Requirement Document Multiplayer Implementation

Functional requirements

1.1 Must Haves

1. The player must be able to play the game with up to 4 other players.
2. Players will all play on a different board, but with the same grid of gems.
3. Each player will start with 2 lives.
4. After the time limit has ended, the player with the lowest score loses the round.
5. Whenever a player loses the round, that player loses 1 life.
6. Whenever a player has no lives left, that player loses this game and cannot play this game anymore.
7. Whenever 2 or more players still have at least one life at the end of the round, another round is started.
8. If there is only one player left in the game, that player wins the game.

1.2 Should Haves

1. Players should be able to join a game that is already in progress if there are less than 4 people in the game.
    a. This player will then join in the next round.
2. Players should be able to host their own game for others to join.
3. Players should be able to join a matchmaking queue to be put into a game with others.
4. The game should display all of the players' scores on the screen.
5. Players should be able to set a nickname, which is displayed during the game.

1.3 Could Haves

1. Players are able to use an account to play the game.
    a. If a player does not have an account yet, he is able to create a new account.
    b. If a player already has an account, that player is able to log in.
2. "Number of Wins", "Number of Losses" and "Numbers of played games" will be tracked for each account.

    a. These statistics can be found on the player's personal page.
3. There could be a 'global' leaderboard to show highscores across all games played.
4. The player could play ranked games so they are matchmade into games with players of a similar skill level.
    a. The skill level is calculated from the Number of Wins and Numbers of Played Games.
5. The player is able to play the game only with friends.
6. The players are able to chat with one another during the game.
7. The host of the game should be able to remove players from the game.

1.4 Won't Haves

1. Players won't be automatically removed from a game if the player is inactive for a certain amount of time.
2. Players are not able to change their username after the account has been created.
3. Players won't be able to remove their account.

Nonfunctional requirements

Note: Since we don't actually have to implement this code, some of these requirements might be a bit more generic than usual. We will write these requirements as if it would be part of the project as one of the assignments*

1. **A fully functional version of the multiplayer function will be delivered on {insert a date here}.**
2. **The time/level function will be developed using the Scrum software development methodology.**
3. **The multiplayer function will be developed using the Responsibility Driven Design technique.**
4. **The implementation of the multiplayer function shall have at least 75% of meaningful line test coverage. (where meaningful means that the tests actually test the functionalities of the multiplayer function and for example do not just execute the methods involved)**

Researching for libraries

We have researched on what kind of library is useful for client and server networking but before starting this research, we had to think about our transmission protocol. This protocol should handle data communication between multiple clients in a network. In our game Bejeweled we want a reliable data transport and also some kind of ordering of packages since we most likely would want to distinguish which player made the first move, which will take priority over the other. This means the order in which the packages arrive matters. So, using the TCP protocol seems better in our case because UDP does not know about ordering. TCP provides more reliability over UDP since it would request a lost packet if the connection was lost whilst the UDP could get lost somewhere during the delivery.

After this decision we were looking for a way in Java that can support this. We know that all of this is possible by studying sockets and writing your own socket code to do this, however, we don't think that is feasible given the time we have for each sprint to do all of that. We started looking for a simple API to handle the TCP communication between a client and a server. We found Kryonet, Apache MINA and Netty. Apache MINA is open source and it is made for high performance network applications. We don’t really have any experience with the mentioned frameworks, however the lack of documentation and tutorials with Apache MINA make it unsuitable for us because we have to realize the implementation in time. We also looked at another framework called Netty. Netty does have a lot more documentation compared to Apache MINA. Netty also has a user guide that shows some demonstrations about writing a network application with Netty, however in our opinion it lacked the simplicity that Kryonet offers and not many people have actually used Netty to implement a multiplayer game, thus we decided to use Kryonet.

