Assignment 5 - Group 26

Exercise 1 - 20-Time, Revolutions

In assignment 4 we were required to list the requirements for the TA exercise. At the meeting with the TA on 20/10/2015 we were told that we could finish this feature as the 20-time in assignment 4. These requirements are therefore not very new, but a few changes have been made. They are shown in italics. The most important thing we'll work on is fixing the client receiving settings and allowing the client to move.

The requirements are listed in the same way as our game, so we will use the MoSCoW method. Firstly, the functional requirements of the logging are listed as must, should, could and won't haves. Secondly, the non-functional requirements are listed.

Functional Requirements

1.1 Must haves

- The game Fish.io must have non-local multiplayer.
    - This must support multiplayer for at least 3 clients.
    - The connection will find place through TCP/IP connections.
- The implementation will be socket-based and will use a server that distributes the necessary information and environment variables.
- The following actions must be implemented:
    - Every client must be able to control its own playerfish.
    - Players should be able to eat another player if they are big enough according to the game rules.
    - Smaller AI fish must spawn and move so that the players can eat those instead if they cannot eat players.
- Multiplayer must be implemented into the GUI.
    - There must be a multiplayer screen on which the player can connect to a multiplayer game by connecting to a specific IP.
    - There must be a button in the main menu that links to the multiplayer screen.

1.2 Should haves

- *Multiplayer mode should have a background.*
- *The client should be able to differentiate different users by seeing them with a different color fish.*
    - *All player fish should still have the original player fish sprite.*
- The settings should be the same on all clients.
    - *The server will send a settings message to a client when that client connects.*
- The playing field should be bigger than a single player playing field..
- The user's view of the playing field should be able to move within the boundaries of the playing field.

    - This is called a viewport and will also work for a single player game in case the screen is smaller than the field.

1.3 Could haves

- The server could spawn certain power ups that players can collect.
    - These might have to be rebalanced in order to be fair to other players.
- There could be extra entities like a large fish that is able to eat all players.

1.4 Won't haves

- The multiplayer implemented won't be local multiplayer version (2 players on one machine).
    - (The items below are requirements of local multiplayer):
    - The user shall be able to start a local multiplayer game using a "multiplayer" button on the start menu.
    - The local multiplayer version shall work similar to the single player version. A few differences:
        - The controls for the extra player(s) will differ from the controls for the first player.
        - Different player fish can have overlapping bounding boxes without any result.
- There won't be any extra entities like:
    - Sea urchins: When the player is bigger than a sea urchin and collides with it, the player gets killed instead of eating it.
    - Obstacles: the player must move around obstacles and cannot move through them.

Non-functional requirements

- The second, working, version of the multiplayer mode shall be ready and submitted to the staff of the course on the 23rd of October.
- The multiplayer will not be tested with the same coverage as the total game (75%), as this is very difficult to do right on travis and it would take too much time to implement.
- The components that can be tested easily on travis will be tested with the same coverage as the entire game (75%).
    - These components will be identified as multiplayer is implemented, and will be noted in the UML.
- The multiplayer implementation will have to adhere to the same non-functional requirements of Fish.io itself, with exception of the modified non-functional requirements in this document.

Please see the CRC path of last week multiplayer. There have been no changes to the responsibilities of multiplayer.

Exercise 2 - Design patterns

Factory pattern:

EnemyFish Factory Pattern

1) *Natural language description of implementation.*

The main reason for wanting factories is simple: moving responsibilities from one class to another. The code is much more structured if you put complicated methods like createRandomEnemyFish or createRandomPowerUp in the EnemyFish class or the PowerUp class. Those classes will then have too many responsibilities, violating the first *SOLID* design principle.

