Exercise 1

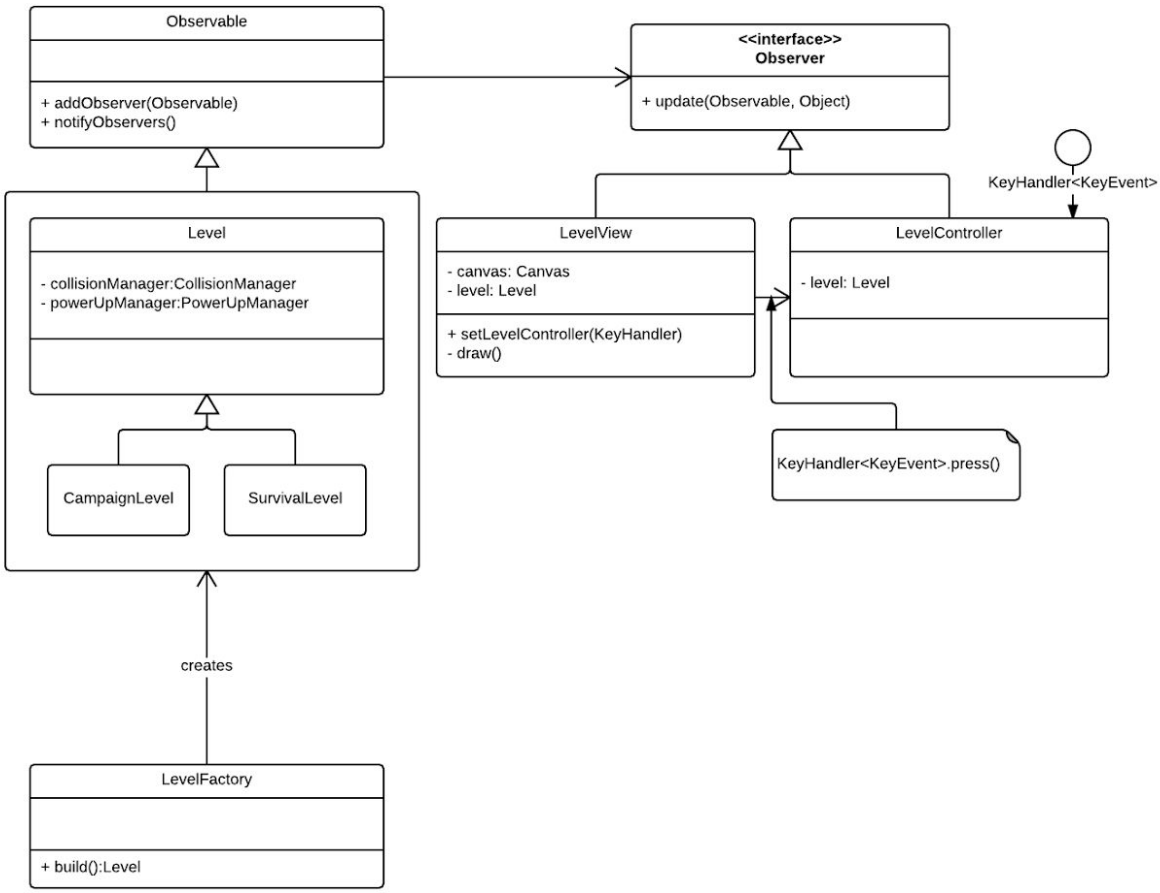
We started with making the CRC cards, which can be found in appendix B. We decided to use the Model View Controller design pattern. When making the CRC cards we forgot that the model had to extend Observable so we fixed that in our actual code.

