Assignment 1

by

Group #6

Delft University of Technology
Faculty of Electrical Engineering, Mathematics and Computer Science.
Mekelweg 4,
Delft

Name	StudentId	Email
Michiel Doesburg	4343875	M.S.Doesburg@student.tudelft.nl
Matthijs Halvemaan	4353803	M.J.W.Halvemaan@student.tudelft.nl
Dmitry Malarev	4345274	D.R.Malarev@student.tudelft.nl
Sunwei Wang	4345697	S.Wang-11@student.tudelft.nl
Clinton Cao	4349024	C.S.Cao@student.tudelft.nl

TA:
Valentine Mairet V.A.P.Mairet@student.tudelft.nl

Course: *Software Engineering Methods (TI2206)*

TU Delft

Table of Contents

Exercise 1:

Deriving classes with CRC cards

Following the Responsibility Driven Design and using CRC cards, we have derived the following classes with their responsibilities and collaborations (figure 1.1 and 1.2):

Game	
Superclass(es): none	
Subclasses: none	
Responsabilities	Collaborations
Starts the game	MainScreen, PlayerFish
Ends the game	MainScreen, LosingScreen and WinningScreen

Fish	
Superclass(es): none	
Subclasses: PlayerFish, EnemyFish	
Responsabilities	Collaborations
Represents a fish object	playerFish, EnemyFish

Screen	
Superclass(es): none	
Subclasses: OptionScreen, MainScreen, LosingScreen, WinningScreen	
Responsabilities	Collaborations
Represents a 2d plane, where all the graphical elements of the game will be shown/drawn.	OptionScreen, MainScreen, LosingScreen, WinningScreen

MainScreen	
Superclass(es): Screen	
Subclasses:	
Responsabilities	Collaborations
Handles actions (clicking of buttons, pressing of buttons etc.)	Game

figure 1.1. CRC cards of the classes Game, Fish, Screen and main screen.

EnemyFish	
Superclass(es): Fish	
Subclasses:	
Responsabilities	Collaborations
Moves across the screen	Game, MainScreen

PlayerFish	
Superclass(es): Fish	
Subclasses:	
Responsabilities	Collaborations
Represents the fish that will be controlled by the player	Game, MainScreen

OptionScreen	
Superclass(es): Screen	
Subclasses:	
Responsabilities	Collaborations
Shows the options and the instructions of the game	Game

LosingScreen	
Superclass(es): Screen	
Subclasses:	
Responsabilities	Collaborations
Shows that the player has lost/died	Game

WinningScreen	
Superclass(es): Screen	
Subclasses:	
Responsabilities	Collaborations
Shows that the player has won the game	Game

figure 1.2. CRC cards of the classes EnemyFish, PlayerFish, OptionScreen, LosingScreen and WinningScreen.

Starting from the *must haves* of our requirements, we highlighted some of the noun phrases that we think that can be classes. These words are the following:

- Game
- Fish
- Player
- Player-fish
- Additional fishes
- Screen

Then from the *should haves*, we highlighted the following noun phrases:

- Losing screen
- Winning screen
- Start screen

There are a lot of noun phrases in the *could haves*, but we did not highlight any of them. The reason is because we did not see the reason why these noun phrases should represent a class for our game. We think that they are in no way an essential component of the game; for example we do not see the reason why we "*sea*" should represent a class in our game. We do not see the benefits for implementing a "sea" class, as it is not essential for our game.

Deriving responsibilities of classes

From the words highlighted in the *must haves*,we think that all of them are essential for our game; game should be a class, as it is the "driver" for our game. This class has the responsibilities of starting/launching the game and ending the game.

Fish should be a class, as it represents the fish that will be spawning on the screen and also the fish that will be controlled by the player, therefore fish has the responsibility to represent a fish object for our game.

We decided that both player and player-fish should be (together) one class. This is because player will the one that will be controlling player-fish. Hence we think that it is better to merge them both into one class instead of two separate classes. This class has the responsibility to represent the fish that the player will be controlling. This class is also a specialization of the class Fish.

