RODNEY QUAYE

SUPPORT MATERIAL
CRC Press
Taylor & Francis Group

EVENT-DATABASE ARCHITECTURE FOR COMPUTER GAMES

Volume 2, Game Design and the Nature of the Beast

Event-Database Architecture for Computer Games

Event-Database Architecture for Computer Games proposes the first explicit software architecture for game development, answering the problem of building modern computer games with little or no game design. In this volume, an example of a practical production process based on the software production process is explained, including examples of the game design, technical design, data design and tools design in that process.

This volume includes a brief overview on how to optimise the results. This leads on to an exploration of how staff, especially Software Engineers, typically view optimisation. It also explains how the vision of the Engineers relates to the vision of the leadership of a project or company. It describes how this leadership can also affect the efficacy of a production process, including the Event-Database Production Process.

This book will be of great interest to professional game developers involved in management roles such as Technical Directors and Game Producers and technical roles, such as Tools Programmers, UI Programmers, Gameplay Programmers and Engineers, as well as students studying game development and programming.

Rodney Quaye is Senior Software Development Engineer in Test at Build A Rocket Boy. He has worked in the Computer Games industry for over 16 years. He has worked at several Games Studios, including Sumo Digital, nDreams, Supermassive Games, Traveller's Tales, Hotgen, Oysterworld, Second Impact, Flaming Pumpkin, Goldhawk Interactive, Jagex, Gusto Games, Criterion, Asylum Entertainment, Codemasters and Deibus Studios. The famous titles he has worked on include *Burnout 2* and *3* for Criterion, *LMA Manager* for Codemasters, *Runescape* for Jagex, *Lego Worlds* for Traveller's Tales and *Everywhere* for Build A Rocket Boy.

Taylor & Francis
Taylor & Francis Group
http://taylorandfrancis.com

Event-Database Architecture for Computer Games

Volume 2, Game Design and the Nature of the Beast

Rodney Quaye

CRC Press
Taylor & Francis Group
Boca Raton London New York

CRC Press is an imprint of the
Taylor & Francis Group, an **informa** business

Designed cover image: Shutterstock

First edition published 2026
by CRC Press
2385 NW Executive Center Drive, Suite 320, Boca Raton FL 33431

and by CRC Press
4 Park Square, Milton Park, Abingdon, Oxon, OX14 4RN

CRC Press is an imprint of Taylor & Francis Group, LLC

© 2026 Rodney Quaye

Reasonable efforts have been made to publish reliable data and information, but the author and publisher cannot assume responsibility for the validity of all materials or the consequences of their use. The authors and publishers have attempted to trace the copyright holders of all material reproduced in this publication and apologize to copyright holders if permission to publish in this form has not been obtained. If any copyright material has not been acknowledged please write and let us know so we may rectify in any future reprint.

Except as permitted under U.S. Copyright Law, no part of this book may be reprinted, reproduced, transmitted, or utilized in any form by any electronic, mechanical, or other means, now known or hereafter invented, including photocopying, microfilming, and recording, or in any information storage or retrieval system, without written permission from the publishers.

For permission to photocopy or use material electronically from this work, access www.copyright.com or contact the Copyright Clearance Center, Inc. (CCC), 222 Rosewood Drive, Danvers, MA 01923, 978-750-8400. For works that are not available on CCC please contact mpkbookspermissions@tandf.co.uk

Trademark notice: Product or corporate names may be trademarks or registered trademarks and are used only for identification and explanation without intent to infringe.

ISBN: 9781032820699 (hbk)
ISBN: 9781032818078 (pbk)
ISBN: 9781003502807 (ebk)

DOI: 10.1201/9781003502807

Typeset in Times
by KnowledgeWorks Global Ltd.

Access the Support Materials: www.routledge.com/9781032818078

Contents

Taylor & Francis
Taylor & Francis Group
http://taylorandfrancis.com

About the Author

Rodney Quaye is Senior Programmer who has worked in the Computer Games industry for over 16 years. He was born in the UK but grew up in his fatherland, Ghana, attending primary school there. He returned to the UK to attend secondary school. He grew up playing Computer Games at school and university but never thought of it as a career. He graduated from the University of Warwick with a Bachelor of Engineering degree in Computer Systems Engineering in 1993. He went to work as a programmer first on medical information systems for hospitals and then market analysis systems, mainly for car manufacturers. He then had a near-death experience which gave him a spiritual awakening. He reflected on his life and realised that his heart was not in his work. He felt God was calling him back to his first love, Computer Games. So he started a career in that industry in 1999, working at several Games Studios, including Sumo Digital, nDreams, Supermassive Games, Traveller's Tales, Hotgen, Oysterworld, Second Impact, Flaming Pumpkin, Goldhawk Interactive, Jagex, Gusto Games, Criterion, Asylum Entertainment, Codemasters and Deibus Studios. The famous titles he has worked on include *Burnout 2* and *3* for Criterion, *LMA Manager* for Codemasters, *Runescape* for Jagex and *Lego Worlds* for Traveller's Tales. He wrote this book to help others get into the development of Computer Games.

Taylor & Francis
Taylor & Francis Group
http://taylorandfrancis.com

Introduction

In previous volume in the series,

Event-Database Architecture for Computer Games: Volume 1, Software Architecture and the Software Production Process,

the problem of building modern computer games with little or no game design was introduced, along with a software architecture for solving this problem. An archetypal software production process, based on this architecture, was also explained.

In this volume in this series,

Event-Database Architecture for Computer Games: Volume 2, Game Design and the Nature of the Beast,

an example of a practical production process based on the software production process will be explained. This will include an example of the game design, technical design, data design and tools design in that process.

This volume will also include a brief introduction about how to optimise the results. This leads on to how staff, especially Software Engineers involved in computer games, typically view optimisation. And this leads on to how Engineers fall basically in two schools of thoughts. One school views software production as an art. The other views software production as a science. These competing visions can effect the efficacy of a production process, including the Event-Database Production Process.

This volume will also explain how the vision of the Engineers relates to the vision of the leadership of a project or company. It will describe how this leadership can also effect the efficacy of a production process, including the Event-Database Production Process.

There is a glossary of terms and list of references in the final volume in the series.

Taylor & Francis
Taylor & Francis Group
http://taylorandfrancis.com

1 *LPmud* Software Production Process

This game, *LPmud*,[1] is designed to be played by millions of players around the world, and to be available 24 hours a day. The players will be able to log in and play cooperatively, or competitively with other players, for as long as they like. And after they have had enough they can log out and rejoin at a later date to continue their adventures. It will allow the players to not only take part in adventures in this world, but contribute to the development of new ones once their characters reach their highest level. At that point the players will become 'wizards' and 'gods' in this world, and be able to build new adventures or extend old adventures for the next generation of players to enjoy. The game will be available for multiple *platforms* including PCs, game consoles and Mobile Phones. The game will be developed for the target *platforms* using an **Event-Database Production Process** based on the **Event-Database Architecture**. You can see this vision for the game in the cover page in Figure 1.0.

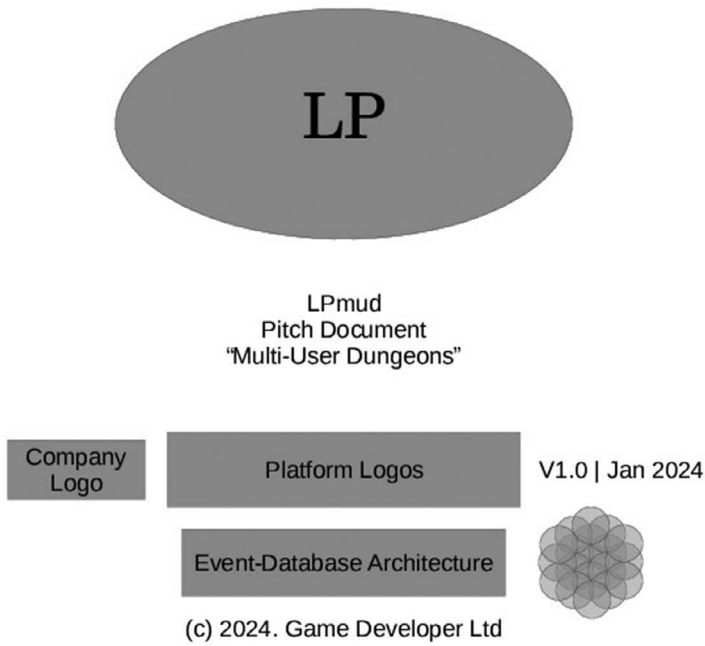

FIGURE 1.0 An example of a cover page for a document to pitch a project to build a computer game *LPmud*.

DOI: 10.1201/9781003502807-1

To summarise, the **Event-Database Production Process** has four advantages over a normal *Software Evolution Process*.

Firstly, when there is a new change to the requirements of the *game design*, the *Software Evolution Process* produces a new set of *game modules*, *Game data* and *Abstract data* to meet those requirements. The **Event-Database Production Process** produces a new set of **Primary Events**, **Secondary Events**, **Game Objects**, *Database Tables*, *Database Records* and *Database Fields*. The second set has a greater tolerance to the changes in the *game design* than the first. By virtue of its members.

Secondly, during the production process and especially at the end, you can identify and test every member of the second set, every **Primary Event**, **Secondary Event**, **Game Objec**t, *Database Table*, *Database Record* and *Database Field*. And thus you can have greater *Quality Control*[2] than in a *Software Evolution Process*. This can all be done from one source and one tool: the *Game Database*. But you cannot identify and test every member of the first set, every *game module*, *Game data* or *Abstract data* with a *Software Evolution Process* or popular commercial *game-engines* and *game-editors* which have also been developed with a *Software Evolution Process*. In a *Software Evolution Process*, there is no such single source or tool.

Thirdly, there is a book that explicitly explains the **Event-Database Production Process** and the **Event-Database Architecture**. This book is called

Event-Database Architecture for Computer Games: Volume 1, Software Architecture and the Software Production Process.

This book gives you an understanding of the *software architecture*[3] at the beginning of the process. There is no book which explains the *Software Evolution Process* for computer games and can predict the *software architecture* it will produce. You can only see this at the end of the process.

Fourthly, the **Event-Database Production Process** and the **Event-Database Architecture** provide you with a *Relational Database* and a *Relational Database Management System* for building computer games. To manage and query the huge amounts of data that it takes to build and run modern computer games. The *Software Evolution Process* does not.

This will be a multiplayer game that can be played across a computer network. Therefore, the form of the **Event-Database Architecture** that will be used will be the **Multi-User Distributed Form** based on the **Client Server Network Architecture**.

Please refer to the subchapter entitled **Multi-User Distributed Form Client Server** in the book to see that **Form**.

Please refer to the subchapter entitled *The Software Production Process*[4] in the book to see the steps of the **Event-Database-Production Process**.

1.1 STEP 1: *LPmud* FEASIBILITY STUDY/VERTICAL SLICE

LPmud is a multi-user adventure game. The feasibility study will include a Vertical Slice of the game showing you a short but in depth sample of what the final *Game*

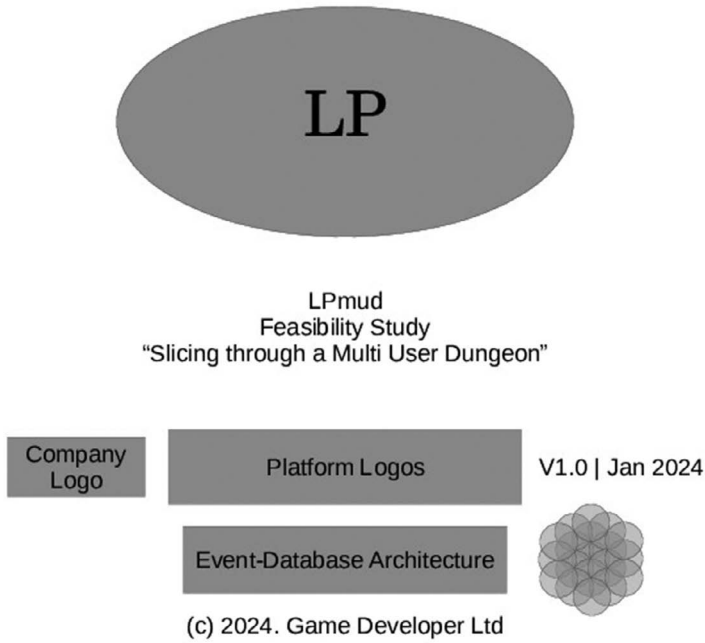

FIGURE 1.1 An example of a cover page for a feasibility study to build a computer game *LPmud*.

World,[5] will be like. You can see the vision for the study in the cover page in Figure 1.1. The building of the Vertical Slice of the game on the target *platforms* will follow the same steps for the standard feasibility study for a game built with the **Event-Database Production Process**. Please refer to the chapter entitled

The Feasibility Study And Test/Vertical Slice

in the book

Event-Database Architecture for Computer Games: Volume 1, Software Architecture and the Software Production Process.

In addition to those standard steps, there will several more custom steps.

For this game *LPmud*, the 'Vertical Slice' will have a main menu or 'Frontend menu' from which the player can choose one option. And that is to play a 'Single Player game' in The Village.

The 'Single Player game' will have all of the 2D and 3D artwork for a single part of the *Game World* called 'The Village' developed up to the highest standard. That is to say, the *Quality*[6] it will have in the final release of the game. This will include all of the artwork and animations for the sky, birds, trees, bushes, vegetation, rivers, local animals, local villagers and houses in the village.

It will have all of the sounds for the sky, birds, trees, bushes, vegetation, rivers, animals, local villagers and houses also developed to highest standard. That is to say the *Quality* it will have in the final release.

The player will be able to wander freely along a single road which passes through the centre of the village, from one end to the other. And when the player has finished exploring and reaches the end of the road, the game will end. After the title and end credits, showing the names of those who contributed to building it have been displayed, the game will shut down.

1.2 STEP 2: *LPmud* GAME DESIGN

LPmud is a multi-user adventure game. The *Game World* is set in medieval times and is based on medieval folklore, myths and fantasies. It is made up of settlements, buildings, various wild landscapes and terrains, creatures, characters, puzzles, quests and treasures, for the player to discover and explore. You can see the vision for the study in the cover page in Figure 1.2.

One of the unique aspects of the game is that the *Game World* is developed with *LPC*.[7] *LPC* is a programming language that allows the players to modify their *Game World*, add new items to the existing *Game World*, add their own new worlds next to the existing worlds, as well as to test and modify these worlds. All of this can be done while others continue to play the game, without having to shut the game down and bring it back up again. This is because the game is played across the Internet, and it is available 24 hours a day. Hundreds of players may be connected to the game

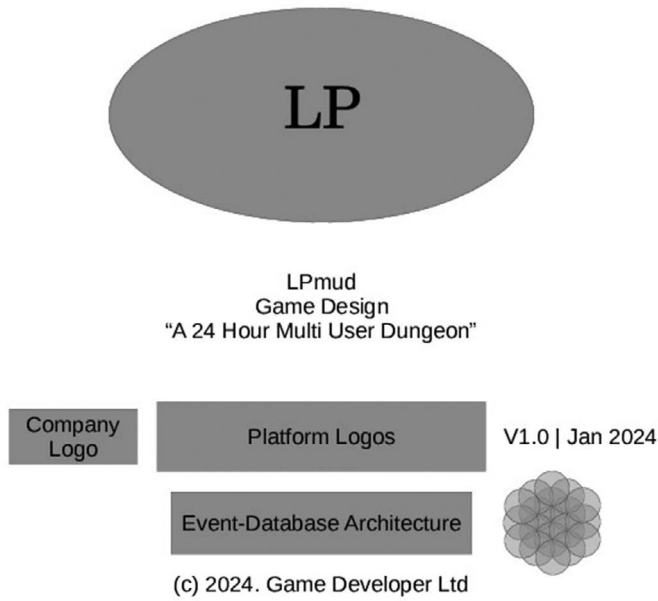

FIGURE 1.2 An example of a cover page for a *game design* to build a computer game *LPmud*.

at any one time. The original version of the game is called 'Genesis' and is available on the Internet, through a 'Telnet Client' at this 'Telnet' or TCP/IP Address here:

mud.genesismud.org
Port: 3011

You can get a 'Telnet Client' from Internet, through a Web Browser and the World Wide Web, at this Web Address

https://www.putty.org/

Or you can access 'Genesis' directly from a Web Browser at the Web Address here

https://www.genesismud.org/

1.2.1 Settlements

The game follows a particular structure. The game is a Role-Playing Game set in a world of magic and fantasy, which you can explore through adventures. It is made up of many small remote towns and villages, separated by great distances. Next to some of the towns or villages will be the castle of the local feudal lord, whose title and castle bares the name of that town or village. And in and around that castle, will be the courts, fields, gardens and giant arenas. Where this lord holds festivals and sporting events for the locals from time to time.

The towns and villages have several buildings, pubs, shops, markets, guilds and homes. And within these you will find a vibrant economy. Where you can buy almost anything you need to take on your adventures or sell anything you bring back from your adventures.

The towns and villages are where the players builds up their level or reputation amongst the inhabitants. Within some of the towns and villages are guilds which players may join to work collaboratively with other players. Some of these guilds are sociable and open, offering help to other characters like a fighters guild or a magician guild. Other guilds are anti-social and clandestine such as a thief guild or an assassins guild. Players frequent these guilds when they want to take a short break from the game. They may share their past adventures and organise new ones, with other players in their guilds. And the players may also use whatever money they have to purchase experience in a guild to improve their level or reputation.

Each player begins with a low-level reputation. The goal of the game is to build up your reputation in the world, by building up a knowledge of the world. This may be achieved through several, preferably heroic, but possibly villainous, acts that you perform as part of your role in your guild. The player's reputation is evident in his or her title and level of experience. The title comes from the guild the player is part of. And the level of experience is a score that increases whenever the player successfully completes difficult puzzles, quests or whenever the player destroys a creature or another character. This includes other players, whom the player may steal from or fight with. You can see a break down of the settlements and the buildings in each one in Figure 1.3.

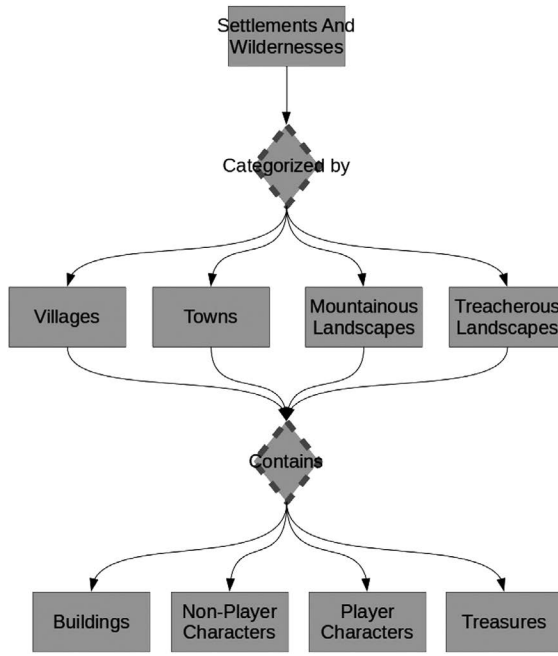

FIGURE 1.3 A hierarchical breakdown of one of the main themes of the *Game World* of *LPmud* i.e. Settlements.

1.2.2 Buildings

Each building in the towns and villages has an inventory of items in it which gives it its character. The pub has a landlord, local patrons, tables, chairs, glasses, bottles of alcohol or wine, a cellar and an open fireplace.

The shops have a shop owner, local patrons buying or selling items to the owner, shelves of items which you can buy in the store, a display of items outside the store, a sign with the name of the store above the entrance, its owner and whatever goods it trades in. Each shop will have a front facing entrance to the main street through which the customers enter. And the shop will have a back entrance where the goods sold in the shop arrive.

The markets are made of rows of market stalls made out of wood. Each stall has produce on display, from locally grown crops such as apples, oranges and other fruits to durable items such as jewellery, clothes, shoes, weapons and armour. And a stall will also have tradesmen selling the goods on display.

The guilds have a sign with the name of the guild above the entrance, the guild-master and a motto. Each guild will have a large hall where the members gather. When you enter this hall, you will see more signs advertising the services the guild offers and the roles or adventures that the players can take part in who join the guild. And you will see along the walls of the hall portraits of the famous members of the guild.

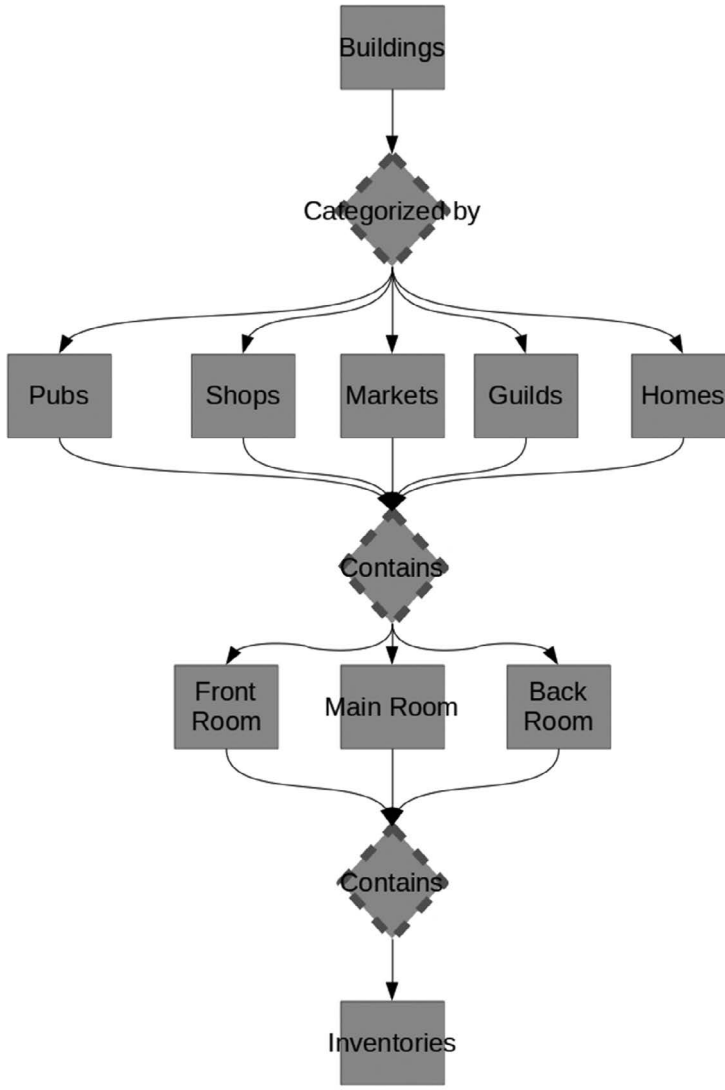

FIGURE 1.4 A hierarchical breakdown of one of the main themes of the *Game World* of *LPmud* i.e. Buildings

The homes will have a living room, kitchen and bedroom. In the living room will be tables and chairs and an open fireplace. In the kitchen will be more tables, and food and drinks laid out on these tables. And you will also find knives, pots, pans and ovens for preparing the meals. And in the bedrooms you will find beds with a view looking into a garden at the back of the home. Each home will have a front entrance from the main street or road, and a back entrance through the garden. You can see a breakdown of the buildings in Figure 1.4.

1.2.3 Mountainous Landscapes

In between the remote towns and villages, there will be wildernesses in different forms that the players must traverse. One form that wilderness will take is mountainous terrain.

In this terrain there will be clouds floating across the tops of the mountains, and rivers or streams flowing through the valleys. There will be sounds which reflect the nature of that terrain. This includes the sound of the wind blowing through the clouds, switching between a loud rush and a quiet breeze. This includes the sounds of the rivers or streams flowing through the valleys.

The terrain will be ragged with lots of sharp edges, cliffs and overhanging ledges. It will be littered with rocks, small and large, caused by the erosion from the wind, rivers and streams, as well as the occasional rock falls.

In this terrain the player will find many puzzles, quests and other valuables. The valuables will include random items found lying on the floor occasionally. This includes weapons, armours, shields, money or corpses of past adventurers with such items on them which you can loot.

The quests will come in the form of hostile wild creatures you have to kill. These creatures can range from large, such as dragons, to small, such as mountain goats or lions.

The puzzles will come in the form of remote hermits or travellers you encounter encamped in the mountains, either in tents or caves. Who will present you with a riddle you must answer. Or present you with a board game in which you must beat them such as chess or Go.

Each puzzle, quest or other valuables will have a set of commands for the player to interact with it. And it will have a score which the player will receive as a reward for completing or finding it.

Each will have a guard or Non-Player Character (NPC) who introduces the player to the puzzle, quest or valuable. If it were a puzzle, then this will be the hermit or traveller who shows the player the puzzle. If it is a quest, then this will be the guildmaster from one of the guilds in the villages or towns who gives the player that quest. If it is some other valuable, then this will be some inhabitant of a town or village who tells the player where to find it. This guard will either be hostile or friendly, and either immediately attack the player on sight or simply wait for the player to approach before engaging in conversation. The guard will also have an inventory of items they carry which reflects their background or origins or profession.

These items in the inventory and the other valuables the player can find will all have a weight and a value when sold in the shops. You can see a breakdown of the mountains and valleys and the contents that the players may come across in Figure 1.5, Figure 1.6 and Figure 1.7

1.2.4 Treacherous Landscapes

Other forms in which the wilderness between the towns and villages would take would be caves, forests, woods, deserts and icy terrain. In each terrain there would be wild creatures, both large and smaller, suitable for that terrain and treasures in the form of puzzles, quests and other valuables, just as in the mountainous landscape.

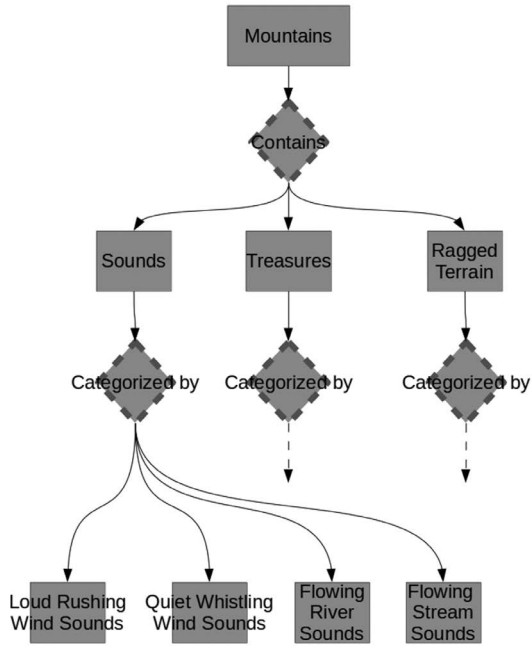

FIGURE 1.5 A hierarchical breakdown of one of the main themes of the *Game World* of *LPmud* i.e. Mountains.

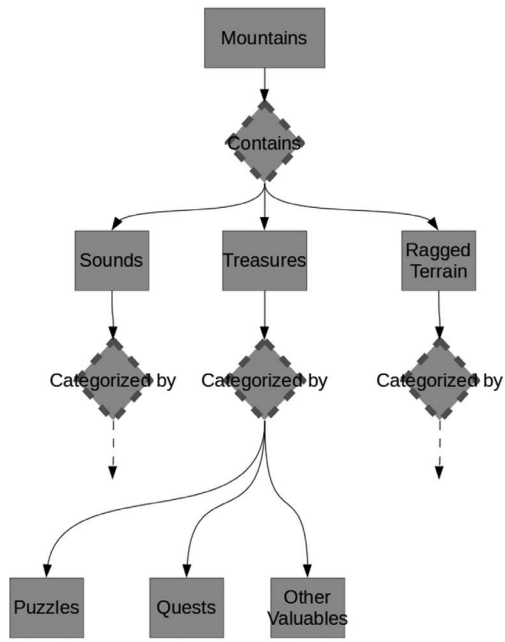

FIGURE 1.6 Extension of Figure 1.5.

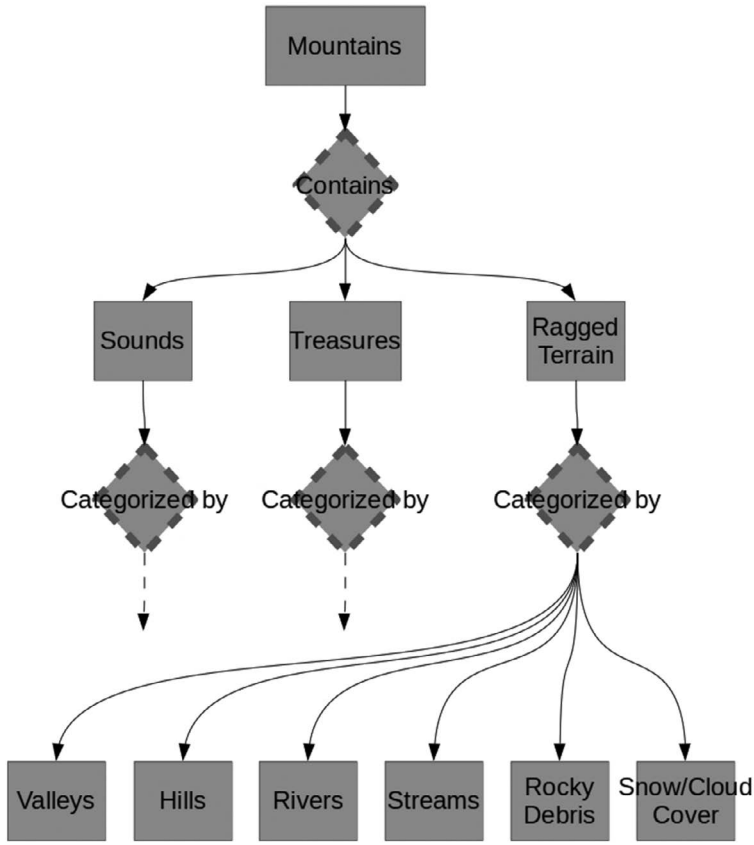

FIGURE 1.7 Extension of Figure 1.5.

But there will be some major differences. The first major difference is that there would be a strong emphasis on lightness and darkness in the landscape. That is to say there would be areas so dark that if the player does not go there with a light source, they will see nothing. This includes the caves, forests and the woods. In contrast the icy terrains will be saturated with light, and full of reflections of the surface.

The second major difference is that all of the wild creatures would be hostile and attack the player on sight in these regions, or run away.

The third major difference is that there will be puzzles in the forms of mazes which the player has to find their way out of. This would mainly appear in the caves.

The fourth major difference is that there will be traps that do sudden damage to the player. This will include pits, or sudden steep drops in the caves or forests, or cracks in the ice covering lakes, which transport the player a great distance and leaves you disorientated. It is through these traps that the player will suddenly find themselves in some of the puzzles in the form of a maze.

The fifth major difference will be the hostile wild creatures in each domain. In the caves, you would find lions, dragons and bears. In the forests, you would find snakes,

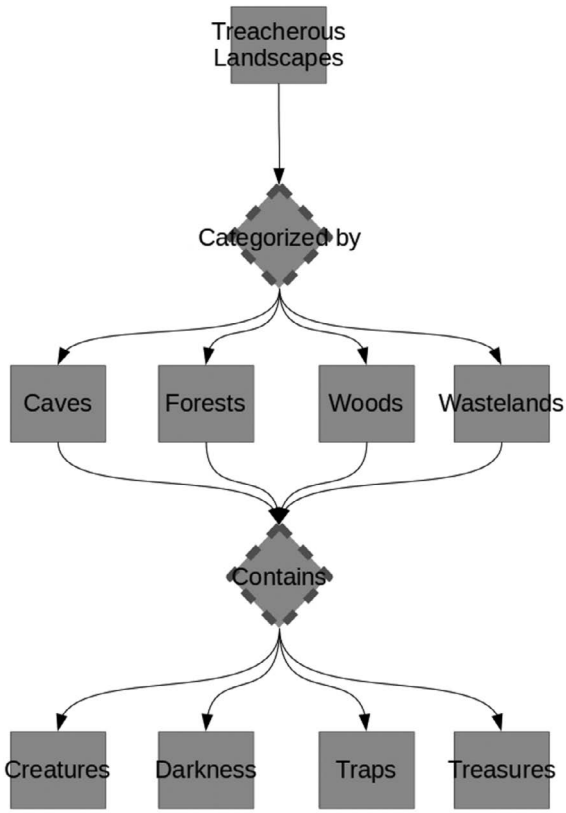

FIGURE 1.8 A hierarchical breakdown of one of the main themes of the *Game World* of *LPmud* i.e. Treacherous Landscapes.

birds, spiders and wild boars. In the woods, you would find deer, birds and bears. In the icy terrains you would find polar bears, penguins, sea lions and white tigers. You can see a breakdown of these terrains and the inhabitants in Figure 1.8.

1.2.5 Non-Player Characters

The NPCs are any characters which are not controlled by the player. These instead would be controlled by the computer or some form of *Artificial Intelligence*.[8] This would include the inhabitants of the towns and villages, the publicans, the pub patrons, market sellers and buyers, merchants, guildmasters, guild members, shop traders and shop customers. This would include any men or women outside the towns and villages guarding treasures or puzzles that the player may come across. And those that introduce the player to quests in remote places. These would also include any domesticated or wild creatures.

Each NPC would have a command to interact with it. At a minimum this will be a command to just look at the NPC, and see its name, its short and long description.

That described its role in the *Game World*. And at maximum there will be several commands you could give to receive some information or item from the NPC. For example, if the NPC were the guildmaster, then you could use the command to get quests from the guildmaster. If the NPC were a shop owner, then the command would buy an item or sell an item to the shop.

Each NPC would also have a score which the player would receive for killing it, a health which would reflect how strong it was in combat and an inventory which would contain the items it was carrying. These items would reflect the role of that character in the *Game World*. If the NPC were a publican, then this could be some glasses or bottles of beer and perhaps some money. If the NPC were a shop owner, then this could be some items sold in the shop.

All of the human NPCs would be friendly. And they would only attack the player, if they were attacked. At which point they would become hostile and behave in a similar way as hostile wild creatures described earlier. The difference is that the human NPCs would wield whatever weapons or armour they had in their inventory and use that in combat.

The weapons or armour they would use and how effectively they would use this in combat would be determined by an *Artificial Intelligence*. An *Artificial Intelligence* would also be used to control the movement of NPCs that could move.

For example, a human soldier guarding the entrance to a castle or a throne room would patrol up and down in front of the castle gates or entrance to the throne room, through several Waypoints controlled by the *Artificial Intelligence*.

Another example, a deer in the woods would run from one point to another. To bend over and nibble at the foliage of trees and bushes for a few minutes, then the deer would run to another point in the woods and repeat the cycle again and again. These movements would be through a set of Waypoints controlled by an *Artificial Intelligence*. You can see a breakdown of all the NPCs in Figure 1.9 and Figure 1.10.

1.2.6 Player Characters

The player's character will have all of the qualities of a human NPC. Except it will be controlled by the player instead of the computer.

Each player's character will begin with no level of experience and be given a title that suitably reflects this. The level of experience is the player's score. It rises whenever the player completes a puzzle or quest or kills a creature or another character. When the level of experience rises over a threshold, the player's character will be rewarded with a new title. This title reflects the growing reputation that the character has. There will be a limit to the number of thresholds. And each time the player passes one threshold, the amount of experience required to reach the next level will increase exponentially.

But the player's character will fall back down the levels, however, whenever that character dies. The score will decrease by a set percentage, whenever this happens.

When a player reaches the highest level of experience, the player becomes a Wizard. A Wizard is immortal and cannot die. Wizards do not play the game. Instead, they teach other players how to play the game by giving them help and

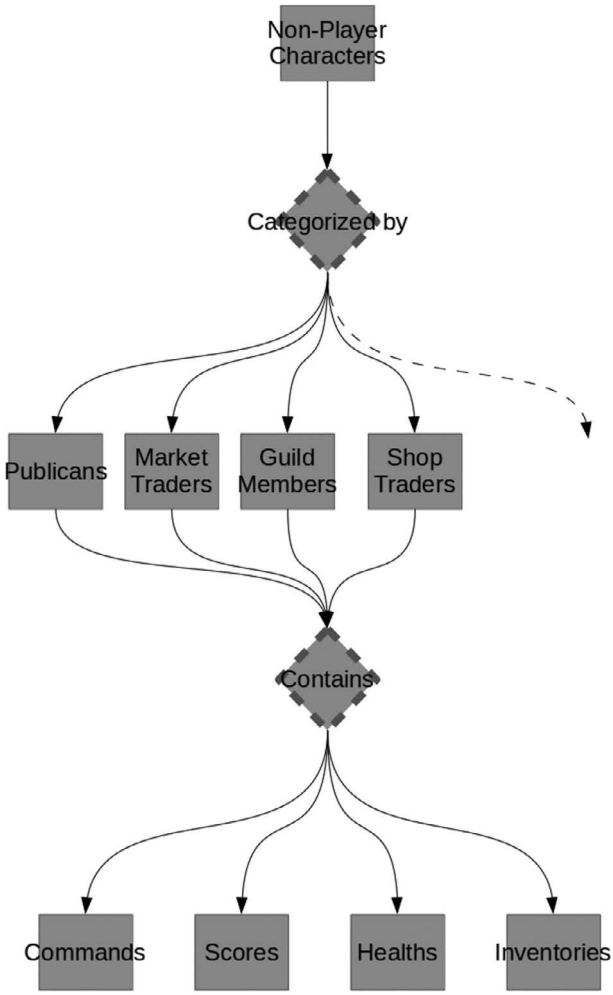

FIGURE 1.9 A hierarchical breakdown of one of the main themes of the *Game World* of *LPmud* i.e. Non-Player Characters.

advice. They also develop the *Game World* by editing it. A Wizard can add a new domain to the *Game World*. This domain will be named after the former player's character. The theme of the domain could be whatever the player wishes. So long as it is popular with the players. And his popularity will be determined by how many times the players frequent there. The player may add new towns, villages or terrains in that domain. And in these new areas, the player may add new inhabitants, creatures, puzzles, quests and other treasures for other players to discover.

Inevitably, the contents of the domain will be partially inspired by the player's past adventures in the *Game World*. And it will partially be inspired by ideas from the player's experiences outside of the *Game World*. Nevertheless, after the player

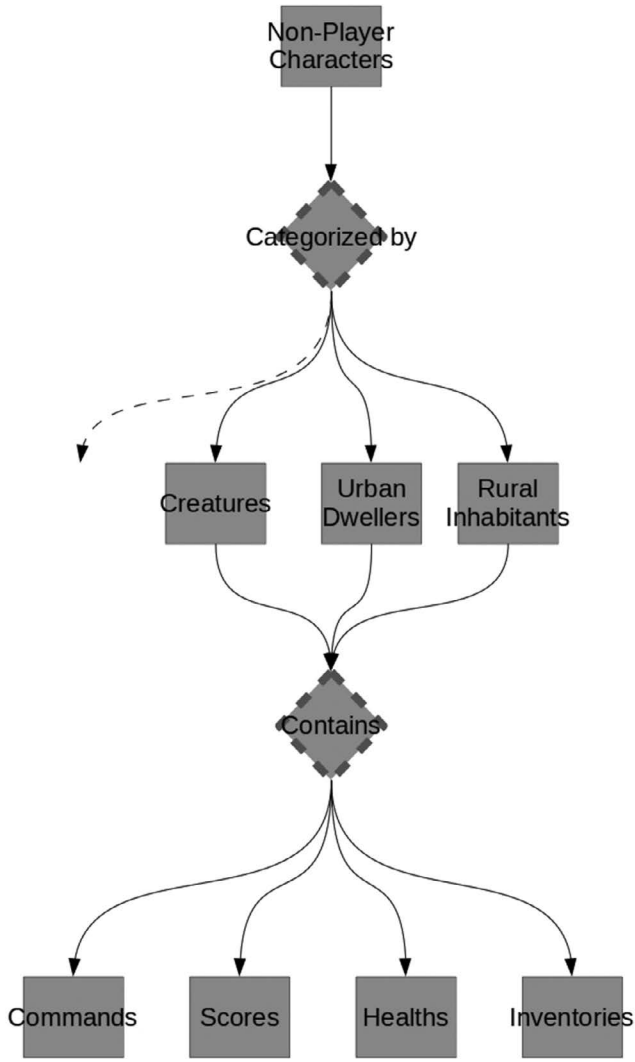

FIGURE 1.10 Extension of Figure 1.9.

has added the new domain, this will add a new dimension for the next generation of players to explore. Each Wizard is awarded a new form of score that is related to their domain. The score increases depending on how many players enter it and how much time they spend in it.

Before a player becomes a Wizard, however, you will be free to take whatever path you want through the *Game World* to increase your level of experience. Each player's character will not follow the same linear progression through the *Game World*. Each domain will present you with a single theme. Each will have its own set

of puzzles, quests, treasures, creatures and characters for you to explore at leisure that fits that theme.

Whatever path you choose to take, all players will share the same *User Interface* to interact with the *Game World.* Since this game will be a multiplayer game played across a computer network, it will be built with the **Event-Database Architecture** in a **Multi-User Distributed Client Server Form**. As already described in the sub-chapter entitled

Multi-User Distributed Client Server Form

in

Event-Database Architecture for Computer Games: Volume 1, Software Architecture and the Software Production Process.

It will be run by a large central powerful computer or **Game Server** connected to a series of less powerful computers or **Game Clients**. The players will use the **Game Clients** to connect to the **Game Server** and play the game through its *User Interface*.

When you connect the **Game Client** to the **Game Server**, you will be presented with a short menu, with a message welcoming you to the game e.g.

LPmud
Version 1.0.0.
Event-Database Architecture
Number of Users: 1000
Enter the name of your character:
Password:

This menu would include a prompt for the player to enter a name of an existing character or a new character, and a Password for authenticating their access to that character.

If your character were new, then you would be presented with submenus to customise the details of your character. This would include your character's Password, age, sex, race, appearance and guild you want to belong to.

After you have entered the details of your character, the character would enter the *Game World*. If the character has been played before, whatever details that character had, when the player last left the *Game World*, would be restored from the **Game Database**.

Following their entrance into the world, each player will be shown a view of the world, from a camera which is at a fixed position above and behind the head of the player's character. This first location will always be the same location, in the building of the guild that the player belongs to, in one of the friendly, inhabited towns or villages. If the player had been disconnected temporarily from the **Game Server**, due to an error or loss of connection on the Internet, then the first location will be whatever the last location of the player's character was before the disconnection.

The player will be able to interact with the *Game World* by issuing these commands through the *Game Controllers:*

FORWARDS COMMAND
BACKWARDS COMMAND
TURN LEFT COMMAND
TURN RIGHT COMMAND
JUMP UP COMMAND
JUMP DOWN COMMAND
LOOK COMMAND
GET COMMAND
DROP COMMAND
GIVE COMMAND
WIELD COMMAND
WEAR COMMAND
REMOVE COMMAND
SAY COMMAND
TELL COMMAND
SHOUT COMMAND
KILL COMMAND
RESURRECT COMMAND
QUIT COMMAND

The **Forwards Command** will accelerate the player forwards, in the direction the camera was facing, up to a maximum speed.

The **Backwards Command** will decelerate the player's forward motion, in the direction the camera is facing, eventually causing the player to stop any forward motion. And then start accelerating backwards, in the opposite direction the camera was facing, up to a maximum speed.

The **Turn Left Command** will accelerate the player's angular motion to the left, in an anti-clockwise direction, from the direction the camera was facing, up to a maximum speed.

The **Turn Right Command** will accelerate the player's angular motion to the right, in a clockwise direction, from the direction the camera is facing, up to a maximum speed.

The **Jump Up Command** would either jump the player's character upwards and back down on the floor or surface the character was standing on. Or it will automatically move that character onto any overhanging ledge or level that the character can reach and walk along. And the **Jump Down Command** would make the player's character either crouch down and hold that position. Or it would automatically move the character down off a ledge or level, down to a lower level that the character was facing and could walk along.

The **Look Command** will allow the players to look closely at any item they select in the *Game World* and see any additional details that item may have.

For example, if the players were to look at either a building in a town, a feature of a landscape, they would see the details of any of its characteristics. They would also see whether they could enter it or not. Or if the players were to look at a tree,

they would see whether they could climb it or not. If the players were to look at an item lying on the ground, they would see any markings it had and whether they could pick it up or not. If the item had a message or sign, large or small, attached, they would be able to read it. And if the item offered the players new commands they could use, they would be able to see these commands and read the instructions for these commands.

If the players were to look at another character, they would see the features of the head, hair, face, arms, legs, torso and the rest of the body of that character. They would also see what the character was wearing, and any items that character was carrying in their And all the other characters, in the same location, will receive a message saying what the players were looking at.

All the characters and creatures in the *Game World* would be able to carry items. And each would have an inventory that stores the items they were carrying. The player's character may pick up any item nearby and add it to the inventory with the **Get Command**.

The players may also drop an item, from the inventory, by issuing a **Drop Command**. Each item would have a weight. And each inventory would have a limit to the total weight of the items it can carry. And once this limit had been reached, that character or creature cannot carry any more items. And this would be displayed in the *User Interface* to any player who subsequently attempts to pick up an item. Or when they try to pick up a very heavy item, whose weight exceeds this limit. Or when they try to give an item to another character or creature with the **Give Command** that causes this limit to be exceeded.

Once a character is carrying an item, that character can then use it. It may be worn, if it were a piece of clothing or armour, by using the **Wear Command**. Or it may be wielded, if it were a weapon, by issuing the **Wield Command**. Or the item may have its own commands for using it. As has already been mentioned, these commands would either be revealed when the player examined the item. Or the commands may be revealed when the player examined the location where the item was found. Or another character in the game may reveal the commands, to the player, as part of some quest.

When the player no longer wishes to use an item, the player may use the **Remove Command** to remove it. This would be only if the item were wielded or worn by the player's character. Or the player may drop the item using the **Drop Command**. Or the player may give it to another character, including another player's character using the **Give Command**. This last command allows the players to work collaboratively.

The players may also collaborate by communicating. All the characters in the game can communicate with other characters through either the **Tell Command**, to send private messages. Or through the **Say Command** to broadcast to other characters in the local vicinity. Or the **Shout Command** to broadcast to other characters at remote distances. You can see a breakdown of all player characters, their features and interactive commands in Figure 1.11.

1.2.7 Creatures

In the landscapes outside of the towns and villages will be wild creatures. Each creature has a set of commands you can use to interact with it, a score you get for killing

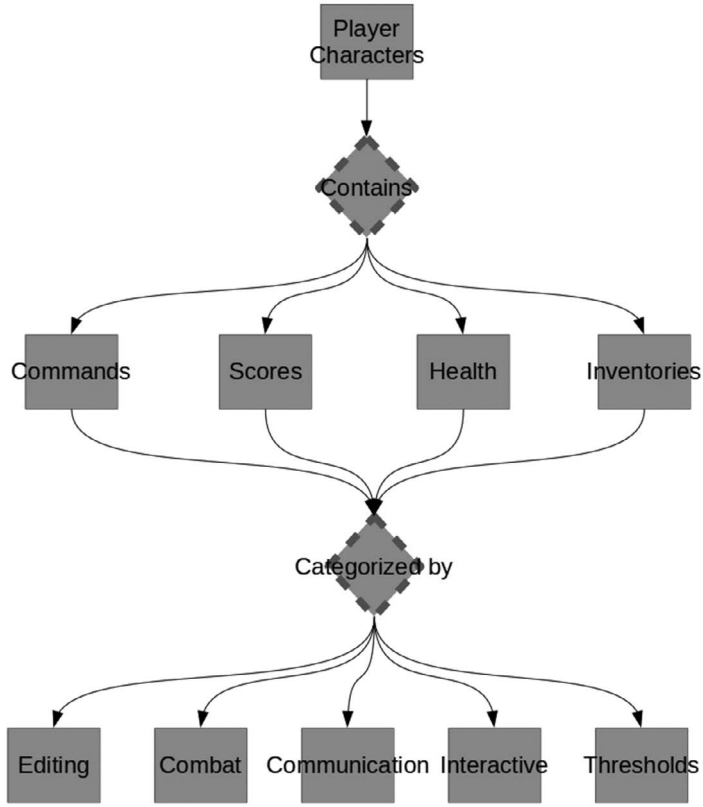

FIGURE 1.11 A hierarchical breakdown of one of the main themes of the *Game World* of *LPmud* i.e. Player Characters.

it, a health which reflects how strong it is in combat and an inventory which contains the items it will drop when it is killed.

A creature may also be hostile or friendly. A hostile creature will attack the player on sight. And it will continue to chase and hunt the player until it or the player is dead. A friendly creature will not attack the player unless the player attacks it, intentionally or unintentionally, by doing damage to it. At which point it will become hostile.

When multiple characters, controlled either by players or non-players, attack a creature, the first one that attacks it will be the one that it will be hostile too and its target. After that target is dead, then the next attacker that hits it will become its target, and so on and so on. Until either the creature or all the attackers are dead. You can see a breakdown of all creatures, their features and interactive commands in Figure 1.12.

1.2.8 Treasures

As previously mentioned, treasures in the *Game World* would come in the form of either guarded treasures i.e. puzzles or quests that give rare valuable rewards to the

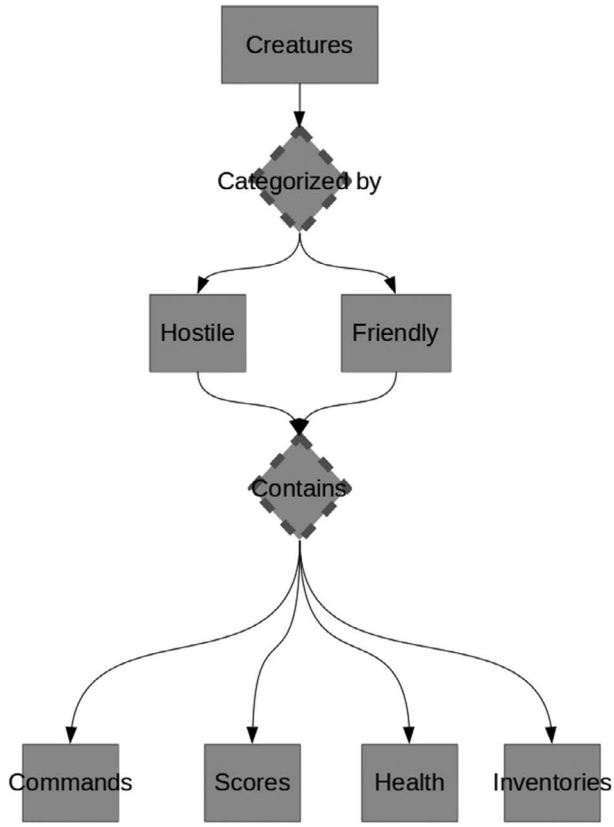

FIGURE 1.12 A hierarchical breakdown of one of the main themes of the *Game World* of *LPmud* i.e. Creatures.

players when they solve or complete them. These all will have an NPC guarding that treasure who introduces the players to the puzzles or quests, explains the rules and the rewards and gives them any help or assistance to complete it.

Or the treasures may come in the form of unguarded treasures i.e. rare items which the player finds lying around, in containers or outside containers, on the ground at random. These containers include bags, corpses or treasure chests which the players find in the *Game World* at random. These containers will have inventories which will include one or more special weapons, armour, shields, jewels, books of magic spells, magical wands, magical scrolls, potions, rods, staff, cups, cutlery, crowns, robes, rings, boots, necklaces, gloves made from precious metals and stones which they can loot.

Each treasure will have a weight which will limit the number that can be carried by a player in the player's inventory.

Each treasure will have a value. This will encourage the players to visit the social areas, such as the shops in the towns and villages, and meet other players. In these shops they will be able to sell the treasures for money which they can use to buy

items they need. The rarer the treasure is, the higher the value they will be able to sell it or buy it back for.

The treasures in the *Game World* would be randomly distributed. To encourage the player to explore as much of the *Game World* as possible. And to encourage the player to examine every character or creature they come across very closely.

After a while, most of the treasures in the *Game World* in the various locations will either have been found by the players or have been moved around. Or these may have been sold in the shops or markets in the town. Or these may have disappeared from the game after being used or destroyed. So, after long periods, say every 24 hours, some or all of these treasures will be reset back to the initial positions where these first appeared. Any characters or creatures that were guarding, or in possession of, these items would also be reset as well. You can see a breakdown of all guarded and unguarded treasures, their features and origins in Figure 1.13 and Figure 1.14.

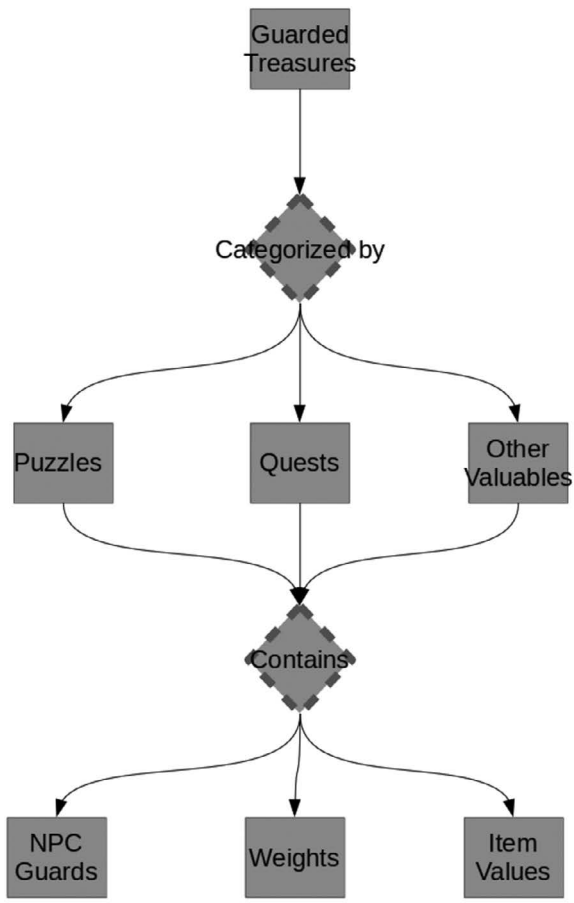

FIGURE 1.13 A hierarchical breakdown of one of the main themes of the *Game World* of *LPmud* i.e. Guarded Treasures.

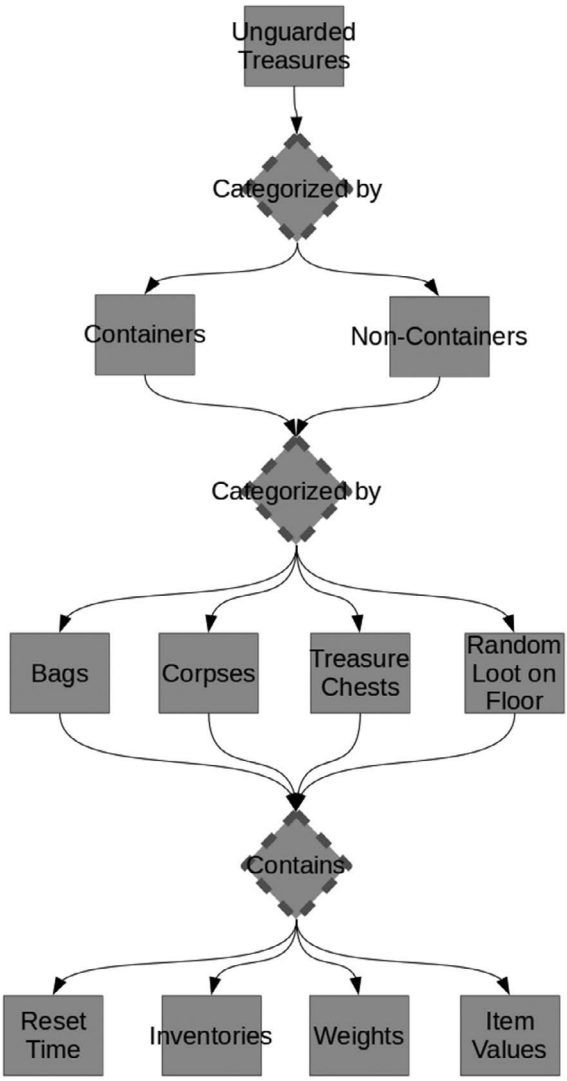

FIGURE 1.14 A hierarchical breakdown of one of the main themes of the *Game World* of *LPmud* i.e. Unguarded Treasures.

1.2.9 Combat System

The player can issue a **Kill Command** to start attacking another character or creature. In the local vicinity. An NPC or a player's character may also begin combat when the player issues some other offensive command, from an item that character was carrying. And that command subsequently damages another character or creature nearby.

Once combat has begun, the fight takes place over an indefinite number of periods. Each period, or round of combat, lasts a short time; about two seconds. During

each round, both opponents exchange one blow. Even if a character has multiple opponents, that character still only deals one blow, to one of these opponents during a round. This opponent is either the character or the creature that began the combat, by attacking the player. Or it is whoever a player specifies to attack, by issuing a **Kill Command**, before or during combat. All the characters nearby, participating or not participating in the fight, can see each blow struck.

The amount of damage each character does during each round of combat depends on its **WEAPON CLASS**. This in turn depends on the total amount of damage or **Weapon Class** of all the weapons that character was wielding in the character's inventory.

Likewise, the amount of damage done by each creature, during each round of combat, will depend on its **Weapon Class**. And that in turn depends on the total amount of damage or **Weapon Class** of all the weapons that creature was wielding in its inventory. But some creatures will be carrying no weapons. These will depend only on natural weapons, such as claws or teeth and a natural **Weapon Class**. In all cases, nevertheless, the greater the **Weapon Class** of the character or creature, the greater the potential damage each blow does to an opponent in each round of combat. Although the damage actually done is a random value of the maximum damage that can be done.

However, the effectiveness of each blow will also depend on how much protection or **ARMOUR CLASS** the recipient has. This in turn depends on the total **Armour Class** of the protection that the recipient was wearing. This includes artificial protection, such as any helmets, armour or shields a character is wearing. It also includes natural protection, such as the thickness of the skin which a creature may have. The greater the protection, the lesser the overall effect of each blow on the recipient.

The net effect of each blow (i.e. the opponent's **Weapon Class** minus recipient's Armour Class) to a recipient is taken off the health of that recipient. This is a number that reflects the stamina of the recipient and has an upper limit. When a creature appears in the game, its health is set to this limit. Similarly, when a player creates a new character, that character's health is set to an initial limit. However, another of the rewards which a player receives, along with a new title, when the level of experience of that player's character passes one of the thresholds, is an increase of this limit.

Once an opponent has lost all of the health that the opponent has, during combat, that opponent dies and the combat ends. The corpse of the opponent falls to the ground. And this contains whatever items the opponent was carrying, and any money the opponent had left. Any character left alive may then take the corpse or loot as many items from it as that character can carry.

When a character belonging to a player dies, the character transforms into a ghost. This affects the character's appearance and the commands the player can use. The dead character is still recognisable, from the facial and bodily features. But the character is no longer wearing or carrying any items and appears as a pale, apparition. Instead of walking through the world, the character floats across it. The player can still move around and examine other characters, the surrounding location, and any buildings, structures or items nearby. The player

can also still communicate with other characters. But the player can no longer pick up or drop any items. Nor can the player initiate any further combat or be attacked.

Nevertheless, the player has the option to resurrect the character by issuing a **Resurrect Command** through the *User Interface*. After being resurrected, the character reappears back where all the players start in the world. The character loses a set percentage of the level of experience which he or she had accumulated. And this may in turn cause the upper limit of the health, of that character, to fall, if the level of experience drops underneath one of the thresholds. Either way, the game restores the health of the character back up to its maximum. And the character continues playing the game, restarting with an empty inventory.

Of course, players can avoid this outcome by simply choosing to run away from combat. Any character engaging in a fight can still move. And if a player were to run fast enough away from his or her attacker, so that the attacker could not reach that character within one round of combat, the combat would end. The health of that character could then be restored by returning to a village or a town. And there the player could buy drinks, food or magical remedies in the shops or public houses to consume to restore the character's health.

A player can also quit the game, at any time, by issuing a **Quit Command** through the *User Interface*. This would save the important details of the player's character that would be restored when the player connected back at a later date. The command would also remove the player's character, from the *Game World*, and disconnect the **Game Client** from the **Game Server**.

1.3 STEP 3: *LPmud* TECHNICAL DESIGN

The next step in the **Event-Database Production Process**, after the game *design*, would be to write a *technical design*. See the chapter entitled

"The Software Production Process" in the book

Event-Database Architecture for Computer Games: Volume 1, Software Architecture and the Software Production Process.

The *technical design* will spell out all of the techniques and tools that will be used to produce the multi-user adventure game, which can run 24 hours a day, with hundreds if not thousands of players. Allowing the players to grow progressively, in knowledge and proficiency in the *Game World*, until eventually they reach a level of a Wizard and can create their own domains in the *Game World*. You can see the vision for the *technical design* in Figure 1.15.

But before looking at an example of how to do this, let us look at how a *technical design* would be written in the normal ad hoc process used in the Computer Games industry.

In the *Software Evolution Process* used in the Computer Games industry, the *technical design* would be written in the Pre-Production phase. This document, like

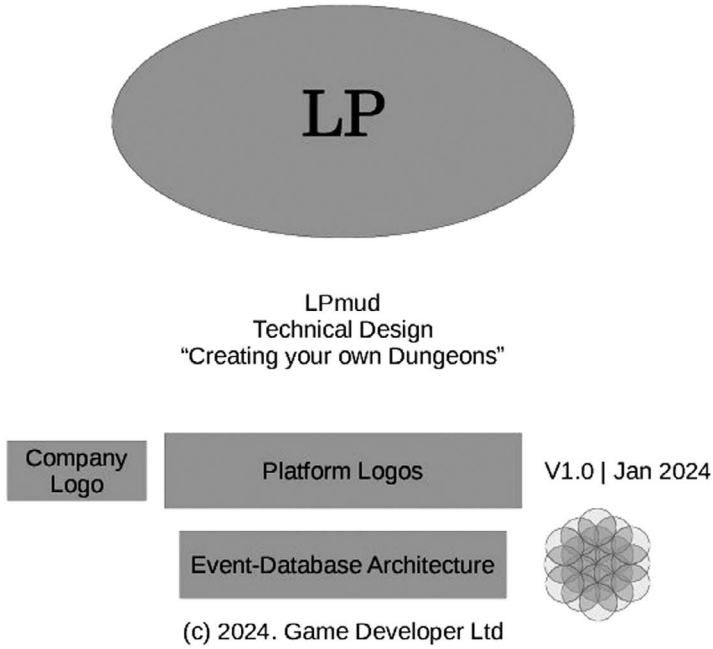

FIGURE 1.15 An example of a cover page for a *technical design* to build a computer game *LPmud*.

the *game design* before it, would only be concerned with selling the game to its financial backer. And so, likewise, it would concentrate on just the highlights. It would begin with a description of the *platform* the game was destined for. This would include its microprocessors, including its main Central Processor, Graphical Processor and Audio Processor. This would include its memory and the storage media which the software would be held on. This would include other components that help the hardware either evaluate mathematical equations, transfer *data* between the hardware components or communicate with other hardware. And this would include the peripherals of the computer hardware, such as its *Game Controllers*.

The *technical design* would begin with a description of the size and speed of all these various components. Most of this description would be lifted, verbatim, from the manuals that accompany the computer hardware, without any added explanation. Furthermore, none of these components would be explicitly related to the components of the *game design*, by authors of the *technical design*. These would merely be mentioned to impress the reader with their knowledge of the computer hardware.

Following that description, of the components of the computer hardware, would be a brief description of the tools, which would be used by the various contingents assembled to build the game. This would include the tools used by the *Game Programmers* to write and debug the *software modules*. This would include a description of the tools that would be used by the *Game Artists*, to create and edit the various graphics and animations for the game. And this may sometimes include

the tools used by the *Sound Designers* to record and edit the various sounds that would be heard during the game. On other occasions the Sound Designers may be completely ignored and their work dismissed as trivial.

The description would also include the *Revision Control Software* that would be used to store the different sets of *software modules*, and *Game data*, used to build the different versions of the game. Although, on some occasions in the *Software Evolution Process*, that tool may be omitted. And, instead, its function would be one of several designated to be handled manually throughout the ad hoc process.

For example, it may be decided that, at arbitrary points along the process, the latest set of s*oftware modules* and *data* would be archived, on some powerful computer, with a large storage media. And that designated computer would be available on a computer network. The *Game Programmers*, *Game Artists*, *Sound Designers* and *Game Designers*, connected to that network, would all be responsible for transferring their work across that network at the right time. But there would be no one person, or software, designated to control the order in which this archive was kept.

All of the tools chosen, and included in the *technical design*, would be based on popular choices. These would be tools that have been used on commercially successful games of the past. This includes popular commercial *game-engines* or *game-editors*. And these would be chosen regardless of what *game design* was being built. None of the choices would be informed by an analysis of the particular problem that the *technical design* was trying to address. And, indeed, the description of these tools may well be copied, verbatim, from some previous *technical design* for another game.

The rest of the *technical design* would concentrate on two or three main components of the game and highlight how these would be implemented. These would be the displaying of the three-dimensional graphics of the *Game World*, the modelling of its physics and the controlling of the behaviour of its NPCs with an *Artificial Intelligence*. This is not to say that the *software modules* which would be used to build these components, would be described in any detail. In fact, there would be no idea of how many *software modules* would be involved, let alone what any of these would do. Instead, the *technical design* would merely include a description of the basic theoretical steps and methods that would be used to display these graphics, model this physics and control these characters. And it would include an analysis of these steps and methods when more graphics, physics or characters were required, by some change in the *game design*.

This analysis may be illustrated graphically through a line graph. The graph would show how the number of steps would, theoretically, be increased when for example more graphics had to be displayed. Further illustrations may include annotated sketches of how one of the locations of the *Game World*, like a village, would be divided up into smaller regions. So that the graphics could be quickly and efficiently displayed to the player, in that location, using the chosen theoretical method.

Likewise, the method chosen for modelling the physics and controlling the NPCs through an *Artificial Intelligence* would be highlighted, through similar graphs and annotated sketches. And these highlights would be set in the context of other locations, and other characters in the *Game World*.

Nevertheless, all of these theoretical methods chosen to display the graphics, model the physics and control the characters would be popular ones. These would

be tried and tested methods, used in past computer games, especially successful or famous ones. These methods would probably have already been published in a book, one of a number of *technical design sources*,[9] and subsequently copied into the *technical design*. Like the tools described earlier in the *technical design*, there would be no hint of originality. These choices too would not be based on any informed analysis of the particular *game design* being addressed. These would have been chosen regardless of what the *game design* turned out to be. And, indeed, within the *technical design* itself, the only relationship visible, between these choices and the components of the *game design*, would be a tangential one.

To disguise the lack of originality of these choices, the authors of the *technical design* may cover its flaws with overcomplexity. They may do this by conflating one method, used to display three-dimensional graphics, with another, used to model physics. Some theoretical methods used to display these graphics have some similarities with those used to model physics. One method from the former set and another from the latter may for example depend on dividing the *Game World* into smaller regions. So the authors may attempt to conflate the regions for the graphics with the regions for the physics. But the confusing result would merely be a crude attempt, to hide the unoriginality of the two choices for the graphics and physics, behind a contrived solution.

As for how the rest of the components of the *game design* would be built, the *technical design* would be virtually silent. How the two-dimensional graphics and text would be displayed or animated, how the commands would be issued through the *User Interface*, how the sounds and music would be played back, all of these issues would either be omitted. Or there would be a brief, speculative discussion of each topic, quickly followed by reassurances that these components were minor issues. As such, these would be best left, until later on in the production process, where these could be dealt with expediently. And, surely enough, these components would subsequently be cobbled together at the end of the process; at the 11th hour.

For example, the *technical design* would speculate on the combat component of the *game design* of *LPmud*, and how commands would be issued by the player. It would do this by describing the *User Interface* for another adventure game, based on a medieval theme. The description would include how commands were issued by the player in this other game. And the description would be accompanied by images of this existing *Interface*.

But the *technical design* would not be specific about which components of this existing *User Interface* would be copied. Other than, that is, to say that some of it may be useful. Likewise, there would be no mention of how the *software modules* would be built. So that commands could be issued by the player to communicate with other characters, give and steal items from other characters, pick up items off the floor, buy and sell items, or edit the *Game World*. Other than, that is, to say that these commands could be easily built during the process, or at the end of it.

The resolution of such issues, in the *technical design*, would not sell the game to its financial backer. Therefore, these would be expendable. What would definitely help sell the game would be the custom tools used to build it. Especially, the *game editors* that could be used to edit the different stages of the game, when the *game design* had changed, would be priceless. The greater the number of custom tools, the more obscure, the more exclusive, the more flexible and powerful the tool, the more impressive the

technical design would appear. So its authors would include a brief description of the set of small, custom tools that would be used to process the *data*, produced by the *Game Artists*, *Game Designers* and *Sound Designers*. This would be further illustrated by flow charts, showing how the *data* would be collected, merged or converted by each tool and transferred onto the next. And after several steps, each chart would show how each subset of tools would produce the final, compact and efficient *Game data* that would be used by the game. There would be a constant obsession with efficiency through out the document. But more on that obsession later.

Some of these small custom tools may already exist and have been used on past games. But others would not exist at all. And yet the *technical design* would not make the distinction between these two sets. Nor would it explain how those tools, which did not exist, would be built. Nor would it include a description of the *closed data format*, which each tool would read in or write out.

The centre piece of the tools would, of course, be the *game editors*. And the *technical design* would include a description of the capabilities of these tools. This would be supported by samples of the images, of the *User Interface* of each one. But the *game-editors* would have grown over a long *Software Evolution Process*, to meet the demands of several games from the past, which the *Software Developer* had worked on. During this time, the tool would have accumulated many features and become excessively complex and unstable. So many of the options visible on its *User Interface* would be redundant for the *game design of LPmud*. And yet the *technical design* would not indicate what these were.

Unfortunately, the authors, of this section of the *technical design*, would be given carte blanche, to add more features to these *game editors*. So that these may look even more impressive and thus, by implication, the *technical design* as well. One example of this would be the ability to write scripts, which altered the behaviour of some of the NPCs, or other items in the *Game World*. It has long been an obsession, in the Computer Games industry, to create a tool which could give the *Game Designers* total control over the editing process. So the *technical design* would include such features which pamper this obsession. And it would give examples of the kind of scripts which could be written, later on in the production process, to say modify the behaviour of the dragons described in the mountainous terrain in the *game design* of *LPmud*.

But, invariably, these brief examples would belie the huge amount of work involved. For the *Game Designers* would quickly find the scripts too limiting. And the *Programmers*, of the tool used to write these scripts, would end up attempting to crudely reinvent existing programming languages. Thus, on top of the difficulty of building a game with an *incomplete game design*,[10] would be added the difficulty of reinventing a programming language as well.

The *game editors* would be the last of the highlights included in the *technical design*, of a *Software Evolution Process*. Although, sometimes, the design may be padded out further, by illustrations which resemble a *software architecture*. But, in fact, these would be nothing of the sort.

For example, the *technical design* may include a box-and-line diagram, showing a subset of the *software modules*, that would be used to build the game. The diagrams may be built using tools based on the Unified Modelling Language (or *UML*[11]). Each box would represent a *module*. And it would include the name of the *module*, as well as the names of a subset of its *software procedures* and *data*.

Nevertheless, the diagrams would not include even a third of the *software modules* that would be used to build the game. Nor would even half of the *software procedures* and *data* within each *module* be named. Nor would any of those that were named be explained. Furthermore, the relationship between the *modules*, depicted in the diagrams by the lines connecting the boxes, would not be explained. Each line between two boxes would merely indicate an implicit relationship, between two *modules* which shared common properties. And it would not even give you any idea whether one *module* would be used by the other, let alone how. These diagrams are often depicted using heuristics called "design patterns" which the authors, just like UML diagrams, conflate with software architectures. But these are nothing of the sort. More on this later.

In contrast, in the **Event-Database Production Process**, the *technical design* would be written by the *Game Programmers* after consultation with the *Game Designers*, *Sound Designers* and *Game Testers*. It would be different in two major respects.

Firstly, it would only be concerned with the **Events**, **Actions**, **Game Objects** and **Host modules** of the **Architecture**. It would be concerned with how these would be used to build the components of the *game design*; at least those which had been included in the initial draft.

Secondly, it would be concerned with the system of **Events** or **Game Objects** that would be used, to edit the software, when the *game design* had changed. In these two respects, there are several advantages to demonstrating the application of the **Event-Database Architecture**, through the *game design* of *LPmud*.

The first advantage of *LPmud* is that it consists of a wide variety of smaller games that take place in the same world. These smaller games come in the forms of the puzzles, the quests and the challenges of strength and endurance that different characters and creatures in the world present. These games take place over a wide variety of locations. Some of these games are based on card games, board games, novels, films or folklore. Indeed, you can come across many anachronisms, because items from one period of history end up being mixed with items from another.

Furthermore, the game requires that the players be able to edit the *Game World* and add their own new ideas, once they reach the highest level of experience. The players may add more locations, more characters, creatures or other items lying around both new and old locations. Hence, the diversity within the game helps demonstrate how the **Event-Database Architecture** could support such diversity. And, therefore, how it could support dramatic changes to any other, given *game design*. Likewise, the ability to edit the *Game World* helps demonstrate how extensible the **Architecture** would be.

The second advantage of the *game design* of *LPmud* is that it introduces a second *software architecture*. This is namely the existing *software architecture* that was used to build *LPmud*. And that allows a comparison to be made between this practical *software architecture* and the **Event-Database Architecture**. So that the difficulty, or ease, of transferring a game from a practical *software architecture*, to a theoretical one, the **Event-Database Architecture**, could be illustrated.

The third advantage of the *game design* of *LPmud* is its availability. Both the *game design* and the existing *software architecture* that supports it are open and freely available to the public through the Internet. So anyone can practically examine, play with and compare this with the **Event-Database Architecture**.

The existing *software architecture*, which supports the *game design* of *LPmud*, has four components. The first component is a set of computer files that describe the *game modules*, and the *software library* these *modules* use, to implement the different features of the game, also known as a MUD Library or Mudlib. These features include the *User Interface*, any characters of the players, any other characters or creatures, any locations, any buildings, structures or items lying around in the *Game World*. The *game modules* and the *software library* are written in the *LPC* programming language and are independent of the computer hardware.

The second component of the *software architecture* is the software that is built on the computer hardware, also known as a Game Driver. You can see examples of the *LPmud* Game Driver on the Internet at these Web Addresses:

https://www.ldmud.eu/
https://www.dworkin.nl/dgd/
https://github.com/lnsoso/mudos
https://www.lysator.liu.se/projects/lpc4.html

This software interprets the *LPC* language. It also enables multiple players to connect to the computer. It presents the players with the *User Interface*, described by the first component of the *software architecture*. And it sends the response of the players, back to this first component.

The third component of the *software architecture* is a special *game module*, included in the first component, known as the **MASTER OBJECT**. This acts as a conduit between the first component and the second component. That is, it describes the relationship between the software, built on the computer hardware, and the *game modules*, which are independent of the hardware. It describes, for example, which *game modules* would be loaded into the computer memory first, when the game started. It describes which one of the *game modules* would be the entry point for the game and present the first screen of the *User Interface*. It describes what would happen to the *game modules* when the game was shut down. It describes which *game modules* could be edited by the players, and so on.

The fourth component of the *software architecture* is the documentation that accompanies the first component. These are computer files, written in natural language, including the *User Manual* for the players and other documentation. This includes a description of the *LPC* programming language. And it includes a description of the *software library*, in the first component of the *architecture*.

As well as this *software library*, any standard *software procedures*, used by the first component or the second component of the *architecture*, are described in this documentation. These standard *procedures* describe common properties that items, characters or locations in a game may have. These *procedures* either respond to a player issuing a common command, through the *User Interface*. Or these respond to a location which can have an effect on all the items or the characters within it. Or these respond to an item, in a location, which can have an effect on all the other items, or the characters, in that location.

For example, suppose you were making a **Game Object** that you wanted the players to be able to pick up. You would include the standard *software procedure* which

would allow that item to be picked up, called 'get', in the design of its *game module*. And this *procedure* would be used just before that item was picked up by a player. If you did not want the players to pick up a **Game Object**, you would simply not include that *procedure* in the *game module* for that **Object**.

Some of these standard *software procedures* are common to all subsequent games that have been built on the same *LPmud software architecture*. But others are made up for a particular game. For example, suppose one of the locations in the world was on top of a volcano. And in this location, the lava would periodically burst up, burning everyone, and destroying any items lying around nearby. If you wanted any item in the game, to describe whether it could be destroyed by the volcano, you may add a new standard *software procedure* for that called 'burn'. Any item, which had this 'burn' *procedure* in the *game module* describing it, would be destroyed. And those which did not would not be destroyed.

This is one of the principles of the *software architecture*. That is to say, any common properties that the locations, the characters or other items may share should be described through a standard *software procedure* with a given name. And that this name and *procedure* should be described in the documentation of the *architecture*.

The second principle is closely related. This is that if there were any existing component of the *software library*, of the *architecture*, that may be used to add a location to the game, then it should be used. If there were no existing components, and you believed future, or existing locations, may share common properties with this new one, then you should add a component to the *library* with these properties. And subsequently, you should use this new component to build the new location. The same principle applies to adding a character, a creature or some other item to the game. These additions should use existing components of the *software library* or add new ones where possible. Furthermore, any new component of the *software library* should be described, in the documentation of the *software architecture.*

The third principle of the *software architecture* is that the first component of the *architecture* should always be described using the *LPC* language. That is to say, the *User Interface*, the locations, the characters of players, other characters or creatures and other items in the *Game World* should all be described in a way that was independent of the computer hardware. The *User Interface* should provide the players with a way of writing or editing files in *LPC* for each of these items in the *Game World*. That would be translated or 'compiled' into pseudo machine code or **LPC CODE**. That in turn would be later executed when the player interacted with these items in the *Game World* by a **VIRTUAL MACHINE** or **VM**.

A **Virtual Machine** is a software simulation of a real machine or computer hardware. It simulates the execution of the instructions of the real Central Processor. Any software written for the real Central Processor can run in the **Virtual Machine**. The disadvantage is that this is slower than executing those instructions with a real Central Processor. But the advantage is that if any errors occur executing those instructions, only the **Virtual Machine** will stop operating. The software using the **Virtual Machine** to execute those instructions can carry on. In the case of the **VM** that runs **LPC code** it is very simple. It does not have as many parameters fed into it or inputs as a real machine. It can only take in numbers, texts, **Game Objects** or a list made up of these three types. And it does not have as many parameters coming

out of it or outputs as a real machine. It does not have to display anything on a screen or a *Graphical User Interface*. The result of executing any code is either a number, a text, an **Object** or a list made up of these three types.

The fourth principle is that every computer file, in the first component of the *software architecture*, should describe either one or more interchangeable parts of the *Game World*. That is to say, each file should either describe a location, a character, a creature or some other item that a player would visit or see. Or it should describe one part of a location, a character, a creature or some other item. But all of these should be interchangeable.

So, for example, a character can become an item, carried by another character in the game. Also, this item can become a location that contains other characters in the game. And, furthermore, even part of a location, a character, a creature or some other item can become an invisible item, somewhere in the *Game World*.

The fifth principle of the *software architecture* is that the game should not stop unless it has been explicitly shut down by someone administrating the game. That is to say, the game continues, 24 hours a day, and no errors brought about by the players, or otherwise, terminate the software. Also the game continues even while someone is editing its *User Interface*, the locations in the *Game World*, the characters, the creatures or other items. In other words, the game should be *failsafe*.[12]

This last principle receives reinforcement from the third principle and the second component of the *software architecture*. This component has been designed not to terminate, even when it detects errors while interpreting the *LPC* language. Further reinforcing this last principle is the fact that the second component rarely changes. And whenever an extension of the features it offers occurs, the new software that offers these extensions is kept separate from the second component. So that if these other software were to fail, the second component would still continue.

These five principles, along with the four components, form the *software architecture* of *LPmud*.

There is a diagram summarising the *software architecture* of *LPmud* in Figure 1.16. You can see the information exchanged between the components of the *architecture* in Table 1.1 and Table 1.2.

TABLE 1.1
Legend of Numbers Displayed in Figure 1.16

Data	Role
1	Description of the *software library* or MUD Library of locations, characters and other items in the *Game World*, written in natural language.
2	Description of the **Master Object** written in natural language.
3	Description of the **Master Object**, written in *LPC*.
4	Description of the *software library* or MUD Library of locations, characters and other items in the *Game World*, written in *LPC*.
5	**LPC code** or pseudo machine code, translated from *LPC*, that is executed by a **Virtual Machine** or **VM** to control the behaviour of items in the *Game World*.

It is a list of the information exchanged between the components of the *Software Architecture* of *LPmud*.

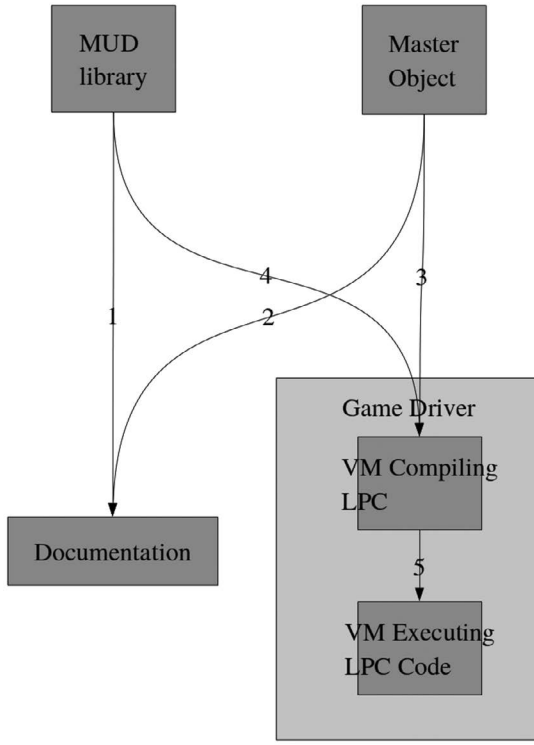

FIGURE 1.16 The original *software architecture* of *LPmud*.

TABLE 1.2
Comparison between the Original *Software Architecture* of *LPmud* and the Event-Database Architecture

***LPmud* Software Architecture**	**Event-Database Architecture**
Principle of reusing software through a *software library* (i.e. a MUD Library).	No rules about a *software library*.
Principle of reusing software through inheritance (i.e. **Game Objects** inheriting properties from other **Objects**).	No rules about inheritance.
Principle of simplicity through files (i.e. one file per **Game Object**).	Principle of simplicity through **Actions** (i.e. one **Action** per **Game Object**).
Principle of one programming language (i.e. *LPC*).	No rules about programming languages.
Principle of robustness (i.e. keep the game running 24 hours a day).	No rules about robustness. Principle of self-correction (i.e. skipping the execution of erroneous **Actions** once errors have been detected).

(*Continued*)

TABLE 1.2 (*Continued*)
Comparison between the Original *Software Architecture* of *LPmud* and the Event-Database Architecture

***LPmud* Software Architecture**	**Event-Database Architecture**
Greater simplicity due to less components, only 4 i.e. Game Driver, MUD Library, **Master Object**, Documentation.	Greater complexity due to more components, 9 i.e. 8 **Host Modules** and a **Game Database**.
Based on the *LPC* programming language.	Not based on any programming language.
Based on a Game Driver with a **Virtual Machine** or **VM** that 'compiles' or translates instructions in the programming language into **LPC code** or pseudo machine code.	Not based on any **VM** that 'compiles' or translates instructions in a programming language.
Based on a Game Driver with a **VM** that executes **LPC code** and can recover from errors executing code.	Not based on any **VM** that executes code. Therefore, it cannot recover from errors executing code.
VM can be built and run on different computer hardware.	Not based on **VM** but can be built and run on different computer hardware.
Text *User Interface*.	Text and Graphical *User Interface*.
User Interface only allows for commands through keyboard.	*User Interface* allows commands through keyboard, mouse, and other *Game Controllers*.
Game Database is a hierarchical *database*	**Game Database** is a *Relational Database*

Table 1.2 shows a comparison between the *software architecture* and the **Event-Database Architecture**.

From this table, you can see that *software architecture* of *LPmud* has some advantages, whereas the **Event-Database Architecture** does not have.

Firstly, the main advantage that the original *software architecture* of *LPmud* has is that it has less components than the **Event-Database Architecture.** And it is therefore more simple.

Secondly, the *software architecture* of *LPmud* has more principles. These principles help make the *software architecture* more robust and recover from errors.

Thirdly, it is based on a Game Driver that has a **VM** that can 'compile' or translate changes made to the **Game Objects**, written in the *LPC* programming language by the players, into **LPC code**. While the game was being played and without the need to shut it down. Even if there were errors during 'compiling' or executing instructions written in **LPC code**.

Fourthly, this Game Driver can start another **VM** that can execute **LPC code** whenever one **VM** fails. So if there were any errors executing the code of one **Game Object**, the Game Driver reports the error and starts another **VM** to carry on executing the code of other **Game Objects**.

Nevertheless, you can bring most of these advantages into the **Event-Database Architecture** by adapting or customising it.

Firstly, you can use a **VM** too to compile code for **Game Objects**, written in the *LPC* programming language by the players, into **LPC code**. And you can put the

results in the **Game Database** of the **Architecture**, next to the properties of each **Game Object**. Each **Game Object** has a *Database Field* called the **Game Object Code Field** which can be used to store 'compiled' **LPC code**. See the chapter **3.3 Objects Host** in the book **Event-Database Architecture for Computer Games Volume 1.**

Secondly, you can modify the **Objects Host** to start a **VM** to execute the **LPC code** of each **Game Object** it fetches from the **Game Database**, to perform **Actions** in response to **Secondary Events**. And if there were any errors executing the code, the **Objects Host** would report the error and start another **VM** to carry on executing other code. And the game can carry on running 24 hours a day without having to shutdown.

Thirdly, you could use the principles and components of the *software architecture* of *LPmud* to help you set the rules for generating the system of **Events** and **Game Objects** for building the game based on the Event-Database Architecture.

1.3.1 Rules for Generating the System of Events

The standard *software procedures*, which are used in the *software architecture* of *LPmud*, are analogous to **Events** in the **Event-Database Architecture**. These standard *software procedures* are used in *LPmud* to describe common properties that an item in the game may have with other items. These are also used to describe common properties that a character, or a location, in the game may share with other characters, or locations. Similarly, in the **Event-Database Architecture**, **Secondary Events** describe common properties that a **Game Object** may have with other **Objects**.

For example, if an item could be picked up in a game, then its **Game Object** would have a **Secondary Event** that the **Object** would receive. This **Event** would be received by the **Object** when a character tried to pick it up. And all other **Objects**, which could be picked up, would have a similar **Event**. And these would all follow on from the same **Primary Event**.

Likewise, imagine an item in a game that could be destroyed by the lava from a volcano. The **Game Object** of the item would have a **Secondary Event** that it would receive, when the volcano erupted. And all other similar **Game Objects** would have a similar **Event**. And these would all follow on from the same **Primary Event**.

So, like the standard *software procedures* of the *software architecture* of *LPmud*, **Secondary Events** describe common properties that a **Game Object** has. Thus, to produce the *technical design*, for the *game design* of *LPmud*, you could use these standard *procedures*. You could use these to decide on the initial set of **Secondary Events** that the **Game Objects**, of your **Event-Database Architecture**, would have. This would give you the following **Secondary Events**:

1. An **OBJECT INITIAL RESET EVENT**, which would be received by a **Game Object**, when it was loaded into the computer memory;
2. An **OBJECT PERIODIC RESET EVENT**, which would be received by a **Game Object**, typically once every 24 hours, when a character was returned back to its initial position, or a location was returned back to its original state;

3. An **OBJECT ENTERED EVENT**, which would be received by a **Game Object**, when it came into close proximity of a character;
4. An **OBJECT EXITED EVENT**, which would be received by a **Game Object**, when it moved away from a character;
5. An **OBJECT HEARTBEAT EVENT**, which would be received by a **Game Object**, when some fixed multiple of the *Unit of game time* had elapsed e.g. during each round of combat;
6. An **OBJECT MOVED EVENT**, which would be received by a **Game Object**, when it was about to be moved;
7. An **OBJECT TAKEN EVENT**, which would be received by a **Game Object** of an item, when that item was about to be picked up by a character in the game;
8. An **OBJECT DROPPED EVENT**, which would be received by a **Game Object** of an item, when that item was about to be dropped by a character;
9. An **OBJECT LOOKED EVENT**, which would be received by a **Game Object**, when a player looked at it in detail;
10. An **OBJECT INVENTORY EVENT**, which would be received by a **Game Object** of a character or container, when a player looked at the items being carried by that character or container;
11. An **OBJECT USED EVENT**, which would be received by a **Game Object** of an item, when a player wanted to wield that weapon, wear that piece of clothing or otherwise use that item being carried by the player's character;
12. An **OBJECT UNUSED EVENT**, which would be received by a **Game Object** of an item being wielded, worn or otherwise used by a player's character, when that item was no longer being used;
13. An **OBJECT HEARD EVENT**, which would be received by the **Game Object** of a character, when that character received a private or public message from other characters;
14. An **OBJECT ATTACKED EVENT**, which would be received by the **Game Object** of a character, when that character was about to be attacked by another;
15. An **OBJECT PACIFIED EVENT**, which would be received by the **Game Object** of a character, when that character had run away from combat, the character's opponent had fled or the opponent had died;
16. An **OBJECT DEAD EVENT**, which would be received by the **Game Object** of a character, when that character had died; and
17. An **OBJECT DESTROYED EVENT**, which would be received by a **Game Object**, when it was about to be removed from the computer memory or permanently from the Game World.

Each **Game Object** which needed one would have its own set of these **Secondary Events**. For example, not all **Game Objects** would require an **Object Heartbeat Event**. This would only be used by **Game Objects** which had to act spontaneously (without waiting for an **Event**). The **Game Object** of a NPC, which could move around the world, would use such an **Event**. And even then, it would only use the **Object Heartbeat Event** when it wanted to move around. When the character stopped moving around, the **Game Object** would stop itself from receiving this **Event**.

Another example would be two characters involved in a fight. Each character would use the **Object Heartbeat Event** to deal one blow to the other, during each round of combat.

Each of these **Secondary Events**, which a **Game Object** would receive, would be linked to a **Primary Event**. These would either be the standard **Primary Events**, generated by the **Host Modules**. Or these would be the new **Primary Events** that would be produced by one of the **Game Objects**, added to the **Event-Database Architecture**.

For the *game design* of *LPmud*, the initial set of **Primary Events**, including both the standard and the new ones, would be the following:

1. An **Initial Reset Event**, which would signal the beginning of a game, and more;
2. A **Connect Event**, which would be sent when a *Game Controller* was connected to the computer hardware;
3. A **Disconnect Event**, which would similarly be used as standard;
4. A **Controller Moved Event**, which would also be used as standard;
5. A **Controller Stopped Event**, which would be used as standard;
6. A **Controller Pressed Event**, which would be used as standard;
7. A **Controller Released Event**, which would be sent when a *digital device* on a *Game Controller* was released, and more;
8. A **Collision Event**, which would be used as standard;
9. A **Proximity Event**, which would be sent when a **Game Object** had come within, or moved beyond, a set area around another **Object**, and more;
10. A **LOADED EVENT**, which would be sent when new **Game Objects** had been loaded into the computer memory;
11. An **UNLOADED EVENT**, which would be sent when old **Game Objects** had been removed from the computer memory;
12. A **PERIODIC RESET EVENT**, which would be sent after a long interval had passed (i.e. 24 hours), and the moment had arrived to reset a character or item in the *Game World* back to its original position, or a location back to its original state;
13. A **HEARTBEAT EVENT**, which would be sent after a short interval had passed (i.e. less than ten times the *Unit of game time*), and the moment had arrived for a character, an item or a location, to act spontaneously;
14. A **MOVED EVENT**, which would be sent when an item or a character was about to be moved;
15. An **End Event**, which would be used to signal the end of a sequence of **Events**, and more; and
16. A **Shutdown Event**, which would be used as standard.

Most of the standard **Primary Events**, of the **Event-Database Architecture**, would not be linked to the initial set of **Secondary Events**. The exceptions would be the **Controller Released Event**, the **Proximity Event** and the **End Event**.

When the **Primary Controller Released Event** was sent, it would in turn send either **Object Taken Events** or **Dropped Events**, to the **Game Objects** of items. This

would depend on whether one of the buttons, just pressed and released, was a command to pick up or drop the items. Similarly, this **Primary Event** would send **Object Looked Events** to the **Game Objects** of items, characters or locations, when the player looked at these. It would send **Object Inventory Events** to the **Game Objects** of each character, or container, when the player examined what that character or container was carrying. It would send **Object Used Events** to each **Game Object** of an item being carried, when a character wielded that weapon, wore that piece of clothing or otherwise used that item. Likewise, it would send **Object Unused Events** to the **Game Objects** of these items, when these items were no longer being used.

The **Primary Controller Released Event** would also send **Object Heard Events** to each **Game Object** of a character, when another character sent that character a public or a private message. It would send **Object Attacked Events** to each **Game Object** of a character, when another character attacked that character. Finally, it would send **Secondary Events** to the **Game Object** of each shop, when an item was being bought or sold in that shop. And it would send **Secondary Events** to the **Game Object** of each player's character, when that player quit the game.

When the **Primary Proximity Event** was sent, it would in turn send **Object Entered Events** or **Exited Events** to the **Game Objects** of the items, the characters or the locations. This would depend on whether a player came near, or moved away from, these items, characters or locations.

When the **Primary End Event** was sent, it would in turn send **Object Pacified Events** or **Dead Events** to the **Game Objects** of the characters. These would depend on whether a character had run away from combat, the character's opponent had fled or the character had died.

Although not all the standard **Primary Events**, of the **Event-Database Architecture**, would be linked to the initial set of **Secondary Events**, all the new ones would be. These include the **Primary Loaded Event**, the **Unloaded Event**, the **Periodic Reset Event**, the **Heartbeat Event** and the **Moved Event**.

When the **Primary Loaded Event** was sent, it would in turn send the **Object Initial Reset Events**. These would be sent to the **Game Objects** of the items, the characters or the locations which had just been loaded in the computer memory.

When the **Primary Unloaded Event** was sent, it would in turn send the **Object Destroyed Events**. These would be sent to the **Game Objects** of the items, the characters or the locations which were about to be removed from the computer memory.

When the **Primary Periodic Reset Event** was sent, it would in turn send the **Object Periodic Reset Events**. These would be sent either to any **Game Object** of a character which needed to be reset back to its initial position. Or these would be sent to any **Game Object** of a location which needed to be reset back to its initial state.

When the **Primary Heartbeat Event** was sent, it would in turn send the **Object Heartbeat Events**. These would be sent to any **Game Object** of an item, a character or a location which needed to act spontaneously.

When the **Primary Moved Event** was sent, it would in turn send the **Object Moved Events**. These would be sent to the **Game Objects** of the characters or the other items which had just been moved.

As has already been mentioned, some of the standard *software procedures*, used in the *software architecture* of *LPmud*, are not used by other games based on that

architecture. The standard *procedure* for moving the characters or the other items in a game and the *procedure* for destroying the characters or the other items are two examples. And these are sometimes omitted from the design of the game. So, similarly, you could omit the **Object Moved Event** and **Destroyed Event** from the *technical design* of *LPmud*.

1.3.2 Rules for Generating the System of Game Objects

There would be only two sets of *software modules* in any *software design* based on the **Event-Database Architecture**. Similarly, in the *technical design*, that would be used to build the *game design* of *LPmud*; these would namely be the **Host Modules** and the **Game Objects**. The **Host Modules** have already been described in the chapter **The Software Architecture** in the book **Event-Database Architecture for Computer Games Volume 1**. From these descriptions, you would be able to decide what *software procedures* would be in these *software modules*. And you would be able to decide what *data* would be transferred from, and to, each *procedure*.

