Theoretical and Experimental Application of Grid in Computer Games

Kristian Kjems

Kgs. Lyngby 2004
IMM-THESIS-2004-80

Technical University of Denmark
Informatics and Mathematical Modelling
Building 321, DK-2800 Lyngby, Denmark
Phone +45 45253351, Fax +45 45882673
reception@imm.dtu.dk
www.imm.dtu.dk

IMM-THESIS: ISSN 1601-233X

Abstract

This thesis examines how applicable grid services are as a foundation for a computer game server. The focus is primarily on server design for a *Mass Multiplayer Online Game* (MMOG). The task of running a MMOG server is so big it cannot be run on a single computer. Therefore the game server is divided into smaller sub-problems according to the *divide and conquer* principle. This project describes how a game server is divided into sub-problems, and how these sub-problems work together using grid services. It is concluded that inter-service communication performance is a problem, when used in a performance critical system like a game server. The problem can be overcome by using normal IP communication between components where performance is critical. Consequently grid services are best used for infrastructural purposes in a game server design.

Resume

Dette eksamensprojekt undersøger hvor anvendelige grid services er som et fundament for en computerspilsserver. Det primære fokus er på server design til et *Mass Multiplayer Online Game* (MMOG). En MMOG kræver for mange ressourcer til at den kan køres på en enkeltstående computer. Computerspilserveren er opdelt i mindre opgaver som *divide and conquer* princippet beskriver det. Dette projekt beskriver hvordan computerspilserveren er opdelt i mindre opgaver, og hvordan disse del-opgaver arbejder sammen ved brug a grid services. Det er konkluderet at kommunikationen med grid services forårsager et performace problem, når grid services bruges i et performanceafhængigt system som en computerspilsserver. Problemet kan løses ved at benytte normal IP kommunikation mellem del-opgaver i serverstrukturen der er performanceafhængig. Resultatet er at grid services bedst anvendes til infrastruktur i forbindelse med computerspilserver design.

CHAPTER 1 INTRODUCTION......11

1.1 INTRODUCTION TO GRID COMPUTING......12
Relation to web services......12
Generic calculations......12
Accounting and payment......13
Grid compared to electric power......13
Hosting agreements......13
Standardization......13
Platform for grids......14
Global research focus......14
1.2 RECENT DEVELOPMENTS IN COMPUTER GAMES......15
Singleplayer games......15
Multiplayer games......15
Evolution towards multiplayer games......15
Console multiplayer Games......16
Authenticity in computer games......16
Security in computer games......16
Design requirements for authenticity and security......17
Broadband and Mass multiplayer games......17
Development of Mass multiplayer online games......18
1.3 APPLICATION OF GRID IN COMPUTER GAMES......19
Assumptions......19
Hosting economy......19
Scalability......19
Simulation and AI calculations on the grid......20
Current solutions using grid: ButterFly.Net......20
1.4 PROJECT SPECIFICATION......22
1.5 TIME SCHEDULE......22
Phase plan......23
Overall Iteration plan......23
1.6 CHAPTER OVERVIEW......24

CHAPTER 2 DESCRIPTION OF TECHNOLOGIES USED IN THIS PROJECT......27

2.1 UNIFIED MODELING LANGUAGE......28
UML......28
2.2 DESIGN PATTERNS......30
Strategy Pattern......31
Memento Pattern......31
Observer Pattern......32
Proxy Pattern......33
Singleton Pattern......34
Transformation Pattern......35
2.3 GRID SERVICES USING WSRF......39
Technical documents this project is based on......40
Web Services Fundamentals......40
Web Service example: SimpleMath......41
Web Services as a Foundation for WSRF......42
Implied Resource Pattern......43
Endpoint Reference Type......43
Lifetime of Web Service Resources......44
Notifications in WSRF......44
2.4 NUNIT......45
Examples......45
2.5 SUMMARY......47

CHAPTER 3 DESIGN PRINCIPLES OF COMPUTER GAME AND GRID SOFTWARE 49

3.1 GRID DESIGN 50
Grid software design elements *50*
Grid application start-up..*51*
Grid service components *51*
Grid component interaction *52*
3.2 COMPUTER GAME ELEMENTS 53
Game Components *53*
Game Objects and states *53*
Distributing Object states...*54*
Relevant Set *54*
3.3 SUMMARY..56

CHAPTER 4 CASE STUDY: APPLICATION OF GRID IN COMPUTER GAMES 57

4.1 SYSTEM DESCRIPTION 58
Definition of game object in this case study *58*
Simulation of the virtual world..*58*
Game client *59*
4.2 ANALYSIS 60
Benefits of using grids in a game serve *60*
Problems with using grids in game server *61*
Scalability in Massively Multiplayer Online Games *61*
Security in Massively Multiplayer Online Games *61*
The game server fundamental design idea *63*
Database Cache *65*
Zone Simulation...*65*
Player Agent...*65*
Zone Simulation and Player Agent interaction in relation to relevant set algorithm.................*65*
4.3 DESIGN AND IMPLEMENTATION...67
Game Object...*68*
Game Actions *74*
XMLFactory *76*
Zone Simulation...*78*
Network Communication *82*
Player Agent...*87*
Test game client..*93*
Service Checker..*95*
Grid Services *96*
ZoneSimulationWorker...*98*
PlayerAgentWorker *99*
PlayerAgentManager and ZoneSimulationManager..*100*
GridInGamesManager and GridInGamesConnector..*101*
Code Documentation..*103*
4.4 TEST..106
Introduction..*106*
Unit Testing..*106*
Performance Testing *109*
Functionality Testing...*111*
4.5 SUMMARY..112

CHAPTER 5 CONCLUSION 113

5.1 A SHORT SUMMARY 114
Chapter 1: Introduction...*114*
Chapter 2: Descriptions of technologies used in this project....................................*114*
Chapter 3: Design Principles...*114*

Chapter 4: Case study: Application of grid in computer games 114
5.2 CONCLUDING REMARKS 115
5.3 FUTURE RESEARCH AND DEVELOPMENT 117

CHAPTER 6 BIBLIOGRAPHY 119

Chapter 1

Introduction

This section will consist of six parts. The first part will be a non-technical introduction to grid computing. The second part will be an introduction to computer games, focusing on the problems with hosting and development of multiplayer games. The third part will be about application of grid in computer games, and how grids may solve some of the problems discussed in the second part of this introduction. The fourth part will describe the purpose of this thesis. The fifth part will be a time schedule for the entire thesis. The sixth part will be a chapter overview.

1.1 Introduction to Grid Computing
1.2 Recent developments in computer games
1.3 Application of grid in computer games
1.4 Project specification
1.5 Time schedule
1.6 Chapter overview

1.1 Introduction to Grid Computing

This section is a non-technical introduction to grid computing.

Grid computing is a promising and fast growing technology, which without a doubt will be an important part of the future information technology. The general goal of grid computing, is to enable diverse resources to be brought together to form a united system. The goal is accomplished by encapsulating resources to conform to a general interface. The encapsulation is what makes each resource into a grid service. New grid services can be constructed using a variety of other grid services.

Relation to web services

There are different implementations of the grid service principle of encapsulating resources. The most common implementations are based on web service technology. Web services can loosely be defined as a description of how messages are exchanged over a network. Web services typically use the http protocol as media for message transfers. A web service is unable to store information making it stateless. In many applications, state is important making it much less desirable to use web services. Grid services add a resource to a web service. The resource can be elements in a database making the otherwise stateless web service able to contain a state. Grid services are more than just an added state, but the state is one of the most important additions to web services.

Generic calculations

There is a large variety of possibilities of grid computing, but the primary focus will be on grids that provide calculation power and storage for use in computer games. In order to increase the amount of calculation power several computers work together and it will examined how grids can be used to manage these computers. Such a set of computers is called a grid cluster. Grid clusters are able to host arbitrary many different grid applications. That means two different companies can run their grid applications/services on the same grid cluster.

Accounting and payment

The grid has an architecture that makes it possible to make accounting on resource use. It is therefore possible for the provider of a grid cluster, to bill the user in accordance to the used resources. The grid architecture provides access control to grid services. Some services can be accessed by all users, while other services are only accessible to a selected group.

Grid compared to electric power

Grid computing has many similarities with electric power, and can in many cases be thought of as the same. The only difference is that in grid computing we want computational power and in electric systems, we want electric power. The consumer has a socket in his wall allowing him to draw power from the net. The provider of power can make statistical prediction on how much power his consumers are going to use in the future. The consumer pays for the amount of power he is using. The provider can buy power from other providers, if he at some point is lacking power to satisfy all his consumers.

Hosting agreements

Because of the standardization, a grid application can be hosted by any grid provider that uses the same standard. This implies a loose coupling between the company that wants a grid application published and the company that hosts the grid application. This form of loose coupling is very desirable in a dynamic business world, where company policy and pricing can change and companies can go bankruptcy. It is an important feature for the consumer of a grid cluster that it will be easy to change from one grid provider to another.

Standardization

For grid services to be able to talk to each other, a general standardization is required. The most common used standard today is the Open Grid Services Architecture (OGSA) and Open Grid Services Infrastructure (OGSI). OGSA defines protocols and formats, to enable grid services to interoperate in a large-scale format. OGSI defines how the OGSA should be implemented using current technology. The latest invention or emerging standard in the world of grid and web services is the (Web Service Reference Framework) WSRF. The WSRF is based on the latest developments in web service technology. The WSRF is developed using the knowledge learned from the development of OGSA/OGSI.

Platform for grids

Grid technology is still a very young, and rapid developments are seen everywhere. The most mature implementation of the OGSI standard is called Globus Toolkit. Globus Toolkit works on any Linux platform. An OGSI implementation for the windows platform is being worked on in two active projects. The projects are called OGSI.NET [OGSI.NET] and MS .NET grid [MS.NET]. The WSRF is still only implemented in one project called WSRF.NET. The first technology preview of WSRF.NET was published late June 2004. Just as web services, grid services can communicate across platforms, and programming languages, which makes them very versatile. The only limitation for a grid service to run on a certain platform is the platforms ability to execute the grid service as a process.

Global research focus

The worldwide focus on grid computing is considerable. An example is the global e-Science collaboration that has founded £236M in the period from 2001 to 2006 on grid computing research [e-Science]. The focus on grid computing in Denmark increased when the Danish Center for Grid Computing [DCGC] was opened in November 2003.

1.2 Recent developments in computer games

This section will describe some fundamentals in computer games. The focus will be on the problems with hosting and developing multiplayer games.

The industry of creating and publishing computer games has been growing fast. Sales of both singleplayer games and multiplayer games are increasing. A multiplayer game is by definition played by multiple players simultaneously, while singleplayer games are played by a single player.

Singleplayer games

Singleplayer games are played with only one interacting human being called the player of the game. The player interacts with the game for entertainment. Most modern computer games present a virtual world, where the player acts as some important identity solving different tasks. Computer games can be grouped into genres like movies. The grouping of computer games into genres makes it easier to talk about and understand. A brief description of the different game genres can be found in [Games-Genres].

Multiplayer games

Multiplayer games are typically played over the internet by multiple players simultaneously. The two most common technologies used to connect players in multiplayer games are client-server. The client-server technology lets the server keep track of all the game related information. The server sends all relevant game information to each of its clients. It is a demanding task to perform the job of a server, so it is usually implied that the server cannot perform the job of a client as well. This leaves a problem of who should host game server.

Evolution towards multiplayer games

Multiplayer games are becoming increasingly more popular. The interaction with other humans through a virtual reality is a lot more challenging and satisfying than most singleplayer games. Most multiplayer games are competitive making individuals or teams fight against each other.

Console multiplayer Games

A console is a special computer designed for computer games. Most consoles are connected to the television. The most common consoles are: Playstation (PS), Playstation2 (PS2), XBOX and GameCube.

Console games have just begun to support network games over the internet. The fact that most consoles are now able to play multiplayer games will generate a huge demand for console multiplayer games. Sony has released their PS2 Online, and Microsoft has released their Xbox Live. Xbox Live provides network infrastructure and online customer service for the publisher/developer [Consoles]. That means developers have to provide server code, and then Xbox live will run the server code on their servers. The Xbox live system does however have some limitations in flexibility. Xbox Live charges a monthly fee from its customers, while PS2 Online is free. PS2 Online does not provide network infrastructure, so the publisher/ developer has to host and maintain their games themselves. Developing server code for Xbox Live and PS2 Online is different, so a lot of time is used on developing the same thing on two platforms.

Authenticity in computer games

Authenticity in computer games is provided by a validation mechanism between the virtual and the real world. In most cases, a player has to use a login and password to gain access to the virtual world. Authenticity can be used to enable all kinds of game related features like: Player ranking, character development, penalties for cheating. Player ranking makes a game more competitive as players can compare themselves to each other. Character development makes a stronger connection between the virtual character and the human player. With authentication, it is possible to punish a player if he is caught cheating or misbehaving in other ways.

Security in computer games

The security in computer games is made so cheating will be as hard as possible. Cheating in singleplayer games, only affect the cheating player, so no need to worry much about that. The cheating in multiplayer games affects other players making security very important. The concept behind many multiplayer games is the competition between players. Cheating in any form of competition makes participation a lot less fun. The sad truth of people playing computer games is that some will cheat if it is possible. If the players host the game servers, it is almost impossible to prevent cheating, because the players will have access to all game data and communication. If a trusted party provides the players with all the game servers, it is much less likely that they will be able to cheat.

Design requirements for authenticity and security

Authenticity and high security have some requirements to the hosting environment, if they are to work in an appropriate way. The game server has to be hosted by a trusted party and an authentication service like a login has to be provided.

Broadband and Mass multiplayer games

The online game experience can quickly become an expensive one, if connection to the internet is made with a modem charged by the minute. Broadband networking enables anyone to gain relatively fast and cheap access to the internet for a fixed monthly cost. A large penetration of broadband in a game market is an important factor for the success of online games. The penetration of broadband on the Korean market has been very good, with approximately 70% of all households having broadband [Korea-BB]. The broadband market growth has had a dramatic impact on the popularity of online games in Korea. Especially Mass Multiplayer Online Role Playing games (MMORPG) [Games-Genres] have been successful. The MMORPG Linage [Linage1],[Linage2] is played by almost 2 million Koreans, which is 4% of the total population. Linage alone is generating an impressive $100 million in annual sales.

Development of Mass multiplayer online games

Game research indicates that MMOG has the potential to make good revenue for the game developer and publisher [Online-Gaming1], [Online-Gaming2]. However for a MMOG to be successful a lot of technical obstacles have to be overcome. Many developers have attempted to make an economic successful MMOG game, but have barely had their expenses covered. The most common genre of MMOG today is Role Playing Game (RPG) [Game-Genres], and is called MMORPG. Developing a MMORPG is one of the most challenging tasks a game developer team can take. One of the problematic technical elements of a MMORPG is the network infrastructure. The network infrastructure has to be able to host large games with thousands of interacting players, making it necessary that several computers work together. It can be very expensive for developers/publisher to buy and maintain a network infrastructure that is required to host a MMORPG.

1.3 Application of grid in computer games

In this chapter, it will be describe how grid can be used to solve some of the problems with multiplayer games described in the previous chapter. The application of grid in computer games also grants new possibilities, which will also be discussed.

Assumptions

Grid technology is well suited for hosting computer games, but requires the availability of a grid cluster. The availability of grid clusters today (Q4 2004) is very low. In this section, it will be assumed that the availability of grid clusters is good. The grid technology is still very young, and might not yet be suited for game server development. In this section it is assumed that the grid technology is matured making it suitable for MMOG servers.

Hosting economy

Under normal circumstances, publishing a computer game with trusted servers will require a lot of investing in equipment. The publisher has to buy a computer park able to host the game when a maximum number of players are playing the game. It is hard to determine exactly how many are going to play a certain game, so the publisher has to buy a little more computer power than necessary. Both the initial investment in a computer park and the following maintenance will be expensive and inefficient.

Grid technology is able to reduce these expenses. The publisher makes an agreement, with a grid hosting company already owning a grid cluster able to host the game. The expenses from hosting a game using a grid cluster will be substantially lower than the traditional methods.

Scalability

Grid technology is a tool that can help developers make server code scalable more easily. Grid technology can only help to provide scalability with regard to hosting the game. Other scalability restrictions may apply, as for instance the graphics engine. Most engines will not be able to show 1000 players on the screen at the same time on current game platforms. Another restriction is the network bandwidth. If a player **A** is in a near vicinity of other players **B**, **C** and **D,** it is required that the positional information of the players **B, C** and **D** are sent to player **A**. If player **A** is in a near vicinity of 1000 other players, the information required to update the positional information of all these players would exceed the bandwidth of most users. This leaves a problem not solvable by grid computing.

Simulation and AI calculations on the grid

Hosting a game on a grid cluster, makes it easier to do all kinds of heavy game related calculations on the grid. Some restrictions or requirements apply to the calculations that can be made on the grid. The amount of data sent to each client from a grid-based simulation, is limited by the speed of the client's internet connection. Suitable calculations will not require large amount of data transferred between the clients and the grid. The time it takes for a client to make the calculation locally should in general be longer than the time it takes to send it to the grid, and get a result back. In some cases a lot of game related data is stored on the grid running the game. In these cases, clients may not be able to perform the calculations themselves, because they lack the necessary information to do so. An example of this attribute is the shortest path problem where the task is to find the shortest path between to points while not colliding with any obstacles. Only the server know where all the obstacles are positioned, and is for that reason the only one able to perform the shortest path calculation. The amount of data sent between the grid and the client is small. The client sends two points A and B to the grid and the grid replies with a list of points representing the shortest path between A and B.

Current solutions using grid: ButterFly.Net

A commercial solution based on OGSA optimized for computer games does exist under the name of Butterfly.Net [ButterFly]. Butterfly.Net provides both a development framework and a hosting solution for a grid-based solution. There currently exists no information on how they use grid technology in their server solution.

1.4 Project specification

A. Grid computing has to be explained. OGSA, OGSI and WSRF must be presented. The Unified Process (UP), Unified Modeling Language (UML) and design patterns must be described.

B. A framework for a multiplayer game server has to be developed. The framework must be constructed as a grid application where scalability is important.

C. A test application able to simulate a simple running computer game must be created. The test application must use the framework from B.

D. Various tests of the running game server simulation of a running computer game must be made using the test application from C. The tests should help clarify if grid technology is applicable in a computer game server.

E. It must be examined how applicable the Unified Modeling Language and design patterns are in the development of grid applications.

1.5 Time schedule

The project will be divided into several iterations divided by a set of milestones. The end of each of the iterations defines a new milestone. The time schedule defines a set of dates to which the milestones must finished. The creation of the time schedule is divided into two parts. A phase plan and an overall iteration plan. The phase plan is a macro level plan where milestones dates and objectives are defined. The overall iteration plan is defined using eight categories, how the workload should be distributed in all iterations. This form of development plan is closely related to the time schedule defined in the Unified Process [UML-Larman].

Phase plan

The purpose of the phase plan is to define the dates and roughly describe the goals for each milestone. The phase plan will remain consistent and unchanged throughout the project period.

Iteration 1: March 2004	Start-up phase
Iteration 2: April 2004	Technology preview
Iteration 3: May 2004	Analysis and design
Iteration 4: June 2004	Design and implementation
Iteration 5: July 2004	Design and implementation
Iteration 6: August 2004	Implementation and Test
Iteration 7: September 2004	Test and Documentation

Overall Iteration plan

The iteration plan below, show a rough distribution of the workload split between eight categories. The workload from each of the categories, sum up to 100% over the seven iterations.

Iteration:	**1**	**2**	**3**	**4**	**5**	**6**	**7**
Project management	50%	10%	10%	10%	10%	5%	5%
Requirements	25%	75%	0%	0%	0%	0%	0%
Analysis	0%	0%	75%	15%	10%	0%	0%
Architecture	0%	0%	70%	10%	10%	10%	0%
Design	0%	0%	30%	40%	30%	0%	0%
Documentation	5%	10%	5%	5%	5%	30%	40%
Implementation	0%	0%	0%	20%	20%	50%	10%
Test	0%	0%	0%	0%	0%	20%	80%

1.6 Chapter overview

Chapter 1: Introduction

The chapter introduces fundamental concepts of grid technology and computer games. Some of the problems with hosting and developing a computer game server are described. It is suggested how grid technology can be used to solve some the problems with the hosting and development of a computer game server.

Chapter 2: Description of technologies used in this project

This chapter describes the technologies used in this thesis. The following will be described:

- Unified Modeling Language (UML)
- The Open Grid Service Architecture (OGSA)
- Open Grid Service Infrastructure (OGSI)
- The Web Service Reference Framework (WSRF)
- NUnit framework for unit tests

The chapter will also contain a presentation of the design patterns used in this project [GoF].

Chapter 3: Design principles of computer games and grid software

This chapter will describe some of the general and fundamental design principles and algorithms used in this project. This includes a grid application design principle called the Manager-Worker design. The Manager-Worker design is made to solve problems using the *divide and conquer* strategy. A game server specific algorithm used for distributing game object state will also be described.

Chapter 4: Case Study: Application of grid in computer games

This chapter will describe how the technologies from chapter 2 combined with the design principles of chapter 3 can be used to solve some of the problems with computer game servers described in chapter 1.

The chapter will describe how the framework and test application is implemented. Key elements from the source-code will be examined. This chapter will also contain both unit tests and performance tests. The test application will be tested according to the project specification.

Chapter 5: Conclusion

A short resume of the chapter summaries will be made. The project specification will be examined to outline how each of the requirements is met. Future work and improvements on this project will be discussed.

Chapter 2

Description of technologies used in this project

This chapter describes the technologies used in this thesis. The following will be described:

- Unified Modeling Language (UML)
- The Open Grid Service Architecture (OGSA)
- Open Grid Service Infrastructure (OGSI)
- The Web Service Reference Framework (WSRF)
- NUnit framework for unit tests

The chapter will also contain a presentation of the design patterns used in this project [GoF].