Additional fishes should be a class, as it represents the fishes that will be moving horizontally across the screen. We gave this class the name of EnemyFish, because it is not controlled by the player and the player might die from colliding with it. This class has the responsibility to move across the screen. This class is also a specialization of the class Fish.

Screen should be a class, as it represents the windows where all the graphical elements (fishes, background, buttons etc.) of the game will be shown/drawn. And of course the responsibility of this class is to represent the 2d plane where the all the graphical elements of the game will be shown/drawn.

From the words highlighted in the *should haves*, we think they all should be a class; although we have already derived a screen class, the *losing screen, winning screen* and *start screen* will each have their own responsibilities beside the responsibility that *screen* has. Therefore they are each a specialization of the class *screen*. Because the player will only get to the *winning screen* when the player has won the game, therefore the responsibility that we have give to to the *winning screen* is to show that the player has won the game.

Just like *winning screen*, the player will only get to the *losing screen* if the player has lost the game, and therefore the responsibility that we have given to the *losing screen* class is to show that the player lost the game.

Start screen is where the player can select to play the game, exit the game or to go to the options of the game. Because of this reason we decided to give it the name *MainScreen* in our CRC cards, as this screen must handle all the events/action (clicking of buttons, keypress etc.) that can happen on the screen and the player will be on this screen most of the time (the player can only get to the other screens only if certain events has happened). Therefore the responsibility that we have to this class is to handle actions (clicking of buttons, pressing of buttons etc.).

Deriving collaborations between classes

From the words highlighted in the *must haves*, *Game* would collaborate with the class *MainScreen* and *PlayerFish* to start the game, since the game should start the main screen and a player fish should be present in the empty sea.

Class *Fish* on the other hand collaborates with class *EnemyFish* and *PlayerFish* to represent a fish object for our game, as it is the superclass of both classes.

To be more specific, class *PlayerFish* represents the fish controlled by the player, it is launched in the *MainScreen* after the game is started, therefore, it collaborates with class *Game* and class *MainScreen*.

The other subclass of Fish is *EnemyFish* and for the same reason as *PlayerFish*, it collaborates with class *Game* and class *MainScreen*.

In the same manner, the class *Screen*, which is the superclass of all the screen classes, collaborates with class *OptionScreen*, *MainScreen*, *LosingScreen* and *WinningScreen* to represent the 2d plane where the all the graphical elements of the game will be shown/drawn.

For all the classes in *should haves*, the *LosingScreen, WinningScreen* and *Start screen* will each have their own responsibilities and collaborations beside the responsibility and collaborations that *screen* has. This reason is that these classes are specializations of the *Screen* class. Because the player will only get to the *WinningScreen* when the player has won the game, therefore the collaboration of the *WinningScreen* is with *Game*.

Just like *WinningScreen*, the player will only get to the *LosingScreen* if the player has lost the game, and therefore the collaboration to the *LosingScreen* class is also with *Game*.

Start screen , also called *MainScreen* is where the player can select to play the game, exit the game or to go to the options of the game. As this screen must handle all the events/action (clicking of buttons, keypress etc.) that can happen on the screen and the player will be on this screen most of the time (the player can only get to the other screens only if certain events has happened), this class will also collaborate with the class *Game*.

OptionScreen shows the instructions of the game, therefore it collaborates with the class *Game*.

The comparison of the result with our actual implementation

In comparison with our actual implementation, we have a lot of classes in common, for example, class *Game, enemy fish, player fish*. The classes *main screen, options screen, winning screen and losing screen* are also implemented in the actual implementation. However, these classes are implemented as *MainScreenController, OptionScreenController, WinningScreenContoller* and *LosingScreenController*. This is because we are working with JavaFX and we implemented the controllers for the screen, rather than the screen itself. The class *Fish* is in our implementation, however we did not name it *Fish*. We gave it the name of *Entity*. We think that it might be better and less confusing to rename the class from *Entity* to *Fish*.
On the other hand, there is a class that we derived which are not included in our actual implementations. It is the class *screen*, which is suppose to be the superclass of all the screen classes. The reason behind this is the same reason we explained why the other screens are implemented as controller and not the screen itself.