We found Kryonet to be the ideal solution for our game. There is some documentation available on their Github page on how to run a server on a TCP port and how to handle incoming connections. There are also a few small examples available on how a client can connect to a server. Besides, there are tutorials on Youtube how to set this up for a game, so we thought this can help us to implement the multiplayer functionality for our game without diving too much into getting a working connection between a server and a client. This library can be included in the project using Maven. After that, we can quickly create a new server object that starts listening on a TCP port to get started.

Multiplayer Architecture

Now that we have determined that we will use the KryoNet API, we need to determine how we could use this API to actually give our game multiplayer functionality. We have arrived at the current possible implementation by looking at the explanations and examples of the KryoNet API, which can be found in their GitHub repository (github.com/EsotericSoftware/kryonet). Now, we will describe the different classes that we've designed.

BejeweledServer

This class is responsible for the server part of the functionality. Clients can connect to the server. When enough clients are connected, a game can be started. In order to do this, the *BejeweledServer* first needs to create a grid of jewels that it then sends to all clients. This way, the clients each have the same grid, so that the game is fair: everyone has the same possible moves in the beginning. Next up, the game is started by the server. This means that all clients get a message that the game has started and that a timer starts running on the server. The players can now play the game as they normally would. While they are playing, they can see the current scores of the other players. Once the timer on the server has reached 0, the round has finished. One life is removed from the loser (see the requirements) and a new round can start. The *BejeweledServer* class also contains an extension of the KryoNet *Connection* class, namely *BejeweledConnection*. This class is simply a *Connection* that additionally contains the name of the connection, which equals the name of the client that uses this connection. Once the *Game* has finished, the *BejeweledServer* needs to declare the winner to the clients.

BejeweledClient

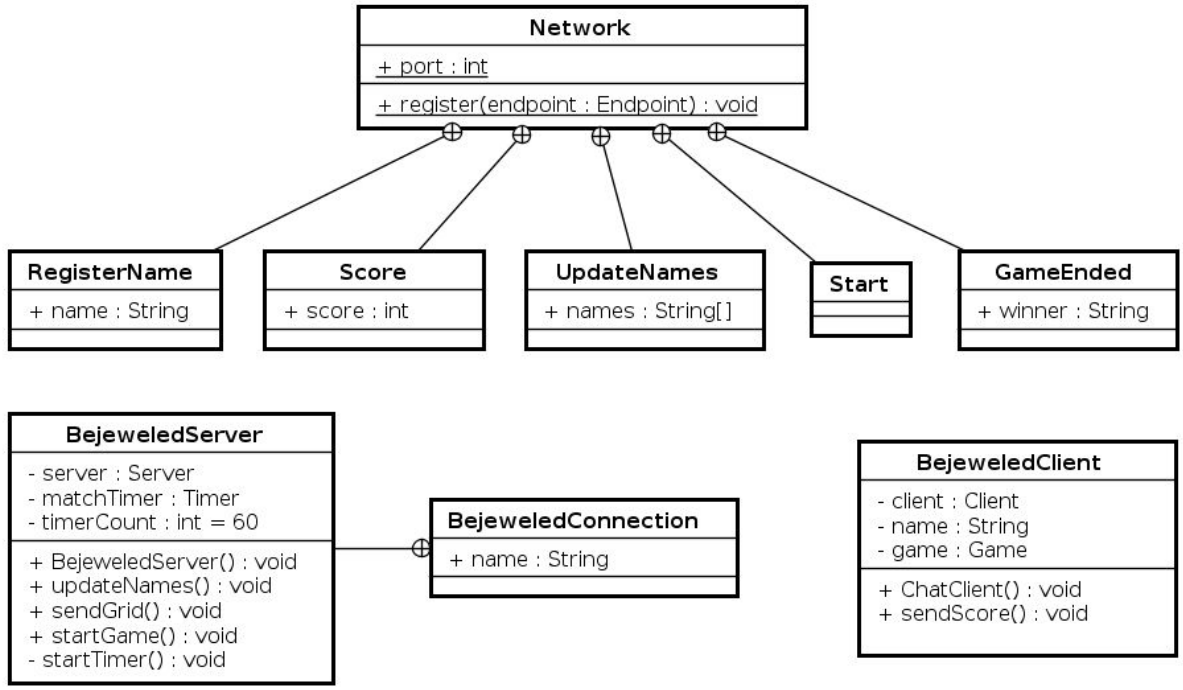
This class is responsible for the communication between a local *Game* and the *BejeweledServer*. It connects to the server and receives and uses the grid send by the server. It waits for the server to announce the start of the game, after which the game can be played normally. When the player makes a move, the updated score is sent to the server by the client. Each client has a name representing the player's name, so that the server and other clients can see who they are playing against.

