Software engineering methods assignment 4

by

Irene van der Blij, Mayke Kloppenburg, Samuel Sital,
Kiran Kaur, Takang Kajikaw Etta Tabe

TI2206 Software Engineering Methods

Group 37

Exercise 1 – Your wish is my command, reloaded

We currently have a fully functioning Bejeweled game. The exercise is to extend this and implement a Multiplayer Bejeweled game in which more players can compete against each other.

A multiplayer game played over a network can be implemented using several different approaches and all of these can be categorized into the *authoritative* and *non-authoritative* approaches.

AUTHORITATIVE APPROACH

In the authoritative group, the most common approach is the client-server architecture, where the authoritative server considered as the central entity or authority controls the whole game. Every participating client connected to the server constantly receives data when there is a change in the game(in our case when the tiles have been swapped) and this corresponds to the state of the game.

If a client performs an action, such as making a valid combination of 3 or more tiles of the same colour in our game, that information is sent to the server. The server checks whether the information is correct, then updates its game state. After that it propagates the information to all the clients, so their game states are also updated.

NON-AUTHORITATIVE APPROACH

In the non-authoritative group, there is no central entity and every peer controls his or her own game state. In a peer-to-peer (P2P) approach, a peer sends data to all other peers and receives data from them, assuming that information is reliable and correct therefore, cheating-free. In this case you really have to rely on the reputation of the peers for transparency in the game.

OUR APPROACH

In this document, we present the implementation of a multiplayer game played over the network using an authoritative client-server approach. The game is a bejeweled game in which valid combinations of 3 or more jewels of the same colour gives a player a certain number of points. We have chosen for the client-server approach because of several reasons. One of them being, we need one central server, or rather storage, so all the clients could actually make use of the same board in turns. Also, since during the turn of player 1, the other players should not be able to make any move, so having a central server who can block this is essential. The turn-by-turn system was chosen because it would've been the easiest way to implement this. There are a few more reasons as to why we chose the client-server approach which we shall describe in the conclusion.
An authoritative multiplayer game has a central entity to control the game state. This central entity is the server and it communicates any changes and important actions performed by one player to all the other players, so every player does not need to control his own game state. As a consequence, the different players see just 1 scenario at a time: the move he is performing/making at that particular time.
The player's moves and actions are sent over to the server with the use of sockets (discussed later in this document) connected at different ports on the server for each player, which in turn checks if this move is a valid one and if that is the case, the player's game state and that of all the other players is updated almost instantly and simultaneously using the

serversockets. For the actions or valid moves performed by all the other players, the player gets his game status updated by the server.

In this process of the server and the client communicating with each other, the player sends messages to the server and the server sends messages back to all the players. These messages obviously take time to travel over the network from the server to all the different computers and this therefore leads to a network lag which needs to be taken care of.

THE IMPLEMENTATION

For the specific case of our game, we have one server, which creates the game and a board filled with tiles, and sends this to all the clients through the serversocket. The clients are all the different participants of the game or the different players involved in the game. The server opens a port per player where it can receive all the incoming communication from the clients, meaning all the changes the clients (players) make. The different ports are to make sure the server knows which player has made a move (identification purposes). Instead, we could also make use of a unique ID for each player, but since we're anyway making use of Sockets, making use of a different port for each player is more convenient for us.
When the first participant (player 1) has made a valid combination, this is sent from the player itself through his or her own socket to the server. The server then receives this changed data and then updates the board and game status of every other player by sending the recent board and game status through its serversocket to all the other players, therefore updating their information at the same time.