2.1 Unified Modeling Language
2.2 Design Patterns
2.3 Grid services using WSRF
2.4 NUnit
2.5 Summary

2.1 Unified Modeling Language

The Unified Modeling Language (UML) is a notation that can be used to describe structure, state and behavior in software. UML is a raw notation that can be used as a tool when designing object oriented software. The Unified Process (UP) describes a set of activities done to transform a set of requirements into a final product. Neither UML or UP guarantee a good object oriented result when making software, but if applied correctly they can be helpful tools for achieving the goal. The UML is a huge language containing more notation than most people will ever need when designing object oriented software. It is practical to choose a subset of UML that suites the project at hand. Not all the activities described in the UP are relevant to each project, so it is also practical to choose a subset of the UP to use. The subset of UP was described in *1.5 Time schedule*. The subset of UML used in this thesis will be introduced below.

UML

The subset of UML used in this thesis will be sequence diagrams and class diagrams. For more information about UML see [UML-Larman]. The diagrams will be created in Borland Together 6.1. Two simple examples of a sequence and class diagram can be seen on Figure 2-1 and Figure 2-2

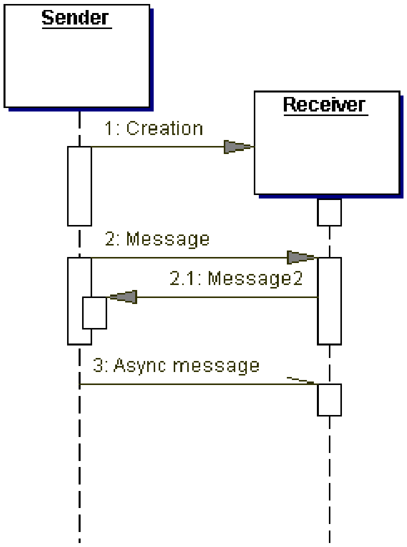


Figure 2-1, Sequence diagram

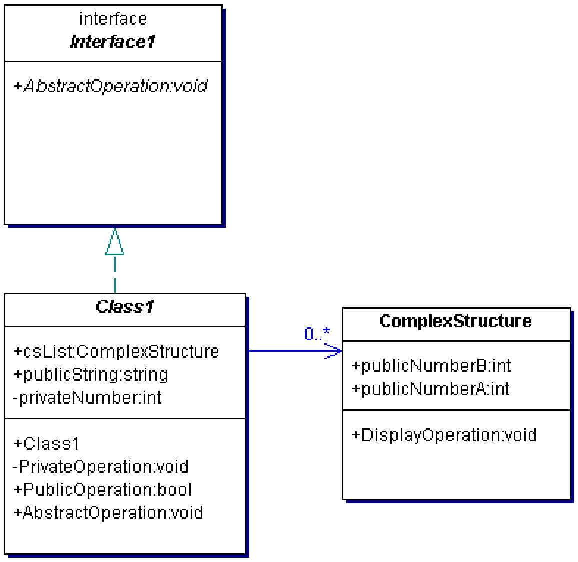


Figure 2-2, Class diagram

2.2 Design Patterns

Design patterns are a very important tool in modern software development. The "bible" of design patterns "Design Patterns, Elements of Reusable Object-Oriented Software" written by Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides [GoF]. The four authors of the book are also known as the Gang of Four.

A design pattern is said to be a solution to a problem in a context. A design pattern can be used repeatedly in the same context, but with different problems. Design patterns are thoroughly tested design components constructed by experienced developers as a good practice for less experienced software designers. Design patterns are an abstraction of the details otherwise necessary to describe a solution to a problem in a context. This abstraction makes it easier to understand and discuss otherwise complex software designs.

In software development, design patterns are divided into 3 groups:

- Creational Patterns
- Structural Patterns
- Behavioral Patterns

The design patterns used in this thesis, will shortly be presented in this chapter. For a thorough description of the design patterns, see the reference [GoF]. There will be presented one new design pattern called the Transformation Pattern, and it has been developed during this project. It would almost be blasphemy to imply that this "home made" pattern can match it self against the patterns presented in [GoF], but it does have its application in this project. The patterns used in this project are:

- Strategy
- Memento
- Observer
- Proxy
- Singleton
- Transformation

Strategy Pattern

The strategy pattern is a behavioral pattern. It defines a common interface for a family of strategies, letting the context be able to exchange them easily.

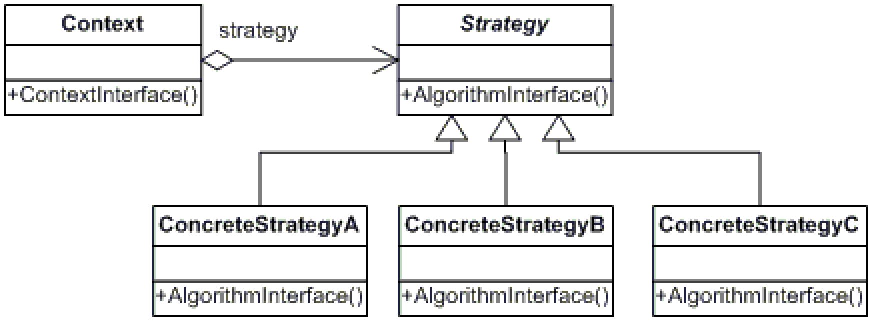


Figure 2-3, Strategy pattern class diagram

The class diagram on the pattern can be seen on Figure 2-3. The strategy pattern is typically used when something can be done in many different ways, and it should be easy to change how they are done.

Memento Pattern

The memento pattern is a behavioral pattern. The pattern makes it possible to extract internal state of an object without violating the encapsulation. The object can be restored to a previous state by inserting a memento extracted earlier.

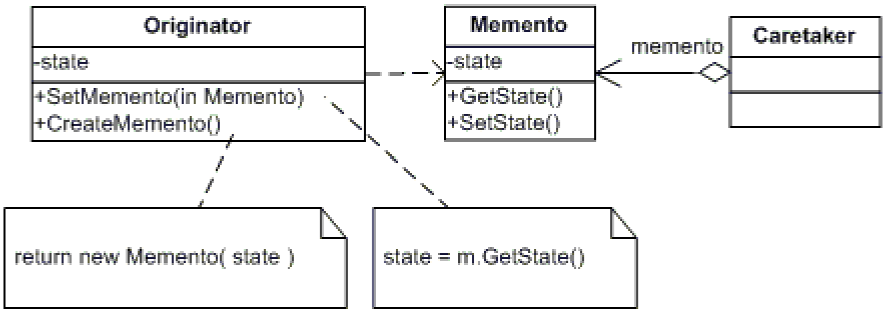


Figure 2-4, Memento pattern class diagram

The class diagram of the memento pattern can be seen on Figure 2-4. In a distributed system, the pattern can also be used to keep two objects synchronized. The internal state can be extracted from a source object, send over the network and inserted into the target object thus keeping the two objects in same internal state.

Observer Pattern

The observer pattern is a behavioral pattern, which notifies objects that have subscribed to be notified when changes occur to a source object. The source object is called the subject.

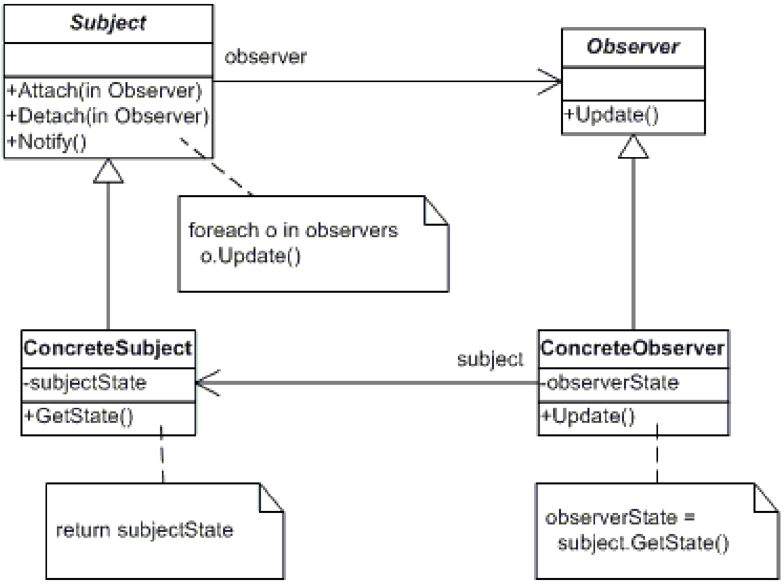


Figure 2-5, Observer class diagram

The class diagram on Figure 2-5 show the structure of the pattern, and the naming used for the different structural parts.

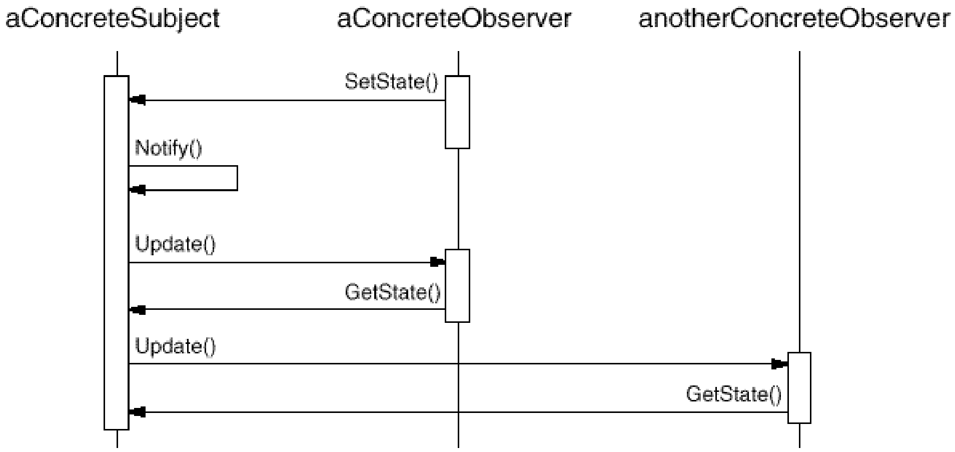


Figure 2-6, Observer sequence diagram

Figure 2-6 shows the sequence diagram of the observer pattern in function. When a change happens to the ConcreteSubject, the Notify method is invoked. The Notify method calls the Update method in all the subscribed observers. All of the observers use the GetState to retrieve the new updated state from the subject.

Proxy Pattern

The Proxy Pattern is a structural pattern. It enables the user to hide or protect the Client from the details of accessing a subject, by providing a placeholder or proxy to the subject.

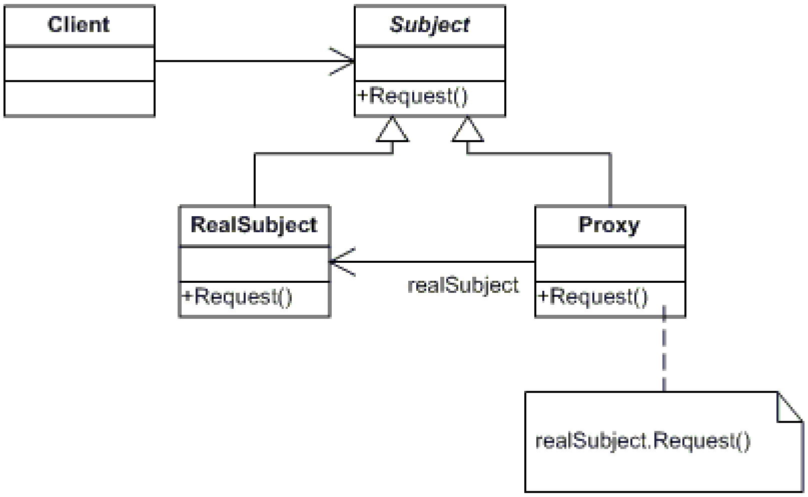


Figure 2-7, Proxy pattern class diagram

The Figure 2-7 shows the class diagram of the pattern containing the naming of the structural elements.

Singleton Pattern

The Singleton pattern is a creational pattern. It ensures that only one instance of an object exist, and provides global access to it.

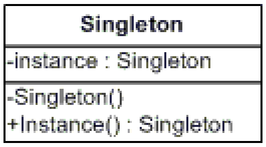


Figure 2-8, Singleton class diagram

Figure 2-8 shows the simple class diagram of the singleton pattern. The constructor is private, and access to the object is achieved using the static method Instance. The object is created the first time the Instance method is called and subsequent calls to the Instance method returns the original object.

Transformation Pattern

The transformation pattern is homemade and not tested or documented elsewhere. The transformation pattern is a structural pattern.

Intent:

Change an interface to another granting access to different part of a class. Transformation pattern hides part of a class interface, which given a certain internal state, is undesirable that clients get access to.

Motivation:

Sometimes classes give access to methods that should not be called unless some internal state constraint is fulfilled. The following simple example illustrates how this can be a problem for a class used to send data over the network. The class for sending data over the network is called UDPSend and has a default constructor. The class has two methods: Open and Send. Open is used for opening a connection to a remote peer, and Send is used to send data to the connected peer. It is problematic that is possible to call the Send method before being connected to the remote peer, thus causing a fault at execution-time. The objective of the transformation pattern is to move this fault to compile-time by making a compile error if Send is called before a connection is established.

The transformation pattern uses interfaces or pure abstract classes to expose certain parts of an underlying object. Look at Figure 2-9, where the transformation pattern has been applied to the UDPSend class.

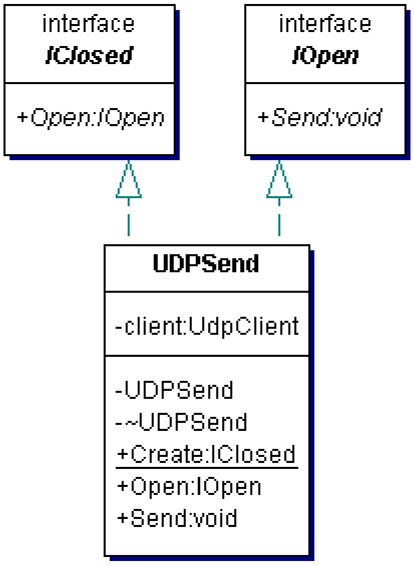


Figure 2-9, Class diagram over transformation pattern

The constructor of UDPSend is private and there is no way to gain access to the object directly. UDPSend has a static method called Create that creates a new UDPSend object but returns it as an IClosed interface. This way only the subset of methods defined in the IClosed interface is available after creation. See Figure 2-10 for the UDPSend source. IClosed supports one method called Open that takes an IP-address and port number as argument and returns the UDPSend object as an IOpen interface. IOpen gives access to the Send method that sends a string given as argument to the connected remote peer.

```
namespace TransformationPattern
{
  public interface IOpen
  {
    void Send(string s);
  }
  public interface IClosed
  {
    IOpen Open(string address, int port);
  }

  public class UDPSend : IOpen, IClosed
  {
    private UdpClient client;
    private UDPSend(){}
    ~UDPSend()
    {
      client.Close();
    }
    public static IClosed Create()
    {
      return new UDPSend();
    }
    public IOpen Open(string address, int port)
    {
      client = new UdpClient(address, port);
      return this;
    }
    public void Send(string s)
    {
      byte[] bytes = Encoding.ASCII.GetBytes(s);
      client.Send(bytes, bytes.Length);
    }
  }
}
```

Figure 2-10, Example code of the transformation pattern

The source on Figure 2-11 shows what is possible and what gives a compile error. The lines of code starting with "//" is commented out and would cause a compilation error.

```
// UDPSend udpSend = new UDPSend();
// UDPSend udpSend = UDPSend.Create();
// IOpen openConnection = UDPSend.Create();
IClosed closedConnection = UDPSend.Create();
// closedConnection.Send("Hello World");
IOpen openConnection = closedConnection.Open("127.0.0.1", 9999);
openConnection.Send("Hello World!");
```

Figure 2-11, Application of the UDPSend class

Application:

Use the transformation pattern when:

- You only want to expose methods in a class that is reasonable to call given a certain internal state of the class.
- You want to make sure that improper sequence of method invocations on a class yields an error at compilation rather than under execution.

Structure:

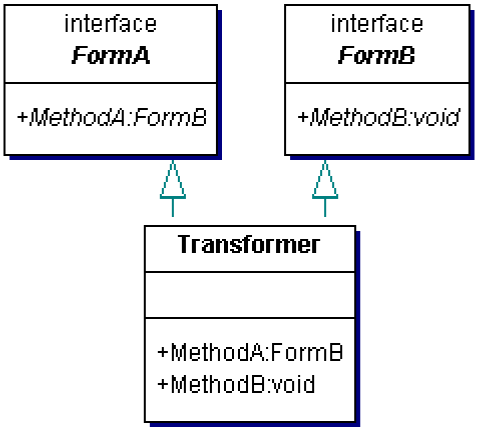


Figure 2-12, Transformation structure

Participants:

- Transformer (UDPSend)

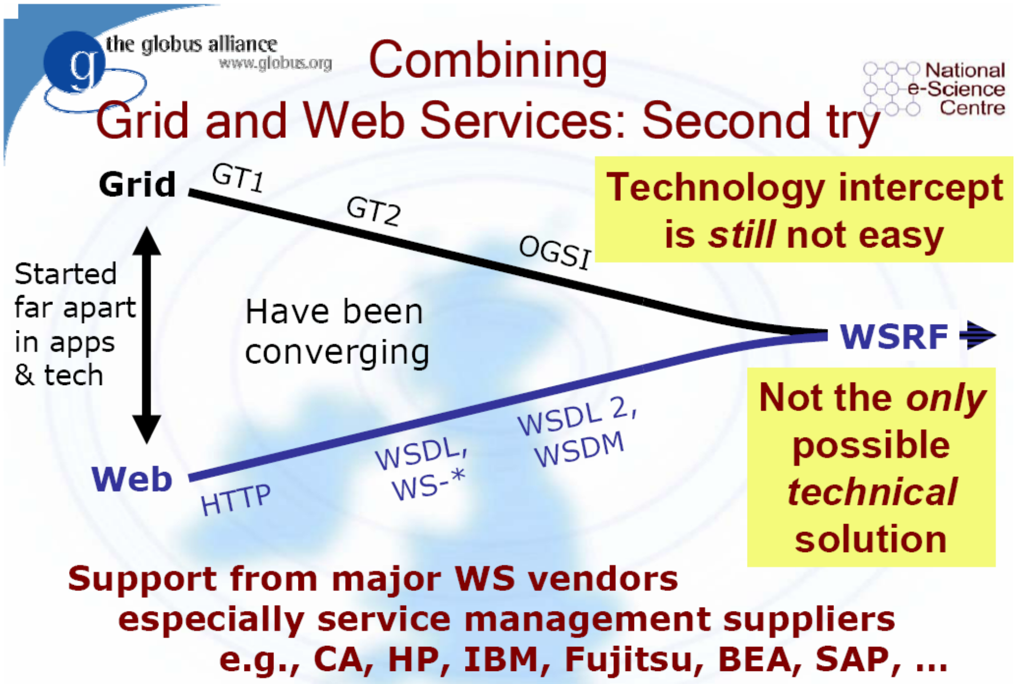
  - The Transformer uses the Forms to show itself, and never reveals its true form.

- Form (IOpen, IClosed)

  - How the Transformer is viewed by outsiders.

2.3 Grid services using WSRF

Grid technology is specified and implemented in various ways. The implementation used in this project is the Web Service Reference Framework WSRF. The WSRF is an evolution from the Open Grid Service Architecture OGSA [OGSA-Spec] and Open Grid Service Infrastructure OGSI[OGSI-Spec]. The following figure depicts how the original grid technology is merging with the web technology. The two technologies started far apart, but have now merged together in WSRF:



Combining
Grid and Web Services: Second try

Technology intercept is *still* not easy

Not the *only* possible *technical* solution

Support from major WS vendors
especially service management suppliers
e.g., CA, HP, IBM, Fujitsu, BEA, SAP, …

Figure 2-13, Merging web and grid technologies [GT-Presentation]

Because WSRF is a result of two merging technologies it is both a web service and a grid service.

Grid technology and web technology has merged in the work on the Web Service Reference Framework WSRF. The term "grid service" and "WS-Resource" denote the same. Both terms will be used in this chapter so remember the relation.

Technical documents this project is based on

There is a series of important documents describing the technology used in this project. OGSA and OGSI is respectively described in [OGSA-Spec] and [OGSI-Spec]. WSRF is described in [WSRF-Spec] that contains four documents, but only WS-ResourceProperties and WS-ResourceLifetime will be used in this project. The four documents are a refactoring of the design principles of the [OGSI-Spec] using newer web service standards. The four specifications can be used independently making solutions that are more lightweight possible.

Web Services Fundamentals

Web service is a new software development technology that abstracts the way software can be constructed. Web services are stateless programmable constructs, in the sense that they are only able to hold session based data. Web services are meant to make the abstraction of how things are done, and where they are done. Web services are black-box constructs, so the user does not know nor care how the job is done. The only thing that defines the web service is the messages generated and accepted. Web services communicate by standard high level protocols like HTML and XML. Any host able to send messages in XML and HTML over the internet is a potential host for web services. A web service is described in a language called Web Services Description Language (WSDL). A WSDL document describes a set of messages, encodings and protocols needed to communicate with the service. When web services are deployed they are also typically published into a registry, so potential users can easily find it. Users search the registry for a suitable web service according to the description of the web service in the registry. After the web service is found a binding is made between the client and web service, as described in the WSDL document. See Figure 2-14 for a state chart that describes this principle.

