Gamesystem

E2-204

19th December 2003

This project was written during the DAT3 semester 2003, at Aalborg University, Department of Computer Science. It was written as part of the course Distributed Systems(DS), and was written by: Carl Christian Sloth Andersen, Tim Boesen, Thomas Pryds Lauritsen, Tommy Rolchau Mathiesen, Jakob Rutkowski Olesen and Dennis Kjærulff Pedersen (Group E2-204). The group was supervised by Piotr Makowski.

Carl Christian Sloth Andersen

Tim Boesen

Thomas Pryds Lauritsen

Tommy Rolchau Mathiesen

Jakob Rutkowksi Olesen

Dennis Kjærulff Pedersen

Synopsis

Our main objectives in this project are:

- To learn about the Java 2 Enterprise Edition(J2EE) platform.

We want to gain a general understanding of the Java 2 Enterprise Edition platform, and a detailed understanding of some of the integrated technologies.

- To create a distributed framework to handle turn based games using J2EE.

We want to create a distributed framework for turn based systems, using some of the technologies available on the Java 2 Enterprise Edition Platform. We will ensure that the system has a high degree of availability, enforces reusability, is scalable, is usable and is transparent in terms of communication.

Contents

1 Introduction **3**

1.1 Goals . . . 3

1.1.1 Problem Area . . . 4

1.2 Structure of the report . . . 4

1.3 Prerequisites . . . 5

2 Technology **7**

2.1 In the Beginning . . . 7

2.2 Hypertext Transfer Protocol . . . 7

2.2.1 HTTP Messages . . . 8

2.2.2 HTTP Advantages and Disadvantages . . . 8

2.3 As Time Went On . . . 9

2.4 Java Naming and Directory Interface . . . 10

2.4.1 Naming and Directory Services . . . 10

2.4.2 Contexts . . . 10

2.4.3 JBoss Naming – a JNDI Provider . . . 10

2.5 Dynamic Content . . . 11

2.6 Architectural Frameworks For Enterprise Applications . . . 11

2.7 Java 2 Enterprise Edition . . . 12

2.8 Java Server Pages and Servlet Technology . . . 13

2.8.1 Properties of Servlets and JSP . . . 14

2.9 Communication Within Applications – Messaging . . . 14

2.10 Java Message Service . . . 14

2.10.1 Asynchronous and Synchronous Message Reception . . . 15
2.10.2 Message Domains . . . 15
2.10.3 Message Selectors . . . 16
2.10.4 Message Types . . . 16
2.11 Abstract Thinking . . . 17
2.12 Enterprise Java Beans . . . 17
2.12.1 Session Beans . . . 17
2.12.2 Entity Beans . . . 20
2.12.3 Message-Driven Beans . . . 21
2.12.4 Deployment Descriptors . . . 22
2.12.5 JNDI and J2EE . . . 22
2.13 Other Frameworks . . . 22
2.13.1 CORBA . . . 22
2.13.2 Microsoft .NET . . . 23
2.14 Summary . . . 23

3 A Model of the System 25

3.1 General Concepts . . . 25
3.2 Prototyping . . . 26
3.3 Requirements . . . 26
3.3.1 Authorization and Authentication . . . 27
3.3.2 Joining a Game . . . 28
3.3.3 Playing a Game . . . 29
3.4 Classification of Objects . . . 30
3.5 Object States . . . 31
3.5.1 User and Registration Objects . . . 31
3.5.2 Room and Table Objects . . . 32
3.5.3 Moderator Object . . . 33
3.6 Object Interactions . . . 33
3.6.1 User Registration . . . 34
3.6.2 Joining and Leaving Game . . . 35
3.7 Summary . . . 36

4 Distribution of Data and Functionality 37
4.1 Thin and Thick Clients . . . 37
4.1.1 Thin Client . . . 37
4.1.2 Thick Client . . . 38
4.1.3 Game System . . . 39
4.2 Stateless and Stateful Servers . . . 40
4.2.1 A Stateless Server . . . 40
4.2.2 A Stateful Server . . . 41
4.3 Summary . . . 41

5 Game System Framework 43
5.1 Features . . . 43
5.2 Framework . . . 44
5.2.1 Inheritance Model . . . 44
5.3 Summary . . . 45

6 Web Applications and Communication 47
6.1 Web Applications . . . 47
6.2 Communication . . . 48
6.2.1 Servlet Layer . . . 48
6.2.2 Messaging and Enterprise Beans . . . 49
6.2.3 Sending and Retrieving Information . . . 50
6.2.4 Messages . . . 52
6.3 Server Interface . . . 53
6.3.1 Stateful or Stateless Session Bean . . . 55
6.4 Summary . . . 55

7 User Authorization and Authentication 57
7.1 User Interface . . . 57
7.2 Persistent Storage of Users . . . 57
7.3 Implementing the Entity Bean . . . 59
7.4 Finding Users in Storage . . . 60
7.5 Authentication of Users . . . 61
7.5.1 Remembering Authentication . . . 62
7.6 Summary . . . 62

8 Game Server Framework **63**
8.1 Games 63
8.2 Managing Rooms and Tables 64
8.2.1 Rooms 64
8.2.2 Tables 67
8.2.3 Sitting Down at a Table 68
8.3 Game Configuration 70
8.4 Framework 70
8.5 Stateful and Stateless Server 72
8.6 Transactions 72
8.7 Summary 73

9 Game Client Framework **75**
9.1 Client Requirements 75
9.2 Lifetime of the Client 76
9.3 The Room Applet 77
9.3.1 Game Properties 78
9.4 Sending and Receiving Messages 78
9.4.1 Sending Messages 78
9.4.2 Receiving Messages 79
9.5 Handling Messages 80
9.6 User Interface 81
9.7 The Framework 81
9.8 The Distribution of Functionality and Data 83
9.9 Summary 84

10 Tic Tac Toe **87**
10.1 The Game 87
10.2 Implementing The Game 87
10.2.1 Game Events 88
10.2.2 The Board 90
10.2.3 Implementing the Server 90
10.2.4 Implementing the Client 90
10.3 Summary 92

11 Conclusion **93**

11.1 J2EE . . . 93

11.2 Distributed Game Framework . . . 94

11.3 J2ME . . . 95

11.4 Other Possible Solutions . . . 95

Chapter 1

Introduction

Information technology has an increasing impact upon our lives. The impact is so great that we start to take many things for granted. This includes different services offered via the Internet, such as weather forecasts, online ordering of different goods, and online banking. But when you start thinking about what is actually going on, things get a bit more complicated. How does it actually work? How is it possible for large enterprises to manage their services?

During the last decade or two, several ways to accomplish the task of creating these enterprise systems have emerged. We will during this report explore one of these technologies, or architectural frameworks as they are also called. First, let us have a look at the main objectives of this report.

1.1 Goals

Seen from a broad point of view, we have two main goals in this project, as stated in our synopsis:

- To learn about the Java 2 Enterprise Edition(J2EE) platform.
- To create a distributed framework to handle turn based games using J2EE.

We would like to explore the J2EE platform, and see what it offers. As such, the platform is quite extensive, and it would not be possible to explore it all in detail, in the amount of time we have available for this project. Therefore, we have chosen to give a general overview of what this platform offers, and dwell on some specific parts.

We will elaborate on those part of J2EE we will be using to develop a distributed framework for turn based games. Why a game system one might ask? We have chosen so because a game system represents a concept which is possible to implement, while we explore the technologies of the platform. A simple game as Tic Tac Toe does not require too much implementation, and therefore suits our needs, since we want to focus on the technologies in use, and not on developing a fancy graphical user interface, and complicated business logic.

1.1.1 Problem Area

The term “A distributed framework for turn based games” is a bit of a mouthful. Let us briefly explain what we mean by that. By distributed we mean, by communication through a network of some kind. It also refers to games with at least two users, since there would be no point in having a single player distributed game. The reason we have chosen turn based games is that they are quite simple to handle, compared to e.g. real time games.

A framework is a set of building blocks for doing something, in this situation to create games. Most games have several things in common, e.g. a large number of games are turn based. You could build each game from the bottom up, but one thing that object oriented technology offers, is the opportunity to gain benefit from general concepts. Why not just build the things that all these games have in common just once, and then only implement the parts which are specific for each particular game? It makes it a lot easier to implement new games. This could be generalized under the terms of reusability and scalability, i.e. we reuse the code for each game, and thus makes it a lot easier to add more games.

The point of making such a system is that someone should use it. Both game developers and gamers. To achieve that, the system should be usable from the developers point of view. There also have to be a high availability to ensure that there are enough gamers actually playing, and not least, details such as how the communication is achieved should be hidden away.

This is, from our point of view, the most important features of a system like the one we are going to design. Therefore, they will be our main design criteria, and we have a belief that the J2EE platform will help us achieve these goals.

1.2 Structure of the report

We will give an overview of each of the chapters in the report. The purpose is to give the reader a taste of what is coming, but also, this section can be used as a reference while reading the report. Each chapter is listed, followed by a description.

Technology In this chapter we give an introduction to the different technologies we use in the project. We explain these in the context of history, and other similar technologies. The intention is not to give a complete introduction to everything available, but to provide the reader with an overview of the used technologies.

Modelling The main objective of this chapter is to develop a model for the system. With starting point in prototyping, we develop a model using the modelling language UML. Also, it gives a more detailed overview of the system, although seen from a general point of view.

Distribution of Data and Functionality Data distribution refers to where the different components of software and data will be located, according to the server and client. We will discuss thick and thin clients and stateful/stateless servers.

Game System Framework A general description of the game framework is presented, and we discuss how game developers can access framework features.

Web Applications and Communication This chapter gives an introduction to web applications and how communication is dealt with in the system. The communication between clients and the server is discussed, and an interface to the server is constructed.

User Authorization and Authentication Here we discuss how users can identify themselves to the system. We look at different approaches for how this can be achieved using various J2EE features.

Game Server Framework The design of the server framework is introduced here. We discuss how the framework is designed, and how it is intended to be used.

Game Client Framework A discussion on the client framework will end the design process of the game system.

Tic Tac Toe This chapter provides a description of a game example, implemented using the game framework. We construct the game Tic Tac Toe.

Conclusion The conclusion will sum up on the development of the game framework, and our experience using J2EE. We will compare the actual results with our main objectives.

1.3 Prerequisites

It is assumed that the reader has some general knowledge of the concepts of programming, databases and the Internet.

When talking about J2EE applications, we need a J2EE container. We have chosen to use JBoss[1] for this. Also, we need a database, and for this purpose we have chosen mySQL[2]. Since we only have a few topics in the project which is specifically connected to the two products, and it is possible to change each product, with only minor changes, we have chosen not to discuss these issues any further.

[1] An open source Java web container. A brief overview can be found at: http://www.jboss.com

[2] An open source database. A brief overview can be found at http://www.mysql.com

Chapter 2

Technology

Since the very beginning of the Internet, the main objective has been communication. Along the path, many advances have been made and new technologies have emerged. Some technologies have been replaced while others have been improved to stay fit for their purpose.

As technologies have been developed, and have given us new opportunities, also our way of thinking of communication has changed. At first, the main thing was just being able to have two computers communicating with each other, but as this was accomplished, many new ideas emerged.

2.1 In the Beginning

Until around the nineties, communication had been rather simple. You had the opportunity to connect to other computers, work on a distant computer, fetching and sending files from them, and similar operations. Since it was a quite heterogeneous environment, many different hardware platforms and software applications were in use. This made it difficult to exchange documents between users. There was a need to represent documents in a way, so that everyone could read them without trouble. Then the World Wide Web (WWW) came up. Through the Hypertext Transfer Protocol (HTTP), it was now possible to read other people's documents via the WWW in an easy manner. Let us take a look at how it works.

2.2 Hypertext Transfer Protocol

HTTP[1] is a relatively simple client-server protocol. The client sends a message requesting something from the server and then waits for a response. The HTTP protocol uses the underlying TCP protocol to send its requests/responses, so arrival or loss of messages is not of any concern to the HTTP protocol. If things should go wrong, the HTTP protocol will simply timeout and the client will receive an error message.

[1]For a complete description of the HTTP protocol see [FIG$^+$99].

2.2.1 HTTP Messages

An HTTP message is either a request from the client to the server, or a response from the server to the client as demonstrated in figure 2.1.

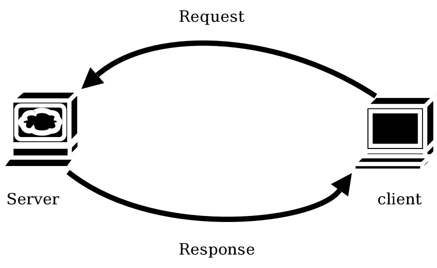


Figure 2.1: HTTP Request/Response.

The HTTP message consists of three parts, namely the status-line, the headers and the body-message. On a request message, the status-line consists of a method invoker, a Request-URI that identifies the resource upon which the request should apply, and a version field, telling which HTTP version that the client is expecting. The method invoker could be one of the methods listed in table 2.1. The response status-line contains the HTTP version, a status code, and a phrase shortly describing the code given (e.g. the server would respond with a status code 200 and a phrase 'OK' if the request were a success).

Operation	Description
GET	Retrieve information from a document
HEAD	Retrieve meta-information about a document
POST	Request the server to accept some data

Table 2.1: Some common methods for HTTP/1.1.

The header in an HTTP message consists of a field-name and a field-value. One could say that a header is a kind of rule given by either the client or the server. For example, a header could tell the client not to fetch documents that it is already keeping as cached data.

The last part, the body-message, is where the "payload" of the message is placed (e.g. file contents or query data). Figure 2.2 illustrates the parts contained in the two types of HTTP messages.

2.2.2 HTTP Advantages and Disadvantages

The HTTP protocol is stateless and does not follow the concept of having an open connection between server and client, so it has to reopen TCP connections whenever needed. Having to reopen TCP connections regularly is quite expensive in terms of performance. Earlier versions

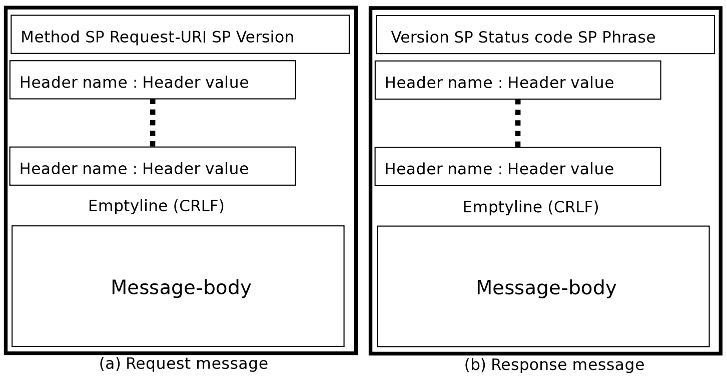

Method SP Request-URI SP Version
Header name : Header value
Header name : Header value
Emptyline (CRLF)
Message-body
(a) Request message
Version SP Status code SP Phrase
Header name : Header value
Header name : Header value
Emptyline (CRLF)
Message-body
(b) Response message

Figure 2.2: The Request/Response messages.

of the HTTP protocol had to reopen a connection for each request and response message, and since a document may consist of parts from different files, fetching a document could be a slow experience. In HTTP/1.1[2], this has been optimised with a persistent connection, where the clients are able to make several requests, without having to wait for each response (this is also referred to as pipelining), before the connection is closed. This obviously reduces the number of TCP requests, and the latency of reopening connections is thereby reduced.

Although optimised in different ways, the HTTP protocol still suffers from slower performance compared to a low level connection, due to the opening/closing of connections and the overhead of the message size. Therefore, if latency is an issue, one would have to reconsider other options than the HTTP protocol. So why even bother using HTTP messages? One of the main reasons is firewalls. Most servers today (also clients) are behind firewalls, leaving only a small number of ports open for incoming traffic. Usually the web-port (80) is open and thereby allowing HTTP messages to pass through.

2.3 As Time Went On

As the Internet grew more and more popular, the number of machines connected to it increased. A new kind of concept had emerged: Client-server architecture. Client computers contact serving computers in order to get hold on different kind of information. A central issue was how to contact the machine from which you wanted information. From early time, all machines had a unique name and number, by which they could be identified. To get hold on these names, you had to get a list from some server via File Transfer Protocol (FTP). But this way soon showed out not to be very scalable, and then the Domain Name Service (DNS) was invented. Naming services have later shown to be useable in many other situations, and that is probably why Java supports this through its JNDI interface. Let us take a closer look at naming services with starting point in JNDI, one of the technologies used in this project.

[2]HTTP/1.1 is the newest version of the HTTP protocol

2.4 Java Naming and Directory Interface

The Java Naming and Directory Interface (JNDI) is an Application Programming Interface (API) which enables naming/directory service functionality in applications written in Java. As such, it consists of two parts. It enables the user to access existing general naming services, such as DNS, but also enables programmers to implement a JNDI provider through the Service Provider Interface (SPI), i.e. a more specific naming service e.g. JBoss' naming service.

2.4.1 Naming and Directory Services

The concept of a naming service is simply to associate a name with an object. An example of such a naming service is the DNS. The name (a value such as: `www.cs.auc.dk`) of some domain is mapped to its IP address (a value such as: 130.225.194.199). When you need the IP address, you look it up by using the name, which is also called resolving the name. Most people have a hard time remembering long numbers. Therefore, it seems as a good idea to refer to the number via a logical, easy-to-remember name such as the domain name. When a name is associated to an object, it is called a binding.

A directory service is an extension to the naming service. It allows the objects to have attributes. An example: If you want to locate the object "John Doe", you may have his first name and his e-mail address only. Then, you can lookup the object using these two attributes. This enables you to lookup an object even though you do not know all details about it. Another option is to look up a list of all objects which adhere to certain criteria.

2.4.2 Contexts

Context is a central term when talking about naming services. A context is a set of bindings, and each context has a naming convention that specifies which syntax the name must follow, i.e. the name that you want to bind. It is the context which provides the user with functionality such as lookup and binding. In the DNS example mentioned previously, the `.dk` domain is a context. A DNS domain named relative to another one is called a sub context.

2.4.3 JBoss Naming – a JNDI Provider

We have chosen to use JBoss as our application container. Here, we will give a brief introduction to how JBoss actually manages its contexts. The JBoss naming service is the service provider interface implemented by JBoss, and thus the part that manages the contexts.

The JBoss naming service may, in general, hold three different kinds of context.

- `java:`
- `java:comp`

- All others

The `java:` name space is only visible under the JBoss virtual machine. `java:comp` and its sub contexts are accessible only to application components located here, and it may be considered as a local name space. For example, each component located under `java:comp\env\` , will each have their own environment under this context. This means that each component can have a `java:comp\env\site` context, where component1's site is referring to the IP address of the server on which it is running, and component2's site is referring to the physical location of the server on which it is running. All other name spaces can be accessed by remote clients as well, and it may be seen as a global name space. An example of its use is an Enterprise Java Bean home interface, which is globally accessible.

2.5 Dynamic Content

For a long time, the contents of web pages had been static, but then the Common Gateway Interface (CGI) appeared. This was an extension to HTTP servers, that allowed you to make calls to programs located on the web server. These programs could e.g. create a web page and return it to the client. CGI supported both compiled languages such as C and C++ but also interpreted ones like Perl, and thus it opened a whole new world.

Up through the mid-nineties new languages such as PHP Hypertext Preprocessor (PHP)[3] and Active Server Pages (ASP) came up. These are also referred to as server side scripting languages. The languages that could be used with CGI were not made especially for the web, even though Perl has quite some support for web programming. One of the more cumbersome things was to write out the actual HTML code for the web page being generated. The new languages did not suffer from this, since they were made especially for this purpose, and they could actually be embedded into HTML documents. Just as CGI programs, these server side script languages are, as the name implies, executed on the server.

Another interesting language in this context is Java Server Pages (JSP), which is a part of the Java 2 Enterprise Edition (J2EE) framework. To understand JSP, it should be seen in context with another technology under J2EE, and therefore it would make more sense to take a general look at the J2EE framework before we go into detail with JSP.

2.6 Architectural Frameworks For Enterprise Applications

An architectural framework is a rather abstract term. Let us try to break it down to something more specific. When we talk about the architecture of a program, we are referring to the way that the program is built. As research within the area of software engineering has shown, it does make sense to structure the way you develop an application, especially when dealing with larger software applications. A framework is something which is made ready for you,

[3]The name is recursive, and is not a spelling error. :)

to support you in some task, in this situation building a software architecture. Therefore, we could say that an architectural framework is ready-made building blocks made to assist you in building software applications. Examples of such frameworks are the Common Object Request Broker Architecture (CORBA), Microsoft .NET, and J2EE. All three frameworks are designed for helping you in dealing with issues connected to building large, distributed enterprise applications.

What do these frameworks do for you? Of course this differs from framework to framework, but the general idea is that they take care of functionality, such as security or network communication, and thereby enable you, as the developer, to stay focussed on the main issue, which is to build your business logic into your system. Instead of creating and maintaining your own framework for network communication, transaction handling, security, etc., you can access functionality through the framework and let it deal with the cumbersome details.

Using a framework enables faster development and code with less errors. It also supports the idea of reuse of code, since you are using existing components to implement your software, and you could even develop your own components and let others use them.

