Exercise 1

Requirements

Powerups are visually shown and can drop from the bubbles. The menu is a start screen where options can be adjusted too.

Must

- Powerups are drawn to the screen with an icon to display that a powerup is in use.
- A powerup has a visual in-game representation that can be picked up by the player.
- On random occasions (once in five splits), the in-game representation of a random powerup drops from a bubble when a bubble is split.

Should

- The shop shows when a powerup is bought with an icon.
- The menu screen is interactive so that options for the game can be set.
- An in-game powerup expires after 30 seconds.
- A power that is bought in the store expires when a level is completed or lost.

Could

- Buttons in the store give visual feedback by adding depth when clicked.
- The player sprite is changed when a powerup is applied.

Would

- The menu is multiple levels deep and screen and audio options can be changed.

CRC

Many of these classes already exist, but we want to look at their responsibilities to see where we apply the changes and lay the new responsibilities.

Classes

- PowerUp
- PowerUpGeneraror
- GameView

PowerUp	
Superclass(es):	
Subclasses: LifePowerUp, SlowPowerUp, TimePowerUp	
Activate powerup	
Draw powerup	GameView

PowerUpGenerator	
Superclass(es):	
Subclasses:	
Generate a powerup based on randomness and previous events	Bubble
	PowerUp

GameView	
Superclass(es):	
Subclasses:	
Draw all elements to the screen	Wall
	Bubble
	Player
	PowerUp

Explanations for decisions

We decided to not implement powerups as decorators, because this would limit the effect of powerups to - for example - the player. We'd also like powerups to affect the timer and bubbles.

Exercise 2

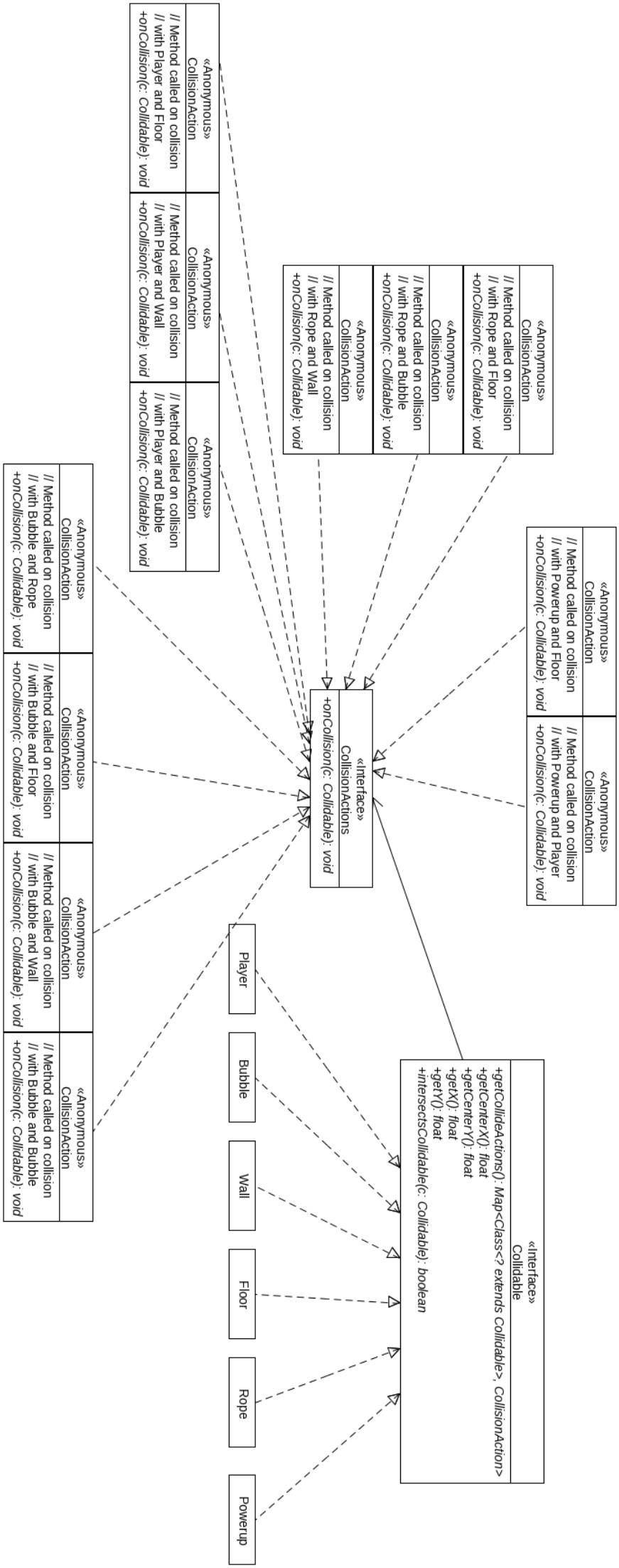
1.a) For the first design pattern, we implemented the strategy pattern to handle different collision actions. Every object needs to invoke different actions on a collision. This depends on the class of the object, and the object it collides into.
To handle these actions in a way that is consistent and easily extendable, we used the strategy pattern.