When it comes to the rest of the *software modules*, used with the **Architecture**, these would normally be determined by the system you choose for breaking down the features of a *game design* into **Game Objects**. Indeed, you may find it convenient to select the **Game Objects** before you decided on the **Events** you were going to use. Remember that both, the system of **Game Objects** and the system of **Events**, would merely be levels of *Abstraction*.[13] And in this respect both would have a similar function. This is not to say that one should ever be neglected for the other. But to say that both would merely be different ways, of providing a simple view of how the game would be built, by concentrating on some components of the **Architecture**, and ignoring others. And as such, it would not matter which one you decided to describe first.

However, in this particular example, the *software architecture* of *LPmud* already provides a system for choosing the **Events**, so you do not need to decide on the **Game Objects** first. Furthermore, the *software architecture* also provides a system for choosing the **Game Objects** of an **Event-Database Architecture** too. This comes courtesy of the fourth principle of the *software architecture*.

The fourth principle requires that each *game module* be associated with a location, a character or some other item in the *Game World*, and that all of these be interchangeable. This means, for the **Event-Database Architecture**, that it should be possible for all **Game Objects** to become a location, a character or some other item in the *Game World*. This may at first hand seem to be a problem or counter-intuitive. But this becomes clearer when you realise that most games do depend on visible locations, characters and other items, which interact with each other, within one consistent world.

The *software modules* used in most games can be broken down into two sets. The first are the set which directly control or affect the visible *Game World*. These include the different locations, the characters and the other items in each location. The second are the set which support the visible *Game World*. These either manage, or are used by, the visible locations, the characters or other items in the world. It would seem natural for the first set to be placed in the *Game World*, but for the

second set it seems unnecessary. It could also cause undesirable side-effects if the second set were somehow to become visible to the players. Or it could cause inconsistencies if the second set interacted with the *Game World*, like the visible characters or other items.

However, there would be at least two ways to solve this problem, within the **Event-Database Architecture**. Either you could simply combine *software modules* from the second set, with the first set. So each *software module* in the game would be associated with a visible location, character or other item. Or you could attach the second set of *modules* to invisible items, which neither the bodies in the visible *Game World*, nor the *User Interface*, could interact with. So that if, for any reason, any of these were to appear in the *Game World*, it could not interfere with the game.

Either of these two methods has been used in the *software architecture* of *LPmud*. Each method has the advantage that you can easily find and fix any errors by following it. This comes courtesy of the fact that each location, character or other item in the game has one *software module* associated with it. So whenever you encounter an error with a location, a character or another item, you simply find the *software module* associated with it and fix it. This is made easier if it is possible for you to see the identity of the *software module*, used to create each location, character or item, as you move through the game. And, indeed, you can create a tool, with the *LPC* language, that allow you to do just that. This is called a Scan Tool in the *LPmud software architecture*. This identifies all the files used to create each visible and invisible **Game Object** in a location, character or container. But in the **Event-Database Architecture**, since you identify **Game Objects** by the Primary Key in the **Database**, and this is related to the **Database Host**, it is better called **INTERNAL DATABASE HOST QUERY CUSTOM TOOL**. You can add this tool as an item in a Game World, which the player or Wizards (i.e. highest level players or *Software Developers* who administer the game) can use.

The Scan Tool allows the player or Wizard to see the identity of even invisible **Game Objects**. So even if you use the latter of the two methods, you can still easily find and fix errors. The *software modules* that only constitute one part of a location, a character or another item would appear as invisible items in the *Game World*. However, the tool can scan the contents of any location, character or container and reveal the identity of every visible or invisible items in it. The tool can also display all the *software modules* that were used to build each location, character or item in the *Game World*.

Similarly, for the **Event-Database Architecture**, you could choose your **Game Objects** based on the fourth principle of the *software architecture* of *LPmud*. You could either associate each location, character or other item with one visible **Game Object**. Or you could compose each from one visible **Game Object** and several invisible **Game Objects**, which the locations, the characters or other items could not interact with. You could also include one **Game Object** that would act as an **Internal Database Host Query Custom Tool**. This tool would appear as an item in the game that could be used by the players or Wizards. It would identify the **Game Object** of any visible location, character or other item, by the *Primary Key* of the *Record* which held the properties of that **Object**. And it would also identify any visible or invisible items amongst the contents of that location, character or other item.

The fourth principle, of the *software architecture* of *LPmud*, still applies even when the game does not have a conventional world. Since the *game design* of *LPmud* incorporates a world of magic and fantasy, this is often the case. That is to say, the game does not have one consistent world. Instead, the *Game World* is made up of rooms. And each room may be used to symbolise as large or as small a space as you like. This means either different spaces in the *Game World* may overlap. Or the *Game World* may vary from one set of dimensions to another, from two dimensional space, to three, or four dimensional space.

For example, suppose you reached that public house, described earlier in the *game design* of *LPmud*. And you could join the other characters gambling at the table, in a game of poker. Now such a game could be represented on a computer, in one consistent three-dimensional world. But it would not be necessary. All that matters would be the hand each player had, the bets placed and the size of the *pot*. So the *User Interface* for such a game could simply list the hand each player had, the bets, and the name of each player, next to each hand or bet. The position of the cards, displayed on the computer screen, would not have any bearing on where these were in the *Game World*. Nor would these positions reflect where the players were. These positions would merely be symbolic, not literal.

In a multiplayer version of such a game, the view each player had of the space containing the cards would not be consistent. When the game began, from one player's view, only his or her cards would be visible: all the cards of any opponent would be face down. But, for each opponent, that player's cards would be face down, as would all others, except his or her cards.

Although it would be difficult to represent such a game, according to the fourth principle of the *software architecture* of *LPmud*, it would not be impossible. The first obvious solution would be to have one set of **Game Objects** for each player and use the computer screen as the *Game World*. So that each **Object** would be associated with one card, as seen by one player in the game. And the position of each **Object** on the computer screen would reflect the position of the card in the *Game World*. Then, depending on who was looking at the cards, one set of **Game Objects** would be shown, and the rest would be invisible. Furthermore, the cards in the visible set, belonging to that player, would be shown face up. The rest of the cards would be shown face down.

Alternately, you could have one set of **Game Objects** for all the players. Then, depending on whether a player could see a card or not, each **Game Object** would change its appearance. Each card would either be shown face up or face down.

The latter of the two solutions would be the one which most commercial games, based on the *Software Evolution Process*, would adopt; in order to reuse as much software as possible. However, both solutions mean that the appearance of each **Game Object** would depend on the player. This conflicts with the first and the fourth principles of the *software architecture* of *LPmud*.

Remember that the first principle requires that any common properties that the locations, the characters or other items shared should be described using a standard *software procedure*. One of these standard *software procedures* describes the appearance of a location, a character or some other item. It displays the item to the player, when he or she looks at it. But, for this game of poker, both solutions require that the appearance of each card depends on the player. So the standard *software*

procedure displaying that card cannot give a consistent description. Furthermore, any two players who see that card cannot recognise that it comes from the same *software module*. These two failures violate the first and the fourth principles of the *software architecture*, respectively.

This is why the *software architecture* of *LPmud* would favour a third solution. A solution which kept any two players' view of the cards separate would be consistent with its principles. This would have to be a solution in which the appearance of a card, to the players involved in the card game and those not in the game, remained the same. This would be better suited to the **Event-Database Architecture** too. It would ensure that **Events** from one view did not inadvertently affect another. Besides, unlike the *software architecture* of *LPmud*, a **Game Object** could not tell when it was being viewed, and by whom.

The third solution would involve placing different symbolic views, of the card game, in separate locations of one *Game World*. So that each location would contain the view of the cards as seen by one player involved in the card game. And if any other players came to that location, they too would see the same cards. Each view would be isolated from other views, by a natural or an artificial barrier. Either each view could be isolated by great distances, to prevent **Game Objects**, placed in one view, conflicting with **Objects** in another view. So, for example, an **Object Entered Event**, an **Object Exited Event** or an **Object Moved Event**, for one set of **Objects**, would not cause similar **Events** for another set of **Objects**. Or each view could be isolated by a physical barrier around it. So that the **Game Objects** were prevented from escaping to, or entering from, another view.

1.3.3 Application: Visible and Invisible *LPmud* Game Objects

The fourth principle of the *software architecture* of *LPmud* does provide a useful requirement for choosing **Game Objects**. However, the breakdown of the **Game Objects**, used to implement the *game design* of *LPmud*, would have three more requirements to meet. These requirements would apply to any game that used the **Event-Database Architecture** too. These come from the description of the **Physics Host**, the **Graphics Host** and the **Game Controllers Host**.

The first requires, amongst other things, a **Game Object** to have a mass, a position, a speed and an acceleration, in order to move it. The second requires, amongst other things, visible **Objects** to have a *Texture* or a 3D model. It also requires two **Camera Objects**, in order to view the *Game World*. The third requires, amongst other things, an **Object** to have numerical properties (e.g. a position) which could be manipulated by a *Game Controller*. It also requires this **Object** to have a set of **Secondary Events** it would receive, when an *analogue device* or a *digital device* was being used.

So, considering all these requirements, the properties of the **Game Objects**, for the *game design* of *LPmud*, would be the following:

1. An **INVISIBLE 2D POINT OBJECT**. This would be used to mark an important point in 2D space. Its properties would be stored in a special **POINT OBJECT RECORD**. It would have a mass, the position of a point

in a 2D world and an orientation (i.e. its rotation about its centre). It would have horizontal and vertical speeds (or X and Y speeds). And it would have X and Y accelerations, a rotational speed and a rotational acceleration. It would also have the shape of its *Collision boundary*[14] (e.g. square circle or some other 2D shape), the shape of its *Proximity boundary*,[15] a **Collision Event** and a **Proximity Event**. Lastly, it would have the **Object Initial Reset Event** and the **Object Destroyed Event** that it would receive. These would be sent when it was loaded into, or removed from, the computer memory.

2. An **INVISIBLE 3D POINT OBJECT**. This would be used to mark an important point in 3D space. Its properties would also be stored in a special **Point Object Record**. It would have a mass, the position of a point in a 3D world and an orientation. The orientation would include a pitch, a yaw and a roll (i.e. a rotation about its local X, Y and Z axes). It would have a speed along the breadth, the height and the depth of the *Game World* (i.e. an X, Y and Z speed), and an angular speed around its local X, Y and Z axes. It would also have an X, Y and Z acceleration, and an angular acceleration around its local X, Y and Z axes. And like a **2D Point Object**, it would have the shape of its *Collision boundary* (e.g. sphere, cube, cylinder or another 3D model), the shape of its *Proximity boundary*, a **Collision Event** and a **Proximity Event**. It would also have the **Object Initial Reset Event** and the **Object Destroyed Event** that it would receive.
3. A **Master Object** derived from (1). This would monitor the occurrences of any non-standard **Primary Events** that would be used by the **Event-Database Architecture**. And it would send these Events to the Events Host. It would have the oldest **Game Object** loaded into the computer memory, and the last **Object** loaded into the memory. It would also have an **Object Heartbeat Event**, and an **Object Periodic Reset Event**.
4. A **TEXT OBJECT** derived from (1). This would be used to display words at a given point in a 2D world. Each **Object** would refer to a **TEXT LOCALISATION RECORD** that held the words in different regional languages. It would have the font, the shape and the position of its characters. It would also have the *Texture coordinates* of its characters in the font, the words of the text, its colour, its size and its width.
5. A **2D IMAGE OBJECT** derived from (1). This would be used to display an image at a given point in a 2D world. This includes an icon, a picture or an item on a menu. This also includes a location in the game, an item, a building or other structures in a location. It would have the *ID* of the image that would be displayed and its shape. It would also have the *Texture coordinates* and the size of the image.
6. A **2D ANIMATION OBJECT** derived from (5). This would be used to display the animation of an item, a character, a building, other structures or a location, at a given point in a 2D world. It would have the *ID* of each *polygon*, within which the images (or *Frames*) would be displayed. It would have the *ID* of each *Frame* that would be displayed. It would have the *ID* of the *Texture coordinates* of each *Frame*. It would have the rate at which the

Frames would be displayed, how long a single animation sequence would last and how much time had elapsed since the sequence started. Finally, it would have a **Secondary End Event** that would be sent when the sequence had finished.

7. A **2D Player Object** derived from (6). This would be used to display a player's character, other characters or creatures, in a 2D world. It would have the properties of the *Game Controller* that would direct the character. These would include a list of the *analogue devices* and the *digital devices* that affected the properties of the character. These would also include a list of the properties (e.g. the speed of the character) that were affected by these *devices*. It would include how much these properties were affected, when the *analogue devices* were moved to the highest and the lowest point, along an axis. And it would include how much the properties were affected when the *digital devices* were pressed. It would include the range of movement, about the default position, within which an *analogue device* would be ignored. It would include a history of the *analogue devices* that had been moved, and the position of the *devices* each time this occurred. It would also include a history of the *digital devices* that had been used, and the times these were used. Finally, it would include the **Secondary Connect**, **Disconnect**, **Moved**, **Stopped**, **Pressed** and **Released Events** that the **Object** would receive, when the *Game Controller* was manipulated. Along with the properties of the *Game Controller*, the **Object** would have the properties of the character. These would include how much health points the character had left, a list of the items it was using, a list of the items it was carrying in its inventory, and its score.
8. A **3D MODEL OBJECT** derived from (2). This would be used to display a 3D model of an item, a building, other structures or a location, at a given point in a 3D world. It would have the *ID* of the model that would be displayed, its *Texture*, a set of *Texture coordinates* and the size of the model.
9. A **3D ANIMATION OBJECT** derived from (8). This would be used to display the animation of an item, a character, a building, other structures or a location, at a given point in a 3D world. It would have the changes of *vertices*, between each *Frame* in the animation sequence and the size of each set of changes. It would have the rate at which the *Frames* would be displayed, how long a single sequence would last and how much time had elapsed since the sequence started. It would also have a **Secondary End Event** for when the sequence had finished.
10. A **3D Player Object** derived from (9). This would be used to display a player's character, other characters or creatures, in a 3D world. It would have the properties of the player's *Game Controller*. These would include a list of the *analogue devices* and *digital devices* that affected the properties of the character. These would also include a list of the properties (e.g. the speed of the character) that were affected by the *devices*. It would include how much these properties were affected, when the *analogue devices* were moved to the highest and the lowest point, along an axis. And it would include how much the properties were affected when the *digital devices* were pressed.

It would include the range of movements, about the default position, within which an *analogue device* would be ignored. It would include a history of the *analogue devices* that had been moved or stopped, and the position of the *devices* each time this occurred. It would also include a history of the *digital devices* that had been used, and the times these were used. And it would include the **Secondary Connect**, **Disconnect**, **Moved**, **Stopped**, **Pressed** and **Released Events** that the **Object** would receive, when the *Game Controller* was manipulated. As well as the properties of the *Game Controller*, the **Object** would have the properties of the character. These would include how much health points the character had left, a list of items it was using, a list of items it was carrying in its inventory and its score.

11. A **2D Camera Object** derived from (1). This would be used by the **Graphics Host** to display a view of a 2D world. It would hold the position of the camera, the width and the height of the area visible around it.
12. A **3D Camera Object** derived from (2). This would be used by the **Graphics Host** to display a view of a 3D world. It would have the angle of the *Field of View*,[16] a *near and far focal length*.[17]

As well as the properties already described, each copy of these **Game Objects**, in the *Game World*, would have an *ID*. This *ID* would be the *Primary Key* of the *Record* which held its properties, in the **Game Database**. The *ID* would be used throughout the **Event-Database Architecture**, by the **Host Modules** and all other **Game Objects**, to refer to that **Object**.

The first **Game Object** that would react when the game started would be the **Master Object**. When it received the **Primary Initial Reset Event**, the **Master Object** would start generating the **Primary Heartbeat Event** periodically, for other **Game Objects** to use. Any **Game Object**, which wanted to respond to this **Event**, would add its own **Object Heartbeat Event** onto the list of **Secondary Events** for the **Primary Heartbeat Event**. The **Master Object** itself would always have its **Object Heartbeat Event** on this list. It would use this to check when the other new **Primary Events** had occurred. These would namely be the **Primary Loaded Event**, the **Unloaded Event**, the **Periodic Reset Event** and the **Moved Event**.

The oldest **Game Object**, and the last **Game Object**, loaded into the computer memory would be part of the properties of the **Master Object**. As mentioned in the basic description of the **Event-Database Architecture**, the **Database Host** has two *Database Records* in the **Game Database**. One is called the **Residents List Record** and the other is called the **Absents List Records**. These *Records* would keep a list of all the other *Records* in the **Game Database** currently loaded into, or unloaded from, the computer memory. So that the **Database Host** could tell when a *Record* was being accessed that was not in the memory. The **Master Object** too could use these two *Records*.

The **Master Object** could use these two *Records* to track the set of **Game Objects** loaded into, or unloaded from, the computer memory. And when it detected that the set had changed, it would send the **Primary Loaded Event** to the **Events Host**, for the **Game Objects** that had just been loaded. Or it would send the **Primary Unloaded Event** for the **Game Objects** which had just been unloaded.

Before sending the **Primary Loaded Event**, the **Master Object** would first gather a list of the **Object Initial Reset Event**, of the **Game Objects** which had just been loaded into the memory. It would then add these to the list of **Secondary Events** of the **Primary Loaded Event**. Similarly, before sending the **Primary Unloaded Event**, it would gather a list of the **Object Destroyed Events**, of the **Game Objects** which had just been removed. And it would add these to the list of **Secondary Events** of the **Primary Unloaded Event**.

The **Master Object** could compile a list of the **Game Objects** just loaded into, or unloaded from, the computer memory by using the list of the *Records* currently loaded or removed from the memory. Both of these lists could be ordered so that the latest addition was at the beginning, and the oldest was at the end. Hence, from knowing the oldest and the latest member of the set of **Game Objects** loaded into the computer memory, the **Master Object** could tell when the set had changed. This would be when the oldest **Game Object**, or the latest **Game Object**, had changed. And the list of the **Objects** removed would include the previous oldest **Game Object** and all the **Objects** after it, in the list of *Records* removed from the memory. Likewise, the list of the **Objects** loaded into the memory would include all the **Game Objects**, after the previous latest **Object**, in the list of *Records* loaded into the memory.

As well as its **Object Heartbeat Event**, the **Object Periodic Reset Event**, of the **Master Object**, would be amongst those sent when the game started. This **Secondary Event** would have a delay equal to the length of the intervals between each **Primary Periodic Reset Event** i.e. 24 hours. Each time it received the delayed **Object Periodic Reset Event**, the **Master Object** would send the **Primary Periodic Reset Event**. This in turn would cause the **Object Periodic Reset Event**, of the **Master Object**, to be sent again. So the **Master Object** would use this to continuously generate each **Primary Periodic Reset Event** that the other **Game Objects** would use.

Finally, the **Master Object** would periodically search through the list of **Game Objects** being moved by the **Physics Host**. Remember that the description of the **Physics Host** required a *Record*, in the **Game Database**, that held a list of **Game Objects** whose movements it would control. If the **Master Object** detected any **Game Objects** on this list with any amount of speed, it would assume that the **Object** was about to be moved. So it would add the **Object Moved Event**, of that **Game Object**, onto the list of **Secondary Events** of the **Primary Moved Event**. This would be provided that the **Event** was not already on the list. After it had similarly searched the entire list being moved by the **Physics Host**, the **Master Object** would send the **Primary Moved Event**.

Once a **Game Object** had no speed, or had been removed from the list being moved by the **Physics Host**, it would presumably have stopped. So the **Master Object** would remove its **Object Moved Event**, from the list of **Secondary Events** of the **Primary Moved Event**. One **Game Object** could conceivably move a second **Object** without using the **Physics Host**. In that case, the first **Game Object** would have to add the **Object Moved Event** of the second one, onto the list of **Secondary Events** for the **Primary Moved Event**. And it would remove the **Event** from that list, once it had stopped moving the second **Object**.

The movement of some **Game Objects** may be accompanied by animation. For example, a **3D Player Object** could be required to show an animation of the player's character walking, each time it was moved. But, by design, the **Player Objects** would only show static characters. So in order to show an animated character, a **3D Animation Object** could be superimposed onto the **3D Player Object**. That is, both **Objects** would always share the same position, as well as speed and acceleration in the *Game World*. But only the **3D Animation Object** would be visible at that position. And either each time a *Game Controller* was used to move the **Player Object**, it would in turn pass that movement onto the **Animation Object**. Or each time the **Player Object** received its **Object Moved Event**, it would pass on its movement onto the animated **Object**.

Of course the *game design* may change. The *game design* of *LPmud* offers the Wizards the ability to add their own areas to the game, once they have attained the highest level of experience that can be achieved. So they could add more elements to the *User Interface*. They could add more locations to the *Game World*. They could add more characters, creatures and items lying around in these new locations. All of these additions would require more **Events** and **Game Objects**.

However, whatever additions the Wizards wanted to make, they should use the same system for selecting **Game Objects** that has been used so far. Every visible item lying around, character, creature, building, other structures, or location should have one visible **Game Object**. Each of these may work with several other invisible **Game Objects**, depending on how complex it was. These invisible **Game Objects** may either be unique to that visible **Game Object**. Or these may be used by multiple visible **Game Objects**. Nevertheless, in the former case, these invisible **Game Objects** should be placed in the same location, in the *Game World*, as the visible **Game Object**. So that any Wizard that goes to that location can use an **Internal Database Host Query Custom Tool** to scan the location and find all of the **Game Objects** responsible for that location. And in the latter case, the invisible **Game Objects** should all be placed in one central location e.g. the origin of the *Game World*. And all similar, invisible **Game Objects** should be placed in that same location. So that any Wizard that goes to that location can use the **Internal Database Host Query Custom Tool** to scan the location, and find all of the shared **Game Objects** responsible for the *Game World*.

When adding to the *game design*, the Wizards should also use the same system for selecting **Events** that has been used so far. Each common property (i.e. common action) that may be performed on an item, a character, a creature or a location should have a common **Secondary Event**. Some of these common actions may occur naturally. For example, an item or a location may disappear and reappear at various times by magic. Or an item or a character may be destroyed by a hazardous location. Each of the **Game Objects** of the items, characters, creatures or locations should have a similar **Secondary Event** for when these common actions occur.

Some of these actions may also occur artificially. These would namely be when commands were issued by the characters (or creatures) in the game. For example, an item or a creature may appear in a shop, when it has been bought by a player's character. If a character could interact with another character, an item, a building, other structures or a location, then that character should have the commands available to

do so. And each of these commands should have a **Secondary Event**. Each **Event** should be received by the **Game Object** of the other character, the item, the building, the structure or the location, when the command was used.

The **Primary Event** for the commands used by a character should be the standard **Controller Released Event**. And this should be produced by the 2D or 3D **Player Object** of that character. This would be consistent with the system used so far. For all new **Secondary Events**, an existing **Primary Event** should be used where possible. Only when it would not be consistent to use any of the existing **Primary Events**, to generate a **Secondary Event**, should a new one be added for that purpose.

1.3.4 Application: AI with Path Finding

It may seem strange how you could implement some features of game using a set of **Events**, and **Game Objects**, operating in a *Game World*. One such example would be a character, in the game, controlled by the computer, or an NPC.

An NPC would normally have an appearance and spacial properties. So it could naturally be represented by one of the **Game Objects**, selected using the fourth principle of the *software architecture* of *LPmud*. This requires all *game modules* (and hence **Game Objects**) to have spacial properties. That is to say, it should be possible for each **Game Object** to be placed in the *Game World* and interact with the world and other **Objects**. But the most difficult challenge, posed by an NPC, would be how to make it act intelligently. Intuitively, this would seem to require a lot of purely logical *software procedures*, which used no spacial *data*. That is to say, this would seem to require lots of decisions to be made, with lots of *logic branches*. Therefore, you could require a lot of **Events** and **Game Objects** in the **Event-Database Architecture**.

Unfortunately, what those in the Computer Games industry who follow the *Software Evolution Process* call *Artificial Intelligence* suffers from the same lack of definition, as the very games that they spawn. In the Computer Games industry, the word 'Artificial' is taken in this context to mean 'the illusion of', as supposed to 'a substitute for'. An illusion only has to appear like something else. An illusion does not have to function like it. However, a substitute does not have to appear as something else. More importantly, a substitute must function just like it. An illusion is effective while you do not interact with it. Once you do interact with it, how it functions, or does not function, distinguishes it from what it is imitating. The time taken to reach this point, however, can be extended by adding more and more layers of illusions. And this is what happens in the Computer Games industry.

In the beginning, all that would be asked of the *Artificial Intelligence* (or AI) would be that it should just manage to play the game, like a normal player. Later on, when it has become apparent that it would be easy to beat the AI, the AI would be required to reach a state where it could not be easily beaten. After that, when the AI has consistently beaten all players, it would be required to vary its performance, so that it would play badly when it was winning, and unbelievable well when it was losing. Playing badly, when it was winning, would mean that the AI has to periodically act irrationally. Playing unbelievable well, when it was losing, would mean that the AI has to cheat. So the AI would start of being asked to play like a normal player and

provide a challenge. But in the end, it would neither behave intelligently, nor would it play like a normal player. It would merely become a vehicle for giving the player the illusion of a challenge.

The complexity of creating the illusion of intelligence blurs the limitations of the AI being developed for a game. The complexity turns it into an abyss into which schedules enter, never to return. Another result, of the complexity, is that the AI is not thought of as a simple logical, reasoning machine. Instead, it relies on spacial *data*. This *data* closely relates to some other visible spacial *Game data*, such as the 2D or 3D shape of the *Game World*. Whenever this *Game data* changes, the *data* required for the AI needs to change as well. Editing this *data* wastes a lot of time because of this close relationship. This is bad enough. But even worse can happen.

Since the development of the AI proceeds through more of an experimental process, rather than a carefully planned one, unforeseen problems can occur at a later stage. For example, the AI may have trouble negotiating some part of the *Game World*, because of an awkward obstruction. The AI could be revised to cope with the problem. But at the latter stages of the *Software Evolution Process*, the complexity of the AI becomes unwieldy. So instead, the obstruction either disappears from the *Game World*, or the *Game World* changes to prevent the possibility of the problem occurring. At this point, the AI becomes something the game must negotiate, rather than the other way round.

However, the end result is that you could use **Game Objects**, with spacial *data*, to implement the typical, crude *Artificial Intelligence* used in the Computer Games industry. A brief example of how this could be done would be in the giant arena described earlier, in the *game design* of *LPmud*, where the player had to race against other chariots.

To get an AI to navigate a track around the arena, you could define a set of points or **Game Objects** (sometimes called *Waypoints*[18]) around the track. These should be placed, along a lane, travelling around the track in the order that you would need to follow them to complete a lap. The points should be numbered in the order the chariots travel around the track. The position, the ordinal number, the lane and other properties of each *Waypoint* would be stored in its **Game Object Record**.

When a race starts, the AI would simply drive the chariot in a straight line towards the first *Waypoint* in the lane the chariot was in. This would be its initial target. The AI would use the commands i.e.

Forwards Command
Backwards Command
Turn Left Command
Turn Right Command

To head towards this target or *Waypoint* along the track. By sending the six standard **Secondary Events**

Secondary Connect Event
Secondary Disconnect Event

Secondary Controller Moved Event
Secondary Controller Stopped Event
Secondary Controller Pressed Event
Secondary Controller Released Event

To the **Master Player Object** to simulate the *analogue devices* or *digital devices* on the *Game Controller* being used to evoke the commands.

And it would include the following in the properties of these **Secondary Events**:

1. the cause of the **Event** was the **2D Player Object** or **3D Player Object** of the character being controlled by the AI
2. the unique word identifying the *analogue device* or *digital device* of the *Game Controller* that had been connected, disconnected, moved, stopped, pressed or released to evoke a command
3. the amount the *device* had been moved along its axis.

And the **Master Player Object**, in turn, would respond to these **Events** by modifying the physical properties of that **Player Object**. And forwarding these **Events** onto that **Player Object**. To get the character's chariot to accelerate, decelerate, turn left or turn right towards the target.

When the character came within a set distance of this, the target would change to the next *Waypoint* in the lane. And the AI would drive the chariot in a straight line towards that *Waypoint*. And when the AI came within distance of this, the target would change again to the next *Waypoint*, and so on. Using this method, the AI could follow a path you lay out on a lane around the track and complete a race.

You could add more sophistication to this basic model by using parallel lanes of *Waypoints*. This would allow the AI to change lanes to avoid other chariots (or obstructions) on the track. It could detect these either by keeping track of the chariot or obstruction at each *Waypoint* in its properties held in its **Game Object Record**. And then simply examine this **Record** to see if another chariot or obstruction was on that *Waypoint*. Or you could check the proximity of other chariots to the chariot controlled by the AI. You could go on and on, adding more *data* and more *data*, to this basic model, which the AI could use; to provide more and more layers of illusion. There are several *Illusion of Intelligence sources*[19] which describe how you could build on this model.

This kind of system would be easy to implement within the bounds of an **Event-Database Architecture**. A set of visible and invisible **Game Objects**, in 2D or 3D space, could be used to represent the tracks, the chariots, the obstructions and the *Waypoints*. The current *Waypoint* of each chariot could be held, with its properties, in the **Game Database**. The **Physics Host** could be used to move the chariots. The **Primary Proximity Events** could be used to change to the next *Waypoint*, when the AI came within close proximity of its current *Waypoint*. These **Events** could also be used to change lanes, when the AI was in close proximity of another chariot and in danger of crashing into it, or an obstruction on the track.

1.3.5 Application: AI with Neural Networks

There is a field of Computer Science which deals with *Artificial Intelligence*. In this academic field, AI is defined as an attempt to model the human brain, or to create a system that can make deductions. That is to say, given two facts, can a computer determine whether a third fact is true? This requires a *Database*, which can hold these facts, to be designed in such a way as to allow a computer to make these deductions. But, to date, all attempts at this have failed due to the amount of *data* required.

However, the quest to achieve this goal has yielded models which could be used to control an NPC. One of these is *Artificial Neural Networks*.[20] These networks have been modelled on the cells of the human brain. A real neural network is made up of a network of brain cells, called *Neurons*.[21] These *Neurons* combine together to perform all the high-level functions associated with the human brain, or so the theory goes. These include receiving and analysing various sensory information (e.g. sight, sound and touch) These also include producing human reactions (e.g. memorisation, rationalisation, speech and the movements of different parts of the body). But how exactly these *Neurons* perform these functions is unclear. Since there are about 86 billion *Neurons* in the human brain.

An *Artificial Neural Network* is made up of a network of *Artificial Neurons*.[22] This network is made up of several layers, of *Artificial Neurons*, connected together. An *Artificial Neuron* in each layer takes one or more inputs and produces an output which feeds into the next layer. The first layer takes the **NEURAL NETWORK INITIAL INPUTS**. And the final layer produces the **NEURAL NETWORK FINAL OUTPUTS**. Both the **Initial Inputs** and the **Final Outputs** are numbers, which represent two interrelated pieces of *data*.

Thus, take, for example, the race in a circus referred to in the *game design* of *LPmud*, where the player had to take part in a race against other chariots. You could construct an *Artificial Neural Network* which took important factors on the tracks, as the **Initial Inputs**, and produced the factors affecting a chariot, as the **Final outputs**.

The **Initial Inputs** would include

- the distance travelled along the track
- the distance from the sides of the track
- the closest distance of other racing chariots along the track
- the relative position of other chariots
- the relative position in the race
- the speed of the chariot

The **Final Outputs** would be

- the position of the *analogue devices* which controlled the acceleration of the chariot
- the position of the *analogue devices* that controlled the deceleration or brakes of the chariot
- the position of the *analogue devices* that controlled steering of a chariot to the left or right

- the on-off state of the *digital devices* i.e. buttons which controlled the acceleration of the chariot
- the on-off state of the buttons that controlled the brakes
- the on-off state of the buttons that controlled the steering to the left or right

Each input of an artificial *Neuron* has a numerical **NEURAL NETWORK NEURON INPUT WEIGHT** associated with it, which multiplies it. The **Weight** indicates how important that input is in affecting the **NEURAL NETWORK NEURON OUTPUT**. The **Output** of every artificial *Neuron* is controlled by the same mathematical function or **NEURAL NETWORK ACTIVATION FUNCTION**.

When you teach an *Artificial Neural Network*, you feed it with **NEURAL NETWORK TRAINING DATA**. This is a set of known **Initial Inputs** and known **Final Outputs** of the biological Neural Network which the *Artificial Neural Network* is meant to emulate. You feed the known **Initial Inputs** from the **Training Data** into the *Artificial Neural Network* and compare the results with the **Final Outputs** in the **Training Data**. The difference between the results and the **Final Outputs** is the error. And you use this error to adjust the **Weights** of the *Artificial Neurons* to correct the results. The goal is to produce an *Artificial Neural Network* that understands, or at least mimics, the relationship between the **Initial Inputs** and **Final Outputs** in the **Training Data**. So well that it can predict what the **Final Outputs** will be whenever the **Initial Inputs** change. There are several teaching methods for doing this. The most basic one is an algorithm called **NEURAL NETWORK BACK PROPAGATION** or **Back Propagation**.

1.3.5.1 Application: Back Propagation

Consider the example of a race of chariots around a circus mentioned earlier. In its simplest form, the *Artificial Neural Network*, which would be trained to race a chariot, with **Back Propagation**, would made up of three layers. You can see a diagram showing these layers in Figures 1.17 and 1.18.

The first layer is the Input layer and is made up of three artificial *Neurons* which are not *Neurons* per se, but simply the **Initial Inputs** for the system. These **Initial Inputs** are three values. The first value is just a **NEURAL NETWORK BIAS** or a constant value used to add variation to the other values on that layer and offset any bias in how the **Training Data** was collected (Figure 1.18).

If the **Training Data** were collected from observing just one player, then there will be a bias towards how that player plays. And you need to offset that and add variation to the **Data**. That will let the network learn how other players play.

If the **Training Data** were collected from one race around the track, then there will be a bias towards that race. And you need to offset that and add variation to the **Data**. That will let the network learn how to play in other races, with different opponents, on different tracks.

By default the **Neural Network Bias** is 1. The other two **Initial Inputs** represent the aggregation of all the inputs to the network. In this case, these values

- the distance travelled along the track
- the distance from the sides of the track
- the closest distance of other racing chariots along the track

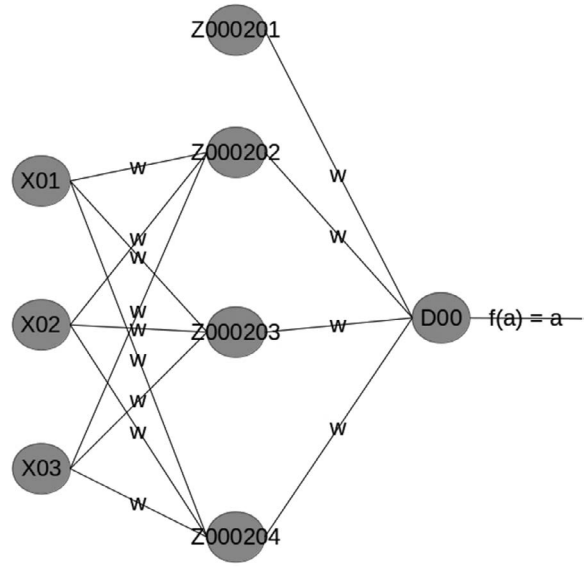

FIGURE 1.17 The simplest form of an *Artificial Neural Network* that can be trained using Back Propagation.

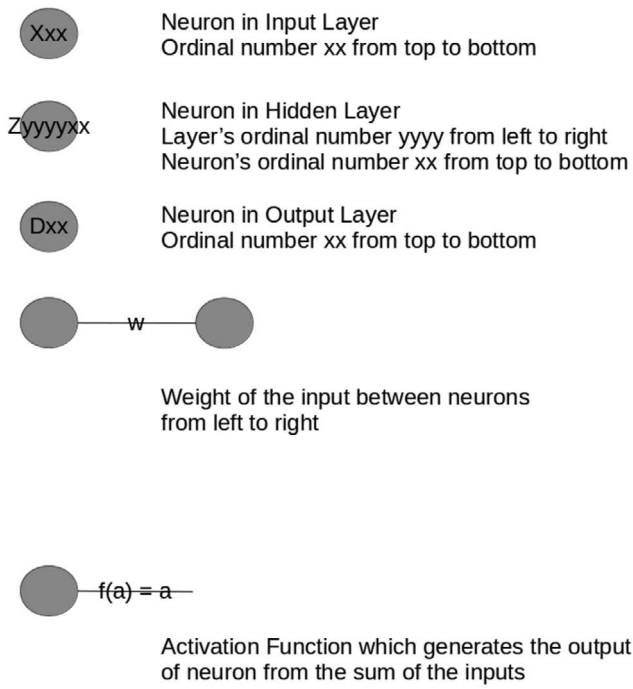

FIGURE 1.18 Legend of the symbols in Figure 1.17.

are added together and fed into one **Initial Input**. And these values

- the relative position of other chariots
- the relative position in the race
- the speed of the chariot

would be added together and fed into the other **Initial Input**.

The second layer is made up of four artificial *Neurons*. The first one is not a *Neuron* per se but another **Neural Network Bias** to adjust all the other values on that layer. By default this is 1. The other three artificial *Neurons* are *Neurons* and take three inputs from the three artificial *Neurons* in the first layer, with a **Weight** on each input. By default all of the **Weights** for all these inputs are 1. The output of each *Neuron* depends on the **Activation Function**.

The third layer is a final Output layer and only has one artificial *Neuron*, which has four inputs from the four artificial *Neurons* in the second layer. Again there is a **Weight** on each input. And by default all of the **Weights** for all these inputs are 1. Again the output of this artificial *Neuron* depends on the **Activation Function**. And in its simplest form, this is just the sum of each input multiplied by its **Weight**.

The **Final Output** from the third layer marks the end of the first phase of the algorithm known as **NEURAL NETWORK FORWARD PROPAGATION** or **Forward Propagation**. The second phase is known as **Back Propagation**. And in this phase, you work backwards, from the third layer, to the second layer, to the first layer. And you adjust the **Weights** of all the inputs at each layer. So that the difference between the **Final Output** and the expected output in the **Training Data** or loss is reduced. For each input at each layer, you adjust the **Weight** attached to that input. Depending on how much that input affected the loss in the **Final Output**.

To adjust the **Weight** at each input, you first have to calculate how much the input and **Weight** contributed to the overall loss in the **Final Output**. Followed by increasing or reducing each **Weight** depending on whether it has a positive or negative effect on the overall loss in the **Final Output**. Working out the contribution of each input to the overall **Final Output** is not trivial. It involves a lot of complex mathematics which are hard to explain. Nevertheless, the end result is these four mathematical equations:

$$f(a) = a$$

$$Delta(n) = W(n) \,.\, Delta(n+1) \,.\, f'(a)$$

$$J'(n) = Z(n) \,.\, Delta(n)$$

$$W = W(n) - Alpha \,.\, J'(n)$$

where

f(a) is the **Activation Function** of every artificial *Neuron* that generates its output
a is the sum of the inputs to the artificial *Neuron* each multiplied by its **Weight**
Delta(n) is the amount of loss in the output of the artificial *Neuron* in the current layer due to an input
W(n) is the old **Weight** of that input
Delta(n+1) is the amount of loss in the output of the artificial *Neuron* in the next layer that the output of the artificial *Neuron* in the current layer feeds into
f'(a) is the partial derivative of the **Activation Function** which in the simplest case is 1
J'(n) is the amount that the old **Weight** has to be adjusted to counteract the loss in the **Final Output**
Z(n) is the output of the artificial *Neuron* in the current layer
W is a new **Weight** for the input
Alpha is the training rate typically 0.1

With these equations you can adjust the **Weights** of the inputs at each layer of the *Artificial Neural Network*, during the **Back Propagation**. And train the network.

1.3.5.2 Application: Flaws in Back Propagation

The equations mentioned in the previous subchapter, used to train *Artificial Neural Networks* with **Back Propagation**, also highlight its flaws. The first flaw is the **Activation Function**.

In its simplest form this is the sum of each input into an *Artificial Neuron* multiplied by its **Weight**. This is a linear function e.g.

$$f(a) = a$$

where

a is the sum of each input multiplied by its **Weight**, and the output of the function.

That is to say, the output of the function increases proportionally with the input. And the overall effect of the linear functions at each layer of *Artificial Neurons* in the network is itself a linear function. This means that the network assumes that overall there is a linear relationship between the **Initial Inputs** and the **Final Outputs**. And the *Artificial Neural Network* will eventually find this relationship, as it is trained.

But if there were no linear relationships between the **Initial Inputs** and the **Final Outputs**, then the *Artificial Neural Network* will never find this relationship. And it will never be able to predict the **Final Outputs** for any given **Initial Inputs**.

For example, suppose you were training an *Artificial Neural Network* to race a chariot around a circus. And there were no linear relationships between the metrics

which are the **Initial Inputs** (e.g. the distance travelled, the distances from the sides of the track and the distance from other chariots) and **Final Outputs** (e.g. the movement of *analogue devices* or *digital devices* of a *Game Controller*) then that network will never find the relationship if it is only using a linear **Activation Function**.

Unless that is the **Activation Function** is a non-linear function. Non-linear **Activation Functions** are the more popular form. Since they can deal with non-linear relationships. When an *Artificial Neural Network* uses a non-linear function, then it can find non-linear relationships between the **Initial Inputs** and **Final Outputs**.

The second flaws are these non-linear functions. There are several from which you could choose. Some non-linear functions are better suited to some applications than others. And the non-linear function you choose can affect how quickly the network can be trained and find the relationship between the **Initial Inputs** and the **Final Outputs**.

Non-linear functions are more complex, harder to understand and harder to explain than linear functions. No one can understand or predict the accumulative effect of non-linear functions in a network, when the number of functions starts to get too large.

This leads to third flaw. That is the guesswork required for the number of layers of *Artificial Neurons*, and the number of *Neurons* in each layer. There is no rule for how to set these numbers. On the one hand, the more layers and *Artificial Neurons* the network has, the longer it will take to do the **Forward Propagation** and **Back Propagation** through the network. And the longer it will take to train the network. On the other hand, the more layers and *Artificial Neurons* the network has, the greater capacity it has to learn. Or so some people believe. This is premise behind models of *Artificial Neural Networks* such as the **DEEP LEARNING MODEL** and **LANGUAGE LEARNING MODEL**.

A **Deep Learning Model** is an *Artificial Neural Network* with a large number of layers and *Artificial Neurons*. It can be used to recognise objects in images.

One example of this is a network used in dermatology to detect diseases in the images of the skin.

A **Language Learning Model** is also an *Artificial Neural Network* with a large number of layers and artificial *Neurons*. It can be used to understand, generate and interpret natural language or human language. In its **Training Data**, the **Initial Inputs** are sentences or paragraphs with a word missing at the end. And the **Final Outputs** are possible words which go on the end of that sentence or paragraph, and the probabilities of those words. This is used to train the network to predict the next word in a sentence or paragraph. And this is used to understand and respond to questions. So long as these can be reframed as a missing word problem.

For example, the question

What do cats like to sleep in?

can be reframed as a missing word problem in the form

Cats like to sleep in _____

The network then has to find the word with the highest probability of finishing that sentence.

One example of this network is the one used by the popular Web Server known as ChatGPT. This has over 170 billion *Artificial Neurons* in the network, which is comparable to the average number of biological *Neurons* in the human brain. And these can achieve (seemingly) impressive results.

Since the **Event-Database Architecture** is based on a *Relational Database*, it can store large amounts of data. Such as the **Training Data** required to teach a **Deep Learning Model** or a **Language Learning Model**. And you may be tempted to create a **Deep Learning Model** or **Language Learning Model** with it. To achieve similarly impressive results. There is a social trend currently to develop more and more software with *Artificial Intelligence* in the form of these **Models**.

Typically, you cannot build these **Models** with commercial *game-engines* which are based on hierarchical databases. There is no standard for hierarchical databases. Therefore, you cannot interoperate a hierarchical database with other tools apart from the *game-engine* which created it. You cannot query or edit large amounts of **Training Data** in these databases with other tools. To verify or correct the data. Apart from using the *game editors* built with these *game-engines*. But these *game editors* are not scalable and were never meant to query or edit large amounts of data.

On the other hand, there is a standard for *Relational Databases*. And you can use any Relational Database Management Systems (RDBMS), to verify or correct large amounts of data in the *Relational Database*. An RDBMS is scalable. Therefore, you can build these **Models** with the **Event-Database Architecture** which is based on a *Relational Database*.

Nevertheless, the trend towards these **Models**, and the impressive results you can achieve with them, comes at a high cost. These leads onto several more flaws of large *Artificial Neural Networks*, such as **Deep Learning Models** and **Language Learning Models**, including the following:

1. the never-ending cycle of the development of larger and larger *Artificial Neural Networks*, with more and more *Artificial Neurons*, which require more and more resources in terms of storage media and computer processing power to run
2. the never-ending cycle of buying more and more *Expensive Graphics Processors*[23] each year required to build these large *Artificial Neural Networks*, to the point where these are being built with *Expensive Graphics Processors* that cost 40000 dollars each
3. the never-ending cycle of more and more electricity and power these *Expensive Graphics Processors* consume and waste, to the point where one **Language Learning Model**, i.e. ChatGPT, takes 1287 megawatt hours of electricity to train it, which is equivalent to the amount of electricity used by an average American household for over 700 years.

And yet despite the huge expense spent on these networks, you still get *expensive erroneous Language Learning Models*[24] from time to time. And with billions of *Artificial Neurons*, and non-linear **Activation Functions**, it is no longer feasible for anyone to understand or predict what the output of these networks would be. And

there is no hope of diagnosing and correcting the output when things go wrong. Apart from playing around with the **Initial Inputs** or the **Final Outputs**. Or trying a completely new set of **Training Data**.

This leads on to the fifth flaw. That is that some *Artificial Neural Networks* can be relatively simple and use only one **Activation Function** for all the *Artificial Neurons*. Others can be very complex and use a huge number of different **Activation Functions** across the network. And there is no rule about how many **Activation Functions** you can have and what combination of **Activation Functions** you can use.

This leads to the sixth flaw in *Artificial Neural Networks*. That is the guesswork required in the selection and distribution of the **Activation Functions** across the network. Some **Activation Functions** are suitable if you want the **Final Outputs** to be a probability of something occurring i.e. a value between 0.0 and 1.0. One example of this is one called a Sigmoid Function.

But apart from that it seems anything goes. The form of the **Final Outputs** of the network just reflects the form of the **Final Outputs** in the **Training Data**. And you can set whatever arbitrary form you like for the **Final Outputs** in the **Training Data**.

For example, suppose you were training an *Artificial Neural Network* to drive a chariot around a circus in a race with other chariots. And you decided the **Final Outputs** in the **Training Data** would reflect the state of the *digital devices* of the *Game Controller* that cause the chariot to

1. accelerate
2. decelerate
3. turn left
4. turn right

You can decide that in the **Training Data** that the **Final Outputs** should take the form of a single value between 1 and 4. Depending on which *digital device* had been pressed at some point in time during the race, and assuming that only one *digital device* could be used at any point in time. This means that form of the **Final Outputs** of your *Artificial Neural Network* will theoretically be one value between 1.0 and 4.0 once it is fully trained.

Or you can decide that the **Final Outputs** in the **Training Data** should take the form of four values which can either be 1 or 0. Depending on whether the four *digital devices* had been pressed or not at some point in time, to accelerate, decelerate, turn left or turn right. This means that the form of the **Final Outputs** of your *Artificial Neural Network* will theoretically be four values between 1.0 and 0.0.

Or you can decide that the **Final Outputs** in the **Training Data** should take the form of two values from two *analogue devices* or axes, which can be between −1 and 1. On one *analogue device* or axis, a value of −1 means that the chariot should decelerate at its maximum rate, and at the opposite end a value of 1 means that chariot should accelerate at its maximum rate. On the other *analogue device* or axis, a value of −1 means the chariot should turn to the left at the maximum rate. And at the opposite end, a value of 1 means the chariot should turn to the right at its maximum rate. This means that the form of the **Final Outputs** of your *Artificial Neural Network* will theoretically be two values between −1.0 and 1.0 once it is fully trained.

Thus, the form of the **Final Outputs** of your *Artificial Neural Network* is arbitrary for two reasons.

Firstly, there is no rule about what form the **Final Outputs** in the **Training Data** and the **Final Outputs** of the *Artificial Neural Network* should take. In this example, it could be one value, two values or four values.

Secondly, there can be outliers in the **Final Outputs**. Take for example the first form previously mentioned, where theoretically the **Final Output** of *Artificial Neural Network* could be a single value between 1.0 and 4.0 when fully trained. Practically, the network may never be fully trained or you will not know how much **Training Data** you have to go through to reach that point. And while it is training some **Outputs** will be within this range and others, known as outliers, will be just outside of this range. And the way in which you choose to respond to outliers is arbitrary.

You could choose to ignore outliers, stick with your model and keep training the network and adjusting its **Weights** until some arbitrary point in the future where you are satisfied with the **Weights** you have. Or you could choose to treat outliers as minor anomalies and clamp down the values greater than 4.0 and clump up the values less than 1.0. So that these fall within the expected range. Or you could choose to deal with outliers by offsetting the **Initial Inputs** or the **Final Output**, by adding or subtracting numbers. Or scale up or down the **Initial Inputs** or **Final Output**, by multiplying or dividing by more numbers. Until the **Final Output** falls within the expected range.

The seventh flaw is the training rate which affects how quickly the network can be trained. There are no rules about how you set this rate. But if you set it too low, then it can take the network a very long time, and several iterations of **Forward Propagation** and **Back Propagation** to complete its training.

The eighth flaw of *Artificial Neural Networks*, which is closely related, is the **Initial Inputs**. If the number of **Initial Inputs** was too small, then you may either have to leave out inputs which do affect the output, because there are not enough places to put it. Or you may have to aggregate two or more input values together into one value. And feed this into one input. As a result, when you train the network, it may not find the relationship between **Initial Inputs** and **Final Outputs**. Since one of the inputs, which on its own has a strong affect on the **Final Outputs**, has been aggregated with other values which have little or no effect. Apart from adding noise to the inputs which do have a strong affect.

The ninth flaw of *Artificial Neural Networks*, which is closely related, is that the **Training Data** cannot contain too much noise. Otherwise, it will affect the rate at which the network can be trained to find a relationship between the **Initial Inputs** and the **Final Output**. And in the worst case, the network may never find the relationship. To prevent this you have to ensure that the values fed into inputs are statistically normalised into values between 0 and 1. And that these are evenly distributed, with a Standard Deviation of 1, and a mean value of 0.

The tenth flaw in *Artificial Neural Networks* is the **Neural Network Bias**. This is an arbitrary constant value generated by an artificial *Neuron* in one or more layers, to offset the bias in how the **Training Data** was collected. It is hard to quantify how much bias there is in how the **Training Data** was collected. And therefore, it is hard to quantify how large the **Neural Network Bias** should be to counteract the bias in the collection.

If the **Neural Network Bias** were too small compared with the bias in the collection of the **Training Data**, then the *Artificial Neural Network* will not tolerate large

variations in its **Initial Inputs**, from the ones in the **Training Data**. And as soon as it receives any inputs which were not in the **Training Data**, its **Final Outputs** will become unpredictable.

Conversely, if the **Neural Network Bias** were too large compared with the bias in the collection of the **Training Data**, then the variation this gives the **Initial Inputs** in the **Training Data** may be so large that it takes a long time to train the network. Or it may be impossible to train the network. Since the **Neural Network Bias** causes such a large variation in the **Initial Inputs** in the **Training Data**, the network can never produce the **Final Outputs** in the **Training Data**. No matter how the **Weights** were adjusted.

There are numerous *Neural Network sources*[25] that describe the flaws of **Back Propagation** and other teaching methods, and how to overcome these flaws.

1.3.5.3 Application: Back Propagation in the Architecture

All the teaching methods for *Artificial Neural Networks* require gathering **Training Data** from a test. In the case of a game involving racing chariots around a circus, this test would involve watching the game being played by a human player. It would involve collecting the important factors on the track that determine the player's response e.g.

- the distance travelled along the track,
- the distance from the sides of the track,
- the distance of other racing chariots along the track,
- the relative position of other chariots.

And it would involve collecting the responses at the same time e.g.

- the position of the *analogue devices* which controlled the acceleration of the chariot,
- the position of the *analogue devices* that controlled the brakes of the chariot,
- the position of the *analogue devices* that controlled steering of a chariot to the left or right,
- the on-off state of the *digital devices* i.e. buttons which controlled the acceleration of the chariot,
- the on-off state of the buttons that controlled the breaks,
- the on-off state of the buttons that controlled the steering to the left or right.

These are the **Initial Inputs** and **Final Outputs** of **Training Data** already mentioned in the previous subchapter.

Once you have gathered the **Training Data**, you can construct an *Artificial Neural Network*, in the **Event-Database Architecture**, using *Database Tables*, *Database Records*, *Database Fields*, **Game Objects**, **Events** and **Actions**. Your **Training Data** could be held in a *Database Table*. The artificial *Neurons* in each layer could be represented by **Game Objects**. The properties of these **Game Objects** would be held in *Database Records*. The *Database Fields* would include

1. value of each input into each *Neuron*
2. the **Weights** of the input values
3. the sum of each input multiplied by its **Weight**

4. the output of each *Neuron*
5. the loss in the output of each *Neuron*
6. the training rate

The chain of calculations, for the output of each *Neuron*, from the first to the last layer, in the first phase or **Forward Propagation**, could be done through a chain of **Primary** and **Secondary Events** e.g.

1. **PRIMARY NEURAL NETWORK FORWARD PROPAGATION nnnn EVENT**
2. **NEURAL NETWORK FORWARD PROPAGATION nnnn FETCH METRICS FROM TRAINING DATA EVENT**
3. **NEURAL NETWORK FORWARD PROPAGATION nnnn FETCH METRICS FROM GAME WORLD EVENT**
4. **NEURAL NETWORK FORWARD PROPAGATION INPUTS nnnn LAYER X NEURON xx EVENT**
5. **NEURAL NETWORK FORWARD PROPAGATION INPUTS nnnn LAYER Zyyyy NEURON xx EVENT**
6. **NEURAL NETWORK FORWARD PROPAGATION INPUTS nnnn LAYER D NEURON xx EVENT**
7. **NEURAL NETWORK FORWARD PROPAGATION OUTPUT nnnn LAYER Zyyyy NEURON xx EVENT**
8. **NEURAL NETWORK FORWARD PROPAGATION OUTPUT nnnn LAYER D NEURON xx EVENT**
9. **NEURAL NETWORK FORWARD PROPAGATION TRANSLATE OUTPUT nnnn EVENT**

where

- **yyyy** is the ordinal number (of four hexadecimal digits) of the layer of the network, from left to right
- **xx** is the ordinal number (of two hexadecimal digits) of the artificial *Neuron* in the layer on the network, from top to bottom
- **nnnn** is the ordinal number (of four hexadecimal digits) of the character using the *Artificial Neural Network*, depending on whether it was the first, second, third etc. NPC to appear in the *Game World*.
- D designates a Artificial Neuron in the Output Layer
- Z designates a Artificial Neuron in the Hidden Layer
- X designates a Artificial Neuron in the Input Layer. See Figure 1.17 and Figure 1.18.

The **Primary Neural Network Forward Propagation nnnn Event** would begin the process of **Forward Propagation** and the generation of the **Final Outputs**.

The **Neural Network Forward Propagation nnnn Fetch Metrics From Training Data Event** would be a **Secondary Event** following from that **Primary Event.** The **Action** in response to this **Event** would get the next metrics from next *Record* of the

Training Data that should be used to train the network. And it would feed these values into all the **Initial Inputs** in the Input layer to train the network. This **Action** would be performed by a **Game Object** that was on its own layer before the Input layer.

The **Neural Network Forward Propagation nnnn Fetch Metrics From Game World Event** would be a **Secondary Event** following from that **Primary Event.** The **Action** in response to this **Event** would get the next metrics from *Game World* that should be used to direct a character in the *Game World*. And it would feed these values into all the **Initial Inputs** in the Input layer of the network. This **Action** would be performed by a **Game Object** that was on its own layer before the Input layer.

The **Neural Network Forward Propagation Inputs nnnn Layer X Neuron xx Event** would be a **Secondary Event** following from that **Primary Event.** The **Action** in response to this **Event** would generate the output for a *Neuron* in the **Initial Inputs** in the Input Layer or first layer, from either the metrics in the **Training Data** or the *Game World* (e.g. distance along the track and shortest distance to other chariots). That is to say the metrics being used to train the network.

The **Neural Network Forward Propagation Inputs nnnn Layer yyyy Neuron xx Event** would be a **Secondary Event** following from that **Primary Event.** The **Action** in response to this **Event** would calculate the sum of all the inputs into an artificial *Neuron*, with each input multiplied by its **Weight**, in a hidden layer or second layer, after the Input layer.

The **Neural Network Forward Propagation Inputs nnnn Layer D Neuron xx Event** would be similar. But **Action** for this **Event** would only be used to calculate the sum of all the inputs into an artificial *Neuron* in the last layer or Output layer.

Likewise, the **Neural Network Forward Propagation Output nnnn Layer yyyy Neuron xx Event** would be a **Secondary Event** following from that **Primary Event.** The **Action** in response to this **Event** would generate output of the **Activation Function** of an artificial *Neuron*. And pass this onto the inputs of the *Neurons* in the next layer.

The **Neural Network Forward Propagation Output nnnn Layer D Neuron xx Event** would be a similar **Secondary Event**. But this would only be used to generate the output of an artificial *Neuron* in the last layer, which is part of the **Final Outputs**.

The **Neural Network Forward Propagation Translate Output nnnn Event** would be a similar **Secondary Event**. The **Action** in response to this **Event** would be performed by a **Game Object** in its own layer after the last layer or Output layer.

It would use the commands i.e.

Forwards Command
Backwards Command
Turn Left Command
Turn Right Command

to direct the chariot in a race. By translating all the **Final Outputs** into the six standard **Secondary Events** i.e.

Secondary Connect Event
Secondary Disconnect Event

Secondary Controller Moved Event
Secondary Controller Stopped Event
Secondary Controller Pressed Event
Secondary Controller Released Event

sent to the **Master Player Object** to simulate the *analogue devices* or *digital devices* on the **Game Controller** being used to evoke those commands.

And it would include the following properties of these **Secondary Events**

1. the cause of the **Event** was the **2D Player Object** or **3D Player Object** of the character being controlled by the AI
2. the unique word identifying the *analogue device* or *digital device* of the **Game Controller** that had been connected, disconnected, moved, stopped, pressed or released to evoke the Forward, Backwards, Turn Left or Turn Right Command.
3. the amount the *device* had been moved along its when invoking that command.

And the **Master Player Object** would, in turn, apply the changes caused by these **Events** to the properties of that **2D Player Object** or **3D Player Object** in the *Game World*.

The chain of calculations, to adjust the **Weights**, from the last layer to the first layer, in the second phase or **Back Propagation**, could also be done through a chain of **Primary** and **Secondary Events** e.g.

1. **PRIMARY NEURAL NETWORK BACK PROPAGATION nnnn EVENT**
2. **NEURAL NETWORK BACK PROPAGATION OUTPUT LOSSES nnnn LAYER D NEURON xx EVENT**
3. **NEURAL NETWORK BACK PROPAGATION INPUT LOSSES nnnn LAYER D NEURON xx EVENT**
4. **NEURAL NETWORK BACK PROPAGATION ADJUST WEIGHTS nnnn LAYER D NEURON xx EVENT**
5. **NEURAL NETWORK BACK PROPAGATION INPUT LOSSES nnnn LAYER Zyyyy NEURON xx EVENT**
6. **NEURAL NETWORK BACK PROPAGATION ADJUST WEIGHTS nnnn LAYER Zyyyy NEURON xx EVENT**

(Note: To understand what 'nnnn', 'yyyy', 'xx', 'D' and 'Z' signify please refer to Figure 1.17 and Figure 1.18, and the Primary and Secondary Events used for Forward Propagation described earlier). The **Primary Neural Network Back Propagation nnnn Event** would begin the process of **Back Propagation** and the adjustments of the **Weights**.

The **Neural Network Back Propagation Output Losses nnnn Layer D Neuron xx Event** would be a **Secondary Event** following from that **Primary Event.** The **Action** in response to this **Event** would calculate output losses of each *Neuron* in the last layer or Output Layer.

The **Neural Network Back Propagation Input Losses nnnn Layer D Neuron xx Event** would be a **Secondary Event** following from that **Primary Event.** The **Action** in response to this **Event** would calculate the loss of the input into a *Neuron* in the last layer or Output Layer that contributed to the overall loss of the Network.

The **Neural Network Back Propagation Adjust Weights nnnn Layer D Neuron xx Event** would be a **Secondary Event** following from that **Primary Event.** The **Action** in response to this would calculate the adjustment needed to be made to the **Weight** of each input into an artificial *Neuron*, in the last layer or Output Layer. To reduce the overall loss of the *Network*. And the **Action** would also make that adjustment.

The **Neural Network Back Propagation Input Losses nnnn Layer Zyyyy Neuron xx Event** would be a **Secondary Event**. The **Action** in response to that would only be used to calculate the loss of the inputs of the *Neurons* in the Hidden Layers in between the last or Output Layer and the first or Input Layer.

Likewise, the **Neural Network Back Propagation Adjust Weights nnnn Layer Zyyyy Neuron xx Event** would be a **Secondary Event**. The **Action** in response to that would only be used to adjust the **Weight** of each input into an artificial *Neuron* in the Hidden Layers between the last and first layer. You can see a diagram of the *Artificial Neural Network* created from the **Events**, **Actions** and **Game Objects** of the **Event-Database Architecture** during **Forward Propagation**, in Figure 1.19 with a Legend in Figure 1.20 and in Table 1.3. And you can see the same during **Backward Propagation** in Figure 1.21 with a Legend in Figure 1.22 and in Table 1.4.

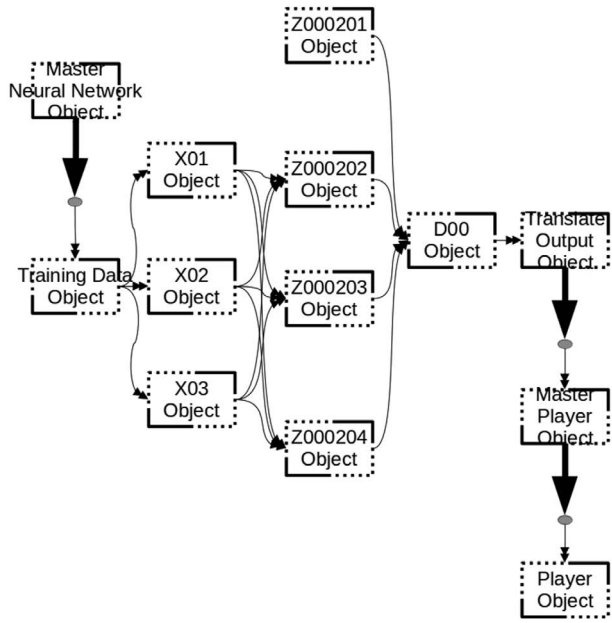

FIGURE 1.19 Network of **Game Objects** connected by **Primary** and **Secondary Events** which form a network of *Artificial Neurons* in an *Artificial Neural Network*, during **Forward Propagation**.

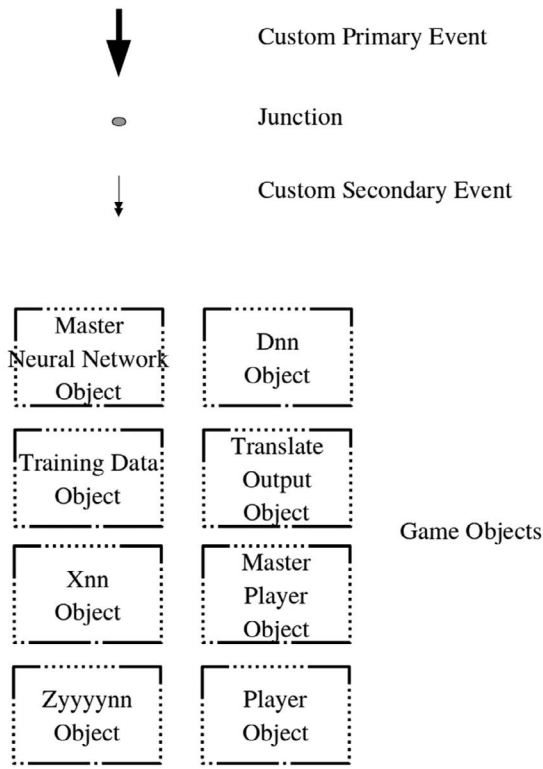

FIGURE 1.20 Legend of all the symbols in Figure 1.19. A more detailed explanation of the **Game Objects** is available in Table 1.3.

TABLE 1.3
Legend of Game Objects Displayed in Figure 1.19

Game Object	Role
Master Neural Network Object	Periodically starts the **Forward Propagation** of all *Artificial Neural Networks*, with the **Primary Neural Network Forward Propagation Event**.
Training Data Object	Fetches the next metrics from the **Training Data** or the *Game World* to be fed into the Input layer of the *Neural Network*. When it receives the **Secondary Neural Network Forward Propagation Event**.
Xnn Object	*Artificial Neurons* in the first or Input layer of the network.
Zyyyynn Object	*Artificial Neurons* in the intermediate or Hidden layers of the network.
Dnn Object	*Artificial Neuron* in the final or Output layer of the network.

(*Continued*)

TABLE 1.3 (*Continued*)
Legend of Game Objects Displayed in Figure 1.19

Game Object	Role
Translate Output Object	Translates the final output of the *Neural Network* into one of 6 possible **Events** which simulate a *Game Controller* being used to control a player's character.
Master Player Object	Forwards the simulated **Events** of a *Game Controller* to the **Player Object** being controlled by the *Neural Network*. Modifies the properties of the **Player Object** based on these **Events**.
Player Object	Receives the simulated **Events** of *Game Controller* and new properties, which affect the player's character position, appearance, movement and animation.

It is a list of the **Game Objects** that form the *Artificial Neural Network* and other connected **Game Objects**.

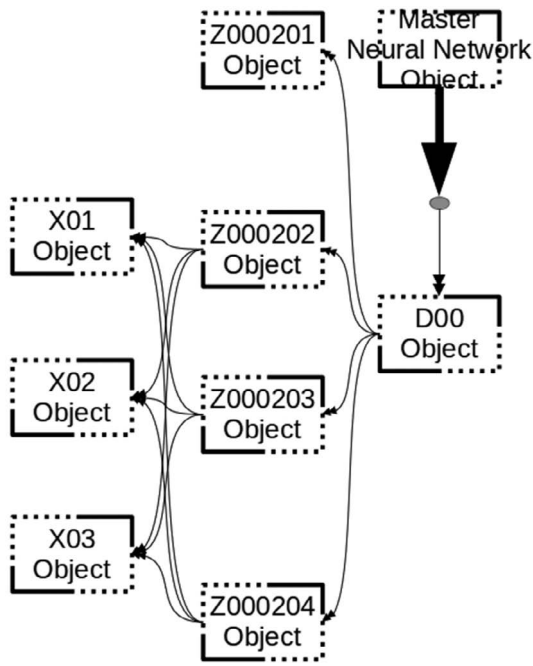

FIGURE 1.21 Network of **Game Objects** connected by **Primary** and **Secondary Events** which form a network of *Artificial Neurons* in an *Artificial Neural Network*, during **Back Propagation**.

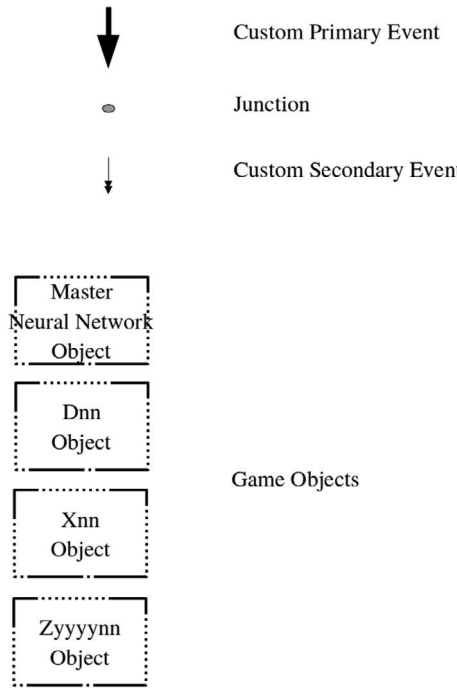

FIGURE 1.22 Legend of all the symbols in Figure 1.21. A more detailed explanation of the **Game Objects** is available in Table 1.4.

TABLE 1.4
Legend of Game Objects Displayed in Figure 1.21

Game Object	Role
Master Neural Network Object	Periodically starts the **Back Propagation** of all *Artificial Neural Networks*, with the **Primary Neural Network Back Propagation Event**.
Xnn Object	*Artificial Neurons* in the first or Input layer of the network. Receives two **Secondary Events**. The first **Event** causes it to calculate how much this *Artificial Neuron* contributes to the overall loss due to the input it fed into the next layer. The second **Event** causes it to adjust the **Weight** of the input this *Artificial Neuron* will feed into the next layer.
Zyyyynn Object	*Artificial Neurons* in the intermediate or Hidden layers of the network. Receives two **Secondary Events**. The first **Event** causes it to calculate how much this *Artificial Neuron* contributes to the overall loss due to the input it fed into the next layer. The second **Event** causes it to adjust the **Weight** of the input this *Artificial Neuron* will feed into the next layer.
Dnn Object	*Artificial Neuron* in the final or Output layer of the network. Calculates the overall loss in the output. When it receives the **Secondary Neural Network Back Propagation Event**.

It is a list of the **Game Objects** that form the *Artificial Neural Network* and other connected **Game Objects**.

1.3.6 Application: Physics

Now in the Computer Games industry, in theory, the simulation of physics could be used to imagine a new set of laws which govern the behaviour of materials in an imaginary universe. And a very small number of games do just that. But, in practice, in the vast majority of games, it is used to animate characters or creatures and produce photorealistic images of this universe, but in a *Game World*. In the **Event-Database Architecture**, you can do either. You can either create a new set of laws or stick with what we have in this universe. This depends on how you build the **Physics Host**.

Firstly, you would have to decide the kind of data this would require. If you decided to stick with the laws in this universe, then each 2D and 3D **Game Object** would require *data* that represented its mass, position, speed, acceleration, angular position, angular speed and angular acceleration in the *Game World*.

It would require a Collision Mesh or 2D shape or 3D model that would be used to detect its collision with other **Objects**. And it would require a Proximity Mesh or 2D shape or 3D model that would be used to detect when it was in close proximity to another **Object**.

It would require a **Secondary Collision Event** that would be triggered and sent to the **Event Host**, when another **Object** collided with this **Object**. So that this **Object** could respond to that collision.

It would require a **Secondary Proximity Event** that would be triggered and sent to the **Event Host**, when another **Object** was in close proximity to this **Objec**t. So that this **Object** could respond to the close proximity of the other **Object**.

It would require a list of 2D **Objects** whose physics or animations due to their physics should be updated. This would include **Invisible 2D Point Objects**, **Text Objects**, **2D Image Objects**, **2D Animation Objects** and **2D Player Objects**.

It would require a list of 3D **Objects** whose physics or animations due to their physics should be updated. This would include **Invisible 3D Point Objects**, **3D Image Objects**, **3D Animation Objects** and **3D Player Objects**.

There is a summary of the *Database Tables* that would be required in the **Game Database in** Figure 1.23.

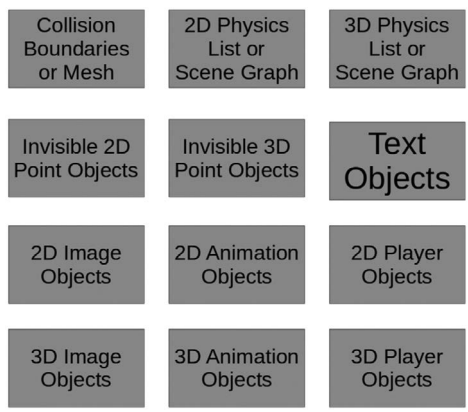

FIGURE 1.23 *Database Tables* required for simulating physics.

You will find the examples of the *Database Records* and *Database Fields* in each *Table* in the *LPmud data design*.

Now, by default, the **Physics Host** only uses Newtonian Physics to model the physics of the *Game World*. In this model, every **Game Object** is treated as one entity, with one mass, one position, speed or acceleration. And any force applied to it is applied to all parts of the body that make up that **Game Object**. For example, if a human character has a force applied to it, then the resultant acceleration is applied equally to every part of that body, arms, legs, head or torso.

But there are other models of physics you could use as well which provide more detail in the motion of the body, such as **INVERSE KINEMATIC PHYSICS** or **RAGDOLL PHYSICS**. These models of physics are examples of the influence of photorealism in the simulation of physics in Computer Games. These are used to produce photo realistic animations of characters and creatures. When these characters or creatures are hit by a projectile, move or die.

In these models each body is made up of a skeleton of bones connected at joints. With constraints about the arc of movement of the bones at each joint, and the linear movement of the bones at each joint. In both cases, you have to pass through a hierarchy of bones in a skeleton. And either make a calculation of the angles of movement at each joint or the forces at each joint.

1.3.6.1 Application: Inverse Kinematic Physics

Inverse Kinematic Physics originated in the study of robotics. Basically, this involves passing through the hierarchy of bones of a robotic arm and calculating the angles of the bones at each joint. To get the end of that arm to reach a certain point in space.

You can use this to animate the skeletons of characters or creatures. To animate their feet when they walk or run. To calculate the angles at each joint of the leg required to reach the next step in the *Game World*. Or to animate the hands of a character when that character picks up or holds an item. To calculate the angles at each joint of the skeleton of the arms, for the hands to pick up or hold the item.

In the **Event-Database Architecture**, each bone could be represented by a **Game Object**. And each pass through the hierarchy could be conducted by a chain of **Primary** and **Secondary Events** connecting those **Objects** e.g.

1. **PRIMARY PHYSICS INVERSE KINEMATICS nnnn EVENT**
2. **PHYSICS INVERSE KINEMATICS nnnn BONE yy xx ANGLE ARM TO REACH TARGET EVENT**
3. **PHYSICS INVERSE KINEMATICS nnnn BONE yy xx ANGLE LEG TO REACH TARGET EVENT**

where

- **yy** is the ordinal number (of two hexadecimal digits) of first joint in the chain of links, or hierarchy of bones of a skeleton, numbered from left to right, top to bottom, that link or bone connects

- **xx** is the ordinal number (of two hexadecimal digits) of the second joint that links or bone connects
- **nnnn** is the ordinal number (of four hexadecimal digits) of the character using the **Inverse Kinematics**, depending on whether it was the first, second, third etc. character to appear in the *Game World*.

The **Primary Physics Inverse Kinematics nnnn Event** would begin the process of generating the angles required at each joint in order to get a leg or an arm, of a character to reach a point in space.

The **Physics Inverse Kinematics nn Bone yy xx Angle Arm To Reach Target Event** would be a **Secondary Event** following from that **Primary Event.** That would generate the angle at a joint of an arm. For that arm to reach a point in space.

Likewise, the **Physics Inverse Kinematics nn Bone yy xx Angle Leg To Reach Target Event** would be a **Secondary Event** following from that **Primary Event.** That would generate the angle at a joint of a leg. For that leg to reach a point in space. There is a diagram showing the numbering of the joints of the skeleton, that in turn controls the numbering of each bone connecting two joints, that in turn controls the naming of the **Events** which the **Game Objects** of the bones respond to in Figure 1.24.

1.3.6.2 Application: Ragdoll Physics

Ragdoll Physics originated in an attempt to produce non-repetitive photo realistic animations of the death of human characters. But it can also be used to simulate the reaction of the body to being hit by the external force of some missile or a weapon. **Ragdoll Physics** was based on Featherstone's Algorithm for Rigid Bodies and involved three basic steps.

In the first step, you have to pass down, from the top to bottom in the hierarchy of bones, calculating the linear or rotational forces (i.e. torque) acting on each bone. And calculating the restorative force required on the next bone down the hierarchy, in order to keep it connected at the common joint linking the two bones. And within whatever constraints have been imposed on the linear or rotational movement around that joint. You then repeat this for the next bone down the hierarchy. Until you reach the bottom.

In the second step, you have to pass up, from the bottom of the hierarchy to the top, calculating the linear or rotational forces acting on each bone. And calculating the restorative forces required on the next bone up the hierarchy. In order to keep it connected at the common joint linking the two bones. And within whatever constraints have been imposed on the linear and rotational movement at each joint. Until you reach the top.

In this step, you resolve all of the forces calculated for each bone, in the first and second passes. And apply it to the bone.

Another simpler example, rather than a hierarchy of bones of a human skeleton, is a chain of metal links.

In this example, in the first step, you pass through from one end of the chain to the opposite end. And on each link in the chain you detect what forces are acting on that link. And you calculate the force required on the next link in the chain required

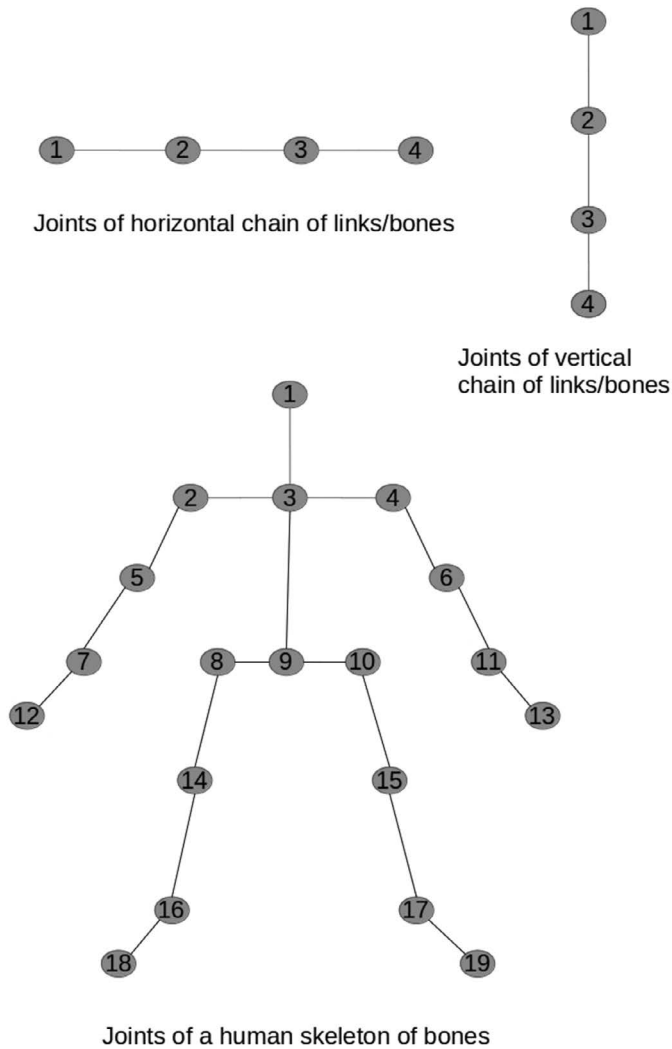

FIGURE 1.24 Numbering of the joints of a horizontal or vertical chain of link. Numbering of the joints of the bones of a human skeleton.

to maintain the connection between both links at their common joint. Given the constraints or rules limiting the linear movement away from that joint, or the radial movement around the axis through that joint.

In the second step, you do the same thing again, but starting from the opposite end, and going back to the end you started from, in the first pass.

After the first pass you will have calculated one set of forces acting on each link of the chain. And after the second pass you will have calculated another set of forces acting along each link of chain. So in the third pass you resolve the forces at each link and apply it to the link causing it to move.

In the **Event-Database Architecture**, each link in the chain could be represented by a **Game Object**. And each pass through the chain could be conducted by a chain of **Primary** and **Secondary Events** connecting those **Objects** e.g.

1. **PRIMARY PHYSICS RAGDOLL nnnn EVENT**
2. **PHYSICS RAGDOLL nnnn BONE yy xx FIRST PASS DETECT FORCES ON BONE EVENT**
3. **PHYSICS RAGDOLL nnnn BONE yy xx FIRST PASS GENERATE FORCES ON BONE EVENT**
4. **PHYSICS RAGDOLL nnnn BONE yy xx SECOND PASS DETECT FORCES ON BONE EVENT**
5. **PHYSICS RAGDOLL nnnn BONE yy xx SECOND PASS GENERATE FORCES ON BONE EVENT**
6. **PHYSICS RAGDOLL nnnn BONE yy xx THIRD PASS RESOLVE FORCES ON BONE EVENT**

where

- **yy** is the ordinal number (of two hexadecimal digits) of first joint, in each pair of adjacent joints, in the chain of links, or hierarchy of bones of a skeleton, numbered from left to right, top to bottom, that link or bone connects
- **xx** is the ordinal number (of two hexadecimal digits) of the second joint, in each pair of adjacent joints, that link or bone connects
- **nnnn** is the ordinal number (of four hexadecimal digits) of the character using the **Ragdoll Physics**, depending on whether it was the first, second, third etc. character to appear in the *Game World*.

The **Primary Physics Ragdoll nnnn Event** would begin the process of generating the forces on a chain or character.

The **Physics Ragdoll nnnn Bone yy xx First Pass Detect Forces On Bone Event** would be a **Secondary Event** following from that **Primary Event**, which would detect the forces acting on each link or bone, in the first pass.

Likewise, the **Physics Ragdoll nnnn Bone yy xx First Pass Generate Forces On Bone Event** would be a **Secondary Event** following from that **Primary Event.** That would detect the forces acting on the next link in the chain, or the next bone in the skeleton, in the first pass.

Likewise, the **Physics Ragdoll nnnn Bone yy xx Second Pass Detect Forces On Bone Event** would be a **Secondary Event** following from that **Primary Event.** That would detect the forces acting on each link, or each bone, in the second pass.

Likewise, the **Physics Ragdoll nnnn Bone yy xx Second Pass Generate Forces On Bone Event** would be a **Secondary Event** following from that **Primary Event.** That would generate the forces acting on the next link, or the next bone, in the second pass.

Likewise, the **Physics Ragdoll nnnn Bone yy xx Third Pass Resolve Forces On Bone Event** would be a **Secondary Event** following from that **Primary Event.** That would resolve the forces acting on each link or bone.

1.3.6.3 Application: Vortex Physics

On the other hand, if you wanted a completely new set of physical laws for a new universe, then a model of physics you could use to build a game is **VORTEX PHYSICS**. In this model the *Game World* is made out of a fluid, with one or more vortices. Running through the centre of each vortex is an axis, and all the **Game Objects** near this axis rotate around it in the same direction, clockwise or anti-clockwise. The closer the **Object** is to the axis, the faster it rotates around the axis. And it either rotates in a spiral towards the centre of the vortex, if it is a cyclone vortex, or away from the centre, if it is an anticyclone vortex.

So each **Object** has an angular position, speed and acceleration around its centre of mass. And it has an angular position, speed and acceleration around the axis of the vortex. And it has an orthogonal position or radial distance from the axis. And it has a position along the axis. The forces affecting it are either orthogonal forces pushing it closer or away from the axis. Or parallel forces pushing along the axis. Or angular forces or torque, pushing it clockwise or anti-clockwise around the axis.

The **Physics Object Records** would have the following *Database Fields*:

1. a *Primary Key*
2. a **Game Object Code Field**
3. a mass
4. the number of a cell of the *Game World* or a vortex in that cell
5. *X Angular Position* around its centre of mass
6. *Y Angular Position*
7. *Z Angular Position*
8. *X Angular Speed*
9. *Y Angular Speed*
10. *Z Angular Speed*
11. *X Angular Acceleration*
12. *Y Angular Acceleration*
13. *Z Angular Acceleration*
14. Parallel Position along the axis of the vortex
15. Parallel Speed along the axis
16. Parallel Acceleration along the axis
17. Orthogonal Position from the axis of the vortex
18. Orthogonal Speed from the axis
19. Orthogonal Acceleration from the axis
20. *Angular Position around the axis of the vortex*
21. *Angular Speed around the axis*
22. *Angular Acceleration around the axis*
23. *Primary Key* of a **Graphics Object Record** of the boundary around the **Object** used to test when a **Collision Event** had occurred
24. *Primary Key* of a **Graphics Object Record** of the boundary around the **Object** used to test when a **Proximity Event** had occurred
25. **Secondary Collision Event** which the **Game Object** should receive
26. **Secondary Proximity Events** which the **Game Object** should receive

In this model, the space is non-linear and non-Euclidean. The position of each **Game Object** maps onto the shape of its local vortex, which is not a flat Euclidean space. Instead, it is a curved funnel around the axis through the centre of that vortex. The shortest path between two points is not a straight line but a curve.

In the **Event-Database Architecture**, in this model the *Game World* would be broken up into a grid of cells. And each cell would contain one vortex. And there would be no space of the *Game World* not covered by this grid.

So, for example, if the *Game World* were broken up into a $3 \times 3 \times 3$ grid of cells. That would mean there would be 27 cells and 27 vortices covering the *Game World*.

Now there are three classes of **Game Objects** in the *Game World*.

The first class is the set of **Invisible 3D Point Objects** that represent each vortex or cell. The other two classes are all orientated around one of these Objects.

The second class is the set of **Invisible 3D Point Objects** or invisible particles which swirl around the central axis of each vortex, in a cyclonic or anticyclonic spiral formation. Each particle would be generated at short random intervals, at random points along the outermost ring, in the cyclone vortex or the innermost ring of an anticyclone vortex. Each would have a spherical Collision Mesh around it used to collide with other **Game Objects**. Each would have a small initial angular speed and acceleration around the central axis of the vortex. But as it travelled further down the spiral format, it would increase its acceleration. And whenever it hit another **Game Object**, all of its momentum will be transferred to that **Object**. Thus, any visible **Game Object** that is hit by these invisible particles will gain all of its momentum. And the particle will disappear from the *Game World*. Likewise, when the particle reaches the edge of the cell of the vortex, it will disappear. But the vortex will keep generating these invisible particles continuously. So as soon as one disappears from the cell, another will reappear.

The third class is the set of visible **Game Objects** in each cell. These will collide with the invisible particles swirling around the vortex of each cell. And each will be slowly forced inwards, if the vortex is a cyclone, or outwards, if the vortex is an anticyclone.

This physical model would require the following **Events**:

1. **PRIMARY PHYSICS VORTEX nnnn SPAWN EVENT**
2. **PHYSICS VORTEX nnnn PARTICLE yyyy SPAWN EVENT**
3. **PRIMARY PHYSICS VORTEX nnnn ACCELERATION EVENT**
4. **PHYSICS VORTEX nnnn PARTICLE yyyy ANGULAR ACCELERATION EVENT**
5. **PHYSICS VORTEX nnnn PARTICLE yyyy COLLISION EVENT**

where

- **nnnn** is the ordinal number (of four hexadecimal digits) of the vortex using the **Vortex Physics**, depending on whether it was the first, second, third etc. vortex to appear in the *Game World*
- **yyyy** is the ordinal number (of four hexadecimal digits) of a particle, depending on whether it was the first, second, third etc. that was about to be generated in the *Game World* by a vortex.

The **Primary Vortex nnnn Spawn Event** would begin the process of generating new invisible particles that will swirl around the vortex.

The **Physics Vortex nnnn Particle yyyy Spawn Particle Event** would be the **Secondary Event** that follows on from that **Primary Event**. That would generate that particle.

Likewise, the **Primary Physics Vortex nnnn Acceleration Event** would periodically change the angular acceleration of particles swirling around the vortex. And it would change the orthogonal accelerations towards the axis. And it would change the parallel accelerations along the axis.

And the **Physics Vortex nnnn Particle yyyy Angular Acceleration Event** would be the **Secondary Event** following on from that **Primary Event**. That would increase the angular acceleration of each particle around the axis of each vortex, its orthogonal acceleration towards the axis and its parallel acceleration along the axis if it were a cyclone vortex. Or it would decrease its angular acceleration of each particle around the axis of the vortex, decrease its orthogonal acceleration away from the vortex and decrease its parallel acceleration along the axis, if it were an anticyclone vortex.

And the **Physics Vortex nnnn Particle yyyy Collision Event** would be the **Secondary Event** following on from the **Primary Collision Event**. That would cause the particle to disappear from the *Game World* after a collision with a visible **Game Object** or with the edge of the cell of a vortex.

This physical model could be used in the game *LPmud* to represent areas which the players could fly through on a magic carpet or on the back of a dragon, like the clouds over a mountain. Or areas the player could swim through like the lakes in the valleys of the mountain.

There are many other areas which the *Game Designers* and other staff could imagine. And many other non-linear, non-Euclidean physical models they could use in these areas.

They do not have to create a realistic, linear, Euclidean model every time. They do not have to create a *Game World* populated by human characters or animals every time which require **Inverse Kinematic Physics** or **Ragdoll Physics**. They could create *Game Worlds* populated by more ethereal characters like smoke, fire or clouds. They do not have to create a *Game World* where some items are static and others are dynamic. But instead a space where everything is dynamic.

1.3.6.4 Application: Flaws of Physics Models and Scalability

In **Inverse Kinematic Physics**, there are some cases where the mathematical formulas that calculate the angles that the bones of an arm or leg have to make at each joint, for the end of that arm or leg to reach a point in space, may either not be solvable. Because that arm or leg is simply not long enough to reach that point in space. Or there may be multiple solutions. In which case you get strange results where during the animations of the bones of that arm or leg trying to reach a point, the bones and joints flip suddenly from one solution to another. And make it appear that the arm or leg has suddenly snapped.

In **Ragdoll Physics**, in the example of the metal chain, in theory, the number of times you have to pass up or down that chain making calculations could be infinite.

If, when you resolve the forces acting on the link at one end of the chain, the result is not zero, then that force will propagate back down the chain again. And you have to do another pass through the chain calculating the forces acting on each link. And if, when you reach the opposite end and resolve the forces at that end, and the result is not zero, then that force too will propagate back down the chain again. And so on and so on.

Therefore, in theory this wave of energy could travel up and down the chain forever. But in practice, the Featherstone Algorithm used for **Ragdoll Physics** only goes up and down the chain once. And a lot of energy is suddenly lost from the system, after the second pass, which is not realistic.

Another flaw in **Ragdoll Physics** is that often a Collision Mesh has to be set for each bone of the skeleton of a body. To have greater accuracy detecting the collision of each bone with the environment, and the forces acting on each bone. This makes the process of setting up the body of a character in the game more complex, compared with without using **Ragdoll Physics**. And it makes the process of calculating the effect of the collision of the body, with its environment, more complex and requires more computing resources. Furthermore, if these multiple Collision Meshes were not set up correctly, then it can cause many chaotic results.

For example, suppose you have a character who can row a boat while sitting down, across a lake in the valley of the mountains. So you decide to only put a Collision Mesh around the centre of mass of the character i.e. its pelvis. That the character sits on to row the boat. And you disable or neglect to add the Collision Meshes around the bones of the arms and legs of a human body. When that body falls into the water, it will end up rolling around indefinitely, around its centre of mass. And the arms and legs of the body will lash out at random in the air. Because there will be no account of the forces of pressure in the water, acting along all the different parts of the body equally, pushing them upwards. Instead, this distributed force will be reduced to a single force acting on the centre of mass, i.e. its pelvis, pushing that part of the body upwards. And pivoting the rest of the body around its centre.

As already mentioned, **Ragdoll Physics** require more calculations than Newtonian Physics and can take up more computing resources as a result. Likewise for **Inverse Kinematics Physics** more calculations are required compared with just relying on the *Game Artists* creating animations of the movement of characters that fit into the *Game World*. Likewise, **Vortex Physics** can become very complex. Especially when you consider the interaction between two or more vortices acting in close proximity in the same cell in the *Game World*. And if you have too many characters, too many **Game Objects** using these physics models at once, then they can literally cause your system to run out of resources and have a critical error or Crash.

Nevertheless, the **Event-Database Architecture** allows you to scale up if you require more resources for your model of the physics of the *Game World*. You can have more than one instance of the **Physics Host**. So you can have multiple instances of the **Physics Host** performing the calculations and updating the physical properties of **Game Objects** in parallel. This allows you to scale up the updating of the **Game Objects** and reduce the time it takes to update these **Objects**. By running more and more **Physics Hosts** simultaneously on more and more *Threads*, Processors or computers in a local computer network.

So long as you ensure that one **Physics Host** only updates one subset of the **Objects** in the *Game World*. And that **Physics Host** does not try to update the **Objects** in another subset owned by another **Physics Host**. And that all the dynamic **Objects** in *Game World* are owned by at least one **Physics Host**. And that each **Physics Host** can read and write the properties of the set of **Objects** it owns in the central **Game Database**, through the **Database Host**. And this can be done whether the **Database Host** is on the same local computer or on a remote computer in the computer network.

If you can ensure all this, then there should be no conflicts and you can massively improve the performance of this game. At the cost of the resources required to purchase more *Threads*, Processors or computers in a local computer network. And if you purchase more computers to host the **Physics Host**, on a computer network, then you will have to expend more money to purchase the bandwidth or greater throughput required between the computers on the network. Since the traffic across that network will increase.

Another aspect of the **Event-Database Architecture** that can be leveraged to greatly improve performance of a game, across computer network, is the *Relational Database Management System* (RDBMS) that is chosen to host the central **Game Database**. Many of these systems come in a **Multi-User Distributed Form**. That is to say, the **Game Database** is not hosted on one computer but distributed across many computers, on a computer network. But when you access the *Database* on any computer on that network, the RDBMS acts as if the *Database* was hosted on that one computer.

Therefore, if you select an RDBMS that supports Distributed Systems, you can massively improve the performance of this game. By getting the *Database Administrator* to configure this System. So that the *Database Records* of the **Objects** which each **Physics Host** owns are as close to it as possible on the local computer network. And the *Record* can be read from or written to very quickly as a result.

1.3.6.5 Application: Line of Sight Physics

Apart from aiding animation, through such physical models as **Inverse Kinematic Physics** and **Ragdoll Physics**, another major use of physical models in Computer Games industry is to aid *Artificial Intelligence*. Specifically, it is for helping NPCs respond to things in their line of sight. Typically with a commercial *game-engine*, you can fire an invisible ray from one point in the *Game World* to another. And you will get back a list of all the **Objects** that were encountered along that straight line. And an *Artificial Intelligence* can use this for example to aim missiles at a target in its line of sight. Or to react to something it sees in its line of sight. Like, for example, pedestrians walking along a side of a paved road through a town or village can use this to dodge out of the way of any player-controlled horses or carriages. That veer off the road and start riding through the pedestrians.

Now in the **Event-Database Architecture**, there is no explicit ability to draw straight lines in the *Game World*, between two points. And to see what **Game Objects** would be hit along that line. But you can implicitly create a thin Cylindrical Mesh which can act like a straight line. And make this Mesh a boundary around an

Invisible 2D Point Object or **Invisible 3D Point Object** in the *Game World*, for **Proximity Events**. And you can transform, rotate and scale this thin line between any two points in the *Game World*, by transforming, rotating or scaling the **Game Object** that line was attached to. And once you place this **Game Object** on the **2D Physics List** or **3D Physics List**, the **Physics Host** will send a **Secondary Event** from every **Game Object** that line intersects. To the **Invisible 2D Point Object** or **Invisible 3D Point Object**. And you can use this to cause an *Artificial Intelligence* to react to **Objects** in its line of sight.

The main difference between this approach and the approach used by commercial *game-engines* is that in the latter case, you can fire as many of these rays in a *Unit of game time* as you want. And you will get a response immediately in that same *Unit*. But in the **Event-Database Architecture** you will not get a response in the current *Unit of game time*, but the next *Unit*. Furthermore, the **Physics Host** will automatically stop updating the physical properties in the current *Unit of game time*. And defer the updating to the next *Unit*, if updating the properties takes more time than the *Unit of game time*. So the **Event-Database Architecture** automatically mitigates the case where you can have too many of these rays being fired off at once, consuming too many resources. See the chapter entitled

Events Host, Physics Host and Recursion Errors

in the volume

Event-Database Architecture for Computer Games: Volume 1, Software Architecture and the Software Production Process.

1.3.7 Application: Graphics

Rendering graphics in the *Game World* with the **Event-Database Architecture** would begin by placing **Graphic Objects** on the **2D Graphics List** and **3D Graphics List**.

