Assignment 5

by

Group #6

Delft University of Technology

Faculty of Electrical Engineering, Mathematics and Computer Science.

Mekelweg 4,

Delft

Name	StudentId	Email
Clinton Cao	4349024	C.S.Cao@student.tudelft.nl
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

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

Course: *Software Engineering Methods (TI2206)*

Table of contents

A Small Update, reloaded…

In this section we will go through some updates regarding the not yet resolved design flaw (*Schizophrenic Class)* in our system, structure of our packages, the *PlayerFishTest* class and also the *MainScreenController* class.

From the previous sprint we were supposed to use *InCode* to check if our system has any design flaws. *InCode* encountered four design flaws in our system, namely, *Feature Envy* design flaw in *AnimationTimerFactory*, two *Message Chain* design flaws in *MainScreenController* and *Data Class* design flaw in Game.

We wanted to resolve all the design flaws in the last sprint. It all seemed to be going according to plan, but at the last moment one team member realized that our system had a new design flaw, namely *Schizophrenic Class* in *MainScreenController*.

A few members tried to resolve this design flaw, but in the end it was not resolved. The reason was that we had no idea how we could have resolved the design flaw and it was too close to the deadline for the assignment. We were disappointed that we got a new design flaw after we had resolved the previously mentioned design flaws. We decided to leave it in our system and see if we can resolve this design flaw in the last two sprints.

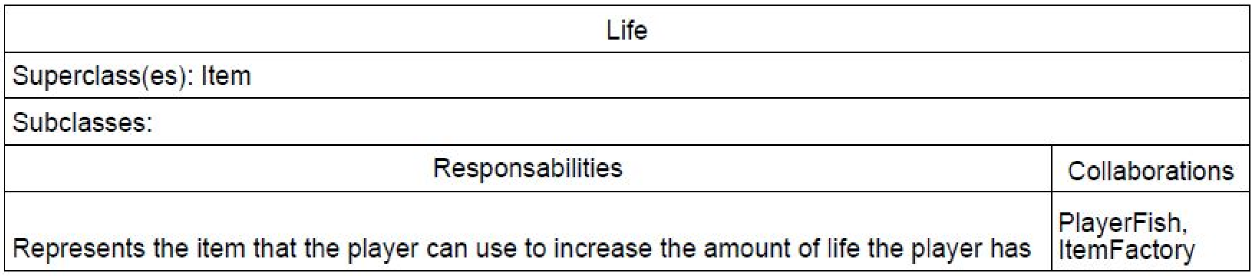
We have resolved this design flaw by creating a new class; *GameLoop*. This class handles all the logic of the game e.g. rendering of the sprites. This removes almost all of the responsibility that *MainScreenController* previously had and thus also making our system less centralized.

We received feedback from our TA that the packages of our repository do not follow the usual Maven packages structure. This issue was not resolved in last sprint, as it took a very long time to try to solve it. After a some researching, this issue is now successfully solved.

In our feedback, our TA also pointed out that there should not be any test class (*PlayerFishTest)* in the main package. The reason that this class was in the main package was that a JavaFx application needs to be started so that we could create an image for the sprite. As the test class starts an Application, it needs to be manually closed so that all the tests can be run. As this might lead to stalling of Travis, when the environment is trying to build our system, we decided not to put it in the test package. Our TA suggested to use a library to test this class, as this test class needs the GUI to finish its tests. We have done some research and we decided to use TestFX to help us test this class. At first there were a lot of build failures, as the environment didn't open a display for the second build job, but this was fixed after configuring the *.travis.yml* file.

(Chapter 4:)[1] A New Feature…

Following RDD and using CRC cards, we have derived the following class from our requirements of this sprint:



Life	
Superclass(es): Item	
Subclasses:	
Responsabilities	Collaborations
Represents the item that the player can use to increase the amount of life the player has	PlayerFish, ItemFactory

The life class is an item that the player can use to increase its life in the game. Thus this class will have the responsibility to represent the item that the player can use to increase the amount of life the player has in the game. Of course the superclass of this class will be the *Item* class. As the player is the only one who can use the life item in the game, it will collaborate with *PlayerFish*. *Life* collaborates with *ItemFactory*, as this is the class that creates the the items of the game.