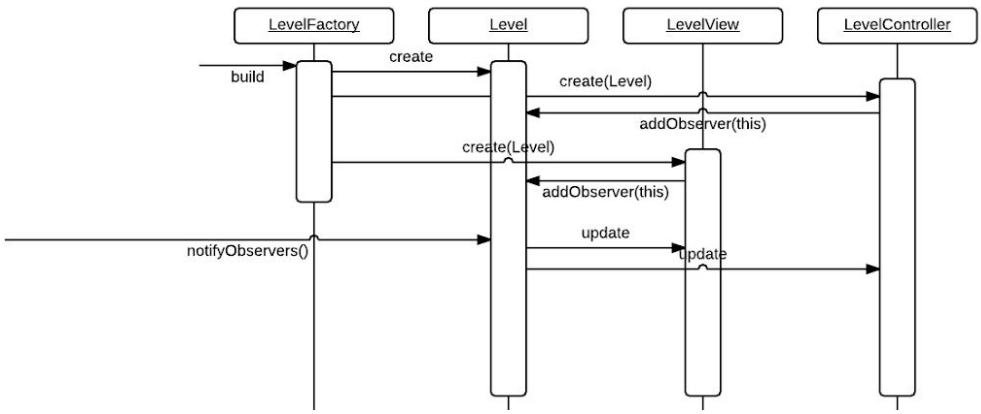
For implementing the custom level-builder we came up with different elements which can be dragged into the canvas. These elements are represented by for example the Ballelement class. We also needed views, for example BallElementView, and a controller, LevelBuilderController. The UML for this exercise can be found in Appendix A. Because we use the MVC-pattern we have implemented that the views observe the models.

Exercise 2

Design pattern 1: Model-View-Controller pattern for Level.

1. The level functionality of the game is split in three classes with each a distinct responsibility. Firstly, the model class, Level, has the responsibility to manage all the necessary data and business logic to be a Level. Secondly, the view class, LevelView, draws all elements in Level that should be displayed on the screen and delegates user input to the controller. Thirdly, the controller class, LevelController handles user input and translates this to notifications to the model class. This communication is done via the Observer pattern. Level is observed by both LevelView and LevelController and they get notified by when data in Level changes, and act accordingly. For example, for LevelView by redrawing the Level. Additionally, LevelController gets notified by LevelView via a listener pattern, LevelController being the listener for key events (user input).
   The reason to use the MVC pattern is because it clearly splits the responsibility of the functions of Level clearly, and promotes clear, standardized communication. Because of this, code is easier to maintain and also easier to test, as not the full level has to be tested. For example, when wanting to test logic of Level, not the whole view (which is usually pointless to test) has to be instantiated.
2. In the class diagram, multiple design patterns are shown (Observer and Factory), as they are necessary to implement to MVC pattern properly. The sequence diagram shows the creation of a Level and it's components and the a typical process flow when the observers are notified.



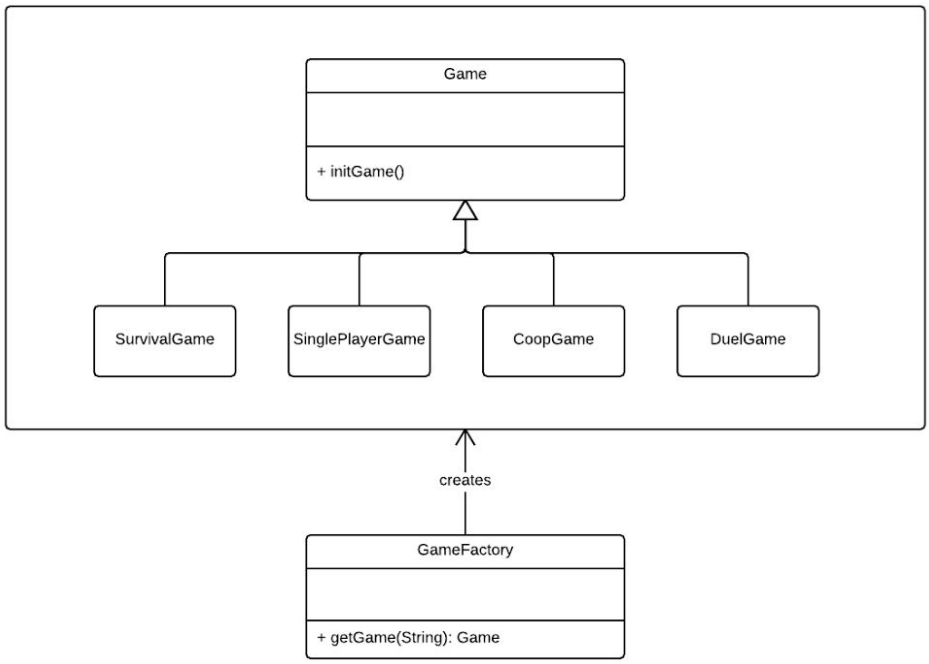
3. 

