Assignment 1

Exercise 1

1) CRC

See enclosure 1 for the CRC cards.

We started with the requirements, in which many classes we had to put on the CRC cards were clear right away. On the CRC cards, we started with the most important class, the GameController class, and worked our way down. This class has connections with almost all other classes through its direct collaborators and keeping this routine we covered almost all classes. A final check if all requirements were met and that's it.

A main difference between our actual implementation and the CRC cards we derived is that we also have a game object and not just a game controller. This way the GameController has a view and model, and not just a view. This is mainly for a testing purpose. We also came up with a gamefactory and some classes for could have or won't have functionality. We do not have these classes like a settingController because we did not yet implement the settings functionality.

2) Main Classes

Main classes:

- GameController
    - Responsibility: keep track of the progress in the game and the game tick is executed here. Contains graphic elements such as the score and live labels.
    - Collaborations: Collaborates with Level, in which all the updates are executed.

- LevelFactory
    - Responsibility: Load levels from the xml files containing them and prepare them to be used.
    - Collaborations: Collaborates with Level (a level is created) and with Level.Builder to build one.

- Level
    - Responsibility: Contains most of the game logic and update methods of the game.

    - Collaborations: Collaborates with Player, in which the player is moved, with Ball to add new Balls and to move them, with Wall to create walls in the game, with projectile to shoot one.

- Level.Builder
    - Responsibility: Build a level with specified elements.
    - Collaborations: Level

- MenuController
    - Responsibility: Manage the main menu
    - Collaborations: App and GameController which is the controller triggered when a new game is started.

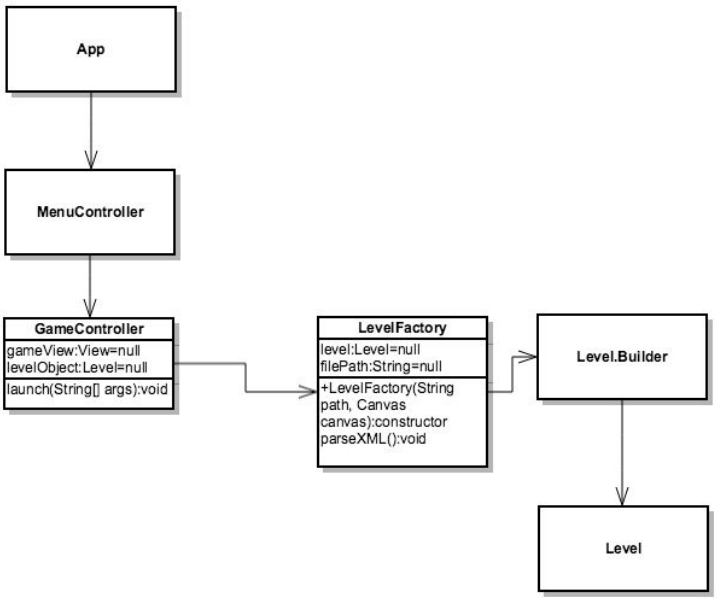
- App
    - Responsibility: Launch the app
    - Collaborations: MenuController

3) Non-main classes

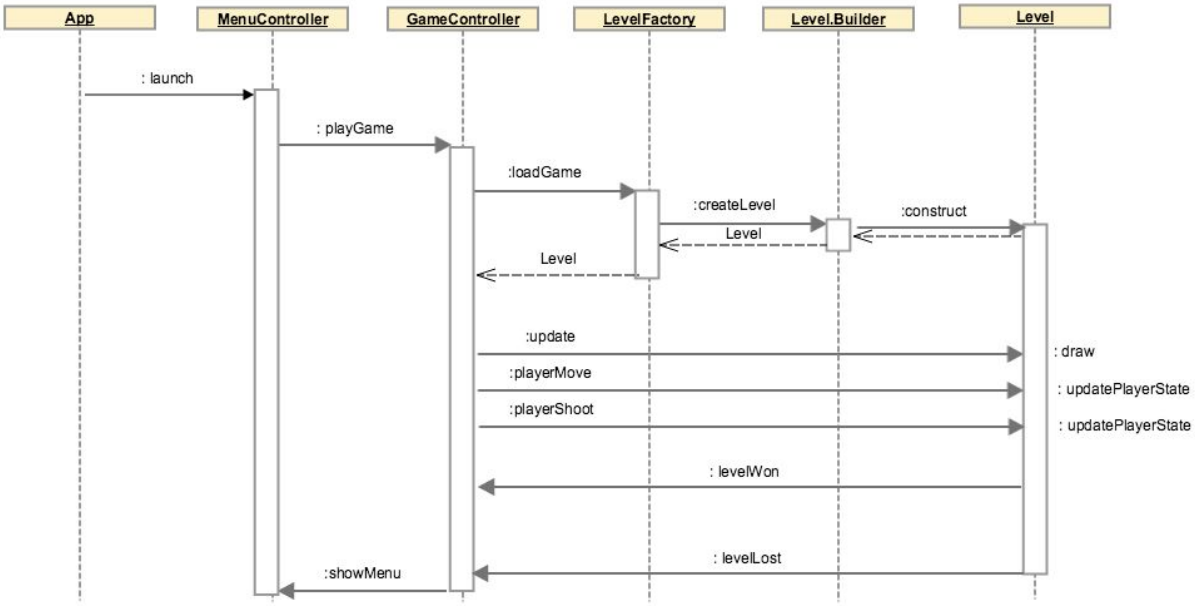
We consider most of the other, non-main classes as less important because they mainly consist of the Model and do not contain much logic or graphic elements. They are mainly objects with only some attributes, getters and setters and one or two other methods, for example to move them. They cannot be merged or deleted though, because we need these different objects to e.g. distinguish Walls from Balls.