Actual implementation

Comparing with our implementation, we implemented the *Life* class just like what we have derived with RDD and the use of CRC cards. We applied the singleton design pattern to *Life*, as we think there should be just one object in our game. Every time that the item spawns on the screen , it is the same object. Because of this, we don't have to keep a list of life items; we just keep the number of lives that the player has and every time that the player picks up the life item, the numbers of lives is incremented, that is if the player does not already have the maximum amount of lives (3).

[1] This is not chapter 4 of this report, but does anyone get the reference? ;)

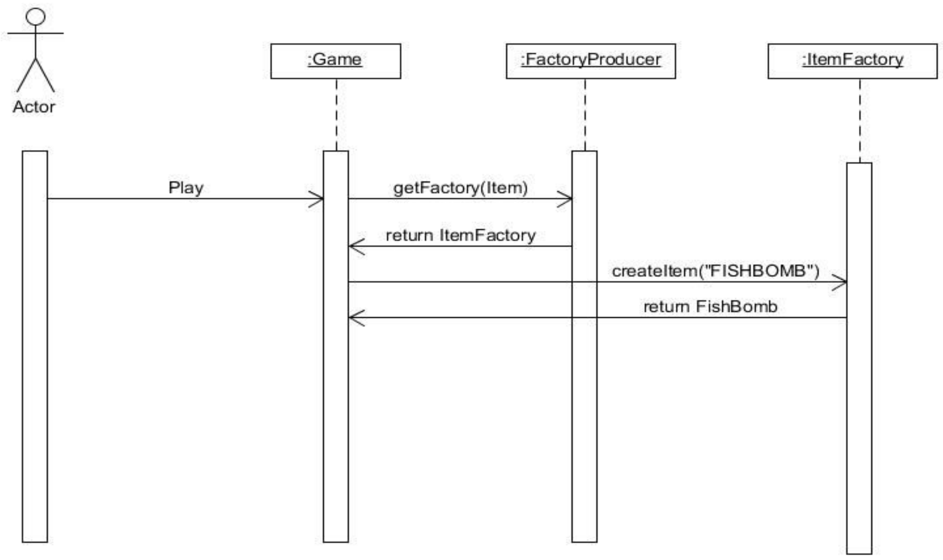
Exercise 2 - Design patterns

2.1 Abstract Factory Pattern.

We chose the abstract factory design pattern because it was a natural follow-up from the factory design pattern which we had used in an earlier sprint. It also gives a nice cohesive way of creating all factories by having a “super-factory” creating those factories.

It is implemented by creating (as implied by the name of the design pattern) an abstract factory class which is extended by all the other factories. The abstract factory has the functionality of all the different factories, meaning that the main factory method for each specific factory was also implemented in all the other factories, but there they will simply return null. Next the “FactoryProducer” was implemented, which is simply put a factory that creates all the different factories. This “FactoryProducer” consist of a simple switch which is in line with all the other factories.

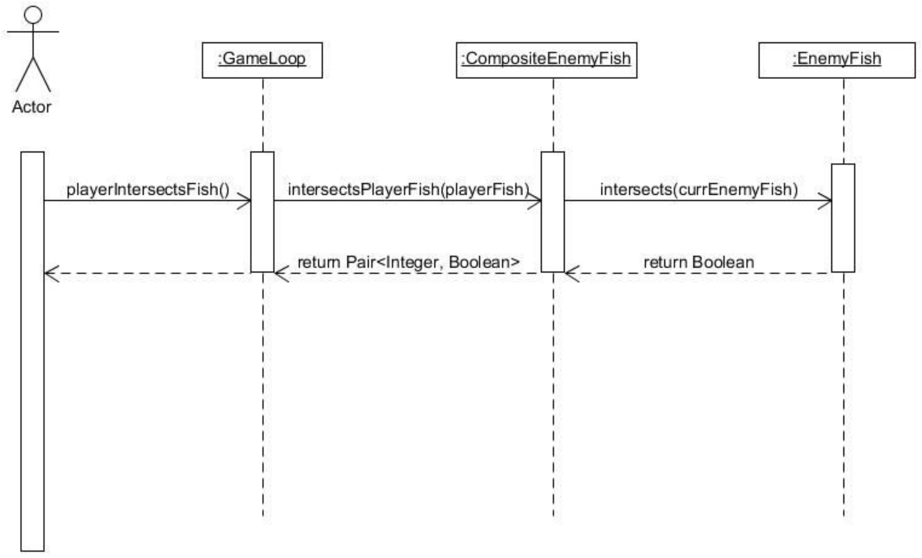
The sequence diagram below is specifically made for the ItemFactory, but works for any of the subclasses of factory.



