**Exercise 1**

1.1

For the CRC diagram , we read through our requirements which were listed in point form for more clarity and we picked out the important noun phrases in our requirements and this is what we used as main classes for our CRC cards. The useless noun phrases were discarded.

Per class, we look at the responsibility of this class and those are written down on the left side of the CRC card and on the right side we have the classes that collaborate with each class to help it perform its responsibilities.This is done per class and at the end we link each card consisting of one class to others if they have a relationship together.

Comparing our actual implementation to what we have on our CRC cards , we have a lot of differences:

In our implementation we have an application class which is used for the starting up the GUI while in our CRC cards, we have the GUI being started up in the game class which is not so handy.

Another extra class is the combinationFinder class which we implemented but is not a class in our CRC diagram.This is obviously because we didn't have any noun phrase called combination finder in our requirements but ofcourse we will along the line of the project notice that we need a method or class that can help us find or recognize possible combinations

In our implementation, we also have a class for animations and mouse events while in our CRC cards, those are missing.This is due to the fact that mouse events are built in functions in java and so do not need to be put in separate classes in the CRC cards.

In our responsibility driven design, we have classes for points, levels which are not separate classes in our implementation.This is because points and levels are methods in one of our main classes and due to this, we did not have any need to implement this as separate classes.Points Is gotten per pefect combination and at a certain number of points, we go to a new level and this was just done in a particular method in our implementation.

CRC-Cards

| POINTS | |
|---|---|
| Superclass(es):None<br>Subclass(es): None | |
| Responsibilities:<br><br>Takes player to next level | Collaborators:<br><br>Combination |

| GAME | |
|---|---|
| Superclass(es):None<br>Subclass(es):None | |
| Responsibilities:<br><br>-Shows Player's level<br>-Shows Player's score<br>-Starts a new game | Collaborators:<br><br>-Levels<br>-board<br>-jewels |

| LEVELS | |
|---|---|
| Superclass(es):<br>Subclass(es): | |
| Responsibilities: | Collaborators:<br><br>Points |

| BOARD | |
|---|---|
| Superclass(es):None<br>Subclass(es):None | |
| Responsibilities:<br>-Contains Grids<br>-Contains jewels | Collaborators:<br>Jewels |

| STAR GEM | |
|---|---|
| superclass(es): Jewels<br>subclasses(es): None | |
| Responsibilities | |

| GRID | |
|---|---|
| Superclass(es):None<br>Subclass(es):None | |
| Responsibility:<br>Contains jewels. | Collaborators:<br>Jewels |

| HYPERCUBE | |
|---|---|
| superclass(es): Jewels<br>subclasses(es): None | |
| Responsibilities | |

| JEWELS | |
|---|---|
| Superclass(es):None<br>Subclass(es):None | |
| Responsibilities:<br>Points earned for 3 or more jewels of thesame colour | Collaborators: |

| FLAME GEM | |
|---|---|
| superclass(es): Jewels<br>subclasses(es): None | |
| Responsibilities | |

| COMBINATION | |
|---|---|
| Superclass(es):None<br>Subclass(es):None | |
| Responsibilities:<br>Points achieved per valid combination of jewels. | Collaborators:<br>-Jewels |

1.2

Looking at our implementation, we have a couple of main classes in terms of responsibility and collaboration.

The first of all of them is the *Game*.java class. This class is a main class because it collaborates with so many other classes such as *Combination*, *CombinationFinder* and *Tile*. It contains a lot of important responsibilities such as making a new game, creating a random board filled with jewels of different colours and a lot more. As a result of this, Game.java is considered as a main class. This class also makes use of combination to be able to print all combinations on the screen and to be able to swap jewels if it can make a move. It contains the important aspect of the game such as deleting jewels when they are a valid combination and letting other jewels fill up the space of the deleted jewels. These are the fundamentals of the game, and because this class contains these fundamentals, it is considered as a main class.

Another main class is the *Main*.java class.

This class just like Game.java collaborates with a lot of other classes and those other classes need the main.java class in order to function properly or perform their responsibilities.This class collaborates with the *Game*, *StatusPanel*, *Logger* and *Animation* class which are also important classes.This class contains important

methods such as the making of the board on which the game will be played, it adds mouse listeners to the game making it possible for mouse events to be recognized and executed, it also adds a logger on to the game making it possible for the start of a new game.It also makes it possible for a player to be able to end the game in which he is currently on. These are all basic functionalities and because of these, this class is considered a main class.

