Project Reflection

TI2206 Software Engineering Methods (2015-2016 Q1)
Date: 30/10/2015
Group Number: 0
Group Members:

Thomas Baars - tcbaars@gmail.com
Lars Ysla - yslars@gmail.com
Boris Schrijver - boris@radialcontext.nl
Adriaan de Vos - adriaan.devos@gmail.com
Dylan Straub - d.r.straub@student.tudelft.nl

Table of Contents

Reflection

Project Description

As part of the course TI2206 Software Engineering Methods, we had to choose a game out of 6 possibilities and create a first working version within approximately 10 days. We chose to implement the Fishy game[1]; which is where the player controls a fish and tries to consume smaller fish to grow in size, while trying to avoid larger fish. Each fish the player consumes contributes to its score. If the player is consumed by a larger fish then they lose, on the other hand if the player reaches the maximum size, then the player wins.

The main priority for delivering the working version, was to first create a requirements document which outlined the functional and non-functional requirements of the game, as well as assigned each requirement a priority. After we finished, and had the requirements document approved, we started working on implementing the first working version of our game based on these requirements.

First Working Version

The first working version of the game meet the majority of the requirements set forwarded by the requirements document. However there were a few large design issues, which were very related to the scalability of the product. The first issue was the fact that there were many classes that had many responsibilities, the main example of this were the classes in the layer package which had the responsibility of drawing and managing the currently displayed layer.

The second issue was the ‘hacky’ way we changed between layers, by linking the desired layer and unlinking the current layer. It was implemented this way to be scalable, by allowing multiple layers to be linked and one could then easily return to the previous layer; however it did not work as intended, so the previous layer had to be unlinked.

The third issue was the use of private final static variables which were used as constants to allow the classes to be easily customised; but these made the classes large and were often copied and pasted over multiple classes.

Iterations

The next few weeks we had to adopt the SCRUM methodology to improve our game. Each week we would have an assignment which would outline the tasks our sprint had to include.

[1] Fishy gameplay video - https://www.youtube.com/watch?v=G3vC6x7CTSIh

And each week we would have to create a sprint reflection of the previous sprint, and create sprint plan which included the tasks for the next sprint. Having these meetings gave us time to look at our current situation and plan adjustments based on the feedback for the next sprint.

Each iteration you could see that the project was evolving, not only in terms of the added features, but also in the code structure. A lot of these improvements were made from comparing the current structure to what it would have looked like if we had followed a responsibility driven design. This included reducing the number of responsibilities each class had and looking at how they collaborate.

The result of this was an implementation of the state design pattern, with a state manager to make changing between the states easier. That way the state would have responsibility of performing the operations necessary to keep it up-to-date, and it would collaborate with its layer to draw the current state.

We also made some improvements to reduce the overall size of the classes. Such as splitting large classes into multiple classes, removing the public static final attributes, and creating utility classes for commonly used functions. The size of the classes where also reduced due to the practices set forward in the implementation of the design patterns. For example implementing the adapter design pattern and implementing multiple singletons, made class' responsibilities more atomic and made access to global information easier.

Comparison to the requirements

The main goal of each iteration was to fulfil the requirements specified, which includes those specified in the initial requirements document and those related to the extensions added each iteration.

Looking back upon the original requirements document, there were fourteen requirements that were classified as "must-haves". All fourteen of these requirements were met. In the "should-haves" category, there were eight requirements, all of which were also successfully implemented in the final version of the game. As for "could-haves", there were no fewer than 21 requirements specified in the original requirements document. Within this category, there were a few requirements which were not fully implemented in the final version.

1. Moving tails: Although we *did* ultimately have fish with moving tails, this was true only of the enemy fish, not of the player's fish.
2. Orientation-dependent consumption: We did not make the fish's ability to eat other fish dependent upon its orientation. In our game, the player's fish is able to eat other fish with its tail.
3. Progress indicator: although our game *does* display the number of fish eaten, this is not done with images of fish skeletons as we had originally envisioned

4. Hunger Indicator & size decrease: We did not implement a hunger indicator or a time-dependent decrease in the player's fish size

The rest of the "could-haves" were successfully implemented. The "would/won't-haves", however, were not implemented, for the most part. Only the power-ups feature was partially implemented. With the exception of 75% test coverage, our non-functional requirements were successfully implemented.

During the project we made use of a number of helping programs to help build the project and oversee the overall process and workflow. There were two problems encountered with travis. Travis did not have updated sound drivers which gave problems with the bubble testing sounds. Travis also gave a false passing build when there were errors in our tests for the player. As the project went on we implemented different features such as multiple enemies and a power up. We also made a lot of adjustments to improve the structure of the code and make future features. Some of the initial requirements that we wanted but not needed we did not implement as the priorities of the project was focused on the structure and testing.

Low test coverage

There were several reasons for the relative lack of test coverage.

- Started from the first working version, and got worse each iteration
    - especially due to restructuring
- GUI not tested
- Launcher not tested
- Enumerations not tested
- Menus contains interfaces
- Exceptions not tested
- Group members were absent

Future Improvements

- Better test coverage
- State manager
    - We could get rid of the switch statement with enough time
- Read via XML instead of enumeration
- Implement would haves