Tamar Christina

Timon Bijlsma

Pieter van Ede

[Multi Useless Dungeon]

September 12

2008

[This document is the accompanying documentation for the project for the Distributed Programming class. Here we outline design choices and architectural decisions made and compromises]

Architectural Documentation

Table of Contents

Introduction

Languages

We are planning to use the programming language Java for our server and primary client. For the secondary client we plan to use C#. As interoperability layer we are planning to use CORBA.

Planning

Work distribution

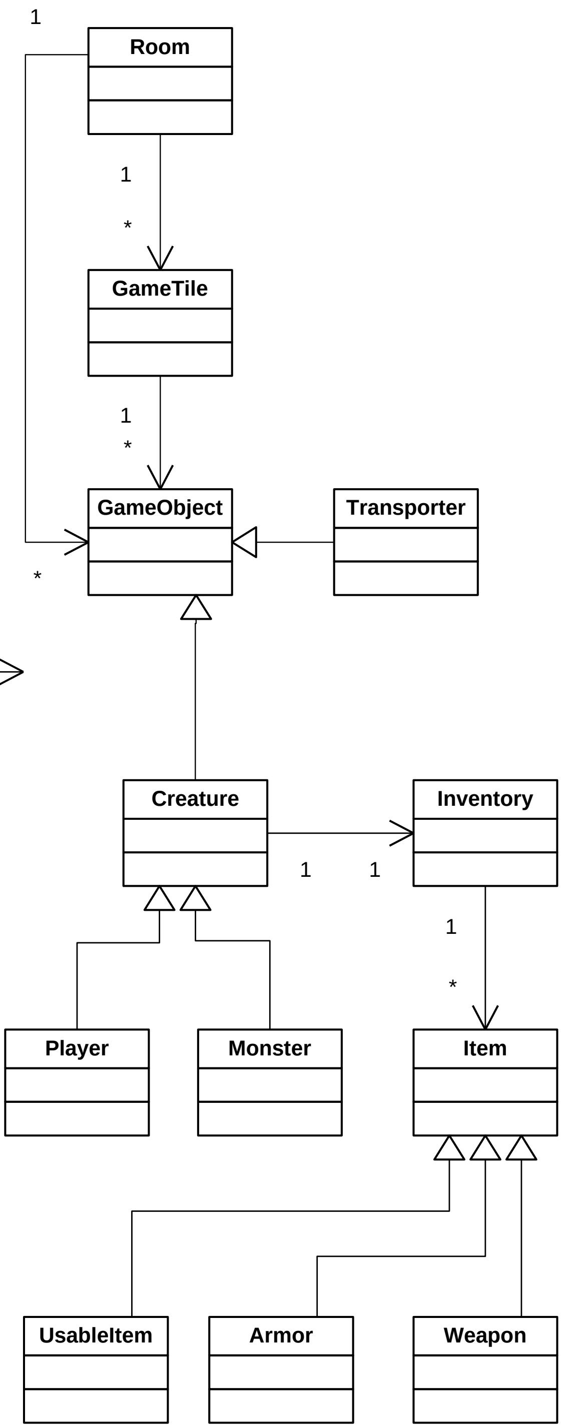
Person	Activity
Pieter	UML Class diagram, document manager, design presentation, Java server implementation, Java client implementation
Tamar	UML Class diagram, IDL, C# client, C# bot, helped with many Java-issues, prepared final presentation
Timon	UML Class diagram, sequence diagram, UML use cases, IDL, GUI, animation, Java server implementation, Java server AI, final presentation

