Assignment 1

Exercise 1

1.
First we search all noun phrases and seperate them into obvious classes, uncertain candidates and nonsense:

- **Level** Obvious class
- **Ceiling** Obvious class
- **Bubble** Obvious class
- **Wall** Obvious class
- **Game** Obvious class
- **Player** Obvious class
- Platform Nonsense a platform consists of wall(s) and floor(s)
- **String** Uncertain candidate
- **Menu** Obvious class
- Start button Nonsense, because it is part of the menu
- Quit button Nonsense, because it is part of the menu
- Background Nonsense, because this is just a part of a level.
- Lives Nonsense, because it is part of a player.
- **Timer** Obvious class
- Score Nonsense, because it is part of a player.
- Points Nonsense
- Height of a bubble Nonsense, because this is a feature of the bubble.
- **Power-up** Obvious class
- Sound effect Nonsense
- **Collision** Uncertain candidate
- Background Music Nonsense

Next we make CRC cards of the candidate classes. First we defined the responsibilities of each candidate class. From here we can see with which other classes it collaborates. To find the requirements we searched in the requirements. We found out that the uncertain candidates: String and Collision, were good candidates. If we wouldn't use the String class, we couldn't percept a collision with either the ceiling or a bubble which is needed. Also if we wouldn't use the Collision class, we couldn't manage the collisions between certain classes. We found it out with the CRC cards, because if we walked through a scenario we missed those two classes.

So all classes we use are:

Level	Ceiling	Bubble	Wall
Game	Player	String	Menu
Timer	Power-up	Collision	

Level	
Superclass(es):	
Subclasses:	
Making a Level consisting of walls and a ceiling	Walls
Send everything to game	Game
Manage a timer	Timer
	Ceiling
	Player
	Bubbles

Ceiling	
Superclass(es):	
Subclasses:	
Representing the ceiling in a level	Level
Block bubbles	Bubble
	Collision

Bubble	
Superclass(es):	
Subclasses:	
Representing a bubble in a level	Level
Fall according to the rules of physics by calculating the next position.	Power-up
Split into two when hit, or disappear hit at smallest size.	Player
May drop power-up when hit.	Wall

Keep track of position.	Collision

Wall	
Superclass(es):	
Subclasses:	
Representing a wall in a level	Level
Block the bubbles and players	Bubble
Specify the location of the wall	Player
Specify the width and height of the wall.	Collision

Game	
Superclass(es):	
Subclasses:	
Creates a new level when "Start game" is selected.	Level
Shows a menu, perform action based on the menu choice.	Menu
Calculating the score	Player
Terminates the game when “Quit game” is selected	Timer

Player	
Superclass(es):	
Subclasses:	
Representing a player in a Level	Level
Move left and right	Bubble
Shoot a string	Wall
Keep track of lives	Power-up
Keep track of position in the level	Collision
	String

Menu	
Superclass(es):	
Subclasses:	
Contain a button to start or quit the game	Game
Send everything to game	

Timer	
Superclass(es):	
Subclasses:	
Count down in a level	Level
Specifies the time left in a game.	Game
Responsible for the countdown at beginning of the game.	

Power-up	
Superclass(es):	
Subclasses:	
Give the player a boost	Player
	Bubble

String	
Superclass(es):	
Subclasses:	
Move upwards when the player shoots it	Player
Splits a bubble in two when it hits one	Bubble
Disappears when it hit the ceiling	Collisions
	Ceiling

Collisions	
Superclass(es):	
Subclasses:	
Manage all collisions	Bubble
	Player
	Wall
	Ceiling

	String

The following classes we have also implemented in our initial version:
Bubble Player String Timer
Collision

The Model and Controller classes have the functionalities of the Game class.
The Model gathers all the data and the Controller works with that data.

The Room and RoomData classes have the functionalities of the Level class.
The Room class uses the data from roomdata to process it. The RoomData class has the initial data for every room.

SlickApp is for the graphical part and to run the game which we didn't have with the CRC cards.

Classes which we haven't implemented:
Ceiling Wall Power-up Menu

The functionalities of Power-up and Menu aren't implemented at all so those classes will come. The Ceiling and Wall classes are implemented as rectangles in an arraylist in a room.

