Exercise 1

Game	
Superclasses:	
Subclasses:	
Hold players Hold levels Keep track of the score Keep track of lives of the player	Level Player Score

Player	
Superclasses:	
Subclasses:	
Move Shoot rope Picked up powerups Die	Rope Powerups

Level	
Superclasses:	
Subclasses:	
Restriction borders Amount of bubbles Starting coordinates of initial bubbles Rope coordinates Player coordinates Powerup coordinates Handle all collisions	Bubble Player Rope Powerup

Bubble (abstract class)	
Superclasses:	
Subclasses: Bubblex8, Bubblex16, Bubblex32, Bubblex64, Bubblex128	
Movement Destruction (popping) Size of the bubble Colour of the bubble How much points it is worth	Level Score

Score	
Superclasses:	
Subclasses:	
Handles the score of the game	Level Bubble

Rope	
Superclasses:	
Subclasses: IceRope	
Moveup and disappears when it hits the roof or a bubble Coordinates Colour	Level Player

IceRope	
Superclasses: Rope	
Subclasses:	
Overrides Moveup and disappears only if it hits a bubble Overrides colour	Level Player

Powerup	
Superclasses:	
Subclasses: SpeedPowerup, IcePowerup, LifePowerup, DoubleRopePowerup, InvinciblePowerup, SlowMoPowerup	
Duration Coordinates of the powerup	Level Player

Settings	
Superclasses:	
Subclasses:	
Keep track of all settings of the game Slowmotion factor for the SlowMoPowerup	Game Level Player Rope Bubble Powerup

Bubblex8	
Superclasses: Bubble	
Subclasses:	
Inherits everything from parent	Level Score

Bubblex16	
Superclasses: Bubble	
Subclasses:	
Inherits everything from parent	Level Score

Bubblex32	
Superclasses: Bubble	
Subclasses:	
Inherits everything from parent	Level Score

Bubblex64	
Superclasses: Bubble	
Subclasses:	
Inherits everything from parent	Level Score

Bubblex128	
Superclasses: Bubble	
Subclasses:	
Inherits everything from parent	Level Score

SpeedPowerup	
Superclasses: Powerup	
Subclasses:	
Inherits from parent Player will move faster	Level Player

LifePowerup	
Superclasses: Powerup	
Subclasses:	
Inherits from parent The player will get an extra life	Level Game

IcePowerup	
Superclasses: Powerup	
Subclasses:	
Inherits from parent The rope won’t disappear until it intersects with a bubble	Level Rope

DoubleRopePowerup	
Superclasses: Powerup	
Subclasses:	
Inherits from parent Player will be able to shoot 2 ropes at a time	Level Player Rope

SlowMoPowerup	
Superclasses: Powerup	
Subclasses:	
Inherits from parent Bubbles will move slower	Level Bubble

InvinciblePowerup	
Superclasses: Powerup	
Subclasses:	
Inherits from parent Player can't lose lives	Level Player

1.1
We started by reading our updated requirements. After reading we started with defining the class which we thought was the biggest. From that point we went by all the points in our requirements and build each class on that basis. It was not easy to create the classes unbiased. When we had the standard classes we started thinking of any possible specializations of certain classes. We came up with a couple of different sub-classes for bigger classes.

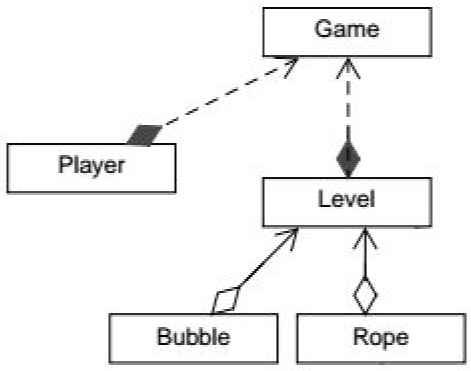
When we compared our new classes, responsibilities and collaborations we noticed that we had much more classes than we had before. Our first overview of classes etc. also lacked many responsibilities and collaborations between classes. The way we build it now has a much better overview of how the overall architecture of the system should look like. What we had before was a fairly simple UML diagram which was not as in-depth as it is now. One of the biggest differences we encountered was that the bubble class ended up to be an abstract class instead of the big class it currently is.