Since our focus is J2EE, we will first have a look at that, and then have brief discussions about CORBA and .NET.

2.7 Java 2 Enterprise Edition

J2EE is an architectural framework, which supports the philosophy of Rapid Application Development (RAD), that enables developers to create large, fast, and reliable, distributed systems. It is a rather large framework supporting the user in many different tasks. Besides the ones that we are working with, it also supports writing applications that make use of XML[4], authentication and authorization, and emails, to mention a few.

J2EE is far too large a technology to cover completely within the frames of this project. Instead, we will only look at those that we use in our project.

- Java Servlet Technology.
- Java Server Pages Technology (JSP).
- Java Messaging Service (JMS).
- Enterprise Java Beans (EJB).

We will give an overview of these technologies in the following sections. Earlier, we were talking about JSP, but to understand what that is, we also need an introduction to servlets.

[4]XML – Extended Markup Language. A language to define the contents of a document. For further explanation see [BPSMM00]

2.8 Java Server Pages and Servlet Technology

JSP and servlets are Java's server side web application technologies. At first, only servlets were available. They are small programs running on the server side, as opposed to applets[5].

With servlets you can do a lot of things that are normally not allowed from applets, such as contacting the underlying file structure of the server, or communicating with other servers around the world. Servlets can also be used to generate dynamic web pages, but code-wise that is rather tedious, because you have to generate every single line of the HTML page from the servlet. That is why JSP was invented. JSP has its own syntax, and is written directly into an HTML document. You request a JSP document the same way that you request an HTML document. Actually JSP pages are transformed into servlets and compiled the first time that they are requested from a client. Figure 2.3 shows this.

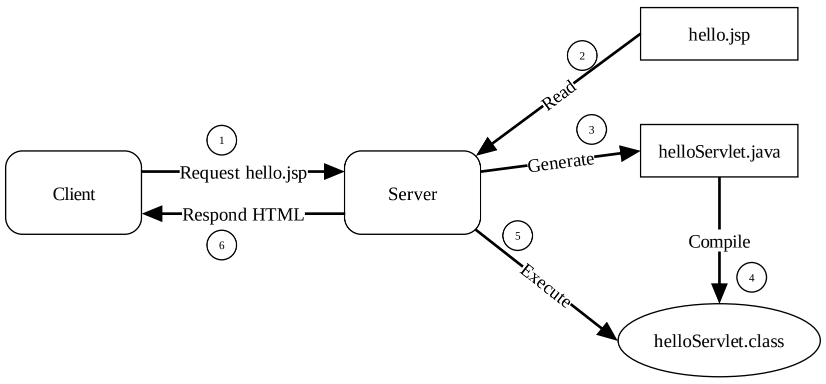


Figure 2.3: First request for a JSP page.

After receiving the request for the JSP page, the servlet is run on the server and the resulting web page is returned to the client. The technology also allows servlets to be called from the client. Usually this is done from within a form, where the action parameter points to a servlet.

JSP is usually thought of as the environment where you write the presentation part of the application, whereas servlets are thought of as the environment where you write the functionality for the application. But when using other functionality containers, both JSP and servlets may be seen as part of the presentation layer.

When writing a JSP page, the HTML code is written as-is, whereas actual Java code, import statements, and other logic require some special XML-like tags to encapsulate it. These tags decide the placement in the corresponding servlet of the seen statements, e.g. import statements in the top of the document, and regular code in the body of the called method. This structure makes JSP easy to use for HTML programmers, and with the use of Java code, the pages become dynamic.

[5]Applet – a small program that can be made available through the web, and is executed on the client computer

2.8.1 Properties of Servlets and JSP

Receiving information from an HTML page is done by e.g. a POST or a GET request. The servlet handles this request with a corresponding HTTP `doPost` or `doGet` method. These methods have a request object and a response object.

The request object can contain important information about the client, such as parameters from a form or whether the user has been authenticated. The response object is used to reply to the client. In this object, the information to be transmitted to the client is stored.

An interesting control mechanism is that of redirect/forward. These commands allow the servlet/JSP page to forward a request to another URL. An application of this is to let a single servlet serve as a junction point for several pages, and then let this handle the redistribution of requests.

Currently, JSP is a very popular choice when building dynamic web applications, because it allows fast, easy, and low-prised development. It is a robust technology, built on top of a robust, fast expanding, platform independent programming language. The specification for JSP and Servlets has become freely available in the hope that it will encourage developers to implement support for it on their application servers.[6]

2.9 Communication Within Applications – Messaging

So far, we have looked at the concept of communication in a way that mostly concerns how we get the information delivered to the user. Another important thing is how communication goes on within a software component, or perhaps between two applications. This can be achieved in several ways, and one of them is called messaging. This enables two clients to communicate through some agent in a loosely coupled manner. The agent enables the client to create, send, retrieve, and read messages, and it works as a kind of buffer. Thus, the two clients need not be aware of each other, but they just have to agree upon the communication format. J2EE has its own messaging service API, called Java Messaging Service (JMS).

2.10 Java Message Service

JMS[7], is an API that allows the programmer to use messaging in a Java application. The API ensures messages to be delivered once and once only.

In JMS, both senders and receivers of messages are called clients; i.e. senders are *not* referred to as servers. Instead, JMS has a JMS provider, which is the system that implements the JMS interfaces and keeps control of sent and received messages from clients. Messages are kept in JNDI name space.

[6]This paragraph is inspired by the site: [sun03].
[7]See [Haa02].

2.10.1 Asynchronous and Synchronous Message Reception

As a whole, JMS is asynchronous in that e.g. the sending client is not blocked until the message is successfully received in the other end. However, reception of messages can be done in both an asynchronous and a synchronous way.

Asynchronous Message Reception

From the receiver's point of view, JMS is asynchronous, i.e. a receiver does not have to request for messages and thereby halt the system for an amount of time until the message is received. This way, it is also not necessary to implement routines for regular checks for new messages; they can be delivered automatically at the receiver.

This is accomplished by registering a message listener which provides asynchronous message reception, in that the JMS provider invokes the implemented message listener's `onMessage` method when a message awaits a client.

Synchronous Message Reception

As an alternative, it is possible to receive messages synchronously by invoking the blocking `receive` method. This is done instead of registering a message listener.

2.10.2 Message Domains

There are two kinds of message domain available; point-to-point and publish/subscribe.

Point-to-Point Messaging

In point-to-point messaging, messages are targeted at one specific receiver. Messages are stored in queues until the receiver has acknowledged its reception or the message expires, even if the client is not present at the time of sending. This is shown in figure 2.4.

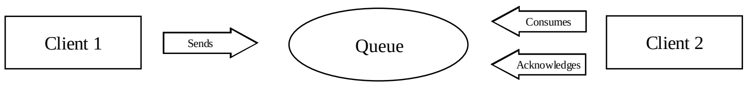


Figure 2.4: Point-to-point messaging.

Publish/Subscribe Messaging

In publish/subscribe messaging, messages are targeted at a topic to which clients may subscribe. When a client sends a message to a topic, the message stays in JNDI name space until it has been delivered to all clients subscribed to that topic. Therefore, the sending client does not have to know anything about the clients receiving its messages; this is handled by the JMS provider. However, since no recipients are specified, a client cannot receive a message sent before it subscribed to the topic. This is shown in figure 2.5.

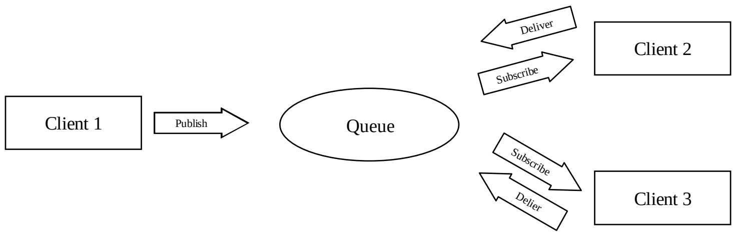


Figure 2.5: Publish/subscribe messaging.

2.10.3 Message Selectors

A receiving client might want to receive particular messages, only. These can be filtered out by using a message selector. A message selector is a string containing an expression in a format that resembles that of SQL[8] rather much. The message selector string can be provided as an argument to methods creating receiver or subscriber objects, telling the JMS provider to relay messages that fit that message selector expression only.

2.10.4 Message Types

There are six different JMS message types available, whereof one is an empty message:

TextMessage: wraps around a Java String object.

MapMessage: contains a set of unordered key/value pairs[9].

BytesMessage: is a stream of bytes.

StreamMessage: is a stream of Java primitive type values.

ObjectMessage: wraps around any Serializable Java object.

Message: is an empty message (i.e. without a body).

[8]SQL (Structured Query Language) is an ANSI standard computer language for accessing and manipulating databases. For an introduction see http://w3schools.com/sql/.

[9]A key is a Java String; A value is a Java primitive type.

2.11 Abstract Thinking

As time has passed, from the introduction of the Internet until now, more and more abstractions have been introduced. As systems grew in complexity, there was a need to get it all to a conceptual level, i.e. to focus on the large picture. Object oriented programming languages and development philosophies support this idea. An advantage of this way of thinking is the fact that it is much easier to reuse components. One way of making large, reusable, distributed components within the J2EE framework is through Enterprise Java Beans.

2.12 Enterprise Java Beans

It is possible to create large distributed systems using only servlets. However, the only way to use such a system is through the HTTP protocol. Even though a lot of technologies (e.g. JMS and JDBC) are accessible from a servlet, we often want more than just accessibility.

An example is managing data stored in a database. Using JDBC directly from a servlet requires that the developer handles the consistency between data loaded into memory and data in the database. The developer could also encounter a problem if changing database system, since the SQL constructed might not be portable. If the container could take care of such problems, then it would allow for even faster application development.

J2EE provides a set of specialised server components called enterprise beans. Each bean type has a unique purpose, like handling data consistency and portability as mentioned above. There are currently three types of enterprise java beans:

- Session Beans
- Entity Beans
- Message-Driven Beans

A session bean is where the developer puts business functionality, while an entity bean encapsulates data (like a database table row). Message-driven beans are used to create asynchronous JMS message listeners running inside the J2EE container.

The following sections give an overview of the three bean types, and are inspired by [RAJ02].

2.12.1 Session Beans

Session beans are the functionality containers in EJB technology, and therefore this is where developers are intended to put their business methods.

There are two types of session beans in J2EE – stateful and stateless session beans. As their names imply, one of them can remember its state during a conversation with a client and the other one cannot. There are of course different trade-offs associated with using each type.

The implementation of a session bean consists of the following:

- A remote and/or local interface to the bean
- An implementation of the methods defined in the interface
- A home object, which can create new instances of the bean
- One or more deployment descriptors, which the container uses to deploy the bean

When a client needs a particular bean, it can look up the home object using JNDI, and then invoke a `create` method to get a new instance of the bean. Figure 2.6 shows this.

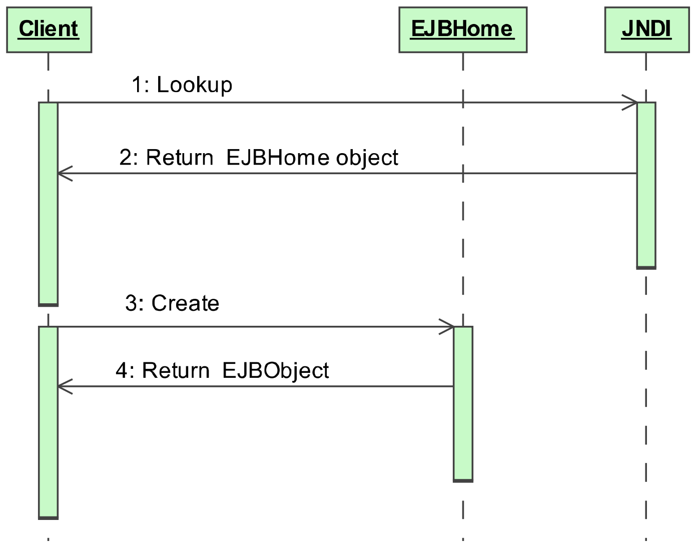


Figure 2.6: A client looking up a EJBHome object, and using that object to create a new EJBObject instance.

Stateless Session Beans

These beans do not maintain state across multiple method calls. A bean's state is the contents of its class variables at a given point in time. So in other words, when using stateless beans, you cannot use class variables to store information used across multiple method calls. You can of course use other means to emulate the existence of a state by using persistent storage or JNDI. In fact we will use such an emulation in this project.

The clever thing about not maintaining state is that it does not matter on which instance of a bean that a particular client invokes a method. This way, the J2EE container can make a pool of already created bean instances, and then feed these beans randomly to requesting clients. Naturally, this makes execution faster, since the container does not have to create new bean instances on client requests.

The trade-off is that you need to pass all necessary client data on each method invocation. This can lead to increased bandwidth usage.

An example of a stateless session bean is a weather forecast service, that delivers the forecast given a particular date. Figure 2.7 shows usage of such a service, when implemented as a stateless session bean. Note that the data must be retransmitted on each method invocation.

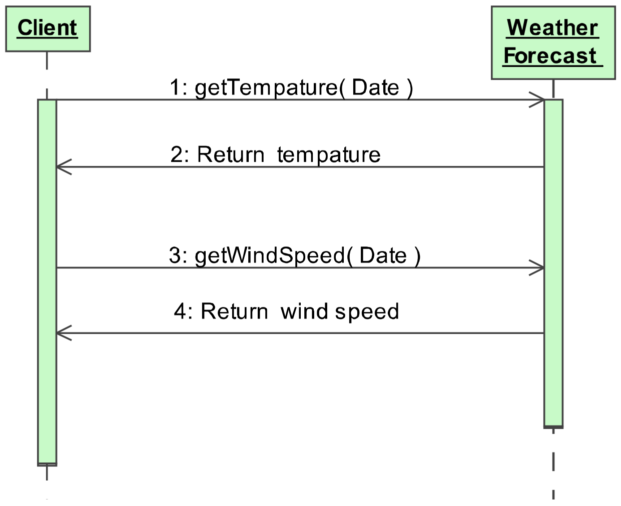


Figure 2.7: A weather forecast service using a stateless session bean.

Stateful Session Beans

Stateful session beans are used when the client needs to maintain state across multiple method invocations. So opposite to stateless beans, the programmer can make use of class variables when implementing stateful beans. This can decrease bandwidth usage, since the client no longer has to pass the same data to the bean more than once.

Here, the trade-off is execution speed. The J2EE container can no longer make an instance pool like it was the case with the stateless counterpart, because now each bean instance is associated with a particular client, and no other clients should invoke methods on that instance. Also, the container has to store the different states in memory, and in case memory runs short, in persistent storage. If not careful, this can lead to bottlenecks, where states must be fetched from persistent storage on every client request.

To show an example of a stateful session bean, we will again use the weather forecast service. Figure 2.8 shows the usage of the service, when implemented as a stateful session bean. Note that here we can set the date once and for all.

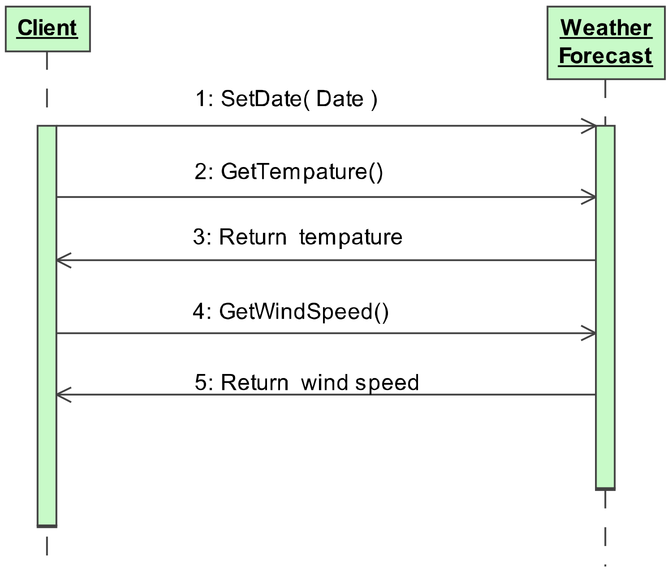


Figure 2.8: A weather forecast service using a stateful session bean.

2.12.2 Entity Beans

An entity bean is a persistent data component. It is an object that represents data and knows how to save its state to the data store when changed. When using entity beans, the programmer also gains some middle-ware services from the EJB container. Some of these services are data consistency, transactions and security. The implementation of an entity bean consists of the following:

- A remote and/or local interface to the bean
- A home object, which can create new instances of the bean
- An entity bean class
- A primary key class
- One or more deployment descriptors, which the EJB container uses to deploy the bean

The entity bean class maps to an entity definition in a database scheme, and it may have methods to alter data in the data store. There are also some required callback methods, which the EJB container will use. These will be mentioned later.

The primary key class is used to uniquely identify an entity bean. The bean itself calls the required method `getPrimaryKey` to know what data in the data store, with which it is associated. This is needed, because entity bean instances can be pooled and reused as will be mentioned later.

Persistence

There are two ways to make the entity bean persistent: Bean-Managed Persistent (BMP) and Container-Managed Persistent (CMP). In BMP, the programmer needs to write the database access code. In CMP, the programmer informs the EJB container of how the data should be persisted, using a container tool. The latter case will reduce the code in the entity beans enormously.

Entity Bean Instance

An entity bean instance is an in-memory representation of the entity bean data. Even though the entity bean instance is a copy of the data, it should be interpreted as the same. The required methods `ejbLoad` and `ejbStore` make it possible to use this interpretation. As the names imply, `ejbLoad` reads data from the persistent storage into the bean instance, and `ejbStore` saves the fields of the bean instance to the storage. The EJB container calls these methods. This is one of the features that make EJB suitable for rapid development; the programmer does not need to worry about data synchronization.

Pooling

Since creating and destroying an object is expensive, the EJB container normally chooses to pool entity beans in order to recycle entity bean instances. This means that the same entity bean can represent different data in two sequential calls. This is handled by the EJB container calling some of the required methods. `ejbActivate` and `ejbPassivate` are called when the bean is taken out of and put into a pool. These methods should acquire and release the used resources, e.g. socket connections.

2.12.3 Message-Driven Beans

This last bean type gives the server-side implementation of an application the ability to use JMS message listeners and thereby the possibility to receive incoming messages asynchronously. Though this is possible without using EJB, it requires the developer to create at least one thread running on the J2EE server. However, the usage of threads is not encouraged in J2EE, because concurrency control can be extremely complicated, and it does not blend well with the J2EE Rapid Application Development philosophy.

Message-driven beans do not have home objects or interfaces, since developers do not create new instances of messages-driven beans or invoke methods on them, so the implementation consists of the following.

- An implementation, which describes what to do with incoming messages
- One or more deployment descriptors, which the container uses to deploy the bean

The developer can simply associate a message driven bean with a specific topic, and the bean will then asynchronously receive messages from that topic.

2.12.4 Deployment Descriptors

When the EJB's are created, they need to be deployed to an EJB container. The deployment descriptor is an XML file that is provided by the bean implementer. It contains information about how the bean is to be deployed. This informs the EJB container about which beans exist and which middleware services they use, e.g. if the beans are stateful or stateless. Since EJB containers are different, often a vendor specific deployment file exists.

The EJB technology is primarily focused on large enterprise software, and it may be entirely unnecessary in many J2EE applications. [RAJ02] has a very good list of when and when not to use EJB.

2.12.5 JNDI and J2EE

When talking about JNDI and J2EE together, one of the key issues is the isolation of an application component (e.g. Enterprise Java Beans). Each application component has its own environment, also referred to as the Enterprise Naming Context (ENC). This is enforced by the application container, in the form of a context. This means that each component has an environment which cannot be accessed by any other components. In the case of Enterprise Java Beans, they can be accessed through a home interface available at a global name space.

2.13 Other Frameworks

As mentioned earlier, we have decided to take a brief look at two other frameworks, which could have been used instead of J2EE. Even though all three of them have been classified as frameworks, they are different in a number of ways. First, we will have a look at CORBA, and then Microsoft .NET.

2.13.1 CORBA

CORBA provides distributed interoperability between objects. It does so by using a language called the Interface Definition Language (IDL) to define interfaces to distributed objects. Other objects can use these interfaces to make requests to the distributed objects.

The central component of CORBA is the Object Request Broker (ORB), which handles location and programming language transparency. The language transparency is obtained by letting the ORB translate requests and responses into whatever format is appropriate.

CORBA introduces no direct support for web applications, since there is no direct way to forward HTTP requests to a system using only CORBA for communication. Another likely approach to our particular system, would be to let the users download a client application, which would communicate with other components through CORBA. There are of course many different security issues connected with users downloading programs and running them directly on their computers. We would not expect many users to take such risks in order to play a computer game.

2.13.2 Microsoft .NET

.NET is Microsoft's platform for making a competitive programming environment for every occasion.

> Microsoft .NET is a set of software technologies for connecting information, people, systems, and devices. This new generation of technology is based on Web services—small building-block applications that can connect to each other as well as to other, larger applications over the Internet [msn03].

Essentially, the information given in this quote might as well have been a description of what J2EE is. Both technologies are tools to help you build distributed applications, and both run in a container that handles tedious tasks such as load balancing and transactions.

