Assignment 1

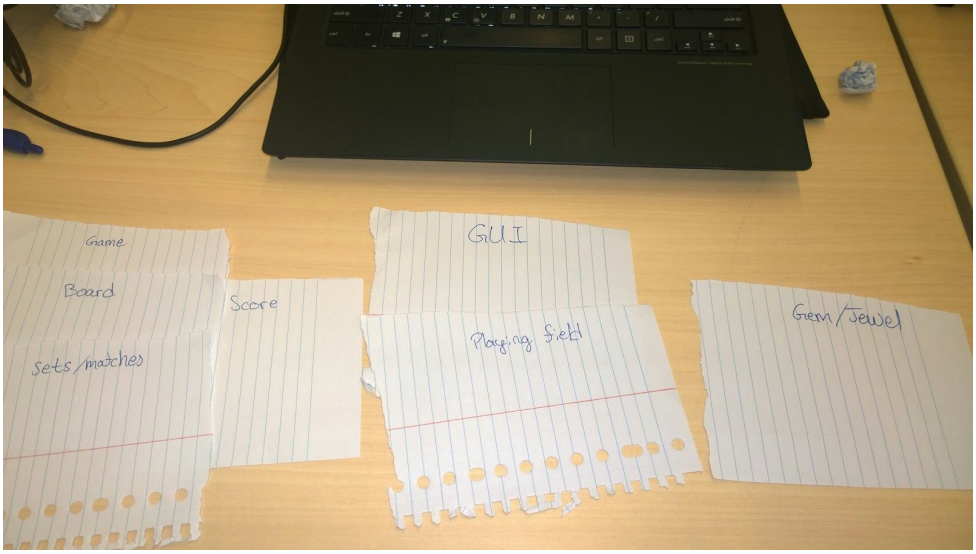
Group 20 Bejeweled

Exercise 1.1

Nouns:	**Classes:**
game board gems sets grid playing field row column moves player message screen score combos blocks matches jewel	Game row column moves Board score playing field blocks Gem / Jewel sets matches combos GUI grid message screen

Selection:

We took Screen out of the GUI because it isn't a needed class. Combos are basically the same as sets / matches so we took it out. Row / Columns are not needed since there is already a board class which will probably use Grid, so Row / Column are just unneeded specifications. Moves does not seem needed either because there is only one move that can happen, a swap, so Gem can handle it.



Differences:
In our implementation we have a scoreboard and a coordinate to handle the score and the position of the gems. In our implementation we also didn't implement the playing field class or the game class. We didn't include these because the playing field, if implemented, would be a weak class, that we think is better to have stay inside the GUI class because of how small it would be. The game class is also not needed because in our implementation it would only act as a manager for board class, which is unnecessary.

Exercise 1.2

The main classes of our implementation are GUI, Board, Jewel.

Board - Is responsible for calculating the layout of the board in the beginning of the game. It adjusts the board for when a match is found (taking out matched jewels and creating the new jewels). It listens to the listeners that process what the player is doing and how it affects what the Board class should do.
The Board class gives information to the GUI as to how to update the board that the player sees. The Board class also gives information about the score to the ScoreBoard. The Board also shares information with its listeners and the jewel class (for what color a jewel is).

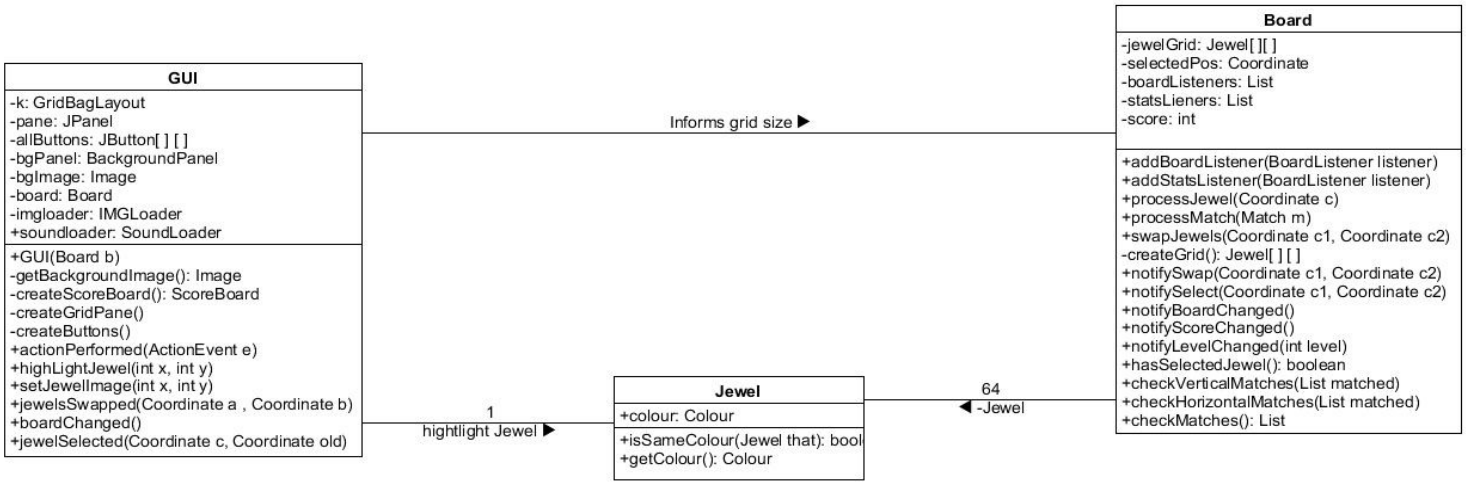
GUI - Is responsible for the GUI of the bejeweled game. Displays the board and the score board. Is responsible for the images the gems use and the sounds in the game. The GUI also handles all the button presses that the player makes.
GUI gives information to the Board class, as to how big the field is. The GUI takes information from other classes to give to the user as visual and audio information.