2.

Bubble	
Superclass(es):	
Subclasses:	
Representing a bubble in a level	Room
Fall according to the rules of physics by calculating the next position.	CollisionEvent
Split into two when hit, or disappear hit at smallest size.	Player
Keep track of position.	Controller

Collisions

Superclass(es):	
Subclasses:	
Manage all collisions	Bubble
	Player
	Room
	Rope
	Controller

Controller	
Superclass(es):	
Subclasses:	
Checking if there is a collision	CollisionEvent
Start new room	Room
Update instances in a room	Player
	Bubble
	Ropes

Model	
Superclass(es):	
Subclasses:	
Gathers all the data for the game	Bubble
Creating the rooms	Player
	Room
	Rope

Player

Superclass(es):	
Subclasses:	
Representing a player in a room	Room
Move left and right	Bubble
Shoot a rope	Controller
Keep track of lives	CollisionEvent
Keep track of position in the level	Rope
Get the players from Model	Model

Room	
Superclass(es):	
Subclasses:	
Making a room consisting of player(s) and bubble(s) from RoomData	Player
Send everything to Controller	Model
Manage a timer	Timer
Get the rooms from Model	Bubbles
	Controller
	RoomData

Rope	
Superclass(es):	
Subclasses:	
Move upwards when the player shoots it	Player
Splits a bubble in two when it hits one	Bubble
Disappears when it hit the ceiling	CollisionEvent
	Controller
	Room

Timer	
Superclass(es):	
Subclasses:	
Count down in a room	Room
Specifies the time left in a game.	Player
Responsible for the countdown at beginning of the game.	

3.
SlickApp is for rendering and running the game. A separate class for rendering can be made, so the SlickApp only has one responsibility.
Currently, the Controller class also handles Collision. Collision handling could be implemented in a seperate class.
Roomdata is a class which can be merged into the Room class. Roomdata has the initial data of a room and this could be a method in the Room class.
The Wall and Ceiling class aren't implemented currently. The Room class uses the Rectangle class for walls and ceilings. This could be changed to a Wall and Ceiling class to allow for more flexibility in the future.

4/5.
See UML class and sequence diagrams in ‘documentation’ folder.

Exercise 2

Exercise 2 - UML in practice

1.
What is the difference between aggregation and composition? Where are composition and aggregation used in your project? Describe the classes and explain how these associations work.
□In aggregation, the different parts can exist independently of each other. In composition, if the parent class is destroyed, the child is also destroyed.□□In our game, we mostly make use of composition. A couple of classes are used in the Room class, and without a Room, these classes don’t exist for any other functional purpose. The Room class uses the following classes: Wall, Floor, Bubble. These classes have no use without a room, so they are used in the composition form. Another use of composition is that the class Rope is used in the Player class. Without a Player, there aren’t any Ropes and the Player has a list of Ropes as an attribute.□□We use aggregation in the Model class. In this class, Room and Player are aggregated to form the Model of the game. Without a Model, these classes can still exist and

could be used by other classes, but they are nicely aggregated in the Model class, so the Controller can get to them in one single place.□

2.

Is there any parameterized class in your source code? If so, describe which classes, why they are parametrized, and the benefits of the parametrization. If not, describe when and why you should use parametrized classes in your UML diagrams.