.Net includes the ability to let different programming languages communicate through the .NET interface. Amongst the supported languages are C#, Visual Basic, and ASP. The largest drawback in connection to our project, is that most .NET specifications are not freely available [RV01]. However, Microsoft.NET would be a reasonable choice, when designing a game system such as ours.

2.14 Summary

We have represented a subset of J2EE technology and possible alternatives to it. The HTTP protocol and JNDI have been described as well, since both play a major role in J2EE development.

As stated, J2EE is part of the project foundation, so the choice of technology was already made. The choice can however easily be justified against the presented alternatives, especially when considering that this is an educational project, where access to technology specifications is important, and that the final solution is likely to use web application technology.

Chapter 3

A Model of the System

In the last chapter, we discussed the technologies that we will use in this project. We will now move on to define the general concepts of our game system, and use these concepts to construct a prototype. This prototype will make it easier to set up requirements to the system, which we need in order to construct a system model.

The system model will be constructed using different parts from the UML standard, since this is a generally adopted standard. We have primarily used UML[1] to clarify the understanding, and have not used it as a step towards automatic code generation. In the following sections, we will use descriptive pictures[2] as well as use cases, state diagrams and sequence diagrams as described in the UML standard.

3.1 General Concepts

We would like to have a turn based game that can be played by two or more persons in a distributed manner, connecting through some kind of terminal. At this point, we can identify some general concepts. The first concept is users. We need at least two users to agree on playing some game. In order to play, the users need to interact, and this leads us to the second concept, communication. The users should be able to write messages to each other, and should be able to notify about game events, e.g. moving a chess piece. Another concept is rules or consistency in a game. There should be a way to enforce that a game actually behave according to its rules.

Figure 3.1 describes the general concept of playing a game. The central point is the actual game. Each user can alter the game, and communicate with each other. Preferably, someone is making sure that the rules are followed (a moderator). The moderator can communicate with all users and can alter the game if, for instance, someone was found cheating.

[1] See bibliography [OMG03].
[2] Our translation of the term "rige billeder" from [MMMNS97].

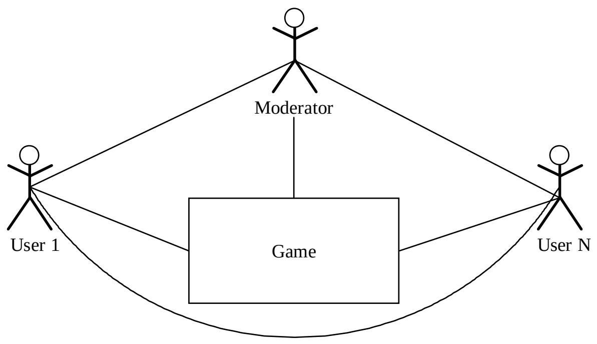


Figure 3.1: Relations between users and game.

With inspiration from other online game systems[3], we will use the abstraction of rooms and tables. Each game type (e.g. chess) can be distributed out to a set of rooms (e.g. beginners and experts). Each room will contain a number of tables on which a game can take place. If a user wishes to play a game, he or she must enter the appropriate room and then sit at a table. Since our system is aimed at turn based games, we also have the concept of a turn.

3.2 Prototyping

Before attempting to model a full-scale distributed system using a new technology, it is generally a good idea to construct a small subset of the system, and thereby getting a greater understanding of the technology. We have therefore made the decision to construct a prototype of the game system. This ultimately gives us a more realistic approach to the development process.

The functionality of the prototype will be limited to the chat component. To send text information back and forth, we need a communication line; in other words, the prototype will help clarify the communication possibilities available in a J2EE system. We will not go into further details regarding the experiments made through the prototype, since most of these considerations are mentioned in the design of the actual game system. Figure 3.2 is a screen shot of the prototype.

3.3 Requirements

Using the prototype and having the experience of trying other existing game systems, we can now setup a list of requirements to our game system. We will model the requirements with use case diagrams. We have three categories of use cases.

[3]We have primarily been inspired by the web site `spil.tv2.dk`, which is a Danish game site.

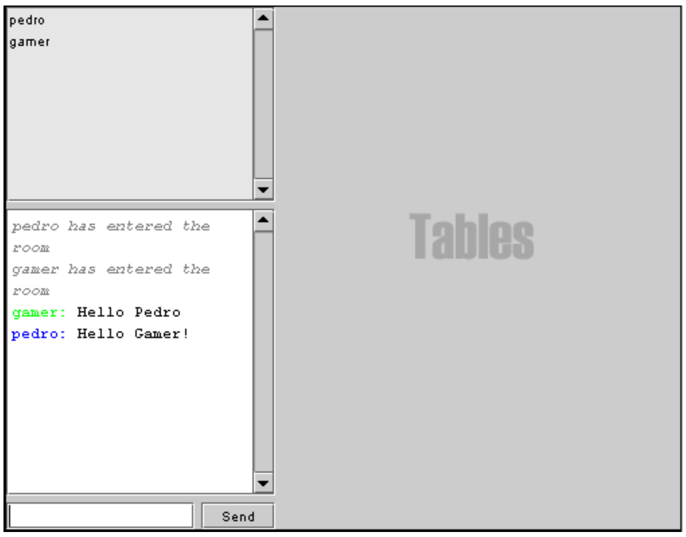

```
pedro
gamer
```
```
pedro has entered the
room
gamer has entered the
room
gamer: Hello Pedro
pedro: Hello Gamer!
```

Figure 3.2: Screenshot of the prototype.

- Authorization and authentication
- Joining a game
- Playing a game

In the following sections, the flow of events in the diagrams will be described.

3.3.1 Authorization and Authentication

Before being able to play a game, each user has to register in the system. When registered, the system should authenticate the user by requesting user name and password. In figure 3.3, the use cases that model this behaviour are shown. The actor in this diagram is a user who wishes to either be authorized or authenticated. A description of each use case follows.

Register and Unregister

The first screen presented to a user should be a register and login screen. If the user chooses to register, he or she should be asked to enter a new user name and password. The entered values would then be accepted by pressing an accept button. Upon pressing the accept button, the user name entry must be checked in order to see if it is already taken. If the user name is accepted, the user should then be passed on to a login screen.

Before a user can unregister, a registration should obviously exist. A user should be able to unregister by entering his or her password, and then pressing an unregister button.

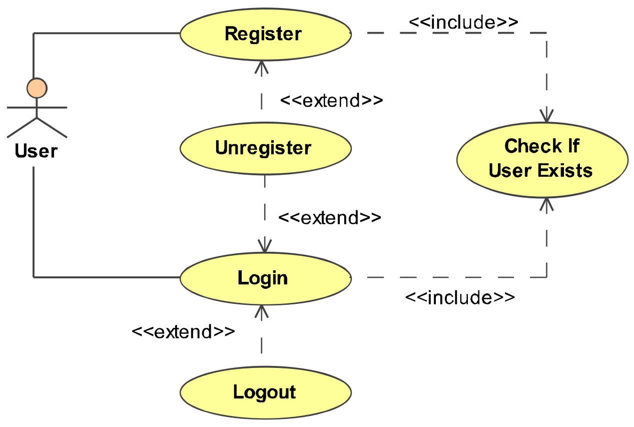


Figure 3.3: Authorization and authentication.

Login and Logout

On the login screen, the user should be asked to enter his or her user name and password so that the system can perform authentication. The system should check if the entered user name is registered, and if the password entered matches the password that the user was registered with.

To logout from the system, the user should simply press a logout button. Before a user can logout, he or she needs to be logged in. If the user does not manually logout then the system should do this after some time has elapsed.

3.3.2 Joining a Game

Once the user has logged in, he or she is presented with a room/table overview. Figure 3.4 illustrates the behaviour of a user in this situation.

Entering and Leaving a Room

After the login screen, the user should be directed to a screen presenting an overview of all the available games. Each game may be listed more than once, depending on how many rooms are dedicated to that particular game. When the user has chosen which game to play, he or she should click on the room representing the game to enter it. Inside a room, the user will see the tables available and a list of the users residing in the room. The user may now choose a table to sit at, or choose to leave the room. Leaving the room would remove the user from the user list.

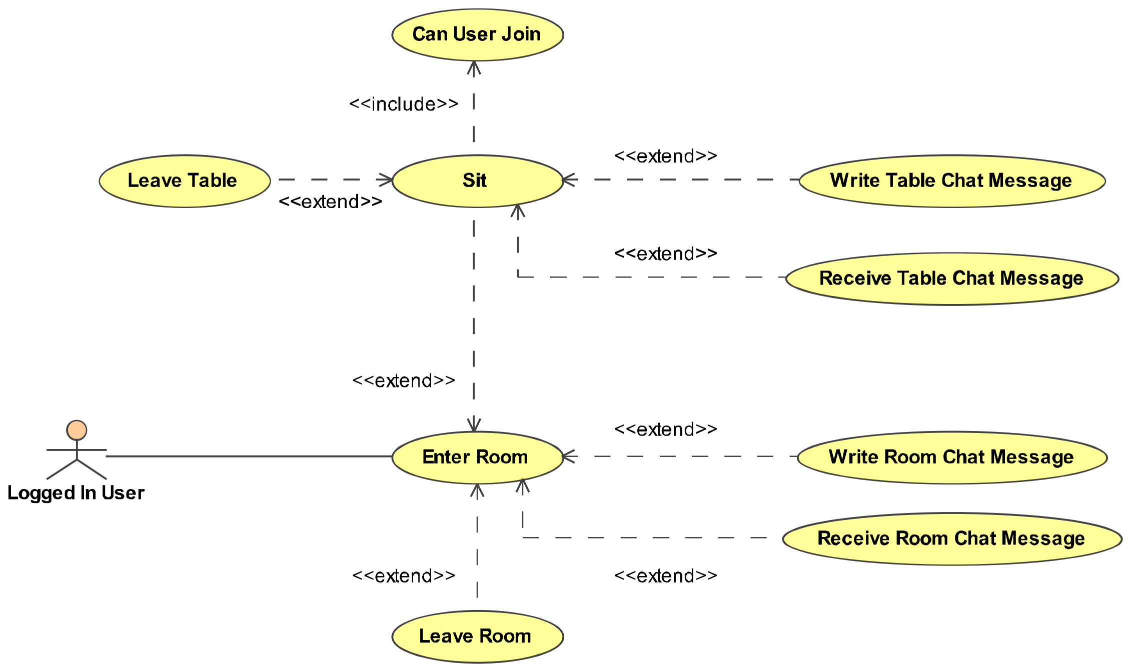


Figure 3.4: Joining a game.

Sitting Down and Leaving a Table

Joining a table is much like entering a room. By clicking on a table, the user should, if the table is not full, be placed at this table and a new screen should open showing the chosen game, e.g. showing a chess board in case of a chess game.

Leaving a table (leaving a game) should close the opened screen.

Writing and Receiving Chat Messages

In order to arrange games, users should be able to write and receive chat messages. After having joined a game, the users should be able to continue communication (it's no fun winning a game, if you cannot laugh at the other guy). Messages written to a room and messages written to a specific table should be held separated.

3.3.3 Playing a Game

Sitting at a table is the last scenario before actually playing the chosen game. The use case diagram in figure 3.5 shows the behaviour of a user and a moderator in this scenario.

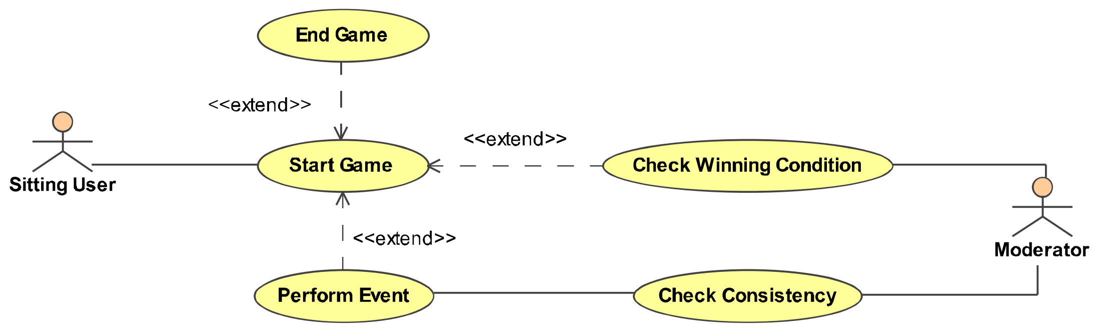


Figure 3.5: Playing a game.

Starting and Ending a Game

When two or more users have agreed on playing a game, they should be able to start the game. A game should start whenever all the users at a table are ready, and have pressed a ready button, or it might be started whenever a table is full. A user should also be able to end a game, e.g. if the opponent lost connection with the server.

Performing an Event

Once the game is started, the users in the game should be able to perform different events, e.g. moving a chess piece or rolling a die, depending on which type of game the user is playing. Each event performed should be validated by a moderator (see section 3.5.3).

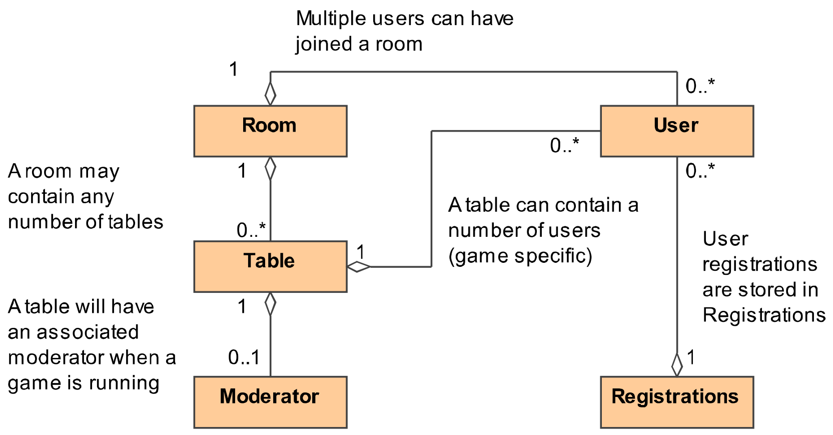
3.4 Classification of Objects

Having completed our requirements to the system, we will now classify the objects in our model, and determine their associations. Intuitively, our model consists of the objects described in table 3.1.

Name	Description
User	A user of the system
Registrations	User registrations
Moderator	Moderating game events
Room	A room dedicated to a specific game
Table	A table in a room. A game is played on a table

Table 3.1: Classified objects.

The object associations are very straight forward given our room and table paradigm. They are modelled by a class diagram in figure 3.6.



Multiple users can have joined a room
1
0..*
Room
User
0..*
A room may contain any number of tables
1
0..*
A table can contain a number of users (game specific)
0..*
User registrations are stored in Registrations
Table
1
A table will have an associated moderator when a game is running
1
0..1
1
Moderator
Registrations

Figure 3.6: Class Diagram.

We can now move on to defining the possible states of our objects.

3.5 Object States

Describing an object by its possible states allows for a better understanding of the particular object. Knowing which states an object can assume, and which events that trigger its state transitions, also brings us closer to an actual implementation. We will describe object states and transitions by using state diagrams.

3.5.1 User and Registration Objects

The User object is an object with a potentially very long life cycle. It is created when a user chooses to register, and destroyed when the same user chooses to unregister. A user is of course not active during the entire object life cycle, but can login and logout. This is modelled by figure 3.7.

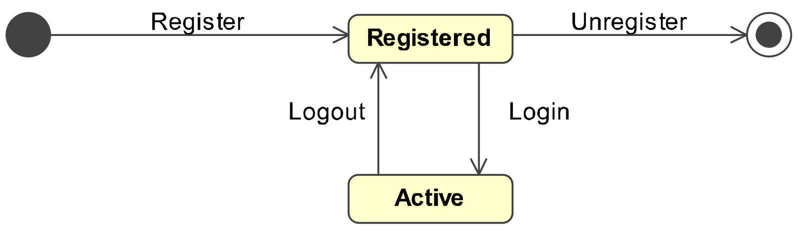


Figure 3.7: State diagram for object User showing registration and activation.

When a user is active, he or she can enter and leave rooms. When in a room, the user can sit down by a table and of course leave the table. The user can also write messages to other users in order to arrange games. This behaviour is modelled in figure 3.8.

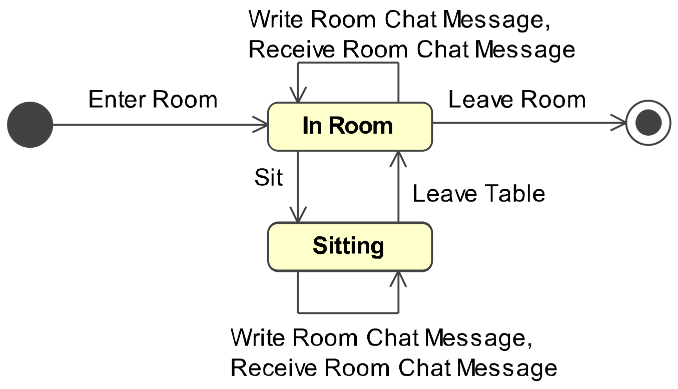


Figure 3.8: State diagram for object User showing an active user.

Finally, a user sitting at a table can start a game. When a game is started, the user performs events associated with the particular game until he chooses to leave the game, or if the game ends for some other reason (like inconsistency, which might indicate that someone is cheating). While in-game, the user can continue to communicate with other users by writing messages. The in-game states and transitions are shown in figure 3.9.

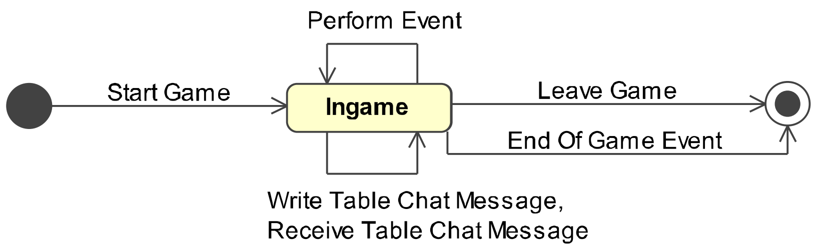


Figure 3.9: State diagram for object User showing an in-game user.

The Registrations object stores user registrations as shown in figure 3.10. To check if a user is registered, a lookup can be performed. This would return a User object if the user was found. This object is needed, since our way of creating user authorizations is through registrations.

3.5.2 Room and Table Objects

Defying real life physics, a room should be able to contain an unlimited number of tables and users. Rooms are created and destroyed, but a room should be viewed as an object with a reasonably long life cycle. That is, as the numbers of users increases and decreases, we create and destroy tables, not rooms. Figure 3.11 shows this.

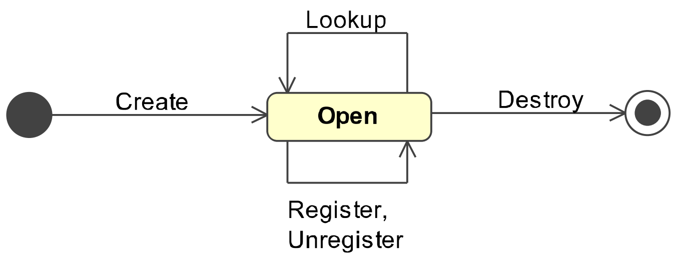


Figure 3.10: State diagram for object Registrations.

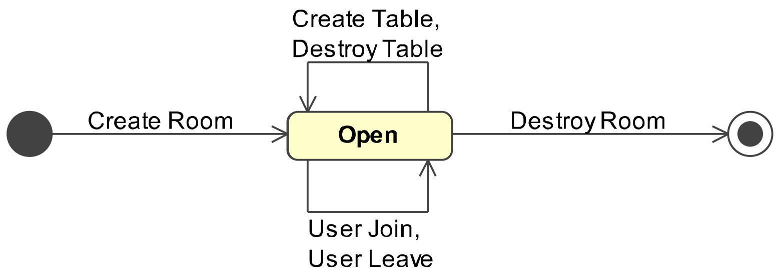


Figure 3.11: State diagram for object Room.

After a table is created, users can join and leave, and at some point a game will start at the table. Some kinds of game might allow users to join while a game is in progress, so this behaviour has also been modelled. Eventually, a game will end, or the table might even be destroyed. The life cycle of a Table object can vary. As mentioned, new tables can be created to deal with an increased number of users. It will be the task of the system to determine the need for tables. The state diagram of the table object is shown in figure 3.12.

3.5.3 Moderator Object

Our last object is the Moderator, which must monitor each ongoing game for consistency and winning conditions[4]. If an inconsistency is found, appropriate actions should be taken – which would be to either repair the inconsistency or end the game. The object state diagram is shown in figure 3.13.

