Comfort Eagle

White Dwarf Map Editor

Internal Documentation

Version 1.1

Revision History

Date	Version	Description	Author
25/02/2002	1.0	Initial Revision	W.D. Development Team
15/04/2002	1.1	Final Revision	W.D. Development Team

Table of Contents

List of Figures

List of Tables

Architecture Document

White Dwarf Map Editor

1 Introduction

1.1 Purpose

This SAD is intented to provide a comprehensive architectural overview of the *White Dwarf Map Editor*. Number of different architectural views will be used to depict the different architectural concepts and decissions that will be used as a base of design for the system.

1.2 Scope

The *White Dwarf Map Editor* will be developed by *Comfort Eagle* corporation to create a map for the *White Dwarf Game*. This SAD document provides an architectural overview of the *White Dwarf Map Editor*. It describes the architecture that will be used to meet the different functional and non-functional requirements derived from the project proposal and inputs from the various stakeholders.

In order to reach a broad audience, the architecture will be presented using a number of different architectural views as proposed by the *Rational Software*[4] 5+1 model for software architecture.

1.3 Definitions, Acronyms, and Abbreviations

The following is a list of definitions, acronyms and abbreviations that will facilitate the understanding of the document.

1.3.1 Definitions

Actor	Object displayed on the screen (e.g. a ship or a bullet). Actors have a type, initial energy level, weapon and state definitions. During the game, at run time, they will be assigned a dynamic position and amount of energy.
AI	Controls the behavior of all the actors in a formation.
Chapter	A playable section of the game. Chapters are invisible in the gamer's perspective since the transitions between them are done in a continous manner, even if some things change like the scrolling speed, the music, the background image, and so on. Beginning of chapters are also used as points where the player ship reappears after its death. The chapter finishes when there are no more formations left within it.
Document Type Definition	A formal grammar to a class of XML documents[1].
Episode	A collection of chapters. It is presented to the gamer as one "level". At the beginning the name of the level is presented and at the end the ship exits the screen to go to the next level.
Extensible Markup Language	A flexible, multi-dimensional, text-oriented markup language[1].
Formation	Group of actors that can be controlled by a single AI. It is often seen in the game as a fleet of ships moving together or as a "boss" with multiple body parts. It can also wrap a single actor or even an item.
Gamer	Someone who plays video games.
Game Engine	Modules of the game responsible of controlling the objects within the game. It is usually closely linked with the AI and, of course, the map system.

Continued on next page...

Media	Sounds, images, animations, and so on.
Scenario	Collection of maps, actors, AI and different media. Scenarios may contain several episodes.
State	Different status that an actor can have throughout the game. Shooting, hit, low on energy or normal are different states that can be associated with an actor. For each state, an image and a sound can be associated.
White Dwarf	The tentative name of the game for which the game editor will be created.

Table 1: Definitions

1.3.2 Acronyms

AI	Artificial Intelligence
DTD	Document Type Definition[1]
FAQ	Frequently Asked Questions
SSME	Space Shooter Map Editor
SAD	Software Architecture Document
GUI	Graphical User Interface
GPU	Graphical Processing Unit
RAM	Random Access Memory
UI	User Interface
XML	Extensible Markup Language[1]

Table 2: Acronymns

1.3.3 Abbreviations

NA	Not Applicable
s/he	He/She

Table 3: Abbreviations

1.4 References

[1] World Wide Web Consortium. Extensible markup language (xml) 1.0 (second edition). `http://www.w3.org/TR/2000/REC-xml-20001006`.

[2] Philippe Kruchten. The 4+1 view model of architecture. `http://www.rational.com/media/whitepapers/Pbk4p1.pdf`.

[3] David Garland Mary Shaw. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice Hall, 1996.

[4] Rational software. `http://www.rational.com`.

1.5 Overview

This SAD presents the architecture of the *White Dwarf Map Editor* system using a set of different views on the system. The views shown in this document are directly derived from the 5+1 model for software

architecture proposed by *Rational Software*[4].

Each view proposed will be accompanied by an explanation of the various components it contains and how they interact with each other. Furthermore, each view will also be depicted visually using one or more diagrams. The diagrams presented to facilitate the understanding will follow the notation described in the 4+1 architectural model[2].

2 Architectural Representation

The architecture of the system will be presented using the following views: use case view, logical view, process view, deployment view, the implementation view and finally the data view.

The use case view will consist of a division of the various tasks that the distinct stakeholders can perform with the system using *actors* and *use cases*. This view will be used to derive, elicit and validate the different requirements of the system.

The logical view will be used to decompose the system into a set of key abstraction that will be utilized to fulfill the functional requirements of the system. It will provide a high-level breakdown of the system in terms of *objects* and *object classes* and their relations.

The process view will divide the system in terms of *processes*, *threads* and *tasks* in order to show how the system will reach the non-functional requirements. This view is useful to show the different threads of control and concurrency present within the system.

The deployment view will also be used to represent non-functional requirements. This view will show how the different processes identified in the process view are mapped to the different *processing nodes* available.

The implementation view will depict how the system will be physically decomposed. This view is intented to show how the system will be organized in terms of *libraries* and *subsystem*. Looking at this view, it will be easy to see which components are shared among the different modules of the system and which ones are reused from existing libraries.

The data view is intended to show how the system and the scenario it creates will be arranged at the *files* and *directories* level.

3 Architectural Goals and Constraints

There are some key requirements and system constraints that have a significant bearing on the architecture. These are:

1. The *White Dwarf Map Editor* must be consistent and useable with *White Dwarf Game* since it will be eventually integrated with the latter.
2. The *White Dwarf Map Editor* must use the *White Dwarf Game* game engine to render and display the map, i.e. the terminology and concepts must be easily recognizable from the game.
3. The *White Dwarf Map Editor* UI must be representative of the *White Dwarf Game*.
4. The *White Dwarf Map Editor* must provide a sophisticated UI for the developers of the game, enabling them to quickly create new exciting scenarios.
5. The *White Dwarf Map Editor* must also please to the new players of the game, by providing an intuitive UI with simple language and concepts.
6. The *White Dwarf Map Editor* must run in the Windows 2000 labs at Concordia University. All hardware and software constraints must be taken into account.

7. The *White Dwarf Map Editor* should use XML to store configurations of the scenario.

Being that the software is an editor, the design of *White Dwarf Map Editor* will have a user-centered approach, that is to say, that the design of the UI will have a significant impact on the architecture of the software.

4 Use-Case View

The following is a list of use cases that represent some significant or central functionality to the final system. However, only a few of the most important use cases will be used to do the main success scenarios. Architecturely speaking, the other use case scenarios are simply a variation of the ones listed in Section 4.1.

The Use-Case Diagrams are in Appendix 12.2 on page 19.

UC-1	Create a New Scenario	**UC-2**	Build a Scenario Using the Wizard
UC-3	Build a Scenario Using the Wizard	**UC-4**	View a Scenario
UC-5	Add a New Episode	**UC-6**	Add an Episode using the Wizard
UC-7	Remove an Episode	**UC-8**	View an Episode
UC-9	Change the Episode Order	**UC-10**	Add a Chapter
UC-11	Remove a Chapter	**UC-12**	View a Chapter
UC-13	Change the Chapter's Order	**UC-14**	Change a Chapter's Music
UC-15	Change a Chapter's Background	**UC-16**	Change a Chapter's Scrolling Speed
UC-17	Add a Formation to a Chapter	**UC-18**	Remove a Formation from a Chapter
UC-19	Move a Formation in a Chapter	**UC-20**	Add a New Actor
UC-21	Add an Actor Using the Wizard	**UC-22**	Remove an Actor
UC-23	View an Actor	**UC-24**	Change an Actor's name
UC-25	Change an Actor's Default Image	**UC-26**	Change an Actor's Type
UC-27	Change an Actor's Weapon	**UC-28**	Change an Actor's Item
UC-29	Add a State to an Actor	**UC-30**	Remove a State to an Actor
UC-31	View the State of an Actor	**UC-32**	Change the Image of a State
UC-33	Change the Sound of a State	**UC-34**	Change the Name of a State
UC-35	Add a New Formation	**UC-36**	Add a Formation Using the Wizard
UC-37	Remove a Formation	**UC-38**	View a Formation
UC-39	Change the Type of a Formation	**UC-40**	Change the Name of a Formation
UC-41	Add an Actor to a Formation	**UC-42**	Remove an Actor from a Formation
UC-43	Change the Role of an Actor in a Formation	**UC-44**	Change the Item of a Formation
UC-45	Import an Image	**UC-46**	Remove an Image
UC-47	Change the Name of an Image	**UC-48**	Import a Sound
UC-49	Remove a Sound	**UC-50**	Change the Name of a Sound
UC-51	Add a New Player Formation	**UC-52**	Add a Player Formation Using the Wizard
UC-53	Remove a Player Formation	**UC-54**	View a Player Formation
UC-55	Change the Type of a Player	**UC-56**	Add an Actor to a Player
UC-57	Remove an Actor from a Player Formation	**UC-58**	Change the Role of an Actor in a Player Formation