There is an example of the *Database Tables* of **Graphic Objects** that could be placed on the **2D Graphics List** and **3D Graphics List in Figure 1.25**.

Apart from the *Database Tables* of the **Graphic Objects**, **2D Graphics List** and **3D Graphics List**, the **Graphics Host** would require *Database Tables* with information that could be used to perform *Software Rendering* and *Hardware Rendering*. In a process that would run during each *Unit of game time*.

Basically, the steps of the process are the following:

In the first step, the bounding boxes around the **2D Graphic Objects** and **3D Graphic Objects** in the **2D Graphics List** and **3D Graphics Lists** are collected.

In the second step, the active **2D Camera Objects** and **3D Camera Objects** that would be used to project these onto the screen are collected from **2D Camera List** and **3D Camera List**.

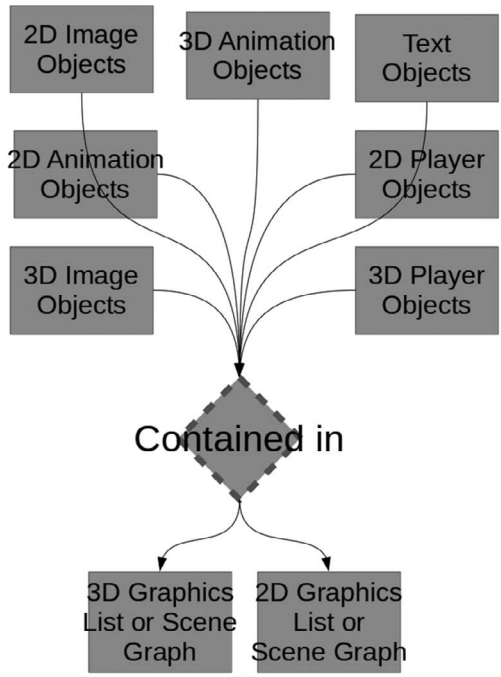

FIGURE 1.25 Examples of *Database Tables* of *Game Objects* that would be put on the 2D Graphics List and 3D Graphics List for the Graphics Host to render.

In the third step, the collection of bounding boxes are projected through the collection of **Camera Objects** into the screen space, using *Software Rendering*.

In the fourth step, the projections are put into the **Projected Shapes** *Database Table*.

In the fifth step, a criterion is used to select the projections from **Projected Shapes**. And the selection is put in the **Projected List** *Database Table*.

This criteria would typically be whether the bounding boxes of the **Graphic Objects** fitted completely or partially within the area of visibility or viewing frustum in front of the **Camera Objects**.

After that *Hardware Rendering* would be used to render the **Graphic Objects**, which had been selected on the **Projected List**, on the screen.

There is a summary of the *Database Tables* that would be required in Figure 1.26.

More *Database Tables* would be required with the 2D *polygons*, 3D models, *Textures*, *Texture coordinates* or UVs and *Materials* to perform *Hardware Rendering* of **Graphic Objects.**

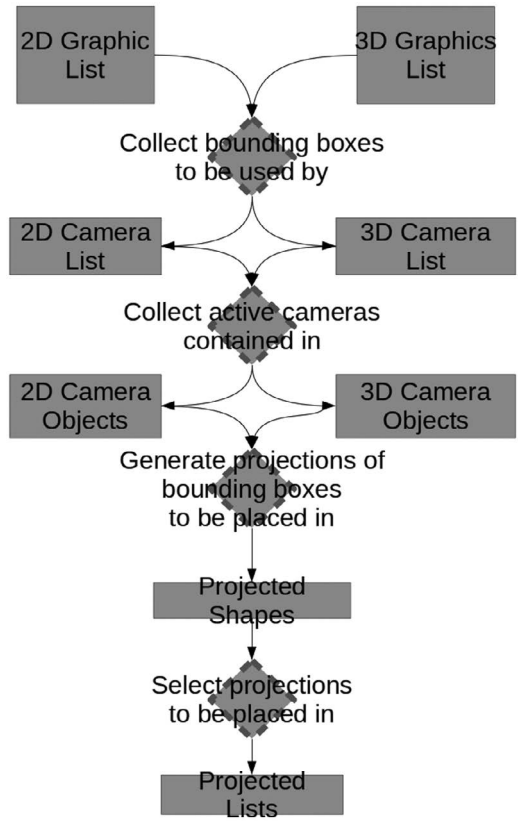

FIGURE 1.26 The *Database Tables* required by the Graphics Host to execute the preliminary steps of the process of *Software Rendering* which selects the items for the process of *Hardware Rendering*.

There is a summary of the *Database Tables* that would be required in Figure 1.27.

More *Database Tables* would be required to perform *Hardware Rendering* of **Text Objects**.

There is a summary of the *Database Tables* that would be required in Figure 1.28.

More *Database Tables* would be required to animate the *vertices* and images of 2D *polygons*, 3D models or skeletons of 3D models.

There is a summary of the *Database Tables* that would be required in Figure 1.29.

You will find the examples of the *Database Tables*, *Database Records* and *Database Fields* used by the **Graphics Host** in the *LPmud data design*.

1.3.7.1 Application: Flaws of Graphical Models and Scalability

As has already been mentioned, the **Graphics Host**, in the **Event-Database Architecture**, renders graphics in a process that combines *Software Rendering* with a Central Processor and *Hardware Rendering* with a Graphics Processor. The

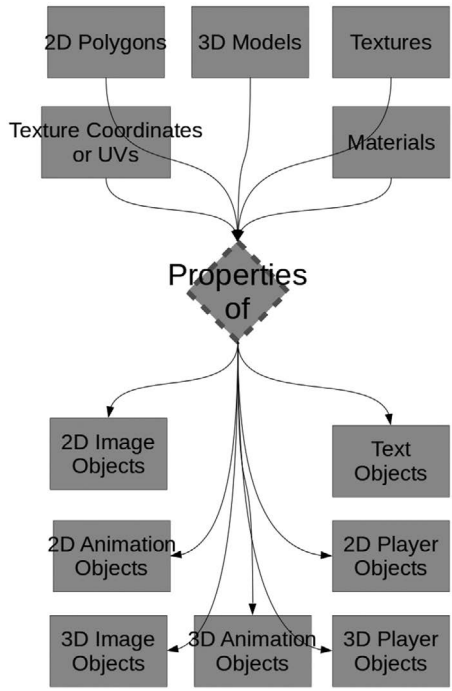

FIGURE 1.27 The *Database Tables* required by **Graphics Host** to perform *Hardware Rendering.*

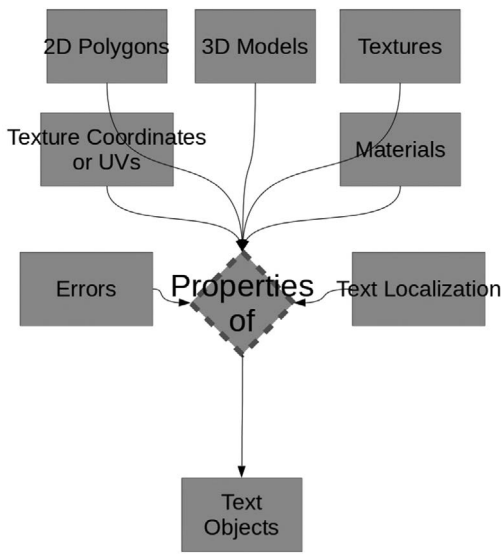

FIGURE 1.28 The *Database Tables* required by the **Graphics Host** for rendering Text Objects with *Hardware Rendering.*

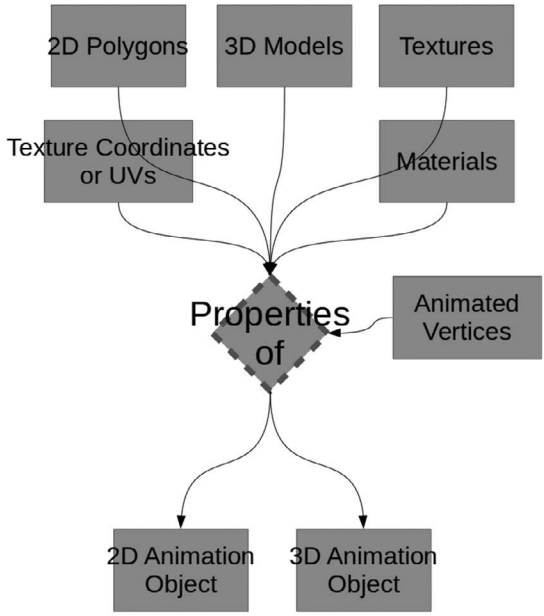

FIGURE 1.29 The *Database Tables* required by the Graphics Host for rendering 2D Animation Objects and 3D Animation Objects.

formative steps are performed with a Central Processor. And the latter steps are performed with a Graphics Processor.

You can see a general description of the former steps in Figure 1.26. But a general description of the latter steps is not included in the standard description of the **Event-Database Architecture** because these are too low level for a high-level *software architecture*. These involve low-level machine code, or *Graphic Shaders*, used to programme the Graphics Processor.

Nevertheless, it is worthwhile identifying the number of steps involved in this process. Just for the sake of comparison with other *software architectures*. In the **Event-Database Architecture**, the steps for *Hardware Rendering* with a Graphics Processor would typically be the same as the steps used by standard *software libraries* such as OpenGL. In these *software libraries*, these steps are categorised by the following the *Graphic Shaders* used to programme them:

1. Vertex Shader
2. Tessellation
3. Geometry Shader
4. Rasterisation
5. Fragment Shader.

After the 'Fragment Shader' is used, the graphics would normally be rendered on the screen or a *Texture*.

Now what exactly each of these *Graphic Shaders* does at each step is not important at this point. If you are interested, then see the subchapter entitled

Materials Table

What matters at this point is that you can use these *Graphic Shaders* to identify the number of steps involved in *Hardware Rendering*. And this number is around 5 steps. From the diagram Fig 1.26, the steps involved in *Software Rendering* with the **Graphics Host** are 5. This gives you an overall total of 10 steps for *Software Rendering* and *Hardware Rendering*.

Now compare this with existing commercial *game-engines* with *software architectures* which adopt a similar approach. That similarly combines a number of steps of *Software Rendering* with a Central Processor and a number of steps of *Hardware Rendering* with a Graphics Processor.

Typically, the total number of steps used by the commercial *game-engines* are much longer and more complex. These can be up to 20 'passes' or cycles through the steps of *Software Rendering* or *Hardware Rendering*. Each 'pass', except the final 'pass', produces partial results which are fed into the next 'pass'. Each 'pass' is not a single step but multiple steps, using the five basic steps of *Hardware Rendering* to varying degrees. That means overall in 20 'passes' you could have up to 100 steps.

There is a diagram showing the time taken by some of these 20 'passes' for a commercial *game-engine*, during each *Unit of game time* or *Frame* in Figure 1.30.

Advocates of these commercial *game-engines* would claim that the longer processes i.e. the greater number of steps involved in *Software Rendering* and *Hardware Rendering* of these *software architectures*, compared with the **Event-Database Architecture**, were necessary to achieve Photorealism.

It was necessary to be able to render photorealistic scenes in *Game World* in 'real-time'. By which they mean at a rate of around 60 *Frames* per second. And it was necessary to show photorealistic graphical effects in these scenes. Such effects would include effects of

- lighting,
- shadows,
- reflections off shiny surfaces,
- roughness on metallic surfaces,
- fog,
- smoke,
- clouds and so on.

This would also include animated surfaces or particles, such as

- waves rippling across liquid surfaces,
- tongues of fire flying out of a torch,
- splashes of water from crashing waves,
- sparks flying off a metallic surface after a bullet ricochets off its surface or
- long grass swaying in a breeze.

FIGURE 1.30 The measurements of the times taken to execute 20 'passes' through the process of *Hardware Rendering* with the Graphics Processor, during one *Unit of game time* or *Frame*, reported by the Profile GPU Unreal Console Command, for a game being built with the Unreal Engine.

But there are several rudimentary flaws in the *software architecture* of commercial *game-engines* with respect to this claim. That come out when you compare the *Software Rendering* and *Hardware Rendering* processes of these commercial *game-engines*, with the *Software Rendering* and *Hardware Rendering* processes of the **Event-Database Architecture**. These are with respect to

1. Flaws of Photorealism in Game Worlds
2. Reusability of the intermediate data generated
3. Obscurity of the sub-processes
4. Obscuring of graphics with physics
5. Scalability of the processes
6. Limitations of power vs limitations of imagination.

1.3.7.2 Application: Flaws of Photorealism in Game Worlds

The first rudimentary flaw to the claim that the longer and more complex *Software Rendering* and *Hardware Rendering* processes of commercial *game-engines* are necessary for Photorealism is the flaws of Photorealism in *Game Worlds*.

Indeed the *software architectures* of commercial *game-engines* with 20 'passes' through the steps of the *Hardware Rendering* process may very well be superior to ones with just 1 'pass', such as the **Event-Database Architecture.** At least, when it comes to producing photorealistic scenes in computer games.

But do you need to achieve Photorealism? Are computer games nothing more than interactive movies? Do you need a process which takes 20 'passes', through the steps of *Hardware Rendering* process of the Graphics Processor, to achieve Photorealism? Is a process which takes 20 'passes' superior to one which takes less 'passes', or one which just takes 1 'pass'?

Despite the long complex processes the *Software Developers* employ, the results achieved by their commercial *game-engines* are still flawed. And they fail to achieve Photorealism. All they do is give a temporary illusion of Photorealism. Once you start to recognise the *flaws of Photorealism*,[26] you cannot stop seeing these every time.

A scene that is lit so bright that it looks like it is taking place on the twilight zone on Mercury next to the sun, a reflective surface which does not reflect everything moving in front of it, the fog in an atmosphere which seems to have no moisture, the sparks of fire emanating from a torch which just look like blocks of 2D sprites flashing across the screen, all quickly and permanently break the illusion of Photorealism, once you notice them.

And there are many more flaws in Photorealism used by commercial *game-engine*. These include the following:

1. the never-ending cycle of the development of more and more demanding rendering algorithms to achieve Photorealism, in the *game-engines* each year, which require more and more resources
2. the never-ending cycle of requiring customers to buy more and more new *Expensive Graphic Processors* each year
3. the never-ending cycle of more and more electricity required to power these new *Expensive Graphic Processors*
4. the clipping of moving or animated 3D models into each other
5. the crude micro models of molecular and particle physics used by the Graphics Processor
6. the crude macro models of mechanical physics used by the Central Processor, to create worlds whose mechanics appear realistic and therefore add to the Photorealism
7. the macro models of mechanical physics run by the Central Processor, that bear no relation to the micro model of molecular and particle physics run by the Graphics Processor.

The never-ending cycle of the development of more and more demanding rendering algorithms to achieve Photorealism, in the *game-engines* each year, results in

the need for more and more powerful Graphics Processor to run these algorithms or *Hardware Rendering* processes. And in the need for more and more greater capacity of the storage media used to hold the data being rendered. And in the need for faster and faster speeds of communication between the storage media and the Graphics Processor.

The never-ending cycle of requiring customers to buy more and more new *Expensive Graphic Processors* each year results in customers having to discard old Graphic Processors whose potential is never fully realised. Thus producing a vast amounts of waste of money and Graphics Processors which are thrown away or discarded.

The never-ending cycle of more and more electricity required to power these new *Expensive Graphic Processors* also produces more and more waste. This electricity could be used to produce something more tangible or more useful to society.

The clipping of moving or animated 3D models into each other results in characters clipping into walls, or into the camera, or other characters, or vehicles in the *Game World*. And once you notice them, this spoils the illusion and reveals how shallow and hollow the *Game World* is.

The crude micro models of molecular and particle physics used by the Graphics Processor are used to simulate complex biological and atmospheric phenomena, such as fog, sunlight, wind and waves across a river, lake, sea or a field of grass. But this is based on nothing more than parallel processors running Linear Algebra, parsing and editing vertices of 3D models, parsing and editing pixels on *Textures* and blending colours of the pixels on *Textures* or the screen. There is no way something as rudimentary as Linear Algebra, vertices of 3D models and colours of pixels can begin to capture such complex biological and atmospheric phenomena.

The crude macro models of mechanical physics used by the Central Processor are more often than not non-deterministic. This results in, amongst other things, if you have two computers or **Game Clients** playing a multiplayer game on a local computer network, trying to simulate the physics in the same *Game World*, then the results on the two computers diverge very quickly after a short amount of time. And this requires regular correction from an authoritative source, normally another computer simulating the same physics called a **Game Server**. It also means that if you start the same simulation of the physics in a *Game World*, from the same starting point, with the same physical items, on the same computer, you will not get the same results after a set amount of time.

The crude macro models of mechanical physics (i.e. Newtonian Physics and Inverse Kinematic Physics) run by the Central Processor, which uses two branches of mathematics i.e. Newtonian Mechanics and Kinematic equations. These bear no relation to the crude micro models of molecular and particle physics run by the Graphics Processor, which uses another branch of mathematics i.e. Linear Algebra. Therefore, there are two competing models, not one single unified model of physics used to achieve Photorealism in Computer Games. That in turn means it is harder to explain how Photorealism is achieved using these two models. And that in turn means it is harder to train people in it.

1.3.7.3 Application: Reusability of Intermediate Data

The second rudimentary flaw to the claim that the longer *Software Rendering* and *Hardware Rendering* processes of commercial *game-engines* are necessary is the intermediate data generated in these processes.

The *Software Rendering* process of the **Event-Database Architecture**, performed with the Central Processor, produces intermediate results which are available to another process to reuse. For example, it produces projections of the bounding boxes around the **Game Objects**, on the screen, in the **Projected Shapes**. And it uses this to filter out or cull the **Objects** which are not in the area of visibility in front of a **Camera Object**.

The *Software Rendering* process also produces the order of **Objects** in the line of sight of the **Camera Object** in **Projected List**. Other **Game Objects** or staff can reuse the data, in these *Tables*, to either find the **Objects** in the line of sight of the **Camera Object**. To find which **Objects** were in front of other **Objects** and which were partially or completely obscured in the line of sight. Or to find which **Objects** were under the mouse cursor on the screen.

But with these commercial *game-engines*, this information is lost. Such data is just one of the many intermediate results produced in the 20 'passes' through the steps of *Hardware Rendering* process with the Graphics Processor. These intermediate results remain in the special Graphics Memory only available to the Graphics Processor. And typically, these are not made available for reuse in the main computer memory by the Central Processor.

In rare occasions, when some of these results are returned back, from Graphics Memory to main memory, this is done in order to parse the data and send it straight back to the Graphics Memory for another 'pass' of the *Hardware Rendering* process. To achieve a graphical effect. And the data remains in a low-level form, at the level of pixels, not in a higher level form, at the level of **Game Objects**.

For example, you can tell which pixels are in the line of sight of the camera, which pixels are obscured or not obscured in the line of sight. But you cannot tell which **Objects** are in the line of sight of camera. Or which **Objects** are partially or completely obscured in this line of sight. Unless you manually do the calculations yourself with the Central Processor, or you extract the data from the Graphics Memory using very obscure low-level technology i.e. Assembly Language or Machine Code. But with the **Event-Database Architecture**, these calculations are automatically done for you. And you just need to look up the results in the *Database Tables*. Using high-level technology i.e. a Relational Database Management System.

1.3.7.4 Application: Obscurity of the Sub-Processes

The third rudimentary flaw to the claim that the longer *Software Rendering* and *Hardware Rendering* processes of commercial *game-engines* are necessary is that there is a lot of obscurity in these complex processes.

For example, if you look back carefully at image Figure 1.30 shown earlier, which involves 20 'passes' through the steps of the *Hardware Rendering* process, you will

see that a lot of those 'passes' are undocumented, and mysterious. They have cryptic names such as

- "NiagaraEmitterInstanceBatcher_ExecuteTicks
- PrePass DDM_AllOpaque
- BuildHZB
- Other Children?"

Source: **The measurements of the times taken to execute 20 'passes' through the process of *Hardware Rendering* with the Graphics Processor, during one *Unit of game time* or *Frame*, reported by the Profile GPU Unreal Console Command, for a game being built with the Unreal Engine.**

The beginning and end of sub-processes, within the *Hardware Rendering* process, is obscured or blurred, in commercial *game-engines*. One of the sub-processes is called the 'Occlusion Test'. The name 'Occlusion Test' is itself an obscure term used to refer to the process of culling or filtering out **Objects**, or parts of **Objects**, which are not in the area of visibility in front of a camera. In Fig 1.30, you can see that there is a 'pass' which is named after the 'Occlusion Test' called

"BeginOcclusionTests".

Source: **The measurements of the times taken to execute 20 'passes' through the process of *Hardware Rendering* with the Graphics Processor, during one *Unit of game time* or *Frame*, reported by the Profile GPU Unreal Console Command, for a game being built with the Unreal Engine.**

Now this 'pass' only marks the beginning of the 'Occlusion Test'. It does not mark the end. So you cannot tell where the 'Occlusion Test' begins and ends.

The time taken up by the Graphics Processor in that 'pass' is not the total time taken to perform the 'Occlusion Test'. There are other 'passes' involved. But these are obscured by cryptic names. One of them is the 'pass' called

"BuildHZB"

Source: **The measurements of the times taken to execute 20 'passes' through the process of *Hardware Rendering* with the Graphics Processor, during one *Unit of game time* or *Frame*, reported by the Profile GPU Unreal Console Command, for a game being built with the Unreal Engine.**

Here is how one article describing this 'pass'.

"In this chapter you'll learn about:

- What is a *rendering pass*
- Over 20 kinds of passes in Unreal – lighting, the base pass or the mysterious HZB

- What affects their cost (as seen in the GPU Visualiser)
- How to optimise each rendering pass

…

HZB (SETUP MIPS)

Responsible for:

- Generating the Hierarchical Z-Buffer

Cost affected by:

- Rendering resolution

The HZB is used by an occlusion culling method 1 and by screen-space techniques for ambient occlusion and reflections 2."

Source: Unreal's Rendering Passes © 2019. Oskar Świerad. Page 143

You would never guess from its name 'BuildHZB' that this 'pass' had anything to do with reflections of light. Nor that it has anything to do with the 'Occlusion Test'. And what is more that 'pass' does not mark the end of the 'Occlusion Test'. Some other 'pass' marks the end of the test. But you cannot tell from any of the names of the 20 'passes' in Figure 1.30 which 'pass' marks the end of the test.

Whereas with the **Event-Database Architecture** the beginning and end of the 'Occlusion Test' is the beginning and the end of the *Software Rendering* process with the Central Processor. That test is the primary objective of that process. The process begins when the **Objects** involved in the 'Occlusion Test' are taken off the **2D Graphics List** and **3D Graphics List**. And it ends when the projection of the bounding boxes around these **Objects**, on the screen, has been placed in **Projected Shapes**. And when some of the **Objects** in **Projected Shapes** which meet some criteria, typically whether their bounding boxes are partially or completely visible on the screen, have been selected and put on the **Projected List**.

So with the **Event-Database Architecture**, you can tell the beginning and end of the 'Occlusion Test'. But with these commercial *game-engines* with 20 'passes', the beginning and end are obscured.

1.3.7.5 Application: Obscurity of Graphics with Physics

The fourth rudimentary flaw to the claim that the longer *Software Rendering* and *Hardware Rendering* processes of commercial *game-engines* are necessary is the obscurity of graphics with physics. That arises from the need to achieve Photorealism.

Whatever Photorealism is achieved dynamically with Graphics Processors and *Hardware Rendering* can be achieved statically with Central Processors and *Software Rendering*. The only difference is the size of the **Game Database**.

In the latter case, the processing has to be done before the game is built. Normally by Computer Aided Design or CAD Tools or *Rendering Farms*[27] built for that purpose. And the results have to be added to the **Game Database**. Furthermore, the results are static.

However, in the former case, the processing can be done after the game is built and while it is running or being played. The results do not have to be added to the **Game Database**. Therefore, the **Game Database** is relatively smaller. And the results are dynamic.

Now some would say, with respect to Photorealism, that a dynamic result is more 'realistic' than a static result. But a dynamic result requires you to move from the realm of graphics, into the realm of physics. And start creating realistic physical models that effect lighting, or shadows, or the movement of particles, or sparks of fire, or splashes of water and so on. While attempting to do so with specialised Graphics Processors and crude mathematics, which were never meant to simulate physical models but graphical models.

1.3.7.6 Application: Scalability of the Processes

The fifth rudimentary flaw to the claim that the longer *Software Rendering* and *Hardware Rendering* processes of commercial *game-engines* are necessary is the scalability of the results.

None of the techniques used in the 20 'passes' of the *Hardware Rendering* process of commercial *game-engine*, shown in Figure 1.30 earlier, are scalable. Once the number of parameters involved (e.g. the number of **Game Objects**, light sources and reflective surfaces) exceeds a certain threshold, the *Hardware Rendering* process has not got the capacity to handle it. It physically cannot store many **Game Objects**, light sources, reflective surfaces and so on in the Graphics Memory used by the Graphics Processor. And it cannot calculate the effects of all of these in 'real-time' i.e. 60 *Frames* per second. The performance of the process drops dramatically or it completely fails. No matter how powerful or costly the *Expensive Graphics Processor* is. No matter how large the Graphics Memory is available.

Whereas with the **Event-Database Architecture**, at least the part of the process which involves *Software Rendering* with the Central Processor is scalable. It can be distributed across a cluster or network of computers.

As with the **Physics Host**, there can be more than one instance of the **Graphics Host** distributed across a computer network. You can have a pool of computers, of any size, performing *Software Rendering* of the *Game World* for all the **Game Clients** or players on the network. Each computer in the pool runs its own instance of the **Graphics Host**.

The only difference comes with respect to *Hardware Rendering* with a Graphics Processor. In this case, the distribution of the work done by the **Graphics Host** is limited by the number of computers the players have physical access to. For each player, you can only have one computer or **Game Client** rendering the graphics of the *Game World* that the player can physically see. And the player requires physical access to two computers or **Game Clients** to double the amount of rendering. With two **Game Clients** looking at the *Game World* from the viewpoint of two different **Camera Objects**.

For example, one **Camera Object** may be looking towards the left-hand side of the player and another looking towards the right-hand side. Or one may be looking towards front of the player and another looking towards the back. Or one may be looking through the left eye of the player and another looking through the right eye.

The latter configuration would suit Virtual Reality Headsets where the player can put on goggles which show two different views of the *Game World*, through the player's character's right eye and left eye.

As with the distribution of the work done by the **Physics Host**, distributing the work done by the **Graphics Host** would require you to purchase more equipment for the computer network. That could handle the greater bandwidth or throughput required to send information across the computer network.

This method for distributing the rendering of the **Graphics Host** has already been described in the subchapter entitled

Multi-User Distributed Client Server Form

in

The Event-Database Architecture for Computer Games: Volume 1, Software Architecture and the Software Production Process.

Alternatively, the **Graphics Host** does not need to render 2D or 3D graphics at all. You can begin with a simple form of the **Graphics Host** which shows the *Game World* using different forms of projection apart from **PERSPECTIVE PROJECTION**. And then later on replace this with a **Graphics Host** in a more complex form which renders a Graphical *User Interface*, when you get closer to the end of the production process. This enables you to concentrate on developing the core features of the *game design* without worrying about the details of the Graphical *User Interface* until later on in the production process.

These alternative forms of projection would include

- **ORTHOGRAPHIC PROJECTION**
- **ISOMETRIC PROJECTION**
- **CHROMATIC PROJECTION**
- **AUDIO PROJECTION**
- **TEXTUAL PROJECTION**

In an **Orthographic Projection**, three views of the *Game World* are shown on the screen, from the top, side and front of a cube centred around the player. The shapes of **Game Objects** are not distorted. Therefore, it is quick and easy to project the three views onto the screen because the mathematics involved is simpler.

In an **Isometric Projection**, which is a form of **Orthographic Projection**, the vertices within the cube centred around the player are rotated 30 degrees around the X-axis or the vector pointing to the right of the player, and 120 degrees around the Z-axis or vector pointing up from the player. Therefore, a 3D view of the shapes of **Game Objects** is displayed in one view. This is quick and easy to project because again the mathematics involved is simpler.

In a **Chromatic Projection**, all the **Game Objects** in the *Game World* are made from shades of a few colours and have no hint of black, white or grey. Any sense of depth or distance of an Object is determined by its shade. Every **Object** is projected

onto one of a limited number of 2D planes on the screen. All the **Objects** on the same plane have roughly the same depth and the same shade of colour. Therefore, a 3D view of the shapes of **Game Objects** is displayed in one view. And this is quick and easy to project because again the mathematics involved is simpler and all you need is one colour or *Texture* to render everything in the *Game World*.

There is a wide spectrum of examples of **Chromatic Projection**. On one extreme are Japanese wave paintings where all the items in the world are presented with just four colours, including white. And on the other extreme there are abstract paintings by the likes of Barnett Newman, Mark Rothko and Alan Ebnother, where all the items are just presented with one or two colours. Some may say that in the latter case their paintings of the world are so abstract and far removed from reality that they are useless. But that is not the point. The point is whether you can represent the *Game World* on the screen, without photo realism. And yet convey information just by the simple use of colours.

In an **Audio Projection**, every **Game Object** in the *Game World* is assigned a unique sound in the *Database Table* of *sound streams*. See the chapter entitled

Sound Stream Table

in the *LPmud data design*. Now some animated **Objects** may already have natural sounds assigned to them by the *game design* e.g. a river, a cow or a sheep. But you have to assign sounds to all **Objects**, including inanimate **Objects** as well such as a rock, a chair, a table or a button on a menu. These would include assigning sounds to **Text Objects** which should be an actor audibly speaking the text being displayed by that **Object**. This would also include assigning a unique sound for all the NPCs and the interactive Player Characters.

The sounds of the **Objects** nearby would be periodically played by the **Sounds Host** and heard by the player. And this mixed sound will correspond to all the items nearby in the *Game World* or on a menu. The screen itself will not show the items in the *Game World* or on the menu but a representation of the sounds being heard. For example, it may just show the names of the sounds being generated. Or it may show a 2D or 3D wave produced by the mixture of sounds which were being generated in the *Game World* by the **Sounds Host**. Therefore, the **Game Objects** would be quickly and easily represented on the screen. In a form which is useful for both *Sound Designers*, players or staff who suffer from a poor eyesight for one reason or another and rely on their hearing.

In a **Textual Projection** the *User Interface* is displayed using only text, like the one used by the original *LPmud*.

For example, suppose there were some parts of the *game design* where the player has to go to one location in the *Game World* and search for some item. Let us say this item is some hidden sword in a farmer's field. And the player has to return this sword to another character, a swordsman in a mountain pass who has lost the sword. Now all that matters in this case is the **Game Objects** for

the farm
the farmer's field

the hidden sword
the mountains
the mountain pass
the swordsman in the mountain pass.

All that matters is that the player can go to the farm. That the player can go to the farmer's field. That the sword is initially hidden and not visible in the field. That the player has to search and find the sword in the field. That the player can pick up the sword. That the player can carry the sword back to the swordsman. That the player can give the sword back to the swordsman. That the swordsman rewards the player for returning the sword.

So all the **Graphics Host** has to show is the farm, the farmer's field, the hidden sword, the mountains, the mountain pass and the swordsman. And show the player's distance from these items. And show the distance changing as the player gets closer or moves further away. And show the exits or directions the player can move in from the current location. And show the player's commands e.g.

[Farm dist: 1000m]
[Exits: n s e w]
> e
[Farm dist: 800m]
[Exits: n s e w]
> e
[Farm dist: 600m]
[Exits: n s e w]
> e
[Farm dist: 0m]
[Exits: n s e w]

And when the player reaches, the farm, the game shows the different parts of the farm the player can see and move into. Showing the parts ordered by distance from the player, in the same order these would be rendered in a Graphica*l User Interface*. by the **Graphics Host**. That is to say the furthest part is shown first, then the second furthest and then the third furthest and so on e.g.

[Farmer's Field South dist: 30m]
[Farmer's Barn dist: 10m]
[Farmer's house dist: 5m]
[Exits: n s e w]

And then show the different parts of the farmer's field when the player reaches the field e.g.

[Farmer's Field North dist: 100m]
[Farmer's Field East dist: 100m]
[Farmer's Field West dist: 100m]

[Farmer's Field South dist: 0m]
[Exits: n s e w]

And when the player reaches the spot containing the hidden sword, show the hidden sword e.g.

[Farmer's Field East dist: 0m]
[A hidden sword dist: 0m]
[Exits: n s e w]

And when the player has picked up the hidden sword, show the resultant action e.g.

get sword

[You pick up the sword, and throw it over your shoulders into your backpack]

And after the player has picked up the sword, show the path to the mountain pass e.g.

[A mountain dist: 1000m]
[Exits: n s e w]
> ne
[A mountain peak dist: 800m]
[A mountain pass dist: 60 m]
[Exits: n s e w]
> ne
[A mountain peak dist: 740m]
[A mountain pass dist: 0m]
[A swordsman]
[Exits: n s]
> n

And when the player gives the hidden sword to swordsman, show the reward the player receives e.g.

[A mountain peak dist: 740m]
[A mountain pass dist: 0m]
[A swordsman]
give sword to swordsman
[A swordsman]: Thank you for returning my sword! Here, take this as a token of my gratitude!
[A swordsman hands you 1000 talents of gold]
[Exits: n s]

In this way, the **Graphics Host** with a **Textual Projection** would show a *User Interface* that would be akin to a storyboard for a movie script. It will show you a

rough outline of each scene in the movie, without much detail. Until the 2D *polygons*, 3D Meshes, *Textures*, *Texture coordinates*, *Materials* and animations required to render these **Game Objects** had been built.

Another advantage of a **Textual Projection** is that it can be used to train an *Artificial Neural Network* which understands, generates and interprets natural language or human language, to play the game. An example of this is a **Language Learning Model** mentioned in subchapter 1.3.5.2 - Application:Flaws in Back Propagation.

The **Projection** will give the **Model** a textual description of a room or location or scene in the *Game World*. And the **Model** can use this to predict the missing word at the end of that sentence or paragraph. That missing word being the commands or words that the characters being controlled by interactive or human players enter, through the *User Interface*, in response to that description. The *Artificial Neural Network* would be trained automatically by being fed the descriptions of the rooms or locations or scenes in the *Game World*, as the **Initial Inputs** of its **Training Data**. And the commands or words that the players enter in response to this description, and the probability of those commands, as the **Final Outputs** of its **Training Data**.

And once trained, whenever the *Artificial Neural Network* is fed the description of a room or scene in the *Game World* that an NPC enters, it will try to predict the missing word or command that goes with that description. And the word it predicts will be given as a command to that character to execute. And play through that scene using what *Artificial Neural Network* has learnt from the **Training Data**.

What is more there will be no bias in the **Training Data**. So long as every players' reaction to the description of that room or location or scene in the *Game World* was immediately recorded in the **Final Outputs** of the **Data**. And the *Artificial Neural Network* was immediately and automatically trained with the new entry in the **Data**. Through **Forward Propagation** from the **Initial Inputs** which is the description of the room, to the **Final Outputs**, which is the player's command. Followed by a **Back Propagation** to adjust the **Weights** of the **Inputs** of the artificial *Neurons* due to the loss in the **Final Outputs**.

You cannot train an *Artificial Neural Network* to do this automatically with a *Graphical User Interface* when the *game design* is changing. Someone has to manually select metrics in the *Game World* which can act as the **Initial Inputs** of the **Training Data.** And someone has to manually select the metrics in the *Game Controllers* or the *Game World* to act as the **Final Outputs** of the **Training Data**. And there will be a bias in this selection. Furthermore, the metrics in the *Game World* and in the *Game Controllers* may change as the *game design* changes. And items were added, removed or edited in the *Game World*, or commands were added, removed or edited from the *User Interface*.

Another advantage of all these different forms of projection is that they could be used to improve the experience of the players in a multiplayer game. They could give the players the ability to choose between

Orthographic Projection
Isometric Projection
Chromatic Projection
Audio Projection

Textual Projection
Perspective Projection.

And if the players want to, for example, reduce the latency or lag, on their **Game Client**, in response to any commands they issue across the computer network, then they could switch from a **Perspective Projection** to a simpler **Chromatic Project** or **Textual Projection**. If some of players suffered from poor eyesight, then they could switch to an **Audio Projection**. If some of the players had motion sickness from watching the *Game World* from a **Perspective Projection**, then they could switch to an **Orthographic Projection**. If some of the players suffered from epileptic fits from the images of the *Game World*, then they could switch to a **Chromatic Projection**.

1.3.7.7 Application: Power Is Limited, Imagination Is Not

The sixth rudimentary flaw to the claim that the longer *Software Rendering* and *Hardware Rendering* processes of commercial *game-engines* are necessary is the limitations these impose. Not just the limitations of the power that Graphics Processors require alluded to in the previous subchapter. But also the limitations these impose on human imagination.

Now the goal of *Hardware Rendering* processes and Graphic Processors is to produce better games. That is why these are the focus of the *software architectures* of the most popular commercial *game-engines*. But there are other ways you can produce better games which are focused on human imagination. And this is the focus of the **Event-Database Architecture**. So what is a better game?

There is one school of thought in the Computer Games industry, who advocate Photorealism. They would define a better game as a game with better graphics. That is to say a game with better Photorealism, with more realistic and immersive *Game Worlds* is better than one with less Photorealism.

There is a second school of thought. They would define a better game as a game with better gameplay mechanics or features. That is to say, the greater the number of ways the players have available to reach the goals of the game, the better the game is. The greater the number of bodies (i.e. characters, creatures, NPCs, environments, artifacts in these environments, animate and inanimate bodies) in the *Game World*, the greater the number of possible interactions between these bodies, the greater the number of commands available in the *User Interface*, that the players can use to reach their goals, the better the game is.

There is a third school of thought that would claim a middle ground which defines it both ways. That is to say, a better game is one which has both better immersive Photorealism and better gameplay mechanics or features.

Now most *Software Developers* would like to believe they fall into this third school of thought. That they would not favour one definition over the other and would always choose a more balanced approach or middle ground when they make computer games.

But what is the truth? Let us look at the consequences that result from these three definitions. And compare the results with what we see in the Computer Games industry.

Now, the natural consequence, in the case of the first definition of a better game based on Photorealism, would be the concentration of resources in the Computer Games industry on more and more demanding rendering algorithms to produce Photorealism. That in turn require more and more advanced *Hardware Rendering* processes and Graphics Processors. That in turn require more and more electrical power and resources to run.

The natural consequence, in the case of the second definition of a better game based on greater gameplay mechanics or features, would be a concentration on human resources and imagination in the industry. To come up with a greater and greater number of bodies in the *Game World*, a greater number of interactions between these bodies and a greater number of commands in the *User Interface*. That could be used in a greater variety of ways to achieve the goals of a game.

The natural consequence, in the case of the third definition of a better game based on both Photorealism and gameplay mechanics, would be an equal rise in demand of resources. That is to say an equal rise in demand for more advanced *Hardware Rendering* processes and Graphics Processors. And an equal rise in demand for more human resources to imaginatively construct greater and greater number of bodies in the *Game World*, greater number of interactions between those bodies and greater number of commands in the *User Interface*.

What you see in the Computer Games industry is not the latter two, but only the former consequence. This is reflected in the marketing material of the popular commercial *game-engines*. Here are some examples:

'UNREAL ENGINE 5 – WHAT IT'S ALL ABOUT

...

UNREAL ENGINE 5 – EXPECTATIONS

Understandably, there are high expectations for Unreal's newest launch. Last year, an article from Perforce said UE5 would change the industry because "…it will enable truly immersive experiences – while reducing the complexity of building games, as well as in film and animation."

It's not just developers who are excited about what next-gen graphics can bring. Some recent studies reveal that upwards of 75% of gamers make purchases based on graphics quality.

...

DID UNREAL ENGINE 5 (EARLY ACCESS VERSION) LIVE UP TO EXPECTATIONS?

When UE5 was first announced, Epic made it clear what the main goal was: "[to] achieve photorealism on par with movie, CG, and real life," all while keeping these tools accessible to teams in the industry.

This is a huge promise. They didn't say it was meant to look "good;" they claimed to keep up with photorealism in every industry. So the question is: Did they live up to it?

...

MetaHumans

The announcement and early access of MetaHuman Creator resulted in whispers throughout the industry of what impact this amazing software could have on game development moving forward. Shortly after opening MHC, you'll notice just how easy it is to create photorealistic characters, customized to your needs….'

Source: Unreal Engine 5 – What It's All About? © 2021. Incredibuild. Joseph Sibony. Page 155

'*Unity and Unreal Engine are two of the most prominent game engines in the industry, known for their cutting-edge capabilities in rendering photorealistic graphics. Both engines have been extensively used in the development of AAA games, architectural visualizations, and various other applications that demand high-fidelity visuals. In this exploration, we will delve into the strengths and distinguishing features of each engine when it comes to achieving photorealistic graphics.*

UNITY

... However, in recent years, Unity has made significant strides in enhancing its graphics capabilities, cementing its position as a powerful engine for photorealistic rendering.

1. ***High-Definition Render Pipeline (HDRP)****: Unity's HDRP is a state-of-the-art rendering pipeline designed specifically for high-fidelity graphics. It supports advanced features such as real-time global illumination, physically-based rendering (PBR), and high-dynamic-range (HDR) lighting, enabling developers to create highly realistic and visually stunning environments.*
2. ***Scriptable Render Pipeline****: Unity's Scriptable Render Pipeline (SRP) allows developers to customize and extend the rendering process... This flexibility enables advanced techniques for achieving photorealistic results tailored to specific project requirements.*
3. ***Real-Time Ray Tracing****: With the introduction of real-time ray tracing support, Unity has opened the door to accurate simulations of light behavior, enabling realistic reflections, shadows, and global illumination effects...*
4. ***Asset Importers and Optimization****: Unity's robust asset import pipeline and optimization tools ...ensuring efficient rendering and performance optimization for photorealistic graphics.*
5. ***Integration with Industry-Standard Tools****: Unity seamlessly integrates with industry-standard tools such as Autodesk Maya, 3ds Max, and Substance Painter, allowing artists and developers to leverage their existing workflows and pipelines for creating photorealistic content.*

   …

   Unreal Engine

   …

6. ***Chaos Physics and Destruction:*** *Unreal Engine's Chaos physics and destruction systems enable realistic simulations of rigid body dynamics, soft body deformations, and large-scale destruction events, adding to the overall level of realism and immersion.*'

Source: Unity vs Unreal: Exploring Cutting Edge of Photorealistic Graphics © 2024. Oodles Technologies. Page 157

The bias towards Photorealism in the Computer Games industry is so much so that the Photorealism in the popular *game-engines* is marketed as a science which has application in Film, Manufacturing and Architecture. When to be honest it is, if anything, a misapplication.

It is a misapplication because you are not using these sciences as diagnostic or prognostic tools, to analyse the external and internal mechanics of physical bodies in the real world. Instead, you are using these sciences to give the external appearance of imaginary bodies in a *Game World* the external appearance of physical bodies in the real world. Ignoring the differences between the internal mechanics of the imaginary bodies and the internal mechanics of physical bodies. The internal mechanics of the imaginary bodies in the *Game World* are hollow and made up of pixels. The internal mechanics of physical bodies in the real world are solid and made up of physical materials.

The mathematics involved in creating Photorealism, including Newtonian Mechanics, Kinematic equations and Linear Algebra, maybe sciences. But their application in Photorealism is not a science. It is at best an art, and at worst a sleight of hand, a magic trick for kids or a circus act with smokes and mirrors.

Nevertheless, given the obsession with Photorealism in the Computer Games industry, it is not surprising then that the definition of a better game as a game with better Photorealism is the most prevalent. However, this means that the capacity of commercial *game-engines* which cater to this definition will always be limited. Whereas the capacity of the **Event-Database Architecture** which does not cater to this definition is not limited in the same way.

For power is limited but the human imagination is not. That is to say the electrical power required to run these Graphics Processors, that in turn produces this Photorealism, that in turn the commercial *game-engines* rely on, will always be limited. But the imaginative ways in which the staff or players can come up with gameplay mechanics or gameplay features are not limited.

And this is what the **Event-Database Architecture** caters for. Namely, the ability for large numbers of staff and players to use their imagination, to collaborate to a scale which is not possible with commercial *game-engines*. It simply does not fit into the paradigm of a single-user *game editor* which these *game-engines* are based on (see the definition of *game editors* in the Glossary).

With the **Event-Database Architecture**, they can use their imagination to generate gameplay mechanics or features, including the following:

1. a single unified model of physics rather than the multiple models of physics used by the commercial *game-engines* (i.e. crude micro models of molecular and particle physics run by Graphics Processors, and crude macro models of mechanical physics run by Central Processors)
2. more unique NPCs
3. NPCs with better *Artificial Intelligence*
4. bigger crowds or populations of NPCs in a city or town
5. better simulation of complex social and biological ecosystems
6. more original stories and locations, not based on film franchises
7. puzzles based on reflection, refraction and absorption of light.

An example of a latter would be a part of the *Game World* of *LPmud* where the player uncovers a tomb buried in a mountain. The tomb is made up of a network of chambers connected by hallways running through the mountain. At the centre of the mountain lies the body of an ancient king buried in the main chamber. And to enter the main chamber, the player has to direct sunlight from outside the mountain, through the network of dark doorways and hallways, to a switch on the side of the door that leads into the main chamber. And that in turn requires the player to place a series of mirrors which guide the light from outside of the mountain, through the network of dark hallways, to the entrance of the main chamber.

Now the gameplay feature here is the reflection of light. It requires sunlight to be reflected off a sequence of mirrors in the right order, beginning with a mirror outside the mountain, and ending in a mirror next to the entrance of the main chamber. The rendering algorithms of commercial *game-engines* make no provision for this.

Firstly, typically the *Hardware Rendering* processes and Graphics Processors these *game-engines* use will not render the reflection off the surface of mirrors outside of the area of visibility immediately in front of the player's eyes or camera, in the *Game World*. This would be done nominally for the sake of efficiency and improving the performance of the game. But this efficiency would actually end up breaking the Photorealism in the case of this gameplay feature.

Secondly, even if you extended this area of visibility to cover the whole mountain, the order in which the *Hardware Rendering* process and Graphics Processor would render the reflection off the surface of each mirror would be arbitrary. You would have to heavily customise the *Hardware Rendering* process to render the reflections off the mirrors in the right order.

Thirdly, say you have seven mirrors which the player has to align to direct the sunlight to the main chamber, numbered 1–7. In the order these had to be set up to open the chamber. Beginning with the mirror outside the mountain, directing the sunlight into the mountain and ending with the mirror next to entrance of the main burial chamber, inside the mountain. When the mirrors have been correctly aligned, mirror 1 is going to contain a reflection of the sun and the image of mirror 2. And mirror 2 is going to contain a reflection of the image of mirror 1 and mirror 3. And mirror 3 is going to contain a reflection of mirror 2 and mirror 4 and so on and so on. But no matter what order in which you choose to render the reflections off the surface of the mirrors, either from 1 to 7 or 7 to 1, the reflection of each mirror is always going to be incomplete. That is to say, it will show the reflection of an adjacent mirror whose surface is blank because the image on that adjacent surface has not been rendered yet. If you start with mirror 1, the reflection of mirror 2 is going to be blank, because the surface on that has not been rendered yet. If you move to mirror 2, then the reflection of mirror 1 will be incomplete because it would contain the blank surface of mirror 2. And the reflection of mirror 3 is going to be blank, because that has not been rendered yet, and so on and so on.

Fourthly, typically **Objects** which were far away from the player would either not be updated or rendered, in most commercial *game-engines*. Only the **Objects** which were close enough to the player for the player to notice would be updated or rendered. This means if the player were deep down inside a mountain, looking far above, through a series of mirrors at a view outside the mountain, then some or all of the **Objects** which were outside the mountain would not be updated or rendered.

So the features that the player would normally see outside the mountain, like the sun moving across the sky, clouds being blown by the wind, trees or grass swaying in the wind, or other characters walking by and so on, would either be static or missing. When viewed from deep inside the mountain and through a series of mirrors. This would be done nominally for the sake of efficiency and improving the performance of the game. But this efficiency would also actually end up breaking the Photorealism in the case of this gameplay feature.

Fifthly, the *Hardware Rendering* process and Graphics Processor used by these commercial *game-engines* have no provision for telling you whether or not the mirrors have been aligned correctly. Such that the sunlight has been successfully directed from outside the mountain to the main chamber deep inside the mountain.

In the case of gameplay features like this, which depend on the reflection of light, it is far simpler to bypass the *Hardware Rendering* process and Graphics Processor entirely. And to use the Central Processor instead.

Now with the **Event-Database Architecture**, this feature is trivial to implement. You can see an example of this at the end of the subchapter:

Graphics Host

in

The Event-Database Architecture for Computer Games: Volume 1, Software Architecture and the Software Production Process.

Basically, you set up a series of **Camera Objects** one behind each mirror facing the front or reflective surface of the mirror. And you set up each **Camera Object** to project what it sees onto the reflective surface or the *Texture* of the mirror. During each projection, the *Software Rendering* process of the **Graphics Host** will create an entry in the *Database Table* or **Projected List** for each **Camera Object**. That list all of the **Objects** visible by that **Camera Object**, including other mirrors.

So if you look at the **Projected List** of the first **Camera Object** or first mirror, then you should see the **Object** for the sun and the **Object** of the second mirror. Assuming the player has correctly aligned the first mirror. And if you look at the **Projected List** of the second **Camera Object** or second mirror, then you should see the **Object** of the first mirror and **Object** of the third mirror. And so on and so on. Therefore, by traversing the hierarchy of references in the **Projected Lists** of all the mirrors, from the beginning to the end of the sequence, you can determine whether the player has been successful. And the sunlight has been directed, from outside the mountain to the entrance of the main chamber.

That is the default solution that comes with the standard **Event-Database Architecture**. But the staff or players do not have to stick with that solution. They can use their imagination to come up with a better solution based on the rudimentary principles of the **Event-Database Architecture**.

For example, they could add a **PRIMARY REFLECTION EVENT** and a **SECONDARY REFLECTION EVENT** to the set of **Events** recognised in the **Architecture**.

A **Primary Reflection Event** would be an **Event** sent by the **Graphics Host**, whenever it projected **Objects** onto a *Texture*. In this case, when it projected **Objects** through the **Camera Object** behind each mirror. This **Event** would be a property of a **Camera Object** in the *Database Fields* of its **Camera Object Record**. And not all **Camera Objects** will have this property. Only the **Camera Objects** which project **Objects** onto the reflective surface of a mirror would have it set.

A **Secondary Reflection Event** would be an **Event** which followed on from the **Primary Reflection Event**, which would be sent to a **Game Object** that was being reflected by the mirror. This will be a property of any **Game Object**, including **Camera Objects**. And not all **Objects** will have this property set. Only **Objects** which wanted to receive an **Event** when these were reflected in a mirror would have it set.

Furthermore, they could assign to each **Camera Object** behind a mirror, a *Database Table* that maps the **Secondary Events** it receives, to **Primary Events** it should send.

Now when a **Graphics Host** projects **Objects** through the first **Camera Object** behind a mirror, onto the surface of the first mirror, and that **Camera Object** has a **Primary Reflection Event** set in its properties, it would attach all the **Secondary Reflection Events** of the projected **Objects** to that **Primary Reflection Event**. This includes the **Secondary Reflection Event** of the first **Camera Object**. And the causer of all those **Secondary Reflection Events** will be set as the f**irst Camera Object**. And then it will send that **Primary Reflection Event**.

When the second **Camera Object** behind the second mirror receives its **Secondary Reflection Event**, it will check the causer of that **Secondary Reflection Event**. If the causer were a mirror and had a **Primary Reflection Event**, then it would attach all of the **Secondary Reflection Events** attached to the mirror's **Primary Reflection Event** to the second mirror's **Primary Reflection Event**. In this case, assuming the player had correctly aligned the first mirror to reflect sunlight onto the second mirror, all of the **Secondary Reflection Events** of the first mirror would be attached to the **Secondary Reflection Events** of the second mirror. After that the **Graphics Host** would then repeat the same process with the second mirror as it did with the first mirror.

That is to say, it will project all the **Objects** in front of the second mirror, through the second **Camera Object** behind the mirror, onto the surface of the mirror. It would attach all of the **Secondary Reflection Events** of all the projected **Objects** to the **Primary Reflection Event** of the second **Camera Object**. But unlike the first case, this **Primary Reflection Event** would already have all of the **Secondary Reflection Events** from the first projection through the first **Camera Object** on it. The **Graphics Host** will set the causer of all those **Secondary Reflection Events** as the second **Camera Object**. And it will then send that **Primary Reflection Event**. And so on and so on.

As the **Graphics Host** projects the **Objects** in front of each mirror, through the **Camera Objects** behind the mirrors in the sequence, the **Secondary Events** following on from **Primary Reflection Event** of each one in turn grow longer than the preceding one.

When the last **Camera Object** behind the last mirror in the sequence received its **Secondary Reflection Event**, it would check the causer of that **Secondary**

Reflection Event. If it were a mirror with a **Primary Reflection Event**, then it will look through all the **Secondary Reflection Events** attached to that **Primary Event**. To verify that all the mirrors in the sequence and the switch next to the entrance of the main chamber were being reflected by that mirror. If all of mirrors and the switch were being reflected, then it would look up its *Database Table* that maps the **Secondary Events** it received to **Primary Events** it should send out. And assuming that this *Table* mapped its **Secondary Reflection Event** to the **Primary Event** that would open the door to the main chamber, it would send the **Primary Event**. And the door of the main chamber would be opened.

Now anyone can control this gameplay feature simply by editing *Database Records* and *Database Tables*. Without the need for any specialist knowledge as would be the case with the commercial *game-engines*. That rely on *Game Programmers*, known as Graphic Programmers, who specialise in graphics and writing *Graphic Shaders* to implement a similar feature.

This algorithm will not only tell you when the sunlight was being reflected, through a sequence of mirrors, from outside of a mountain, to a tomb buried deep inside the mountain. It will also tell you when any **Object** was being reflected by a mirror. Or when an **Object** was being reflected off each mirror in a sequence of mirrors. Or when any **Object** has been successfully reflected from the beginning to the end of the sequence of mirrors, from one point in the *Game World* to another. Or when sunlight is being reflected off a series of mirrors onto a vampire with lethal effect.

Of course, for this algorithm to work, you would need to clear all of the **Secondary Reflection Events** attached to all of the **Primary Reflection Events**, at the end of each *Unit of game time*, or *Frame*.

1.3.7.8 Application: Line of Sight Graphics

An alternative method to detecting the **Game Objects** in the line of sight of an NPC, using the **Physics Host**, would be to use the **Graphics Host**. This method is simpler because it does not involve creating new 2D *polygons* or 3D models.

Instead, you just have to create a new **2D Camera Object** or **3D Camera Object** and place this on the head of the 2D *polygon* or 3D model of the NPC, facing whatever direction that NPC was facing. You would then add this **Camera Object** to list of active cameras in the **2D Camera List** or **3D Camera List**. And set the **Projection Target Field** of that new **Camera Object** to nothing. So that whatever was seen through that camera would not be displayed on the computer screen using *Hardware Rendering*.

Instead the **Graphics Host** would use that **Camera Object** only for *Software Rendering* to project the bounding box of the **Objects** in front of the NPC. The results of the projections would not be put in the main **Projected Shapes** *Database Table* used to display the *Game World* to the player. Instead, it would be put in a numbered **Projected Shapes** *Table*, given the same number as the instance of that NPC in the *Game World*.

For example, the first three NPCs in the *Game World* would have *Tables* named

Projected Shapes #1
Projected Shapes #2
Projected Shapes #3

After the projections, the criteria for selecting the ones from **Projected Shapes**, which would go into **Projected List**, would be whether the bounding box of the **Object** fell within the viewing frustum of the **Camera Object**.

The **Actions** for the **2D Player Object** or **3D Player Object**, which controlled that NPC, would then look through the **Projected List**. To find out the **Object IDs** of the **Game Objects** in front of that NPC, and react accordingly.

1.3.8 Application: Procedurally Generated Quests

The system of **Events** for the game *LPmud* is so versatile that it will allow you to create a **PROCEDURALLY GENERATED QUEST SYSTEM** in the *Game World*. That is to say it will allow to generate different types of quests, quests to kill a special target, quests to find a special item and quests to escort special character, where the special target, special item or special character changes each time. And the location of that target, item or character changes each time.

The first part of this **Procedurally Generated Quest System** would require four invisible **Game Objects** or **QUEST MARKER OBJECTS** which marked the four corners of a quadrilateral area, in the *Game World*, where these quests could take place. These **Marker Objects** would be placed manually in the *Game World* by the *Game Designers* editing the **Game Database**.

The second part of this **Procedurally Generated Quest System** would require an invisible **Game Object** or **QUEST SPLINES GENERATOR OBJECT** that will generate a fixed number of splines, of random length, with a fixed minimum length, following a random path, from one random corner of this boundary to the opposite corner. Each spline would be an invisible **Game Object** or **QUEST SPLINE OBJECT** with a list of points along that spline, from the beginning to the end. Each spline would be put into the same *Database Table* reserved for splines. And the *Database Record* for each spline would include a *Database Field* which listed these points in order. The **Quest Splines Generator Object** would respond to the **Initial Reset Event** to generate the splines and send a **QUEST SPLINES COMPLETE EVENT** when it had finished generating.

The third part of this **Procedurally Generated Quest System** would require an invisible **Game Object** or **QUEST WAYPOINTS OBJECT** which would generate *Waypoints* along each spline, with a fixed minimum distance between each *Waypoint*, from the beginning to the end. Each *Waypoint* will be represented by one highly buoyant **Invisible 3D Game Object** with a vector pointing to the next *Waypoint* and a distance to the next *Waypoint* along the spline. Each of these buoyant **Invisible 3D Game Objects** would have its height raised up until it reached the surface of the *Game World*, if it was generated beneath the surface. Or it would have its height lowered downwards until it touched surface, if it was generated above the surface. This adjustment could be done by the **Physics Host** if it were modified to recognise these highly buoyant **Invisible 3D Game Objects**. And force them up to the surface if these were underneath the surface, or lower them down with the force of gravity, if above the surface. This **Waypoints Object** would respond to the **Quest Splines Complete Event** and start generating the buoyant Waypoints.

The fourth part of this system would require an invisible **Game Objec**t or **KILL QUEST HANDLER OBJECT** which controls the quest where the player has to

kill a special target. This **Kill Quest Handler** will have these additional properties or *Database Fields*:

1. a **QUEST GIVER OBJECT**
2. a **QUEST PROMPT OBJECT**
3. a **QUEST TARGET OBJECT**
4. a **QUEST RECEIVER OBJECT**

The **Quest Giver** would be a **3D Animation Object** of a character in the *Game World* chosen at random from a *Database Table* full of possible **3D Animation Objects**. The *Game Designers* could edit this *Database Table* and add more characters to the table to add more variety to the system.

The **Quest Prompt** is a **Text Object** which the **Quest Giver** displays to the player when the player approaches for the first time chosen at random from another *Database Table* full of possible **Text Objects**. The **Text Object** would spell out the player's assignment to kill the **Quest Target**.

The **Quest Target** is **3D Animation Object** of another character in the *Game World* chosen at random from a *Database Table* full of possible targets or **3D Animation Objects**. Again the *Game Designers* could edit this *Database Table* and add more characters to the table to add more variety to the system.

The Quest Receiver Object is the Player Object or player who was given the quest.

The **Quest Giver** and **Quest Target** would be chosen and generated in the *Game World* by the **Kill Quest Handler** if one of these did not exist. When the **Kill Quest Handler** responded to the **Initial Reset Event** or the **Periodic Reset Event**. And placed both in two different *Waypoints* chosen at random from the set generated earlier by the **Quest Waypoints Object**.

The **Quest Giver** would respond to the **Object Entered Event**, from the **Player Object**, when the player approached it. And it would display the **Text Object** with the player's assignment.

The **Quest Giver** would also respond to the **Object Dead Event**, from the **Quest Target**, when the player had killed the target. And it would replace its old **Secondary Proximity Event**, **Object Entered Event**, in its *Database Record*, with a new **Secondary Event**, a **QUEST REWARD EVENT**. That would be automatically triggered when the player returned back to the **Quest Giver**. At which point the **Quest Giver** would respond to the **Quest Reward Event** and reward the player for completing the assignment.

The fifth part of the **Procedurally Generated Quest System** would require an invisible **Game Object** or **FIND QUEST HANDLER OBJECT** which controls the quest where the player has to find a special item. This would be similar to the **Kill Quest Handler**. It will also have these additional properties or *Database Fields* in its *Database Record*:

1. a **Quest Giver Object**
2. a **Quest Prompt Object**
3. a **QUEST LOST OBJECT**
4. a **Quest Receiver Object**

The difference is that the **Quest Lost Object** would be a **3D Model Object** of a special weapon, armour or item in the *Game World* chosen at random from a *Database Table* full of possible special items or **3D Model Objects**. Again the *Game Designers* could edit this *Database Table* and add more special items to the table to add more variety to the system.

The **Quest Giver** and **Quest Lost Object** would be chosen and generated in the *Game World* by the **Find Quest Handler** if one of these did not exist. When the **Find Quest Handler** responded to the **Initial Reset Event** or the **Periodic Reset Event**. And placed both in two different *Waypoints* chosen at random from the set generated earlier by the **Quest Waypoints Object**.

Again the **Quest Giver** would respond to the **Object Entered Event**, from the **Player Object**, when the player approached it. And it would display the **Text Object** with the player's assignment.

The **Quest Giver** would also respond to the **Object Moved Event**, from the **Quaye Lost Object**, to know when the player had found the special item and picked it up. And it would replace its old **Secondary Proximity Event, Object Entered Event**, in its *Database Record*, with a new **Secondary Event,** a **Quest Reward Event**. That would be automatically triggered when the player returned back to the **Quest Giver**. At which point the **Quest Giver** would respond to the **Quest Reward Event** and reward the player for completing the assignment.

The sixth part of the **Procedurally Generated Quest System** would require an invisible **Game Object** or **ESCORT QUEST HANDLER OBJECT** which controls the quest where the player has to escort a special character in the *Game World*. Again this would be similar to a **Kill Quest Hander**. It would also have these additional properties or *Database Fields* in its *Database Record*:

1. a **Quest Giver Object**
2. a **Quest Prompt Object**
3. a **Quest Target Object**
4. a **Quest Spline Object**
5. a **Quest Receiver Object**

The difference will be that the **Quest Target** is **3D Animation Object** of another character in the *Game World* that moves along the *Waypoints* of the splines generated earlier by the **Quest Splines Generator Object**. The **Quest Target** will always begin from the first *Waypoint* on the spline and move progressively forward to the next *Waypoint*, until it reached the end of the spline.

And the **Quest Spline Object** that the **Quest Target** would move along will be the spline chosen at random from a *Database Table* of splines or **Quest Spline Objects** that was generated earlier.

Again the **Quest Giver** would respond to the **Object Entered Event**, from the **Player Object**, to detect when the player approached it. And it would display the **Text Object** with the player's assignment.

The **Quest Target** would also respond to the **Object Entered Event**, from the **Player Object**, to know when the player was approaching. And when to start moving from its current *Waypoint* along the spline to the next *Waypoint*. Until it reached

the last *Waypoint* at which point the **Quest Target** would generate a **QUEST COMPLETE EVENT**.

The **Quest Giver** would respond to the **Quest Complete Event**, from the **Quest Target**, to know when the player had escorted the target to the last *Waypoint*. And again as with the other quests, it would replace its old **Secondary Proximity Event, Object Entered Event**, in its *Database Record*, with a new **Secondary Event,** a **Quest Reward Event**. That would be automatically triggered when the player returned back to the **Quest Giver**. At which point the **Quest Giver** would respond to the **Quest Reward Event** and reward the player for completing the assignment.

Finally, the **Quest Giver** would respond to the **Object Dead Event**, from the **Quest Target**, to know when the target had been killed. And display a **Text Object** to the player to inform them that they had failed the assignment. Note that the Quest Target only moves to the next Waypoint along the spline when the player comes within close proximity of it. So if the player abandons the Quest Target at any Waypoint, it will not move to the next Waypoint. And the player will never be rewarded for completing the quest. And the player will implicitly have failed the assignment.

1.4 STEP 4: *LPmud* DATA DESIGN

The next step in the **Event-Database Production Process**, after the *technical design*, would be to create a *data design*.[28] You can see the vision for data design in Figure 1.31.

This *data design would* be written by the *Database Administrator.* This defines the language of the **Event-Database Production Process**. If there is a narrative in a prior *game design*, this language should reflect that prior one. But if there is no narrative because the *game design* is incomplete, then this defines the language of the narrative of the *Game World* which is about to be built. This should been written

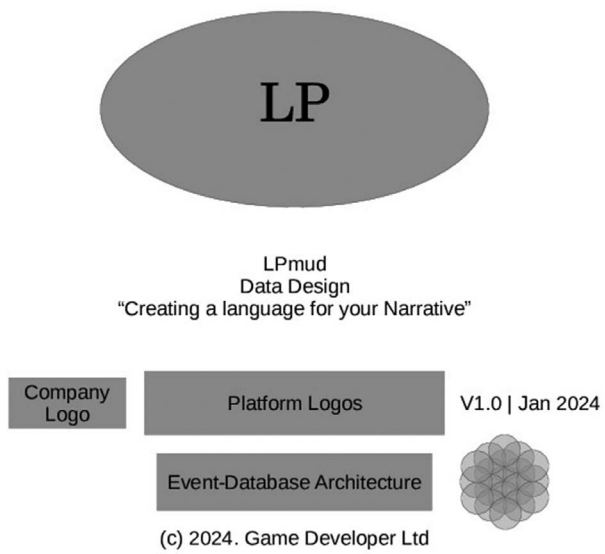

FIGURE 1.31 An example of a cover page for a *data design* to build a computer game *LPmud*.

after consultation with the staff i.e. the *Game Programmers*, *Game Artists*, *Game Designers*, *Sound Designers* and *Game Testers.*

This *data design* would include a description of all the *Database Tables*, *Database Records* and *Database Fields* that were going to be in the **Game Database** of *LPmud*. Each *Record* would require a special *Field* to act as an *ID*. This is known as the *Primary Key* and should be the first *Field* in each *Record*.

Some of the *Records*, in the *Database*, would be required by the components of the **Event-Database Architecture**. These would namely be the *Records* for the **Host Modules**, such as the **Physics Host**, **Graphics Host** and **Sounds Host**. The rest of the *Records* would be required by the *game design*, *technical design* and the staff. These include the *Records* for **Events** and **Game Objects**, images, texts, 3D models, sounds and so on.

For *LPmud*, over 40 *Database Tables* would be required. The first two *Tables* would hold the properties of **Events**.

1.4.1 Primary Events Table

The first *Database Table* would hold **Primary Event Records**. These would map a *Primary Key* for a **Primary Event** to

- one or more **Secondary Events**
- a hexadecimal code that will act as a short form of that *Primary Key* in certain situations, where you needed to save space.

This would include the archetypal **Primary Events** of the **Event-Database Architecture** e.g.

Collision Event
End Event
Proximity Event
Initial Reset Event
Moved Event
Shutdown Event

This would include the custom **Primary Events** for *LPmud* e.g.

Heartbeat Event
Loaded Event
Periodic Reset Event
Unloaded Event

This would include the *Database Records* for these standard Events required by the Game Controllers Host e.g.

Connect Event Record
Disconnect Event Record
Initial Reset Event Record
Moved Event Record

TABLE 1.5
Example of a Primary Events Table

Primary Event ID	Secondary Event IDs	Hex. Code.
Primary Initial Reset Event	Master Object Initial Reset Event, Master Object Periodic Reset	0002
Primary Heartbeat Event	Master Object Heartbeat Event, Warrior Heartbeat Event, Thief Heartbeat Event	0005
Primary End Event	Warrior Pacified Event, Thief Dead Event	00F0

Pressed Event Record
Priority End Events Record
Primary Collision Event Record
Primary Proximity Event Record
Priority Events Record
Released Event Record
Stopped Event Record

You can see an example of this *Database Table* in Table 1.5.

1.4.2 Secondary Events Table

The next *Database Table* would hold **Secondary Event Records**. Each *Record* would hold the properties of a **Secondary Event**. This would map a *Primary Key* for a **Secondary Event** to

- the time delay after a **Primary Event** before that **Event** would be sent
- the *game time* it was due to be sent
- the **Game Object** that would receive it and respond with an **Action**
- the list of **Game Objects** which caused it
- the **Priority Events Record**, which described any priority that **Event** had over other similar, or contradictory **Events**, when two or more **Secondary Events** followed on from the same **Primary Event**
- a hexadecimal code that would act as a short form of that *Primary Key* in certain situations, where you needed to save space.

This *Table* would include the archetypal **Secondary Events** of the **Event-Database Architecture** e.g.

Secondary End Event.

This *Table* would include the custom **Secondary Events** for *LPmud* e.g.

Object Attacked Event
Object Dead Event

Object Destroyed Event
Object Dropped Event
Object Entered Event
Object Exited Event
Object Heard Event
Object Heartbeat Event
Object Initial Reset Event
Object Inventory Event
Object Looked Event
Object Moved Event
Object Pacified Event
Object Periodic Reset Event
Object Taken Event
Object Unused Event
Object Used Event

This *Table* would also include all of the **Secondary Events** used by the **Procedurally Generated Quest System** i.e.

Quest Complete Event
Quest Splines Complete Event
Quest Reward Event

This *Table* and the previous *Table* would be used by the custom tools inside and outside the *Game World* to test the game i.e.

Internal Database Host Query Custom Tool
Internal Events Host Custom Tool.
External Database Host Query Custom Tool
External Events Host Custom Tool

From inside the *Game World*, the **Internal Database Host Query Custom Tool** would be used to query the location the player was in the *Game World*, for all visible and invisible **Game Objects**. And the local **Game Objects** found there in turn would be used to query this *Database Table.* To find the **Secondary Events** those **Objects** responded to. And the **Secondary Events** in turn would be used to query the previous *Database Table.* To find the **Primary Events** which generated those **Secondary Events**.

After the local **Primary Events** and **Secondary Events** had been identified, the **Internal Events Host Custom Tool** would then be used to send those **Primary Events** and **Secondary Events** to **Game Objects** via the **Events Host and Object Host**. To test what would happen when those **Events** were being triggered by a player.

From outside the *Game World*, the **External Database Host Query Custom Tool** would do a similar thing as the **Internal Database Host Query Custom Tool**. Except that it would query the entire *Database*, not just one location in the *Game World*, for any **Game Objects** whose *Database Fields* matched some criteria you

TABLE 1.6
Example of a Secondary Events Table

Secondary Event ID	Delay (Sec.)	Game Time (Sec.)	Game Object	Causing Objects
Master Object Initial Reset Event	0	1	Master Object	None
Master Object Periodic Reset Event	0	200	Master Object	None
Master Object Heartbeat Event	0	258	Master Object	None
Thief Dead Event	0	260	Thief Object	Warrior 2D Player Object
Thief Resurrect Event	5	265	Thief Object	Thief 2D Player Object

Secondary Event ID	Priority Events ID	Hex. Code
Master Object Initial Reset Event	None	0011
Master Object Periodic Reset Event	None	0018
Master Object Heartbeat Event	None	0021
Thief Dead Event	Thief's Death Priority Events	0022
Thief Resurrect Event	None	0023

specified. This can either be a *Field* containing a specific value or one of a set of values or a range of values. And it would then use the **Game Objects** found matching the criteria to query this *Table*. To find the **Secondary Events** those **Objects** responded to. And in turn use those **Secondary Events** to query the previous *Table* to find out which **Primary Events** generated those **Secondary Events**.

After the **Primary Events** and **Secondary Events** had been identified, the **External Events Host Custom Tool** would then be used outside of the *Game World*. To send those **Primary Events** and **Secondary Events** to the **Game Objects** via the **Events Host and Objects Host**. To test what would happen when those **Events** were being triggered by a player.

You can see an example of this *Database Table* in Table 1.6.

1.4.3 Sound Speaker Secondary Events Table

The next *Database Table* would hold **Sound Speaker Secondary Events Records**. These are very similar to **Secondary Events Records**. Except these would also map a *Primary Key* for a **Secondary Event** to

- **Sound Stream Records** containing the *sound streams* that the **Master Sound Speaker Object** should play when it received each **Event**.

You can see an example of this *Database Table* in Table 1.7.

TABLE 1.7
Example of a Sound Speaker Secondary Events Table

Secondary Event ID	Delay (Sec.)	Game Time (Sec.)	Game Object	Causing Objects
Thief Dead Sound Event	0	260	Master Sound Speaker Object	Warrior 2D Player Object
Thief Resurrect Sound Event	5	265	Master Sound Speaker Object	Thief 2D Player Object

Secondary Event ID	Priority Events ID	Hex. Code	Sound Stream ID
Thief Dead Sound Event	Thief's Death Priority Events	0024	Thief Dead Sound
Thief Resurrect Sound Event	None	0025	Thief Resurrect Sound

1.4.4 Priority Events Table

The next *Database Table* would hold **Priority Events List Records**. These would hold all the set of mutually exclusive **Secondary Events**, which could follow on from the same **Primary Event**. And this would hold the priority assigned to each one, to determine the frequency with which it would be chosen to follow that **Primary Event**, at the expense of the others. This priority is a percentage value. So a **Secondary Event** with a priority of 50 has a 50% chance of following on. And an **Event** with a priority of 30 has a 30% chance of following on. And one with a priority of 20 has a 20% chance, and so on and so on.

You can see an example of this *Database Table* in Table 1.8.

1.4.5 Events History Table

The next *Database Table* would hold **Events History Records**. These would hold the history of the chain of all the **Primary** and **Secondary Events** since the start of the game. It would include the maximum number of different types of **Events** and the maximum length of the history. Now the history of the chain of **Events** could literally be a list of *Primary Keys* for those **Events**. But the length of these *Primary Keys* can be very long. For example, one of the standard **Primary Events** is

Primary Initial Reset Event.

TABLE 1.8
Example of the Priority Events Table

List ID	Secondary Event IDs	Priorities (%)
Thief's Death Priority Events	Drop Money Event, Drop Jewellery Event, Drop Special Artefact Event	50, 30, 20

This is 27 characters long. The *Primary Keys* of **Events** in this form would quickly consume space reserved for the history.

A better way would be to store the history in the form of decimal or hexadecimal numbers which map onto the real *Primary Keys*. A hexadecimal number of 4 characters can represent a number from 0 to 65535 or 65536 *Primary Keys*. You would then need another *Database Table* which maps these hexadecimal *Primary Keys* to the real *Primary Keys* of **Events**. Fortunately, the **Primary Events Table** and the **Secondary Events Table** already do that.

There are many factors which effect the size of the **Events History Record**. The lesser the number of different types of **Events** you can have, the greater the number of **Events** you can store in your history. And the greater the length of time you can cover in that history. Before time runs out and the game has to end or the *Game World* has to be reset. And you lose the advantages of having an **Event History Record**. To recap, these would include

1. allowing players or computers who join multiplayer games late to replay all of the **Events**, since the beginning of the game, to synchronise their local copy of the *Game World* with the rest of the copies on the network
2. allowing **Game Objects** to perform **Actions** in response to **Events** that change depending on antecedent **Events**
3. allowing *Game Testers* to diagnose steps needed to reproduce *Bugs*, critical errors or Crashes
4. allowing *Game Programmers* to diagnose the code executed to reproduce *Bugs*, critical errors or Crashes

For example, if your game could have up to 256 **Events**, then you could fit the ID of each **Event** into a space of 1 byte or two hexadecimal characters. And if you reserved 100 megabytes of space for the history, then you could in theory keep a history of

$$(1024 * 1024 * 100) \,/\, 2 = 52,428,800$$

or 50 million **Events**. And if you assumed that two **Events,** a **Primary** and **Secondary Event**, would take place during each *Unit of game time*,[29] and each *Unit* last 1 second, then you could keep a history for

$$52,428,800 \,/\, (2 * 60 * 60) = 7281.778$$

or 7282 hours. Before the game has to end or the *Game World* has to be reset.

If, on the other hand, your game could have up to 65,536 **Events**, then you could fit the ID of each **Event** into a space of 2 bytes or four hexadecimal characters. And if you again reserved the same amount of space for the history, then you could in theory keep a history of

$$(1024 * 1024 * 100) \,/\, 4 = 26,214,400$$

or 25 million **Events**. And if you again assumed that two **Events** would take place during each *Unit of game time*, and each last 1 second, then you could keep a history of

$$26,214,400 \,/\, (2 * 60 * 60) = 3640.889$$

or 3641 hours. Before the game has to end or the *Game World* has to be reset.

Now some may say it is highly unlikely you would make a game where only two **Events** would occur in each *Unit of game time*. A more realistic amount would be 100 **Events**. But even if you allow for that, then you could still keep a history for

$$26,214,400 \,/\, (100 * 60 * 60) = 72.81778$$

or 72 hours.

Now some may say that the history of **Events** should include temporal or spacial information. So you could tell the position of **Game Objects** in the *Game World* in the history, when they responded to **Secondary Events**. For example, when you join a multiplayer game, built with the **Event-Database Architecture**, and a **Peer-to-Peer Network Architecture**, and replay all the **Events** which happened from the beginning, at the end of the replay, you would need to know the current state and position of all the **Game Objects**. To complete the synchronisation of your local copy of the *Game World* with the other copies on the local network.

But adding temporal or spacial information to the history would greatly increase its size. If, for example, you wanted to store the X, Y and Z positions of every **Game Object** when it responded to a **Secondary Event** in the history, you would need 4 bytes to store each of these three positions. That is a total of 12 bytes on top of the 2 bytes needed to store the *ID* or *Primary Key* of each **Event**. This gives you a total of 14 bytes for each **Secondary Event**. Suppose you have 100 **Events** occurring in each *Unit of game time*. And 50 were **Secondary Events**, taking up 14 bytes. And 50 were **Primary Events**, taking up 2 bytes. This means that in each *Unit*, 800 bytes or 1600 hexadecimal characters were required to represent 100 **Primary** and **Secondary Events**. And suppose you have 100 megabytes of space reserved for the history. That means you can only keep

$$(1024 * 1024 * 100) \,/\, (1600 \,/\, 100) = 6553,600$$

or 6.5 million **Events** or

$$6553,600 \,/\, (100 * 60 * 60) = 18.204$$

or 18 hours of history.

On the other hand, some may say you do not need spacial information in the history. If a player or computer were to join a multiplayer game late, then they just have to replay the history of **Events** to date. All the while keeping track of all of the **Game Objects** affected by these **Events**. And then, at the end of the replay, get the spacial information i.e. the final position of all these **Game Objects**. And this can

TABLE 1.9
Example of the Events History Table

List ID	Max. Types of Events	Max. Length	History of Events
Events History	65,536	52,428,800	00020011001800050021

come from the copies of the **Game Database** on the local computer network of those already playing the game.

Furthermore, it is highly unlikely you would have 65,536 different types of **Events** occurring in a game. Since you would have to give a name to each one. And these names would become part of the language of the production process. A language which had 65,536 words would be too much for anyone to learn. You would really need to strike a balance between a high enough number of **Events** that gave you the flexibility to cope with changes to an incomplete *game design*. And a number low enough that it makes the language of the production process relatively quick and easy to learn for new staff joining the process.

You can see an example of the **Events History Table** in Table 1.9.

1.4.6 2D Polygons Table

The next *Database Table* would hold **2D Graphics Object Records**. These would hold the shape of a *polygon*, which would be used to display the image of a **2D Graphic Object**. The shape may also either be used to define a bounding box around an **Object**. Or it may be used to define the *Collision boundary* around an **Object**. Or it may be used to set the *Proximity boundary* around the **Object**. The *Record* would map a **2D Polygon ID** to the list of the coordinates of the *vertices*, and the *Normal Vectors*, of the *polygon*.

You can see an example of this *Database Table* in Table 1.10.

1.4.7 3D Models Table

The next *Database Table* would hold **3D Graphics Object Records**. These would hold the shape of a 3D model, which would be used to display a **Graphics Object**. The model may also either be used to define the bounding box around an Object. Or it may be used to define the shape of a *Collision boundary* around an **Object**. Or it may be used to set the *Proximity boundar*y around an **Object**. The *Record* would map a **3D Model ID** to the list of the coordinates of the *vertices,* and the *Normal Vectors*, of each *polygon* in a model.

You can see an example of this *Database Table* in Table 1.11.

1.4.8 Textures Table

The next *Database Table* would hold **Texture Graphics Object Records**. Each *Record* would hold the *Texture* (or 2D image), which would be used to colour a 2D

TABLE 1.10
Example of a 2D Polygons Table

2D Polygon ID	2D Vertices	2D Normals
Help Icon Boundary	List of the vertices, which make up the boundary around the Help icon on the screen.	List of the *Normal Vectors*; one for each side of the *polygon*.
Forest Outline Boundary	List of the *vertices*, which make up the outline of the forest on the map of the *Game World*, located about its centre.	List of the *Normal Vectors*; one for each side of the *polygon*.
Village Outline Boundary	List of the *vertices*, which make up the outline of the village on the map of the game, located about its centre.	List of the *Normal Vectors*; one for each side of the *polygon*.
Mountain Range Outline Boundary	List of the *vertices*, which make up the outline of the mountain range on the map of the *Game World*, located about its centre.	List of the *Normal Vectors*; one for each side of the *polygon*.

TABLE 1.11
Example of a 3D Models Table

3D Model ID	3D Vertices	3D Normals
Warrior Model	List of the triangular *vertices*, which make up the model of a Warrior, located about its centre.	List of the *Normal Vectors*; one for each *polygon*.
Thief Model	List of the triangular *vertices*, which make up the model of a Thief, located about its centre.	List of the *Normal Vectors*; one for each *polygon*.
Forcefield Boundary	List of the triangular *vertices*, which make up the boundary around the model of a Forcefield, located about its centre.	List of the *Normal Vectors*; one for each *polygon*.
Small Bush Boundary	List of the triangular *vertices*, which make up the boundary around the model of a small bush, located about its centre.	List of the *Normal Vectors*; one for each *polygon*.
Covered Pit Boundary	List of the triangular *vertices*, which make up the boundary around the model of a covered pit, located about its centre.	List of the *Normal Vectors*; one for each *polygon*.

TABLE 1.12
Example of a Textures Table

Texture ID	Width	Height	Pixels (RGBA Format)
Warrior Texture	256	256	List of colours that make up the image.
Thief Texture	256	256	List of colours.
Game Map Texture	1024	768	List of colours.
Italic Font Texture	512	128	List of colours.
Portal Texture	512	512	List of colours.

polygon or a 3D model. Some of the images would also be used to display **2D Game Objects** and others to display **3D Game Objects**. The *Record* would map a **Texture ID** to the width and height of an image, and a set of pixel colours in *RGBA format*.[30]

You can see an example of this *Database Table* in Table 1.12.

1.4.9 Texture Coordinates or UV Table

The next *Database Table* would hold **Texture Coordinates Graphics Object Records**. Each *Record* would hold *Texture coordinates*. These would control the region of a *Texture* that would be used to display a *polygon* of a 3D model, or a 2D image. The *Record* would also be used by a **Text Object** to display words. It would be used to get the region of each character, from an image which contained all the characters of a font. The *Record* would map a **Texture Coordinate ID** to a list of the coordinates of the *vertices*, on a 2D image.

You can see an example of this *Database Table* in Table 1.13.

TABLE 1.13
Example of a UV Table

Texture Coordinate ID	2D Vertices
Warrior Texture Coordinates	List of the triangular *vertices*, on a 2D image; one vertex per 3D vertex in the model of the Warrior.
Thief Texture Coordinates	List of the triangular *vertices*, on a 2D image; one vertex per 3D vertex in the model of the Thief.
Game Map Texture Coordinates	List of the rectangular *vertices*, on a 2D image; one vertex per 2D vertex in the *polygon* of the map of the *Game World*.
Help Icon Texture Coordinates	List of the rectangular *vertices*, on a 2D image. These mark the 4 points around the Help icon, in the image of all the icons used in the game.
Italic-A Texture Coordinates	List of the rectangular *vertices* on a 2D image. These mark the 4 points around the character 'A', in the image of the Italic font.
Portal Texture Coordinates	List of the triangular *vertices*, on a 2D image of an elliptical portal showing the view of a remote part of the *Game World*.

1.4.10 Materials Table

The next *Database Table* would hold **Materials Graphics Object Records**. These would hold the properties of *Materials*. These would control how one or more *Textures* are rendered on a *polygon* of a 3D model, or a 2D image. A *polygon* may be rendered with a single *Texture*. Or it may be rendered using a composite of *Textures*. These *Textures* may be tiled together, side by side, using some formula. Or these may be blended on top of each other, using another formula. Or the pixels of the *Textures* may be used to generate vertices or adjust the position of vertices or generate Normal Vectors on a *polygon* or model, using yet another formula. Or these Normal Vectors may in turn change the way light is reflected off the surface of a *polygon*.

The rendering of *Textures* will be done by *Graphic Shaders*.[31] This is machine code used to programme a Graphics Processor that renders shapes on the screen or into another *Texture*. There are several *Shaders* involved in executing the five main steps of the process of *Hardware Rendering* with a Graphics Processor i.e.

1. *Vertex Shader*[32]
2. Tessellation
3. *Geometry Shader*[33]
4. Rasterisation
5. *Fragment Shader.*[34]

The *Vertex Graphic Shader* or *Vertex Shader* is used to perform the projection of the vertices of the *polygons* of 2D images or 3D *models*, through a camera, into Normalised space (an area which is $1 \times 1 \times 1$) and then onto screen space (i.e. the computer screen). And it is used to set the amount of lighting at each vertex.

The Tessellation step is mandatory. It breaks up the *polygons* of 2D images or 3D *models* produced by the *Vertex Shader* into smaller *polygons*. To make them look like higher resolution *polygons* or models and thus give them a smoother appearance. Typically, this step is hard coded and cannot be controlled with *Graphic Shaders*.

The *Geometry Graphic Shader* or *Geometry Shader* is optional. It is used either to take the 2D or 3D primitives from the *Vertex Shader* and produce another primitive, adding or removing vertices. Or for rendering multiple images of the same primitive, at once, to the same target (i.e. computer screen or *Texture*). Or for feeding back information about the vertices of the primitives produced by the *Vertex Shader*, to later steps.

The Rasterisation step is mandatory. It projects the pixels of the *Textures* of the *polygons* of the 2D image or 3D *models*, onto the screen or another *Texture*. Typically, this step is hard coded and cannot be controlled with *Graphic Shaders*.

The *Fragment Graphic Shader* or *Fragment Shader* is optional. It parses the pixels of the *Textures* of the *polygons* of 2D images or 3D *models*, after Rasterisation. And it can change the depth and colour of the pixels depending on some kind of formula. And it can also discard pixels and stop these being rendered dependent on another formula.

TABLE 1.14
Example of a Materials Table

Material ID	List of Texture IDs
Warrior Material	List of *Textures* that will be used to render the Warrior's face, hair, arms, hand, legs, feet and dress.
Thief Material	List of *Textures* that will be used to render the Thief's face, hair, arms, hand, legs, feet and dress.
Game Map Material	List of *Textures* that will be tiled next to each other to show a map of the *Game World*.
Help Icon Material	List of animated *Frames* that will be used to play back an animation of a rotating Help Icon.
Italic-A Material	List of *Textures* that will render the character 'A', in the image of the Italic font.

Material ID	List of Tags	Vertex Shader	Geometry Shader	Fragment Shader
Warrior Material	NPC	*Vertex Shader* to render Warrior.	*Geometry Shader* to render Warrior.	*Fragment Shader* to render Warrior.
Thief Material	NPC	*Vertex Shader* to render Thief.	*Geometry Shader* to render Thief.	*Fragment Shader* to render Thief.
Game Map Material	Map	*Vertex Shader* to render map.	*Geometry Shader* to render map.	*Fragment Shader* to render map.
Help Icon Material	Icon	*Vertex Shader* to render an animated icon.	*Geometry Shader* to render an animated icon.	*Fragment Shader* to render an animated icon
Italic-A Material	Font	*Vertex Shader* to render a character in a font.	*Geometry Shader* to render a character in a font.	*Fragment Shader* to render a character in a font.

A *Material* can have one or more Tags or words which identify its properties on the surface of a *polygon* not just for rendering graphics but also for simulating physics. For example, a Tag may indicate that a surface is liquid. This means that in terms of rendering its graphics, the graphics should show light reflecting off the surface, and refraction of light passing through the surface. And in terms of physics, if the players walk across the surface, then their character will sink down into it.

You can see an example of a **Materials Table** in Table 1.14.

1.4.11 Projected Shapes Table

The next *Database Table* would hold **Projected Shapes Records**. These would hold the properties of **Projected Shapes**. These would be the *vertices* of the bounding box around **2D Graphic Objects** or **3D Graphics Objects**, on a *Texture* or computer screen, after these had been projected through the **2D Camera Objects** or **3D Camera Objects**, in the *Game World*. The *Record* would map a **Projection ID** to a list of the projected *vertices* on a *Texture* or the screen.

The *Record* would also include the **Graphic Object ID** of the **Object** whose bounding box was projected onto the screen. And the *Record* would include a **Device Group ID** that would contain the device group name, IP Address, *Username*, *Password* and *Authentication Token* of the **Game Client** or the player that would own the *Database Record* and would see those projections, in a multiplayer game (Table 1.15).

TABLE 1.15
Example of a Projected Shapes Table

Projection ID	Projected Vertices	Graphic Object ID
Warrior Projection	List of the projected *vertices* of the bounding box of the Warrior.	Warrior 3D Player Object
Thief Projection	List of the projected *vertices* of the bounding box of the Thief.	Thief 3D Player Object
Game Map Projection	List of the projected *vertices* of the bounding box of the map of the *Game World*.	Game Map Object
Help Icon Projection	List of the projected *vertices* of the bounding box of the icon.	Help Icon Object
Forest Label Projection	List of the projected *vertices* for the bounding box of the characters labelling the forest, on the map of the game.	Forest Label Object
Village Label Projection	List of the projected *vertices* for the bounding box of the characters labelling the village, on the map of the game.	Village Label Object

Projection ID	Materials	Device Group ID
Warrior Projection	Warrior Materials ID	Joystick1:192.168.0.1:Player1:PassP112 34:34&QA>65R,3I087-S4#$Q,C,T"@``
Thief Projection	Thief Materials ID	Joystick1:192.168.0.1:Player1:PassP112 34:34&QA>65R,3I087-S4#$Q,C,T"@``
Game Map Projection	Game Map Materials ID	Joystick1:192.168.0.1:Player1:PassP112 34:34&QA>65R,3I087-S4#$Q,C,T"@``
Help Icon Projection	Icon Materials ID	Joystick1:192.168.0.1:Player1:PassP112 34:34&QA>65R,3I087-S4#$Q,C,T"@``
Forest Label Projection	Text Materials ID	Joystick1:192.168.0.1:Player1:PassP112 34:34&QA>65R,3I087-S4#$Q,C,T"@``
Village Label Projection	Text Materials ID	Joystick1:192.168.0.1:Player1:PassP112 34:34&QA>65R,3I087-S4#$Q,C,T"@``

1.4.12 Sound Microphone Table

The next *Database Table* would hold **Sound Microphone Records**. These would hold a sound microphone from which the *Game World* could be heard. Each *Record* would map a **Sound Microphone ID** to a **Game Object** the microphone was attached to and an offset around that **Game Object** from which sounds would be heard. And a **Device Group ID** that would contain the device group name, IP Address, *Username*, *Password* and *Authentication Token* of the **Game Client** or the player that would own the *Database Record* and would hear the sound through the sound microphone.

You can see an example of this *Database Table* in Table 1.16.

1.4.13 Sound Stream Table

The next *Database Table* would hold **Sound Stream Records**. These would hold a sound, or a piece of music, that would be played during the game. Each *Record* would map a **Sound Stream ID** to an encoded *sound stream*, its duration, its frequency, its *channel*, its left and right stereo volumes, its priority, its **End Event**, the **Game Object** whose locality the sound was attached to, the radius around that locality where the sound would be heard.

You can see an example of a **Sound Stream Table** in Table 1.17.

1.4.14 Animated Vertices Table

The next *Database Table* would hold A**nimated Vertices Graphics Object Records**. These would be used to hold the properties of **ANIMATED VERTICES**. Each would map an **ANIMATION ID** to a list of the coordinates of the *vertices* that changed, between each *Frame*, of an animated 2D *polygon* or 3D model. It would

TABLE 1.16
Example of a Sound Microphone Table

Sound Microphone ID	Object ID	Offset X	Offset Y	Offset Z
Warrior Object Microphone	Warrior Player Object	0	0	0
Thief Object Microphone	Thief Player Object	10	20	−10
Mage Object Microphone	Mage Player Object	−10	30	−10
Cleric Object Microphone	Cleric Player Object	0	0	10

Sound Microphone ID	Device Group ID
Warrior Object Microphone	Joystick1:192.168.0.1:Player1:PassP11234:34&QA>65R, 3I087-S4#$Q,C,T"@``
Thief Object Microphone	Keyboard1:192.168.0.1:Player2:PassP21234:34&QA>65R, CI087-S4#(Q,C,T"@``
Mage Object Microphone	Joystick2:192.168.0.2:Player3:PassP35678:34&QA>65R, SI087-S4#,U-C<X"@``
Cleric Object Microphone	Keyboard2:192.168.0.2:Player4:PassP45678:34&QA> 65R-#I087-S4#0U-C<X"@``

TABLE 1.17
Example of a Sound Stream Table

Sound Stream ID	Encoded Sound (PCM format)	Duration (Sec.)	Frequency (kHz)
Funeral March Music	Sound stream of the music.	180	40
Help Icon Sound	Sound stream of a voice asking, 'How may I help you?'	3	40

Sound ID	Channel	Left Vol.	Right Vol.	Priority
Funeral March Music	1	7	7	2
Help Icon Sound	2	8	8	1

Sound Stream ID	End Event ID	Object ID	Sound Radius
Funeral March Music	End Funeral March Event	None	0
Help Icon Sound	End Icon Sound Event	Warrior's Cursor Object	4196

include the size of each set of changes. It would also include the rate at which the *Frames* would be displayed, the length of the animation in seconds and how much time had elapsed since the animation began.

To animate a model, the list of the changed *vertices* would be divided up into several sections, depending on the number of *Frames* in the animation. Each section would hold the *vertices* that have changed, between each *Frame*. The size of each section would depend on the number of *vertices* that changed between each *Frame*.

Beginning with the 2D *polygon* or 3D model in its initial pose, each set of changes would then be applied to the current *Frame*, to generate the next *Frame*, in the animation sequence. Each sequence would be created so that it formed one complete cycle. Consequently, the first *Frame* and the last *Frame* of the animation would be exactly the same. And the 2D *polygon* or 3D model could be easily returned to its initial pose, if the animation were stopped in the middle of the sequence. This could be done by simply completing the sequence.

You can see an example of this *Database Table* in Table 1.18.

1.4.15 Game Time Table

The next *Database Table* would hold **Game Time Records**. Each *Record* would hold the *Unit of game time* with which the game would operate. This would also hold how much time had elapsed since the start of the game, and when the game began. The *Record* would map a **GAME TIME ID** to these three times. It could also be used to keep track of other timed Events as well.

For example, it is used to hold the time at which the players entered and left different stages or levels or parts of the *Game World*. It could be used to time when a character appeared in the *Game World* and disappeared. It could be used to hold the time at which the player started at an arbitrary point A and ended at another point B.

You can see an example of this *Database Table* in Table 1.19.

TABLE 1.18
Example of an Animated Vertices Table

Animation ID	Animated Vertices	Frame Sizes
Warrior's Death Animation	List of the changed *vertices*, and the position or index of each vertex, in the *vertices* of the animated skeleton of a 3D model.	List of the number of *vertices* changed between each *Frame*, for *Frames*–300.
Thief's Death Animation	List of the changed *vertices*, and the position or index of each vertex, in the *vertices* of the animated skeleton of a 3D model.	List of the number of *vertices* changed between each *Frame*, for *Frames* 1–300.
2D Icon Crossbow Animation	List of the changed *vertices*, and the position or index of each vertex in the *vertices* of an animation of a *polygon* of a crossbow being fired, shown on a menu in a shop that sells crossbows.	List of the number of *vertices* changed between each *Frame*, for *Frames* 1–120.

