Assignment 1

Alex Geenen, Bart de Jonge, Mark van de Ruit, Menno Oudshoorn, Stefan de Vringer

Software Engineering Methods 2015 - Group 1

Exercise 1.1: responsibility driven design

In this exercise, we start from our requirements (without considering our implementation) to derive, step by step, all the classes, responsibilities and collaborations of our game.

1. Deriving the classes, responsibilities and collaborators from the requirements.

Our game is composed of different 'states'. For example: a state for the main menu, the settings screen, the main game itself, the pause screen and the end game screen. We now have separate classes for these states, each of which derives from the Slick2D library class BasicGameState. These classes are called StartState, GameState, SettingsState, PausedState and GameOverState. They will be the foundation of our game.

Next up, we require a player - something we can control and move through the environment. For this we derive a Player class.

Another group of vital elements of our game is the bubbles that you, the player, have to shoot. For these bubbles, we have a Bubble class, deriving from the more generic Circle class of the Slick2D library.

To contain the player and the bubbles, we require walls, a floor and a ceiling. For this, we use a class called Rectangle, which we use to create rectangles in all shapes and sizes in different locations in the game.

Bubbles in our game can be shot with a weapon, so we now derive a Weapon class. This class will keep track of the current type of weapon, and will handle interaction between said weapon and the bubbles.

For implementing multiple levels, we require two classes. Level, to keep track of all things in a level itself, and levelContainer, which stores all the levels in the game and initializes them when necessary.

Because we wish to keep track of player highscores, we have derived three classes. Score, which will keep track of a single score and attaches a name to said score. HighScores, which in turn combines all these scores to form a top 10 sorted list. And finally HighScoresParser, which will write and read to/from the highscores file, so that highscores can be saved for later.

Another feature among our requirements was the addition of gates to certain levels. For this, we have a Gate class. This means the Level class will now also keep track of the gates present in a level.

We wish to also implement some kinds of powerups for the player to use. For this, we create a Powerup Class. Powerup will be able to modify certain aspects of the game, such as the Weapon, to give the player an advantage. A subclass of Powerup will be called Coin. This is a special powerup which grants the player extra points, and nothing else.
There are a few more relatively small classes that may be necessary. To keep track of a player's points during the game, we have derived a Points class. To keep track of a player's lives during the game, we

have derived a Lives class. To keep track of timers and remaining time during a level, we have derived a CountDown class.

Because one of the requirements is multiplayer, multiple Players should be possible. Thus, we have derived a PlayerList class, which can keep track of multiple players.

To make our game look stunning, we have an Image class which can overlay images. Character is a special type of Image, which will represent the player's image.

And finally, to bind everything together, we use a class called MainGame. This class will initialize and start all necessary parts to get a game running, and handles Slick2D boilerplate code.

Below we show an overview of all the currently derived classes. Furthermore, we have derived a set of all the CRC cards for all these classes. These are displayed after part two of this exercise, as they take up multiple pages.

Requirement	**Class**	**Important?**
Start Menu/Main menu	StartState	Yes
Settings menu	SettingsState	Yes
Game over screen	GameOverState	Yes
A player entity	Player	Yes
Implement highscores	HighScores, Score	No
Bubble go bouncy	Bubble	Yes
Parse highscores to file	HighScoresParser	No
Game Screen	GameState	Yes
Clickable buttons	Button	No
Gates in levels	Gate	No
Weapons to shoot bubbles	Weapon	Yes
Multiple levels	Level, LevelContainer	Yes, No
Coin powerups	Coin	No
Powerups	Powerup	No

Pause	PauseState	No
Floors, walls, ceilings	Rectangle	Yes
Lives	Lives	No
Countdown	Countdown	No
Points	Points	No
Different Characters	Character	No
Assets	Image	No
Multiplayer	PlayerList	No
Overall Game + main method	MainGame	Yes

For the CRC cards that we derived from the requirements, see below

2. Differences between the CRC cards and the existing implementation

Our current implementation has the following additional classes: RND, FloatingScore, InstantLaser, MyButton, MyRectangle, ScoreComparator, Spiky & WeaponList.

MyRectangle and MyButton are implemented to facilitate testing. MyRectangle extends the Rectangle class from the Slick2D library, which did not, sadly, have an equals() method. The MyButton class, however, seems to be completely unnecessary as it can be incorporated into the Button class, which we implemented.

The ScoreComparator class consists of a single method (which simply compares two score objects) and could and should be incorporated into the Score class.

The InstantLaser and Spiky classes are now types of Powerups, besides Coin, and hence have their own classes.

The WeaponList class is used to store and update the weapons on screen. In the crc cards we have allocated this task to Weapon, but as there can be multiple weapons due to multiplayer, this is a very clean solution.

The FloatingScore class is used to create a small animation for floating text in the game. It did not follow from any of the requirements, and hence is not represented in the crc cards beow.

RND is a class that takes responsibility for drawing all of the elements on the screen and was not included in the crc cards because we did not think it necessary to separate it. It did not follow from any of the original requirements. However, due to what it does, it saves us a lot of repetitive code and is a welcome addition.

Now we arrive at the classes that our implementation does not contain, but which were specified in our crc cards: PauseState, Lives, Countdown, Points & Character.

Lives, Countdown, Points & Character are all not implemented due to the fact that they can be assigned as attributes to other classes. Lives is an attribute of GameState, just like Countdown and Points. Instead of Character we have just used Image, a Slick2D class, as an attribute of the Player class.

