Requirements
Fishy Sprint 3

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)*

Table of Contents

1 FUNCTIONAL REQUIREMENTS FROM TEAM

The functional requirements of the features implemented by our group for the game “Fishy” are described below. These requirements are divided according to the MoSCoW methods. The functional requirements are for the improvements that we are going to make in sprint #3.

1.1 Must Haves

- Classes must have one clear responsibility each, according to responsibility driven design.
- Methods must be of a reasonable complexity and length.
- All data must be in the right place (according to each class’ responsibility).

1.2 Should Haves

- A package containing interfaces for each class.
- These interfaces have comments for each method, clearly explaining their purpose.

Since this improvement has a clear goal of refactoring the code into a much better state (which follows RDD), we do not have any could/won’t haves.

2 FUNCTIONAL REQUIREMENTS FOR APPLYING SINGLETON DESIGN PATTERN

The functional requirements for the applying singleton design pattern to our code are described below. These requirements are divided according to the MoSCoW methods.

2.1 Must Haves

- It must ensure that only one instance of *PlayerFish* class is created.
- There must be maximum of one instance of *Logger* existing in the game at any given time.
- It must prevent the simultaneous invocation of the getter method by 2 threads or classes simultaneously.
- The *PlayerFish* and *Logger* classes' default constructor is made private, which prevents the direct instantiation of the object by other classes.

2.2 Should Haves

- It should provide a global point of access to the *PlayFish* object.
- It also should provide a global point of access to the *Logger* object.

2.3 Could Haves

As we are trying to apply the singleton design pattern according to what we have learned in the lecture, we can't think of any could haves for the singleton design pattern. We think that we are supposed to follow the rules of this design pattern, as taught in the lecture.

2.4 Won't Haves

- We won't create a new *PlayerFish* or *Logger* object in every class that requires it.

3 FUNCTIONAL REQUIREMENTS FOR APPLYING FACTORY DESIGN PATTERN

The functional requirements for the applying factory design pattern to our code are described below. These requirements are divided according to the MoSCoW methods.

3.1 Must Haves

- Instances affected by the factory design pattern must be made through a factory class.
- Classes affected by the factory design pattern must implement an interface.
- The factories only responsibility must be to create instances of other objects.
- There must be only one factory implemented for each interface.

3.2 Should Haves

- In the case of multiple factories, the abstract factory design pattern should be followed.

Because for this assignment we are applying the design patterns to our code, we couldn’t think of any could or won’t haves.