We have also two additional classes in our actual implementation which were not derived from our requirements, these are class *bounding box* and class *sprite*. They are not included in our requirements, but these classes made it easier to construct the game in a more organized way.

Description of main classes in terms of responsibilities and collaborations.

Main classes:

- BoundingBox
- Sprite

- Entity → EnemyFish (Both of these classes extends the entity class.)
  → PlayerFish

- Game
- LosingScreenController
- MainScreenController
- OptionsController
- WinningScreenController

BoundingBox is one of the most basic classes. Its responsibility is to represent a bounding box (axis-aligned) which will function as a hitbox for calculating collisions between the PlayerFish and the EnemyFish. It collaborates with the Sprite class to achieve this. The BoundingBox consists of four attributes: X and Y integer values, to represent the BoundingBox's location withing a 2-dimensional plane, and width and height integer values, to represent the size of the BoundingBox. The BoundingBox class has a boolean intersects method which accepts another BoundingBox and update methods which enable it to 'move' over the screen.

Sprite builds on top of BoundingBox. The responsibility of the Sprite class is to represent an actual ingame 'model'. This model should be visible to the player, but it should also have a 'behind the scenes' numeric representation of its size and location to calculate any collisions. This is why the Sprite object consists of an Image and a BoundingBox. The BoundingBox is used for aforementioned calculations, and the Image is what is actually drawn on the screen. The Sprite class has methods for updating its location, a method to render it on the screen and a boolean intersects method which accepts another Sprite. The Sprite class collaborates with the Entity class to model a creature.

The **Entity** class represents a creature of some kind. So far, there are only fish present in the game, so the Entity class could be described as the most basic way of modelling a fish. In our minds, a 'fish' is just a Sprite moving over the screen. So the Entity class consists of a movement speed integer attribute, and a Sprite. It has a boolean intersects method, to calculate the collisions between entities. And it has several boolean intersects methods which calculate the collisions with the screen. These are used to check if the fish controlled by the player should stop moving (if it's intersecting the screen edge). Entity collaborates with EnemyFish and PlayerFish, since these classes both extend Entity.

The **EnemyFish** class' responsibility is to represent an enemy fish. It extends Entity, so it inherits the attributes Sprite and movement speed. In Fishy, enemy fish of randomly varying size spawn at a random side of the screen, moving at a random movement speed. The movement speed and size attributes are already contained within the Entity class. EnemyFish adds a boolean attribute to indicate whether the fish spawns at the left or the right side of the screen. Most importantly, EnemyFish has a generator method which generates a new enemy fish with random attributes. This method is used to spawn new enemy fish in the game. Naturally, EnemyFish collaborates with Entity, namely through the inheritance from Entity, the intersects method, and the render method. It also collaborates with Sprite and BoundingBox since it is built from these objects, and MainScreenController, where instances of EnemyFish are held within an arraylist.