Table 4: Use Cases List

Some use case scenarios will be investigated further in Section 4.1. For now lets look at a brief description of some scenarios to try to understand the motivation of the user in taking these actions.

- **UC-2 Build a Scenario Using the Wizard**

  New users can use the "Scenario Wizard" to easily create a new scenario and all its contents, including the episodes, chapters, formations, actors, and so on.

- **UC-4 Add a New Episode**

  The developers and avid users will want to build episodes for the game. Episodes are commonly known as a levels in the gaming jargon.

- **UC-9 Add a Chapter**

  The user can add new chapters to already existing episodes or new episodes they created themselves.

- **UC-34 Add a New Formation**

  The user can create new formations of actors. They can be placed in any chapter they desire once they are created.

- **UC-19 Add a New Actor**

  The user can add any number of actors to any formation.

- **UC-28 Add a State to an Actor**

  As stated in Section 1.3.1, actors are objects displayed on the screen. Each actor has states, for example a dead state, shot state, normal state, boosted state, and so on. For example we could add an invicible state to a ship.

- **UC-42 Change the Role of an Actor in a Formation**

  Formations are goups of actors that are a controlled by a single AI. Each actor has a specific role from the point of view of the AI. This use case allows the user to change the role of an actor in its formation. For example, there could be a ship formation, and one of the ship could be the "head ship" of the formation. This use case allows the user to specify which ship in the formation is the "head" one.

- **UC-13 Change a Chapter's Music**

  Each chapter has a music associated to it. A user may decide to add or change the music played in some chapter.

- **UC-44 Import an Image**

  The user can import an image in a scenario. The image can then be used for an actor's state or for the background of a chapter.

4.1 Use-Case Realizations

To better understand the interaction of the user with the systemn, for each scenario listed in Section 4 a detailed description and a sequence diagram will be given.

4.1.1 Build a Scenario Using the Wizard

In this Use-Case, the user must be helped to create a new scenario. To create a proper scenario, the user must at minimum create one player formation and one episode. Thus, this Use-Case will make use of the Use-Cases "*UC-51 Add a Player Formation Using the Wizard*" and "*UC-5 Add an Episode using the Wizard*". Those wizards will be called once for each new player formation or episode that the user want to create.

The state diagram for this Use-Case is in Figure 13 on page 22.

4.1.2 Add an Episode Using the Wizard

The episode wizard guides the user into the creation of a new episode. The user is allowed to add chapters and change the settings of each of Additionally, the user may decide to create a new formation before adding it to the chapter. In such a case, the creation of the new formation is delegated to the Formation Wizard.

The state diagram for this Use-Case is in Figure 14 on page 23.

4.1.3 Add a Formation Using the Wizard

The formation wizard allows the user to create a new formation with step by step instructions. S/he will be given a chance to add actors to the formation and even to create a new actor if it doesn't already exist. In that case, the work is delegated to the Actor Wizard.

The state diagram for this Use-Case is in Figure 15 on page 23.

4.1.4 Add an Actor Using the Wizard

The actor wizard guides the user into a step by step creation of an actor. This involves setting the parameters of the actor and its states.

The state diagram for this Use-Case is in Figure 16 on page 24.

5 Logical View

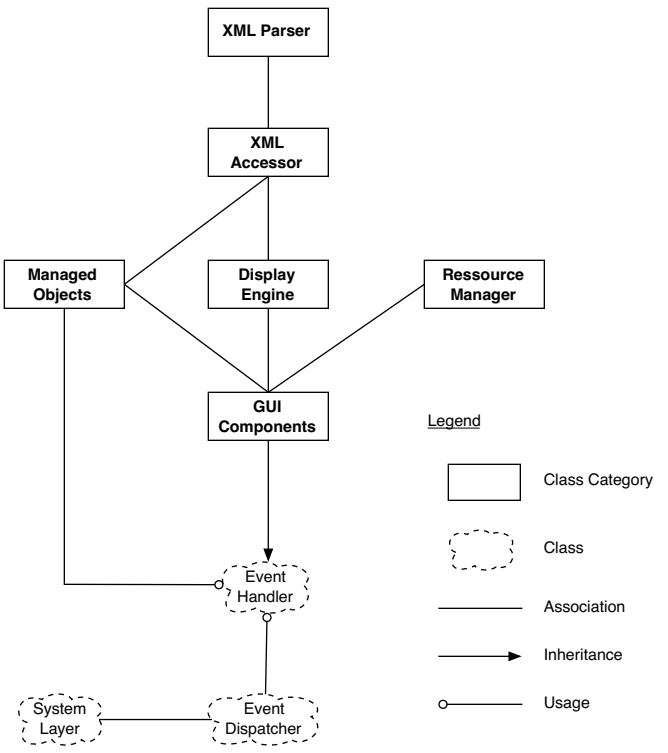

Figure 1: Logical View Design Diagram

5.1 Overview

Editors normally adopt the *blackboard* architecture style[3] since they are a collection of tools that operate on a shared data space.

5.1.1 System Layer

This is the UI part of the operating system and is shown here only for completeness. It is the System Layer itself that displays the GUI widgets and stores their properties. The System Layer has access to the Event Dispatcher to create and receive new events from and to the different GUI widgets it manages.

5.1.2 Event Dispatcher

The Event Dispatcher is also part of the operating system and is shown here only for completeness. The Event Dispatcher is in charge of sending and receiving events to the registered Event Handler of the user application. Here, the events are UI events which mostly correspond to user input and output.

5.1.3 Event Handler

The Event Handler is part of the *White Dwarf Map Editor*. Whenever the Event Dispatcher of the operating system needs to send an event to the *White Dwarf Map Editor* application, it does so by calling the Event Handler function with the event parameters.

5.1.4 GUI Components

The GUI Components provide an interface to the user to interact with the system. They are buttons, windows, scrollbars, etc. They receive user events through the Event Handler interface and react by either performing the task requested by the user or by delegating the task to another component by sending an internal event. More information on this topic in Section 5.2.1

5.1.5 Rendering Engine

A group of functions that are specialized at rendering the game components on screen.

5.1.6 Resource Manager

A set of utility functions made to simplify access to the various media files the editor needs to move, copy, remove and preview.

5.1.7 Managed Objects

The various objects which the editor needs to add and modify in the scenario data. More information on this topic in Section 5.2.2.

5.1.8 XML Accessor

A set of functions used as a proxy to the actual XML data, as produced by the XML Parser. More information on this topic in Section 5.2.3.

5.1.9 XML Parser

This is a *validating XML parser and generator*. In other words, it can put in memory a structure representing an XML file on disk ("*parser*"), validate the memory structure based on a XML DTD ("*validator*"), and properly put back on disk the memory structure ("*generator*").

5.2 Architecturally Significant Design Packages

5.2.1 GUI Components

The GUI components provide an interface for the user to interact with the system. They respond to GUI events by changing the state of the software.

