Group 11 - Assignment #1

Exercise 1.1

The classes we have derived based on the initial requirements, and the reasoning for including them:

- *Entity* – The actual entities with which the game is played. Both the NPC fish and player's fish are entities and therefore subclasses of this. This is an abstract class.
- *Player* – The fish which is controlled by the user.
- *Enemy* – The NPC fish that spawns at the sides of the screen and the player has to eat/avoid.
- *Spawner* – The class which takes care of the actual spawning of NPC fish.
- *ScreenHandler* – The class which takes care of the switching between screens.
- *MenuScreen* – The screen which is first shown when the game is started.
- *PlayScreen* – This would be the actual screen the user sees and interacts with.
- *WinScreen* – The screen which is shown after a game has been won.
- *LoseScreen* – The screen which is shown after the player is killed.

CRC Cards:

Class Name	Entity
Superclass	-
Subclasses	Player, Enemy
	Provide a basis for entities

Class Name	Player
Superclass	Entity
Subclasses	-
	Controllable by the user Grows when eating smaller fish

Class Name	Enemy
Superclass	Entity
Subclasses	-
	Move in one direction, left or right

Class Name	Spawner	
Superclass	-	
Subclasses	-	
	Spawn Enemy fish from the sides	Enemy
	Vary Enemy fish size based on Player	Enemy, Player
	Take care of collisions	Enemy, Player

Class Name	ScreenHandler	
Superclass	-	
Subclasses	-	
	Take care of switching between screens	All the other Screen classes
	Start the game (with MenuScreen showing)	MenuScreen

Class Name	MenuScreen
Superclass	-
Subclasses	-
	Show a play button Show an exit button Show instructions

Class Name	PlayScreen	
Superclass	-	
Subclasses	-	
	Show the 'battlefield'	
	Show player and its movement	Player
	Show enemy fish	Spawner, Enemy

Class Name	WinScreen
Superclass	-
Subclasses	-
	Show that the player has won Show a button to restart a game

Class Name	LoseScreen
Superclass	-
Subclasses	-
	Show that the player has lost Show a button to restart the game

Compared to our current implementation there are very few differences. All of the above classes are present in the actual version, albeit named differently, with the exception from *WinScreen* as we have changed our win condition. All the others have been used and mostly serve the same purpose. We do, however, have 5 additional classes in our implementation. These 5 classes have been created to streamline working with the two different coordinate systems used in a library we use. One of them had (0,0) in the top left corner while the other had it in the bottom left corner. This caused some problems so we have made some classes to unify these coordinate systems. Apart from that the classes are fairly similar to the ones on our CRC cards, so we have followed the responsibility driven design concept fairly well.

Exercise 1.2

The main classes from our implementation are Main.java, OpponentHandler.java, Player.java and Opponent.java

The Main class is responsible for actually starting the game, and for switching between different states such as the menu or playing states. In order to achieve this the class collaborates with the various 'State' classes to swiftly switch between them when necessary.

The *OppentHandler* class is responsible for the majority of actual game logic. It spawns the enemy fish, varying in size and speed, in from the sides of the screen. It also detects collisions between the player and the enemy fish and checks whether either the player loses the game, or eats the enemy fish and grows a little. In order to achieve this the class collaborates with both the Player and *Opponent* classes.

The Player class is the fish which the user actually controls. It is mostly responsible for making the game interactive by being controllable by keyboard inputs, and by growing when smaller fish are eaten. We have also made sure that the movement is very smooth and gradual to make the experience feel more polished. It does not directly collaborate with other classes (Other than Entity, which is its superclass)

The *Opponent* class is the class which is responsible for the enemy fish. Every enemy fish on the screen is an *Opponent* object. An *Opponent* has a size and speed which can vary. The size depends on the size of the player, and the speed is randomly set from around 5 options when the *Opponent* is created It does not directly collaborate with other classes (Other than Entity, which is its superclass).

Exercise 1.3

Amongst the other classes are Entity.java, GameEnd.java, LevelState.java, MenuState.java and ActionLogger.java. All of these classes are less important, they are not fully required to implement the game. The Entity class is a superclass to Player and Opponent, the class is not used in the game itself, it contains the "framework" of the two subclasses. The *WinGame*, *LoseGame*, *MenuState* and Level*State* classes are required for the states in the game. These classes are not specifically required to run the game, these classes are specific to the framework. The game can be implemented in another way, omitting these classes. The ActionLogger class is not required at all for the game, the main purpose of the class is to keep track of all the actions happening in the game, it is for observability purposes only.

Exercise 1.4

Exercise 1.5

Exercise 2.1

The difference between aggregation and composition can be found in the lifecycle of objects that are parent and child. When the lifecycle of the parent and the child are independent , which means the child won't be destroyed when the parent does, this is called aggregation. It is an example of a "has-a" relation. When the parent object is destroyed and the child goes with it, this

is called Composition. The parent object has complete control over the child's create and destroy methods in this case.

In our project usage of composition and aggregation can be found at a couple of places. There exists a composition between the OppentHandler and *Opponent* classes, because the *OppentHandler* class is in complete control over the lifecycle of the instances of *Opponent*. Also, these child instances are destroyed when the *OppentHandler* is.

A form of aggregation that exist in our code can be found in the *actionLogger* class. This class is referenced from other classes, which can be seen as a "has-a" relation. For example, the *OppentHandler* class has a static instance of *actionLogger*. When the *OppentHandler* class is destroyed the *actionLogger* Instance won't be affected by this action. They have a different lifecycle.

Exercise 2.2

There aren't any parametrized classes in our code at the moment. The benefit of using this type of classes is that they are suitable for reusing the class into a different context. You should use parameterized classes in your UML diagrams to make your class definition as abstract as possible. This makes it easier to reuse your code and will result into more adaptable code when it has to change.

Exercise 2.3

Exercise 3.1

See the implementation in the *ActionLogger* class.

Exercise 3.2

Requirements for ActionLogger

Functional Requirements

Must Haves:

- Logger must output to file
- Logger must add a timestamp
- Logger must add the origin (classname) of the log
- Logger must be able to be turned off
- Logger must be able to disable writing to file
- Logger must open a new file with every game start
- Logger must log all errors
- Logger must print additional text if a log is an error
- Logger must log all navigation inside the game
- Logger must log all collisions
- Logger must log all eat and die actions
- Logger must log score
- Logger must log growth
- Logger must log in the format: "[timestamp] – [classname] – [action]"

Should Haves:

- Logger should be able to only print errors

Could Haves:

- Logger could log errors to a different file (logs in log.txt and errors in error.txt)

Won't haves:

- Logger will not manipulate existing log files

Non-functional requirements:

- Logger must be written in Java 1.8
- Logger must be delivered on 18th of September 2015
- Logger must be tested with a minimum of 75% line coverage

ActionLogger	
Superclass(es):	
Subclasses:	
Write given text to file	FileWriter
Get timestamp	Current Time
Open new log file with data & time in filename	FileWriter
Close logger	FileWriter

<<user interface>> **ActionLogger**
- filewriter: FileWriter - dateFormat: SimpleDateFormat
+ logLine(text: String, className: String) + logLine(text: String, className: String, isError: boolean + close() - getTimeStamp()