Deliverable 6

TI2206 Software Engineering Methods
Delft University of Technology
The Netherlands

Piet van Agtmaal 4321278
Paul Bakker 4326091
Jochem Heijltjes 1534041
Jente Hidskes 4335732
Arthur Hovanesyan 4322711

25 oktober 2014

1 Requirements specification sprint 7

The goal is to develop different difficulty levels by changing the spawning of new tiles.

1.1 Functional Requirements

F1: Player should be able to choose between three difficulties (Random, Medium and Hard) of the singleplayer game.
F2: Player should beuable to choose between the two different solvers in the settings screen.
F3: The player's chosen solver should be saved.

1.2 Non-Functional Requirements

NF1: Man in the middle attacks should be circumvented.
NF2: Fonts shouldn't be aliased.

2 Sprint plan 7

Game: 2048
Group: 21

User Story	Task	Assigned to	Estimated Effort
	Exercise 1: True type font	Piet	10 hours (hard)
Story 1	Exercise 1: Select between solvers	Piet	1 hour (easy)
Story 2	Exercise 1: Choose difficulty	Jente	20 hours (hard)
	Exercise 1: Improved communications security	Paul	10 hours (medium)
	Exercise 2: Essay	Everyone	15 hours (medium)

2.1 User Stories

2.1.1 Story 1

As a user, I want to be able to choose which solver I want to use. I want the chosen solver to be saved.

2.1.2 Story 2

As a user, I want to be able to set a difficulty in the singleplayer game. This means that the spawn of a new tile isn't random anymore. It spawns the new tile on difficult places to increase the difficulty of the game.

3 Exercise 1 - 20-Time, revolutions

In this section we will describe the extra features we have implemented.

We implemented the following extra features:

1. The use of TrueType fonts;
2. An option to switch between solver types;
3. An option to set the delay between solver moves;
4. An option to set the difficulty of the game;
5. Secure communication over the network.

3.1 TrueType Fonts

The use of TrueType fonts was an improvement over the BitmapFont we used to use. The latter turned out to be quite blurry on HDPI screens. Therefore, we wanted to improve the sharpness. Actually, this was on the bucket list for quite some time, but we never managed to get it working properly. Some more research eventually led to a solution, which is creating BitmapFonts on the fly from a .ttf file. We then add the generated fonts to the Skin object, so that labels and textbuttons are able to use them. Since no major changes to the code were necessary to accomplish the goal, no UML diagrams have been created.

3.2 Additional Settings

Since we created two different AI classes for solving the grid, it is obvious that an option has to be added to use the desired one. Furthermore, to achieve even more flexibility, we wanted to add an option to set the delay between moves. Doing this enables the user to either carefully see how it is done or faint from pleasure when he lets the solver show off.

One major change we made to the game is the option to set the difficulty. We have accomplished this by modifying the manner in which tiles are spawned after a move. We created an `AnnoyingSpawner` class, that calculates the location at which tile placement of a certain value (a 2 or a 4) would harm the player the most. Depending on the difficulty level set, the `AnnoyingSpawner` is picked instead of the `RandomSpawner` to spawn a tile. These probabilities are as follows:

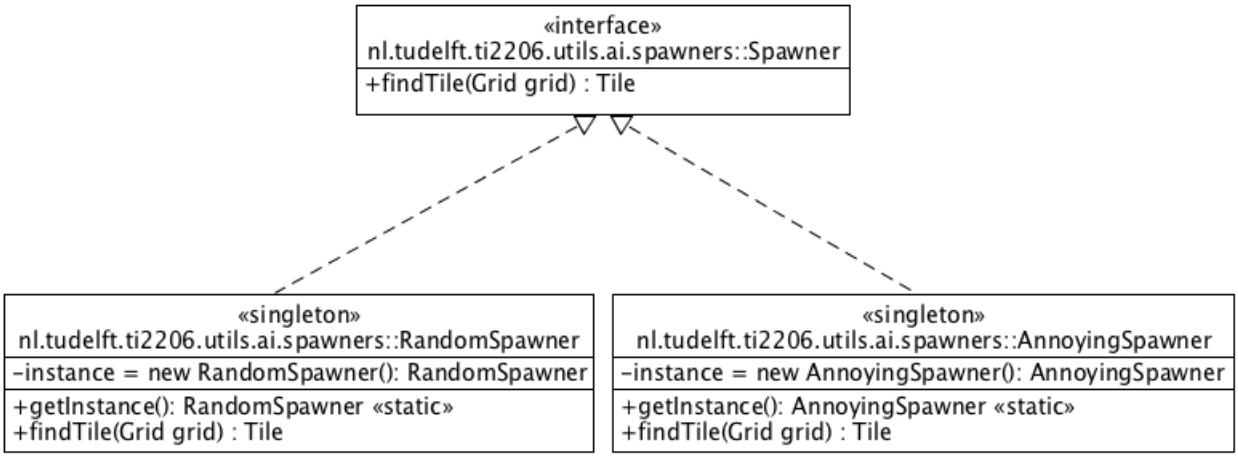
Random 100% random, 0% annoying;