3.6 Object Interactions

Given the objects, we will now – moving even further toward an implementation – model the interactions that occur between our objects. Since many interactions are trivial and

[4]Some games might of course end in tie, and here this is also covered by a “winning condition”

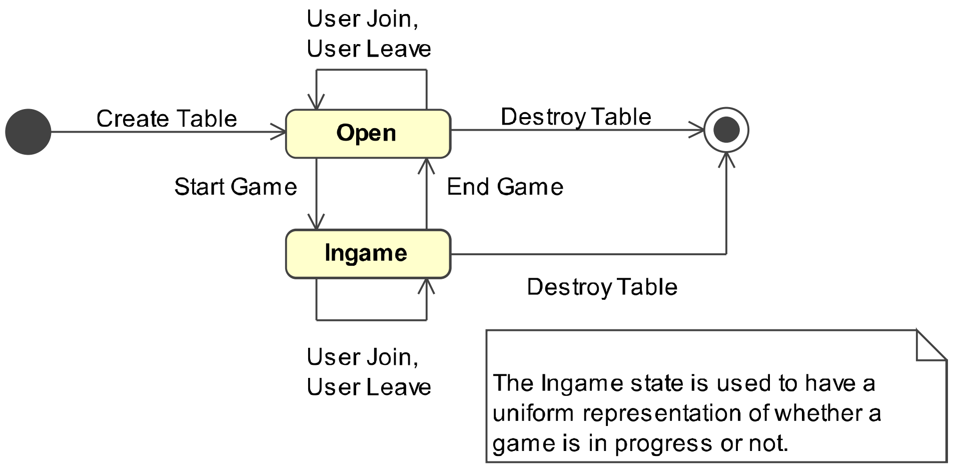


Figure 3.12: State diagram for object Table.

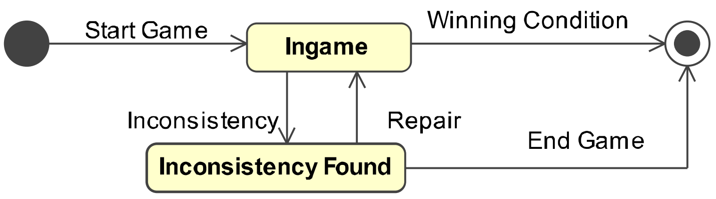


Figure 3.13: State diagram for object Moderator.

somewhat identical, we will only look at two of them; the interactions that happen when a user is registered, and when a user joins a game, respectively. We will use sequence diagrams to model interactions.

3.6.1 User Registration

Figure 3.14 shows the sequence diagram representing interaction between the User object and the Registrations object.

When registering, the User Object calls the `register` method on the Registrations object. The Registrations object then completes the registration, but not before checking if the user already exists. A reply is returned to the User object to indicate whether or not the registration was successful.

The object interaction occurring when a user logs into the system will of course be almost identical to the previous one. A user should not be able to interact with rooms or tables if he or she is not logged into the system. This topic will be described further in the design of the system.

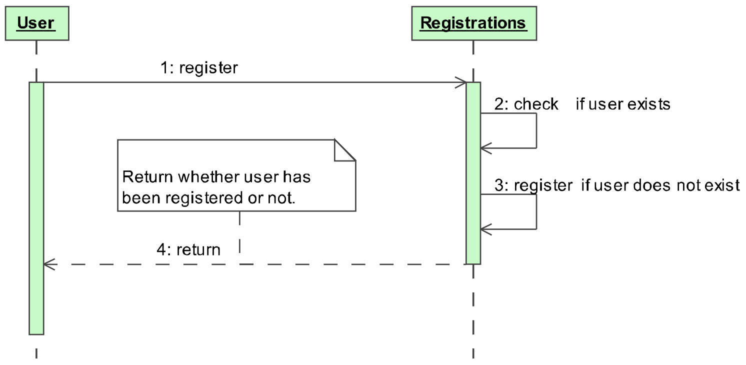

User
Registrations
1: register
2: check if user exists
Return whether user has been registered or not.
3: register if user does not exist
4: return

Figure 3.14: User registration.

3.6.2 Joining and Leaving Game

The most important object interactions in the system occur as a user joins, plays a game, and leaves a game, respectively. Figure 3.15 shows these interactions.

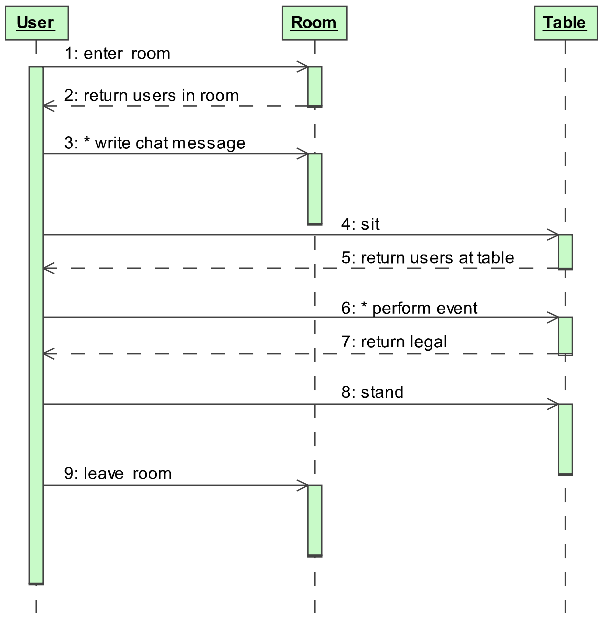


Figure 3.15: Joining a game.

When the user enters a room, the Room object will return a list of users that are currently in the room. While in the room, a user interacts by writing and receiving chat messages.

At some point, the user invokes the `sit` method, which means that the user will sit down at a table. The Table object will return a list of users at the table. The interactions now consist of game events, and whether or not these events are legal. Eventually the user will leave the table and leave the room.

Observing these interactions, it becomes clear that the interactions which will occur most frequently are the game event interactions, and the confirmation to whether a given event is a legal one. This is important, since it suggests that allowing these interactions to be executed very fast will greatly increase the performance of the system.

3.7 Summary

In this chapter, we have modelled our requirements to the game system with use case diagrams. To define the requirements, we constructed a prototype inspired by other existing game systems.

We have classified objects and described their states and interactions, and we now have a model of the system. The model itself gives a clear description of the intended system functionality, which will be helpful in the following design progress.

The next chapter will focus on the distribution of data and functionality, which is important for the performance of a distributed system.

Chapter 4

Distribution of Data and Functionality

In the previous chapter we constructed a model of our system. In this chapter, we will take a look at how data can flow from the client to the server, and we will discuss where to put the processing of data to minimize the network traffic, but still maintain a high level of control on the server.

We will clarify the advantages and disadvantages of thin and thick clients, and move forward to discuss how a server can be stateful or stateless. Being stateless means, that a server will not remember any requests previously made by any client.

We will not make any actual choices for our game system in this chapter; we will leave this for the chapters regarding the design of the system.

4.1 Thin and Thick Clients

When building a web application, the decision of whether to choose a thin or a thick client is very important. This discussion has to do with how much data and functionality should be handled on the client side, and how much data and functionality should be handled on the server side. The problem does not only pertain to new systems, but also to systems already in use, systems which could benefit from being web enabled.

In this section we will describe what thin and thick clients are, as well as consider the advantages and disadvantages of each. For further details refer to [app] and [Bar02].

4.1.1 Thin Client

The definition of a thin client according to hyperdictionary.com[hyp03] is as follows.

> A simple client program or hardware device which relies on most of the function of the system being in the server...

We will expand on this definition and say, that a thin client is a software application that is implicitly downloaded to the client.

The pure thin client is one extreme, when talking about how data and functionality is distributed between the client and server. Thin clients can be compared to dumb terminals in that all the functionality is located on the server[1], and that nothing can be done offline. In our project it is easy to see that we require the client to always be online while in use, because it is a requirement when playing against other players. So unlike some other distributed systems, the client in the game system cannot do anything sensible while being offline.

The main advantage when talking about thin clients, is that explicit download of software is not required. This is all handled automatically when connecting to the site where the application is located. This means that all clients that log onto the system, in our case the game server, will have the same version of the client, so no inconsistencies will exist between clients. The only exception maybe browser and platform specific GUI's[2].

Another important aspect is the matter of security. In the case of the game server, we do not want a client to be able to send the wrong data to the server e.g. try to fool the server to think that the game move, just performed, by a client was a legal move, when in fact, it was not. If all the functionality resides on the server, a client cannot manipulate the result as easily as if the functionality was located on the client i.e. rewriting the client.

The main disadvantage of using thin clients is the speed of the program. Because all functionality resides on the server, every time something happens on the client e.g. a player move, or a simple calculation, the server must be contacted. This means heavy network traffic and much more work for the server, which in turn results in the server not being able to handle as many clients.

4.1.2 Thick Client

The thick client is the other extreme. A thick client is usually something that is installed from a CD-ROM or explicitly downloaded from a remote site and installed onto the client machine. It usually has far more functionality located on the client side, and is more flexible than its counterpart, and the overall speed of the application will be far greater due to the fewer calls to the server.

An example of a thick client may be a complicated online game such as a Massive Multiplayer Online game (MMO).

[1]The difference between a thin client here and a dumb terminal, is that a dumb terminal does not have any harddisk, or software on the local machine. Everything is handled on the server

[2]Buttons and other graphics may vary from platform to platform. However Java does provide the opportunity to streamline this

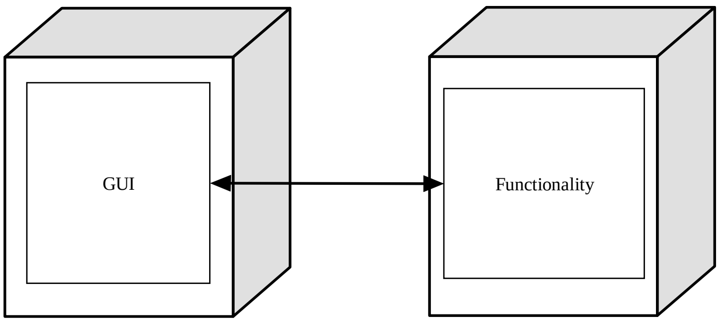


Figure 4.1: data distribution for a thin client.

An important advantage is, that a thick client is able to contact the underlying file system and operating system. This will add to the functionality of the program e.g. adding the possibility to access a local printer directly. One of the advantages of the thin client is the added security. With thick clients, this is just the opposite. First there is the fact that the installed software will have access to the underlying system, which is not always a good thing(think of the damage a malicious client program could do to the system). In trusted environments, this is not a big problem, however in the case of the Internet, where our game system is to be found, this is not so. Who can say what intentions the developers of an application have.

Another aspect of the lacking security in thick clients is, the ability to change functionality to something other than what was intended, this topic was touched upon earlier in the thin client section.

The last disadvantage we will mention here, is the problem of multiple client versions. Because the application is installed on the client computer, one is not automatically guaranteed that a client version is the latest available. There are two solutions to this problem. Support older versions of the clients i.e. make the server backward compatible to older versions of the client software. This means that the complexity, along with the space required of the server will increase as more and more versions appear. The other solution is automatic version checks performed by the client every time it logs into the server, this of course requires, that if there are multiple servers, they all provide the same version for download.

4.1.3 Game System

As can be seen from the above review of the thin and thick clients it is a trade-off between speed and security. What we will choose for our game system will naturally be something in between. A further discussion of this can be found in the chapters 5 to 9 regarding the design of the system.

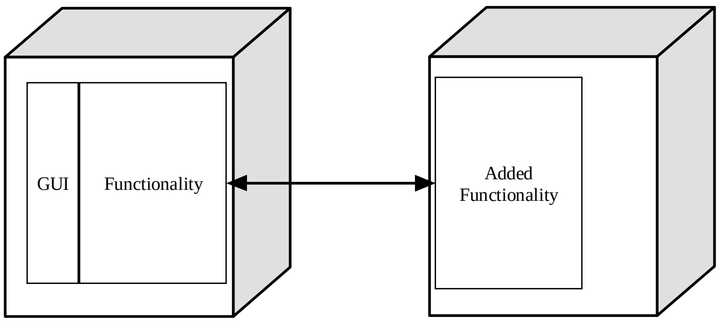


Figure 4.2: data distribution for a thick client.

4.2 Stateless and Stateful Servers

When developing a client/server system, we have the options of both a stateless and a stateful server. Below follow advantages and drawbacks of both.

4.2.1 A Stateless Server

A stateless server does not keep track of the states in which its visitors are. When clients communicate with a stateless server, they must know that for every new query they make, the server has not remembered any previous ones.

An example of a stateless server is a typical web server. Basically, a web server has a directory structure of files, that are each requestable via the URL line. All files forming a web page are requested one by one[3] as if they had nothing to do with each other. In between these requests, the server returns to its default "state", happily unaware of what is to come and what has just happened. It might, of course, keep a log file but that does not affect its state.

This principle is implementable in our case, too. However, since the server must return to a default state for each request, it is necessary for the clients to keep track of the game state. Intuitively thinking, this opens a possibility for cheating in the game: A cracked client could make illegal moves. However, since in most games there are two or more players, one crooked client's actions will interfere with those of its legal opponent(s).

Even though the server is stateless, it *can* actually still play the role as game moderator and see to it that the rule set is adhered to. This requires the clients to actually transfer the complete game state (e.g. piece positions) in addition to every query. Needless to say, this generates considerably more network traffic, but for a game of relatively few server enquiries, it may not matter.

[3]Only in HTTP 1.0; in HTTP 1.1, which is most commonly used nowadays, the socket connection can be kept open until all files of a web page are transferred, and all files are transferred over this one connection. See section 2.2.

We still have the problem of crooked clients, though; the relation between game states is still handled by the clients themselves, and a client may choose to ignore the server's reply to e.g. whether a move is valid or not. The only way to overcome this in a fair way is to actually end a game where one or more clients' views of the game state differ from that of others.

A stateless server would also mean that we cannot have a central storage of user data (such as game statistics, etc.). Again, we would have to trust every single user that it is providing the correct score, etc.

4.2.2 A Stateful Server

The opposite, a stateful server, is when we allow the server to remember states. This means that a command to a stateful server might trigger different actions depending on its current state. An FTP server is an example of such. Once you have logged in, you can perform an array of commands without authenticating yourself to the server, and for instance the command `ls` gives a file list relative to the directory that you are currently in. To be able to do this, the server must remember certain information between the received commands, namely the user who has logged in and the current directory.

In a game server system such as ours, the clients now hold only a representation of the current game state[4], and every server query does not include this state. Again, the server checks for illegal moves, but now it does so on the basis of its own perception of the game; not the clients'. This way, if a client tries to perform an illegal move, the server can easily deny it. We do not have the problem of deciding whose perception of the game is correct since there is one perception that we know is always right; that of the server.

A stateful server is of course also a nice place to store users' game statistics. Since we can rule out illegal moves, by controlling e.g. scores on the server, we can also prevent tampering with these; a user would have to actually win a game in order to increase his score.

This section summed up the advantages and disadvantages of stateful and stateless server implementations and presented an overview of how a game server system could make use of either one.

4.3 Summary

In this chapter we described some fundamental issues concerning how data and functionality are distributed. We discussed the advantages and disadvantages of distributing the functionality to the client, contra having it located on the server. We saw that the advantage of thin clients was the higher security in keeping functionality on the server. This is the disadvantage of the thick client. This client in turn, has the advantage of being faster and more flexible.

[4]To save bandwidth, though, one could implement a checking algorithm on the client in addition to the one on the server, but the server must have the final word.

The next issue discussed was the problem of whether to choose a stateless or a stateful server. We saw that if we were to choose a stateless server, the server would not have to keep track of the states of the individual games. However, it had the drawback, that more data needs to be transferred with each transmission. The stateful server, on the other hand, has the advantage of being able to keep track of the individual games, which makes cheating more difficult.

Chapter 5

Game System Framework

We are now ready to design our game system framework. This chapter is dedicated to a discussion of the design of the primary features, that the system should make available to game developers. Our approach will be to build an extensible application, where anyone can add their own games.

The main objective is to allow game developers to think in terms of rooms, tables, users and moderators, and not worry about how these concepts are implemented below. In other words, we want to make the task of creating a distributed turn based game as straight forward as possible. Also, implemented games should be easy to access for potential users.

5.1 Features

We can split up the game system into two main features, which must be provided by the system. These are user handling and game handling.

User handling can be viewed as a system in itself. Authorization and authentication of users will be centralized to this part of the system, and not implemented for each game. This means that if you are authorized to play one particular game, then you are also authorized to play any other game in the system.

The game handling is the engine that will run each game installed in the system. The game developers will implement their games using this part of the system. It should provide features, such that game developers can stay on an abstract level of thinking. The following is a basic list of features that this part of the system should provide.

- The game developer should implement one game instance only, and not worry about how many instances are running at any given time
- Receiving information, at either client or server, should happen asynchronously from the game developers' point of view

- When sending information, the underlying communication protocol should be transparent to the game developer

The first requirement is very important. Keeping track of multiple games, and which game some incoming information belongs to, is not necessarily an easy task. However, keeping track of only *one* game, and the incoming information belonging to that particular game, should be straight forward (some games are of course more complex than others).

Receiving information asynchronously is a nice feature for developers, since they do not have to ask for new information on their own initiative. The underlying communication protocol should be transparent, because it should be possible to change it without changing already implemented games.

5.2 Framework

A game developer needs a way to make use of the features that the game system provides. We will construct the system as a framework, where Java inheritance will be the primary means for extending the system. Instead of relying mainly on inheritance, we could use an API as the only means of accessing the framework. However this would clutter the game code with API calls. The inheritance model makes it possible to make a relatively clean joining of games and framework. Figure 5.1 shows this concept. As shown, we can divide the framework into a client framework and a server framework. We will use this separation as we design the framework.

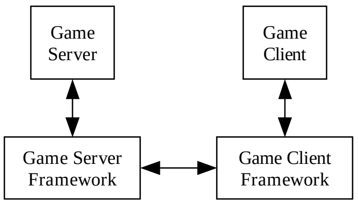


Figure 5.1: Basic framework design.

5.2.1 Inheritance Model

Using inheritance to extend framework features makes it possible to force game developers to implement certain methods[1]. To illustrate this principle, we can look at the task of handling a game specific message. Handling such a message is the responsibility of the specific game implementation, so we need a means to deliver the message to the game extension. So, we force the game developers to implement the method `onMessage`, and supply the message through

[1]Using abstract methods

this method. The message can then be processed inside the game extension. Figure 5.2 shows the inheritance between an abstract message handler, and a game specific message handler. Exactly how the game extensions uses the features offered by the framework is discussed in chapter 8 and 9 regarding the client and server framework.

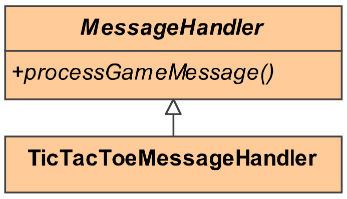


Figure 5.2: Example of framework inheritance.

5.3 Summary

We have now specified which features the game system should provide to a game developer, and divided the system logically into user handling and game handling.

We have also looked at how features should be accessed, that is, through a framework using Java inheritance, abstractions, and API calls.

In the next chapter, the focus will be on the communication between the game system client and server.

Chapter 6

Web Applications and Communication

In the previous chapter we introduced the basic design of the game system. We will now focus on the communication in the system, in particular the communication between the game client and the game server.

We need to make a design decision regarding the interface to the server, and how the client should use it. We will also discuss how a user gains access to client software.

High accessibility is listed as a requirement to the system, and this naturally leads us toward the World Wide Web.

6.1 Web Applications

Allowing users to play a game directly from a web page without explicitly downloading anything, will ensure that all users, regardless of their administrative computer skills, will be able to participate. The J2EE standard provides an integrated web server for this purpose, and has an abstraction of a web site called a web application.

The term web application in J2EE covers an application implementation using servlets and/or JSP. The entire application does not necessarily have to be implemented using only those technologies, but they are the interface between the client and the server.

The user interface for authorization and authentication is well suited to be implemented directly on a web page using HTML forms. Such tasks only require updates from the server when the client performs some action, e.g. submitting a login request.

However, this is not the best solution for the user interface for an actual game. When a game is running, there will be a lot of communication between clients and the server, and if the user interface were designed in HTML, this would require a page reload every time new information becomes available. This is not very dynamic and would be tiresome for players, and they would probably find another game to play.

Java Applets are an intuitive approach for creating a game running on a web site. Using applets, it is possible to make requests for new information without a web page reload. It is also possible to communicate through the network using other means than servlets or JSP. An applet could e.g. communicate directly with an enterprise bean.

6.2 Communication

Communication in a J2EE application consists of the communication between a client and the server, but also of communication between various server components, which do not have to be running on the same machine. Figure 6.1 shows this principle.

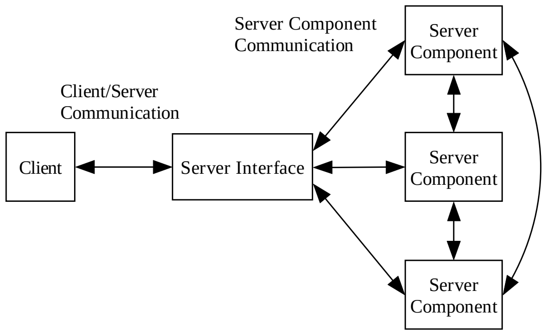


Figure 6.1: Application communication.

There are two distinct paradigms to be considered when implementing communication; method calls and messaging. A common property of turn based games is the way that they are played; one move at a time, where some moves are only allowed after other moves has occurred. This makes turn based games well suited for using messaging, where each message can serve as an event. If a user moves a chess queen from field A1 to A2, then a message like `queen, move from A1 to A2` could be transmitted. When the opponent receives this message, it will update the view of the game, and the game can continue. The messaging service of J2EE is JMS, and we will use this service as the backbone of our message communication.