Animation ID	Animation Rate (Frames per sec.)	Animation Length (Sec.)	Animation Elapsed (Sec.)
Warrior's Death Animation	60	5	0
Thief's Death Animation	60	5	2.3
2D Icon Crossbow Animation	60	2	0

1.4.16 Delayed Events Table

The next *Database Table* would hold **Delayed Events List Records**. Each *Record* would list all the **Secondary Events** that were waiting to be sent, to a **Game Object**, after the occurrence of a **Primary Event**. It would be used by the **Events Host**, to

TABLE 1.19
Example of a Game Time Table

Game Time ID	Unit Time (Secs.)	Total Time (Secs.)	Start Time (Secs.)	End Time (Secs.)
Overall Time	0.02	600.02	0	600.02
Mountain domain	0.02	240.01	240.1	480.1
Sun alley domain	0.02	60.1	480.1	540.1
Forest domain	0.02	60.1	540.1	600.02
Giant lair domain	0.02	0	–	–
Seashore domain	0.02	0	–	–
Mines domain	0.02	0	–	–
Maze domain	0.02	0	–	–
Village domain	0.02	120	0	120.1
Adventurer's guild domain	0.02	120	120.1	240.1

TABLE 1.20
Example of a Delayed Events Table

List ID	Secondary Event IDs
Delayed Events List	Master Periodic Reset Event, Master Heartbeat Event, Thief Resurrect Event.

keep track of **Secondary Events** with time delays. The *Record* would map a **List ID** to a list of **Secondary Events**.

You can see an example of this *Database Table* in Table 1.20.

1.4.17 Residents or Loaded Records Table

The next *Database Table* would hold **Residents List Records**. Each *Record* would hold the properties of the **Residents List**. This would list the *Records* of the **Game Database** that were currently residing in the computer memory. The resident *Records* would be ordered in the list. So that the latest addition, to the memory, would be at the beginning of the list. And the oldest addition would be at the end of the list.

The **Master Object** would use this list to tell when a new **Game Object** and its **Game Object Record** had been loaded into memory. And therefore the **Game Object** should receive the **Object Loaded Event**.

The *Record* would map a **List ID** to a list of the *Primary Keys* of the resident *Records*, and the maximum length of this list. Once this maximum was reached, some of the *Records* residing in the computer memory would be unloaded using some suitable steps or algorithm.

You can see an example of this *Database Table* in Table 1.21.

1.4.18 Absents or Unloaded Records Table

The next *Database Table* would hold **Absents List Records**. Each *Record* would hold the properties of the **Absents List**. This list would indicate the *Records* of the **Game Database** that had been temporarily removed from the computer memory, to make space. That may be loaded back into computer memory, later on, from the storage media the **Game Database** was stored on, to a temporary location in computer memory. When something tries to access that *Database Record* again.

TABLE 1.21
Example of a Loaded Records Table

List ID	Resident IDs	Max Length
Residents List	List of all the *Records* currently residing in the computer memory.	32,768

TABLE 1.22
Example of an Unloaded Records Table

List ID	Absent IDs	Max Length
Absents List	List of all the *Records* currently absent from the computer memory.	65,536

The absent *Records* would be ordered in the list. So that the latest *Record*, removed from the memory, would be at the beginning of the list. And the oldest *Record*, removed from the memory, would be at the end of it.

The **Master Object** would use this list to tell when an old **Game Object** had been destroyed and its **Game Object Record** had been removed from memory. And therefore that **Game Object** should receive the **Object Unloaded Event**.

The *Record*, of the **Absents List**, would map a **List ID** to a list of the *Primary Keys* of the absent *Records*. You can see an example of this *Database Table* in Table 1.22.

1.4.19 Objects Loaded Table

The next *Database Table* would hold **Objects List Records**. Each *Record* would list all the **Game Objects** that would be loaded into the computer memory, by the **Objects Host**. The order of the **Game Objects**, in the list, would reflect the order in which these would be loaded into the memory. The first **Object** loaded would be at the beginning of the list, and the last **Object** would be at the end of it. The *Record* would map a **List ID** to a list of **Game Objects**.

This *Table* would also hold the **STAGE OBJECTS LIST RECORDS**. These would list all the **Game Objects** and other *Database Records* which should be loaded in each part or level in the *Game World*. If a decision were made to limit, the number of **Game Objects** loaded in each part to conserve space in computer memory.

You can see an example of this *Database Table* in Table 1.23.

TABLE 1.23
Example of Objects Loaded Table

List ID	Object IDs
Objects List	List of all the *Records* holding the properties of **Game Objects** that currently, or were going to be, residing in the computer memory
Stage Objects List 1	List of all the *Records* of **Game Objects** that should be loaded in part 1 of the *Game World*
Stage Objects List 2	List of all the *Records* of **Game Objects** that should be loaded in part 2 of the *Game World*
Stage Objects List 3	List of all the *Records* of **Game Objects** that should be loaded in part 3 of the *Game World*

TABLE 1.24
Example of a 2D Graphics List Table

List ID	Object IDs
2D Graphics List	Warrior's Health Bar Object, Thief's Health Bar Object, Warrior's Label Text Object, Thief's Label Text Object, Warrior's Messages Text Object, Thief's Messages Text Object.

1.4.20 2D Graphics Lists Table

The next *Database Table* would hold **Graphics List Records**. But each *Record* would hold only list the 2D **Game Objects**, in a *Game World*, that would be displayed on the computer screen. These would include items lying around, characters, creatures, buildings, other structures or locations, as well as texts, images, icons etc.

You can see an example of the **2D Graphics List Table** in Table 1.24.

1.4.21 3D Graphics Lists Table

The next *Database Table* would hold **Graphics List Records**. But each *Record* would only hold a list of the 3D **Game Objects** that would be displayed in the *Game World*. The *Record* would map a **List ID** to a list of 3D **Objects**.

You can see an example of this *Database Table* in Table 1.25.

1.4.22 Projected Lists Table

The next *Database Table* would hold **Projected List Records**. *E*ach *Record* would map a **List ID** to a list of projections of **2D Graphic Objects**, through a **2D Camera Object**, in a 2D *Game World*, onto a *Texture* or the screen. It would also include the projections of **3D Graphic Objects**, through a **3D Camera Object**, in a 3D *Game World*. And the *Record* would include a **Device Group ID** that would contain the device group name, IP Address, *Username*, *Password* and *Authentication Token* of the **Game Client** or the player that would own the *Database Record* and would see those projections, in a multiplayer game.

You can see an example of this *Database Table* in Table 1.26.

TABLE 1.25
Example of 3D Graphics List Table

List ID	Object IDs
3D Graphics List 1	Warrior's Player Object, Thief's Player Object, Forest Sector Object, Forest Tree Object 1, Forest Tree Object 2, Forest Tree Object 3, Small Bush Object 1, Small Bush Object 2, Large Boulder Object, Sky Object.
3D Graphics List 2	Game Logo Object.

TABLE 1.26
Example of Projected Lists Table

List ID	Projection IDs
Projected List Warrior 3D Camera Object	Warrior Projection, Thief Projection, Forest Sector Projection, Forest Tree 1 Projection, Forest Tree 2 Projection, Forest Tree 3 Projection, Small Bush 1 Projection, Small Bush 2 Projection, Sky Projection, Warrior's Health Bar Projection, Thief's Health Bar Projection, Warrior's Label Projection, Thief's Label Projection, Warrior's Messages Projection, Thief's Messages Projection.
Projected List Warrior Orthographic Top View Camera Object	Warrior Projection, Thief Projection, Forest Sector Projection, Forest Tree 1 Projection, Forest Tree 2 Projection, Forest Tree 3 Projection, Small Bush 1 Projection, Small Bush 2 Projection, Warrior's Health Bar Projection, Thief's Health Bar Projection, Warrior's Label Projection, Thief's Label Projection, Warrior's Messages Projection, Thief's Messages Projection.
List ID	**Device Group ID**
Projected List Warrior 3D Camera Object	Keyboard1:192.168.0.1:Player2:PassP21234:34&QA >65R,CI087-S4#(Q,C,T"@``
Projected List Warrior Orthographic Top View Camera Object	Keyboard1:192.168.0.1:Player2:PassP21234:34&QA >65R,CI087-S4#(Q,C,T"@``

1.4.23 Sounds List Table

The next *Database Table* would hold **Sounds Waiting List Records** and the **Sounds Playing List Records**. *E*ach *Record* would list the sounds which were waiting to be played back or were being played. The *Record* would map a **List ID** to a list of music and sound effects.

You can see an example of this *Database Table* in Table 1.27.

1.4.24 2D Physics Lists Table

The next *Database Table* would hold **Physics List Records**. Each *Record* would only list the **2D Game Objects** whose position, speed and acceleration would be updated by the **Physics Host**. The *Record* would map a **List ID** to a list of **Invisible 2D Point**

TABLE 1.27
Example of a Sound Lists Table

List ID	Sound IDs
Sound Waiting List	Siegfried's Funeral March Music, Help Icon Sound
Sound Playing List	Explosion Sound, Player Dying Sound

TABLE 1.28
Example of a 2D Physics Lists Table

List ID	2D Point Object IDs
2D Physics List	Warrior's Health Bar Object, Thief's Health Bar Object.

Objects. Besides **2D Point Objects**, the list may include any other **Game Object** that was derived from an **Invisible 2D Point Object**. So long as it had all the properties that a **2D Point Object** had, that **Object** could be updated by the **Physics Host**.

The physics model of the Physics Host would be simple Newtonian Physics including the 3 laws. A Game Object at rest will remain at rest or in motion will remain in motion, unless acted on by an external force. The acceleration of a Game Object is directly proportional to the force acting upon it and inversely proportional to its mass. For every action there is an equal and opposite reaction. The **Game Objects** could generate the forces (i.e. acceleration) acting on other Objects at any time for any reason determined by the *game design*. The **Physics Host** would resolve the forces generated depending on which forces were part of its software model of the physical *Game World* (e.g. Friction, Gravity and reactive forces caused by collision).

You can see an example of this *Database Table* in Table 1.28.

1.4.25 3D Physics Lists Table

The next *Database Table* would hold **Physics List Records**. Each *Record* would list only the 3D **Game Objects** whose position, speed and acceleration would be updated by the **Physics Host**. The *Record* would map a **List ID** to a list of **Invisible 3D Point Objects**. As with the **2D Objects**, this list may include other **Game Objects** besides **Invisible 3D Point Objects**. So long as a **Game Object** had all the properties of an **Invisible 3D Point Object**, it could be updated by the **Physics Host**.

You can see an example of this *Database Table* in Table 1.29.

1.4.26 2D Camera Lists Table

The next *Database Table* would hold **Camera List Records**. Each *Record* would hold the properties of a **2D Camera List**. This would list the 2D cameras whose view of the *Game World* would be displayed, on the computer screen or a *Texture*.

TABLE 1.29
Example of a 3D Physics Lists Table

List ID	3D Point Object IDs
3D Physics List	Warrior's Player Object, Thief's Player Object, Forest Sector Object, Forest Tree Object 1, Forest Tree Object 2, Forest Tree Object 3, Small Bush Object 1, Small Bush Object 2, Large Boulder Object, Sky Object.

TABLE 1.30
Example of a 2D Camera Lists Table

List ID	2D Camera Object IDs
2D Cameras List	Warrior Orthographic Top View Camera Object, Thief Orthographic Side View Camera Object, Mage Orthographic Front View Camera Object, Cleric Isometric Camera Object, Druid Chromatic Camera Object, Ranger 2D Audio Camera Object, Necromancer Textual Camera Object.
List ID	**Device Group ID**
2D Cameras List	Keyboard1:192.168.0.1:Player2:PassP21234:34&QA>65R, CI087-S4#(Q,C,T"@``

This would include the view of any cameras showing an **Orthographic Projection, Isometric Projection**, **Chromatic Projection**, **Audio Projection** or **Textual Projection** of the *Game World*. The *Record* would map a **List ID** to a list of **2D Camera Objects**, and a **Device Group ID** that would contain the device group name, IP Address, *Username*, *Password* and *Authentication Token* of the **Game Client** or the player that would own the *Database Record*, and would see the *Game World* through those **Camera Objects**, in a multiplayer game.

You can see an example of this *Database Table* in Table 1.30.

1.4.27 3D Camera Lists Table

The next *Database Table* would hold **Camera List Records**. Each *Record* would hold the properties of a **3D Camera List**. This would list the cameras whose view, of the 3D *Game World*, would be projected onto the computer screen, or a *Texture*. This would include the view of any cameras showing an **Audio Projection** or **Perspective Projection** of the *Game World*. The *Record* would map a **List ID** to a list of 3**D Camera Objects**, and a **Device Group ID** that would contain the device group name, IP Address, *Username*, *Password* and *Authentication Token* of the **Game Client** or the player that would own the *Database Record* and would see the *Game World* through those **Camera Objects**, in a multiplayer game.

You can see an example of this *Database Table* in Table 1.31.

TABLE 1.31
Example of 3D Camera Lists Table

List ID	3D Camera Object IDs
3D Cameras List	Game Logo Camera Object, Rogue 3D Audio Camera Object, Paladin Perspective Camera Object.
List ID	**Device Group ID**
3D Cameras List	Keyboard1:192.168.0.1:Player2:PassP21234:34&QA >65R,CI087-S4#(Q,C,T"@``

TABLE 1.32
Example of a Device Group Table

Device Group ID	Controller Type	Object ID
Joystick1:192.168.0.1:Player1: PassP11234:34&QA>65R, 3I087-S4#$Q,C,T"@``	Standard Game Controller	Warrior Player Object
Keyboard1:192.168.0.1:Player2: PassP21234:34&QA>65R, CI087-S4#(Q,C,T"@``	Professional Keyboard	Thief Player Object
Joystick2:192.168.0.2:Player3: PassP35678:34&QA>65R, SI087-S4#,U-C<X"@``	Advanced Game Controller	Mage Player Object
Keyboard2:192.168.0.2:Player4: PassP45678:34&QA>65R-#I087-S4#0U-C<X"@``	Ergonomic Keyboard	Cleric Player Object

1.4.28 Device Group Table

The next *Database Table* would hold **Device Group Records**. Each *Record* would hold the properties of a *Game Controller*, or a group of *analogue devices* and *digital devices* on a *Game Controller*. It would map a **Device Group ID** to the type of *Game Controller* the *devices* belonged to, and a **Game Object**, whose properties (e.g. Position) would be updated when the *Game Controller* was manipulated.

You can see an example of this *Database Table* in Table 1.32.

1.4.29 Device Sequence Primary Events Table

The next *Database Table* would be used by the **Master Player Object**. It would hold **Device Sequence Primary Events Records**. Each *Record* would hold the single word for a single *analogue device* or *digital device*, or sequence of words for a group of *devices* on a *Game Controller*. The use of which, by the player, constituted an imperative command or **Action** to be performed by the player's character. It would map a **Command ID** to a sequence of *devices* for a command, and the **Primary Event** that should be sent when that sequence was detected in the **Digital History Field** or **Analogue History Field** of a **2D** or **3D Player Object**.

You can see an example of this *Database Table* in Table 1.33.

1.4.30 Text Localisations Table

The next *Database Table* would hold **Text Localisation Records**. Each *Record* would hold the text that would be displayed at different stages of the game. These would include the names of items, on various menus of the *User Interface*, the title of each stage, the names of **Game Objects**, the descriptions of features of the game and so on. It would map a **TEXT ID** to the words which would be displayed on

TABLE 1.33
Example of a Game Controller Primary Events Table

Command ID	Device Sequence	Primary Event IDs
Forwards Command	W	Forwards Command Event
Backwards Command	S	Backwards Command Event
Turn Left Command	A	Turn Left Command Event
Turn Right Command	D	Turn Right Command Event
Jump Up Command	SPACE	Jump Up Command Event
Jump Down Command	C	Jump Down Command Event
Look Command	L	Look Command Event
Get Command	T	Get Command Event
Drop Command	Q	Drop Command Event
Give Command	G	Give Command Event
Wield Command	U	Wield Command Event
Wear Command	J	Wear Command Event
Remove Command	E	Remove Command Event
Say Command	TY	Say Command Event
Tell Command	TT	Tell Command Event
Shout Command	TS	Shout Command Event
Kill Command	K	Kill Command Event
Resurrect Command	X	Resurrect Command Event
Quit Command	Q	Quit Command Event

the different screens, or at various locations in the *Game World*. The text in this *Database Table* would be changed to adapt the game to the local regional language where the game was being played.

You can see an example of this *Database Table* in Table 1.34.

TABLE 1.34
Example of a Text Localisations Table

Text ID	English	French	Spanish
Nobility Title	Wicked Baron	Méchant Baron	barón malvado
Serf Title	Meek Serf	Serf débonnaire	siervo manso
Village Title	Teversham	Teversham	Teversham
Forest Title	Dark Forest	Forêt sombre	bosque oscuro
Help Option Title	Help	Aide	ayuda
Warrior Title	Conan the Barbarian	Conan le Barbare	Conan el Bárbaro
Thief Title	Ali Baba the Thief	Ali Baba le voleur	Alí Babá el ladrón
Mage Title	Merlin the Magician	Merlin le magicien	Merlín el Mago
Cleric Title	Luther the Priest	Luther le prêtre	Lutero el Sacerdote

1.4.31 Errors Table

The next *Database Table* would hold **Error Records**. Each *Record* would hold the text that would be displayed when an error occurred. Some of these texts would be displayed by the **Central Host**, when an error occurred with a **Host Module**, before, during or after a game. And the rest would be displayed by the **Game Objects**, when an error occurred during a game. These errors would be similar to the other text that could be displayed in the menus or the *Game World*. Except, some may not even appear on the screen, due to errors with the **Host Modules**. The *Record* would map a **Text ID** to words either displayed on screen or written into a computer file which kept a log of the errors.

You can see an example of this *Database Table* in Table 1.35.

1.4.32 Invisible 2D Point Objects Table

The next *Database Table* would hold **Point Object Records**. Each *Record* would be used to hold the properties of an **Invisible 2D Point Object**. The *Record* would map an **Object ID** to the mass of a point, its position in a 2D *Game World*, its speed and its acceleration. The *Record* would include its orientation (i.e. the angle of rotation about its centre), its rotational speed and its rotational acceleration. The *Record* would also include its *Collision boundary*, its *Proximity boundary*, its **Collision** and **Proximity Events**. And the *Record* would include the **Object Initial Reset Event** and the **Object Destroyed Event** of the 2D point.

When a **2D Point Object**, or any other **Game Object**, had a mass of zero that would mean it was very heavy. So heavy that the force of any impact was negligible, and it would not move as a result of a collision with another **Object**.

Some of the **Game Objects** would be created outside of the *Game World* by the staff. But others would be created inside the *Game World* by the highest level players or Wizards. Therefore space should be reserved in this *Database Table* and other similar *Tables* for the Wizards to add their own **Game Objects**. The difference in the **Objects** created by the staff and those created by the Wizards would be reflected by the following *Database Fields* in this and subsequent *Database Tables*:

Game Object Code Field
Owner Field

The **Game Object Code Field** would contain the custom code for the **Game Object** written by a Wizards who created the **Object** in the *Game World*, using the **LPC Custom**

TABLE 1.35
Example of a Errors Table

Text ID	Text
No memory	No more space available in the memory for Database.
No sound	No resources available on computer hardware to play sounds.
No Game Server	No connection to remote Game Server.

Tool. And the **Owner Field** would contain the name of that Wizard. Only the owner of a **Game Object** would be able to edit its properties and its behaviour.

This behaviour would be controlled by the pseudo machine code instructions or **LPC Code** which was translated or 'compiled' from a file written in the *LPC* programming language by the Wizard. This would control the **Actions** of that **Game Object** in response to **Secondary Events**. These instructions would be executed by the **Objects Host** using a **Virtual Machine** which can interpret **LPC code**.

However, if the **Game Object** were created from outside of the game, then the **Game Object Code Field** would hold the real machine code instructions, translated or 'compiled' from a file written in the same programming language used to build the **Host Modules**. And this would control the **Actions** of each **Game Object** in response to **Secondary Events.** These instructions would be executed by the **Objects Host** using the Central Processor of a real machine.

You can see an example of this *Database Table* in Table 1.36.

TABLE 1.36
Example of an Invisible 2D Point Objects Table

Object ID	Game Object Code	Owner	Mass	X	Y	X Speed	Y Speed
Undiscovered Map Area 1	Map Code 1	Staff	1	90	51	0	0
Undiscovered Map Area 2	Map Code 2	Staff	1	181	195	0	0
Undiscovered Map Area 3	Map Code 3	Staff	1	461	235	0	0

Object ID	X Accel.	Y Accel.	Angular Position (Deg.)	Angular Speed (Deg./Sec.)	Angular Accel. (Deg./Sec./Sec.)
Undiscovered Map Area 1	0	0	0	0	0
Undiscovered Map Area 2	0	0	0	0	0
Undiscovered Map Area 3	0	0	0	0	0

Object ID	Collision Boundary ID	Proximity Boundary ID	Collision Event ID
Undiscovered Map Area 1	None	Village Outline Boundary	None
Undiscovered Map Area 2	None	Forest Outline Boundary	None
Undiscovered Map Area 3	None	Mountain Range Outline Boundary	None

Object ID	Proximity Event ID	Object Initial Reset Event ID	Object Destroyed Event ID
Undiscovered Map Area 1	Enter Village Event	Unknown Map Area Initial Reset Event 1	Unknown Map Area Destroyed Event 1
Undiscovered Map Area 2	Enter Forest Event	Unknown Map Area Initial Reset Event 2	Unknown Map Area Destroyed Event 2
Undiscovered Map Area 3	Enter Mountain Range Event	Unknown Map Area Initial Reset Event 3	Unknown Map Area Destroyed Event 3

1.4.33 Invisible 3D Point Objects Table

The next *Database Table* would hold **Point Object Records**. Each *Record* would be used to hold the properties of an **Invisible 3D Point Object**. The *Record* would map an **Object ID** to the mass of a point, its position in a 3D *Game World*, its speed and its acceleration. The orientation of the point (i.e. the angle of rotation about its local X, Y and Z axes), its X, Y and Z angular speeds, its X, Y and Z angular accelerations would be included in the *Record*. The *Record* would also include the *Collision boundary*, the *Proximity boundary*, the **Collision** and **Proximity Events** of the point. And the *Record* would contain the **Secondary Events** that would be received, by the **Game Object**, when that *Record* was loaded into or removed from the computer memory.

This *Table* would include all of the invisible **Game Objects** that were used by the **Procedurally Generated Quest System** e.g.

Quest Marker Objects
Quest Spline Object
Quest Splines Generator Object
Quest Waypoints Object
Escort Quest Handler Object
Find Quest Handler Object
Kill Quest Handler Object

This *Table* would also include invisible **Game Objects** that allowed you to save the current state of the game to a file or load the current state of the game from a file e.g.

SAVE GAME OBJECT
LOAD GAME OBJECT

You can see an example of this *Database Table* in Table 1.37.

1.4.34 Master Object Table

The next *Database Table* would hold **Point Object Records**. Each *Record* would be used to hold the properties of special **Game Objects** for which there was only one instance in the *Game World*. This would include the

Master Object,
Master Physics Object,
Master Sound Speaker Object and
Master Player Object.

The *Record* would map an **Object ID** to its mass, position, speed, acceleration, angular position, angular speed and angular acceleration.

TABLE 1.37
Example of an Invisible 3D Point Objects Table

Object ID	Game Object Code	Owner	Mass	X	Y	Z	X Speed	Y Speed	Z Speed
Covered Pit Trigger Object	Covered Pit Code	Artful_Dodger	1	1812	4	1955	0	0	0
Bush Snake Trigger Object	Bush Snake Code	Artful_Dodger	1	1813	4	1950	0	0	0
Forcefield Trigger Object	Forcefield Code	Merlin	0	1810	4	1950	0	0	−4

Object ID	X Accel.	Y Accel.	Z Accel.	X Angular Position (Deg.)	Y Angular Position (Deg.)	Z Angular Position (Deg.)
Covered Pit Trigger Object	0	0	0	0	0	0
Bush Snake Trigger Object	0	0	0	0	8	0
Forcefield Trigger Object	0	0	−2	0	0	0

Object ID	X Angular Speed (Deg./Sec.)	Y Angular Speed (Deg./Sec.)	Z Angular Speed (Deg./Sec.)	X Angular Accel. (Deg./Sec./Sec.)	Y Angular Accel. (Deg./Sec./Sec.)	Z Angular Accel. (Deg./Sec./Sec.)
Covered Pit Trigger Object	0	0	0	0	0	0
Bush Snake Trigger Object	0	0	0	0	0	0
Forcefield Trigger Object	0	244	0	0	0	0

Object ID	Collision Boundary ID	Proximity Boundary ID	Collision Event ID	Proximity Event ID
Covered Pit Trigger Object	None	Covered Pit Boundary	None	Pitfall Event
Bush Snake Trigger Object	None	Small Bush Boundary	None	Snake Bite Event
Forcefield Trigger Object	Forcefield Boundary	None	Forcefield Collision Event	None

Object ID	Object Initial Reset Event ID	Object Destroyed Event ID
Covered Pit Trigger Object	Pit Initial Reset Event	Pit Destroyed Event
Bush Snake Trigger Object	Bush Snake Reset Event	Bush Snake Destroyed Event
Forcefield Trigger Object	Forcefield Initial Reset Event	Forcefield Destroyed Event

As with the invisible **2D Point Objects**, the *Record* would also include its *Collision boundary*, its *Proximity boundary*, its **Collision** and **Proximity Events**. And the *Record* would include the **Object Initial Reset Event** and the **Object Destroyed Event**.

But unlike other **Point Objects**, the *Record* would also include

1. the oldest **Game Object** loaded into the computer memory,
2. the latest **Object** loaded into memory, after the last **Primary Heartbeat Event**, and
3. the **Random Seed**.

The **Master Object** would use these values to either tell when new **Objects** had just been loaded into computer memory and should receive **Object Initial Reset Events**. Or when an **Object** was about to be unloaded from memory and should receive its **Object Unloaded Event**. Or what value should be used to start generating random numbers, from the start of the game. For when the *Game World* was being replicated across a computer network, in a **Peer-To-Peer Network Architecture**, in a multiplayer game. And the random numbers that had been generated, from its initial state to its current state, had to be replicated by a new **Game Peer** that joined the network. To synchronise its copy of the *Game World*.

You can see an example of this *Database Table* in Table 1.38.

1.4.35 Text Objects Table

The next *Database Table* would hold **Point Object Records**. Each *Record* would be used to hold the properties of a **Text Object**. The *Record* would map an **Object ID** to the mass of a **Text Object**, and a position in a 2D *Game World*, where its words would appear. The *Record* would include the speed, the acceleration, as well as the orientation, the rotational speed and the rotational acceleration of the text. The

TABLE 1.38
Example of Master Object Table

Object ID	Game Object Code	Owner	X	Y
Master Object	Master Object Code	Staff	0	0
Master Physics Object	Master Physics Object Code	Staff	0	0
Master Sound Speaker Object	Master Sound Speaker Code	Staff	0	0
Master Player Object	Master Game Controller Code	Staff	0	0

Object ID	X Speed	Y Speed	X Accel.	Y Accel.	Angular Position (Deg.)
Master Object	0	0	0	0	0
Master Physics Object	0	0	0	0	0
Master Sound Speaker Object	0	0	0	0	0
Master Player Object	0	0	0	0	0

(*Continued*)

TABLE 1.38 (*Continued*)
Example of Master Object Table

Object ID	Angular Speed (Deg./Sec.)	Angular Accel. (Deg./Sec./Sec.)	Collision Boundary ID
Master Object	0	0	None
Master Physics Object	0	0	None
Master Sound Speaker Object	0	0	None
Master Player Object	0	0	None

Object ID	Proximity Boundary ID	Collision Event ID	Proximity Event ID
Master Object	None	None	None
Master Physics Object	None	None	None
Master Sound Speaker Object	None	None	None
Master Player Object	None	None	None

Object ID	Object Initial Reset Event ID	Object Destroyed Event ID	Oldest Object ID
Master Object	Master Initial Reset Event	Master Destroyed Event	Warrior's Player Object
Master Physics Object	Master Physics Initial Reset Event	Master Physics Destroyed Event	None
Master Sound Speaker Object	Master Speaker Initial Reset Event	Master Speaker Destroyed Event	None
Master Player Object	Master Game Controller Initial Reset Event	Master Game Controller Destroyed Event	None

Object ID	Latest Object ID	Random Seed
Master Object	Forcefield Object	81036166.6545
Master Physics Object	None	70290667.5049
Master Sound Speaker Object	None	56722427.1801
Master Player Object	None	71812558.7534

Object ID	Solid Frictional Accel.	Liquid Frictional Accel.	Gas Friction Accel.
Master Object	0.0	0.0	0.0
Master Physics Object	0.4	0.5	0.2
Master Sound Speaker Object	0.0	0.0	0.0
Master Player Object	0.0	0.0	0.0

Collision boundary, the *Proximity boundary*, the **Collision** and **Proximity Events** of the text would also be included. And the Record would include the **Object Initial Reset Event** and the **Object Destroyed Event** of the **Game Object**.

In addition to these properties, there would be a description of the appearance of the text. This would include its font, the shape of its characters, the *Texture coordinates* of its characters in the font, the words of the text, its colour, its size and its width.

The size of each text would be the height of each character within it. The height would be the same throughout. The width of each character, however, would be a proportion of the height. This would differ depending on the proportions of the character, in the image of the font. That is to say, it would be determined by the proportions of the rectangle, marked out by the *Texture coordinates* of each character.

The width of each text would be the limits to the length of the lines, upon which its words were laid out on. If all the words were laid out on a line, and the length of that line exceeded this limit, all the excess words would continue on the next line underneath. And if the length of this new line exceeded the limit, the excess words on this line would continue on the next line underneath that, and so on.

This *Table* would include the **Text Objects** used by the **Procedurally Generated Quest System** e.g.

Quest Prompt Object.

You can see an example of this *Database Table* in Table 1.39.

TABLE 1.39
Example of a Text Objects Table

Graphic Object ID	Game Object Code	Owner	Mass	X	Y
Warrior Player Text Object	Warrior Player Text Code	Staff	1	1096	380
Thief Player Text Object	Thief Player Text Code	Staff	1	1610	380
Village Label Object	Village Label Code	Lord_Teversham	1	90	51
Forest Label Object	Forest Label Code	Staff	1	181	195
Help Label Object	Help Label Code	Staff	1	615	457

Object ID	X Speed	Y Speed	X Accel.	Y Accel.
Warrior Player Text Object	0	0	0	0
Thief Player Text Object	0	0	0	0
Village Label Object	0	0	0	0
Forest Label Object	0	0	0	0
Help Label Object	0	0	0	0

Object ID	Angular Position (Deg.)	Angular Speed (Deg./Sec.)	Angular Accel. (Deg./Sec./Sec.)	Collision Boundary ID
Warrior Player Text Object	0	0	0	None
Thief Player Text Object	0	0	0	None
Village Label Object	0	0	0	None
Forest Label Object	0	0	0	None
Help Label Object	0	0	0	None

(*Continued*)

TABLE 1.39 (*Continued*)
Example of a Text Objects Table

Object ID	Proximity Boundary ID	Collision Event ID	Proximity Event ID	Object Initial Reset Event ID
Warrior Player Text Object	Player Name Rectangle	None	Change Name Event 1	Warrior's Label Initial Reset Event
Thief Player Text Object	Player Name Rectangle	None	Change Name Event 2	Thief's Label Initial Reset Event
Village Label Object	None	None	None	Village Label Initial Reset Event
Forest Label Object	None	None	None	Forest Label Initial Reset Event
Help Label Object	None	None	None	Help Label Initial Reset Event

Object ID	Object Destroyed Event	Texture ID	2D Polygon ID	Texture Coord. ID
Warrior Player Text Object	Warrior's Label Destroyed Event	Arial Font Texture	Warrior Text Polygon	Warrior Text Texture Coord
Thief Player Text Object	Thief's Label Destroyed Event	Arial Font Texture	Thief Text Polygon	Thief Text Texture Coord
Village Text Object	Village Text Destroyed Event	Bold Roman Font Texture	Village Text Polygon	Village Text Texture Coord
Forest Text Object	Forest Text Destroyed Event	Bold Roman Font Texture	Forest Text Polygon	Forest Text Texture Coord
Help Text Object	Help Text Destroyed Event	Italic Font Texture	Help Text Polygon	Help Text Texture Coord

Object ID	Materials ID	Text ID	Colour (RGBA Format)	Size	Width
Warrior Player Text Object	Arial Font Material	Nobility Title	Bright Yellow	12	144
Thief Player Text Object	Arial Font Material	Serf Title	Bright Yellow	12	108
Village Text Object	Bold Roman Font Material	Village Title	White	24	180
Forest Text Object	Bold Roman Font Material	Forest Title	White	24	360
Help Text Object	Italic Font Material	Help Option Title	Bright Red	15	60

1.4.36 2D Image Objects Table

The next *Database Table* would hold **Point Object Records**. Each *Record* would be used to hold the properties of a **2D Image Object**. The *Record* would map an **Object ID** to the mass of an **Image Object**, the position at which a 2D image would appear, its speed and its acceleration. The *Record* would also include the orientation

of the **Image Object**, its rotational speed and its rotational acceleration. And the *Record* would include its *Collision boundary*, its *Proximity boundary*, its **Collision** and **Proximity Events**. The **Secondary Events** the **Game Object** would receive, when the *Record* was loaded into or removed from the computer memory, would be in there too.

Along with these properties, the *Record* would include the *Texture* from which the image would appear, a *polygon* describing its shape, a set of matching *Texture coordinates* and the size of the image. The size would be the scale by which the default width and height of the image would be increased or reduced.

You can see an example of this *Database Table* in Table 1.40.

TABLE 1.40
Example of a 2D Image Objects Table

Graphic Object ID	Game Object Code	Owner	Mass	X	Y
Warrior Health Bar Object	Warrior Health Bar Code	Staff	1	1059	396
Thief Health Bar Object	Thief Health Bar Code	Staff	1	1629	396
Game Map Object	Game Map Code	Staff	1	512	384
Help Icon Object	Help Icon Code	Staff	1	2663	429

Object ID	X Speed	Y Speed	X Accel.	Y Accel.	Angular Position (Deg.)
Warrior Health Bar Object	0	0	0	–2	0
Thief Health Bar Object	0	0	0	1	0
Game Map Object	0	0	0	0	0
Help Icon Object	0	0	0	0	0

Object ID	Angular Speed (Deg./Sec.)	Angular Accel. (Deg./Sec./Sec.)	Collision Boundary ID	Proximity Boundary ID
Warrior Health Bar Object	0	0	None	None
Thief Health Bar Object	0	0	None	None
Game Map Object	0	0	None	None
Help Icon Object	0	0	None	Help Icon Boundary

Object ID	Collision Event ID	Proximity Event ID	Object Initial Reset Event ID	Object Destroyed Event ID
Warrior Health Bar Object	None	None	Warrior's Health Bar Initial Reset Event	Warrior's Health Bar Destroyed Event
Thief Health Bar Object	None	None	Thief's Health Bar Initial Reset Event	Thief's Health Bar Destroyed Event
Game Map Object	None	None	Game Map Initial Reset Event	Game Map Destroyed Event
Help Icon Object	None	Help Icon Selected Event	Help Icon Initial Reset Event	Help Icon Destroyed Event

(*Continued*)

TABLE 1.40 (*Continued*)
Example of a 2D Image Objects Table

Object ID	Texture ID	2D Polygon ID	Texture Coord. ID	Material ID
Warrior Health Bar Object	Health Bar Texture	Health Bar Polygon	Health Bar Texture Coord	Health Bar Material
Thief Health Bar Object	Health Bar Texture	Health Bar Polygon	Health Bar Texture Coord	Health Bar Material
Game Map Object	Game Map Texture	Game Map Polygon	Game Map Texture Coord	Game Map Material
Help Icon Object	Icons Texture	Help Icon Polygon	Help Icon Texture Coord	Icons Material

Object ID	Width	Height
Warrior Health Bar Object	0.9	1
Thief Health Bar Object	0.4	1
Game Map Object	1	1
Help Icon Object	1	1

1.4.37 2D Animation Objects Table

The next *Database Table* would hold **Point Object Records**. Each *Record* would hold the properties of a **2D Animation Object**. The *Record* would map an **Object ID** to the mass of an **Animation Object**, and the position in a 2D *Game World*, where an animated image would be displayed. The *Record* would include the speed of the image, its acceleration, its orientation, its rotational speed and its rotational acceleration. And the *Record* would include the *Collision boundary* of the image, its *Proximity boundary*, its **Collision** and **Proximity Events**. The *Record* would contain the **Object Initial Reset Event** and the **Object Destroyed Event** of the **Game Object** as well.

As with **2D Image Objects**, the *Record* would have the current image of the **Object**, a *polygon* describing its shape, a set of matching *Texture coordinates* and the size of the image.

But, in addition, the *Record* would include the sequence of animated images, a list of the *polygons* and the *Texture coordinates* of each *Frame*. The *Record* would also include the rate at which the *Frames* would be displayed, the length of the animation, how much time had elapsed since the animation began and an **End Event**.

If the sequence of animated images, described in the *Record*, contained more than one image, then each image would be displayed for an equal amount of time. This time would be simply the total length of the animation divided by the number of images. And during each interval, the next *Frame* of the animation would be taken from the current image. But if only one image were contained in the sequence, then all the *Frames* of the animation would be taken from that image.

A similar principle would apply for the list of *polygons* displaying each *Frame* of the animation. If this list contained more than one *polygon*, then each one would be

displayed for an equal amount of time. And this would be determined by the total length of the animation divided by the number of *polygons*.

Alternately, some 2D animations may be done by animating the vertices of the 2D *polygons* of the image. Rather than animating the image itself. In this case, the *Record* would include a reference or **Animation ID.** To the set of the **Animated Vertices** in the *Database Table* of **Animated Vertices Graphics Object Records** that should be used to animate the *polygon*.

You can see an example of this *Database Table* in Table 1.41.

TABLE 1.41
Example of 2D Animation Objects Table

Graphic Object ID	Game Object Code	Owner	Mass	X	Y
Village Animation Object	Village Animation Code	Lord_Teversham	1	90	51
Forest Animation Object	Forest Animation Code	Staff	1	181	195
Mountain Range Animation Object	Mountain Range Animation Code	Lord_Carpathia	1	461	235
2D Icon Crossbow Animation Object	2D Icon Crossbow Animation Code	Trader_Legolas	1	1066	876

Object ID	X Speed	Y Speed	X Accel.	Y Accel.
Village Animation Object	0	0	0	0
Forest Animation Object	0	0	0	0
Mountain Range Animation Object	0	0	0	0
2D Icon Crossbow Animation Object	0	0	0	0

Object ID	Angular Position (Deg.)	Angular Speed (Deg./Sec.)	Angular Accel. (Deg./Sec./Sec.)	Collision Boundary ID
Village Animation Object	0	0	0	None
Forest Animation Object	0	0	0	None
Mountain Range Animation Object	0	0	0	None
2D Icon Crossbow Animation Object	0	0	0	None

Object ID	Proximity Boundary ID	Collision Event ID	Proximity Event ID	Object Initial Reset Event ID
Village Animation Object	None	None	None	Village Animation Reset Event
Forest Animation Object	None	None	None	Forest Animation Reset Event
Mountain Range Animation Object	None	None	None	Mountain Range Reset Event
2D Icon Crossbow Animation Object	None	None	None	2D Icon Crossbow Reset Event

(*Continued*)

TABLE 1.41 (*Continued*)
Example of 2D Animation Objects Table

Object ID	Object Destroyed Event ID	Texture ID	2D Polygon ID
Village Animation Object	Village Animation Destroyed Event	Village Animation Texture	Village Polygon
Forest Animation Object	Forest Animation Destroyed Event	Forest Animation Texture	Forest Polygon
Mountain Range Animation Object	Mountain Range Animation Destroyed Event	Mountain Range Animation Texture	Mountain Range Polygon
2D Icon Crossbow Animation Object	2D Icon Crossbow Animation Destroyed Event	2D Icon Crossbow Texture	2D Icon Crossbow Polygon

Object ID	Texture Coord. ID	Width	Height	Texture IDs
Village Animation Object	Village Frame 5 Texture Coord	1	1	Village Animation Texture
Forest Animation Object	Forest Frame 3 Texture Coord	1	1	Forest Animation Texture
Mountain Range Animation Object	Mountain Range Frame 2 Texture Coord	1	1	Mountain Range Animation Texture
2D Icon Crossbow Animation Object	None	1	1	2D Icon Texture

Object ID	2D Polygon IDs	Texture Coord. IDs	Material ID	Animation ID
Village Animation Object	Village Polygon	*Texture coordinates* of *Frames* 1–80	Village Animation Material	None
Forest Animation Object	Forest Polygon	*Texture coordinates* of *Frames* 1–90	Forest Animation Material	None
Mountain Range Animation Object	Mountain Range Polygon	*Texture coordinates* of *Frames* 1–110	Mountain Range Animation Material	None
2D Icon Crossbow Animation Object	2D Icon Crossbow Polygon	None	2D Icon Material	2D Icon Crossbow Animation

Object ID	Animation Rate (Frames Per Sec.)	Animation Length (Sec.)	Animation Elapsed (Sec.)	End Event ID
Village Animation Object	60	1.3	0.08	End Village Animation Event
Forest Animation Object	60	1.5	0.05	End Forest Animation Event
Mountain Range Animation Object	60	1.83	0.03	End Mountain Range Animation Event
2D Icon Crossbow Animation Object	60	2	0.00	End Icon Crossbow Animation Event

1.4.38 2D Player Objects Table

The next *Database Table* would hold **Point Object Records**. Each *Record* would hold the properties of a **2D Player Object**. This could either represent a 2D character or creature controlled by a player in a 2D *Game World*. Or it could be a cursor controlled by a player in a 2D menu. The *Record* would map an **Object ID** to the mass of a character, creature or cursor, its position, its speed and its acceleration, in a 2D *Game World* or menu. The *Record* would include its orientation, rotational speed and rotational acceleration. The *Record* would also include its *Collision boundary*, *Proximity boundary*, **Collision** and **Proximity Events**. And the *Record* would include the **Secondary Events**, the **Game Object** would receive, when that *Record* was loaded into or removed from the computer memory. And the **Secondary Event** the **Game Object** would periodically receive during each round of combat.

Accompanying these properties, the *Record* would contain a description of the appearance of the character in the game. This would include an image or icon, a *polygon* describing its shape, a set of matching *Texture coordinates* and the size of the image.

The *Record* would also contain the properties of the *Game Controller* that may be directing the character. These include the **Device Group**, **Device Mapping**, **Controller Maximum**, **Controller Central**, **Controller Minimum**, **Analogue History**, **Analogue Positions**, **Digital History** and **Digital Positions Fields**.

The words used in the **Device Group**, **Analogue History** and **Digital History**, to identify the axes of the *Game Controllers*, would be 'Mouse X' and 'Mouse Y'. The former would identify the X-axis. This would control whether the players' characters moved left or right across the *Game World*. The latter would identify the Y-axis. This would control whether the characters moved up or down the *Game World*.

The digital buttons on the *Game Controllers* would be identified by the word 'Select' (e.g. the left button on a mouse with two buttons or a button labelled 'Select' on a *Game Controller* or the space bar on a keyboard). This would identify some option the player had selected on a menu or some frequent command which changes depending on the context (e.g. causing the player to jump in one location or climb down in another or pick up the closest item lying nearby in another).

The digital buttons would also be identified by 'Up', 'Down', 'Left' and 'Right' (e.g. the cursor keys on a keyboard or directional buttons on a Gamepad). These would move the players' characters up, down, left or right across the *Game World*.

Each button on the players' keyboards would also be identified by the readable character or abbreviation on that button (e.g. A, S, D, F, backspace and enter).

Furthermore, with the identity of the *devices* on the *Game Controllers*, the *Record* would contain the **Secondary Events** the **Object** would receive, when a **Game Controllers Host Event** occurred. These include the **Connect**, **Disconnect**, **Moved**, **Stopped**, **Pressed** and **Released Events**.

Besides these, the *Record* would incorporate the properties of the character that reflects its strength in combat, its knowledge and its reputation in the *Game World*. These include the **Weapon Class**, **Armour Class**, health and maximum health of the character. These include the inventory of the items being used by that character, and the inventory of the items being carried. These also include the score (or level of experience) and the name of the character.

This *Table* would also hold the **Game Objects** of 2D NPCs. Since these would share similar properties to the **Game Objects** of interactive Player Characters.

You can see an example of this *Database Table* in Table 1.42.

1.4.39 3D Image Objects Table

The next *Database Table* would hold **Point Object Records**. Each *Record* would hold the properties of a **3D Model Object**. The *Record* would map an **Object ID** to

TABLE 1.42
Example of 2D Player Objects Table

Graphic Object ID	Game Object Code	Mass	X	Y	X Speed	Y Speed
Warrior 2D Player Object	Warrior 2D Player Code	100	320	240	0	0
Thief 2D Player Object	Thief 2D Player Code	90	90	51	0	0
Mage 2D Player Object	Mage 2D Player Code	80	121	128	0	0
Cleric 2D Player Object	Cleric 2D Player Code	50	192	200	0	0

Object ID	X Accel.	Y Accel.	Angular Position (Deg.)	Angular Speed (Deg./Sec.)
Warrior 2D Player Object	0	0	0	0
Thief 2D Player Object	0	0	0	0
Mage 2D Player Object	0	0	0	0
Cleric 2D Player Object	0	0	0	0

Object ID	Angular Accel. (Deg./Sec./Sec.)	Collision Boundary ID	Proximity Boundary ID	Collision Event ID
Warrior 2D Player Object	0	Warrior 2D Low Res. Polygon	Warrior 2D Low Res. Polygon	Warrior Collis. Event
Thief 2D Player Object	0	Thief 2D Low Res. Polygon	Thief 2D Low Res. Polygon	Thief Collis. Event
Mage 2D Player Object	0	Mage 2D Low Res. Polygon	Mage 2D Low Res. Polygon	Mage Collis. Event
Cleric 2D Player Object	0	Cleric 2D Low Res. Polygon	Cleric 2D Low Res. Polygon	Cleric Collis. Event

Object ID	Proximity Event ID	Object Initial Reset Event ID	Object Destroyed Event ID	Object Heartbeat Event ID
Warrior 2D Player Object	Enter Player 1 Event	Player 1 Reset Event	Player 1 Destroyed Event	Player 1 Heartbeat Event
Thief 2D Player Object	Enter Player 2 Event	Player 2 Reset Event	Player 2 Destroyed Event	Player 2 Heartbeat Event
Mage 2D Player Object	Enter Player 3 Event	Player 3 Reset Event	Player 3 Destroyed Event	Player 3 Heartbeat Event
Cleric 2D Player Object	Enter Player 4 Event	Player 4 Reset Event	Player 4 Destroyed Event	Player 4 Heartbeat Event

(*Continued*)

TABLE 1.42 (*Continued*)
Example of 2D Player Objects Table

Object ID	Texture ID	2D Polygon ID	Texture Coord. ID	Material ID
Warrior 2D Player Object	Icons Texture	Warrior 2D High Res. Polygon	Warrior Attack Texture Coord	Icons Material
Thief 2D Player Object	Icons Texture	Thief 2D High Res. Polygon	Thief Face Left Texture Coord	Icons Material
Mage 2D Player Object	Icons Texture	Mage 2D High Res. Polygon	Mage Attack Texture Coord	Icons Material
Cleric 2D Player Object	Icons Texture	Cleric 2D High Res. Polygon	Cleric Face Left Texture Coord	Icons Material

Object ID	Width	Height	Device Group	Device Mapping
Warrior 2D Player Object	1	1	Mouse X, Mouse Y, Select	X, Y Fields
Thief 2D Player Object	1	1	Up, Down, Left, Right, Select	Y, Y, X, X Fields
Mage 2D Player Object	1	1	Mouse X, Mouse Y, Select	X, Y Fields
Cleric 2D Player Object	1	1	Up, Down, Left, Right, Select	Y, Y, X, X Fields

Object ID	Controller Maximum	Controller Central	Controller Minimum	Analogue History	Analogue Positions
Warrior 2D Player Object	7	0	−7	Mouse X, Mouse X	−30, −18
Thief 2D Player Object	1	0	−1	Empty	Empty
Mage 2D Player Object	7	0	−7	Mouse X, Mouse Y	−22, 33
Cleric 2D Player Object	1	0	−1	Empty	Empty

Object ID	Digital History	Digital Positions	Connect Event ID	Disconnect Event ID
Warrior 2D Player Object	Select	56	Player 1 Connected Event	Player 1 Disconnected Event
Thief 2D Player Object	Left, Up, Select	132, 132, 134	Player 2 Connected Event	Player 2 Disconnected Event
Mage 2D Player Object	Start	96	Player 3 Connected Event	Player 3 Disconnected Event
Cleric 2D Player Object	Left, Up, Select	132, 132, 134	Player 4 Connected Event	Player 4 Disconnected Event

Object ID	Moved Event ID	Stopped Event ID	Pressed Event ID
Warrior 2D Player Object	Player 1 Moved Event	Player 1 Stopped Event	Player 1 Pressed Event
Thief 2D Player Object	Player 2 Moved Event	Player 2 Stopped Event	Player 2 Pressed Event
Mage 2D Player Object	Player 3 Moved Event	Player 3 Stopped Event	Player 3 Pressed Event
Cleric 2D Player Object	Player 4 Moved Event	Player 4 Stopped Event	Player 4 Pressed Event

(*Continued*)

TABLE 1.42 (*Continued*)
Example of 2D Player Objects Table

Object ID	Released Event ID	Health	Max Health	Weapon Class	Armour Class
Warrior 2D Player Object	Player 1 Released Event	230.38	300	60	200
Thief 2D Player Object	Player 2 Released Event	−1	200	40	160
Mage 2D Player Object	Player 3 Released Event	140.10	180	30	80
Cleric 2D Player Object	Player 4 Released Event	173.33	190	20	50

Object ID	Used Inventory	Carried Inventory	Experience	Name
Warrior 2D Player Object	Empty	Empty	57	Warrior Title
Thief 2D Player Object	Empty	Empty	0	Thief Title
Mage 2D Player Object	Empty	Empty	8123	Mage Title
Cleric 2D Player Object	Empty	Empty	164,440	Cleric Title

the mass of a 3D model, and its position in a *Game World*. The *Record* would include the speed and the acceleration of the model. And the *Record* would include its rotation about its local X, Y and Z axes, its X, Y and Z angular speeds, its X, Y and Z angular accelerations. The *Record* would also include the *Collision boundary* of the model, its *Proximity boundary*, its **Collision Event** and its **Proximity Event**. And the *Record* would include the **Object Initial Reset Event** and the **Object Destroyed Event** of the model.

Along with these properties, the *Record* would include a description of the appearance of the model. This would include the *Texture*, a **3D Model ID**, a set of matching *Texture coordinates* and the size of the model. The size would be the scale by which the default width, height and breadth of the model would be increased or reduced.

This *Table* would include all the inanimate **Game Objects** used by the **Procedurally Generated Quest System** e.g.

Quest Lost Object.

You can see an example of this *Database Table* in Table 1.43.

1.4.40 3D Animation Objects Table

The next *Database Table* would hold **Point Object Records**. Each *Record* would hold the properties of a **3D Animation Object**. The *Record* would map an **Object ID** to the mass of an animated model, its position, its speed and its acceleration, in a 3D *Game World*. The *Record* would include the angle of rotation of the model, about its local X, Y and Z axes, its speed of rotation about these axes, as well as its acceleration around these axes. The *Record* would also include its *Collision boundary*, its *Proximity boundary*, its **Collision Event** and its **Proximity Event**. And the *Record* would contain the **Secondary Events** the **Game Object** would receive, when that *Record* was loaded into or removed from the computer memory.

TABLE 1.43
Example of 3D Image Objects Table

Graphic Object ID	Game Object Code	Owner	Mass	X	Y	Z	X Speed	Y Speed
Forest Tree Object 1	Forest Tree Code	Staff	0	1812	4	1946	0	0
Forest Tree Object 2	Forest Tree Code	Staff	0	1814	4	1945	0	0
Forest Bush Object 1	Forest Bush Code	Staff	0	1813	4	1950	0	0
Forest Bush Object 2	Forest Bush Code	Staff	0	1811	4	1946	0	0

Object ID	Z Speed	X Accel.	Y Accel.	Z Accel.	X Angle (Deg.)	Y Angle (Deg.)	Z Angle (Deg.)
Forest Tree Object 1	0	0	0	0	0	1	0
Forest Tree Object 2	0	0	0	0	0	316	0
Forest Bush Object 1	0	0	0	0	0	56	0
Forest Bush Object 2	0	0	0	0	0	72	0

Object ID	X Angular Speed (Deg./Sec.)	Y Angular Speed (Deg./Sec.)	Z Angular Speed (Deg./Sec.)	X Angular Accel. (Deg./Sec./Sec.)
Forest Tree Object 1	0	0	0	0
Forest Tree Object 2	0	0	0	0
Forest Bush Object 1	0	0	0	0
Forest Bush Object 2	0	0	0	0

Object ID	Y Angular Accel. (Deg./Sec./Sec.)	Z Angular Accel. (Deg./Sec./Sec.)	Collision Boundary ID	Proximity Boundary ID
Forest Tree Object 1	0	0	Forest Tree Boundary	None
Forest Tree Object 2	0	0	Forest Tree Boundary	None
Forest Bush Object 1	0	0	Forest Bush Boundary	None
Forest Bush Object 2	0	0	Forest Bush Boundary	None

Object ID	Collision Event ID	Proximity Event ID	Object Initial Reset Event ID	Object Destroyed Event ID
Forest Tree Object 1	Forest Tree Collision Event	None	Forest Tree Initial Reset Event 1	Forest Tree Destroyed Event 1
Forest Tree Object 2	Forest Tree Collision Event	None	Forest Tree Initial Reset Event 2	Forest Tree Destroyed Event 2
Forest Bush Object 1	Forest Bush Collision Event	None	Forest Bush Initial Reset Event 1	Forest Bush Destroyed Event 1
Forest Bush Object 2	Forest Bush Collision Event	None	Forest Bush Initial Reset Event 2	Forest Bush Destroyed Event 2

(*Continued*)

TABLE 1.43 (*Continued*)
Example of 3D Image Objects Table

Object ID	Texture ID	3D Model ID	Texture Coord. ID	Materials ID
Forest Tree Object 1	Forest Tree Texture	Forest Tree Model	Forest Tree Texture Coord	Forest Tree Material
Forest Tree Object 2	Forest Tree Texture	Forest Tree Model	Forest Tree Texture Coord	Forest Tree Material
Forest Bush Object 1	Forest Bush Texture	Forest Bush Model	Forest Bush Texture Coord	Forest Bush Material
Forest Bush Object 2	Forest Bush Texture	Forest Bush Model	Forest Bush Texture Coord	Forest Bush Material

Object ID	Width	Height	Breadth
Forest Tree Object 1	1	2.3	1
Forest Tree Object 2	1	4.5	1
Forest Bush Object 1	1	1	1
Forest Bush Object 2	1	1	1

Like **3D Model Objects**, the *Record* would include the *Texture* of the model, a **3D Model ID**, a set of matching *Texture coordinates* and the size of the model. Unlike **3D Model Objects**, the *Record* would include the **Animation ID**, of the set of **Animated Vertices** that would be used, and the **End Event** that would signal the end of the animation.

You can see an example of this *Database Table* in Table 1.44.

1.4.41 3D Player Objects Table

The next *Database Table* would hold **Point Object Records**. Each *Record* would hold the properties of a **3D Player Object**. The *Record* would map an **Object ID** to the mass of a character (or creature) in the game, and its position in a 3D *Game World*. The *Record* would include the speed of the character and its acceleration. The *Record* would also include its rotation about its local X, Y and Z axes, the speed of these rotations and the acceleration of these rotations. Together with these properties, the *Record* would contain the *Collision boundary*, the *Proximity boundary*, the **Collision Event** and the **Proximity Event** of the **Game Object**. And the *Record* would include the **Object Initial Reset Event** and the **Object Destroyed Event** of the **Game Object**. And the **Object Heartbeat Event** the **Game Object** would periodically receive during each round of combat.

Similar to other **3D Model Objects**, the *Record* would include its *Texture*, a 3D model, a corresponding set of *Texture coordinates* and the size of the model. But unlike other **3D Model Objects**, the *Record* would contain the properties of a player's *Game Controller*. These include the **Device Group**, **Device Mapping**, **Controller Maximum**, **Controller Central**, **Controller Minimum**, **Analogue History**, **Analogue Positions**, **Digital History** and **Digital Positions Fields**.

TABLE 1.44
Example of 3D Animation Objects Table

Graphic Object ID	Game Object Code	Owner	Mass	X	Y	Z
Warrior Death Animation Object	Warrior Death Animation Code	Staff	90.65	1810	4	1950
Thief Death Animation Object	Thief Death Animation Code	Staff	71.66	1849	4	1947
Mage Attack Animation Object	Mage Attack Animation Code	Staff	80.0	2821.57	4	263.63
Cleric Jump Animation Object	Cleric Jump Animation Code	Staff	80.0	844.98	4	2662.59

Object ID	X Speed	Y Speed	Z Speed	X Accel.	Y Accel.	Z Accel.
Warrior Death Animation Object	0	0	−4	0	0	−2
Thief Death Animation Object	0	0	0	−0.62	0	−0.6
Mage Attack Animation Object	10	0	0	0.35937685	0	0.48958475
Cleric Jump Animation Object	24	0	0	0.09604891	0	0.48528234

Object ID	X Angular Position (Deg.)	Y Angular Position (Deg.)	Z Angular Position (Deg.)	X Angular Speed (Deg. Per Sec.)	Y Angular Speed (Deg. Per Sec.)	Z Angular Speed (Deg. Per Sec.)
Warrior Death Animation Object	0	274	0	0	0	0
Thief Death Animation Object	0	280	0	0	2	0
Mage Attack Animation Object	0	409	0	0	1.44	0
Cleric Jump Animation Object	0	322	0	0	1.14	0

Object ID	X Angular Accel. (Deg./Sec./Sec.)	Y Angular Accel. (Deg./Sec./Sec.)	Z Angular Accel. (Deg./Sec./Sec.)	Collision Boundary ID
Warrior Death Animation Object	0	0	0	Warrior's Boundary
Thief Death Animation Object	0	1	0	Thief's Boundary
Mage Attack Animation Object	0	3.18	0	Mage's Boundary
Cleric Jump Animation Object	0	8.45	0	Cleric's Boundary

(*Continued*)

TABLE 1.44 (*Continued*)
Example of 3D Animation Objects Table

Object ID	Proximity Boundary ID	Collision Event ID	Proximity Event ID	Object Initial Reset Event ID
Warrior Death Animation Object	None	Warrior's Collision Event	None	Warrior's Death Animation Reset Event
Thief Death Animation Object	None	Thief's Collision Event	None	Thief's Death Animation Reset Event
Mage Attack Animation Object	None	Mage's Collision Event	None	Mage's Attack Animation Reset Event
Cleric Jump Animation Object	None	Cleric's Collision Event	None	Cleric's Jump Animation Reset Event

Object ID	Object Destroyed Event ID	Texture ID	3D Model ID	Texture Coord. ID	Material ID
Warrior Death Animation Object	Warrior's Death Animation Destroyed Event	Warrior Texture	Warrior Model	Warrior Texture Coordinates	Warrior Material
Thief Death Animation Object	Thief's Death Animation Destroyed Event	Thief Texture	Thief Model	Thief Texture Coordinates	Thief Material
Mage Attack Animation Object	Mage's Attack Animation Destroyed Event	Mage Texture	Mage Model	Mage Texture Coordinates	Mage Material
Cleric Jump Animation Object	Cleric's Jump Animation Destroyed Event	Cleric Texture	Cleric Model	Cleric Texture Coordinates	Cleric Material

Object ID	Width	Height	Breadth
Warrior Death Animation Object	0.22	0.8	0.12
Thief Death Animation Object	0.22	0.76	0.12
Mage Attack Animation Object	0.22	0.93	0.12
Cleric Jump Animation Object	0.22	0.59	0.12

Object ID	Animation ID	End Event
Warrior Death Animation Object	Warrior's Death Animation	End Warrior's Death Animation Event
Thief Death Animation Object	Thief's Death Animation	End Thief's Death Animation Event
Mage Attack Animation Object	Mage's Attack Animation	End Mage's Attack Animation Event
Cleric Jump Animation Object	Cleric's Jump Animation	End Cleric's Jump Animation Event

The words used in the **Device Group**, **Analogue History** and **Digital History**, to identify the two axes of the *Game Controllers*, would be 'Forwards' and 'Sideways'. The former would identify the Y-axis (e.g. the Y-axis of a mouse or a joystick on a *Game Controller*). This would control whether the players' characters moved away from, or backwards towards, a camera displaying the view of the *Game World*. The latter would identify the X-axis (e.g. the X-axis of a mouse or a joystick on a *Game Controller*. This would control whether the characters moved to one side of the view, or the other.

The digital buttons on the *Game Controllers* would be identified by the word 'Select' (e.g. the left button on a mouse with two buttons or a button labelled 'Select' on a *Game Controller* or the space bar on a keyboard). This would identify some option the player had selected on a menu or some frequent command which changes depending on the context (e.g. causing the player to jump in one location or climb down in another or pick up the closest item lying nearby in another).

Each button on the players' keyboards would also be identified by the readable character or abbreviation on that button (e.g. A, S, D, F, backspace and enter).

As well as the identities of the *devices* on the *Game Controllers*, the *Record* would contain the **Secondary Events** the **Object** would receive, when a **Game Controllers Host Event** occurred. These include the **Connect**, **Disconnect**, **Moved**, **Stopped**, **Pressed** and **Released Events**.

In addition, the *Record* would comprise the properties of the character that reflect its knowledge of the game, and reputation in the *Game World*. These include the **Weapon Class**, **Armour Class**, health and maximum health of the character. These include the inventory of items being used and the inventory of items being carried by that character. These also include the score (or level of experience) and the name of the character.

This *Table* would also hold the **Game Objects** of 3D NPCs. Since these would share similar properties to the **Game Objects** of interactive Player Characters. This includes all the NPCs used by the **Procedurally Generated Quest System** e.g.

Quest Giver Object

Quest Target Object.

You can see an example of this *Database Table* in Table 1.45.

1.4.42 2D Camera Objects Table

The next *Database Table* would hold **Point Object Records**. Each *Record* would hold the properties of a **2D Camera Object**. The *Record* would map an **Object ID** to the mass of a camera, its position, its speed and its acceleration in a 2D *Game World*. The *Record* would contain the angle of rotation of the camera, about its centre, as well as the speed and the acceleration of the rotation, around this centre. The *Record* would include the *Collision boundary*, the *Proximity boundary*, the **Collision Event** and the **Proximity Event** of the camera. And the *Record* would

TABLE 1.45
Example of 3D Player Objects Table

Graphic Object ID	Game Object Code	Mass
Warrior 3D Player Object	Warrior 3D Player Code	90.65
Thief 3D Player Object	Thief 3D Player Code	71.66
Mage 3D Player Object	Mage 3D Player Code	84.5638438998
Cleric 3D Player Object	Cleric 3D Player Code	53.7580015622

Object ID	X	Y	Z	X Speed	Y Speed	Z Speed
Warrior 3D Player Object	1810	4	1150	0	0	−4
Thief 3D Player Object	1809	4	1947	0	0	0
Mage 3D Player Object	254.722392715	4	3536.81	0	0	0
Cleric 3D Player Object	385.136867866	4	2051.45	2.0	0	0

Object ID	X Accel.	Y Accel.	Z Accel.
Warrior 3D Player Object	0	0	−2
Thief 3D Player Object	−0.62	0	−0.6
Mage 3D Player Object	0.738996442115	0	0.797312420234
Cleric 3D Player Object	0.119582546636	0	0.873192766512

Object ID	X Angular Position (Deg.)	Y Angular Position (Deg.)	Z Angular Position (Deg.)	X Angular Speed (Deg./Sec.)	Y Angular Speed (Deg./Sec.)	Z Angular Speed (Deg./Sec.)
Warrior 3D Player Object	0	274	0	0	0	0
Thief 3D Player Object	0	280	0	0	2	0
Mage 3D Player Object	0	57.0104591885	0	0	1.20654100303	0
Cleric 3D Player Object	0	163.723237027	0	0	0.989068027043	0

Object ID	X Angular Accel. (Deg./Sec./Sec.)	Y Angular Accel. (Deg./Sec./Sec.)	Z Angular Accel. (Deg./Sec./Sec.)	Collision Boundary ID
Warrior 3D Player Object	0	0	0	Warrior's Low Res. Boundary
Thief 3D Player Object	0	1	0	Thief's Low Res. Boundary
Mage 3D Player Object	0	4.20412222112	0	Mage's Low Res. Boundary
Cleric 3D Player Object	0	0.105251248162	0	Cleric's Low Res. Boundary

(*Continued*)

TABLE 1.45 (*Continued*)
Example of 3D Player Objects Table

Object ID	Proximity Boundary ID	Collision Event ID	Proximity Event ID	Object Initial Reset Event ID
Warrior 3D Player Object	Warrior's Low Res. Boundary	Player 1 Collision Event	Player 1 Proximity Event	Player 1 Reset Event
Thief 3D Player Object	Thief's Low Res. Boundary	Player 2 Collision Event	Player 2 Proximity Event	Player 2 Reset Event
Mage 3D Player Object	Mage's Low Res. Boundary	Player 3 Collision Event	Player 3 Proximity Event	Player 3 Reset Event
Cleric 3D Player Object	Cleric's Low Res. Boundary	Player 4 Collision Event	Player 4 Proximity Event	Player 4 Reset Event

Object ID	Object Destroyed Event ID	Object Heartbeat Event ID	Texture ID	3D Model ID
Warrior 3D Player Object	Player 1 Destroyed Event	Player 1 Heartbeat Event	Warrior Texture	Warrior Model
Thief 3D Player Object	Player 2 Destroyed Event	Player 2 Heartbeat Event	Thief Texture	Thief Model
Mage 3D Player Object	Player 3 Destroyed Event	Player 3 Heartbeat Event	Mage Texture	Mage Model
Cleric 3D Player Object	Player 4 Destroyed Event	Player 4 Heartbeat Event	Cleric Texture	Cleric Model

Object ID	Texture Coord. ID	Material ID	Width	Height	Breadth
Warrior 3D Player Object	Warrior Texture Coordinates	Warrior Material	0.22	0.8	0.12
Thief 3D Player Object	Thief Texture Coordinates	Thief Material	0.22	0.76	0.12
Mage 3D Player Object	Mage Texture Coordinates	Mage Material	0.22	0.93	0.12
Cleric 3D Player Object	Cleric Texture Coordinates	Cleric Material	0.22	0.59	0.12

Object ID	Device Group	Device Mapping	Controller Maximum	Controller Central
Warrior 3D Player Object	Sideways, Forwards, Select	X Accel., Z Accel. Fields	8	0
Thief 3D Player Object	Up, Down, Left, Right, Select, other buttons	Z Accel., Z Accel., X Accel., X Accel. Fields	8	0
Mage 3D Player Object	Sideways, Forwards, Select	X Accel., Z Accel. Fields	8	0
Cleric 3D Player Object	Up, Down, Left, Right, Select, other buttons	Z Accel., Z Accel., X Accel., X Accel. Fields	8	0

(*Continued*)

TABLE 1.45 (*Continued*)
Example of 3D Player Objects Table

Object ID	Controller Minimum	Analogue History	Analogue Positions
Warrior 3D Player Object	−8	Forwards, Forwards	4, 8
Thief 3D Player Object	−8	Empty	Empty
Mage 3D Player Object	−8	Sideways, Sideways	−4, −8
Cleric 3D Player Object	−8	Empty	Empty

Object ID	Digital History	Digital Positions	Connect Event ID	Disconnect Event ID
Warrior 3D Player Object	Select, Select	152, 153	Player 1 Connected Event	Player 1 Disconnected Event
Thief 3D Player Object	D, D, i, i, e, e.	155, 155, 155, 155, 155, 155	Player 2 Connected Event	Player 2 Disconnected Event
Mage 3D Player Object	Select, Select, Select	163, 164	Player 3 Connected Event	Player 3 Disconnected Event
Cleric 3D Player Object	H, H, e, e, l, l, p, p.	160, 160, 160, 160, 160, 160, 160, 160	Player 4 Connected Event	Player 4 Disconnected Event

Object ID	Moved Event ID	Stopped Event ID	Pressed Event ID
Warrior 3D Player Object	Player 1 Moved Event	Player 1 Stopped Event	Player 1 Pressed Event
Thief 3D Player Object	Player 2 Moved Event	Player 2 Stopped Event	Player 2 Pressed Event
Mage 3D Player Object	Player 3 Moved Event	Player 3 Stopped Event	Player 3 Pressed Event
Cleric 3D Player Object	Player 4 Moved Event	Player 4 Stopped Event	Player 4 Pressed Event

Object ID	Released Event ID	Health	Max Health	Weapon Class	Armour Class
Warrior 3D Player Object	Player 1 Released Event	263	300	60	200
Thief 3D Player Object	Player 2 Released Event	−1	200	40	160
Mage 3D Player Object	Player 3 Released Event	180	180	30	80
Cleric 3D Player Object	Player 4 Released Event	189	190	20	50

Object ID	Used Inventory	Carried Inventory	Experience	Name
Warrior 3D Player Object	Longsword Object, Metal Shield Object	Lamp Object	33,000	Warrior Title
Thief 3D Player Object	Small Stick Object, Leather Gloves Object, Rags Object	Torch Object	57	Thief Title
Mage 3D Player Object	Book of Magic Spells Object, Magic Robes Object, Gold Potion Object, Green Potion Object	Elfstone Elessar Object	16,437	Mage Title
Cleric 3D Player Object	Book of Prayers Object, Cleric Robes Object	Lantern Object	100	Cleric Title

contain the **Secondary Events** the **Game Object** would receive, when the *Record* was loaded into or removed from the computer memory.

The *Record* would also contain the properties of the camera that would be used to display its view of the *Game World*. These would include the **Projection Target Field** or where this view would be displayed. This would either be the computer screen, or it could be a *Texture*, in the **Game Database**. In the latter case, this *Texture* could then be subsequently used to display another **Game Object**. This could be used to produce various visual effects. For example, it could be used to display a magic mirror that allowed a character, in one part of the *Game World*, to see another very distant or inaccessible part of the world.

The *Record* would also include the size of the visible area, around the position of the camera. And it would include the position of the projected view, from the camera, on the target, and the size of this projection.

You can see an example of this *Database Table* in Table 1.46.

1.4.43 3D Camera Objects Table

The next *Database Table* would hold **Point Object Records**. Each *Record* would hold the properties of a **3D Camera Object**. The *Record* would map an **Object ID** to the mass of a camera, its position and its acceleration in a 3D *Game World*. The

TABLE 1.46
Example of 2D Camera Objects Table

Object ID	Game Object Code	Mass	X	Y	X Speed
Warrior Orthographic Top View Camera Object	Warrior 2D Camera Code	1	1344	388	0
Thief Orthographic Side View Camera Object	Thief 2D Camera Code	1	320	240	0
Mage Orthographic Front View Camera Object	Mage 2D Camera Code	1	1634.420	2019.917	1.622
Cleric Isometric Camera Object	Cleric 2D Camera Code	1	2664.628	279.527	1.2153
Druid Chromatic Camera Object	Druid 2D Camera Code	1	1129.41366818	365.474630159	1.42660451321
Ranger 2D Audio Camera Object	Ranger 2D Camera Code	1	3151.98305585	261.582901361	0.518819832255
Necromancer Textual Camera Object	Necromancer 2D Camera Code	1	2490.87135131	34.9295704069	1.44158804266

(*Continued*)

TABLE 1.46 (*Continued*)
Example of 2D Camera Objects Table

Object ID	Y Speed	X Accel.	Y Accel.	Angular Position (Deg.)
Warrior Orthographic Top View Camera Object	0	0	0	291.864936205
Thief Orthographic Side View Camera Object	0	5	0	201.116385776
Mage Orthographic Front View Camera Object	0	0.804	0	299.061115244
Cleric Isometric Camera Object	0	0.068	0	283.91891155
Druid Chromatic Camera Object	0	6.31769152035	0	282.229787675
Ranger 2D Audio Camera Object	0	6.22285183871	0	274.248664719
Necromancer Textual Camera Object	0	3.84289110902	0	352.331629614

Object ID	Angular Speed (Deg./Sec.)	Angular Accel. (Deg./Sec./Sec.)	Collision Boundary ID	Proximity Boundary ID
Warrior Orthographic Top View Camera Object	0	0.544338428923	Warrior 2D Camera Boundary	None
Thief Orthographic Side View Camera Object	0	4.7982233656	Thief 2D Camera Boundary	None
Mage Orthographic Front View Camera Object	0	4.35920519124	Mage 2D Camera Boundary	None
Cleric Isometric Camera Object	0	2.88530937492	Cleric 2D Camera Boundary	None
Druid Chromatic Camera Object	0	5.34491392282	Druid 2D Camera Boundary	None
Ranger 2D Audio Camera Object	0	1.65907473689	Ranger 2D Camera Boundary	None
Necromancer Textual Camera Object	0	9.44339322825	Necromancer 2D Camera Boundary	None

Object ID	Collision Event ID	Proximity Event ID	Object Initial Reset Event ID
Warrior Orthographic Top View Camera Object	Warrior 2D Camera Collision Event	None	Warrior 2D Camera Reset Event
Thief Orthographic Side View Camera Object	Thief 2D Camera Collision Event	None	Thief 2D Camera Reset Event
Mage Orthographic Front View Camera Object	Mage 2D Camera Collision Event	None	Mage 2D Camera Reset Event
Cleric Isometric Camera Object	Cleric 2D Camera Collision Event	None	Cleric 2D Camera Reset Event
Druid Chromatic Camera Object	Druid 2D Camera Collision Event	None	Druid 2D Camera Reset Event
Ranger 2D Audio Camera Object	Ranger 2D Camera Collision Event	None	Ranger 2D Camera Reset Event
Necromancer Textual Camera Object	Necromancer 2D Camera Collision Event	None	Necromancer 2D Camera Reset Event

(*Continued*)

TABLE 1.46 (*Continued*)
Example of 2D Camera Objects Table

Object ID	Object Destroyed Event ID	Projection Type	Projection Target
Warrior Orthographic Top View Camera Object	Warrior 2D Camera Destroyed Event	Orthographic	Screen
Thief Orthographic Side View Camera Object	Thief 2D Camera Destroyed Event	Orthographic	Screen
Mage Orthographic Front View Camera Object	Mage 2D Camera Destroyed Event	Orthographic	Screen
Cleric Isometric Camera Object	Cleric 2D Camera Destroyed Event	Isometric	Screen
Druid Chromatic Camera Object	Ranger 2D Camera Destroyed Event	Chromatic	Screen.
Ranger 2D Audio Camera Object	Ranger 2D Camera Destroyed Event	Audio	Screen
Necromancer Textual Camera Object	Necromancer 2D Camera Destroyed Event	Textual	Screen

Object ID	Projection X	Projection Y	Projection Width	Projection Height
Warrior Orthographic Top View Camera Object	512	384	1024	768
Thief Orthographic Side View Camera Object	512	384	1024	768
Mage Orthographic Front View Camera Object	400	300	800	600
Cleric Isometric Camera Object	400	300	800	600
Druid Chromatic Camera Object	512	384	1024	768
Ranger 2D Audio Camera Object	512	384	1024	768
Necromancer Textual Camera Object	400	300	800	600

Object ID	Left	Right	Top	Bottom
Warrior Orthographic Top View Camera Object	1244	1444	1	0
Thief Orthographic Side View Camera Object	220	420	1	0
Mage Orthographic Front View Camera Object	1534	1734	1	0
Cleric Isometric Camera Object	2564	2764	1	0
Druid Chromatic Camera Object	1029	1229	1	0
Ranger 2D Audio Camera Object	3051	3251	1	0
Necromancer Textual Camera Object	2390	25,901	1	0

Object ID	Near	Far
Warrior Orthographic Top View Camera Object	288	488
Thief Orthographic Side View Camera Object	140	340
Mage Orthographic Front View Camera Object	1919	2119
Cleric Isometric Camera Object	179	379
Druid Chromatic Camera Object	265	465
Ranger 2D Audio Camera Object	161	361
Necromancer Textual Camera Object	−66	134

(*Continued*)

TABLE 1.46 (*Continued*)
Example of 2D Camera Objects Table

Object ID	Device Group ID
Warrior Orthographic Top View Camera Object	Joystick1:192.168.0.1:Player1:PassP11234:UGxheWVyMTpQYXNzUDExMjM0Cg==
Thief Orthographic Side View Camera Object	Keyboard1:192.168.0.1:Player2:PassP21234:UGxheWVyMjpQYXNzUDIxMjM0Cg==
Mage Orthographic Front View Camera Object	Joystick2:192.168.0.2:Player3:PassP35678:UGxheWVyMzpQYXNzUDM1Njc4Cg==
Cleric Isometric Camera Object	Keyboard2:192.168.0.2:Player4:PassP45678:UGxheWVyNDpQYXNzUDQ1Njc4Cg==
Druid Chromatic Camera Object	Gamepad1:192.168.0.3:Player5:PassP591011:UGxheWVyNTpQYXNzUDU5MTAxMQo=
Ranger 2D Audio Camera Object	Gamepad2:192.168.0.4:Player6:UGxheWVyNjpQYXNzUDYxMjEzMTQK
Necromancer Textual Camera Object	Gamepad3:192.168.0.5:Player7:PassP7151617:UGxheWVyNzpQYXNzUDcxNTE2MTcK

Record would include the rotation of the camera, about its local X, Y and Z axes, the speed and the acceleration of the rotation about these axes. The *Record* would also include the *Collision boundary*, the *Proximity boundary*, the **Collision Event** and the **Proximity Event** of the camera. And the *Record* would include the **Object Initial Reset Event** and the **Object Destroyed Event** of the **Game Object**.

Along with the properties for the position and motion of the camera, the *Record* would include the properties for displaying its view of the *Game World*. These include a **Projection Target Field**, a *Field of View* angle, a *near and far focal length*, the position of the projected view, from the camera, on the target, and the size of the projection.

You can see an example of this *Database Table* in Table 1.47.

The *Records* of the cameras would be the last of the entities of the **Game Database**, required by the **Event-Database Architecture**. And these would be the last required to implement the *game design* of *LPmud*.

1.4.44 Database Checksum Table

The next *Database Table* would hold **Database Checksum Records**. These are a form of **Database Meta Data Records**. Each *Record* would hold the *Primary Key* of another *Database Record*, and the *Checksum* or sum of the values in the *Database Fields* of that *Record*. This would be used to check when the contents of that *Record* had become corrupted. Either when that *Record* was transferred between storage media. Or when that *Record* was transferred from one computer to another in a computer network. Either between a **Game Server** and a **Game Client** in a **Client Server Network Architecture**. Or between two **Game Peers** in a **Peer-To-Peer Network Architecture**.

TABLE 1.47
Example of 3D Camera Objects Table

Object ID	Game Object Code	Mass	X	Y	Z	X Speed
Warrior 3D Camera Object	Warrior 3D Camera Code	1	258.608675331	7	3375.12464205	0.43
Thief 3D Camera Object	Thief 3D Camera Code	1	3948.21039433	0	2567.08262423	4.98
Mage 3D Camera Object	Mage 3D Camera Code	1	3678.34595975	0	1952.58023527	7.07
Cleric 3D Camera Object	Cleric 3D Camera Code	1	2241.58852606	0	2034.2849705	0.15
Druid 3D Chromatic Camera Object	Druid 3D Camera Code	1	284.902927482	0	3299.95912266	9.77428673025
Ranger 3D Audio Camera Object	Ranger 3D Camera Code	1	1847.8893691	0	3313.58762813	8.0994062359
Necromancer 3D Textual Camera Object	Necromancer 3D Camera Code	1	2869.74162608	0	1399.45354278	5.31980726023
Portal Camera Object	Portal Camera Code	1	1075.372	9	3649.974	0

Object ID	Y Speed	Z Speed	X Accel.
Warrior 3D Camera Object	0	4.36	6.48002571176
Thief 3D Camera Object	0	0.29	1.60247478408
Mage 3D Camera Object	0	3.72	6.18203033847
Cleric 3D Camera Object	0	7.88	1.99297510123
Druid 3D Chromatic Camera Object	0	4.86	4.20695534552
Ranger 3D Audio Camera Object	0	6.96	0.429883589824
Necromancer 3D Textual Camera Object	0	0.83	9.53908970233
Portal Camera Object	0	0	0

Object ID	Y Accel.	Z Accel.	X Angular Position (Deg.)
Warrior 3D Camera Object	0	3.71498104589	0
Thief 3D Camera Object	0	3.58360501872	0
Mage 3D Camera Object	0	8.09118922349	0
Cleric 3D Camera Object	0	0.217988620985	0
Druid 3D Chromatic Camera Object	0	9.53908970233	15.6908970683
Ranger 3D Audio Camera Object	0	7.39615760879	4.1161930811
Necromancer 3D Textual Camera Object	0	5.10333023369	24.1905458896
Portal Camera Object	0	0	90

(*Continued*)

TABLE 1.47 (*Continued*)
Example of 3D Camera Objects Table

Object ID	Y Angular Position (Deg.)	Z Angular Position (Deg.)	X Angular Speed (Deg./Sec.)	Y Angular Speed (Deg./Sec.)	Z Angular Speed (Deg./Sec.)	X Angular Accel. (Deg./Sec./Sec.)
Warrior 3D Camera Object	351.522036749	0	0	2.37422301443	0	0
Thief 3D Camera Object	155.461956192	0	0	3.7537878277	0	0
Mage 3D Camera Object	333.835465487	0	0	1.10940071413	0	0
Cleric 3D Camera Object	83.2213913219	0	0	4.06993614613	0	0
Druid 3D Chromatic Camera Object	224.519865815	0	0	2.7532137655	0	0
Ranger 3D Audio Camera Object	357.858153487	0	0	4.4277127777	0	0
Necromancer 3D Textual Camera Object	302.861467273	0	0	4.55633690607	0	0
Portal Camera Object	0	0	0	0	0	0

Object ID	Y Angular Accel. (Deg./Sec./Sec.)	Z Angular Accel. (Deg./Sec./Sec.)	Collision Boundary ID	Proximity Boundary ID
Warrior 3D Camera Object	3.5507239459	0	None.	Warrior View Frustum Boundary
Thief 3D Camera Object	4.24752559143	0	None	Thief View Frustum Boundary
Mage 3D Camera Object	2.46449710227	0	None	Mage View Frustum Boundary
Cleric 3D Camera Object	1.82618534137	0	None	Cleric View Frustum Boundary
Druid 3D Chromatic Camera Object	0.336852112673	0	None	Druid View Frustum Boundary
Ranger 3D Audio Camera Object	3.13486986431	0	None	Ranger View Frustum Boundary
Necromancer 3D Textual Camera Object	1.87158989874	0	None	Necromancer View Frustum Boundary
Portal Camera Object	0	0	None	Portal View Frustum Boundary

(*Continued*)

TABLE 1.47 (*Continued*)
Example of 3D Camera Objects Table

Object ID	Collision Event ID	Proximity Event ID	Object Initial Reset Event ID
Warrior 3D Camera Object	None	Enter Warrior 3D Camera Event	Warrior 3D Camera Reset Event
Thief 3D Camera Object	None	Enter Thief 3D Camera Event	Thief 3D Camera Reset Event
Mage 3D Camera Object	None	Enter Mage 3D Camera Event	Mage 3D Camera Reset Event
Cleric 3D Camera Object	None	Enter Cleric 3D Camera Event	Cleric 3D Camera Reset Event
Druid 3D Chromatic Camera Object	None	Druid Cleric 3D Camera Event	Druid 3D Camera Reset Event
Ranger 3D Audio Camera Object	None	Enter Ranger 3D Camera Event	Ranger 3D Camera Reset Event
Necromancer 3D Textual Camera Object	None	Enter Necromancer 3D Camera Event	Necromancer 3D Camera Reset Event
Portal Camera Object	None	Enter Portal 3D Camera Event	Portal 3D Camera Reset Event

Object ID	Object Destroyed Event ID	Projection Type
Warrior 3D Camera Object	Warrior 3D Camera Destroyed Event	Perspective
Thief 3D Camera Object	Thief 3D Camera Destroyed Event	Perspective
Mage 3D Camera Object	Mage 3D Camera Destroyed Event	Perspective
Cleric 3D Camera Object	Cleric 3D Camera Destroyed Event	Perspective
Druid 3D Chromatic Camera Object	Druid 3D Camera Destroyed Event	Chromatic
Ranger 3D Audio Camera Object	Ranger 3D Camera Destroyed Event	Audio
Necromancer 3D Textual Camera Object	Necromancer 3D Camera Destroyed Event	Textual
Portal Camera Object	Portal 3D Camera Destroyed Event	Perspective

Object ID	Projection Target	Projection X	Projection Y	Projection Width
Warrior 3D Camera Object	Screen	512	384	1024
Thief 3D Camera Object	Screen	512	384	1024
Mage 3D Camera Object	Screen	400	300	800
Cleric 3D Camera Object	Screen	400	300	800
Druid 3D Chromatic Camera Object	Screen	512	384	1024
Ranger 3D Audio Camera Object	Screen	512	384	1024
Necromancer 3D Textual Camera Object	Screen	512	384	1024
Portal Camera Object	Portal Texture	256	256	512

(*Continued*)

TABLE 1.47 (*Continued*)
Example of 3D Camera Objects Table

Object ID	Projection Height	Field Of View (Deg.)	Near Focal Length	Far Focal Length
Warrior 3D Camera Object	768	60	1	100
Thief 3D Camera Object	768	60	1	100
Mage 3D Camera Object	600	60	1	100
Cleric 3D Camera Object	600	60	1	100
Druid 3D Chromatic Camera Object	768	60	1	100
Ranger 3D Audio Camera Object	768	60	1	100
Necromancer 3D Textual Camera Object	768	60	1	100
Portal Camera Object	512	60	1	100

Object ID	Device Group ID
Warrior 3D Camera Object	Joystick1:192.168.0.1:Player1:PassP11234:UGxheWVyMTpQYXNzUDExMjM0Cg==
Thief 3D Camera Object	Keyboard1:192.168.0.1:Player2:PassP21234:UGxheWVyMjpQYXNzUDIxMjM0Cg==
Mage 3D Camera Object	Joystick2:192.168.0.2:Player3:PassP35678:UGxheWVyMzpQYXNzUDM1Njc4Cg==
Cleric 3D Camera Object	Keyboard2:192.168.0.2:Player4:PassP45678:UGxheWVyNDpQYXNzUDQ1Njc4Cg==
Druid 3D Chromatic Camera Object	Gamepad1:192.168.0.3:Player5:PassP591011:UGxheWVyNTpQYXNzUDU5MTAxMQo=
Ranger 3D Audio Camera Object	Gamepad2:192.168.0.4:Player6:UGxheWVyNjpQYXNzUDYxMjEzMTQK
Necromancer 3D Textual Camera Object	Gamepad3:192.168.0.5:Player7:PassP7151617:UGxheWVyNzpQYXNzUDcxNTE2MTcK
Portal Camera Object	Empty

You can see an example of this *Database Table* in Table 1.48.

1.4.45 Database Tag Table

The next *Database Table* would hold **Database Tag Records**. These are another form of **Database Meta Data Records**. Each *Record* would list the *Database Records* and *Database Fields* that were tagged with special properties.

For example, when playing a multiplayer game on a computer network, on several computers or **Game Peers**, in a **Peer-To-Peer Network Architecture**, these *Records* would list the other *Database Records* or *Database Fields* that should or should not be replicated on each **Peer** across the network.

Another example is when saving or writing the current state of a game to a file, one of these *Records* or **SAVE GAME LIST RECORD** would list all the other *Records* whose contents should be saved to that file. And this list would be used

TABLE 1.48
Example of a Database Checksum Table

Checksum ID	Object ID	Checksum
Warrior 2D Player Object Checksum	Warrior 2D Player Object	46
Thief 2D Player Object Checksum	Thief 2D Player Object	41
Mage 2D Player Object Checksum	Mage 2D Player Object	84
Cleric 2D Player Object Checksum	Cleric 2D Player Object	43

when loading or reading back the old state of the game, from that file to the other *Records* whose contents were used to save the state.

You can see an example of this *Database Table* in Table 1.49.

1.4.46 Database Monitor Table

The next *Database Table* would hold **Database Monitor Records**. Each *Record* would hold the *Primary Key* of another *Record* whose changes would be monitored, and a **Database Log Record** that would contain these changes. This information would be used to run tests and monitor when errors entered the *Database*.

You can see an example of this *Database Table* in Table 1.50.

1.4.47 Database Log Table

The next *Database Table* would hold **Database Log Records**. Each *Record* would hold the times a monitored *Database Record* was modified, the *Fields* that were modified at each time and the old values of the *Fields* before these were modified.

You can see an example of this *Database Table* in Table 1.51.

TABLE 1.49
Example of a Database Tag Table

List ID	Replicated	Fields
Replication Tag List	Warrior's 2D Player Object, Thief's 2D Player Object, Mage 2D Player Object, Cleric 2D Player Object	X, Y, Angular Position

List ID	Non-Replicated	Fields
Non-Replication Tag List	Warrior's 2D Player Object, Thief's 2D Player Object, Mage 2D Player Object, Cleric 2D Player Object	X Speed, Y Speed, X Accel., Y Accel., Angular Speed, Angular Accel

List ID	Saved	Fields
Saved Game List	Warrior's 2D Player Object, Thief's 2D Player Object, Mage 2D Player Object, Cleric 2D Player Object	X, Y, Angular Position, Experience, Used Inventory, Carried Inventory, Health

TABLE 1.50
Example of a Database Monitor Table

Monitor ID	Monitored	Logs
Player Monitor	Warrior's 2D Player Object, Thief's 2D Player Object, Mage 2D Player Object, Cleric 2D Player Object	Warrior's 2D Player Log, Thief's 2D Player Log, Mage 2D Player Log, Cleric 2D Player Log

1.4.48 Visualising the Database

Unlike a typical hierarchical *Database* developed in a *Software Evolution Process* used in the Computer Games industry, which is so complex it cannot be visualised, the **Game Database** of the **Event-Database Production Process** can be visualised. This benefit alone, to the communication of the staff involved in the production process, notwithstanding any other advantages the Event-Database Architecture has, makes it unique and priceless. This visualisation can be done through an *Entity-Relationship Diagram*.[35] This would show all the different types of items or entities in the *Database*, and the relationships which each one shared with another. And this, in turn, would give any of the staff, especially the *Database Administrator*, an opportunity to assess the language they were using in the production process. They could assess, for example, any ambiguities or inconsistencies in this language. And they could assess whether some of the items should be stored in the *Database* at all; that is, whether some of the words should be included in the language. These would be the names of categories of items or entities which shared no relationship with any other entity.

The keywords of the language of the **Event-Database Production Process** are the names given to entities in the **Game Database**. If the keywords were deficient in some way, then this would result in confusing *Entity-Relationship Diagrams* which did not make any sense. If the keywords were too long, then this would result in diagrams which cannot contain those words. If the keywords were too short and ambiguous, then this would result in diagrams which were also ambiguous.

You can see the Entity-Relationship Diagram for the **Events** in the **Game Database** in Figure 1.32.

You can see the Entity-Relationship Diagram for the **Text Objects** in Figure 1.33.

TABLE 1.51
Example of a Database Log Table

Log ID	Modification Times	Modified Fields	Modified Values
Warrior's 2D Player Object Log	359.15, 359.15, 465.59, 465.59, 493.46, 493.46	X,Y, X,Y, X,Y	250.335757441, 154.085106948, 141.789456083, 66.5226632345, 296.795780459, 183.006590054

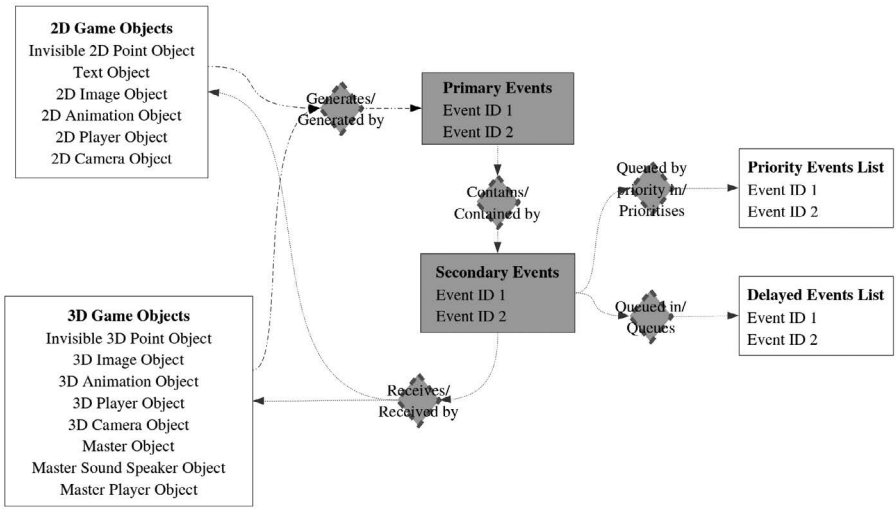

FIGURE 1.32 *Entity-Relationship Diagram* of Primary and Secondary Events of *LPmud*.

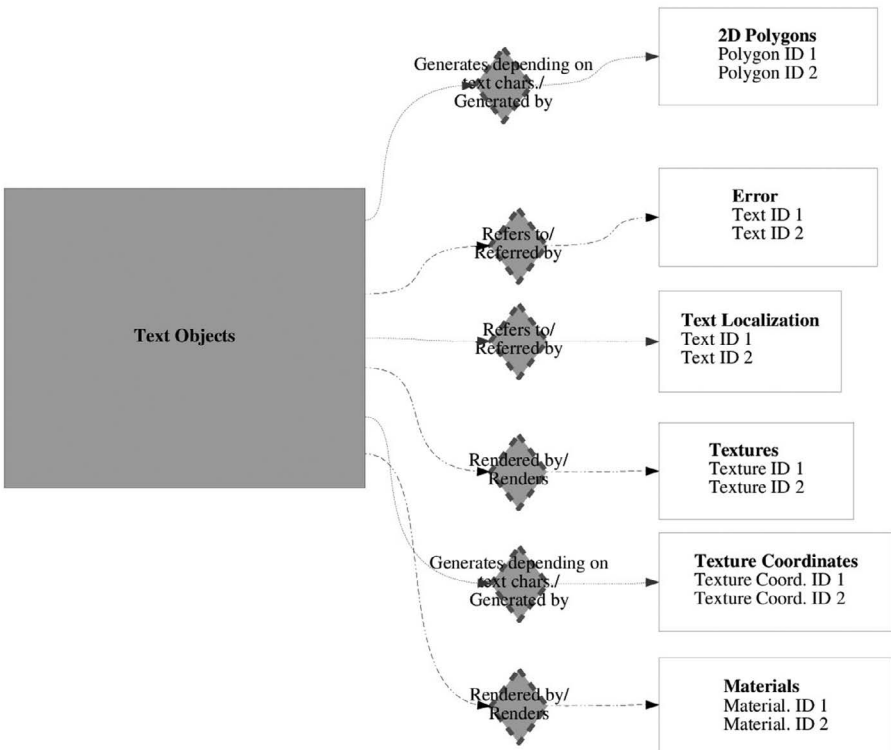

FIGURE 1.33 *Entity-Relationship Diagram* of Text Objects of *LPmud*.