Design

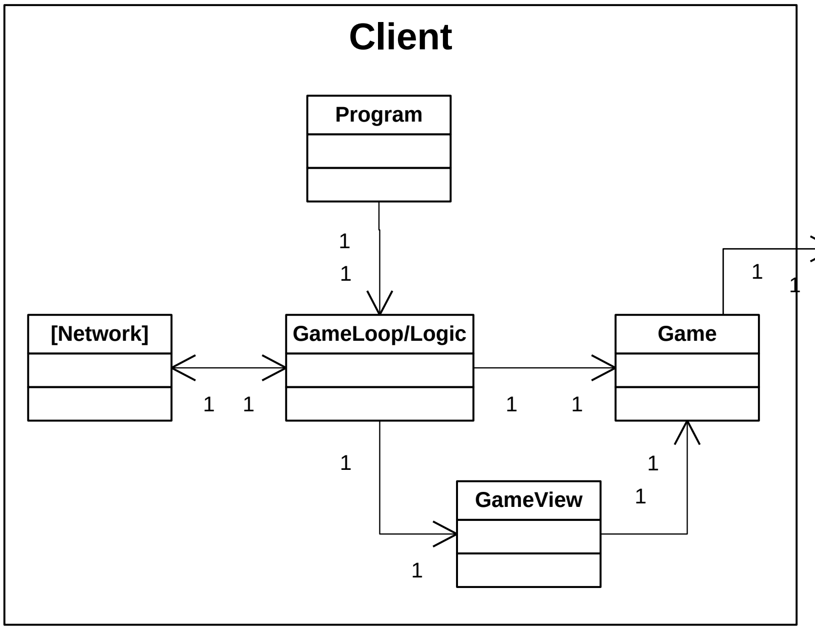
UML

Class diagram

On the right you see the first part of our class diagram. This part models the game itself. It contains a room which consists of multiple GameTiles and each GameTile can contain multiple GameObjects. These could be creatures, which consist of multiple subtypes and for example transporters or just plants and tables. The special thing to note, is that a creature has an inventory of several types of items, which he may drop after a battle.

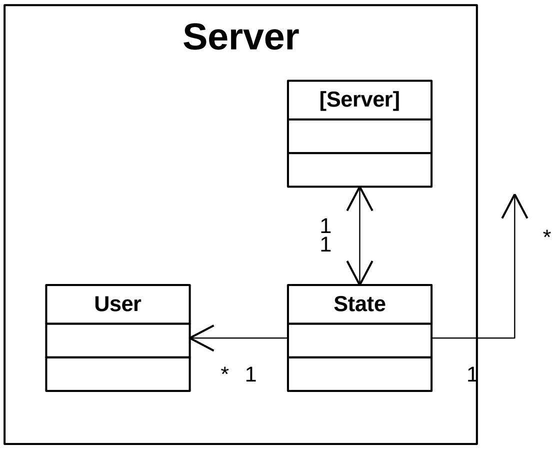


Below you can see the client part of the design document.



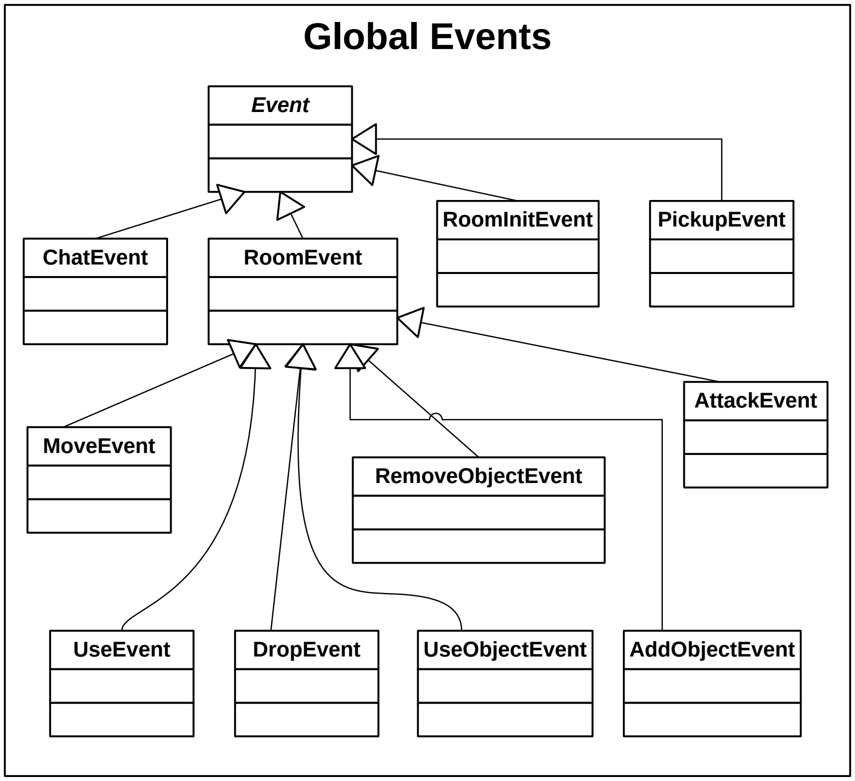
As you can see, we have a base class which starts a typical model-view-controller architecture. We also have a network class, which handles all communication with the server. The idea of our architecture is that the game (the model) has a local copy of the Room with its accompanying GameTiles and GameObjects. We make these local copies to enable real time animation in the client. How we will keep the clients updated is the topic which we will discuss when we arrive at the schema of the events.

For now it is worth noting that the client gets a local copy of a room from the server with all the objects within that room. Then all things that change in the room are being sent to the client via push technology, whereas when the client changes to another room, this new room is acquired via pull technology.