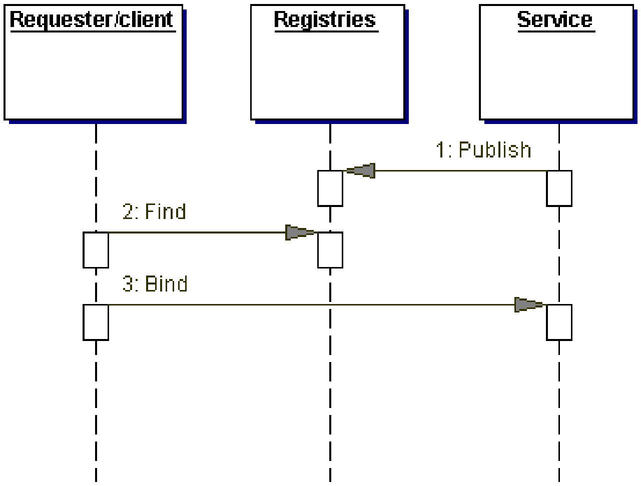


Figure 2-14, Web service registry model

In the classical version of web services, only one instance of a web service can exist.

Web Service example: SimpleMath

To illustrate how a web service works, an example of a very simple one is given here. The web service is called SimpleMath and is only able to add two integer values together. The source code for SimpleMath can be seen on Figure 2-15.

```
Namespace SimpleMath
{
  [WebService(Namespace="http://www.kjems.org/webservices/")]
  public class SimpleMathService : System.Web.Services.WebService
  {
    public SimpleMathService()
    {
      InitializeComponent();
    }
    #region Component Designer generated code
      // Default designer generated code left out
    #endregion

    [WebMethod(Description=
                 "Adds two integer values together and return the result")]
    public int Add(int x, int y)
    {
      return x+y;
    }
  }
}
```

Figure 2-15, SimpleMath web service code

The web service can be found on:
http://www.kjems.org/webservices/SimpleMath/SimpleMath.asmx
A test form using the web service can be found on:
http://www.kjems.org/webservices/SimpleMathClient/SimpleMathForm.aspx

Web Services as a Foundation for WSRF

After a short introduction to the added features in WSRF, the following sections will look into more details about each of the features.
WSRF is an extension of web services that defines a standardized way of dealing with stateful resources:

- WS-Resource: Defines way to handle web service resources.
- WS-ResourceLifetime: Defines mechanisms to destroy WS-Resources.
- WS-ResourceProperties: Defines method to change, retrieve and delete resource properties.
- WS-Notifications: Defines mechanisms for event subscription and notification using the idea of the Observer Pattern[GoF].
- WS-RenewableReferences: Defines a way that an Endpoint Reference can be renewed if the old one has been invalidated.
- WS-ServiceGroup: Defines a way to handle collections of WS-Resources
- WS-BaseFaults: Defines a fault schema for returning errors in a web service using XML.

Implied Resource Pattern

WSRF manages resources using the *implied resource pattern*. The implied resource pattern is defined as the following quote [WSRF-Expl]:

> The *implied resource pattern* refers to the mechanisms used to associate a stateful resource with the execution of message exchanges implemented by a Web service.
>
> - The term *implied* is used because the stateful resource associated with a given message exchange is treated as implicit input for the execution of the message request. By implicit, we mean to say that the requestor does not provide the stateful resource identifier as an explicit parameter in the body of the request message. Instead, the stateful resource is implicitly associated with the execution of the message exchange. This can occur in either a static or a dynamic way. We say that the stateful resource is associated with the Web service statically in the situation where the association is made when the Web service is deployed. We say that the stateful resource is dynamically associated with the Web service when the association is made at time of message exchange execution. When performed dynamically, the stateful resource identifier used to designate the implied stateful resource may be encapsulated in the WS-Addressing endpoint reference used to address the target Web service at its endpoint.
> - We use the term *pattern* to indicate that the relationship between Web services and stateful resources is codified by a set of conventions on existing Web services technologies, in particular XML, WSDL, and WS-Addressing.

WSRF.Net uses static association between a web service and resource, meaning the association is made when the web service is deployed.

Endpoint Reference Type

The Endpoint Reference Type EPR is used to address a specific web service resource. There is a one-to-one relation between and EPR and a web service resource. The sequence of actions needed to create a new web service resource can be seen on Figure 2-16. It is possible to make more than one instance of a web service resource, but they will all have different EPRs.

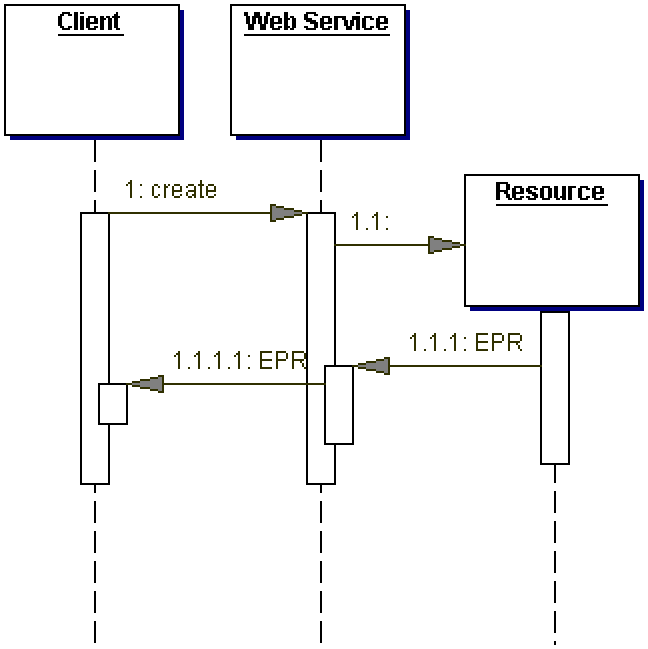


Figure 2-16, Web service creation sequence diagram

Lifetime of Web Service Resources

A web service resource instance is created with a lifetime. If the lifetime of the instance expires, then the instance is considered not longer needed, and may be destroyed to free up resources. The lifetime of a web service resource can be extended if needed. It is good programming practice to let the client of a web service resource extend the lifetime in a recurring manner. If something goes wrong in the client, the web service resource will stop getting life extensions and the resource can be freed. The importance of lifetime management is obvious in large-scale grid systems where resources would clutter up, if unused resources were not closed down in the occasion of the inevitable errors.

Notifications in WSRF

Notifications in grid services work like the observer pattern from [GoF]. In grid service terminology, the objects that subscribe to be notified are called sinks and the object that sends out notifications on a change is called the source.

2.4 NUnit

NUnit is a unit-testing framework designed for the .Net languages. The framework is designed to make it easy to perform and define unit tests. NUnit uses a series of attributes to denote the type of the code in the unit test. Assert statements are inserted into the code, so that if an assertion fails, the test of that code unit or component will fail. The newest version of Visual Studio called Whidbey/2005 has incorporated NUnit directly, making it very easy to perform unit tests. A class or method is simply marked to be unit tested, and Visual Studio generates most of the code necessary to perform the tests.

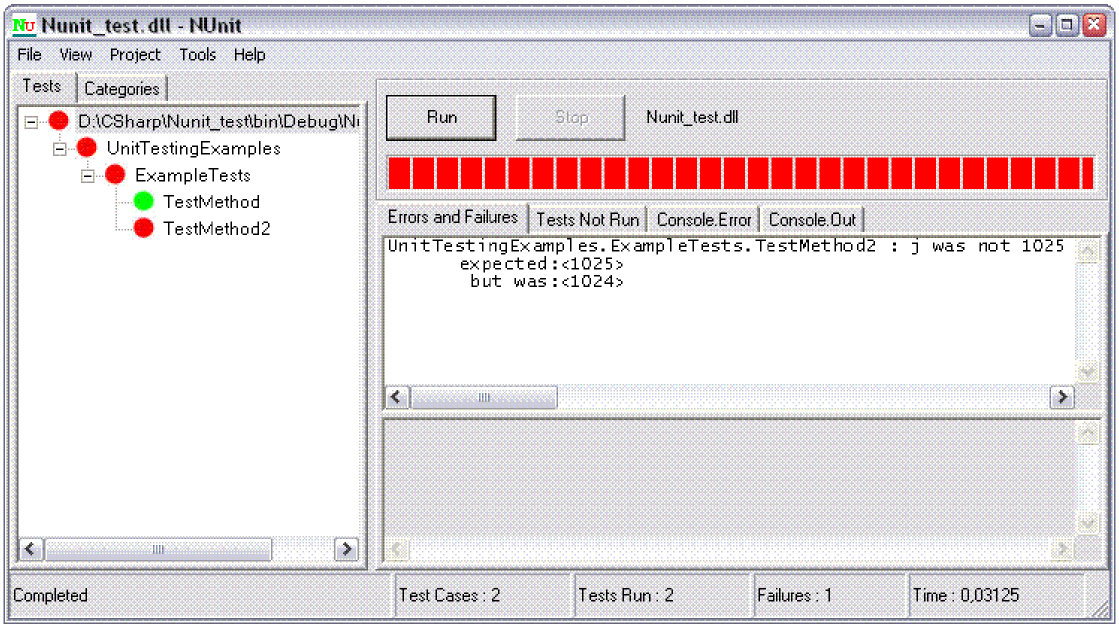
Examples

An example of how NUnit is used can be seen on Figure 2-17.

```
namespace UnitTestingExamples
{
  using System;
  using NUnit.Framework;

  [TestFixture]
  public class ExampleTests
  {
    private string testString;
    [SetUp]
    public void Setup()
    {
      testString = "";
    }

    [TearDown]
    public void TearDown()
    {
      // explicit destruction
    }

    [Test]
    public void TestMethod()
    {
      Assert.IsNotNull(testString, "testString was null");
      Assert.AreEqual("", testString, "testString was not empty");
      testString = "Hello";
      Assert.AreEqual("Hello", testString, "testString was not 'Hello'");
      testString += " World!";
      Assert.AreEqual("Hello World!", testString,
                      "testString was not 'Hello World!'");
    }

    [Test]
    public void TestMethod2()
    {
      int i = 2;
      int j = (int)Math.Pow(i,10);
      Assert.AreEqual(1024, j, "j was not 1024");
      Assert.AreEqual(1025, j, "j was not 1025");
    }
  }
}
```

Figure 2-17, Example code used for NUnit

The [TestFixture] attribute on class ExampleTests on line 6 is used to indicate that the class contains methods that should be tested with the test runner application. The [Test] attribute on line 22 and 34 indicates that these methods should be included as tests. The [SetUp] attribute on line 10 indicates a method that should be called before each of the [Test] methods. The [TearDown] attribute on line 16 indicates a method that should be run after each of the [Test] methods.



```
UnitTestingExamples.ExampleTests.TestMethod2 : j was not 1025
       expected:<1025>
        but was:<1024>
```

Completed | Test Cases : 2 | Tests Run : 2 | Failures : 1 | Time : 0,03125

Figure 2-18, NUnit Test Runner GUI

The result from executing the test code from Figure 2-17 can be seen on Figure 2-18.

2.5 Summary

The sections describing UML and design patterns are merely a presentation on the subjects. The following books describe the subject more thoroughly: [UML-Larman], [GoF]. Grid service technology is a new technology in rapid development. Even through the period of this project, groundbreaking events have taken place. The work on this project started out using OGSA and OGSI as a foundation for the grid services. During the project period, the WSRF specification was released and was clearly the way grid services would be made in the future. WSRF combined the technologies of OGSI and the new web service standards making a much smoother implementation. It was decided to use WSRF instead of OGSI for this project. The WSRF specification was made public February 2004 and the first technology preview implementation (WSRF.NET) was made public in July 2004. There was no documentation of the first technology preview, so the source of information was the source code.

NUnit has been used as a unit test method, and it gives an easy method of unit testing software.

Chapter 3

Design principles of computer game and grid software

This chapter will describe some of the general and fundamental design principles and algorithms used in this project. This includes a grid application design principle called the Manager-Worker design, which will be explained in the section *Grid Design*. The Manager-Worker design is made to solve problems using the *divide and conquer* strategy. A game server specific algorithm used for distributing game object state will be described in the section Computer game elements.

3.1 Grid Design
3.2 Computer game elements
3.3 Summary

3.1 Grid Design

Note: Some of the grid service features discussed in this chapter are still under development.

Software design using grid services requires a different way of thinking. The abstraction of grid services being distributed resource managers give a new way of designing software. Large grid software applications are composed from a set of more fundamental services. Some fundamental grid services can be shared between several different higher-level services. The WSDL description of services interfaces advocate reusability and interoperability in grid software design. Scalability is an important feature of grid service designs. To solve the scalability problem a divide and conquer model can be used. If a specific grid service instance is running low on resources on a specific host computer, it should be able to move to another host, or split between two hosts. Grid services are able to move and split between specific hosts or computers because of the transparency to the underlying computers that actually run the grid service. One grid service application might be using services on several different machines, and one machine might be running services from several different grid service applications. There is no coupling between a grid service, and the host computer that executes the grid services.

The following list summarizes the important benefits of grid software design:

- Scalability (Divide and conquer)
- Transparency
- Interoperability and reusability

Grid software design elements

To be able to accommodate the requirement for scalability in the game server, grid based structural elements has to be found or invented. The structural elements are some sort of untested grid design pattern, or a way of how a certain problem will be solved in this project. The grid software design elements are generic building blocks for grid applications.

Grid technology is still very young and a lot of work will be done to standardization and development of fundamental tools. As the technology matures, aspects like the simple structural elements created for this project will be obsolete due to the global development effort in this field.

Grid application start-up

When the game server start, an initiator grid service will be responsible for starting and keeping all basic grid services alive. The basic grid services are those fundamental for a server. If one of the basic grid services fails to respond due to computer malfunction or other errors, it is the initiator grid services job to start it again on a suitable computer. The initiator grid service uses a configuration file for the basic functions of starting up, running and shutting down the game server. The existence of some sort of start-up or global manager service is very common in large-scale grid applications.

Grid service components

The grid service component is a design idea developed for this project, but could also be applicable in other scenarios. A grid service component is a set of grid services with a manager service in charge. The grid service component serves to solve a common problem using the divide and conquer principle. The manager grid service is in charge of the lifetime and assignments of jobs to its worker grid services. The manager will create a new grid service instance to solve a specific problem, and kill it if it is no longer needed. The workers can communicate directly with other grid services without the intervention of the manager. To get a better understanding of the dynamics in a grid service component an example will be used as seen on Figure 3-1.

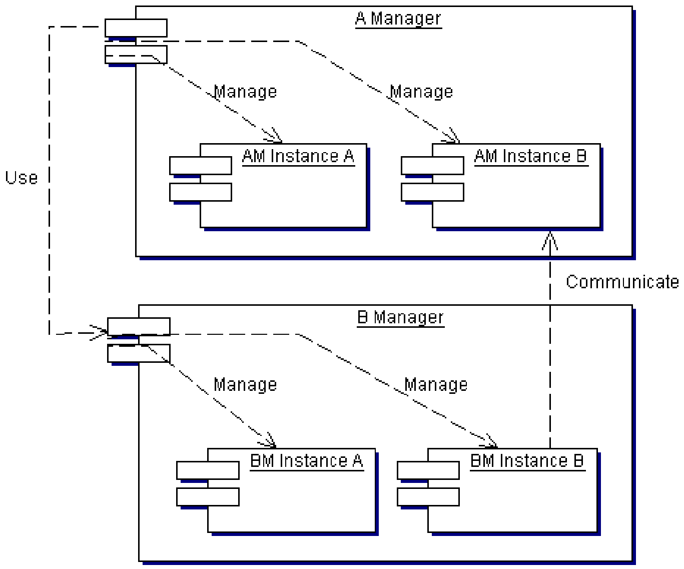


Figure 3-1, A manager and B manager interaction

Grid component interaction

Some of the grid component managers have to interact according to some specified logic. The A Manager(AM) and B Manager(BM) from Figure 3-1 shows an example of the interaction between two grid components. The nature of the interactions is a part of the application logic. That means the logic behind the interactions is a part of the final application, so the design principle will not dictate how the interactions should be done. The instances managed by the Manager will be called Workers.

3.2 Computer game elements

Game Components

A game server consists of a set of logical components that has to be modeled. For the system to be scalable, each of the components should be designed with a possibility of splitting the work into two separate grid service instances in accordance with the divide and conquer principle.

The following game components have been identified in a typical large-scale computer game:

- Object management (Active data/game rules)
- Player management
- Database management (Static/Persistent data)
- World simulations
- Path finding

The three core components: Object management, player management and database management is critical for the functionality of a large-scale game server. The two other components world simulation and path finding is not critical for the functionality of the game server, but gives some added tools for a good game experience. In this project, the focus will be on the object management and player management of both computer and human controlled players. It will be investigated how these two components can be implemented with grid service technology.

Game Objects and states

Most games consist of game objects, and each of these objects can have a set of states. The understanding of the terms objects and states is easier with a few examples.

Examples of objects are:

- A player character (PC)

- A monster
- A box made of wood
- A weather condition
- A non playing character (NPC)

Examples of object states are:

- A player character is *moving* from point A to point B.
- A monster is *shooting* in a direction.
- A box made of wood is *positioned* at a location *pointing* in a direction.
- A weather condition is currently *raining***.**
- A player character is *trading* with a non playing character.

An object can have multiple states, and states can include interaction with multiple objects. A running game simulation will constantly change states of objects to make the game world a dynamic environment for players to interact with. In multiplayer games, these constantly changing states have to be distributed to the clients.

Distributing Object states

Computer games are getting increasingly more complex, and the number of objects and states required to create the game world is growing. It is important to keep the amount of object states that have to be communicated over the network in multiplayer games to a minimum. To accommodate the requirement for reducing the communication between clients and servers, an algorithm for distributing game objects states is going to be used [Object-State].

Relevant Set

The relevant set algorithm is defined as the set of objects that is relevant for a given client at a given time. If an object is relevant to the client, it needs to be distributed to that client. Relevant sets typical consist of game objects in a near vicinity of a client, but they can also be other things like in-game weather conditions relevant for the client.

This simplified pseudo code will illustrate how the idea works.

- Extract the relevant set from the game world representation.
- Send data needed to update the objects in the relevant set to the client.
- The client applies the relevant set data to the local game world representation.

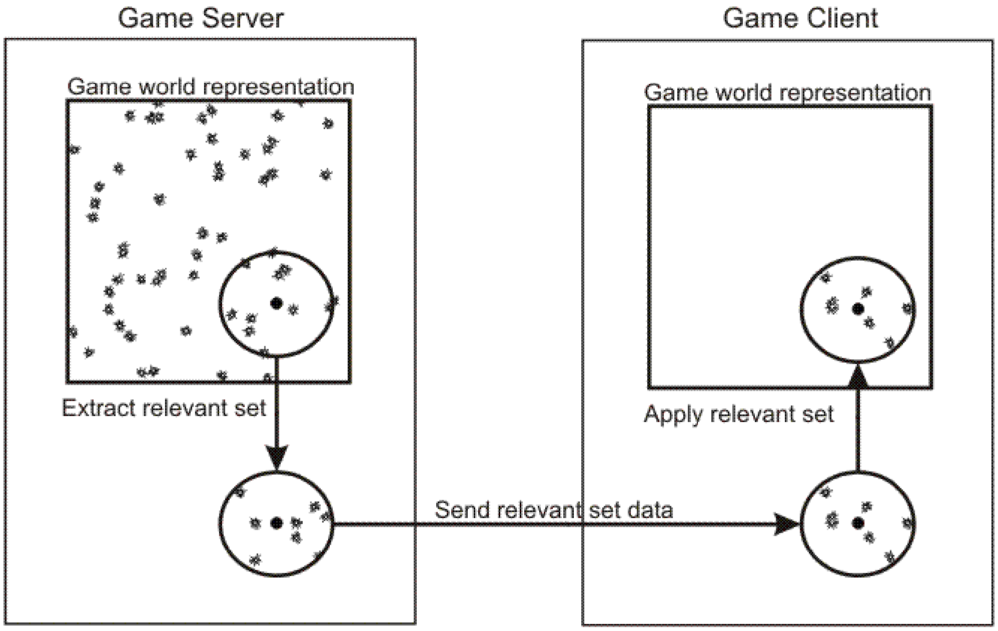


Figure 3-2, The relevant set algorithm

This algorithm works in both ways, so the client also extracts relevant data and sends it to the server.

3.3 Summary

This chapter has investigated the two aspects of this project: Grid services and game server design. Grid technology is a new technology, so there is not much written material on grid application design. The Manager-Worker design principle is developed for use in this project. It is a design made for solving large problems using the *divide and conquer* principle. The manager controls a set of workers, working on a common problem. The problem has been divided into *n* parts, where the workers solve a part each. The manager is in charge of the work process and controls the lifetime of the workers.

Formulating general theories of game server design is very hard, because game design differ a lot. The area of theories for game server design becomes very wide and is in most cases specialized solutions to a specific game. One thing that most MMOG type of games share is some sort of information filtering, so only relevant information is sent to the client computer. The algorithm for doing that in this project is called the Relevant Set algorithm and is described above.

Chapter 4

Case Study: Application of grid in computer games

This chapter will describe how the technologies from chapter 2 combined with the design principles of chapter 3 can be used to solve some of the problems with computer game servers described in chapter 1.

The chapter will describe how the framework and test application are implemented. Key elements from the source-code will be examined. This chapter will also contain both unit tests and performance tests. The test application will be tested according to the project specification.

4.1 System description

4.2 Analysis

4.3 Design and implementation

4.4 Test

4.5 Summary

4.1 System description

The purpose of the case study is to examine, if and how applicable grid services are as a foundation for a MMOG server. To make the examination, a simplified test a game server will be made along with a very simplified game client. The game server will be tested to clarify how a game server will benefit from a grid service foundation, and where it will be a disadvantage.

Definition of game object in this case study

A game object as described in section 3.2 will consist of the following:

- A current location
- A destination location
- A movement speed
- A unique identification
- A set of interaction rules called Actions that define a behavior

The game object exist in a 2-dimensional world, so all locations are defined in (X,Y) coordinates. Game objects can be many things, from a box of wood to an animal moving around in the virtual world. The interaction rules is what makes each object behave differently.

