

# Software Engineering Methods TI2206

## Assignment 4

*Group 47*



**Course Manager: Alberto Bacchelli**

**TA: Aaron Ang**

Lillian Lehmann 4009037

Karin van Garderen 4144384

Christian Maulany 4097238

Rogier Vrooman 4001257

Fieke Miedema 4141369

Bas Böck 4366174

## ***Exercise 1 – Your wish is my command, Reloaded (45 pts)***

### **1. Requirements:**

#### **Must have:**

- The magiron shall appear when a level is not cleared within 45 seconds
- The magiron shall move towards the player
- The magiron shall be visualized by a flying magikarp-Aaron-combination
- The magiron shall kill the player upon collision
- The magiron shall be invincible
- Fruit shall appear when enemies are killed, close to the enemies
- Fruit shall fall to the ground
- Player shall get points for picking up fruit
- A score screen shall appear after the game is over
- The player can enter their name in the score screen
- The score screen shall keep track of the 10 best scores

#### **Should have:**

- Every level shall have an animated background
- The magiron shall be animated

#### **Could have:**

- Every level shall have a different background
- All enemies shall be animated
- Projectiles shall be animated

#### **Won't have:**

- The game shall have bonus stages

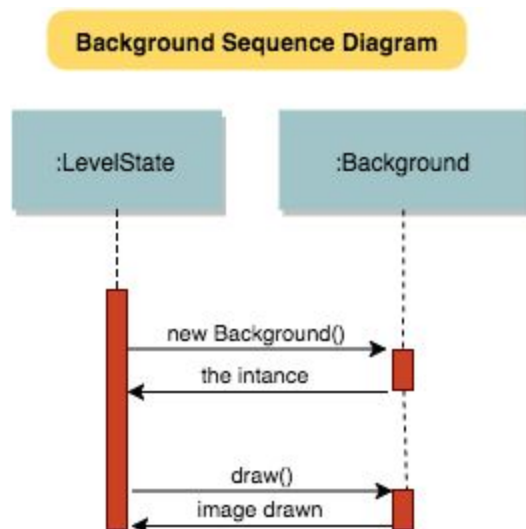
## 2. Responsibility Driven Design

### Background:

The responsibilities of the background feature are shown in the CRC card below. The background class uses on instantiation the location of the to be used image. The class creates a buffered image from this .gif file, and creates an Animation instance of it. In the draw method in Background, the animation is updated and drawn.

Background	
Superclasses: none	
Subclasses: none	
Show animated background image	Animation

The sequence diagram of the background is as following:

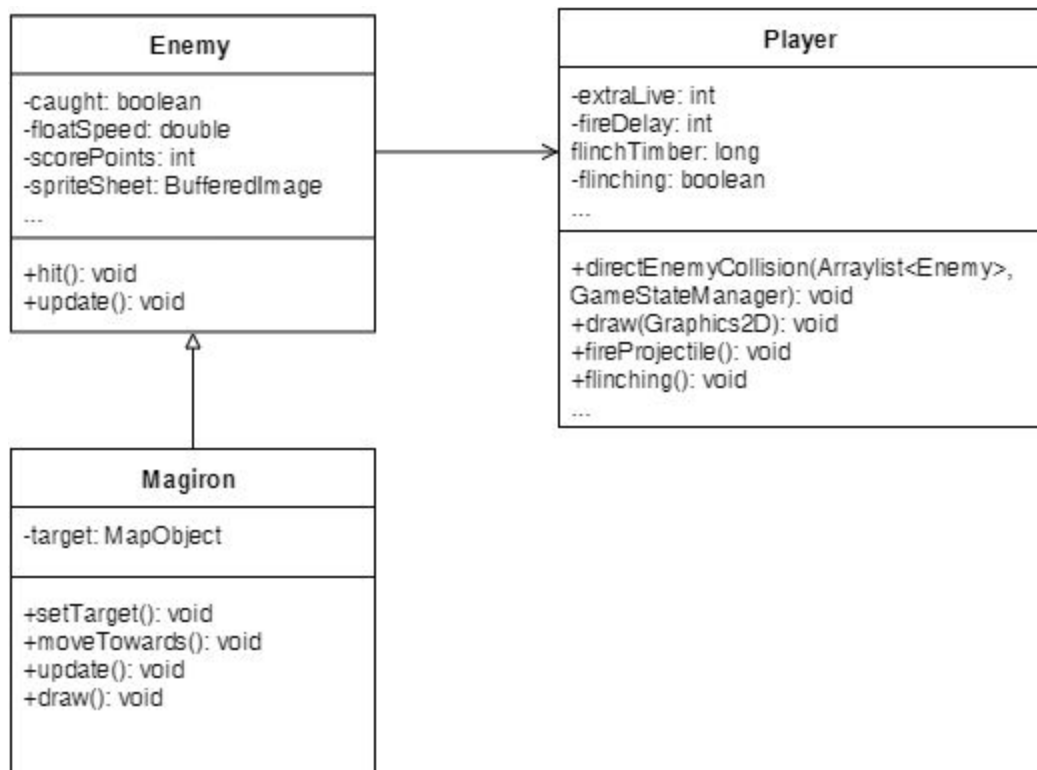


## Magiron

Magiron is a magic fish like creature with face of Aaron. The character appears when a player is spending too much time on a single level. Aaron follows the player and kills it when the two collide.

