TI2206 Software Engineering
Bubble Shooter report: Assignment 5
EEMCS/EWI

Gerlof Fokkema 4257286
Owen Huang 4317459
Adam Iqbal 4293568
Nando Kartoredjo 4271378
Skip Lentz 4334051

October 26, 2014

Exercise 1: 20-Time, Revolutions

In this section improvements made to the Bubble Shooter of Doom during the '20-Time-Revolutions' will be elaborated. Additionally an in depth description of what has been implemented and the resulting class diagrams can be found in this section.
One of the major changes that were planned for this sprint was the introduction of an animation system. This task turned out to be more daunting than expected. Therefore we decided to focus on improving the existing game features, which allowed us to deliver a good working and polished final product.

Polishing of all game features
During this sprint lots and lots of bugs have been fixed. Apart from that, all kinds of tiny improvements have been added to the game as well. Most of these bugs and/or improvements were not really critical problems, but they do influence the gameplay experience of the user.
A few things that have been fixed are for example:

- Synchronization of EndingConditions in Multiplayer-mode.
- Displaying opponent score in Multiplayer-mode.
- Properly closing Sockets when a client disconnects.
- Adding textures to textfields.
- Addition of extra levels for Arcade-mode.
- Difficulty balancing of level progression in Arcade-mode.
- The *Zen Mode* is playable again.

As all these changes are minor changes, the corresponding design of the classes involved did not really change. Therefore the structure of these classes did not change, thus no new class diagrams were required. After all these fixes, the game now gives the user a pretty smooth game play experience.

Responsibility Driven Design applied to GameFactory
Before, we instantiated the *GameMode* and *EndingCondition*s of the game in the *Listener*s of the buttons in the *MainMenuScreen*. Since this wasn't really a responsibility of the *MainMenuScreen*, we decided to relieve the *MainMenuScreen* of this responsibility, and create a *GameFactory* interface. This setup also enabled us to add an additional feature to the game, which is discussed in the next section.
A UML class diagram can be seen in **Fig. 1**.

Abstract Factory pattern applied to GameFactory, to facilitate viewing the opponent's score
With the 'old' design, it was not possible to view the opponent's score: the process of building the *GameUI* and instantiating the *MultiPlayerMode* happened independently. This caused problems when trying to access the *Score* object of the opponent, as there was no way of doing this.
It did not take long to realize that a *GameFactory* was needed, which can create *GameUIs* and *GameModes*. By introducing these new factory classes, access to the *Score* object of the opponent was gained. This meant that *StatsObservers* could be added to display the opponent's score on the screen. By applying the *Abstract Factory* pattern to the *GameFactory* class, the *BubbleShooterScreen* can create the *GameUI* and *GameMode*, without knowing anything about what kind of game it is (arcade or multiplayer, for example).

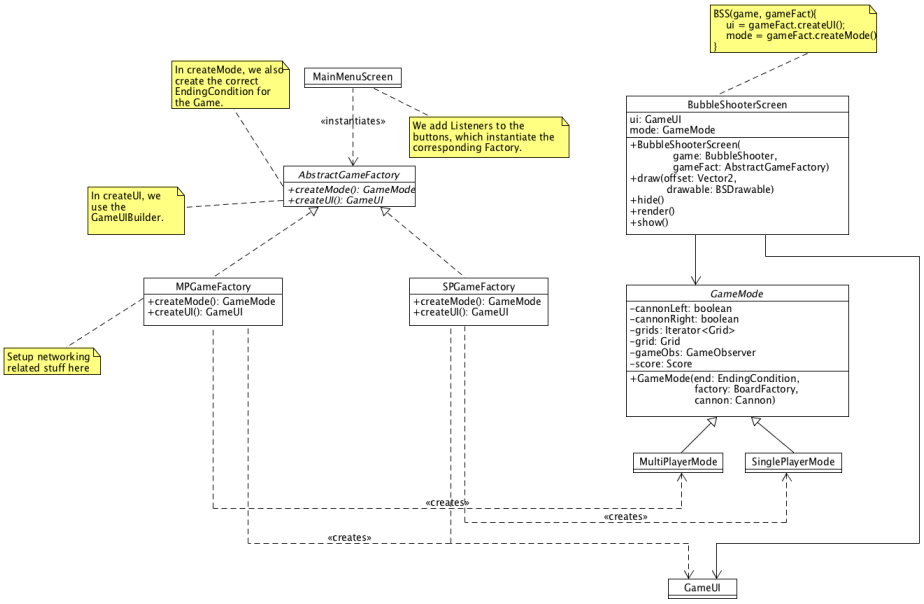

Fig. 1: Class diagram of *GameFactory* and most of its decendants.

