Assignment 5

SEM Group Project, sprint 5, week 1.8.

Published on 23 October 2015.

TU Delft Software Engineering Methods 2015-2016 (TI2206)
Supervisor: Dr. A. Bacchelli
Teaching Assistant: A.W.Z. Ang

Group 12

David Alderliesten	4368703	J.W.D.Alderliesten@student.tudelft.nl
Jesse Tilro	4368142	J.Tilro@student.tudelft.nl
Jeroen Meijer	4382498	J.Meijer-5@student.tudelft.nl
Floris Doolaard	4362748	F.P.Doolaard@student.tudelft.nl
Niels Warnars	4372069	N.Warnars@student.tudelft.nl

Table of contents

1 Requirements for Sprint 5

1.1 Functional Requirements

1.1.1 Must haves

1. The player shall be able to pick up power-ups in the game, which will cast a temporal effect on the player meant to increase the ease with which the current level can be completed.
2. Three different types of power-ups shall be introduced, each with a different effect on the player: Blueberry Bubble, Pyro Pepper, Turtle Taco and Chili Chicken.
3. A power-up will be dropped with a characteristic probability distribution when an enemy is killed.
4. The chance of an enemy dropping a Blueberry Bubble power-up shall be 10%.
5. The chance of an enemy dropping an Chili Chicken, Turtle Taco or Pyro Pepper power-up shall be 5%.
6. The Blueberry Bubble power-up shall provide the player the ability to shoot enlarged bubbles for 10 seconds after pickup.
7. The Turtle Taco power-up shall provide the player with invulnerability for 5 seconds after pickup, meaning the player will not die upon collision with an enemy.
8. The Pyro Pepper power-up shall provide the player the ability to shoot fireball projectiles for 5 seconds after pickup.
9. The Chili Chicken power-up shall provide the player the ability to move with 150% of its normal movement speed for 5 seconds after pickup.
10. A Blueberry Bubble power-up shall be able to encapsulate up to three enemies at the same time.
11. A Blueberry Bubble power-up shall only be destroyed when it hits a wall or platform.
12. The physics of the Blueberry Bubble shall be identical to the normal bubble.
13. A Fireball projectile shall instantly kill an any enemy on collision.
14. A Fireball projectile shall move horizontally in a straight line.
15. A Fireball projectile will only be destroyed when hitting a wall or a platform.
16. The current power-up of the player shall be overridden by another power-up if it is being picked up or shall give bonus points to the score of the player(s).

1.1.2 Should haves

1. All levels should be modified to ensure that power-ups spawn within the level by default.
2. A player should not be able to climb the walls or move against the underside of a platform by taking advantage of a bug in the game.
3. The enemies should jump randomly to enable them to climb up stairs and move up to platforms.
4. A new type of enemy will be introduced next to the current type of enemy Zen-Chan, namely the Mighta enemy.
5. The Mighta enemy will not be affected by Gravity, and will therefore be floating through the level in a non-deterministic manner.
6. The Mighta enemy will only be introduced in the last 50% of the levels of the game.

1.1.3 Could haves

1. CSS for the keybinding options in the settings menu could be a adjusted in such a way that tabs are equally aligned.

1.2 Non-Functional Requirements

1. The total production codebase, excluding code for the Graphical User Interface implementation, shall have a line test coverage of at least 80%.
2. No fixable CheckStyle/FindBugs/PMD warnings shall be left unfixed at the moment of hand-in on October 23rd 2015.
3. All the issues assigned for Sprint 5 should be fixed before October 23rd 2015.

2 20-Time Revolutions: Power-Ups

Motivation

For the final official assignment, the decision was made to focus on implementing a feature that was intended to be in the game from the beginning, but had not been implemented so far due to timing issues and other priorities. This feature was the power-up system.