There are some classes though, which are not yet used at all. BallController, AbstractProjectile, PowerUp, TimePowerUp, AbstractGame, SinglePlayerGame and MultiPlayerGame are currently not used. But we want to keep some of them to be able to easily implement further functionalities such as powerups. Ballcontroller, SinglePlayerGame, AbstractGame and MultiPlayerGame are the only classes that can be deleted because they are empty and not needed for further extension of the game.

4) UML Diagram of important classes and their connections



5) Sequence diagram



DooB CRC

Requirements specification

Game Controller	
Superclass: -	
Subclasses: -	
Core game logic & data	Game Object
Display game board to user	Game View

Menu Controller	
Superclass: -	
Subclasses: -	
Trigger other controllers based on options	Game Controller
Show menu to user	Menu View
Show high-score to user	High-score View
Start the application	

Settings Controller	
Superclass: -	
Subclasses: -	
Show settings to user	Settings View
Change game parameters in Properties file	

Game Object	
Superclass: -	
Subclasses: SinglePlayerGame, MultiPlayerGame	
Represents levels in game	Level Object
Represents players in game	Player Object

Level Object	
Superclass: -	
Subclasses: -	
Current position and state of balls in game	Ball
Represents player playing the level	Player
Borders of the level	Wall
Powerup logic and management of thi level	PowerUp

Player Object	
Superclass: -	
Subclasses: -	
Player movement	
Player state (score, location, life)	
Keep track of the current projectiles	Projectile

Projectile	
Superclass: -	
Subclasses: Spike	
Collision detection with other collidables	Ball, Wall

PowerUp	
Superclass: -	
Subclasses: -	
Influence level state for player	Level
Create different types of powerups	

Ball	
Superclass: -	
Subclasses: -	
Collide with projectiles, walls and players	Projectile, Wall, Player

Wall	
Superclass: -	
Subclasses: -	
Collide with other collidables	Player, Ball, Projectile

Spike	
Superclass: Projectile	
Subclasses: -	

Level Factory	
Superclass: -	
Subclasses: -	
Build level objects	Level

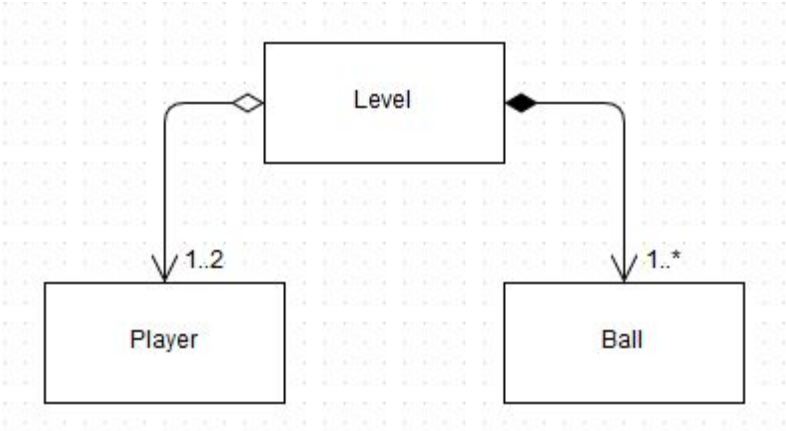
Game Factory	
Superclass: -	
Subclasses: -	
Create game objects	Game

Exercise 2

1) Aggregation and composition

The difference between aggregation and composition is that while they both describe a “has-a” relationship, composition implies ownership. This means that if the owner part of a composition relationship gets deleted, so does the owned part of the relationship. A good example for this is the relationship between “Document” and “Page”: A document is composed of pages and when the document gets deleted, so do the pages.

We use a composition relationships between Level and Ball and a aggregation relationship between Level an Player like so:



A level has 1 or 2 players, but they get carried over between levels. A level has 1 or more balls, however once a level gets deleted, so do all the balls the level contains. The level has ownership of the balls but not of the players.

