Exercise 1 - 20-Time, Revolutions

1.
We chose moving walls as an extra feature this time. This means that we had to restructure our wall package so that it could handle movements.

We used multiple elements of java to implement it. There are abstract classes, inheritance elements and interfaces. First we wanted to move bubble, player and duowalls. But our player can only move left and right, not up or down. So making the player and duowall moving would be useless.

We used a movement class to implement the movement behavior. Because not all walls need movement. The class self is abstract and vertical movement, horizontal movement and no movement extend from it, which aren't abstract.
Then we created the three type of movement walls for the bubble wall. Those walls use the movement classes and extends from bubble wall, which is made abstract. So in the game there are only childs from the bubblewall (HorizontalMoveBubbleWall, VerticalMoveBubbleWall and NoMoveBubbleWall).

This way of implementing the moving walls is clear and structured. You can easily create a moving wall and give the begin coordinates, width and height, moving borders and the speed. So this is really flexible. We also created a coordinates object for cleaner code and shorter constructors. And because it's used often in the code.

As you can see in the code and the crc cards it's implemented following responsibility driven design. By splitsing movement to separate classes. And also using multiple wall classes for horizontal movement and vertical movement.

The CRC cards:

Wall (abstract)	
Superclasses:	
Subclasses: BubbleWall, PlayerWall, DuoWall	
Will setup the basics of every wall.	- Level - Coordinates - Dimensions

PlayerWall	
Superclasses: Wall	
Subclasses:	
Wall that will only block a player.	- NormalLevelFactory - WallPlayerCollision

DuoWall	
Superclasses: Wall	
Subclasses:	
Wall that will block both player and bubbles.	- NormalLevelFactory - WallPlayerCollision - WallBubbleCollision

BubbleWall (abstract)	
Superclasses: Wall	
Subclasses: HorizontalMoveBubbleWall, VerticalMoveBubbleWall, NoMoveBubbleWall	
Will setup the basics of a wall that will only block bubbles.	- WallBubbleCollision

HorizontalMoveBubbleWall	
Superclasses: BubbleWall	
Subclasses:	
Wall that will move horizontally and block bubbles.	- MoveHorizontally - NormalLevelFactory

VerticalMoveBubbleWall	
Superclasses: BubbleWall	
Subclasses:	
Wall that will move vertically and block bubbles.	- MoveVertically - NormalLevelFactory

NoMoveBubbleWall	
Superclasses: BubbleWall	
Subclasses:	
Wall that will not move and block bubbles.	- NoMove - NormalLevelFactory

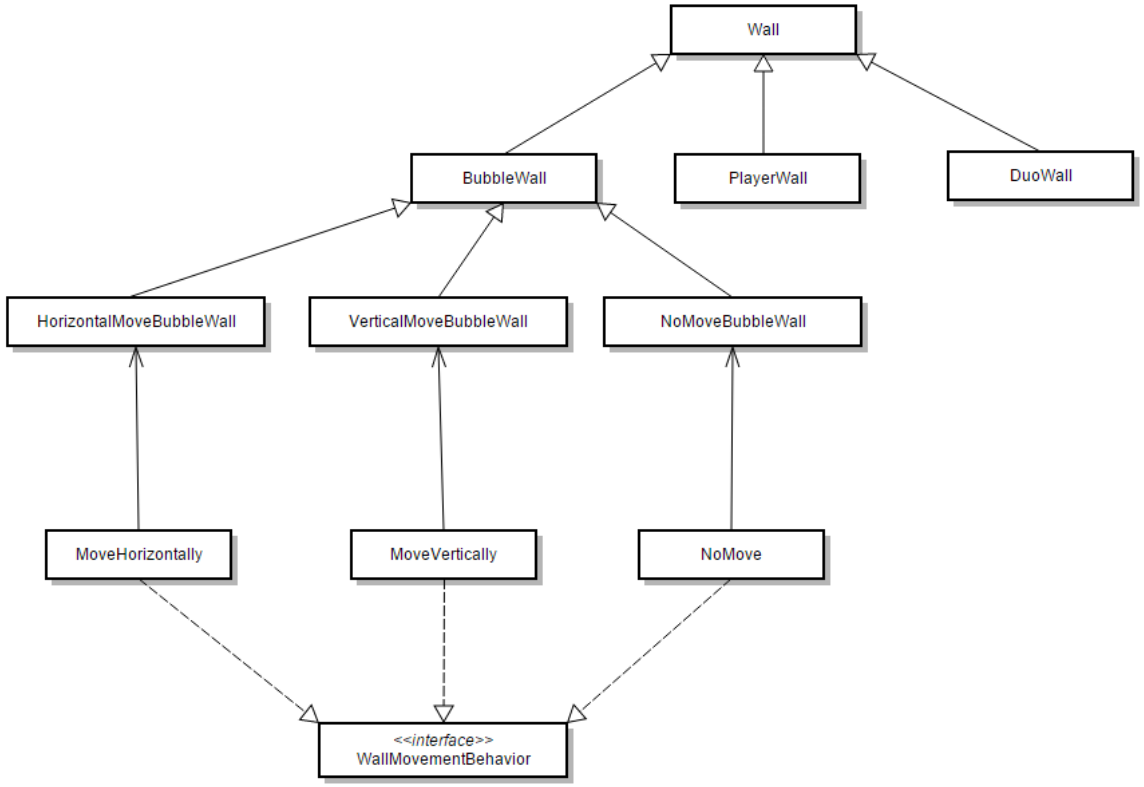
WallMovementBehavior (interface)	
Superclasses:	
Subclasses:	
Defines the methods that a mover has to implement.	- NoMove - MoveHorizontally - MoveVertically