The power-up system exists in the native Bubble Bobble game, in which the player character has the ability to fire more powerful projectiles, become faster, or be temporarily invincible. We wanted to implement a simple yet extendable version of this system. Based on the original game, we decided to implement the following power-ups for the initial implementation:

- **Blueberry**
  Grants larger bubbles to the player that can encapsulate multiple enemies upon collision, approximately 225% original bubble size for 10 seconds.

- **Pyro Pepper**
  Grants the player the ability to shoot fireballs, if possible for 5 seconds.

- **Chilli Chicken**
  Grants the player increased movement speed for 5 seconds. Movement should be approximately 150% the base speed.

- **Turtle Taco**
  Grants the player temporary invincibility, lasting about 5 seconds.

At the time, these requirements seemed simple enough to implement. After the creation of the requirement (which can be found above), a CRC session was held to decide how the implementation of the underlying power-up system should exist. It was decided that power-ups should utilize a decorator design pattern, meaning that every power-up featured its own implementation and could be applied upon the player.

It was decided that the player should be under the effect of one power-up at a time, where others simply grants points instead of applying a double power-up bonus. It was also decided that Power Ups should be level elements that had to be picked-up, and not necessarily as drops from enemies or randomly generated occurrences.

The result of the above was synthesized in the CRC diagram displayed on the next page.

CRC Diagram

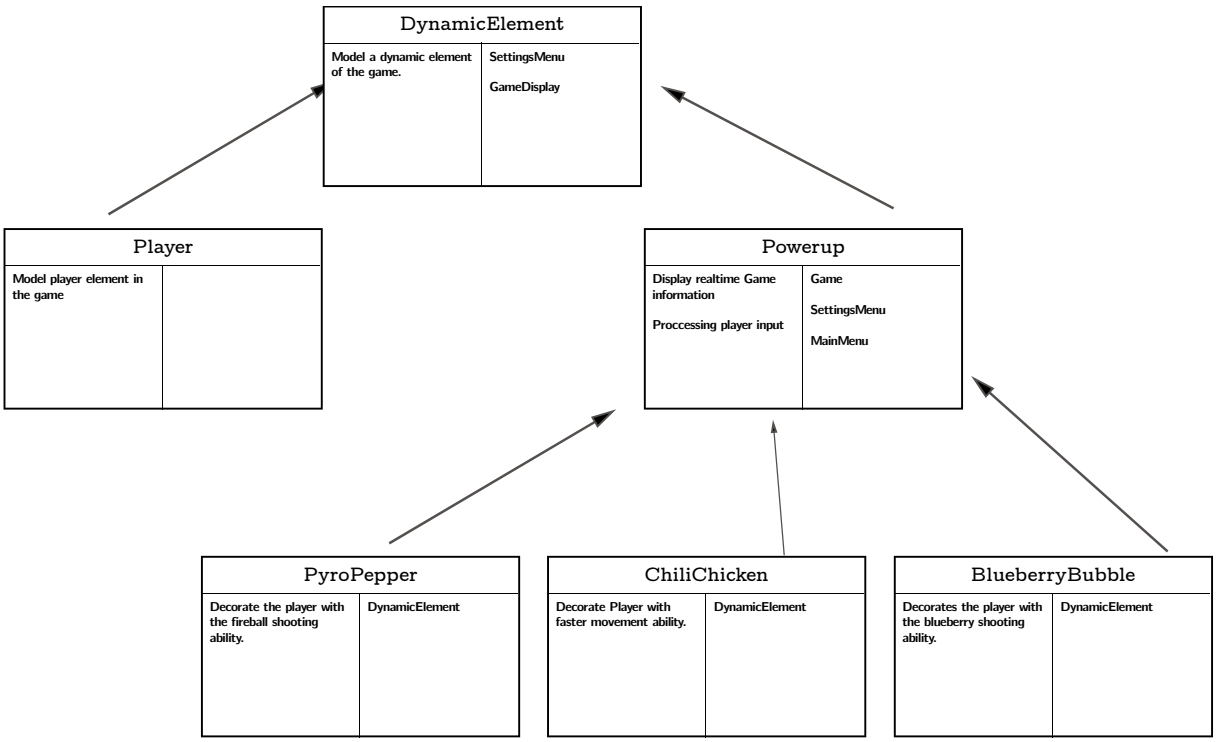

Figure 1: A diagram representing the intended Power-Up implementation from the CRC session. Please note that during the CRC session the Turtle Taco was not included yet.