GUI events can originate from the operating system or from the Managed Objects. The events received from the operating system will be handled through a callback system, while the events received from the Managed Objects will be handled through a subscriber system.

The events from the operating system will reach the event dispatchers first. These behave as adapters to dispatch the events to the GUI Components. Those events are direct consequence of an action from the user.

The events from the Managed Objects will be sent directly through the Event Handler interface and result from a modification to the XML data (e.g. if a node was removed, added or modified). When the GUI components need to display information about an XML node, they provide their Event Handler interface and expect to be informed whenever the XML node is modified. That way, they can react by updating the screen or closing if the node was deleted.

5.2.2 Managed Objects

The Managed Objects are object-oriented representations of the various information that need to be stored for a scenario. Each class is a proxy of the actual operations done on the parsed XML data, in memory, through the XML Accessor functions.

The primary goal of the Managed Objects is to ensure that the XML data remains valid, i.e. follows the rules of the DTD file (refer to Section 9). This simplifies the use of the XML data, as it is not required to constantly worry about what rules need to be followed to make sure the XML data remains valid.

Also, the Managed Objects do whatever conversion is needed to convert the data, as it is logically seen, into properly formatted XML data. This includes converting numbers between their numerical value in memory and their latin-iso-1 character representation.

The term "Managed" in "Managed Objects" means that this component is always aware of what elements in the XML data are currently referenced. This means, for example, that any class that makes use of some Managed Object will be informed when another class removes that object from the XML data. This is done by sending some events to the Event Handler.

5.2.3 XML Accessor

A collection of straightforward, low-level functions which read and modify the parsed XML data. In this module, the data is modified without any restriction.

All operations that might need to be done on the XML data need to be implemented here. This includes, but is not limited to:

- Find an element from its ID.

- Add a new element.
- Enumerate through the list of attributes of an element.
- Change the ordering of two elements.
- Change the value of an attribute.

This is done primarily because the system will use a "third-party" XML Parser. Using this architecture scheme, it will be easily feasible to change to any other kind of XML Parser, without requiring too much overhead since only this component will need to be changed.

6 Process View

Since every window of the system must concurently proccess information in order to be correctly updated each of them will executed in a separate thread of control.

The window were the actual chapter is rendered and display, called *Chapter Display*, will be threated differently from the other windows since it will need to use the OpenGL libraries and the game engine.

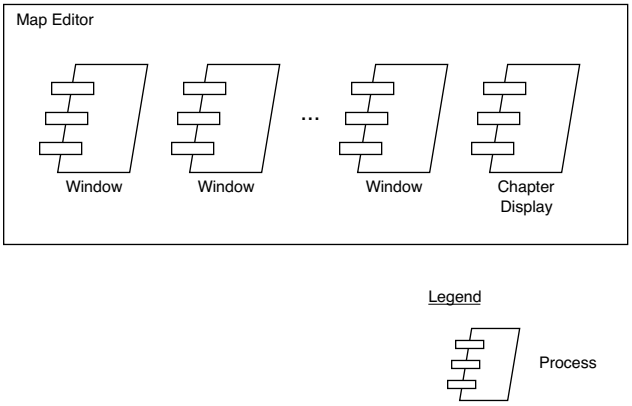

Figure 2: Process View Design Diagram

7 Deployement View

Although the system would take advantage of a system with multiple processors, where the load of each window thread could be shared, the minimal requirement is a single processor machine with a graphics card supporting OpenGL hardware acceleration. Typically, the system should work with 50MB of disk space and 32MB of available RAM.

All the threads except the *Chapter Display* will be processed by the main processor. The *Chapter Display* will be processed on the GPU of the graphics card.

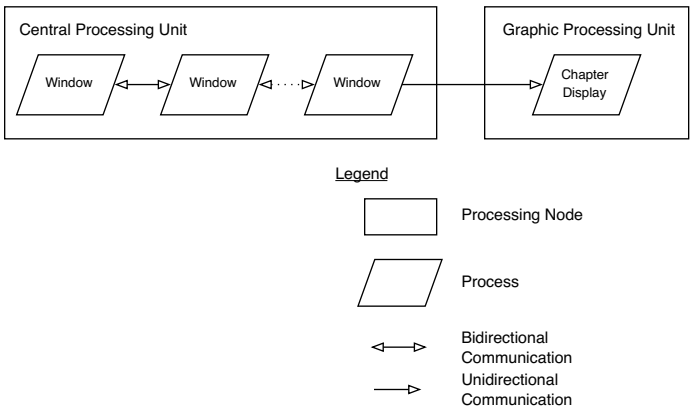

Figure 3: Deployment View Design Diagram

8 Implementation View

8.1 Overview

Application The application layer provides a graphical interface that allows the user to access and modify the XML data of the scenario. It relies entirely on the servies of the layer immediately below it, and thus it is platform independant. The application is the logic of the software, and it implements this logic with using the low-level library services.

Low-Level Libraries Low-level libraries consist of shared code between the game and the editor. Those will be the core of the editor, and their reuse in the game will ensure consistency between the two. They are also platform independant and can be thought as specialisations of the third-party libraries in order to provide high level services to the application. This layer is as platform independant as the third-party library they are built with.

Third-Party Librarties Third-party libraries are the building blocks of the application. They provide the most basic services, such as XML validation and parsing, display of windows and widgets and 2D graphics rendering.

8.2 Layers

8.2.1 Application

Editor Engine: The Editor Engine provides graphical tools to manage the XML Objects. It is written in C++.

8.2.2 Low-Level Libraries

XML Object LIbrary: The XML Object Library provides services to reading and writing Managed Objects from XML data. This library will be shared between the editor and the game. Although internally it will be written in C++, it will provide a C interface so that it can be loaded and linked dynamically.

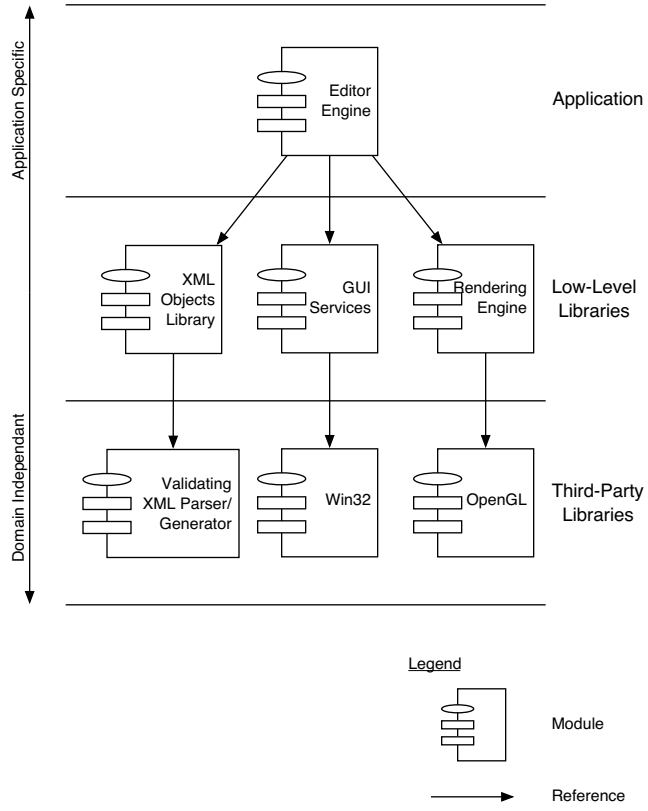

Figure 4: Implementation View Design Diagram

GUI Services: The GUI Services play the role of an adapter between the operating system GUI components (windows and widgets) and the Engine Editor. This scheme provides a way to keep the Editor Engine platform independant. It will be written in C++. Although it is considered that the system will eventually have different ports, it will initially be written using the MFC services. The GUI Services implement the GUI Components. As the public interface of this module is in C++, it will be statically linked with the Editor Engine.

Rendering Engine: The Rendering Engine provides services to display a chapter on the screen. This library will be used both in the editor and in the game and will be written in C using the OpenGL libraries.