Simulation of the virtual world

Section 3.2 described how a simplified game server works. A game server constantly changes the state of the game objects. The rules of how the game objects change state, is game specific, and details concerning that have no relevance for this project. The simplification of the game server in this case study will primarily be on the rules of how objects change state. The simulation is constantly changing the states of the game objects according to the world rules and the objects interaction rules. The clients should only be updated with changed states that are relevant to them, so the information flow between server and client is kept to a minimum.

Game client

The game client will be very simplified as it is not the focus of this case study. The game client should be able to display the position of the game objects in a 2D virtual world. It should also be possible to move around.

4.2 Analysis

This section analyses how a game server should be designed and implemented using grid services.

Benefits of using grids in a game serve

A grid based game server solution should if possible support following set of important features:

- Scalability
- Robustness
- Easy maintenance
- Low equipment cost

These are all features desirable in any server based solution. The grid services provide an infrastructure that is well suited to form a fundament for a game server that supports the features above.

Scalability: Scalability is possible because grid service instances are created and managed so the grid application is transparent to the computers that actually run the grid application.

Robustness: Robustness can be achieved using the lifetime management and heartbeat abilities that grid services provide. Lifetime management is typically used to shut down services that are no more needed. Heartbeats are used to check if specific grid services are always running.

Easy maintenance: A well-designed grid application will not fail under a computer breakdown or shutdown, but continue after relocating some components to another computer.

Low equipment cost: Because the price of a computer increases exponential in compared to the computer speed, it is beneficial to use a lot of small computer rather than few big ones. Spreading the load of a grid application over many computers, will reduce the equipment compared to using only one big computer.

Problems with using grids in game server

Some of the aspects that might become problematic when using grid services for a game server are:

- Higher development cost (because grid is a new technology)
- Lower performance in regard to response time

The development cost when using a new technology is higher because everyone who works with the new technology has to learn it first. New technology also tend to change as it matures, so work done in the start of the development phase might become obsolete in the end. All the infrastructural benefits that grid services can provide come with a price. The marshalling and interacting grid services causes an overhead that might have a negative impact on the response-time of the application. Computer games are a type of application where the response-time is critical. Bringing the response-time to a minimum is one of the biggest challenges developing a grid based game server.

Scalability in Massively Multiplayer Online Games

The complexity of the work that a MMOG server is doing depends upon the following factors:

- Number of players
- Number of objects
- Number of interactions

It is important to make the MMOG server scalable in relation to the complexity elements above.

Security in Massively Multiplayer Online Games

As described in chapter 1.2, the security in computer games is very important. The fundamental server design should be made so it is possible to achieve a high level of security. Encrypting data between server and client is problematic as seen on Figure 4-1.

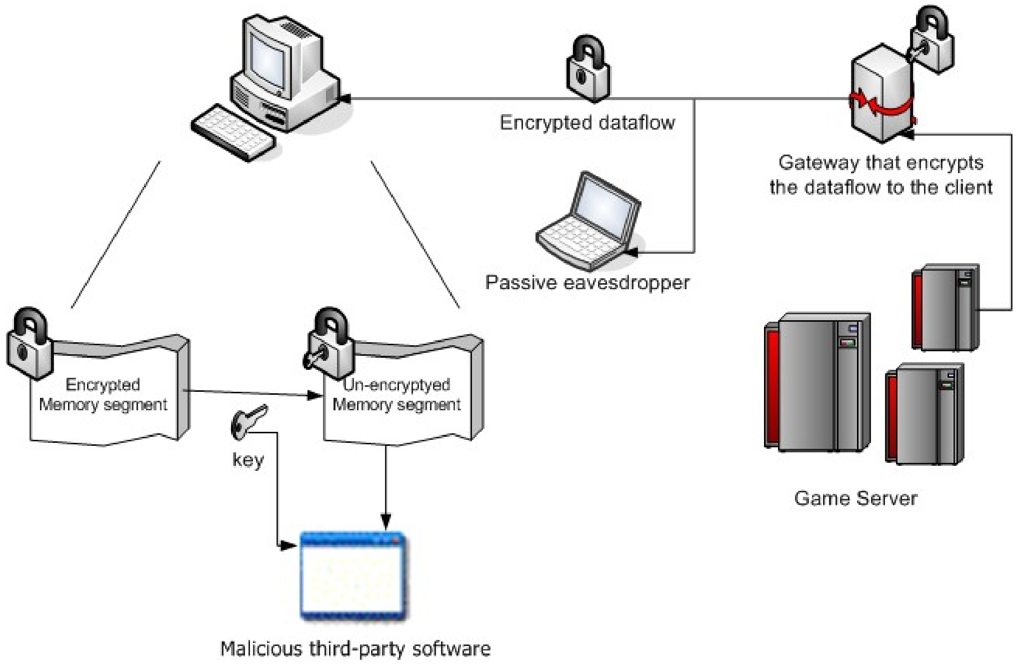

Encrypted dataflow
Gateway that encrypts the dataflow to the client
Passive eavesdropper
Encrypted Memory segment
Un-encryptyed Memory segment
key
Game Server
Malicious third-party software

Figure 4-1, Security issues with game server/client communication

The malicious user in the scenario of computer game security is the cheating player. The malicious user has access to the game data before it reaches his computer by letting another computer on the network make a passive listening to the data flow. This kind of security issue is called a passive eavesdropping. If the key negotiation between the server and client is done correctly, and the data is encrypted, the passive eavesdropper attack is impossible. The game client receives encrypted data from the server, and will have to decrypt the data before it can be used. That causes two security problems: The client must have the decryption key, and the un-encrypted data must reside in the memory somewhere. The malicious user has access to both the key and the un-encrypted data making him able to listen in on the information sent between the game server and game client.

Another type of problem with cheating, is tampering or infusion of malicious packets into the network stream between the server and client. The server should be resilient to attacks like this by enforcing integrity control on the communication.

The focus in this case study is not game security, but it should be noted that security in computer games is very important.

The game server fundamental design idea

The game server is composed of three main grid components, two auxiliary grid components and a database as seen on Figure 4-2. The three main grid components are essential for the operation of the game server. The main grid components are Player Agent, Zone Simulation and Database Cache and they are marked with bold on the figure. The other two grid components on the figure are given as examples of auxiliary grid components. It is also possible to come up with other auxiliary grid components that can solve specific game related problems. The focus in this project will primarily be on the two basic grid components: Player Agent and Zone Simulation. The number of game objects in this case study is very low, so there is no need for a Database Cache.

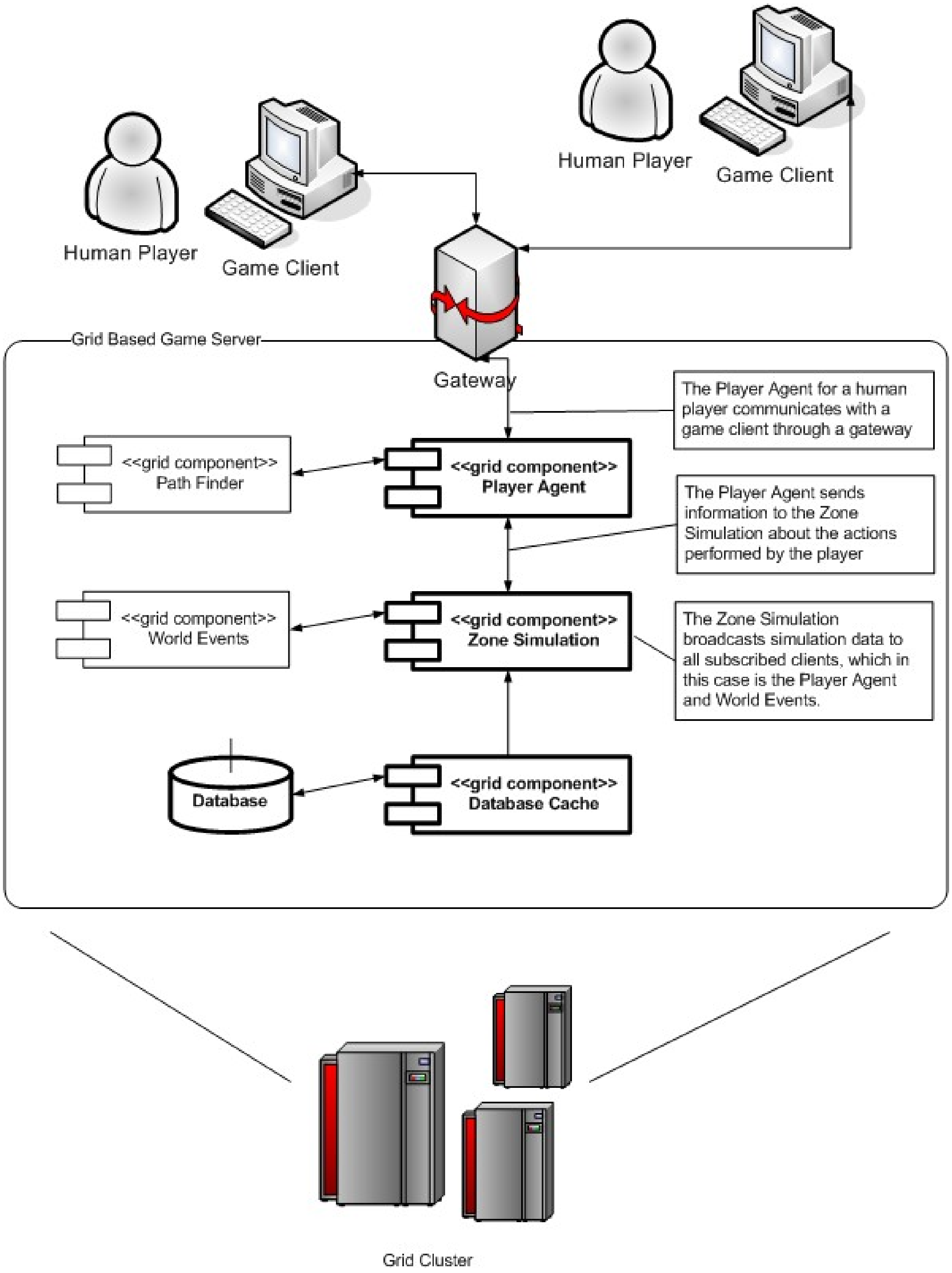

Human Player
Game Client
Human Player
Game Client
Grid Based Game Server
Gateway
The Player Agent for a human player communicates with a game client through a gateway
<<grid component>> Path Finder
<<grid component>> Player Agent
The Player Agent sends information to the Zone Simulation about the actions performed by the player
<<grid component>> World Events
<<grid component>> Zone Simulation
The Zone Simulation broadcasts simulation data to all subscribed clients, which in this case is the Player Agent and World Events.
Database
<<grid component>> Database Cache
Grid Cluster

Figure 4-2, Game server

Database Cache

The database cache(DC) grid component works as a cache for the global persistent database. Data retrieved from the global database will be stored until a specified cache size has been filled. In most cases, other grid component instances using a specific DC instance will need the same information repeatedly over a given period. The purpose of the DC is to decrease the workload of the global database and to improve response performance.

Zone Simulation

A Zone Simulation(ZS) instances handle all interaction between game objects in a specific area of the game. Game objects can be other players, monsters, boxes, doors and so on. The ZS instance only handle a small subset of the entire game world the players can interact with. Each ZS instance is associated with one DC instance, and all static game information is retrieved through that ZS instance.

Player Agent

The Player Agent(PA) manager is in charge of all the PA instances that handle the communication with players and AI Players. A PA instance receives data from a ZS instance that is relevant to a specific player. It then makes necessary filtering and marshalling of the data before it is sent to the player. A PA instance only has to subscribe to data from the ZS instance that the player exists in. If a player moves from ZS instance A to B, it is the job of the ZS to make sure the PA instance get an updated reference to ZS instance B.

Zone Simulation and Player Agent interaction in relation to relevant set algorithm

The relevant set algorithm is performed by the Zone Simulation(ZS) and Player Agent(PA) in collaboration. A ZS instance is sending all state changes to subscribing PA instances. The PA instances filters the data sent to them, so only what is relevant is being sent to the client or player. Any encoding, marshalling or encryption will also be done in the player manager. Complex game objects on the server might hold more information than the client need, so a PA instance will also extract only relevant data from complex game objects. A PA instance will know what the client knows, and will only send state changes compared to the current state of the client. It is important for game servers that only the necessary data is sent to the client because of the low bandwidth of clients.

4.3 Design and implementation

This chapter describes how the game server is implemented using grid services in the form of WSRF. At the time of writing, there was only one implementation of WSRF available and it was called WSRF .Net v1.0. [WSRF.Net]. WSRF.Net v 1.0 is based on C# and Microsoft's .Net framework v. 1.1. The WSRF.Net implementation is only partial, but the most essential things are included. In section 3.1, it was described how grid services were transparent to the physical computer they were executed from. This is not the case for this very young version of WSRF.NET. There still needs to be done a lot of work in the service utility layer. The way WSRF services is made transparent to the physical computer has to be standardized so all WSRF implementations use the same method. Bottom line is that the implementation in this case study is made assuming among others, transparency to physical computers is working even though they are still being worked on.

Game Object

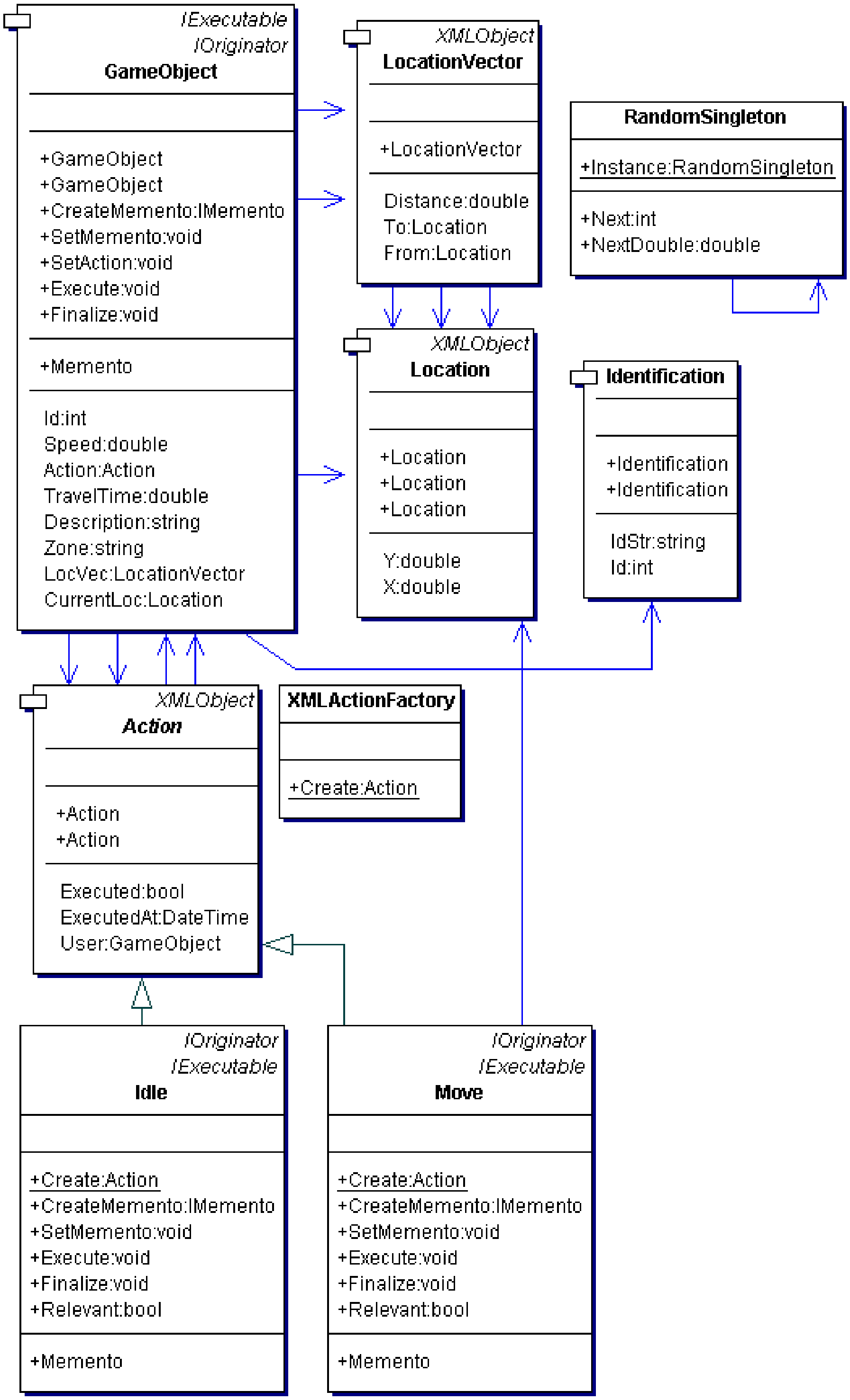


Figure 4-3, GameObject and Action class diagram

The following description of the requirements is used as a fundament for how the construction of the GameObject and Action was made. The GameObject contains information to describe a game object. Action has an Execute method, that when invoked will change the internal state in the coupled GameObject, according to the logic of the action. The GameObject and Action objects are tightly coupled, as neither of them is fully functional without the coupling. Action describes an abstract object that all the concrete actions inherit from. The GameObject and Action is coupled using the strategy design pattern. GameObject is the context, Action is the strategy and the concrete strategies are Move and Idle (see Figure 4-3). The Action that the GameObject is performing can be changed with the SetAction method. It should not be possible accidentally to invoke the Execute method, if the GameObject and Action has not been coupled. To avoid accidentally invocation of the Execute method, the Transformation pattern is used. IExecutable, Action and IOriginator are Forms. The concrete actions Move and Idle are Transformers. See section 2.2 for a description of the transformation pattern. First, an interface called IExecutable is made.

```
public interface IExecutable
{
  void Execute();
  void Finalize();
}
```

Figure 4-4, IExecutable interface

The concrete actions will now both inherit from Action and IExecutable. The constructor of the concrete actions is made private and a Create method is made to construct an object of the concrete type, returned as the abstract type Action. An example of this can be seen on Figure 4-5 where the Create method from the Move class is shown.

```
public static Action Create(Location toLoc)
{
  return new Move(toLoc) as Action;
}
```

Figure 4-5, Create method in Move class

As the abstract class Action does not have an Execute method, it is not possible to accidentally execute a concrete action without having a coupling to a GameObject.

The GameObject is part of a large distributed system, and it will be necessary to work with identical copies of GameObjects in different parts of the system. The easy solution would be to send the entire GameObject each time a change was made. That would however cause a lot of overhead as GameObjects change constantly and they contain more information than needs to be sent. To reduce the overhead, the memento design pattern will be used. The Action contained in the GameObject is an abstract type, so it will be necessary to implement the memento pattern in both the GameObject and concrete actions. The interfaces seen in Figure 4-6 will be used in the implementation of the memento pattern.

```
public interface IMemento
{
}

public interface IOriginator
{
 IMemento CreateMemento();
 void SetMemento(IMemento memento);
}
```

Figure 4-6, Memento interfaces

The IMemento interface is empty because the memento should not be used or modified by external holders. The CreateMemento method extracts the essence of the state of the originator and stores it in a memento. The SetMemento method restores the state of the originator based on the information in the memento given as argument.

The implementation of the memento pattern in the class Move can be seen on Figure 4-7. The Memento class is a nested class in the Move class. CreateMemento and SetMemento are methods for supporting the IOriginator interface that Move inherits from. Note that the strategy- and memento design patterns are mixed together as the GameObject is the context, IOriginator is the strategy and Move is a concrete strategy in the strategy pattern.

```
#region Memento Pattern
[Serializable]
public class Memento : IMemento
{
 private Location toLoc;
 private Location fromLoc;
 private TimeSpan timeSpan;
 private bool bExecuted;
 public Memento(){}
 public Location ToLoc{get{return toLoc;}set{toLoc=value;}}
 public Location FromLoc{get{return fromLoc;}set{fromLoc=value;}}
 public TimeSpan TimeSpan{get{return timeSpan;}set{timeSpan=value;}}
 public bool Executed{get{return bExecuted;}set{bExecuted=value;}}
}
public IMemento CreateMemento()
{
 Memento mem = new Memento();
 mem.ToLoc=user.LocVec.To;
 mem.TimeSpan=DateTime.Now-executedAt;
 mem.FromLoc=user.LocVec.From;
 mem.Executed=this.Executed;
 return mem;
}

public void SetMemento(IMemento memento)
{
 Memento mem = memento as Memento;
 this.executedAt=DateTime.Now-mem.TimeSpan;
 user.LocVec.To=this.toLoc=mem.ToLoc;
 user.LocVec.From=mem.FromLoc;
 this.Executed=mem.Executed;
}
#endregion
```

Figure 4-7, Memento implementation in the concrete class Move

Note that CreateMemento returns an object of type IMemento(empty interface) and not Memento, so it will not be possible to modify or read any of the information contained inside the returned memento. There is a scenario to be aware of when using the memento this way: What if a created IMemento is given as argument to a SetMemento method in another type of object than it was created from? That is a problem and it should be made impossible to do by accident. The solution is the transformation design pattern discussed in chapter

Design Patterns on page 30. The concrete actions, Move and Idle are *transformers* and IExecutable, IOriginator and Action are *forms*. The only way of creating a concrete action is by using the Create method that returns the object masked as the abstract type Action. Action does not have the CreateMemento or SetMemento methods.

```
1  public class Memento : IMemento
2  {
3    public Memento(){}
4    private Action action;
5    private IMemento actionMemento;
6    public Action Action
7    {
8      get{return action;}
9      set{action=value;}
10   }
11   public IMemento ActionMemento
12   {
13     get{return actionMemento;}
14     set{actionMemento = value;}
15   }
16 }
17
18 public IMemento CreateMemento()
19 {
20   Memento memento = new Memento();
21   IOriginator actionOriginator = action as IOriginator;
22   memento.ActionMemento = actionOriginator.CreateMemento();
23   memento.Action = action;
24   return memento;
25 }
26
27 public void SetMemento(IMemento memento)
28 {
29   Memento mem = memento as Memento;
30   this.SetAction(mem.Action);
31   IOriginator actionOriginator = action as IOriginator;
32   actionOriginator.SetMemento(mem.ActionMemento);
33 }
```

Figure 4-8, Memento implementation in the GameObject class

The memento implementation in the concrete Action classes (Move and Idle) is only used by the GameObjects memento implementation as seen on Figure 4-8. The GameObject memento is made using the CreateMemento method on line 18-25. First the memento from the coupled Action is extracted and saved in the GameObject.Memento on line 22. Then the action itself is saved on line 23, and the memento is returned as an IMemento. If the GameObject is coupled to an Action of one concrete type, and the ActionMemento is created from an Action of another concrete type an runtime error will occur. To avoid the error from being able to happen, the Action stored in the GameObject.Memento is coupled with the GameObject before the SetMemento in the Action memento is called.