A few examples: We only have 1 big bubble class, but it would be better to split it up into smaller classes. The same applies to the powerup class.

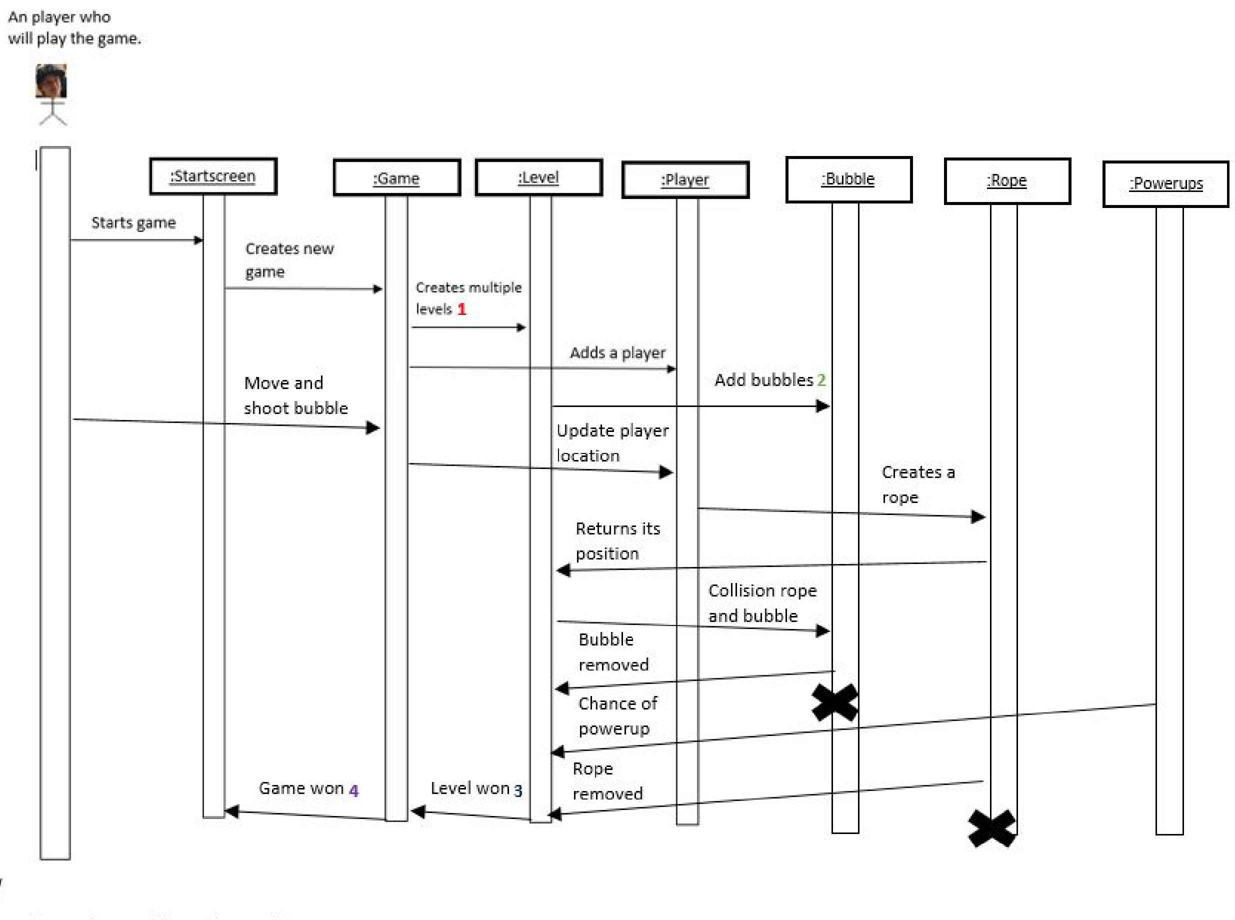
1.2
Our main classes are the classes that the game as a whole depends on. So: without bubbles the game isn't bubble trouble, without the player the game can't be played, without the levels the game can't be played, and so on…
The main classes are: Player, Game, Level, Bubble, Rope. These are the crucial building stones for having a functional and working game. Those classes together share the most responsibilities and collaborations with each other. The other remaining classes do have some responsibilities and collaborations, but those aren't as crucial as those of the just stated main classes.
Also, we didn't implement any subclasses, but they are not so important to let the game work. We implemented all the responsibilities, which are necessary to let the game function correctly, from the CRC cards.