Network

This class holds all networking information that is shared between the client and the server classes. This includes the port number of the port that is used for communication and the classes of objects that are used in the communication (as nested classes). A *register()* method is also contained in the *Network* class, which registers all objects that are going to be sent over the network. The registration of these objects is demanded by the API.

Overview

When the above classes are put together, the following UML diagram results:



Testing

If we actually implemented this multiplayer feature, we would need to test it to keep our code quality at the same level as before. Before testing the new feature, it is more important to test the existing code. We think the first thing that you would have to do, is to apply regression testing. We have to make sure that our old tests still pass after the new changes and if there are some errors, we will have to solve those first. There is also a chance that some tests become no longer valid since there might be some big changes in several classes, those tests have to be removed or adjusted to the new structure. Secondly, we have to make sure that our builds are also passing on our server integration service Travis which is using our testing tool set before writing new tests. Finally, we could also have a comparison of coverage by using the cobertura coverage report and see for ourselves how this new change has affected our code coverage and why this has happened compared to before this implementation.

The multiplayer feature can be most effectively tested by using a combination of Junit and manual testing. For Junit we could write test cases that check if a connection has been established between a server and a client by making some changes in the client and sending that over to the server, and verify that the server has received the changes. We could also test it the other way around, a server sends changes to a client and the client should have received and applied those changes correctly (we could use asserts for that). Those are the type of tests that we would use Junit for, and when we want to test more complicated changes between server and client, we could use manual testing. We could write scenarios and test cases beforehand and have an actual developer verify if the expected outcome is true or not. For example we could test whether or not a change in the board that is being sent from the server to the client, is applied properly on the board of the client that is being shown on the screen. These kind of tests are more suitable for manual testing instead of Junit testing.

Exercise 2

We used the inCode program to check our code for any flawed classes. The result showed us that we only have one flawed class, namely the BoardTest. (This result can be found in the git repository, in the “inCodeResults” folder) However, we would have thought that the Board class en GUI class would come out as flawed classes. For this exercise, we will look at these 3 classes.

BoardTest Design Flaw

a) In the BoardTest class, we have 6 cases of internal duplication. This means that we have 6 methods in our BoardTest which are very similar. Having multiple cases of internal duplication leads to an increased complexity, because it might be possible to combine code in a separate method and just calling that method.

b) We think it is not useful to remove any of these fragments. Test cases often have a lot of similarities between them, most of the times just changing a few parameters. Therefore, it is only logical that they have a lot of similarities between them. We will discuss 2 possible solutions to these internal duplications, and why we think they should not we implemented.

The internal duplications almost all consider checking the Board for Matches. We have separate cases for both vertical and horizontal Matches, and each of those have a few tests of their own, mostly changing the size of the Matches. For each of the methods, we create an almost exact same grid of Jewels and set it to the board. As a solution we could add a specific grid in the before method. Then we would just need to change a few Jewels in the grid for each test case. However, changing this would make the code less clear. If we would make a standard grid in the before() method, and change some of the Jewels for each test case, it is less clear what is happening in the test. You won't be able to immediately see where the actual Match is, so you have to constantly look at both the test case and the standard grid. We don't want that, because then it will be more difficult to understand the situation, raising the complexity of the class. This is the opposite of what we want by removing the internal duplications, so we won't implement this solution.