Responsibility Driven Design applied to Settings
In order to introduce the ability to switch between themes, there was a sudden need to store *settings*. It would be nice if after changing the theme, when the user closes the application this *setting* would be remembered. This way the user would not keep having to change the theme after starting the application. So to make this possible, a new class called *Settings* had to be introduced. Thinking back on what newly introduced responsibilities came with this feature two key responsibilities can be noted:

- The first being: able to store information about the selected theme.
- The second being: able to actually switch between themes.

The new *Settings* class would bear these two responsibilities. Additionally since the *Settings* should be something which is only instantiated once, because there is only one *'settings'* the use of the *Singleton* pattern was also applied here. How this is done can be read in the next paragraph.

Singleton pattern applied to Settings
To ensure that there is only one unique instance of *Settings*, a singleton pattern can be used. As previously mentioned, to make theme switching possible the possibility to store information about the selected theme is necessary. So this means that it is also necessary to write to a file and of course reading from it. Additionally when loading in the new theme (which are essentially all textures), this means that the old used textures should be unloaded. This required that *Assets* would be able to unload all the textures in order to load in the new ones. This is essentially what switching themes accomplishes. In **Fig 2.** the class diagram of the implementation of *Settings* is depicted.

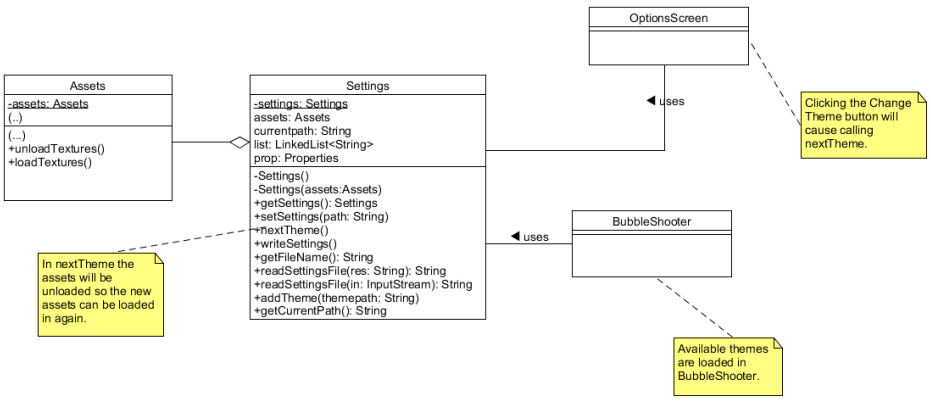

Fig. 2: Class diagram of *Settings*.

Exercise 2: Wrap up - Reflection

In this section a fairly in depth reflection on the entire project is done. This includes looking back at the development cycle, reflecting on the things that have been learnt and how this may improve future projects.

This semester, we took part in a lab assignment for the course Software Engineering Methods with a project kind of setting. We worked in a team of five people, designing a Java based game of *Bubble Shooter*. A great difference between this project and previous ones was that we had to deliver a complete/final product. One of the first things learnt in the course was using *Responsibility Driven Design*. This we also had to immediately apply to our project. During the development a requirements document should be constructed. The final product should meet the requirements listed in the document, and furthermore the product should be thoroughly tested and finished in the available time.

Before this project we had often done only small assignments (in other courses), where the task given was not a very broad one, meaning there was little freedom in choosing what we would want to do. All these past assignments often made you use only a small part of the topics discussed during the course. This project required all of the previously practiced skills acquired from the other courses as well.

Reflecting on experience is an important capability and even a necessity for IT professionals, as this can prevent you from making mistakes which will cost a lot of time and money. Ignoring reflection could even lead to disastrous or even rejected final products, if the errors made are so grave and impact the final product immensely.

Having experience on how to tackle, and even more important, how not to tackle the 'problem' from a given assignment up to a final product is essential, because this might save you a lot of unhappy customers in the end. A great deal of our work in the IT section is delivering good products, these products are almost always made through project based teams.