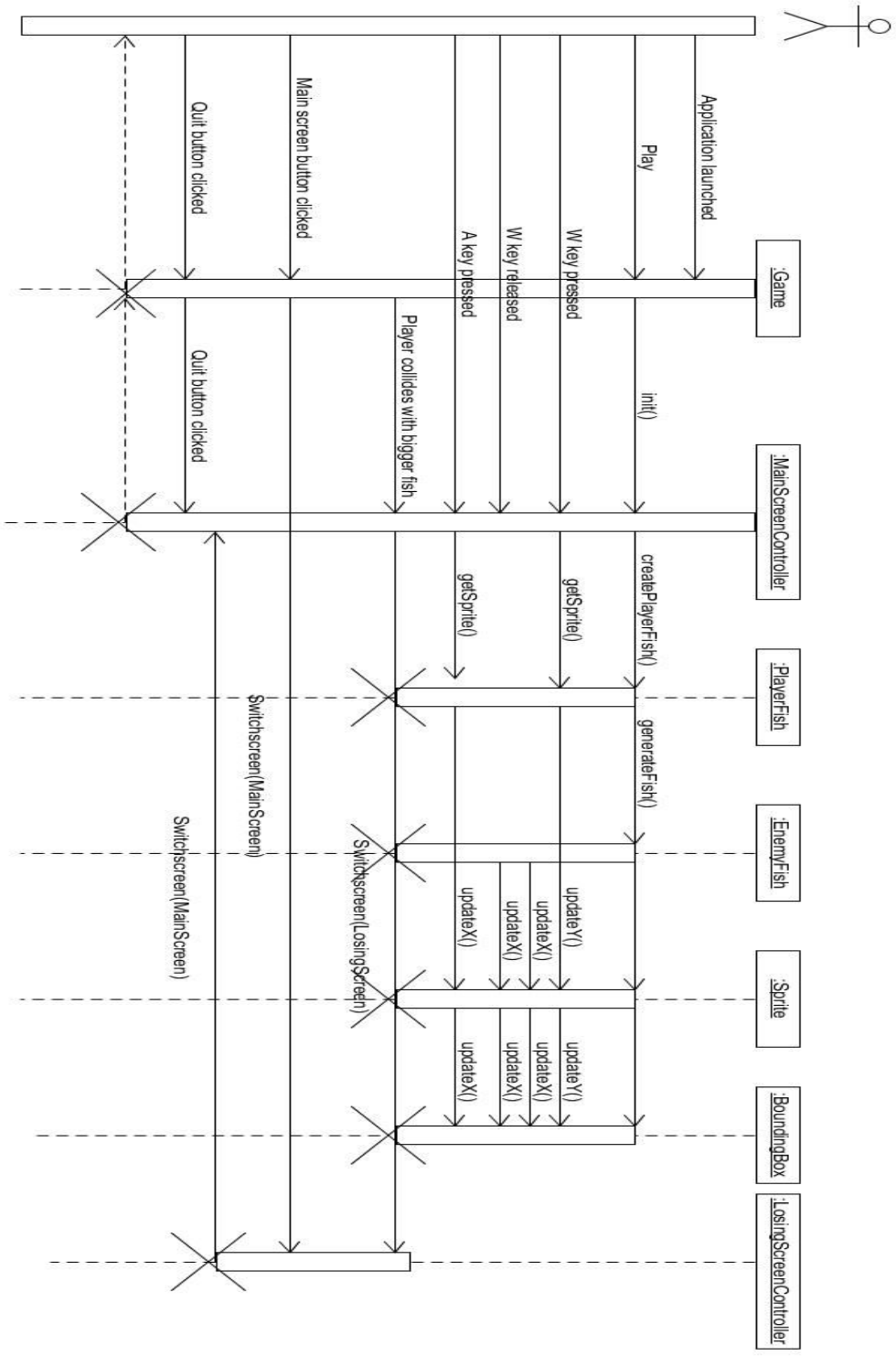
The **PlayerFish** class's responsibility is to represent the player fish. It extends Entity as well, so the Sprite and movement speed attributes are inherited. Furthermore, the player fish has a boolean attribute to indicate whether the player is alive, and a score integer attribute to hold the player's score. The PlayerFish class has a method to create the player fish at the start of a new game, a method to grow the player fish when it eats another fish, and a method to check if the player dies when it collides with another fish. The 'growing' is achieved through scaling the Image and updating the BoundingBox subsequently. PlayerFish collaborates with Entity, Sprite. BoundingBox and MainScreenController. It is built from or extends the former three classes. MainScreenController holds an instance of PlayerFish and controls the player fish in regards of the actual game.

The **Game** class' responsibility is to represent the conceptual entity of a game instance. It controls the switching of screens according to the player's inputs, and loads the necessary resources when the game is launched. It collaborates with LosingScreenController, MainScreenController,
OptionsController and WinningScreenController. It switches between these screens when necessary.