The factories were easily implemented. The were assiociated with certain “Spawner” classes that automatically put the created objects in a PlayingField. So the Factories have no direct contact with the PlayingField. The Spawners use the Factories to place in the PlayingField.

class diagram:

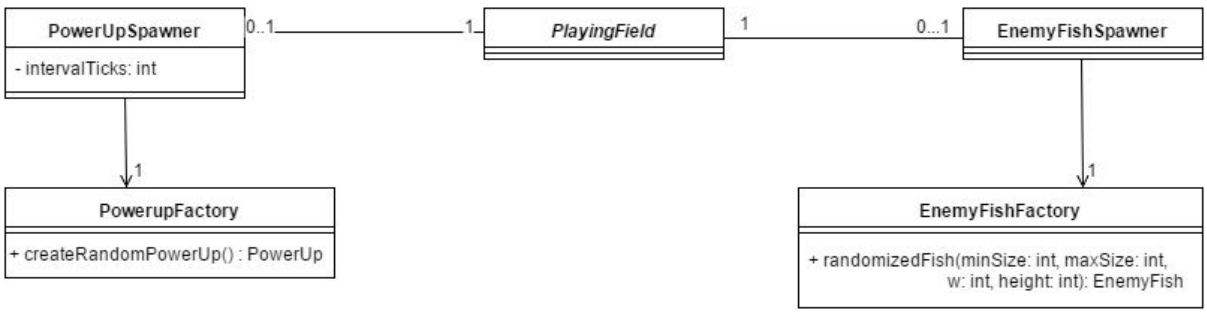

Figure 1: The class diagram of the factory pattern implemented by the EnemyFishFactory.

sequence diagram:

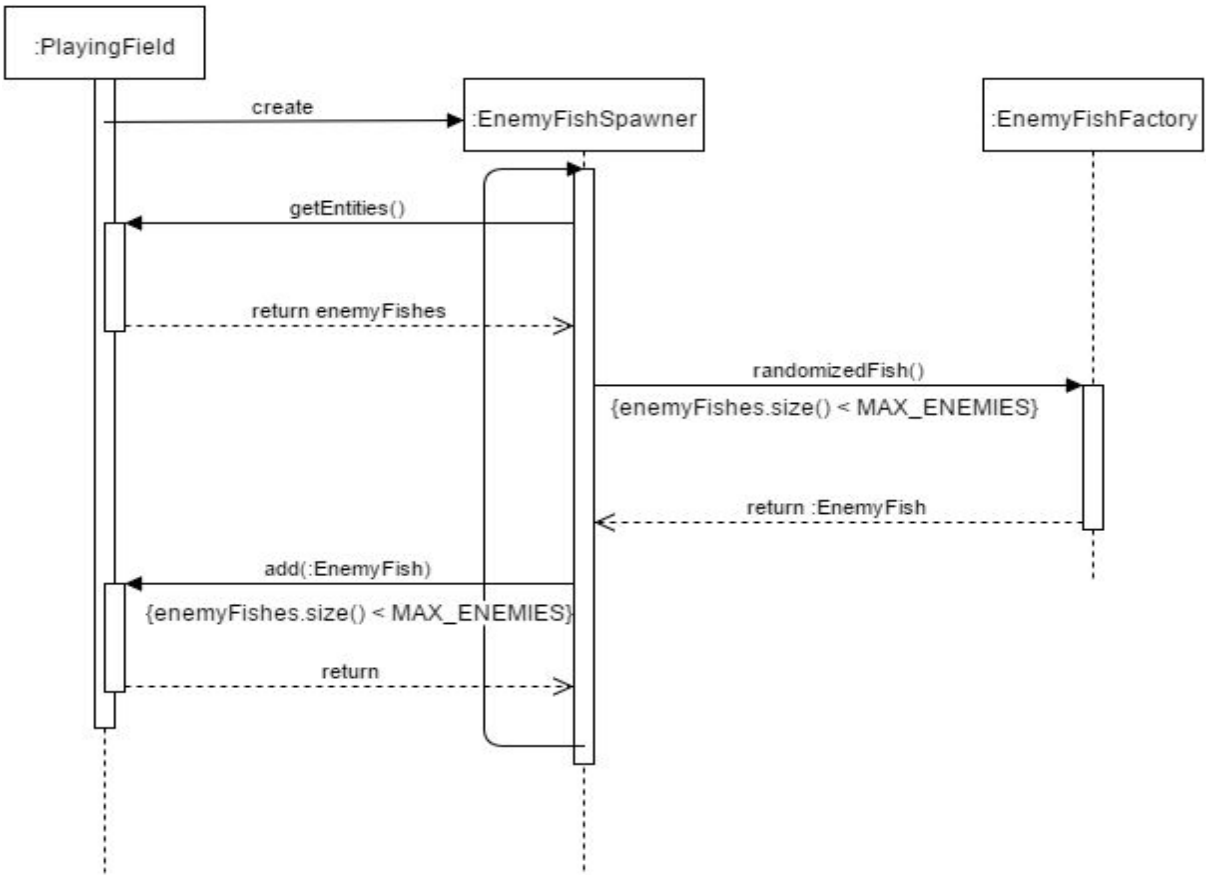

Figure 2: The sequence diagram of the factory pattern implemented by the EnemyFishFactory.

Observer pattern:

Observer Pattern Ticks

1) *Natural language description of implementation.*

When we implemented the basic structure of our game, we needed a way to synchronize and organize the updating of the entities in the game. We also wanted to be able to execute some methods before updating the entities or after the updates.
The nicest way to implement this is using game ticks. These game ticks act like a clock for your system. All the entities then would listen to this tick and then update. This way all entities are updated in an organized way. This also allows for changing the time between the updates of the listeners by simply changing the times between two ticks.

The part for updating entities was implemented by calling methods in the PlayingField each tick. These methods move the player, add new enemies and remove the dead entities.
The ticks at the beginning (the pre-ticks) and the ticks at the and (the post-tick) are implemented using the observer pattern because this pattern facilitates this behavior by definition. In an observer implementation there are two types of classes. The first is the Observable (in our case the clock) and the second is the Observer (in our case the classes

that listen to the ticks of the clock). The Observable has a list of Observers that are notified about changes using a method call on the Observers. Observers can register themselves as new Observers to the Observable to be added to this list or unregister to be removed from the list.

In our project two classes implement an Observable called Listenable. These classes are the GameThread and Renderer. Classes implementing the TickListener interface act as observers in our implementation. These classes are PowerUpSpawner and PowerUpDuration. What these classes have in common is that they use a duration to time their actions. By counting the ticks they can precisely calculate the durations.

Class Diagram:

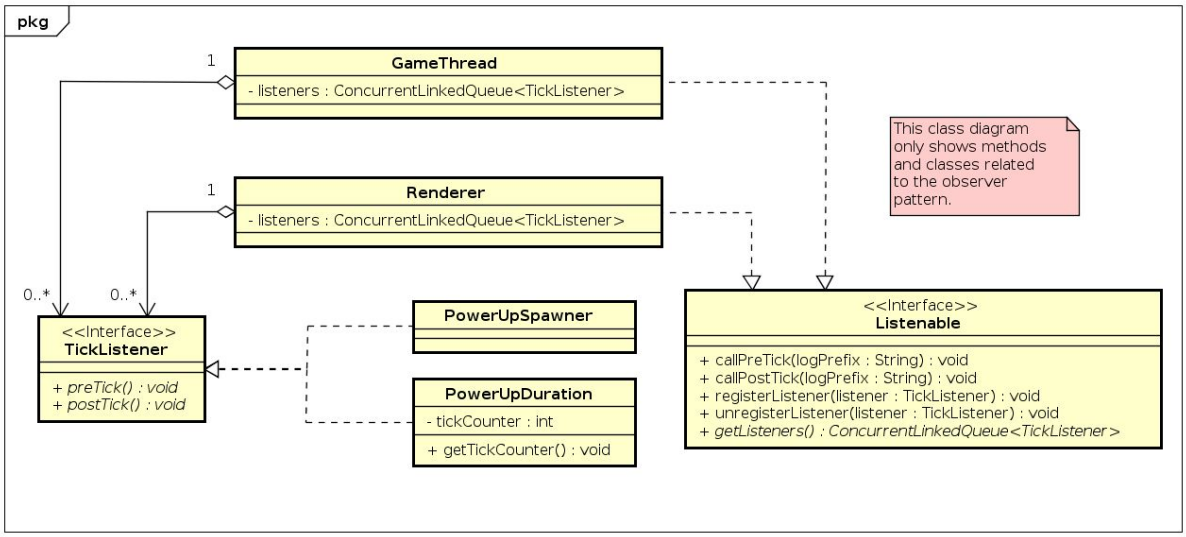

Figure 3: The class diagram of the tick implementation. For the explanation of the implementation please see the previous exercise. A full sized version can be found on the GitHub repository.

Sequence Diagram:

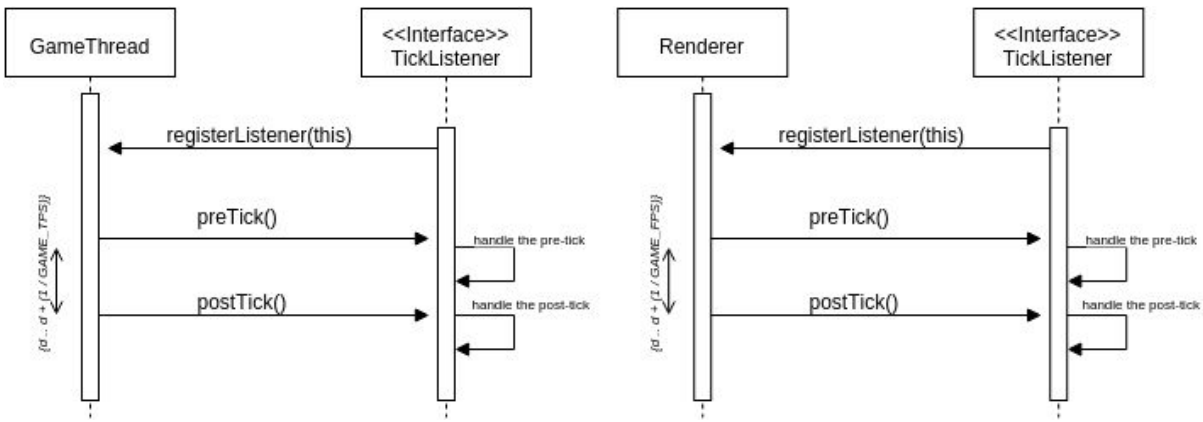

Figure 4: The sequence diagram for the interaction between the two listenable classes and their attached TickListeners. For the explanation of the implementation please see the first sub-exercise. A full sized version can be found on the GitHub repository

Exercise 3 - Wrap up – Reflection

Before we can start on the reflection, we will explain the structure of it briefly. The first half of the reflection consist out of a progress report on our process throughout the project. This half contains a walkthrough through our source code progress and contains a list on things that we improved on. The second half is split into three parts: a summary on what we learned during the lab, followed by our thoughts on what we learned on working in a programming group and lastly, a word on how we will use what we learned in future to design and implement software projects. Note that we don't count this structure explanation into account for the word total.

We would also like to note that three of our group members had a quite hefty deadline on the same day and because the assignment was not put online in the weekend, almost everything had to be done during this week. This means that did simply not have time for creating some graphs about the grades going up, adding a small video of our git project history generated by a tool (Gource) and some pictures of inCode comparing significant changes. We apologize for this missing content.

Reflection (1020 words long):

Starting from the beginning of the course, we established our source code by having one team member create a general *skeleton* for our game. This skeleton contained a few basic classes and appropriate folders for different kinds of resources and documentation. After the first few commits the rest of the team joined in and started to work on independent features of the base game. At the end of assignment zero we had created a fishy-clone with basic features *such as swimming around on a background and the ability to eat fish smaller than your own fish*.