The combinationFinder.java class is a class which is considered to have some certain degree of responsibility but it collaborates with board and combination. This class is considered as an important class because this is where we find all the possible valid combinations, be it a simple combination of 3 jewels of the same color or the more complicated combinations consisting of The T and L shapes.This is also considered an important characteristic of our game because it is a must have or a functional requirement. Due to this functionality of finding all the appropriate combinations in the game, this class is considered an important class in our implementation.

The StatusPanel.java class is used to keep record of a player's score and level and to update the score and level if any changes occur or when he makes a possible move or has a combination.This class just updates the player's scores and shows it on the screen and as such, it is considered to be an important class because it has a responsibility of showing the player his score and collaborates with the other classes in which the players scores and levels are calculated.Due to this we consider it as an important class.

The tile.java class is also an important class in our implementation.This is where our tiles are made and colors are given to the different tiles on our board.Also,this class is considered important, this is because it contains our hash function which is an important aspect of our implementation, without which the newly added tiles will be put in the wrong position on the board after a valid combination has been made.Due to this functionality, the tile.java is considered an important class.

1.3

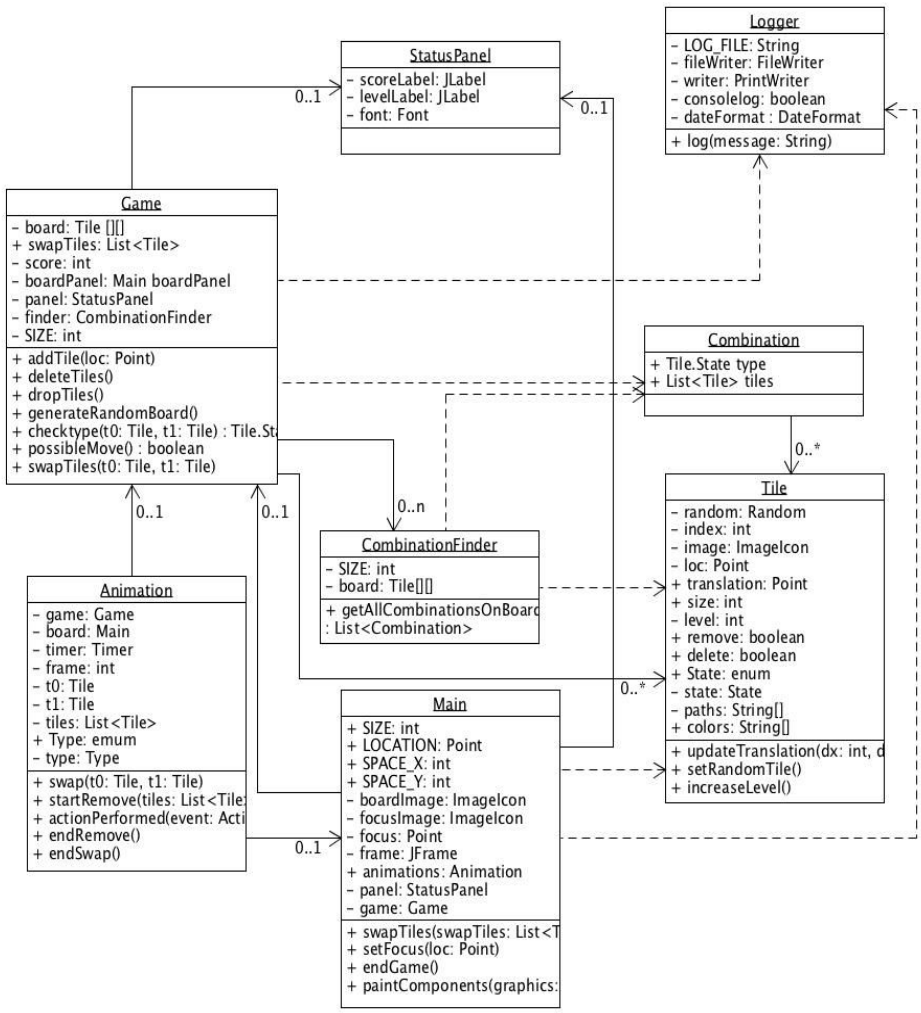
Looking at our implementation, we observe that some classes have less responsibilities than our main classes. Some of these classes are:

The other classes which we implemented are the event or mouse listener classes which are used for the execution of every mouse event such as a click or a drag. All these actions lead to subsequent changes and this is needed for the game to be able to be played appropriately. For example, if a player tries to make a move by switching 2 tiles, this is a mouse click and a mouse drag event which requires that the 2 tiles be switched. However, the aforementioned classes do not have the same degree of responsibilities as our main classes.