The PausedState class has not been implemented, because it would cause unnecessary overhead for simply pausing/unpausing the game. The player wishes to continue exactly where he paused the game, without any signs of slowdown. Switching to another state away from GameState, and then back, would mean storing all the level data and placing everything back where it was.

3. The CRC cards

On the next few pages we've stored all the crc cards for all the classes we derived for this exercise. The rest of the exercises continues after the next few pages.

Class Name: StartState	
Superclass: BasicGameState	
Subclasses: none	
Responsibilities	**Collaborators**
Is showed when starting the game	MainGame
Move to actual game	Button, GameState
Move to settings screen	Button, SettingsState
Exit the game	Button

Class Name: SettingsState	
Superclass: BasicGameState	
Subclasses: none	
Responsibilities	**Collaborators**
Change character	Character, Player, Button
Return to the main menu	Button, StartState

Class Name: GameOverState	
Superclasses: BasicGameState	
Subclasses: none	
Responsibilities	**Collaborators**
Enter Highscore	HighScores, Score, HighScoresParser, Button
Exit game	Button
Play again	GameState, Button
Main menu	StartState, Button

Class Name: Player	
Superclass: none	
Subclasses: none	
Responsibilities	**Collaborators**
Shoot laser	Weapon
Pick up powerups	Powerup
Intersect with circle	Bubble
Intersect with gate	Gate
Intersect with walls	Rectangle
Move through the environment	GameState
Assign a character	Character

Class Name: HighScores	
Superclasses: none	
Subclasses: none	
Responsibilities	**Collaborators**
Keep a list of sorted highscores	Score
Keep the list after you close the game	HighScoresParser

Class Name: Score	
Superclasses: none	
Subclasses: none	
Responsibilities	**Collaborators**
Record a score linked to a game	GameOverState
Store the score in the highscores (if high enough)	HighScores

Class Name: Bubble	
Superclasses: none	
Subclasses: none	
Responsibilities	**Collaborators**
Interact with walls, gates & player	Rectangle, Gate, Player
Split when shot (or disappear	Weapon

Class Name: HighScoresParser	
Superclasses: none	
Subclasses: none	
Responsibilities	**Collaborators**
Write and read the highscores to a file	HighScore

Class Name: GameState	
Superclasses: none	
Subclasses: none	
Responsibilities	**Collaborators**
Manage collisions	Player, Gate, Rectangle, Bubble, Weapon
Keep track of lives	Lives
Keep track of player death, switch to necessary state	GameOverState, Lives
Spawn powerups	Powerup
Show pause menu when game is paused	PauseState
Keep track of time left	Countdown
Keep track of points/coins	Points, Coin
Keep track of levels	Level, LevelContainer

Class Name: Button	
Superclasses: none	
Subclasses: none	
Responsibilities	**Collaborators**
Switch states	GameState, SettingState, StartState, GameOverState, PauseState
Confirm score	Score, HighScores, HighScoresParser

Class Name: Gate	
Superclasses: Rectangle	
Subclasses	
Responsibilities	**Collaborators**
Open when certain bubbles are shot	Bubble
Interact correctly with player(s) and Bubbles	Bubble, Player

Class Name: Weapon	
Superclasses	
Subclasses	
Responsibilities	**Collaborators**
Be the correct weapon after powerup	Powerup
Handle collisions with bubbles	Bubble
Handle collisions with ceiling	Rectangle

Class Name: Level	
Superclasses: none	
Subclasses: none	
Responsibilities	**Collaborators**
Keep a list of balls in the level	Bubble
Keep a list of gates in the level	Gate
Store the time that can be used in this level	Countdown

Class Name: LevelContainer	
Superclasses	
Subclasses	
Responsibilities	**Collaborators**
Create levels	Level

Class Name: Coin	
Superclasses: PowerUP	
Subclasses: none	
Responsibilities	**Collaborators**
Give points	Points
Disappear if picked up by player	Player

Class Name: Powerup	
Superclasses: none	
Subclasses: Coin	
Responsibilities	**Collaborators**
Interact correctly with other objects in the level	Player, Rectangle
Give the correct effect when picked up	Player, Weapon

Class Name: PauseState	
Superclasses: BasicGameState	
Subclasses: none	
Responsibilities	**Collaborators**
A way to return to the game	Button, GameState
A way to quit the game	Button
A way to go back to the main menu	Button, StartState

Class Name: Rectangle	
Superclasses	
Subclasses: Gate	
Responsibilities	**Collaborators**
Interact correctly with the other components in a level	Rectangle, Gate, Player

Class Name: Lives	
Superclasses: none	
Subclasses: none	
Responsibilities	**Collaborators**
Stop the game if there are no more lives	GameOverState
Deduct a life when a player is hit by a bubble	Bubble, Player
Deduct a life when the time is up	CountDown

Class Name: Countdown	
Superclasses: none	
Subclasses: none	
Responsibilities	**Collaborators**
Deduct a life if there is no more time left	Player, Lives
Give bonus points for completing a level with time left	Level, Countdown, Points

Class Name: Points	
Superclasses: none	
Subclasses: none	
Responsibilities	**Collaborators**
Register events that reward points	Bubble, Coins, Countdown
Have points contribute to the score	Score