The entire construct is composed of 3 strategy-, 2 memento- and 1 transformation design pattern.

The GameObject class is modeling a game object. Game objects in this project are described by the following:

- **Id** : Unique identification
- **Speed**: Travelspeed when moving
- **TravelTime**: Calculated traveltime based on speed and distance
- **Description**: Name of the game object
- **GameAction**: The game action is executed when the game object is executed.
- **Zone**: String identifying the zone the game object exists in.
- **LocVec**: Vector describing the movement of the game object.
- **CurrentLoc**: Calculated based on time/speed/direction where the game object is currently located in the virtual world.

```
1    public Location CurrentLoc
2    {
3      get
4      {
5        if (locVec.Distance==0) return locVec.From;
6        Location currentLoc = locVec.From.Clone() as Location;
7        TimeSpan delta = DateTime.Now-action.ExecutedAt;
8        double distance = delta.TotalSeconds*speed;
9        double modifier = distance/locVec.Distance;
10       if (modifier>1) modifier=1;
11       if (modifier<0) modifier=0;
12       currentLoc.X+=(locVec.To.X-locVec.From.X)*modifier;
13       currentLoc.Y+=(locVec.To.Y-locVec.From.Y)*modifier;
14       return currentLoc;
15     }
16   }
```

Figure 4-9, CurrentLoc calculation

The CurentLoc property can be seen above in Figure 4-9. On line 5 it is checked that the object is actually moving, and if it is not then locVec.From is returned. On line 7-9 it is calculated how far in percent the object has moved from the start location locVec.From, to the end location locVec.To. On line 10 and 11 it is checked that the object has not moved beyond the boundary of the locVec. On line 12 and 13 the current location is calculated, and the result is returned at line 14.

A GameObject can be executed with the Execute method as it adheres to the IExecutable interface. The Execute in GameObject calls the Execute method in the coupled Action. The GameObject can be set to use a new Action with the SetAction method.

Game Actions

Game actions support the IExecutable interface. Each action is able to be executed when the Execute method is invoked. The Execute method alters the internal state of the Action, and alters the internal state of the GameObject that it was called from. The simplified game server in this project only supports two actions: Move and Idle action. The concrete game actions Move and Idle are basic actions. Complex actions can be performed by AI players by composing them from the basic ones. It should only very rarely be necessary to add new basic game actions to the system.

```
[Serializable]
public abstract class Action : XMLObject
{
  #region Private Member Variables
  [NonSerialized]protected GameObject user;
  [NonSerialized]protected DateTime executedAt;
  protected bool bExecuted;
  #endregion

  #region Constructor
  public Action():this(null){}
  public Action(GameObject gameObject)
  {
    this.user = gameObject;
    executedAt=new DateTime(0);
    bExecuted = false;
  }
  #endregion

  #region Public Properties
  public DateTime ExecutedAt{get{return executedAt;}}
  public GameObject User{get{return user;}set{user=value;}}
  public bool Executed{get{return bExecuted;}}
  #endregion
}
```

Figure 4-10, Abstract Action class

The abstract Action class can be seen on Figure 4-10. The [Serializable] attribute on line 1 is necessary if Action objects have to be serialized to XML. Elements marked with [NonSerialized] will be omitted from the serialization as seen on line 4 and 5. The executedAt remembers when an action was executed. If the action is on a timer it will be possible with the executedAt to se how much of the action has been performed. CurrentLoc from Figure 4-9 uses executedAt. The Executed property of type bool, indicates if the Action has been executed yet and is used to prevent an action from be executed twice.

```
#region IExecutable
public void Execute()
{
  if(this.User!=null && !Executed)
  {
    User.LocVec.To=toLoc;
    executedAt=DateTime.Now;
    bExecuted = true;
  }
}
public void Finalize()
{
  if(this.User!=null && Executed)
  {
    User.LocVec.From = User.CurrentLoc;
  }
}
#endregion
```

Figure 4-11, IExecutable implementation in class Move

The implementation of the IExecutable interface can be seen on Figure 4-11. The Finalize method in a concrete Action should be called just before the action is replaced by another.

```
public bool Relevant(GameObject gameObject)
{
  // Simple example Relevance critera
  const double relevantDistance = 10;
  double dis = Math.Min(
    Math.Min(Location.Distance(fromLoc, gameObject.LocVec.From),
Location.Distance(fromLoc, gameObject.LocVec.To)),
    Math.Min(Location.Distance(toLoc, gameObject.LocVec.From),
Location.Distance(toLoc, gameObject.LocVec.To)));
  return (dis<relevantDistance);
}
```

Figure 4-12, Relevance criteria in the Move class

The relevance criteria seen in Figure 4-12 is a simple example, and in a real game server more refined relevance calculations should be done. If the action is relevant to an object it must either move from or to the objects relevance radius.

XMLFactory

The XMLFactory is a method of constructing objects from XML. The ability to store objects as string-based XML is very important in this project. It makes it possible to communicate objects throughout the system using the network. It also makes it possible to store objects in a database. Some of the objects in this project are only constructed based on information stored in an xml-based database. The construction information in the database is stored in XML, so it makes sense XMLFactory takes an xml-based string as parameter. If the xml-based string can represent several different objects, it is necessary to use the XMLFactory to analyse the XML and create the right type of object. Some objects are partly contained in the XML, while others are completely contained. The Move action seen in Figure 4-13 illustrates an object that is completely contained in the XML. An object completely contained in XML can at any state be deconstructed to XML and then reconstructed to the original object and state again.

```
<Move>
  <toLoc>
    <X>5</X>
    <Y>5</Y>
  </toLoc>
</Move>
```

Figure 4-13, Move object in XML

The code for the XMLActionFactory can be seen on Figure 4-14. It analyses the root node of the XML, and creates a new object of the type of the root using XML serialization based on the XML given as parameter. This is possible because the Move object is completely contained in the XML. If Move had only been partly contained in the XML, the object should have been manually reconstructed instead. XMLActionFactory throws an ArgumentException if the reconstruction fails.

```
public class XMLActionFactory
{
 protected XMLActionFactory(){}
 public static Action Create(string parms)
 {
  XmlDocument xmlParms = new XmlDocument();
  try
  {
   xmlParms.LoadXml(parms);
   XmlNode root = xmlParms.LastChild["Action"].FirstChild;
   string s = root.LocalName;
   switch(s)
   {
    case "Move":
     return (Move)ObjectXMLSerializer.StringToObject(
                                          root.OuterXml,typeof(Move));

    case "Idle":
     return new Idle();

    default:
     throw new ArgumentException("Object specified for creation does not
                                    exsist: "+s);
   }
  }
  catch(XmlException e)
  {
   throw new ArgumentException(string.Format("Parameters passed to XMLFactory
                                 is not valid XML:{0}.\n{1}",parms,e.Message));
  }
 }
}
```

Figure 4-14, XMLActionFactory

The implementation of the generic StringToObject used in Figure 4-14 can be seen on Figure 4-15. Note that the type needs to be known. In the case where the XML describes an abstract object like Action, the XMLFactory has to be used to determine the concrete object type. In the case of Action, the concrete object types are Move and Idle.

```
public class ObjectXMLSerializer
{
 private ObjectXMLSerializer(){}
 public static object StringToObject(string s, Type type)
 {
  XmlSerializer xmlSerializer = new XmlSerializer(type);
  StringReader stringReader = new StringReader(s);
  return xmlSerializer.Deserialize(stringReader);
 }
}
```

Figure 4-15, ObjectXMLSerializer

All classes that inherit from abstract class XMLObject is able to convert itself into a string. The string can be based on either XML or SOAP. As seen on Figure 4-10, Action inherits from XMLObject. The implementation of XMLObject can be seen on Figure 4-16.

```
public abstract class XMLObject
{
 public string XML
 {
  get
  {
   XmlSerializer xmlSerializer = new XmlSerializer(this.GetType());
   StringWriter stringWriter = new StringWriter();
   xmlSerializer.Serialize(stringWriter,this);
   return stringWriter.ToString();
  }
 }
 public string SOAP
 {
  get
  {
   return ObjectSoapSerializer.ObjectToString(this);
  }
 }
}
```

Figure 4-16, Abstract class XMLObject

The source code in Figure 4-17 illustrates how the different types of XML/Object conversions can be used. An Action object of concrete type Move is converted into an XML-based string on line 2. The string is then converted back to the original object on line 3.

```
1   Action action = Move.Create(new Location(5,5));
2   string actionXml = action.XML;
3   Action action2 = XMLActionFactory.Create(actionXml);
```

Figure 4-17, Examples of XML/Object conversions

Zone Simulation

The Simulation class is running the actual simulation, by invoking the execute method on game objects. A queue of game objects that has an action to perform is maintained by the ActionSequenceSingleton class. The Simulation and ActionSequenceSingleton are the internal parts of the grid component ZoneSimulation shown on Figure 4-2.

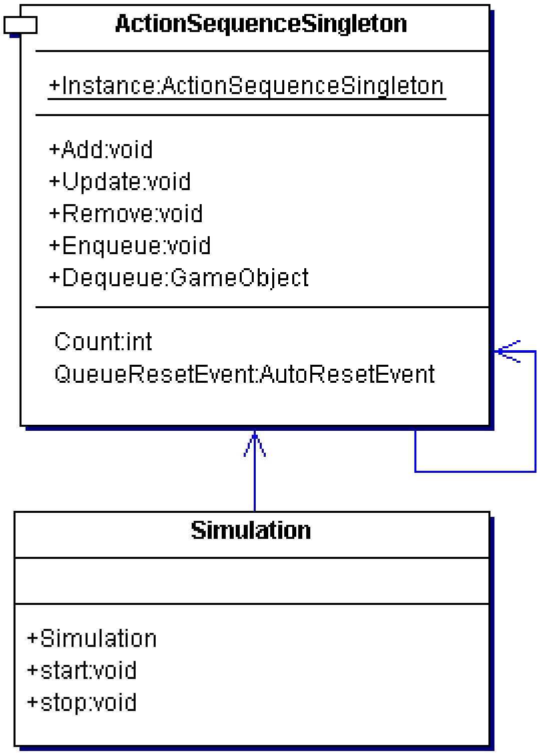


Figure 4-18, Simulation class diagram

Singletons are very easy to make in C# as it can be done with only one line of code as seen on Figure 4-19. There is small difference between the implementation of the singleton pattern seen on Figure 4-19 and the one described in the [GoF] book. The difference is that the [GoF] singleton instance is first created when the instance is requested the first time, while the singleton instance used here will be created at the start of the program.

```
public static readonly ActionSequenceSingleton Instance =
                         new ActionSequenceSingleton();
```

Figure 4-19, ActionSequenceSingleton declaration

The ActionSequenceSingleton maintains an internal queue of type Queue. The queue is accessed from different threads so it is necessary to synchronize the queue.

```
private void initialization()
{
  actionSequence = ActionSequenceSingleton.Instance;
  queueResetEvent = actionSequence.QueueResetEvent;
}
```

Figure 4-20, Using the ActionSequenceSingle

Synchronization of all C# collection classes including the Queue is very easy, and can be done with the code seen in Figure 4-21.

```
private ActionSequenceSingleton()
{
 unsyncedActionSequence = new Queue();
 actionSequence = Queue.Synchronized(unsyncedActionSequence);
 gameObjects = new Hashtable();
}
```

Figure 4-21, ActionSequenceSingleton constructor

The BroadCasterSingleton seen on Figure 4-22 is used for sending information from the Simulation to all subscribed listeners.

```
public class BroadcasterSingleton
{
 public static readonly BroadcasterSingleton Instance =
                                     new BroadcasterSingleton();
 private IPSender sender;
 BroadcasterSingleton()
 {
    sender = new IPSender();
 }
 public IPSender Sender
 {
   get{return sender;}
 }
 public void AddRecipient(ISendStrategy sendStrategy)
 {
   sender.AddRecipient(sendStrategy);
 }
}
```

Figure 4-22, BroadcasterSingleton

The simulation is started by calling the public method start, from the simulation class. The simulation is stopped by calling the stop method. Start and stop methods can be seen on Figure 4-23.

```
public void start()
{
 running = true;
 run();
}
public void stop()
{
 running = false;
 queueResetEvent.Set();
}
```

Figure 4-23, Start and Stop methods

The simulation is running inside the private run method as seen on Figure 4-24. To avoid using busy-wait where the actionSequence is pulled for new information constantly, an AutoResetEvent is used. Whenever a change is made to the queue, or if the run method should wake up for other reasons then queueResetEvent.Set() is called. Line 6 in the run method is waiting for a signal from .Set() before it continues. On line 9 and 10, a GameObject that has an action to be executed is de-queued and executed. The new updated GameObject memento is broadcasted to all listeners on a specified UDP multicast group on line 11.

```
1    private void run()
2    {
3      GameObject gameObject;
4      while(running)
5      {
6        queueResetEvent.WaitOne();
7        while (actionSequence.Count>0)
8        {
9          gameObject = actionSequence.Dequeue();
10         gameObject.Execute();
11         broadcastSender.Sender.Send(gameObject.CreateMemento());
12       }
13     }
14   }
```

Figure 4-24, Simulation run method

The Enqueue method in ActionSequenceSingleton takes a GameObject.Memento as argument as seen on Figure 4-25. The ActionSequenceSingleton maintains a Hashtable with the key being the ID and the body being the GameObject. The Hashtable contains all the GameObjects that exist in its zone.

```
public void Enqueue(GameObject.Memento memento)
{
  GameObject gameObject = gameObjects[memento.Id] as GameObject;
  gameObject.SetAction(memento.Action);
   actionSequence.Enqueue(gameObject);
  queueResetEvent.Set();
}
```

Figure 4-25, ActionSequence Enqueue method

The sequence diagram on Figure 4-26 show the actions performed when a GameObject or Action is added to the simulation. The GridInGamesConnector and ZoneSimulationWorker in the figure will be described later.

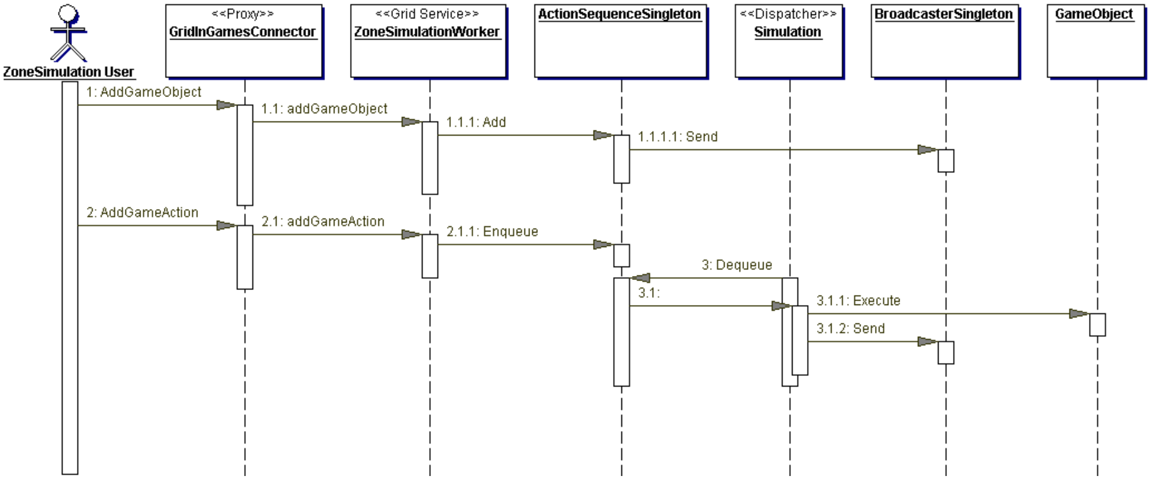


Figure 4-26, Simulation sequence diagram

Network Communication

A small custom-made network library is made for sending and receiving data with TCP and UDP protocols. The IPReciever class can receive data from both UDP and TCP depending on how it is constructed. The received data is put into a local Queue as it arrives. Users of the IPReciever class can Dequeue the information as they see fit. The IPSender class is able to send data over the network using UDP, TCP or UDP multicast protocol. A class diagram of IPSender class can be seen on Figure 4-27. The type of protocol is chosen at construction of the object. The strategy pattern is used where IPSender is the context, ISendStrategy is the strategy and UDPSend, TCPSend and UDPMulticast are the concrete strategies.

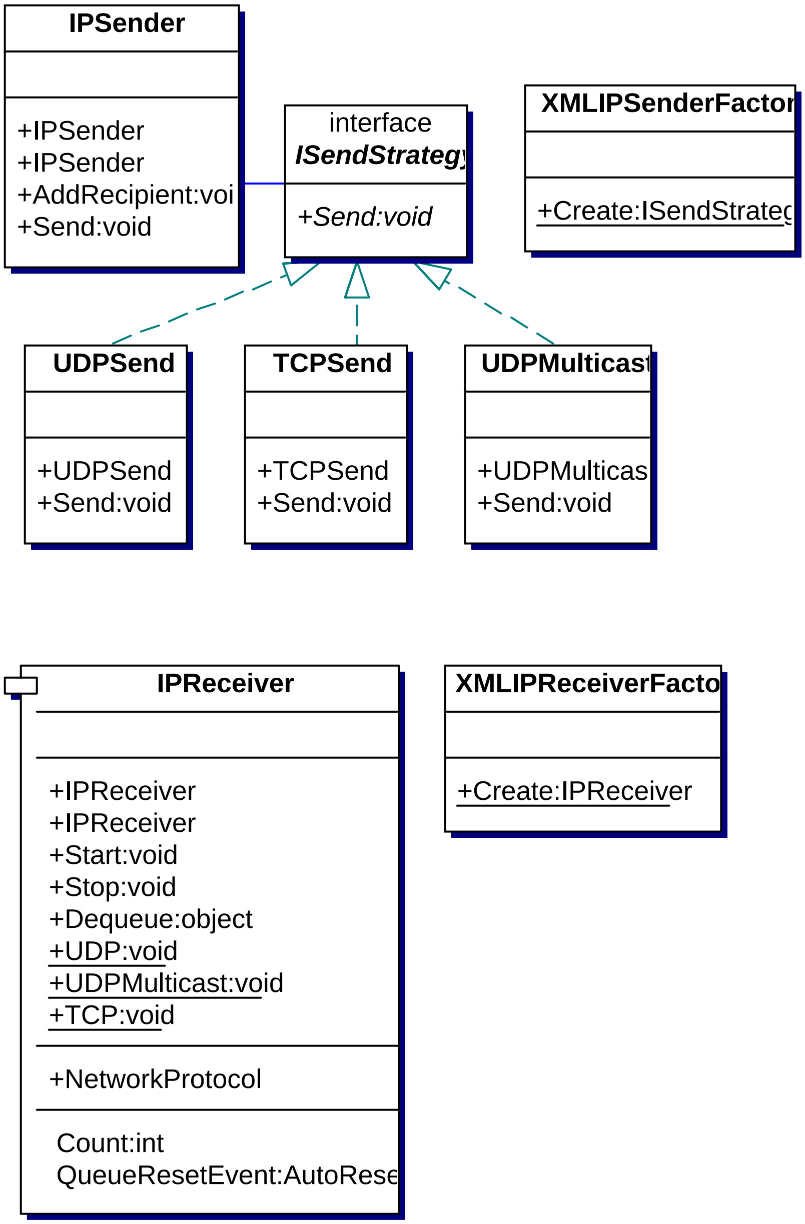


Figure 4-27, Network Communication

The ISendStrategy supports three functions all named Send that takes a string, byte-array or object as an argument.

```
public interface ISendStrategy
{
 void Send(string s);
 void Send(byte[] bytes);
 void Send(Object obj);
}
```

Figure 4-28, ISendStrategy interface

UDPSend, TCPSend and UDPMulticast have very similar implementation, so only UDPSend will be shown here. UDPSend uses the UdpClient from the .NET framework, which makes the class very simple and straightforward. The string to be send is converted to a byte array with the build-in encoding conversions on line 14 in Figure 4-29.

```
1    public class UDPSend : ISendStrategy
2    {
3     private UdpClient client;
4     ~UDPSend()
5     {
6      client.Close();
7     }
8     public UDPSend(string address, int port)
9     {
10     client = new UdpClient(address, port);
11    }
12    public void Send(Byte[] bytes)
13    {
14     client.Send(bytes, bytes.Length);
15     Thread.Sleep(0);
16    }
17    public void Send(String s)
18    {
19     Send((Object)s);
20
21    }
22    public void Send(Object obj)
23    {
24     Send(ObjectSoapSerializer.ObjectToBinary(obj));
25    }
26   }
```