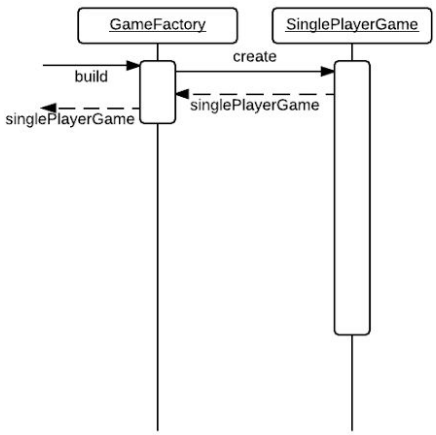
Design pattern 2: Factory pattern for Game

1. The Doob game has multiple game modes: SinglePlayerGame, DuelGame, CoopGame and SurvivalGame. Each mode has distinct features, which should be

spread among separate classes. In order to easily create a certain version of Game, it is logical to use the factory pattern. This makes sure the logic for determining what game to use is handled in the GameFactory, and not in the Game class itself, or any caller class (such as the MenuController, that loads a certain game upon a button action). The GameFactory determines what Game to load and returns the right one. The reason to use this pattern is that it spreads out the responsibility and improves code readability by removing “if-instanceof” logic away from the other logic.

2. The UML is as follows:


Game
+ initGame()
SurvivalGame
SinglePlayerGame
CoopGame
DuelGame
creates
GameFactory
+ getGame(String): Game


GameFactory
SinglePlayerGame
create
build
singlePlayerGame
singlePlayerGame

3.

Exercise 3: Reflection

This project started without us ever thinking about any software engineering principles. When we started working on the first working version, a basic model of the program was made. From there everything was coded on the fly. We would just add methods and classes where we would see fit. This lead to many design mistakes in our code, which would all come to surface when making the assignments and improving our code.

One mistake that comes to mind is the god class mistake we made when making the Level class. Almost everything was handled in the Level class: collisions, event handlers, drawing and even more. When the class was first made, and later when extended with several methods, no one gave thought to the design principles. Because of this the Level class became a huge mess.

With assignment 3 the Level class was refactored. Collisions were now handled by their own class, as with drawing and power ups. After this huge refactor we immediately saw how much better our code was when following proper software engineering methods. Not only did the code look better and was it more formatted and easy to read, it was also clear that it was significantly less difficult to extend the game with new functions.

But thinking about it, the way of working was also a big cause of the problems we encountered. Before when working on projects, we would first determine what features our program would have, figure out everything the program would need and only then start thinking about how to model it. With the iterative way of working, we first created the bare minimum of a working game. And while each week we were thinking about how we would implement the features of the next iteration, we weren't thinking much about how that would affect the iterations of the following weeks. This lead to us writing code for one feature, but then when later that same code would need to be used for another feature, the whole code would turn into a mess of design flaws and would later need to be refactored.

The different game modes we implemented are a good example of this. If we would have thought of all the different game modes at the initial design of the game, it would be logical to first create a clear hierarchy. There would be an abstract Game, which would be extended by an abstract SinglePlayerGame and MultiPlayerGame, which in turn would be extended by their respective game modes: Campaign and Survival or Duel and Coöp.