First of all, we will make use of a queue for all our participants so that there is an order in which each player can make a move and only one player can make a move at a time. So if one client is making a move and has sent some changes to the server, the ports of all the other players are being blocked so that they can't interact with the server at that particular time or make a change to the game status. We currently have the Main and Game classes in our implementation. To implement the Server/Client we need a seperate class called SocketHandler that receives input from Main and sends it to the client. The client connects to the aforementioned (communication) port of the Server using a socket and sends the changed board and tiles. In the view package of our implementation, we have a MouseSocketHandler class and a MouseMotionSocketHandler. When it is a particular player's turn to make a valid or play the game, the MouseMotionSocketHandler and MouseSocketHandler give us which tiles were swapped or clicked on, eventually making a call to the swaptiles method in the *main.java* class, and this change is then sent to the server through the player's socket. The Server receives this changed data and if the move was a valid one, the server adjusts the board and tiles of the Game class and gives the particular player its points based on which specific port was the one to send the change. Since the Game class is now changed through the server, the server informs the next player that it is his turn to make a valid move and the next participant will make use of this changed board and try to make a valid combination.
This goes on from one player to the other depending on the order in which they appear in the queue and when the last player has made a move and gotten his or her points, the first player or client gets the chance to make a move or take an action again. This goes on until there are no more possible combinations on the board. When this is the case, the server informs all participants of the fact that there are no possible combinations left and that the game has ended. After this the server announces the player with the highest score so far as the winner, and this is shown to all the players.

LIBRARIES AND CLASSES

We need to make use of 2 main libraries for the implementation:

1. *Java.net:*

This provides the classes for implementing networking applications. A network application is any application running on one host and provides a communication to another application running on a different host, the application may use an existing application layer protocols and depends on socket programming to communicate to other applications. Through this library, communication between the client (our multiplayers) and the server can be established. For communication purposes, this library makes use of the serversocket and socket classes which will be explained below:

- ServerSocket

Through this class we handle the server side, it has a different port for communication with the clients. A server socket waits for requests to come in over the network. It performs some operation based on that request, and then possibly returns a result to the requestor.
The ServerSocket works like this:
Gets the socket's input and output stream and opens readers and writers on them.
Initiates communication with the client by writing to the socket.
Communicates with the client by reading from and writing to the socket.

- Socket

This is a TCP client API, and will typically be used to connect to a remote host. Through this class we handle the client side of the communication between the client and the server and when a player makes a move or change on the board of the game, messages are sent with the changed data to the communication port of the Server and can receive messages from the Server.

2. Java.IO:

Another important library which is being used for the implementation of this multiplayer functionality is the Java.io package which provides for system input and output through data streams, serialization and the file system. This library makes use of the BufferedReader and PrintWriter classes which are used by both the client and the server to write and read from the socket and serversocket respectively, thereby sending data to and receiving data from the both the client and the server.

- BufferedReader:

Reads text from a character-input stream, buffering characters so as to provide for the efficient reading of characters, arrays, and lines. This is used by both the server and the client or players to be able to read the information sent from the other. The server opens the bufferedReader on the socket of a player when a change is made on the board of the game and reads this change and the player opens the bufferedReader on the serversocket to get the change in the status of each player’s game after one player makes a move.

- PrintWriter:

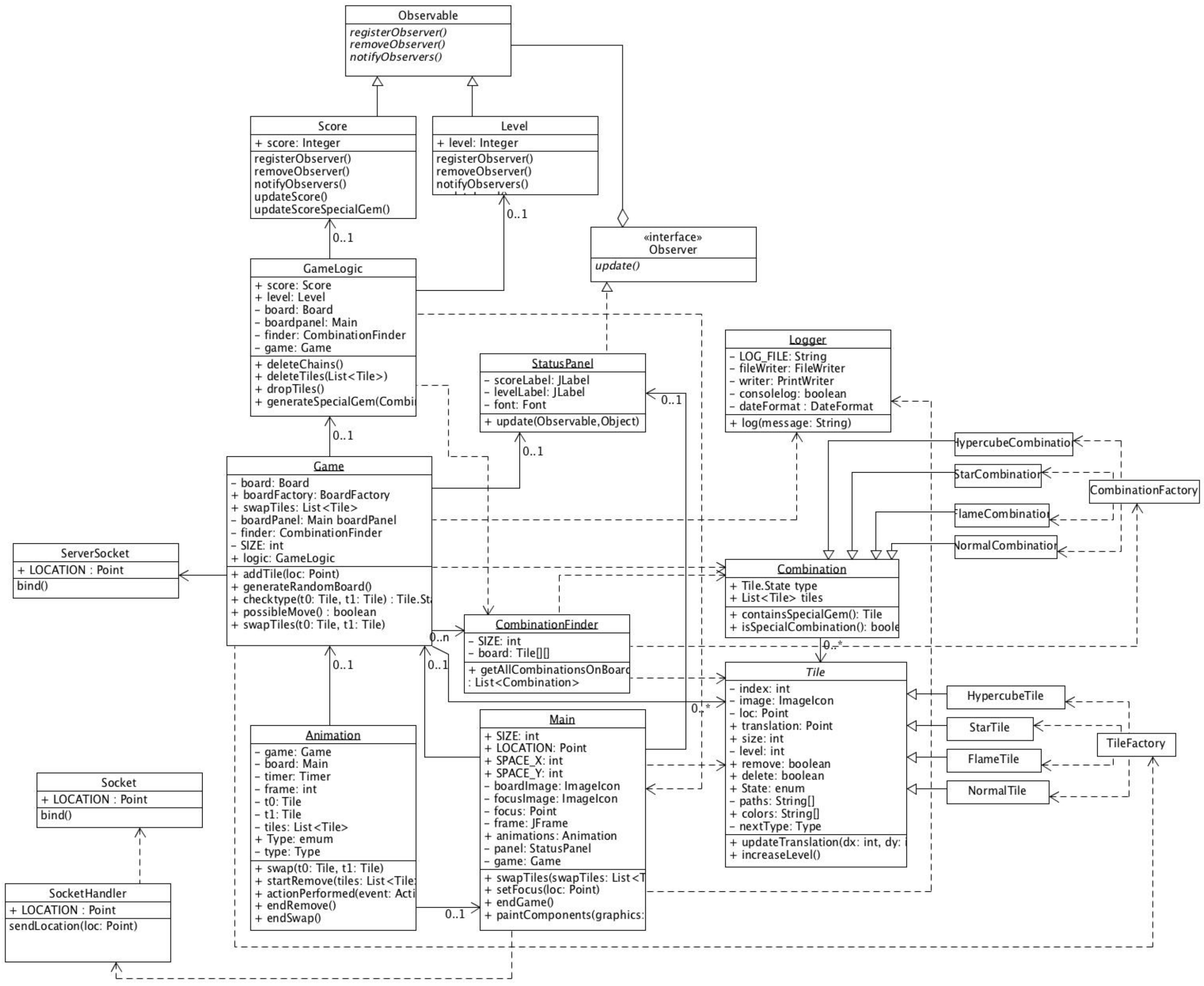
Prints formatted representations of objects to a text-output stream.This is used by clients to send data to the server through the socket and the server uses this to send a change in the game’s status to each of all the multiplayers through the serverscoket.

- InputStreamReader:

An InputStreamReader is a bridge from byte streams to character streams: It reads bytes and decodes them into characters using a specified charset.This inputStreamReader is wrapped around the BufferedReader for top effieciency.

UML

Here we represent the UML corresponding to the extension:



There might be more methods needed for each new class (Socket, SocketHandler and ServerSocket) but without implementing it first we can’t decide precisely which new methods would be required.

CONCLUSION

In this document we presented the authoritative approach in which the server has control over the entire game but this game could also have been implemented using the peer to peer or non-authoritative approach. The main reasons why we decided to choose the authoritative is because of the transparency and we want to make sure that no player can cheat. Cheating is almost impossible in this case because the server has control over the game logic and game status and not the client.

Also another reason why we did not choose the non-authoritative approach is because of the network lag. In this case, clients have to send messages to each other with which move or action they have carried out, but considering the fact that these messages take time to travel to across the network from one computer to the other, this may cause the players to have different boards with different tiles at a particular time. This will therefore cause different participants to have different game states or status when they ought to have the same.