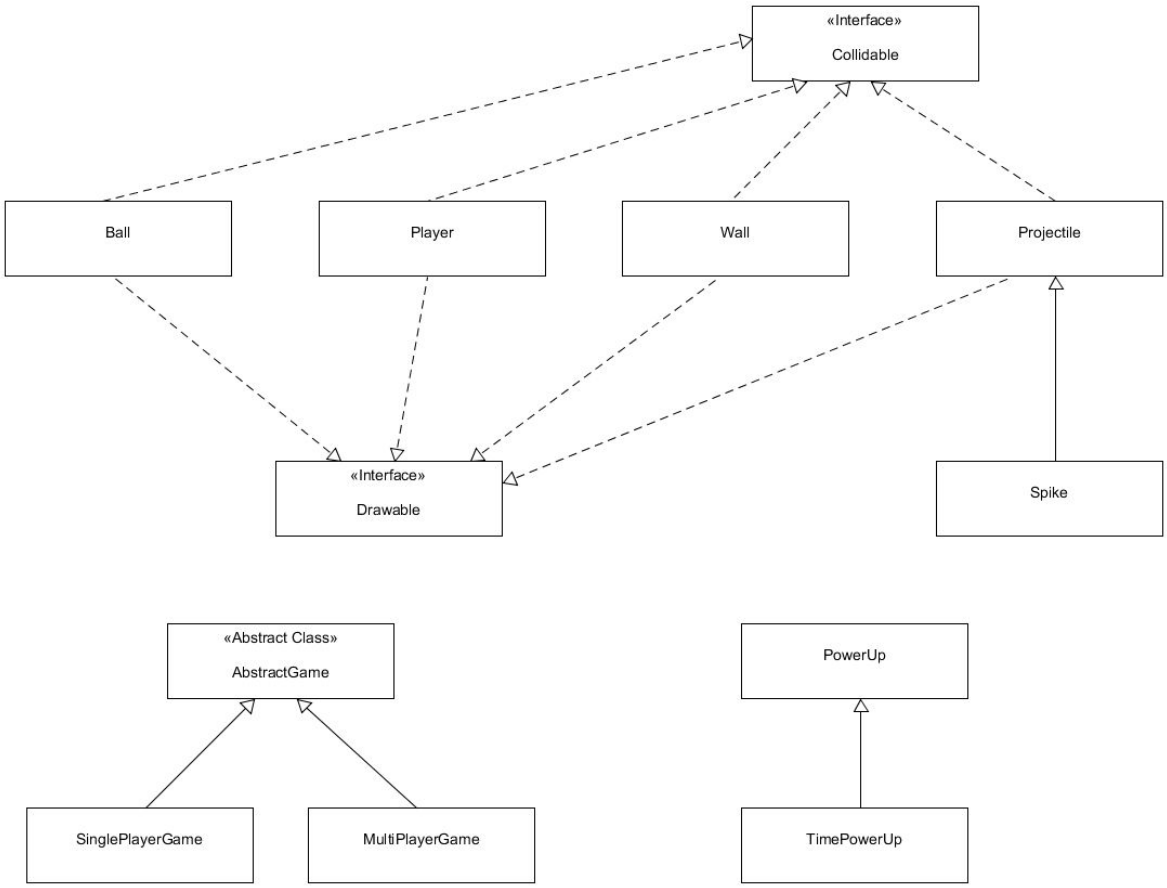
2) Parameterized classes

We don’t use parameterized classes.
Parameterized classes should be used in UML when they are used in the source code and the parameter has influence on the UML. For example, let’s say a parameterized class is used for a list, in which the parameter stands for what kind of objects the list holds. The parameter then in turn decides what kind of object will return when using a get() function or what kind of object should be given with an insert(Object o) function.
So when using a parameterized class in UML, it is easy to see what functions the Parameter has effect on and how it effects those functions.

3) Hierarchie

‘Projectile’ and ‘Spike’ have a “is-a” relationship, a spike is a projectile. Inheritance is used in this case for scalability, so that it is easy to make different types of projectiles. Even though at the time there is only the spike-projectile, others could be added.
‘AbstractGame’, ‘SingePlayerGame’ and ‘MultiPlayerGame’ are a form of polymorphism. Inheritance is used here because ‘SingePlayerGame’ and ‘MultiPlayerGame’ share most attributes, except for the amount of players.
‘PowerUp’ and ‘TimePowerUp” also have a “is-a” relationship. Inheritance is used here for scalability. This way we can easily add different types of power-ups.
‘Ball’, ‘Player’, ‘Wall’ and ‘Projectile’ all implement the interfaces ‘Collidable’ and ‘Drawable’ as they all need a set of functions in order to compute collisions and draw them on screen.

We believe there aren’t any hierarchies that should be removed. All the hierarchies present are of good use and are not being used for unnecessary code reuse.



Exercise 3: Logging

DooBLog	
Superclass:	
Subclasses:	
Can write to log file	File specified in doob.properties
Can empty log file	