The Application.java class is also less important than our main classes but is still essential for the game as this is where we create our GUI(graphic user interface). It's also where the animations for our tiles are being made. Other important functionalities of it include the animation for the swapping or removing of tiles which is needed in order to notice that two tiles are being swapped or removed from the board. These functionalities are the reason we have to keep this class because without them, these actions could occur without the player being able to observe any changes (how the tiles or jewels are being refilled or swapped).

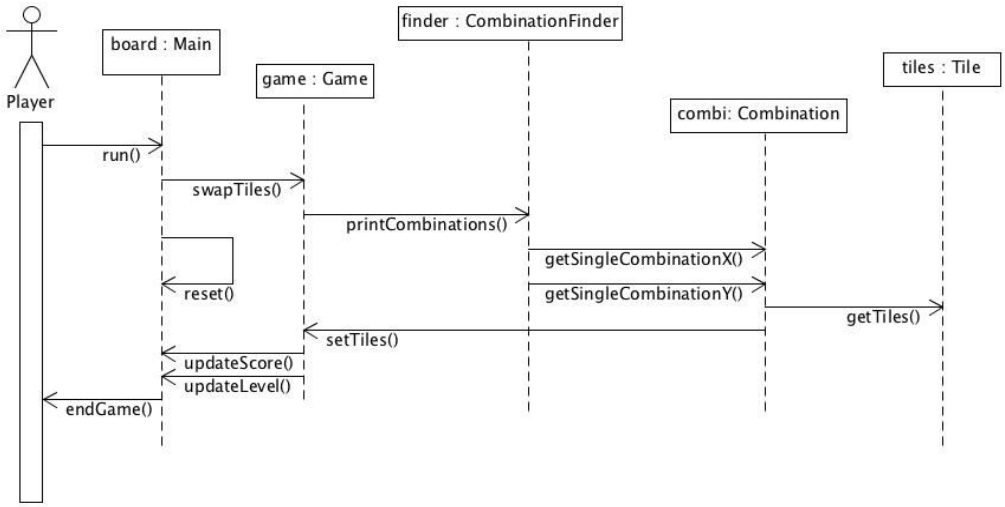
The flameTile.java, hypercube.java, starFile.Java are not important classes because they have no collaborators and also almost no responsibilities. But these classes will be needed later on for further implementation of our game and because of this reason, we will keep this class and not delete or merge them.

## 1.4



1.5

This sequence diagram describes the main elements of our game.



**Exercise 2**

2.1

The difference between aggregation and composition is that aggregation is a relationship where the child class can exist without the parent class, whereas composition describes a relationship where the child class cannot exist without the parent class.

In our project we have the classes Combination and Tile. A combination is made up of several tiles, and so these classes have a relation of the type aggregation, because the Tile class can exist independent from the Combination.

We also have the classes Board and Tile. A board consists of tiles, so these classes have a relation of the type aggregation, because the Tile class can exist independent from the Board class. (The Board class is not yet in use, but we plan to split the Main class into Main and Board).

The classes Application and StatusPanel also have a relationship. This is a composition, because the StatusPanel class cannot exist without the Application. This is due to the fact that in the StatusPanel we update the scores in the JFrame of the application. The JFrame is made in the application, so we need the application for the StatusPanel.

The classes Main and MouseListener, and Main and MouseMotionListener both have a composition relationship for the same reason. In the Main the jewels are set on the board, and the MouseListener and MouseMotionListener need this to be able to click on the jewels; therefore the listeners cannot exist independent from the Main class.

Lastly, the class animation has a composition relationship with the classes Game and Board. In the Animation class, the jewel images on the board are moved, and for this a board is needed to have the actual tiles with jewels, and a game is needed to be able to switch the jewels.