1.3
The other classes are less important, since they don’t form a crucial part for the functional game (the core). However, none of the less important classes can nor should be merged. These classes themselves do carry responsibilities that can’t be taken over by other classes, at least not in an easy manner. We kept our base pretty small, we made no subclasses of classes. Because of our small scale class diagram we only have the most crucial classes for our game to function sufficiently. This results in some real big classes that have to be split up into subclasses. We did create those subclasses that are needed in exercise 1.1. The updated newer structure from exercise 1.1 will be implemented in our code. Those new classes will make things a lot more manageable. The child classes will have new and more specific methods (ex. the movement of a bubble) for the object that it is. This also fixes our sometimes deep cyclomatic complexity. So the changes won’t consist out of merging or removing but instead will consist out of splitting and adding. All of that to improve our overall quality and to strive for a better implemented Responsibility Driven Design.

1.4



1.5



1. {number of levels > 0}
2. {number of bubbles >0}
3. {all bubbles are popped}
4. {all levels are won}

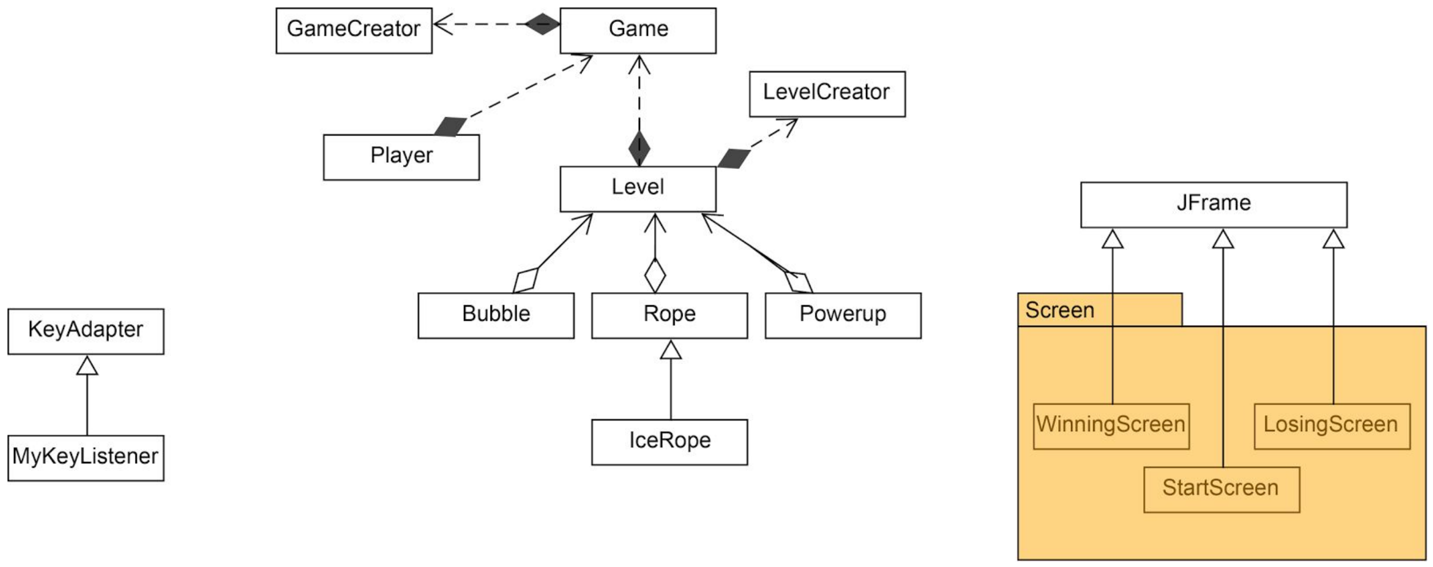
Exercise 2

2.1 In a (java) project there exists a relationship between two objects. This relationship is also known as association. There are different kinds of association, namely composition and aggregation. However, these two types of association have a different meaning. In the case of aggregation the two different objects can exists without each other. In the case of composition, one class cannot exist without the other class.