*Please look in our repository (Assignment 5 folder) for the class diagram. Both the sequence and the class diagram has been simplified, so it doesn’t look cluttered and to avoid confusion.

For our second design pattern we initially chose decorator. We wanted to apply the decorator design pattern to our item class. We were thinking something along the lines of creating a general item and then adding the specific attributes we needed to make it a specific item (like a life powerup). We chose a little hastily however, because after some reading and some looking at some example code, it turned out a decorator does not do that. Fortunately we figured this out the on tuesday afternoon, so no real time was lost. Instead of the decorator pattern we switched to the composite pattern.

In fishy the PllayerFish constantly interacts with a whole list of enemy fish. This lead to a lot of for loops in the game logic. For checking intersections with the player fish, and for removing enemy fish which were off screen, and for checking intersections with the fish bomb explosion, etc. The composite pattern actually turned out to be a great idea for our code. Instead of all these pesky for loops strewn around everywhere, we could just introduce a new object; CompositeEnemyFish. Now all our other objects like the PlayerFish and the screenbox can just interact with the composite enemy fish, but all the ‘responsibility’ of looping over all the enemy fish is ‘outsourced’ to the composite class. This really cleans up the code. We are really big fans of this design pattern! Below is the sequence diagram:


Actor
:GameLoop
:CompositeEnemyFish
:EnemyFish
playerIntersectsFish()
intersectsPlayerFish(playerFish)
intersects(currEnemyFish)
return Pair<Integer, Boolean>
return Boolean

*Just like the previous sequence diagram, this one is also simplified so it does not look cluttered and and to avoid confusion.

Exercise 3 - Wrap up – Reflection

Through almost eight weeks of Software Engineering Method lab, we have a learned a lot as a team. Firstly, we learned how to apply SCRUM methodology to our project as a team, where we first construct an initial working version of our project and then improve from that. Instead of following U-Boat design, which only resurface at the end of the project, meaning only get a working version for the end project. Our working pattern followed dolphin model, because we kept resurfacing and building a better working version of the project every weekly iteration, taking our TA's feedback into account.

We learned how to respond fast to changes, such as unexpected bugs in our code or different requirements of game features from TA. We also learned to value the feedback we got after every iteration and improve our project based on the feedback and also learn from our mistakes.

Secondly, we learned how to understand the problem and agree upon the nature of the problem to produce requirement documents. We wrote requirements for our project and new features implemented in every iteration. We have also learned to use the help of tools such as Class Responsibility Collaboration(CRC) cards and UML diagrams in the process of finishing our project. We derived classes from CRC cards and compared it with our initial version and improved our project based on that.

Next, we have also learned about how to review our code as a team, and what are the good practices of doing code reviews, such as we shouldn't review too many lines our code per member of the team, and we should take our time and look carefully. We also learned to annotate the source code before the other members start to review the code. And it resulted better code quality from our team.