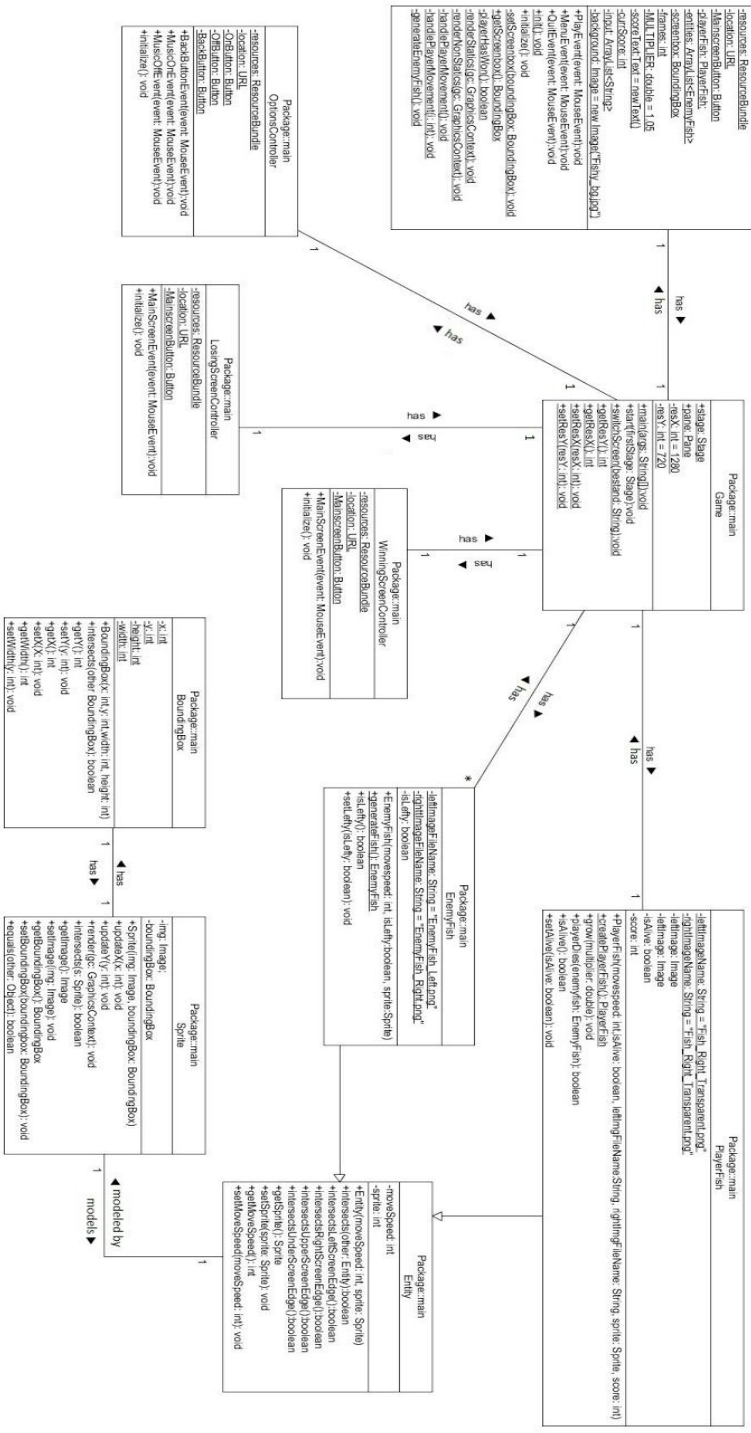
All the **Screen Controller** classes model the screen objects. The panel, background image, buttons, event listeners and handlers are all contained within these classes where necessary. Each class' responsibiliy is to handle everything pertaining to their 'own' screen. Out of all these classes, MainScreenController is the most important one. The 'main screen' being the screen where the player is actually playing the game. As such, MainScreenController contains the actual game loop. It handles the movement of the player, the movement of all the enemy fish, collisions, removal of enemy fish when they move outside the screen and the death or victory of the player.

At the moment, we feel like the classes we have now maintain a good balance of responsibility versus class length. We could merge some classes, but it would lead to some classes with too much responsibility and decrease the modularity of the system. For instance, there are a lot of different screen controller classes. It is possible to merge these classes into one. We have thought about this, but ultimately decided against it. We feel every screen controller class now has a clear responsibility of dealing with everything pertaining to its particular screen. Merging these classes would lead to a class with too much responsibility, decrease the modularity of the system, and reduce the overall clarity and structure in the code. The changes we did make involve switching around some methods in different classes which were not in the right place according to each class' responsibility.