Also in the non-authoritative approach, we may also assume network lag and just the aforementioned point causes or makes the game uninteresting. This network lag comes from the fact that messages have to be transfered from one peer to another and sometimes when these are large, it requires more CPU time to pack, send and unpack all these messages and also reduces the throughput.

This document desribed the implementation of a multiplayer game using an authoritative server approach. All the concepts presented can be expanded to implement different multiplayer games. This doesn’t seem challenging enough? then let the multiplayer game making begin!

REFERENCES

These are the resources we used for the entire research.

- http://www.pepwuper.com/unity3d-multiplayer-game-development-unity-networking-photon-and-ulink-comparison/
- http://www.iue.tuwien.ac.at/phd/fasching/node29.html

For the different libraries and classes we used the oracle documentation.

- http://docs.oracle.com/javase/7/docs/api/java/awt/Point.html
- https://docs.oracle.com/javase/7/docs/api/java/io/BufferedReader.html
- http://docs.oracle.com/javase/7/docs/api/java/io/PrintWriter.html
- http://docs.oracle.com/javase/7/docs/api/java/io/InputStreamReader.html
- http://docs.oracle.com/javase/7/docs/api/java/net/ServerSocket.html
- http://docs.oracle.com/javase/7/docs/api/java/net/Socket.html
- http://docs.oracle.com/javase/7/docs/api/java/net/package-summary.html
- http://docs.oracle.com/javase/7/docs/api/java/io/package-summary.html

Exercise 2 – Software metrics

1. The analysis file can be found here: D:\Users\Irene\Documents\School\SEM\Bejeweled-Group-37\doc\Analysisfile.result

2. The most severe design flaw according to the tool is in our Game class.

   a) The first flaw in our Game class is that this is a God class. We can see this because the class uses many attributes from external classes, the class is large and complex and it is non-cohesive. The other design flaw is that the class is a schizophrenic class. We can see this because the client classes use disjoint fragments of the class in an exclusive way and the class is non-cohesive. These properties lead to the following warnings:
   - There are many methods
   - There are many accessor methods
   - There are extensive calls to external accessors
   - The outgoing intensity is strong
   - The incoming intensity is strong

   The design choices that led to these warnings are that we put too much functionality in the Game class; for example also the functionality of swapping tiles. This should be done in a separate class.
   There is also made use of a lot of calls to methods in other classes, which should be reduced. Some methods that are in the Game class do not necessarily belong there, and should be placed to a more suitable class. This class also has a lot of attributes; this number can also be reduced.

   b) First of all we made a separate class called SwapHandler, which handles the swapping. This already reduced the severity from 9 to 4. The complexity is now average, and also the incoming intensity is now average. We still also had the swaptile attributes in the game class. When we also removed this, and moved one other method to the SwapHandler class, the severity was reduced to 0.
   When we put the method printCombinations() from the game class into the Combination class, and the method createsCombination() from the game class into the swapHandler class, the outgoing intensity was reduced to average, and so we have reduced the coupling.
   We have not fixed the weak common attribute access. The game class, at the moment, has three attributes: board, size and swapHandler. The board and size attributes are used in almost every method. The swapHandler attribute, however, is only used in one method, and thus we thought that removing the swapHandler from this class would fix this issue. We made swapHandler static in order to do so. Unfortunately, this did not fix the common attribute access. Since moving the swapHandler attribute only made the code more complicated, we decided to leave this in the Game class. We already reduced the number of attributes of Game from 7 to 2.
   By making the SwapHandler static we did, however, reduce the incoming intensity, because this changed several method calls to calls to static methods.
   The only warning left is that we have extensive calls to external accessors. We decided to leave this warning, because most of the external calls go to

the board class, but we do not want to move this intelligence to the Board class. Our Board class only contains actions that directly apply to the board, whereas the methods in Game need the information from the Board object.

<u>The second most severe design flaw according to the tool is in our ButtonActionListener class.</u>

a) The following negatively stands out when looking at this class:
   - External data is accessed extensively.
   - The cohesion is weak.