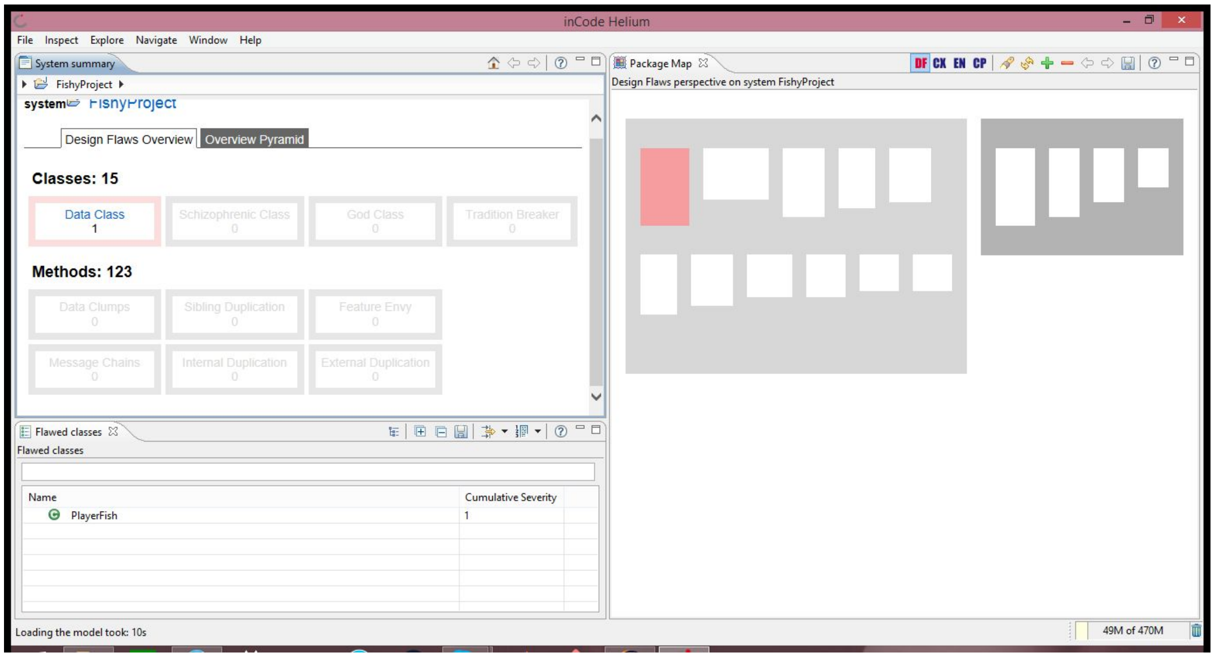
Furthermore, we learned how to apply different design patterns to our code, so we can program in an efficient way and increase the usability and maintainability of our software. For example, we applied composite design pattern to our code; we made a new *CompositeEnemyFish* class which we use in the *MainScreenController*, so it does not interact with multiple EnemyFish objects, but just one *CompositeEnemyFish* object. We used the Singleton design pattern for objects of which we only needed one, like the playerfish, the end boss, the lance and the life item. We used factories for the creation of many of these objects (and for, for example, generating enemy fish). And because we ended up with so many factories, we ended up implementing an abstract factory design pattern.

And last but not least, we learned to use software metrics through the *inCode* tool to detect design flaws in our code. Through detecting design flaws and fixing them, we have now a more insightful knowledge about software engineering of the project.

All the aforementioned skills will give us a huge benefit when it comes to creating software from a client's idea. Looking at the progress we managed to achieve during development by comparing what we handed in after the first two weeks and the version we currently have, we can say that the lab gave us a new viewpoint on how to efficiently build software and how to make the code in it expandable without the need of a total refactor, allowing us to further develop specific features in Fishy without wasting time.

When it comes to what we learned as a team, we learned how to implement different design patterns to make our code more clean, this will allow us to implement these patterns during the initial release, giving us the remainder of the time implementing new features.
These design patterns also allow us the ability to extend our game without changing the code.
Another thing we learned is how to manage our time during sprints, a skill which is gonna be essential to allocate enough time for each task to be completed and give us time to fix any errors that have been found last minute.

