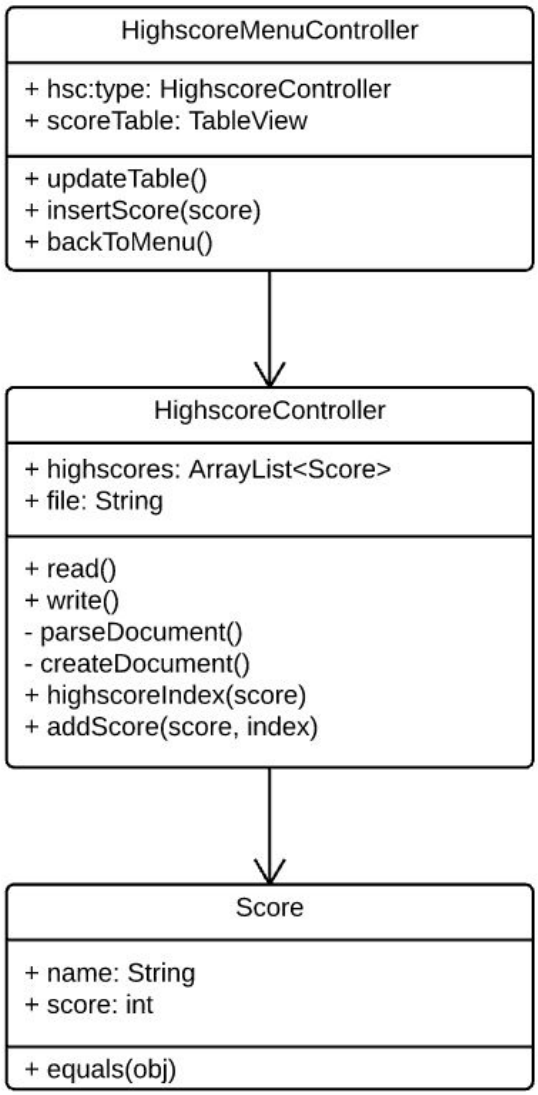
Exercise 1 - New Features

See the appendix for the Requirements and CRC cards.

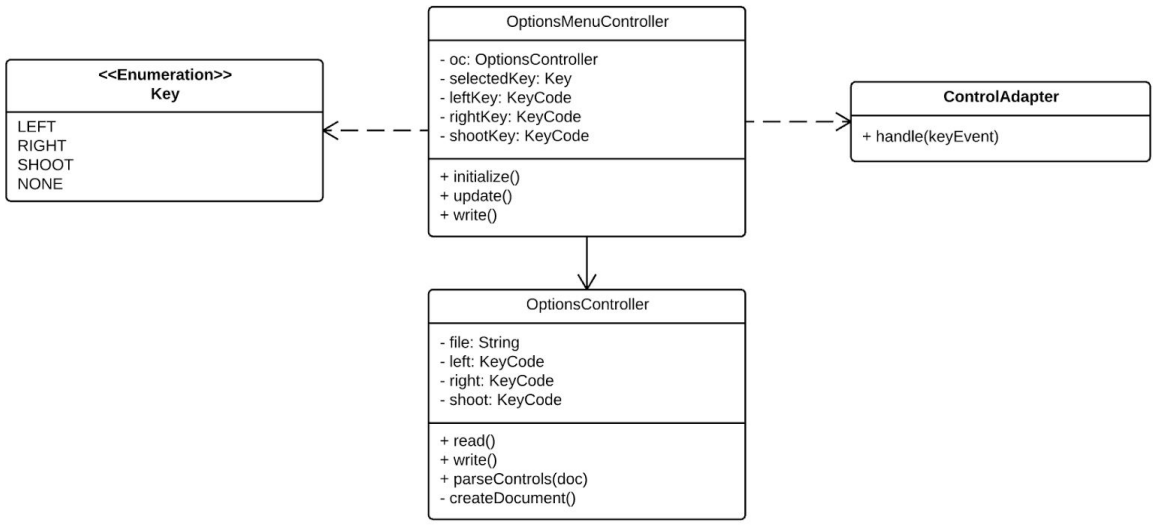
Highscores

We implemented a system that keeps track of highscores. There is a xml file with a maximum of 10 highscores. To display those highscores we created a menu with a table with the scores.
Every time a game ends when the player is out of lives or the last level is completed the highscores menu is automatically opened and the scores are displayed. Also the system checks whether the achieved score is high enough to be among the top 10 scores. The score is also added to the highscore list when there are nog 10 highscores yet.



Options

It is now possible to change the controls of the player. By going to the Options menu, you can click on a button and input a new key. Upon leaving the options menu, all options are saved in an xml file.
Also there is a scrollbar to change volume. But since there is no sound in the game yet this will later be implemented with functionality.



Back to menu

The game now has a back to menu buttton. By clicking on it you navigate back to the main menu.