Class Name: Character	
Superclasses: Image	
Subclasses: none	
Responsibilities	**Collaborators**
Display correctly on the players	Player

Class Name: Image	
Superclasses: none	
Subclasses: Character	
Responsibilities	**Collaborators**
Be a visual asset to the game	GameState, GameOverState, StartState, SettingsState, PauseState

Class Name: PlayerList	
Superclasses: none	
Subclasses: none	
Responsibilities	**Collaborators**
Keep track of all the players in the game	Player

Class Name: MainGame	
Superclasses: none	
Subclasses: none	
Responsibilities	**Collaborators**
Create a screen to play in	GameContainer
Initialize states	GameState, PausedState, SettingsState, GameOverState, StartState
Initialize levels	LevelContainer
Load images	Image
Load constant parameters used in the game	

Exercise 1.2: Main classes of implemented project

Below are listed what we consider to be the 'main' classes of our previously implemented project, as well as their responsibilities and collaborations.

This means that functioning of the game is heavily dependent on these classes. If any of these classes were to currently drop out, the game as a whole would cease to function, not just from a programming perspective, but also functionality-wise.

Furthermore, for each class, a short description is given in terms of the responsibilites and collaborations of this class.

MainGame

The responsibilities of the MainGame class are as follows:

- Create a screen for the game to run in.
- Initialise states.
- Initialise levels.
- Store widely reused images.
- Store widely reused constant parameters used in the game

This means that the MainGame class is the central anchor of the game. It contains most of the boilerplate code necessary for running even the simplest of applications, but, more importantly, stores most of the parameters and images reused throughout the game.

The collaborations of the MainGame class are as follows:

- GameContainer: a Slick2D class MainGame depends on heavily for the creation of the application.
- All state classes: Without state classes, the application would have no content. As the MainGame is used to create one, it collaborates with state classes to achieve its purpose.
- LevelContainer: MainGame collaborates with the LevelContainer class, as it cannot load in levels (which are necessary for a simple game) on its own.
- Image: the MainGame class stores most of the images that the game uses in multiple places for efficiency, where they are easily accessible.

The MainGame class collaborates mostly with Slick2D classes to set up a running application. Some of our implemented classes come in here as well, to take responsibilities (such as storing levels) off its load.

GameState

The responsibilities of the GameState class are as follows:

- Update most of the game's objects.
- Manage collisions between most objects, using those objects' classes.
- Spawn powerups and coins.
- Show the pause menu when the game is paused.
- Keep track of remaining time.
- Keep track of the current score.
- Keep track of the current level.

This means that the GameState class is mainly responsible for managing gameplay. Most of the code in it makes sure everything in the game keeps running, moving and calculating. It manages an update loop, as well as a render loop, both of which constantly re-run to update all objects in the main game.

The collaborations of the GameState class are as follows:

- Player: the GameState does not by itself manage the game's player. It does, however, tell the player when to update.
- Powerup: the GameState creates and places powerups. Once this is done, the Powerup class itself manages these powerups.
- Coin: the GameState creates and places coins. Once this is done, the Coin class itself manages these coins.
- GameOverState: the GameState handles control of the game to the GameOverState the moment it considers the player 'dead'.
- Level, LevelContainer; the GameState may display the levels, but they are stored in LevelContainer, and their data in Level. If these classes did not exist, GameState would become a behemoth of a class.

The GameState class collaborates mostly with gameplay-specific classes. Most of its collaborators are classes that represent objects displayed and updated in GameState.

Player

The responsibilities of the Player class are as follows:

- Shoot the laser.
- Pick up powerups when it intersects with them.
- Keep track of intersections with circles.
- Keep track of intersections with gates
- Keep track of intersections with walls
- Move the player through the environment
- Assign a character to the player

This means that the Player class is mainly responsible for managing the player and its interactions with the environment around it.

The collaborations of the Player class are as follows:

- Weapon: The player manages the different weapon types, one of which is standard and others that can be obtained via powerups. When a shot is fired, the Player passes its current weapon to the GameState to animate/manage.
- Powerup: The player checks if it has intersected with a powerup. If so, the powerup's ability/weapon is added to the player and is managed accordingly, as well as removing the Powerup from gameplay via the GameState.
- Coin: The player checks if it has intersected with a coin. If so, the player's score is incremented in the GameState and the coin is removed from play.
- Gate: The player checks to see if it has intersected with a gate, if so it can no longer move in that direction (as a gate is in the way)
- GameState: The player communicates with the GameState to check on in-game items and to adjust the environment based on the player's actions or to restrict the player's actions.
- SpriteSheet: The player uses the SpriteSheet to display images in differing states of animation quickly and efficiently. It can load in a single image, instead of 5 separate images, for a single animation.

Level

The responsibilities of the Level class are as follows:

- Store and keep track of all circles in the level
- Store and keep track of all gates in the level
- Store and keep track of remaining time for this level.

This means that the level class is mainly a storage class that holds lists of all objects that should be displayed in a level. It allows other classes to access this storage easily.

The collaborations of the Level class are as follows:

- BouncingCircle: the level class directly stores these objects. If this class did not exist, it would have to store far more information and could quickly become a behemoth of a class. It requires the BouncingCircle class to function properly.
- Gate: for the same reasons as for the BouncingCircle class, the Level class stores Gate objects, not information about gates. It requires the Gate class to function properly.

Weapon

The responsibilities of the Weapon class are as follows:

- Manage the weapon’s properties, such as speed, width and shape
- Update the weapon’s position after firing.

