Assignment 4

TI2206 Software Engineering Methods

Group 25
Pim Veldhuisen (4153448)
Jan-Gerrit Harms (4163400)
Amritpal Singh Gill (4419820)
Jeroen Vrijenhoef (1307037)

16 oktober 2015

Exercise 1

Requirements

It was decided that game shall be extended to allow multiple players to simultaneously play the same game on different devices. The following user story and subsequent requirements describes this new functionality.
User stories

- As a user I want to compete among other people while playing the game.

These user stories can be translated into a number of requirements:

Requirements

- Functional requirements
  - Must Have
    1. Starting a new game shall provide an option to choose multiplayer game mode.
    2. The game shall allow sending an invitation to second player, when first player selects the multiplayer game mode.
    3. The game shall allow second player to accept or reject an invitation to play multiplayer game.
    4. The game shall convey invitation accept or reject message to first player, and former resulting in starting of the new multiplayer game.
    5. Starting a multiplayer game shall present identical board to first and second player.
    6. The multiplayer game shall only allow a Jewel to be selected by one player at a time.
  - Should Have
    1. The multiplayer game shall display individual scores along with names of first and second player.

2. The multiplayer game shall use red and blue colored cursors to point out the Jewel selected by the first and second player respectively.
3. The multiplayer game shall use same scoring matrix for first and second player.
4. The multiplayer game shall announce the winner along with his/her scores, when no move is possible.

– Could Have

1. The multiplayer game shall disable the hint option.
2. The multiplayer game shall not allow pausing the active game.
3. The multiplayer game shall cancel the selection of Jewel by a player if no move is made for more than 10 seconds.

Non-functional requirements

1. The multiplayer game shall be played on two different devices.
2. The multiplayer game shall use peer to peer network for establishing game connection between first and second player.
3. The multiplayer game shall update board and scores simultaneously, on active devices, with difference in update being smaller than 10 milliseconds.
4. The documentation should be done before the 16th of October, 23:55PM.
5. The game shall be playable on Windows (7 or higher), Mac OS X (10.8 and higher), and Linux (Ubuntu 14.04 and higher).
6. The game shall be implementable in Java 8.
7. The game shall be designed in an agile way, using (parts of) the SCRUM methodology.

FRAMEWORK

The requirements describe a peer-to-peer multiplayer game, that is there is no central server to which the clients connect but the first client is hosting the game and another client can then connect to the first client.

There are two main concepts for communicating between two devices, namely blocking and non-blocking connections.

Non-blocking connections are usually used in large peer-to-peer networks or client server structures with a large amount of concurrent connections. This is the case for most web-applications and massively multiplayer online games. The advantage in these cases is that one thread can handle multiple connections.

Blocking connections are usually simpler to implement but are less responsive in the case of many connections because each connection requires a new thread.

In this case there will be only one connection needed between the two players and as the game is not likely to be expanded to a much larger amount of connections, a blocking IO library will be sufficient for our needs.

Java 8 already has a native library that can be used to connect to devices through the use of sockets, for obvious reasons it is called java.socket. This allows sending messages from one device to another device which is all we need. The actual communication between the players will be quite basic and therefore no sophisticated high-level library is needed.

IMPLEMENTATION IDEA

Implementation of multiplayer game mode will need some modifications to existing classes, and introduction of some new classes.