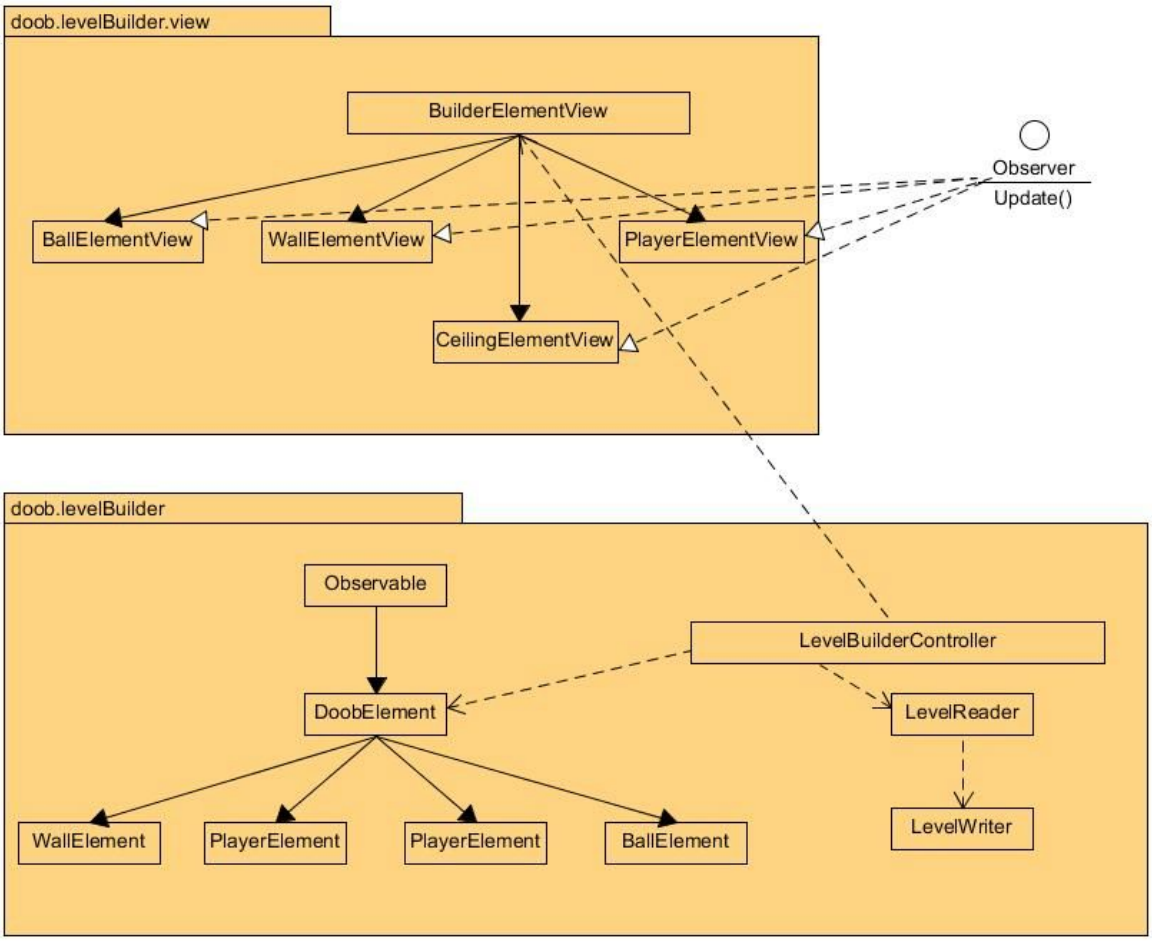
When actually creating these game modes though, creating a clear hierarchy wasn't the first thing we thought of. Having an already working game, we thought about what we would

need to add to implement the different game modes. Refactoring whole classes wasn't something we saw as necessary and so we just added what we needed.