Every different action must be an implementation of the *CollisionAction* interface. Every object that handles collisions implements the interface *Collidable.* This interface requires all implementations to implement a method *getCollideActions().* All CollisionActions a particular collidable object can perform, are specified in that method.
The method returns a HashMap with implementations of *CollisionAction.* Depending on the class of the object it collides into, a different implementation is used.

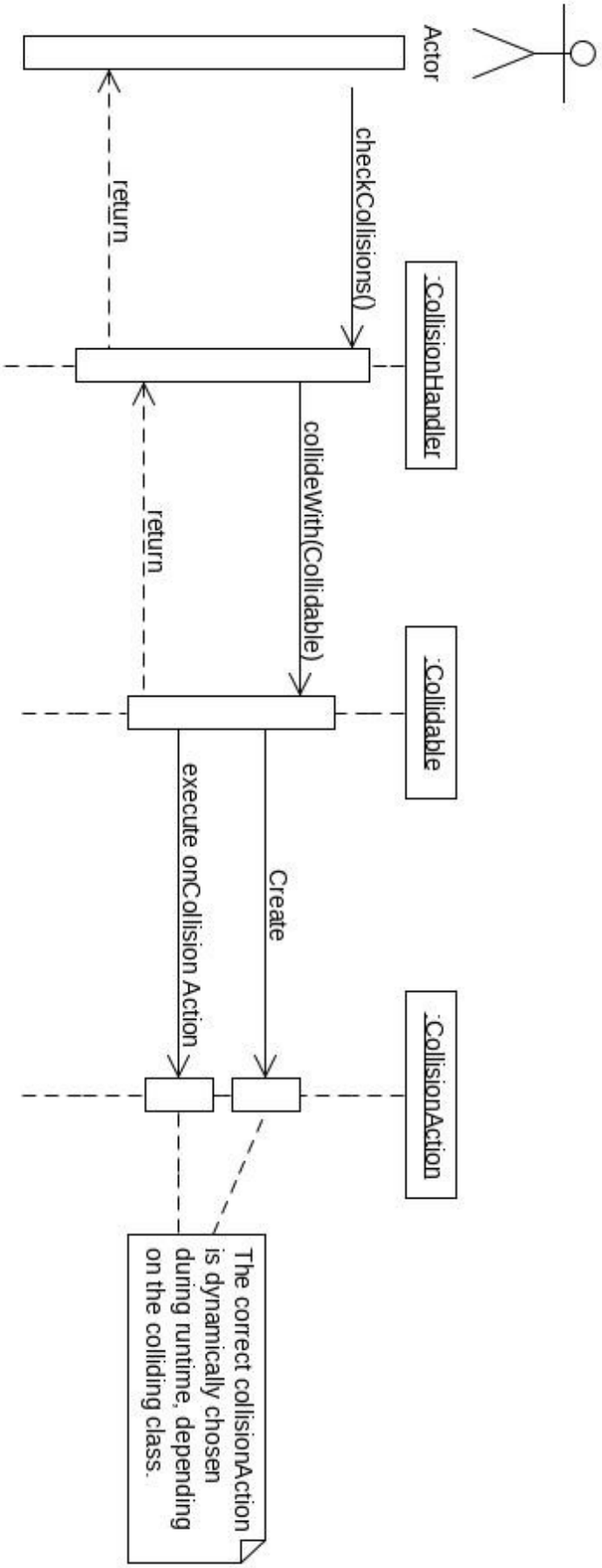
We implemented this with a HashMap to avoid class checking(instanceof etc). The class is used as the key, and a CollisionAction implementation is returned as the value.

