Assignment 5
Bubble Bobble
Group 15

Exercise 1

In this sprint we are adding the new feature 'highscores'.

Requirements - highscores

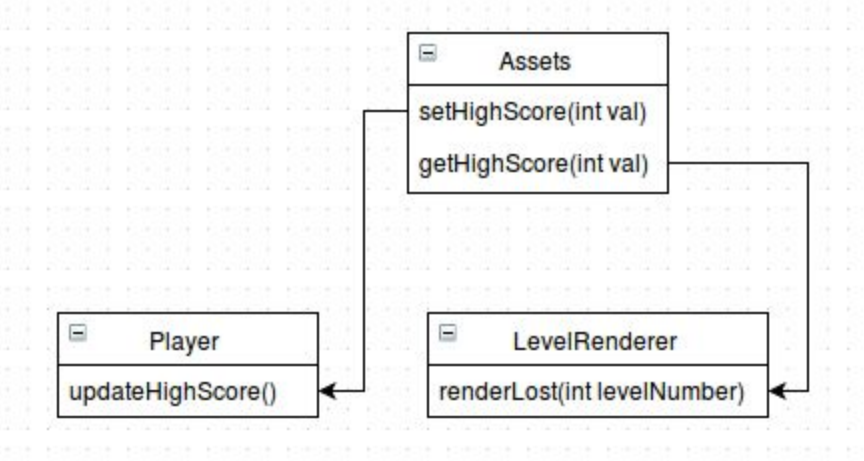
Must haves

- When the player wins or loses the game, then the player shall see the high score on the screen.
- When the player has more points then the current high score, then the player's points will be the new high score.

Assets superclass: none subclass: none	
Responsible for loading and saving the highscore.	

LevelRenderer superclass: none subclass: none	
Responsible for drawing the highscore on the screen.	

Player superclass: GameObject subclass: none	
Responsible for comparing the score with the highscore.	



Libgdx provides an easy way to store files alongside the application. In Libgdx one can create a preference instance, which is a hash map that can hold different values. Everything stored in the hash map is written to a XML file called preferences.

In the Assets class an instance of preferences is created. One key-value pair is initialised with the string 'highscores' and the value is set initially to 0. When the player dies the score is checked with the highscore. If the score is greater than the high score the value is changed and stored in the hash map.

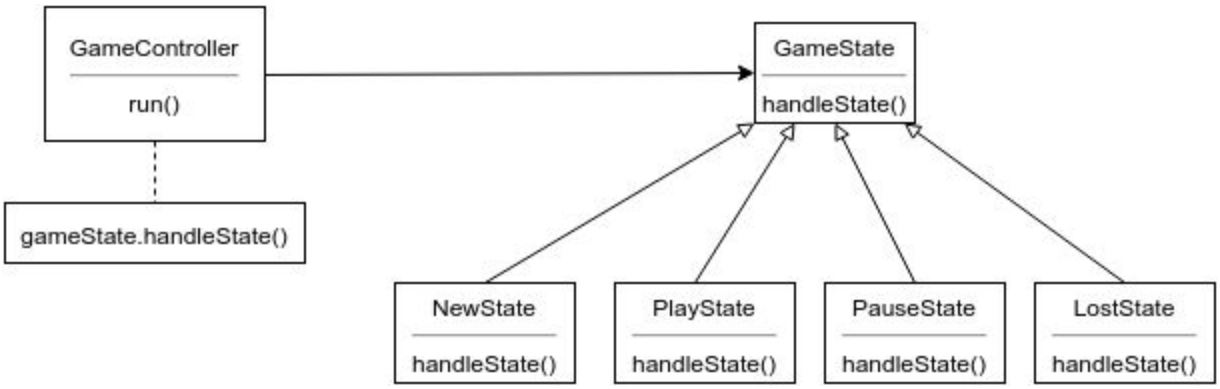
In the LevelRenderer the high score is displayed on the lost screen when the method renderLost is called.

Exercise 2

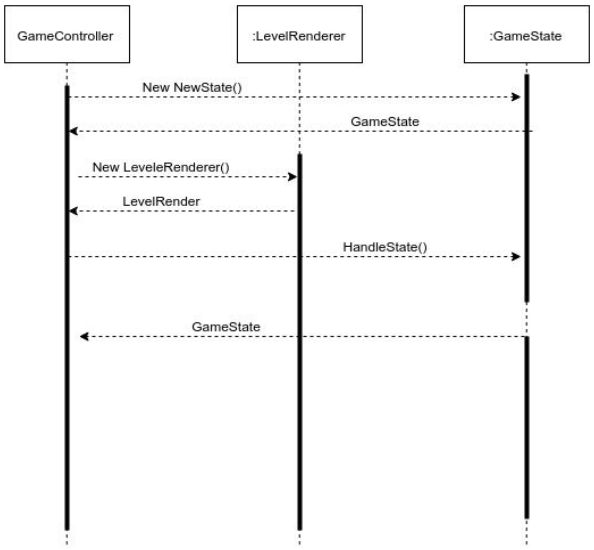
The first design pattern we choose to implement is the **State Pattern**.

1) We used this pattern because the game controller can behave in different ways depending on the state of the game. This can be (at the moment) New, Play, Pause or Lost. Before we implemented the State Pattern this was handles with an enum used to describe the state, following a switch statement decided which function to call to make the GameController work properly. This of course screams for the State Pattern. The pattern is now implement using a GameState interface, which only has the function handleState(GameController controller, float elapsed). For each state a Class is made which implements this interface. The handleState function always return the new state, because in each iteration, the state of the game changes depending on what happens in the handleState function.