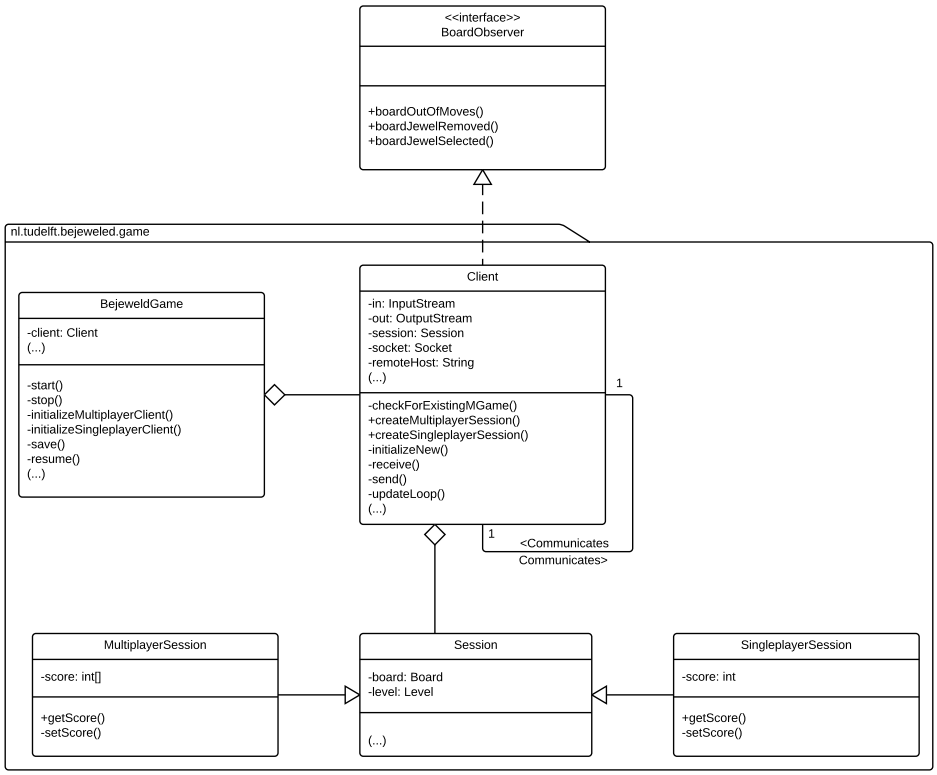
Currently BejeweledGame class only have support for singleplayer game mode. It will require some modifications to incorporate support for multiplayer game mode as well.

An implementation of new class will also be required in order to keep track of synchronization of board and scores within both the devices. The main purpose of this new class would be to tackle any kind of communication that will take place between devices linked through multiplayer game. This class will keep track of the current game sate and any changes in it will be communicated to the second player playing the game. It is required because, in multiplayer game, players will be competing on how quickly one player could figure out the combination of Jewels than the other. Thus it is very important for both competing players to have same board on their devices all the times, else one player will have an advantage over the other in earning scores.

In depth analysis of this implementation idea is discussed in Classes and Sequences section.

CLASSES

Figure 1 shows the implementation of a client class in the existing structure of classes. The following paragraphs will explain the changes that need to be done on the existing classes as well as the new class *Client*.



Figuur 1: Diagram of class structure

Session The session game needs to allow for single and multiplayer games and therefore shall be divided into two subclasses of Session, namely *MultiplayerSession* and *SingleplayerSession*. These will mainly differ in the scoring system, as the multiplayer session keeps track of both players' scores. This will ensure that one session incorporates a full representation of the complete game state.

BejeweledGame The *BejeweledGame* class currently incorporates a session instance, which keeps track of the board, score etc. This is fine in single player mode because *BejeweldGame* is the only class that can influence the session. For the implementation of a multiplayer mode this situation changes. Therefore the *BejeweledGame* class will have a client attribute which in turn has a multiplayer or singleplayer session. To start these session there will be two functions, *initializeMultiplayerClient*() and *initializeSingleplayerClient*().

Client The communication between different devices running the game will be handled by the *Client* class. Just like in a large peer-to-peer network all peers are created equal. In this case both instances of the game will run a client for the communication. The client also keep track of the session, which is a full representation of the game state. The client will be informed of changes to the local players board, that is selections and moves by implementing the *BoardObserver* interface. Selections and moves on the remote board will be communicated through a socket connection between the two devices, the *updateLoop*() constantly checks for data and sends data if needed. If a selection event is received through the connection it will then directly call the function through the attribute structure *session.board*. The mechanism will be further dicussed in the following section.

BoardObserver In order to inform the client of selection changes on the local instance of the game through the *BoardObserverInterface*, it will need to include a function *jewelSelected*.

SEQUENCES

Figure 2 shows the sequence of actions performed to start a multiplayer game.

It shows two devices which each have an actor, a Launcher, a *BejeweledGame* and a Client instance. Consider that they both want to play a multiplayer game of Bejeweld, so they fire up the Launcher and start a Multiplayer game by pressing the button in the GUI. This will inform the *BejeweldGame* instance which then initiates a multiplayer Client, a client class with a *MultiplayerSession*. This happens on both devices in the same way.