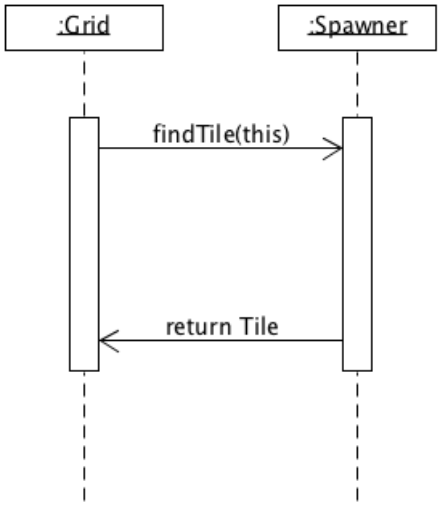
Easy 67% random, 33% annoying;

Medium 33% random, 67% annyoing;

Hard 0% random, 100% annoying.

Even our best player (obviously the Expectimax solver and not Piet!) did not reach a higher tile than 512 at hard difficulty, so we feel like we have reached our goal pretty well.

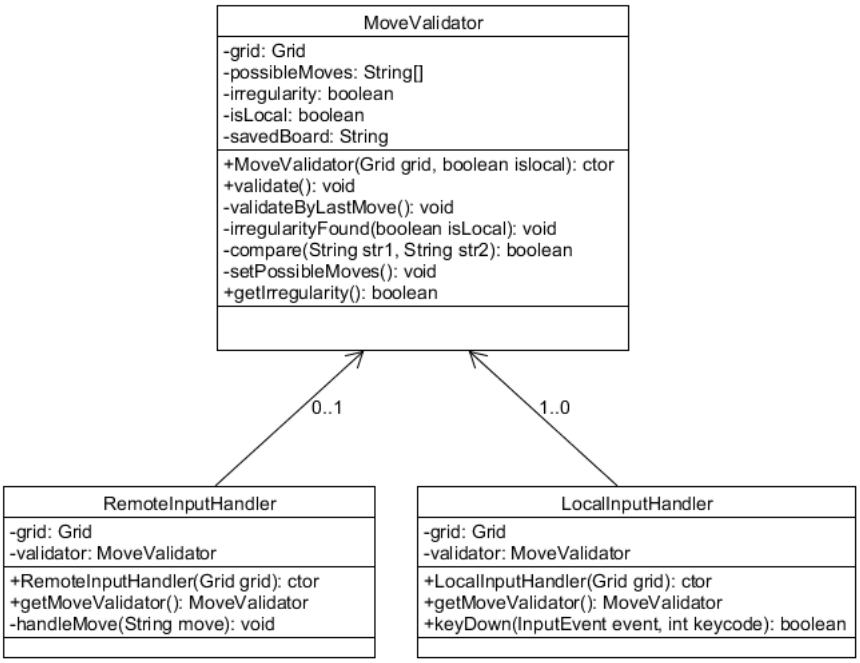


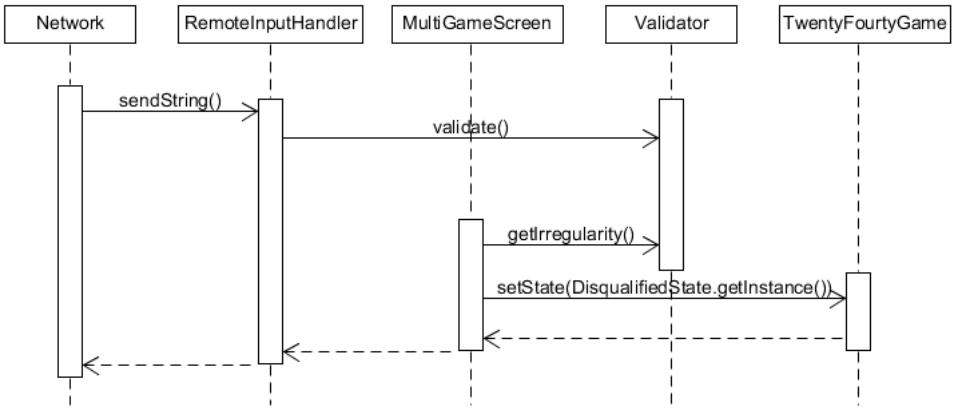


3.3 Secure Networking

Multiplayer was one of the first extra features we brought to the game. Ever since we added the feature into the game we had problems with players tampering with the messages sent to their opponent. Doing this would give the player an unfair advantage. This might be all fun the first couple of times but after a while this becomes very dull. To keep the game fun for everyone we have implemented a move validator that checks whether the received move was actually plausible from the position the tiles were in previously.