Sequence Diagram of Fishy



Class diagram of Fishy



Exercise 2:

Aggregation and Composition

The main difference between aggregation and composition is the way the two objects reference each other.In Composition, the child class is fully dependent on its parent, meaning that, if the parent gets deleted, the child would be deleted as well.On the other hand, aggregation is mainly a simple reference, meaning that the child class is independent from its parent.

In our implementation, here are three main cases where association occurs:

- The Class PlayerFish, which extends Entity,
- The Class EnemyFish, which also extends Entity
- The Class Game, which extends Application.

The classes that extend Entity (PlayerFish and EnemyFish) are compositions, since they use their superclass' constructor, since both classes are different types of entities, which means that PlayerFish and EnemyFish (heavily) depend on their superclass to function.

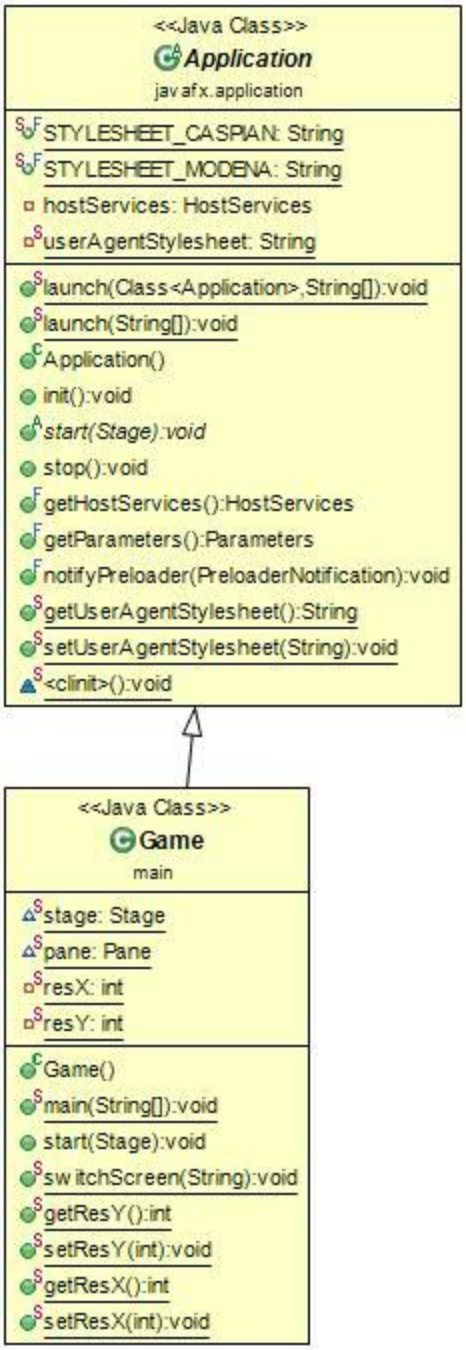
The association between Game and Application, however, is an aggregation. Mainly because the Game class shares itself with the controller classes of the game.

Parameterized Classes

There are currently no parameterized classes in our code. Parameterized classes are most useful when it comes to generic 'holder' type classes, like a list. If you write a class for a list of Strings, you can construct a new list an add String to it, but nothing else. If you want to add some integers as well, you have to make a completely new class which does the exact same thing but with integers instead. Now if you want to add some doubles, or perhaps an entirely new object you made in a different class, you'd have to make a whole new list class again. Obviously this is a big waste of time and code. Parameters solve this problem by letting the list class be defined once, and only defining the *type* of the list when an instance of the list is made. This does not just solve the time and code problem. Instead of making a list class with, for instance, type Object, to hold many different kinds of Objects, you can use generics. This enables stronger type checking at compile time (if you're trying to add the wrong type to a certain list) and it eliminates explicit casts.