NoMove	
Superclasses:	
Subclasses:	
Will ensure that a wall will not move. Implements WallMovementBehavior.	- WallMovementBehavior - NoMoveBubbleWall

MoveHorizontally	
Superclasses:	
Subclasses:	
Will ensure that a wall will move in a horizontal direction. Implements WallMovementBehavior.	- WallMovementBehavior - HorizontalMoveBubbleWall

MoveVertically	
Superclasses:	
Subclasses:	
Will ensure that a wall will move in a vertical direction. Implements WallMovementBehavior.	- WallMovementBehavior - VerticalMoveBubbleWall

UML Class Diagram

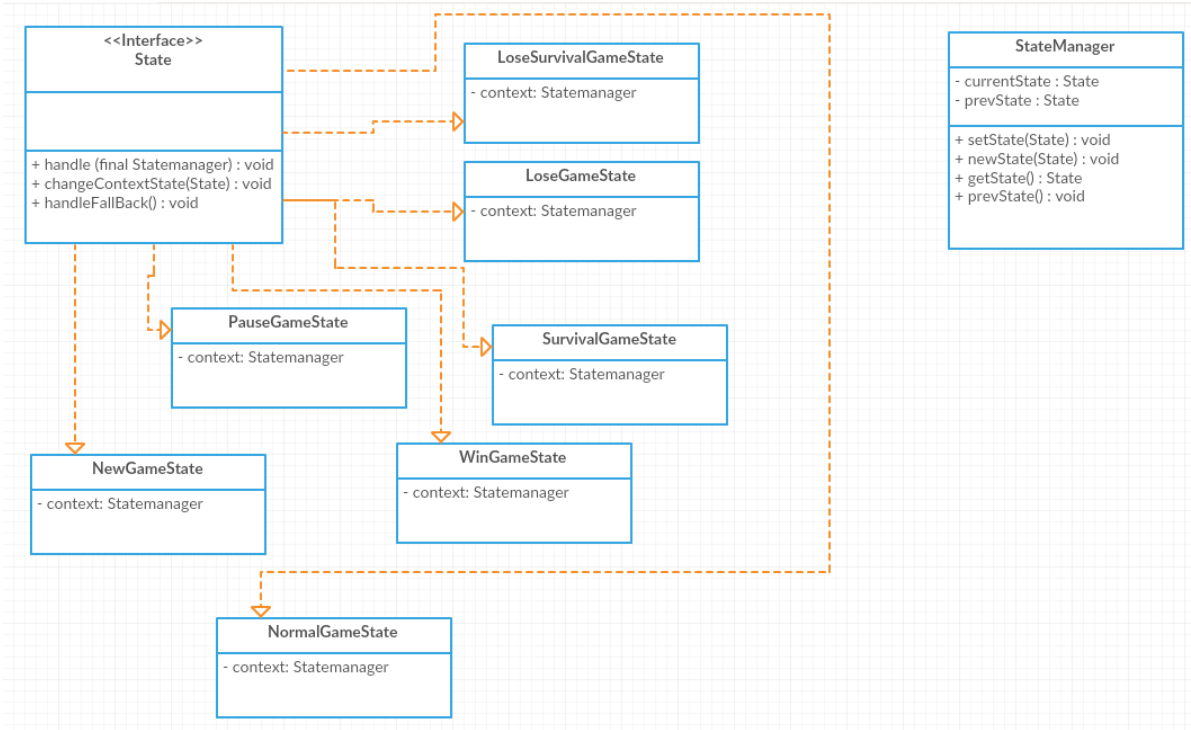


Exercise 2 – Design Patterns

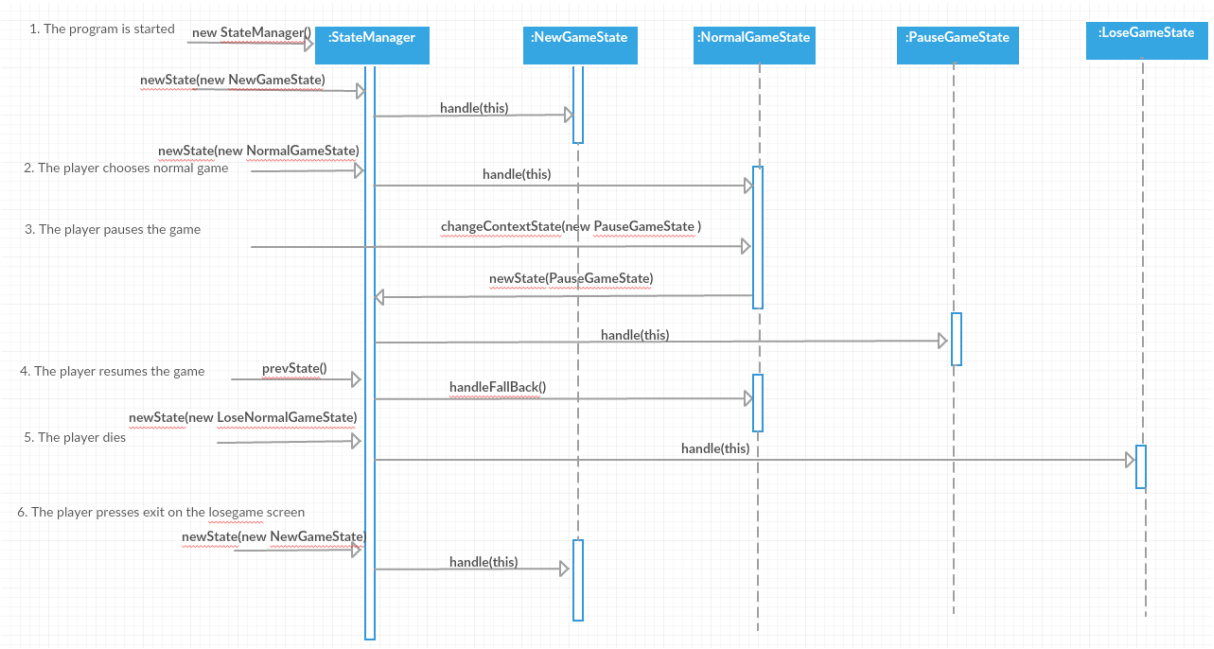
State pattern

1. We chose to implement the state pattern, because we could easily use it for the states the game can be in. States like when the player is playing the game or when the game is paused or when the player dies. When using this pattern we can easily add another state to the game without having to refactor a big part of the code. Every state has its own rules and its own execution. Thus adding a state is easy. It also adds up to the structure of the overall code. When you want to find something that happens in a current state you can just look in the respective state class and find what you are looking for or what you want to change in that class. We implemented this by making one manager class and a state interface with other classes implementing that interface. The StateManager class is created in our MainRunner class (the class that has the main method). From here we initiate the first state, the new game state. From there on the game starts and the states should change. When a state change occurs the StateManager jumps in. The StateManager changes its current state and executes the state its handle function (sort of initiate function).

2.



3.