This sprint featured a complete refactoring of the Level Element implementation, which meant that the player was no longer a special entity but a normal level element. As such, in order to place Power-Up items within the level, we decided to have the interface of the Power-Ups to be implemented as a level element.

To conform to the decorator pattern, it was decided that each power-up would exist as a unique extension of the Power Up interface. This not only applied the decorator pattern, but it also allowed modifications and possible future extensions to be made easily by simply extending the Power Up interface.

Earlier attempts at making a streamlined implementation from the CRC session resulted in Power Ups being special classes and objects, and being separate from all other segments of the application. With the level elements refactor in mind it was decided to not do this, as extension was relatively simple and allowed the Power Ups to exists in the location where they would be used, which was as level elements that modify the player.

The UML Class Diagram further visualizing the implementation of the Power-Up feature (figure 2) can be found in section 3.1.2.

3 Design Patterns

3.1 Decorator Design Pattern

The first design pattern that was implemented fort his exercise is the Abstract Factory Method design pattern. This section discusses why and how this design pattern was implemented and includes the UML Class and Sequence diagrams visualizing this implementation.

3.1.1 Why Decorator DP was applied on Power-Ups

When brainstorming on how to spend the 20-Time for this iteration, it was concluded that the time allotted were to be spent on implementing a Power-Up feature. This was documented as the following requirement during the requirements engineering process: “The player shall be able to pick up power-ups in the game, which will cast a temporal effect on the player meant to increase the ease with which the current level can be completed.” (section 1.1, requirement 1).

Triggered especially by the words temporal effect, implementing this functionality was recognized as a good opportunity to apply the Decorator design pattern. Therefore the choice was made to combine the 20-Time and Design Pattern 1 exercises of this week’s Assignment, as the solutions to both exercises concerned the implementation of this Decorator design pattern.

The Player LevelElement behaviour and the way the player interacts with other components was to be interchangeable at runtime, though still uniformly treatable by clients. This was realized by wrapping the player in a powerup element, which implements the same interface as the one clients expect when interacting with the player; hence the Decorator design pattern.

This solution could not be implemented right away, however. The hierarchical structure of the LevelElement components needed a major refactor in order to allow the new Powerup interface to have a place in the system correctly implementing the desired functionality. The complete refactored hierarchical structure of the LevelElement components as well as the new components implementing the Power-Up functionality have been visualized in the UML Class Diagram (2.1.2).