Figure 4-29, UDPSend strategy class

The IPSender seen on Figure 4-30 maintains a list of recipients to which the class must send to when the Send method is invoked. To add a recipient the AddRecipient method is called and to remove a recipient the RemoveRecipient method is called. When the Send method is invoked, the argument is sent to all “subscribed” recipients on the list.

```
public class IPSender
{
 private ArrayList recipients;
 public IPSender()
 {
  recipients = new ArrayList();
 }
 public IPSender(ISendStrategy sendStrategy): this()
 {
  AddRecipient(sendStrategy);
 }
 public void AddRecipient(ISendStrategy sendStrategy)
 {
  recipients.Add(sendStrategy);
 }
 public void RemoveRecipient(ISendStrategy sendStrategy)
 {
  recipients.Remove(sendStrategy);
 }
 public void Send(byte[] bytes)
 {
  foreach(ISendStrategy recipient in recipients)
  {
    recipient.Send(bytes);
  }
 }
 public void Send(Object obj)
 {
  foreach(ISendStrategy recipient in recipients)
  {
    recipient.Send(obj);
  }
 }
 public void Send(string s)
 {
  foreach(ISendStrategy recipient in recipients)
  {
    recipient.Send(s);
  }
 }
}
```

Figure 4-30, IPSender class

The XMLIPSenderFactory on Figure 4-31 is similar to the other XMLFactories seen. It constructs an IPSender object with a specific send strategy and parameters based on an XML argument.

```
public class XMLIPSenderFactory
{
  protected XMLIPSenderFactory(){}
  public static ISendStrategy Create(string parms)
  {
    XmlDocument xmlParms= new XmlDocument();
    try
    {
      xmlParms.LoadXml(parms);
      XmlNode root = xmlParms.LastChild;

      if (root.Name!="IPSender") root = root["IPSender"];
      string s = root["protocol"].InnerText;
      string address = root["IP"].InnerText;
      int port = int.Parse(root["port"].InnerText);
      switch(s)
      {
        case "UDP":
          return new UDPSend(address, port);

        case "TCP":
          return new TCPSend(address, port);

        case "UDPMulticast":
          return new UDPMulticast(address, port);

          default:
        throw new ArgumentException("Object specified for creation does not
exsist: "+s);
      }
```

```
   }
   catch(XmlException e)
   {
    throw new ArgumentException(string.Format("Parameters passed to XMLFactory
                                 is not valid XML:{0}.\n{1}",parms,e.Message));
   }
  }
}
```

Figure 4-31, IPSenderFactory

IPReceiver can receive data from TCP, UDP or UDPMulticast protocols. The type of protocol is selected at creation by selecting one of three functions that adhere to the NetworkProtocol delegate signature. The constructor and delegate declaration can be seen on Figure 4-32.

```
public delegate void NetworkProtocol();
private NetworkProtocol protocol;
// Rest of member declaration left out

public IPReciever(NetworkProtocol protocol, string _address, int _port)
{
  this.protocol=protocol;
  // Rest of constructor body left out
}
public void Start()
{
  done=false;
  listenThread = new Thread(new ThreadStart(protocol));
  listenThread.Start();
}
```

Figure 4-32, IPReciever construction and startup

An example of the protocol selection can be seen on Figure 4-33, where the UDP protocol is selected.

```
IPReciever listener =
   new IPReciever(new IPReciever.NetworkProtocol(IPReciever.UDP), port);
```

Figure 4-33, External construction of IPReciever class using UDP

There are three Protocol delegates in the IPReciever class and they are much alike, so only the UDP version it shown here. It can be seen on Figure 4-34.

```
public static void UDP()
{
 UdpClient listener = new UdpClient(port);
 IPEndPoint endpoint = new IPEndPoint(address, port);
 try
 {
  while (!done)
  {
   byte[] bytes = listener.Receive(ref endpoint);

   syncMessageQueue.Enqueue(bytes);
   queueResetEvent.Set();
  }
  listener.Close();
 }
 catch (Exception e)
 {
  Console.WriteLine(e.ToString());
 }
}
```

Figure 4-34, UDP delegate function

The IPReciever class maintains a local Queue of objects containing the incoming data. The Queue is synchronized using the same method as in the ActionSequenceSingleton shown on Figure 4-19. External users of the IPReciever can use the queueResetEvent to avoid using busy-wait methods getting data from the queue.

Player Agent

The PlayerAgent class serves as an agent for the player. The PlayerAgent and the concrete players AIPlayer and HumanPlayer are internal parts of the grid component named Player Agent on Figure 4-2.

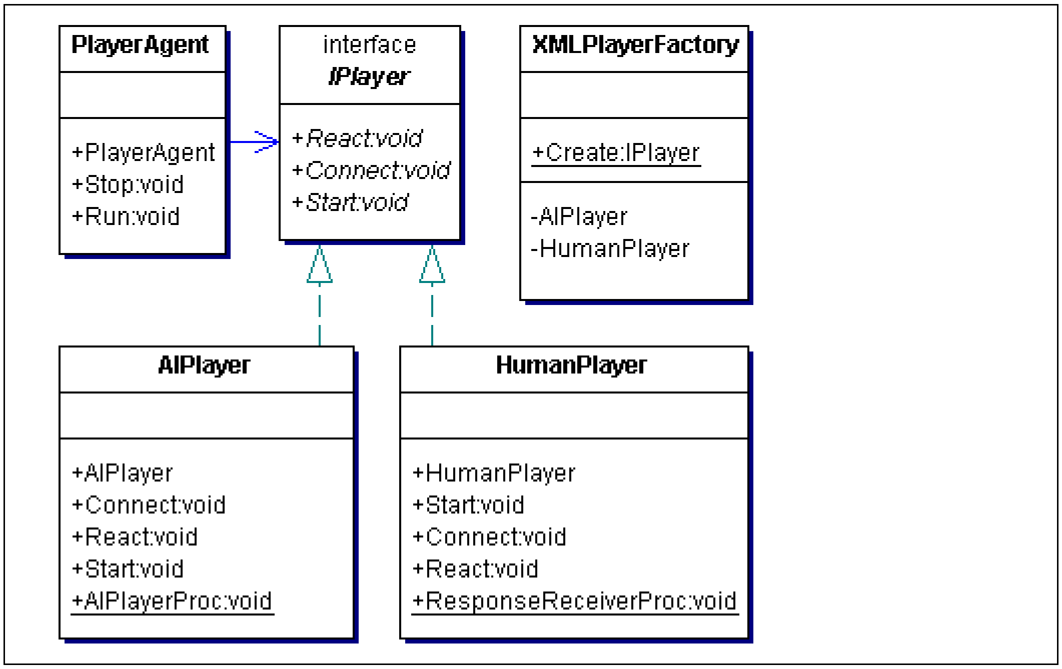


Figure 4-35, Class diagram of the PlayerAgent

Any information recived in the PlayerAgent is passed on to the Player's React method with the information as an argument. The information is processed in the respective concrete Player classes. There are two very different Player's in this project: AIPlayer and HumanPlayer. The HumanPlayer class handles the communication with the game client. The information relevant is sent to the game client. The response from the human player using his game client is received by HumanPlayer and passed on to the simulation. The interface to the simulation is through a WSRF webservice. The AIPlayer is the artificial intelligence of game objects. The AIPlayer in this project is very simple, but in a real computer game, the calculations made in the AIPlayer could potentially be very heavy and complex. The XMLPlayerFactory creates instances of either AIPlayer or HumanPlayer based on an XML-based string. Both the AIPlayer and HumanPlayer are private inner classes in XMLPlayerFactory. They are made private inner classes because then only the XMLPlayerFactory can create those two objects.

Note that the strategy design pattern is used. Player is the strategy while AIPlayer and HumanPlayer are the concrete strategies. PlayerAgent is the Context.

The PlayerAgent class seen on Figure 4-36 takes three arguments at construction. The first argument IPlayer, can be either a HumanPlayer or AIPlayer. The second argument is an IPReceiver that receive the data sent from the ZoneSimulation. The third argument is the EPR for the GridInGamesManager service that keeps track of all services in the system. The GridInGamesManager service is used when connecting to a ZoneSimulation instance. The PlayerAgent is oblivious to what kind of player it is agent for.

```
bool done = false;
private IPReceiver ipReceiver;
private AutoResetEvent queueResetEvent;
private IPlayer player;

public PlayerAgent(IPlayer player, IPReceiver ipReceiver, string
gridInGamesEPR)
{
  this.ipReceiver = ipReceiver;
  queueResetEvent = ipReceiver.QueueResetEvent;
  this.player = player;
  player.Connect(gridInGamesEPR);
}

private void Initialize()
{
  ipReceiver.Start();
  player.Start();
}

public void Stop()
{
  done=true;
}
```

Figure 4-36, PlayerAgent class constructor and startup

The Run method seen on Figure 4-37 uses the queueResetEvent to avoid a busy-wait. Information received is dequeued on line 9, and sent on to the Players React method.

```
1    public void Run()
2    {
3      done=false;
4      Initialize();
5      while(!done)
6      {
7        while(ipReceiver.Count>0)
8        {
9          player.React(ipReceiver.Dequeue());
10       }
11       queueResetEvent.WaitOne();
12     }
13   }
```

Figure 4-37, Run method in PlayerAgent

IPlayer seen on Figure 4-38 defines the interface of the objects created by XMLPlayerFactory. The interface defines the method React that is called from PlayerAgent when information is received. The Connect method is used for connecting the Player to the ZoneSimulation grid service. The Start method is used to activate the Player, by sending the GameObject of the Player to the ZoneSimulation .

```
public interface IPlayer
{
  void React(Object obj);
  void Connect(string url);
  void Start();
}
```

Figure 4-38, IPlayer interface used with XMLPlayerFactory

The XMLPlayerFactory creates objects that inherit from IPlayer. The two classes HumanPlayer and AIPlayer are both private inner classes of XMLPlayerFactory as seen on Figure 4-39. This way HumanPlayer and AIPlayer can only be constructed using XMLPlayerFactory. The safety measures to catch errors from a faulty XML parameter are put in the XMLPlayerFactory.

```
public class XMLPlayerFactory
{
  protected XMLPlayerFactory(){}
  public static IPlayer Create(string parms)
  {
    XmlDocument xmlParms= new XmlDocument();
    try
    {
      GameObject gameObject;
      xmlParms.LoadXml(parms);
      XmlNode root = xmlParms.LastChild;
      string s = root.LocalName;
      switch(s)
      {
        case "HumanPlayer":
          gameObject = new GameObject(root["name"].InnerText,
                                      root["zone"].InnerText);
          return
            new HumanPlayer(gameObject, new IPSender(
XMLIPSenderFactory.Create(root["ClientCommunication"]["IPSender"].OuterXml)),
XMLIPReceiverFactory.Create(root["ClientCommunication"]["IPReceiver"].OuterXml));

        case "AIPlayer":
          gameObject = new GameObject(root["name"].InnerText,
                                      root["zone"].InnerText);
          return new AIPlayer(gameObject);


        default:
          throw new ArgumentException("Object specified for creation does not
                                       exsist: "+s);
      }
    }
    catch(XmlException e)
    {
```

```
    throw new ArgumentException(string.Format("Parameters passed to XMLFactory
                                is not valid XML: {0}.\n{1}",parms,e.Message));
   }
 private class HumanPlayer : IPlayer
 {
  // Content of the HumanPlayer class
 }
 private class AIPlayer : IPlayer
 {
  // Content of the AIPlayer class
 }
}
```

Figure 4-39, XMLPlayerFactory class

The HumanPlayer constructor seen on Figure 4-40 takes three arguments. The first argument is the GameObject that contains the information about the player. The second argument is the IPSender for sending data to the user's game client. The third argument is the IPReceiver that receives data from the user's game client.

```
public HumanPlayer(GameObject _gameObject, IPSender _sender, IPReceiver
_receiver)
{
 gameObject = _gameObject;
 sender = _sender;
 listener = _receiver;

}
```

Figure 4-40, HumanPlayer constructor

The Connect method seen on Figure 4-41 connects the HumanPlayer to the zone or ZoneSimulation instance that the player resides in.

```
public void Connect(string gridInGamesURL)
{
 zoneSimulationConnect = new
    GridInGamesConnector.ZoneSimulationConnect(gridInGamesURL);
 zoneSimulationConnect.ConnectToWorker(gameObject.Zone);
}
```

Figure 4-41, HumanPlayer Connect

The Start method on Figure 4-42 sends the GameObject of the HumanPlayer to the ZoneSimulation.

```
public void Start()
{
 zoneSimulationConnect.AddGameObject(gameObject);
}
```

Figure 4-42, HumanPlayer Start

The React method uses the *Relevant* method in the *Action* from the incoming *GameObject.Memento* to determine if the information is relevant to the *player*. The HumanPlayer.React sends on all GameObject's and only relevant GameObject.Memento's (performed actions) to the client.

```
public void React(Object obj)
{
  if (obj is GameObject.Memento)
  {
    GameObject.Memento memento = (GameObject.Memento)obj;
    if (memento.Id==gameObject.Id)
    {
      gameObject.SetMemento(memento);
    }
    if ((memento.Action as IAction).Relevant(gameObject))
    {
      sender.Send(obj);
    }
    else
    {
       EventDebug.Send("Non-relevant action received");
    }
  }
  if (obj is GameObject)
  {
    sender.Send(obj);
  }
}
```

Figure 4-43, React method in HumanPlayer

It would be possible to lower the bandwidth requirements of server/client communication in two ways: Instead of using a SOAP serializer a binary one could be used. Secondly, the data could be compressed. It would also be possible to apply encryption to avoid passive listening attacks.

```
public static void ResponseReceiverProc()
{
  done=false;
  AutoResetEvent queueResetEvent = listener.QueueResetEvent;
  listener.Start();

  Object obj;
  while(!done)
  {
    queueResetEvent.WaitOne();
    while(listener.Count>0)
    {
      obj = listener.Dequeue();
      if (obj is GameObject.Memento)
      {
        GameObject.Memento memento = obj as GameObject.Memento;
        zoneSimulationConnect.AddGameAction(memento.Id, memento.Action);
      }
    }
  }
}
```

Figure 4-44, HumanPlayers ResponseReceiverProc thread

The ResponseReceiverProc thread seen on Figure 4-44, receives information from the game client. The information received is analysed and acted upon typically by sending it to the ZoneSimulation.

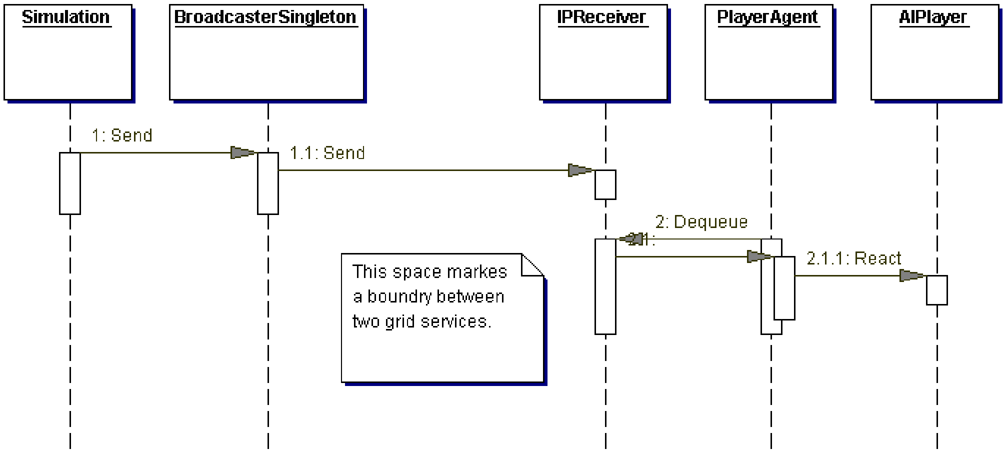


Figure 4-45, PlayerAgent sequence

The sequence diagram on Figure 4-45 shows the interaction between the Simulation and the PlayerAgent. Later it will be described how both the Simulation and PlayerAgent are encapsulated inside the grid service.

Test game client

The test game client is able to display information from the game server. It is also able to send a Move action with the location taken from a mouse-click to the game server. The class diagram of the test game client can be seen on Figure 4-46. The game client uses the observer pattern where GameObjectModel is the subject. ViewArea and GridInGamesGUI are observers.

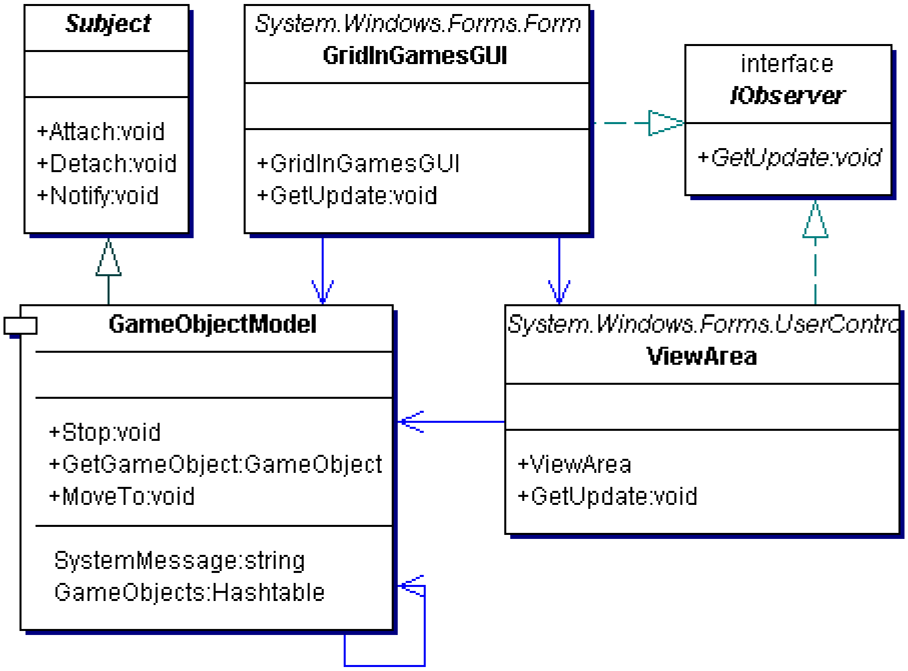
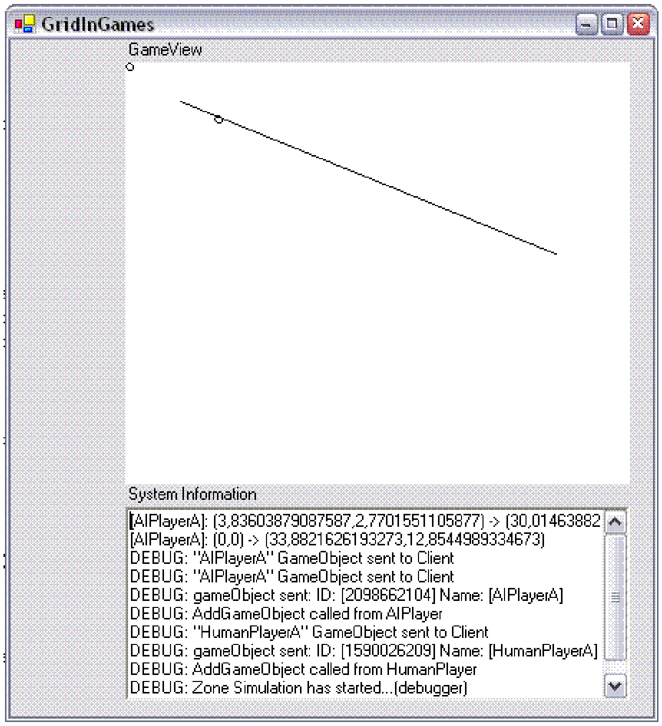


Figure 4-46, Test Game Client class diagram

The graphical user interface (GUI) of the test game client can be seen on Figure 4-47. The line represents the From and To Location of a GameObject. The Location of the dot on the line is repeatedly calculated using the CurrentLoc method in the local GameObject.

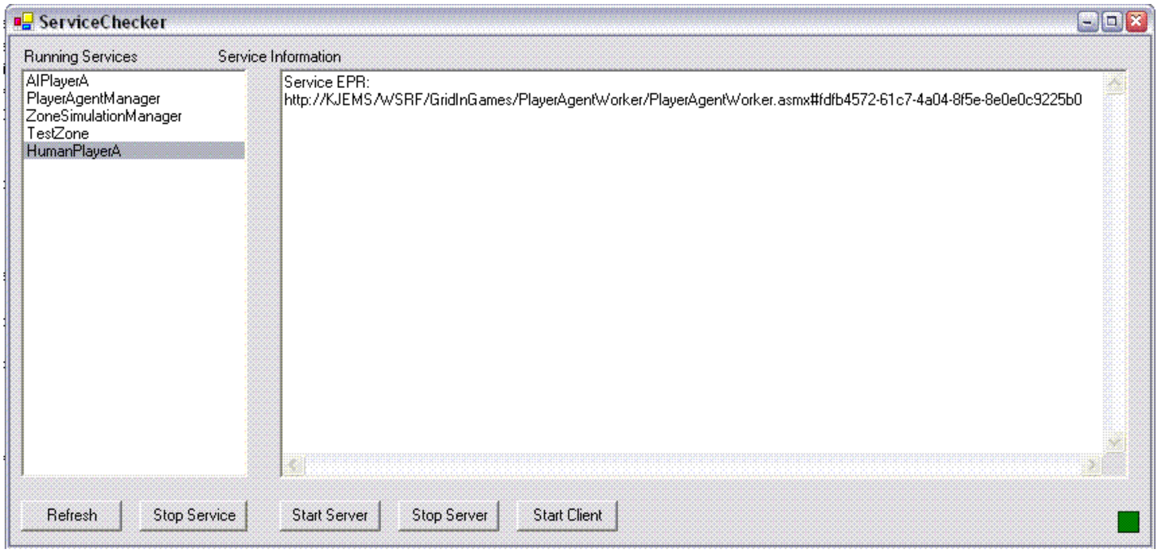


```
[AIPlayerA]: (3,83603879087587,2,7701551105877) -> (30,01463882
[AIPlayerA]: (0,0) -> (33,8821626193273,12,8544989334673)
DEBUG: "AIPlayerA" GameObject sent to Client
DEBUG: "AIPlayerA" GameObject sent to Client
DEBUG: gameObject sent: ID: [2098662104] Name: [AIPlayerA]
DEBUG: AddGameObject called from AIPlayer
DEBUG: "HumanPlayerA" GameObject sent to Client
DEBUG: gameObject sent: ID: [1590026209] Name: [HumanPlayerA]
DEBUG: AddGameObject called from HumanPlayer
DEBUG: Zone Simulation has started...(debugger)
```

Figure 4-47, Test Game Client GUI

Service Checker

The Service Checker is used to monitor the grid services that compose the game server. It is possible to retrieve information about each of the services. In this early version, only the service EPR is given, but it could easily be evolved to include more of the grid service information. The Service Checker is also able to start all the services of the game server, and close them down again. The Service Checker is a hollow construct not containing much else than what the Visual Studio project wizard and GUI editor has made. The Service Checker can be seen on Figure 4-48.