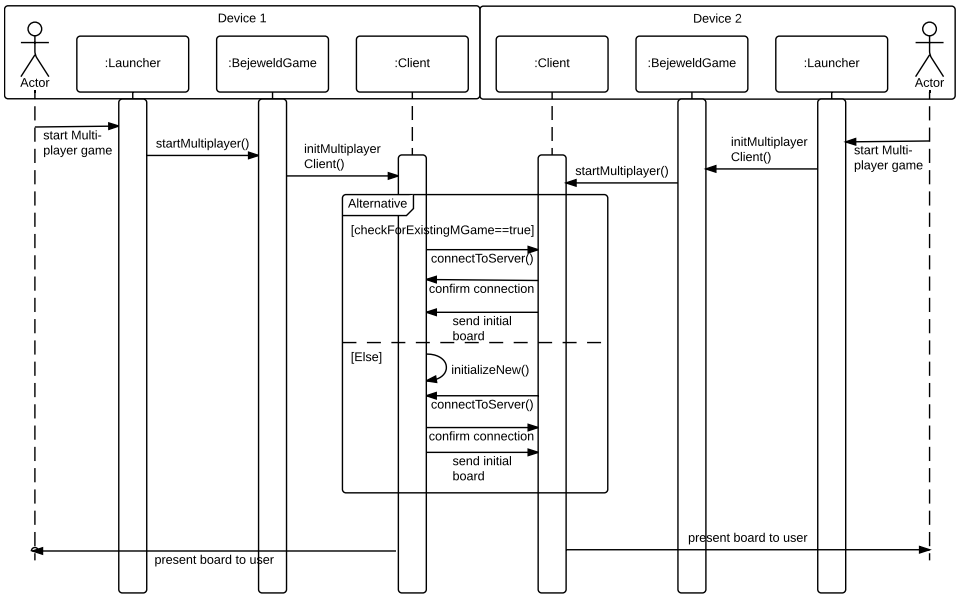
Now the Client which is started first, in this case the left side of the diagram will check for an existing multiplayer game by checking it's local network or some other set of IP adresses for a waiting Client on a fixed Port to be used. If no existing game is found a new one will be created, waiting for a client to connect.

Another client will find the running instance and connect to the Client. The connection will be confirmed by the initiating client, completing the handshake. Then the initiating client will calculate a new board and send it to the receiving client.

Both Clients can then present the user with the game, starting the the main loop.

Figure 3 shows the sequence of actions performed to make a move.

This diagram shows how a move is synchronized between the two devices. One of the players will select a jewel at some point, which is done through the board which is part of the session (in this diagram the board is omitted for reasons of simplicity). The session will then call the *BoardObserver* function *jewelSelected* to inform the Client of the selection. The local boards representation will automatically update this through the existing logic but the remote board also needs to be updated. Therefore the local client will send a message to the remote Client that a selection was made. Because session is an attribute of client and board is an attribute of session, the client



Figuur 2: Sequence diagram of starting of a multiplayer game

can directly call the function *addSelection* of the board class. The same happens for the second jewel selection. Then the Session itself will detect that a move was made, update the score and animate the board.

These diagrams should explain the most important features of a multiplayer system. The description only covers a two player game, although multiplayer could mean many more players participate. The general system would look the same. Instead of informing one client of the moves, each other client would need to be informed. With a larger board it could be possible to make 4 player games of even more.

Furthermore to implement the multiplayer mode there are some difficulties which need to be solved, for example what happens if both players select the same jewel at the same time. This will probably boil done the one with the faster computer/connection getting the advantage. Another thing that can happen is that the connection breaks during an active game. It would then be nice the game is paused and can be resumed later on. When actually implementing the system, probably some more difficulties will become apparent.