We didn't create concrete classes implementing the *CollisionAction* interface. This could be done, but for our implementation it wasn't convenient. The amount of concrete classes needed would make it disorganized quickly. All CollisionActions are anonymous inner classes instead.

Class diagram Strategy pattern



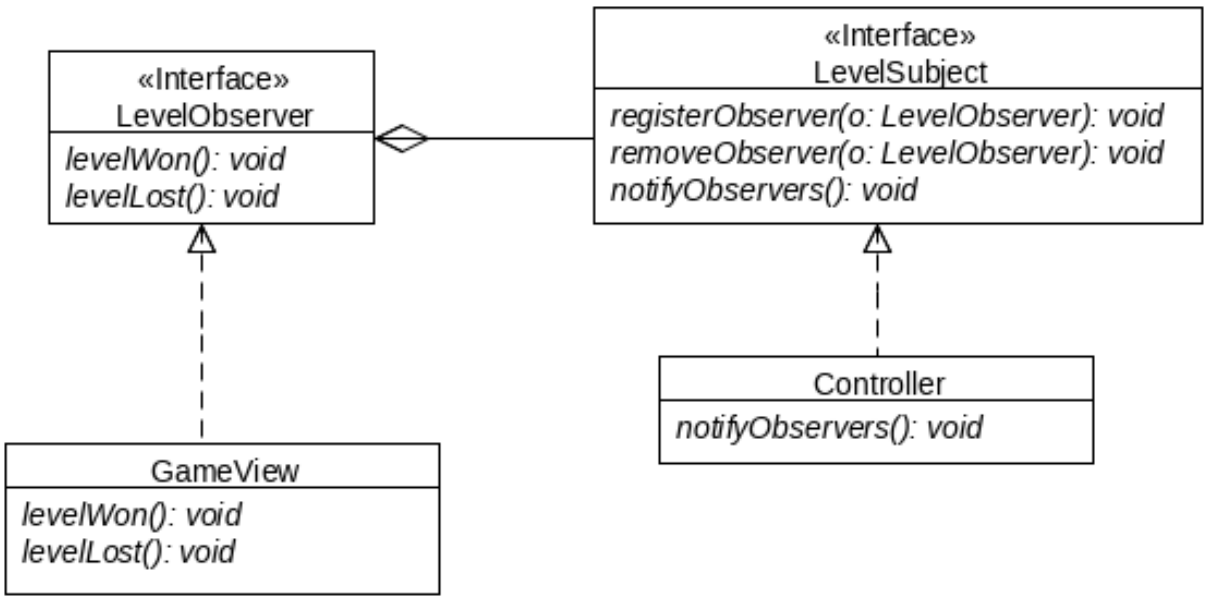
Sequence
diagram
Strategy pattern



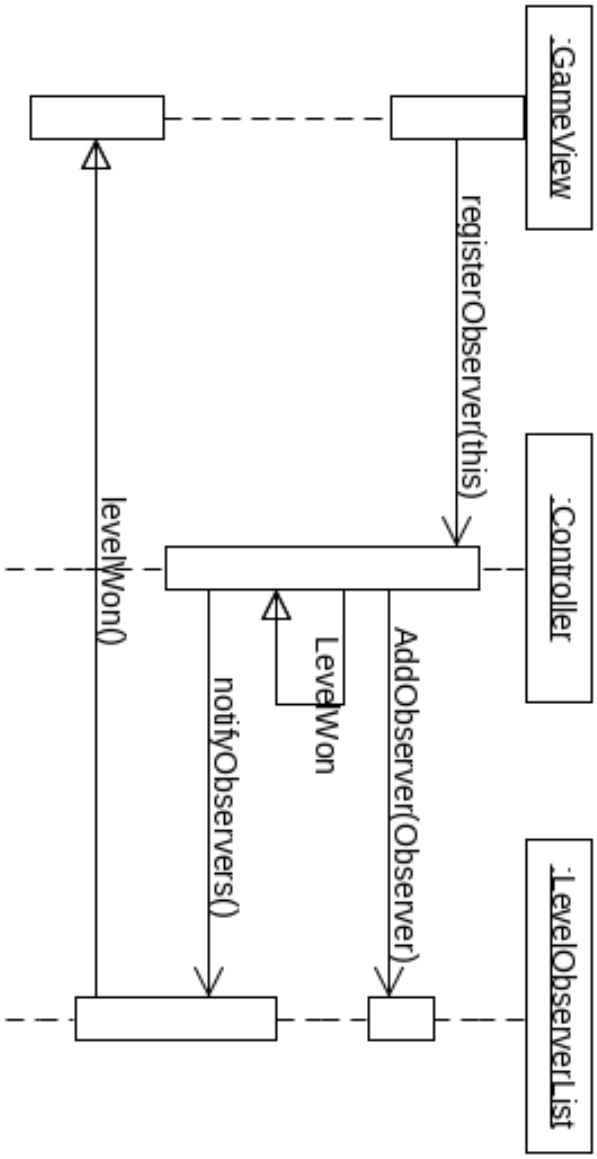
1.b) For the second pattern, we implemented the observer pattern to notify observers about winning or losing a level. Using this pattern allows for loose-coupling between classes. Classes can interact, while having little knowledge about each other.

We implemented this so an observer can be notified about winning or losing levels, and handle the corresponding actions independently. The controller is the subject, and will notify observers on level state changes. The GameView class implements the observer, so the game state can be changed(using Slick2D StateBasedGame) accordingly.

On winning a level, the game state will be changed to the ShopView. Upon losing a level, the game state will be changed to the LostLevelView. In this state, information about
the player is shown. Pressing enter will return to the GameView.



Class Diagram Observer pattern



Sequence diagram Observer pattern

Exercise 3

1.
In this research they have used a repository with projects from three different companies. This repository contains 352 project. All projects will be compared with the average of this repository. First, an average of costs and duration is calculated per size category. With this information two graphs can be made. One with the average costs per size category and one with the average duration per size category. In this graphs, each project can be plotted. If the project is plotted higher than the average line, the project has a higher cost and/or duration. Else the project has a lower cost and/or duration. With this information a cost/duration matrix can be made. If the project was in both the cost and duration graph lower then average, this project is in the good practice quadrant (which is the right upper quadrant). Else if the project was in both the cost and duration graph higher then average, this project is in the bad practice quadrant (which is the left under quadrant).