Magiron	
Superclasses: Enemy	
Subclasses: None	
Follow the player	Player
Kill the player on collision	

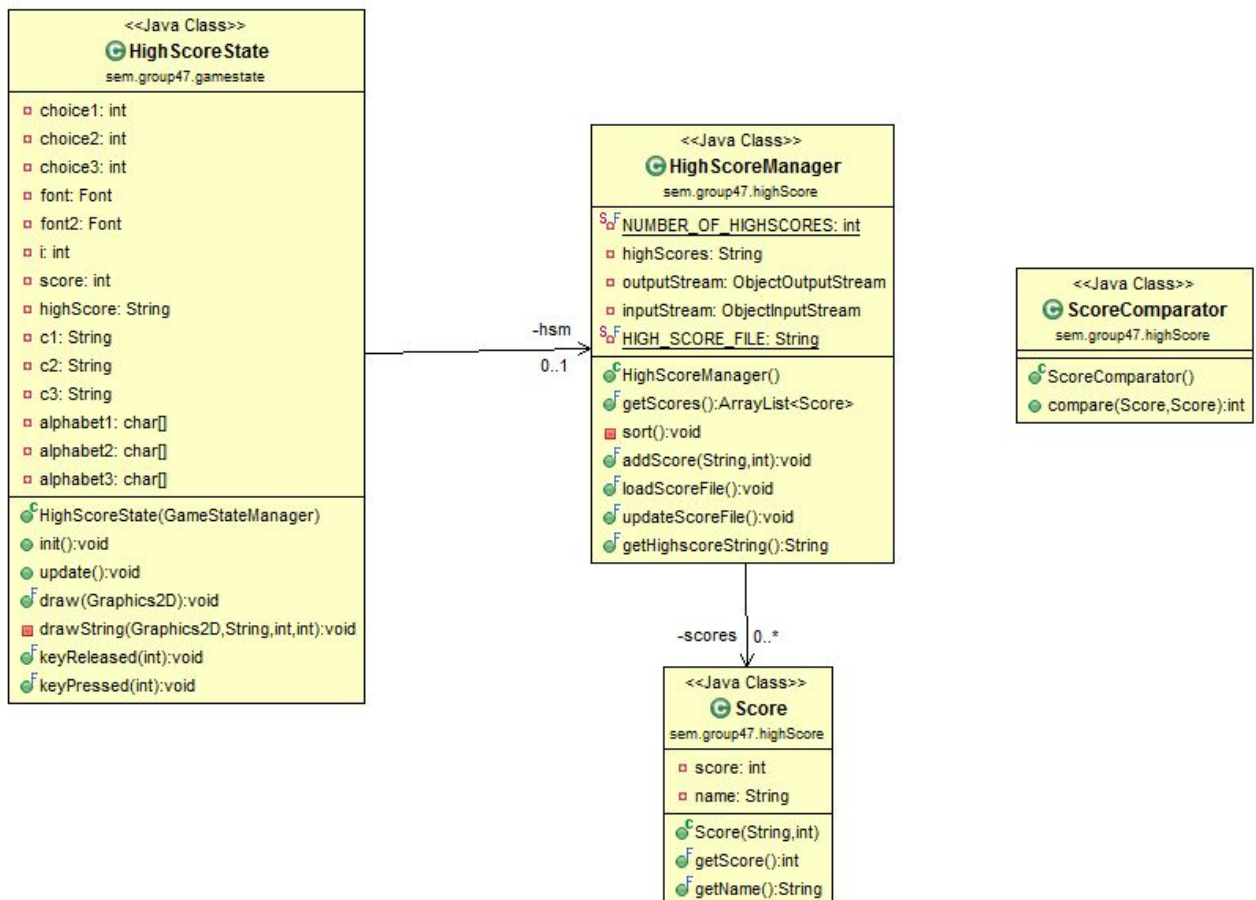
By using Enemy as a Superclass, much of the collision detection logic is already present and only the movement of the Magiron needs to be defined.