8.2.3 Third-Party Libraries

XML Parser We will use libxml to parse the XML from the ascii file. This component will be statically linked to the XML Object Library.

Win32 The GUI Services component will be dynamically linked to Win32.

OpenGL The graphics for the preview of the chapters will be done using the services of OpenGL. This component will be dynamically linked to the Rendering Engine.

9 Data View

9.1 Files

Figure 5 shows how the media files and libraries will be organized on the file system. A word in italics means that it is a directory (the trailing slash can be ignored), otherwise it is a file. The names shown here were chosen only as examples and may change during development.

- White Dwarf Game
- White Dwarf Map Editor
- *Libraries/*
  - Rendering Library
  - XML Library
  - ...
- *Resources/*
  - Image1.png
  - Image2.png
  - ...
  - Sound1.aif
  - Sound2.aif
  - ...
- *Scenarios/*
  - *Scenario1/*
    - data.xml
    - *Resources/*
      - NewImage.png
      - NewSound.aif
      - ...
  - *Scenario2/*
  - ...

Figure 5: Organization of the Media Files and Librairies

The directory "*Libraries*" contains code libraries that will be used both by the *White Dwarf Game* and the *White Dwarf Map Editor*. This is why the directory is at the same level as both the Game and the Map Editor.

The directory "*Resources*" which is at the same level as the *White Dwarf Game* and *White Dwarf Map Editor* contains the media shared amongst all the different scenarios of the game.

The directory "*Scenarios*" contains the different scenarios that can be played in the *White Dwarf Game* or edited in the *White Dwarf Map Editor*.

Even if a scenario is self-contained, since each one has its own "*Resources*" directory containing their own media files, it can also access the shared media found in the top-level "*Resources*" directory. Obviously, the

White Dwarf Map Editor will not allow the user to change the shared media, since all scenarios will assume that it was not modified and they might not work if the shared media was changed.

Finally, the information for a scenario is contained in a single XML file (here, "*data.xml*") that was generated earlier by the *White Dwarf Map Editor*.

9.2 XML Data

As often mentioned earlier, the information for a scenario is stored in an XML file. The advantages of doing so are:

1. Several third-party libraries can already parse, validate and generate XML files;
2. XML can easily represent multi-dimentional structures;
3. XML documents can be validated by a DTD which formally captures almost all the requirements of the data;
4. XML documents can be examined and edited with any text editor.

Figure 6 on page 18 shows the DTD file that will be used by the *White Dwarf Map Editor* and the *White Dwarf Game* to validate the data XML file of a scenario. Note that the file can change during the development.

From Figure 6, we can see several things.

An *episode* contains at least one *chapter*, a *chapter* contains at least one reference to a *formation*, a *formation* has at least one reference to an *actor*, and so on.

Here, references are used whenever an element can be referenced more than once. Otherwise, if an element can be referenced only once, we put the element within the other element that contains it.

Note that it is a limitation of XML that while an *IDREF* references an *ID* attribute of another tag, it cannot be specified what tag it should be specifically referencing. This is currently the only data requirement that cannot be fully covered by the DTD.

The first *ELEMENT*, which is for the whole document, lists all the elements that are at the "root level" of the XML data. All the other elements are rooted at another element.

Here's a quick explanation of the different elements that the *White Dwarf Map Editor* can generate. Most of the terms used here are defined in Section 1.3.1.

episode

An *episode* should contain at least one chapter.

chapter

A *chapter* contains at least one reference to a *formation*. It needs to be within a single *episode*.

The attribute *music* refers to a *sound* element. *speed* is a numerical value which represents the scrolling speed of the chapter.

speed is a human-readable integer in decimal representation. It can be negative.

background

A *background* is an image that scrolls as the chapter scrolls on screen.

The attribute *image* refers to an *iamge* element. *layer* is a numerical value which represents the relative depth of the background layer.

layer is a human-readable floating point number in decimal representation.

formationRef

A *formation reference* is a reference to a *formation* element. It contains some additional information to make it relevent in a chapter, for example its position (x and y).

x and y, like most of the other numerical values in the DTD, are represented in human-readable decimal values. They can be negative.

formation

A *formation* is the information globally meaningful to the formation of *actors*. It contains zero or more references to an *actor*, which are the initial actors available in the formation.

The attribute *ai* is a constant which refers to some game function that will control the actors in the formation. *item* refers to a *formation* which will be "dropped" in the game whenever the formation is destroyed.

player

A *player formation* is a special kind of formation which is controlled by the gamer. At least one *player formation* must exist in the XML data.

actorRef

An *actor reference* is a reference to an *actor*. It is used by *formation* and *player*.

actor

An *actor* is referenced by a *formation* or a *player*.

The attribute *image* refers to an *image* element. The attribute *type* refers to some game function that will control the actor's states. *weapon* refers to a *formation* which is controlled by the *actor*. *item*, what is "dropped" when the actor is destroyed, refers again to a *formation*.

The *actor* has zero or more *states*.

state

A *state* simply has a *name* and refers to an *image* and a *sound*.

image

An *image* simply refers to a *fileName* given some*id*. The *id* is always the one used in the XML data to refer to the image so that the file name can be changed without having to change all the elements that refer to it.

sound

Similar to *image*, a *sound* refers to a *fileName* given some *id*.

10 Size and Performance

The chosen architecture will support the following size and performance requirements:

1. The *White Dwarf Map Editor* must require less than 50MB of disk space.
2. The *White Dwarf Map Editor* must require less than 32MB of RAM.
3. The *White Dwarf Map Editor* must be able to save the current work within 10 seconds.
4. The size of the scenario created by the *White Dwarf Map Editor* must be sufficiently small to permit reasonable download time.

11 Quality

The chosen architecture will support the following quality requirements:

1. The GUI must be compliant with Windows 2000.
2. The GUI must be presented with intuitive concepts. Using the system main functions should be very intuitive. Anyone familiar with side scroller games should be able to use the basic functionalities of the SSME by interacting with it less than 30 minutes.
3. The GUI of the SSME must be representative features supported by the *White Dwarf Game*.
4. The GUI must reflect the organization of a scenario.
5. The *White Dwarf Map Editor* must create scenario in a format that allows easy on-line distribution.
6. The *White Dwarf Map Editor* must create backup intermittently.
7. The software will contain a help menu which will provide the first source of help. It will be useful to solve simple problems and answer general questions that the user might have.

12 Appendices

12.1 DTD Specifications

```
<!ELEMENT scenario (episode+, player+, formation*, actor*, image*, sound*)>

<!ELEMENT episode (chapter+)>
<!ELEMENT chapter (formationRef+) (background*)>
<!ATTLIST chapter music IDREF #IMPLIED>
<!ATTLIST chapter speed CDATA #IMPLIED>
<!ELEMENT background EMPTY>
<!ATTLIST background image IDREF #REQUIRED>
<!ATTLIST background layer CDATA #REQUIRED>
<!ELEMENT formationRef EMPTY>
<!ATTLIST formationRef x CDATA #REQUIRED>
<!ATTLIST formationRef y CDATA #REQUIRED>
<!ATTLIST formationRef id IDREF #REQUIRED>

<!ELEMENT formation (actorRef*)>
<!ATTLIST formation id ID #REQUIRED>
<!ATTLIST formation ai CDATA #REQUIRED>
<!ATTLIST formation item IDREF #IMPLIED>
<!ELEMENT player (actorRef+)>
<!ELEMENT actorRef EMPTY>
<!ATTLIST actorRef name IDREF #REQUIRED>

<!ELEMENT actor (state*)>
<!ATTLIST actor  name ID #REQUIRED>
<!ATTLIST actor  image IDREF #REQUIRED>
<!ATTLIST actor  type CDATA #REQUIRED>
<!ATTLIST actor  weapon IDREF #IMPLIED>
<!ATTLIST actor  item IDREF #IMPLIED>
<!ELEMENT state EMPTY>
<!ATTLIST state name CDATA #REQUIRED>
<!ATTLIST state image IDREF #IMPLIED>
<!ATTLIST state sound IDREF #IMPLIED>

<!ELEMENT image EMPTY>
<!ATTLIST image id ID #REQUIRED>
<!ATTLIST image fileName CDATA #REQUIRED>
<!ELEMENT sound EMPTY>
<!ATTLIST sound id ID #REQUIRED>
<!ATTLIST sound fileName CDATA #REQUIRED>
```