Play/Pause Button

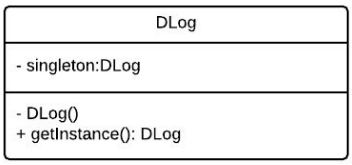
The game now has a play/pause button. By clicking on it the game plays or pauses depending on whether the game is paused or not.

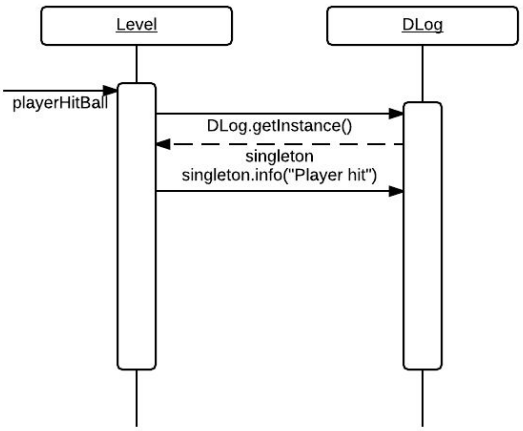
General

The projectile image is improved and it shoots more realistically.
Checkstyle improvements in some classes.

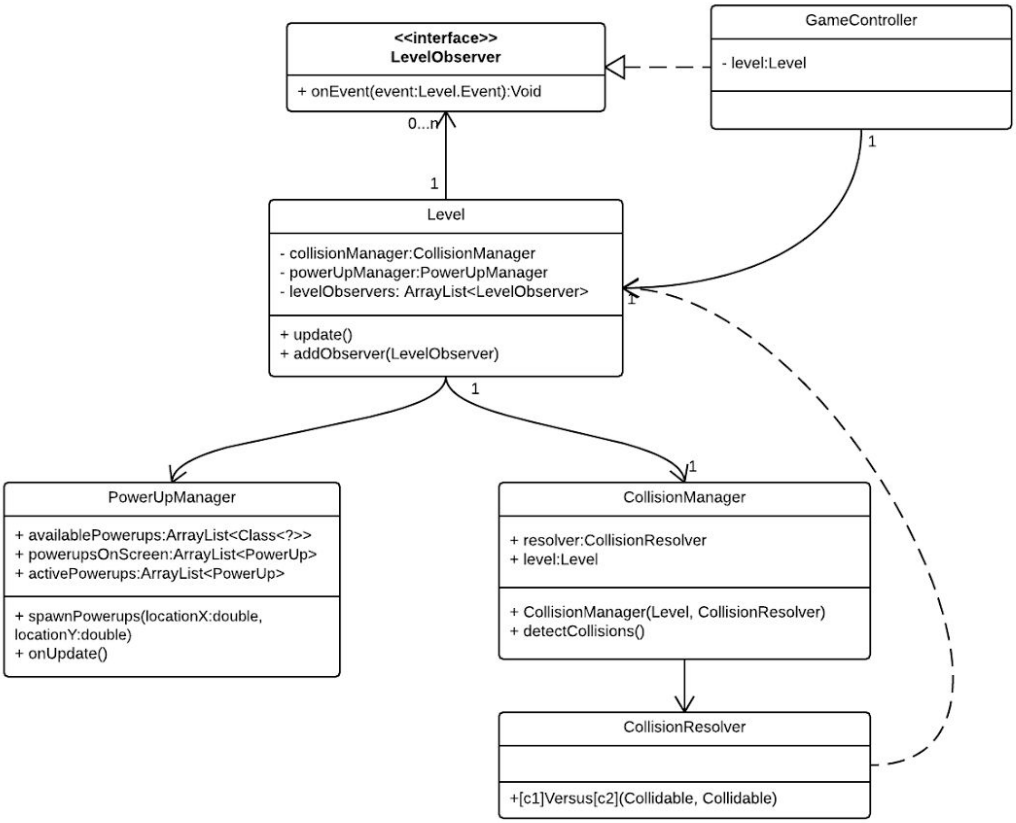
Exercise 2 - Design patterns

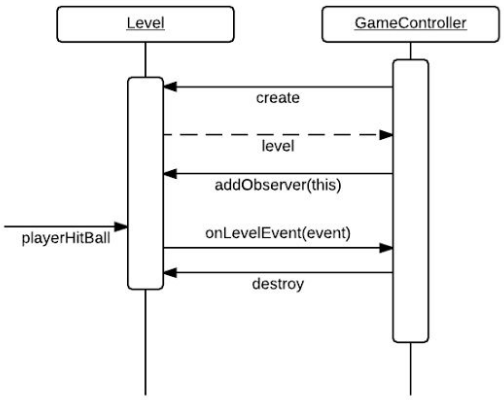
Design pattern 1: Singleton instance of Log class
Why: The Log class is responsible for the logging functionality of Doob. Only one instance of this class is needed throughout the whole project. However, it is a hassle to instantiate the Log class when it is needed, and since one instance suffices, a singleton can be used.