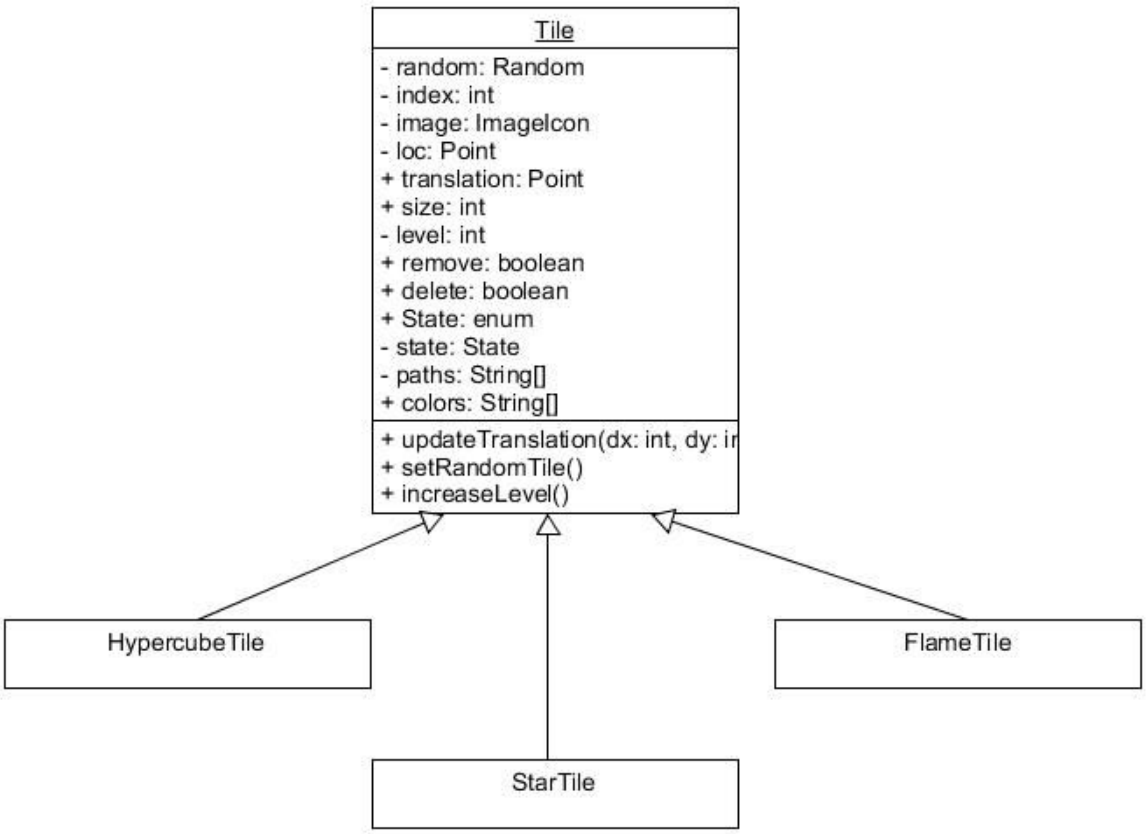
2.2

We do not have any parameterized classes in our source code. Therefore we will describe here when and why you should use parameterized classes in an UML diagram. A parameterized class is a class which has a parameter of which the class is not yet defined. This means that the parameterized class can be used with parameters from many different classes. Therefore parameterized classes can be very useful when you want to use a certain class (for example a list) multiple times, but with different parameters.  In UML such a class is shown by a dashed rectangle

in the right top corner of the class symbol. In the dashed rectangle the formal parameters of the parameterized class are shown.

2.3

We have a hierarchy with the classes Tile, HypercubeTile, StarTile and FlameTile, where Tile is the superclass and the other classes are the subclasses of Tile. We made this hierarchy, because all of these classes have the same basic characteristics and method, but there are several methods where the outcome is different for the subclasses. This hierarchy is useful because we can put all overlapping and basic methods in the Tile class, and override methods in the subclasses where the outcome is different for a specific type of tile. This will reduce the number of switch statements in the Tile class, which makes the class more organized. The type of the relation is an “Is-a” relation, since HypercubeTile, StarTile and FlameTile are all tiles.



**Exercise 3**

3.1

The logger has the following requirements:

- The log file is a .txt file;
- Ability to write to log from every class without instantiating an object(e.g. static log method);
- The logger is not buffered but writes immediately to the file.
- Write error messages to log;
- The log file displays date and time of log message, e.g. "15-09-15 12:29:32.573 - Swap tiles: java.awt.Point[x=0,y=4], java.awt.Point[x=0,y=5]".