This means the Weapon class, while small, is responsible for the manaing of one of the main features of the game. Namely, the laser. It keeps track of and updates a wide variety of values concerning the weapon’s properties, and makes these easily accessible to other classes/objects.

The collaborators of the Weapon class are as follows:

- BouncingCircle: the Weapon class itself does not handle any of the collisions between the laser and circles. The GameState class does most of the heavy lifting here, using values supplied by the Weapon class and the BouncingCircle class.
- Powerup: the Weapon class itself does not handle any of the powerups and their effects. Its properties are modified externally by powerups, instead.

BouncingCircle

The responsibilities of the BouncingCircle class are as follows:
- Manage the circle's properties, such as speed w.r.t. gravity
- Return splitted circles when the circle is going to be destroyed

This means that the logic is reasonably small, as it is responsible for its own physics, and for game features when it is about to be destroyed.

The sole collaborator of the BouncingCircle class is the GameState class, which the BouncingCircle uses for physics calculations. The GameState uses BouncingCircle for environment actions and to get the circles that are the result of a "split" circle.

Exercise 1.3: less important classes of implemented project

In this exercise, we reflect upon the less important classes of our implementation.

The classes which weren't classified as important in the previous question were excluded because they weren't essential to the core gameplay. The definition of core gameplay consisted of being able to play a single level with the main idea and requirements of Bubble Trouble implemented. This would be a game with a player that could shoot splitting bubbles, and nothing else. Most of the classes described in this document serve to fill up the rest of our game.

Per class, we will reflect on their responsibilities, to see what could be changed, merged or removed. If we find classes for which this counts, code changes will be performed. If not, an explanation will be given.

Button
The Button class is used throughout the game to represent clickable elements. It holds a large amount of the responsibility concerning user input, and will not be removed. The MyButton class, however, could be merged into the Button class, as it simply adds a single method.

Coin
The coin class is only used to add extra functionality. It is used for coins that spawn in the game while shooting bubbles, which grant you some extra points upon pickup. This is definitely not important for core gameplay, but just a nice extra feature. We could possibly merge this class with Powerup, because its dropping functionality is similar, but we want to keep a clear distinction between things that grant you points (Coin) and things that give you a special ability (Powerup)

FloatingScore
The FloatingScore class is used to make our game more dynamic, but is not actually needed to play the game, which is why we have considered it a less important class. This class is used to display the score you get when you shoot a bubble, floating near the bubble. We do not want to omit this class because it has a specific responsibility that we don’t want to put anywhere else.

Gate
The Gate class is, as far as we’re concerned, quite an important class because it adds really nice functionality, but because this class can be omitted without affecting the ability to actually play the basic version of the game, we still put it as a “less important class”. However, we do not want to delete this class, because it has a very specific responsibility that is important to making our game a lot more fun

HighScores
You can still play our game without high scores, which is why we haven’t considered the HighScores class, which keeps track of the high scores in the game, as a core class of our game. However, we definitely want to keep this class, because it keeps all the highscores nicely in one place.

HighScoresParser
This class takes care of writing and reading the high scores to file. We haven’t considered it one of our core classes because, just like HighScores, high scores are not a core element of the game, but just added functionality. We could have added the static methods in this class to the HighScores class, but we wanted to keep the in-game high scores and the communication with the “outside” filesystem separated for a clear division, which is why we won’t merge this class with HighScores

InstantLaser
InstantLaser is a subtype of the Weapon class. Because we wanted to make a clear distinction between the different types of weapons, we have created the InstantLaser and the Spiky class. We could have implemented these in the Weapon class itself, but this would probably make the Weapon class a lot more difficult to understand, because you would get a very high cyclomatic complexity. This is why we chose to use this inheritance design.

LevelContainer
While a simple game could do with just a small set of levels, a game with, say, 10 or more levels would soon become a behemoth of code. In order for us to efficiently store a large set of levels, a LevelContainer was required. It will not be changed.

MyButton
This class simply extends the Button class, and overrides the equals and hashcode methods, and adds a serialVersionUID. These things don’t actually override anything in the Button class, so these changes can be incorporated into Button and MyButton instances can be replaced with Button instances.

MyRectangle
The MyRectangle class has little responsibility of its own. It is required for some tests, however, as the class it extends does not and can not support an equals() method.

PlayerList
The PlayerList class is less important because it is not necessary for a simple game. However, it has many responsibilities, because it is required for multiplayer. No other class could serve its functions

Powerup
The Powerup class serves to expand our game by introducing an extra new gameplay mechanic. Its responsibilities include keeping track of and creating new powerups for the player. Due to its pivotal function for the powerup mechanics, it will not be removed.

Score
The Score class serves to store score objects. It was implemented so we could easily combine game scores with player names, before writing all of this into a file. While it could easily function as a part of the HighScores class, we shall not merge them because it would make no sense from a Responsibility Driven Design point of view.