For now we will concentrate on the communication between the client and the server interface, and in particular the communication between an applet running a game and a servlet.

6.2.1 Servlet Layer

When a client communicates through servlets, we say that it communicates through a servlet layer. The layer offers an interface to the server-side application. When communicating through a servlet layer, all network traffic is being wrapped in the HTTP protocol. This has advantages, but also drawbacks. Note that servlets are capable of containing functionality, and not just serve as interfaces to the server.

The main and greatest advantage is that most firewalls will let HTTP communication pass through, where other protocols most likely will fail. If a user is behind a firewall, which does not allow HTTP communication, then even browsing the World Wide Web would be impossible. In that case, the user would not have discovered the web site containing the game system. This is not normally the case, so using HTTP means that almost all users will be able to use the system.

The drawback is that a client using a servlet layer needs to design everything in terms of requests and replies, since this is how HTTP works. Figure 6.2 shows the principle of using a servlet layer.

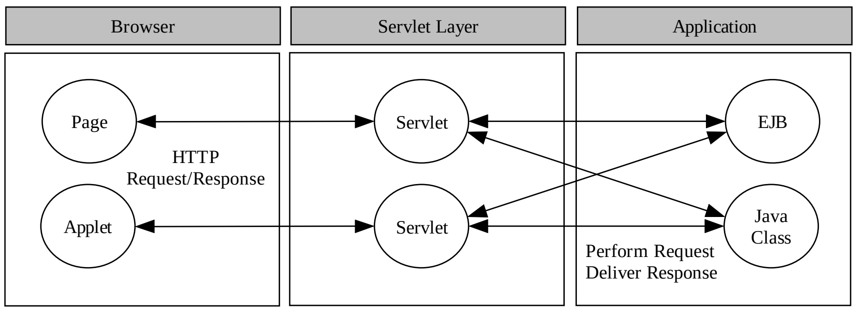


Figure 6.2: Servlet Layer example.

6.2.2 Messaging and Enterprise Beans

Instead of using a servlet layer, we can send JMS messages directly from the applet. This offers a higher network transparency, since communication is now done through messages to a queue or topic. Unfortunately JMS messages are not likely to pass through a firewall, because most firewalls will have no reason to allow communication using this protocol.

We could also communicate directly with deployed enterprise beans using their remote interfaces, but this uses the RMI/IIOP[1] network protocol, and such communication will most likely not pass through a firewall either. Figure 6.3 shows how an applet can make a call directly to an enterprise bean.

Using either of these technologies directly from an applet will in addition require additional technology Java packages[2] attached with the applet download. This means that the download would take up longer time, and keep the user waiting.

[1]A further description of RMI/IIOP can be found in [RAJ02]

[2]E.g. a JAR file containing EJB classes

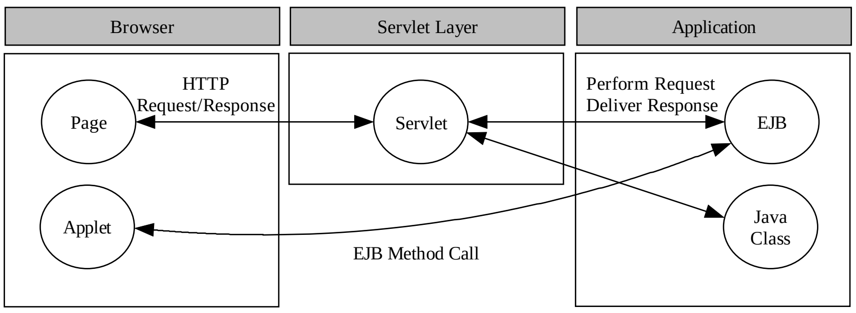


Figure 6.3: Example of using EJB remote interfaces.

```java
//Create an output URL connection to servlet
URL servlet = new URL(urlToServlet);
URLConnection servletConnection = servlet.openConnection();

//Wrap an object output stream around the connection output stream
outputToServlet = new ObjectOutputStream(servletConnection.getOutputStream());

//Write objects to be transmitted to the stream
outputToServlet.writeObject(myObject);

//Close the stream
outputToServlet.close();
```

Listing 6.1: Sending an object to a servlet.

Another option would be to use JMS messages or remote method calls when possible, and then switch to HTTP if a firewall is detected. This solution would, however, still require the additional download of client software, and would make the client network code more complex.

To accommodate the requirement of high accessibility, we will implement a servlet layer. Another requirement is communication transparency, which means that we will have to wrap HTTP requests and responses in a more abstract component - thereby hiding from game developers the fact that they are actually using HTTP.

6.2.3 Sending and Retrieving Information

Having decided to use a servlet layer, we now consider how a client applet can send and retrieve information using the layer. Sending information is straight forward; the client can simply make a request and wrap it around any objects that should be transmitted. Listing 6.1 is an example of sending an object to a servlet.

On the servlet side, objects transmitted from the applet can be fetched from the request input stream. Listing 6.2 is an example of this.

```java
public void doPost(HttpServletRequest request , HttpServletResponse response){
  try{
    //Wrap an object input stream around the request input stream
    ObjectInputStream inputFromApplet =
      new ObjectInputStream(request.getInputStream());

    //Read an object from the stream
    Object obj = inputFromServlet.readObject();

    //Close the stream
    inputFromApplet.close();
  }
  catch(Exception ex){
    System.err.println("Error receiving object from applet: "
      + ex.getMessage());
  }
}
```

Listing 6.2: Retrieving an object from an applet in a servlet.

Retrieving information from the server, e.g. that another player has completed a turn, can be done in a similar fashion, where object streams are wrapped around an HTTP connection's input and output streams. However, this way of communicating does not allow the server to broadcast information to clients on its own initiative. The server can only prepare data for the next time the client chooses to make a request. This can cause unnecessary network traffic, since clients will keep asking for new information even though none might be present.

Instead of closing a connection whenever a request and response cycle is complete, the connection could be kept open. This requires a small work-around in the servlet, which must enter a loop, so it does not exit its primary method. Doing so will keep the connection open, and the server may transmit information to the client at any time. In fact, it will wrap an entire user session into one HTTP request and response. This technique is the basic idea behind some streaming technologies, which e.g. can be used to view video on the Internet. However, a user of our game system could potentially stay logged into the system for a very long time, and would occupy a server port throughout the entire session. It is difficult to predict the exact performance when applying this concept to our problem, and it should be tested before applied. We will not go that far, but instead just note that this could perhaps be a reasonable solution.

We will instead use the traditional way of communicating, that is, letting the client initialise requests both when transmitting and retrieving information. From now on we will use the term "client message listener" when referring to the clients' method of retrieving information from the server on a regular basis. Listing 6.3 shows the basic idea of a message listener.

Here we also introduce the client message handler, which is the object responsible for handling incoming messages from the server. Notice that the delay in the example is hard-coded to two seconds. It is possible to regulate this delay to match the actual server load. If the server is very busy, the delay could be set to e.g. four seconds for all clients, and thereby cutting the

```java
public class MessageListener extends Thread{

  MessageHandler messageHandler;

  public void run(){

    while(running){

      //Fetch messages from server using HTTP
      messages = getNewMessagesFromServer();´

      //Process the incomming messages
      messageHandler.processMessages(messages);

      //Wait for two seconds before looping
      try{
        Thread.sleep(2000);
      }
      catch{Exception ex){
      }
    }
  }
}
```

Listing 6.3: A client message listener retrieving information from a server.

amount of requests in half.

6.2.4 Messages

Having decided to use messaging, we need to define which message types can be sent, and when each type should be used. Even though we are using a servlet layer (and not JMS directly from the client), the communication should still consist of messages.

Intuitively we have three types of information in the system, namely chat information, game information and system information. The latter could e.g. be a message stating that a user has joined a room or a table. Figure 6.4 shows a small class diagram of message types.

The class `GeneralMessage` is an abstract class, which only contain the attributes `from` and `destination`. These must indicate which user sends a particular message and whether the message destination is a room or a table. The `SystemMessage` is used to send information that the framework will use to manage in which rooms and tables users are located, and contains a `type`, which indicates a particular event associated with the message[3]. The `ChatMessage` obviously contains a message to be relayed to other users, while the `GameMessage` is an abstract class not meant to be used directly. Instead game developers must inherit from this class when they implement their own game messages. This allows the system to relay

[3]An example of an event is a user leaving a table

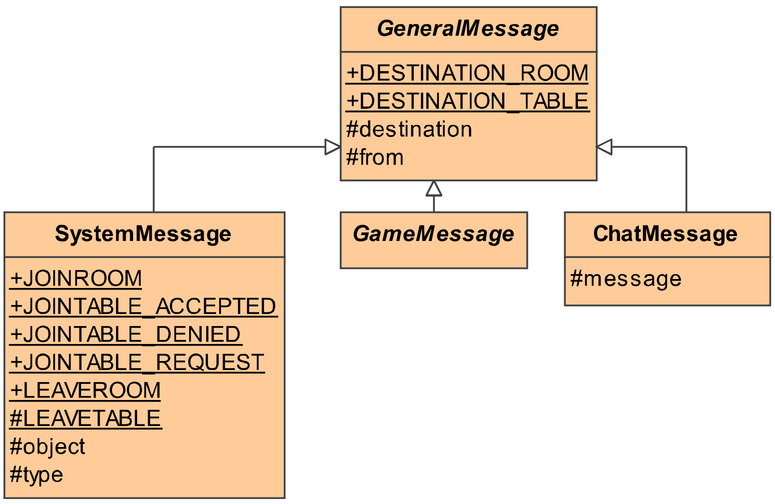


Figure 6.4: Message types.

game messages to the actual game implementations by determining that a given message is a descendent of `GameMessage`.

6.3 Server Interface

We need to create a well defined interface to the game server, so that clients can access it appropriately. We have already chosen to implement a servlet layer, so this will be the actual interface. The client should be able to access the following functionality.

- Register
- Login
- Enter and leave a room
- Sit down and stand up from a table
- Send game events
- Receive game events

We could choose to implement only one servlet, which would take a parameter indicating which function should be executed. Instead we will implement a servlet for each function, because this allows us to change the object called, when a specific URL is entered, without

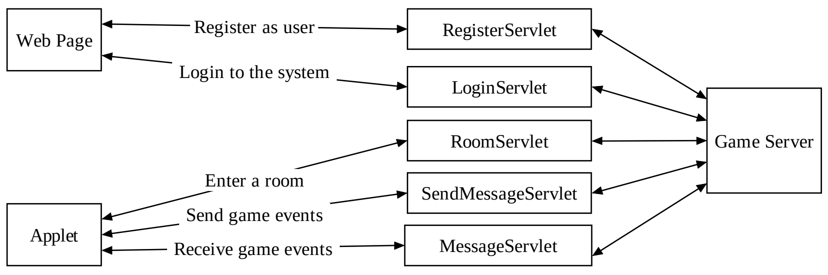


Figure 6.5: The servlet layer, which constitutes the game system interface

```
public interface UserHandlerLocal extends EJBLocalObject {
  void register(...);
  void login(...);

  void registerInRoom(...);
  void unregisterInRoom(...);
  void sendMessageToRoom(...);

  void registerAtTable(...);
  void unregisterAtTable(...);
  void sendMessageToTable(...);

  void getNewMessages(...);
}
```

Listing 6.4: `UserHandler` interface.

modifying any existing Java code[4]. Figure 6.5 shows the servlet layer interface to the game system.

The functionality of sitting down by a table is encapsulated in the ability to send game events. If a user wants to sit down, then an appropriate game event can be sent.

If we later want to add a client, that can communicate directly with the enterprise beans of the game system, we should allow this[5]. We must therefore also provide an interface for this case. To construct such an interface, we can use a session bean, let us call it `UserHandler`, but there is the question of whether it should be stateful or stateless. We will consider the pros and cons to this in a moment. First we will describe the interface to the session bean, which is naturally very much like the servlet layer. Listing 6.4 shows the local interface to the bean.

We will let the servlet layer call through the `UserHandler` bean to execute any functionality, since this couples the two means of access together, and makes it easier to change.

[4]By modifying servlet mappings in the web application configuration files
[5]Because constructing a client might be easier if not forced to use the HTTP protocol

```java
public class UserHandlerBean implements SessionBean {

  Object myClassVariable;

  public UserHandlerBean() {
    myClassVariable = new Object();
  }
}
```

Listing 6.5: Example of using class variables inside a session bean.

6.3.1 Stateful or Stateless Session Bean

Intuitively a stateful session bean would be an obvious choice for our application since we need to store information about each user (e.g. whether the user is logged in, or which room is entered). Creating a stateful bean would allow us to simply represent this information as class variables in the bean. Listing 6.5 shows how information can be stored inside a stateful bean using class variables.

The problem is, that we have stated that we want the system to be as scalable as possible within the boundaries of this project. As described in chapter 2, stateful beans can not be pooled and reused as easily as stateless beans. Given enough users, we could end up in a scenario, where each bean is stored on disk because of memory shortage, and then fetched into memory on each user request. This would slow down the system considerably.

Instead we can make the bean stateless, and then use JNDI to emulate the notion of a state. This means that instead of storing data in class variables, we will move it to the bean environment in JNDI. Listing 6.6 shows this concept. We will avoid the problem of the bean being swapped to disk, because the amount of stateless beans does not have to match the amount of users. We do, however, introduce another small problem. Storing information in JNDI means that we need to look it up every time it is needed. This is, of course, slower than if the data was available directly as class variables, but this problem has a more constant time factor than the memory problem. That is, the time it takes to make a lookup in JNDI is independent of the amount of users present in the system. We therefore choose to make the `UserHandler` session bean stateless.

6.4 Summary

We have made the choice to implement the game system as a web application. The user interface for authorization and authentication will be implemented directly on a web page, while the actual games will be run in applets. In general, the communication in the system will consist of messages.

The interface to the server will be a servlet layer, so that the game system will work even in

```
public class UserHandlerBean implements SessionBean {

  public UserHandlerBean() {
    InitialContext ctx = new InitialContext();
    myClassVariable = ctx.lookup("java:comp/env/myClassVariable");
  }
}
```

Listing 6.6: Example of looking up variables using JNDI.

the presence of most firewalls. Whether we will put even more functionality in servlets will be discussed in the following chapters.

We have also made the choice *not* to keep an open connection between the client and the server, while running a game. Instead the applet will make an HTTP request on a regular basis to check if new information is available.

The information flowing between a client and the server has been defined as three message types, system messages, chat messages and game messages.

In the next chapter we will focus on creating a platform for authorizing and authenticating users.

Chapter 7

User Authorization and Authentication

An authorization of users is needed so that users can authenticate themselves to the game system, when they want to play a game. Experience shows that users behave more seriously, when they are registered[1]. Registration of users will also make it possible to implement a statistics of the games played, so that the users not only compete from game to game, but also in overall performance.

7.1 User Interface

We have described how a user should register and login in our use cases in chapter 3. This kind of interaction is well suited for a user interface implemented in HTML. Since we use dynamic contents, this means that we will implement the user interface in JSP.

To allow users to register, we construct an HTML form allowing the users to enter their desired username and password. We set the `action` parameter to point at the `RegisterServlet`, which is responsible for relaying the registration. We have chosen to use the HTTP method `POST`, since we are posting information to the server.

The login process is handled similar. Listing 7.1 shows the login HTML form, which allows the user to enter his username and password. Here we send login information to the `LoginServlet`.

7.2 Persistent Storage of Users

The registration of users would not be very useful if it were only stored in the memory of the server. In that case, registrations would of course disappear every time the server was

[1]This is the reason why serious forums on the Internet often require registration

```
<form method="post" action="Login">
  <h3>Login</h3>
  <p>
  Enter your username and password and press <b>Login</b>.
  <br>

  Username<br>
  <input type="text" name="username">

  Password<br>
  <input type="password" name="password">
</form>
```

Listing 7.1: Part of JSP-login.

e.g. restarted. We therefore have the need for a persistent storage of users. As mentioned, we use mySQL as the underlying database in the project, and this is where we will store user registrations.

The registration could be done by letting a servlet connect directly to a database using JDBC. This would give us full access to SQL database functionality, but our Java code would not be portable because of the SQL-variations on the market. Another disadvantage is the fact that the code for data modification needs to be hand made, and the programmer should therefore make sure that the data in the database and the data in memory is consistent.

If we instead chose entity beans with container managed persistency (CMP), we would have portable Java code, because then there would be no SQL-calls in the code. We would also move the responsibility of data consistency onto the EJB container (as mentioned in chapter 2). A disadvantage is that there is a larger call overhead when using entity beans in contrast to using JDBC directly from a servlet. This is because of the extra layer the call has to go through. If the access to the entity bean is through a remote interface, it gets even worse. This is because entity beans represent data logic with get/set methods and not functionality. Executing any business method would normally require several calls to an entity bean, and each call would require network transmission. This can lead to unnecessary network traffic.

We choose to implement database access using entity beans, because of the advantages mentioned. We now have to decide whether to use bean managed persistency (BMP) or container managed persistence. We will choose to use the latter, since this makes the solution portable to other databases. We also avoid introducing errors, since we do not need to implement database access code directly. If we just make sure, that the entity beans are never accessed through a remote interface, the call overhead will not differ much from the JDBC-servlet solution. Should very fast read access to the database be needed at some later point, this can be handled with a Fast Lane Reader[sun02b] pattern, where some database reads can be implemented directly using JDBC. This could, however, compromise the portability of the Java code.

```xml
<entity>
    <ejb-name>GameUser</ejb-name>
    ...

    <!--Chooses container managed persistency-->
    <persistence-type>Container</persistence-type>
    ...

    <!--The name we give the bean-->
    <abstract-schema-name>GameUserBean</abstract-schema-name>

    <!--Container managed fields-->
    <cmp-field>
    <field-name>username</field-name>
    </cmp-field>
    <cmp-field>
    <field-name>password</field-name>
    </cmp-field>
  </entity>
```

Listing 7.2: Deployment descriptor.

7.3 Implementing the Entity Bean

We will implement an entity bean called `GameUser`. Since we do not have to write the data access code, we have to somehow inform the container of the mapping between a bean and the data in the database. This is done by using deployment descriptors. Here we can define the container managed fields, which must match get and set methods in our bean class. Listing 7.2, which is a section of our deployment descriptor, shows this. The EJB container can then generate the database access code by linking the CMP-fields with the corresponding get and set methods in the bean.

In addition, we need to specify to JBoss, which database and field the entity bean should be associated with. We do this in a container specific deployment descriptor. Listing 7.3 shows a part of our JBoss specific descriptor.

We stated that we do not want entity beans invoked directly through their remote interface. There exists a design pattern for solving this problem called Session Facade[sun02a], which puts session beans in front of all entity beans. The session beans and the entity beans must exist in the same virtual machine, so that the local interface of the entity beans can be used, thus making the remote interface superfluous. This is a good idea, because, as mentioned, a business method normally requires several database calls, and this way the caller will only make one remote method invocation. Remote interfaces to entity beans also forms a security risk, since it reveals the structure of the database. We will use our session bean `UserHandler` as a facade for `GameUser`.

```
...
<!--Name of database-->
<datasource>java:/GameServerDS</datasource>

<!--Type mapping-->
<datasource-mapping>mySQL</datasource-mapping>
...
<entity>
  <!--Name of entity bean-->
  <ejb-name>GameUser</ejb-name>
  ...

  <!--The name of the table in the database-->
  <table-name>GameUser</table-name>

  <!--Container managed fields-->
  <cmp-field>
    <field-name>username</field-name>
    <column-name>username</column-name>
    <jdbc-type>VARCHAR</jdbc-type>
    <sql-type>varchar(32)</sql-type>
  </cmp-field>
  <cmp-field>
    <field-name>password</field-name>
    <column-name>password</column-name>
    <jdbc-type>VARCHAR</jdbc-type>
    <sql-type>varchar(255)</sql-type>
  </cmp-field>
</entity>
```

Listing 7.3: JBoss deployment descriptor .

7.4 Finding Users in Storage

We have to specify a finder method to be able to find existing users in the database. We write the required method `findByPrimaryKey` in the home interface of our entity bean, which searches the database for a user that already exists. In our case we have defined the username as the primary key for a user, so the search is done by looking for a particular username. The primary key is defined in GameUserPK.java (shown in listing 7.4).

Since we have chosen to use container managed persistence, the container will search the database for us. Because username is a primary key, we do not have to specify how the database should be searched. If we also wanted a finder method like `findByAge(int age)` we would have to do some more work. Of course we would need to introduce an age-field in the deployment descriptors. We would also need someway to specify how the code for `findByAge(int age)` should be generated. This is done with EJB Query Language(EJB-QL)[2], which is similar to SQL in syntax. There is a `SELECT` clause, a `FROM` clause, and an optional `WHERE` clause. In our example the EJB-QL query would look like listing 7.5.

[2]EJB-QL is explained in [RAJ02] page 569

```java
public class GameUserPK implements Serializable {

  public String username;

  public GameUserPK(String username) {
    this.username = username;
  }

  ...
}
```

Listing 7.4: GameUserPK.java.

```
  SELECT OBJECT (a)
  //GameUserBean is our abstract schema name defined in ejb-jar.xml
  FROM GameUserBean AS a
  // ?1 means the first parameter
  WHERE a.getAge = ?1
```

Listing 7.5: EJB-QL.