2.
It is likely that projects which used Visual Basic are less complex. Another reason might be that the Visual Basic skills were better than other skills. But the real cause isn't possible to be identified. This is due by the upfront collection of the data from the projects.

3.

- The independent testers. This would end in the good practice quadrant. If the testers are not independent they will ignore some doubtful statements. This because he thinks to know the code. On the other hand, if the testers are independent they will go into every doubtful statement.

- The use of Code Review. Also this factor would end in the good practice quadrant. If the project use code review regular this would improve the project. Most of the time code review improves the code quality. Also code review finds other defects than dynamic testing does.

- The non regular availability of the team members. This factor would end in the bad practice quadrant. If the team members are available at different times there is less time to plan and reflect the project. This would have a negative effect on the project because some features might be missed or might be done twice.

4.

- Once-only project. Such a project would end in the bad practice quadrant. In such a project team members don't have experience with earlier releases in the project. The effect of this is that faults are made which otherwise would have been encountered during earlier releases.

- Technology driven project. A project which works driven by a particular technology ends in the bad practice quadrant. This project isn't designed for the requirements why some classes will have wrong or missing functionalities and some classes will be missed.

- Rules and Regulations driven project. Team members in such a project don't

have the freedom anymore to work. An effect is that team members design the system differently which is a disadvantage for the functionality of the system.