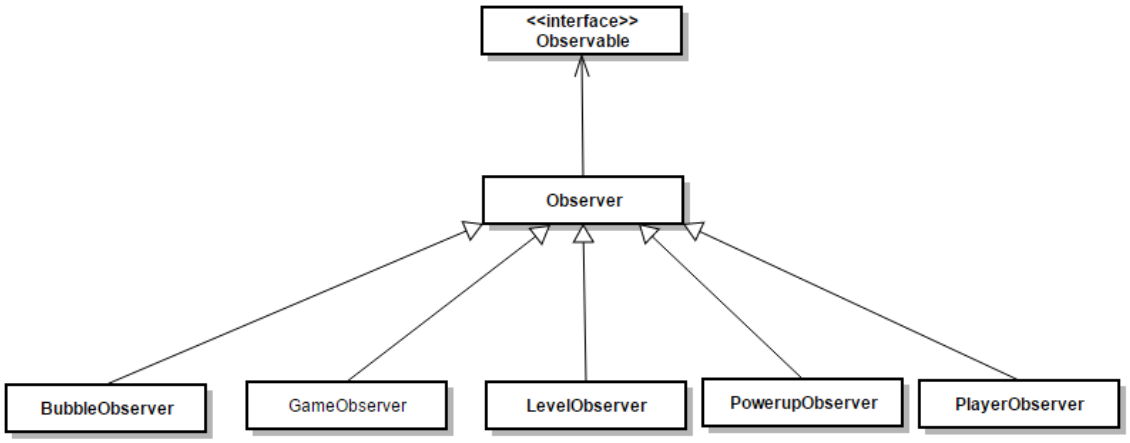


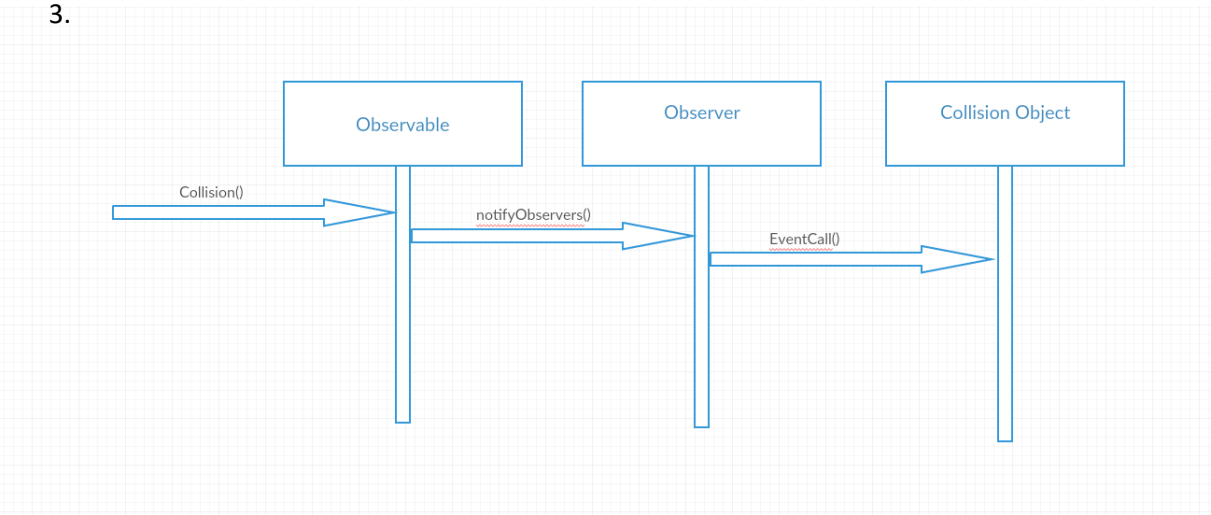
The arrows pointing inwards from the far left are from various other objects. Those objects from top to bottom are

- MainRunner
- MainRunner
- StartScreen
- NormalGameDriver
- PauseScreen
- NormalGameDriver
- LoseScreen

The actions and handling are the same for the survival game type. Now only the normal game is shown. The exit button on the losescreen works exactly the same and calls exactly the same states as the pausescreen and winscreen. That is why I did not take those into account.