Since there can be more users of the same age, the method will return a collection which should be defined in the method header.

7.5 Authentication of Users

The system should verify a user that logs in by comparing the password typed on login with the password that the particular user was registered with. We will solve this by letting the `LoginServlet` invoke a `verify` method on `UserHandler`. The servlet passes the entered username and password of the user to the method. The `UserHandler` bean then calls the `findByPrimaryKey` on `GameUser` with the username, which returns the entity bean responsible for the data related to the particular user. From this bean, the registered password can be fetched and matched with the typed password. Figure 7.1 shows this flow.

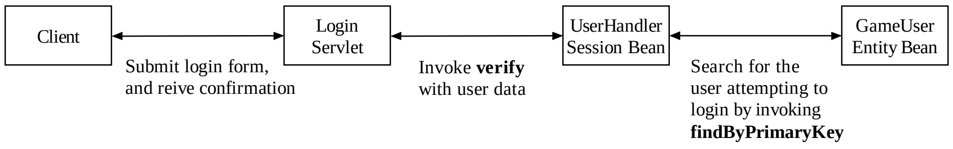

Client
Login Servlet
UserHandler Session Bean
GameUser Entity Bean
Submit login form, and reive confirmation
Invoke **verify** with user data
Search for the user attempting to login by invoking **findByPrimaryKey**

Figure 7.1: The flow of control when a user logs in.

7.5.1 Remembering Authentication

When a user sends a request to one of the servlets, the servlet should only relay the request if the user is logged in the system. This means that the servlets should remember whether a user is logged in or not. We will let the login servlet store that the user is logged in in the session context. Each time a request is sent to a servlet, the logged in status can then be looked up in the particular user's session context. If the user is not logged in, the servlet redirects to the login servlet.

This solution means that authentication only works if the system is accessed through the servlet layer. It is possible to remember authentication by storing it another place than the servlet context e.g. using JNDI. We do in fact store users through JNDI (for keeping track of which room a user is in etc.), so this would not be a major change to the system.

7.6 Summary

We have made the choice to implement the access to user information using an entity bean with a session facade pattern, in order to obtain data consistency and portable code.

The user interface is implemented using JSP, and we remember the users login status in their servlet session context.

Next we will look at how the game server framework is created.

Chapter 8

Game Server Framework

Having decided how user authorization and authentication is managed, we can move on to how games should be handled on the server side of the game framework. We will discuss the distribution of functionality in servlets and enterprise beans, and how this affects our choice of using messaging as the primary means of communication.

Indirectly, we have already made the choice to make the game server stateful by saving information about users in JNDI. We will continue discussing this subject to describe why we want the server to be stateful.

8.1 Games

Since the game framework should be able to handle the presence of multiple game types (e.g. chess and backgammon), we need a way to differentiate the game types. Having chosen messaging as the communication paradigm, this means that we must somehow separate messages from different games. We have already defined our message types, but as mentioned in chapter 2, JMS has several other message types, that we can use. JMS messages can also contain properties, which can be used by subscribers to choose only specific messages from a queue or topic.

When a game is played, each game event performed by a client should be transmitted to all other clients of that particular game. This is one-to-many communication and we will therefore use the topic publish/subscribe message domain that JMS offers. This means that events are published to a topic, and clients interested in certain events must subscribe to the specific topic. We needed a way to separate messages from different games, so we can create a topic for each game type that is installed in the game system. Figure 8.1 shows this principle.

A client playing chess will send events to the topic ”Chess Topic”, and this will naturally keep the messages from different games separated. But a user playing chess does not want to receive messages about other chess games running simultaneously. Therefore, we must separate the messages even further, so that we can differentiate both rooms and tables.

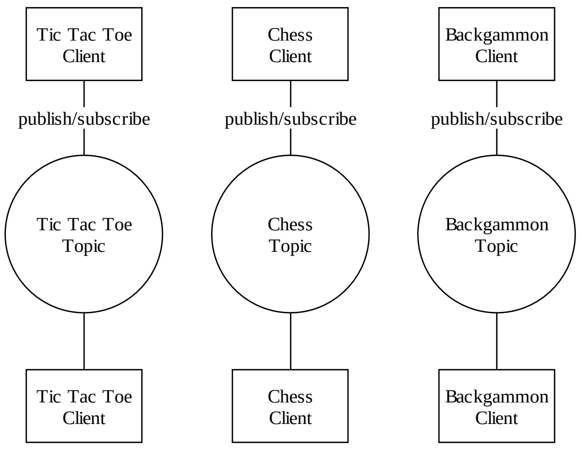


Figure 8.1: Games and topics

8.2 Managing Rooms and Tables

Our model described in chapter 3 states that rooms are objects with a reasonably long life cycle, while tables should be created and destroyed to match the amount of users present in the system. From this description it follows that the system should be able to create tables dynamically while running. The creation of rooms, however, can be done as a configuration of a particular game. That is, if a game developer wants two rooms (e.g. a beginner and an expert room), then this can be stated as a configuration of the particular game.

We still have the issue of where to put the basic functionality of the game framework. We have already defined a servlet layer, the `UserHandler` session bean and the `GameUser` entity bean. One more enterprise bean type, which we have not yet used, still exists; the message driven bean. The fact that we are using messaging as communication, and that this project is about exploring J2EE, we will of course bring this bean type to use.

Having stated that, using EJB technology is not always the optimal solution. Not all projects may benefit from using enterprise beans, and some might indeed even suffer from it. They are recommended if you want to be able to develop different user interfaces on your existing business code. Not having done this for the project, we still like the idea that without too much trouble, one can develop e.g. a mobile phone client that uses the same J2EE beans as the current web client – especially that a web client and a mobile phone client would then actually be able to play a game against each other.

8.2.1 Rooms

Since we need to differentiate messages sent to different rooms, we create a new message driven bean, the `RoomMessageBean`. As mentioned, JMS messages can be selected by their properties, and we would therefore make due with deploying only one message driven bean for each game. The bean would then have the responsibility of relaying messages to the correct rooms. However, it will be easier to implement the bean if each instance is only responsible for one particular room. This follows the idea of letting game developers create rooms as a configuration to their games, since each message driven bean would need to be deployed into the game system. Figure 8.2 shows this principle.

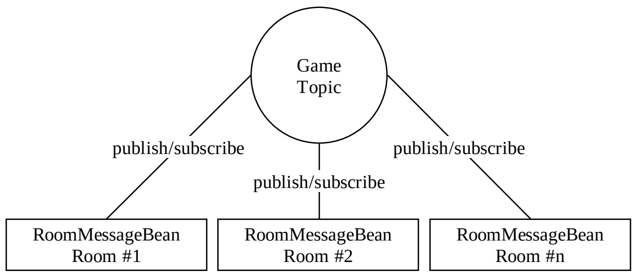


Figure 8.2: A game topic and deployed message driven beans

Just letting the message driven bean subscribe to a game topic will not do the trick, since this would still mean that it would receive all messages regarding that particular game type. If we set the property `room` with an identification number (1,2,3...) on each message sent in the system, we can configure the bean only to receive messages with a specific room identification number by using a message selector. So, a message driven bean deployed to handle a backgammon expert room with identification number 1 will subscribe to the topic "Backgammon Topic" with the message selector `room = 1`.

As a technology, message driven beans only support asynchronous subscription to a topic. On figure 8.2 we have stated that each bean must also be able to publish messages to the particular topic, and this must be implemented independently. Also, as mentioned, each client will need the ability to publish messages. In fact, it does not really matter which publish object either a client or other components will use, so we only need one for each game (that is, for each topic). This is good, because we want to attach as little information as possible to each client in order to save memory.

To publish messages to a room we again use the `UserHandler` session bean. In chapter 6 we constructed a local interface to the session bean, which included the method `sendMessageToRoom`. This method can be used by both clients and other components (such as the message driven bean) to send messages, but we need to expand the interface a little bit to allow the possibility to send private messages to a specific user in a room. We introduce the method `sendPrivateMessageToRoom`. An example of this is an answer to a request to sit down by a table, which will be handled in a moment. Figure 8.3 shows the road of a client message from being transmitted by the client to ending up in the message driven bean. Listing 8.1 shows the implementation of the method `sendMessageToRoom`.

Having defined how rooms are implemented, we still need to clarify how the system is informed of a user wishing to enter a room. First of all, according to our model, there is always space for one more user in a room, so we only need to register the particular user in the particular room.

```java
public void sendMessageToRoom(GeneralMessage m) {

  //Lookup the user who is sending the message
  JMSUser user = getUser(m.getFrom());

  //Set the destination of a message to be a room
  m.setDestination(GeneralMessage.DESTINATION_ROOM);

  try {
    /*Create an JMS object message, which can contain
    the GeneralMessage object */
    ObjectMessage om = getTopicSession().createObjectMessage(m);

    /*Setting the property username to an empty
    string indicates that this message is intendeded for all clients*/
    om.setStringProperty("username", "");

    /*Set the room property to the room
    that the user is currently in */
    om.setStringProperty("room", String.valueOf(user.getRoomId());

    //Publish message
    getTopicPublisher(game).publish(om);
  }
  catch (JMSException e) {
    System.err.out("Error sending message to room: " +
      e.getMessage();
  }
}
```

Listing 8.1: Sending a JMS object message to every user at a table (from the `UserHandler` bean).

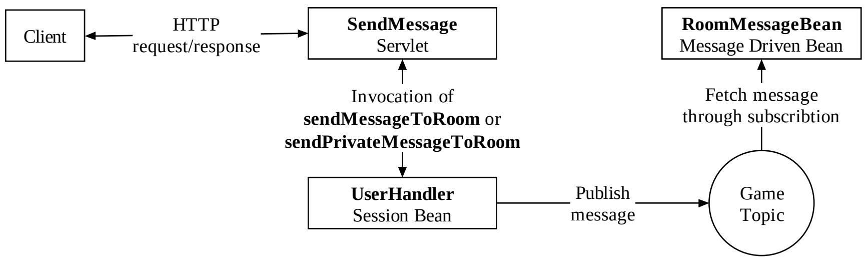
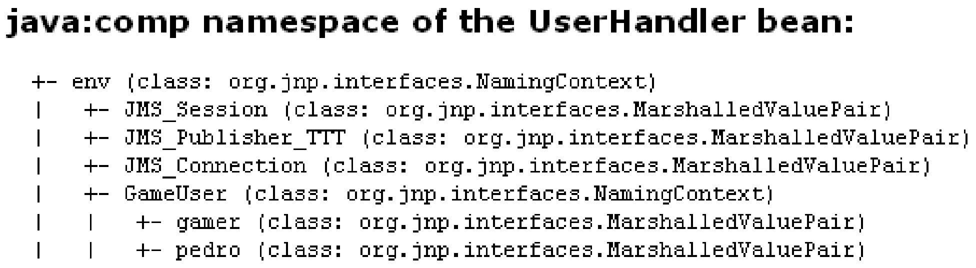

Client
HTTP request/response
SendMessage Servlet
RoomMessageBean Message Driven Bean
Invocation of sendMessageToRoom or sendPrivateMessageToRoom
Fetch message through subscribtion
UserHandler Session Bean
Publish message
Game Topic

Figure 8.3: The road of a client send message

We defined the `RoomServlet` in chapter 6, and making a request to this servlet means making a request to join a room, where the room identification number can be specified as a parameter to the servlet. We also defined the method `registerInRoom` in the `UserHandler` bean. The servlet will relay a request to this method, which in turn will create a topic subscription to the particular room (again using a message selector to filter out messages to other rooms) on behalf of the client. The subscription is saved in a user object placed in the `UserHandler` beans' private JNDI environment along with the room identification number. Figure 8.4 is a screenshot of how users are stored in JNDI. The figure also shows the object `JMS Publisher TTT`, which is the publisher dedicated to the game Tic Tac Toe. As mentioned, this is the publisher that is used every time a message is sent to any room dedicated to that particular game[1].



java:comp namespace of the UserHandler bean:

```
+- env (class: org.jnp.interfaces.NamingContext)
|   +- JMS_Session (class: org.jnp.interfaces.MarshalledValuePair)
|   +- JMS_Publisher_TTT (class: org.jnp.interfaces.MarshalledValuePair)
|   +- JMS_Connection (class: org.jnp.interfaces.MarshalledValuePair)
|   +- GameUser (class: org.jnp.interfaces.NamingContext)
|   |   +- gamer (class: org.jnp.interfaces.MarshalledValuePair)
|   |   +- pedro (class: org.jnp.interfaces.MarshalledValuePair)
```

Figure 8.4: A screenshot of users and JMS objects stored in JNDI

8.2.2 Tables

We now have a number of `RoomMessageBean` instances dedicated to receiving messages for their particular room. A message can either be sent to a room or a table in a room. If a received message is sent to the room, then the bean can handle it internally. Messages sent to the room can e.g. be information about users sitting and leaving tables. However, if a message is game specific, and thereby intended for a specific table, then the message bean needs to distribute the message to the correct room. Furthermore, the final destination of a game specific message is at the actual game implementation provided by a game developer. Before being able to handle this, we must be able to handle presence of game specific tables.

As stated in chapter 5, we have chosen to implement our framework based primarily on inheritance. We therefore need to construct an abstract table class, which game developers must use to construct their server side implementation of a game. We must force the developers to implement methods to handle incoming messages as well as the joining and leaving of users. We construct the abstract class `GameTable`, which is described in listing 8.2.

Besides its abstract methods, which the game developer must implement, the class itself implements functionality for sending and broadcasting messages to the clients playing the particular game that a table implementation is dedicated to. This is needed, because the server needs to coordinate ongoing games in order to e.g. check game consistency.

The system does still not know how to create table instances of specific games, and this actually presents a problem. How can the game system know the names of game specific classes not even implemented yet? As mentioned, each game implementation should come

[1] There is a small problem when saving JMS objects in JNDI, since they are not serializable. We have avoided this issue by wrapping the objects in a serializable adaptor, and making sure that the objects are still valid after each read

```
public abstract class GameTable implements Serializable{
  ...
  public abstract void onMessage(GameMessage m);
  public abstract boolean join(String username);
  public abstract void userHasJoined(String username);
  public abstract void leave(String username);

  protected void broadcast(GameMessage m) { ... }
  protected void send(GameMessage m, String username){ ... }
}
```

Listing 8.2: The `GameTable` class.

with a configuration. In this configuration, the game developer must indicate which class he has developed that extends the `GameTable` class. The game system can then read the configuration and create the needed table instances. We will go into further details about the configuration of a game later in this chapter. We now have a list of table objects associated with each `RoomMessageBean`, and the bean can now distribute messages to specific tables simply by looking up tables in this list. Figure 8.5 shows this.

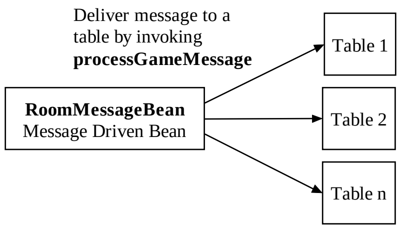

Deliver message to a table by invoking **processGameMessage**
RoomMessageBean
Message Driven Bean
Table 1
Table 2
Table n

Figure 8.5: `RoomMessageBean` distributing messages to tables

Just like when a message destination was a room, a message sent to a table needs to be tagged such that the system knows that it is meant for a specific table. The `UserHandler` method `sendMessageToTable` takes care of this by setting a table property on messages as shown in listing 8.3.

8.2.3 Sitting Down at a Table

When a client wants to sit down at a table, we have to make sure that no synchronization problems occur e.g. if two users were allowed to sit down at a table with only one free seat. It is clear that there has to be some control of access to a table, so this problem does not occur.

At any given time, each client will have a view of how tables in a room are occupied by other clients. At some point, the client will sit down at a table with room for one more player, but another client might have made the same choice before being notified about the first client

```
public void sendMessageToTable(GeneralMessage m) {
  ...
  //Set the destination of a message to be a table
  m.setDestination(GeneralMessage.DESTINATION_TABLE);

  try {
    ObjectMessage om = getTopicSession().createObjectMessage(m);
    om.setStringProperty("username", "");
    om.setStringProperty("room", String.valueOf(user.getRoomId());
    om.setStringProperty("game", user.getGame());

    //Set the property table indicating which destination this message has
    om.setProperty("table", user.getTableId());

    getTopicPublisher(game).publish(om);
  }
  catch (JMSException e) {
    System.err.out("Error sending message to room: " +
      e.getMessage();
  }
}
```

Listing 8.3: The method `sendMessageToTable`.

sitting down. To avoid this problem, each client will need to ask for permission to sit down by a table. The server can then give the free seat to the first client asking, and deny the request from the second client. Figure 8.6 shows this protocol.

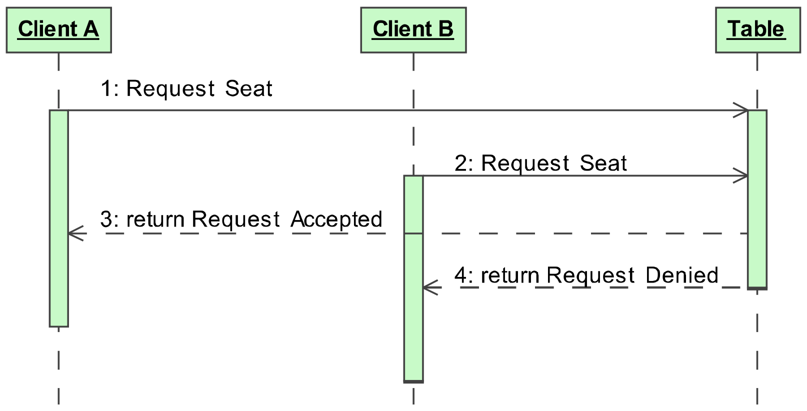


Figure 8.6: Clients asking for permission to sit down by a table with only one free seat

We have defined the method `join` in our `GameTable`, which must be implemented for each game by a game developer. This method should be thread safe, since we otherwise would still have the problem of more clients being assigned the same seat. The game developer should not be forced to make the method thread safe himself, so the game system will take care of this by using synchronized methods. When a client calls `join`, the game specific implementation should reply with a response indicating if it allows the client to join or not.

This solution distributes the responsibility of handling the users present at a table to the game specific implementation. The task could, however, be handled by the framework in a more elegant manner by allowing a parameter to `GameTable`, which should indicate the amount of players allowed in a specific game[2]. The game implementation would then be able to access the users present at a table by invoking methods on `GameTable`.

8.3 Game Configuration

As mentioned, each game implementation must be followed by an associated configuration. We have already determined that this configuration must at least contain the name of the game-specific class implementing the abstract class `GameTable`. Each game type was represented by a JMS topic, and since the game system cannot predict which topics future game implementations will use, a topic name must also be provided in the configuration.

As a security precaution, we also have to protect the game topics from unwanted access. Unauthorized people subscribing or publishing messages to a topic, might create chaos as messages could potentially be “spammed” to a topic. Protecting a topic can be done with username and password in JBoss, so we also need the configuration to contain a valid topic username and password. We can use a simple security scheme like this, because only the server side of the game system will ever publish or subscribe to game topics[3]. This means that the username and password will never exist on the client side.

Intuitively, a game configuration could be supplied as a simple text file. However, direct file access inside enterprise beans is prohibited by the J2EE specification for different reasons[Bro03]. Instead, the specification suggests using a resource manager API, such as JDBC, to store data. Following this suggestion, game configurations should be put in a table in the database, and accessed through an entity bean with a session facade. We will construct the entity bean `GameProperty`, which encapsulates one configuration property, and the session bean `PropertyHandler`, which will be the session facade. The implementation of this concept is almost identical to the user handling of the game system described in chapter 7.

8.4 Framework

Until now, our server side framework inheritance model only consists of the one class `GameTable`. However, according to our model, we also need to design some kind of moderating service to the game developers. We have the problem that the rules for different board games obviously

[2]Some games allows an unlimited amount of players, so this should be handled too
[3]We determined this in chapter 6 about communication

```java
public abstract class GameModerator implements Serializable {

  public static final int STATE_ERROR = -1;

  protected int state;

  public abstract int event(int event);
  public abstract boolean ingame();
  public abstract boolean error();
  public abstract boolean waitingForPlayers();
  public abstract int getInitialState();
  public abstract int getInitialIngameState();
  public abstract int getEndOfGameState();
  public abstract int isWinningCondition();

  ...
}
```

Listing 8.4: The class `GameModerator`.

differ to a great extent. In other words, we cannot create a standard moderating service that will work for all board games.

Instead, we will introduce a `GameModerator` class as an optional feature, that game developers can choose to use to enforce their rules. The `GameModerator` object can implement common properties of board games like the ability to be in a state, execute an event and check for a winning condition. The game developers will, however, still need to implement the actual states, event executions and winning conditions themselves. Listing 8.4 shows the `GameModerator` class.

As mentioned, a game developer might find that he could benefit more from constructing an independent moderator, and this will be allowed in the framework. This leads us to the final inheritance model, which is shown in figure 8.7.

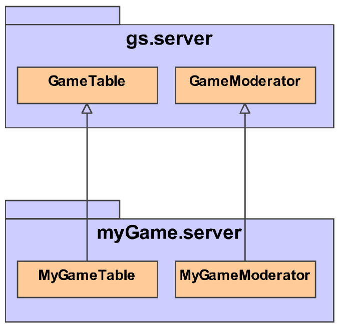


Figure 8.7: The two framework server classes and a simple example of inheritance from them.