Figure 6: DTD specifications for White Dwarf

12.2 Use-Case Diagrams

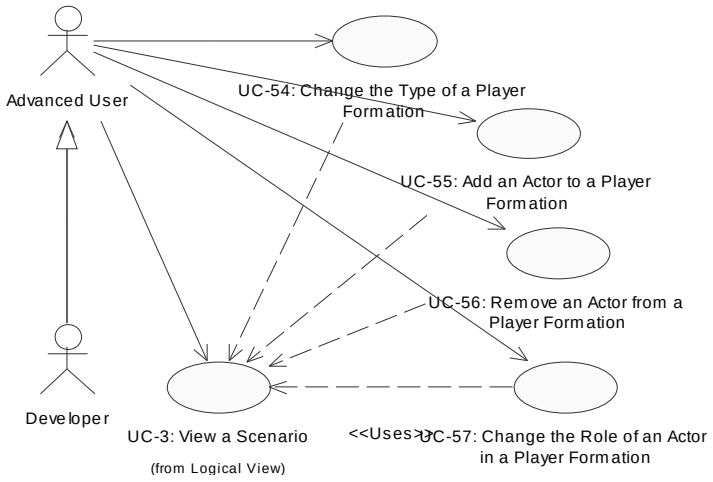

Figure 7: Use-Cases (Player Formation Related)

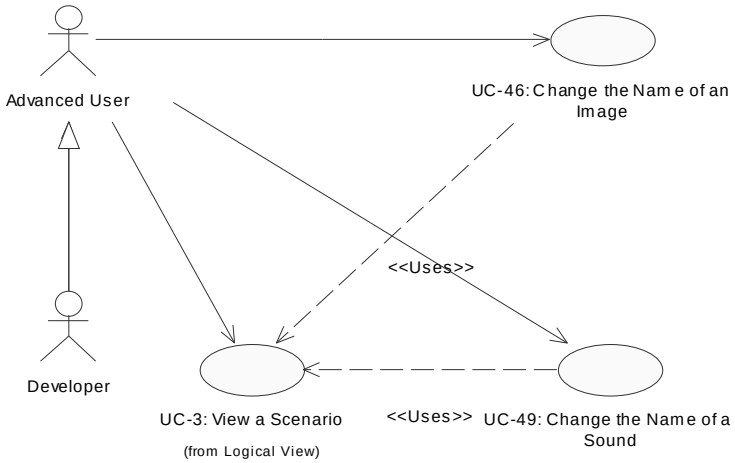

Figure 8: Use-Cases (Image and Sound Related)

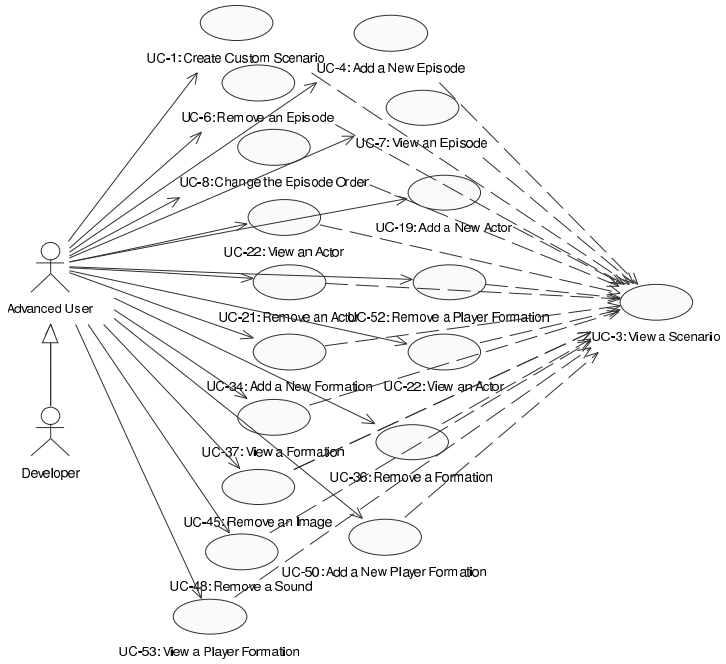

Figure 9: Use-Cases (Advanced)

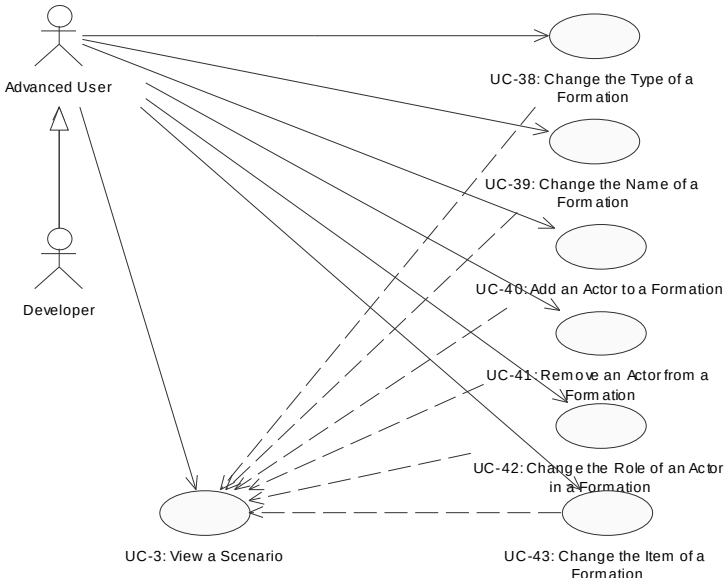

Figure 10: Use-Cases (Formation Related)

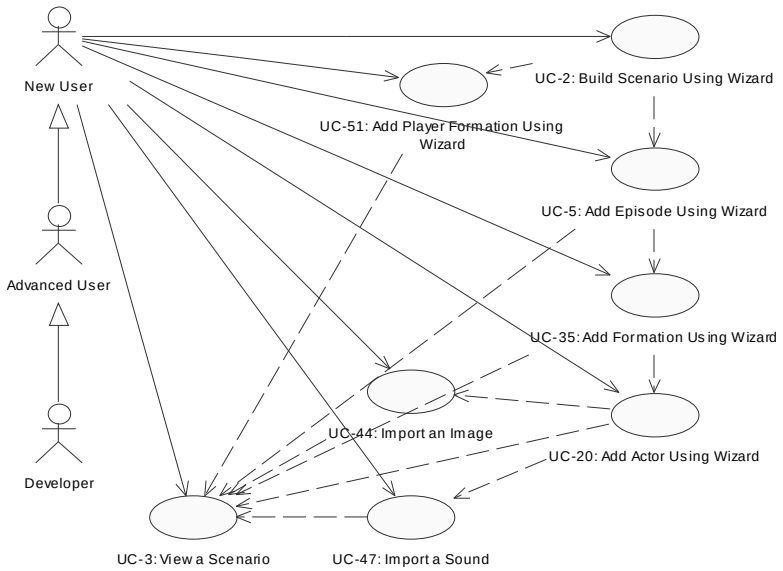

Figure 11: Use-Cases (Simple Interface)