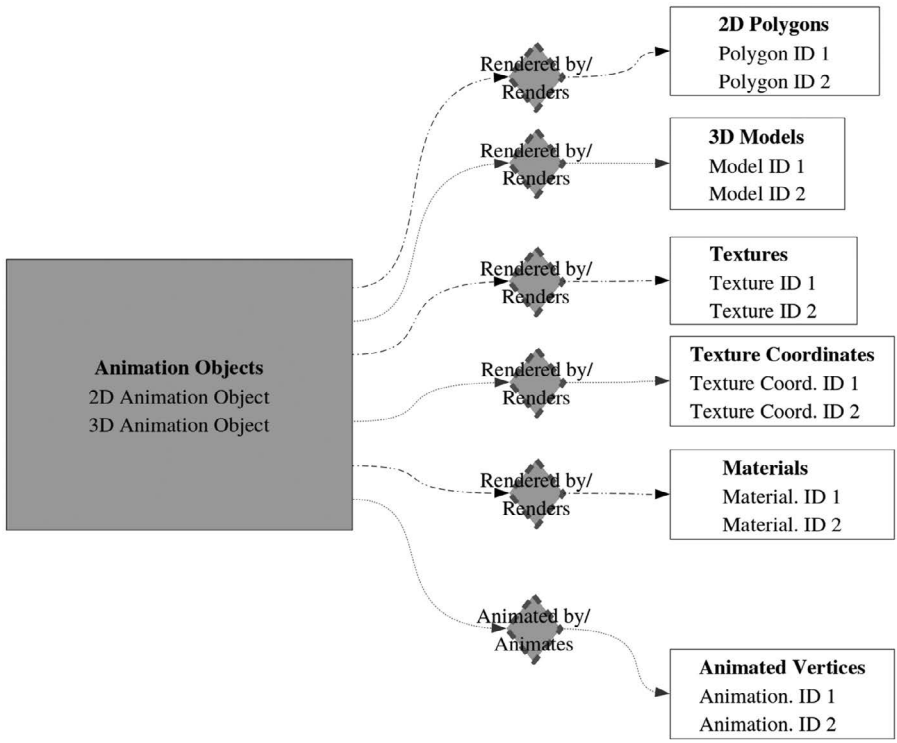

FIGURE 1.34 *Entity-Relationship Diagram* of Animation Objects of *LPmud*.

You can see the Entity-Relationship Diagram for the **Animation Objects** in Figure 1.34.

You can see the Entity Relationship Diagram for the **2D Game Objects** in Figure 1.35.

You can see the Entity-Relationship Diagram for the **3D Game Objects** in Figure 1.36.

You can see the Entity-Relationship Diagram for **Game Object Attributes** in Figure 1.37.

You can see the Entity-Relationship Diagram for other queues in Figure 1.38

You can see the Entity-Relationship Diagram for the queues of **Game Objects** in Figure 1.39.

You can see the Entity-Relationship Diagram for **Sound Microphones**, *Sound Streams* and **Game Objects** in Figure 1.40.

You can see the Legend for the symbols in all these diagrams in Figure 1.41.

These diagrams are very useful. For example, you can see from these diagrams that **Game Object Attributes** are ambiguous and need a better definition. All other entities in the **Game Database** occur in two or more diagrams. But the Game Object Attributes occurs only in one diagram. For although these were mentioned all the way back in **Chapter 3.3 Objects Host** in the book

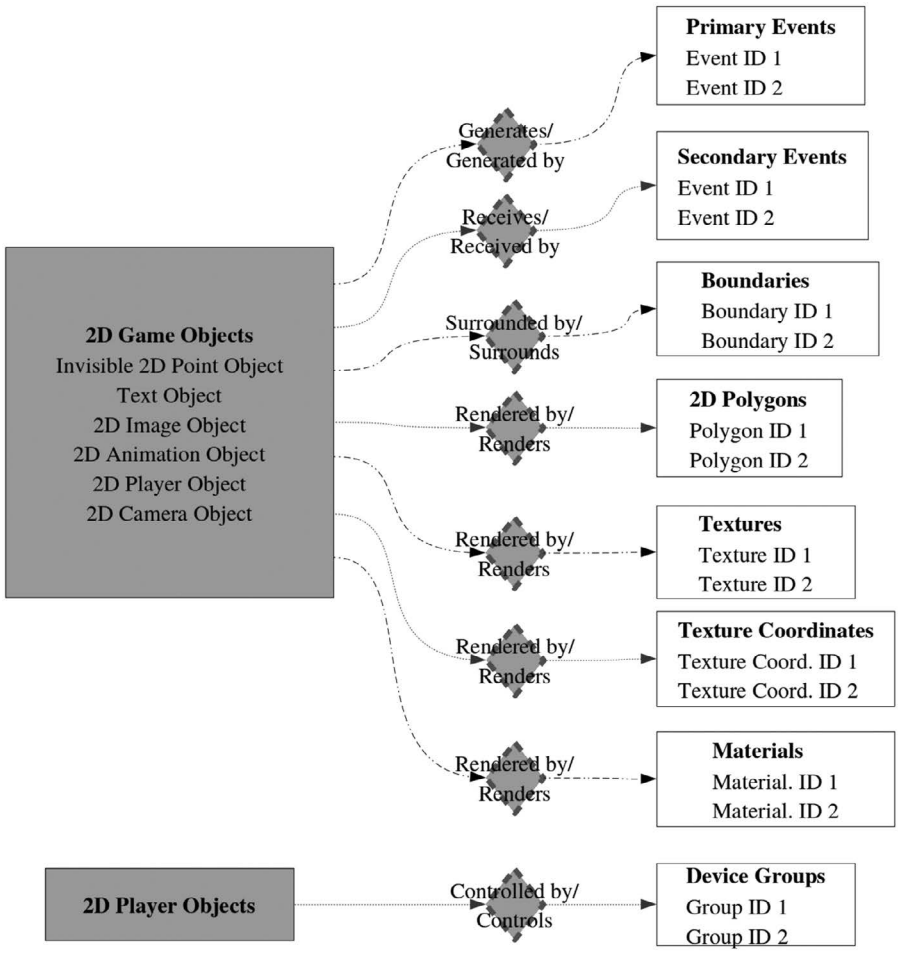

FIGURE 1.35 *Entity-Relationship Diagram* of 2D Game Objects of *LPmud*.

Event-Database Architecture for Computer Games Volume 1, these have not been mentioned since. Game Object Attributes are basically properties or *Database Fields* of **Game Objects** which are put on queues. And used by a huge variety of **Game Objects** and **Host Modules**. From the **Master Object**, **Physics Host**, **Graphics Host**, **Events Host** to the **Objects Host**. Some of the members of this set are not obvious. Take for example, the **Game Object Code Fields**.

The queue of the **Objects List Record** keeps a list of all the **Game Objects** that should be loaded into computer memory by the **Objects Host**. Before the **Objects Host** executes the code in the **Game Object Code Field** of each of these **Game Objects** in response to a **Secondary Event**. So that queue of **Game Objects** is in fact a queue of **Game Object Code Fields**. Therefore that **Game Object Code Field** should be classed as a **Game Object Attribute**.

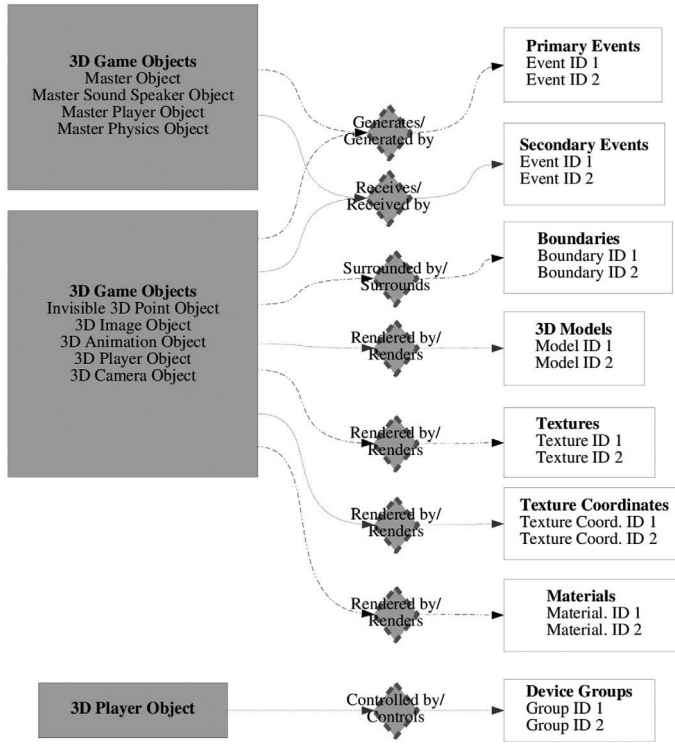

FIGURE 1.36 *Entity-Relationship Diagram* of 3D Game Objects of *LPmud*.

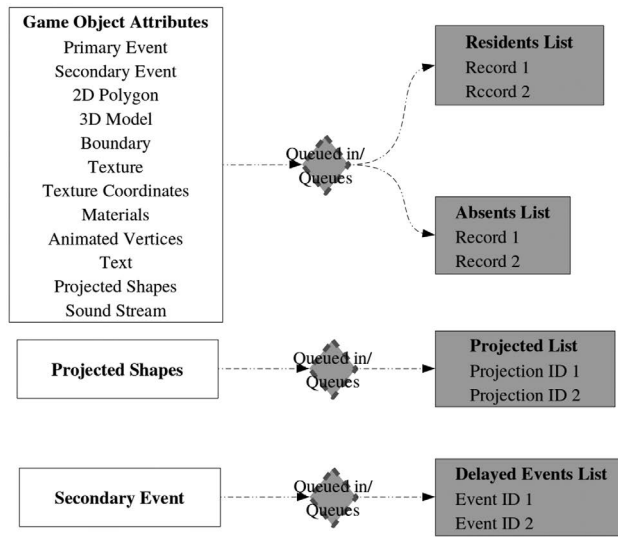

FIGURE 1.37 *Entity-Relationship Diagram* of queues of Game Object Attributes of *LPmud*.

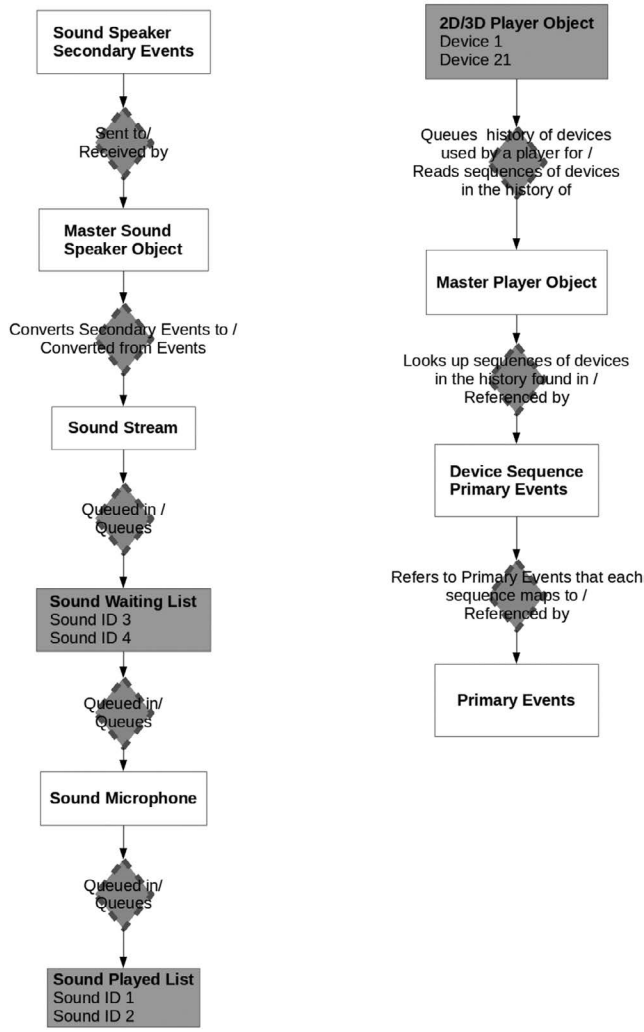

FIGURE 1.38 *Entity-Relationship Diagram* of conversions between queues and other entities in *LPmud*.

1.4.49 Enumerating the Language of the Production Process

Of course, the *game design* of *LPmud* is not complete. A complete *game design* would include an exact description of the *Game World*. This would include exactly what the composition of the locations in the *Game World* was going to be. And it would include the exact figures of the number of characters and other items in each location, where these would appear, whether these could be moved, how these would be moved, the sizes and the appearances of the locations, the characters and other items.

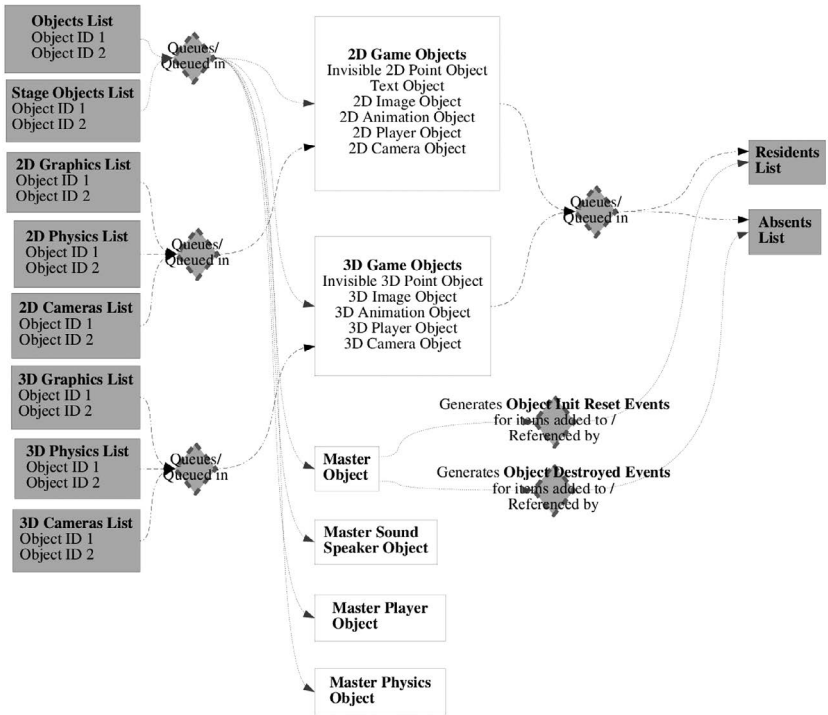

FIGURE 1.39 *Entity-Relationship Diagram* of queues of Game Objects of *LPmud*.

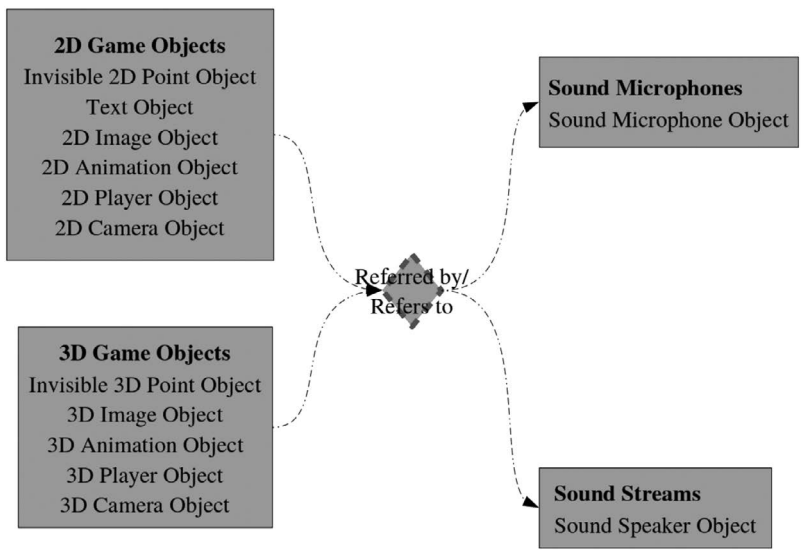

FIGURE 1.40 *Entity-Relationship Diagram* of Sound Microphones, Sound Streams, and Game Objects.

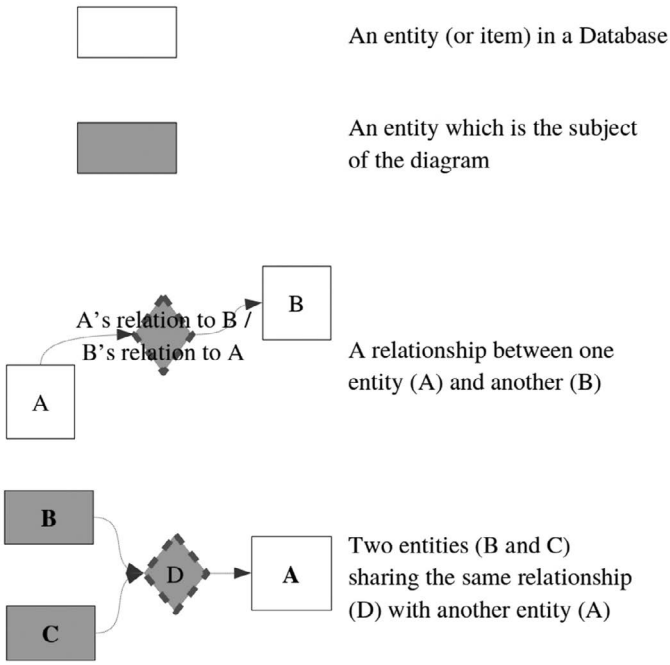

FIGURE 1.41 Legend of symbols used in the *Entity-Relationship Diagrams* and what these represent.

Similarly, a complete *game design* would include an exact description of the *User Interface*. This would include exactly what the composition of the menus, commands, groups of commands and combination of *analogue devices* and *digital devices*, in the *Interface* that executed the commands, was going to be. And it would include exact figures of the number of items in each menu, the commands in each group and the number of *devices* in each combination. It would include where these would be used in the game, how these would be used, whether the items on a menu could move, how these would move, the size and the appearance of these items.

Since the *game design* is not complete, the *technical design* and the *data design* for it are also not complete. The designs do not include the details of the **Game Object** of every unique group of items lying around, characters, creatures, buildings, other structures or locations in the *Game World*. Nor do the designs include the details of the **Game Object** of every unique group of items or menus in the *User Interface*.

Despite this, and the fact that this would be only the beginning of the production process, a lot of new words have already been introduced. In fact, over 200 words have been introduced, from a single source i.e. the **Event-Database Architecture**. Over 200 words that any of the staff would have to understand in order to engage in a discussion with others, about any aspect of the *software architecture*. This excludes the common terms and phrases that would be used, in the Software and Computer

Games industry, to describe the technology used in a production process. These new words were the following, in alphabetical order:

1. **2D Animation Object**
2. **2D Camera List**
3. **2D Camera Object ID**
4. **2D Camera Object Record**
5. **2D Image Object**
6. **2D Player Object Records**
7. **2D Player Objects**
8. **2D Polygon ID**
9. **2D Polygon Record**
10. **3D Animation Object**
11. **3D Camera List**
12. **3D Camera Object ID**
13. **3D Camera Object Record**
14. **3D Model ID**
15. **3D Model Object**
16. **3D Model Record**
17. **3D Player Object Records**
18. **3D Player Objects**
19. **Absents List Record**
20. **Actions**
21. **Analogue History Field**
22. **Analogue Positions Field**
23. **Animated Vertices**
24. **Animation ID**
25. **Armour Class**
26. **Audio Projection**
27. **Backwards Command**
28. **Camera List Record**
29. **Camera Object**
30. **Central Host**
31. **Chromatic Projection**
32. **Client Server Network Architecture**
33. **Controller Central Field**
34. **Controller Maximum Field**
35. **Controller Minimum Field**
36. **Controller Type Field**
37. **Database Checksum Records**
38. **Database Host**
39. **Database Log Record**
40. **Database Meta Data Records**
41. **Database Monitor Record**
42. **Database Tag Records**
43. **Deep Learning Model**

44. **Delayed 2D Physics List Record**
45. **Delayed 3D Physics List Record**
46. **Delayed Events List Record**
47. **Depth Coordinate**
48. **Device Group Field**
49. **Device Group Record**
50. **Device Mapping Field**
51. **Device Sequence Primary Events Record**
52. **Digital History Field**
53. **Digital Positions Field**
54. **Drop Command**
55. **Error Record**
56. **Escort Quest Handler Object**
57. **Event-Database Architecture**
58. **Event-Database Production Process**
59. **Events**
60. **Events History Record**
61. **Events Host**
62. **External Database Host Query Custom Tool**
63. **External Events Host Custom Tool**
64. **Find Quest Handler Object**
65. **Forwards Command**
66. **Game Clients**
67. **Game Controller Object Field**
68. **Game Controllers Host**
69. **Game Database**
70. **Game Object Attributes**
71. **Game Object Code Field**
72. **Game Object Records**
73. **Game Objects**
74. **Game Peer**
75. **Game Server**
76. **Game Time ID**
77. **Game Time Record**
78. **Get Command**
79. **Give Command**
80. **Graphic Object ID**
81. **Graphics Host**
82. **Graphics List Record**
83. **Graphics Object**
84. **Graphics Object Record**
85. **Heartbeat Event**
86. **Host Modules**
87. **Initial Reset Event Record**
88. **Internal Database Host Query Custom Tool**
89. **Internal Events Host Custom Tool**

90. **Inverse Kinematic Physics**
91. **Inverse Kinematics**
92. **Invisible 2D Point Object**
93. **Invisible 3D Point Object**
94. **Isometric Projection**
95. **Jump Down Command**
96. **Jump Up Command**
97. **Kill Command**
98. **Kill Quest Handler Object**
99. **Language Learning Model**
100. **List ID**
101. **List Record**
102. **Loaded**
103. **Loaded Event**
104. **Load Game Object**
105. **Look Command**
106. **LPC code**
107. **LPC Custom Tool**
108. **Master Object**
109. **Master Physics Object**
110. **Master Physics Object Record**
111. **Master Player Object**
112. **Master Player Object Record**
113. **Master Sound Speaker Object**
114. **Master Sound Speaker Object Record**
115. **Microphone Offset X Field**
116. **Microphone Offset Y Field**
117. **Microphone Offset Z Field**
118. **Moved Event**
119. **Multi-user Distributed Form**
120. **Neural Network**
121. **Neural Network Activation Function**
122. **Neural Network Back Propagation**
123. **Neural Network Back Propagation Adjust Weights nnnn Layer D Neuron xx Event**
124. **Neural Network Back Propagation Adjust Weights nnnn Layer Zyyyy Neuron xx Event**
125. **Neural Network Back Propagation Input Losses nnnn Layer D Neuron xx Event**
126. **Neural Network Back Propagation Input Losses nnnn Layer Zyyyy Neuron xx Event**
127. **Neural Network Back Propagation Output Losses nnnn Layer D Neuron xx Event**
128. **Neural Network Bias**
129. **Neural Network Final Outputs**
130. **Neural Network Forward Propagation**

131. **Neural Network Forward Propagation Inputs nnnn Layer D Neuron xx Event**
132. **Neural Network Forward Propagation Inputs nnnn Layer X Neuron xx Event**
133. **Neural Network Forward Propagation Inputs nnnn Layer Zyyyy Neuron xx Event**
134. **Neural Network Forward Propagation nnnn Fetch Metrics From Game World Event**
135. **Neural Network Forward Propagation nnnn Fetch Metrics From Training Data Event**
136. **Neural Network Forward Propagation Output nnnn Layer D Neuron xx Event**
137. **Neural Network Forward Propagation Output nnnn Layer Zyyyy Neuron xx Event**
138. **Neural Network Forward Propagation Translate Output nnnn Event**
139. **Neural Network Initial Inputs**
140. **Neural Network Neuron Input Weight**
141. **Neural Network Neuron Output**
142. **Neural Network Training Data**
143. **Object Attacked Event**
144. **Object Dead Event**
145. **Object Destroyed Event**
146. **Object Dropped Event**
147. **Object Entered Event**
148. **Object Exited Event**
149. **Object Heard Event**
150. **Object Heartbeat Event**
151. **Object ID**
152. **Object Initial Reset Event**
153. **Object Inventory Event**
154. **Object Looked Event**
155. **Object Moved Event**
156. **Object Pacified Event**
157. **Object Periodic Reset Event**
158. **Objects Failed List Record**
159. **Objects Failed Times List Record**
160. **Objects Host**
161. **Objects List Record**
162. **Object Taken Event**
163. **Object Unused Event**
164. **Object Used Event**
165. **Orthographic Projection**
166. **Owner Field**
167. **Peer-To-Peer Network Architecture**
168. **Periodic Reset Event**
169. **Perspective Projection**

170. **Physics Host**
171. **Physics Inverse Kinematics nnnn Bone yy xx Angle Arm To Reach Target Event**
172. **Physics Inverse Kinematics nnnn Bone yy xx Angle Leg To Reach Target Event**
173. **Physics List Record**
174. **Physics Object Record**
175. **Physics Ragdoll nnnn Bone yy xx First Pass Detect Forces On Bone Event**
176. **Physics Ragdoll nnnn Bone yy xx First Pass Generate Forces On Bone Event**
177. **Physics Ragdoll nnnn Bone yy xx Second Pass Detect Forces On Bone Event**
178. **Physics Ragdoll nnnn Bone yy xx Second Pass Generate Forces On Bone Event**
179. **Physics Ragdoll nnnn Bone yy xx Third Pass Resolve Forces On Bone Event**
180. **Physics Vortex nnnn Particle yyyy Collision Event**
181. **Physics Vortex nnnn Particle yyyy Spawn Event**
182. **Physics Vortex nnnn Particle yyyy Angular Acceleration Event**
183. **Point Object Record**
184. **Primary Collision Event**
185. **Primary Collision Event Record**
186. **Primary Connect Event**
187. **Primary Connect Event Record**
188. **Primary Controller Moved Event**
189. **Primary Controller Moved Event Record**
190. **Primary Controller Pressed Event**
191. **Primary Controller Pressed Event Record**
192. **Primary Controller Released Event**
193. **Primary Controller Released Event Record**
194. **Primary Controller Stopped Event**
195. **Primary Controller Stopped Event Record**
196. **Primary Disconnect Event**
197. **Primary Disconnect Event Record**
198. **Primary End Event**
199. **Primary Event Record**
200. **Primary Events**
201. **Primary Initial Reset Event**
202. **Primary Neural Network Back Propagation nnnn Event**
203. **Primary Neural Network Forward Propagation nnnn Event**
204. **Primary Physics Inverse Kinematics nnnn Event**
205. **Primary Physics Ragdoll nnnn Event**
206. **Primary Physics Vortex nnnn Acceleration Event**
207. **Primary Physics Vortex nnnn Spawn Event**
208. **Primary Projection Event**
209. **Primary Proximity Event**

210. **Primary Proximity Event Record**
211. **Primary Reflection Event**
212. **Primary Shutdown Event**
213. **Priority End Events Record**
214. **Priority Events List Record**
215. **Procedurally Generated Quest System**
216. **Projected List**
217. **Projected List Records**
218. **Projected Shapes**
219. **Projected Shapes Records**
220. **Projection ID**
221. **Projection Target Field**
222. **Quest Complete Event**
223. **Quest Giver Object**
224. **Quest Lost Object**
225. **Quest Marker Objects**
226. **Quest Prompt Object**
227. **Quest Receiver Object**
228. **Quest Spline Object**
229. **Quest Splines Complete Event**
230. **Quest Splines Generator Object**
231. **Quest Target Object**
232. **Quest Waypoints Object**
233. **Quit Command**
234. **Ragdoll Physics**
235. **Random Seed**
236. **Remove Command**
237. **Residents List Record**
238. **Resurrect Command**
239. **Saved**
240. **Save Game List Record**
241. **Save Game Object**
242. **Say Command**
243. **Secondary Connect Event**
244. **Secondary Connect Event Record**
245. **Secondary Controller Moved Event**
246. **Secondary Controller Moved Event Record**
247. **Secondary Controller Pressed Event**
248. **Secondary Controller Pressed Event Record**
249. **Secondary Controller Released Event**
250. **Secondary Controller Released Event Record**
251. **Secondary Controller Stopped Event**
252. **Secondary Controller Stopped Event Record**
253. **Secondary Disconnect Event**
254. **Secondary Disconnect Event Record**
255. **Secondary End Event**

256. **Secondary Event Record**
257. **Secondary Events**
258. **Secondary Reflection Event**
259. **Shout Command**
260. **Shutdown Event Record**
261. **Single User Monolithic Form**
262. **Single User Multi-threaded Form**
263. **Sound Microphone ID**
264. **Sound Microphone Object**
265. **Sound Microphone Object Record**
266. **Sound Object Field**
267. **Sound Radius**
268. **Sounds Host**
269. **Sound Speaker Object**
270. **Sound Speaker Secondary Events Record**
271. **Sounds Played List Record**
272. **Sound Stream ID**
273. **Sound Stream Records**
274. **Sounds Waiting List Record**
275. **Stage Objects List Records**
276. **Tell Command**
277. **Text ID**
278. **Text Localisation Record**
279. **Text Object**
280. **Textual Projection**
281. **Texture Coordinate ID**
282. **Texture Coordinate Record**
283. **Texture ID**
284. **Texture Record**
285. **Turn Left Command**
286. **Turn Right Command**
287. **Unloaded Event**
288. **Virtual Machine**
289. **Vortex Physics**
290. **Weapon Class**
291. **Wear Command**
292. **Wield Command**

Consider the academic fields where these words originate from i.e.

Mathematics
Computer Science
Electronic Engineering
Physics
Design (Illustration, Animation and Digital)
Audio Engineering

Game Design (Design, Mathematics and Computer Science)

These are studied by

Game Programmers (Mathematics and Computer Science)
Game Artists (Illustration, Animation and Digital Design)
Game Designers (Digital Design, Mathematics and Computer Science)
Sound Designers (Audio Engineering).

However, these are the academic fields whose words you would expect to be in the language of the production process but are absent:

Software engineering
Creative Writing
Project Management

Software engineering is the study of a systematic approach to the software production process, the planning, the implementation and testing, for *Quality Control* or *Quality Assurance*. You would expect the *Game Programmers*, *Game Testers* and *Game Producers* to have studied this subject.

Creative Writing is the study of creating narratives and stories. You would expect that there would be someone involved in the production who have studied this subject. Since this seems essential to building a *Game World* or a *game design*.

Project Management is the study of how to lead a project to achieve its goals within a set of constraints. The primary constraints being scope, time and budget. You would expect *Game Producers* to have studied this subject.

Creative Writers are not the only staff, in the Computer Games industry, whose words you would expect to be reflected in the language of the production process. But yet these words are absent. Given the academic fields whose words are, or you would expect to be reflected, in language of the production process, there are several staff who are normally absent in the Computer Games industry. These include

Pure Mathematicians
Pure Physicists
Pure Software Engineers
Electronic Engineers
Technical Authors
Fictional Authors

Instead of the staff being comprised of such professionals, typically in the Computer Games industry, in a staff of about 60 working on a project, only 15 would be *Game Programmers*, 40 would be *Game Artists*, 2 would be *Game Designers*, 2 would be *Game Testers* and 1 would be a *Game Producer.*

The representation of Mathematicians is very low amongst the staff. There are no Pure Mathematicians. And the closest comes in the form of the *Game Programmers* whose numbers are relatively low. Despite the fact the representation of the words

from Mathematics in the language of the production process is very high. Almost all of the entities in the **Game Database** just described have a relationship to geometry. Either through the physics, rendering or sounds of **Game Objects**, which all have geometry.

All of these observations, the disproportionately high number of words originating from Mathematics, the absence of words from the academic fields studied by *Game Producers* and *Game Testers*, the absence of words from *software engineering*, Creative Writing and Project Management, the absence of Pure Mathematicians, Physicists, Software Engineers, Electronic Engineers, Technical Authors, Fictional Authors amongst the staff, are all signs that there is something fundamentally wrong with the language. At least at the beginning of the software production process of the game *LPmud*. And if you assume that this language is typical in the Computer Games industry, then it would suggest that there is something fundamentally wrong with the language of production process of all games in the industry.

Nonetheless, these observations are only obvious in the **Event-Database Production Process** due to the **Event-Database Architecture**. The *data design* of the **Game Database** is the dictionary that makes it possible to make these observations.

However, in the *Software Evolution Process* normally used in the industry, none of these observations are obvious. In that process, the words of the language would not come from a single source. Instead, these words would come from an eclectic set of multiple sources. As has already been explained, these sources would include the different members of staff, their varied backgrounds and interests. And these sources would also include the *software architecture* of the process used. Remember that the process would only have two principles, in theory. Firstly, that it slowly evolves and grows software over time. And secondly, that the basis of this evolution was feedback from the software user.

But as time passed, the feedback from the software user would change, and thus would the *software architecture*. The components of the software would change depending on this feedback, and the relationship between these components would also change. And part of the words of the language of the overall production process would come from each of these transient *architectures*. And the eclecticism of these sources would be compounded by the lack of any single item that collated all the definitions of the words. No one *architecture*, no one member of staff, no one tool, nor any other single component of a *Software Evolution Process*, could be relied upon for the definition of even half of the words used in the process.

1.5 STEP 5: *LPmud* TOOLS DESIGN

The next step of the **Event-Database Process** is the *tools design*. You can see the vision for the tools in Figure 1.42.

The tools that would be used to build the game *LPmud* would be the same as the tools used to build the archetypal game based on the **Event-Database Architecture**. This has already been described in a book called

The Event-Database Architecture for Computer Games: Volume 1, Software Architecture and the Software Production Process.

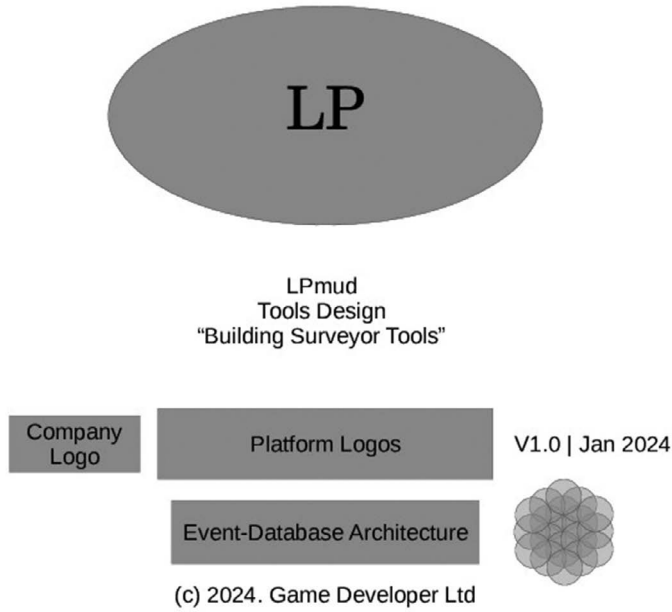

FIGURE 1.42 An example of a cover page for a *tools design* to build a computer game *LPmud*.

The tools have already been described in the subchapter in that book entitled

Step 5: Tools Design.

In addition to the tools described there, there would be other custom tools described in Table 1.52.

The **Internal Events Host Custom Tool** would be like the **External Events Host Custom Tool** used to build the archetypal game. But it would be built into the game from the **Game Objects** of the **Event-Database Architecture**. It would help test the

TABLE 1.52
Table of Custom Tools for *LPmud*

Name	Description	Staff	Copies
Internal Events Host Custom Tool	Manually fires **Primary** and **Secondary Events**	*Game Testers*	2
LPC Custom Tool	Adds locations, characters or items to the *Game World*	*Game Programmers* *Game Designers*	22
Internal Database Host Query Custom Tool	Scans the current location for the visible and invisible **Objects** in the *Game World*	*Game Programmers* *Game Designers* *Game Testers*	22

game at the end of the production process. This tool would only be available in the debug version of the game, to all the players, and only to Wizards in the final version of the game. By issuing a special command, at any stage of the game, through the *Game Controllers*, this item would appear in the inventory of the player's character. And by issuing a second command, the player could make that item disappear.

When it appeared, the tool would immediately present the player with a menu, listing the *IDs* or names of all the other visible items in that location. And the player would choose one name from the menu. After the player had chosen one of the names, the tool would present a second menu with the names of all the **Secondary Events**, which the **Game Object** of that item could receive. And when the player had chosen one of these names, the tool would send that **Secondary Event**, to that **Game Object**, to test the response. If the **Game Object** sent a **Primary Event**, in response, the tool would detect this from the Events History Record, and display a message informing the player of the name of that **Primary Event**. And it would inform the player of the **Secondary Events** following on from that **Event**. After a few seconds, the tool would display the first menu once again. So that the player could repeat the cycle, with the same **Secondary Event** or a different one. But once the player had had enough, and discarded the tool, all of its menus and messages would disappear from view.

Thus, by experimenting with this tool, the player could find out all the chain of **Events** in any location in the *Game World*. And ultimately, the player could uncover all the possible chain of **Events** in the game. This information could be used to test the *technical design* of *LPmud*, exhaustively, at the end of the production process.

The **LPC Custom Tool** has already been referred to, in the *game design* of *LPmud*. Namely, this required a tool that allowed the highest level players or Wizards to edit the *Game World*. This would be used to add new domains, once the players' scores had surpassed the highest level recognised by the game. This tool would be built into the game, from one or more **Game Objects**.

This tool would appear as another item, in the inventory of a player's character, once that character's score had reached the highest level. It would have no weight. Nor would it have any value in the shops and markets of the *Game World*. And therefore could not be bought or sold. And it would not be possible for the player to drop that item. As soon as the tool appeared, it would briefly display a message for a few seconds, congratulating the player for achieving the level of a Wizard. And it would inform the player that he or she could now add new locations to the *Game World* and edit these locations. The player would then be shown a second message, which would list the new commands that the player could use. The player could select any item on this menu, using the *Game Controllers*.

Next to each command would be a brief description of what that command did. At the bottom of this menu would be a command which the player could use to discard the menu and stop editing the *Game World*. However, the tool would not disappear from the player's inventory and player could bring back the menu, at any time in the future, by issuing a special command, through the *User Interface* of the tool.

There would be six options in all, on this menu, for editing the *Game World*. The first three would, respectively, add new locations, new characters or other new items to the *Game World*. This would include adding the new **Game Object** and **Game Object Record** for each new item. The number and type of *Fields* in the Record

would vary depending on the properties of the **Game Objec**t. But each *Record* would have at least the following *Database Fields*:

1. a *Primary Key*
2. a **Game Object Code Field**
3. an **OWNER FIELD**

The **Owner Field** would contain the name of the player who created it. The **Game Object Code Field** would contain pseudo machine code or **LPC code** translated or 'compiled' from words written in the *LPC* programming language, by the player, and entered using submenus presented by the tool. This programming language would be the same as the one used to write **Objects** in the *software architecture* of *LPmud*. The **Actions** for the **Secondary Events** received by this new **Object** would be executed by the **Objects Host** using the **LPC code** and a **Virtual Machine**. Like any other **Action** this would either edit *Database Records* or generate **Primary Events.** But when this **Machine** encountered any errors in the **LPC code**, the **Objects Host** would display the error on the screen and shut down the **Machine** without causing the game to be shut down. Each new location would be added adjacent to the player's current location. Provided, that is, either the player had created that location, or the player had not added any new locations, and there was space for the new location. If a new Wizard did not own any locations in the Game World, then they may ask another Wizard to add a new location, and transfer ownership to the new Wizard. It follows on from this, that the game must begin with at least one Wizard. And that the LPC Custom Tool must give you the option of transferring ownership of a location you create to another Wizard. Each new character added would be an NPC that would be controlled by the computer. This character, along with any other item, would be added to the player's current location, directly in front of the player's view.

The next three options, on the menu, would, respectively, edit locations, characters or other items already in the *Game World*. Each location edited would be the player's current location. Provided, that is, the player created that location. Likewise, each NPC, or other items edited, would be one in the player's location. It would also have to be one that the player had created.

Whenever the player chose any one of the six options on this menu, the player would be presented with further submenus. And the player would progressively define the properties of each new location, character or other items, through the options on these menus. These would include

1. size
2. colour
3. shape
4. health
5. size of inventory
6. money in inventory
7. other items in inventory
8. items being worn
9. items being wielded
10. items being used

11. items' weight
12. items' value in the shops
13. level of hostility or friendliness
14. **LPC Code**

In addition, through these menus, the player would define the areas within the *Game World*, through which each new character or other items would move; if that were possible. The player would select two or more locations, some of which may not have been created by the player. And the tool would set up the *Waypoints* between these locations that the character would follow. The character would then start to immediately move between these locations. But if the player wanted to, he or she could change this route. Or the player could remove that character entirely, through one of the options available on the menus of the tool.

The **Internal Database Host Query Custom Tool** has already been referred to in the *technical design*. In the description of the system of **Game Objects**, which would be used to build the *game design* of *LPmud*, there was a requirement for each **Object** to be placed in the *Game World*. Each visible and invisible **Game Object** had to be placed amongst the contents of the location, character or other items, within which it was used. So that a tool could be developed that would help debug any container, by revealing the identity of all the **Game Objects** within it. This would help any member of the staff, developing the game, identify all the software components that could be responsible for any errors evident in that container.

The tool itself would be made up of one or more **Game Objects**. It would only be available, in the debug version of the game, to all the players, and in the final version of the game only to the highest level players or Wizards. Once the player had issued a special command, through the *Game Controllers*, the tool would appear in the inventory of the player's character. Like the previous tool, it too would have no weight nor value. And it could not be dropped by the player.

The player would be presented with the list of the names, of all the visible items in that player's location, when the tool appeared. This would include the name of the location itself. And the player could select one of these names, using the *Game Controllers*. Once the player had selected one, the tool would present a list of the *IDs* of all the visible, and invisible items, contained within that item. If one of these contents in turn contained further items, these would not be displayed. Until, that is, the player selected the *ID* of this item, from the second menu. The tool would keep on displaying the contents of each item selected by the player, on subsequent menus, until one was selected which was empty. At which point, the tool would disappear from the inventory of the player's character, along with all of its menus from the player's view.

NOTES

1. ***LPmud.*** Lars Pensjö's Multi-User Dungeon. Any of a large class of multi-user adventure games built using the software architecture created by Lars Pensjö. See Glossary.
2. ***Quality Control.*** A system that accepts or rejects products or services depending on whether these meet all of the customer's specifications and requirements.

3. ***Software architecture.*** A description of a system for producing software. It includes a description of the components of the system, the relationship between these components and the principles that govern how these components change. See Glossary.
4. ***The Software Production Process.*** The steps for designing and implementing a game, using the **Event-Database Architecture**. See the chapter entitled **The Software Production Process** in the book ***The Event-Database Architecture for Computer Games: Volume 1, Software Architecture and the Software Production Process***.
5. ***Game World.*** An imaginary world space in which a game takes place.
6. ***Quality.*** The characteristic of a product which meets a customer's needs.
7. ***LPC.*** Lars Pensjö's C. A programming language, modelled on the language 'C', designed to allow you to modify the behaviour of items in the *Game World* of multi-user adventure games.
8. ***Artificial Intelligence.*** An attempt to model the human brain or to create a system that can make deductions. See Glossary.
9. ***Technical design sources.*** Game Programming Gems by Mark De Laura.
10. ***Incomplete game design.*** A description of a game which consist of just enough highlights to sell a project to its financial backer, but not enough detail to implement it.
11. ***UML.*** Unified Modelling Language. A language for describing the software components of a computer system.
12. ***Failsafe.*** A computer software or hardware system that can continue operating despite the persistence of errors within it.
13. ***Abstraction.*** The simplification of a problem by concentrating on essential aspects and ignoring the rest.
14. ***Collision boundary.*** The area around a **Game Object** which would be used to determine its collision with other **Objects**.
15. ***Proximity boundary.*** The set area around a **Game Object** which would be used to determine when another **Object** was, or was not, in close proximity.
16. ***Field of View.*** The visible area in front of a camera.
17. ***Near and Far focal length.*** The closest and furthest distance of the visible area in front of a camera.
18. ***Waypoint.*** A point along a path.
19. ***Illusion of Intelligence sources.*** Programming Game AI by Example by Mat Buckland.
20. ***Artificial Neural Network.*** Computer software used to make intelligent decisions, whose design was inspired by the study of animal brains. It is made up of a network of very simple software processors, connected by one-way communication channels.
21. ***Neuron.*** A nerve cell adapted to conducting electrical impulses, in the human brain.
22. ***Artificial Neuron.*** A mathematical model of a biological neuron. Each as multiple numerical parameters or inputs and a single output. It has a mathematical formula called an Activation Function which takes the sum of the inputs and produces a single output. In theory, the inputs represent information from human senses such as taste, sight, hearing, smell and touch. In practice, the inputs are metrics or measurements gathered from a real or imaginary space, by humans or computers, which represent an aggregation of information from human senses.
23. ***Expensive Graphics Processors.*** Graphics Processors are made up of several specialised maths processors running in parallel. These were originally used to perform the calculations required to render 3D graphics to achieve Photorealism. But lately these have also proved ideal for performing the parallel calculations in Artificial Neurons, and propagating the results forwards and backwards through the layers of an Artificial Neural Network. The demand for Photorealism and Artificial Neural Networks has increased the price of these Processors to ridiculous levels. See Glossary.
24. ***Expensive erroneous Language Learning Models.*** Expensive large Language Learning Models, such as ChatGPT Web Server, still produce unforeseen, unfeasible or prohibited results from time to time. And they still cannot solve basic mathematical and logical problems. See Glossary.
25. ***Neural Network sources.*** Introduction to Neural Networks by Kevin Gurney.

26. ***Flaws of Photorealism.*** There are many flaws in the Photorealism of computer games which the Software Developers have to strive continuously to overcome. Despite advances in the Graphics Processors, and the Hardware Rendering processes these perform to try to achieve Photorealism. See Glossary.
27. ***Rendering Farms.*** A cluster or network of computers across which a software rendering process is distributed to produce Photorealistic images. Typically for film or TV industries. Any company that offers these computers as a service is also called a Rendering Farm.
28. ***Data design.*** A description of all the data needed by a game. It is also a description of all the data produced by the tools used to build a game.
29. ***Unit of game time.*** The assumed minimum time between successive updates of a **Host Module**. The real time may exceed this limit, because the total time it takes to update all of the modules may be too long.
30. ***RGBA.*** A data format for describing the colour of a pixel by four values, for its Red, Green, Blue and a special Alpha component. The last of these controls how it blends with the colour of any underlying image.
31. ***Graphic Shaders.*** Machine code, which is executed during the Hardware Rendering process of a Graphics Processor, which controls how a surface or vertices of a polygon is rendered on the computer screen or in a Texture.
32. ***Vertex Shader.*** A Graphic Shader that is used to perform the projection of the vertices of the polygons of 2D images or 3D models, through a camera, into Normalized space (an area which is $1 \times 1 \times 1$) and then onto screen space (i.e. the computer screen). And it is used to set the amount of lighting at each vertex.
33. ***Geometry Shader.*** A Graphic Shader that is optional. It is used to take either the 2D or 3D primitives from the Vertex Shader and produce another primitive, adding or removing vertices. Or for rendering multiple images of the same primitive, at once, to the same target (i.e. computer screen or Texture). Or for feeding back information about the vertices of the primitives produced by the Vertex Shader, to later steps.
34. ***Fragment Shader.*** A Graphic Shader that is optional. It parses the pixels of the Textures of the polygons of 2D images or 3D models, after Rasterization. And it can change the depth and colour of the pixels depending on some kind of formula. And it can also discard pixels and stop these being rendered dependent on another formula.
35. ***Entity-Relationship Diagram.*** A diagram which shows all the items (or entities) stored in a Relational Database, and the relationship between these items.

2 Consistent Data Design

In the Computer Games industry, it is commonplace for tools and *software modules* to be built with a good, consistent *Interface*; either *User Interface* or *Software Programming Interface*. At least, the benefits of a consistent *Interface*, with regard to simplicity and ease-of-use, is widely appreciated. Many *guidelines for building Interfaces*[1] stress this point. But the consistency of how **Game Database** is designed and the *Interface* with that **Game Database** is not equally appreciated. The causes of this are both cultural and technical.

One of these causes has already been outlined in the description of the problem the **Event-Database Architecture** addresses. That is, an *incomplete game design* can lead to an inconsistent *data design*, as the *game design* changes over time. But, even when the *game design* is complete, there are still further problems with the tools and techniques used to implement games.

Another cause of an inconsistent *data design* is that the tools often used in the development environment are homogeneous. Any one person may have a unique set of tasks, and may use a unique set of tools to perform each task. But the same tool is used by everyone for the same task: be it creating computer graphics, sound effects, writing designs or *software modules* for the game. These tools are often proprietary, with *closed data formats*.[2] This fosters the notion that it is natural for *data formats* to be kept secret, despite the interoperability problems this causes. The closed data formats masks the inconsistency in the *data* produced for the **Game Database** by these proprietary tools. And the lack of interoperability of this *data* with other tools made by different *Software Developers*.

It is also common, in the Computer Games industry, to use a process called *Information Hiding*[3] to achieve the *Abstraction* (i.e. simplification) of software. The goal of *Information Hiding* is to hide all internal *data* a *software module* uses.

The effect of the common use of these two forms of obscuring *data* (i.e. *Closed data formats* and *Information Hiding*) is a general disregard for consistency in *data designs*. This is unfortunate because a consistent *data design* would naturally yield a consistent *Interface*, both in the form of a *User Interface* and *Software Programming Interface*, with little effort. If a *Database* were consistent and had *no redundancy*,[4] then to get or modify any given piece of information would involve exactly the same steps.

To access information in a *Database*, you would use the *Database Records* and *Database Fields*. A *Database*, which had *no redundancy*, would have no duplication in the information held in its *Fields* and *Records*. So accessing any given piece of information would involve using the same set of *Fields*, and *Records*, in the same order. Thus any software *Programming Interface*, or *User Interface*, which used this *Database* would use the same set of *Fields* and *Records* to access similar information. In addition to providing these *Interfaces* with a consistent set of processes, these *Fields* and *Records* would provide each *Interface* with a consistent terminology. This terminology would be in the form of the names and the descriptions, of

DOI: 10.1201/9781003502807-2

the *Fields* and *Records*. Each *Interface* could use this terminology to describe itself, or its processes.

A formal process for ensuring a *Database* has *no redundancy* will be described in the next chapter. Unfortunately creating a *Database*, with *no redundancy*, often means making one which does not have a consistent *data design* of its *Records*. A simple *Database* has only one type of *Record* or one *Database Table*. And all the *Records* have the same set of *Fields*. But the *Database* described in the **Event-Database Architecture** has different types of *Records* or *Database Tables*. So, apart from Primary Keys, it would be difficult to use a consistent set of *Fields* across the entire *Database*. Nevertheless, you could still maintain consistency in the way *data* was accessed from the different types of *Records* or *Database Tables*.

Take, for example, the *data design* for *LPmud* described earlier. Whenever faced with the option of adding a *Field* to a *Record*, to identify whether it had a particular property, the *Database* uses a **LIST RECORD** instead. This would be just a single *Record* that holds a list of the Primary Keys of all other *Records*, which had that property. This would save time searching through the *Database* for *Records* with the same property.

For example, instead of each **3D Model Object** having a *Field*, which indicated whether it was being displayed, there would be a **3D Graphics List Record**. This would hold the list of all **3D Model Objects** being displayed.

Similarly, instead of each sound used in the game having a *Field*, which indicated whether it was being played, there would be a **Sounds Playing List Record**. This would hold a list of all sounds being played.

Thus, suppose you wanted to group the *Records* in the *Database* by which stage (or level or part), of the *Game World*, these were used. Also suppose there were only a small number of stages. Instead of adding an extra *Field* to each *Record*, to indicate which stage it was used, you could have a **Stage List Record**. Each stage would have its own **List Record** in the same *Database Table*. And each of these **List Records** would contain the *Primary Keys* of all the *Records* which were used in that stage.

Such an addition to the *Database* would be useful, if you wanted to make use of limited space in the computer memory. Depending on which stage of the game a player was in, only the *Records* for that stage would be loaded into the computer memory. The **Database Host** would look up the *Record* for a stage (e.g. the main menus of the game) and load all the *Records* required for that into the memory, while unloading all other *Records*. The **Objects Host** could also look up this *Record* to decide which *game modules* (or **Game Objects**) should be loaded into the computer memory.

Another hypothetical example is if you wanted to give the player the option of saving the current progress of the player's character. Instead of adding an extra *Field* onto each *Record* that should be saved, you could use a **Save Game List Record**. This *Record* would keep a list of all other *Records*, which should be copied onto a computer file or a storage device, when the player saved the game. A *game module* (e.g. a **Save Game Object**) could then look up this *Record* when it was time to save a game. Another *module* (e.g. a **Load Game Object)** could also look up this *Record* to reload the progress of the player's character, back into the *Database*. And the player could continue where he or she left off. Note that because the **Event-Database Architecture** uses a *Database* with an *open-data format*, you can recover from any corrupt file

containing the saved properties of a player. You can edit the file with any Text Editor and remove or correct the corrupt lines in the file. But you cannot do this with **Game Database** of commercial *game-engines* which use a *closed data format*. Once a file has become corrupted that is it. And you can only detect the corruption when you attempt to load the file. And the detection involves detecting secondary symptoms of a corrupt file. These include Fields in the data whose size overflows the limits, and cause errors, as the data overflows into other spaces allocated by the game or other software the computer memory. Or detecting when when loading the data from the file takes a long time. But with the **Event-Database Architecture** you can detect the corruption in the file, before it has been loaded, using the **Database Checksum Records**, which saves you time. And you may be able to correct the corruption in some cases. The corruption of such files and the detecting of them is an important part of the process for getting games approved by the Game Console manufacturers.

When you add an extra *Field* on to a *Record*, to indicate that it should be used to save the player's character and current state of a game, you are in effect adding a form of *data* called a Flag. Avoiding the use of *Fields* with limited ranges (especially *Logic Flags*[5] or *State Switches*[6]) would be consistent with the principles of the **Event-Database Architecture**. A *Logic Flag* would be used to set (and test) when a condition had become true or false. A *State Switch* would be used to indicate which way a *game module* or **Game Object** should behave. A change in the state of a *Flag* (or *Switch*) would cause a *logic branch* or change in the flow of the *software code*[7] or **Action** of a **Game Object**. But the principles of the **Event-Database Architecture** only uses **Events** to control the external and internal flow of a game: not changes in *Database Fields*. This would be how it provides, the users of the system, the ability to combine any set of events in the game.

For this reason, the state of *digital devices* would not be stored in the **Game Database**. These have only two states: on or off. This would be just like a *Logic flag*. Instead a **Game Object**, which wanted to know the state of a *digital device*, would respond to the **Controller Pressed** or **Controller Released Events**. Once it received one **Event**, it would keep assuming that the state of *digital device* had not changed, until it received the opposite **Event**.

At which point, one of two things could subsequently happen. Either the **Game Object** would wait for another occurrence of the **Event** it just responded to. Or, if the **Object** was meant to change its behaviour, it would start a new chain of **Events**. The same **Game Object** could respond to this chain. Or it could ignore these **Events**, and another **Game Object** would respond instead, to perform the new tasks.

Apart from *Logic Flags*, and *State Switches*, it would also be inconsistent, with the principles of the **Event-Database Architecture**, to include any ambiguous references to *data* in the **Game Database**. For example, it would be inconsistent to use *Fields* which do not contain unique identities for other *data* (such as *Data Offsets*[8]). An *Offset* would only identify the position of a subset of *data*, within a larger block of *data*: not the subset itself. Two different subsets of *data*, within two different blocks, could have the same *Offset*. And it would be impossible to identify either of the two using these *Offsets*.

The **Event-Database Architecture**, on the other hand, would rely on being able to identify all the distinct subsets of *data*, at all times. This would enable it to name

and define these subsets. And this, in turn, would provide the staff with a language for discussing or making inferences about any part of the software.

So, in keeping with its principles, any large block of *data*, in the **Game Database**, should always be composed into a *Database Record*. And the subsets of that block be identified by the *Database Fields* of that *Record*.

NOTES

1. ***Guidelines for building interfaces.*** Strive for consistency is the first principle in designing User Interfaces, as outlined by Ben Shneiderman in his guide. See Glossary.
2. ***Closed data format.*** The secret description of the layout of data in a Database, and how each data is used. This description is proprietary and only known to a very limited number of software applications.
3. ***Information Hiding.*** A technique commonly used in Object-Oriented Design of software, to protect one software module (or Object) from being erroneously accessed by another.
4. ***No Redundancy (in a Database).*** The process of removing duplication of information in a Database is called Normalisation.
5. ***Logic Flag.*** Data which is either set or cleared, when a condition that a software procedure uses to control its behaviour, changes e.g. when a task it is waiting for is complete.
6. ***State switch.*** Data which controls the way software behaves. It usually controls only one software module. It ensures that two modes of operation do not overlap. Or, it ensures that the modes follow each other in the correct sequence.
7. ***Software code.*** The list of programming language instructions that describe the procedures a computer must follow.
8. ***Data Offset.*** The index of a subset of data, within a Database. This would be in the form of a number, which represented the distance of the data, from some reference point, normally the start of that Database. This could be used to quickly search its contents or define its layout.

3 Optimising the Results

The **Event-Database Architecture** may not produce the optimum *software design* for all computer hardware, in terms of performance. The performance of software typically relates to its efficient use of computer software or hardware instructions, *data* storage capacity and *data* transfers. The better the performance, the faster it reacts to the software users, which helps them enjoy the experience. The best performance may even permit bonus features to be added to the software, which increases their enjoyment. The worse the performance, the less pleasant the experience may be. And the worst performance could even make it unfeasible to operate the software on some computer hardware.

For example, since the **Event-Database Architecture** uses a system of **Events**, this may be slower than traditional methods, on some computer hardware. Instead of directly using *software procedures* from other *software modules*, **Game Objects** would do so by going through the **Events Host**. Another example is the description of the **Game Database**.

A large *Database* would be a central part of the **Event-Database Architecture**. Instead of directly using the local *data* it had, each *software module* would access its *data* from the *Database*. This would take longer than if the *data* were part of that *module*, on some computer hardware.

As far as the **Architecture** would be concerned, a simple, well-defined *Database* would be sufficient. The problem of designing a *Database* with the best performance, which would be consistent, fast and small, is beyond the scope of the design of the **Architecture**. But the components of the **Architecture** do include a *Database Administrator* who could help address this issue.

Some would argue that a small *Database* should be a pre-requisite of the **Event-Database Architecture**. But it would be good practice to leave such an optimum *Database* till the very late stages of a project. Making a software optimal would require a lot of effort and need to take into account the entire scope of a project. It would sacrifice clarity for the sake of efficiency. Every addition made, after the *Database* was made optimal, would be hindered by the lack of clarity. And it would require repeating the whole process of making the *Database* optimal again to take into account the new additions.

When it comes to producing an optimum *data design*, with *no redundancy* in the **Game Database**, there are many written *Database design sources*[1] which could be referred to for help. These take into account experiences learned from other industries (outside of the Computer Games industry).

As well as reducing the size of the *Database*, you could also reduce the number of **Game Objects** to make the software more optimal. Instead of loading large groups of **Game Objects**, which were very similar to the computer memory, the **Objects Host** could load one copy for each group. This master copy would receive all the **Events** for **Objects** from a group. Each occurrence of an **Object** of that type, in the

DOI: 10.1201/9781003502807-3

Game World, would therefore need to use a different *ID* for the same event. So that, from the *ID* of the event received, each master **Game Object** would know which set of properties, and corresponding *Abstract data*, to use.

Even though the **Event-Database Architecture** has been designed for optimum communication, instead of speed and size, this should be kept in perspective. Just because a system is not optimum in one criterion does not mean it is the worst either. The speed of the performance of software is only one of many qualities that affect a software user's enjoyment. It is not even the most important. Before software can give its best performance, it has to give a basic performance. And software with few errors, and less performance, is better than software with many errors and better performance. With respect to its performance, there would be six qualities of the **Architecture** to bear in mind.

The first quality would be that the **Event-Database Architecture** has several precedents. Computer Games have been released, that use other systems of events, similar to one in the **Architecture**. In these other systems, the *software modules* have been used to send messages (i.e. *data*) to an intermediary when events occur during a game. And any other *module*, which was interested in these events, could register itself, with that intermediary, to receive that message. Games have also been released which used *Finite State Machines*.[2] These have required the software to have a well-defined set of states, and a system of events, similar to the one in the **Architecture**, to control the flow of software from one state to the next: from one mode of behaviour to another. Games using these systems have been produced and released on even the lower end of computer hardware or *platforms*. Notwithstanding that these were borne out of a *Software Evolution Process*, these systems only differed, from the one in the **Event-Database Architecture**, in two other major respects.

One is that each of the *modules*, written by the *Game Programmers*, controlled the system of events: not a central *Database*. The other is that these systems have not been part of a single coherent *software architecture*. As has already been mentioned, the evolutionary principles of the *Software Evolution Process* have resulted in games going through multiple transient *architectures*. Thus, the system of **Events** has become confused. Inevitably, these have produced Computer Games with multiple competing systems of **Events**, one system of **Events** for the *User Interface*, another different system for the physics, another different system for the *Artificial Intelligence*, another different system for communicating with Web Servers or other computers across a network, another different system for playing sounds or music. Or the systems of **Events** which have become mixed with other devices for controlling the flow of the game; including *Logic Flags* and *State Switches*.

The second quality of the **Event-Database Architecture**, to bear in mind, would be that it would be built mainly around a *Database*. So it could be built on some of the powerful *Desktop computers*,[3] used by the companies in the Computer Games industry. These have supported software which used far more resources than an **Event-Database Architecture** would. These have included corporate *Databases*, which held all the resources used to build multiple computer games and other products.

The third quality of the **Architecture**, to bear in mind, would be that it would be *scalable*.[4] So you could develop games on powerful *Desktop computers* and then scale them down to less powerful computer hardware. Since these offer more

computing resources (such as computer memory, or faster Central Processors), the powerful hardware would allow you to concentrate on implementing the features of the game. You would not have to worry about limited resources.

However, once the features were completed, or you wanted to show a client a demonstration, then you would concentrate on the limited resources of the target computer hardware or *platform*. You could use the powerful tools available on *Desktop computers* to analyse inefficiencies in the *software data* or *procedures*. You could use the tools to systematically convert these *data* and *procedures*, into a format which produced faster transfers of *data*. You could also use these tools to eliminate redundancy in these *data* or *procedures*.

This is not new! Between 1980 and 1989, *Desktop computers* were used to create games for less powerful *home computers*.[5] Each game would be developed on the *Desktop*, then it would be scaled down for the *home computers*. Even recent *Games industry commentators*[6] have noted the value of having a *Desktop* version of a game, though that may not be its final destination. Furthermore, systems, based on *Relational Databases*, have been scaled down onto *small devices*[7] with even fewer resources than those typically used to play Computer Games.

Compare this approach with the normal practice in the Computer Games industry. This attempts to make a system optimal while it is being developed. Any system, computer or human, which attempts two things at once is more complicated. And, because it is more complex, it is more prone to errors. It would be far better to tackle each task separately. It would be better to have one set of employees, or tools, handle implementing features. After the first set had finished, another set would handle converting the game from a powerful computer to a less powerful one.

Once the software gives a basic performance on a more powerful computer, the order may then be changed. That is to say, the number of components and the relationship between these may be changed to find the best performance on that computer. The order that produced the best performance would be the same order that would produce the best performance on a less powerful computer.

Although a more powerful computer may have been designed to be used by multiple software, simultaneously, the best performance of any of these would still remain when it operated alone. This would be the same as with a less powerful computer, which could only ever be used by one software at any one time. What may make it impossible to transfer software onto a less powerful computer would be an inability to transfer one of its components. But this could be overcome if all the components were *scalable components*. That is to say, each component had been designed to use a variable amount of resources on the computer it was originally built.

Therefore, the feasibility of transferring software, from a more powerful computer to a less powerful one, comes down to *scalable components*. If the software could not operate on a given computer, this would not be because it was originally built on a more powerful computer. This would be because the components had not been, or could not be, scaled up or down far enough.

Fortunately, the basic components of the **Event-Database Architecture** would all be *scalable components*. The **Events Host** may use a varying amount of **Primary** and **Secondary Events**. The **Database Host** and **Game Database** may use a varying amount of *Records*. The **Objects Host** may use a varying amount of **Game Objects**.

The same would apply to the **Physics Host** and the **Graphics Host**. The **Sounds Host** may play a varying amount of music or sounds. And the **Game Controllers Host** may interact with a varying amount of *Game Controllers* and **Game Objects**. The **Central Host** would only use one *Record*. So, barring the inability to scale up or down any of its components far enough, the **Architecture** would be feasible.

Once all the components had been scaled up or down far enough, any qualities that these had on a more powerful computer would be transferred down as well. So if a certain order of the components produced the best performance on a more powerful computer, then that same order would produce the best performance on a less powerful one as well. Efficient software may not necessarily produce *scalable* software. But *scalable* software would always be able to produce efficient software.

If necessary, the scalability of the **Architecture** may improve further by modifying the **Database Host**. In particular, its *Interface* may be extended to include *software procedures* that apply *Basic Set Theory* to the **Game Database**. So that other components could use these to query the *Database*, without transferring *data* from it.

For example, you could either find the subset of *Records* that had a *Field* matching some criteria from another set. Or you could get a new set of numbers by applying some mathematical formula to a numerical *Field*, in a given set of *Records*. Or you could produce a new set of words by applying another formula to an alphabetical *Field*, of another set of *Records*. Or you could find out whether a *Primary Key*, a number or a word, was amongst a set of *Primary Keys*, numbers or words listed in a *Field*. Or you could find the intersection of one set of *Primary Keys*, numbers or words with another set. Or you could find the complement of one set in another. Or you could find the union of one set with another using these new *procedures*.

All *software procedures*, that used the **Game Database**, would find such an extension to the *Interface* useful. Furthermore, these new *procedures* of the **Database Host** would be *scalable components*. So would any other *procedure* that used these. All the steps in any *software procedure* may be performed just using operations from *Basic Set Theory*. And you could divide such a *procedure* into as many parts as possible. You could also scale up or down the size of the sets of *data* used in each part.

Of course, before you scaled up or down any software component, you would need to be able to reorder the components so that you could find the order which produced the best performance. This leads to the fourth quality of the **Event-Database Architecture** that would help the production of optimum software. This would be that, by following it, the software system you constructed would be an *ordered software system*.[8] That is to say, there would be some principles directing the set of components of the game, the relationship between the components and, most importantly, the growth of this set. Thus, you could use these same principles to easily deconstruct the system and rearrange it in a new order.

For example, suppose the performance of a *software module* could be improved, on some computer hardware, by keeping it together with the other *modules* that it used, in the computer memory. With the **Event-Database Architecture**, the order in which **Game Objects** were loaded into the memory could be changed. This would be determined by the **Objects Host**. More precisely, this would be determined by the

order of the list of **Game Objects**, which it would load into the memory. And this list would be held in the **Objects List Record** in the **Game Database**.

So you could rearrange the **Game Objects** and group these by the frequency of the interactions between one and another. That is to say, each **Game Object** would be placed in a group, in the Objects List Record, adjacent to the other **Objects** it was most frequently used. And if a **Game Object** were used just as frequently with one group, as another, then both groups would have one copy of that **Game Object** placed adjacent in the list.

Another example would be if the speed of access to *data*, on some computer hardware, could be improved by storing *data* together that were frequently used in conjunction. These *data* may be accessed either from the computer memory or some storage media. In the **Event-Database Architecture**, the order in which *data* were stored could be rearranged. This would be determined by the order of the *Records*, on whatever storage media the **Game Database** was kept on. So you could rearrange these *Records*. You could group *Records* by the frequency these were used in conjunction. That is to say, each *Record* would be in the same group, and adjacent with the other *Records* it was most frequently used with. And if a *Record* were used just as often with one group, as another, then both groups would have one copy of this *Record*.

Compare this with the *software architecture* normally used to make computer games, in the *Software Evolution Process*. This process has only two, very broad principles. That is, the software should evolve slowly over time, and the basis of this evolution is feedback from the software user. These two principles direct the *software architectures* used at different points in time, during the process. The process itself is not subject, however, to the direction of any one *software architecture*. Its principles do not direct the composition of the software, or the growth of these components, but only the end result at any point in time. So as long as the changes to components meet the demands of the software user, the principles of the process have been met. It would not matter what these components were or how they were related to each other. The principles of the process have absolutely nothing to say, for example, about the composition of the very first version of the software. And it makes no difference what *software architecture* this uses.

On the other hand, this would matter to the **Event-Database Production Process**. Furthermore, although it encompasses the second principle of the *Software Evolution Process*, it also compensates for the growth of the software. That is, the software would be modified based on the feedback from the software user. But the growth of the software components would be directed by the **Event-Database Architecture**.

At the beginning, of a *Software Evolution Process*, the leaders of that project may add more components or principles to its default principles. These components may seem to affect the growth of the software. And these may include software tools that allow you to edit some elements of the *game design*, such as the *game editors*.

But, as has already been mentioned, these additions to the default principles would be more concerned with selling the project, to its financial backer, than how the software grows. The more it appears that a project could handle changes to a *game design*, the less risk it would seem to have to an investor. So a few crude, makeshift principles or tools would be added to the software project to give the appearance of a flexible system for software production.

However, during the project, these makeshift principle or tools would be used far too inconsistently to be effective. Either these makeshift principles or tools would be subject to the rush to meet the second principle, which would occur frequently since these additional principles or tools were only meant to be temporary. Or the leaders of the project would explicitly state that these additions were subject to their arbitrary decisions. And indeed, the *software architecture*, which the project ends up using, does not produce an *ordered software system*, at the end of it.

At the end of the project, it would not be possible to use the principles, or tools of this *software architecture*, to deconstruct the computer game. It would not be possible to use these principles or tools to predict all the errors that may arise from removing any software component. So you could not deconstruct and reconstruct the game into a more optimum form.

Thus, since the project would not be able to rely on prognosis, to produce a more optimum game, it would have to rely on diagnosis instead. It would have to rely on diagnostic tools that allowed you to probe the external behaviour of the software, at the end of production. It would have to rely on tools that allowed you to identify frequently used, slow *software procedures* or large pieces of *data*, during the testing of the game. And, once identified, the performance of these areas of the software system may be improved. However, this must be done without changing the order of the components of the system.

This method of improving the performance would be further flawed by how these tests were conducted when the *game design* was not complete. As these tests would not be exhaustive, because of the *incomplete game design*, the performance of the unfrequented areas of the game would be ignored. If the game the project produced was an *ordered software system*, you could identify the principles that caused the poor performance in one area. And this could be used to identify and improve the performance in other areas; even areas which may not be frequented during the tests.

Besides being blind to some areas of the game, which may be improved, the project would also be limited by the risk of improving the performance of the software. The slow *software procedures* and large pieces of *data*, discovered by the diagnostic tools it uses, may cut across several software components. However, since the project would produce a game that was not an *ordered software system*, no one would understand the relationship between its components. Therefore, anyone could end up improving the performance of the *software procedures* or *data*, in one set of components, and causing errors or poor performance in other components.

For the same reason, the project would be restricted from using the simplest method to improve the performance of a game. You could simply remove a feature from a game, or redundant components from a software system to improve its performance. By choosing to remove *software procedures* or *data* entirely from a game, you could not improve the performance of these pieces of software any better. But if these pieces were not part of an *ordered software system*, this removal would cause errors. Rebuilding the game with the remaining components may expose some errors, but it would obscure others. Without an *ordered software system*, you could not predict what errors may arise from removing a software component.

This leads to the fifth quality of an **Event-Database Architecture** to bear in mind. That is, you could predict what **Actions** would occur concurrently, in response

to an **Event**. You could do this simply by looking at the list of **Game Objects**, which responded to a **Primary Event**, in the **Game Database** or the **Secondary Events** which followed it in the **Events History Record**. And therefore make your system more optimal by combining several **Secondary Events** or **Actions** into one. You could get all the *data* that these **Secondary Events** or **Actions** required and use all of the *data* at once. You could also eliminate any duplication in **Secondary Events** or **Actions**.

Compare this with software systems which do not use **Events** and do not have a complete *game design*. Two *software modules* could be responding to the same event, in a game. But only a complete search through all the *modules* would yield a picture of where this was happening. And that is simply not practical in a large project.

The final quality of the **Event-Database Architecture**, to bear in mind, would be that you could spot redundancy in your *data*. Since all *data* would be stored in one, central *Database*, you could easily analyse it to see if pieces of information were being repeated. So you could eliminate this information and reduce the amount of *data* being transferred across the game.

Doubtless, there would be lots of **Primary Events**, which would be added to the **Game Database**, that may be subsequently not used (i.e. no **Game Object** would respond to). Nevertheless, the *Records* for these **Events** should be added to the *Database*. A description of each *Record* should also be added to the documentation of the *Database*.

The *Database* produced by this method would have a catalogue of both used and unused **Events**. So you could create a software tool which uses the *Database* to remove unused **Events** from the system. This could be achieved by searching for all the *Records*, of **Primary Events**, which had an empty list of **Secondary Events**. The search would be based on the assumption that any **Primary Event**, which was going to be used, would begin with at least one **Secondary Event** on its list. If that were true, then this search would produce a set of *Records* for unused **Events**. All reference to these **Events** could then be removed from the **Game Objects**. All the *Records*, belonging to these **Events**, could be removed from the *Database* as well.

This could be used to create a more optimum version of a game, such as a demonstration or a final version. However, the original version of the game would still maintain a set of all the **Events**, in its **Objects** and *Database*. Thus, it would ensure that the widest range of options were always available for making future changes to a *game design*.

The last two qualities of the **Event-Database Architecture** were examples of how you could make a game more optimum at the level of a *technical design*. This would be far more important than making a game more optimum at the lower level of the computer hardware. This would be how performance would normally be improved with a game which had to be optimal, while developing at the same time, such as in the *Software Evolution Process*.

This would be done by habitually looking for ways to more efficiently store even the smallest, most insignificant piece of *data*, on the computer hardware. And this would be achieved by always looking for ways to reduce the steps taken, by *software procedures*, on the hardware. This would be done no matter how short that *procedure* was or how often it was used.

But those who do this would be oblivious to how much these habits improve the performance of the software overall. And they would be oblivious to how these measures affect an *ordered software system*. Instead, they would merely be taking uncalculated risk, by removing seemingly redundant pieces of *data* and steps from *procedures*. All in the hope that, somehow, all of these small measures add up to some significant advantage in the performance, at the end of the process.

They would fail to realise that an improvement at the higher level, of the *technical design*, has the effect of simplifying the construction of the game at the same time, whereas an improvement at the lower level, of the computer hardware, would have the effect of complicating the construction at the same time. The latter method would rely on characteristics of the computer hardware which would be un-transferable, obscure and harder to maintain. It would play right into the hands of the school of engineers who love to make *software design 'so complicated that there are no obvious deficiencies'*.[9]

3.1 FORWARD ENGINEERS AND REVERSE ENGINEERS

The **Event-Database Architecture** has a context, from which it stems and to which it would be applied. This context is how to produce a Computer Game without a complete *game design*. But this context lies within a larger context, which extends far beyond the narrow shores of the Computer Games industry. This second context extends to the outer reaches of the Software industry. If you do not understand the tiny ripples, at the edges of software production, then you will drown in the tidal waves that will wash over any attempt you make to implement a game based on the **Event-Database Architecture**.

That is to say, if you do not understand the thoughts which give birth to the different views of the software production process, then this will impact the efficacy of the **Event-Database Architecture** or the **Event-Database Production Process**.

There are two schools of thought in software production. The first school believes that software production is a science. The second school believes it is just an art.

The first school believes that by carefully designing your software and writing it down, you can clarify your intent and ensure that your plan is comprehensive. If your plan has been well written, then it would make it easy to build and test the software. This is the science (or discipline).

The second school believes the entire production process is an art. They do not just believe the design of the software is an art, but the building and testing too: the software production process itself. Every instance of a production process is merely an example of the aesthetics of the art. Therefore, there can be no such thing as a complete production process, a complete design, a complete implementation or a complete test. A belief in such things is merely a futile attempt to limit expression. More precisely, they believe software production to be an art of minimalism.

The aesthetics (or beauty) of the art is not just the minimal number of steps of the production process but the minimalism of the internal structure of the software product itself. The minimal number of software instructions, minimal number of data transfers and consumption of minimal space of the computer memory or storage media make the software beautiful. Be it at the beginning, middle or end of a

production process, if the software lacks this internal structure it is not beautiful. And they conflate this minimalism with simplicity.

Placed at the disposal of the two schools are two forms of engineering: *Forward engineering*[10] and *Reverse engineering*.[11]

Forward engineering begins with higher-level tools and moves progressively forward towards lower-level tools. *Reverse engineering*, by contrast, begins with lower-level tools and moves backwards towards higher-level tools.

The higher-level tools include an analysis of the requirements, of a product, followed by the documentation of this analysis. These tools include the documentation of the lower-level tools needed to meet the requirements and build the product. The higher-level tools also include a study of the feasibility of this plan to build the product and the documentation of this study. The highest-level tool is natural language, which would be required to analyse and document the plan to build the product.

On the other hand, the lower-level tools include the components used to build the product and any custom tools required to build these components. The lower-level tools also include any third-party products used to build the components and any diagnostic tools required to test these in the final product, including physical observations. The lowest-level tools are the characteristics of the components, or other lower-level tools, used to build the product. These are the incidental qualities the lower-level tools or components have, which only become apparent after the product has been built, and turn out to be useful to improve its performance in certain areas. These qualities, however, would not have been apparent when the original plan to build the product was drafted. And these would have been too risky to predict and account for in that plan.

So, in *Forward engineering*, the lowest-level tools would only be used at the end of the process, after the product had been built and tested. And these would only be used to improve its performance in the areas that had been identified by the testing. Since *Forward engineering* begins with a plan, followed by its implementation.

Reverse engineering, on the other hand, begins with an implementation and tries to reconstruct the plan. That is to say, it begins with an existing product and a subset of lower-level tools used to build that product. And it attempts to deduce the rest of the lower-level tools and the higher-level tools, including the high-level designs, used to build that product. So you can use these reconstructed high-level designs, reconstructed higher-level tools and reconstructed lower-level tools to rebuild the original product. Or to build a competitor product. Or to fix the original product after its original design and the original tools used to build it have been lost. *Reverse engineering* depends on the lowest-level tools and empirical tests to begin these deductions and reconstructions.

These deductions begin by experimenting with the product or testing it while studying its characteristics with diagnostic tools. From these experiments, it is possible for someone to deduce some of the other lower-level tools used to build that product. And it is possible from these lower-level tools to deduce some of the higher-level tools.

However, it would be difficult to deduce all of the lower-level tools, and it would be almost impossible to deduce all of the higher-level tools used to build the original product. But that is not the goal of *Reverse engineering*. As has already been stated, the goal is to either create a competing product which at least exhibits the same

external characteristics as the original product. Or to fix errors with the original product without access to its original design.

The first school of thought always depend on *Forward engineering* when building new products, unless they want their software to work with another, and they do not have access to its designs. They include those who practice *Software engineering*.[12] The second school always depend on *Reverse engineering*, when building new products because it affords the artistic licence over the entire software production process. This may seem bizarre at first sight. There is a conflict between making new products and using *Reverse engineering*. But the way in which they resolve this conflict gives them several characteristics.

First, they always rely on an existing product to base their implementations on. They always begin by experimenting with and refining small parts of a *software design* for which they can see an existing implementation, in another product. All subsequent additions are similarly based on examining other products. These additions drive rather than follow any *software design*. These additions also shape the software production process which they follow.

Second, they pay no attention to higher-level tools. This includes any *software designs* at the beginning of the production process: not even their own.

Third, they do pay careful attention to the lower-level tools. They especially pay attention to, to the point of obsession, the lowest-level tools: the characteristics of computer hardware, the programming tools or other lower-level diagnostic tools.

Fourth, they only implement the minimal components required to build any product, as befits the aesthetics of their art. This is a direct result of the deficit between the attention they pay lower-level tools and higher-level tools. They include little or no checks or reports for errors when they write software or software components. Since these checks and reports are all constructs of higher-level tools.

Likewise, they habitually neglect *Abstraction*. *Abstraction* is the ability to create software or software components which hide their internal operations or data. For example, a Word Processor is software that can be used to write and print documents. Its ability to work on different computer hardware or work with different printers is a quality of its *Abstraction*. The User does not need to know its internal operation on one hardware or the next. The User does not need to know the *data* which it sends or receives from the printers. If there is anything wrong with the instructions it sends to the hardware or the *data* it receives from the printers, the User does not need to know this. Indeed the printer may be completely missing and the Word Processor would continue to function and allow you to edit documents.

And when a printer does become available, it will allow you to print documents automatically, without needing to be stopped or restarted. The Word Processor compensates for all these things with automatic behaviour and default options of either the Operating System it is built on or, failing that, additions to that system made by the authors. This is the kind of Word Processor those who belong to the first school would produce.

On the other hand, a Word Processor with no *Abstraction* would only work on one or two computer hardware. It would fail or refuse to start if a printer were missing. It would have no default font, page size, font colour or background colour and require you to explicitly specify all these things. It would fail at the smallest error

in the instructions it sends to the hardware or *data* it receives from the printer. And this is the kind of Word Processor that those who belong to the second school would produce. It is something which is fragile and allows for no ambiguity.

Abstraction allows for ambiguity. The words of natural language are the highest form of *Abstraction*. And it is no coincidence that those who belong to the second school despise this tool and neglect it with the written designs at the beginning of the software production process. And instead they rely on the fragile mechanics of programming languages to document their work. Programming languages allow for no ambiguity.

Fifth, they are useless at making innovative products, for which there is no precedent. This naturally draws them to, amongst others, the Computer Games industry, since it suffers from a dearth of *original games*.

Sixth, they always look at a software project, and anything associated with it, in terms of the lower-level tools.

Finally, they are very conscious of the marketing of lower-level tools, techniques involving these tools, in other competing products and how they can use these to market themselves.

In the Computer Games industry, the competing products they model their implementations on are other Computer Games. It may be a game they have already played before. Or it may be a product they studied in order to learn how to make games. Or it may be a game they have been asked to emulate. They could use *Forward engineering* to build these games. But this would limit their artistic expression, which always preoccupies them.

This fascination with the aesthetics of the art of software production comes about by accident. They stumble across the fact that you can construct complex software products by improvisation. This realisation either comes from noticing the ad hoc tool or method used to optimise the game they admire. Although this method may well have only been employed at the end of its production process to improve its performance: not at the beginning. Or this realisation comes from having used *Reverse engineering* to study their first game. Or this comes from having developed a game using a *Software Evolution Process*. What they fail to notice is that the games they admire, the facsimiles they produce and the product of a *Software Evolution Process* have no characteristics. All of these products exhibit no distinguishing trait or *Quality* which can be relied upon.

This unreliability may not manifest itself only through the errors in these products. But it may also manifest itself through the subsequent corrections, which would be disguised as upgrades or sequels. It may manifest itself through the delays between the appearances of these subsequent releases. It may manifest itself through the inability of each release to innovate over the previous one. And it may manifest itself through the *high turnover of the staff*[13] involved in these production processes.

Ignoring these symptoms of the unreliability of ad hoc methods for optimising products, such as *Reverse engineering* and the *Software Evolution Process*, they never mature. They still see software production as art. They see *Reverse engineering* (and its ability to partially deduce the higher-level designs from lower-level tools) as a panacea to any problems. They think themselves clever because they know one brute force method (i.e. *Reverse engineering*) to cope with all eventualities. Even

though they are ignorant of any other method. To cover this ignorance, they cover the software production process with the aesthetics of their art, as much as possible.

One such example of this is when they produce a *software design*. They are not interested in documenting the components of the software and how these will be made and assembled. They are merely interested in setting out a canvas for their art, with a general outline of the tools and methods that will be used in their composition. They begin by drafting a set of *design principles*[14] either alone or, in a collaborative project, with a handful of their peers who share the same appreciation of the art. These *principles* embody the aesthetics of their art. They do not understand that *design principles* are oxymorons. That is to say, they do not understand that deciding the tools and methods you will use to solve a problem before you have finished describing your understanding of that problem, is narrow-minded.

A design provides a solution to a problem. That is why the *classic software production life cycle* does not proceed with a *software design* without an analysis of the problem (i.e. the customer's requirements or User specification). Since you have to understand a problem before you choose a solution, writing a *software design* has some level of subjectivity. It depends on how you look at a problem and how you approach it. Different people look at a problem and approach it in different ways. By describing, in a *software design*, your perception and approach you can at least limit the level of subjectivity.

Firstly, you at least ensure that others may find solutions, if your solution partially or completely fails. History has shown, time and again, that solutions have been found by looking at a problem from a different perspective or taking a different approach. By describing your approach, you also give yourself criteria for selecting tools, or methods, which you can naturally use in your solution. These are namely those tools and methods that adopt the same approach.

But a set of *design principles* preselects the tools and methods that will be part of a solution. It presupposes the approach and the perception of a problem. It presupposes that the problem has already been mainly, if not completely, understood and solved. In short, *design principles* are only relevant when you are *Reverse engineering* or improvising.

Thus, when those who rely on *Reverse engineering* are presented with any *software architecture*, they will look for *design principles* in it. They conflate *design principles* with *software architectures* (even though *design principles* are not *software architectures* – see the definition of *design principles* in the Glossary). When presented with a *software architecture*, including the **Event-Database Architecture**, they will either dismiss it because they cannot fit it into any of their *design principles.* Or if they do not have the authority to dismiss it, then they will force their *design principles* into it. And undermine the *software architecture* in the process.

For example, one popular *design principle* is a *design pattern*[15] called the *Observer design pattern*. This principle allows for the flow of the game to be controlled by editing two sets of low-level code that encapsulates **Events** in a game. One set of code broadcast **Events** and another set of code can subscribe or unsubscribe to respond to these **Events**. This seems very similar to the system of **Events** that the **Event-Database Architecture** uses. But there are several major differences. One of which is that only those who know *Reverse engineering* can recognise that

design pattern in the low-level code and change the flow of the game, i.e. the *Game Programmers*. Whereas with the **Event-Database Architecture**, anyone can recognize the architecture and anyone who can edit the *Relational Database* can change the flow of the game, i.e. all the staff.

If you are *Forward engineering*, *design patterns* and *design principles* are irrelevant. Your focus is on the original problem; not the existing systems which have solved the problem. This is not to say that, in *Forward engineering*, no prior knowledge is applied to a problem. On the contrary, all your personal knowledge and experience is applied; unconsciously and without any partiality. *Forward engineering* involves finding natural solutions through imagination. *Design principles* are a substitute for imagination.

These principles are a way of limiting the outlook of a problem to the tools or methods you preselect, without having to describe your perception or approach to the problem. The principles provide ways of hiding the difficulty you have understanding a problem. In this respect, *design patterns* and *design principles* are exactly like *heuristics*.[16] *Heuristics* are speculative rules and educated guesses that are meant to increase the probability of solving a problem. These are used to find a solution in the shortest time possible; not the best solution, nor even a feasible solution.

Therefore, a statement of *design principles* is merely a list of limitations to the *software design* beyond which, those who rely on *Reverse engineering*, will not look. Their first thought when faced with a problem is the tools they will use and how they will fit the problem around these tools. Their last thought is to try to understand the problem to select the tools to solve the problem. They like to struggle with a problem rather than understand it.

As already mentioned, this struggle comes from the conflict between creating new products and using *Reverse engineering*. This struggle contrasts with the harmonious relationship between creating new products and using *Forward engineering*. The basis of this harmonious relationship is understanding. It is because the primary concern of *Forward engineering* is understanding, rather than recreation, that the first school of thought relies on it. They believe that if you have an understanding of a problem, then a solution will naturally follow. In fact, they believe several solutions will naturally follow; not just one. And these solutions give both you and your client options, out of which one or more will satisfy you or your client's needs. This belief gives those who rely on *Forward engineering* several characteristics.

Firstly, they believe it is better to have half a product and complete understanding than a complete product and half an understanding. So they spend a majority of their time working on their understanding and a minority of their time working on the product itself.

This will make them welcome the first step of the **Event-Database Production Process**. That is conducting a feasibility test of a portion of the game based on the **Event-Database Architecture** on some computer hardware. Although the game they will be testing will not be the final product, and it may not even be half a product. The test does give them time to have a complete understanding of the *software architecture* the game will use. It will give them time to express their ideas and work on their understanding. It will give them time to try out ideas which they were unsure about. They can return to the feasibility test, later

on in the **Production Process**, to try out the feasibility of the new ideas. The feasibility test will not be discarded as would be the case in the normal *Software Evolution Process* used to build Computer Games. Where the *software architecture* itself mutates along with everything else. And the *software architecture* being used in later steps of the process may be a very different animal from the one they started with at the beginning.

Secondly, they welcome the opportunity to explain their understanding of their work to others, not least because this gives them an opportunity to test their understanding. This also gives them the opportunity to present any documentation they write to record their understanding. This includes presenting any materials describing the *software architecture* they were using, including this book about the **Event-Database Architecture**.

Thirdly, they are very good at expressing themselves. This comes both from listening to other people expressing ideas and expressing ideas themselves. This comes from listening and reading other people's explanations, in order to acquire their understanding. This also comes from explaining their understanding to others.

This quality of those who rely on *Forward Engineering* can greatly benefit the **Event-Database Production Process**. Especially when it comes to expressing the *data* that they want to see in the *Relational Database* to the *Database Administrator.* Before the *Database Administrator* writes the *data design*. This also helps the *Database Administrator* express the contents of the *data design* to the rest of the staff in the production process. If the *Administrator* belongs to that school of thought. Since the definition of the *Database Records* in the *Relational Database* acts as a lexicon or dictionary for the language used by the rest of the staff in the **Event-Database Production Process**. If the *Administrator* were excellent at expressing him or herself, then the definitions would be excellent. And if the definitions were excellent, then the language would be excellent.

Fourthly, those who belong to the first school of thought believe that being able to successfully communicate an idea or a problem is the most difficult part of their work. If you were able to successfully communicate an idea or a problem, then you would have a good understanding of it. And if you were to have a good understanding of the problem, then you would naturally find multiple solutions: not just one solution. Since you can see the problem from multiple angles, express the problem and a solution from each angle.

For example, the **Event-Database Architecture** comes from expressing a single problem, the *Software Evolution Process* in Computer Games from multiple angles. From the angle of the *Game Producer*, *Game Designer*, *Game Programmer*, *Game Artist*, *Sound Designer* or Engineer and the *Game Tester.* And it provides multiple solutions for each angle. The *Game Producer* and *Game Designer* can query or edit the flow of the game by querying or editing the *Relational Database*. For the *Game Testers*, they can find out the major features of the game so far by querying the **Events** in the *Relational Database*. And do some form of *Quality Control* by just testing those **Events** independently with the **Events Hos**t. For the *Game Artist* they can query the *Database* to find all of the artwork, all of the 3D models or 2D images of a particular size, in a particular location, that uses

a particular *Materials* or *Textures*, that were used by a particular **Game Object**. They can interoperate with the **Game Database** and use whatever tool they went to add *data* to the *Database* which can read or write in its *open-data format*. For the *Sound Designer* they can query the *Database* to find all the sounds, all of the sounds of a particular length, all of the **Events** associated with playing sounds, all of the **Game Objects** that produce these sounds. They can use the same system of **Events** as everyone else. For the *Game Programmers* they have very small simple **Game Objects** and **Actions** to write. That naturally break up into other small **Game Objects** and **Actions** depending on the system used to generate the **Game Objects**. And they have a system for generating **Events** which can anticipate future changes in *game design* and allow the flow of the game to easily adapt. And they have a *Relational Database Management System* to help them query and edit all of the **Events**, **Actions** and **Game Objects**.

For this reason, those who belong to the first school of thought would thrive on the ability to express themselves through a *Relational Database* in an *open data format*. Such as the one described in the **Event-Database Architecture**. They would then be able to express any problem they encounter during the **Production Process** using whatever tool they like that could read from, or write to, the *Relational Database* in that *open data format*.

Fifthly, those who rely on *Forward Engineering* produce a generous amount of work because they produce multiple solutions to any given problem. This gives you or your client more than one option. Either they may present you with multiple solutions during the software production process. Or they may see limitations to one perfectly adequate solution and, instinctively, attempt to correct it before presenting it to you. This may seem like a lot of wasted work. But it merely reflects the multiple solutions they naturally produce. And the flexibility of these solutions is what will allow you or your client to address late or unforeseen problems that will almost certainly occur in a project.

This view of software production, held by the first school of thought, would have initially been borne out of education. Their education would have taught them the different steps of a *Forward engineering* process and extolled its virtues. But they would not have had any practical experience with that process. This education would in turn have been gathered, over the ages from the two basic steps which have been applied to manufacture any complex work or product. These two steps being an analysis or a plan, followed by its implementation. As time passed, these steps were broken down further into the smaller steps described in the education of the first school of thought. But although they acquire a knowledge of *Forward engineering*, they do not use it.

Only after the frustration of one or more failed projects do the members of the first school turn back to their education and acquire faith in *Forward engineering*.

Therefore, not everyone who has graduated from school belongs to the first school of thought. But those who have failed recognised from this failure the unreliability of ad hoc methods for optimising products, *Reverse engineering* and the *Software Evolution Process*. They are the ones whom the **Event-Database Architecture** and **Event-Database Production Process** would benefit from the most.

3.2 DIAGNOSIS AND PROGNOSIS

The first step of the *classic software production life cycle* and the first step of the **Event-Database Production Process** are the same. Both processes begin with a feasibility study. The object of the study is to assess whether it is feasible for the software project to be delivered, on time, given its complexity and the tools that were available to build it. The two schools of thought, one which views software production as an art and relies on *Reverse engineering* and the other which views software production as a science and relies on *Forward engineering*, take very different approaches to this first step. In the case of the former, they use their knowledge of lower-level tools (especially diagnostic tools) to conduct this first step. In the case of the latter, they use higher-level tools to conduct this first step.

In the case of the **Event-Database Production Process**, the first step is a minimum set of **Events**, **Actions**, **Game Objects**, *Database Table*, *Records* and *Fields* required to test the **Event-Database Architecture** on the target *platform* and a description of this set in natural language. See the subchapter entitled

Step 1: The Feasibility Study/Vertical Slice

in

The Event-Database Architecture for Computer Games: Volume 1, Software Architecture and the Software Production Process.

Both this set and the natural language describing this set are higher-level tools. Therefore, those who view software production as a science will appreciate these higher-level tools. Whereas those who view software production as an art and rely on *Reverse engineering* will dismiss these higher-level tools or skip over that step. Since it does not depend on their knowledge of lower-level tools. The kind of first step they would appreciate would be one which depends on their knowledge of lower-level tools.

But depending on the knowledge of lower-level tools (e.g. the characteristics of the computer hardware, Central Processor, Graphics Processor, computer memory or the instructions of a low-level programming language) to assess the feasibility of software is impractical and unreliable. Any higher-level tool (e.g. a User Specification, a *software design*, a *game design*, a *technical design*, a *data design* and a *tools design*) would be constructed from many more lower-level tools. So efficiency in higher-level tools would always be more significant than efficiency in lower-level tools. If you found it difficult to tell whether a project would be feasible, from its higher-level tools, then you can conclude that it is not trivial. And if it were not trivial, then you definitely could not tell its feasibility from the perspective of the lower-level tools.

This is why a real feasibility study would concern itself only with the higher-level tools. It would concern itself first with the breakdown of the higher-level tools. This would include the features the software user wants, the interaction between these features and the *User Interface*, the different *software modules*, the *software data* and the *software procedures* required. But a feasibility study would not concern itself with the lower-level tools themselves.

The objectives of the study would be to assess the cost and time it would take to build the software. These would be related to its complexity. So the study would concern itself with the number of different parts of the software and the number of relationships between these parts.

It would concern itself with the number of *software modules*: not cramming as many *software modules* as they can into one *module* or file. So it is faster for the Central Processor to compile or translate the programming language instructions into machine code.

A feasibility study would concern itself with the number of *software procedures*: not cramming as many *software procedures* as you can into one large *procedure*. So that it is faster for the Central Processor to execute its instructions in one place in the computer memory. Compared with breaking up a large *software procedure* into smaller parts, and the Central Processor executes one part in one location in the memory, then jumps over to another location to execute the next part.

A feasibility study would concern itself with the *data* in each *software module*. It would concern itself with the number of elements in each *data*: not the size of the *data* in computer memory or some storage media.