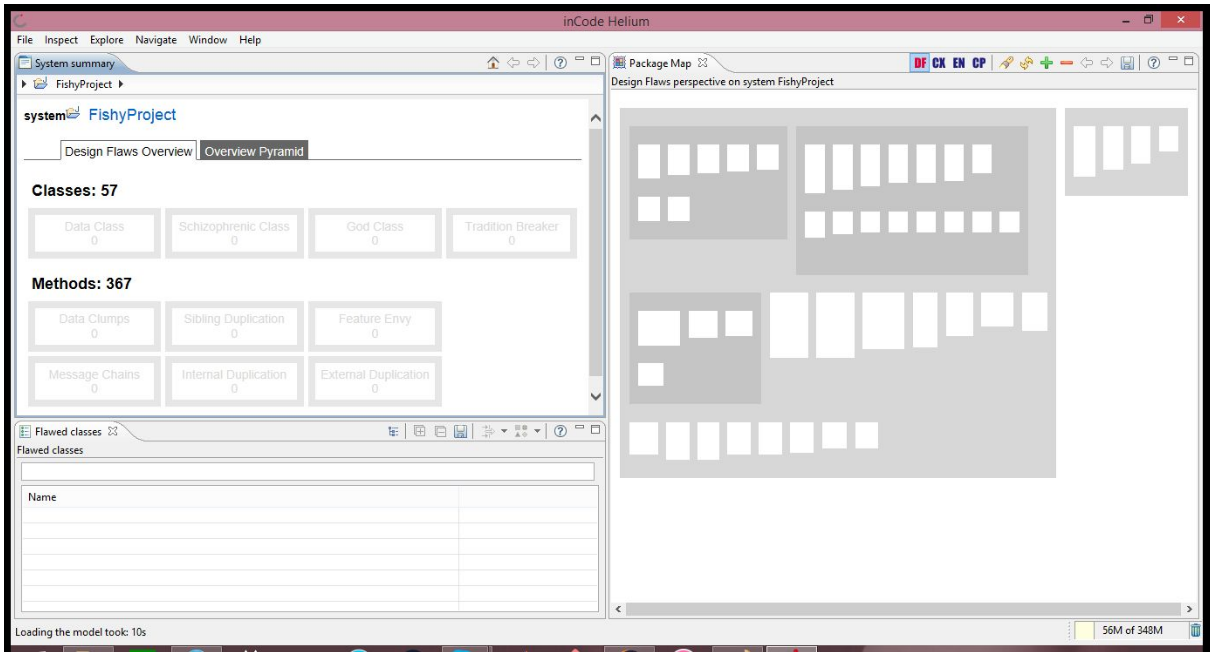
During the project, we clearly saw how the game was evolving and developing from the first release to our current version. In the first release, our game was pretty simplistic: it had no clear goal or any specific bonus features. The game would just continue until the player fish eventually died. The simplicity can be also seen in the way the code was written in the first release:



Name	Cumulative Severity
PlayerFish	1

As can be clearly seen, the code is divided within the test and main package, and the main package has classes without any sorting system. What was surprising was the fact that the release only had one design flaw, namely a Data Class Flaw.
In the initial release, there was also the issue that the MainScreenController class had a lot of responsibilities, violating one of the SOLID Principles. The reason was because the class took care of both the rendering of enemies and handling their intersections with the screen or the player.

When we look at the current version, we can see how the project became more complex:



system FishyProject

Classes: 57

Data Class	Schizophrenic Class	God Class	Tradition Breaker
0	0	0	0

Methods: 367

Data Clumps	Sibling Duplication	Feature Envy
0	0	0

Message Chains	Internal Duplication	External Duplication
0	0	0

Design Flaws perspective on system FishyProject

Loading the model took: 10s

56M of 348M

We can see that the classes became properly sorted and divided in different packages, the design flaws are gone and the way methods are implemented became less straightforward. The code now also implements different design patterns, which it did not have when the game was made during the first two weeks.
The issue with the MainScreenController having multiple responsibilities was also fixed by creating the GameLoop class, a class that handles the logic of the game i.e. MainScreenController has to render all the sprites, handle the collisions between the fishes, and also handle the modes of our game. With the new GameLoop class, most of these responsibilities has been removed from MainScreenController and is now shifted to the GameLoop class.The addition of new features, such as powerups and the ability to save and load the highscore also contributed to the improvement of the project as a whole. So in conclusion, we have learned a lot from this project e.g. we understand not just the theory of the design patterns, but also how we can apply it in practice .