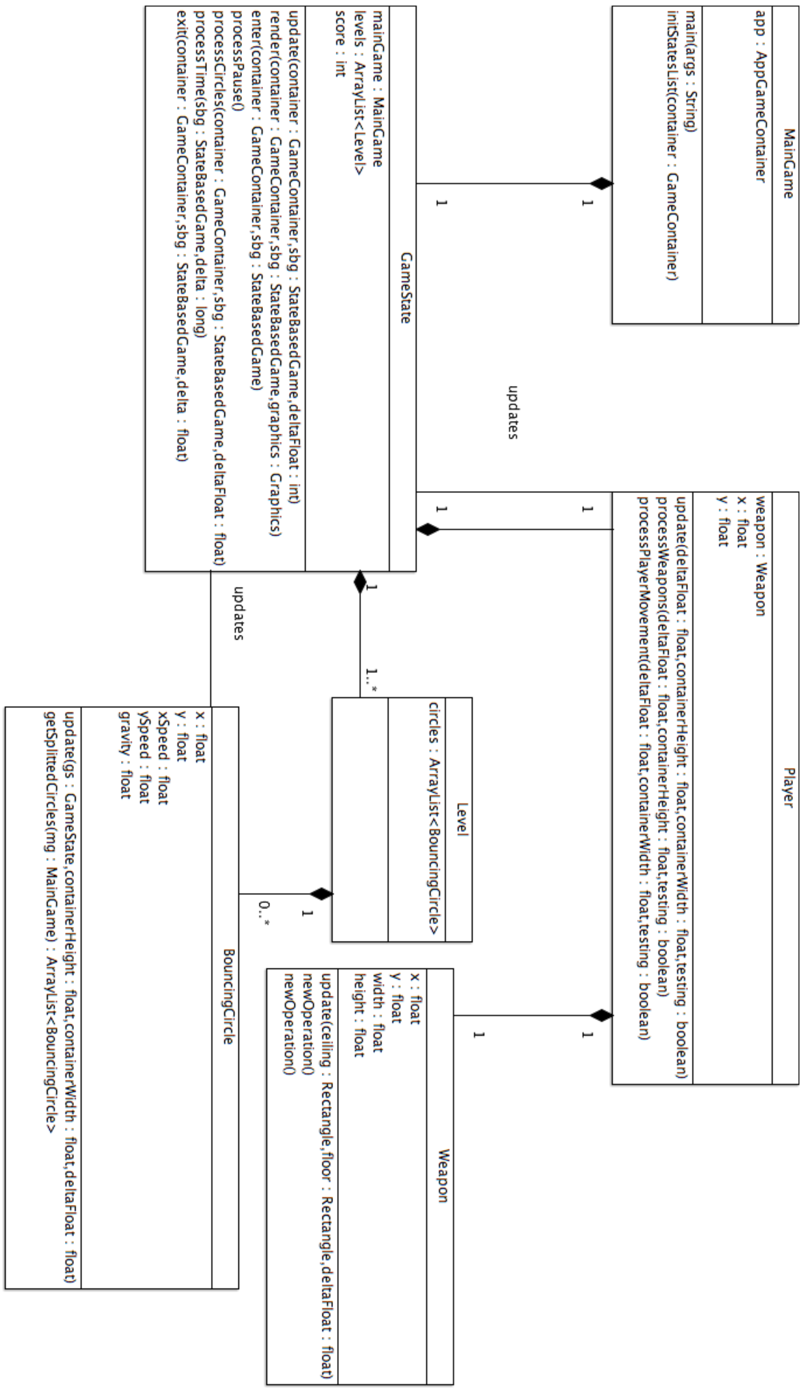
ScoresComparator
ScoresComparator is a class that implements the Comparator interface and is used to compare two scores (to sort the highscores). We didn't consider this one of our core classes because we could easily do without, actually, we have really deleted this class and made the compare method one of the Score class itself.

Spiky
Spiky is a subtype of the Weapon class just as InstantLaser is. It shall not be merged for the same reasons as with its brother, the InstantLaser class.

WeaponList
The WeaponList class is used because we are having multiple possible weapons in our game. We didn't consider this class as important because more weapons are part of powerups and powerups are extra functionality rather than a core element of the game. We could, in theory, omit this class, and put the weapons in the Player class (because each player can only have one weapon at the same time). But because this class is already heavily integrated, we chose not to do that at this time. We might do it in the future