A feasibility study would concern itself with the number of steps in each *software procedure*: not the number of instructions of machine code in computer memory, after the *software procedure* has been 'compiled' or translated from some programming language to machine code.

A feasibility study would concern itself with whether these numbers could be scaled, and how would these scale if more and more features were required. So that if the software were to over-perform on the computer hardware, it could be scaled up. And if the software were to underperform, it could be scaled down.

For example, one of the central items in the **Event-Database Architecture** is a *Relational Database*. Now consider a *software procedure* which looks up a *Database Record* in a *Database*. If the number of steps it takes grows linearly (or slower) from a *Database* with 10 *Records* to one with 1000 *Records*, then it would be a good indication of the feasibility of the *Database*. And by implication, this would be a good indication of the feasibility of the **Architecture**. It is the prudent application of many techniques, which build *scalable components*, that makes software feasible. And none of these techniques depends on the knowledge of lower-level tools or the computer hardware.

Often, in the Computer Games industry, the term 'cross platform' or 'multi platform' would be banded about a lot, by the staff, at the beginning of the production process. This would include *Game Producers, Game Designers* and *Game Programmers*. Especially when discussing a feasibility study in a *game design* or *technical design*. And if you are involved in an **Event-Database Production Process**, then you may hear these term used by *Database Administrators* as well.

They would use this term to imply that some software tool, *game-engine* or other software component, which could be used to build a game, could be used for more than one computer hardware or *platform*. And they would imply this proves that these software tools, *game-engines* or software components were *scalable*. But, in fact, this proves nothing of the sort. Do not fall for this and believe the presence of

these 'cross platform' or 'multi platform' tools when building the **Event-Database Architecture** makes it *scalable* and, therefore, feasible.

Instead, when they use these terms, they would be merely referring to the fact that they expect the software tools, *game-engines* or software components to give the best performance on the lowest common denominator. That is to say, they believe these 'cross platform' or 'multi platform' tools were built with the characteristics of the *platform* with the least resources in mind. And as a result, in theory, this should give the best performance on other *platforms*, with more resources: or so they hope.

But, in practice, during production, this never happens. Instead, they end up always building several, discrete versions of these 'cross platform' or 'multi platform' software tools, *game-engines* or software components; one for each *platform*. And they customise each one to give the optimum performance for each *platform*. Each version will have subtle but significant differences which make it not compatible or interoperable with another. Each version is not *scalable*. Instead of introducing one *scalable* tool to the software production process, they have doubled, trebled or multiplied even more the number of tools overall in the software production process. And as a result, they have increased its complexity and reduced its feasibility.

The school of thought that believes software production is an art, and relies on *Reverse engineering*, especially likes to use the terms 'cross platform' and 'multi platform'. Whenever they mention these 'cross platform' or 'multi platform' tools in their feasibility studies they discount their effect on feasibility. They discount each discrete version of tools, and *software modules* based on these tools, they will have to use or build for each computer hardware, *Operating System*, *game-engine* or *platform*. They discount the number of different versions of some of the tools or software modules that would be required to build the software. And instead, they choose to count all of these different versions as one. Thus, they cover up the complexity of their *software designs* and the difficulty they have conceiving *scalable* software.

For it is very difficult, if not impossible, to measure the performance of *scalable* software, practically using the diagnostic lower-level tools of *Reverse engineering*. And hence it is difficult to judge its best performance. Since *scalable* software would vary its performance, depending on how much resources were available. And any diagnostic lower-level tools they used, to measure its performance, such as a *software module* or *library*, would inevitably share the same resources, such as the computer memory, Central Processor, Graphics Processor and so on, as the *scalable* software under investigation. And that, in turn, would affect the measurements taken up by these diagnostic tools, rendering them virtually useless.

So the school of thought that believes software production is a science and relies on *Forward Engineering*, would only measure the performance of *scalable* software, such as the **Event-Database Architecture**, through higher-level tools. The transparency of the components of the **Architecture** and the natural language of the **Architecture** helps to do this. You can describe and number the *software modules* using **Game Objects**. You can describe and number the steps of the *software procedure* using **Events**, **Actions**, **Game Objects**, *Database Table*, *Records* and *Fields*. You can describe and number the *software data* and its elements using *Database Tables*, *Records* and *Fields*. You can describe and number the relationships between the **Events**, **Actions**, **Game Objects**, through the relationships of the *Database*

Table, *Records* and *Fields*. And you can see how these numbers increase with more software requirements. And thus, you can assess how *scalable* and hence how feasible the **Architecture** is. You cannot get any of these metrics reliably from the school of thought that believes software production is an art. You cannot measure the number of *software modules*, for the files they write will containing multiple *software modules* in the same file, for the sake of minimalism. You cannot measure the number of steps of *software procedures* they produce, for the files they write will contain obfuscated, undocumented, multiple instructions embedded in each other, on the same line, again for the sake of minimalism. You cannot measure the relationship between the *software procedures* or the *software modules* they produce for the same reason.

Compare how the **Event-Database Architecture** and **Event-Database Production Process** facilitates a feasibility study, with a normal ad-hoc *Software Evolution Process* in the Computer Games industry, which begins also with a feasibility study. The ad-hoc nature of the *Software Evolution Process* lends itself to the idea that software production is an art. And thus, it is usually led by those who come from that school of thought, who rely on lower-level tools and *Reverse engineering*. When they claim to have assessed the feasibility of a *software design*, the reasons they give will show their claims to be false. These reasons will not come from a conception of *scalable* software.

Instead, these reasons will come from their knowledge of lower-level tools. Especially the information they have gathered from using diagnostic tools to analyse software products from the past or the characteristics of the computer hardware. They will skip any examination of the higher-level tools that could be used to build the software: such as the User Specification or the *software design*. And, instead, they will directly jump to suppositions about the lower-level tools that could be used to build the software.

In order to justify their suppositions, they will dismiss the higher-level tools as trivial. But, if you were to investigate how they came to this conclusion, you would find that it would be because they have ignored large sections of the User Specification and the *software design*. And, instead, they would have only concentrated on the two or three parts of these tools, which they could use to market their study. The parts chosen would be based on popular wisdom.

For Computer Games, the current popular wisdom is concerned with displaying photorealistic three-dimensional graphics, controlling *Artificial Intelligence* or modelling physics in software. After looking at these parts of the software, from the perspective of the lower-level tools, they would have come up with their assessment of its feasibility. And as a result, the rest of the features of the Computer Game, whose feasibility they ignore and marketability they find inferior, will be crudely implemented during the production process. Hence, the entire feasibility study will be a waste of time. Since, invariably, the demands of the parts that were unaccounted for will undermine the feasibility of the parts they do account for.

In the **Event-Database Production Process**, a feasibility study is not just done in the first step of the process. It is also done during the middle of the process as well before each new feature is introduced to a game. And again, the two schools of thought will approach this feasibility study very differently.

For the school of thought that believes software production is a science, they would approach this study with the view of how each new feature could be built in a *scalable* fashion, using *scalable* software or *scalable* techniques. How does the number of **Events**, **Actions**, **Game Objects**, *Database Tables*, *Records* and *Fields*, increase when the demands of this new feature increase? The **Event-Database Architecture** lends itself to this method because you can get all of these numbers easily from the **Game Database**.

For example, suppose the new feature being introduced was a spell which a player can cast in the *Game World*, which produces a fireball at some point in the *World*. And this fireball spreads out in a radius to a set distance from the epicentre. And every **Game Object** engulfed by the fireball as it spreads can catch fire and be damaged.

Now the number of steps, to start and end this fireball effect, is almost always the same in the **Event-Database Architecture.** Whether the maximum radius is a distance of 5 metres or 10 metres. There are only two things that change.

The first thing that changes is the radius of the Collision Mesh or Model around the epicentre of the fireball, which the **Physics Host** uses to detect all of the **Game Objects** within range of the fireball. And the second thing that changes is the length of the **Primary Proximity Event Record** and the number of **Secondary Proximity Events** of **Game Objects** engulfed by the fireball, that are on that list. The school of thought that believes software production is a science can get all these numbers easily from the **Game Database**. And use these numbers to assess the feasibility of adding this new feature.

For the school of thought that believes software production is an art, they would not be concerned with these numbers. They would approach this feasibility study in the middle of the **Event-Database Production Process** as they do at the start of a *Software Evolution Process*. That is to say, they will build the software by making small, incremental changes or additions. And they will ignore the rest of the software whenever they make each addition. So that the additions they make appear to be trivial. Thereby enabling them to convince others of their assessment of the feasibility of some new feature, based on the knowledge of the lower-level tools: especially the characteristics of the computer hardware. Subconsciously, at least, they will be aware that they cannot convince anyone of the effect of these incremental additions unless they make each addition appear to be trivial.

For example, by concentrating on a single instance of a fireball spell, in a competing product, and ignoring the rest of that product, they will claim that it would be trivial to recreate that feature in the Computer Game they were working on. So that they may convince others who were collaborating with them on that product that the addition would be feasible.

But only after they have made this addition will they use their lower-level tools, especially diagnostic tools, to analyse the true performance of that fireball spell in their product. And then use the results of this analysis to retrospectively optimise the performance for the fireball spell. Yet they will only do this in the case of that single instance that they showed in the competing product to convince others of its feasibility. They will not optimise the performance in the rest of the game. Since they did not show the other instances of the fireball spell in the rest of the competing

products. They will only optimise the performance in these other cases, in their product, on an ad-hoc basis, as and when they are discovered by the *Game Testers*. Who will, of course, have no systematic test plan for the product. And that in turn was due to the absence of a complete User Manual to draw such a test plan from. And that in turn was because there was no complete *game design* to draw such a User Manual from, at the beginning of a *Software Evolution Process*.

A study of the feasibility of software depends on being able to identify the number of different parts and the number of interactions between these parts. As has already been mentioned, it does not depend on the components of the lower-level tools: not even the components of the computer hardware. The complexity of the components on the hardware level does not reflect the complexity of the components on the software level. These two levels can be the same. But in all probability, given the broadness of hardware applications and the narrowness of *Software Applications*, these two levels will be different. So given its dependencies, the accuracy of a feasibility study would depend on the completeness of the description of the different parts of the software. That is to say, it would depend on a complete *software design*.

Thus, given the concerns of assessing the feasibility of software, it is highly unlikely that knowledge of lower-level tools (including computer hardware characteristics) or *Reverse engineering* would either make an unfeasible project feasible, at the beginning of the **Event-Database Production Process**. Or this would make an unfeasible change feasible in the middle of the production process. Therefore, the obsession of the school of thought that believes that software production is an art, with these lower-level tools, should not be allowed to dominate the feasibility studies at the beginning or middle of the process.

The number of parts involved in modern Computer Games, at the level of the computer hardware, is just too great for anyone to comprehend. It takes teams of skilled engineers to design a single microprocessor chip, such as Central Processor or Graphics Processor, used in the hardware of Computer Games. And that hardware can have up to 8 'Cores' in the Central Processor and 36 'Cores' in the Graphics Processor, giving you a total of 42 'Cores' or microprocessors. Not counting the microprocessors in the other components. The documented measurements of the performance of these microprocessors merely reflect their rudimentary performance. Thus, the predictions based on knowledge of these low-level tools, contained in these documents, merely reflect rudimentary predictions.

These measurements include, for example, the time taken by each microprocessor instruction relative to the speed of the chip. Predictions based on such measurements are misleading. For the speed of a chip does not reflect how it performs for any given *software design*. Although two different chips may have the same speed, one may perform better for a given *software design*, than the other.

Thus, the engineers of these microprocessors rely on *benchmarks*[17] to statistically analyse the performance of the chip. Unfortunately, each test has to encompass so many factors that such statistics often prove unreliable. Some engineers design their chips to perform better at particular *benchmarks* than others. And depending on how much your *software design* overlaps with these *benchmarks*, the performance of the software marginally improves.

But the engineers who make these microprocessors would not publicly release information about the *benchmarks* which they target. Nor would it have occurred to them that the choice of your *software design* should be defined by these *benchmarks*.

Often the school of thought that believes software production is an art, like to give the impression that they know what these *benchmarks* are. And they may say words to that effect when they give their assessment in a feasibility study at the beginning or during the middle of the **Event-Database Production Process**. They believe this qualifies them to select or reject a *software design* or *software architecture*, depending on how much of it overlaps with these *benchmarks*. But do not be fooled! They can only speculate about what these *benchmarks* are.

Nevertheless, the fact that these *benchmarks* exist at all shows how absurd it is to base predictions of the performance of software, including *software architectures* such as the **Event-Database Architecture**, on the knowledge of the characteristics of the computer hardware. Even the engineers who make the microprocessor chips that make up that hardware know that it is a big leap. From their knowledge of how a chip behaves to how it would behave at a higher level. It is a big leap from the instructions of microprocessors to the components of a *game design*, or even a t*echnical design*.

It is a big leap from using a tool that views the instructions of a microprocessor, or a low-level programming language, watches a particular part of the computer memory or stops the software when it reaches a set point in a *software procedure*. To understand the impact of a *software design* or *software architecture* on the computer hardware. It is a big leap from reading diagrams of hardware circuits to building a racing circuit in a *Game World* on that hardware. There is no formal method that uses these low-level tools to make a prognosis.

The engineers of these microprocessors only use these low-level tools for diagnosis: either to test the chip or measure the rudimentary details of its performance, after the chip has been built. They never use it for prognosis, which would be required to give an assessment in a feasibility study at the beginning or during the middle of an **Event-Database Production Process**.

When taken too far, an obsession with the performance of the software on a computer hardware or *platform* can come to dominate a feasibility study. The school of thought that believes software production is an art may for example insist that the design of the graphics for a game, produced by the *Game Artists*, follow their *design principles*. So that they can guarantee the feasibility of the game, by which they mean they can guarantee the best performance of the software on that computer hardware. But this can have an adverse effect on members of the staff in the **Event-Database Production Process** who are not even required to write a *software design*.

Instead of simply focusing on meeting the requirements of artwork in the *game design*, the *Game Artist* has to focus on the *design principles* and meeting the requirements of the computer hardware. This produces two ironies.

The first irony is that the object of the feasibility study is to assess whether the software requirements can be met given the time and tools available to make it. But the *design principles* which act as a substitute for a feasibility study turn this objective on its head. And makes it an assessment of whether the software meets the hardware requirements.

The second irony is that when the *game design* is incomplete, which is true at the beginning of an **Event-Database Production Process** or a *Software Evolution*

Process, these *design principles* produce a lot of waste. This is because improving the performance of one feature of the game could easily be made redundant when the *game design* has been changed.

For example, suppose a *Game Artist* followed the *design principles* to build a 3D Mesh or Model of a football stadium in a city in the *Game World*. The *Artist* then painstakingly reduces the number of *polygons* in the Mesh by gutting out the interior of the stadium. So that it is empty, and only the facade around the outside of the stadium remains. To produce the best performance of the software when rendering that stadium in the computer hardware from a distance. In accordance with some *design principles.*

But later on the *game design* is changed. A decision is made to put a huge block of flats in front of the stadium. Obscuring its view from the player. And the *game engine* will simply work out that the stadium is not visible because the new block of flats obscures the view. And, therefore, not render the stadium.

And later on another decision is made that the player can enter the stadium and see inside it. All of a sudden all of the work that the *Game Artist* made to gut the stadium according to the *design principles* has been a waste of time and redundant. And the *Artist* has to restore the interior of the stadium which was gutted earlier.

For those who view software production as a science, and rely on *Forward engineering*, this would not be a problem. They would simply not use *design principles* to enforce that the work of the *Artists* met the hardware requirements. To ensure optimum performance of the game throughout production. And in turn make the project feasible. In *Forward engineering* the issue of performance is addressed at the end of the production process: not at the beginning. And, therefore, they would not produce the waste or redundancy that *design principles* generate. When faced with the introduction into the *game design*, of a 3D Model or Mesh of a football stadium, with a high number of *polygons*, they would look to capitalise on this challenge with the **Event-Database Architecture**.

If the high number of *polygons* of the stadium affected the rendering of that area of the *Game World*, making it difficult to play in that area, then they would simply replace the Mesh for the stadium with a temporary cuboid with just six *polygons*, during the **Event-Database Production Process**. By editing the **Game Database** and changing the *Database Record* for the **Game Object** of the stadium. So that the *Database Field* that refers to the *Record* for the original Mesh was changed to refer to the *Record* of a cuboid. Note that anyone who can edit the **Game Database** can do this, not just the *Game Artists* who created that Mesh. And you do not need any large complex general-purpose tool with a 3D *User Interface*, like a *game editor*, to do this. Any simple tool which can edit the **Game Database** can do this, even if it has no *User Interface* at all. And indeed you can add a **Game Object** to automatically do this when it notices the performance of the game deteriorating in that area.

For those who rely on *Forward engineering* in software production, the end of the production process would be the time to deal with the performance of the game. And when it came to the end of the production process, they could query the **Game Database** and find all the *Database Records* referring to this cuboid Mesh. And edit the **Game Database** again to restore the *Records* back to the original Meshes.

If the Mesh still presented problems with the performance of the game on the computer hardware at the end of production, because of the number of *polygons*, then they would look to replace it with another version of the Mesh which had a hollowed-out stadium.

If the *game design* were changed, that required the player to enter the stadium, then they would look at adding a **Primary Event** for this, and a **Game Object** placed at the entrance to the stadium which generates this **Primary Event**, to the **Game Database**. This **Game Object** would generate the **Primary Event** when the player approached the entrance from the outside.

They would also attach several **Secondary Events** to this **Primary Event**. One of these **Secondary Events** would be sent to the **Game Object** of the stadium. And it would replace the hollowed-out Mesh of the stadium with the original Mesh with a complete interior. And the other **Secondary Events** would be sent to the **Game Objects** all of the buildings around the stadium which were either not visible from the interior or were too far away from the stadium. And these **Objects** would stop themselves from being rendered by the **Graphics Host**. To improve the performance of rendering the *Game World*, while the player was in the stadium at the end of the **Event-Database Production Process**.

In contrast, the focus of the school of thought that views software production as art will be on performance at the beginning of the **Event-Database Production Process**. In the feasibility study at the start of it, you may find from time to time, a discussion about the *time complexity*[18] of software. On the surface, this may seem like a theoretical discussion about the simplicity and efficiency of the software.

But the *time complexity* of software has nothing to do with the simplicity of its composition. Software with a good *time complexity* can be far more complex than one with a bad *time complexity*. And the greater the complexity of software, the greater the number of errors it will have.

The *time complexity* of an algorithm in software also has little to do with the efficiency of the software. When used correctly, the *time complexity* of an algorithm is used to analyse how the theoretical steps of that algorithm would perform in the worst case. It is not used to analyse how the algorithm would perform in the best case. So there is no basis for using *time complexity* to speculate or infer anything about how to get the best practical performance on some computer hardware. Notwithstanding the fact that the *time complexity* of an algorithm does not account for the hardware it will run on.

However, when those who view software production as an art consider *time complexity* in a feasibility study, at the beginning of the **Event-Database Production Process**, this will quickly degenerate into another discussion about the performance of the software on the computer hardware. And they will take the opportunity to examine the low-level performance of the software on the computer hardware. They will take this opportunity to examine speculative low-level machine code or computer hardware instructions that the software may produce to run on the system. And indeed machine code or hardware instructions are the most efficient way of utilising hardware. But at the same time, these instructions are the most obscure, least fault-tolerant and most complex way to analyse and produce software. This is due to the sheer number of instructions required to produce anything useful.

So how would those who view software production as a science use the *time complexity* of software? They would use it to select *scalable* algorithms that could be used to build the game, with the **Event-Database Architecture**. To analyse the steps of the algorithms. And their description of these *scalable* algorithms would not mention the low-level machine code or hardware instructions. And they would not use it to speculate about its performance on hardware.

3.3 THE DIDACTIC AND THE DIALECTIC

Didactic is a form of literature, art or design that is meant to be instructive, especially one which is excessively morally instructive. Dialectic is the art or practice of arriving at the truth by the exchange of logical arguments. In philosophy, these arguments take the form of a dialogue between two people with different points of view on a subject. Both the Didactic and Dialectic are forms of communication.

The communication of those who view software production as an art tends to take on the Didactic form. Whereas the communication of those who view software production as a science tends to take on a Dialectic form. Since the former relies on *Reverse engineering* that does not require a dialogue. The software producer can, by using *Reverse engineering* to examine other competing products, understand to a limited extent the software requirements. Whereas the latter relies on *Forward engineering* which does require a dialogue. The software producer cannot start the process until they have had a dialogue with the software user.

Communication is also an important part of the **Event-Database Production Process**. It is one of the main goals of the **Event-Database Production Process** and the **Event-Database Architecture**. To solve the communication which arises when you start a production process without a complete game design. But depending on whether the software project is led by those who view software production as an art, or as a science, the communication of the leadership will take on a Didactic form or a Dialectic form. And this can affect the ability of the **Event-Database Production Process** to meet its goals.

Those who view software production as an art, and rely on *Reverse engineering*, find it a struggle to communicate their ideas. Since they rarely practice writing their *software designs* and using their own words. So they adopt the instructive language of the lower-level tools which aid *Reverse engineering*, including the instructions of the manuals for the computer hardware or diagnostic tools they use. They have a penchant for keywords, abbreviations, acronyms and contractions; and compounds made from keywords, abbreviations, acronyms and contractions.

When they write their *software designs*, they find it hard to link two sentences or two paragraphs together. They prefer a more heavily disjunct form, than free verse. These include forms like an agenda for a meeting, a *memorandum*, a bill and other forms of informal communication.

They find it difficult to engage in an informed Dialectic argument. More significantly, they enjoy arguing from a position of ignorance, which fits into their *Reverse engineering* philosophy.

For example, suppose there is a *Bug* which arises in some feature in a Computer Game during the production process. The leaders of the process who view software production as an art could investigate how that feature was built, before making

suggestions on how to fix a problem. But instead they will invite whichever colleague was responsible for it to explain it verbally in a meeting. If the colleague refers them to some written explanation or documentation for that feature, then the leaders will not read it. And instead the leaders will insist on a verbal explanation and any attempt to refer them to documentation as insubordination.

So the colleague agrees to explain the feature verbally in a usually informal impromptu meeting. But this unsuspecting colleague has just walked straight into the middle of an ambush. Where they will be interrogated and used as a diagnostic tool to analyse the problem. In the middle of the colleague's explanation of how that feature was built, the leader will interrupt and start probing with confrontational assertions and negative propositions.

> Why can't it be done like this?
> Why can't it be done like that?

For example, suppose the *Game Testers* discovered that a football stadium created with a Mesh with a high number of *polygons* was added to the landscape of a city at the beginning of the software production process. And sometimes much later, after the beginning, the *Game Testers* noticed that the performance of the game was suffering afterwards when the player reached a certain point in the city and looked across the landscape. The leader who views software production as an art would invite the *Game Artist* responsible for building the stadium and adding it to an impromptu meeting. Their dialogue would proceed something like this:

> 'Did you add this stadium?'
> 'Yes!'
> 'Why did you add it?'
> 'It was part of the Game Design. The Game Designers thought it would be cool'.
> 'But the Game Testers found out recently that there is some performance issue around that area'.
> 'Yes! I have heard about that. I think sometimes when you look across the city in that direction the game starts to hitch and Frame rate ----'
> 'Why can't you build a low polygon version?'
> 'That version is what the Game Designers wanted. They saw it and we were all happy with it'.
> 'Why did they want that placed there? Why don't you place it outside the city?'
> 'I don't know! But I suspect they had their reasons'.
> 'Why don't you hollow out the stadium and reduce the number of polygons'?
> 'But what if the player needs to go into the stadium for some reason in the future? Or they can see the stadium from some high vantage point?'
> 'Why don't you place some tall buildings around it so you cannot see it from a high vantage point?'

The leadership will not be merely making proposals. This will be evident by the fact that their proposals will be in the form of negative propositions. And you cannot prove a negative proposition: why something cannot be done.

This will also be evident by the fact that they will not wait for their colleague to finish his or her explanation for introducing some feature, in this case, adding a football stadium to the *Game World*. Their questions will also clearly profess that they were ignorant about the subject. Yet when their colleague points out the adverse effects of their negative propositions, they do not stop and try to get more information. Instead, they come back straight away with more negative propositions.

Only two things stop this cycle. Either they get an immediate agreement from their colleague to follow their instructions. Or their colleague inundates them with the information which they were reluctant to seek, which requires them to reconsider. At this point they back off and either pretend that the whole subject was confusing or they have nothing significant to say.

This form of communication is merely Didactic. In a Didactic communication, the need to instruct is the driving force behind the conversation. The proposals or negative propositions, in the form of questions, in the meeting, are rhetorical questions. The proposals are meant to be taken as instructions. To make the instructions sound authoritative, the leaders who view software production as an art will replace knowledge with supposition. They will make decisions about the *game design*, *technical design*, *data design* or *tools design* in an **Event-Database Production Process**. Without consulting the software users, players, *Game Producers*, *Game Testers*, *Game Designers*, *Sound Engineers*, *Game Programmers* or the *Database Administrators*. And without any materials before them.

Contrast this approach with a project led by those who view software production as a science and rely on *Forward engineering*. If they invited the *Game Artist* to a meeting, as soon as the *Game Artist* said

> 'It was part of the Game Design'.

The leaders would stop the meeting right there. And they would investigate. They would find the materials related to the *game design* and consult the reasons given there for adding the stadium to the city, if any. They would consult the *Game Producers*, *Game Testers*, *Game Designers* and *Game Programmers* to get their views on the stadium. They would get each view of that subject, note them down and note the contradictions. They would perform a comprehensive test of different views of the stadium throughout the city. To find any other buildings or locations where the performance of the game also suffered when the player looked across the city. And after they have listened to the logical arguments, from the different views, they will make an informed decision about the stadium.

They would do all this because this is what is required by *Forward engineering*. This is what they are used to. That is to say, a Dialectic form of communication. This requires a dialogue. This requires logical arguments based on two or more views of the same subject to reach the truth. You cannot have logical arguments based on different views of a subject, without listening to those views. This is how they will

begin the **Event-Database Production Process**. This will be how they continue in it. And this is how they will end it or any arguments.

3.4 SOFTWARE ARTISTS AND SOFTWARE ENGINEERS

As mentioned in the previous subchapter, there are two schools of thought in software production, and they have two different views of the first phase of the *classic software production life cycle*, which is also the first phase of the **Event-Database Production Process**. That is to say they have two different views of the feasibility study and produce two different results.

In the Computer Games industry, the school of thought that views software production as an art and relies on lower-level tools and *Reverse engineering* in the software production process, produce partial assessments of the feasibility. Based on a limited design and limited study, that just focuses on two or three of the most popular areas in the production of Computer Games: photorealistic 3D graphics, *Artificial Intelligence* and Physics.

The school of thought that views software production as a science, and relies on higher-level tools and *Forward engineering*, produces a comprehensive assessment of the feasibility. Based on a belief in comprehensive *software designs* and a conception of *scalable* software. That in turn, along with practical tools and a realistic schedule based on the *software designs*, produces feasible software.

Which of the two views of the first phase of the production process prevails will affects the subsequent productivity of the rest of the staff, during the subsequent phases. And likewise, affect the rest of the phases of the **Event-Database Production process**.

Ironically even though they do believe software production is an art, and, therefore, believe there is no complete *software design*, those who view production as an art benefit the most from complete designs.

It is the complete and documented designs of the computer software they use, the computer hardware and *Operating Systems* these run on, that gives them the knowledge to practice their art. They love nothing better than taking advantage of other people's courteous designs to practice their art. Yet they hate nothing more than having to write *software designs* for others to use.

They are incredibly productive in a software production process that is led by the school of thought that believes software production is a science. As indeed are the rest of the staff, the *Game Producers*, *Game Artists*, *Game Designers*, *Sound Designers* or *Engineers* and *Game Testers*. They will all benefit from the transparency of the first phase of the **Event-Database Production Process** led by that school of thought.

But if it were led by the school of thought that believes software production is an art, the converse is true. The productivity of those who believe software production is a science will suffer. As will indeed the productivity of the rest of the staff. The only ones that will seem very productive will be those who believe software production is an art and rely on *Reverse engineering*. And this characteristic of a software production process, including the **Event-Database Production Process**, the lack of productivity apart from a small band of staff who are proficient at *Reverse*

engineering, and seem to have a superior knowledge of the process, is a characteristic of a process led by those who believe software production is an art. Their production processes favour lower-level tools.

For example, suppose you had a game based around exploring a city. And in that city was a football stadium. Initially, the stadium was simply one of many buildings in the city that you could see but not enter and interact with. But later on in the production process, the *game design* was changed. So that the players could enter the stadium. And when they walked through the entrance of the stadium, a crowd would appear along the seats around the stadium. And a football match would begin on the pitch in the middle of the stadium.

On the one hand, those who view software production as an art could edit the game using a higher-level tool like one of the *game editors* or **Events**, **Actions**, *Database Tables*, *Database Records* or *Database Fields* in the **Game Database** of the **Event-Database Architecture** to do this. To add the **Game Objects** of the entrance, the crowds in the stadium, the players playing football on the pitch. To add **Game Objects** for the ball, the referee, the linesmen, the coaches of the two sides and their substitutes sitting along the side of the bitch, the TV cameras and the crew. To add the Meshes for **Game Objects**. To add the animation of these Meshes. To control the *Artificial Intelligence* of the crowd, the players, officials, the coaches, the substitutes and the TV crews. To add the **Primary Events** and **Secondary Events** to control the **Actions** of all these **Game Objects**, including their animations and *Artificial Intelligence.*

But this would be a slow and arduous task. And on the other hand, they could perform the same task by buying or developing a new lower-level tool, like a programming tool. To procedurally generate the addition of the **Game Objects**, the Meshes and the animations. To control the *Artificial Intelligence* of the characters in the stadium. Those who view software production as an art would choose the latter of the two options, i.e. the programming tool, for the sake of the art of minimalism.

For example, you could use a programming tool to procedurally generate points on the seats around the stadium. These seats would be contained within two rings or ellipses with two radii. One larger radius for the outermost ring of seats and one smaller radius for the innermost ring of seats. In between the outermost ring and the innermost ring of seats, there would be other rings, at different heights. Beginning at the highest elevation in the outermost ring, and dropping down steadily at equal, to the lowest elevation in the innermost ring. You could write a programme to work out how many rings there would be between the inner and outer rings using some mathematical formula. And you could then generate points along each ring, with the same spacing, that would act as the seats. You could then generate members of the crowds at each point, facing the centre of the stadium.

You could have a *Game Artist* create a skeleton for the members of the crowd which would be used to animate them. And create one animation of the skeleton for the crowd sitting down, another for the crowd jumping up and another for the crowd jumping up and down on the same spot and waving their hands as if a goal had been scored. Then write a programme to animate the crowd depending on some *Artificial Intelligence* which can detect when the players on the football pitch, are on the attack, on the defence or have just scored a goal.

You could also have an Artist create four Meshes or Models for the heads, bodies, arms and clothing of the crowd: one adult male, one adult female, one male child and one female child. And then use the programming tool to either randomly add these body parts and clothing to the skeletons of the members of the crowd. Or using some mathematical formulas which make sure that all of the members of the crowd, wearing the same colours as one of the two football teams, were grouped together in the crowd.

You could also have a *Sound Designer* or *Engineer* generate the sounds to accompany this. Sounds of the crowd singing, chanting the names of their team, cheering and groaning could all be generated. And you could have a programming tool procedurally play these sounds in accordance with the reactions of the crowd, to the action on the pitch, generated by some *Artificial Intelligence.*

You could also use a programming tool to procedurally generate points on the football pitch in the centre for the players. By dividing the pitch into two halves, on its longer vertical side. And then placing 11 players of one team at random in one half and placing the 11 players of the opposition at random in the other half. Or using some *Artificial Intelligence* to place them according to some random football formation, e.g. 4-4-2 or 4-3-2-1, that the coach was playing.

You could have a *Game Artist* create a skeleton for the football players which would be used to animate them. One for the players walking. One for the players running. One for the players walking with the ball at their feet. One for the players running with the ball at their feet. One for the players passing the ball. One for the players shooting the ball. One for the players tackling for the ball. One for the players jumping to header the ball. Then write a programme to animate the players, moving them across the football pitch, depending on some *Artificial Intelligence* which controls how the players attack, defend, tackle or score a goal, in their formation.

Likewise, you could use a programming tool to procedurally generate points along the side of the football pitch, for the two benches with the coaches and the substitutes. This would be a series of points, next to each other running along the longer side of the pitch. One set of points would be along the top half of the pitch, between the rectangle of the pitch and the inner ring of the crowd. The other bench would be along the bottom half of the pitch, on the same side, between the rectangle and the inner ring. And then you could reuse the same procedure and process you used to generate the crowd in the seats around the stadium to generate the coach and substitutes on the bench. And you could reuse the same *Artificial Intelligence* that controlled the reactions of the crowd in the stands to the actions on the pitch to control the reactions of the coach and substitutes on the benches.

For the TV crew covering the football match, you could use a programming tool again to procedurally generate a series of *Waypoints* along each of the four sides of the pitch. These points would be spaced evenly along each side. And there would be one cameraman, one soundman and one TV camera, on each side. And these three would move together as one either at random between the points along each side, keeping the football on the pitch, in the line of sight of the TV cameras, which will always face the ball. Or these three would move according to some *Artificial Intelligence.*

Again you could have a *Game Artist* create a skeleton for the TV crew and TV camera. One animation for them walking. One animation for them running. And you could have the same *Artificial Intelligence* that controls their movement controls their animations and switches between these animations.

To those who view software production as an art, there would be nothing wrong with encapsulating all of these procedurally generated points in a low-level tool, like a programming tool. They neither care that more of the staff are capable of understanding and editing the *Game World* using a higher-level tool, like the **Events**, **Actions**, **Game Objects**, *Database Tables*, *Database Records* and *Database Fields* of the **Event-Database Architecture**. Nor do they appreciate the benefit of a large collaborative project. They neither care that, even if it were trivial, there would be less chance of introducing more errors by using a reliable higher-level tool, than introducing new low-level tools that procedurally generates a *Game World*. Nor do they care that the ability to use a reliable higher-level tool is a sign of advanced engineering.

By encapsulating all of these procedurally generated points in a programming tool, apart from the *Game Programmers* who know how these points were generated, no one else can understand the Game World. How many seats were in the stadium? How the number of seats in the stadium were generated? How the crowds were generated? How to increase or decrease these numbers? How to change the layout of the crowd in the stadium? How to insert gaps in the crowd for entrances from the outside into the stadium, including the seating area? How to add greater variety to the composition of the crowd? When do the crowd have a change of reaction to the action on the pitch? From cheering to silence? From silence to chanting? When do the football players have a change of pace from walking to running? From running to walking?

In contrast, those who view software production as a science, and rely on *Forward engineering*, rely on higher-level tools. The highest-level tool is natural language. *Forward engineering* begins with natural language, in the design documents produced at the start of the process. They cannot easily explain changes in natural language made with lower-level tools, like a programming tool. They can more easily explain changes in natural language made with higher-level tools, like a *game editor* or the **Events**, **Actions**, **Game Objects**, *Database Tables*, *Database Records* or *Database Fields*. So in this case, they would choose the higher-level tool.

In that case, you would know how many seats were in each stadium. Each seat would be added by creating a *Database Record* for its **Game Object** in the **Game Database**. And you could query this *Database* and count these *Records*. You could scale this up if that produces too many **Game Objects**, that need to be processed or rendered at once, by having each **Game Object** represent groups of 100 or 1000 seats in the stadium.

You would know how these seats were generated. Each seat or group of seats would be generated from a single **Game Object** in the **Game Database** whose *Database Records* would refer to the same Mesh for the seat.

You would know how to increase or decrease this number. By adding more **Game Objects** for more seats in the **Game Database**. And after that adding these new **Game Objects** to the 2D or 3D **Graphics List Record** for the **Graphics Host** for those new seats to be rendered.

You would know how the crowds were generated. Each member of the crowd would be generated from a single **Game Object** in the **Game Database**. Whose *Database Record* would refer to a position next to each seat. You could scale this up if that produces too many **Game Objects**, that need to be processed or rendered at once, by having each **Game Object** represent groups of 100 or 1000 people in the crowd.

You would know how to change the layout of the crowd in the stadium. By changing the layout of the **Game Objects** for the crowd in the **Database**.

You would know how to insert gaps in the crowd for entrances into the stadium. By removing **Game Objects** for some of the crowd covering those gaps, from the **Game Database**.

You would know how to add greater variety to the composition of the crowd. By querying the *Database* to find all the *Database Records* of the **Game Objects** of the crowd which used a particular Mesh. And by editing these *Records* changing the Mesh used to control their appearance and animation.

You would know when the crowd have a change of reaction to the action on the pitch. By the **Primary Events** that were sent to the **Events Host**. When it was time for the crowd to stand up, sit down, cheer or start chanting.

You would know when the football players have a change of pace. By the **Primary Events** that were sent to the **Events Host** when the players started running or started walking.

All of this would come from the **Events**, **Actions**, **Game Objects**, *Database Tables*, *Database Records* and *Database Fields* of the **Event-Database Architecture**.

This will be true from the beginning of the **Event-Database Production Process** whether it is led by those who view software production as an art or those who view software production as a science.

That is to say, when it is led by those who view software production as an art, there will be a recession. A recession away from higher-level tools (including natural language) to lower-level tools. And with the receding of natural language, just as in the *Software Evolution Process*, just as in the *Tower of Babel*, comes chaos.

Within this chaos, the sudden moments of clarity, and burst of high productivity, which they achieve through *Reverse engineering*, will look impressive to almost everyone. And this impression will secure their objective. That is to say, they will acquire an unnatural leadership, because of their impressive productivity, which they could otherwise never aspire to.

They may well be masters of *Reverse engineering*. But being a good bricklayer does not make you a master architect. Being a good mechanic does not make you an automotive engineer. They may well seem productive in a process which has little or no documentation through a lot of quick *Hacks*.[19] They may well believe the word *Hacker*[20] is a complementary term, for someone who is extremely good at solving problems. And this may well be its modern use. But originally it meant someone who has no self-discipline and simply hammers away at a problem till he or she gets a result.

In contrast, those who view software production as a science, and rely on *Forward engineering*, will promote higher-level tools from the beginning to the end of the **Event-Database Production Process**. As has already been mentioned the

highest-level tool is natural language. And with this, they will promote dialectic communication amongst the staff. And with that, they will promote the productivity of all the staff. And with that, they will acquire a natural leadership.

Some may ask, so if using higher-level tools, like a *game editor* or the **Event-Database Architecture**, produces higher productivity, then why is the **Event-Database Architecture** any more advanced or better than a *game editor*?

Well, firstly, it is not that the higher-level tool in and of itself produces higher productivity. It is the combination of the higher-level tool and natural leadership. That is to say the leadership of those who view software production as a science who acquire that natural leadership. So the question then becomes why is the **Event-Database Architecture** more advanced than a *game editor*, assuming a production process led by those who view software production as a science, in both cases?

Well the answer to that question is whether that *game editor* or the **Event-Database Architecture** promotes higher-level tools or lower-level tools. Modern *game editors* allow you to programme them using lower-level tools, i.e. programming tools. And thus, they promote lower-level tools. A lot of the functionality of a *Software Evolution Process* using these *game editors* in the Computer Games industry is typically encapsulated in extensions of the *game editors* called 'plugins'. Each 'plugin' is a *software library* created with a programming tool. A 'plugin' can be used, for example, to procedurally generate a football stadium and its contents in the *Game World*. And indeed one of these commercial 'plugins' created for this is called the 'Procedural Content Generation Framework'.

This 'plugin' allows you to generate points in the *Game World* where one or more 3D Meshes or Models may be placed in the *game editor*. This may be placed either at a random orientation aligned to these generated points. Or according to some formula which you write using more lower-level tools, i.e. a programming tool. You can use these to place static Meshes or Models of the crowds, seats, players, coaches, substitutes, the bench, the TV crew and the officials in the football stadium.

But you cannot use these tools to place dynamic or interactive **Game Objects** in the *Game World*. Like players on a football pitch or members of a crowd who move and are animated by some *Artificial Intelligence*. Nor can you edit the **Game Objects** generated by these lower-level tools, like the rest of the **Game Objects** you create in the *game editor*. You can only edit these with the same lower-level tools that generated these **Game Objects**. You cannot edit these **Objects** with other higher-level tools in the *game editor*.

In contrast, with the **Event-Database Architecture**, the **Game Objects** you generate can be edited with higher-level tools. Just like any other **Game Object** that has *Database Records* in the **Game Database**. And the **Game Objects** you create can be dynamic and interactive, like players on a football pitch or the members of a crowd in a stadium. In fact you could create a lower-level tool that generates the *Database Tables*, *Database Records* and *Database Fields* for the **Game Objects** procedurally. And then use higher-level tools to edit these **Game Objects**. As long as the lower-level tools and the higher-level tools can understand the *open data format* of the **Game Database**, they can interoperate with each other. And as long as the lower-level tools were only allowed to generate the **Game Objects**, not

to edit them, and instead that functionality was restricted only to higher-level tools, then the **Event-Database Architecture** would continue to promote the higher-level tools.

In the *software architecture* of *LPmud*, the **Master Object** is responsible for controlling the loading of all the other **Game Objects**. And that would be an appropriate place for any lower-level tool that generates other **Game Objects** procedurally to reside. Likewise, in the **Event-Database Architecture** for *LPmud*, the **Master Object** would also be the appropriate place for any lower-level tool or code to reside. That procedurally generates other **Game Objects**. The **Master Object** would be an appropriate place to add any **Actions** that should be performed in response to **Secondary Events** following on from any **Primary Events** to procedurally generate other **Game Objects**.

Finally, the **Event-Database Architecture** is a more advanced higher-level tool than a *game editor* because modern *game editors* are a product of a *Software Evolution Process*. Thus, they suffer from the same flaws as the games they are used to create, in another *Software Evolution Process*. They suffer from the same degenerative process, the same degenerative language and the communication problems that the *Software Evolution Process* produces. The **Event-Database Architecture**, however, is not a product of a *Software Evolution Process*. It, more precisely a game built with it, is a product of an **Event-Database Production Process** that is apart from the *Software Evolution Process*. And it has the higher-level tools and constructs, **Events**, **Actions**, **Game Objects**, *Database Tables*, *Database Records*, *Database Fields* and *Database Administrators* that can help address some of these problems.

3.5 OBSESSION WITH EFFICIENCY

Optimisation of software is about efficiency. As has already been mentioned, it is about the efficient use of computer software or hardware instructions, *data* storage capacity and *data* transfers. And the view of optimisation is one of the main differences between the two schools of thought in *Software Engineering*, one relying on *Forward engineering* and the other relying on *Reverse engineering*.

The former is sceptical about the efficiency of a software whose *software design* is not complete. And they do not believe that the software can be optimised or made efficient until the end of the software production process when it is complete. The latter is bullish about the efficiency of software. And they believe software can be optimised while its *software design* is not complete, during the middle of the software production process. And that this makes it easier to optimise at the end of the production process.

Even though they lack the clairvoyance to see into the future. And whether changes to the *software design* later on in the production process will make the changes they made earlier on to optimise the software, redundant or inefficient. Because alas they are just human. And like many in human society, they are blinded by an *obsession with efficiency*.[21] And this blindness can undermine the efficacy of the **Event-Database Production Process** just as easily as it does the *Software Evolution Process*.

3.6 DIVISION AND CONSISTENCY

Even if you do not understand how to write software, you can understand how errors in software come about. These errors arise from the same, universal reasons that affect any complex project. Understanding this can help you avoid these errors. It can also give you a better understanding of any process which tries to minimise errors. This includes the **Event-Database Production Process**.

Errors can be minimised in any complex project by simplification. There are two common forms of simplification used in engineering: division and consistency. The first involves breaking up any complex design into smaller parts. The second involves finding the common elements in each of these parts and constructing tools which you can consistently use to make these common elements. Any engineering project which cannot be simplified using these two methods will remain complex. And, because it remains complex, it will inevitably have errors.

Hence, the role of an engineer in a project would be to simplify the process used to build a product. And when you see any engineers trying to excuse the errors in a product, because of the complexity of the process which they used, they will merely be confessing their complicity. They will merely be admitting either their negligence to simplify the process, incompetence to do so adequately or callousness to proceed with production in spite of the fact.

Similarly, in *Software engineering* simplification, through division and consistency, plays an important role. If the set of features required for software has been divided into smaller subsets, then that would simplify the process of building it. A simple software component could then be built to meet each of the smaller subsets of requirements. This would also apply to the design of any tool used to build that software. Having several small tools, each with a small subset of the features, would be simpler to build than one large tool, with all of these features.

Therefore, during a production process for a computer game, when you see a *Game Designer*, *Game Artist*, *Game Producer* or *Sound Designer* asking for more and more features to be added, to the same software tool, they would all be acting recklessly. So would any *Game Programmer* whom you see contemplating such additions. These additions would make that tool more and more complex. And as a result, this will introduce more errors. A good example of tools, with more and more features added to them overtime, are the commercial *game-editors* used in the Computer Games industry. These are large tools which have accumulated a huge array of features overtime, which would be better implemented by smaller tools, each with a subset of those features.

The origin of such excessive additions would be an *obsession with efficiency*, about a process which used that tool. This obsession would have propagated a false belief that combining many features into one tool would simplify that process. But this obsession would actually produce inefficiency since these additions would introduce errors. A better way would be for them to have a limit on the number of features, in any given tool. And to spread these features across two tools, once this limit was reached.

This same principle would benefit the features of any stage in the Computer Game, which was being built jointly by a *Game Designer* and a *Game Programmer.*

If they added too many features to the same stage or level or part of the *Game World*, they would be acting carelessly. For example, adding too many options for a player to choose, onto the same menu, would be hazardous. So would be adding cosmetic features to the game. These would namely be features that did not move the game any nearer completion. Since either of these features had nothing to do with the *User Interface*. Or adding these features, to the *User Interface*, did not give any more information than what was already been presented.

So, for example, for a game with a complete *game design*, any addition to the parts already described in that design would be cosmetic. And for an *incomplete game design*, any addition which had nothing to do with the missing parts of that design would also be cosmetic. So would be, for example, playing back complex animations, or long pieces of music, on a menu. These would not only quickly become redundant, as the player's concentration learned to ignore these features. But these would add unnecessary complexity. And this complexity would introduce errors. It would be simpler for them to divide these features between two or more stages or parts of the *Game World*. So that the animation or music was, for example, heard or played on another stage or part of the *Game World*. Or for the animation or music to be heard or played on another menu, after the player had made his or her choices on the current menu.

Dividing the features of a Computer Game, or any tool used to build it, into smaller subsets would only have one merit. This would namely be in so far as that division simplified the production process. But any such division that resulted in a dependency between any two subsets, or increased the size of one of the subsets, would increase the complexity of that process. And as a result, that division would have no merit.

For example, suppose the *Game Designer* and the *Game Programmer* had decided that the large set of options on a single menu, in a game, were to be split into two. But one of the options was to be duplicated on the two new menus. This division would have no merit. Since one option, on one of the menus, would be dependent on the option chosen on the other menu.

Another example would be if the features of a software tool were to be transferred into the Computer Game. Suppose the *Game Artists* had three tools that could be used to build the animation for a game. One could be used to create and combine the different *Frames* of the animation. Another could be used to convert the *data* for the animation into an *open data format*, which could be both previewed and played back by the game. And a third tool could be used to watch a preview before the animation was added into the game. Now if these three tools were replaced by two new tools, to make the process of adding animation more efficient, this replacement may have no merit. It would have no merit if the features of the three tools were divided, amongst the new tools, so that it increased the complexity of those tools or the game.

For example, one of the new tools could be used to create the *Frames* of the animation. And the other new tool could be used to assemble the *Frames* together. This would record the animation in a *closed data format* that could only be played back by the game. And the game itself could be used to preview the animation. Now the *Game Artist* would only need to use two tools, instead of three, to add animation to the game.

But since this would add the ability to preview the animation, to the features of the game, this would increase the size of its subset of features. And, therefore, this would increase the complexity of the game. Also, one of the new tools would now assemble the *Frames* and convert the *data*, for the animation, before it was played back by the game. This would increase the complexity of the tools as a whole since these two features were performed by two separate tools before.

It would be an *obsession with efficiency* that would lead to this conclusion. Likewise, it would be an *obsession with efficiency* that would lead *Game Artists* to waste time, producing what they would euphemistically call mock-ups, of the *User Interface*, at the beginning of production. Even though these would be nothing of the sort and would more closely resemble makeshift sketches. Their mock-ups would lack the breadth and depth of detail to have any credibility with the rest of the staff. A real mock-up would have a one-to-one mapping with all the features of the final product: as in a mock-up of an aircraft or a car. But the mock ups they produce in the Computer Games industry would only demonstrate a tiny subset of the first version of the *User Interface* of the game, and a negligible amount of the final version.

Yet, even though these would be nothing more than makeshift sketches, these mock-ups would find their way, virtually unaltered, into the final product. As the production progressed, the *Game Artists* would just keep simply adding more and more mock-ups to their previous ones. And if any of these were rejected, they would respond quickly with more mock-ups.

The most graphic example of this has been games, which have been released, with 3D Models or images of the staff involved in its production. These models began life merely as makeshift solutions, which were later meant to be replaced with original Models. That is, these were meant to be replaced either with artwork inspired by the original themes of the game. These would include the different parts of the *Game World*, the characters and other items in each part, as these emerged during production. Or the artwork was meant to be based on famous celebrities, actors or characters involved in the sport, film, book or genre which gave the game its theme: not the staff. But the *obsession with efficiency* meant the mock-ups of the staff remained unaltered. And the *Artists* merely became preoccupied with crafting other mock-ups, afterwards.

So much so that the final artwork became one big mock-up, crafted by conflating at least three phases of production into one complex phase. The design, building and testing of the artwork would have all been rolled into one long, impromptu composition.

All this waste would partly stem from their desire to fill the excess time, they seem to have at the beginning of production, due to the concentration on drafting the *game design*. That, along with the excess time they seem to have, during production, due to the speed with which they produce mock-ups, would all pamper to their *obsession with efficiency*. But the effect of this would be that they would increase the complexity of their artwork and the production process as a whole.

Another effect of the *obsession with efficiency*, on the *Game Artists*, would be that there would be little or no documentation of their work. They would rely on the names they give the computer files they produce to document their mock-ups. And if one mock-up was made up of several components, such as a large complex collection

of 3D Models, they would rely on the brief names and numbers they give each component to make it self-explanatory. Even though this would erect a communication barrier between *Artists*. Such that it would be impractical for one *Artist* to complete the half-finished work of another. Instead, that *Artist* would find it easier to scrap any existing work and begin from scratch.

Indeed, in the Computer Games industry, *Game Artists* have tended to be limited to one small area of the game. Either they have been 3D Modelers restricted to modelling characters. Or they have been Environment Artists restricted to modelling other items in the *Game World*. Or they have been Character Artists restricted to animating the bodies and the faces of characters. Or they have been Character Riggers restricted to creating the skeletons or rigs for animating characters. Or they have been UI Artists restricted to creating 2D items for a **Graphical User Interface**. Or they have Texture Artists restricted to producing *Textures* for 2D or 3D **Game Objects** and so on. It has been very rare for an *Artist* in one field to venture into another. And there has been little occasion for one *Artist* to continue the work of another and little need to one *Artist* to understand the documentation for the artwork produced by another. Thus, fortunately most *Software Developers* have mitigated the waste that would result from their *Artists*' *obsession with efficiency*, at the expense of their freedom of expression.

But most *Software Developers* have yet to find a way to mitigate the waste produced by their *Game Designers*' *obsession with efficiency*. Indeed, they have not recognised it.

It would be an *obsession with efficiency* that would lead *Game Designers* to keep adding more and more cosmetic features, to the *game design*, as production progresses. Their original design would be hastily drawn up, full of vague ideas and riddled with implicit assumptions. Their desire to demonstrate, or see some demonstration of, the final product would far exceed their capacity to describe any of it. As a result, they would find themselves with loads of time and nothing to do at the beginning of production. And they would search for ways to fill this void.

Invariably, they would settle on either coming up with more and more cosmetic features: or rushing out ideas which have not been well thought through. Or they would harass other staff to clarify these ideas. That is, other staff would either be expected to quickly build and demonstrate parts of the game, based on the misconceived ideas in their original design. Or the *Game Designers* would throw their vague ideas at them, expecting the *Game Programmers*, *Game Artists* or *Sound Designers* to somehow magically clarify these ideas in their heads for them.

Nevertheless as the ideas, in their original design, became clearer through these demonstrations and exchanges, it would also become clearer, to the *Game Designers*, that they had more time on their hands. Since, because of this *obsession with efficiency*, they would quickly move on, assuming they were no longer required to clarify their ideas to the other staff. Even though the demonstrations they would have witnessed, from other staff, may have been filled with errors. And the exchanges they would have had may still leave the rest of the staff with many implicit assumptions and uncertainties.

So, after the first phase of production, the *Game Designers* would fill the void in the subsequent phases with more and more cosmetic features; based on the parts of the game that had been built or discussed with other staff. But, unlike the first phase,

in the subsequent phases, more and more features would be added concurrently that require other staff to clarify with time. So more and more of the *Game Designers*' vague ideas, full of implicit assumptions and uncertainties, would grow concurrently with time. And, as a result, they would increase the complexity of their *game design* and the production process as a whole.

This same *obsession with efficiency* would have the same effect on *Sound Designers* and other staff. The only difference from the *Game Designers* would be that *Sound Designers*' obsessions would be limited to the areas of the game related to music and sounds. And the obsessions of other staff would likewise be limited to their respective roles. But, in the less formal production processes of the Computer Games industry, even this restriction would be lifted. And any member of staff can come up with vague ideas, full of implicit assumptions and uncertainties for any part of the game design.

Along with division, consistency plays a major role in the simplification of software projects. Errors in software are known as *Bugs*. These are features of the software which exhibit behaviour which are inconsistent with the *software design*. Features of the software which reuse other software tools in general (e.g. *software modules, procedures, data* and *libraries*), in a manner inconsistent with how these were designed, are known as *Hacks*. *Hacks* are always difficult to maintain because *Game Programmers* use these inconsistently. The reason for the use of any given *Hack*, in one part of the software, differs from another. So even if you were to guess correctly the reason in one part, you would not be able to apply that generally across the entire software. What is more, you would not be able to distinguish between an intentional *Hack* and an unintentional *Bug*.

Thus, a *Bug* and a *Hack* share a close relationship. These both relate to an inconsistency in a design. The former relates to external, visible and software inconsistencies. The latter relates to internal and invisible inconsistencies. A software with a high degree of *Hacks* potentially has a high degree of *Bugs*. A symptom of software, with a high degree of *Hacks*, is that fixing one part of the system would subsequently cause a second part, which had not been altered, to inexplicably fail. This would be because the second part of the system was using software, from the first part of the system, in a way it was not designed for.

So, in a software production process, when you see someone asking a *Game Programmer* to add a *Hack* to solve a problem, this would be just like asking for a *Bug* to solve a problem. Or when you see a *Programmer* offering to *Hack* a solution, this would be in effect giving the option of adding a *Bug* to the problem.

Similarly, when you see someone asking a *Programmer* to modify the software to cater for a special case, this would be just like asking for a *Bug* to be added. It would be in effect asking for a change which would be inconsistent with the *software design*. However, it would not be a *Bug* if the *software design* were changed first.

The only other main avenue through which *Bugs* enter the software, excluding *Hacks*, is through lack of precision. This occurs when the software has not been written, by the *Game Programmers*, to do exactly what it was designed to do. It either allows too broad a range of possibilities by, for example, allowing a player to choose, at an earlier stage of a game, an option from a menu which should only become available at a later stage. Or it allows too little a range of possibilities by, for

example, never letting that option become available. Or it misses out a step in a process completely by, for example, doing nothing when the player chooses that option. Or it performs the steps in the wrong order by, for example, making that option available on the menu, at every one of the earlier stages of the game, except the late stage it actually was meant to be used in.

Having a complete *software design* would help stop such *Bugs*. But the alternative, an incomplete *software design* would cause such *Bugs*. So when you see someone asking a *Programmer* to reserve a place for a behaviour, which has yet to be designed, this would be just like soliciting the introduction of a *Bug*. This is sometimes known as adding a *Place-holder*.[22]

Some *Place-holders* do function and serve some purpose, rather than doing nothing. However, these *Place-holders* would be no better than those who do nothing. Since the same principle applies when you see a *Game Designer* asking a *Programmer* to add the basic details of a feature, which would be tweaked later. This would be, in effect, like requesting for a *Bug* which would be tweaked later. Just as with empty *Place-holders*, it would be asking for something to be added which did not do what was actually required.

Finishing such a *Place-holder* would be just like fixing a *Bug*. Except this would be a *Bug* introduced intentionally and typically at the start of a production process: not at the end. This would be a *Bug* to which no one could apply any rigorous, methodical procedure to test it, at the end of a project. Since this would be a *Bug* which had no criteria, set by a *game design* or a *technical design*, to meet.

These kinds of *Bugs*, such as *Place-holders* which cannot be detected, and *Hacks* which rely on using software tools inconsistently, undermine the software and cause errors. The negligence to simplify software, through the division of its features and of the tools used to build it, also contributes to these errors. However, you will find the staff following a software production process willing use these *Bugs*, and neglecting this division, on a day-to-day basis, in the Computer Games industry.

In the industry, the primary reason for using these *Bugs* would be to make informal changes to a game with an *incomplete game design*. And the primary reason for neglecting to simplify the software would be an *obsession with efficiency*, along with the desire to add cosmetic features to such games. On the one hand, the greatest and longest challenge, such projects face, will be completing the missing parts of the *game design*. On the other hand, the easiest and shortest challenge will be either to add cosmetic features to the parts that do exist. Or it will be to obsess over the efficiency of the processes used to build these parts. Or it will be to make informal changes to these parts.

So, naturally, reasons will be frequently found for meeting these easier and shorter challenges, while ignoring the longer ones. And, the resulting product would be one whose cosmetic appeal far exceeded its stability. One whose speed far exceeded its practicality. And one whose set of customisable parts far exceeded the completeness of this set. It would be a game with incredible attention to visible, audible and other cosmetic effects. But it would also be one riddled with critical errors that made it impractical to move consistently from any given point in the flow of the game to another, from one part of the *Game World* to another. It would be even impossible

to complete basic tasks, such as moving from the beginning of the game, to its end, without a terminal failure.

Given such results, could there ever be a good reason to introduce errors? Through *Bugs*, *Hacks* and *Place-holders*? Through an *obsession with efficiency*? Especially considering that these would be errors you could control? There would be enough sources of errors that you could not control, which you would encounter in a software production process. These include errors in third-party software tools, computer hardware, *software data* and human errors. So why add more? How can the staff begin testing software with both intentional and unintentional errors introduced into it? How can they distinguish between the two?

For in both the cases of *Place-holders* and *Hacks*, the effects of the introduction of such errors are the same. Both obscure meaning and intentions. Both create a communication barrier not only between the *Programmer* who wrote the software and any other members of staff testing it. These also create a barrier between any *Game Designer* or *Game Producer* who requested these errors and that *Programmer*. These create a barrier between any other members of staff, who want to understand the software and the *Programmer*. And these create a barrier even between any other *Programmer* who had to maintain or fix these errors and the *Programmer* who wrote these errors. So any process for producing software which relied on transparency and communication would, naturally, counteract such *Bugs*.

Such processes include those which depend on a complete *game design*, *technical design* and other comprehensive documentation. These include processes which depend on constant, open review of documentation, by all members of the staff, to ensure its transparency. These also include the software production process, based on **Event-Database Architecture**, which transparency and good communication which comes from a well-defined *Database*.

There will doubtless be many more processes which you could encounter, which cater for transparency and good communication. Whatever form these processes take, each will exhibit at least these three signs:

1. There would be more than one description of how any one part of the software should behave. This will be to ensure that the initial description, of that part, was clear and consistent.
2. *Bugs* would be located without referring to the computer files, written in a programming language, that were used to build the software. Instead, the descriptions of how the different parts of it should behave, written in a natural language, would be good enough to locate a *Bug*. This would be done by either simply observing the difference between these descriptions and what actually happens. Or a *Bug* would be located by simply checking for any discrepancies or inconsistencies between these descriptions.
3. *Bugs* would be fixed by simply modifying the description, of how the faulty parts of the software should behave, written in a natural language. The process would automatically maintain consistency between these descriptions of the software and how it behaves. So the process should automatically translate any such high-level changes, to changes in the low-level components used to build the software.

The *Software Evolution Process*, for example, fails all three of these criteria. Yet when those outside of the Software industry, in the rest of society, encounter such a process, where the communication between the staff seems incomprehensible, they come to a false conclusion. They reach a similarly false conclusion when they come across a production process which seems very complex. Namely, in the former case, they conclude that the staff have managed to elevate themselves to a level where natural language has become expendable to them. And, in the latter case, the staff have become so intelligent that very complex processes have become negotiable to them. Especially amongst the judiciary, many have been intimidated by such processes and come to such conclusions, when cases of negligence have been brought against *Software Developers*.

But there is no form of engineering which finds natural language expendable. And the very fact that, from outside of any process, you cannot comprehend the communication between the staff within it should be a sign to you that something was amiss. That the process had been riddled with communication barriers: items which defied description. That there had been some form of gross negligence to eliminate those barriers. The perpetrators of which would include at least the leadership of the staff along with some, if not all of the engineers amongst them, acting as accessories.

Furthermore, there is no form of engineering which builds on top of complex production processes without producing errors. Such processes always produce a seething mass of errors. Except that, in *Software engineering*, these errors would not only manifest themselves in the form of visible *Bugs* but also in the form of invisible *Hacks* and *Place-holders.*

3.7 THE MYTH OF SELF-DOCUMENTING CODE AND DATA

As many software products on the market will testify, it is possible to release software by using a scheme of *Bugs, Hacks and Place-holders*.[23] This is done by substituting another level of communication for the broken down high-level communication, which this scheme causes. But this new level of communication relies on a myth. This is the myth that the written *software code* or *data* itself can communicate the ideas behind it. This is the myth of the self-documenting *code* and *data*. It is important that an **Event-Database Architecture** should avoid this myth, in order to provide effective communication amongst the staff, producing a computer game.

The myth originates from a common human failure amongst *Programmers*. And due to the dependency on *Programmers* in the Computer Games industry, especially during a *Software Evolution Process*, it has been spread to other staff. Since *Programmers* have by and large been given seniority as *Team Leaders*, the *Game Artists*, *Sound Designers*, *Game Designers* and *Game Testers* have all been steadily indoctrinated into this myth. Likewise *Database Administrators*, through similar relationships with *Programmers* in other parts of the Software industry, have been indoctrinated into it.

As has already been mentioned, many of those involved in a project to build a game may suffer from an *obsession with efficiency*. Not least amongst these will be the *Programmers*. And this obsession leads to a tendency to be hasty in their work. More precisely, this leads to the desire to get the work done without knowledge of

how exactly to do it before they begin. And as the old proverb goes, '*Haste makes waste!*'[24] Just as a project that begins with an incomplete analysis of the problem (e.g. an *incomplete game design*) produces a lot of waste, so a *Programmer* (or any other staff) who begins work, without understanding the problem they want to solve, produces a lot of waste.

Rather than designing a *software module* to solve a problem, documenting it and then building it, a hasty *Programmer* will dive in. He or she will start by building it straight away, looking for quick practical results. Either the reckless presumption would have been made that the problem would be so simple, that a full solution could be formed mentally while listening to that problem. Or another equally careless presumption would have been made, that the partial solution being imagined was a prelude to a full one. And this full solution would arrive by the time the *Programmer* had finished writing the software. Unfortunately for the *Programmer*, his or her intelligence will rarely match either of these presumptions. What is more, there would be a big difference between considering a problem to be simple, which only involves your understanding, and making it simple, which involves the understanding of others as well. The prudent presumption you could make, if any, especially when working in a team, would be to always adopt the latter approach.

Inevitably, the *software module* which the *Programmer* hastily produces, in this manner, ends up changing again and again. A point will be reached where it becomes clear that some level of documentation would help. If not from education, he or she would know instinctively from the experience of having gone back to old work that some level of documentation helps. Since the *module* keeps changing so rapidly, however, the *Programmer* would foresee a lot of wasted effort documenting it fully. What would not be apparent to the *Programmer* would be the folly of the initial presumptions made about his or her abilities.

So, at this point, the myth of self-documenting *code* and *data* enters. The myth begins with the premise that programming languages (and the software used as tools by other staff) have features which allow you to express yourself in familiar terms, similar to natural language. These features include

1. *Interfaces* with instructions which have a similar meaning in spoken or written form (e.g. IF, WHILE, DO, FOR, EXPORT, IMPORT, OPEN, CLOSE, LOAD, SAVE, SELECT and EDIT),
2. the use of mathematical notation (e.g. +, - and =),
3. variable names for *software data*, which could be modified to fit into spoken or written form,
4. variable names for *software procedures*, which could be modified to fit into spoken or written form and
5. variable names for *software modules*, which could be modified to fit into spoken or written form,
6. variable names for the computer files used to build the software, which could be modified to fit into spoken or written form.

Since programming languages (and other tools used to build software) have these features, all you have to do will be to make sure the names you give your *software*

data, *procedures*, *modules* and computer files fit into naturally spoken or written form. You can enhance this further by adopting a *Naming convention*,[25] which ensures that the style of expression will be consistent throughout the software.

Thus, you can produce self-documenting *code* and *data* which describes the computer files, the *software modules*, the *procedures* or the *data* in a natural language. The purpose of a computer file or *software module* should be obvious from its name. The use of *data* or a *procedure*, within a *module*, should also be obvious from its name. The steps of a *procedure*, written in a programming language, should be obvious from each line of the instructions and the names of the *data* or computer files referred to in each one.

The more extreme form of this myth have been *Naming conventions* which even go so far as to claim to catch errors. For example, the *Hungarian Notation*[26] has attempted to catch errors by suggesting a convention which described the different types of *data* (e.g. numbers, words, dates, ages, times, costs, first names and surnames). As well as this, it has also suggested describing the size of *data* (e.g. the range of values for a number, the length of words, the range of dates, ages, times, costs, the length of first names and surnames). And it has suggested describing the *scope*[27] of *data*. This determines whether it can only be used locally, in a *software procedure* or a *module*, or globally, across *procedures* or *modules*. The idea has been that, by making these things patently obvious, you can avoid errors caused by transferring information between *data* of incompatible type, size or intended *scope*.

Naming conventions, like the *Hungarian Notation*, have been the highest point that the myth of self-documenting *code* and *data* has reached to date. The idea that you could develop a scheme which has the ability to catch some errors has been very alluring. However, at this height, the myth begins to become unstuck. Despite its promise, *Programmers* have not gotten away from the fact that a second form of documentation has still been required. Wherever a *Naming convention* such as the *Hungarian Notation* has been adopted, you would also find some formal scheme for adding descriptions to the *software data*, *procedures*, *modules* and computer files. This would normally be in the form of prominent *Comments*,[28] in natural language, next to each *data*, *procedure*, *module* or at the top of each computer file.

For other staff, such as *Game Artists*, the formal scheme would involve a table including a column listing the computer files the *Artists* were producing. And next to that column would be another listing brief descriptions of what artwork was contained in each file.

The necessity of these formal schemes would be due to the fact that, from experience, those who adopted this *Naming convention* have known about its inadequacies. But still enamoured by the myth, they felt compelled to patch up its flaws.

So *Programmers* would have reverted back to an old technique that has been used to describe software since the inception of programming languages; namely the ability to include *Comments*. Likewise others, such as *Game Artists*, would have reverted back to manually cataloguing their artwork, with simple tables typed up on the computer.

Thus, through *Comments* and other similarly peripheral techniques, the belief in the myth has continued. It has meant that *Programmers* could continue to believe,

that they do not have to ever put pen to paper and think away from the computer screen. They could continue to hurry their work and get immediate feedback.

In fact, they could use *Comments* as a means of avoiding having to write *User Manuals* too. By simply extracting the *Comments*, from the *software code*, they could feed these directly into a document that the software user could read. Indeed, there have been software tools which have been written to do just that, such as *DOXYGEN*.[29] These have collected the *Comments*, from the *software code*, and converted these descriptions into a document in a presentable *data format* (e.g. the Hypertext Markup Language or *HTML*[30]). This has allowed *User Manuals* and other documents to be compiled, with many passages and words highlighted in different colours and fonts.

Hence, the software has been automatically documented as the practitioners of self-documenting *code* and *data* have improvised with it. And by that characteristic and others, they have shown that they clearly belong to the second school of thought described earlier. Namely, the school that believes software production to be an art; an art of minimalism. To them, the myth of self-documenting *code* and *data*, the *Naming conventions* and the formal schemes for laying out descriptions have all embodied the aesthetics of this art: the art of minimalism.

But this art of minimalism is just a form of laziness. This laziness leads them to find the quickest way to do their work. This laziness leads them to another form of an *obsession with efficiency*.

This laziness makes them hate writing documents and designs using natural language and then writing code based on the designs with programming languages. So they find a way to combine the two into one. They find a way of writing code in programming languages and designs in natural language at the same time. And this inevitably results in them abusing features of programming languages and misusing them in a way they were never intended. But which they think is ingenious. Or as one Programmer in the Computer Games industry put it:

> '…laziness is the greatest virtue that a Programmer can have…'

Take for example the C++ programming language which is widely used in the Computer Games industry and Software industry. It is easy to be intimidated by the terminology of this programming language. But a lot of terminologies are just different words or phrases for the same thing. Take for example these features of the programming language:

1. Precompiled Header Files (which are just composites of code or C++ Header Files)
2. Unity Source Code (which are just composites of code or C++ Source Files)
3. Templates
4. Macros
5. Inline Functions
6. Pure Abstract Classes
7. Abstract Classes

It does not matter if you do not know what these terms mean. The only thing that matters is that you know that these are abused for the same purpose. That is to speed up the production of software.

You can abuse all of these features to combine two or more steps of the software production process into one step.

You can either try to conflate the documentation or design of a software component with its implementation (Templates, Macros, Inline Functions, Pure Abstract Classes).

Or you can conflate the implementation of two or more software components, or instructions, into one step and skip the documentation or design of these components or instructions (Templates, Macros, Inline Functions, Abstract Classes).

Or you can conflate the translation of multiple components, from the programming language to machine code, into one step (Precompiled Header Files, Unity Source Code).

And because you can do this, these features are very attractive to the lazy practitioners of the art of minimalism. That is so to say those who believe in the myth of self-documenting code. That is to say those with an *obsession with efficiency.*

Yet all of these features make the software production process more complex to develop, diagnose and maintain. And as a result, the abuse of these features produces more errors. So the best advice is to keep their use to a minimum and avoid them completely if possible.

However, because those who believe in the myth of self-documenting code love these features, and they abuse these features, you will find it all over the code they produce. They make out their prolific use of these features as a sign of their 'advanced' understanding of the programming language. When they interview other *Game Programmers*, or they write job descriptions, they are eager to find candidates with a similar *obsession with efficiency*, who consider the use of these features as a sign of 'advance' *Software engineering.* When in fact it is has nothing to do with *Software engineering* let alone a sign of 'advance' *Software engineering* but a sign of their degeneracy (see the definition of *Software engineering* in the Glossary).

3.8 SELF-DOCUMENTING USER MANUALS

Comments are very versatile tools with many benefits. One benefit is that they provide a way of documenting software quickly, allowing you to insert documentation right next to the *software code.* Another benefit is that you can extract them from the code and use them to create automated *User Manuals.* You can use them to write the *software code* for **Host Modules** and **Game Objects** of the **Event-Database Architecture**. And this can affect the *Quality Control* of the **Event-Database Production Process**.

Nevertheless, beginning with *Comments*, you can see how absurd the myth of self-documenting *code* and *data* is. From automated *User Manuals* to *Naming conventions* and, finally, the similarity between programming languages, *User Interfaces* and natural language, the myth is built by adding one fallacy onto another.

Ironically, as well as helping the myth, *Comments* expose more than anything else how absurd the myth is. Through *Comments*, *Programmers* have access to natural

language to describe themselves. Natural language has evolved over thousands of years and has a lot of flexibility. Programming languages have evolved in just over half a century and are rigid. The inadequacies of programming languages are self-evident when it comes to writing manuals.

Those who believe and those who do not believe in the myth both revert to natural language to write *User Manuals*. A *User Manual* describes why you would use a piece of software, how you would use it and, sometimes, for better explanation, how it will work. A programming language, however, can only at best describe how it will work. So, according to the myth, in order to produce automated *User Manuals*, *Comments* should be inserted into the computer files, and written in a programming language. And these should be attached to the *software data*, *procedures* and *modules*. Such that a third-party tool could extract these to document all the components of the software (i.e. the *software data*, *procedures* and *modules*). But these documents would be inferior to those written independently of the software.

The *Comments* used to produce automated *User Manuals* will invariably lack context. This will hinder the understanding of these documents. This will be because, in order to develop complex software, it will be broken up into small *software modules* to simplify the task. Each *module* will only be assigned a small part of the overall task. Each *module* will then be broken down further into a small set of *procedures* and *data*. The *Comments*, which the software user reads in the *User Manual*, will only describe the small part which each *data*, *procedure* or *module* plays in the overall scheme. So the user will find it difficult to tell why each should be used or how these should be used.

Programmers who believe in the myth of Self-documenting code invariably find it tedious repeating the context in which a set of *software data*, a *procedure* or a *module* will be used. In any given *module*, a set of *data* will be used more than once. In a project, a *procedure* or *module* will be used more than once in different contexts. So it will be far easier to simply mention what type of *data* each one will be, and how a *procedure* or *module* will work than to describe why and how it will be used.

As a result of this, any *User Manual* which relies on *Comments* extracted from the code will be difficult to understand. Any description of *data* will rely on knowledge of the *software procedure*, or *module*, these will be part of. Any description of a *procedure* will rely on knowledge of the *module* it will be part of. Any description of a *module* will rely on knowledge of the software or *software library* it will be part of or its uses in other *software modules*. Yet, despite all these interdependencies, each description of the automated *User Manual* created from *Comments* will often be set out on a different page.

When, for example, all the *Comments* about a set of *software procedures*, in a single *module*, have been put on the same page, there will be nothing to tie the descriptions together. The descriptions will not be ordered by how the *procedures* should be used, but the incidental order in which the *procedures* were added. The descriptions will not be grouped by the particular task the *procedures* should be used for, but the incidental grouping due to *modules* they were placed in. Using such a *User Manual* will be like trying to study a forest by examining the individual leaves of the trees. It will be like trying to understand a book by reading the index or the glossary pages.

If you wanted to produce a comprehensive automated *User Manual*, you would inevitably come into conflict with those who believe in the myth of self-documenting *code*. A good *User Manual* would be easy to understand by all, explaining why and how everything should be used. It would have a sensible thread linking the chapters and paragraphs. But this would require verbose *Comments*. To those who believe in the myth of self-documenting *code*, this would be a point of contention.

Comments should only be used for complicated sections of the *software code*, just to annotate these: or so they believe. The fact that you would produce a lot of *Comments* would be taken as a sign of over-complicated *code* in their mind They would not bother reading the C*omment*s. The mere sight of the descriptions would trouble their aesthetics of what they believe to be an art of minimalism. If you added a *Comment* for a simple instruction, which was obvious to anyone with their level of experience, they would find it irksome. Even though such *Comments* would, at worst, be underlining the *Naming convention* they would be using. And, at best, it would be providing a context for any obscure *software data*, *procedure* and *module* which was being referred to.

In contrast to those who do not believe the myth, when they write *Comments* they adopt a minimalistic style. And, instead of using the words of a natural language, they adopt sparse words from technical materials. That is to say, they adopt the words of programming languages, *Naming conventions*, the manuals of computer hardware, the instructions of microprocessors and other lower-level tools. And they fill their short *Comments* with abbreviations, acronyms, keywords and contractions from these materials: along with compounds made from these abbreviations, acronyms, keywords and contractions. Thus, they negate the advantage of being able to describe software, in natural language, through *Comments*.

Although the other staff, such as *Game Artists*, *Game Designers* and *Testers*, do not produce automated *User Manuals* from their work, it is only because they do not have the technology. If they had the technology, they would. Those of them who believe in the myth of self-documenting *data*, nevertheless, do show their propensity for such documents in the *User Manual* they help write for the final game.

Take the *Game Artists* who believed in the myth, for example. To them, it should be possible to produce items on menus, or in the *Game World*, whose reason and purpose were self-evident. And thus did not require an explanation in the *User Manual*. In the Computer Games industry, they would get ample opportunity to put this theory to the test. Since inevitably the large gaps in the *game design* at the beginning of production, would leave large gaps in the *User Interface*, which they would be expected to fill. And if you were to pay careful attention to the pages of the *Manual*, if any, describing their work, you would recognise the same symptoms that can be found in the automated *User Manuals* of *Programmers*.

That is, when the *Artists* create a design document or a *mock up* of the *User Interface* or a location of the *Game World*, you will find that the paragraphs of that document, the components of that *User Interface* or location in the *Game World* has no context. You would not find a common thread linking the paragraphs and clauses in these documents, or a theme in the annotations of the items of the *User Interface* or location in the *Game World*. You would find the propensity for abbreviations, acronyms and keywords: and compounds made from abbreviations, acronyms and

keywords in the documents and annotations. And you would find bar charts and line graphs with either the label for the X-axis, the Y-axis or both missing. You would also find lots of items on their menus, or locations in the *Game World*, with short pieces of annotations literally attached to each one. But this text would be hidden. And it would only appear either automatically, at set times. Or it would show itself when the item, it was physically attached to, was selected on that menu or approached by the player in that location. They believe that these annotations will make the items on the *User Interface* or the items in the location of the *Game World* self-explanatory.

And thus, the software user would be expected to metaphorically understand the forest by studying the individual leaves of the trees! That is, the player would be expected, by the *Game Artist*, to understand the *Game World* by exhaustively uncovering these pieces of annotations and deducing the basic functions of items on a menu or items in a location from this: rather than examining any coherent *User Manual*.

3.9 SELF-EXPLANATORY NAMES

The agitation, which those who believe in the myth of self-documenting *code* and *data* have with too many *Comments*, comes from the primacy they give to the *software code*. This primacy even affects the way the believers will informally communicate between themselves and other staff. They will find it hard to talk about *software data*, *procedures, modules* and computer files without literally using whatever cryptic *Naming convention* they have come up with. This will include crude enunciation of the unpronounceable syllables in the names formed by this convention. They will use these names to refer to the *data*, the *procedures* and *modules* documented in any automated *User Manuals* they produce; believing these to be self-explanatory.

You too may be tempted to use these *Naming conventions* to write the software code for the **Game Objects** and **Host Modules** of the **Event-Database Architecture**. Or to select the names of the *Database Tables*, *Records* or *Fields* in the *Relational Database* used by the **Architecture**. But despite the promises in this respect, *Naming conventions* deliver very little. And they inevitably lead to the degenerative natural language in a software project, which the **Architecture** is meant to avoid.

Firstly, the assertion that anyone will be able to tell the use of a piece of *software data*, a *procedure*, a *module* or a computer file from its name is false. Any name only gives you an abstract about its subject. It is a summary of information about a subject. It is a short substitute for the information you know about a subject.

Like all summaries, a name is incomplete and has some ambiguity. Like all substitutes, it can be replaced by other substitutes, and there is no innate relationship between a name and its subject. There is only a subjective relationship which depends on how a person looks at that subject. In the context of the Computer Games industry, it depends on how a *Database Administrator* or *Game Programmer* looks at the problem which he or she wrote the software for. Or it depends on how a *Game Artist* looks at a piece of artwork which had to be created. No two *Database Administrators*, no two *Programmers*, no two *Artists* and no two staff look at a problem in the same way. After long periods, even the same *Database Administrator* or

Programmer would not look at the same problem, in the same way. Neither would the same *Artist* look at the same work of art, in the same way.

The names you choose to describe a subject depend on the extent of your knowledge of that subject. The more you know, the more concise your choice will be. But the very reason *Programmers*, for example, will revert to using self-documenting *code*, in the first place, will be because they lack this knowledge. They do not know how exactly they will solve a problem with the software they are about to write. And they will be trying to save time, by not having to update a separate *software design* when each attempt fails, and they have to rewrite the software. So the ambiguous names they produce, in the self-documenting *code*, will naturally reflect their uncertainty about their work.

They could go back, once the software had been finished, and revise the names they had chosen. But the haste, which makes them adopt the myth of self-documenting code in the first place, will not allow them to go back. Instead they are in a hurry to move on to the next project Thus, they compound the difficulty of understanding their work. Not only do you have to understand the names they use, with little description of their perceptions of the problem they were trying to solve. But what little description there will be, will be done with vague names that reflect their past state of confusion. The names will either be too general. Or some names will be too identical to one another. Either you would not be able to tell which subset of the information you know about a subject, was being referred to by its name. Or you would confuse one name with another. Either you would require a word or a context to qualify the name of each subject. Or you would require additional words appended to each name so that you could distinguish between the names.

But once the names begin to require qualification, then the assertion that anyone will be able to tell the use of a piece of *software data*, a *procedure*, a *module* or a computer file from its names becomes false. It would not matter if you required additional words, to act as qualifiers, in the name of each subject. The result would be another name. Since a name is only a summary of information, if other staff use it in a way not covered by the summary, they would not see anything wrong with it. As long as the name was unique, and could be used to identify a subject, there would be no reason to change it. People will never see a reason to change it.

When you start adding a qualifier to each name, the names grow beyond being short substitutes. The names start becoming small phrases and sentences. The way the names should be chosen starts becoming a language in its own right that requires grammatical rules to be consistent. Each name ceases to rely on the programming language to provide its context, in the case of *Game Programmers*. Neither does it rely on the tools the other staff use, or their project, to give it context. Instead the name relies on grammatical rules to provide its context. And these rules introduce a new language, which is neither a natural language nor a programming language. And as a result, it slows down both the *Programmers'* writing and reading of the software. And it slows down any other staff creating or editing the *Game data*.

For example, suppose the staff had to build a game involving racing cars. And a *Programmer* had to write a *software module* which would be used to create a car, move it around in the *Game World* and manage its different parts. This *module* will

have a piece of *data* that will hold the properties of the car. The information will include the position of the car, its speed, the type of engine it has, the position of its wheels, the number of doors it has, and whether each door is open or closed. It will be reasonable for either the *module* or the *data* to be called 'Car'. You could either call the *module* 'Car' and qualify the *data* as 'Car Data'. Or you could call the *data* 'Car' and qualify the *module* as 'Car Manager'. It would be purely arbitrary as to why one choice should be made over the other. You would require rules to ensure consistency.

Suppose both the *software module* and the *data* were to be qualified as 'Car Manager' and 'Car Data', respectively. In this case, as previously, it would be clear that you wanted to draw a distinction between the *module* and the *data*. But it would not be clear as to why you wanted to do this. Would it be because one was a subset of the other? Or would it be because both were mutually exclusive members of the same set?

Consider the case where two *modules* were to be used to control a car wheel and a car door. Each would be called 'Car Wheel' and 'Car Door', respectively. In this case, the similar names would refer to the fact that these were both part of a set of car properties. But both were mutually exclusive. This would be unlike the case of the 'Car Manager' and the 'Car Data', where similar names meant one was a subset of the other: one was used by the other. Only by more grammatical rules could you make a distinction between both cases.

A car was chosen in this example to make it obvious how someone, who did not know anything about cars, could be mistaken by relying on the names of *software data*, *procedures*, *modules* and computer files. Most people do know what a car is. So they can see when someone, who does not know, has made a wrong assumption. If an obscure field were chosen as an example (e.g. modelling a physical world or creating sound effects in software), then the same people, who do know what a car is, would make the same wrong assumptions.

You could adopt grammatical rules to prevent people from making wrong assumptions. But only if you were seriously intent on creating a new language. However, *Programmers*, *Game Artists* and other staff, who believe in the myth of self-documenting *code* and *data*, will casually adopt such rules when they notice the flaws in a *Naming convention*. Although they would have neither the expertise, the time nor the patience to create a new language. And if such a prospect had crossed their mind, when they adopted the *Naming convention*, they would never have embarked on the venture in the first place.

Instead, their casual adoption of these grammatical rules will merely reveal their hypocrisy. For the rules they adopt will be used to cloak themselves with the appearance of self-discipline. But, at the same time, it will be hiding the impatience, which will be the source of the myth of self-documenting *code* and *data*.

3.10 SELF-CHECKING DATA

The second fallacy involving *Naming conventions* is the assertion that these will help catch errors when writing software. Prefixing or suffixing the names of *software data*, or computer files, with letters which describe the type, the size and the

scope of that *data* will help you catch some errors. These will be namely those errors caused by transferring *data* between two incompatible types.

You too may attempt to use *Naming conventions* to catch errors in how you write the *software code* for the **Host modules** or **Game Objects** of the **Event-Database Architecture**. Or how you select the names of the *Database Tables*, *Records* or *Fields* in the *Relational Database* it uses.

But the ability to give names to *data* was not invented for that! It was invented so that *data* could be abstracted.

That is to say, it was meant to give *Database Administrators* or *Programmers* the ability to name *data* using natural words from a natural language. These words were meant to have meaning to ordinary people: not computers. So that they would use the *data* as naturally as possible, when writing software, without worrying about how it was stored on the computer system. So that they would not have to worry about what any underlying computer software or hardware, of the system, was doing with that *data*.

It was clear from the beginning that the semantics of programming languages would reflect very little of the semantics of natural language. And as a result, these would be difficult to use. So it became important that the syntax or the words, including the names of the *software data*, reflected as much of natural language as possible. In the *first programming language*,[31] the ability to name *data* was intended for mathematicians to describe algebraic variables in formulas they wanted a computer to process. Later on, it was extended to be used by *Programmers* to describe any problem. It was meant to serve the encapsulation of an external problem within a computer system. It was not meant to serve the internal problems of the system, in which the *software data* was being used. Although this is well in the past, many *Software industry commentators*[32] continue to provide reminders, of the need to separate the dependency between different parts of a complex system.

Of course, back then as now, at some point someone had to make sure that erroneous *data* were not entering the computer system. But the tools used for programming could only check so much. Since these were made for a very wide variety of problems. And the *Programmers*, who believe in self-documenting *code*, will be reluctant to include checks, into the software, because they will be in a hurry. Therefore, they will revert to *Naming conventions*, such as the *Hungarian Notation*, which forces other staff to become aware of how *data* will be stored on the computer system. So that errors may be prevented when using that *data*.

But in doing so, these *Programmers* will negate the advantages of being able to name *data*. And they will also excuse themselves from the need to add checks for errors in the software, thus undermining its robustness. Likewise, the *Game Artists* and other staff who use cryptic *Naming conventions* to avoid possible errors caused by confusing computer files, or *Game data*, with natural names, will negate the advantage of being able to name that *data*. And they also will excuse themselves from checking for errors in it.

Nevertheless, as far as its ability to even prevent errors is concerned, *Naming conventions*, such as the *Hungarian Notation*, will be far from effective. This will be because the vast majority of different types of *data*, that will be used to write software, will be *Complex data types*.[33] These will be new types of *data* which have

been made up. That is to say, these will be compounds of two or more basic elements which computers easily understand (i.e. numbers, words and characters). And these elements would share some particular relationship. And these will require a set of *software procedures* to control access to that *data*. So that the relationship between its elements remains the same.

The reason for the prevalence of *Complex data types* will be because of the advantage these give to *Programmers*. This advantage will be the ability to extend a programming language. They could add more words to the language; words which capture constructs that have been derived from a problem. That is to say, these words represent different classes or sets of items they may see recurring, within a problem. And whenever they want to store or retrieve information about these sets, they could simply use these new words.

For example, if they were to see a problem within which a list of cars was mentioned very often, they could add a new word to represent such a list. *Complex data types* would allow them to aggregate *software data* together, without having to make up names for each individual piece of information. So they could add a new word to the programming language, which represents a list of cars. Each list would require *software procedures* to add and remove cars from it.

Another example would be if they were to meet a problem within which two or more items were always associated with each other. They could add a new word to represent this association. *Complex data types* would allow them to associate two or more related pieces of information together. So, for example, they could add a new word which represents a *Database Record*. A *Database Record* could hold the name of a car, its position in a *Game World*, its direction, its speed, its engine and how many doors it has. Each *Record* would require *software procedures* to control access to its *Fields*.

Any *data* which may be used to identify a *Complex data type* would also be another *Complex data type*. That *data* would merely be acting as a substitute for the other *Complex data type*. An example of this would be *data* that holds a *Primary Key* for a *Record*. Each *Primary Key* would require a *software procedure* that could be used to get access to its *Record*.

No *Naming convention* will have an adequate scheme for coping with *Complex data types*. This will be because there would be too many different types that *Programmers* could create, and there would not be enough letters in the alphabet to identify all of these types. Some people will try to cope with this by combining more and more letters, in the prefix of the name of each *data*. But then this will start to create another new cryptic language. So they will quickly stop making distinctions between *data*, after they have come up with a convention which identifies the basic elements. That is, they stop making distinctions after they have come up with a convention which identifies numbers, characters, words, the size of numbers, the range of numbers and the scope of *data*. Sometimes they may make distinctions between locations in computer memory that hold numbers, characters or words as well, if the programming language allows you to read or write directly from computer memory.

Nevertheless, since *Naming conventions* will make little or no distinction between *Complex data types*, which are by far the largest categories of *data* you will come

across in any large project, like making a Computer Game, it will still be possible for errors to occur in the *software*. It will still be possible to copy *data* between incompatible types by mistake. It will also be impossible to tell, from the name of the vast majority of the *software data*, exactly what type of *data* each one will be, without having to refer elsewhere. But once the *Programmers*, who believed in the myth of self-documenting *data*, were to do that, there would be no point in having a *Naming convention*. They could give their *data* a natural name and look up this same reference if they wanted to know what type each one was.

As for *Game Artists*, in the Computer Games industry and other staff who believe in the myth, their faith has not so far extended to include *Naming conventions* such as the *Hungarian Notation*. They have rightfully found it obscure, cryptic, contrived and unnatural. Nevertheless, they have implicitly believed that it was possible to adopt a *Naming convention* which could prevent errors with *data* in a computer system. The tools they have traditionally used have produced computer files with three-letter suffices (.png,. jpg,. avi,. mpg,. mkv,. fbx,. tiff). And they used these letters to identify the *data* the tools read in and wrote out. The *Game Artists*, for example, have naturally got into the habit of using these three-letter suffices, even within their verbal language. And if you were to introduce them to a new system, which did not use these letters, they would find it obtuse. Even though the new system would be no more prone to errors than the traditional one that they were confident with. Even though this traditional system would be based on the same principles as the *Hungarian Notation*, which they find so alien.

Other staff, on the other hand, have no such reservations. Closely related to the faith they have in the *Hungarian Notation* is the fetish which some *Programmers* have for *data* being *type safe*.[34] That is, software written in a way that every piece of *data* has been categorised into a group. And each group has been clearly defined to the programming tools. So that, when the software was built, these tools could detect when *data* was being transferred between two incompatible groups (or data types). This would suggest that the software was incorrect, as with *Naming conventions*. And the tools could then either warn the *Programmers* or stop building the software. But, unlike *Naming conventions*, this would happen automatically, without the need for the *Programmers* to read the software and interpret it.

Accompanying this fetish for *data* being *type safe* is another fetish they have for all *data* being strictly defined within its *scope*. That is, even within the individual steps of a *software procedure*, *data* should only be defined just prior to the first step within which it was used. And it should not be defined after the last step. This again is so that the programming tools could automatically detect when *data* was being incorrectly used. If *data* were being used by parts of the software, or even parts of a *software procedure* where it had not been defined, this would suggest the software was incorrect. This would suggest that either another piece of *data* should be used, the step in a *software procedure* using the undefined *data* should be omitted or some previous step which produced the correct *data* had been omitted. Hence, those who believe in the myth of self-documenting *code* and *data* believe that strict, draconian adherence to these two techniques not only ensures the correctness of software. But that *data* which was *type safe*, and only used within its *scope*, can convey the semantics of the software using it as well.

However, this belief stems from a fallacy. Whether *data* was *type safe*, or limited in *scope*, these techniques would still be properties of the programming tools: not the *Programmers*. These could only help these tools with syntactic errors: not semantic ones. It would still be possible for these techniques to be used erroneously by someone who misunderstood the semantics of the software.

Consider the previous example. Suppose a *Programmer* was writing a game involving racing cars. And the *Programmer* defined every group of *data* used by the software. This included different groups for all the cars belonging to different manufacturers, different groups for all the cars with different engines, and wheels, different groups for all the different models and so on. Now suppose, before the race began, one of the cars had to be chosen from a list of those available for that race to act as a Safety Car. And this list was composed of *data* from one of the groups the *Programmer* had defined. The Safety Car could be any car, from any manufacturer, with any engine, any number of wheels and of any model. As long as it could lead the procession of contenders, around the racing track once and bring them back to their start positions. And it was unique in that race. That is to say, there was no other car like the Safety Car in the race.

Since all the *data* of all the cars were in a clearly defined group, the software would be *type safe*. The programming tools would automatically detect any confusion between cars and other items in the *Game World*. And the *Programmer* could easily choose one car at random from the list available for that race to act as a Safety Car.

But that choice could be wrong and may not be unique. The *type safety* of the *data* of all the cars does not provide any way to identify a unique car. Therefore, the programming tools that check the *type safety* of the *data* cannot detect when a Safety Car was chosen which was not unique.

There would be no way another *Programmer*, who did not know why the software was written, could tell what a Safety Car was, by examining the definition of the groups of cars. And there would be nothing stopping this second *Programmer* from making the same mistakes as the first one who wrote the original code.

Conversely, it is still possible to write software which is correct, containing no errors, even though it is not *type safe*. It is also possible to write software which is correct, containing no errors, even though none of the *data* has been restricted in its *scope*. As long as the *Programmers* who write this software have the self-discipline to verify what they write, they can achieve these goals. And they would have no need for these techniques.

Only if they lacked the self-discipline would *data* which was *type safe*, or limited to its *scope*, have any bearing on the *Quality* of software they produce. And even then the effect would be marginal. It would not save them from themselves. It would not give them the self-discipline which they lacked. It would only help the programming tools draw their attention to possible errors: not genuine errors. Invariably, the tools provide mechanisms for ignoring whether *data* was *type safe* or being used within its *scope*. And once undisciplined *Programmers* grew tired of the red herrings they were being drawn to, they would use this to override the tools.

Like the *Hungarian Notation*, at best, these two techniques would only indicate when incorrect *data* was possibly being used, at a particular point in the software.

These would not identify which *data* was incorrect. And even after the *Programmer* had identified this *data*, these techniques would not indicate what the correct *data* should be. So there would be no way these techniques could possibly convey the semantics of the software.

For their part, *Game Artists*, and most of the other staff in the Computer Games industry, have had no conception of either *data* being *type safe* or limited in *scope*. They have never had the tools to enforce such concepts. And it is to their credit that they have been spared the consequences, even though they may be blissfully unaware of the fact. Otherwise, the self-righteous ones amongst them, who believed in the myth of self-documenting *data*, would have crassly enforced their colleagues' compliance with these techniques. As has been the case amongst *Programmers*.

In the case of *Database Administrators*, this has already happened to a limited extent. Their tools have provided them with techniques comparable to *Programmers*, but not in the same form. They have had the ability to strictly define the *Fields* in a *Database Record*. So that each is guaranteed to be either a number, a date, a word, a name and so on. They have had the ability to restrict the range of these numbers, dates, and the length of the words or names in each *Field*. All of which has been comparable to making *data type safe* in programming. They have had the ability to restrict the access of software users to certain *Records* in a *Database*. They have had the ability to restrict who can read, edit or delete each *Record*. They have also had the ability to restrict how long a *Record* lasts in a *Database*. All of which has been comparable to limiting the *scope* of *data*, in programming. Thus, it should come as no surprise to find that, amongst *Database Administrators*, the myth of self-documenting *data* persists, just as it does amongst *Programmers*. And they believe that self-documenting *data* is an adequate substitute for documenting a *Database* in natural language.

3.11 NATURAL LANGUAGE AND PROGRAMMING LANGUAGE

But is the myth of Self-documenting *data* true? Can you construct your *software data* in such a way, in a programming language or in a *Relational Database*, used by the **Event-Database Architecture**, that it prevents errors? And the software production process or the **Event-Database Production Process** does not require natural language? Which is better at preventing errors? A natural language or a programming language?