   Our design choice that led to the severity of this class was actually a mistake on our side. In the method where the clicking of the New Game button is handled, we had some code initializing the new game; for example setting the score and level in the new panel and in the game. This meant that we had to access the StatusPanel and other classes multiple times. However, this was not necessary, since the initializing is already done in the constructor of the StatusPanel. The code that caused the warning is thus redundant and can be removed.
   We thought that the design choices that led to the weak cohesion were that we gave Launcher, StartScreen and Statuspanel objects as attributes to this class. But since you only need one startscreen and one launcher, we have to make these static in the launcher class. This means it is no longer necessary to have them as attributes.
   We had a sort of circle: startscreen had the launcher as attribute, statuspanel had a launcher and a startscreen. This was only because it was needed for the ButtonActionListener class.

b) In order to reduce the severity to 0, we simply deleted the redundant code in the handleNewGame() method, as described in section a.
   In order to fix the weak cohesion, we made the StartScreen and Launcher static attributes of the launcher class. In this way, we do not have to give these attributes along with the ButtonActionListener.
   The only warning left for this class is that we access external data extensively. The ButtonActionListener handles the actions for all our button, which involves setting new screens. In order to do this, we need to access the Main and StatusPanel object to repaint them.

<u>The third most sever design flaw according to the tool is the class Main.</u>

a) The following warnings are present:
   - External data is accessed extensively.
   - Extensive use of external accessors calls.
   - Strong outgoing intensity.
   - Weak cohesion.

   In the last two lines of the constructor of the class Main, we set the level and score of the game to the panel. To do this we need a lot of calls to other classes, and we have to use a lot of getters

(`panel.setLevel(game.logic.getLevel().getLevel());`). These lines of code caused the warnings above, since it uses a lot of external data and makes a lot of external accessor calls. Also, this piece of code does not belong in this place. In the back of our heads we probably already knew this, since we also already put this functionality in the method loadGame() of the class SavedGame. Therefore, these redundant lines of code can be deleted.
In the constructor of Main, a lot of initializing is done, which is not really neat to do in this place. Here we also access a lot of external data, so we should change this.
We also have to check whether there are methods in the class Main which do not really belong there. For example, there is still the method swapTiles in Main, which might be placed better in the SwapHandler class. This will also reduce the number of calls to external accessors, since we use these multiple times in this method. This will also increase the cohesion.

b) As stated above, we removed the redundant lines of code, which reduced the severity of this class to 0. Also the use of external accessor calls is now limited, the outgoing intensity is now weak, and the cohesion is now average to tight. While reorganizing other classes, however, these values went back to their old values, and we still had to resolve some flaws to reduce the warnings.
In the constructor of Main we initialized the Score and Level object, and we set the observers for the observer pattern. Instead of putting all this in the constructor, we now made a method in GameLogic, called init(), which does all this initializing. Instead of the several lines of code that did not really belong there, we now have one method call. This reduces the external data access to limited, and it also changed the outgoing intensity to average. Later we also changed the GameLogic class to a static class, and we initialized one GameLogic in the constructor of Game. This changed the external data access for Main to none. This also had a good influence on other classes, for example SavedGame, which had severity 3 now has severity 0.
In order to reduce the extensive use of external accessor calls, we removed the method swapTiles from Main, and put it into the SwapHandler class. This reduced the calls to external accessors to limited.
To increase the cohesion, especially tightening common method calls, we moved the method endGame() from Main to StatusPanel, since this method mainly uses the statuspanel, and StatusPanel and endGame are both responsible for putting text and labels on the screen.
The only warning left in Main is that the common attribute access is weak. This is weak, because several attributes are only used in specific methods. Therefore, we tried splitting the class Main up into two classes. This indeed fixed the cohesion, but the problem was moved to other classes, since these other classes now had to do more calls to external accessors. Overall this solution did not really give a good outcome, and therefore we chose to leave this warning as it is.