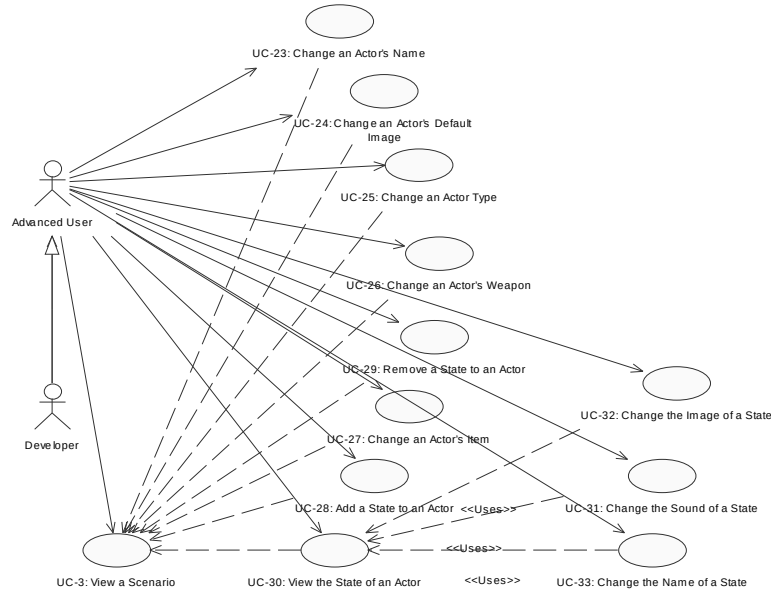

Figure 12: Use-Cases (Actor Related)

12.3 State Diagrams

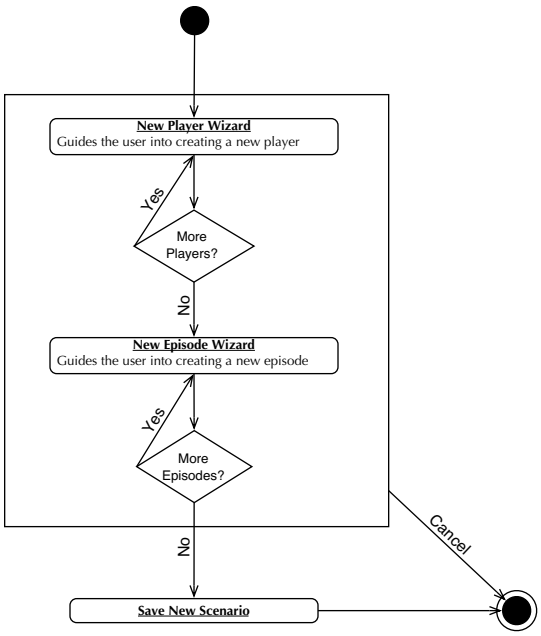

Figure 13: UC-2 Build a Scenario Using the Wizard

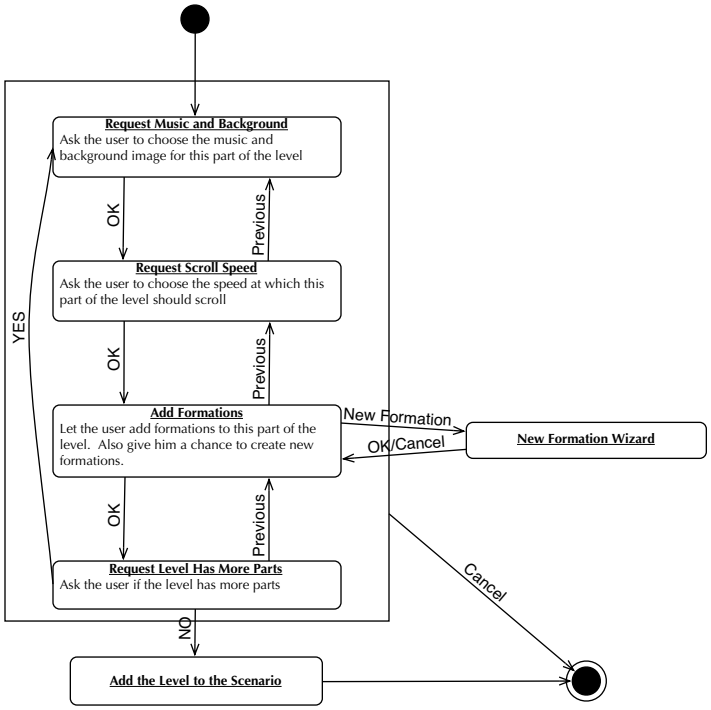

Figure 14: UC-5 Add an Episode Using the Wizard

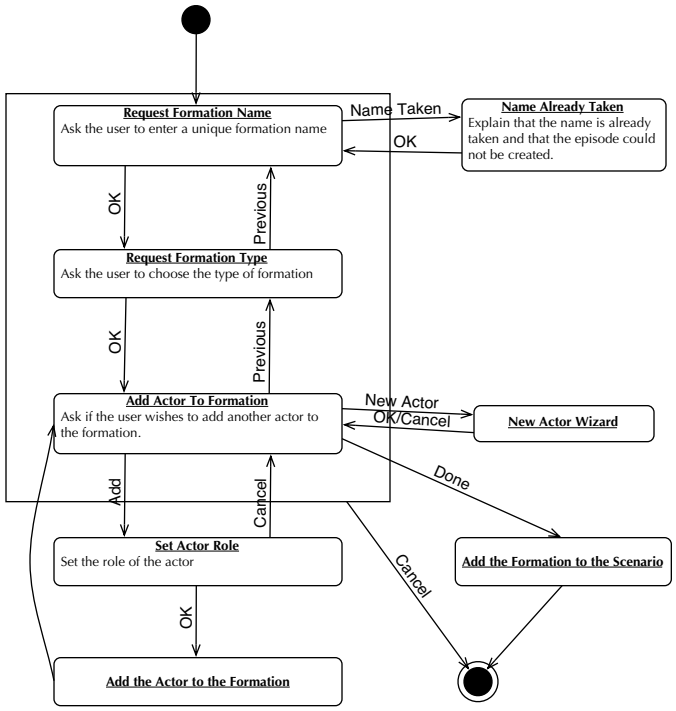

Figure 15: UC-35 Add a Formation Using the Wizard

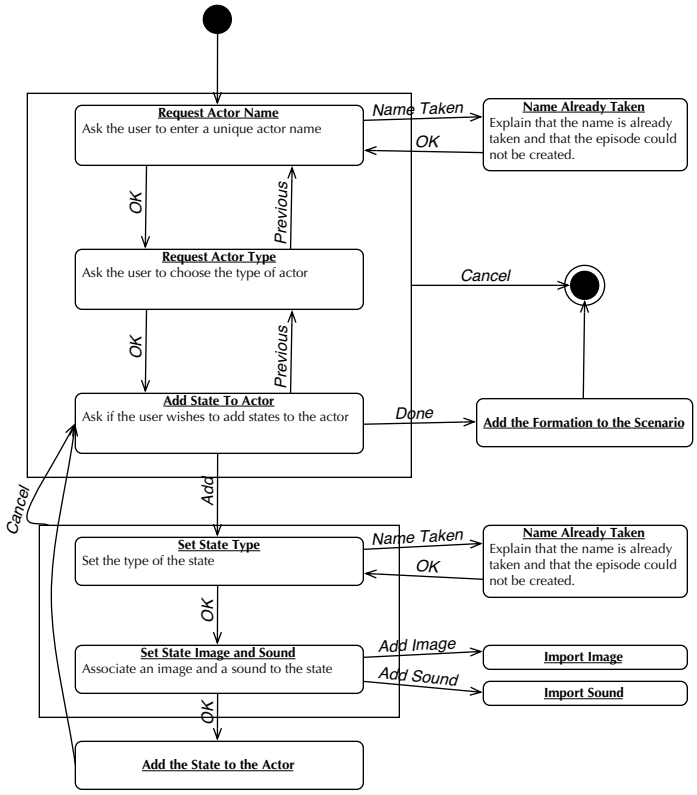

Figure 16: UC-20 Add an Actor Using the Wizard

12.4 Incremental Development Plan

The development will be splited in three parts. One team will work on the XML services while the other one will work on the implementation of the GUI. Both teams will work separately using a driver and a stub respectively. When the projects are mature enough, they will be liked together through an additional even-based layer.

Table 5 presents the incremental plan.