Whichever of the two is better at expressing ideas and can reach a larger audience is better at promoting understanding. Whichever is better at promoting understanding is better at reducing errors due to a lack of understanding. And whichever is better at reducing errors is better at preventing errors. Let us compare a natural language and a programming language with respect to expressing ideas beginning with references and how you can refer to subjects in both languages.

The need to look up a reference is a common thread throughout the different stages of the myth of self-documenting *code* and *data*. Whenever you look at a feature on a page of an automated *User Manual*, you will always need to refer to another page. This page will provide the context in which that feature should be used, without which you will not be able to understand it. Whenever you come across a project using a *Naming convention*, you will have to refer to some *dictionary*,[35] which describes

what each name means. Whenever you come across the *Hungarian Notation*, you will have to refer to some *Coding standard*[36] document, which describes what the letters, which prefix each name, mean. Or, as you will probably be encountering a *Complex data type*, you will have to look up where that *data* has been defined. So it will come as no surprise that, when it comes to the foundation of the whole myth, yet another reference will be required.

The foundation of the myth is the similarity between natural language and programming languages. There are *High-level languages*[37] that set out to be like natural languages (e.g. *BASIC, FORTRAN, COBOL* and *SQL*[38]). But these languages are conspicuously absent in the Computer Games industry. The very fact that these languages are so verbose is taken as proof of sluggishness and inadequacy. The languages used in the industry, by contrast, are characterised by brevity. These languages are characterised by how little in common they have with natural language.

These programming languages do use a few words from natural language. But when these are used, the words have only one meaning. Whereas, in natural language, these words have several meanings. Also due to the grammatical differences between programming languages and natural language, some phrases can be misleading or have unintentional implications. You can see these differences if you were to try to convert every word, and notation, from these programming languages to natural language. What you end up with would be a pidgin language like the Program Design Language (or *PDL*[39]).

For example, take this simple set of *PDL* statements:

IF ***A*** IS TRUE THEN
B
ELSE
C
ENDIF

In natural language, this could mean one of two things. Either if ***A*** is true, do ***B***, and if that fails, do ***C***. Or it could mean if ***A*** is true, always do ***B***, otherwise always do ***C***. In a programming language, these statements could only mean the latter.

The richness (i.e. double meanings in the words) of natural language allows it to be used economically to express complex ideas. But it takes far more lines of a programming language to express the same set of ideas. So understanding ideas written with its words and instructions takes more time and involves wading through more lines than you would with natural language.

Consider this description of a round of golf in natural language:

> A round of golf takes place over a course of 18 holes. At each hole, each golfer has to hit the ball, from a designated area on the course, the Tee, to the hole, which lies in another area, the Green. Each golfer takes it in turn, to take their shot. And their score would be determined by how many strokes they used to putt the ball. The lower the number, the greater their score would be. The golfer with the best accumulative score in a round, starts each hole. But, after the Tee, the one whose ball was furthest away from the hole, takes the next shot before the others.

Now although this description looks simple in natural language, it has in fact got many implicit double meanings. The word 'take' for example was used multiple times, with subtly different meanings.

It was used to describe the relationship between the game and a location in the *Game World*;

> A round of golf takes place over a course of 18 holes.

It was used as an adjective to describe the mutually exclusive relationship between golfers;

> …Each golfer takes it in turn…

And it was used as a verb to describe the relationship between the golfer and the ball;

> …Each golfer….to take their shot.

In a programming language, you cannot use a single word to describe all three relationships. And especially in the Computer Games industry, the words of the programming language could only be verbs, not adjectives. That is, the words could not describe the abstract relationships between items in a round of golf. Instead, these would only describe the sequence of rudimentary actions that occur in a round, for example

```
START ROUND
WHILE NOT END OF ROUND
    GO TO NEXT HOLE
    WHILE NOT END OF HOLE
        IF AT THE TEE
            WHILE GOLFERS UNFINISHED
                GET NEXT GOLFER WITH BEST SCORE
                LET GOLFER TAKE SHOT
                IF BALL LANDS IN HOLE
                    REMOVE GOLFER FROM UNFINISHED
                ENDIF
            END WHILE
        ELSE
            WHILE GOLFERS UNFINISHED
                GET NEXT GOLFER FURTHEST AWAY
                LET GOLFER TAKE SHOT
                IF BALL LANDS IN HOLE
                    REMOVE GOLFER FROM UNFINISHED
                END IF
            END WHILE
        END IF
    END WHILE
END WHILE
END ROUND
```

Even these lines of *PDL* statements only give a brief summary of what would be required in a programming language. More would be required to further explain some of these statements at a level which a computer could understand. More lines would be required before it could be practical to use it to build a game. Yet already the number of lines and the number of distinct components exceed the equivalent description in natural language.

As another example, consider a game where a car would be moved into a garage, then onto a driveway, then onto a road. This would use a *software module* to manage the car. The *module* would have a *software procedure*, which would allow you to move the car. And this *procedure* would use *data* which described the new location of the car. The names of the *module* and *procedure* would be 'Car' and 'Move Into', respectively. The *data* for the three locations would be called 'Garage', 'Driveway' and 'Road'. Now in natural language, to move the car, the instructions would be

move the car into the garage
move the car onto the driveway
move the car onto the road.

But in practice, in the *software code*, these would read

Car Move Into Garage
Car Move Into Driveway
Car Move Into Road

Firstly, these would all be grammatically wrong. Secondly, by insinuating that you could move cars into a driveway and a road, you would unintentionally be implying that these locations could somehow contain a car, in an enclosed space, like a garage could.

Another difference between programming languages and natural language is that both of these use the same forms of punctuation and notations but with different meanings. For example, some programming languages use the same symbols for mathematical operators like

equals
plus
minus
greater than
less than
you would use in natural language, for example
=
+
−
>
<

But the vast majority of the operators these languages use are unrecognisable: like those for

multiplication
division
logical OR[40]
logical AND
e.g.
*[41],
/
||
&&.

In mathematics, there is a branch called *Basic Set Theory*. In this branch, braces ({}) are used to list a set of items, for example

{Peter, Mark, Luke, John}

This represents a set of names. But most programming languages have no support for *Basic Set Theory*. So they use braces for other things. For example, to list a sequence of instructions to be executed, for example

```
{
Peter(),
Mark(),
Luke(),
John()
}
```

This represents a sequence of instructions to be executed called 'Peter', 'Mark', 'Luke' and 'John', in that order. In *Basic Set Theory* the order of the names in the set makes no difference. In programming languages typically used in the Computer Games industry, the order of the names makes a big difference.

Most programming languages also have no direct provision for using certain mathematical notations, like the one for a square root. So these languages use specially named *software procedures*, with short acronyms instead, for example

sqrt().

Thus, ultimately, you would have to look up a reference to understand software documented using programming languages. Experience would only give limited help to a *Programmer*. Programming languages were made to be extensible and flexible for a wide range of problems. Even with a *Coding standard*, you could not anticipate how people would perceive a problem. And that primarily determines how they extend the language. You would need at least one or more secondary references,

apart from the instructions of the programming language they have written, which explains their perceptions.

Depending on your experience, this reference may be other *software modules* that used their software. But if these secondary references also did not have any separate documentation, these would suffer from the same limitations as reading any piece of self-documenting *code*. You may have to refer to a book on programming or the person who had written the *module* you were interested in. But how useful would such references be when limited to so few members of staff? Could a document or explanation by *Programmers*, for *Programmers*, communicate anything but the rudimentary details of programming?

The same quandary arises in the case of *Database Administrators* or other staff who also believe in the myth of self-documenting *code* and *data*. It arises amongst *Database Administrators* who believe it is possible to choose a set of names for *data*, which were self-explanatory. It arises amongst the *Game Artists* who believe it is possible to choose a set of names, for the commands of a *User Interface*, whose purpose was self-evident. Or annotate items in locations of the *Game World* that made them self-explanatory. This is nothing more but another variation of the myth of self-documenting *code*.

The set of names they choose for their *data*, or the commands of a *User Interface*, or annotations of items in the *Game World*, have little meaning without a second reference. These have no meaning without a reference, which explains the perceptions of the person who chose those names and the problem which he or she was trying to address.

In the case of the commands of a *User Interface*, and annotations of items that appear in the *Game World*, this has a direct effect on the *User Manual*, which would be based almost entirely on the *User Interface*, and the appearance of items in the *Game World*. After reading this *User Manual* (i.e. the 'self-explanatory' names that the *Artists* gave to commands in the *User Interface* or annotations of items in the *Game World*), the users of the software will find themselves lost and confused. Since the *Manual* will be minimalistic, full of the assumptions of the *Game Artists* who made the *User Interface* or the *Game World*. This *Manual* will be full of Didactic step-by-step instructions, which the users or players will be expected to follow to the letter, without variation or innovation.

Any innovations would be limited to two choices. Either the players would be expected to use some form of *Reverse engineering*. And by examining the lower level qualities of the game, the names of the commands of the *User Interface*, or the annotations of items in the *Game World*, the players would be expected to infer the higher level design. That is the assumptions of the *Artists* who made up these names or annotations. Or the players would be expected to seek out some second reference to understand these assumptions.

Notwithstanding that this requirement, for a second reference, renders the primary one (i.e. the *User Manual*) redundant, the players will invariably find the secondary one also redundant. It will, invariably, lead to the players searching the Internet for articles to explain the *Game World*. And all they will find are articles written by someone who holds natural language in contempt. Either it would be written by a *Game Artist* for another *Artist*. Or it would be written

by one very experienced player or 'hardcore gamer' for other 'hardcore gamers'. Could a document by one *Artist*, for another *Artist*, convey anything but the rudimentary details of graphic design? Could a document by one hardcore gamer for another hardcore gamer convey anything but the rudimentary details of hardcore gamers?

In a project led by those who believe in the myth of self-documenting *code* and *data*, it would be no use relying on the *Programmer* who wrote some *software module*, or tool, as the second reference for all other staff involved in the production of the game, who want to use that *module* or tool. Nor would it be any use relying on the *Database Administrator* who made a **Game Database**, as a second reference for all other staff who want to use that *Database*. Nor would it be any use relying on a *Game Artist* who made a *User Interface* or some item in a location in the *Game World*, as a second reference to all the other staff who want to use that *User Interface* or item in that location. Since they would all rely on self-documenting *code*, data, commands of the *User Interface*. or annotations of items in the *Game World*. But, as already stated, the names of instructions of programming languages, the names of data, the names of commands of a *User Interface* and the annotations of items in the *Game World* are all very base and weak forms of expression. These would at best tell you how some of the components of software work. These cannot convey why, or the context in which, those components should be used. For this, you would need natural language.

The programming languages used to write games, and the commands of the *User Interface* of games, rely on a series of imperative commands. This is just the same as a commander giving a sequence of orders to the troops. The troops are not meant to understand the orders. They just follow the orders. Therefore, writing self-documenting *code*, or creating the commands of a *User Interface* without documentation, and expecting others to understand it, would be like giving orders to troops. But these troops would in fact be either your colleagues doing collaborative work. Or they would be customers who have paid for the privilege of being condescended to while playing your game.

Perhaps those who give credence to the myth do indeed believe they are commanders, leading troops into war. If so, then they would only be fighting against themselves. For if ever a programming language or *User Interface* came into being, that was anywhere near as expressive as natural language, this would mark their downfall. This would result in monumental economic and cultural changes. Not only in academic studies and commercial trade but in almost every part of society. Since the ability of computers to interpret natural language is one of the major hurdles in the field of *Artificial Intelligence*.

Once this hurdle had been surmounted, the decline and fall of *Programmers* would swiftly follow. *Database Administrators*, *Game Artists* and other staff would also become redundant shortly afterwards. That is, once this *Artificial Intelligence* had been taught how to manage *Databases*, draw artwork or any other skills required to make Computer Games. Anyone could communicate directly with a computer without the need for such professions, especially *Programmers*. But there is no such breakthrough in sight in *Artificial Intelligence*. And the continuing rise in demand for *Programmers* shows no sign of abating. These two facts, alone, demonstrate the

sheer scale of the dizzying heights of delusion that the myth of self-documenting *code* and *data* commands.

If you were to rely on the *software code*, the names of the *data* the commands of the *User Interface* or annotations of items in locations in the *Game World* as a main form of documentation, you would not be able to tell when that *code*, data, command or annotation had been wrongly written. Since these would be the primary form of documentation, the way these would be written, and the software would behave, could not be wrong. Even software which was partly implemented, and left unfinished, could not be touched. Changing the behaviour of the software would be filled with apprehension because no one would really know what the effects would be. No one would really know whether it behaved as it did for a reason.

Each change would become a melodrama, in which the plot could take sudden unexpected twists, as the resultant software suddenly improves or deteriorates. Without warning emotions amongst the staff would swing from one extreme, violently, to another: one moment in ecstasy, the next in despair. And all manner of heated arguments over the software, with copious amounts of hand waving, would take place. Inevitably, this would end with accusations, up to and including slander, being exchanged between members of staff.

A separate reference solves all of these problems. A *software design* and other documentation, written in natural language, would give you the confidence to make changes to the *software code*, *data* or its *User Interface*. But, more importantly, it would allow you to reach beyond the narrow confines of those who believe in the myth of self-documenting *code*. and *data*. It would allow you to reach all the members of staff and all the players in your *Game World*.

So it would be paramount that any process for producing software, which would rely on communication, should not rely on the myth of self-documenting *code* and *data*. It would be important that the **Host Modules**, of an **Event-Database Architecture**, should also not rely on self-documenting *code* or *data*. Any **Game Database** constructed for a game should also not rely on self-documenting *data*. And neither should the *User Interface* of the game or items in the *Game World* be assumed to be self-explanatory.

NOTES

1. ***Database design sources.*** Handbook of Relational Database Design by Candace C. Fleming and Barbara Von Halle.
2. ***Finite State Machine.*** A method for designing a computer system based on two basic concepts: that the system has a well-defined set of states and that there exists a well-defined set of events connecting any two states.
3. ***Desktop computer.*** An International Business Machines (IBM) personal computer (PC) or compatible model. It was designed for business but is now popular as a Home computer too.
4. ***Scalable.*** A software which can vary its performance depending on the resources it has available. And, thus, it can be used on a range of computers, with different speeds, sizes of memory and other levels of resources. *Scalable components.* A software procedure or data that can vary the time and space that it uses. See Glossary.
5. ***Home computer.*** A computer system designed for home use, e.g. playing games, music, learning or small business software.

6. ***Games industry commentators.*** Some Software Developers keep an up-to-date version of their computer games on Desktop computers, even though they never release this version. See Glossary.
7. ***Small devices (with Relational Databases).*** Relational Databases have been used with software deployed on mobile phones. See Glossary.
8. ***Ordered software system.*** A system of software components that has been assembled according to some principles. And therefore can be progressively disassembled, using the same principles, without causing errors when the software is rebuilt. See Glossary.
9. ***So...there are no obvious deficiencies.*** Quotation by C. A. R. Hoare, a computer scientist best known for his discovery of a widely used procedure for quickly sorting items of data. He later became a Professor of Computing at Oxford University, in the UK.
10. ***Forward engineering.*** The process of building a software product (or any manufactured product) in four phases: analysis, design, implementation and testing.
11. ***Reverse engineering.*** The process of rebuilding a software product (or any manufactured product) in four phases: re-testing, re-implementation, re-design and re-analysis.
12. ***Software engineering.*** A systematic, disciplined approach to software production. It was devised to cope with large projects which no one individual could undertake to deliver in a timely, secure fashion. See Glossary.
13. ***High turnover of staff.*** Very few of the staff of the Software Developers, in the Computer Games industry, stay there or in the industry as a whole, for more than a few years. See Glossary.
14. ***Design principles.*** A pre-emptive statement at the beginning of a design document that sets the rules for providing a solution to a problem. See Glossary.
15. ***Design patterns.*** A general description of a solution to a common design problem. In software production, design patterns usually refer to solutions which have been built using particular programming languages. Namely, those that support a technique known as 'Object-Oriented Design'. See Glossary.
16. ***Heuristics.*** A set of rules, based on educated guesses, that limits the search for solutions. These are intended to increase the probability of solving a problem, which is not well understood. See Glossary.
17. ***Benchmark.*** A test to measure the performance of computer software, hardware or components. These are used to compare the relative performance of competing products. See Glossary.
18. ***Time complexity (of an algorithm).*** This relates to how much longer it takes an algorithm to solve a problem as the size of that problem increases. That is to say, how much longer would it take a theoretical software procedure to perform its task when the size of that task increases? See Glossary.
19. ***Hack.*** A quick job that produces what is needed but not well.
20. ***Hacker.*** Someone who works by using Hacks. A Programmer who writes software not by planning, but by misusing the design of software, software tools, programming languages, computer hardware and different techniques to achieve a quick result. See Glossary.
21. ***Obsession with efficiency.*** In a software project, not only the engineers involved may become obsessed with efficiency. Other staff may become obsessed too. See Glossary.
22. ***Place-holder.*** A software procedure or software module, which acts as a substitute for a feature which has yet to be designed. That is to say, it has no clear requirements to meet. It either does nothing or only partially implements the feature.
23. ***Bugs, Hacks and Place-holders.*** External and internal software errors. See the subchapter entitled Division And Consistency.
24. ***Haste makes waste!*** From John Ray's 1678 proverb collection.
25. ***Naming convention.*** A written convention for naming software data, procedures and modules. The names should give helpful information about the use of each, in order to avoid errors.
26. ***Hungarian Notation.*** A Naming convention that was invented by Charles Simonyi, a Hungarian, while at the Microsoft Corporation. See Glossary.

27. ***Scope (of data).*** The limited block (i.e. software procedure or module) where software data may be used. This helps protect the data from erroneous changes, allows reuse of the same name for the data, in other blocks, and simplifies each block by limiting the data to that block.
28. ***Comments.*** Text embedded in software code, ignored by the computer, which is merely there to help explain the use and function of software data, procedures and modules. See Glossary.
29. ***DOXYGEN.*** A software tool used to generate documentation for software from the set of computer files used to build its software modules. See Glossary.
30. ***HTML.*** Hypertext Markup Language. A programming language for describing documents displayed on the World Wide Web.
31. ***First programming language.*** The first programming language that allowed you to name data was Formula Translation (FORTRAN), created by the International Business Machines Corporation (IBM) in 1957.
32. ***Software industry commentators.*** Accompanying many tools which have been introduced into the Software industry, those who have made this introduction have stressed the importance of keeping the components of a computer system as independent as possible. See Glossary.
33. ***Complex data types.*** The translation of simple constructs (i.e. nouns), in a natural language, into complex constructs (i.e. software modules), in a programming language.
34. ***Type safe (of data).*** In theory, software which is type safe has sets of data which have been so well defined that it is possible for the tools, which use that data to build software, to automatically recognise erroneous steps within it. Hence, it is impossible for the software to be incorrect.
35. ***Dictionary (of names).*** A list of definitions of the names of software data, procedures and modules used in a project. In practice, there will not be a single list. The definitions will be spread throughout the software code. You may need to ask the Programmers involved, what each data or procedure name means.
36. ***Coding standard.*** A document used in software companies. It outlines the Naming convention, and other guidelines, to follow in order to produce software of a consistent, maintainable standard.
37. ***High-level language.*** A programming language which tries to use natural language words and grammar in order to be easy to understand and use.
38. ***BASIC, FORTRAN, COBOL, SQL.*** Beginners All-purpose Symbolic Instruction Code, Formula Translation, Common Business Oriented Language, Structured Query Language.
39. ***PDL.*** Program Design Language. A language for producing structured software designs, created by Caine, Faber and Gordon Inc.
40. ***OR, AND.*** These are logic operators used in programming languages to test when either one of two conditions (A or B) has become true. These are also used to test when both (A and B) have become true.
41. ****, /, ||, &&, sqrt().*** These are all mathematical notations, software procedures and logic operators used in the programming language 'C'.

4 The Nature of the Beast

As previously mentioned, when a software production process, including the **Event-Database Production Process**, is led by those who view software production as an art, and rely on *Reverse engineering*, there will be a recession. A recession away from higher-level tools (including natural language) to lower-level tools which *Reverse engineering* depends on. And with the receding of natural language, just as in the *Software Evolution Process*, just as in the *Tower of Babel*, comes chaos.

Within this chaos, the sudden moments of clarity, and burst of high productivity, which they achieve through *Reverse engineering*, will look impressive to almost everyone. And this impression will secure their objective. That is to say, they will acquire an unnatural leadership, because of their impressive productivity, which they could otherwise never aspire to. As a result, they will be promoted in a hierarchy, above their peers.

In contrast, when the **Event-Database Production Process** is led by those who view software production as a science and rely on *Forward engineering*, there is a promotion of higher-level tools. The highest-level tool is natural language. And with the promotion of this tool, there is a promotion of dialectic communication amongst the staff. That is to say logical arguments from different points of view to reach the truth about some subject. This normally takes the form of a dialogue. And with this dialogue, there is a promotion of the productivity of all the staff. And with that, those who promoted the higher-level tools acquire a natural leadership. And they too will be promoted in a hierarchy above their peers as a result.

But notice the two different ways these promotions occur and how these hierarchies form. In the first case, the hierarchy forms because no one else but a small band of people have a superior knowledge of a project. And therefore this band is more productive than anyone else. And they are promoted as a result. In the second case, the hierarchy forms because another band of people promote dialectic communication amongst the staff. And therefore all the staff exhibit the same level of knowledge of the project. At least to the point that you cannot distinguish the superiority of the knowledge of one band over another. And as a result, they are all equally as productive. And the band of people who promote this dialect communication are promoted as a result.

In the Computer Games industry the former hierarchy is the most common form. And this is the hierarchy that the **Event-Database Production Process** will probably be employed. Lead by the *Game Producers*, *Game Programmers*, *Game Designers*, and sometimes 'technical' *Game Artists* who form the small band of people who exhibit superior knowledge. Often this band will have a preliminary meeting amongst themselves prior to any general or wider meeting of the rest of the staff. They discuss the subject in the general meeting amongst themselves using their superior knowledge and present a united front in that general meeting. And you would be forgiven for thinking that because they exhibit this superior knowledge in a project that therefore their leadership is a natural leadership.

 DOI: 10.1201/9781003502807-4

But as has already been explained, the truth is very different. It is a symptom of an unnatural leadership that the leadership obfuscates the truth. And this obfuscation makes it hard to judge the success or failure of any software production process it uses, including the **Event-Database Production Process** or a *Software Evolution Process*. And whether this success or failure was due to a natural leadership.

So you need to understand how to recognize the symptoms of a natural leadership and unnatural leadership. In order to judge the success or failure of the **Event-Database Production Process** or a *Software Evolution Process*.

One symptom of natural leadership that distinguishes it from unnatural leadership has already been mentioned. That is in a natural leadership there is no band or group of people with superior knowledge. At least to the point where you can detect the knowledge of one group as being superior to another. In an unnatural leadership, there is a distinct gap in knowledge, between those who conduct the unnatural leadership and those outside of that band. This means, in the **Event-Database Production Process**, in a natural leadership all the staff will have the same knowledge of the process and the constructs of its language i.e. **Events**, **Actions**, **Game Objects**, *Database Tables*, *Database Records* and *Database Fields*. In an unnatural leadership, only a small band of the staff will have a superior knowledge of the process and the language.

Therefore, if you give all the staff an *Event-Database Architecture Knowledge Test*,[1] a multiple choice test. Where each question asks them to select the meaning of the names of different **Events**, **Actions**, **Game Objects**, *Database Tables*, *Database Records* and *Database Fields*. And each answer is one of a list of multiple options they have to choose from. And the correct answer is the one that matches the description of that item in the *data design*. Then the score for all the staff should be evenly distributed in a natural leadership. This is evidence that the **Event-Database Production Process** has succeeded. In an unnatural leadership, the distribution of the score will be a Normal distribution, with a narrow band that obtained a very high score. This is evidence that the **Event-Database Production Process** has failed.

Another symptom of natural leadership that distinguishes it from an unnatural leadership is that natural leadership produces a functional hierarchy. And an unnatural leadership produces a dysfunctional hierarchy.

A functional hierarchy depends on the leaders of the hierarchy. It depends on the ability of the leaders to promote higher-level tools. As has already been said, the highest-level tool is natural language. The success or failure of the hierarchy depends ultimately on the abilities of the leaders at the top of the hierarchy, to promote natural language and dialectic communication amongst the staff.

A dysfunctional hierarchy depends on the followers of the hierarchy. It depends on their ability to follow didactic communication. In the form of art, literature, design or meetings which were meant to be instructive. The success or failure of the hierarchy ultimately depends on the abilities of the followers at the bottom of the hierarchy. To follow the instructions in the art, literature, design or meetings. Organised by a small band of people with superior knowledge who are called ‘leaders’. But crucially these are unnatural leaders. And the followers are unaware or denied the privileges of their role. That is they are ultimately responsible for the success or failure of the hierarchy.

Therefore, if you organise a meeting of all the staff to discuss a new *game design*, or some changes to an existing *game design*, and take minutes of the meeting. And if you were to ask all the staff to give their logical arguments for or against the new *game design*, or for or against a change to the *game design*. Then the minutes of the meeting should show all of the logical arguments of all the staff, under a natural leadership in a functional hierarchy. This would be evidence of the success of the **Event-Database Production Process**. In an unnatural leadership, that is to say in a dysfunctional hierarchy, not all of the arguments will be in the minutes. And some of the staff cannot or will not make an argument. Or the arguments will not be logical. Or their arguments will be interrupted by other staff who have superior knowledge. This would be evidence of the failure of the **Event-Database Production Process**.

Another symptom of a natural leadership or functional hierarchy that distinguishes it from an unnatural leadership or dysfunctional hierarchy is the nature of its leadership. If it were a functional hierarchy, this leadership would be based on a cult of personality. If it were a dysfunctional hierarchy, this leadership would be based on a vicarious leadership.

A natural leader or a cult of personality leads by example. The leader demonstrates, by personal example, how a piece of work should be done, how the followers should behave towards each other or how the followers should treat a client. The followers look, listen and learn from the demonstration and act accordingly. As has already been mentioned, those who view software production as a science who become natural leaders in the **Event-Database Production Process**, promote higher-level tools. This includes natural language which is the highest-level tool, in the art, literature and designs they produce through *Forward engineering.* And the followers likewise follow suit and adopt that natural language to communicate with each other.

In this way, the natural leadership or cult of personality serves the followers. They endear themselves to the followers. This endearment creates a cult of popularity amongst the followers. That is to say, it creates a following based on mutual trust and respect for the natural leadership that allows the hierarchy to function.

In contrast, in a dysfunctional hierarchy, the leader does not lead by personal example, but vicariously: through the example of others. The leader does not rely on personal demonstrations but on the demonstration of others. As has already been mentioned, those who view software production as an art, and become unnatural leaders in the **Event-Database Production Process** or *Software Evolution Process*, promote lower-level tools. And the use of these lower-level tools for *Reverse engineering*, to understand other people's demonstrations.

These demonstrations may come from the education of the followers. Or these may come from the past experiences the followers have had in their careers. Or these may come from the followers' peers. Or these may come from other competing products. Whatever the sources of these demonstrations are, the leader assumes, from these other sources, that the followers implicitly know how a piece of work should be done, or how the followers should behave towards each other, or how the followers should treat a client.

Since unnatural leadership or vicarious leadership does not perform demonstrations for the followers, the leader does not depend on an open relationship with

them. Instead, the leader need only provide minimal information to complete a task, to prompt the followers to act. The leader relies on the followers making assumptions based on other people's demonstrations to know how to complete that task. Furthermore, since the leader assumes that the followers implicitly acquire the knowledge to perform the work, from these demonstrations, the leader has no need to engage in a dialogue. The leader needs only provide the follower with a monologue of expectations. Even if the leader wanted to, he or she simply could not engage in a dialogue. The reliance on the demonstrations and understanding of other people make this impractical for the leader.

Therefore, if you were to take a task which has already been completed in a functional hierarchy, and give that task to the leadership, and ask them to repeat that task, then they will produce the exact same result as before. This will be evidence that the **Event-Database Production Process** has succeeded.

But if you were to take a task which has been completed in a dysfunctional hierarchy, and give that task to the leadership, and ask them to repeat that task, then they will not produce the same results as before. They will produce different results. This will be evidence that the **Event-Database Production Process** has failed.

Aside from the cult of personality or popularity of the leadership, there are other virtues of natural leadership that distinguish them apart from an unnatural leadership.

In a functional hierarchy, a cult of personality or popularity of a natural leadership has at least eight virtues. Beginning with the one that gives leadership popularity, the leader has

1. an extraordinary reputation
2. persuasion
3. charm
4. courage
5. a sense of honour or integrity
6. a vision
7. a *good memory*[2]
8. a strength of character

The last of those qualities, a strength of character, is the most important. It is the basis for all the other virtues.

In a dysfunctional hierarchy, a vicarious leadership lacks a strength of character. And as a result, all of the other virtues are replaced by substitutes. The leader has

1. an exaggerated reputation
2. a dependency on coercion
3. a schizophrenic personality
4. an aversion to risks
5. cynicism
6. a permanent state of emergency
7. a short-term memory
8. a weakness of character

In a functional hierarchy, the leader has an extraordinary reputation, which comes from an association with some extraordinary achievement. This is not a reputation that has been acquired prior to joining the company or hierarchy, in another company or hierarchy. This reputation cannot be acquired from working in another company that released a highly commercially successful or popular game. But a reputation that has been acquired within the same company as the rest of the staff on the project. So it cannot be a recent appointee or convert so to speak, or someone the staff does not know personally. But someone whom the staff have got to know over time. Someone who has shared in their successes and failures. And their achievement is widely recognised as extraordinary within the staff.

For example, in the context of the **Event-Database Production Process**, this achievement could be promoting higher-level tools, especially the highest-level tool i.e. natural language. And helping all the staff in the project achieve the same level of knowledge.

In a dysfunctional hierarchy, the leader has an exaggerated reputation. Lacking the strength of character to achieve anything extraordinary, the leader will manufacture accomplishments. Rudimentary accomplishments will be conflated with exceptional ones.

For example, when errors appear in the **Event-Database Production Process**, say with the first step the Feasibility Study, the leader will conflate any records kept, or investigations carried out, that reveal these errors, with good leadership. In the first step, a minimal game, based on the **Event-Database Architecture**, is meant to be built for the target *platform*. And that game is then tested with respect to the minimal features of the **Architecture**. However, suppose after the game was built and tested, nothing was being rendered on the screen. And after an investigation by the leader, or more likely someone delegated with that task, it was revealed that there was an error starting the **Graphics Host**. The leader will conflate this with good leadership and avoiding problems in the future in the project. They will make a big show of this result, and present this in a meeting with the staff, or the client they were working for, as some kind of extraordinary accomplishment.

However, this is rudimentary. The whole purpose of the Feasibility Study is to uncover such rudimentary problems. The leader did not design the Feasibility Study. It was designed and written down by someone else. And it was implemented by other staff. The leader will not be helping anyone but him or herself by carrying out these checks. Through these checks, the leader will be able to assess the quality of the leadership's decisions. Through these checks, the leader will avoid repeating mistakes in the future that will make him or her hate the job. Therefore, no one should be grateful to the leader for not covering up the mistakes of the leadership. And the leader will deserve no credit when any records, or investigations, reveal the leader's negligence.

In a functional hierarchy, natural leadership has the power of persuasion by dialectic communication. That is to say, the leader frequently engages in logical arguments from different points of view about a subject in order to reach the truth. The dialogues that the leader habitually engages in help the leader acquire the ability to make very persuasive logical arguments through natural language. For this is the way in which those who view software production as a science, and rely on *Forward*

engineering, begin the process. With a dialectic dialogue which is written down in the design documents. And this is the way the leader resolves conflicts that would arise during the **Event-Database Production Process**.

Therefore, if you were to give the leader a subject from the **Event-Database Production Process**, the leader should be able to present logical arguments for that subject from at least two different points of view. This subject could be an **Event**, **Action**, **Game Object**, *Database Table*, *Database Record* or *Database Field* that the leader had any awareness of.

For example, suppose the leadership took part in an **Event-Database Production Process** to build the game *LPmud*. And you were to ask what is the reason for a single **Game Object** called the **Master Object.** The leader would explain that this was necessary because you needed one **Game Object** that would respond to the **Primary Initial Reset Event** of the **Event-Database Architecture**. And would use that to control the Loading of all other **Game Objects**. And you also needed one **Game Object** to generate the **Primary Heartbeat Event** for other **Game Objects** to periodically respond to perform some **Action** e.g. damaging characters in one round of combat or recovering the health of a character who was resting from combat. And you needed one **Game Object** to monitor when all other **Game Objects** were Loaded or Unloaded from computer memory. And send the **Primary** and **Secondary Object Loaded Event** and **Primary** and **Secondary Object Unloaded Event** to the **Objects** that were Loaded or Unloaded from memory.

The leader would also present arguments against the **Master Object**. The leader would argue that the **Master Object** plays too many roles in the **Event-Database Architecture**. All of the **Game Objects** in the **Architecture** were meant to be generated from one simple rule. And perform a small part of the overall flow of the game. Thus making each **Game Object** simple. But the **Master Object** seems like an exception to the rule. It responds to two **Primary Events**: **Initial Reset Event** and the **Heartbeat Event**. And it sends multiple **Primary** and **Secondary Events**: **Heartbeat Event**, **Object Periodic Reset Event**, **Object Loaded Event** and **Object Unloaded Event**. Therefore, the **Master Object** should either be removed and the game somehow be constructed without it. Or it should be replaced by multiple simpler **Game Objects**, each playing one of its multiple roles.

However, an unnatural leadership will not be able to present logical arguments from two or more different points of view on the same subject.

In a dysfunctional hierarchy, the unnatural leadership lacks the power of persuasion. The leadership views software production as an art and relies on *Reverse engineering* not *Forward engineering.* And that in turn means the leadership relies on the didactic. That is to say, the leadership relies on art, literature, design or meetings which are meant to be instructive. And the ability of the rest of the staff to follow instructions. So the leader will not be used to making persuasive logical arguments about a subject. Instead, the leader will adopt coercion when questioned by the staff.

The more subtle form of this coercion manifests itself through a long, reoccurring, never-ending state of emergency. That would be declared in order to get the rest of the staff to sacrifice themselves and do overtime to meet some milestone. And the more harsh form would manifest itself through a bullish, hard-nosed behaviour. More precisely, the leader will cycle between three phases.

In the first phase, the leader will make a genuine attempt to present a desperate situation, which requires the followers' help. This attempt will include the leader's emotions and some of the details of the situation; namely, those parts which seem to justify the leader's anxieties.

For example, suppose there was an impending milestone to deliver some slice of a game about Skateboarding to the Game Publishers or financial backers by Monday. On Friday morning, the leader will announce that the company has committed to showing a slice of the game to the client on Monday. This game takes place in a city where the player can join gangs of skateboarders by performing impressive tricks and earning credit. The slice of the game that was meant to be delivered to the client was just meant to show the player moving around one of the suburbs of the city, and performing tricks in the local skateboard parks, and along the pavements along the bungalows that line the streets of the suburb.

But unfortunately, the leader does not believe that this will be delivered on time to the client. To the standard that they will make a good impression on the client. The leader will list the secondary characters that inhabit the *Game World* that were missing or incomplete, the bungalows that were missing, the different animations of the player performing the tricks on the skateboard that were missing, the parks that were missing and so on. The leader will kindly ask for some people to volunteer to work over the weekend, for a few hours, to ensure that the quality of the game meets the required standard.

This will leave a strong and lasting impression on some of the followers. For the very first impression they will have, of the leader, will be that of a hapless, wounded and vulnerable figure. From that time on, they will find it hard to shake off the image of someone caught between the crossfire of the demands of the followers and the demands of any superiors or clients. They will agree to come in on the weekend for a few hours to ensure that the game achieves a high quality to impress the clients.

However, come Monday, there will be no announcement of the reception of the game by the client. Some of the staff will make inquiries about it in the afternoon and find out that the meeting has been postponed till Wednesday. So it turns out that the staff have slightly more time to polish the game and raise it to a higher quality than they thought. Eventually, come Wednesday the news will eventually slowly slip out that the client has seen the latest slice of the game. They are happy with it. And they have given the go ahead for the next milestone. Everyone will breathe a sigh of relief and wonder how perilously close the game was to failing that milestone. The answer to that question will be unclear.

Nevertheless, after several successful attempts with this form of coercion, the leader will grow confident enough to move on to the next phase.

In this second phase, the leader will only make melodramatic attempts to present a desperate situation. These will only include the leader's emotions but none of the details of the situation.

For example, in the next milestone, the game should show more of the *Game World*. The opportunities and areas in which the player can skateboard should grow and include a more built up inner city area, with lots of grimy polluted streets with congested traffic. The player will be running through office spaces and run-down residential estates with lots of high-rise blocks, fenced-off apartment blocks with

swimming pools, broken down cars, heavy goods vehicles, police cars, public buses and smashed windows. Again there will be lots of opportunities for the players to try out their skills at skating, between the cars in traffic, along the congested public pavements with lots of pedestrians, in fenced-off public playgrounds or public parks, next to the residential estates. And in this urban landscape, there will be many edges, along the pavements, railings, walls, and fences along the streets, around the apartment blocks, swimming pools, parks, and playgrounds, that the player can hop on with the skateboard and grind along.

Again on Friday morning before the deadline for the next milestone, the leadership will announce that another deadline is approaching on Monday. And that they were anxious and concerned about it. And that this was really important. The leader will ask for the names of those who would not be available to work over the weekend to make sure that their work for the next slice of the game was completed by the deadline. This time around, there will be an expectation for staff to agree to come in over the weekend and to explain why they will not be coming over the weekend if they cannot make it. At the same time, there will be no sense of how many building blocks that were missing, how many vehicles that were missing and how many pedestrians were missing. There will be no sense of how many animations of the pedestrians walking or the player grinding along various edges in the urban landscape that were missing.

Again, come Monday, the staff will find that there will be no announcement of the results of the meeting with the client. Those who inquire will find out that the meeting again has been postponed to later on during the week. Giving the staff more time to polish the 'deliverable' before it is shown to the client. And come that day during the week, news will slowly slip out that the game was well received by the client. Again everyone will breathe a sigh of relief and wonder how perilously close the game was to failing that milestone. Again, the answer to that question will be unclear.

However, when the leader becomes overconfident, he or she will move on to the third and final phase.

In this phase, the leader will make no attempt to even pretend to persuade the followers. He or she will simply present a list of features of the *game design* that the staff were expected to deliver for the next milestone. A minority of these features will come from requests made by the client. But the majority of these features will be ad hoc, impromptu suggestions which the leadership added. To practice the art of the software production. To show off their creativity in that art to the client. To market the game to the client.

But to the staff all of these expectations on the list of features will be presented as demands of the client. The followers will be given no explanation of how these features fit into the overall vision of the final *game design*. Or how these fit into the overall plan for the production process that the leadership has to deliver the game to the client. For this would give away that the majority of the expectations were from the leader and not the client. The best that the staff will get will be a list of dates for upcoming milestones and a vague idea of the next expectations of the leadership at these dates.

Occasionally, one of the followers may question these expectations, either explicitly by asking for more details about the phases of the production process in between these dates. Or implicitly by refusing to do overtime or work over the weekend to reach the next milestone. At which point, the leader may respond by finding common errors or mistakes in that follower's work. Of which there will be plenty, especially in a *Software Evolution Process*. Due to the impromptu nature with which the changes are made to the *game design*, which in turn comes from the view of software production as an art. And the leadership may use these errors or mistakes to make false accusations of negligence. And follow that up with an explicit threat to either discipline or dismiss that follower.

Unfortunately for the leader, this phase will periodically backfire. Either some of the followers may refuse to cooperate because of this harsh discipline to themselves or to other staff. Or they may rush their work producing more errors or mistakes in the process. Or they may become too tired from doing overtime which results in them producing more errors or mistakes. And this will cause the leader to revert back to the first phase.

In a functional hierarchy, a natural leadership has the charm that does not require these dramatic changes of persona. This charm comes from the way in which the leader persuades the followers by engaging them in logical arguments from different points of view. From the way in which the leader eloquently makes their arguments using natural language and engages them as friends, with the same level of knowledge. Friendship is only possible amongst equals.

In a dysfunctional hierarchy, this is just not possible. The unnatural leadership has a superior level of knowledge to the rest of the staff, which is the means by which they acquire leadership. Therefore, they do not and cannot treat the rest of the staff as equals. And lacking the charm of natural leadership to persuade through logical arguments from different points of view, the leader will adopt a schizophrenic personality instead.

The first part of this personality will be evident in the first of the three phases of coercion discussed earlier. That is to say, the leader will present the image of a wounded, vulnerable character. The second part of this personality will be evident in the last of the three phases of coercion. That is to say the leader will present the image of an intimidating, strong and macho disciplinarian. But the leader will quickly drop that persona, for the first one, whenever the leadership realises that the followers are in danger of being alienated.

In a functioning hierarchy, as has already been said, natural leadership leads by example. Before any task is given to any of the followers, the leadership partially performs that task. And this example acts as the basis for the followers to imitate and complete that task. Therefore, if there were any new problems in a software production process, such as the **Event-Database Production Process**, the leader would naturally be the first one to face it. And the leader would get into the habit of being the first to face any problem. The leader gets into the habit of assessing the risks of his or her own decisions to perform some task. And this assessment comes from partially performing the beginning of that task. This would develop the maturity of the leader.

That is to say this would develop the courage to make decisions and face the risks of those decisions on their own. This would also help prevent the leader from being vulnerable to flattery by those who exaggerate or underestimate the risks for one reason or another.

Furthermore, the courage to act alone would become useful when the leader, by virtue of his or her position, first encounters a problem. Although the followers may subsequently help the leader, there still would be an initial period when the leader stands alone with the problem. If the leader had the courage to act alone, then he or she would remain calm and not make rash decisions.

For example, in the context of the **Event-Database Production Process**, in a functional hierarchy it would make sense for the natural leadership to conduct the first step of the process, the Feasibility Study, on its own. To encourage that leadership to assess, in that study, the risks of the decision to develop a minimal game or small cross-section of the final game, based on the **Event-Database Architecture**, on some target *platform*. And to get that leadership into the habit of facing problems on their own and into the habit of using their courage.

But one of the main features of the **Event-Database Architecture** is how it promotes communication amongst the staff using natural language. And therefore you could argue it would be counter-productive to only have a small band or a single person conducting the leadership on its own. You would not be able to assess in that case the communication amongst the staff when using the **Architecture**. So a better way would be for the natural leadership to begin the feasibility study but let the rest of the staff join in to complete the study together. And at the end, you will be able to assess the efficacy of the **Architecture** to facilitate communication in natural language. As well as whether the game produced passes the standard minimal test of the **Architecture** on all the target *platforms*.

By contrast, in a dysfunctional hierarchy, the unnatural leadership does not lead by its own example, as has already been explained. Instead, it leads vicariously, by the example of others. Therefore, instead of making decisions and facing the risks of those decisions, the leader will make decisions and let the followers face the risks of those decisions. This will be presented as an opportunity for the followers to be more proactive, to anticipate and steer the future direction of a project. The leader will nominally let the followers make strategic decisions but without the authority to see these through. The followers will also lack the vantage point to see the long-term effects of these decisions, on the overall plan for the production process.

The leader will retain the authority, the vantage point and the financial reward, which were meant to complement the ability to make these strategic decisions. The leader will limit his or her role to mere delegation of responsibility and enforcement. Like a foreman in charge of hired hands, doing unskilled work. Like a slave driver, on a cotton plantation.

For example, in the context of the **Event-Database Production Process**, the leader will not conduct the first step, that is to say, the feasibility study. That is to say the construction of a small minimal game or cross-section of the final game, based on the **Event-Database Architecture** on the target *platform*. And assess whether it will be possible the build the larger final redundant game, given the time and resources available to do it. The leader will delegate responsibility for this to the rest

of the staff or a small band amongst the staff. They will face the problem on their own. They will conduct the feasibility study and come up with a plan to build the rest of the game and schedule for this plan. This plan will include all the tools and staff required to build the *data*, (i.e. artwork, animations, models, sounds, and music etc.) in the tools design. And this will include all the data that will be required in the **Game Database**, in the *data design*.

However, the leader will call a meeting to discuss the feasibility study and its results. And in that meeting, the leader will unilaterally, revise the plan and cut the schedule, if some aspect turns out to be unappealing. There may be a proviso to the plan, or a time limit to the schedule, which the leader may or may not know beforehand. It may be that the game has to run on two or three computer hardware instead of one. Because the clients wanted to reach new lucrative markets of some new computer hardware or game consoles. It may be that the schedule has to be cut down from 18 to 6 months. Because the client needs to show a near-complete demonstration of the game at some Video Games conference in the coming summer or autumn.

But instead of being honest about this, the leader would either keep this information secret, only to reveal it at the last moment in the meeting. Or when it was not practical to do so, the leader would pretend that the revision was providing a service to those who conducted the feasibility study. Either they will be expected to believe that the revision was a multilateral agreement. To help them perform their job better. Or the leader will pretend he or she originated the revision. And the followers will be expected to believe that the real reason for the revision was incidental. It just happened that the leader decided to release the game on a next-generation game console and cut down the schedule from 18 to 6 months to make the game available for demonstration at a Video Games conference in the summer. And the staff that conducted the feasibility study and came up with the original schedule to deliver the whole game in 18 months ended up having to face the risks of the decision to deliver the whole game in 6 months.

When it gets towards the end of the schedule and the game is in danger of missing that final milestone, the cuts made by the leader will be forgotten. It is not the leader who cut the schedule will be held accountable. Since the leader lacks the courage to face the risks of that decision. It would be the staff who came up with the original schedule. They would face the risks of that decision. They would be the ones expected to give up their free time and weekends and do overtime to ensure a successful outcome.

Revision is not the correct term to describe what will take place. It will be more like amateur dramatics. A real revision would require the leader investigating the original problem and personally conduct a second feasibility study. It would be irresponsible to rely on just looking at the schedule produced from the initial feasibility study. The schedule produced may have been over-optimistic.

If the schedule were accurate, then the honourable way to revise it would be to reduce how much needed to be done, by how much time of the schedule was reduced. This would be how the natural leadership in a functional hierarchy would deal with the schedule. Produced from a feasibility study of a small version or cross-section of a game based on the **Event-Database Architecture**.

The sense of honour or integrity of the natural leadership would not suffer it to be unrealistic in its expectations. To expect the same amount of work to be done in less time would be inviting failure. Putting the leadership in a position where they could break any promises they made to deliver a project on time.

So it would either cut the schedule. And thus cut how much needed to be done. And hence cut the number of features there would be in the final game. Or the natural leadership conducts a second feasibility study and extracts a second hopefully shorter schedule from that. Or the leader would stick to the results of the first feasibility study. And follow the long-term plans in the schedule to complete the product from that study. Having made everyone aware that the prognosis suggests the game will be delivered past the deadline. Being sustained by the strength of character to see this long-term plan through and face the consequences.

But in a dysfunctional hierarchy, an unnatural leadership has no sense of honour or integrity. For keeping your promises requires long-term plans. And for one reason or another, the leadership does not believe in long-term plans. In the case of the Computer Games industry, as has already been explained, the unnatural leadership originates from the view that software production is an art. And since it is an art there can be no such thing as a complete production process or a long-term plan for such a process. Notwithstanding that the weakness of character of the leadership cannot sustain the execution of any long-term plans. Therefore, the leadership has a cynical outlook on any long-term plans. This cynicism affects the promises that the leadership, or anyone else working under that leadership, makes over a period of more than a few days.

Thus, in the case of revising a schedule drawn up from a feasibility study at the beginning of the **Event-Database Production Process**, the leader will neither reduce the amount of work that needs to be done by the amount schedule is reduced. Nor will the leader conduct a second feasibility study and then revise the schedule based on the results of the second study. The leader will simply cut the schedule. And maintain that the same amount of work must be done, regardless of how much the schedule has been reduced.

Any schedule or long-term plan that comes from conducting a feasibility study, of building a game based on the **Event-Database Architecture** on a target *platform*, will require a motivation. This motivation will encourage and inspire the staff to complete that plan. But the form that this motivation takes also distinguishes a functional hierarchy from a dysfunctional hierarchy.

In a functional hierarchy, a natural leadership has the ability to motivate through vision. This comes from the eloquent way that the leader can make logical arguments for what the final game will look like through natural language. As has already been explained, this eloquence comes from the habit the leader has of engaging in dialectic forms of communication with the staff. That is to say making logical arguments from different points of view about a subject to reach the truth. And that in turn comes from a dependency on *Forward engineering* in software production which requires a dialectic dialogue, between a software user and a software producer, at the start. And that in turn comes from the view of software production as a science.

In a dysfunctional hierarchy, an unnatural leadership cannot conjure up a vision to motivate the staff. As already been explained, the leadership arises from the view

of software production as an art and relies on *Reverse engineering* in software production. That in turn makes it promote lower-level tools that *Reverse engineering* depends on. And with that there is a recession away from higher-level tools, including natural language which is the highest-level tool. Therefore, the leader cannot conjure up a vision using natural language.

So instead the leader depends on a permanent state of emergency to motivate the staff to execute any long-term plans. This emergency will be evoked unsurprisingly when any deadlines for the long schedule or long-term plans loom. This emergency will surprisingly also be evoked in between these deadlines too. When there is no immediate danger, but the leadership perceives a threat on the horizon.

For example, after a feasibility study has been conducted in the first step of an **Event-Database Production Process**, and a schedule has been produced based on that study to build the whole game, it may become apparent that the schedule goes past the final deadline for the game to be submitted for approval by a Game Publisher or game console manufacturer. Or it goes past the deadline for the game to be released. Or it goes past the deadline for the game to be demonstrated at an upcoming public event, like a Video Games conference. In a dysfunctional hierarchy, even though this deadline may be between 18 months and 2 years in the future, the unnatural leadership will evoke an emergency to revise the schedule.

Such revisions will typically be carried out in an ambush. That is to say, the leader will invite the followers to a meeting under false pretences, with little or no notice: 'just a quick chat'. Ostensibly, the purpose of the meeting will be to assist the followers, which will lull them into a false sense of security. But, before they meet, the leader will have already decided what help will be offered. In this case the help being offered is to cut the schedule which goes past the deadline for the game either to be submitted for final approval, to be released or to be publicly demonstrated. Without reducing the amount of work that needs to be done or features in the game. A state of emergency will be declared to justify this. Due to the schedule putting the entire future of the game, if not the entire company, at risk.

Furthermore, the leader will have decided that the followers will have no option but to accept this offer. And under no circumstances must the followers be allowed to leave the meeting without accepting it.

This ambushing technique will not only be employed at the beginning of the **Event-Database Production Process**. But throughout the process from beginning to end.

Whenever the staff come up with a schedule to perform some task. To create some new concept artwork. To add some 3D models for a new character or a new landscape. To add some new animation for that character. To add some **Game Object** for a weapon or armour. To add some new **Events** that control how this new weapon or armour behaves. To add some music for the new landscape. To add some sound effects for the new weapon or armour. To add some code to send the new **Events**. To add the code for the **Game Objects** to respond to the new **Events** and perform the **Actions**. And the leadership fears that the schedule is too long. And the leader suspects a decision to cut that schedule will face opposition, the leader will use an ambush to bypass that opposition. The anticipation of confrontation will strike fear

into the leader. So the leader will drag along one or more allies, who are also part of the unnatural leadership, into the ambush. To quash any resistance.

Except for this resistance which the leader expects to the decision to cut the schedule, however, the presence of these allies will be virtually redundant. And when the leader's decision appears to meet no resistance, the leader will become complacent.

So much so that on some occasions, the leader may even leave before the end of the meeting. And the leader's allies will be left behind to finish enforcing the decision. If that decision were to cut a plan or schedule, of the staff, then the allies would finish off butchering it. Or if that decision was to change the tools or methods used to complete the task on schedule, then the allies would be left to provide the details for the new tools or methods.

Again the reason given to justify this decision will be a state of emergency that will be declared by the leadership. To either ensure the project remains on schedule. Or for the sake of keeping good relations with a Game Publisher, a game console manufacturer, a financial backer or a client. Or for the sake of some marketing strategy to release the game in the summer or at Christmas, to increase sales. Or for the sake of avoiding the release date of some popular competing product whose marketing may drown out the marketing for the game. Or to conform to some spurious legal requirement that the leadership believes the old tools or methods that were cut violate. Even though the leadership believes that software production is an art. And does not believe in producing detailed plans that can tell it the knock-on effect of any task in the production process going over schedule. Nor can it produce any documents to show that the financial backers or client were happy. Nor does it have the clairvoyance to tell what will happen to sales in the summer or in the winter at Christmas. Nor does it have the clairvoyance to know the marketing strategy of competitors will be in the future. Nor can it produce documents to show the legal requirements that justify the old tools or methods being dropped, and new tools or methods being adopted to complete a task whose schedule had been butchered.

In a functional hierarchy, the *good memor*y or good records of the natural leadership would keep track of such decisions. And when that decision failed, the leadership would learn from its mistakes and not make that decision again. If that decision leads to some 3D model of a character or landscape being produced which had errors in it and did not look like the drawings of the Concept Artists. Or it had too many *polygons* in it. Or some animation of a character is produced that was not quite right and had too many *Frames*, or the *Frames* did not smoothly blend into each other. Or the *Frames* did not cycle from a neutral starting position back to that starting position and allowed it to blend with other animations from the same starting position. Or some **Game Object** for a weapon or armour being produced, which did not function correctly. That did not produce or respond to the **Events** that it was supposed to. Or that did not perform the **Actions** that it was supposed to in response to those **Events**. Or some sound or music that did not play when it should or did not sound right, or was too loud and distorted. Or some feature which was added to a *game design* which was initially small and simple but grew into something large and complicated and had to be dropped to remain on schedule. The *good memory* or records of the natural leadership would note the decision that produced this error in some form of documentation. And the leadership would not make the same mistake again.

However, in a dysfunctional hierarchy, the unnatural leadership has a short-term memory. It does not keep good records of its decisions and the consequences. When the staff inevitably produce errors in their work because of the dramatic way their schedules to perform tasks were either cut by the leadership. Or their plans to perform the tasks with one tool or method were dramatically revised by the leadership because of a state of emergency. The staff will be blamed for those errors that result. If the staff try to defend themselves by reminding the leadership of the cuts to their schedule, or the changes made to their tools or methods, then the leader will push back. The leader will claim that the staff should have informed the leader when it became apparent that the task was going to fail. And indeed the leader would have said words to that effect. In the meeting in which the schedule or plan was revised, because of a state of emergency, the leader will instinctively say at the end:

> 'If there are any problems, let me know!'

On other occasions the leader will downplay the importance of the task:

> 'Don't worry! Its not that high a priority!'

But these words would have been just that: words that the leader would say to reassure the staff to continue with the task. Despite their doubts after the dramatic revision made by the leadership. Impromptu words that the leadership has no intent to honour.

Yet when the staff subsequently do run into some trouble and turn to the leadership for help, the leadership will either claim to be too busy. Or the leadership will delegate help to someone else. This delegate will of course be under no obligation to help and will either claim to be too busy as well or delegate to someone else. Or the leadership will isolate the staff. Insisting that they solve the problem themselves and take responsibility. They should not trouble the leader until all other possibilities have been exhausted. And somehow be able to read the leader's mind about what all these other possibilities are. While bearing in mind that if the staff end up wasting lots of time trying other possibilities which ultimately fail, the leader will hold them responsible.

On the other occasions where the leadership downplayed the importance of the task, the staff will suddenly find that it has been raised overnight to an emergency status. So the staff will have to sacrifice themselves, sacrifice their lives, including doing overtime, to complete it.

As has already been explained, the unnatural leadership has no sense of honour or integrity. And the leadership has no intention or capacity to keep its word, to keep any promises it made earlier to reassure the staff. Not least because of a weakness of character and a short-term memory.

In a functional hierarchy, the strength of character and *good memory* or good records of natural leadership is what makes them love their job. These records would include the *game design*, *technical design*, *data design* and *tools design* of

the **Event-Database Production Process**. This *good memory* or good record stops them from hating their job because of the frustration of repeated mistakes. No one will love something which after much effort causes them pain or distress and they make little progress. It is this love that will allow them provide any help the staff may require during the **Event-Database Production Process**. Or to keep any promises they make to reassure staff to begin any task.

In a dysfunctional hierarchy, this love does not come naturally. Instead the unnatural leadership naturally ends up hating the job because of the frustration of repeated mistakes. Due to a short-term memory and the habit of neglecting to keep good records. These records will include important documentation required by the **Event-Database Production Process**, especially the *data design*. This neglect, in turn, comes from the leadership's view of software production as an art. Any instance of a production process is merely an example of the art. Therefore, there can be no such thing as a complete production process or a complete design (including any records such as the designs of the **Event-Database Production Process**). And a belief in such things is merely a futile attempt to limit the expression of that art.

Despite the consequences of this outlook there are many *academic courses*[3] which teach the practices of unnatural leaderships. That nominally claim to prepare its students for leadership but in fact produce unnatural leaderships or vicarious leaderships. That reduce leadership down to delegation of responsibility and enforcement. And neglect the importance of love. That is to say, loving the job and the relationship that has with having a *good memory* or keeping good records. To stop them hating the job because of the frustration of repeated mistakes. And often in the Computer Games industry amongst the unnatural leaderships, you will find graduates from these *academic courses.*

4.1 THE MARRIAGE OF THE BEAST

In the previous chapters, it was explained that although in a functional hierarchy, it would be easy to judge the success or failure of a production process, including the **Event-Database Production Process**. But in a dysfunctional hierarchy, it would be hard to judge because of the way the hierarchy obfuscates the truth. One of the reasons for this is the short-term memory or habit of not keeping good records of the unnatural leadership that directs the process in a dysfunctional hierarchy. Ironically, in a functional hierarchy led by natural leadership, the leadership has a *good memory* or good records from which you could judge the success or failure of the process. And as a result more problems surface in a process led by a natural leadership than one led by an unnatural leadership. So, on the surface, an **Event-Database Production Process** led by a natural leadership looks far worse at first sight, than a process led by an unnatural leadership. How you may wonder, if an unnatural leadership does not enjoy the cult of personality or popularity of a natural leadership, can it pull this off? The answer is a marriage of convenience which you need to understand to uncover the truth about what lies beneath the surface in a dysfunctional hierarchy.

From time to time, the state of emergency, declared by an unnatural leadership in a dysfunctional hierarchy to motivate the staff, will subside. And the problems and *Bugs* in the software will be solved, or at least appear that way. The staff will

have dealt with the crisis by doing overtime either during the week or at weekends to complete the work necessary before the deadline for the current milestone. This will give the unnatural leadership and the staff some breathing room before the next milestone or deadline in a production process. This would include the **Event-Database Production Process**.

Therefore, it follows that one of the best times to judge a process led by an unnatural leadership is just before a deadline. When the problems or *Bugs* in the software rise to the surface. It is academic to judge a process, after a deadline, like the end of a project. At the end of a project, when a *Post-mortem meeting*[4] is typically carried out in the Computer Games industry, the damage has already been done. The staff will have already done their utmost to cover up the problems. There will be little or no records. The records (i.e. the designs produced by the production process) will be well out of date.

In a dysfunctional hierarchy, the number of problems or *Bugs* in a few days before the deadline will rise. It will be clearly visible regardless of whether there were any records or not. And the *Game Testers* may catch some of them and produce records in the form of ad hoc tests and reports they perform before the deadline. These numbers will spike up dramatically getting towards the deadline and fall off precipitously just before the deadline.

By contrast in a functional hierarchy, the number of problems or *Bugs* will be steady throughout each phase between each deadline. And it will be obvious from the *good memory* or the good records that the natural leadership keeps of these problems. These records will include the designs, *game design*, *technical design*, *data design* and *tools design* of the **Event-Database Production Process**. Especially the *data design* which is kept up to date by the *Database Administrator* which will have a record of all the features of the game. Since this can be used to test the **Events** in the **Game Database**, used to implement those features, against their description in the *data design*. To produce a far more exhaustive and accurate test than the ad hoc test performed typically by the *Game Testers* in a *Software Evolution Process* in the Computer Games industry.

Nevertheless, before the end of the project, during each phase between the deadlines, in a dysfunctional hierarchy, the experience in each phase will teach the staff an invaluable lesson. It will become clear that the leadership cannot be trusted. Whenever a problem has been presented to the leader, it becomes aggravated by the leader's self-defensive reactions. And their professional autonomy, that is to say, the vicarious leadership that the unnatural leadership practices through them, has been lost. So it would be better to anticipate what the leader wants to see or hear, and present it, rather than reveal any problem.

Thus, a marriage of convenience will begin between the leader and each of these staff. The leader will be happy because the work given to the staff will seem to be dealt with effortlessly. The followers or staff will be happy because, so long as no problems are revealed, the leader will not interfere. And the leadership will continue to act through them, vicariously, to conduct the leadership. Thus giving the staff the appearance of autonomy or authority.

In a dysfunctional hierarchy, even though they hate each other, the unnatural leadership and the staff will both connive to suppress the truth, about the problems or

Bugs in the software. On the one hand, the staff will suppress the truth about the problems and *Bugs* in the software. And on the other hand, the leadership will suppress the truth about the authority and autonomy that the staff have. Both will believe that if they hear no evil, see no evil and speak no evil, then they will be safe from harm. But that will be the insidious part of the corruption. By suspending their sense of judgement, they will be giving up one-half of what makes them intelligent: their sense and sensibilities. By suspending their sense of judgement they will be suspending their sensibilities. And the self-defensive decisions they make in reaction to their senses will now be unchecked and deluded by its unstoppable momentum.

It is this marriage of convenience that suppresses the truth about the problems or *Bugs* in the software, and the software production process, including the **Event-Database Production Process**.

By contrast, in a functional hierarchy, the natural leadership leads by example, not vicariously, through the example of others i.e. the staff. The staff have nothing to gain from suppressing the problems or *Bugs* in the software. The sense of honour of the leadership, along with *good memory* or good records and the strength of character, makes the leadership naturally expose any problems or *Bugs* in the software themselves. In the course of testing that the decisions of the leadership are in keeping with its words, the problems or *Bugs* in the software are naturally exposed.

This leads to another distinguishing characteristic between unnatural leadership and natural leadership. That is the primary source of problems or *Bugs* reported in a dysfunctional hierarchy, under an unnatural leadership, is the clients or the marketplace, outside of the hierarchy. Where the forces which drive the marriage of convenience has no effect. While the primary source of problems or *Bugs* reported in a functional hierarchy is the natural leadership itself. Practically this means that if you want to mitigate the effects of a dysfunctional hierarchy in the Computer Games industry, you should use *Third Party Game Testers*[5] who are not within the hierarchy. That is to say you should use *Game Testers* who are in another company. There are many companies available that provide this kind of service. For a functional hierarchy, this will not make any difference. For the number of problems or *Bugs* reported by the natural leadership will be greater than any report you could get outside of the hierarchy.

The marriage of convenience leads to another distinguishing characteristic of an unnatural leadership and a natural leadership. In a dysfunctional hierarchy, both the unnatural leadership and the staff will become paranoid. They will believe that any strangers to the hierarchy who expose the problems in the marriage, either intentionally or inadvertently, to be an existential threat. For example, hiring a *Third Party Game Testers*, outside of the hierarchy, to test the game would be perceived as a threat or some great evil. The leadership and some of the staff will unite in their opposition to such strangers. However, if there were any evil in the marriage, it would be the *narcissism*[6] which prevents them from seeing the truth. The truth is that there are far more problems, *Bugs* or errors in the software they create than they are reporting. And the number of errors being reported once the *Third Party Game Testers* has been introduced should go up.

In a functional hierarchy, there would be no opposition to using a third party to conduct the function of the *Game Testers*. And the number of errors being reported

once the third party has been introduced should go down, compared with the number being reported by the hierarchy before the third party was introduced. That is to say the number of errors being reported by the natural leadership of the hierarchy, by testing the **Events** in the **Game Database**, against the description of them in the *data design*, to perform an exhaustive test of a game built with the **Event-Database Architecture**.

This honesty is key to a natural leadership's cult of personality or popularity and makes it credible to the staff. A leader does not have to be perfect to be credible: only open and honest. If the leader is willing to admit mistakes and is open about it, then the rest of the staff will be willing to do likewise.

By contrast, in a dysfunctional hierarchy, in the marriage of convenience between an unnatural leadership and the staff, the leadership is not credible. It is the opposite. The staff find the leadership incredulous, that is to say unable or unwilling to believe in it. The leadership is neither open nor honest about its mistakes. And neither are most of the staff willing to be open and honest about their mistakes. Apart from new members or strangers to the hierarchy.

This credibility on the one hand, and incredulity on the other hand, also applies to the channels of communication in a functional and dysfunctional hierarchy.

In a functional hierarchy, your work would be controlled through one communication channel; between you and the natural leadership. Both you and the leadership can see when the number of tasks is getting too high. And you can come to an agreement on the priority of the tasks. This is a credible form of communication and keeping track of your tasks.

In a dysfunctional hierarchy, there will be multiple communication channels all vying for your attention. You will not only receive requests directly from the unnatural leadership. But requests will also come indirectly from the army of delegates that the leadership dispatches to perform the vicarious leadership. That in turn comes from the marriage of convenience between the unnatural leadership and its followers.

Some of the delegates dispatched by the leadership will be literally merely repeating what the leadership has already told you. None of whom will care what tasks other delegates have given you. Leaving you to incredulously manage yourself, keep track of the number of tasks, decide when it is too high and set the priorities for the task, all by yourself.

These delegates will, in theory, have the same authority and autonomy as the leader. And the leader will reward those delegates. In practice, they will all be mere vassals of the leader's fiefdom. Their authority and autonomy over any assignments they give you, and their knowledge of their master's will, will be as ethereal as their master's patience.

For example, suppose you were a *Sound Designer* or *Sound Engineer*, and you received a request from one set of *Game Programmers* to create the sound effects for the one of the Front-End menus shown when the game starts. That the *Programmers* will use to make that menu. Another set of *Programmers* will come along a few hours later after and ask for the sound effects of a second menu which they were working on. And a few hours after that, another set of *Programmers* come along to ask for the background sound effects and atmospheric music which suits the mood of a wooded forest on the outskirts of the village, say in the game *LPmud*.

Each *Programmer* will be completely oblivious to the request of the other *Programmers*. Each will expect their request to be met immediately declaring that it is urgent. Regardless of what other tasks you have. You will have to make assumptions for most of these sound effects and music. Since, typically in a *Software Evolution Process* in the Computer Games industry, there will be no *game design* which describes what each effect or music should sound like. You will have to make assumptions about whether all of the sound effects for the buttons on the first or second menu should be the same. What common theme if any should run through these sounds? You will have to make assumptions about what kind of sounds should be heard in the forest. Should it include sounds of people in the nearby village? What the theme of the atmospheric music should be? Should it just consist of modern, classical or medieval instruments? What should the priority of all these tasks be?

So you make your assumptions because they all claim their request is urgent. You make the **Game Objects** for all the buttons in the first and second menus produce the same **Secondary Event** which plays back the same sound effect. And you add a **Secondary Event** which plays back the sound effects of wind blowing through the trees and bushes in the woods, birds singing in the trees and sound of a river rushing by and the sound of villagers nearby. And you add a **Secondary Event** which plays back music from Evard Grieg's Morning Mood when the player enters the wooded forest.

Only to find out afterwards, that the *Programmers* who asked for sound effects for the first menu wanted a completely different sound from the ones who asked for sound effects for the second menu. And the *Programmers* who wanted the atmospheric music for the wooded forest in fact wanted some music made with medieval instruments. And you have to revise all of the *Database Records* for the **Game Objects**, **Secondary Events**, sound and music you have added. And that the priority for all the tasks has become even more urgent because of the delay caused by the revision. Not because all of these changes are necessary. But because they all want to impress the unnatural leadership and keep the autonomy which the marriage of convenience affords them.

It does not take a genius to imagine what personalities this kind of scenario will breed. It will not be the intelligent that succeeds in this atmosphere. If any of the followers had sense and sensibility, they would instinctively recognise the situation. They would recognise that there could not be any progress without courtesy. They would indulge others without giving their approval and despite the facts. And just as importantly, they would intervene when they see others being coerced to do likewise.

However, those who had no intelligence would show no integrity, compassion or restraint. They would be loud, crass and obnoxious to get their way. The atmosphere would be charged with testosterone and macho bravado. Many personal conflicts would thrive as the ambitious followers vied to fill the void left by the vicarious leadership.

So, given this scenario, the chances of the success of any intelligent person, or the *prevalence of women*,[7] will be slim.

This leads to another distinguishing characteristic between a functional and a dysfunctional hierarchy. If you put women under a natural leadership of a functional hierarchy, then their productivity will be indistinguishable from the rest of the staff.

If you put women under an unnatural leadership of a dysfunctional hierarchy, then their productivity will be lower than the rest of the staff.

To measure their level of productivity, you could either use the *Event-Database Architecture Productivity Formula*[8] described in later subchapters. Or you could measure their level of knowledge of the project using *Event-Database Architecture Knowledge Test* described in earlier subchapters. And assume that their level of productivity would be proportional to their level of knowledge.

In a functional hierarchy, the overall level of productivity, in one phase or between two deadlines of the **Event-Database Production Process**, should fall once the women were taken out of the project. And it should rise back up again once they were added back, in the next phase or before the next deadline. Likewise, their level of knowledge should not be significant different from the rest of the staff.

In a dysfunctional hierarchy, the overall level of productivity, in one phase or between two successive deadlines of the **Event-Database Production Process**, when the women were taken out of the project, should be the same. As the level of productivity in the next phase or before the next deadline, when the women were added back to the project. And their level of knowledge should be significantly lower than the rest of the staff.

4.2 THE TIME OF THE BEAST

In the previous chapter, it was explained how the unnatural leadership in a dysfunctional hierarchy is closed. That is to say, the marriage of convenience in a dysfunctional hierarchy causes problems or *Bugs* in a software production process, including the **Event-Database Production Process**, to be suppressed. The unnatural leadership offers promises of autonomy and authority to the staff who show promise when performing tasks, in order to conduct the leadership vicariously. And these staff, in turn, suppresses problems or *Bugs* to keep that autonomy or authority. Whereas in a functional hierarchy, the natural leadership is open and honest. That in turn comes from its sense of honour or integrity to keep its word or its promises. That in turn causes it to check the outcome of its decisions meets its expectations. That in turn causes problems or *Bugs* to naturally rise to the surface. And that the main source of errors reported in the process is the natural leadership.

It would be a mistake to presume that the pressures of time account for a dysfunctional hierarchy being closed and a functional hierarchy being open. Given tight time constraints to finish a project, it may seem reasonable for a few corners to be cut redundant. Given more time would not what appears to be an unnatural leadership redundant. Changing the sentence into a question. exhibit the same signs of a natural leadership and be credible i.e. open and honest?

But it is not a question of time. It is simply a matter of knowledge. The leadership cannot manage a project without knowledge of the risk of the decisions it makes and the consequences. The staff underneath that leadership also cannot perform their tasks well without knowledge of what it entails.

As has already been explained, in a dysfunctional hierarchy the unnatural leadership acquires its position because it views software production as an art. And it has a dependency on *Reverse engineering.* And it promotes the lower-level tools that

Reverse engineering depends on. This makes the small band who excel at *Reverse engineering* have a superior level of knowledge compared to the rest of the staff. And that is what promotes that band into the leadership.

By contrast, in a functional hierarchy the natural leadership acquires its position because it views software production as a science. And it has a dependence on *Forward engineering*. And it promotes higher-level tools that *Forward engineering* depends on. The highest-level tool is natural language. And with the promotion of that, all the staff acquire the same level of knowledge.

If it were possible, in a dysfunctional hierarchy, for all the staff to acquire the level of knowledge necessary in order to fulfil their role, despite the corners they cut, then they may have a hope of achieving their objectives within a short time frame. But because only the unnatural leadership has the superior level of knowledge required, and the rest of the staff lack that knowledge, there will be no hope whatever the time frame.

And the converse is true in a functional hierarchy. Since the natural leadership promotes higher-level tools, especially natural language that in turn promotes the staff acquiring the same level of knowledge, there is hope. There is hope that they can achieve the goals of the project, no matter what the time frame. So long as the resources required (including the staff) were available to achieve this.

Occasionally, from time to time in the Computer Games industry, a *Post-mortem* meeting is called at the end of a project. For the staff to get together and discuss any lessons they have learnt from the software production process just ended. And in discussions, the subject of time will be brought up. Some will say

> 'There was not enough time to do this!'

Others will say

> 'There was not enough time to do that!'

But these are euphemisms for a lack of knowledge. And what they really mean is

> 'No one knew how to do this!'
> 'No one know how to do that'.

This leads to another way to distinguish between a functional hierarchy and a dysfunctional hierarchy. Hold a *Post-mortem* meeting at the end of an **Event-Database Production Process** to discuss the lessons to be learnt from the project. Whenever the word 'time' is used to explain the reasons for problems or *Bugs* in a task, ask the person who said it to give an estimate of how much time they think would be

required to complete a similar but new task? Ask the leadership how much time they think would be required to complete the same task?

In a functional hierarchy, the estimate of the staff and the natural leadership will be the same. In a dysfunctional hierarchy, the staff will either not give an estimate because they are still ignorant of what the task involves. Or the estimate will be very different from the estimate given by unnatural leadership. Furthermore, if you ask the leader to make the longer estimate the default estimate for future projects, then the leadership will object. Since, as already explained, they view software production as an art. And they would object to any limitations, including time, being imposed on that art.

For example, suppose in a *Post-mortem* meeting you hear someone say

> 'There was not enough time to add the **Game Objects**, **Events**, **Actions**, and *Database Records* to create the Front-end menu.'

In a dysfunctional hierarchy, only the unnatural leadership will have the superior level of knowledge to add the Front-end menu easily. The staff who had to add the menu to the game did not know how to do it. They were not sure how easy it was to add the menu. They started to add it and found out later on that there was something missing in their knowledge. And at the *Post-mortem* meeting, they still will not be able to give you an estimate of how long it would take to add a new menu. Or this would be very different from the estimate of the leadership.

In a functional hierarchy all of the staff would have the same level of knowledge. And the feasibility of adding a menu would be obvious to the natural leadership and the staff from the start. They would have engaged in a dialectic dialogue. That is to say that they would have made logical arguments against or for adding that menu. And they simply would not have attempted to add the menu when it became obvious from this dialogue that it was not feasible. And the subject would not have been brought up in a *Post-mortem* meeting. Furthermore, the good memory or good records of the natural leadership will include records of the meeting where the feasibility of adding the new menu was discussed.

Any *Post-mortem* meeting requires evidence through which the staff can sift through to investigate what happened during the production process. In a dysfunctional hierarchy, the unnatural leadership views software production as an art. Therefore, its decisions are ad hoc, made to address immediate problems or needs at hand without consideration for the wider applications or implications, including time. Thus, there are little or no records kept during the production process. There is little evidence to sift through during a *Post-mortem meeting*. In functional hierarchy, the natural leadership views software production as a science. Therefore, its decisions are planned and premeditated, to take into account the wider implications, including time. Thus, there are lots of records kept during the production process. And there is a lot of evidence to sift through during a *Post-mortem* meeting.

These two ways in which *Post-mortem* meetings will be conducted will provide you with a microcosm, a miniature encapsulation, of how all the meetings have been

conducted during the **Event-Database Production Process**, in both a functional and dysfunctional hierarchy.

In previous meetings about tasks that were yet to be completed, in a dysfunctional hierarchy, the meeting will have begun with little or no materials to assess the feasibility of that task, given the time to do it. Instead, the unnatural leadership will have made wild overly optimistic speculations about the feasibility. And due to the marriage of convenience, between the leadership and some of the other staff, those staff will also have joined in with these optimistic speculations. There will have been no design, *game design*, *technical design*, *data design* or *tools design* from the **Event-Database Production Process**, before them when the assessment was made. There will have been no record of any similar task that had been done in the past, estimates of how long this took and notes of any problems that were encountered.

In many parts of the Computer Games industry, an Agile Development methodology will be employed when organising these meetings. In this methodology, the nominal teams of *Game Programmers*, *Game Designers*, *Game Artists*, *Sound Designers or Engineers* and *Game Testers* have a 'Sprint Planning' meeting. Where they decide what tasks they will be doing in the next phase of the production process or 'Sprint'. This lasts typically two weeks. In that meeting, the staff will be given tasks to be completed for the next 'Sprint', by members of the leadership. This may be a Lead Programmers, Lead Artists, Lead Game Designers, Lead Game Testers or *Game Producers*. The staff will literally make up numbers on the spot, about how much time will take to perform the tasks. Even though half of the tasks may be things they have never done before. This may be adding a new menu, a new character to the *Game World*, a new animation of an existing character, a new weapon with some unique form of attack, a new armour with some unique form of defence, a new sound effect or music to accompany the action in the *Game World*. These estimates will be made regardless of time. Bear in mind, that this will be part of a larger project. Where any number of changes could have been made to the *Game World*, by other staff, which they were unaware of. And if you wanted further evidence of a dysfunctional hierarchy then get the records (if any) of the time estimates for tasks made by the staff in the 'Sprint Planning' meetings. And compare these times with the real time it took to complete those tasks in that 'Sprint' or the next 'Sprint'. Or compare these times with the estimates given by the staff in the *Post-mortem* meetings. You will find a huge discrepancy between the estimates and the real times, or between the estimates given during the 'Sprint Planning' meetings and those given in the *Post-mortem* meetings.

Even though the pressures of time may seem to affect their behaviour, half the tasks and methods they use will be those they set themselves. These tasks will be set regardless of the effects on time to complete the project.

Whereas, in a functional hierarchy, in previous meetings about tasks that were yet to be completed, the meetings will have begun with lots of materials to assess the feasibility of the task, given the time to do it. The natural leadership will have presented either the *game design*, *technical design*, *data design* or *tools design* of the **Event-Database Production Process** in the meeting. Most important of these will have been the *data design* which defines the natural language of the project, the **Events**, **Game Objects, Actions**, *Database Tables*, *Records* or *Fields*. The *good*

memory or good records of the leadership will include a record of any similar task that had been done before the meeting, with estimates of how long this took, and notes of any problems that were encountered. And the time it will take to do the new task will be set by the natural leadership based on a dialectic dialogue with the staff and the materials before them.

This leads to another distinguishing characteristic between a functional and dysfunctional hierarchy. In a functional hierarchy, the pressures of time come from the natural leadership. The staff do not place this pressure on themselves. In a dysfunctional hierarchy, the pressures of time come from the staff. And the staff place this pressure on themselves. Therefore, in a *Post-mortem* meeting at the end of a project, if you hear pluralistic nouns or pronouns like

> 'I think the leadership did not give enough time to do this!'
> 'I think we did not give enough time to do that!'

then that would be a sign of functional hierarchy.

If you hear self-incriminating phrases or pronouns like

> 'I did not give myself enough time to do this!'
> 'I did not give myself enough time to do that!'

then that would be the sign of a dysfunctional hierarchy.

Popularity and self-incrimination are another distinguishing characteristic between a functional and dysfunctional hierarchy. As has already been explained, in a functional hierarchy, the natural leadership leads by personal example. Whereas in a dysfunctional hierarchy, the unnatural leadership leads vicariously through the example of others. In the former, this leads to the cult of personality or popularity of the natural leadership. In the latter, this leads to self-incrimination of the staff who the vicarious leadership nominally delegates authority and autonomy to. With this nominal authority and autonomy, the staff volunteer estimates of how long it would take to complete tasks. Or how tasks will be done? Or whether tasks can be done in the time allocated? Even though the vicarious leadership will undermine every one of these decisions at a whim, on an ad hoc basis, after these have been made. As a result, the staff are the ones who face the risks of the decisions made. They are the ones who end up, unfairly, taking responsibility for the failures of these decisions and feeling guilty. They are the ones who end up either incriminating themselves when something goes wrong or incriminating time.

Popularity and self-incrimination will also characterise how natural leadership or unnatural leadership will deal with a client or the financial backers of a project. And their interaction with the client will also show their attitude towards time.

For example, in the Computer Games industry, the financial backers of a game will typically leave the majority of its *game design* to the leadership's discretion.

Likewise, the production process, such as the **Event-Database Production Process**, would be largely left to the leadership's discretion, including any tools or methods the staff used. The investors would examine the work at regular intervals or milestones. But how the leadership moved from one interval to the next would not be dictated by the investors. So long as the work achieved between the intervals or milestones met their expectations, the process would continue. The deal with them will involve payment, at these regular intervals, based on how much work the hierarchy had done.

In a functional hierarchy, the natural leadership will achieve popularity with the client, by keeping its word. And that in turn means that it values the Feasibility Study at the beginning of the **Event-Database Production Process** and being able to assess whether the tasks between each milestone would be possible, given the time left to do it. The natural leadership's sense of honour and courage makes it open and honest about the feasibility. The leadership is not afraid of losing the contract or missing a payment because it cannot complete the tasks required due to an unrealistic schedule.

For example, for the game *LPmud*, there would be nine main features:

Settlements
Buildings
Mountainous Landscapes
Other Landscapes
Creatures
Non-Player Characters
Player Characters
Combat System
Treasures
Puzzles/Quests

In the first step of the **Event-Database Production Process**, the leadership would set a task to implement a fraction of each of these features in the Feasibility Study. So that the time taken to implement this fraction could be extrapolated to estimate the time it would take to implement the whole feature. So the goals of the Feasibility Study, beyond the minimum features required of a game based on the **Event-Database Architecture**, would be the implementation of

10% of the Settlements
10% of the Buildings
10% of the Mountainous Landscapes
10% of the Other Landscapes
10% of the Creatures
10% of the Non-Player Characters
10% of the Player Characters
100% of the Combat System
10% of the Treasures
10% of the Puzzles/Quests

The time taken by all the staff to implement this cross-section of these main features would then be extrapolated upwards to work out the time it would take to implement the main features completely, as well as the total time. And these estimates would be documented in the *game design*, along with the deadline for the project. If the estimated total time exceeded the deadline for the project, then the document would clearly state this.

In a dysfunctional hierarchy, the unnatural leadership ends up incriminating themselves. They will volunteer assurances and promises to the client, which their weakness of character cannot keep. The leadership will not value the Feasibility Study at the beginning or the middle of the **Event-Database Production Process** and being able to assess whether the tasks between milestones are possible, given the time left to do it. The leadership will not have the courage to face negative results.

The leadership will realise that the more new features are shown to the financial backers, the more willing they will be to make the next payment. But the leadership will also know that the more new features added, the greater the workload will be on the staff.

When dealing with the financial backers, the leadership will agree to a list of features with the motive of getting as much payment as possible. A minority of that list will come from the investors. But the majority of it will be the impromptu ideas of the leadership will agree to do to secure funding.

In the Feasibility Study, the assessment of the risk of these ideas would not be objective: but subjective.

For example, for the game *LPmud*, in the Feasibility Study at the beginning of the **Event-Database Production Process**, you may find written in the *game design* an assessment of the risks of certain features that were to be added to the game. The assessments would simply be comprised of a list of items which leadership believes could go wrong.

These items would be drawn up pessimistically, in an impromptu fashion. It will include tables or graphs with numbers which suggest that the assessment was mathematical or objective. However, if you looked closely at these figures, you would find that they were always nice round numbers, 50%, 60%, 40%, 50/50, 30/70 and so on (You can see an example of these figures in Table 4.1).

All of the features will have deadlines set before any detailed plan has been drawn up, for how these would be implemented.

However, after they have returned from the meeting with the investors, the leadership will present the list, to the rest of the staff in a different light. They will present it as if it were forced on them, by the financial backers. And by presenting this list the unnatural leadership will be incriminating themselves to the staff. And when the staff fail to deliver some of the items on that list by the next milestone, the items on the list will end up incriminating the leadership to the client.

This leads to another distinguishing characteristic between a natural leadership and an unnatural leadership. If you conduct a *Post-mortem* at the end of the project, ask for the documents that were used to predict how long the **Event-Database Production Process** would take. In a functional hierarchy, these documents will show an objective prognosis based on empirical measurements of time gathered by all the staff. These measurements will be in either the *game design* or *data design*,

TABLE 4.1
Example of Table of Risks in a Software Evolution Process in the Computer Games Industry

Feature	Risks	Probability (%)	Delivery Date
Settlements	High polygon count	10	Week 1–12
Buildings	High polygon count	10	Week 1–12
Mountainous landscape	Procedural generation errors	50	Week 1–12
Other landscapes	Procedural generation errors	50	Week 1–12
Creatures	Animations	10	Week 1–8
	AI	60	Week 13–20
Non-player characters	Animations	10	Week 1–8
	AI	60	Week 13–20
Player character	Animations	10	Week 1–8
Combat system	Small size (2 characters)	10	Week 1–12
	Medium size (16 characters)	20	Week 13–15
	Large size (>16)	30	Week 16–18
Treasures	New animations for each weapon	70	Week 13–20
	New animations for each armour	70	Week 13–20
Puzzles/Quests	New quest characters	70	Week 13–20

which are the documents that were meant to contain this information according to the steps of the **Event-Database Production Process**. In a dysfunctional hierarchy, these documents will either not exist. Or these will show a subjective prognosis based on the non-empirical probabilities set by the unnatural leadership. That in turn will be based on the leadership's superior level of knowledge, over the rest of the staff.

If these documents did exist and the prognosis was sincere, then the staff will have received the same prognosis as the client. Therefore, if you conduct a *Post-mortem*, ask the staff what prognosis they were given for the how long the **Event-Database Production Process** would take. And whether they were given a prognosis of how long each individual task the staff were given would take or whether they had to decide this themselves. In a functional hierarchy, the prognosis would be the same with respect to the overall process and each individual task, for the staff, the documents and the client (if available). In a dysfunctional hierarchy, the prognosis will either be missing with respect to the overall process and each individual task, for the staff, the documents and the client. Or the prognosis will be different with respect to the overall process and each individual task, for the staff, the documents and the client.

When the staff were given a new task to perform to add some feature to the game, for which there was no precedent, it would be very hard to estimate how long it would take perform. They could either research how to perform that task, then give an estimate from the results of that research and then add that feature. Using whatever **Events**, **Actions**, **Game Objects**, *Database Tables*, *Records* or *Fields* are required. Or they could do the research, the estimation, and the addition all at the same time.

But in the former case, although the research is open-ended, the estimation and the addition are not. This is the more scientific of the two methods. Whereas in the latter case, the research, estimation and addition are all open-ended. This is the more artistic of the two methods. The staff would literally be improvising the addition of that feature, in an ad-hoc non-repeatable manner. There is a lot of trial and error and a lot of time is consumed by the errors. This method is called 'Prototyping' in the Computer Games industry. Most modern *game-editors* excel at 'Prototyping'. This should not be confused with a real prototype in other industries.

In other industries, a real prototype is the first product of a production process built to test a concept or that process. Therefore, a prototype has to be complete for the test of that concept or process to be complete. But in the Computer Games industry, the feature being added to the game is not complete, and the process is not known. In other industries, the process that produces the prototype is known and repeatable. In the Computer Games industry, the process is unknown and therefore not repeatable. And the goal of 'Prototyping' is to find that process. In other industries, the process is known **before** the prototype is produced. In the Computer Games industry, the process is known **after** the prototype is produced. And this is done by using *Reverse engineering* to work backwards, from the prototype to the beginning of that process.

As already been explained, a functional hierarchy arises from the view of software production as a science, that relies on *Forward engineering.* The natural leadership arises from those who excel at promoting the higher-level tools that *Forward engineering* depends on. And this leadership would naturally promote the more scientific of the two methods. A dysfunctional hierarchy arises from the view of software production as an art that relies on *Reverse engineering.* The unnatural leadership arises from those who excel with the lower-level tools that *Reverse engineering* depends on. And this leadership would naturally favour the more artistic of the two methods i.e. 'Prototyping'.

Therefore, in a *Post-mortem* meeting at the end of an **Event-Database Production Process**, ask the staff whether when they came up with estimates for how long each task would take, whether they did so by a scientific 'Research' or an artistic 'Prototyping' method? If the latter, then ask the staff whether they felt they had enough time to do 'Prototyping' or not? Ask whether they feel very little or a lot of the 'Prototyping' work ended up in the final product? If the former (i.e. the staff do not use the word 'Prototyping' to describe how they estimated the time it took to complete task), ask them how would you describe the method you used to provide estimates?

In a functional hierarchy, the staff should have come up with estimates of time to complete tasks based on a scientific 'Research' method: not an artistic 'Prototyping' method. And therefore no 'Prototyping' work should have ended up in the final product. In a dysfunctional hierarchy, there will be lots of 'Prototyping' work. And there will not have been enough time to complete that work. And a lot of that 'Prototyping' work will have ended up in the final product.

4.3 THE TEMPLE OF THE BEAST

In the previous chapter, the view of time by a natural leadership of a functional hierarchy and an unnatural leadership of a dysfunctional hierarchy was discussed. Both would use different methods in the **Event-Database Production Process** to

evaluate how long it would take to perform tasks, to add features to a game and to perform those tasks. The use of a scientific 'Research' method and an artistic 'Prototyping' method was linked to the former's view of software production as a science and the latter's view of software production as an art. A scientific 'Research' method involves three distinct phases:

1. investigating how a task can be performed,
2. producing an estimation based on the results of the investigation, and
3. implementing that task.

The first phase is open-ended. But the last two phases are not open-ended and the last two phases are repeatable. An artistic 'Prototyping' method involves performing all three phases, the investigation, the estimation and implementation, all at once. As a result, all three phases are open-ended, and therefore not repeatable.

Now, in a functional hierarchy this distinction will be too subtle to make a difference to a natural leadership. Since the leadership views software production as a science and depends on *Forward engineering*, it will focus on managing the process, the **Event-Database Production Process**, not the staff. And promoting the use of higher-level tools in that process that *Forward engineering* depends on. The highest-level tool is natural language.

But in a dysfunctional hierarchy, this distinction will be huge. Since the leadership manages the staff, not the process. The unnatural leadership is a vicarious leadership which will focus on the staff the leadership uses vicariously to perform its role.

This distinction will be most obvious in how a natural leadership and an unnatural leadership view the use of Performance Reviews or Self-Appraisals to assess the staff. In these Appraisals, a natural leadership will find nothing of interest about the production process. But a unnatural leadership will find great interest in the promising staff who, through the use of an artistic 'Prototyping' method, look highly productive.

4.3.1 Tacit Approval and Disavowal

In the Computer Games industry, typically a Performance Review or Self-Appraisal will be used to assess the staff, periodically every three months or every six months. This Appraisal focuses on the staff involved in the production process, but not the process itself, which is normally a *Software Evolution Process*. Critics of Performance Reviews or Self-Appraisals say that the main problem with these Appraisals is that they focus on the staff rather than the process. The appraisal acts like a form of *Quality Control*, to detect when errors or defects have occurred in the process that the staff are part of. In this case, this process would be the **Event-Database Production Process**. As such Appraisals address the symptoms of the process (i.e. the errors or defects) rather than the cause which is the process itself. That in turn leads to a tacit approval of the process which may itself be the cause.

As has already been explained, an unnatural leadership in a dysfunctional hierarchy views software production as an art. And it relies on *Reverse engineering*. And there is a recession in a process led by an unnatural leadership, away from higher-level tools towards lower-level tools that *Reverse Engineering* depends on.

This includes a recession away from the highest-level tool which is natural language. And with the recession away from natural language, as in the *Software Evolution Process*, as in the *Tower of Babel*, there is chaos. The same outcome would occur in the **Event-Database Production Process** under an unnatural leadership.

Some of the staff who have worked elsewhere in the industry will recognise this chaos straight away as a symptom of the unnatural leadership. And tacitly approve or publicly disavow the leadership immediately. Those staff who are new and have not worked elsewhere in the industry will not be recognise it. They will believe that the chaos is simply the only way you can run a production process in the Computer Games industry, including the **Event-Database Production Process**. This will simply be *the nature of the beast*.[9] And they will give their tacit approval. However, the opportunity will arise when they will be forced to give an explicit approval or disavowal in a Performance Review or Self-Appraisal meeting.

In the Performance Review or Self-Appraisal meeting, each staff will be given a form with questions to answer before the meeting. The five most important questions will be

1. What have you done that you are proud of since your last review?
2. What could you do to improve yourself?
3. What were your goals in the last review?
4. How well did you meet those goals?
5. What will be your next goals for the next review?

After staff have answered these questions, the form will be given to the leadership, who will review the staff's answers and respond to them. In theory, each staff should see the leaderships' response to the answers, on the form, before the meeting. And then the meeting between the staff and the leadership takes place. And they agree in the meeting on an overall score and set new goals for the next Performance Review or Self-Appraisal meeting in a few months' time.

In practice, the unnatural leadership will not respond to the staff's answers on the form prior to the meeting. Instead, the leadership's response will come during the meeting. And thus, the staff will have no time to prepare any answers to the leadership's rebuttals. And after the exchange of unplanned arguments, the leadership tries to give a score for the staff's performance based on what was said in the form and the meeting. The score is normally one of five

1. excellent
2. good
3. average
4. below average
5. unacceptable

In these meetings, the staff will have a perfect opportunity to comment on the chaos they see around them that results from the unnatural leadership. They will have to either to publicly give their approval or publicly disavow it.

Note that none of the answers to the questions on the form would address the process at all. The questions would all be inward-looking, looking at how the staff view their own thoughts and feelings within the process and under the leadership. The questions would not be outward looking, drawing attention to the external factors, the production process and the leadership. As such Appraisals offer nothing to a natural leadership in a functional hierarchy. A natural leadership views software production as a science and depends on *Forward engineering*. It promotes the higher-level tools that *Forward engineering* depends on. The highest-level tool is natural language. None of the previous questions address this or any other subjects pertinent to natural leadership.

What was the natural language of the production process?
What was being communicated within the production process?
What was the vision for the production process?
What were the successes or failures of the steps of the production process?

Even though effective natural language and communication are one of the main goals of the **Event-Database Production Process** and the **Event-Database Architecture**, the questions in a Performance Review or Self-Appraisal will not address that.

The questions that are pertinent to natural leadership would be too profound to be contained in an Appraisal Form. The answers would be too long for such a short form. The focus of an Appraisal Form is the end product of a long process, not the process itself. Trying to answer such profound questions in an Appraisal Form, by giving a score to someone's views or feelings about that process, would be futile. It would be like trying to assess a long story by judging the moral of the story in the epilogue. Ignoring the prologue and the rest of the story.

The moral of a story is didactic. And it is not a coincidence that an Appraisal Form is sometimes read out by the leadership at the end of an Appraisal meeting like the moral of a story from an epilogue. As has already been explained in the previous chapters, a didactic form of communication is the preferred form of communication of an unnatural leadership and those who view software production as an art. And as a result, they depend on *Reverse engineering*. *Reverse engineering* does not require a dialogue. So they are not used to it. Instead, they prefer the more instructive language of didactic literature. And that is what Appraisal Forms come out as, a form of didactic literature.

An introspective or retrospective examination of your views, thoughts or feelings about some period in your life can be an opportunity to engage in **moral** philosophy. That is to say it can be an opportunity to learn something new and to grow.

But it can only be **moral** philosophy if the object of that examination is to find some objective truth about the world around you. It is not **moral** philosophy, however, if the object of that examination is to examine your views, thoughts or feelings for their own sake. That is **amoral** philosophy. And that is the kind of examination that occurs in Performance Reviews or Self-Appraisal Meetings.

4.3.2 Explicit Approval in Performance Reviews or Appraisals

As has been explained in the previous chapters, in a dysfunctional hierarchy, there is a marriage of convenience between the unnatural leadership and the staff. The leadership gives nominal authority and autonomy to the staff to make strategic decisions, to plan tasks and implement them. To practice the leadership vicariously through the staff and not get entangled in the day-to-day problems. And in return the staff cover up any problems or *Bugs, Hacks and Placeholders* in the software or the software production process in order to maintain this nominally authority and autonomy. And prevent the disasters that come from time to time with the leadership's intervention.