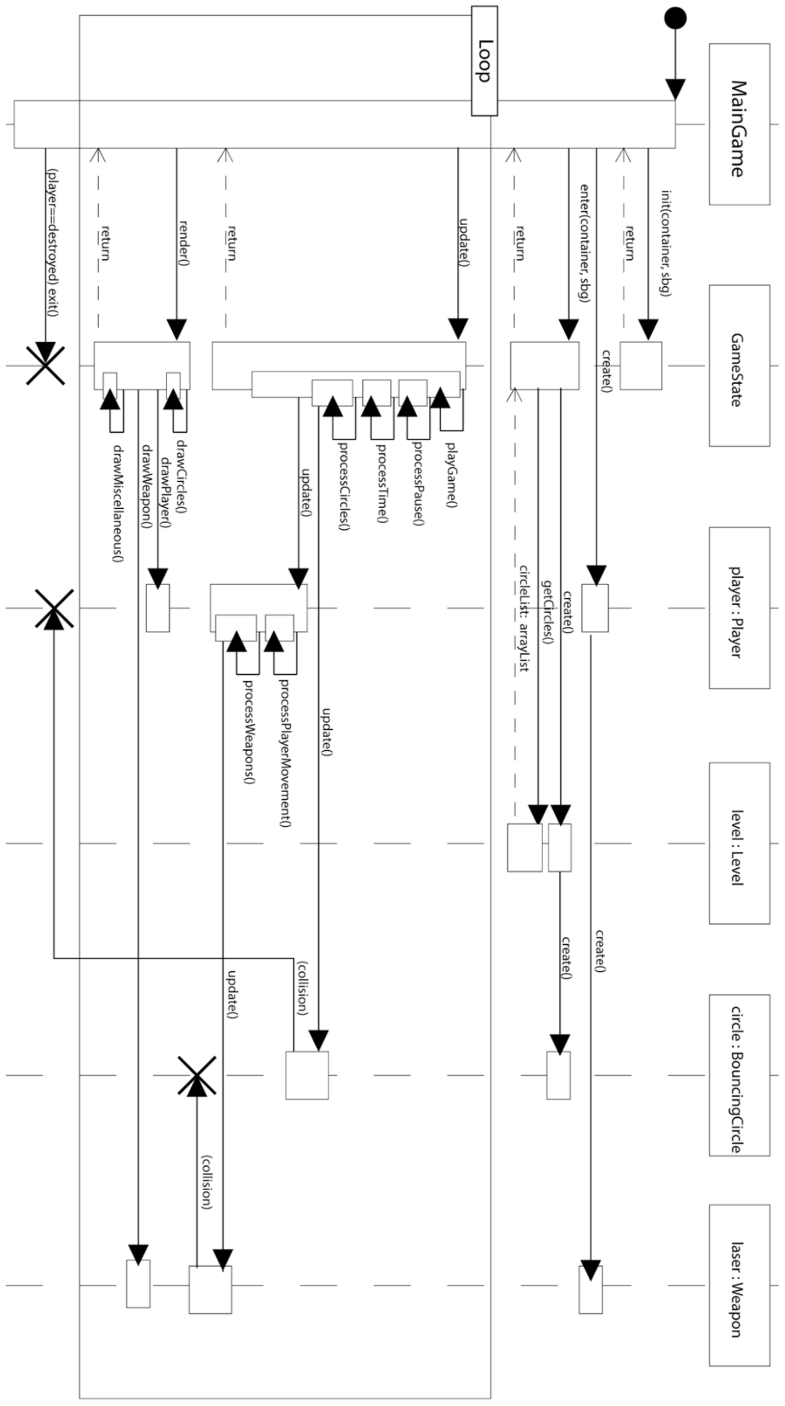
Exercise 1.4: Class diagram of main elements

Below we show a class diagram of the main elements of our game. This means that classes such as Powerup are not incorporated. **This diagram is also included separately.**


MainGame
app : AppGameContainer
main(args : String)
initStatesList(container : GameContainer)
GameState
mainGame : MainGame
levels : ArrayList<Level>
score : int
update(container : GameContainer,sbg : StateBasedGame,deltaFloat : int)
render(container : GameContainer,sbg : StateBasedGame,graphics : Graphics)
enter(container : GameContainer,sbg : StateBasedGame)
processPause()
processCircles(container : GameContainer,sbg : StateBasedGame,deltaFloat : float)
processTime(sbg : StateBasedGame,delta : long)
exit(container : GameContainer,sbg : StateBasedGame,delta : float)
Player
weapon : Weapon
x : float
y : float
update(deltaFloat : float,containerHeight : float,containerWidth : float,testing : boolean)
processWeapons(deltaFloat : float,containerHeight : float,testing : boolean)
processPlayerMovement(deltaFloat : float,containerWidth : float,testing : boolean)
Level
circles : ArrayList<BouncingCircle>
BouncingCircle
x : float
y : float
xSpeed : float
ySpeed : float
gravity : float
update(gs : GameState,containerHeight : float,containerWidth : float,deltaFloat : float)
getSplittedCircles(mg : MainGame) : ArrayList<BouncingCircle>
Weapon
x : float
y : float
width : float
height : float
update(ceiling : Rectangle,floor : Rectangle,deltaFloat : float)
newOperation()
newOperation()
updates
updates
1
1
1
1
1
1..*
1
0..*
1
1

Exercise 1.5: Sequence diagram of main elements

Below we show a sequence diagram of the interaction between main elements of our game. Classes such as Powerup are not incorporated. **This diagram is also included separately.**



Exercise 2.1: Aggregation and composition in our project

Aggregation and Composition are both types of association.

An aggregation relation means that there is one object that consists (partially) of another object. The relation means that there is a reference between the two classes and the latter object may be shared with other classes.

A composition relation means that there is one object type that contains another object. The relation means that the former implements the latter. The second object is only relevant in light of the first, as it is part of a greater whole.

In our implementation, aggregation and composition can be found in multiple locations, which we will discuss one by one below.

State-specific elements
Button and TextField are instantiated as part of a state which again is to be displayed as part of the window. These elements are useless and nonexistent outside of their respective states. This marks a composition. The same goes for Input, which only allows input in the state it is created in.

Lists
HighScores keeps track of the different Score objects created. This is an aggregation relation, as Score objects are created - and exist - separately. This is the same relation as WeaponList has with Weapon, and PlayerList with Player.

LevelContainer and Level share a similar relation, but on top of this, all levels are hardcoded to be initialized in the LevelContainer itself.

MainGame
The class MainGame has a composition relationship with all of the state classes (GameState, GameOverState, SettingsState & StartState). The state classes are created and displayed by means of the MainGame class. A PlayerList is also constructed inside the MainGame class as a part of the game, which makes this another composition relationship.