Jewel - Handles the Jewels. Is responsible for the color of the jewel. Will eventually be responsible for the different jewels that can appear in the game.
The Jewel class gives information about the jewels.

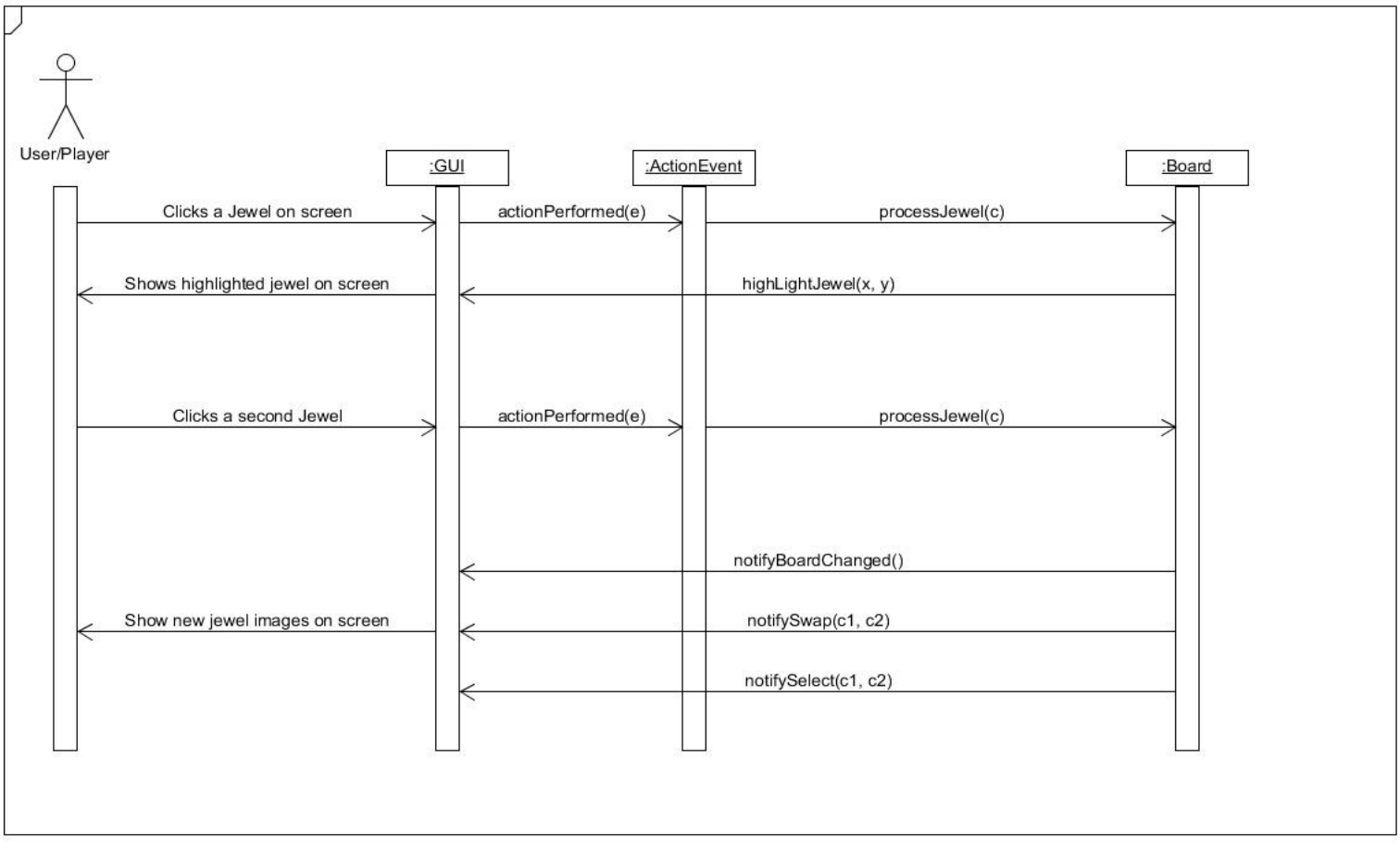
Exercise 1.3

The other classes are less important because they are not essential. They are needed for the game to work correctly, but they are, for the most part, more helpers for the main classes listed above. These non-main classes have not been removed because they all serve a specific purpose. If we were to remove some of these classes we would have to delegate their responsibilities to another class and then that other class would probably end up with too many responsibilities. On a side note, there are one or two classes that don't have many / any responsibilities. We are electing to keep these classes because we have an idea of what we will be using them for in the future, they just haven't been implemented yet.