3.1.2 UML Class Diagram of Decorator DP

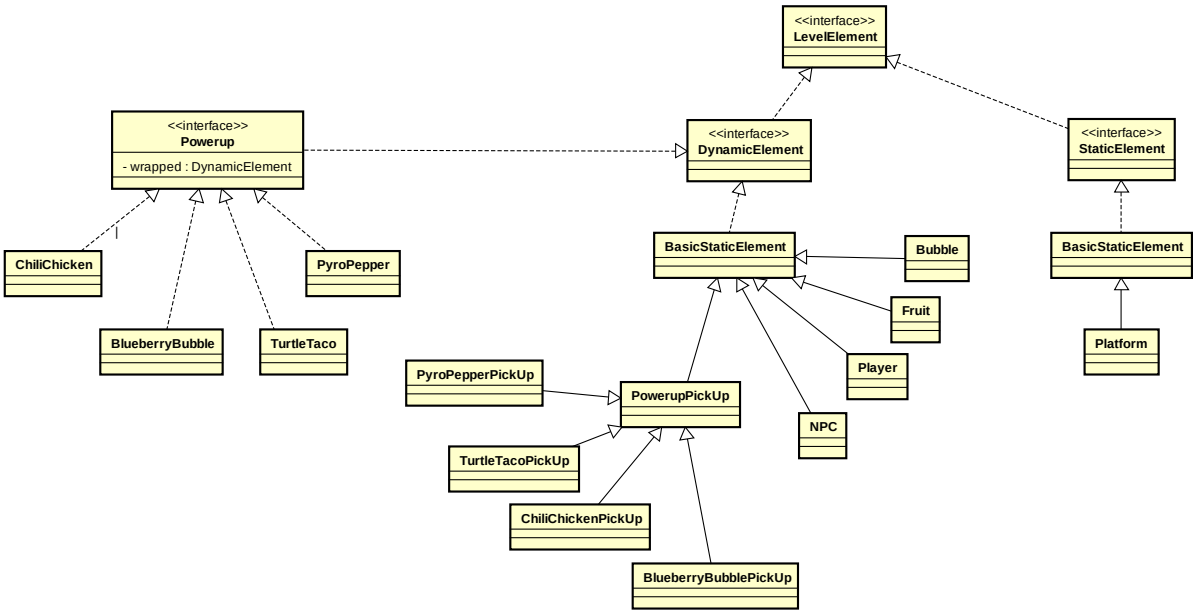

Figure 2: UML Class Diagram visualizing the implementation of the Decorator design pattern, applied on the Powerup classes.

3.1.3 UML Sequence Diagram of Decorator DP

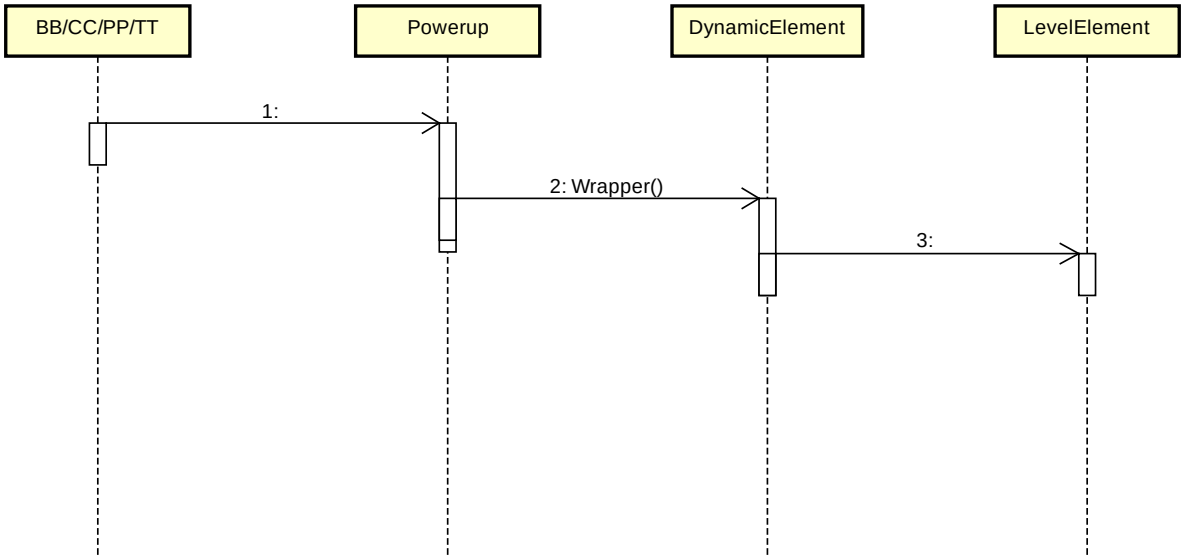

Figure 3: UML Sequence Diagram visualizing the interactions between objects that are part of the Decorator design pattern implementation.

3.2 Abstract Factory Design Pattern

The second design pattern that was implemented fort his exercise is the Abstract Factory Method design pattern. This section discusses why and how this design pattern was implemented and includes the UML Class and Sequence diagrams visualizing this implementation.