In our project we used composition in several places. Level and levelCreator, as well as Game and GameCreator use composition because a game and level simply cannot exist without their Creator. Level and Game also use composition because if there is no Game, you also cannot have a level. The classes Bubble, Rope and Powerup have the relationship aggregation with Level because they can exist without each other but they are associated (parent-child).

2.2 We did not use any parameterized classes. A generic type is a reference type that has one or more type parameters. These type parameters are later replaced by type arguments when the generic type is instantiated. This is useful when you want to use certain methods or algorithms that work on collections of different types. But in our game there are no different objects with the same kind of behaviour, but if this were the case here would the use of parameterized classes really benefit the code.

2.3
Game is the uppermost class in the class diagram, together with the GameCreator class. There is dependency between Game and Gamecreator because Game requires GameCreator. This is the same case with Level and LevelCreator. Level cannot exist if there's no game and it also 'requires' Game so this relationship is composition and dependency as well. The Player class cannot exist without a Game so their relationship is the same as the ones above. The relationship between Bubble and Level, as well as the relationship between Powerup and Level, is aggregation because a bubble/rope is a part of level but can exist without it. This is the same for Rope and Level. Then we continue to IceRope that inherits from Rope, with type Is-A. Other classes that inherit from other classes with type polymorphism are the classes WinningScreen, StartScreen and LosingScreen that inherit from JFrame. Another class that inherits from another class is MyKeyListener that inherits from KeyAdapter with type Is-A. All of the hierarchies are used in our game so none should be removed.



3.1

Log Requirements

Globally the log will works as following: Every event we want to log, calls a method from the log class that creates a log object. A logObject exists of a category, severity and content. All those objects are stored in a simple list and is written to a text file for backup when the game crashes. The log is visible in an extra frame (see design below). This frame can filter on message and/or severity type. There will also be extra logging settings that can enable or disable certain features of the logger to improve performance and readability.

levels in logging

- message category (list)
  - Player input
  - Player
  - Bubble movement
  - Bubble
  - Collisions
  - Rope
  - Powerup
  - Game
  - Level