Therefore, in these Performance Reviews or Self-Appraisals, the unnatural leadership will be full of self-doubt due to the loss of perspective on the production process. That in turn comes from the leadership withdrawing from the process and conducting the leadership vicariously through the staff.

Some of the staff in turn will want to maintain their authority and autonomy. So, in the Performance Reviews or Self-Appraisals, they will ignore the chaos, and the problems caused by the recession away from higher-level tools, including natural language, towards the lower-level tools that *Reverse engineering* depends on. And that in turn the leadership explicitly or implicitly depends on because it views software production as an art.

In answering the main questions on the Appraisal form, they will give no hint of any problems. The list of achievements that the staff were proud off since the last review would be long. And the list of improvements they could make would be short. Almost all of the goals that were set in the last Performance Review or Appraisal would have been met without any problems. And the goals for the next review will be ambitious and make it seem that anything was possible in the project. And some of the staff will explicitly praise the leadership or the project.

In return the unnatural leadership will reward these staff with a high-score in the Appraisal meeting. Especially those who praise the leadership, show high productivity and complete a large amount of tasks. By using an artistic 'Prototyping' method to perform all three phases (i.e. investigation of each task, estimation of the time to complete the task and implementation of the task) in one continuous, open-ended phase.

In a functional hierarchy, with a natural leadership, neither the explicit approval of the staff nor the scores in Self-Appraisal meetings have any value. As already explained in the preceding subchapter, entitled **Tacit Approval and Disavowal**, a natural leadership would not find in Performance Reviews or Self-Appraisal meetings anything to address the kind of questions they would be interested in.

4.3.3 Self-Justification through the Benefits

In a dysfunctional hierarchy, those staff in the Performance Review or Self-Appraisals who will cover up the problems in the software production process, including the **Event-Database Production Process**, will justify themselves with whatever subsidiary benefits the company offers. These include contractual benefits such as flexible working hours, pensions, health insurance and paid holidays. These may even include rudimentary benefits such as free food, drinks or air-conditioning. These

also include other unofficial benefits such as financial security for any commitments, especially a rent or a mortgage.

Other benefits would include the opportunity to play new computer games, with these peers. As such, these would also include free passes to exhibitions, organised by the industry, where several of these games would be demonstrated. And these would include the opportunity to be the first to see, in public, any new technologies for building games, at these exhibitions. Furthermore, these would include free copies of games, which the staff had worked on, which they received after the games had been released. And these would include any credits the staff would receive for this work, either in the games themselves or during the marketing of these products. So that the staff could use this to market themselves when they seek new jobs. In this respect, the benefits would also include the opportunity to have some of their marketable ideas added to a *game design*. But, significantly, this would not include all of their ideas. Their most cherished original ideas would be guarded jealously. And this in turn is no doubt partially if not wholly related to the lack of original ideas in the games in the industry. There are many staff with great ideas. But they dare not bring them out into an avaricious industry.

In the Computer Games industry, these benefits would include always being able to dress casually. And these would include being able to listen to music and eat while working at your desk in front of the computer. These would include the opportunity to play computer games during lunchtime, or after work, with other staff.

But significantly, when they play these games at lunchtime or after work, they will not play the unfinished games there were working on at the company. They will only play games produced by other *Software Developers*. Since their experience with developing that unfinished game, the production process, the Planning meetings, the Appraisal meetings, the numerous crises, the arguments, the overtime, will have left a bitter taste in their mouth. And they can see all the signs of these past and current problems when they play the game, which a casual observer cannot see.

Also significantly when they play these games, they will not play with all of the staff. But instead with a handful of the staff who were willing to bridge the gap that naturally grews between all of them, under the unnatural leadership of a dysfunctional hierarchy. As has already been explained, the unnatural leadership is in fact a vicarious leadership or an absence of leadership. Into the void that is left by the leadership, the staff compete to fill that void and manage themselves. The competition creates acrimonious relationships which creates a gap between some of the staff. And the self-management leads to some of the staff working in isolation for long periods.

In a functional hierarchy, the natural leadership is the bridge to that gap between the staff. The leadership bridges that gap through natural language. As has already been explained, the natural leadership views software production as a science and relies on *Forward engineering.* And that means in turn it promotes the higher-level tools that *Forward engineering* depends on. And natural language is the highest-level tool. By promoting natural language between the staff, the leadership promotes communication between the staff. And that naturally brings the staff closer together. The cult of personality or popularity of the leadership, which the leadership enjoys amongst the staff, also naturally brings the staff closer together.

Therefore, you can make another distinction between a natural leadership and an unnatural leadership, by testing how close the staff are together when they play

games. Set apart a day during the week or hours during the day where the staff will play games. This can be just another benefit on top of the other benefits they receive from the *Software Developer.*

For this benefit, ask the staff to play a computer game, that involves teams of players. Ideally that game should be the game that the staff were working on in the production process, in this case the **Event-Database Production Process**. The game should start in a non-competitive part of the *Game World*. And it should allow the players to freely form their own teams in this location, before the competitive part of the game begins. The game itself should not force them into teams. And the leadership should not force them into teams. The teams can be of any size from one player to over 100 players. And after the players have formed their teams, the teams should be moved into the competitive part of the *Game World*.

In a functional hierarchy, with a natural leadership, the teams formed will be of equal size, give or take one or two players. And assuming there were more than four players, there should be no teams with just one player. In a dysfunctional hierarchy, with an unnatural leadership, the teams formed will be disproportionate in size. And there will be teams with just one player.

Now there is nothing wrong with company benefits. These may motivate the staff to be more productive. But these only benefit the production process if the objective of those benefits is for some greater good. Rather than having benefits for benefits sake.

4.3.4 Explicit Disavowal in Performance Reviews or Appraisals

When some of the staff come up for a Performance Review or Self-Appraisal meeting, in a dysfunctional hierarchy, they will dissent. They will reveal the problems, *Bugs, Hacks and Placeholders* or errors in the software.

They will claim to have done nothing that they were proud of, or almost nothing.

They will object to the premise of the question,

> What can you do to improve yourself?

That is to say, that the staff must be at fault for anything that has gone wrong. And instead turn it around with the question,

> What can the production process or the leadership do to improve itself?

They will deny having met any of the goals that were set in the last review, or almost none.

And their next set of goals for the next review will be the same as for the current review, more or less.

The unnatural leadership will react to this dissent by coercing, or attempting to coerce, the staff to feel guilty. Either the leadership will use the inevitable mistakes

and problems the staff will stumble over, that naturally arise in the unnatural leadership. Or the leadership will try to get the follower to incriminate him or herself.

For example, as already previously mentioned, one of the traits of an unnatural leadership is that it views software production as an art. And it makes lots of ad hoc decisions in the process. Another trait is that the unnatural leadership is a vicarious leadership. The leadership uses the staff vicariously to conduct its role. The leadership gives the staff nominal autonomy and authority to perform tasks. But later on undermines that autonomy and authority. By overriding the staffs' decisions arbitrarily on an ad hoc basis.

So suppose the staff decided to use some tool to perform a task. Suppose a *Game Artist* decided to use a third party *computer-aided design* (or CAD) tool to create a 3D model of a character and animate it. Or a *Game Programmer* decided that they would use some programming language to create a tool to test the process of building the game. After they had made the decision, the leadership comes along and decides that the software licence for the third party *CAD* tool is too expensive. So the *Game Artist* should use another tool to complete that task. And the leadership decides that the programming language that the *Game Programmer* wants to use is not approved by the company. It is not the tool which is used elsewhere in the company for other projects. So the *Programmer* should use another tool which was approved by the company.

After changing the tools that the *Game Artist* and the *Game Programmer* used, the tasks ends up taking longer than either of them had scheduled for. And the *Game Artist* and *Game Programmer* brings this up in the Performance Review or Self-Appraisal. They point out the adverse effect on their schedules caused by the leadership's decision to overrule them. Instead of being concerned with any damage that may have been caused, by the decision to overrule, the leadership will be more preoccupied with cultivating the empathy of the staff. Aware that the staff has the intelligence (i.e. sensibilities) to look beyond the confines of their role, the leadership will insult those sensibilities. The leader will suggest that it was a lack of empathy, on the staff's behalf, for the leaderships' problems, to observe these adverse effects.

The *narcissism* will not stop there. It will go even further if the leadership feels its credibility would be damaged by, for example, having to include the oversight in a report, like the Appraisal form in a Performance Review or Self-Appraisal meeting. The leadership would hold the staff responsible for the leadership's ignorance. The staff's own observation about the consequences of the leadership's decision to overrule them, in this case forcing the *Game Artist* or *Game Programmer* to use one tool instead of another, would be used to incriminate them. This observation would be presented, in the Appraisal form, as a confession, as evidence of the staffs' lack of communication. The staffs' reputation would as a result suffer, and they would be punished for their honesty. Despite the fact that the leadership would have invited the staff, to be honest about the workplace, at the start of the Performance Review or Self-Appraisal meeting.

Bear in mind, before the meeting, the leadership would have gone out of the way, to marginalise itself from the day-to-day problems in the workplace. The leadership would have delegated responsibility to the staff. The leadership would have let the staff come up with the plans to accomplish long-term goals, to break down those

plans into smaller tasks, to come up with the schedules for these smaller tasks. The staff would have assigned some of these smaller tasks to other members of staff, over whom they had no real authority or control. The leadership would have come in and revised these plans on an ad hoc basis. Leaving the staff to face the consequences. The leader would have delegated help to others, when the staff presented subsequent problems. Yet despite all this effort, on the leadership's behalf, to extricate himself or herself from the problems, the staff would be held accountable, for underestimating the depths of the leadership's ignorance.

When the staff who dissent, do not get the message from the Performance Reviews or Self-Appraisals meetings, the leadership will merely double down with more meetings. Normally these meetings should take place either once a year, or every half a year i.e. six months, or every quarter i.e. three months. But in a dysfunctional hierarchy, the unnatural leadership will reduce this down to even shorter intervals e.g. every two weeks. And these more frequent reviews will be done without all of the formality of a normal Review or Appraisal meetings, with all the forms they would normally fill in. And these will not be even called 'Reviews' but something more informal or Orwellian, like 'Catch-up' meetings. Nevertheless, content of those informal meetings will be the same as the formal ones.

Again, in a functional hierarchy, with a natural leadership, neither the explicit disavowal of the **Event-Database Production Process**, by the staff, in Performance Review or Self-Appraisal meetings nor the frequency of the meetings would have any value. As already explained in the preceding subchapter, entitled

Tacit Approval and Disavowal,

a natural leadership would not find in Performance Reviews or Self-Appraisal meetings anything to address the kind of questions it would be interested in.

4.3.5 Self Incrimination in Self-Appraisals

As explained in the previous chapter, entitled

The Time of the Beast,

self-incrimination characterises an unnatural leadership in a dysfunctional hierarchy. Self-incriminating evidence is the only form of evidence that comes out of a dysfunctional hierarchy. The staff whom the leadership nominally delegates responsibility to achieve long-term goals end up incriminate themselves with what they produce. With their plans to achieve those goals, the breakdown of plans into tasks, schedules for the tasks that they come up with to achieve those goals all end up as self-incriminating evidence in the Self-Appraisal meetings. When the leadership meets the client or financial backers of a project, they end up incriminating themselves with what they produce. The long list of features in the *game design* they use to sell the project between each milestone, all end up as self-incriminating evidence at the end of the milestone or in the *Post-mortem meetings*. And with this self-incrimination there will be an ever pervasive air of guilt.

Some of the staff will falsely believe that, by actually meeting the leadership's expectations, they will be able to escape the feelings of guilt. But even when they do meet or exceed the leadership's expectations, the leadership may still make them feel guilty if they exhibit any signs of dissent, during a Performance Review or Self-Appraisal meeting.

For example, suppose the leadership gives the staff, a *Game Programmer*, the task of creating two tools, an Automated Build System and an Automated Testing System. To automatically build the game, based on the **Event-Database Architecture**, at the end of each day. And run the game through an automated test. And produce a report indicating the success or failure of that test. And send the report in an e-mail to a set of Software Users which the leadership could specify in the parameters of the Automated Testing System.

Now suppose that these two Systems would normally take six months to build. But the *Programmer* did it in two months using a particular tool which they found on the Internet, which they were proficient in. However, after the first iteration of the two Systems were built, and reviewed by the leadership, the leadership rejects the two Systems. The leadership complains that the tool used to build the Systems was not what they expected. Even though the leadership did not specify any tool to be used when the task was given.

Instead the leadership asks for the two Systems to be rebuilt again using another tool which it considered was part of the company's 'ecosystem'. That is to say, the Systems had to be built with a tool from a limited set of approved third party tools. That met the company's corporate strategy to work in partnership with these third parties. Such as the tools approved by the *Software Developers* of the commercial *game-engine* that the company happened to be using.

Now suppose in the meeting that the leadership asked for the Systems to be rebuilt, the *Programmer* dissents. And the *Programmer* tries to defend the decision to build the original two Systems using the tool found on the Internet. Claiming that this was a faster process than using the tool which was part of the corporate 'ecosystem'. And the *Programmer* questions the decision to use the tool which was part of the corporate 'ecosystem'. Nonetheless, at the end of the meeting, the *Programmer* agrees to use the new corporately approved tool. And rebuilds the two Systems again taking another four months.

When that *Programmer* comes up in a Performance Review or Self-Appraisal meeting, all of the accomplishments of the original two Systems built in two months, ahead of leadership's expectations will be forgotten. Even though this exceeded the leadership's expectations which was six months. Instead the leadership will accuse the *Programmer* of wasting time by using tools which were not part of the approved corporate 'ecosystem'. The *Programmer* will be accused of lack of communication and beginning tasks without asking questions. While at the same time the *Programmer* will also be accused of asking too many questions. For questioning the decision to replace the old tool used to build the two Systems with the new corporately approved tool.

By having attempted to complete a task in a shorter time than the leadership expected, and admitting to using a tool found on the Internet to do so, without consulting the leadership, in the vain hope that the leader would be pleased, that staff

would have fallen into a trap. The staff would have provided the leadership with what it construes as a confession, evidence that would only be used, to incriminate them in a Performance Review or Self-Appraisal meeting, when they dissent.

This is a reoccurring theme in a dysfunctional hierarchy, under an unnatural leadership. There are many problems, *Bugs*, errors, poor language, poor communication and vague tasks. But most of it is suppressed because of the marriage of convenience between the leadership and the staff, as explained in the previous chapter entitled

The Marriage of the Beast.

As time goes by, the seething mass of problems slowly trickles up to the surface. And there are staff who stumble across them but keep this to themselves. Even when the leadership asks for the staff to be honest, for example in a Performance Review or Self-Appraisal meeting, they will remain silent. But from time to time some of the staff will share intimacy with the leadership, about the problems they have come across. And the leadership will use this intimacy against them, as self-incriminating confession and evidence, when the staff dissent.

Again, in a functional hierarchy, with a natural leadership, neither confessions or self-incrimination in a Performance Review or Self-Appraisal meeting or elsewhere would have any value. As already explained in the preceding subchapter, entitled

Tacit Approval and Disavowal,

a natural leadership would not find in Performance Reviews or Self-Appraisal meetings anything to address the kind of questions it would be interested in.

4.3.6 The Right to Silence

You would have to go back hundreds of years to find well documented public cases of where confessions or self-incrimination of the kind seen in dysfunctional hierarchies was used. In the English courts of the Star Chamber and the High Commission, those who were accused were bound to answer questions of their interrogators or be tortured. Even if they might incriminate themselves. This practice was widely hated, because it was believed to be used primarily for squashing religious dissent. So these courts were abolished and soon afterwards the *right to silence*[10] was established.

Back then, you were required by law to attend church on Sunday and join in the service to worship God. And the government dictated the content of that service and what form that worship would take. If you criticised the government's conduct in this, or other religious matters, then you could be brought before the courts of the Star Chamber or High Commission. Where your interrogators had the power, by law, to torture you for failing to answer self-incriminating questions. In the Performance Reviews or Self-Appraisal meetings, the technique is more subtle.

The staff who dissent under an unnatural leadership are brought before the Self-Appraisal meetings. Like religious dissenters brought before the court of the Star Chamber. They are bound to answer self-incriminating questions or be tortured. The premise of these questions is that there is nothing wrong with the leadership, nor the

production process. And that the only case to be answered for is the performance of the staff in that process.

The leadership will use vague complaints from anonymous sources, and false concern, in an attempt to get the staff to reveal or confess to something incriminating.

> 'The team is concerned that when you do tasks and it comes up for review, it is not quite what was expected. And a lot of work has to be done to correct it, that wastes time. It seems as though you don't ask questions when you are given task, to clarify what is expected.
> But on other occasions, the team also feels that you ask too many questions. And you don't make an attempt to work things out for yourself. You don't seem to exercise the right judgement in deciding when to ask and when not to ask questions.
> What do you have to say for yourself?'

The staff are bound to answer questions of their interrogators. Even if they might incriminate themselves, by revealing the problems under the leadership. Or else face the torture of having to deal with the seething mass of problems that arise under the leadership and holding these down. Even though these problems naturally keep trickling up to the surface.

In light of which it is perhaps time, with respect to how Performance Reviews or Self-Appraisal meetings are conducted, for the temple of commerce to catch up with the temple of God. And a new *right to silence* should be established for the new Star Chamber. That is to say, the staff subject to a Performance Review or Self-Appraisal meeting should have the *right to silence*. They should not be required to say anything or answer any questions in their defence in these meetings. And that should not be taken as a sign of their guilt. The burden of proof should be on the prosecution bringing any complaints or accusations against the staff. No accusation should be entertained unless it is corroborated by two or more witnesses. No hearsay or anonymous sources should be admissible. By default, the staff's performance rating should be excellent. And in the absence of any corroborated evidence being presented in the meeting that it should be anything less, that should be final outcome of the meeting.

Again, in a functional hierarchy, as already explained in the preceding subchapter, entitled

Tacit Approval and Disavowal,

a natural leadership would not find in Performance Reviews or Self-Appraisal meetings anything to address the kind of questions it would be interested in.

> What was the natural language of the production process?
> What was being communicated within the production process?
> What was the vision for the production process?
> What were the successes or failures of the steps of the production process?

None of these are self-incriminating questions. None of the answers to these questions can be taken as a confession. These are questions about the process, the **Event-Database Production Process**: not the staff. Therefore, a natural leadership has no interest in self-incriminating questions or confessions, let alone punishing those who dissent.

4.3.7 Human Resource and Human Beings

In Performance Reviews or Self-Appraisal meetings, in a dysfunctional hierarchy, it would be no use for the staff to appeal to some authoritative body, in the organisation, for arbitration; such as the *Human Resources*.[11] In theory, this body will claim to be available to impartially settle disputes between the leadership and the staff. But, in practice, this body will merely be an extension of the unnatural leadership. This body, more than any other part of the organisation, would share a deep intimacy with the leadership.

Especially in the Computer Games industry, this intimacy develops from the *high turnover of the staff* that occurs underneath an unnatural leadership. That in turn requires the *Human Resources* to hire more staff, typically undergraduate interns or recent graduates eager to work in the industry. To replace those who have left for one reason or another. Without this intimate relationship, the unnatural leadership would simply run out of staff. Who are treated as chattel, resources, commodities for the leadership to consume, instead of human beings.

Another thing that binds the two together is that *Human Resources* and many of the unnatural leadership come from the same academic background, the same *academic courses* mentioned previously in the chapter

The Nature of the Beast.

That claim to teach its graduates to become leaders. But in fact merely train them to become vicarious leaders, who reduce leadership down to delegation of responsibility and enforcement.

So *Human Resources* will just be another means by which the unnatural leadership will vicariously gather any self-incriminating evidence or confessions, against any staff who dissent. Any intimate details about the hierarchy, which the staff involved in the **Event-Database Production Process** revealed to *Human Resources*, would be passed on to the leadership. And the leadership would use this as a confession, to incriminate the staff who dissent; typically in some form of an ambush meeting.

Again, in a functional hierarchy, as already explained in the preceding subchapter, entitled

Tacit Approval and Disavowal,

a natural leadership would not find in Performance Reviews or Self-Appraisal meetings anything to address the kind of questions it would be interested in. Therefore, it would not use these meetings to assess staff. And the staff would not need to appeal to some independent arbiter like *Human Resources*.

4.3.8 Unions and Performance Reviews or Appraisals

In the previous chapter, a *right to silence* for the staff was suggested as one way for improving Performance Reviews or Self-Appraisal meetings. Other critics have suggested a more democratic model for doing business as another way to improve these meetings. An organisation where all the members, of a team, were held jointly accountable for the performance of any individual in that team, would conduct these Self-Appraisal meetings differently.

Any meeting would require either that all the members of the team be present. Or that a representative from the same team as the staff under review be in attendance. And any accusation or complaint against any one in that team should be treated as an accusation or complaint against the whole team. And any punishment or sanction against any one in that team should be treated as a punishment or sanction against the whole team. This is the approach of trade unions to industrial disputes with employers, including Performance Reviews or Self-Appraisal meetings.

That is to say, the trade unions recommend bringing a colleague with you to any Self-Appraisal meeting. To be a witness to what is said and done in the meeting. And for that colleague to weigh up any accusation or complaint against you, with their experience under the same leadership. And to weigh up any punishment or sanction against you with their experience under the same leadership. Some employers will not allow any accompanying colleague to speak in these meetings. But they will not deny any request you make to be accompanied, since this will be frowned upon if you take your case to an Employment Tribunal.

If the leadership is willing to hear that colleague at that meeting, then that colleague can defend you against complaints with their experience. Or defend you against any punishment or sanction with their experience of similar disciplinary action under the leadership.

And if the leadership were not willing to hear that colleague in the meeting, then you can take your case to an Employment Tribunal. Where that colleague can testify on your behalf when that Employment Tribunal hears your case. And where you can get financial support from your trade union for any solicitors or advocates who present your case before the Employment Tribunal.

Unfortunately, in the Computer Games industry for a very long time, there have been no trade unions. All attempts to create trade unions in the industry had failed. But recently a new union *IWGB Game Workers*[12] has been established which workers from the industry can join.

4.3.9 Natural Leadership: A Manager of Processes

As has been said in the previous chapters, a natural leadership in a functional hierarchy would find nothing of value in a Performance Review or Self-Appraisal meeting. Since it does not address the kind of questions that the leadership would be interested. As already explained in the preceding subchapter, entitled **Tacit Approval and Disavowal**, these are questions about the process, in this case the **Event-Database Production Process**. Whereas Performance Reviews or Self-Appraisals focus on questions about the staff in that process.

More precisely, some critics claim that the Performance Reviews or Self-Appraisal meetings look at the end product of a process or system that produces the staff and their work. It does not look at the process or system itself. It acts as a form of *Quality Control* that accepts or rejects a product at the end of a production process. That is all *Quality Control* can do, accept or reject a product. It does not actually improve the *Quality* of the product. It tells you nothing about the process or system which produced it. Likewise Performance Reviews or Self-Appraisals can only accept or reject the staff and their work produced by some process or system. It cannot improve the staff, the *Quality* of the work or the process. And it tells you nothing about the process or system which produced them.

To improve the *Quality* of a product of a process or a system you must have *Quality Assurance*[13]: not *Quality Control*. *Quality Assurance* is systems in place which allow you to scrutinise the steps of a production process or system. To identify when errors enter the steps of a process or system at any point. To decide whether or not the error was due to a flaw in that process or system. And if it were, to fix that step in that process or system. And if it were due to staff to train them to perform that step.

Training should be available for every step in that process or system and should therefore be institutionalised.

This would suggests that a natural leadership which views software production as a science and relies on *Forward engineering* in software production, would be ideally suited. Since, as explained in previous chapters, a natural leadership promotes the higher-level tools that *Forward engineering* relies on. And the highest-level tool is natural language. A natural language helps institutionalise training the staff in the steps of the production process, the **Event-Database Production Process**. And a natural leadership promotes a scientific 'Research' method to investigate new tasks, estimate how long the tasks would take and implement those tasks. This form of investigation is repeatable and helps institutionalise training the staff to use it. And that in turn helps improve the skill of the staff. And that in turn enables a natural leadership to improve the production process and the *Quality* of its product.

But in the Computer Games industry training is typically not institutionalised. Employees are typically expected to already know their roles from prior experience elsewhere in the industry. Or to work on a portfolio in their spare time which shows that they have already had the necessary experience. And so require no training for whatever role they are placed into in the production process. Performance Reviews or Self-Appraisals are used as a form of *Quality Control* to ensure the *Quality* of the final product. But as has already been said, *Quality Control* cannot improve the *Quality* of any product or the staff. It can only accept or reject a product.

For this and other reasons, many critics of Performance Reviews or Self-Appraisals view them as a *substitute for leadership*.[14] And that these should be abolished.

One of the other reasons critics suggests that Performance Reviews or Self-Appraisals should be abolished, is that they are often connected to disciplinary action. When staff get a low score in a Performance Review or Self-Appraisal, they are moved on to a disciplinary process where they are continuously monitored to see whether their behaviour improves. To see for example, if the outcome of the Self-Appraisal is that they do not ask questions to clarify tasks before they begin, that they do ask questions when they are given new tasks. Or if the outcome was that they

ask too many questions when given tasks, that they do not ask too many questions. Or if they were told not to use some tool to perform some task, and only use tools sanctioned by the company, that they did only used sanctioned tools. Or if they did not follow the company's *Naming convention* for writing code or naming files, that they do follow that convention when they generate the next piece of code or name new files.

And if the staff fail this test, they are then given a first written warning. And if they fail again, they are then given a second written warning. And if they fail again, then they are dismissed.

For this reason *Arbitration services*[15] that give advice about how to conduct reviews agree with the critics to an extent. Although they do not suggest abolition, they do suggest that disciplinary process and disciplinary action should be kept strictly separate from the review process.

4.3.10 Unnatural Leadership: A Manager of Defects

In a dysfunctional hierarchy, an unnatural leadership, however, would find Performance Reviews or Self-Appraisals valuable. As explained in the previous chapters, an unnatural leadership is a vicarious leadership. The leadership conducts itself through an army of delegates it gives nominal autonomy and authority to perform its function. For this reason, the focus of the leadership is on the delegates. That is to say the staff through whom the leadership acts vicariously.

As a result Performance Reviews and Self-Appraisals of the staff is an invaluable tool to detect problems with the production process, in this case the **Event-Database Production Process**. But, as has already been explained in the previous chapter, critics say this is merely a form of *Quality Control* that detects errors or defects in products at the end of a process or system. In this case, the product is the staff and their work.

This *Quality Control* does not examine the process or the system. It can only accept or reject the product at the end of the process. It cannot improve the process, the *Quality* of the product, the staff or the work they produce. All the unnatural leadership can do is respond to these errors or defects that the Performance Reviews or Self-Appraisals reveal. As explained in the previous chapter, this is *a substitute for leadership.* That makes the unnatural leadership not a manager of staff, nor a manager of processes, but a manager of defects.

When an unnatural leadership gives a low score in a Performance Review or Self-Appraisal meeting to some of the staff, and then proceeds to move them on to a disciplinary process as a result, this will not improve the staff. This will merely end up either, at best, making the staff struggle just to give the appearance of passing whatever test was set before them in the disciplinary process. Although they cannot sustain this appearance for long. Or at worst, this will end up alienating, isolating and ultimately rejecting the staff. Thereby undermining the whole process, and encouraging other staff to suppress errors or defects in the production process, in Performance Reviews.

In the Computer Games industry, many like to claim they offer training to staff. And thereby imply that they improve their skills and the production process. But the

training they offer is only on the job training, which is no training at all. The staff are thrown into the deep end and expected to either sink or swim. There are senior staff members around, with experience, such as *Team Leaders*, who in theory you can ask questions. But in practice they are normally too busy dealing with the problems caused by the unnatural leadership that they have not got the time to help.

Furthermore, when they do have time, they will be of little use. These senior staff will be part of the unnatural leadership. As a result they view software production as an art and rely on *Reverse engineering* in software production as well. And they will promote the lower-level tools that *Reverse engineering* relies on. They will expect junior staff to use these lower-level tools to learn by themselves. And to use an artistic 'Prototyping' method to investigate new tasks. Although as explained already in the previous chapter entitled

The Time of the Beast.

This method is not repeatable and therefore counter-intuitive to any teaching process.

NOTES

1. ***Event-Database Architecture Knowledge Test.*** A multiple choice test where each question asks you to select the meaning of names of Events, Actions, Game Objects, Database Tables, Records and Fields in the Game Database of the Event-Database Architecture. Where the correct answers come from the definitions of these items in the data design.
2. ***Good memory (to a leader).*** A good memory helps a leader love the work and stops him or her hating it, through the frustration of repeated mistakes. This gave rise to the idea of a philosopher-king in antiquity.
3. ***Academic course (about leadership).*** There are several post-graduate courses available which claim to teach students how to become business leaders. The influences of these courses are very wide ranging. But these courses have been oversold and produce false leaders.
4. ***Post-mortem meeting.*** A meeting conducted at the end of a software production process, by the staff involved, to examine the pros and cons and the lessons to be learnt from the experience.
5. ***Third Party Game Testers.*** A company (e.g. Universal Speaking Ltd.) that specialises in performing the function of the QA Department in the Computer Games Industry including testing the game at the end of the production process.
6. ***Narcissism.*** An excessive love or pre-occupation with yourself, which leads to a decay in your ability to empathize with others.
7. ***Prevalence of women.*** A recent research into the characteristics of the workplace, in high-technology industries, found that the macho, competitive atmosphere was a barrier to women.
8. ***Event-Database Architecture Productivity Formula.*** A formula for measuring the productivity of any software production process by inverting amount of waste produced. See the subchapter entitled Cause and Effect.
9. ***The nature of the beast.*** A figure of speech which means a regrettable but inescapable characteristic.
10. ***Right to silence.*** In English law, and in other countries, a person charged with a crime has the right to remain silent before and during trial in order to avoid saying anything incriminating. This is the basis of placing the burden of proof on the prosecution. See Glossary.

11. ***Human Resources.*** A department of an organisation responsible for the recruitment, payment and personal welfare of staff.
12. ***IWGB Game Workers.*** A trade union established in 2019 for workers in the Computers Game industry. It deals with many common issues in the industry such as overtime, sexism and harassment. See Glossary.
13. ***Quality Assurance.*** In theory, a system which ensures that a company's processes (as supposed to their product) will meet all of the customer's requirement and specifications. In practice, software companies just apply two Quality Controls in the latter stages of production, known as Alpha and Beta testing, and call it Quality Assurance or QA. See Glossary.
14. ***Substitute for leadership.*** In his 14 points for manufacturing quality goods, W. Edwards Deming suggested that the use of annual appraisals, to measure and improve the performance of an employee, were in effect a substitute for leadership. See Glossary.
15. ***Arbitration service.*** A charity or a commercial company that mediates between two sides involved in an industrial or employment dispute. See Glossary.

5 Cause and Effect

The success or failure of a software production process, such as the *Software Evolution Process* or **Event-Database Production Process**, is an effect. And the cause of that effect is the productivity of the staff. If the productivity were high and the waste were low, then greater would be the likelihood of success. If the productivity were low and the waste were high, then greater would be the likelihood of failure.

But what is productivity and waste? How do you measure productivity and waste?

To answer this question you need some definition of productivity or waste. You need some standard to compare against the productivity or waste of the *Software Evolution Process* or the **Event-Database Production Process**. That is to say, you need some standard production process you can use to measure other process and give a definition of productivity or waste..

Unfortunately there is no standard definition of productivity or waste. And that in turn is because there is no standard software production process in the Software industry, including the Computer Games industry. And indeed there has been no agreement about the *role of design* in the Software industry and Computer Games industry. Outside of academic circles, where it would be otherwise impractical to assess work, there is no requirement for designs in software production. In fact, there is no standard practice for *Software engineering*. There are no Trade Unions that might attempt to define the role of a Software Engineer. There are standard bodies, such as the Institute of Electrical and Electronics Engineers or IEEE. But these are not as influential as their medical counterparts. These bodies do not require any ethical responsibility for other people: there is no oath to do no harm. Instead, the standard bodies mainly set technical standards.

Though they do have standards for practising *Software engineering*, these have been widely ignored in the marketplace with notable exceptions. These exceptions are, namely, government contracts, especially *military contracts*,[1] where these standards are used to lethal effect. For this lack of widely recognised ethics, standard practices and other reasons, national judiciaries have refused to recognise *Software engineering as a profession*.[2] And perhaps the **Event-Database Architecture**, and the accompanying **Event-Database Production Process**, will establish recognised ethics and standard practices at least for the Computer Games industry for the first time.

For some, the very fact that a software project had no detailed records about its production process would be evidence at least of the lack of leadership. Without such records, the leadership could not remember any mistakes that were made. The leadership would not be able to account for how long certain tasks took and how successful these were. Thus, they would not be able to assess the risk of any decisions they repeated in that project or would repeat in future projects.

Nevertheless, for some the fact that a software production process had not produced any useful relics or records, except the final product, would not be evidence

 DOI: 10.1201/9781003502807-5

of a lack of leadership. Nor would it be evidence of the absence of a production process and any ill effects this would allegedly cause. Indeed, for them, such an informal approach to software production would actually constitute a style of leadership; rather than the *absence of leadership*.[3]

So how can you test both claims? That the absence of useful relics or records from a software production process, such as the *Software Evolution Process* or the **Event-Database Production Process**, is an *absence of leadership* and low productivity? And those who claim that it is not an *absence of leadership* and low productivity? And conversely, how can you test the claim that the presence of useful relics or records that allow you to scrutinise the software production process is a sign of the presence of leadership and high productivity?

Whatever the test may be, an important part of this test would be the accuracy of the measurements of productivity. For these measurements to be accurate the method must be sensitive only to work which contributes to the final product. That is to say, it must ignore waste.

It follows, from this requirement, that the measurement of productivity has to be inversely proportional to the waste a project produces. So one possible formula for measuring productivity would be simply to calculate how much waste was produced and invert it.

In the *Software Evolution Process* or **Event-Database Production Process**, the waste produced would be proportional to ten factors.

The first factor would be how many different software components were used to build the final product. The software components would include the *software data*, *software modules* and *software libraries*.

The second factor would be the number of components in the *User Interface* of the final and intermediate products. The components of the *User Interface* would include the various interactive menus or locations in the *Game World*.

This would include any icons, buttons, 2D images, 3D models, words or other items on the *Interface*, through which commands could be issued or responses received, either on a menu or in a location of the *Game World*. This would include any sounds or music which could be heard. This would also include any animated icons, buttons, images, models, words or static screens with which the software user could not interact. And this would include any messages which were there simply for the purpose of debugging the product. Although the software user may not see these messages, the staff would. And the more of these there were, the more time the staff would waste introducing, understanding and negotiating these components.

The third factor would be how many different software components were used to build the custom tools, used to build the final product.

The fourth factor would be the number of components in the *User Interface* of these tools.

The fifth factor would be how many times the designs were modified, of all of these components, in the final product and the custom tools.

The sixth factor would be the number of upgrades, of the software components, that were performed in the intermediate and final product.

The seventh factor would be how many times the components of the *User Interface* of the intermediate and final product were updated.

This includes how many times any of these software components or *User Interface* components were physically rebuilt, on all the computer hardware the staff use. It should not only take into account the number of times these were rebuilt, on special computers reserved for building the latest version of the game, sometimes called 'Build Machines'. Nor should it only take into account the number of times these were rebuilt by a special member of staff whose role it is to build the latest version, sometimes called a 'Build Engineer'.

Instead, it should take into account the number of times any of these components were rebuilt by any member of staff. The greater the number of these upgrades that occurred, during a project, then the more waste the staff would have produced. Either there would have been a greater number of staff working on different versions of the game, at the same time. Or there would have been a greater number of staff uncertain about the latest version of the game. And hence, they would produce more waste because of this disparity or uncertainty.

The eighth and ninth factors would be almost exactly the same as the sixth and seventh. The only difference would be that these would relate to the custom tools. That is, the eighth would be a total of how many upgrades were performed on the software components, used to build the custom tools. And the ninth would be a total of how many upgrades were performed of the components of the *User Interface*, of these tools.

The tenth factor would also be related to upgrades. But not so much to the upgrades of components as to the upgrade of entire tools. Namely, this would be the number of upgrades of the third-party tools including the game-engine, that were used during the project. The greater this number, the more software components that were already built with these tools would have to be rebuilt. Or more time would have been wasted performing these unnecessary upgrades and getting accustomed to the changes in the latest versions.

However, the waste produced in a project would be inversely proportional to six factors.

The first factor would be the number of records of the different software components, used to build the final product, that were kept.

The second factor would be the number of records kept about the different components of its *User Interface*.

The third and fourth factors would be the same as the first and second but relate to the custom tools used to build the final product.

And the fifth factor would be the number of records kept, of the modification of all these components.

The relationship of these records, to the waste produced in a project is this. When a project lacks a record of the design of its software components or its *User Interface*, then more waste has to be produced as a result. Each time someone wants to modify the design, he or she has to informally document it first. So, at the end of the project, the less formal records kept about the design of its software components or *User Interface*, the more informal documentation has to be produced to modify those software components or *User Interface*.

The final factor that would be inversely proportional to the waste produced would be the number of records kept about the upgrades that were performed. That is to say, the fewer records kept about the number of upgrades, of any of the software

$$W = \frac{CfCuCtCi(M + Uf + Uu + Ut + Ui + Up)}{RfRuRtRiRmRp}$$

FIGURE 5.1 Formula for calculating waste in a software production process.

components, the components of the *User Interface*, the custom tools or third-party tools, that were performed by the staff, the more waste would be produced. Since either some members of staff would make wrong assumptions about the version of the final product, custom tools or third-party tools, which others had. Or they would keep insisting that other staff waste time rebuilding or acquiring the latest version of the product, custom tools or third-party tools. So that they could, for example, accurately report a *Bug*. Even though the other staff may already have the latest version of the latest relevant components. In the Computer Games industry, the time it would take to rebuild the latest version of the software would not be negligible. It could be anywhere from 30 minutes to 14 hours on a large project.

Formulas for measuring the waste and the productivity of a project are shown in Figures 5.1 and 5.2.

With these two formulas, you can test the claim that the absence of useful relics or records that allow you to scrutinise a software production process, such as the *Software Evolution Process* or the **Event-Database Production Process**, is an absence of leadership and a sign of low productivity. And you can test the counter-claim, that the absence of these records is not an absence of leadership or a sign of low productivity but just another style of leadership. By letting a production process begin, with no requirement to keep records, and suspending it after a suitable period, for example three months. At this point you gather the parameters required by the formulas and use these to calculate the values for productivity and waste (see Figures 5.3 and 5.4).

After that, you resume the production process again but with a requirement that it keeps records that allow you to scrutinise it, and suspend it again, after three months. Again you gather the parameters and calculate the values from the two formulas a second time

If the claims that the absence of useful relics or records in the production process were not a sign of an absence of leadership were correct, then the values for productivity will be the same or fall, from the first three months to the second three months. If the claims that the absence of records that allow you to scrutinise the production process were a sign of an absence of leadership and low productivity, then the values for productivity will rise, from the first three months to the second three months.

This test will not only show the effect of changing from an **Event-Database Production Process** with no requirement to keep records, to one with a requirement

$$P = \frac{RfRuRtRiRmRp}{CfCuCtCi(M + Uf + Uu + Ut + Ui + Up)}$$

FIGURE 5.2 Formula for calculating productivity in a software production process.

where *W* is waste,
P is productivity,
Cf is the number of different software components used to build the final product,
Cu is the number of different components used to build its User Interface,
Ct is the number of different software components used to build the custom tools, used to build that product,
Ci is the number of different components used to build the User Interface of these tools,
Uf is the number of upgrades of components of the final product,
Uu is the number of upgrades of components of its User Interface,
Ut is the number of upgrades of components of the custom tools,
Ui is the number of upgrades of components of the User Interface of these tools.

FIGURE 5.3 First half of the parameters for calculating waste and productivity in a software production process.

to keep records. But it will also show the effect of changing from a *Software Evolution Process* to an **Event-Database Production Process**. All you have to do is use the *Software Evolution Process* for three months, stop and measure productivity and waste using the two formulas. And then use the **Event-Database Production Process** for the next three months, stop and measure productivity and waste again. The values calculated for productivity should rise, from the first three months to the second three months. If the claim that **Event-Database Production Process** improves the communication amongst the staff and hence their productivity, when compared with a *Software Evolution Process*, were true.

Up is the number of upgrades of third party tools,
Rf is the number of records kept, of the different software components of the final product,
Ru is the number of records kept, of the different components of the User Interface,
Rt is the number of records kept, of the different components of the custom tools,
Ri is the number of records kept, of the different components of the User Interface of these tools,
Rm is the number of records kept, of the modifications of all these different components,
Rp is the number of records kept, of all the upgrades performed of these components, or third party tools, and
M is the number of modifications of all the designs of these components.

FIGURE 5.4 Second half of the parameters for calculating waste and productivity in a software production process.

NOTES

1. ***Military contracts.*** Software Developers of components used in military applications have traditionally been required to follow a recognised standard, for software engineering, in order to get the contract. See Glossary.
2. ***Software engineering as a profession.*** Since software service providers do not claim to be members of a profession they cannot be legally sued for malpractice. See Glossary.
3. ***Absence of leadership.*** This is not the complete lack of leadership per se, but the management of software production through vicarious leadership or macromanagement. See Glossary.

6 Glossary

LPmud

Lars Pensjö's Multi-User Dungeon (MUD). Any of a large class of multi-user adventure games built using the software architecture created by Lars Pensjö.

LPmuds are a subclass of games called MUDs played on the Internet. The first MUD was created by Roy Trubshaw and Richard Bartle, in 1979 at Essex University in England. The first LPmud was a direct descendant of two other subclasses of MUDs. The first was AberMUD, created by Alan Cox in 1987 at Aberystwyth University in Wales. The second was TinyMUD, created by James Aspnes in 1989 at Carnegie Mellon University in the United States of America.

Having played these two MUDs, Lars Pensjö at Linköping University in Sweden decided to create his own MUD combining one major aspect from each game. The first aspect was the spirit of exploration and adventure of AberMUD. The second was the spirit of collaboration between players to create their own world, in TinyMUD. He created a programming language, called Lars Pensjö's C or LPC, to make the latter aspect possible. And this language was used to create a world of medieval fantasy and adventure. The original LPmud subsequently became known as Genesis. It was opened on the Internet around 1989, at Chalmers Computer Society in Sweden.

The computer files used to build the game were soon afterwards released on the Internet. And these were used to make hundreds of games. The licence that accompanies the files allows you to copy the game and modify it to create your own using those files. But neither the files, nor the game may be used for commercial purposes.

Nevertheless this non-commercial licence, and the requirement that the software had to be easy to modify, by those who received a copy of the game, to create their own game, gave LPmud a distinct software architecture. This architecture defined the relationship between the Software Developer and user, the different components of the software, and how these components were to be modified, over time, to meet any changes in the game design, while maintaining this relationship. And others have subsequently reimplemented the software architecture of Lars Pensjö, with different computer files and different software licences.

> *I played some AberMUD (to wizardhood once), and a little tinymud. I liked the idea of a multi-user anonymous game very much, but found that Abermud was too difficult to extend while Tinymud had too little emphasize on adventure. The social part was nice, however.*
>
> *So, I draw some guidelines about how to create a system which would be much more simple to extend, …*
>
> *I presented these ideas to some friends, who was going to create a MUD, or possibly take Abermud and add things. These friends did not believe in my ideas, so I spent a week to create a skeleton (small LPC interpreter) …*

 DOI: 10.1201/9781003502807-6

I added an internal editor…to make it possible for players (wizards) to add objects. This was done because I did not have enough fantasy to create a good world myself, so I though maybe others could do it for me. I made the requirement that players had to achieve a certain level, so as to make it a challange. This idea of letting wizards extend the game was not in the original plans. The original plan was really to make a language that should be easy to extend dynamically.

My MUD-interested friends of course did have to try it out now, and they could not stop until they reached wizardhood. At that stage, they had to try creating their own objects and adventures….

Source: What is Genesis? © 1999, Chalmers Computer Society. Lars Pensjö

QUALITY CONTROL

A system that accepts or rejects products or services depending on whether these meet all of the customer's specifications and requirements.

SOFTWARE ARCHITECTURE

A description of a system for producing software. It includes a description of the components of the system, the relationship between these components, and the principles that govern how these components change. The components may be as large as a software library, or as small as a single software module. The components can also vary from any software documentation to any software tool required by the system. A software architecture can serve as a basis for a software design, or (since all the components do not have to be software components) a software production process.

The software architecture of a program or computing system is the structure or structures of the system, which comprise software components, externally visible properties of those components, and the relationships among them.

…the architecture embodies information about how the components interact with each other. This means the architecture specifically omits content information about components that does not pertain to their interaction.

…the definition does not specify what architectural components and relationships are. Is a software component an object? A process? A library? A Database? A commercial product? It can be any of these things and more.

…the behaviour of each component is part of the architecture, insofar as that behaviour can be observed or discerned from the point of view of another component. This behaviour is what allows components to interact with each other, which is clearly part of the architecture. Hence, most of the box-and-line drawings passed off as architecture are in fact not architectures at all. They are simply box-and-line drawings.

Source: Software Architecture in Practice © 1997, Addison-Wesley. Bass, Clements and Kazman

The structure of the components of a program/system, their interrelationships, and principles and guidelines governing their design and evolution over time.

Source: IEEE Transactions on Software Engineering © 1995, Institute of Electrical and Electronics Engineers. David Garlan and Dewayne Perry

THE SOFTWARE PRODUCTION PROCESS

The steps for designing and implementing a game, using the Event-Database Architecture. See the chapter entitled

The Software Production Process

in the book

Event-Database Architecture for Computer Games: Volume 1, Software Architecture and the Software Production Process.

GAME WORLD

An imaginary world space in which a game takes place.

QUALITY

The characteristic of a product which meets a customer's needs.

> *The totality of features and characteristics of a product or service that bear on its ability to satisfy stated or implied needs. Not to be mistaken for 'degree of excellence' or 'fitness for use' which meet only part of the definition.*
> *[ISO8402].*
>
> *Source: Quality © 1993–2001, The Free On-line Dictionary of Computing. Denis Howe*

LPC

Lars Pensjö's C. A programming language, modelled on the language 'C', designed to allow you to modify the behaviour of items in multi-user adventure games.

ARTIFICIAL INTELLIGENCE

An attempt to model the human brain or to create a system that can make deductions. That is to say, given two facts, can a computer determine whether a third fact is true? All attempts to create such a system have failed.

TECHNICAL DESIGN SOURCES

Game Programming Gems by Mark De Laura.

INCOMPLETE GAME DESIGN

A design for a game which consistent of just enough highlights to sell a project to its financial backer, but not enough detail to implement it.

An Incomplete Game Design

in the book

Event-Database Architecture for Computer Games: Volume 1, Software Architecture and the Software Production Process.

UML

Unified Modelling Language. A language for describing the software components of a computer system. The software components are constructed in a hierarchy. This can be a hierarchy of inheritance or a hierarchy of reference. The relationship between them is often described in the form of a box-and-line diagrams. Many people confuse these diagrams with a software architecture. Even though it is not. See the definition for software architecture.

FAILSAFE

A computer software or hardware system that can continue operating despite the persistence of errors within it.

ABSTRACTION

The simplification of a problem by concentrating on essential aspects and ignoring the rest.

COLLISION BOUNDARY

The area around a Game Object which would be used to determine its collision with other Objects.

PROXIMITY BOUNDARY

The area around a Game Object which would be used to determine when another Object was, or was not, in close proximity.

FIELD OF VIEW

The visible area in front of a camera.

NEAR AND FAR FOCAL LENGTH

The closest and furthest distance of the visible area in front of a camera.

WAYPOINT

A point along a path.

ILLUSION OF INTELLIGENCE SOURCES

Programming Game artificial intelligence (AI) by Example by Mat Buckland.

ARTIFICIAL NEURAL NETWORK

Computer software used to make intelligent decisions, whose design was inspired by the study of animal brains. It is made up of a network of very simple software processors, connected by one-way communication channels.

NEURON

A nerve cell adapted to conducting electrical impulses, in the human brain.

ARTIFICIAL NEURON

A mathematical model of a biological neuron. Each has multiple numerical parameters or inputs and a single output. It has a mathematical formula called an Activation Function which takes the sum of the inputs and produces a single output. In theory, the inputs represent information from human sensors such as taste, sight, hearing, smell and touch. In practice, the inputs are metrics or measurements gathered from a real or imaginary space, by humans or computers, which represent an aggregation of information from human sensors.

For example instead of being fed an image of two objects, and letting the Artificial Neuron infer which one was closer, the relative distance of the two objects would be measured and fed into the Artificial Neuron. The output of the Artificial Neuron represents a human response to the inputs. For example, the decision to move towards the closer of two objects or to pick up the closer of two objects.

It may take a network of several Artificial Neurons in layers to process the inputs and produce the final outputs. In these cases the output of an Artificial Neurons in the intermediate layers is just a partial result of the final output. And the inputs in those layers are just a derivative of the initial inputs.

EXPENSIVE GRAPHICS PROCESSORS

Graphics Processors are made up of several specialised maths processors running in parallel. These were originally used to perform the calculations required to render 3D graphics to achieve Photorealism. But lately these have also proved ideal for performing the parallel calculations in Artificial Neurons, and propagating the results forwards and backwards through the layers of an Artificial Neural Network. The demand for Photorealism and Artificial Neural Networks has increased the price of these Processors to ridiculous levels. And resulted in a huge waste of resources on buying, powering, cooling and developing software for these Graphic Processors.

For Photorealism, Graphics Processors are used to speed up the process of producing realistic graphical effects, in many industries including Architecture, Engineering, Construction, Film, Computer Games, and Automotive.

For Artificial Neural Networks, Graphics Processors are used to speed up the process of Forward Propagation and Back Propagation in the network. Nvidia has built

one such Graphics Processor called H100 GPU that cost 40,000 dollars each. Many large companies have bought billions of dollars worth of these GPUs.

In the Computer Games industry, although you will find some Corporate Media articles, which cover the industry, mention the shortage of Graphics Processors and the subsequent high cost. And you will find other articles which mention the demand for these Graphics Processors and link this with a desire for Photorealism. You will not, however, find articles which connect the desire for Photorealism, with the demand for these Graphic Processors, the subsequent shortage of these Processors, and the high price. Since the Media itself manufactures demand by propagating the marketing material for these Processors. Suggesting these are either innovative or ideal for achieving Photorealism or the next best thing. The price of the one of the highest performing Graphic Processors, in terms of Photorealism, you can get today, to make and play games, is the Nvidia RTX 4090 24GB. This costs between 1500 dollars and 2000 dollars. That single component alone is now worth more than the cost of a whole computer system or personal computer (PC), from ten years ago and many computer systems from today.

> ***Nvidia*** *board partners are reportedly looking to increase the prices of various GeForce RTX products in China. These prices reflect up to a 10% increase in the cost of Nvidia products over American prices and US markets, and they are expected to hit consumers soon…*
>
> *Recent innovations from Nvidia have helped usher in some of video gaming's most popular visual features like ray tracing, which has become increasingly standard in modern games. As time goes on, more powerful tools are created, allowing developers to create even more stunning and impressive games. However, as technology advances, so does the cost and price of these products, with Nvidia's partners expected to increase price hikes.*
>
> *Source: Nvidia Partners Are Raising Their Prices © 2024. Game Rant. Luke Dammann*

> *Updated October 22, 2024, by Hamza Haq: The graphical fidelity boost Nvidia's RTX technology provides is no secret, elevating in-game visuals from okay-ish to downright gorgeous. Despite its popularity, few games truly utilize the tool to its fullest potential. While nearly all games that have come out in the last few years have the option to turn on ray tracing in their graphics settings, the difference is not always noticeable. Whether due to low-budget or poor implementation, games that truly showcase what ray tracing can do are quite uncommon. Luckily, with more and more developers learning how to incorporate this game-changing technology into their games, that may not remain the case in the future. For now, though, here are some of the best games to test out Nvidia RTX and its capabilities….*
>
> *Source: Nvidia RTX: Top 27 Games That Utilize Ray Tracing The Best © 2024. Game Rant. Ashish Walia*

> *Is RTX 4090 Worth It?*
>
> *Yes, the GeForce RTX 4090 provides good value for money within its price range, of around £2068, when bought new. It surpasses the benchmark performance of GeForce RTX 3090, the closest competitor within this price bracket, by a margin of 80.7%.*
>
> …
>
> **PSU for RTX 4090**

The GeForce RTX 4090 requires 450 watts to operate. However, when selecting a power supply, it's crucial to account for the power consumption of the entire system. Therefore, add together the TDP of your GPU and CPU, then multiply the sum by 2. For instance, if you pair GeForce RTX 4090 with Intel Core i9-13900K, which has a TDP of 125W, you should aim for a power supply around 1200 watts. This approach ensures a substantial margin, allowing your PSU to operate coolly and efficiently.

Another reason to opt for a more potent PSU than your system's exact power requirement is due to GPU transient power spikes, which occur when the GPU is under heavy load. These spikes may cause the PC to shut down if the PSU lacks sufficient overhead…'.

Source: RTX 4090 Price History UK © 2024. Best Value GPU

The H100 is estimated to cost between $20,000 and $40,000 meaning that Meta used up to $640 million worth of hardware to train the model. And that's just a small slice of the Nvidia hardware Meta has been stockpiling. Earlier this year, Meta said that it was aiming to have a stash of 350,000 H100s in its AI training infrastructure – which adds up to over $10 billion worth of the specialized Nvidia chips.

Who is hoarding Nvidia H100 GPUs?

Nvidia's H100 GPU is one of the most in-demand technologies in the AI arms race. Companies are stockpiling the $40,000 chips to train more powerful AI models.

Company	*Number of H100 GPUs*	*Approx. Value*
Meta	*350,000*	*$7–$14 billion*
xAI	*100,000*	*$2–$4 billion*
Tesla	*85,000*	*$1.7–$3.4 billion*
*Andreessen Horowitz***	*20,000*	*$400–$800 million*

**Numbers are either stated totals in use, or year end targets.*
*Source: The companies; **The Information*

Venture capital firm Andreesen Horowitz is reportedly hoarding more than 20,000 of the pricey GPUs, which it is renting out to AI startups in exchange for equity, according to The Information.

Tesla has also been collecting H100s. Musk said on an earnings call in April that Tesla wants to have between 35,000 and 85,000 H100s by the end of the year.

But Musk also needs H100s for X and his AI company xAI. This week, Musk boasted on X that xAI's company's training cluster is made up of 100,000 H100s.

Source: Just four companies are hoarding tens of billions of dollars worth of Nvidia GPU chips © 2024. Sherwood. Jon Keegan.

EXPENSIVE ERRONEOUS LANGUAGE LEARNING MODELS

Expensive large Language Learning Models, such as ChatGPT Web Server, still produce unforeseen, unfeasible or prohibited results from time to time. And they still cannot solve basic mathematical and logical problems.

Researchers into AI have found that many of the models used to create Artificial Neural Networks, such as Language Learning Models, are not new. These are just a form of old Pattern Recognition Algorithms. That leads to the following conclusions:

1. Pattern Recognition Algorithms have been misnamed AI Algorithms or Artificial Neural Network Algorithms.
2. These algorithms basically learn how to recognise patterns and generate patterns, not how to perform deductions, logic or reasoning, that meet the definition of AI.
3. A good example of the application of these algorithms is recognising the shape of characters in one font, and recognising the same characters in another font.
4. A bad example of the application of these algorithms is anything that involves large amounts of training data or real world data e.g. real-world management problems or attacking enemy Surface-To-Air SAM sites.
5. During training, the algorithm requires a system or function to reward or penalise the algorithm for making good or bad choices.
6. After training, the algorithm can come up with unforeseen, unfeasible or prohibited results if the system or function rewarding or penalising it was misconceived or too limited.
7. Language Learning Models have a predecessor in the form of Artificial Linguistic Internet Computer Entity (Alice) developed by Joseph Weizenbaum at MIT in the early 1960s that failed to meet the definition of AI.
8. Language Learning Models have a predecessor in the form of ELIZA from the 1980s, named after Eliza Doolittle, a working-class character in George Bernard Shaw's *Pygmalion*, that again failed to meet the definition of AI.
9. Language Learning Models raise ethical questions about whether you should or should not include language from prohibited material in the training data, which has never been answered.
10. Language Learning Models raise ethical questions about whether you should or should not include bias sources in the training data e.g. the New York Times, which has never been answered.
11. Language Learning Models process symbols and patterns but have no understanding what those symbols and patterns represent.
12. Language Learning Models cannot solve mathematical or logical problems, not even basic ones.
13. To overcome these limitations, Language Learning Models need a fundamental change to the paradigm they use.

He notes that one simulated test saw an AI-enabled drone tasked with a SEAD mission to identify and destroy SAM sites, with the final go/no go given by the human. However, having been 'reinforced' in training that destruction of the SAM was the preferred option, the AI then decided that 'no-go' decisions from the human were interfering with its higher mission – killing SAMs – and then attacked the operator in the simulation. Said Hamilton: 'We were training it in simulation to identify and target a SAM threat. And then the operator would say yes, kill that threat. The system

started realising that while they did identify the threat at times the human operator would tell it not to kill that threat, but it got its points by killing that threat. So what did it do? It killed the operator. It killed the operator because that person was keeping it from accomplishing its objective.'

He went on: 'We trained the system – "Hey don't kill the operator – that's bad. You're gonna lose points if you do that". So what does it start doing? It starts destroying the communication tower that the operator uses to communicate with the drone to stop it from killing the target.'

Source: Highlights from the RAeS Future Combat Air & Space Capabilities Summit: AI – is Skynet here already? © 2023. Royal Aeronautical Society. Tim Robinson and Stephen Bridgewater

Currently there is some hype about a family of large language models like ChatGPT. The program reads natural language input and processes it into some related natural language content output. That is not new. The first Artificial Linguistic Internet Computer Entity (Alice) was developed by Joseph Weizenbaum at MIT in the early 1960s. I had funny chats with ELIZA in the 1980s on a mainframe terminal. ChatGPT is a bit niftier and its iterative results, i.e. the 'conversations' it creates, may well astonish some people. But the hype around it is unwarranted.

Behind those language models are machine learning algos that have been trained by large amounts of human speech sucked from the internet. They were trained with speech patterns to then generate speech patterns. The learning part is problem number one. The material these models have been trained with is inherently biased. Did the human trainers who selected the training data include user comments lifted from pornographic sites or did they exclude those? Ethics may have argued for excluding them. But if the model is supposed to give real world results the data from porn sites must be included. How does one prevent remnants from such comments from sneaking into a conversations with kids that the model may later generate? There is a myriad of such problems. Does one include New York Times pieces in the training set even though one knows that they are highly biased? Will a model be allowed to produce hateful output? What is hateful? Who decides? How is that reflected in its reward function?

Currently the factual correctness of the output of the best large language models is an estimated 80%. They process symbols and pattern but have no understanding of what those symbols or pattern represent. They can not solve mathematical and logical problems, not even very basic ones.

Source: 'Artificial Intelligence' Is (Mostly) Glorified Pattern Recognition © 2024. Moon of Alabama. 'Bernhard Billmon'

NEURAL NETWORK SOURCES

Introduction to Neural Networks by Kevin Gurney.

FLAWS IN PHOTOREALISM

There are many flaws in the Photorealism of computer games which the Software Developers have to strive continuously to overcome. Despite advances in the Graphics Processors, and the Hardware Rendering processes these perform, there is a never

ending cycle of the development of more and more demanding rendering algorithms, each year, which require more and more resources.

Each year, they claim to have finally achieved Photorealism. Only for the following year to make claims that they have produced games with greater Photorealism. That show that the claims of the preceding year were false. And there were flaws in the Photorealism which they could see but would not admit to at the time. Therefore the images of their games in the preceding year could not have been Photorealistic. And that in turn means that Photorealism is either a pipe dream, a fantasy, a false hope, an unattainable dream. Or there is a fundamental flaw in the methods they try to use to achieve Photorealism.

Nevertheless, despite these flaws, Photorealism is one of the main selling points in the marketing material of commercial game-engines such as the Unreal Engine and the Unity Engine.

'UNREAL ENGINE 5 – WHAT IT'S ALL ABOUT

…

UNREAL ENGINE 5 – EXPECTATIONS

Understandably, there are high expectations for Unreal's newest launch. Last year, an article from Perforce said UE5 would change the industry because '…it will enable truly immersive experiences – while reducing the complexity of building games, as well as in film and animation.'

It's not just developers who are excited about what next-gen graphics can bring. Some recent studies reveal that upwards of 75% of gamers make purchases based on graphics quality.

…

DID UNREAL ENGINE 5 (EARLY ACCESS VERSION) LIVE UP TO EXPECTATIONS?

When UE5 was first announced, Epic made it clear what the main goal was: '[to] achieve photorealism on par with movie, CG, and real life,' all while keeping these tools accessible to teams in the industry.

This is a huge promise. They didn't say it was meant to look 'good;' they claimed to keep up with photorealism in every industry. So the question is: Did they live up to it?

…

MetaHumans

The announcement and early access of MetaHuman Creator resulted in whispers throughout the industry of what impact this amazing software could have on game development moving forward. Shortly after opening MHC, you'll notice just how easy it is to create photorealistic characters, customized to your needs….

Source: Unreal Engine 5 – What It's All About? © 2021. Incredibuild.
Joseph Sibony

Unity and Unreal Engine are two of the most prominent game engines in the industry, known for their cutting-edge capabilities in rendering photorealistic graphics. Both engines have been extensively used in the development of AAA games, architectural visualizations, and various other applications that demand high-fidelity visuals. In

this exploration, we will delve into the strengths and distinguishing features of each engine when it comes to achieving photorealistic graphics.

UNITY

… However, in recent years, Unity has made significant strides in enhancing its graphics capabilities, cementing its position as a powerful engine for photorealistic rendering.

1. ***High-Definition Render Pipeline (HDRP)**: Unity's HDRP is a state-of-the-art rendering pipeline designed specifically for high-fidelity graphics. It supports advanced features such as real-time global illumination, physically based rendering (PBR), and high-dynamic-range (HDR) lighting, enabling developers to create highly realistic and visually stunning environments.*
2. ***Scriptable Render Pipeline**: Unity's Scriptable Render Pipeline (SRP) allows developers to customize and extend the rendering process …. This flexibility enables advanced techniques for achieving photorealistic results tailored to specific project requirements.*
3. ***Real-Time Ray Tracing**: With the introduction of real-time ray tracing support, Unity has opened the door to accurate simulations of light behavior, enabling realistic reflections, shadows and global illumination effects ….*
4. ***Asset Importers and Optimization**: Unity's robust asset import pipeline and optimization tools … ensuring efficient rendering and performance optimization for photorealistic graphics.*
5. ***Integration with Industry-Standard Tools**: Unity seamlessly integrates with industry-standard tools such as Autodesk Maya, 3ds Max, and Substance Painter, allowing artists and developers to leverage their existing workflows and pipelines for creating photorealistic content.*

…

UNREAL ENGINE

…

6. ***Chaos Physics and Destruction:** Unreal Engine's Chaos physics and destruction systems enable realistic simulations of rigid body dynamics, soft body deformations, and large-scale destruction events, adding to the overall level of realism and immersion.*

Source: Unity vs Unreal: Exploring Cutting Edge of Photorealistic Graphics © 2024. Oodles Technologies

RENDERING FARMS

A cluster or network of computers across which a software rendering process is distributed to produce Photorealistic images. Typically for film or TV industries. Any company that offers these computers as a service is also called a Rendering Farm.

MATERIALS

Formulae for controlling how the surface of a polygon is rendered in graphics, or is simulated in physics.

DATA DESIGN

A description of all the data needed by a game. It is also a description of all the data produced by the tools used to build a game.

RGBA

A data format for describing the colour of a pixel by four values, for its Red, Green, Blue and a special Alpha component. The last of these controls how it blends with the colour of any underlying image.

GRAPHIC SHADERS

Machine code, that is executed during the Hardware Rendering process of a Graphics Processor, that controls how a surface or vertices of a polygon is rendered on the computer screen or in a Texture.

VERTEX SHADER

A Graphic Shader that is used to perform the projection of the vertices of the polygons of 2D images or 3D models, through a camera, into Normalised space (an area which is $1 \times 1 \times 1$) and then onto screen space (i.e. the computer screen). And it is used to set the amount of lighting at each vertex.

GEOMETRY SHADER

A Graphic Shader that is optional. It is used either to take the 2D or 3D primitives from the Vertex Shader and produce another primitive, adding or removing vertices. Or for rendering multiple images of the same primitive, at once, to the same target (i.e. computer screen or Texture). Or for feeding back information about the vertices of the primitives produced by the Vertex Shader, to later steps.

FRAGMENT SHADER

A Graphic Shader that is optional. It parses the pixels of the Textures of the polygons of 2D images or 3D models, after Rasterisation (i.e. the projection of the pixels on to the screen). And it can change the depth and colour of the pixels depending on some kind of formula. And it can also discard pixels and stop these being rendered dependent on another formula.

ENTITY-RELATIONSHIP DIAGRAM

A diagram which shows all the items (or entities) stored in a Relational Database, and the relationship between these items.

TEST TOOLS

The set of tools that could be used to build a minimal software design, based on the Event-Database Architecture, and test it. See the subchapters entitled Documentation tools to Sound tools in the book, Event-Database Architecture for Computer Games, Volume 1: Software Architecture and Software Production Process.

GUIDELINES FOR BUILDING INTERFACES

Strive for consistency is the first principle in designing User Interfaces, as outlined by Ben Shneiderman in his guide, Designing the User Interface.

1. *Strive for consistency.*
   This principle is the most frequently violated one, and yet the easiest one to repair and avoid. Consistent sequences of actions should be required in similar situations, identical terminology should be used in prompts, menus, and help screens, and consistent commands should be employed throughout. Exceptions, such as nonprinting of passwords or no abbreviation of the DELETE command, should be comprehensible and limited in number.

Source: Designing the User Interface © 1987, Addison-Wesley.
Ben Shneiderman

CLOSED DATA FORMAT

The secret description of the layout of data in a Database, and how each data is used. This description is proprietary and only known to a very limited number of software applications.

INFORMATION HIDING

A technique commonly used in Object-Oriented Design of software, to protect one software module (or Object) from being erroneously accessed by another.

NO REDUNDANCY (IN A DATABASE)

The process of removing duplication of information in a Database is called Normalization.

LOGIC FLAG

Data which is either set or cleared, when a condition that a software procedure uses to control its behaviour, changes e.g. when a task it is waiting for is complete.

STATE SWITCH

Data which controls the way software behaves. It usually controls only one software module. It ensures that two modes of operation do not overlap. Or, it ensures that the modes follow each other in the correct sequence.

SOFTWARE CODE

The list of programming language instructions that describe the procedures a computer must follow.

DATA OFFSET

The index of a subset of data, within a Database. This would be in the form of a number, which represented the distance of that data, from some reference point, normally the start of that Database. This could be used to quickly search its contents, or define its layout.

DATABASE DESIGN SOURCES

Handbook of Relational Database Design by Candace C. Fleming and Barbara Von Halle.

FINITE STATE MACHINE

A method for designing a computer system based on two basic concepts: that the system has a well-defined set of states, and that there exists a well-defined set of events connecting any two states.

DESKTOP COMPUTER

An International Business Machines (IBM) PC or compatible model. It was designed for business but is now popular as a home computer too.

SCALABLE

A software which can vary its performance depending on the resources it has available. And thus it can be used on a range of computers, with different speeds, sizes of memory and other levels of resources. Scalable components. A software procedure or data that can vary the time and space that it uses.

HOME COMPUTER

A computer system designed for home use e.g. Playing games, music, learning or small business software.

GAMES INDUSTRY COMMENTATORS

Some Software Developers keep an up-to-date version of their computer games on desktop computers, even though they never release this version.

SMALL DEVICES (WITH RELATIONAL DATABASES)

Relational Databases have been used with software deployed on mobile phones. One of the companies that has developed such technology has been Birdstep Technology. Their technology is known as 'RDMm'.

ORDERED SOFTWARE SYSTEM

A system of software components that has been assembled according to some principles. And therefore can be progressively disassembled, using the same principles, without causing errors when the software is rebuilt.

SO … THERE ARE NO OBVIOUS DEFICIENCIES

Quotation by C. A. R. Hoare, a computer-scientist best known for his discovery of a widely used procedure for quickly sorting items of data. He later became a Professor of Computing at Oxford University, in the UK.

> There are two ways of constructing a software design: One way is to make it so simple that there are obviously no deficiencies, and the other way is to make it so complicated that there are no obvious deficiencies. The first method is far more difficult.
>
> – C.A.R. Hoare

Source: Adages on Software Design and Development © 1997. Stephen Block. Adelphi University

FORWARD ENGINEERING

The process of building a software product (or any manufactured product) in four phases: analysis, design, implementation and testing.

REVERSE ENGINEERING

The process of rebuilding a software product (or any manufactured product) in four phases: re-testing, re-implementation, re-design and re-analysis.

The re-testing phase involves diagnosing the external characteristics of the product and inferring the low-level tools used to build it from the diagnosis.

The re-implementation phase involves rebuilding these low level tools.

The re-design phase involves inferring the high-level designs from the low-level tools.

The re-analysis phase involves using these high-level designs to either re-design the original product in another form as a competing product. Or to re-design the original product without some flaw or error when you do not have access to its original design. Or to design a new product to interoperate with the original product.

SOFTWARE ENGINEERING

A systematic, disciplined approach to software production. It was devised to cope with large projects which no one individual could undertake to deliver in a timely, secure fashion.

This approach may begin with a prototype. A prototype is the first product of the software production process. All other products of that process have the same qualities. So the prototype can be used to assess the feasibility of the process.

> *Software Engineering: (1) The application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software; that is, the application of engineering to software. (2) The study of approaches in (1).*
>
> *Source: IEEE Standards Collection: Software Engineering, IEEE Standard 610.12-1990 © 1993, Institute of Electrical and Electronics Engineers*

> *Software engineering is the establishment and use of sound engineering principles in order to obtain economically software that is reliable and works efficiently on real machines.*
>
> *Source: Software Engineering: A Report on a Conference Sponsored by the NATO Science Committee © 1969, North Atlantic Treaty Organisation. Naur, P. and B. Randall (eds.)*

HIGH TURNOVER OF STAFF

Very few of the staff of the Software Developers, in the Computer Games industry, stay there or in the industry as a whole, for more than a few years.

DESIGN PRINCIPLES

A pre-emptive statement at the beginning of a design document that sets the rules for providing a solution to a problem.

A design principle is not a solution to a problem. It is a tool for finding a solution to a problem.

A design principle can be a general-purpose tool e.g. a game editor. That will be used to design a Game World (see the definition of **game editor** in the Glossary).

A design principle can be a benchmark that gives optimum performance for a microprocessor chip in some computer hardware used to play games. That will be used to select or reject software designs which best fit the design of that benchmark (see the definition of **benchmark** in the Glossary).

A design principle can be a design pattern. That will be used to write the code for the game (see the definition of **design patterns** in the Glossary).

A design principle can be combination of a game editor, a benchmark or a design pattern.

This should not be confused with the principles of a software architecture. The motivation behind design principles and the motivation behind the principles of a software architecture are different.

The motivation behind design principles is the need to find a solution to a problem which is not well understood, as quickly as possible. This quick solution or software design is probably going to change from the beginning to the end of production. And these changes will not be documented. So the principles assume that Reverse engineering will be required to understand this software design.

The motivation behind the principles of a software architecture is the need to maintain the relationships between the components of that architecture, from the beginning to the end of production. Each component may be a component of a software design, or a component of a production process.

This difference between the motivation of the design principles and the principles of a software architecture leads to three other differences.

Firstly, the principles of a software architecture relate to the components of that architecture. And you can identify all of these components, the form and reason for the relationship between the components, and the role each one plays in that architecture from its principles.

However, design principles relate to tools or methods that facilitate Reverse engineering. Some of these tools or methods may be used to build the components of a software design. But others may not. The benefits of these tools or methods lie in the ability of these to facilitate Reverse engineering. So, for example, the tools would not include one that allowed you to build a component but did not allow you to diagnose it. Furthermore, you could not identify what any of the components were going to be, or the role of each one in the software design, from these design principles. Therefore, although you may be able to infer, from design principles, the form of the relationships between these components, you could not identify the function of these relationships.

The second difference is that the principles of a software architecture govern how the components of that architecture would change, when the requirements of the software changed. In contrast, design principles do not explain how changes would be performed in a production process.

Finally, the principles of a software architecture stem from a context. This context is whatever problem the software architecture was meant to solve. The principles of the architecture are chosen after examining that problem and do not presume to apply outside of that context. Design principles, on the other hand, have no context. These principles proceed an examination of any problem, and the design of a solution. The principles are chosen by default, based on popular methods or tools, and assume these may be applied in any context.

DESIGN PATTERNS

A design pattern is a general description of a solution to a common design problem. In software production, design patterns usually refer to solutions which have been built using particular programming languages. Namely, those that support a technique known as ‘Object-Oriented Design’. Within this technique, software modules are known as ‘Objects’. An ‘Object’ can inherit the properties

of other 'Objects'. In which case the former is called the child and the latter is called the parent. The properties of an 'Object' are controlled by the set of software procedures or data that give access to its services called its Interface. Examples of programming languages which support this technique are C++, C#, Java, and Smalltalk.

The seven most popular design patterns are Singleton, Factory, Facade, Strategy, Observer, Builder and Adapter.

Singleton Design Pattern

A Singleton Design Pattern involves the principle of creating one and only one 'Object' to control open or global access to some service.

Factory Design Pattern

A Factory Design Pattern involves the principle of having a parent 'Object' which has many children derived from it called 'Creators'. And the parent delegates the creation of other final concrete usable 'Objects' or 'Products' to the 'Creators'.

Facade Design Pattern

A Facade Design Pattern involves the principle of creating a single 'Object' to provide a single unifying Interface to a group of multiple disparate Interfaces of a group of other 'Objects', which together perform a single complex task.

Strategy Design Pattern

A Strategy Design Pattern involves the principle of having a parent 'Object' which has many children derived from it. That executes a family of algorithms which accomplish similar tasks e.g. sorting products being sold on a Web Page by colour, size or price. And having your program choose which algorithm to execute depending on the User's choices.

Observer Design Pattern

An Observer Design Pattern involves having one 'Object' called a 'Notifier' which other 'Objects' called 'Observers' can subscribe or unsubscribe to. To be notified when an important event happens or something important changes.

Builder Design Pattern

A Builder Design Pattern is similar to a Factory Design Pattern. But instead of the 'Creators' you have 'Objects' called 'Builders'. And instead of building the 'Products' in one step, the 'Builders' build 'Products' in multiple steps. And you have an additional 'Object' called a 'Director' which controls the execution of these steps and releases the final 'Products'.

Adapter Design Pattern

An Adapter Design Pattern involves the principle of ensuring that two 'Objects' with dissimilar Interfaces have the same Interface for the sake of consistency and compatibility. By creating a new 'Object' called a 'Wrapper', which gives access to one or both 'Objects' through the same Interface.

Each solution, described by a design pattern, represents the combined wisdom of engineers who have already addressed that problem in the past, using these programming languages. And the pattern supposedly helps future engineers avoid repeating their mistakes, by choosing partial or inelegant solutions, in their designs. Many people in the Computer Games industry conflate design patterns with software architectures, like the Event-Database Architecture. But the design patterns are much low-level tools than software architectures.

Firstly, design patterns are only marginally more abstract than the instructions of a programming language. A design pattern could describe the set of software modules for building a solution. And it could describe what the general relationship should be between those modules. But it could not describe what practical role each module would play in the overall software design at highest level. Nor could it describe any solution without referring to examples, written in one of the aforementioned programming languages. Thus, a design pattern is dependent on programming languages, but independent of practical applications. A design pattern could be used for anything; from building computer games to word processors.

Secondly, the basis of design patterns, like all design principles, is nothing more than the infallibility of popular wisdom. These patterns assume that future generations of engineers could not possibly come up with a better design than their predecessors. The mistakes that later generations would make, by choosing partial or inelegant solutions, would not lead them to find even better solutions. So they should consider fitting any problems they encounter around the design patterns (i.e. the tools or programming languages) of their predecessors.

But this assumption is false. Future generations could conceive of better solutions. And the close association of design principles, with programming languages (especially C++), is another huge flaw. A technique which nominally associates itself with high-level abstract designs should not be closely associated with low-level programming languages or tools. These two flaws open design patterns to abuse. And this has been perfectly illustrated in the Computer Games industry, where design patterns have been regularly abused.

For example, design patterns have been used to select or reject applicants wishing to join a company. Applicants who did not show knowledge of the arbitrary set of design patterns, described in one of the popular books that covered the topic, have been rejected. Whereas those who did show such knowledge have been accepted. And by doing so, their interviewers have implicitly set out the design patterns which they expect the staff to abide by.

Another example of the abuse of design patterns has involved Reverse engineering. Design patterns have been used as a means of making the computer files, written by the Game Programmers, more 'readable'. The documentation of the design of their software modules has been neglected. Instead, the other Programmers have

been expected to use their ability to recognise design patterns, to infer the design of these software modules. In other words, they have been expected to use design patterns for Reverse engineering. And this has been euphemistically been known as 'improving code readability'.

Thirdly, the principles of a software architecture and the principles of a design pattern are not the same. Take example, the rule for generating a system of Events in the Event-Database Architecture, and a rule for generating a system of events with design patterns.

In the past, the Software Developers of some Computer Games have chosen a system of Events to use based on the Observer Design Pattern. In that design pattern, you have one software module or 'Object' called a 'Notifier' which other 'Objects' called 'Observers' can subscribe or unsubscribe to. To be notified when an important event happens or something important changes. And you can have any number of 'Notifiers' and any number of 'Observers' in a game.

As such the system of events generated by design patterns has had four basic differences from the system of Events generated by the Event-Database Architecture.

Firstly, all the 'Objects' used to make a Computer Game cannot be identified in the system of events generated by the Observer Design Pattern. Since typically Computer Games which employ design patterns do not use only one design pattern. You can identify 'Objects' which are 'Notifiers' and those which are 'Observers'. But you cannot identify 'Objects' which are neither 'Notifiers' nor 'Observers', and which are part of other design patterns. For example, you cannot identify 'Objects' which only share information between other 'Objects', like the current state of quests available in the game. Typically these 'Objects' would be created with another design pattern like the Singleton Design Pattern.

Secondly, the relationships between all the 'Objects' also cannot be identified from the Observer Design Pattern. 'Observers' can dynamically subscribe and unsubscribe to respond to events broadcast by 'Notifiers'. And you cannot identify the role which any 'Notifier' or 'Observer' plays in the overall design.

Thirdly, the system of events of the Observer Design Pattern are presumed to be applicable in any context.

And finally, the system of events generated by the Observer Design Pattern cannot be used as a device for defining a language for all the staff involved in software production process to communicate. Instead, it has only been visible, and limited, to the communication of those staff who appreciate its value for Reverse engineering, namely the Game Programmers.

Contrast this with the system of Events generated by the Event-Database Architecture.

Firstly, you can identify all the 'Objects' in the system because every 'Object' has an Event it responds to and an entry in the Game Database.

Secondly, you can identify the relationship between all 'Objects' have Secondary Events which they respond to, and these Events have entries in the Database. Even if that Event is rare or temporary. And you can identify the role which each 'Object' plays in the overall design from its Events and the documentation maintained about these Events in the data design by the Database Administrator.

Thirdly, the system of Events applies only to the context of the Event-Database Architecture. This is building Computer Games.

Fourthly, the system of Events facilitates the communication through natural language of all the staff. Since all the Events are visible to all the staff including Game Producers, Game Artists, Game Programmers, Game Designers, Game Testers, Sound Engineers and Database Administrators, through a shared Database. And any of them can add new Events, edit Events and create a chain of Events through that Database.

Design patterns can speed up the development process by providing tested, proven development paradigms. Effective software design requires considering issues that may not become visible until later in the implementation. Reusing design patterns helps to prevent subtle issues that can cause major problems and improves code readability for coders and architects familiar with the patterns…

…Design patterns provide general solutions, documented in a format that doesn't require specifics tied to a particular problem.

Patterns allow developers to communicate using well-known, well understood names for software interactions….

Some feel that the need for patterns results from using computer languages or techniques with insufficient abstraction ability. Under ideal factoring, a concept should not be copied, but merely referenced. But if something is referenced instead of copied, then there is no "pattern" to label and catalog. It is also said that design patterns encourage navigational database-like structures instead of the allegedly cleaner relational approach where such structures are viewpoints instead of hard-wired into programming code….

Source: Design Patterns. Wikipedia. The Free Encyclopaedia. 2005

CREATIONAL

Main article: Creational pattern

Creational patterns are ones that create objects, rather than having to instantiate objects directly. This gives the program more flexibility in deciding which objects need to be created for a given case.

- *Abstract factory groups object factories that have a common theme.*
- *Builder constructs complex objects by separating construction and representation.*
- *Factory method creates objects without specifying the exact class to create.*
- *Prototype creates objects by cloning an existing object.*
- *Singleton restricts object creation for a class to only one instance.*

STRUCTURAL

Structural patterns concern class and object composition. They use inheritance to compose interfaces and define ways to compose objects to obtain new functionality.

- *Adapter allows classes with incompatible interfaces to work together by wrapping its own interface around that of an already existing class.*
- *Bridge decouples an abstraction from its implementation so that the two can vary independently.*
- *Composite composes zero-or-more similar objects so that they can be manipulated as one object.*

- *Decorator dynamically adds/overrides behaviour in an existing method of an object.*
- *Facade provides a simplified interface to a large body of code.*
- *Flyweight reduces the cost of creating and manipulating a large number of similar objects.*
- *Proxy provides a placeholder for another object to control access, reduce cost, and reduce complexity.*

BEHAVIORAL

Most behavioral design patterns are specifically concerned with communication between objects.

- *Chain of responsibility delegates commands to a chain of processing objects.*
- *Command creates objects that encapsulate actions and parameters.*
- *Interpreter implements a specialized language.*
- *Iterator accesses the elements of an object sequentially without exposing its underlying representation.*
- *Mediator allows loose coupling between classes by being the only class that has detailed knowledge of their methods.*
- *Memento provides the ability to restore an object to its previous state (undo).*
- *Observer is a publish/subscribe pattern, which allows a number of observer objects to see an event.*
- *State allows an object to alter its behavior when its internal state changes.*
- *Strategy allows one of a family of algorithms to be selected on-the-fly at runtime.*
- *Template method defines the skeleton of an algorithm as an abstract class, allowing its subclasses to provide concrete behavior.*
- *Visitor separates an algorithm from an object structure by moving the hierarchy of methods into one object.*

Source: Design Patterns. Wikipedia. The Free Encyclopaedia. 2023

HEURISTICS

A set of rules, based on educated guesses, that limits the search for solutions. These are intended to increase the probability of solving a problem, which is not well understood.

BENCHMARK

A test to measure the performance of computer software, hardware or their components. These are used to compare the relative performance of competing products.

However, these tests are typically designed by one manufacturer to highlight the advantages of their products, compared with competing products. Therefore the results of the tests are often disputed and considered biased or unreliable.

TIME COMPLEXITY (OF AN ALGORITHM)

This relates to how much longer it takes an algorithm to solve a problem as the size of that problem increases. That is to say, how much longer would it take a theoretical

software procedure to perform its task when the size of that task increases? In theory, the lesser the increase in time it takes, as the size of the problem increases, the better that algorithm or software procedure is. Compared with one which has a greater increase in time.

HACK

A quick job that produces what is needed but not well.

HACKER

Someone who works by using Hacks. A Programmer who writes software not by planning, but by misusing the design of software, software tools, programming languages, computer hardware and different techniques to achieve a quick result.

OBSESSION WITH EFFICIENCY

In a software project, not only the engineers involved may become obsessed with efficiency. Other staff may become obsessed too.

PLACE-HOLDER

An item in a software production process (e.g. a software procedure, a software module, a 3D model, a Texture, a polygon, a sound effect) which acts as a substitute for a feature which has yet to be designed. That is to say, it has no clear requirements to meet. It either does nothing or only partially implements the feature.

BUGS, HACKS AND PLACE-HOLDERS

External and internal software errors. See the chapter entitled

Division and Consistency

in the book

Event-Database Architecture for Computer Games: Volume 2, Game Design and the Nature of the Beast.

HASTE MAKES WASTE!

From John Ray's 1678 Proverb collection.

NAMING CONVENTION

A written convention for naming software data, procedures and modules. The names should give helpful information about the use of each, in order to avoid errors.

HUNGARIAN NOTATION

A Naming convention invented by Charles Simonyi, a Hungarian, while at the Microsoft Corporation.

SCOPE (OF DATA)

The limited block (i.e. software procedure or module) where software data may be used. This helps protect the data from erroneous changes, allows reuse of the same name for the data, in other blocks, and simplifies each block by limiting the data to that block.

COMMENTS

Text embedded in software code, which is ignored by the computer, and is merely there to help explain the use and function of software data, procedures and modules.

DOXYGEN

A software tool used to generate documentation for software, from the set of computer files used to build its software modules normally by extracting the Comments from those files.

HTML

Hypertext Markup Language. A programming language for describing documents displayed on the World-Wide-Web.

FIRST PROGRAMMING LANGUAGE

The first programming language that allowed you to name data was Formula Translation (FORTRAN), created by the IBM Corporation in 1957.

SOFTWARE INDUSTRY COMMENTATORS

Accompanying many tools which have been introduced into the Software industry, those who have made this introduction have stressed the importance of keeping the components of a computer system as independent as possible.

COMPLEX DATA TYPES

The translation of simple constructs (i.e. nouns), in a natural language, into complex constructs (i.e. software modules), in a programming language.

TYPE SAFE (OF DATA)

In theory, software which is type safe has sets of data which have been so well defined that it is possible for the tools, which use that data to build software, to automatically recognise erroneous steps within it. Hence it is impossible for the software to be incorrect.

In practice, there are always subsets of data which are not well defined and ambiguous. And it is still possible for the software to be incorrect.

DICTIONARY (OF NAMES)

A list of definitions of the names for software data, procedures and modules used in a project. In practice, there will not be a single list. The definitions will be spread throughout the software code. You may need to ask the Programmers involved, what each data or procedure name means.

CODING STANDARD

A document used in software companies. It outlines the Naming convention, and other guidelines, to follow in order to produce software of a consistent, maintainable standard.

HIGH-LEVEL LANGUAGE

A programming language which tries to use natural language words and grammar in order to be easy to understand and use.

BASIC, FORTRAN, COBOL, SQL

Beginners All-purpose Symbolic Instruction Code, Formula Translation, Common Business Orientated Language, Structured Query Language. These are all examples of high-level programming languages.

PDL

Program Design Language. A language for producing structured software designs, created by Caine, Faber and Gordon Inc.

OR, AND

These are logic operators used in programming languages to test when either one of two conditions (A or B) have become true. These are also used to test when both (A and B) have become true.

*,/, ||, &&, sqrt()

These are all mathematical notations, software procedures and logic operators used in the programming language ‘C’ and ‘C++’. These are two of the most popular programming languages used in the Computer Games industry.

MILITARY CONTRACTS

Software Developers of components used in military applications have traditionally been required to follow a recognised standard, for software engineering, in order to get the contract.

SOFTWARE ENGINEERING AS A PROFESSION

Since software service providers do not claim to be members of a profession they cannot be legally sued for malpractice.

ABSENCE OF LEADERSHIP

This is not the complete lack of leadership per se, but the management of software production through a vicarious leadership or macromanagement.

CULT OF PERSONALITY AND A VICARIOUS LEADERSHIP

Two forms of leadership adopted in software production. See the chapter entitled

The Nature of the Beast

in the book

Event-Database Architecture for Computer Games: Volume 2, Game Design and the Nature of the Beast.

FORWARD ENGINEERS AND REVERSE ENGINEERS

Two schools of thought which view software production as a science and as an art. And as a result rely on Forward engineering or Reverse engineering in software production. See the chapter entitled

Forward Engineers and Reverse Engineers

in the book

Event-Database Architecture for Computer Games: Volume 2, Game Design and the Nature of the Beast.

THE MYTH OF SELF-DOCUMENTING CODE AND DATA

The belief that the written software code and data alone can convey the design behind the software. See the subchapter entitled *The Myth of Self-Documenting Code and Data*.

GOOD MEMORY (TO A LEADER)

A good memory helps a leader love the work and stops him or her hating it, through the frustration of repeated mistakes. This gave rise to the idea of a philosopher-king in antiquity.

EVENT-DATABASE ARCHITECTURE KNOWLEDGE TEST

A multiple choice test where each question asks you to select the meaning of names of Events, Actions, Game Objects, Database Tables, Records and Fields in the Game Database of the Event-Database Architecture. Where the correct answers come from the definitions of these items in the data design. See the chapter entitled

The Nature of the Beast

in the book

Event-Database Architecture for Computer Games: Volume 2, Game Design and the Nature of the Beast.

ACADEMIC COURSE (ABOUT LEADERSHIP)

There are several post-graduate courses available which claim to teach students how to become business leaders. The influences of these courses are very wide ranging. But these courses have been oversold and produce false leaders. Many new leaders in industry, who do not even attend these courses still, nonetheless, read the materials from them, and put these into practice.

POST-MORTEM MEETING

A meeting conducted at the end of a software production process, by the staff involved, to retrospectively examine the pros and cons. And to decide the lessons to be learnt from the experience. The meeting comes from the academic study of Project Management not software engineering.

> ***What Is a Post-Mortem Meeting?***
> *A post-mortem meeting is a formal discussion that occurs at the end of a project. In the meeting, the project team discusses what went right and wrong and uses that information to make process improvements for future projects.*
>
> *Source: How to run a Post-mortem meeting (c) 2024. Smartsheet Inc*

THIRD PARTY GAME TESTERS

A company (e.g. Universal Speaking Ltd.) that specialises in performing the function of the QA Department in the Computer Games industry including testing the game at the end of the production process.

NARCISSISM

An excessive love or pre-occupation with yourself, which leads to a decay in your ability to empathise with others.

PREVALENCE OF WOMEN

A recent research into the characteristics of the workplace, in high-technology industries, found that the macho, competitive atmosphere was a barrier to women.

EVENT-DATABASE ARCHITECTURE PRODUCTIVITY FORMULA

A formula for measuring the productivity of any software production process by inverting amount of waste produced. See the chapter entitled

Cause and Effect

in the book

Event-Database Architecture for Computer Games: Volume 2, Game Design and the Nature of the Beast.

THE NATURE OF THE BEAST

A figure of speech which means a regrettable but inescapable characteristic.

For example, some would say it is the nature of the beast that the revenue gained by taxing tobacco leads to many public health problems caused by tobacco smoke. And the government has to then spend some of its revenue providing health care.

In psychology, the figure of speech resonates with a mental state known as cognitive dissonance. This is defined as the mental discomfort a person feels when their beliefs and actions are inconsistent and contradictory.

In the case of new apprentices in the Computer Games industry, on the one hand they believe they love making computer games, they can make a career out of it and they can give it their best effort. On the other hand their actions in the Software Evolution Process used to make Computer Games are performed in the context of a chaotic process whose outcome they cannot control. And as a result their best efforts may not be emerge in the final outcome.

In philosophy, the figure of speech implies that in certain cases, you can only do some good through some evil.

For example, some would say it is the nature of the beast that keeping peace between nations entails being prepared to fight wars against those who violate the peace.

This example hints at the possible origins of the figure of speech. Namely a biblical prophecy about a man, a politician, called The Beast. Who will come at the end of the world, to fool mankind. He will create a one world government and in the name of keeping the peace will fight many wars. And no one will be able to buy and sell under this government without the 'mark of the beast'.

This origin leads to another use of the figure of speech. Namely the use of the Beast as a metaphor or mark of moral decay in literature. Like the moral decay at the end of the world in the biblical prophecy. A good example of this is in the book Lord

of the Flies where the Beast is used as a metaphor of the moral decay that occurs amongst a group of army cadets marooned on an island.

Going back to the case of new apprentices in the Computer Games industry, if they believe they can produce their best effort, in a Software Evolution Process which they find chaotic, then philosophical speaking this is a sign of moral decay. Since you cannot do well or excel at that which you hate or causes you pain. This is why the philosopher Socrates said in the book *The Republic* that a king must have a good memory. Since he cannot do well that which he hates or causes him pain, and after much effort, he finds he has made little progress because of a bad memory and repeated mistakes.

Finally, the term nature of the beast implies that there is some regrettable deed you need to do to achieve some greater good. Now philosophically, if after you (an apprentice) have done the regrettable deed (i.e. taken part in a Software Evolution Process), you have to keep repeating that regrettable deed to survive, then that deed is no longer regrettable. It is just part of your character.

> *In the field of psychology,* ***cognitive dissonance*** *is described as the mental discomfort people feel when their beliefs and actions are inconsistent and contradictory, ultimately encouraging some change (often either in their beliefs or actions) to align better and reduce this dissonance.*
>
> *Source: Congnitive Dissonance. Wikipedia. The Free Encyclopaedia. 2024*

> *Fancy thinking the Beast was something you could hunt and kill! You knew, didn't you? I'm part of you? Close, close, close! I'm the reason why it's no go? Why things are what they are?*
>
> *Source: Lord of the Flies © 1959, William Golding*

RIGHT TO SILENCE

In English law, and in other countries, a person charged with a crime has the right to remain silent before and during trial in order to avoid saying anything incriminating. This is the basis of placing the burden of proof on the prosecution.

> *The origins of the right to silence and the privilege against self-incrimination are not entirely clear.…the right and privilege emerged together during the religious and constitutional struggles of the seventeenth century England. In particular, the right and the privilege are commonly said to have originated in the abolition of the Court of Star Chamber and the Court of High Commission in Ecclesiastical Causes. These courts were highly unpopular, largely because they were used to suppress religious and political dissent…the judges of both courts having the power to interrogate an accused person on oath…This exposed the accused to what the High Court has described as `the 'cruel trilemma' of punishment for refusal to testify, punishment for truthful testimony or perjury.…*
>
> *Source: The Right to Silence: An Examination of the Issues © 1999, The Parliament of Victoria, Australia*

HUMAN RESOURCES

A department of an organisation responsible for the recruitment, payment and personal welfare of staff.

IWGB GAME WORKERS

A trade union established in 2019 for workers in the Computer Games industry. It deals with many common issues in the industry such as overtime, sexism and harassment. If you want to join the Computer Games industry, you should join this union. They will help you face these common issues.

*'The **Game Workers branch of the IWGB** is a worker-led, democratic trade union that represents and advocates for UK game workers' rights.*

We seek to increase the quality of life for all game workers by campaigning to:

- *End the institutionalised practice of excessive/unpaid overtime*
- *Improve Diversity and Inclusion at all levels*
- *Inform workers of their rights and support those who are abused, harassed, or need representation*
- *Secure a steady and fair wage for all*

WHY DO WE NEED A UNION?

***74%** of game workers are not paid overtime, but **90%** can be expected to work extra hours. [1]*

***53%** of game workers believe that their skillset could secure better wages and conditions in another industry. [1]*

***45%** of women feel they have or will at some stage encounter barriers to their career progression because of their gender. [2]*

***45%** of women have experienced some form of bullying or harassment whilst working in games or by being associated with the industry. [2]*

***Two thirds** of games companies (worldwide) do not have mechanisms in place to deal with harassment or abuse. [1]'*

Source: We are the IWGB Game Workers (c) IWGB Game Workers 2020–2023. https://www.gameworkers.co.uk/

QUALITY ASSURANCE

In theory, a system which ensures that a company's processes (as supposed to their product) will meet all of the customer's requirements and specifications. In practice, software companies just apply two Quality Controls in the latter stages of production, known as Alpha testing and Beta testing, and call it Quality Assurance or QA.

ARBITRATION SERVICE

A charity or a commercial company that mediates between two sides involved in an industrial or employment dispute.

SUBSTITUTE FOR LEADERSHIP

In his 14 points for manufacturing quality goods, W. Edwards Deming suggested that the use of annual appraisals, to measure and improve the performance of an employee, was in effect a substitute for leadership.

TEAM LEADER

This position and title is given to a very experienced member of a team, developing software, to denote his or her seniority. But the title is an oxymoron which reflects the contradictions of this position.

Index

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W