Increment ID	**Description**	
1	Implementation of the Validation and I/O XML functions	Implementation of the GUI primitives
2	Implementation the XML Node and XML check-out and check-in services	Implementation of the Dialogs
3	Implementation of the Managed Object Layer	Implementation of the Chapter Rendering
4	Implementation of the event callback support and replacing the stub and driver with actual modules.	

Table 5: Incremental Development Plan

Detailed Design

White Dwarf Map Editor

1 Module Implementation

1.1 Architecture Changes

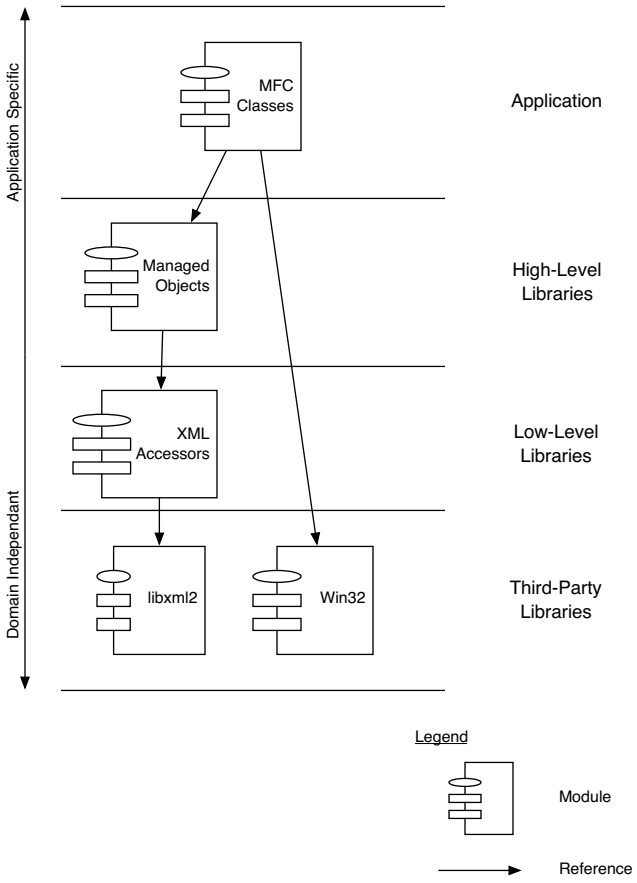

Figure 17: Revised Implementation View

If you compare Figure 17 with Figure 4 on page 13, you can see that several changes were made. The "GUI Components" have been replaced with MFC Classes, the "GUI Services", "Rendering Engine" and OpenGL modules were removed, and the MFC Classes use directly Win32.

Also, if you look at Figure 1 on page 8, the logical modules "Display Engine" and "Resource Manager" were removed.

Below is explained why those changes were made to the architecture during development.

1.1.1 GUI Components and Services

GUI Components to MFC Classes

GUI Components and Services would have required a complete cross-platform GUI wrapper, which is a new project by itself...

MFC forces us to greatly increase coupling between GUI elements and the "logic" they implement...

1.1.2 Resource Manager

Resource Manager (gone)

1.1.3 Rendering Engine and OpenGL

Rendering Engine will be done with the game itself...

1.2 Third-Party Modules

Some modules were already implemented by other developers.

The "XML Parser" module is libxml2[1], part of the open-source Gnome Project[2].

The modules "Event Handler", "Event Dispatcher" and "System Layer" are either part of Microsoft's Win32 library or the Microsoft Foundation Classes (MFC).

2 Design Notes

2.1 XML Accessors

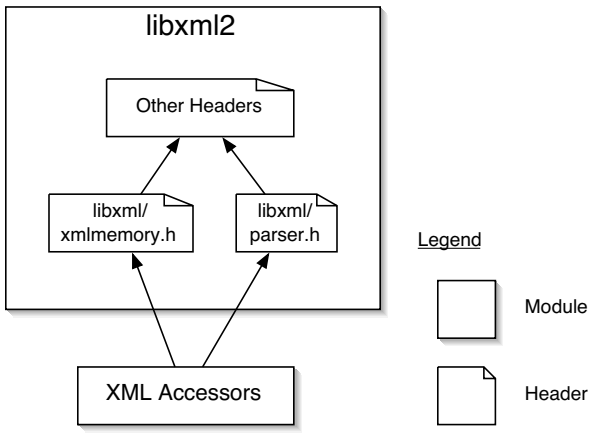

Figure 18: "XML Accessors" Module

The "XML Accessors" module is basically a facade to the third-party XML parser, generator and validator, here libxml2. This is required due for the following reasons:

- The XML library (libxml2) might have not fully supported all the features we required, thus we could have been forced to use more than one XML library at the same time.
- libxml2, while powerful, is often too complex and unintuitive to be used directly.
- For testing purposes, it is a good place to verify for errors and data corruption that otherwise would be written directly to the XML file.

[1] `http://www.xmlsoft.org/`
[2] `http://www.gnome.org/`

This module is the only one that has to and can maintain the in-memory copy of the XML data. But at the same time, it has to do so with possibly several distinct XML documents. As a result, all the context of the different functions is has to support has to come from the input data (the function's arguments).

As a result, it is possible to make "XML Accessors" stateless, that is simply a collection of free functions. Thus, for efficiency, this module was made in C. Being completely independant of the rest of the application, and being in C, we decided that it should be an external static library that will be used by the editor.

2.2 ManagedObjects

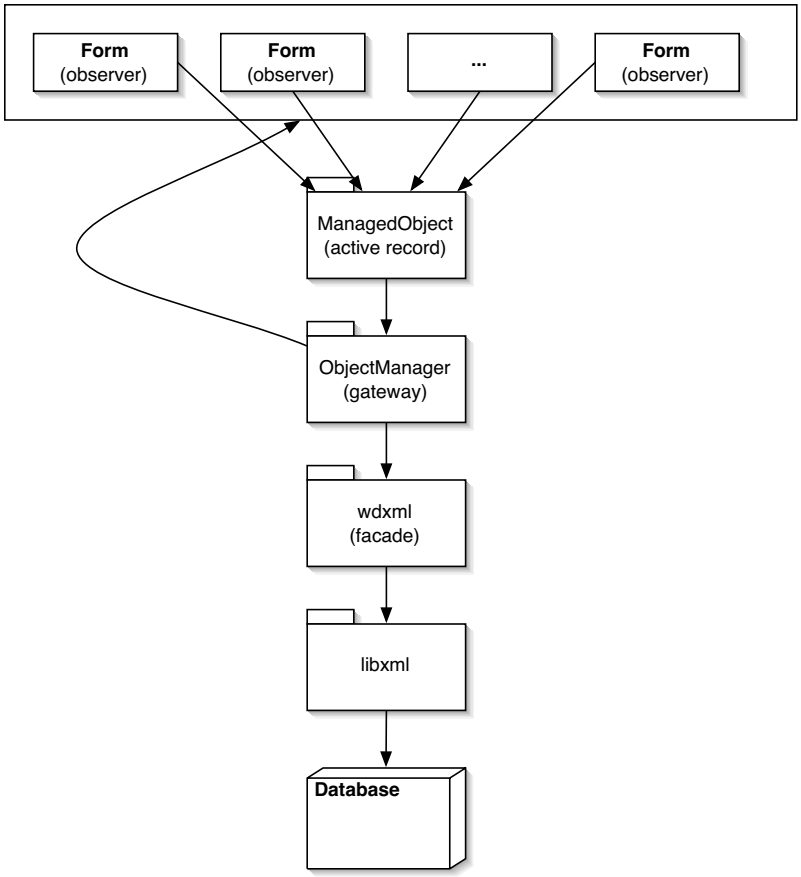

Figure 19: "ManagedObjects" Module

The ManagedObject module provides subscription and access services to the database. We needed to keep the data displayed on screen in sync with the database all the time, and since editing one object can modify an other one (for instance, changing the name of an Actor should update the corresponding ActorRefs in every Formations), we had two solution: polling the system or being notified by the system. Although the former was probably easier to implement, we chose to implement the latter, in part for the challenge, but mainly because it was considered more efficient.