All that was required for the survival mode was a couple of methods for spawning balls and some change in the behavior of the update method. So we just added that in the level class and survival mode was now a working feature. This of course wasn't the right way to do it , which meant it would have to be refactored the next iteration.
At first, the tools such as inCode seemed to only be annoying. However, after applying the suggested changes or over-applying design patterns, the code seemed to become more durable in several ways. Firstly, it was easier to read by a single person. Secondly, the code became easier to discuss because there was a clear structure and thirdly, the project became more fun because it felt good to do something the right way.
The lab learned us that if we keep correctly using design patterns we will be able to avoid the mess we made in our game. The god class problem we created was mentioned in one way or another in the reflections of sprint 2, 3 and 4, showing it took us quite some time to fix it. But we will learn from our mistakes. The lab made us see the direct benefits of all these design patterns. Now when writing new code, we will think about the SOLID design principles and check for each one if our code suffices. Taking the short route will not pay off in the long run.

Luckily there were also a lot of things that went well. Before we started this project, everyone in our group were already friends with each other which made communication a bit easier. We weren't afraid to argue with one another or to call someone out on his mistake. Because of this everyone always agreed on the decisions we made as a team. Sprint plans were made and everyone always had something to do. If someone didn't, he could simply ask on WhatsApp and someone else would have a task ready to do for him.
Us being all being friends did lead to a little problem though: all the discussion we had, was either held face-to-face or through WhatsApp, while our TA wanted to see the discussions we had on GitHub. For the last few iterations we did take the discussion to GitHub.
All in all, the lab learned us a lot about how to maintain proper coding. We made a lot of mistakes but precisely through those mistakes we learned so much. Although this isn't the final release yet, we're all pretty content with the program we created. We can all see how more structured and well written our code became by following the design principles learned in class. It really shows that just following these relatively simple principles can immensely help you write responsible code.

Appendix A



Appendix B

Custom Level CRC

Requirements specification

LevelBuilder (Model)	
Superclass: -	
Subclasses: -	

LevelBuilderController (Controller)	
Superclass: -	
Subclasses: -	
Controls the user input	LevelBuilder, LevelBuilderView
Write the created level to a fxml file.	FXMLWriter

LevelBuilderView (View)	
Superclass: -	
Subclasses: -	
When the model changes, the change is drawed	LevelBuilder
Respond to user input	LevelController

(Abstract) BuilderElement	
Superclass: -	
Subclasses: WallElement, BallElement, PlayerElement	
Keep data	
Keep shared logic of subclasses	

BallElement (Model)	
Superclass: BuilderElement	
Subclasses: -	

WallElement (Model)	
Superclass: BuilderElement	
Subclasses: -	

PlayerElement (Model)	
Superclass: BuilderElement	
Subclasses: -	

BuilderElementView (View)	
Superclass: -	
Subclasses: BallView, WallView, PlayerView	

BallView (View)	
Superclass: BuilderElementView	
Subclasses: -	

WallView (View)	
Superclass: BuilderElementView	
Subclasses: -	

PlayerView (View)	
Superclass: BuilderElementView	
Subclasses: -	

FXMLWriter	
Superclass: -	
Subclasses: -	
Writes the created level to a fxml file	LevelBuilder

Requirements Custom levelbuilder

Must have:

- As a user, I want to be able to make my own level with a levelbuilder.
- As a user, I want to be able to add all possible elements to my created level: balls, walls and players.
- There must be a maximum of two players per level.
- There must be a maximum of three walls per level.
- Two elements cannot overlap. Two players can overlap a bit.
- Make
- As a user, I want to be able to drag the elements I want in the level to a position in the level.
- There must be an option to discard the changes made in the level.
- There must be an option to save the level.
- Players should only be movable on the x-axis, so they may not ‘float’ in the air and can only be on the floor of the level.

Should have:

- There should be an option to set the size of the added balls.
- There should be an option to use the level in the actual game.
- There should be an option to set properties of added walls.
- A button to play the created custom level from the levelbuilder view.
- A button to play all custom made levels
- There should be an option to distinguish which level is made: singleplayer or multiplayer.
- There should be an option to set which player is player 1 and which player is player 2.
- It should be possible to set the amount of time available to complete the level.

Could have:

- It could be possible to set which powerups may be used in the level.

Won’t have:

- A possibility to add custom elements with custom shapes.