8.5 Stateful and Stateless Server

By choosing to store data about each client in JNDI, we have already made the choice to implement a stateful server. Given our type of application this makes perfect sense, since the job of the server is to keep track and control of games being played. As suggested in section 4.2.1 we *could* probably have used a stateless server implementation, but this would require a considerably larger amount of network traffic for transferring the complete game state to the server at every enquiry. Furthermore, we would not be able to judge which client's game state update was correct if they differed from each other, unless we sacrificed yet *more* network traffic and transferred both the old game state and the new proposed ones (or the proposed changes to the old state).

Implementing the server as a stateful server is much more straight forward, since by doing that, the only correct representation (per definition) of the game state is kept by the server (the clients have only copies for their GUI representation) and the only messages, we need to send between clients and server, are those of events in a game. There is, though, the disadvantage that if a stateful server crashes, game states will not be saved. In a stateless server implementation, the clients would just have to wait until the server comes back up, whereafter they can continue their games, since state handling is moved from the server to the clients.

8.6 Transactions

When handling events on the server, it is often necessary to perform a list of actions. These actions will normally be tied together in such a manner, that either the entire list of actions are performed, or none are performed. This is handled by using a transaction handler. A list of actions can be connected into one transaction, and J2EE will ensure that the actions are handled according to ACID properties[4].

J2EE supports three kinds of transactional boundary[5]; programmatic, declarative and client-initiated. A declarative boundary allows us to define an entire method as a transaction, and thereby avoiding direct call to the transaction handler inside our Java code. Which methods that should be handled as transactions can be controlled through the enterprise bean deployment descriptor.

Instead of analysing each bean method to determine the need for transaction handling (which of course requires some time), we have chosen to define all bean methods as transactions. This obviously adds some overhead because every method call will have to go through a transaction handler. While it does ensure the atomicity of our actions, this design choice should be reevaluated if the game system ever makes it to a real production environment.

[4]Transactions and J2EE support are further explained in [RAJ02] chapter 10

[5]The notion of who initialises and commits a transaction and when

8.7 Summary

We have now constructed the game server framework, and by doing so, we have looked at the advantages and disadvantages of using J2EE beans for business code instead of servlets. We have also looked at how the abstraction of rooms and tables could be implemented given our defined model.

The chosen representation of a game type is a JMS topic, while rooms are represented by message driven beans. Tables are implemented by game developers, and the distribution of messages to them are handled inside their associated rooms. This distribution ensures that developers can focus on implementing one instance of their game, and not worry about where or how many instances are actually run, which was a requirement set in chapter 5.

The final server inheritance model consists of only the class `GameTable`, and the optional class `GameModerator`. The extent to which the system manages users joining and leaving tables can be argued, and we will look at this principle at the end of the project.

Before our framework is complete, we need to construct the client side of the system.

Chapter 9

Game Client Framework

In the previous chapter we discussed the design of the server side framework. In this chapter we will design the client side of the framework. We will look at which requirements our abstract model, described in chapter 3, sets for the client framework, and then model the lifetime of a potential game client. Defining the lifetime will clarify the states that each game client will go through during a session with the system.

Each game will take place in an applet[1], so we will look at how applets are initialised, and what a game developer needs to do in order to make this initialisation possible.

All communication between a game and the server is handled through messages. We defined different message types in chapter 6, and we will describe how these can be used to handle events inside the framework as well as events associated with actual games. The main problem, in this context, is how the framework can distribute game messages to game implementations.

To allow users to play a game, we need to define a user interface. Some of the user interface will be general for all games, while other parts are specific for each game. We want to minimize the effort, that game developers will have to put into creating their game specific user interfaces.

Last in this chapter is a continued discussion of the advantages and disadvantages of having functionality on the client, contra having it on the server side.

9.1 Client Requirements

In chapter 3, we constructed a model of the game system. Using this model, we can define a list of requirements for the client framework.

Send and Receive Events When an event occurs, there should be some way to send this event to the correct clients. Our model states that the following events and actions can occur.

[1]This was decided in chapter 6

1. Enter room or table
2. Leave room or table
3. Receive room or table user list
4. Write and receive chat message
5. Perform and receive game events

How these events are handled by the client will be described in section 9.5.

Knowing Other Clients in Room and at Table The sequence diagram shown in figure 3.15 in chapter 3 shows the next requirement. Here the model states, that when entering a room or a table, a list of the active client names should be returned. This requirement makes it possible for a client to see who is in a given room, as well as see the names of the clients participating in ongoing games.

Client Name Following from the previous requirement, it should obviously be possible for game developers to get the name of the client logged in on the local machine. This requirement is not stated explicitly in our model, however it is a logical requirement.

User Interface The client should have a user interface available. In chapter 6, we decided that each game will take place in an applet, so this is already defined.

In the following sections we will describe how these requirements are designed. First, we will look at the lifetime of a game client.

9.2 Lifetime of the Client

Our model defines the states that will be entered during the lifetime of the system. We will expand on this by modelling the lifetime of a game client. The authorization and authentication of users are already dealt with in chapter 7, so we will only consider the lifetime from the point where a client enters a room, and until this room has been left. Figure 9.1 shows an activity diagram describing the client activities.

When a room is entered, the graphical user interface is changed to show the particular room. The view shows the room chat, along with available tables in the room. The chat has a list of clients currently in the room, and each table shows the clients playing a game.

When a client chooses to sit down at a table, a new window should open, showing the game being played. The room view should still be visible, since the client might want to arrange future games in the room chat. When a game ends, the table view is shut down. The lifetime of a game client (in this context) ends when a room is left.

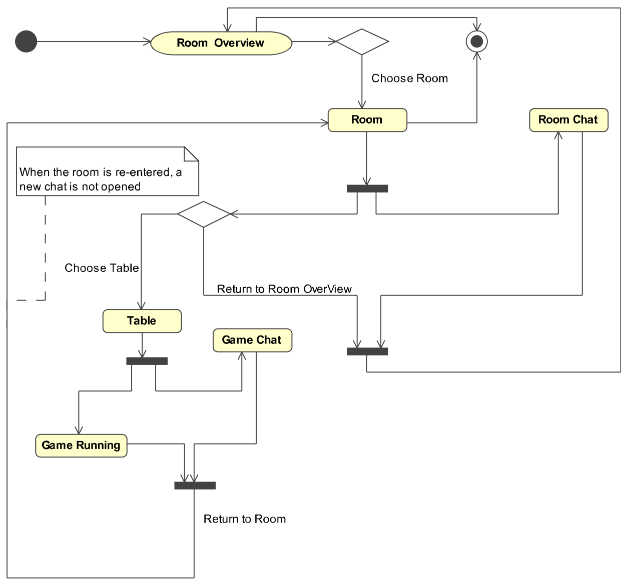


Figure 9.1: Activity diagram showing client activities.

```
<applet code="gs.client.RoomApplet"
        archive="TicTacToeClient.jar,GameSystemApplet.jar" ... >
                <param name="<%=Constants.KEY_USERNAME%>" value="<%=username%>">
                <param name="<%=Constants.KEY_ROOM%>" value="<%=roomId%>">
                <param name="<%=Constants.KEY_GAME%>" value="<%=game%>">
</applet>
```

Listing 9.1: Initialising the `RoomApplet`.

The central elements in our model are rooms and tables. We will therefore define an applet to be the representation of a room, and let the applet open a window to represent the table, that the local user is sitting at.

9.3 The Room Applet

Before entering a room, the user interface that a client uses to navigate the system is constructed in HTML[2]. When a room is chosen by a client, the user interface becomes an applet. We introduce the `RoomApplet` as the applet representing a room. To initialise applets, they must be integrated into an HTML page by using the `<applet>` tag. Listing 9.1 shows how the `RoomApplet` is inserted into a JSP file.

[2]As a normal web site

The `code` argument shows which applet to run i.e. the `RoomApplet`. The `archive` parameter defines the names of the archives containing the needed classes. We have two archives; `GameSystemApplet.jar` which contains the game client framework, and `TicTacToeClient.jar`, which contains a Tic Tac Toe[3] implementation.

The applet is passed three parameters; the username of the user entering the room, the room identification number, and the name of the game being played in the room. Having the username in the applet means that game developers will have access to it, thereby fulfilling one of our requirements. The room and game parameters are used by the applet to request the correct data from the server, i.e. the data for the entered room and associated tables.

9.3.1 Game Properties

In section 8.3, we described how the server framework uses game properties to know which game specific classes to create instances of. In effect, the server framework uses properties to “glue” the framework together with game implementations. This solution supports the concept of a framework very well, since no changes are needed inside the framework in order to support new games.

Similarly, we need game properties on the client. We can let the game developers define the properties in a file, but the name and location of this file should always be the same, so that the system can locate it. We therefore define, that each game extending the framework should have a file, `game.properties`, included in its distribution – thereby defining the names of the extending classes. The client framework will look for this file as part of its initialisation.

9.4 Sending and Receiving Messages

As stated earlier, the client needs a way to communicate with the server. In chapter 6, we described how Java objects can be transmitted through an HTTP connection, and introduced the concept of a message listener (see figure 6.3). Since our message types are obviously Java objects, we can send them directly using HTTP.

9.4.1 Sending Messages

Sending a message from a game should be straight forward for the game developer. One of our design criteria is communication transparency, so we need to construct a level of abstraction, hiding from the game developer that we are using HTTP. This will also ensure, that the means of communication can change from HTTP to another protocol without any changes needed to the already existing game implementations. We do, however, not find it necessary to hide completely from the developer, that he is in fact sending information through a network.

[3]A Tic Tac Toe implementation using the game framework is described in chapter 10

```java
        public abstract class GameMessageHandler {
                ...
                public abstract void onMessage(GameMessage m);
                ...
        }
```

Listing 9.2: The abstract class `GameMessageHandler`.

We will therefore construct the method `broadcast`, which the game developer can use to broadcast messages to the table, that the current client is sitting by.

However, having defined the method, we also need to give developers access to it. We will again use inheritance, and let the developers inherit from certain classes to obtain the functionality of the framework. We will discuss this matter in section 9.7.

We talked about message types being Java objects. We do not want game developers to broadcast just any type of objects to the table, since we need certain properties following every message. In chapter 6, we defined the message type `GameMessage`. By forcing developers to build their game events upon this message type, we can easily separate specific game events from other events occurring in the framework. As an example, a developer constructing the game Tic Tac Toe will need to represent the event of placing a piece in a board cell. He should then construct a message type named `SetPieceMessage`, and let it extend `GameMessage`.

9.4.2 Receiving Messages

In chapter 5, we listed a requirement of providing asynchronously message reception for game developers. Obviously, our design of a message listener using the HTTP protocol to fetch messages from the server at a fixed interval is not asynchronously. So, again we need to create an abstraction in order to obtain this criteria.

We introduced the term message handler in chapter 6, and this is the mechanism which we will use to obtain asynchronously message reception. By forcing game developers to implement an `onMessage` method to handle incoming messages, we can call this method every time our message listener fetches a message from the server. To force the method implementation we construct the abstract class `GameMessageHandler` shown in listing 9.2. Game developers should implement a class that extends `GameMessageHandler`, and specify the name of their implementation in the game properties file.

The path that an incoming message will take inside the system is illustrated in figure 9.2. When the framework message handler receives a message, the first thing it does is determine if the destination of the message is a room or a table. If the destination is a room, then the framework should handle the message, otherwise it should be relayed to the game specific message handler.

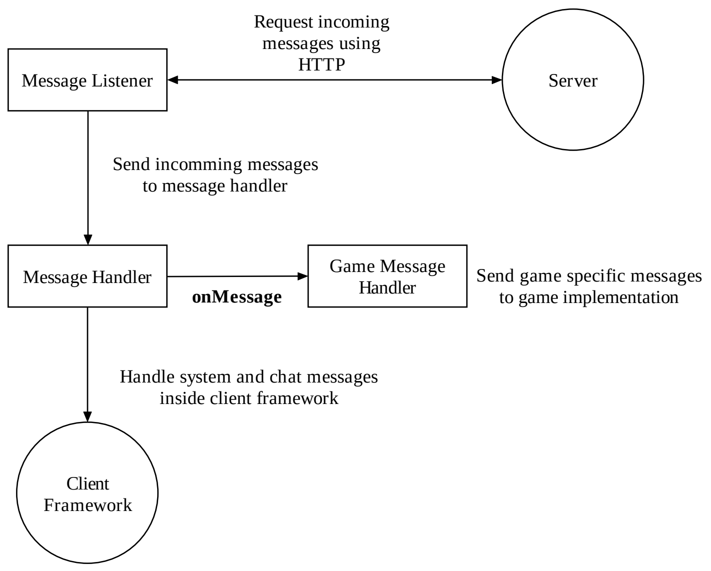

Request incoming messages using HTTP
Message Listener
Server
Send incomming messages to message handler
Message Handler
onMessage
Game Message Handler
Send game specific messages to game implementation
Handle system and chat messages inside client framework
Client Framework

Figure 9.2: Diagram showing the handling of incoming messages

From the game developer's point of view, this concept will provide asynchronious message reception as required.

9.5 Handling Messages

We have now specified how messages are received by the system, and where they are handled. We still need to design how they are handled. In the last section, we described how table messages are handled by the game specific `GameMessageHandler`, so we only need to design how room messages should be handled by the client framework.

A message for a room is either of the type `SystemMessage`, or of the type `ChatMessage`. Chat messages are obviously easy to handle, since they can be inserted almost directly into the graphical user interface of the client. Handling system messages is, however, a bit more complex. We will describe each possible incoming system message, and how we can handle it, in the following.

Userlist Message

Upon entering a room, a client will receive a list of users currently inside the particular room, as well as information of which tables they are sitting at. This information should of course be relayed to the graphical user interface, such that the client can begin to arrange games with other players.

Client Joining or Leaving

Instead of resending the entire list of users and their table location every time someone enters or leaves a room or table, each client should receive notification of these events. This will allow each client to maintain their own information regarding the presence of other players, and save a lot of network bandwidth. As described in chapter 8, the client must follow a specific protocol when requesting to sit down at a table. This means that the client can receive a message stating that his request to sit down has been accepted. In this case, the system should open a new window containing the particular game being played.

While waiting for a seat request response, the system should not allow the client to request any other seats. So if a request to sit down is denied, then the system should again allow the client to request other seats.

9.6 User Interface

All of the work done earlier would have been for nothing, if there is no way for the user to interact with a game. Therefore, we need to design how game developers should construct their graphical user interface. We have already designed that the room interface is constructed in an applet. Figure 9.3 shows a screenshot of the `RoomApplet`. Each game should open in a new window. A game developer could therefore just construct a new frame, and open it when his game is started. But using this solution would make it harder to provide additional features to the developer, since his user interface would be totally separate from the framework.

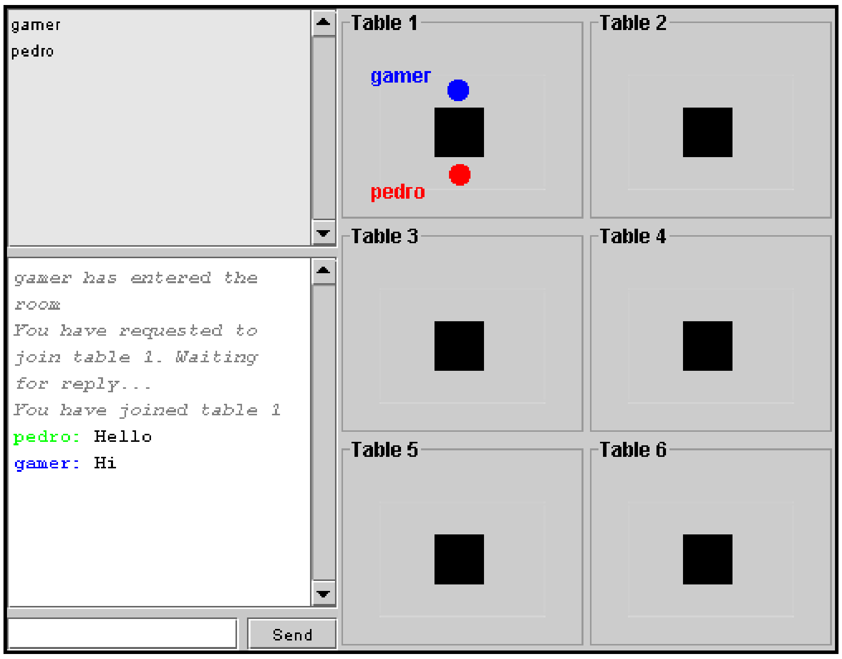
Instead we can construct an abstract class inheriting from `JFrame`, and use this class to provide features. One feature that will come in handy for developers, is the possibility to load images in an easy manner – since all games contain images like cards or pieces. We will name our abstract frame class `GameFrame`.

Here we can make use of our properties file once again. First of all, the user must specify the name of the class inheriting from `GameFrame`. Secondly, we can allow users to simply type the names of the images they want loaded as a property, and then let the framework load them. This way, all game images will be ready to use when a game is started.

9.7 The Framework

We said that we would get back to how game developers would be allowed to access the `broadcast` method. Having defined `GameFrame`, we can put the method into this class. Listing 9.3 shows the `GameFrame` class.

This completes the features provided by our client framework. To give an overview of the important classes of the client framework, we have constructed a class diagram showing the important classes and methods. The class diagram is illustrated in figure 9.4.



gamer has entered the room
You have requested to join table 1. Waiting for reply...
You have joined table 1
pedro: Hello
gamer: Hi

Figure 9.3: Screenshot of the `RoomApplet`

```
public abstract class GameFrame extends JFrame {
        ...
        public GameFrame() {...}

        public int getTableId() {...}
        public void setTableId() {...}
        public String getUsername() {...}
        public ImageIcon getImageIcon(String imageName) {...}
        public void broadcast(GameMessage m) {...}
        ...
}
```

Listing 9.3: The abstract class `GameFrame`.

The design of the client framework allows for an easy implementation of games. This is accomplished by having all the functionality, that is not game specific, handled by the framework Figure 9.5 shows the inheritance model of the client. Here we see the abstract classes meant for inheritance mentioned in this chapter, namely the `GameMessageHandler`, the `GameFrame`, and the `GameMessage`.

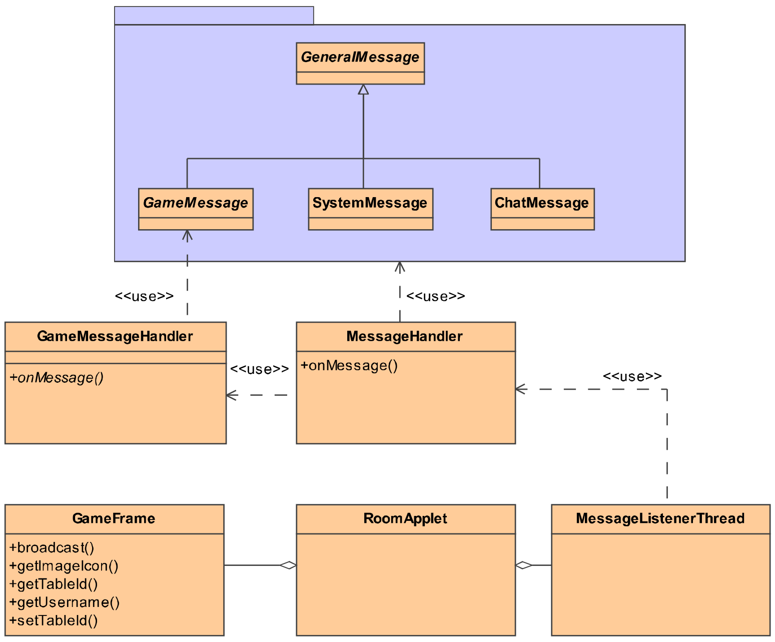


Figure 9.4: Class diagram showing important client framework classes

9.8 The Distribution of Functionality and Data

Following from the discussion in chapter 4 about the distribution of functionality and data[4], we will here describe the choices made in the client framework.

Some choices have already been made on behalf of the game developer. First, as mentioned above, there is the fact that the client is running on a web page inside an applet. This choice would indicate that this is a thin client, in that there is no explicit download of software. Another factor is how the communication is implemented. If EJB's are contacted directly from the client, there will have to be some implicit download of EJB functionality onto the client, making it thicker. The same is the case if JMS is used directly from the client. If, instead, the connection is through one or more servlets, less data needs to be downloaded, and thus making the client that much thinner.

The game developer also has some influence on how thick the client is going to be. Obviously, there is some required functionality on the client due to the design choices made in the framework, but the developer still has to choose how often a game event is transmitted to the server.