- severity type (list)
  - error
  - exception

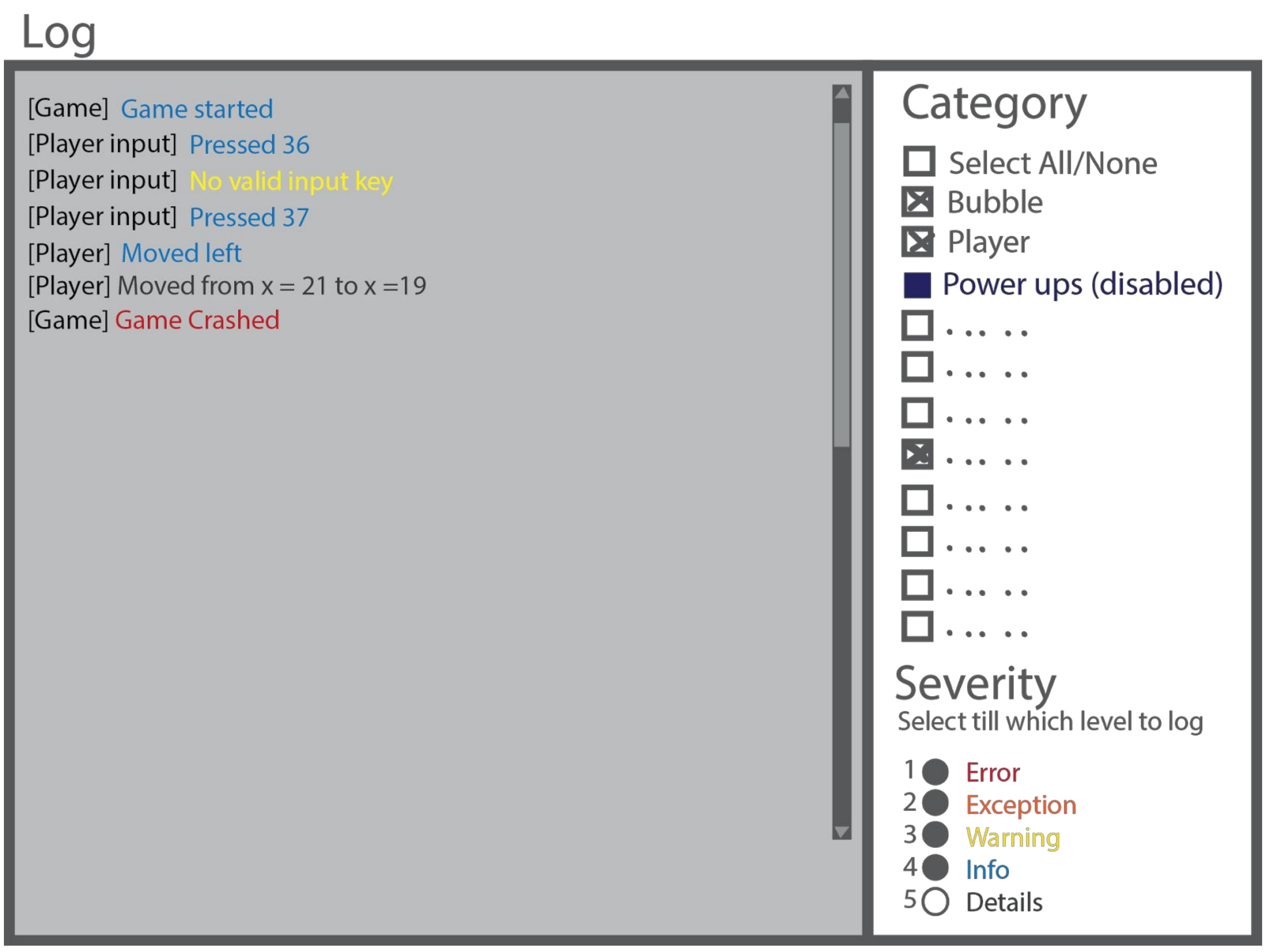
- warning
- info
- details

Logging Settings

- Toggle enable or disable Log
- Select or disable category for logging
- Select or disable severity type for logging
- Toggle write log to txt

Syntax for logging:
Log.message(“category”, “severity”, “content”);

Display of the log (draft):



```
[Game] Game started
[Player input] Pressed 36
[Player input] No valid input key
[Player input] Pressed 37
[Player] Moved left
[Player] Moved from x = 21 to x =19
[Game] Game Crashed
```