Exercise 1.4



Exercise 1.5



Exercise 2.1

Aggregation and composition are both used to describe that Object A is part of Object B. The main difference lays within the importance of Object B. If one only cares about Object B because it is part of Object A, then one should use a composition. However, if Object B can be looked at independently, an aggregation should be used. To determine whether a relationship is an aggregation or a composition, the question “Will Object B still have function when Object A is removed?” can be asked. If the answer is yes, then it is an aggregation. Otherwise, a composition.

In our project, this can be asked for the following classes:

- BackgroundPanel class. This class is used in the GUI class, in which it functions as the background. This would be an aggregation, because BackgroundPanel is only important as a part of the GUI. Without the GUI class, BackgroundPanel is useless.

- ScoreBoard class. This class is also used as part of the GUI class, in which it keeps track of and displays the score. This relationship would be a composition, because the ScoreBoard would still have function without the GUI, namely keeping track of the score.

- Jewel class. This class is part of the Board class. This is an aggregation, because without a Board class, the Jewel class would be useless. Every method in other classes act on the grid of Board, not the Jewels themselves.

- Coordinate class. This class is part of the Match class, in which multiple Coordinates are stored to determine the location of the Match. This relation is a composition, because the coordinate still has function besides the Match class. For example, it can be used to describe certain positions on the Board.

- Colour class. This class is part of the Jewel class, in which it contains one out of seven colours to describe different types of Jewels. This relationship is a aggregation, because the Colour class only exists to differentiate Jewels. Without Jewel, Colour would have no function.

- Board class. Board is part of the GUI class, in which the Board is shown on the screen. This relationship is a composition, because Board still has function without the GUI class. The Board can still be modified and Matches can be made to play the game.

Exercise 2.2

In our project, we make no use of any parameterized class. The Parameterized function should be used for classes that can be used with other different classes. For example, the ArrayList can store any Object, while ArrayList<int> stores integers, ArrayList<Board> stores Board Objects and ArrayList<E> stores any Object E. In our implementation, if we would want an instance of Board not to contain Jewel Objects, but any other Object, we could make the Board class parametrized, in which case we wouldn't have to make a different Board class for each Object.

Exercise 2.3

We have three classes that inherit from classes in the Java package.

The first class is the GUI class. This class is a subclass of the JFrame class. We extended from the JFrame class, because we wanted a class that would make a frame, but specializes by automatically adding a board and a scoreboard to it. The type of this hierarchy is "Is-A", because we do not extend any other classes from JFrame.

Next up, we have the BackgroundPanel and the Scoreboard classes. Both of these extend from the JPanel class. We extended from the JPanel, because we wanted to have different panels in our GUI, which are both specialized in a different task. This hierarchy would be a polymorphism, because we have multiple classes inherit from the JPanel.

Currently, we do not have any (useful) classes that inherit from any of our own classes. (We do still have the AdvancedJewel class, but that class is redundant). However, if the power gem and the hypercube would be implemented in a future version of the game, those would probably both inherit from the Jewel class. That's because they would have the same features as the Jewel, only adding the fact that it is a power gem or a hypercube.

We do not think any of these hierarchies should be removed. These hierarchies are made when the subclass has much in common with its superclass, but has some additional features. Most of the time, a hierarchy should be removed if the hierarchy's main function is to reuse code from the superclass.

Exercise 3.1

Logger requirements

Functional requirements

1.1 Must Haves

1. The logger must be able to log messages to the console (eg. the Eclipse IDE console).
2. The logger must be able to log messages to a plain text file.
3. The logger must be able to log messages to all available output forms at the same time (for example, it must be able to log to the console and to a text file simultaneously).

1.2 Should Haves

1. Each individual log (message) should support multiple levels of importance.
2. The logger itself should have an importance level of logging.
3. All logs that have an importance level that is higher than or equal to the importance level of the logger, should be logged.
4. All logs that have an importance level that is lower than the importance level of the logger, should not be logged.

1.3 Could Haves

1. The time at which a message is logged could be included in the (final) log message.
2. The priority of a message that is logged could be included in the (final) log message.

1.4 Won't Haves

1. The logger won't include the name of the class from which the logging method is called.

Nonfunctional requirements

1. A fully functional version of the logger will be delivered on 18-09-2015.
2. The logger will be developed using the Scrum software development methodology. (The development will take one Scrum iteration)
3. The logger will be developed using the Responsibility Driven Design technique.
4. The implementation of the logger shall have at least 75% of meaningful line test coverage (where meaningful means that the tests actually test the functionalities of the logger and for example do not just execute the methods involved).

Exercise 3.2