On the right you see the server. It maintains the state of all game rooms, as well as the global game state. This includes the table of all users registered to the game. Also the state has a local copy of all game rooms present in the game, along with their contents. The server object handles all communication with the clients and the authentication. It also handles all the logic, for example it calculates the damage done by an attack and updates the state and also sends this modified state via events to all clients currently in the affected room. It should be noted that the Server object here and the Network object in the client are remote objects.

And then last but not least the events hierarchy. Here you see the events that are being sent between the server and the client. This way, each client can update it's own local game state and enable proper real-time animations.



To wrap this description up, the idea of our architecture is as follows: the client enters a room and receives a local copy of the room and it's contents. Then for every action the client performs in this room, it sends an event to the server. The server will compute the effects on the game room and it's contents and sends the result of this to all other clients. And when the client finally leaves the room and enters another, the whole process starts again. This puts a relative high strain on the server, but for the small amount of expected users this won't be a problem. The advantage of this approach, is that it allows the clients to have realtime animation.

IDL

Because our IDL grew between the design phase and the implementation phase, including it inside this document would make it hard to read. Therefore we direct you to the accompanying file DOS.idl for our interface definitions.

The above describes the remote interfaces the clients and the server use. Basically, the client calls methods on the server to request certain actions like moving and picking up items. These methods can fail for a number of reasons, so they return a boolean indicating whether or not the action succeeds. We could also throw an exception when a request is denied, but we believe that throwing exceptions should be reserved for the actually exceptional cases, not for operations that fail very often.

The server may also push data to the client on its own accord. Event objects are used to transport messages from the server to the clients. We use a subclass for every different type of event that can be sent.

Implementation

As stated in the introduction section, we implemented our game in Java. We created both the server as well as a real-time animated client in Java. We made them communicate via CORBA and we used the standard Java libraries for this. Then we made a second real-time animated client in C#, using the .NET framework. We made this client communicate with the external library IIOP.NET.

Difficulties

One of the major difficulties was having the C# client communicate with the Java server. The cause of this problem was that the used library (which was apparently the only mature CORBA library for C#) was poorly documented, and the documentation that was available was for RMI only. Therefore we had to decompile the library to inspect its internal structure to find out how we should use it.

Interoperability

From the previous section you would conclude that interoperability is not guaranteed, but fortunately that is not true. The only real problem is finding a good library and understand it's quirks, and a client in another programming language can be easily made. The new client of course has to have stubs and skeletons for the functions the server offers and the messages it can receive from the server. To prevent odd problems, we serialized the objects as XML, so the client can start easily by parsing and processing the events it receives.

Security

For this application we are going to authenticate users by means of username/password combinations. We are going to use an authentication key which is returned by the login procedure after successful login attempts.

The returned key will be used to prevent unauthorized changes by users to the game world and/or to other players' data. However since CORBA normally operates over unsecure channels, this setup is vulnerable to man-in-the-middle attacks, where someone intercepts the authentication key. With this key they can disguise as a player and harm the game world or other players. To prevent this kind of abuse, we need to make CORBA use SSL (1). This is a feature we won't implement, but will instantly solve the man-in-the-middle attacks. The drawback is some additional overhead.

Usage

Before we wrap up this discussion of how we designed and implemented everything, you should know how to start the game. You should first start the ORB, which is located in the bin directory of your Java runtime system. The command is:

```
orbd -ORBInitialPort someport -ORBInitialHost somehost
```

Here you specify which port and host the orb is going to use. In the typical test setup, you run everything from the same machine, so the ORBInitialHost parameter can be omitted. Then you start the server and the client, giving it the same parameters as you gave the orb (you are here also allowed to leave out parameters):

```
java -jar mudserver.jar -ORBInitialPort someport -ORBInitialHost somehost
```

```
java -jar mud.jar -ORBInitialPort someport -ORBInitialHost somehost
```

Then you start the C# client with this command:

```
MUD.Client.exe --ORBInitialPort someport --ORBInitialHost somehost
```

Conclusion

We implemented a Multi User Dungeon, using CORBA as the intermediary and Java en C# as participating programming languages. We communicate between clients and server, by letting clients pull the server to let the server know something is about to change. On the other hand the server pushes the results of these requests to all the clients. We also duplicated the game state in the clients, to allow local animations. Additionally, we implemented a bot in C# and some basic AI in the Java server as well. At the end we also described some measures that could be taken to make this game more secure, so players cannot be hijacked and to prevent other nasty stuff from happening.

Works Cited

1. **Iona Technologies.** What does CORBA SSL/TLS Provide? . *Supports Docs.* [Online] Iona Technologies. [Cited: September 23, 2008.] http://www.ionatechnologies.com/support/docs/e2a/asp/5.0/mainframe/ssl/html/Intro2.html.