Continuing, in assignment one we implemented a robust logging framework for our game, we added a *help screen*, added *the ability to revive* and we started working on implementing *achievements* in our game. In assignment two we continued working on the *achievements* implementation, added *power ups* and did some *code refactoring* as ordered by our TA. Then in assignment three we again continued with *achievements*, as the feature was more extensive than we thought. Furthermore, we added a *sound engine*, a *settings screen* and did some needed *bug fixes*. In assignment four we *refactored* a bit of our code with the help of inCode and we introduced rough implementations of *screen resizing* and *multiplayer* due to time constraints.

Currently, during assignment five, we are focusing on improving the *multiplayer features*, implementing *high scores* and *refactoring*. Next week, we will have to deliver the final version. Thus, in the following sprint we will try to *create better coverage*, *finish up features* like achievements and might implement one final feature: *a boss fish*.

Throughout the course, we think that our quality of our project work went up a lot. The quality of code improved throughout the weeks. Especially after enabling PMD and FindBugs, which we forgot to enable for the first two weeks. Adding to that, in assignment two, three and four we had to do some kind of refactoring as part of the assignment, which also improved our code.

The improvement of the code can be found back in the grades, which were good from the start, but improved over time. This mainly has to do with our development process. At start of the project we did not have much of an idea of what SCRUM meant or how to use continuous integration development correctly. During the course we had lectures on SCRUM, requirements engineering, responsibility driven design, advanced OO and Code review. Each lecture had a positive impact because we discussed these principles during our daily SCRUM meetings and tried apply them to our current development process.

As a closing note on the project progress and last but not least: testing. We always had some tests in our code, but the unit testing coverage went up and down quite a lot. This mostly had to do with addition of hard to test features like multiplayer, and due to being unable to test a large part from the GUI (on travis). We did manage to break the 75% coverage mark several times, but with the new Multiplayer feature it might be hard to keep that mark intact.

During the lab we tried to ensure everybody was doing something that they were good at, and then try to alternate this with stuff that they didn't really understand yet. This meant that everybody had chances to add features at which they were good at, but it also gave everybody a chance to figure out stuff which they hadn't mastered yet. We think this is a good methodology to follow when studying at an university.

In the lab we learned a lot about both the documentational side and the practical aspect of development. To start with, we learned how to use UML to somewhat effectively to show how classes, and responsibilities between them, exist in the project. We learned how to write requirement documents as contract. At practical aspect we learned how to do a basic SCRUM sprint and how keep track of progress of sprints items. We learned how we can implement certain design patterns into our project to improve the extensibility of our project. And lastly, ways to improve our code reviewing process while doing pull-based development.

During the project we also gained quite some experience as working in a development team. During the course we tried to hold up the SCRUM methodology from day one. Sometimes it was hard to do a daily sprint meeting, especially in the weekends. So, later on we decided that we shouldn't do the meetings in the weekend so we had less restrictions in total. We didn't switch roles that much during meetings, but we did try to improve the meetings so that we could hold them in 15 minutes or less

Some other things that we learned in cooperation were code review, helping people in need and how to keep up a good working atmosphere. Code review at start of the project was pretty bad, but (mostly thanks to the lecture on code review) we started really checking each other's code for improvements and constructive criticism. During the project there several times that people couldn't finish their assignments but with the help of the team, we were able to finish them on time. And finally, we had a lot of fun during the project and so far, we really liked what we have created as a team.

In the future, we will we think it would be a great idea uphold many of the matters and methodologies discussed during the course. We think SCRUM does have it valid points but we shouldn't follow it to the letter as it is a way to help development and not to delay it by taking too much time away from programming. Perhaps the most important point is that we

should be careful on how much time we plan in for each team member. We enjoyed this course thoroughly, but we spent way too much time on it. We sometimes had to forsake other courses in order to get everything done that we planned. This resulted in stressful situations and thus we should all really be more careful at the planning meetings.