Assignment 3

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

Looking back at our code..

Looking at the code we have now, these are the actual responsibilities of each class:

- Green: Correct responsibility.
- Purple: Might warrant another class or refactor in the future. Might also be left as is.
- Yellow: Will be fixed by implementation of factory design pattern.
- Red: Needs to be refactored or relocated.

Class	Responsibility
BoundingBox	Represent a rectangle in a 2-D plane Handle intersections. Handle movement of bb's.
EnemyFish	Represent an EnemyFish. Generate EnemyFish.
Entity	Represent Entity. Handles intersections between Entities.
FishBomb	Represent FishBomb. Generate new FishBombs. Handle FishBomb movement. Handle rendering. Handle intersections.
Game	Launch the game. Switch Screens. Contains a lot of data.
Logger	Log actions of player.
PlayerFish	Represent PlayerFish. Generates PlayerFish. Handle growth of playerFish. Handle player death. Handle generation of fishbombs for the player.
Sprite	Represent ingame model. Handle movement. Handle rendering.

LosingScreenController	Controls the losing screen input.
MainScreenController	Control the main screen. Handle generation of fishbombs for player. Renders all objects on screen. Handle playerfish input. Contains game loop. Generates enemyfish.
WinningScreenController	Controls the winning screen.
OptionsController	Controls the options screen.

Exercise 2 - Design Patterns

Singleton pattern

Singleton design pattern is implemented in our code because of its usefulness, it allows the creation of one instance of a specific class and one only. In our Fishy game, we have indeed some classes which we only want one instance of these classes, for example, as stated in our requirement, *PlayerFish* class and *Logger* class.

For *PlayerFish* class, we only allow one player fish exist in the entire game, therefore Singleton design pattern guaranteed that at any time there can be only one instance of *PlayerFish* object created. Same principles apply for *Logger* class, at any moment of the game, there should be only maximum one *Logger* class exist.

Singleton design pattern is implemented by firstly, adding a static variable in both classes to hold the one instance of class *PlayerFish* and *Logger*.

Secondly, the constructors of both classes were changed from public to private, so that only they can instantiate themselves.

Lastly, we implemented getInstance() methods for both classes, so it gives us a way to instantiate the classes and also to to return an instance of them.

Factory design pattern

The factory design pattern is very useful in our game, as there are entities and items in our game. Using the factory design pattern, we can create classes that have the single responsibility of creating the right objects for our game. Therefore the entities classes and item classes does not have the extra responsibility to create themselves.

We implemented this design pattern by creating an *ItemFactory* and an EntityFactory. The EntityFactory is responsible for creating the right entities (*PlayerFish* or *EnemyFish)* and the ItemFactory is responsible for creating the right items (*FishBombs)*. So when the game is initializing the game objects in the *init* method, the factories will create the object for the game. As there should be only one *EntityFactory* and one *ItemFactory*, we also applied the singleton design pattern to these factories.

The above mentioned factories are not the only factories we created in this sprint. As extra we also implemented several other factories (visible in our source code folder) that we think it might be helpful to make our code easier to read.

* Please have a look in our repository for the new class diagram and sequence diagram after we applied the singleton and factory design pattern.

Exercise 3 - Software Engineering Economics

Recognising Good and Bad Practices

Good and Bad practices are recognized by comparing how much time and money was spent on the project compared to the average among the project's size in function points. Good Practices indicate that the team spent less money and took less time than what's expected with the average. Bad Practices indicate the exact opposite: the team spent more time and money than the expected amount based on the average.

Visual Basic

According to the paper, the reason why Visual Basic is not such an interesting find is because in many cases, it's more about how companies deal with "shortage versus availability" cases rather than the implementation itself.

Additional Factors

3 additional factors that can be used for either Good or Bad Practices are the following:

1. **Tool-assisted Code review**: A technique that could be tested is the usage of tool-assisted code review during development to assist the team. This factor would belong to Good Practices as it will save time for code review and is autonomous, meaning that it does not require to set up a concrete meeting to do it.

1. **U-boat development**: U-boat development is when the team working on the project only releases the product once, and that is when it is finished. This factor would belong in the Bad Practices as this type of development heavily reduces the amount of feedback the team gets from the client, making it difficult to edit the product if the needs of the client change.

1. **Use of Design Patterns**: The use of Design Patterns can facilitate the creation and improvement of the project, allowing to save time when extending code that would otherwise be used refactoring the code to allow those extensions. That reason is why this factor would belong in the Good Practices when comparing projects.

Bad Practice Factors

Once-only Projects: Once-only Projects mean that the development team that wants to start the project up has never worked on something similar before and will only be doing this type of work once. The reason why this belongs in bad practices, is mainly due to the fact that the team is inexperienced in the project's “field”, which results in a lot of defect being created and mistakes being made, reducing the quality of the project.

Dependencies with other systems: This factor indicates that the project relies on a system other than itself to function properly. The reason why this belongs in bad practices is because the dependency heavily limits the freedom that the development team has during implementation.

Rules and Regulation Driven: This factor indicates that the project need to follow specific external protocols bound by law. The reason why this applies to bad practices is because, like dependencies with other systems, this heavily limits the development of the projects.