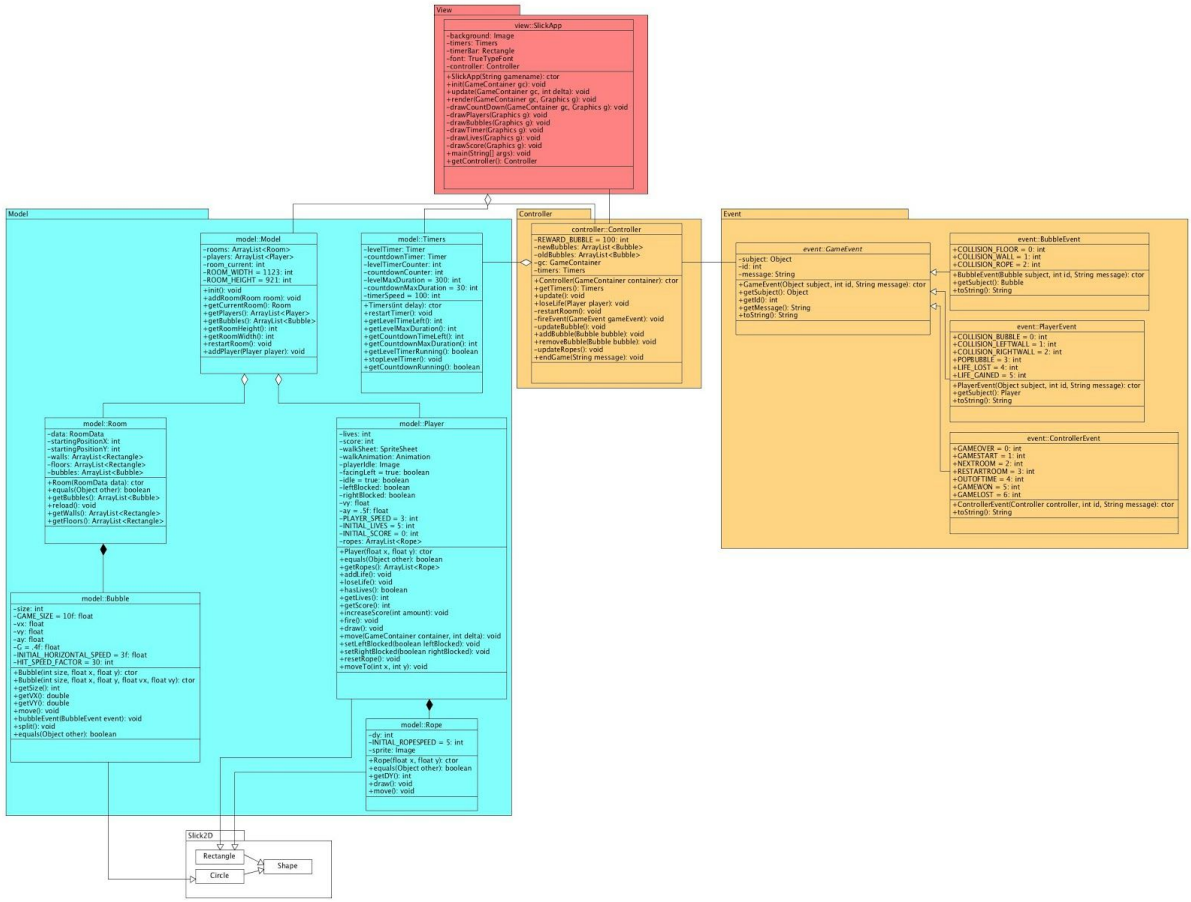
□We don't make use of parameterized classes in our source code. You should use parameterized classes in your UML diagrams when you have classes that use generics and classes that implement these generics. You can use this when you have a class that could take several types of classes as it's content. One use in our game could be collisions; The collision would take a generic type of collision Collision<T>. Another use is lists, like the ArrayList<T>. We use this to make lists of Bubbles, Players, Ropes, Walls, Floors. The benefit of this is that we can use the same class, ArrayList, and its methods on all kinds of objects of different classes.□

3.

Draw the class diagrams for all the hierarchies in your source code. Explain why you created these hierarchies and classify their type (e.g., "Is-a" and "Polymorphism"). Considering the lectures, are there hierarchies that should be removed? Explain and implement any necessary change.□

Also see UML diagram in 'documents' folder. In this diagram are displayed the aggregation, composition and use of subclasses and superclasses. We made a division in packages for the Model, View and Controller and also have a package specifically for Events. In this package, we see use of a hierarchy in the form of Polymorphism. The GameEvent is the superclass and comes in different types and forms: BubbleEvent, PlayerEvent and ControllerEvent. We don't make use of Reuse, so we didn't have to remove any hierarchies or classes for this reason. Another type of hierarchy is in the form of Is-a. We see this in the Model, where Bubble, Player and Rope are all an extension of either the Rectangle or Circle class, which are extensions of Shape themselves. We did this, so we can make use of all the geometric functions and values associated with Shapes in Slick2D. This gives us the possibility to treat the objects as geometric entities which can intersect, move, transform, etc.