GameState
Powerup and its subclasses are in a compositional relation with GameState. GameState controls when Powerups are dropped during the game and when they disappear. As a result, powerups are only relevant in the context of GameState. LevelContainer and WeaponList are the same, also only relevant while the game is progressing and therefore only relevant in GameState.

FloatingScore is in a composition relation with GameState, it is an animation constructed in GameState and it can only active when GameState is the state used to display the window.

SettingsState
Some instantiations of Rectangle are in a composition relation with SettingsState. This is when Rectangle objects are used as another type of window element.

Level
BouncingCircle and Gate are only relevant when loaded into GameState as part of a level, which means a composition relation is present.

Exercise 2.2: Parameterized classes

There are no parameterized classes in our source code, mainly due to the fact that there are not many classes that repeat functionality with different types of inputs or types.

Parameterized classes are useful when there are a bunch of classes doing the exact same thing, while operating on different types. When a parameterized class is used instead, all of the classes with identical functionality can be abstracted away to a single generic class which can then be bound to the different types either explicitly, or implicitly.

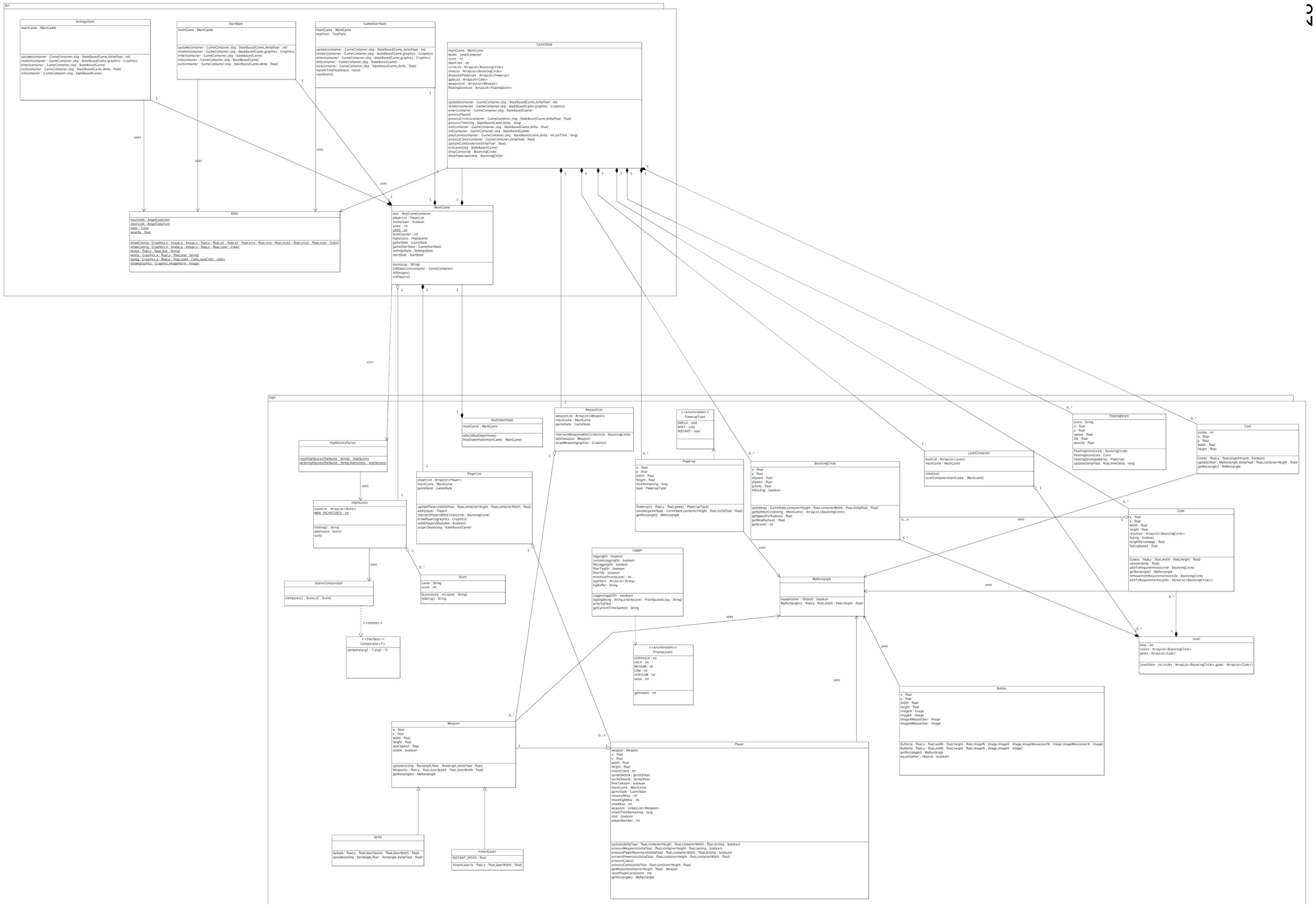
This saves space in the UML, reducing multiple models to a single one, from which classes with specific types are derived. In practice, this promotes the reuse of code and reduces the amount of repetitive code in your code base.

Exercise 2.3: Class diagram for all hierarchies

Looking at our UML class hierarchies, we simply don’t see why any of those hierarchies should be removed.