The next question was to decide who would be notifying the observers. We did not want to couple the database directly to the observers, so we brought in the ObjectManager class. There is only one for a given database, and all access must go through it. Being aware of all the transactions, it is easy for the ObjectManager to notify the observers about the change.

Finally, we wanted to limit the coupling between the forms displayed on the screen and the ObjectManager (the forms shouldn't have to be aware of the database formatting and technicalities) so we introduced the ManagedObjects which would be used as active records, providing hidden access to the ObjectManager.

2.2.1 The EventHandlers

The event handlers are observers. They expect to be notified when the data they subscribed to has been modified. The only requirement for an object to be elligible as an observer is to derive from the IEventHandler interface.

This makes it easy to register a form from the user interface, or other objects wishing to remain in sync with the database.

2.2.2 The ManagedObjects

The ManagedObjects are used as mediators between the database and another object accessing or changing information. They behave as active records, resolving references between data tables and providing an abstraction from the actual database data format. ManagedObjects can be manupulated with the IManagedObject interface when the user does not need to know specific things about the object (for example, it's name, or the number of children), but can also use the specific interface (Actor, Formation, Episode, ...) to access functionalities specific to the object (Actor::setImage, FormationRef::setPosition, ...).

For convenience, we also designed the ObjectFactory, which can build ManagedObject of the correct type from an IManagedObject interface or a simple data node of unknown type.

2.2.3 The ObjectManager

The ObjectManager is a gateway for the database. Since the ObjectManager is aware of all the changes, it is also in charge of notifying the observers.

2.3 MFC Classes

The MFC Classes implement the forms the user can interact with. Most of the design is straight forward, since they are meant to provide a direct access to the ManagedObject's interface, it is interesting to note the design of the ObjectWindow and the Wizard.

2.3.1 The ObjectWindow

Throughout the system, very often the type of the managed object is unknown (it is used as an IManagedObject interface). Thus we designed the ObjectWindow, which behaves as a form factory. The ObjectWindow takes an IManagedObject and will find the form which should be used to display.

2.3.2 The Wizard

There are many wizards (Scenario, Episode, Formation, ...) but they all the same logic. Thus, using a state pattern, we designed a generic wizard which is initialized with it's initial state. The state knows which form to display and knows the previous and the next step. Thus, coupling between the steps is limited to the next and the previous step only, and the Wizard does not need to know what is being displayed.

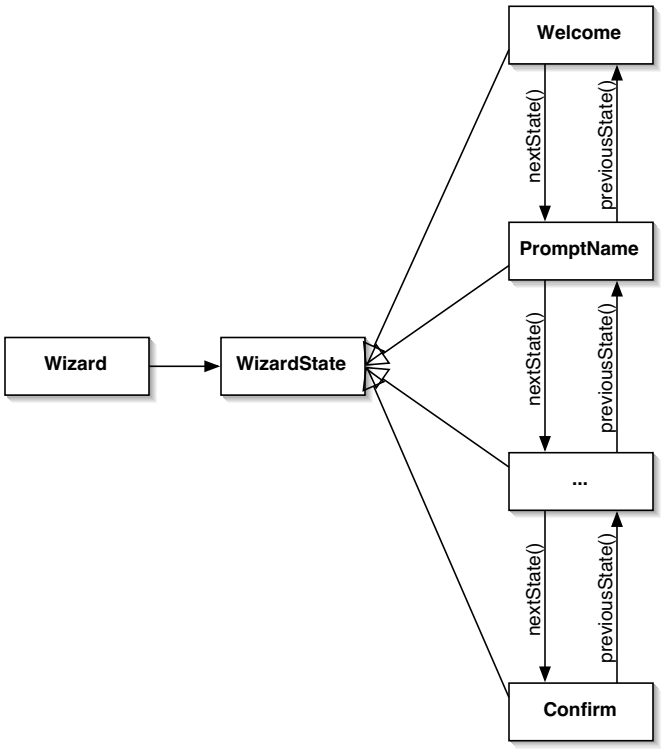

Figure 20: "Wizard" Module

3 Reference Manual

A complete low-level reference manual for the interfaces of the "XML Accessors" and "Managed Obejcts" modules is available on-line at `http://whitedwarf.sourceforge.net/refman`.

Testing

White Dwarf Map Editor

1 Test Plan

Here are the various test plans for the "XML Accessors" module, the "Managed Objects" module and the MFC Classes, as described in Section 1 on page 29.

1.1 XML Accessors

This is the most critical module we had to write. Any error this module can produce will destroy most of the functionalities of the editor.

Extreme programming was used to produce this module. The test code and source data were made and revised as the functions in the module were made. As a result, the testing was "clear box", since the tests were made knowing in advance the way the functions were implemented. Finally, code inspection was made by Benoit Nadeau and, for some functions, Gaspard Petit.

Note that this module is totally dependant on the libxml2 library. As a result, code inspection is not enough, since we had to test that libxml2 works as expected.

The functions in this module are extremely string on the input data, since any "strange" input, even if acceptable by the function, might be indicative that the higher-level modules are unstable. As a result, several assertions were added to reduce the time between the defect has an effect and when it is detected. The assertions are removed in the final, non-debug version of the editor when the other modules are properly tested.

1.2 Managed Objects

Similarly to the "XML Accessors" module, this module was tested with both extreme programming and clear-box testing. Also, code assertions were placed to detect defects as soon as possible.

The major difference between this module and "XML Accessors" concerning testing is that since this module requires at least 10 times more code than "XML Accessors", code inspection would have been too long. As a result, the defects have to be detected with the test code or during integration testing with the MFC Classes.

1.3 MFC Classes

At the beginning of the project, we wanted to test the User Interface with a automatic scripting system wich would reproduce the actions of a user. But since the "GUI Components" module was dropped and that the logic of the application were moved inside the MFC Classes, it was then almost impossible, and much too difficult, to produce any kind of UI scripting system.

As a result, the User Interface was tested simply by using the application, as some kind of black-box testing. The tests were made as GUI functionality was added to the system, which makes it so that there were so explicit "test code" or "test plan" for this module.

2 Testing Results and Quality Assessment

2.1 Process Used

At the beginning, we wanted to use the bug tracking system offered by sourceforge [3], but we did not do so since the amount of defects to track at any one time was always small. Also, because we are a small team, here is the process we used for development and correcting defects.

[3] `http://sourceforge.net/tracker/?group_id=45915&atid=444472`

1. The developer implements a small part of the module, as small as possible.
2. The developer tests the code on his own.
3. As long as defects are found, until the code "seem" to work correctly, the defects are removed by the same developer.
4. Only when the developer considers its own code "bug-free", you copy the code in the CVS repository.

Thus, we always assume that the code in the "main branch" of the CVS repository is relatively stable. It can happen that some defects are placed in the "main branch", but since our team is only of 4 people, and that we often "check in" our code, the person that made the error is quickly found and, usually, the error is quickly corrected.

This process results in a more "pass or fail" situation for the code, where having some code of poor quality in the "main branch" product is not acceptable. As a result, producing quality code was actually part of the process, and bug tracking for some "test phase" was not needed.

Obviously, if the project had been of much higher complexity, or if the team size had been at least 8 people, it would have been more difficult to quickly find and correct the defects, so a bug tracking system would have been useful. This is what might happen for the SOEN 490 project

2.2 Results

The results of the test code for the "XML Accessors" and "Managed Objects" modules can be verified by examining the XML documents produced by those modules and by validating, with the help of libxml2, the documents against the expected structure described in a DTD document.

The results of the tests made for the MFC Classes were seen "visually" by the tester as development was made.

The results were not logged, since we were in the situation described in Section 2.1. But the test code used is kept in a CVS repository[4] so that the other team members can quickly learn, by example, how the module works, and to re-run the same tests if the team member changed some code that is not his own.

[4]The WhiteDwarf/Tests repository in SourceForge.