3.2
For the logging part of the project we also just responsibility driven design.

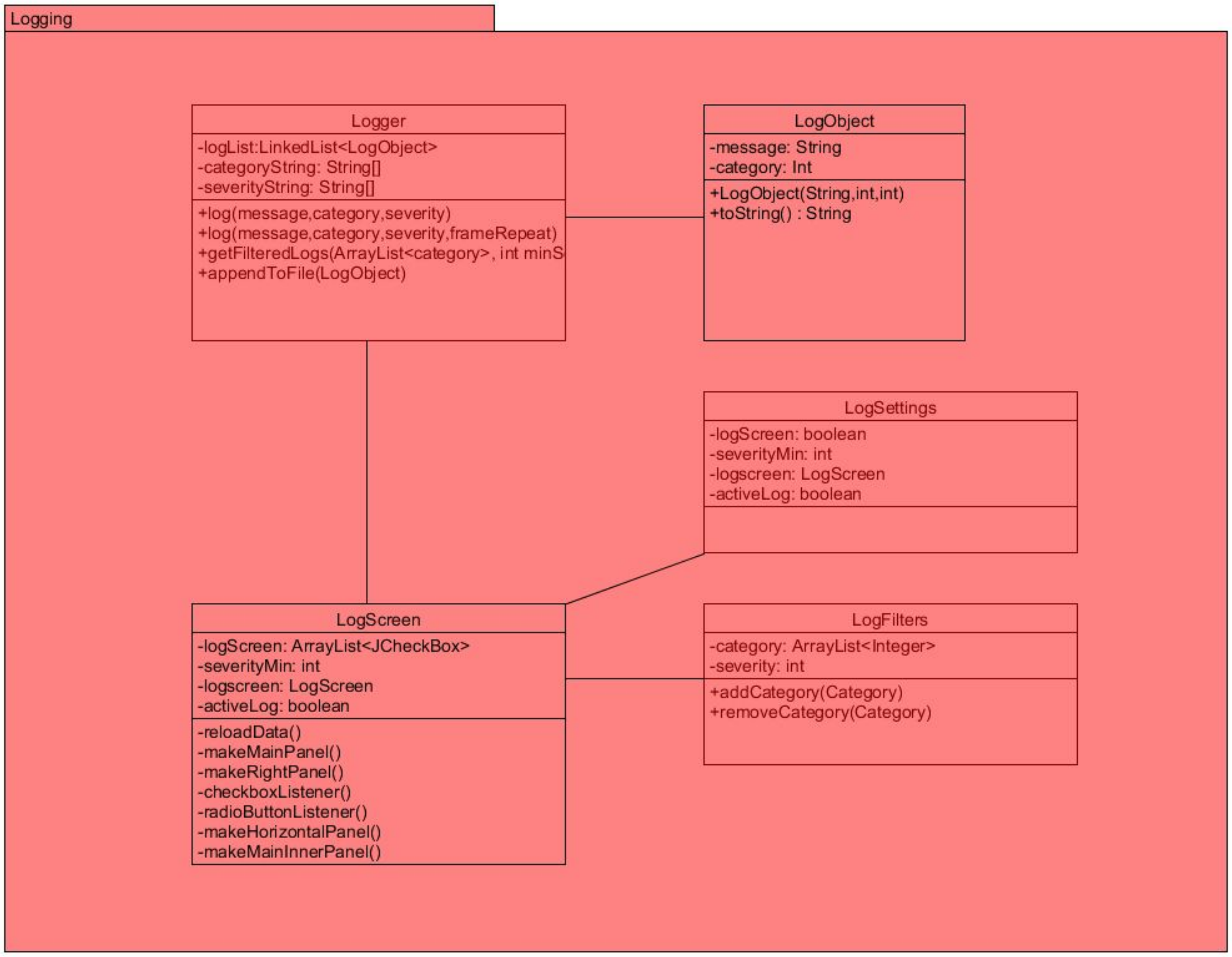
LogObject	
Superclasses:	
Subclasses:	
Store Log message with type	Logger

LogSettings	
Superclasses:	
Subclasses:	
Controls the settings of the logger	LogScreen

LogScreen	
Superclasses:	
Subclasses:	
Shows an overview of all the logs	Logger LogSettings LogFilters

LogFilters	
Superclasses:	
Subclasses:	
Stores all the information of the filters	LogScreen

Logger	
Superclasses:	
Subclasses:	
Stores and creates LogObjects	LogObject LogScreen


Logging
Logger
-logList:LinkedList<LogObject>
-categoryString: String[]
-severityString: String[]
+log(message,category,severity)
+log(message,category,severity,frameRepeat)
+getFilteredLogs(ArrayList<category>, int minS
+appendToFile(LogObject)
LogObject
-message: String
-category: Int
+LogObject(String,int,int)
+toString() : String
LogSettings
-logScreen: boolean
-severityMin: int
-logscreen: LogScreen
-activeLog: boolean
LogScreen
-logScreen: ArrayList<JCheckBox>
-severityMin: int
-logscreen: LogScreen
-activeLog: boolean
-reloadData()
-makeMainPanel()
-makeRightPanel()
-checkboxListener()
-radioButtonListener()
-makeHorizontalPanel()
-makeMainInnerPanel()
LogFilters
-category: ArrayList<Integer>
-severity: int
+addCategory(Category)
+removeCategory(Category)