3.2.1 Why Abstract Factory DP was applied on Games

When the multi player functionality had been introduced half way during it was limited to two player functionality only with no possibility to play a single player game. This implementation was incomplete and an if-else structure for selecting a single player or multiplayer game would have resulted in a dirty implementation.

At this point of the project - with the separation of the Single Player Game and Multi Player Game in mind - it was decided to implement an Abstract Factory Method design pattern.

When multi player functionality was implemented for the first time it had the following limitations:

- Single player and Multi player selection would be hard to implement
- All Level Modifiers were still added in a special construct() method in the Game class.

The solution for these problems was to implement the Abtract Factory Method Design Pattern.

With the Factory Method implemented it is now up to the MainMenu to create a correct Game Factory when a player clicks on a button to start a single player game or a multiplayer game. In this case a new instance of a Single Player or a Multi Player Game Factory is created. This Game Factory is passed on to the GameDisplay.createGame() method which is unaware of the type of Factory it is dealing with and consequently calls the GameFactory.makeGame() method which initializes the instantiation of a new Single Player or Multi Player Game.

3.2.2 UML Class Diagram of Abstract Factory DP

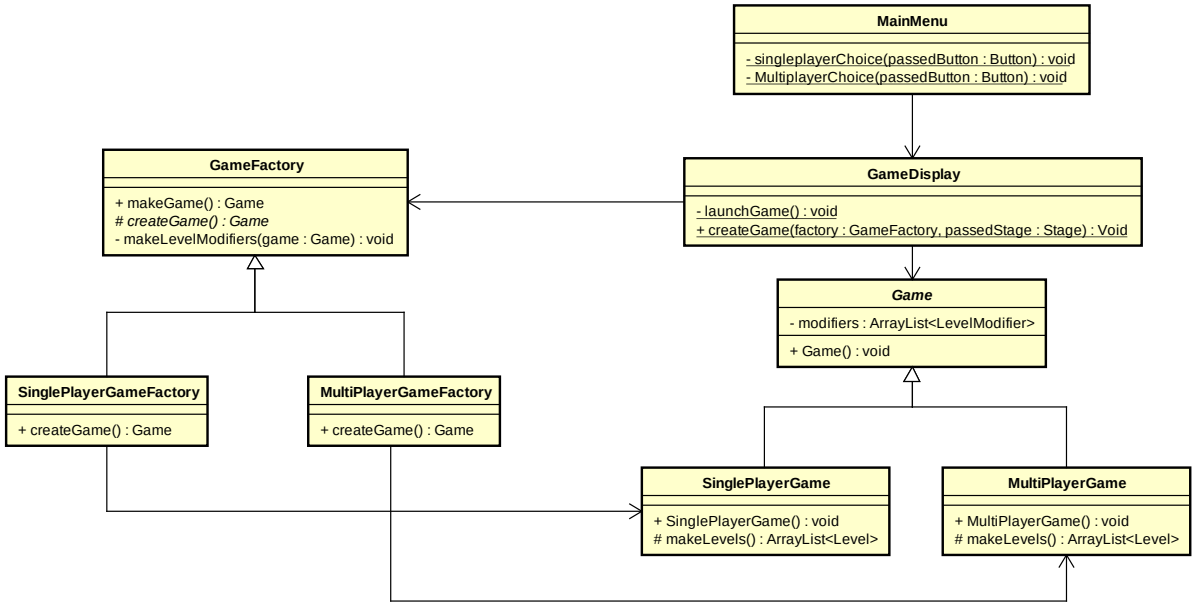

Figure 4: UML Class Diagram visualizing the implementation of the Abstract Factory design pattern, applied on the GameFactory as the abstract factory and the Game as the abstract product.