Please note that this diagram was made during development, so may be a tad bit out of date.



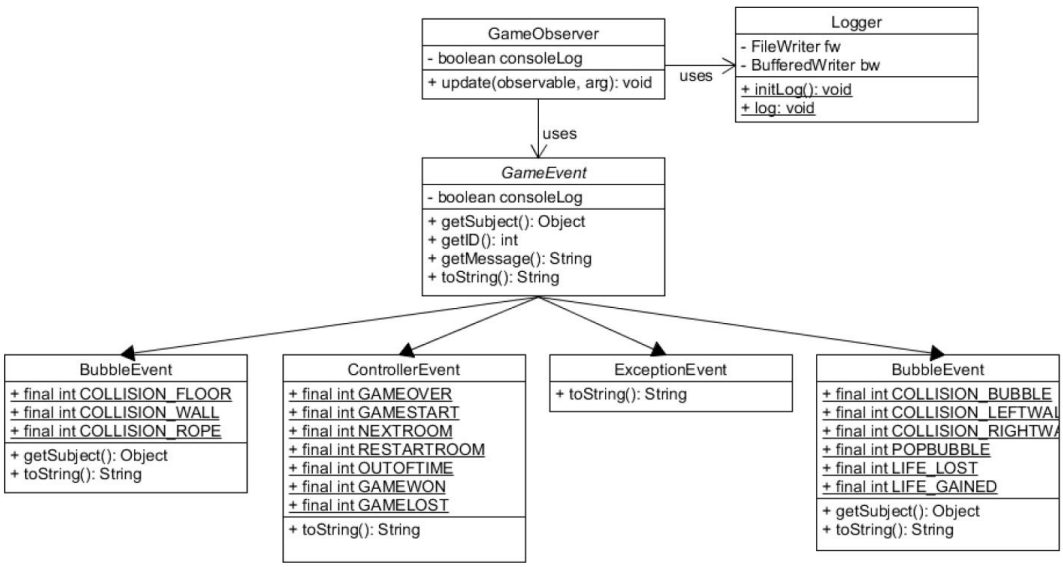
Exercise 3

1.
We've implemented a logger from the requirements in the 'requirements' folder in 'documentation'.

The requirements were:

- The logger logs every event happening in the game. These events are: Bubble related events, Controller related events, Exception events and Player related events.
- The logger writes its output lines to a file
- The logger writes a log file for every game session
- Every output line begins with a date-time stamp
- Every output line contains the source of the event Every output line describes the nature of the event

The UML for this Logger is also found in the Logger.pdf file, in the 'requirements' folder.
There's one error in the diagram: the arrows from the GameEvent class are in the wrong direction. This is done correctly in the complete UML diagram in the uml folder. Another error, is that the BubbleEvent on the most right is supposed to be a PlayerEvent.



2.
From the requirements, we took the following nouns and made a selection of classes: Logger, Event, GameEvent, BubbleEvent, ControllerEvent, ExceptionEvent, BubbleEvent, File.

Other nouns that haven't become classes are: Event (merged with GameEvent), File (we can use the default Java filewriter classes.

Another class we decided to add (not from the requirements) is the GameObserver class, which checks for Events and sends them to the Logger.

The CRC cards came out as following and were implemented in the code:

Logger	
Superclass(es):	
Subclasses:	
Notice when events are triggered	GameObserver
Log events that are triggered in a file	GameObserver

GameObserver
Superclass(es):
Subclasses:

Observe a GameEvent that happens and log it to the console if desired	GameEvent
	Logger

GameEvent	
Superclass(es):	
Subclasses: BubbleEvent, ControllerEvent, ExceptionEvent, PlayerEvent	
Record an Event	

BubbleEvent	
Superclass(es): GameEvent	
Subclasses:	
Record a BubbleEvent	

ControllerEvent	
Superclass(es): GameEvent	
Subclasses:	
Record a ControllerEvent	

ExceptionEvent	
Superclass(es): GameEvent	
Subclasses:	
Record a ExceptionEvent	

PlayerEvent	
Superclass(es): GameEvent	
Subclasses:	
Record a PlayerEvent	