Here, we have the problem of how much functionality we want to have on the client. Does a game developer want his client to handle the rules of the game on the client side, leave it to the server, or have both perform the checks. The advantage of leaving it to the client is of course

[4]Also known as the Thin Thick problem

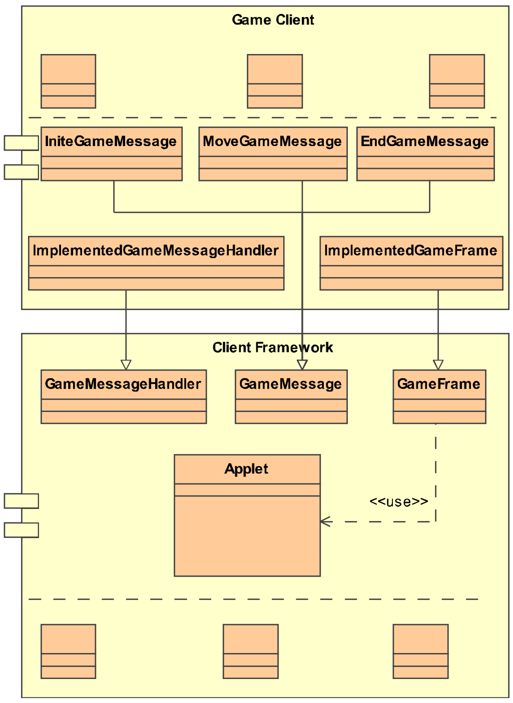


Figure 9.5: Implementation diagram showing the client inheritance model

the less bandwidth usage, which in turn adds to the speed of the application. The downside is the lacking security. The opposite is true, for the case of leaving it all to the server.

The third alternative is a hybrid of the two, a way to get the best of both worlds i.e. the high security of the thin client, along with the speed of the thick client[5]. An example of the advantages of the hybrid is the following: Consider a player turn in backgammon. When the dice have been rolled, the user can, depending on the roll and the state of the game, move one to four times. If there are rule checks on the client side, then the moves performed can be checked here, and, if found acceptable, be transmitted to the server in *one* message – thereby avoiding three server transmissions. Of course, the server will still need to double check the moves.

9.9 Summary

In this chapter we have discussed the design of the client framework. We used the model constructed in chapter 3 to setup a list of requirements to the framework. We showed how the `RoomApplet` was inserted into a JSP page, and that the parameters passed to the applet, was the backbone of the applet initialisation.

[5]This solution would probably be leaning towards the thick client solution

Next we designed how the functionality of sending and receiving messages should be handled in order to meet the requirements. It was shown that the distribution of messages to a specific game implementation was handled by letting the game developer inherit from certain framework classes.

To provide a user interface, and keeping the possibility to add future features to this user interface, we constructed an abstract framework table class. Through this class, we provided the possibility to obtain loaded images in an easy manner, and broadcast information to other clients at the table.

Last in the chapter, the issue of thick and thin clients in relation to the framework was discussed. Because the framework provides some functionality already, the choices regarding distribution of functionality is not entirely up to each game developer. However, as was shown, the game developer still have a say in the matter.

This completes the design of the game framework. We will now test whether the framework does in fact make the task of creating turn based games easy. We will therefore attempt to design the game Tic Tac Toe using the framework.

Chapter 10

Tic Tac Toe

To illustrate how a game developer would implement a game using the game framework, we have created a Tic Tac Toe game. Although the game is a rather simple game, it gives a good picture of what is required when creating a game using our framework.

10.1 The Game

Before digging into the code, we will go over some of the things a player would experience during a game. When a player sits at a table, the screen shown in figure 10.1 will be presented. When the player is the only one at the table, the text "Waiting For Players..." will be shown at the top of the screen, and nothing, except leaving the table, will be possible. Once another player joins, the game will commence and the player who joined first will be granted the first turn. While waiting for another player to complete a turn, a "Waiting For Other Player..." message will be displayed. Hopefully, both players will keep on playing until one of them wins and receives the "Congratulations" or "Loser" message, but it might happen that one of the players will drop out in the middle of a game. This results in the game resetting and the player left, will again see the "Waiting For Players...".

A thing to notice about our Tic Tac Toe game, is that the game reuses its noughts and crosses. In other words, whenever a player has used all his three crosses, he will have to move an already placed cross. This is simply done by pressing the cross, which changes the image of the cross as shown in figure 10.2 and then click on an empty space.

10.2 Implementing The Game

In the previous chapters, we presented the classes that the game developer had to inherit from. Just to sum up on these, they are all, along with their abstract methods, shown in figure 10.3. Although not required, the `GameModerator` class is also in the figure, since we will use the class in the game.

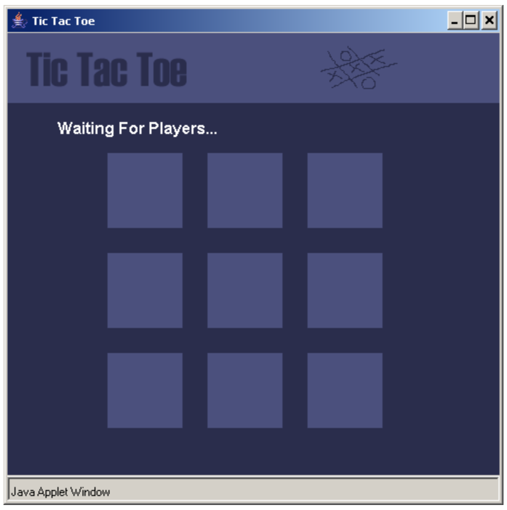


Figure 10.1: The first screen presented when a player enters a table.

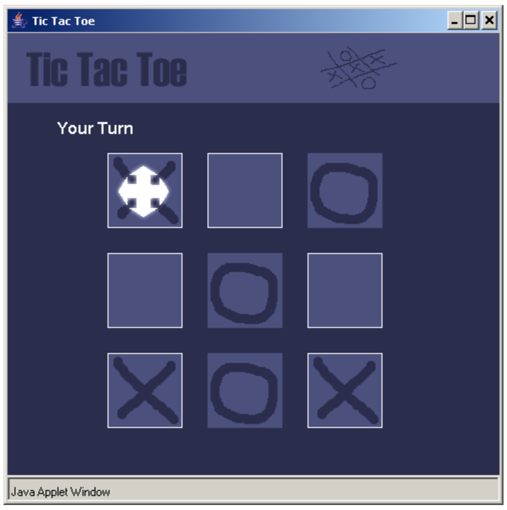


Figure 10.2: A player has used all his crosses and now moves a already placed cross

The actual classes implemented for the Tic Tac Toe are listed in figure 10.4. The model packages in the figures represent classes that are used both on the client-side and the server-side, e.g. messages that are being sent back and forth between the client and the server.

10.2.1 Game Events

In the Tic Tac Toe game, there are four different types of events or messages. The messages are listed in table 10.1 along with a description of their purpose. Notice that there are two

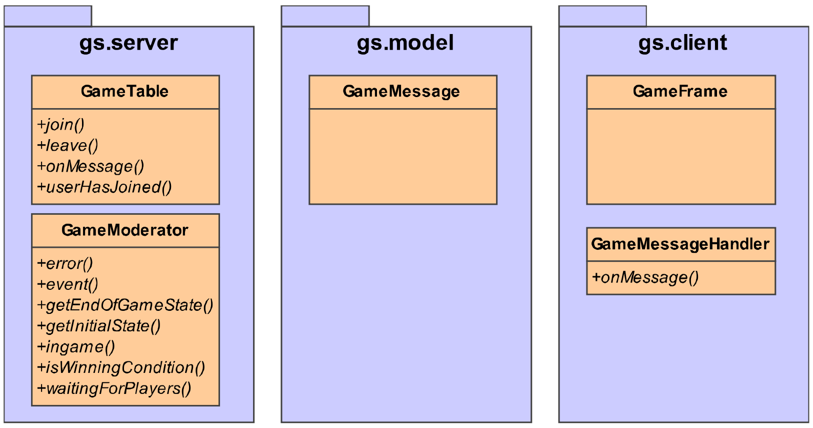


Figure 10.3: The classes required to inherit from

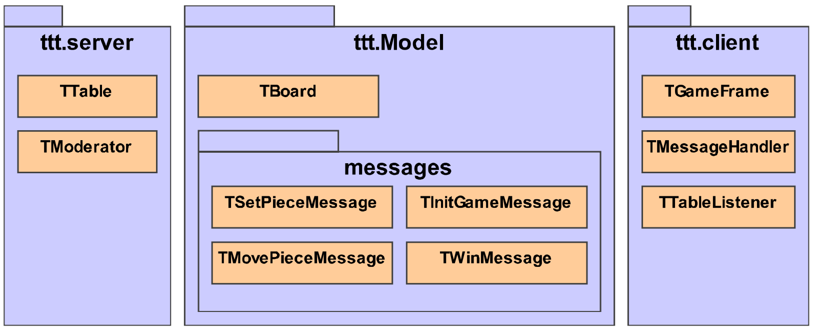


Figure 10.4: The classes implemented in the Tic Tac Toe game.

messages that involve pieces, a set and a move. This is due to the reuse of pieces as explained earlier.

The messages represent the events that can happen in the game. For example the `TInitGameMessage` is the event that triggers the game start state. This particular message is sent from the server, when it determines that a game should start.

Name	Description
TInitGameMessage	Initialize the game
TSetPieceMessage	Set a new piece
TMovePieceMessage	Move a piece
TWinMessage	Who has won

Table 10.1: The different types of messages used in the Tic Tac Toe game.

```
        board = new int[3][3]; //Initialize matrix
        for (int i = 0; i < board.length; i++) {
                for (int j = 0; j < board[0].length; j++) {
                        board[i][j] = CELL_EMPTY; //Fill the matrix with the
                            empty value
                }
        }
```

Listing 10.1: The board matrix is created and filled with the CELL_EMPTY value.

10.2.2 The Board

The `TBoard` class represents, as its name suggests, the actual board of the game, i.e. it keeps track of where the noughts and crosses are placed. The way it does this, is by using a 3x3 matrix where each space in the matrix has a value of either zero (for empty), or one or two (for player one or two, respectively). At the creation of a game, this matrix is filled with zeroes. This is shown in listing 10.1.

Both the server and the client will use this class to represent the game state as they see it (at any point in time).

10.2.3 Implementing the Server

As described in chapter 8, we need to extend the class `GameTable` to create a game server. We create the class `TTable`, which will be responsible for broadcasting and receiving messages on the server. The class will also keep track of the game state, so that it can determine if a game is played correctly.

All incoming game messages will be delivered to the method `onMessage`, which is implemented in `TTable`. When a message is received, the type should be established, and the message should be handled accordingly. In our game, the only messages handled here are `TMovePieceMessage` and `TSetPieceMessage`. In listing 10.2 the message type check is shown.

As mentioned, we use a moderator in the game (`TModerator`). It is responsible for handling the states in the game, i.e. are we waiting for a player, how many pieces have been set and so on. Besides just remembering states, it also checks for valid moves in the `event` method. In listing 10.3, we check for the case that player one has set a piece.

10.2.4 Implementing the Client

To receive messages on the client, we need to extend the framework class `GameMessageHandler`, which we will do with the class `TMessageHandler`. This class will be responsible for determining the type of message received, and thereby how the message should be handled. As an example of this, listing 10.4 illustrates how the `TMessageHandler` deals with a `TSetPieceMessage`.

```
public synchronized void onMessage(GameMessage m) {
        ...
        if (moderator.ingame()) {
                ...
                //Check which message we have received, and process it
                    accordingly
                if (m instanceof TMovePieceMessage) {
                        processMoveMessage((TMovePieceMessage) m);
                }
                else if (m instanceof TSetPieceMessage) {
                        processSetMessage((TSetPieceMessage) m);
                }
        }
}
```

Listing 10.2: The method `onMessage` in `GameTable`.

```
case EVENT_PLAYER1_SET: //Player one has set a piece
        //Check whether the state is one of the three set states
        if (state == STATE_PLAYER1_SET_1 || state ==
                STATE_PLAYER1_SET_2 || state == STATE_PLAYER1_SET_3) {

                state++; //Increase the state
                break;
        }
        else { //If something is wrong enter the STATE_ERROR state.
                state = STATE_ERROR;
                break;
        }
```

Listing 10.3: Checking the validity of a set event.

```
if (m instanceof TSetPieceMessage) { //Check the type
        //Cast the message to set piece message
        TSetPieceMessage sp = (TSetPieceMessage) m;
        //Set the piece on the table
        ((TGameFrame) applet.getGameFrame()).
                setPiece(sp.getRow(), sp.getCol(), sp.getFrom());
}
```

Listing 10.4: Handling a `TSetPieceMessage`.

```
        if (!movingPiece) { //Is it set or move
                if (frame.canSetPiece()) { //Can we place a piece
                        //Create a new set piece message
                        TSetPieceMessage sp = new TSetPieceMessage();
                        sp.setRow(row); //set the row the piece in in
                        sp.setCol(col); //set the column the piece is in
                        frame.broadcast(sp); //Send the message
                        frame.enableTurn(false); //Disable the players turn
                }
        }
```

Listing 10.5: A piece has been set by a player and the `TTableListener` sends the appropriate message.

As the example in listing 10.4 shows, the `TMessageHandler` uses a method in `TGameFrame` to set the piece. This class is the Tic Tac Toe extension of the framework class `GameFrame`. It serves two major functions: drawing the screen, and enabling/disabling player moves. The latter function makes sure that a player cannot place a piece on an already occupied field, or place a piece when it is not his turn. The `TGameFrame` class itself does not know whether or not there is a piece in a particular field, but instead asks the `TBoard` which, as explained, holds all the board information.

To capture user interface events, we have the class `TTableListener`. This class listens for and handles all the events a player generates. In the Tic Tac Toe game, the number of events are rather limited; moving a piece and setting a piece. When an event like setting a piece occurs, the `TTableListener` creates a message representing this event (e.g. `TSetPieceMessage`), and then broadcasts it. Sending messages from the client-side is again something the designer does not need to worry about. All that is needed, is to call `broadcast`. The set piece event is shown in listing 10.5.

10.3 Summary

In this chapter we have implemented the game Tic Tac Toe using the game framework that was constructed during this report. The implementation shows that the framework works as intended, and that creating turn based games is indeed relatively easy with the framework.

Tic Tac Toe is obviously one of the simplest turn based game in existence, but since it follows the same principles as all other turn based games, it serves well as an example implementation.

Chapter 11

Conclusion

In the preceding chapters we have described the development of the game system. We have modelled, designed and implemented the game system. Also, we have discussed different solutions to problems that have emerged during the project, and chosen the solution most appropriate for our problem. Now we will consider our solutions and compare them to our two main objectives for this project, namely:

- To learn about the Java 2 Enterprise Edition(J2EE) platform.
- To create a distributed framework to handle turn based games using J2EE.

11.1 J2EE

We wanted to gain insight into the J2EE platform, and what it offers. We soon realised, that this is a huge platform, which has a lot of opportunities. The main focus was therefore on a number of integrated technologies, i.e. JNDI, JMS, Servlets, JSP and EJB's. Our approach to learning about the technologies was through prototyping. We decided to make the chat component of the system, and make it work. The reason for this, was that the chat to a large extend, represented the way the system would work in general, and we therefore gained knowledge about the whole communication path from client to server.

It was straight forward to gain knowledge about the technologies, since there was a high availability of documentation on the framework, from both Sun Microsystems[1] and many others. This is a major strength of the platform.

The framework allows different styles of constructing each component. A developer can choose to let the container manage services (e.g. transaction handling), or choose to handle it by calling the services directly from the Java source code. This feature has allowed us to construct the game framework with almost no direct calls to J2EE services inside the Java code.

[1]Can be accessed at http://www.java.sun.com

Another central issue in J2EE is encapsulation. Many details have been hidden, and this enables you to focus on the main thing, i.e. the core functionality in the application. Let us give an example as an argument for this. The communication between a web client and a server is made easy, in the sense that it is quite straight forward to send requests back and forth, using applets and servlets/JSP. You do not need to consider details on network programming, and still you have the power of the application components behind. The concept of encapsulation is central to EJB's, since they are meant to be generally usable components. Even though JNDI is not part of J2EE as such, it is tightly bound to J2EE, and adds the strength of location transparency to EJB technology. The user of a component does not need to worry about where it is located, or which component the user is dealing with, i.e. components may be replicated to serve more users. Being able to replicate components without actually interfering with the code makes it scalable as well.

We have discovered that the J2EE framework is very powerful. The fact that it is built upon J2SE, with the support of programming both web applications and other distributed systems, emphasizes this. However, there is a weakness to the concept one may argue. As long as you are only dealing with Java, there is no problem, but if you want to integrate the technologies into an existing enterprise system, this might not be possible without the use of additional software e.g. middleware such as CORBA. CORBA serves as a good example since J2EE support the CORBA IIOP protocol.

11.2 Distributed Game Framework

Regarding our second goal, we have been concerned with two things. We wanted to use the J2EE technologies, and we wanted to design a system with a high degree of availability, reusability, scalability, usability and transparency in terms of communication.

To make the system available to gamers, we constructed the framework around web application technology. The use of HTTP secures that most people will be able to play, despite the presence of a firewall. We defined an inheritance model "gluing" the framework and game implementations together – thereby avoiding any game specific code inside the framework, and thus making it reusable for other games than just Tic Tac Toe. To support the usage of the scaling possibilities inside J2EE, all components have been constructed to allow replication. Replication is e.g. used when clustering J2EE containers. The usability of the framework was tested by creating Tic Tac Toe, which showed that the framework is sufficient for creating any turn based game in an easy manner.

Communication transparency was partially achieved. The communication protocol is hidden from game developers, but the fact that they are broadcasting information through a network is not transparent. This should arguably be changed, since it goes against the principle of allowing game developers to only think in terms of rooms and tables etc.

11.3 J2ME

Even though we have not dealt with it in the report, there is one interesting technology closely connected to the J2EE framework, which is Java 2 Micro Edition(J2ME). It is another Java framework, with support for mobile/wireless application development. This means that you can program terminals, e.g. mobile phones, which can communicate through HTTP, with J2EE applications. Introducing such a technology to the system would increase availability.

11.4 Other Possible Solutions

In chapter 2 about technology, we discussed how the technologies used in distributed applications have evolved. It is interesting to consider how a framework for turn based games could have been designing using only CGI or server-side scripts.

If a game is run directly on a webpage in pure HTML, then the system would need to make a page reload every time new information should be fetched from the server. But even though we are not considering J2EE right now, we can still use Java applets to get a more dynamic game platform. The first real problem arises in the communication between an applet and a non-Java server. A primary feature that we use in the communication between an applet and the server in our game framework, is the possibility to send and receive whole Java objects through the network. This is obviously not possible, when the server does not recognize Java. All data would therefore need to be serialized by hand into a representation that can be wrapped in the HTTP protocol (e.g. a string representation).

It gets even worse if the game system should rise to fame, and get a lot of playing users. In J2EE, there are techniques integrated to handle scaling (the design of a system is obviously also important), but this is not the case if using e.g. PHP. Of course, such scaling techniques can be directly programmed by the system developer, but this is exactly what we want to avoid when using middleware.

Bibliography

[app] *Thick vs. thin client comparison.* http://www.appdesigngroup.com/papers/Thick-vs-Thin.pdf.

[Bar02] Robert Barnett. *Solving the 'thin client'–'thick client' dilemma*, 2002. http://www.rbainformationdesign.com.au/Forms%20Perspective%2003.pdf.

[BPSMM00] Tim Bray, Jean Paoli, C. M. Sperberg-McQueen, and Eve Maler. *Extensible Markup Language (XML) 1.0*, second edition, 2000. W3C Recommendation 6 October 2000, http://www.w3.org/TR/REC-xml.

[Bro03] Simon Brown. *File access in EJB*, 2003. http://weblogs.java.net/pub/wlg/547.

[FIG+99] R. Fielding, UC Irvine, J. Gettys, Compaq/W3C, J. Mogul, Compaq, H. Frystyk, W3C/MIT, L. Masinter, Xerox, P. Leach, Microsoft, and T. Berners-Lee. *Hypertext Transfer Protocol – HTTP/1.1*, june 1999 edition, 1999. http://www.w3.org/Protocols/rfc2616/rfc2616.html.

[Haa02] Kim Haase. *Java Message Service API Tutorial*, 2002. http://java.sun.com/products/jms/tutorial/.

[hyp03] *Thin Client*, 2003. http://www.hyperdictionary.com/computing/thin+client.

[MMMNS97] Lars Mathiassen, Andreas Munk-Madsen, Peter Axel Nielsen, and Jan Stage. *Objectorienteret analyse og design.* Marko, 1997.

[msn03] *What Is .NET?*, 2003. http://www.microsoft.com/net/basics/.

[OMG03] *OMG Unified Modelling Language Specification*, 1.5 edition, 2003. http://www.omg.org/cgi-bin/doc?formal/03-03-01.

[RAJ02] Ed Roman, Scott Ambler, and Tyler Jewell. *Mastering Enterprise Java Beans.* Wiley Computer Publishing, second edition, 2002.

[RV01] Ed Roman and Chad Vawter. *J2EE vs. Microsoft.NET*, 2001. http://www.theserverside.com/resources/article.jsp?l=J2EE-vs-DOTNET.

[sun02a] *Core J2EE Patterns – Session Facade*, 2002. `http://java.sun.com/blueprints/corej2eepatterns/Patterns/SessionFacade.%html`.

[sun02b] *Java BluePrints – Fast Lane Reader*, 2002. `http://java.sun.com/blueprints/patterns/FastLaneReader-detailed.html`.

[sun03] *JavaServer Pages – Dynamically Generated Web Content*, 2003. `http://java.sun.com/products/jsp/`.