AIPlayerA
PlayerAgentManager
ZoneSimulationManager
TestZone
HumanPlayerA

Service EPR:
http://KJEMS/WSRF/GridInGames/PlayerAgentWorker/PlayerAgentWorker.asmx#fdfb4572-61c7-4a04-8f5e-8e0e0c9225b0

Figure 4-48, Service checker GUI

Grid Services

Until now, this Design chapter has only been working with non-grid based software development. It will now be described how the code explained until now can be used as components for a grid based server solution. Figure 4-2 on page 64 shows how the game server is constructed, using a number of grid components. Not all grid components on the figure have been implemented in this project, but those that have will be described here. The code needed to make a grid service is somewhat different from the code seen until now. Attributes are used to augment the code with extra information for the grid service compiler to use.

```
namespace ZoneSimulationManager
{
  [WsdlBaseName("ZoneSimulationManagerService",
      "http://www.kjems.org/GridInGames")]
  [WebService(Description="Manager for ZoneSimulationWorker instances")]
  [WebServiceBinding]
  [WSRFPortType(typeof(ImmediateResourceTerminationPortType))]
  [WSRFPortType(typeof(GCGResourceFactoryPortType))]
  [ResourceInitializerType(typeof(string))]
  public class ZoneSimulationManagerService : ServiceSkeleton,
IManagerService
  {
```

Figure 4-49, Grid Service header

The attributes of the grid services in this project can be seen on Figure 4-49. The three first attributes seen on line 3, 5 and 6 are web service attributes. The following three attributes on line 7, 8 and 9 are WSRF web service (grid service) specific. The attribute on line 3 defines the name of the service, and it will be used in the WSDL description of the service. The attribute on line 5 add additional information to the web service. In this case a description of the web service is added. The attribute on line 6 declares the binding of web service methods defined in the web service. The functionality is somewhat similar to inheriting from an interface in traditional C# code. The attributes on line 7 and 8 are what is called port type aggregation [WSRF.Net-Tur]. A port type is typically able to perform some standard functionality. By aggregating a port type in a WSRF web service, the service gains the functionality defined in the port type. The port type on line 7 makes it easier to destroy a WSRF web service, and the port type on line 8 makes it easier to create a WSRF web service. The attribute on line 9 declares the types used for initialization of the web service. In this case, a single parameter of type string is used.

```
[Resource]
private Hashtable EPRList;
```

Figure 4-50, WSRF Resource declaration in WSRF.NET

The thing that makes a WSRF web service (grid service) different from a normal web service is the ability to manage persistent resources. The WSRF.Net implementation has done the declaration of persistent resources incredibly easy. It is simply done by putting a [Resource] attribute on the member variable that should be persistent as seen on Figure 4-50. The WSRF.Net framework manages communication with a database to store the resource. When a WSRF session start, the resource is loaded from the database, and when the session stops the resource is saved back to the database if it was changed.

```
[WebMethod]
[SoapDocumentMethodAttribute(
   "http://www.kjems.org/GridInGames" + "/getEPR",
   Use=System.Web.Services.Description.SoapBindingUse.Literal,
   ParameterStyle=SoapParameterStyle.Bare)]
[return: XmlElementAttribute(
   "getEPRResponse", Namespace="http://www.kjems.org/GridInGames")]
```

```
public string getEPR(string ident)
{
 return EPRList[ident] as string;
}
```

Figure 4-51, WSRF web service method declaration

The WSRF method declaration can be seen on Figure 4-51. The attribute header of the web method describes the formatting of the incoming arguments and the returned results. The method returns the EPR of the service with identity given as argument. The list of identification and EPR-string relations are stored in the Hashtable EPRList shown in Figure 4-50.

The ZoneSimulationWorker and PlayerAgentWorker both start a new thread executing the actual Simulation and PlayerAgent. These threads are executed under the security permissions of the IIS webserver. If the threads are not closed down manually when a ZoneSimulationWorker or PlayerAgentWorker resources is destroyed, they will continue running as ghosts. The only way to destroy the ghost threads is by restarting the web server. This is very undesirable, and is a matter that should be handled safely by the WSRF.

ZoneSimulationWorker

The ZoneSimulationWorker service works in close relation to the ZoneSimulation described on page 78. A new thread running the simulation is started when the start method is invoked. The simulation is executed on the privilege of the web servers host environment that, in the case of WSRF.Net is the Internet Information Service IIS [IIS]. The ZoneSimulationWorker service is responsible for closing the thread when it is no longer needed. If it is not closed, it will keep running as a ghost process in the web server. The interface of the ZoneSimulationWorker can be seen on Figure 4-52.

The following list describes the methods of the ZoneSimulationWorker:

- **create**: Creates a new instance of the WSRF service
- **getManagerEPR**: Returns the service EPR as a string.
- **getGridInGamesEPR**: Returns the EPR for the GridInGamesManager service.
- **setGridInGamesEPR**: Sets the EPR for the GridInGamesManager service.
- **addGameObject**: Adds a GameObject to the Simulation.
- **addGameAction**: Adds a new Action to a GameObject in the Simulation.
- **subscribe**: Adds a recipient to the information broadcasted from the Simulation.
- **start**: Start a new thread running the simulation.
- **stop**: Stop the thread running the simulation.

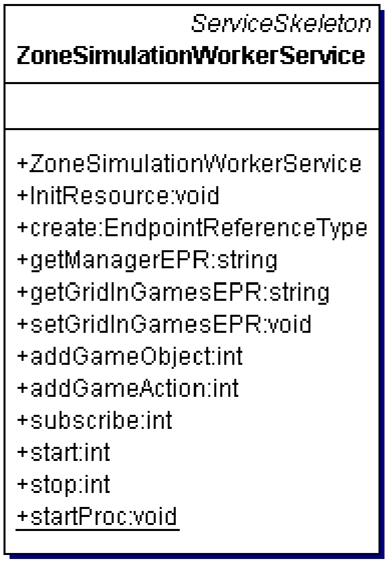


Figure 4-52, ZoneSimulationWorker interface

PlayerAgentWorker

The PlayerAgentWorker works much like the ZoneSimulationWorker. It manages a thread running a PlayerAgent. See page 88 for a description of the

PlayerAgent class. The interface of the PlayerAgentWorker can be seen on Figure 4-53. The following list describes the methods in PlayerAgentWorker:

- **create**: Creates a new instance of the WSRF service.
- **getGridInGamesEPR**: Returns the EPR for the GridInGamesManager service.
- **setGridInGamesEPR**: Sets the EPR for the GridInGamesManager service.
- **startPlayer**: Start thread running the PlayerAgent.
- **stop**: Stops the thread running the PlayerAgent.

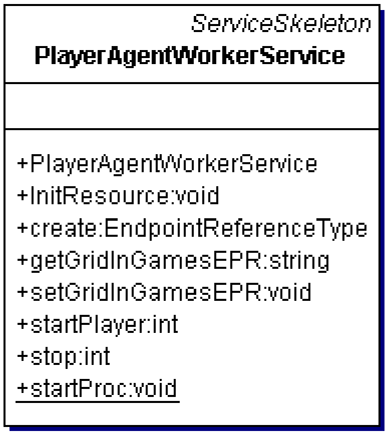


Figure 4-53, PlayerAgentWorker interface

PlayerAgentManager and ZoneSimulationManager

The two manager services are identical besides the name, so only one of them will be described here. The interface of the manager services can be seen on Figure 4-54. The manager service creates new instances of the workers and saves a reference as an EPR in a local resource. Following list describes the methods of the manager services:

- **create**: Creates a new instance of the WSRF service.
- **getWorkerCounter**: Returns the number of worker managed
- **getGridInGamesEPR**: Returns the EPR for the GridInGamesManager service.

- **setGridInGamesEPR**: Sets the EPR for the GridInGamesManager service.
- **getEPR**: Returns the EPR of the service
- **createWorker**: Creates a new worker and stores a reference to the EPR

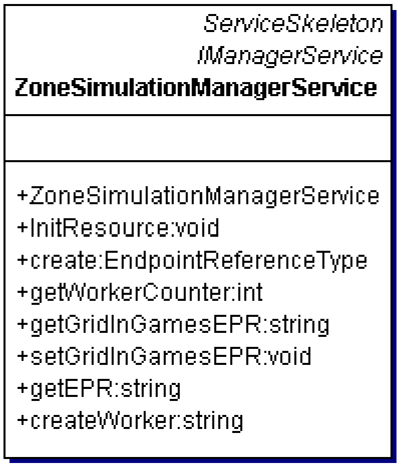


Figure 4-54, Manager Service interface

GridInGamesManager and GridInGamesConnector

The code for actually creating and using WSRF web services is complicated, and it requires a lot of specialized code. To avoid that the user of the services has to know or write that code, the (remote) Proxy design pattern is used to mask the details.

The GridInGamesManager service is the (remote) Proxy and GridInGamesConnector is the RealSubject. The Proxy and RealSubject both inherit from the same interface Subject, but that is not suitable here. Some of the return types from the GridInGamesManager service that acts as the proxy need to be processed before returning to the user. The result is that eventhough both the GridInGamesManager service and GridInGamesConnector conceptual support the same interface there are subtle differences. Figure 4-55 Show the interface of GridInGamesConnector.

```
public interface IGridInGamesConnector
{
  string CreateService(string ident, string URL);
  string GetEPR(string ident);
  void SetEPR(string ident, string epr);
  string GetThisEPR();
  void DestroyService(string ident);
  int DestroyAllServices();
  Hashtable GetEPRList();
}
```

Figure 4-55, IGridInGamesConnector interface

One of the subtle differences between GridInGamesManager and GridInGamesConnector is the implementations of GetEPRList(). Because a Hasbtable is unable to be transferred from GridInGamesManager using web services, it must be converted into a SOAP string before sending. The Hashtable is then reconstructed in GridInGamesConnector using the SOAP string. The user of the WSRF web service GridInGamesManager will never know the problems with Hashtable's not being an eligible web service return type because he only sees the GridInGamesConnector. The two implementations of GetEPRList can be seen on: Figure 4-56 and Figure 4-57.

```
public Hashtable GetEPRList()
{
  return ObjectSoapSerializer.StringToObject(_gridInGamesProxy.GetEPRList()) as
Hashtable;
}
```

Figure 4-56, GetEPRList implementation in GridInGamesConnector

```
[WebMethod]
[SoapDocumentMethodAttribute(
   "http://www.kjems.org/GridInGames" + "/GetEPRList",
   Use=System.Web.Services.Description.SoapBindingUse.Literal,
   ParameterStyle=SoapParameterStyle.Bare)]
[return: XmlElementAttribute("GetEPRList",
   Namespace="http://www.kjems.org/GridInGames")]
public string GetEPRList()
{
  return ObjectSoapSerializer.ObjectToString(EPRList);
}
```

Figure 4-57, GetEPRList implementation in GridInGamesManager

The following list describes the methods in the GridInGamesConnector:

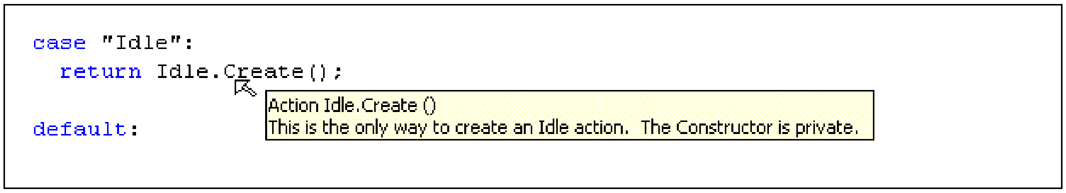
- **CreateService(ident, URL):** Creates a new instance of the service with the *URL* given as argument. GridInGamesManager saves the EPR of the created service instance under the name of *ident*.
- **GetEPR(ident):** Returns the EPR of the service saved under the name of *ident*.
- **SetEPR(ident, EPR):** If a service was created externally, a reference can be added to the GameObjectManager database of that service.
- **GetThisEPR():** Returns the EPR of the service itself.
- **DestroyService(ident):** Destroys the service saved under the name of *ident*.
- **DestroyAllServices():** Destroys all the services saved in the database, effectively closing down the server.

Code Documentation

Code documentation in the .Net languages can be done using XML tags. All public identities are tagged with some information describing it. In this project I use 4 of the tags available:

- <summary> : Short description.
- <returns> : Description of the return value from method
- <remarks> : Detailed description containing special considerations.
- <param name=”name”>: Description of input parameter *name*.

The information added in the XML tags can be used to a lot of different things. The information is available as ToolTips so if the mouse is hovering over an identity the relevant information is showed as a small popup textbox as seen on Figure 4-58**.**

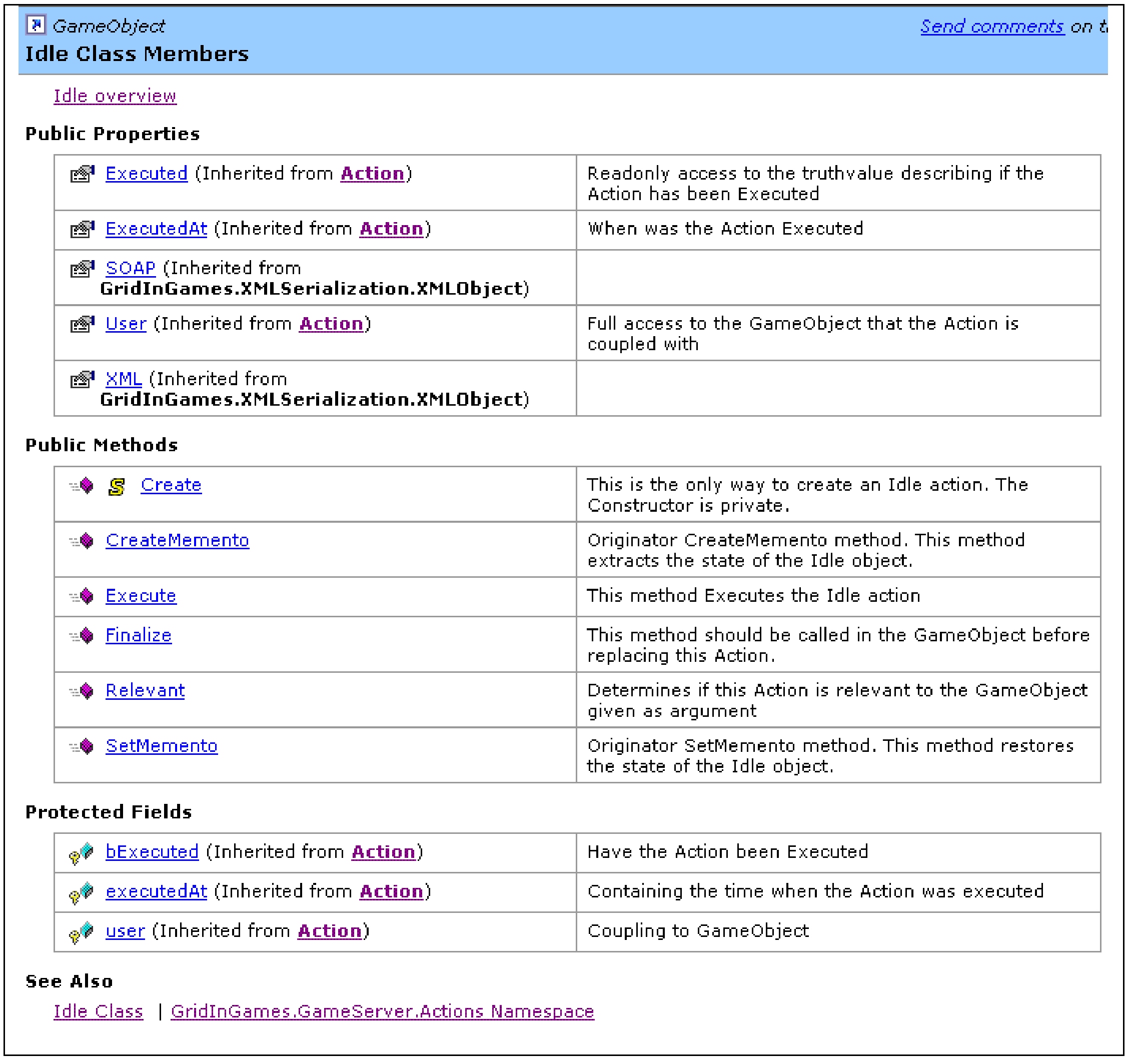


```
case "Idle":
  return Idle.Create();

default:
```

Action Idle.Create ()
This is the only way to create an Idle action. The Constructor is private.

Figure 4-58, Tooltip example

It is also possible to generate a series of documentation web pages using the XML tags as source. The XML tags are compiled into an .xml file that can be shipped with an associated library .dll file.



GameObject *Send comments on t*

Idle Class Members

Idle overview

Public Properties

Executed (Inherited from **Action**)	Readonly access to the truthvalue describing if the Action has been Executed
ExecutedAt (Inherited from **Action**)	When was the Action Executed
SOAP (Inherited from **GridInGames.XMLSerialization.XMLObject**)	
User (Inherited from **Action**)	Full access to the GameObject that the Action is coupled with
XML (Inherited from **GridInGames.XMLSerialization.XMLObject**)	

Public Methods

Create	This is the only way to create an Idle action. The Constructor is private.
CreateMemento	Originator CreateMemento method. This method extracts the state of the Idle object.
Execute	This method Executes the Idle action
Finalize	This method should be called in the GameObject before replacing this Action.
Relevant	Determines if this Action is relevant to the GameObject given as argument
SetMemento	Originator SetMemento method. This method restores the state of the Idle object.

Protected Fields

bExecuted (Inherited from **Action**)	Have the Action been Executed
executedAt (Inherited from **Action**)	Containing the time when the Action was executed
user (Inherited from **Action**)	Coupling to GameObject

See Also

Idle Class | GridInGames.GameServer.Actions Namespace

Figure 4-59, Document! X generated documentation

Reflection combined with the XML tag file contains a lot of useful information to the user of the library. An example of a third party program that uses the XML tags and reflection to generate documentation is Document! X [Document-X]. Document! X generates documentation very similar to the MSDN documentation. An example of the generated documentation can be seen on Figure 4-59. A partial example of the code used to generate the documentation in Figure 4-59 can be seen on Figure 4-60

```
/// <summary>
/// Concrete class Idle of type Action. Constructor is private
/// because the object should be created with the Create method.
/// </summary>
/// <remarks>
/// This is part of the strategy design pattern.
/// GameObject is context, Action is strategy, Move is concrete strategy,
/// Idle is concrete strategy. This is part of the memento design pattern.
/// Action is the Originator, Action.Memento is the memento.
/// The constructor is private to avoid construction of this concrete Action.
/// The reason why a conrete object of this type is undesireable is
/// because it would be possible to invoke methods in the IExecutable
/// interface on an Action decoupled from a GameObject.
/// Use the Create method to create an instance
/// of the Idle object as an abstract Action.
/// </remarks>
[Serializable]
public class Idle : Action, IOriginator, IExecutable, IAction
{
  #region Constructor
  /// <summary>
  /// This is the only way to create an Idle action.
  /// The Constructor is private.
  /// </summary>
  /// <returns>
  /// Abstract Action containing the concrete Idle object
  /// </returns>
  public static Action Create()
  {
    return new Idle() as Action;
  }
  private Idle():base(){}
  #endregion

  // Rest of the Idle implementation is left out.

}
```

Figure 4-60, Code Documentation example

4.4 Test

Introduction

The testing of the written code in this project will be done in three ways: A unit test, performance test and functionality test. The unit test uses the NUnit framework explained in 2.4.

A unit test, tests the fundamental building blocks of the system. If small implementation changes are made, the unit tests can be rerun to ensure that the code still behaves like expected.

The performance test in this project will primarily be a test on a WSRF grid service method invocation. The overhead of using grid services in this project is the most crucial to the system performance.

The functionality testing should prove that the system as a whole is working. The test will simply be to test different situations, and ensure that the program reacts like expected.

Unit Testing

Unit test is done to the public part of classes. Unit tests can be used to test if fundamental building blocks are functioning as intended. It can also be used to retest part of a program if implementation changes have been made, to ensure everything is still functioning as intended. The NUnit framework described in section 2.4 is used to perform the unit tests. Three examples of unit tests will be shown here, and the rest can be found in the appendix with the source code.

CurrentLoc

The CurrentLoc property uses system time to calculate how far an object has moved in a certain period. Results based on a system timer are not entirely exact, so it has to be insured that it lies within an acceptable range. The conceptual steps of the test can be seen on Figure 4-61 and the source code for the test can be seen on Figure 4-62.