So for an IT professional an essential skill is to be able to effectively work with others in a team. While we practice this in the lab sessions on a relatively small scale, in the industry projects can get quite large. This usually means the projects get complex, because more people are involved: more stakeholders and more developers.

The second issue is with the *final products*, these tend to be 'not as final' as they perhaps should be. The IT world is very dynamic, and changes occur everyday, so our products should be sustainable enough to withstand these changes. We often do not know what the future brings us, so our code needs to be maintainable and extendible too by third parties after delivering the final product.

Before this project, code that we wrote was often not very maintainable and/or sustainable. We did not test our code thoroughly enough and we surely did not think about using design patterns. We did have a little bit of experience with working in groups, but not with the a SCRUM based setting and the workload combined. This lab was a great gain for us to grow into great IT professionals.
During this project we have also seen what effects unmaintainable code can have. The *BSMode* class is an example for this. While it was designed to be reusable and maintainable, it turned out that this class was still very rigid and could not easily be removed from the design again. This has taught us to first rethink our design at least twice before implementing it.

The required use of sprint plans and reflections on those, made us think really hard about what was achievable and what should be achieved. Prioritizing tasks was something we did not really think that much about previously. Knowing when to drop a task was also a valuable experience for all of us. Although we did not have any prior experience working with sprint plans, they were utilised rather quickly. We did not have a lot of issues with it, so we were able to use them properly without too much trouble. We felt a bit strangled in the begin by the tasks set in the sprints. During the sprints we often tried to ensure that only the sprint tasks we set in the sprint plan are done. However the need to change something in order to be able to implement something new, for instance a feature occurred more often than predicted. Such changes/tasks are then difficult to document, since it was not part of the initial sprint plan.

As we grew into the project, we started to appreciate the secureness the sprint plan brought us. We knew exactly what we wanted to get done within a week after we added priorities and estimated time to the sprint plan. Using these estimates we then allocated the available time to the different tasks. This made it very easy to determine whether to continue or switch to another task with higher priority when some task caused more difficulties. By reflecting every week on the sprint, and estimated effort, we became much better at estimating what we could do within the given time frame. This made our sprint plan much more efficient as the sprint-weeks passed us by. As we started to have rather efficient sprints, our work became of higher quality. We noticed this by how and what we delivered, and last but not least through our feedback by the TA.

At some point we were taught about how to apply design patterns and how to use UML to ducument your design. Every one of us had no prior experience using design patterns. UML on the other hand was not something completely new, but prior to this very little time was spent on carefully explaining the right way to use it and the reasoning behind it. Surprisingly

some of the patterns taught felt as we had known them before as the solution with the least amount of possible errors just because our experience told us so. We had to adjust our experience based guidelines sometimes a little bit, but the design patterns felt rather nice and welcoming to use. The patterns also helped us think more about ease of extendability of our work in the future. Additionally it made us think about what kind of responsibilities classes should bear. This resulted in having less cluttered classes.

Design patterns are claimed to be one of the most powerful methods for building large software systems, as patterns provide tested/widely used solutions to re-occurring problems that we, code developers, often may face while developing new software. However, design patterns should not be over-used and should only be applied when they solve a problem in your design. Furthermore, they should be properly implemented.

In order to (partially) analyze the design of a program we were also introduced to the concept of software metrics. Since the start of the project a lot of design patterns had been introduced into our design. Using inCode, we analyzed our design and the use of these design patterns. The use of *inCode* to analyze our project was quite interesting and made us realize some of the design problems we created.

As *inCode* also showed us, applying various design patterns was sometimes not that easy, the question on how to exactly use them in the current code structure often arised. This resulted in often major changes in code structure. However in the end we noticed that introducing the new patterns improved our code quality greatly, as they gave us a structure which made our code far better sustainable and re-usable.

To conclude our experience of this project, in the beginning of the project at the end of our first few sprints we were often rather dissatisfied, but as our experience grew in planning, our sprints became better and much more efficient. We learnt how to work with design patterns, we learnt how to progress from assignment to a fully tested final product using the SCRUM methodology. We learnt how to use UML to help documenting changes in the implementation. We learnt how to effectively work in groups and we learnt how to produce re-usable code. Re-usable as in that our code/final product could be extended without edits in the source code. All these experiences together gave us the feeling that we learnt a lot from this project, we feel very satisfied about our final product. These experiences also gave us great confidence on how to tackle future group projects and create a final product with some (high) IT standards.