3.2.3 UML Sequence Diagram of Abstract Factory DP

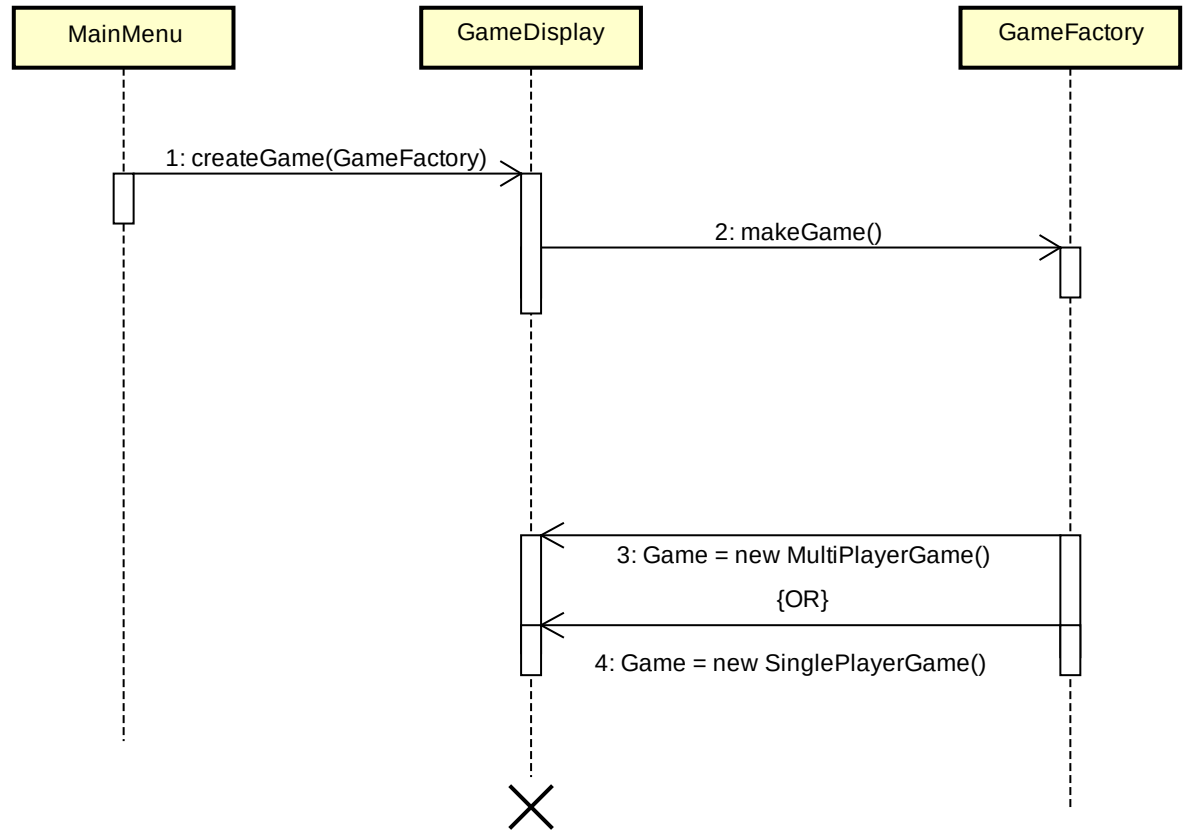

Figure 5: UML Sequence Diagram visualizing the interactions between objects that are part of the Abstract Factory design pattern implementation.

4 Reflection on the group project

In the past few weeks our source code has evolved from a simple, barely playable version of Bubble Bobble to a fully fledged version of the game. Using a well designed and properly thought out engine under the hood we were able to recreate the game and enhance on some of its features.

A lot of effort was put into the design of the project to ensure easy maintainability and extendability for both features and engine changes. This helped a tremendous amount in the later stages when the development of the project focussed more on additional features and when time was limited, as the design choices that were made early on allowed, for the most part, simple extensions.

4.1 The Design Process, lessons learned and discoveries

One of the key things learned in this period is that it is always better and way easier to go into something with some sort of plan or design at hand. If done right, this will make it possible for all the pieces of code to fall in their places like a jigsaw puzzle. We noticed this when we began to couple the graphical user interface to the actual game. Due to our design, we had enabled hooks from the user interface to almost every major component of the game (player, level element, and when coupling this in the GUI we could access almost every graphical element and desired modifier (such as movement of the player with key input), it was a very easy task.