CurrentLoc	**1.** Make a new GameObject at location (1,1). **2.** Add a move action to the GameObject to locatation (5,5). **3.** Execute the action and wait half the travel time. **4.** Test that object is located at (3,3) which is halfway between (1,1) and (5,5).

Figure 4-61, CurrentLoc Unit test steps

```
[Test]
public void CurrentLoc()
{
  const string str = "Monster";
  GameObject ma = new GameObject(str,"TestZone");
  ma.Speed=2;
  ma.LocVec.From=new Location(1,1);
  ma.SetAction(Move.Create(new Location(5,5)));
  ma.Execute();
  Thread.Sleep((int)(ma.TravelTime*1000/2));
  Location loc = ma.CurrentLoc;
  Console.WriteLine("CurrentLoc: ({0},{1})", loc.X, loc.Y);
  Assert.IsTrue(loc.X>2.9, "X < 2.9");
  Assert.IsTrue(loc.X<3.1, "X > 3.1");
  Assert.IsTrue(loc.Y>2.9, "Y < 2.9");
  Assert.IsTrue(loc.Y<3.1, "Y > 3.1");
}
```

Figure 4-62, CurrentLoc test source code

XML Serialization

The test of XML serialization is relatively easy, because and object can be converted to and from XML. The test steps can be seen on Figure 4-63, and the source code can be seen on Figure 4-64.

XMLSerialization	**1.** Make a new LocationVector from (1,2) to (3,4) **2.** Convert the LocationVector to XML **3.** Convert the XML back to a new LocationVector **4.** Test that the content of the two LocationVector are the same.

Figure 4-63, XML Serialization test steps

```
[TestFixture]
public class XMLTest
{
  [Test]
  public void Test()
  {
    LocationVector locVec = new LocationVector();
    locVec.From=new Location(1,2);
    locVec.To=new Location(3,4);
    string xmlLocVec = locVec.XML;
    LocationVector locVec2 = (LocationVector)ObjectXMLSerializer.StringToObject(
                                              xmlLocVec, typeof(LocationVector));
    Assert.IsTrue(locVec.From.X==locVec2.From.X&&locVec.From.Y==locVec2.From.Y);
    Assert.IsTrue(locVec.To.X==locVec2.To.X && locVec.To.Y==locVec2.To.Y);
  }
}
```

Figure 4-64, XML Serialization test source code

IPSender and IPReceiver

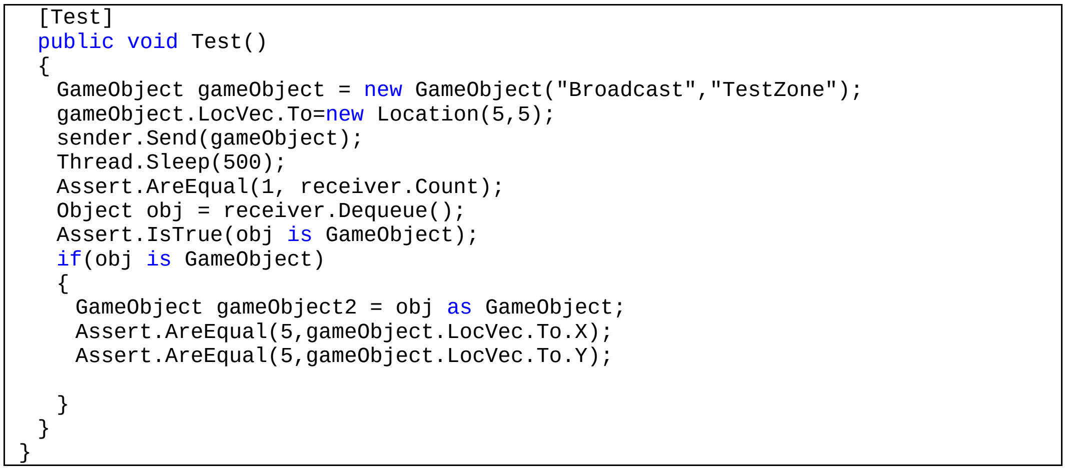
The IPSender and IPReceiver are a little more difficult to test because they use the network. To test the functionality an object is sent over the network, and it is checked that the object is the same after it has been received. This test assumes the serialization from Figure 4-63 is working.

IPSender, IPReceiver	**1.** Make a new IPSender, and IPReceiver so they can communicate with each other. **2.** Create a new GameObject with ToLoc equal to (5.5) **3.** Send the GameObject to the local receiver. **4.** Dequeue an object from the receiver. **5.** Check that the object is a GameObject **6.** Check that the ToLoc of the GameObject is (5,5)

Figure 4-65, IPsender and IPReceiver test steps

The source code for the unit test of the IPSender and IPReceiver can be seen on Figure 4-66.

```
[TestFixture]
public class UDPTest
{
  private IPReceiver receiver;
  private IPSender sender;
  [SetUp]
  public void Initialize()
  {
    sender = new IPSender(new UDPSend("127.0.0.1",25000));
    receiver = new IPReceiver(
          new IPReceiver.NetworkProtocol(IPReceiver.UDP), "127.0.0.1", 25000);
    receiver.Start();
  }
  [TearDown]
  public void TearDown()
  {
    receiver.Stop();
  }
```



```
  [Test]
  public void Test()
  {
    GameObject gameObject = new GameObject("Broadcast","TestZone");
    gameObject.LocVec.To=new Location(5,5);
    sender.Send(gameObject);
    Thread.Sleep(500);
    Assert.AreEqual(1, receiver.Count);
    Object obj = receiver.Dequeue();
    Assert.IsTrue(obj is GameObject);
    if(obj is GameObject)
    {
      GameObject gameObject2 = obj as GameObject;
      Assert.AreEqual(5,gameObject.LocVec.To.X);
      Assert.AreEqual(5,gameObject.LocVec.To.Y);

    }
  }
}
```

Figure 4-66, IPSender and IPReceiver test source code

Performance Testing

The performance of a game server is important because it dictates the response time of the server under heavy load. Having a quick response time is important for the game experience.

Grid Service Invocation

The critical point in this implementation of a game server using grid services is the invocation of grid service methods. A test has been made to determine the invocation time of grid service method. The grid service method tested is the AddGameAction in ZoneSimulationWorker, which is the grid service method, used the most in a running simulation. The AddGameAction calls the Enqueue method in ActionSequenceSingleton. The simulation is not running, so none of the added actions are executed. The invocation time of both methods was tested to determine the difference:

Method	Total invocations	Used time	Invocation time
AddGameAction	1.000	17,6s	17,6ms
Enqueue	10.000.000	7,5s	0,00075ms

The overhead of the grid service invocation is clearly the dominant part when adding a new action to the simulation. It is not surprising that a grid service method invocation is so heavy compared to a local method invocation, but it gives an idea of how many actions the system can handle per second.

Running Simulation

To test the running simulation, *n* actions are added to the simulation and all the actions are executed. The actions used will be Move. Instead of moving to the same location repeatedly, two different locations are used to move between.

	Total actions	Used time	Time per Action
Simulation	1.000.000	4,9s	0,0049ms

Sending information from the Simulation to the subscribers is relatively demanding, because the GameObject.Memento has to be converted into a SOAP string and send via UDP. Figure 4-67 shows the time used when running the Simulation in relation to how many that has subscribed to the Simulation. The relation is clearly linearly.

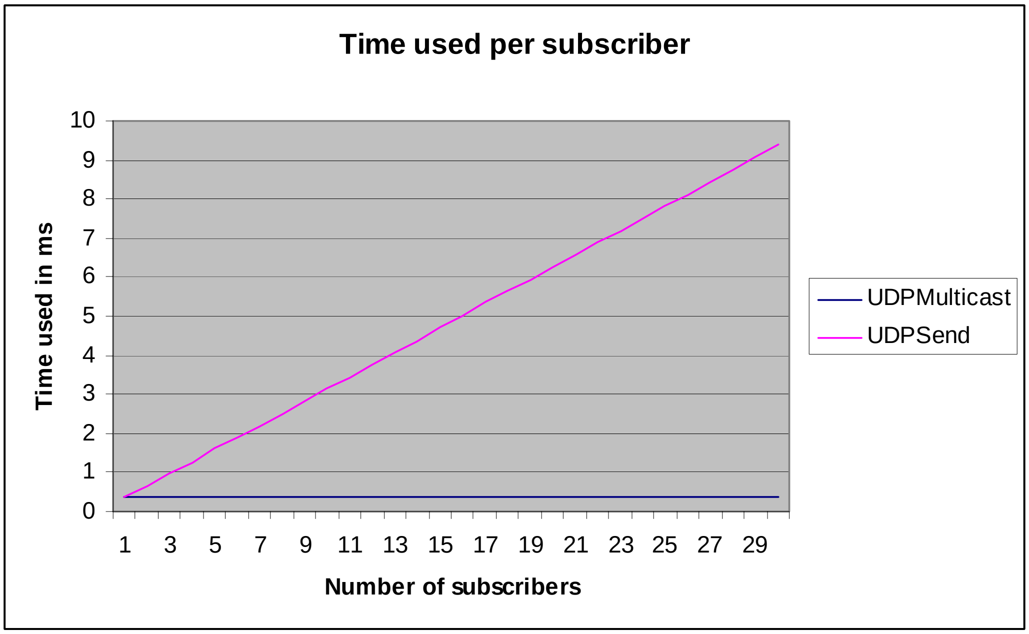


Figure 4-67, Time used for one action per subscriber

When using UDPSend the GameObject.Memento has to be converted into SOAP and send to each of the subscribers in contrast to the UDPMulticast that only have to do it once. Imagine a simulation having 60 subscribers and using the UDPSend: The time needed to send an action to all subscribers would approximately be the same as adding one action through the grid service method.

The grid service invocation is as seen considerable and should in some cases be avoided by using traditional network coding instead. An example of where traditional coding is used, is the broadcasting to all subscribers in the Simulation. The Simulation would have been very slow if all the PlayerAgents should have been notified using grid service invocation. With the test results seen here, the AddGameAction would also be a subject that could be changed to traditional network code instead. It is however hard to predict how the performance of grid service invocations using the WSRF will evolve in the future. WSRF is merely a technology preview, and there has not been done anything to improve the performance yet. The primary use of WSRF is typically applications with few service invocations and where performance is not an issue.

Functionality Testing

This test uses the final system to ensure that the provided functionality is working. The test game client that provides the interface for the functionality test is very simple, so the functionality test is also very simple. The things that can be tested are:

- Is it possible to move the player represented by a small dot around the client area?
- Do the AIPlayers move around the screen as expected?
- Do the simple relevance criteria work?

The basic functionality of the game server is working as expected.

4.5 Summary

The goal of the case study was to investigate if grid services are applicable in the development of a MMOG server. The grid service implementation used was based on Web Service Reference Framework WSRF. The task of making a fully functional MMOG is larger than what can be accomplished during the project period of a master thesis done by one person. The partial implementation seen here does however reveal many of the benefits and problems when using grid services. The problems are primarily related to performance, and an alternative method has been used to solve the problem. Instead of using, the relatively slow inter grid service communication, normal UDP is used instead. The ZoneSimulation broadcasting information is an example of that. Server/Client communication is also done using the UDP protocol. One of the bottlenecks of the system is the grid service interface that the ZoneSimulation is accessed through. Each time a new action is added to the ZoneSimulation, it is done through the grid service interface. The WSRF.Net implementation used as grid service is still very young, and not at all minded towards good performance, so future versions may improve the performance tremendously. One of the benefits from grid services is that a large problem can be divided into smaller pieces and solved on different computers using the principle of Divide and Conquer. The challenge when designing a system using grid services is to determine where the problem should be divided. Each time a problem is divided some performance will be lost in the overhead. Not dividing a problem can result in a problem being too large for one grid service instance to handle. The fact that the WSRF does not handle the lifetime of threads created in a service is troublesome. If the developer of a WSRF service uses a thread and makes a resource destruction error, the thread will continue running as a ghost process in the web server. The handling of threads or computational resources in the WSRF is defiantly an area where future research can be made.

Chapter 5

Conclusion

A short resume of the chapter summaries will be made. The project specification will be examined to outline how each of the requirements is met. Future work and improvements on this project will be discussed.

5.1 A short summary

5.2 Concluding remarks

5.3 Future research and development

5.1 A short summary

This project has analyzed how applicable grid technology is as a fundament for a game server. The WSRF is used as grid technology implementation. A simple game server have been developed using the WSRF as a foundation.

Chapter 1: Introduction

This chapter introduces fundamental concepts of grid technology and computer games. The problems with hosting and developing a computer game server are investigated. It is suggested that grid technology can be used to solve some of the problems with the hosting and development of a computer game server.

Chapter 2: Descriptions of technologies used in this project

Technologies used in the project are described. A new design pattern named Transformation pattern is developed. The problems using a very new technology like the WSRF is explained.

Chapter 3: Design Principles

This chapter describes two general concepts, one for grid application and one for computer games. The first concept describes how a grid application is designed if it should be used to solve problems using the divide and conquer method. The divide and conquer design principle is named Manager-Worker. The second design concept called Relevant Set is used to improve the distribution of game object states in a computer game server

Chapter 4: Case study: Application of grid in computer games

This chapter contains the major part of the work in this project. The chapter describes how the technologies from chapter 2 combined with the design principles of chapter 3 can be used to solve some of the problems with computer game servers described in chapter 1. It is concluded that grid services are applicable as a foundation for a game server, but it has to be designed in a way that avoids some of the performance related drawbacks. Grid services are especially well suited for problems that can be divided into small parts by using the divide and conquer design principle. It is concluded that grid services in the form of WSRF services is still too young a technology to base a final large product upon.

5.2 Concluding remarks

The job of making a game server using grid technology is very big and the time available when making a master thesis is simply not enough to cover all the details.

The true strength of grid service development will first be seen when a foundation of core services and tools have become available to the developer. Because the technology is so new, most of these tools are still on the drawing board. In the future one of the main reasons for using grid services in a game server design will be the vast availability of tools. There is a lot of focus on web development, and big companies like Microsoft and IBM are focusing many of their resources in this field. Grid- or web services will very likely be the fundament of many distributed applications in the future.

The following results are some of the most important in this project:

- Algorithms for game servers and the divide and conquer principle of grid applications were combined into a simple but functional game server.
- The use of design patterns proved to be very beneficial when creating the fundamental components of the game server.
- UML was a helpful tool when planning the project and to document different aspects of the development throughout the written report.
- The development of a game server based on grid services using the WSRF. The WSRF is not yet practically ready to be used in a fully functional game server. It was however demonstrated that the concept of using grid services in game server design is suitable.

Review of Project Specification

The project specification from chapter one, can be seen below with a remark on how each of the specification points have been reached:

A. Grid computing has to be explained. OGSA and OGSI must be presented. The WSRF must be presented. The Unified Process (UP), Unified Modeling Language (UML) and design patterns must be described.

OGSA, OGSI and WSRF are explained in section 2.3. UP is briefly described in section 1.5. UML is described in section 2.1 and design patterns in section 2.2.

B. A framework for a multiplayer game server has to be developed. The framework must be constructed as a grid application where scalability is important.

The scalability of the multiplayer game server framework is archived using the design principle d*ivide and conquer* explained in section 3.1. The entire development of the framework is described in the case study of Chapter 4.

C. A test application able to simulate a simple running computer game must be created. The test application must use the framework from B.

The simple test application is explained in section 4.3 in the subsection called *Test Game Client*.

D. Various tests of the running game server simulation of a running computer game must be made using the test application from C. The tests should help clarify if grid technology is applicable in a computer game server.

The various tests on the server simulation are explained in section 4.4. The clarification of the grid technology being applicable in a game server is discussed in section 4.5 and in this chapter.

E. It must be examined how applicable the Unified Modeling Language and design patterns are in the development of grid applications.

UML and design patterns have been used throughout the case study. UML is good to describe structure and behavior of software components. Design patterns are good as guidelines for how software components should be created, structured and behave.

5.3 Future research and development

There is a lot of future research and development that can be done in this project. Many aspects of creating a computer game server have only been partially dealt with. The technology of grid services and web services used for calculations like in a computer game will still change and develop a lot in the future. The following list shows the most important areas where future work can be done:

- Security protection against passive and active listening server/client communication.
- Security protection against network packet infusion or altering of the packets sent between sever and client.
- Algorithms solving problems with the time difference between client and server.
- Development of the Database Cache service
- Development of artificial intelligence for non-playing characters.
- Development of performance minded serialization methods for sending GameObjects and GameObject Mementoes over the network.
- Development of game specific features like more Actions and more complex GameObjects.

The WSRF is not currently designed to handle heavy calculations and spawning of new threads in the hosting environment. One of the most important areas of future research related to this project is to make WSRF services able to handle thread lifetime management better. So if WSRF is to be used, as a grid service that provides computational power, it will be required that thread management is supported or another mechanism for managing computational resources is provided.

Chapter 6

Bibliography

[XML]
Flexible text format used to store information in a hierarchy manner.
http://www.w3.org/XML

[SOAP]
The fundamental message enveloping mechanism used in Web services.
http://www.w3.org/TR/SOAP.

[OGSA-Spec]
C. K. Ian Foster, Jeffrey M. Nick, Steven Tuecke, "The Physiology of the Grid," [online], 2002,
http://www.gridforum.org/ogsi-wg/drafts/ogsa_draft2.9_2002-06-22.pdf

[OGSI-Spec]
C. K. Ian Foster, Jeffrey M. Nick, Steven Tuecke, “Open Grid Services Infrastructure (OGSI)” [online], 2003, http://www.gridforum.org/ogsi-wg/

[GT]
Globus Toolkit [online], http://www.globus.org/

[OGSI.NET]
OGSI.NET, [online], http://www.cs.virginia.edu/~humphrey/GCG/ogsi.net.html

[MS.NET]
MS .NET Grid, [online]
http://www.nesc.ac.uk/action/projects/project_action.cfm?title=145

[AA]
U.S. Army, ”America’s Army”,[online], http://www.americasarmy.com/

[Butterfly]
IBM, “Butterfly.net: Powering Next-Generation Gaming with Computing On-Demand”, [online], 2002, http://www.butterfly.net/platform/technology/idc.pdf

[GoF]
E. Gamma, R. Helm, R. Johnson, J. Vlissides, ”Design Patterns, Elements of Reusable Object-Oriented Software”, 1995.

[UML-Larman]

Craig Larman, “Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and the Unified Process”, 2002

[Web Services]

Ferguson, D., Lovering, B., Shewchuk, J., Storey, T. Secure, Reliable Transacted Web Services
http://www-106.ibm.com/developerworks/webservices/library/wssecurtrans/

[WS-Addressing]

WS-Addressing, an XML serialization and standard SOAP binding for representing network wide “pointers” to services.
http://www.ibm.com/developerworks/webservices/library/ws-add/

[WS-Arch]

The W3C Web Services Architecture working group, public draft, August 2003.
http://www.w3.org/TR/2003/WD-ws-arch-20030808/

[WSRF-Spec]

The Web Service Reference Framework specification. March 2004
http://www-106.ibm.com/developerworks/library/ws-resource/

[WSE-Spec]

Microsoft Web service Enhancement
http://msdn.microsoft.com/webservices/building/wse/

[GT-Presentation]

Globus Toolkit presentation of WSRF
http://www.ogsadai.org.uk/docs/OtherDocs/GridsAndWebServices.pdf

[WSRF.Net]

WSRF implementation in C# and .Net.
http://www.cs.virginia.edu/~gsw2c/wsrf.net.html

[WSRF-Expln]

Modeling stateful resources with web services
http://www-106.ibm.com/developerworks/library/ws-resource/ws-modelingresources.pdf

[Event]

Event driven programming paradigm
http://www.fact-index.com/e/ev/event_driven_programming.html

[WSRF.Net-Tur]

WSRF.NET developer tutorial
www.cs.virginia.edu/~gsw2c/WSRFdotNet/WSRF.NET_Developer_Tutorial.pdf

[NUnit]
NUnit Framework and documentaion
http://www.nunit.org/

[DCGC]
Dansk Center for Grid Computing
http://www.dcgc.dk/

[e-Science]
e-Science budget distribution on grid computing.
http://www.nesc.ac.uk/talks/mpa/ResearchGridExperiencesPanelMPA20040609.pdf

[Game-Genres]
Description of the most common computer games genres
http://en.wikipedia.org/wiki/Computer_game_genres

[Korea-BB]
Operator Source, "South Korea", [online], 2003, http://www.point-topic.com/content/operatorSource/profiles/South+Korea/South+Korea+broadband+overview+0310.htm&comp_id=612

[Linage1]
John Barrett, "Koreans know how to play", [online], 2003, http://www.eurescom.de/message/messageSep2003/Online_gaming_Koreans_know_how_to_play.asp

[Linage2]
Benjamin Fulford, "Koreas weird wired world", [online], 2003, http://www.forbes.com/technology/free_forbes/2003/0721/092.html?partner=newscom

[Consoles]
David Carnoy, "The game developer's take on Xbox Live vs. PS2 Online", [online], 2002, http://att.com.com/4520-3423_7-5021357-2.html?legacy=cnet

[Online-Gaming1]
Jessica Mulligan, "Online Gaming: Why Won't They Come?", [online], 1998, http://www.gamasutra.com/features/business_and_legal/19980227/online_gaming_why_intro.htm

[Online-Gaming2]
Paul Palumbo, "Online vs. Retail Game Title Economics", [online], 1998, http://www.gamasutra.com/features/business_and_legal/19980109/online_retail.htm

[Object-State]

Rick Lambright, “Distributing Object State for Networked Games Using Object Views”, 2002, [online], http://www.gamasutra.com/resource_guide/20020916/lambright_01.htm

[Document-X]

Third-party software that generates documentation files based on reflection and XML tags.
http://www.innovasys.co.uk/products/documentx.asp

[IIS]

Microsoft Internet Information Server.
http://en.wikipedia.org/wiki/IIS