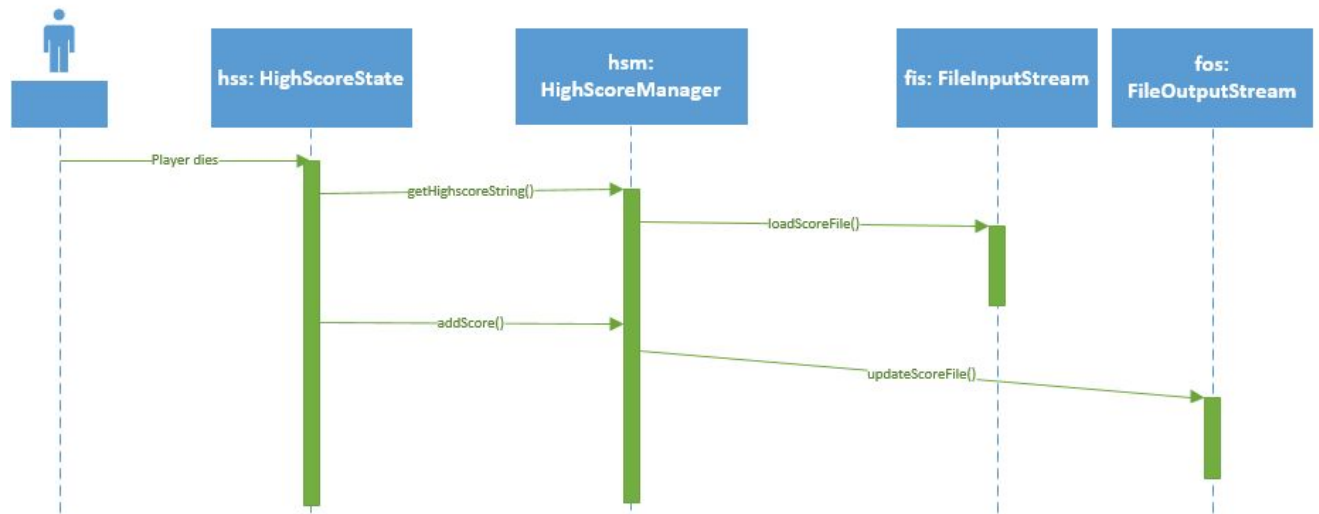


## HighScoreList:

The high score list keeps the highscores of the player. When a player dies the highscore list shows the top 10 players with their score (read out from the highscore.dat file). The player can enter his name with it's high score. The highscore manager then writes the score to the highscore.dat file.

HighScoreManager	
Superclasses: none	
Subclasses: none	
Show Player highscore	Player





## Exercise 2 – Software Metrics (45 pts)

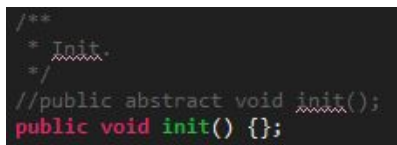
The Incode Analysis file is located in sprints->assignments->incode\_analysis.result

### Flaw 1 - Player

The biggest flaw was Player being a God Class. This is certainly true, because it combines game logic with animations and physics. Refactoring this into separate classes proved to be too difficult, but we did manage to reduce the severity. This was done by moving the projectiles for each player into a separate class, which holds a list of them and has some methods for updating and checking collisions. This caused the severity of Player's Godlike status to decrease from 6 to 4. We believe the next step would be to move the animations, which are now managed inside Player, into a separate PlayerAnimation class and to move the physics to a superclass. An attempt was made, but to do this effectively would require a lot more time.

### Flaw 2 - GameState

The inCode program resulted in several Flawed Classes. Most of them are classes which extended GameState: OptionsState, MenuState, HelpState, GameOverState, HighScoreState. The report of flawed classes stated that there were methods in these classes that were not used. This was because all the gamestate variants extended the abstract class GameState, and therefore the classes had to implement all the abstract methods of GameState. The problem was solved by transforming the abstract methods in GameState to regular classes with no implementation. The constructor method is still abstract, so no instances of the class GameState can be made (with empty methods). So now, the subclasses of GameState can ignore the classes which will not be used. Thus, after this, we removed all the unused methods in all the GameState extensions, and the flawed classes report did not report the extension of GameState as a flawed class.

A screenshot of a code editor showing a change to the GameState class. The original abstract method signature is commented out with /\*\* and \*/. A new concrete method signature is added below it.

```
/**  
 * Init.  
 */  
//public abstract void init();  
public void init() {};
```

*The changes to the GameState method(old line is commented out)*

### Flaw 3 - Projectile

The last two design flaws are concerning Data Classes. Both the PlayerSave and the Projectile Class are marked as such. For the Projectile Class this could be solved by removing the unused getter and setter methods. There are reasons to keep the getter and setters, but since we do not plan on using them anyway we can remove them. For the PlayerSave this flaw is interesting, because the Class actually is supposed to be a Data Class. It holds the information about the state of the player which should be persistent across GameStates, such as amount of lives and score. Therefore this could not be solved, without changing the implementation of how this data is saved. And considering the scope of this project, one could argue that this is the better implementation.

```
/**
 * Checks if is floating.
 *
 * @return true, if is floating
 */
//public final boolean isFloating() {
//    return floating;
//}

/**
 * Sets the floating.
 *
 * @param pFloating
 *        the new floating
 */
//public final void setFloating(final boolean pFloating) {
//    this.floating = pFloating;
//}

/**
 * Gets the float speed.
 *
 * @return the float speed
 */
//public final double getFloatSpeed() {
//    return floatSpeed;
//}

/**
 * Sets the float speed.
 *
 * @param pFloatSpeed
 *        the new float speed
 */
//public final void setFloatSpeed(final double pFloatSpeed) {
//    this.floatSpeed = pFloatSpeed;
//}
```

*The removed getter and setter methods for Projectile*