Observer pattern

1. This pattern is not that much a new one, but it got changed a great deal. Every class that should have their own listener got their own listener and their own methods that handles the notifier method calls. We made a couple of different methods that handle the different collisions. Not all the subclasses of observer implement these methods, since that would result in too many empty methods. Therefore the methods that need to be implemented are overridden from the abstract parent class. The why we implemented this pattern question is almost the same as last time we explained it. Only this time we came to the conclusion that the observer pattern does not fit the collisions well enough, but since we were already using it we saw no way to remove this pattern completely and just go with this pattern with some minor adjustments. For example the specific overridden methods.
2. 

3. 

Exercise 3 – Wrap up

Start of the project

When we started this course, we knew absolutely nothing about software engineering, most of us didn't even know what it meant. This first assignment, to create a fully working game,

was therefore quite hard. Especially given the small time window in which we had to build it. Another difficult factor was that we needed to create documents like function requirements and sprint plans, which we had no experience with. With the help of the lectures and our TA we were able to quickly build a game that fulfilled our requirements, even though the code quality was quite terrible.

Iterations
The weekly iterations were the most interesting part of the course, especially implementing the techniques we learned during the lectures. The best part about the weekly iterations was the diversity between the different exercises. Some weeks all groups had to implement the same feature, but other in others we were able to improve our game in a way we wanted (and the TA agreed on). While we had some difficulty with creating the sprint plans and sprint reflections at first, they became second nature during the final iterations.

What we learned during the lectures
During the lectures we learned many different aspects of software- engineering and design, and to understand all of them was a challenging task. Integrating most of these designs in our project helped us a lot with seeing their potential value. When you're learning about a design pattern during a lecture you'll probably think something along the lines of: "Interesting, but I'll never use it in my project". But when you are forced to use these patterns in your project you will truly learn about their advantages and disadvantages. And a lot of the design patterns turned out te be very useful. Sometimes you can use a design pattern exactly as described in the lectures, but mostly you can use a principle to improve the code structure.

The difficulties we came across
The biggest difficulty with implementing the design patterns introduced during the lectures, was that our initial version was hacked (Definition of Erik Meijer) together very quickly, and didn't follow any design strategy whatsoever. This made it extremely difficult for us to add these design patterns to our code base, since most of the times we had to rewrite a lot of our code. Now with the final version coming up it seems like this whole project was fixing the bad code we hastily introduced in our initital version. If we would have to recreate this project from scratch, I'm sure we could do it in a very short time and the code would be of higher quality because of all the design patterns we now know how to implement.
It took us quite long to get all of our tools up and running. Maven was working during the third iteration and we only found out how to generate reports during the final iteration. Because we couldn't generate these reports each iteration, we couldn't check the amount of warnings our tools gave us, which lead to a big number of warnings that we will have to fix for the final project.

Final version
With the final version coming up we want to focus on a few specific things the coming week. There are two things that we will focus on. One is the code quality, we want to have good structured code. If your code is good, then the chance is great that your game is also good. Secondly a graphical overhaul of the entire game, because we want our final project to look good.
Another smaller goal is to increase our test coverage, so we can assure the quality of our final delivery. With this comes the goal to keep our checkstyle, pmd and findbugs warnings as low as possible.

Future projects
We all agree that future projects we have to work on will be much easier, since we have learned all these engineering strategies. Next time our code will be of good quality from the start of the project, and we would have to spend a lot less time on refactoring and

restructoring the code. I hope we can work on a similar project with the same team in the nearby future, so we can see if we actually improved.

This team
We created this team because we were a group of friends, who happened to sit next to each other as the teacher told us to create groups of five, so it wasn't that difficult of a decision. In the beginning we had some communication issues like 'who can work on the project on which days?' and 'who can hand it in on friday?'. Later in the project we became used to each others schedule, so without needing to discuss it we all knew who was working on it and when. The biggest difficulty we had to overcome as a group was when Naomi left our project group during the second sprint. Suddenly we all had to put much more time in the project, as we weren't prepared for this to happen. We adjusted the following sprint plans to take our smaller group into account. Unfortunately since the assignments didn't scale, we all simply had to do much more work.

Conclusion
All in all we had a lot of fun with this project, and learned a lot from it. Even though the end result is not perfect in the slighest way, we're happy with it. Because we now know how to improve our future projects. And although the start wasn't that great the end product is of much better quality.

Project feedback
Some advise for this project next year would be to move the TA meeting to a monday. To give the groups a day more to implement the features requested by the TA. We all agreed that the deadline on friday night is fine and should not be moved.