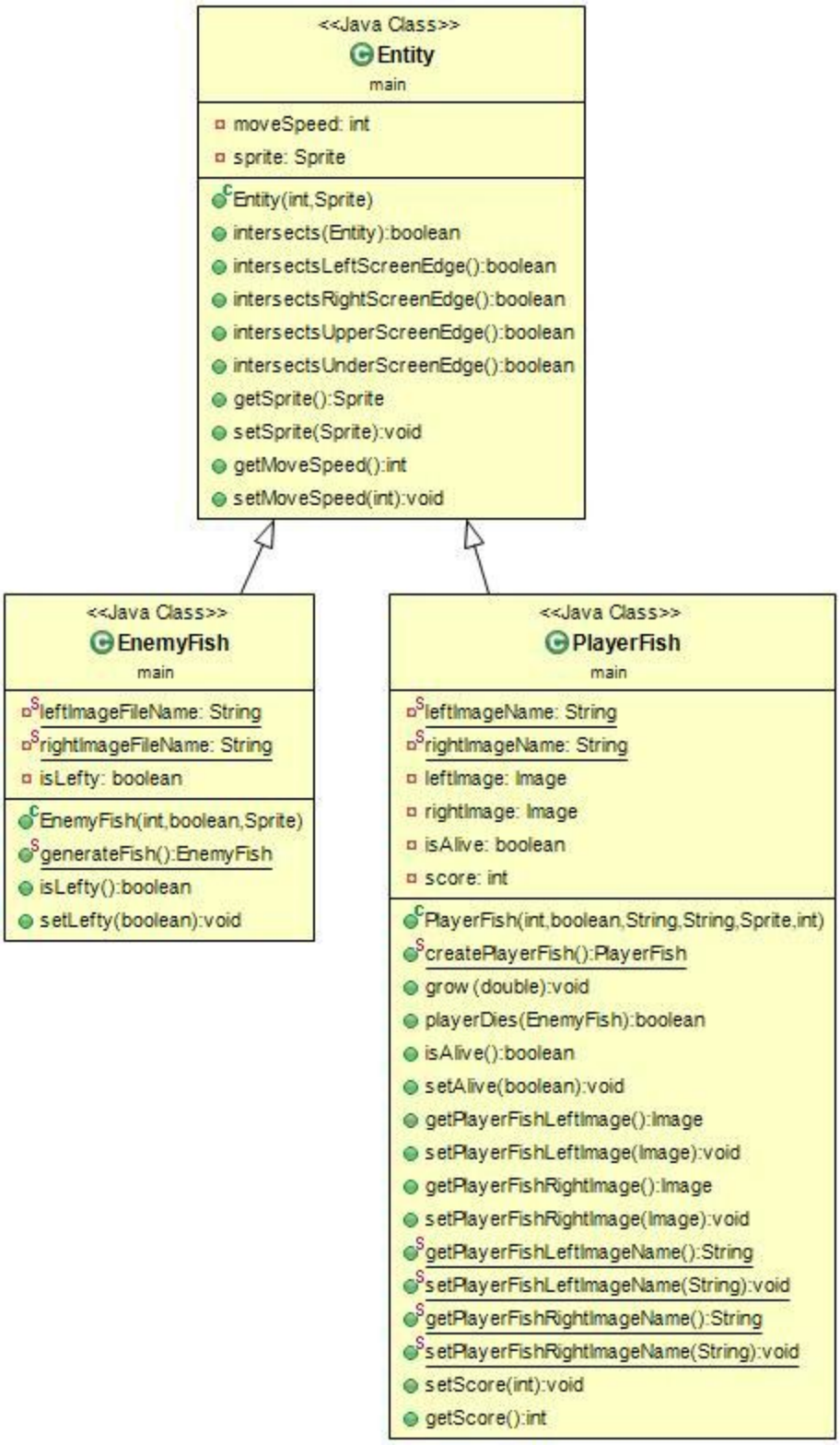
Class Hierarchies

Currently the following hierarchies exist, the first of which is the following:



The hierarchy displayed above is of the “Is-a” type.

Apart from this one, there is one other hierarchy that exists in the application.



This one is of the “Polymorphism” type. Since we are not using any inheritance of the “Reuse” type, the system is not in any real danger that can be caused by this type of inheritance. This means that we do need to remove any hierarchies from our application.

Exercise 3

Please look at the Requirements_Logger.pdf for the requirements of the logger and also please look at our source code for the implementation of the logger.