Initial Report Group 22 End project FourUp 2015

Peter Beers s120

Stephan Braams s1359185

Table of Contents

[Discussion of the Overall Design 3](#_Toc408928729)

[Server Class Diagram and explanation 4](#_Toc408928730)

[Client Class Diagram and explanation 6](#_Toc408928731)

[Model implementation for FourUp Class diagram and explanation 7](#_Toc408928732)

[Discussion per Class 8](#_Toc408928733)

[Server 8](#_Toc408928734)

[ServerController 8](#_Toc408928735)

[GUI 8](#_Toc408928736)

[GameController 8](#_Toc408928737)

[Model 9](#_Toc408928738)

[ConnectionHandler 9](#_Toc408928739)

[ServerSocketListener 9](#_Toc408928740)

[Error 9](#_Toc408928741)

[Client 11](#_Toc408928742)

[Classname 11](#_Toc408928743)

[Model 12](#_Toc408928744)

[Game 12](#_Toc408928745)

[Board 12](#_Toc408928746)

[Field 12](#_Toc408928747)

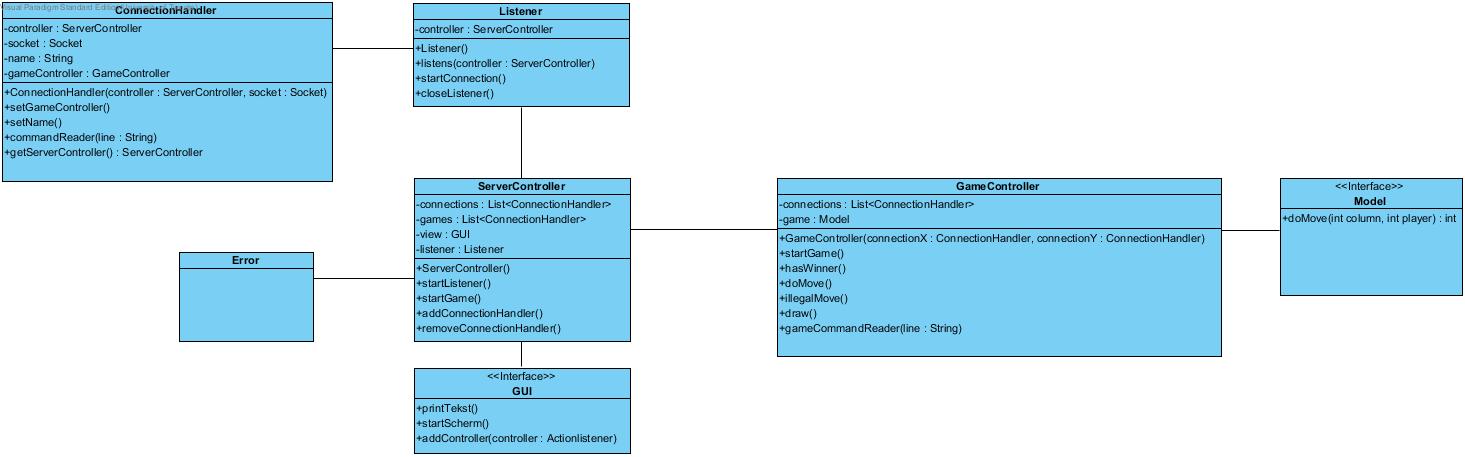
[Security 13](#_Toc408928748)

[Process 14](#_Toc408928749)

# Discussion of the Overall Design

1. Class diagrams with explanation
2. Systematic overview of which parts of the requirements are implemented in which Classes
3. The use of Observer and Model View Controller patterns
4. Formats for data storage and communication protocols

## Server Class Diagram and explanation



The Model Class models the FourUp game and is responsible for assigning turns, checking if moves are correct and notifying the GameController if the game has ended. As the name suggests it fulfills the Model role in the Model-View-Controller pattern. It was decided to use an interface Class for the model for two reasons, on the one hand to add a level of abstraction to the system and create the hypothetical possibility to swap the game type and more important to the developers, ensure that different parts of the system can more easily built independent from each other.

The View is implemented through the GUI Class. We decided to let the ServerController handle the logic for the GUI through action handlers to ensure a degree of separation between Controller and View. As with the Model we use an interface Class to streamline our development process and to add interchangeability to the application.

The main Class in the Server architecture is the ServerController Class, taking the controller role in the MVC pattern. It initializes all other Classes and hence facilitates the creation of games, houses the logic for the GUI, creates the ServerSocketListener for the serversocket and will implement the functionalities of any of the facultative extensions.