2)


GameController
run()
gameState.handleState()
GameState
handleState()
NewState
handleState()
PlayState
handleState()
PauseState
handleState()
LostState
handleState()

3)

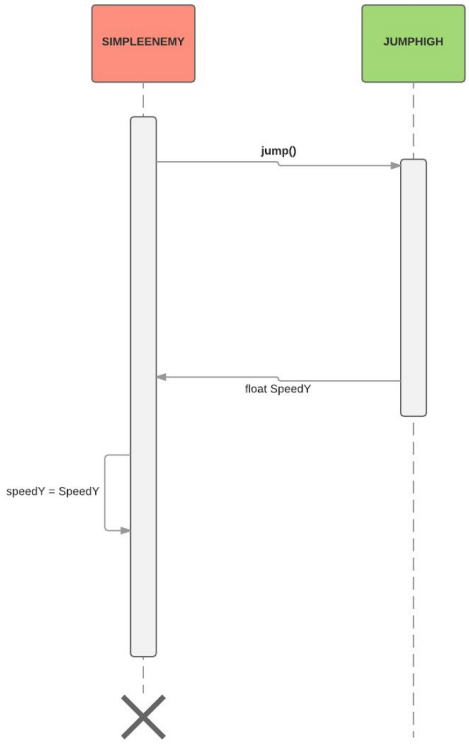

GameController
:LevelRenderer
:GameState
New NewState()
GameState
New LeveleRenderer()
LevelRender
HandleState()
GameState

The second design pattern we decided to implement is the **Strategy Pattern**.

1) This pattern was chosen to better implement the jumping of the enemies. Since all enemies should jump, but not all should jump in the same way, we encapsulate this behaviour in a new interface called JumpBehaviour. This interface is then used to create a protected variable in the Enemy superclass; jumpBehaviour. The enemy superclass also implements a method jump(), which connects the jump to the Y speed of the enemy. The various enemy subclasses then set their wanted behaviour in the constructor. For now, we keep the different behaviours to two: a high jump, and a jump with a random height. Using this design pattern improves our ability to change the behaviour of the enemies (which could even be done on the fly).

2)

3)



Exercise 3 - Reflection

This project was a great learning experience for all of us. After a very mediocre start, with a barely working initial version, we were a bit stressed about finishing the project on a good note. There was little communication, almost no testing being done and tons of checkstyle errors.

However, as the project progressed, we started working more structurized. We believe this had to do with a number of things: for one, our game became playable. Adding more features and improving the game became fun, which meant more time was put into the project which really improved the work being done.

Secondly, we started to develop a well-working routine. The more the project progressed, the shorter the sprint meetings became, as they became much more streamlined and efficient. Testing became rule instead of exception: when a new method was written, a test accompanied it.

These things accumulated into a better code quality, which made the addition or refactoring of code much easier, and therefore more fun.

For us, this is the mayor thing we've learnt: having a well defined project strategy and a good product improves productivity, which in turn improves the product. Creating a needed addition or refactoring takes far less time if a small initial investment is made by following SOLID principles.

These insights were used during the project as well: our current product is well scalable in terms of content (more levels, sounds and enemies are easy to add). Testing was an issue at first, as LibGdx doesn't initialize OpenGL during testing and therefore all textures return a null pointer exception. However, changing our sprint planning (as an Agile team does) around to have one team member focus completely on this major problem, allowed us to up our coverage to around a very decent 80%, with all testable code tested.

The relatively high workload of this course forced us to put in more hours than we did in the beginning. After the realisation came that the initial product was not of sufficient quality, we reevaluated our process. As a result, the sprint planning was made more detailed and we started using Trello more intensive.

Another positive point was that everyone in the group started better doing their share after the first iteration: everyone was responding, collaborating and committing more.

However, we do believe we could've start doing this earlier. For instance, in the fourth assignment the test coverage went up by about 40%. Had we done this earlier, we would've had a higher grade and less effort doing it in hindsight. This once again is our main lesson: **a small initial investment avoids lots of work after.** In other words, we've experienced technical debt in real life.

As a point of critique: at the start of the course, it wasn't immediately clear which deadline was when. Also the exact wanted files for RDD design for instance were not

completely clear. This resulted in a less structured sprint iteration the first time (as we did not completely understand what was due when).

Furthermore, the use of some tools, such as Octopull, wasn't exactly clear to us. For instance the tagging of commits, which we've faithfully done, is something we're still not sure of the purpose of.

When implementing the different design patterns, we sometimes had to refactor code in order to make the pattern work. While this is not necessarily bad, we sometimes had to change large parts of the code in order to implement a small feature with a certain pattern. In some cases this wouldn't make practical sense (as it did not pay of terms of code quality, in some cases even making it worse). This was however, a great way to learn how to implement these patterns in a real life scenario.

We believe our game represents the original closely, with things as multiple enemies, level transitions and the original sounds. Where the initial version was lacking in features and code quality, we as a group have had a strong learning curve, with our final game being well polished, fully featured and well written. It is easily expandable, properly tested and well structured. Our group collaboration has become more structurized as well, with more communication, more productivity and more results being booked. All in all, we believe we've become better programmers and better team players as a result of this course.