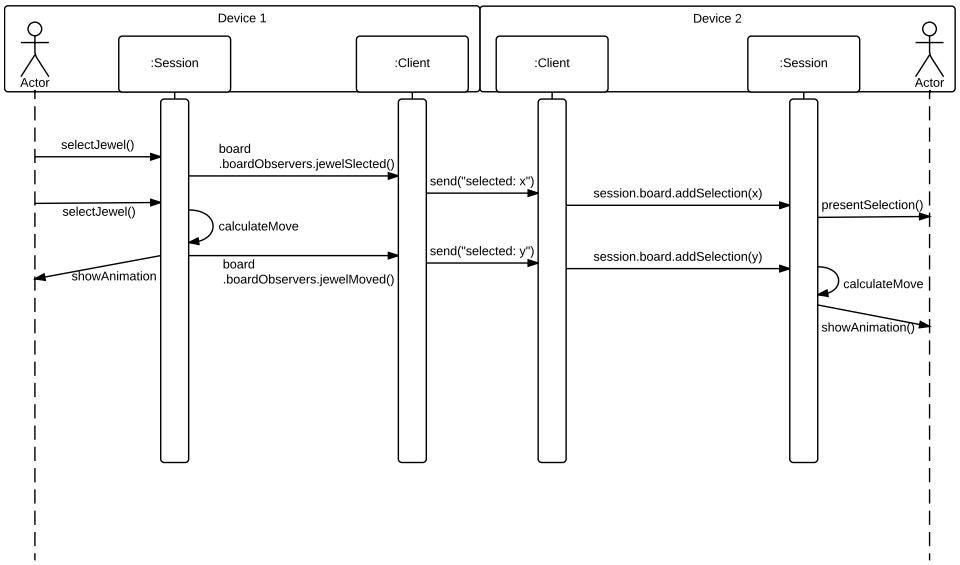
EXERCISE 2

The resulting inCode analysis file is located in the doc/incode snapshots/ directory. The top three design flaws detected are *SchizophrenicClass*, *GodClass* and *Dataclass* and will be discussed in order of severity.

Design Flaw 1: God Class (Board)

1. The board class is excessively large and complex because of two major reasons. The first being that the Board class simply is the most complex class as this class handles all the game logic. The methods that handle combo detection and the hint functionality have a particular high cyclomatic complexity (they for a group of 3 methods that have a cyclomatic complexity of 10 or more).

   The second and perhaps most important reason is that the Board class is also the most reworked class in



Figuur 3: Sequence diagram of a move

the project. Many times the code has been changed (there have been at least three major updates of the game logic) and all group members have worked on specific functionality inside the class. This has lead to duplicate methods and low encapsulation.

2. To fix this design issue the methods that are not used have been removed. The issue with cyclomatic complexity is one we accept because the board is actually fixed in size and therefore an acceptable limit on the time complexity of these methods exists.

Design Flaw 2: Schizophrenic Class (Board)

1. The design errors leading to this design flaw are similar to the ones mentioned in the discussion about the design errors that lead to the *GodClass* design flaw. There were some variables that are used solely by one method and the interface was not very consistent in defining private and public accesor methods.

2. This design flaw actually is not totally fixed. The members of Board that were used solely by one method have been transformed to local variables and public methods that were only used by Board itself have been changed to private methods. The design flaw that remains cencerns several disjoint groups of client classes that use disjoint fragments of the class interface. These clients are mainly BejeweledGame and Session. The other groups of clients are test classes that use specific groups of methods in Board for testing purposes. These test classes are BoardTest, ValidCombosTest, RemoveCombosTest and ValidMovesTest. The BejeweledGame class uses one method excusively called boardClicked(). This method let's the Board know that it has been clicked but no Jewel is selected (resulting in removing the selection). When the BejeweledGame creates the board it first creates a JavaFX Pane to hold all visual nodes in one group. It is at this moment, and only at this moment a handle can be set to detect a mouse click on this board Pane. This is why the method exists and is exclusively used by BejeweledGame.

   The other client class Session uses the methods addObserver(), implodeGrid() and update() in an exclusive fashion. These methods are used by the session to manage the Board class. AddObserver is there for

obvious reasons, so the Session can register itself as an observer. ImplodeGrid() is called when a new level is started, which is detected by Session and update() is part of the overall game loop and is also functionality that should only be used by one client.

Design Flaw 3: Data Class (Sprite)

1. The Sprite class was considered flawed by being a data class, meaning that there was too much usage of its data by other classes. A important user of the internal data was the Board class, which modified all the parameters of the Jewel (which is a subclass of sprite) extensively, setting the position of the Jewels manually.

2. To fix this issue, more abstract modification methods were implemented for the sprite class. This means that the sprite can now be moved to an absolute position or by a relative amount using the implemented functions for this, instead of modifying the individual fields of the Sprite class. This improves the readability of the code, and better separates the Board class from the Jewel class. Additionally, the constructor was modified to include an x and y position on a pixel basis. Until now this was always done right after the instantiation of the object, but it makes more sense to do this in the constructor itself. These changes improved the code quality, which was immediately visible by the fact that the swap function in the Board class could now be implemented in a more sensible way.

Two design flaws were not addressed in the assignment. The first is a message chain in the testMoveJewelNoWin() method. This message chain is part of a test case, and would not normally occur in the code. It is therefore not a problematic issue. The other flaw is the BejeweledGui being a Data class. The exposure of data however makes sense in this case, and the analysis tool probably does not consider its special role as a GUI.