For exeption handling we use an independent error class.

The GUI and ServerController Classes are coupled in the observer pattern, with the ServerController being the sole observer and the GUI being the observable. The Model and GameController Classes fit a similar pattern with the Model being the observable and GameController being the observer.

The ConnectionHandler Class plays an important part in the implementation of the AMULET tcp protocol by receiving the commands and sending them to the other relative parts of the system. We decide to leave as much of the logic regarding the commands in the respective Classes instead of building a massive switch which would call methods on the other objects. In practice, this means that the ConnectionHandler sends the commands through to the ServerController and GameController with which it doesn’t do anything itself.

## Client Class Diagram and explanation

## 

## C:\Users\Marnix\git\project\Documentation\Model.jpgModel implementation for FourUp Class diagram and explanation

For the specific rules and dimensions of the FourUp game the model is implemented by a structure of three Classes. The Game Class takes the similar role to controller and the Board Class to Model.

The Board consists of 42 Field classes who keep track of their value in an Observable-Observer pattern. The Board Class handles the checking of the legality of a move and ascertains if there is a winning player and if the game has ended.

The Game Class initializes the Board, assigns the turns, and cleanly ends the game if the game has ended.

# Discussion per Class

1. The role of the Class
2. The responsibilities
3. The other Classes that are used by this Class
4. Any special cases
5. Precautions to fulfill preconditions7

## Server

### ServerController

#### Role

#### Responsibilities

#### Useage by other classes

#### Special cases & Precautions

### GUI

#### Role

#### Responsibilities

#### Useage by other classes

#### Special cases & Precautions

### GameController

#### Role

#### Responsibilities

#### Useage by other classes

#### Special cases & Precautions

### Model

#### Role

#### Responsibilities

#### Useage by other classes

#### Special cases & Precautions

### ConnectionHandler

#### Role

#### Responsibilities

#### Useage by other classes

#### Special cases & Precautions

### ServerSocketListener

#### Role

#### Responsibilities

#### Useage by other classes

#### Special cases & Precautions

### Error

#### Role

#### Responsibilities

#### Useage by other classes

#### Special cases & Precautions

## Client

### Classname

#### Role

#### Responsibilities

#### Useage by other classes

#### Special cases & Precautions

## Model

### Game

#### Role

#### Responsibilities

#### Useage by other classes

#### Special cases & Precautions

### Board

#### Role

#### Responsibilities

#### Useage by other classes

#### Special cases & Precautions

### Field

#### Role

#### Responsibilities

#### Useage by other classes

#### Special cases & Precautions

# Security

1. What risks is our system exposed to?

(SQL)injection?

Man-in-the-middle

Impersonation

Java workarounds?

1. What measures did we take to minimize those

**SQLinjection?**

Sanitazion of input, no direct connection to any database

**Man-in-the-middle**

RSA private key method

**Impersonation**

Registering players

**Java workarounds?**

No protection possible

# Process Description of the Developers

#### Building the foundation first

We developed the class diagrams for the Server Client and Model first in our development process, after putting all the requirements in a planning roster. Subsequently we documented the class diagram explanations and per-class discussions to create a solid base for our application. With the “create code” function of Visual Paradigm and our usage of interfaces we had a good and inter-independent construct to fill in. We relied predominantly on this report for the Javadoc and agreed that while filling in the bodies of our methods to create Javadoc, JML and test first and add logic later because both of us have had bad experiences in the past with trying to come up with an application structure while coding at the same time. For the tests we created 3 separate JUnits Suites, one for the model, one for the client and one for the server. To implement a class we would typically start with the method stubs generated from the class diagrams, add Javadoc and JML. Subsequently we would add Junit test classes and only then write the method bodies. During the development process we still found flaws in our design and the diagram, Javadoc, JML and tests still needed to be edited but the overall development method helped us add a lot of structure to our process. We can confidently say that this method, although a little unrewarding in the setup phase, introduced a lot of peace and clarity to the development process and in our experience enabled us to create a system of higher quality and elegance.

#### Tools used

We used a couple of tools to smooth the development process for us. First of all Github and for integration with eclipse EGit. The advantages of git are widely known and we won’t bore the reader with repeating them here.

For planning and task overview purposes we used the online service Trello, which uses cards to create an interactive S.M.A.R.T task assignment, tracking and planning environment.

For GUI Design we used the google-developed WindowbuilderPro which supports a visual drag-and-drop interface for GUI design.