All hierarchies of containment are clear (such as aggregation hierarchies as LevelContainer and Level, or composition hierarchies such as MainGame and GameState). Removing these hierarchies would mean the class diagram would not represent our source code anymore. Furthermore, we have some inheritance hierarchies which are important to show, and some “uses” associations (to MyRectangle for example), which are important to show which classes use some kind of rectangle to represent themselves in the game.

The enormous monstrosity of a class diagram itself is shown on the next page. **Because of its size, it is also included separately next to this document as the file exercise2.3.png!**


gui
SettingsState
StartState
GameOverState
GameState
RND
MainGame
logic
ShutDownHook
WeaponList
<<enumeration>> PowerupType
FloatingScore
Coin
HighScoresParser
PlayerList
Powerup
BouncingCircle
LevelContainer
HighScores
Gate
Logger
Score
ScoresComparator
MyRectangle
<<interface>> Comparator<T>
<<enumeration>> PriorityLevels
Level
Button
Weapon
Player
Spiky
InstantLaser
uses
<<realize>>

Logger Requirements Document

On this page we show the requirements document for our logger functionality.

Functional Requirements

Must Haves

- Logger shall log all visible actions
- Logger shall log all state transitions
- Logger shall log all other important backend actions
- Logger shall print to console
- Logger shall write log to file
- Logger shall include a timestamp for every log action
- Logger shall log all errors
- Logger console prints shall be able to be turned on and off
- Logger file writes shall be able to be turned on and off

Should Haves

- Logger should include a predetermined priority level with every log action
- Logger should be able to only log things that meet a predetermined minimum priority level (i.e. verbosity)

Could Haves

- Logger could be able to include a tag specifying the origin of the log
- Logger could be able to only log things that have (a) certain tag/tags

Won’t Haves

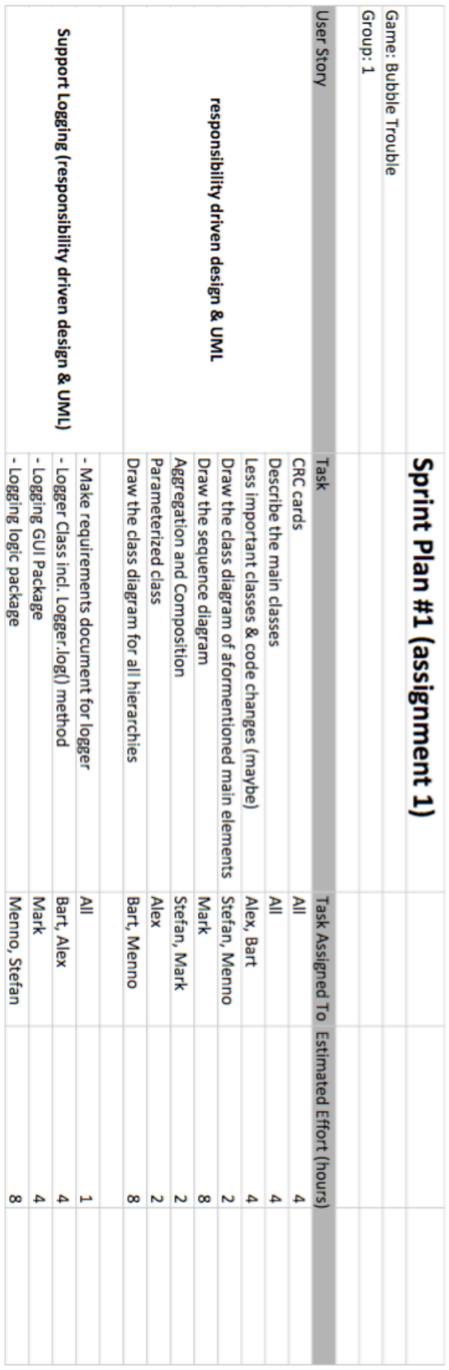
- Logger won’t be able to manipulate existing logs
- Logger won’t have colored output

Non-functional requirements

- The logger shall be tested with a minimum of 75% line coverage
- The logger shall be written using Java 1.8
- The logger shall be delivered on Friday, the 18th of September
- The logger shall be built in a single SCRUM sprint

Sprint plan and user story

On this page we show the sprint plan for this sprint. Furthermore, down below we show the user story for the logger we developed during this sprint.



Sprint Plan #1 (assignment 1)

Game: Bubble Trouble
Group: 1

User Story	Task	Task Assigned To	Estimated Effort (hours)	
responsibility driven design & UML	CRC cards	All	4	
	Describe the main classes	All	4	
	Less important classes & code changes (maybe)	Alex, Bart	4	
	Draw the class diagram of aforementioned main elements	Stefan, Menno	2	
	Draw the sequence diagram	Mark	8	
	Aggregation and Composition	Stefan, Mark	2	
	Parameterized class	Alex	2	
	Draw the class diagram for all hierarchies	Bart, Menno	8	
Support Logging (responsibility driven design & UML)	- Make requirements document for logger	All	1	
	- Logger Class incl. Logger.log() method	Bart, Alex	4	
	- Logging GUI Package	Mark	4	
	- Logging logic package	Menno, Stefan	8	

User story: Logger

Title: Player interacts with Logger

Narrative:

As a player of the game,
I want there to be a logger,
So that I can inspect the history of my actions.

Acceptance criteria:

Scenario 1: Recording important events.

Given a running game,
And certain events have happened,
When I look at the console or the log file,
Then I should see all these important events,
And they should be organised by timestamp,
And they should include priority levels.