A second thing learned is that with a well designed and flexible application it is very easy to add additional features without much of a hassle. This application was therefore designed to be highly extensible and modular. An example of this would be the multiplayer feature. Our player class was implemented by the game, but the game class and level parser classes could support an additional player with the addition of a single method. This would not have been possible if the player class had not been designed with flexibility and extensions in mind.

In the course we learned that a good way to keep the project clean and extendible is by using design patterns. This project therefore incorporated various design patterns in the application. Some examples from our project are:

- *Singleton design pattern*
  The singleton design pattern required the implementation of feature(s) that should only exist once within a program. We were able to implement this with our logger/logging class. The logger implementation utilizes the singleton design pattern to ensure that only one copy of the logger is active at a time. This is required in order to prevent deadlocks or multiple logging files being created per session. It was also implemented to prevent overriding, as multiple loggers could cause the existing logging file to be overridden by a new one.

- *Observer design pattern*
  The observer design pattern was implemented within our level modifier classes. The game notifies it's observant level modifiers when to update the game's current level. Since the Game is propagated by the StepTimer to update each cycle, all modifications made to the Game's current Level are synchronized with the StepTimer.

It is almost certain that all of us will need to use design patterns such as the above (and the ones that were implemented in the project) again at some point in our lives. This course and project has taught us to use them and to apply and design with them, allowing us to gain an introductory knowledge of them.

4.2 Development, issues and organization

The creating of the project proved to be more challenging and complex than we expected at the start. Even after the first weeks there were still some configuration mistakes found, such as checkstyle normalization issues and lackluster documentation. By using the knowledge we learned from this project we can ensure in future projects that these issue are solved immediately at the kick-off, and not halfway through development. The identification of these issues in a team requires time and experience, identifying them without those two factors would be almost impossible for a beginning developer.

A lesson that was learnt in this project is that a project deadline should feature content completion well in advance of the deadline. Both the first and last deadlines had timing issues (the first due to mismanagement, the last due to serious bugs late in development caused by the extension of the project), and the deadlines in between we far more relaxed and allowed for additional time to review code and do manual testing, which was far less stressful.

4.3 Meetings, Organization, and Reflection

At the start of the project the team was rather inexperienced in the ways of SCRUM, having used it only once or twice before in guided projects at the university. This naturally provided a great learning opportunity for the team to get some much needed experience with this industry accepted and widely used method.

Throughout the project there was a great amount of improvement to the weekly SCRUM meetings. When the first few meetings were held every member of the team had their laptop open. Naturally this led to the team being distracted, which should generally be avoided during meetings and planning. This issue surfaced during one of the meetings and since then the rule was established to keep all laptops shut, with the exception of the leader/secretary. During the same meeting a more standardized format for the weekly meetings was adopted, which resulted in efficient meetings in which a lot was done in a very little amount of time. As the first few meetings were rather disorganised the team devised a document which had the structure for the meetings. An important part was the discussing of- and improving the way the previous sprint was handled, addressing issues such as team communication and review standards. The result of these changes was a far greater level of efficiency within the team.

For the first few assignments there was a lack of organization with regards to the tasks and the responsibility that came with each task. For example, in the first assignment a team member had designed a major feature to enhance the overall game, and when this was implemented and we were at the deadline it turned out that responsibility driven design and planning such as UML diagrams had not been created, let alone made. After the second assignment the team decided to make a clearer distribution, aided in part by task tracking and distribution over the Github “issues” system.

4.4 Lessons for the future

The lessons learned and issues that were handled and dealt with within this project will be of great relevance once we enter the professional industry, an industry that is continually interested in improving its methods and designs in order to prevent costly and/or damaging delays. Thanks to both the course and the project we now understand how the processes behind software engineering influence the eventual products, and how these products can be enhanced upon and troubleshooted in a superior fashion thanks to good documentation and clear designs.