For most cases, we make a new list, call checkHorizontalMatches() or checkVerticalMatches() and set an expected Coordinate. We then check the size of the list, the size of the first match and the first Coordinate of the first Match (which is the earlier set expected Coordinate). Since we want to check all of these 3 things, we can't change a lot about these methods individually. Therefore, we consider making a Parameterized test case of this. The Parameters will include the following:

1. If the method checks horizontally, vertically or both for matches.
2. Expected size of the List of Matches.
3. The size of the first Match in List.
4. The Coordinate of the first Match in the List.
5. The grid that the check will cover.

As you can see, this will not make a good Parameterized test case. We need at least 5 parameters, with the 5th one even being an entire grid, consisting of 64 Jewels. This makes a total of 68 parameters per test case. We think this is just ridiculous to put in a Parameterized test case. This will make the parameter method tremendously big, making it unclear what actually happens in it, raising the complexity. Therefore, we will also not use this solution to remove the internal duplications.

So, even after considering these 2 possible fixes, we still don't think it's useful to change the BoardTest class. These solutions will almost every time raise the complexity of the class instead of lower it. Therefore, we will keep this "flawed" class in, because in our opinion, this class is not flawed.

Is the Board class a God Class?

The first flaw that we thought we would see was that the Board class was a god class. This flaw means that the class in question, in this case Board class, had too many responsibilities and was simply too big (not in size per-say, but in its authority over what happens inside the program).

This flaw probably wasn't detected because, while Board class has a lot of lines of code and a lot of responsibility, it may not look that way to an analysis tool. The analysis tool most likely saw the board class and assumed that because the majority of the methods are either private or have to do with getting / setting the internal variables, it assumed that the class was not responsible for as much as it actually is. Another possible reason could be that, because we implemented the observer pattern, and the Board class is communicating with other classes a lot, it assumed that even though it is a big class, it is sharing its responsibilities with the other classes it is communicating with, which is true to a certain extent.

The reason we thought that this was the case was because, from what we can see, the Board class does have a lot of responsibilities, and we thought that it may have too many. For instance, the Board class is responsible for creating the initial grid, keeping track of selected jewels, swapping two jewels, checking if two swapped jewels create a match, removing the board when a match is made, and refilling the board when a match is made. To us, this seemed like a lot of responsibility and that it maybe should be split up into a group of different classes that either work under the Board class, or simply get rid of the Board class all together and create a group of classes that just work together to do what Board currently does.

Is the GUI class a God Class?

The second flaw that we thought would be caught by the inCode analysis tool is, yet again, that we would have a god class. The GUI class in our project is responsible for multiple facets of our project and, with the exception of the scoreboard, is responsible for everything that the player actually sees. Though it is responsible for less than the Board class, and is not as big as the Board class in terms of size, it is still a very sizable class.

Just like with the previous flaw that wasn't detected, there could be multiple reasons why this wasn't detected. The GUI class does incorporate more communication than the Board class, so if the analysis tool sees the abundant communication (not just recieving information but also sending information to other classes) as a factor as to why the GUI class is not considered a god class, then that is understandable. In the eyes of the analysis tool, it could be very possible that it thinks that the GUI class is distributing its responsibilities more than it actually is.

The reason we think the GUI class is a god class, or at least bordering on a god class, is simply because of how much it is responsible for. When looking at the responsibilities that it has, it is clear to see that it does not follow the design principle of one class one responsibility, and it certainly doesn't abide by the general rule of thumb of keeping one's classes small and concise. But it is logical to have the GUI class be responsible for what it is responsible for. If we were to seperate it into more classes the project as a whole would become much more complex (not mentioning that creating the mandatory uml diagrams would become such a heavy workload it would be feared by all in the group).