Design pattern 2: Observer pattern for Level
Why: As events happen in Level during the game, other components of the game might need to be notified and react to these events. An example is GameController that loads or reloads a Level when a Player dies or finishes a Level. Because of the pattern to depend on abstractions, it is viable to create an interface, LevelObserver. Level has an attribute that holds all it’s observers and notifies them when necessary. This way, no concrete implementation has to be made in Level for all it’s dependencies.


<<interface>>
LevelObserver
+ onEvent(event:Level.Event):Void
GameController
- level:Level
0...n
1
1
Level
- collisionManager:CollisionManager
- powerUpManager:PowerUpManager
- levelObservers: ArrayList<LevelObserver>
+ update()
+ addObserver(LevelObserver)
1
1
1
PowerUpManager
+ availablePowerups:ArrayList<Class<?>>
+ powerupsOnScreen:ArrayList<PowerUp>
+ activePowerups:ArrayList<PowerUp>
+ spawnPowerups(locationX:double, locationY:double)
+ onUpdate()
CollisionManager
+ resolver:CollisionResolver
+ level:Level
+ CollisionManager(Level, CollisionResolver)
+ detectCollisions()
CollisionResolver
+[c1]Versus[c2](Collidable, Collidable)


Level
GameController
create
level
addObserver(this)
playerHitBall
onLevelEvent(event)
destroy

Exercise 3 – Software Engineering Economics

1. Good practice and bad practice are recognised in the paper by looking at 352 different software projects and splitting them on (duration vs size) and (project cost vs size). When taking the average of both, the projects can be split in an cost/duration matrix. Projects that perform both better than average on cost and duration are recognised as good practise. Projects that perform both worse than average on cost and duration are recognised as bad practise.

2. Visual Basic is a programming language and cannot be used in every project. A possible reason Visual Basic is recognised as good practise might be that projects that use this language are less complex than others. Another reason might be that in the three companies investigated in this paper the best people dealt with Visual Basic. And even if visual basic is a good practise in general, it is not so easy to adopt this quickly. Furthermore, only six projects used visual basic and personally I find this a number which is too small to say something useful about.

3. 1. Decentralization of programmers. It happens more and more often that programmers in for example India are hired to do some cheap programming. I think this will be part of the bad practise quadrant, because of the complications in communicating and managing people from a different culture far away. Eventually this will cost more and the project will take longer. It might also be part of the cost over time quadrant but this was no possible answer on the question.
   2. Experimental software. I think this would belong to the bad practise quadrant because with experimental software the final goal and thus the requirements are not so clearly defined. This inevitably leads to many costly software changes and might even require projects to start over again after a few months. This means they are both not time and cost efficient.
   3. Code review. I think this is a good practise because it forces developers to dive into other code from the project and to find problems in others code. Because you are a bit blind for your own code, others might be able to find defects you were not able to find. It does cost a little bit more time in the begin of the project, but in fixed defects and other benefits such as extendibility, I expect this to be both time and cost-efficient.

4. 1. Many team changes, inexperienced team. This is part of the bad practise group because every time a new team-member is added to the team he has to get to know the project which takes time and thus costs money. If this new member is also inexperienced it will take even more time and money. Furthermore, when an important team-member leaves the team, his responsibilities have to be taken up by the other team-members which might slow down the entire process.

2. Technology driven. This is part of the bad practise group because you depend very much on things that still have to be invented. This means it is very uncertain how long things will take and often things will not go according to the plan. This means projects will be of a longer duration and/or more costly.

3. Bad relation with external supplier. This is part of the bad practise group because you now depend on someone who is not willing to work fast for you. This means he might not meet deadlines or deliver a bad product. Furthermore, you are depending on something you cannot control yourself. This is very likely to cause delays which are costly in time and money.

Appendix - Requirements

Requirements Menu/Highscore DooB

Must have

- As a user I want to be able to navigate to the options menu and enable/disable certain options.
    - It must be possible to change the control keys used
    - It must be possible to change the volume of the sound
    - The settings must be saved.
- It must be possible to navigate back to menu while playing the game, when doing this the currently running game will be ended.
- It must be possibe to see the highscore

Could have

- It could be possible to change the background.
- It could be possible to choose from different collor palletes for the balls.

Won’t have

- It won’t have an option for mulitplayer

Appendix - CRC Cards

HighscoreMenuController	
Superclass: -	
Subclasses: -	
Display the highscores menu	HighscoreController
Show popup when achieved a highscore	

HighscoreController	
Superclass: -	
Subclasses: -	
Manage a list of highscores with a predetermined length.	Score Object
Read scores from a file	
Write scores to a file	

Score	
Superclass: -	
Subclasses: -	
Store a name and a score	

OptionsMenuController	
Superclass: -	
Subclasses: -	
Display options menu	
Handle the reassignment of keys	ControlAdapter

ControlAdapter	
Superclass: -	
Subclasses: -	
Handle the event of pressing a key to reassign controls	ControlManager

OptionsController	
Superclass: -	
Subclasses: -	
Manage the keys that control the player	
Read the control keys from a file	
Write the control keys to a file	