A problem here was that making the same move as the player would usually not give us a grid that is exactly the same as the player's grid. This is because of the tiles that are randomly spawned after a move is made. Since we do not know the actual position or the value that the random tile will take, we can only determine if a move was actually possible from the previous move. During the comparison we count the amount of differences that exist between the tiles. These differences can only be random tiles in wrong positions or values. If the two tile sets have more than two differences between them, they are considered not similar. When the new grid is not similar to the previously set possible moves, we consider the grid tampered with.




Network
RemoteInputHandler
MultiGameScreen
Validator
TwentyFourtyGame
sendString()
validate()
getIrregularity()
setState(DisqualifiedState.getInstance())

4 Exercise 2 - Wrap up - Reflection

As a team, we have spent many hours working on this project. Many (manly) tears were shed and lots of laughter was brought to us. In this chapter we will reflect on what we have learned throughout this course about ourselves as a team and as developers.

To start off, we have learned how to adequately define requirement specifications for a project and how to plan the iterations that go with this. We had to do this for every single sprint, so that each sprint it was clear to us what our tasks were and how the features had to be implemented. At first we considered this superfluous; for such a small project as this it sure was nonsense. However, as the weeks progressed and as our game became more feature-rich, we started to see the value of defining requirements and creating a proper sprint plan.

We have also learned how to properly structure a software product. Amongst the things we have learned to do this are responsibility-driven design (RDD) and design patterns. RDD aims to describe what the responsibility of an object is and which collaborators it has. This improves the design of the code, as for each class the purpose it serves in the system is clear. Another important skill to support this and that we have developed is implementing design patterns in a software project. A design pattern is an abstract solution to a reoccurring problem within a specific context in object-oriented programming. When these design patterns are implemented correctly, maintaining a system and controlling software changes is much easier. We have now experienced first hand that this is indeed true.

Furthermore, to clarify the structure of our code, we have increased our knowledge on UML-diagrams so we could use these for explaining our project graphically and to visualize its structure. People not familiar with our project, like teaching assistants, were able to use these diagrams to quickly get an idea of how our project is constructed.

Using these skills does indeed help in constructing a solid software package, that is (relatively) easy to extend. Throughout the lab, we have spent a lot of focus on keeping our structure solid and clear. Responsibility-driven design and design patterns have definitely helped us to accomplish this. Last academic year we had a similar project that required us to develop a spreadsheet application. We were not equipped with the same skills and knowledge that we have today. Whenever we wanted to add a feature or change existing code, it would result in making changes to almost every class. Not having a clear structure also led to a bad understanding of the code by several group members and a poor quality of the code in general. If we compare the process of development back then to the one we have just finished, the improvements are huge!

As said, during this project we insisted on producing high quality and clean code with a proper and solid structure. Many unit tests were used to assure the quality of our code, together with the continuous integration server provided by DevHub. Besides this, many visual testing has been conducted. So much so, that we will never play 2048 again.

Our game's first release did not include any design patterns and the internal structure was not as solid as we would have liked it to be. This made it hard to maintain and extend our project and moved us to completely rewrite our game during the second sprint, as implementing multiplayer functionality in that codebase would have been very difficult. As time passed we learned how to implement multiple design patterns and how to apply them. Our latest versions include several design patterns, making our code easier to maintain and extend. The fact that we have made the right decision to rewrite our game is reflected in InCode's analysis of our project. As a course requirement, we had to let the program InCode analyze our project. InCode is a code analysis tool for discovering design flaws. After our first release (v1.0.0), InCode did not detect any design flaws, most likely due to the extensive refactoring we have done after this release. Since we learned how to keep the design flaws out of our project, it also didn't detect any design flaws in later iterations of our project.

In summation, we have learned a lot in this course and much of what we have learned improves how we manage a software project. We can (and will) use this in the future to create projects that adhere to a higher quality standard and that are more likely to succeed. Over time, our product and our code both have changed numerous times. Many features have been added and the code quality has improved greatly by implementing everything we have learned during this course. We have also improved our workflow regarding Git.

Our communication went mostly by Telegram Messenger, but also via email and Skype. Our teamwork has always been solid. Delegating tasks and helping eachother out always went without issues. Of course we've all been tired or grumpy but no major problems ever occurred within our team. The atmosphere within our team has been a pleasant one overall.

We are very satisfied with our final product. We have enjoyed working together to create a game that is fun to play and we enjoyed improving our understanding of software development while doing so.

5 Test report

In this section we will explain how we tested our game. We will start by explaining how often we tested our game. Afterwards, we will explain what kinds of testing we have done. Lastly, we will present the results of the testing procedure.

5.1 Test frequency

Due to the fact that only few new features have been implemented, not much testing had to be carried out. When someone was done refactoring, most of the times some tests failed and of course they had to be fixed before committing to prevent a build failure. When new features had been implemented, testing followed soon after to make sure the extension had been implemented correctly.

5.2 Testing methods

Visual tests involved actually playing the game and analyzing logging output manually. Unit tests simply check object properties with certain input. Visual testing was used a lot again this sprint when testing the new fonts and verifying the layout behaved correctly. This is because visual testing is the only way to make sure the layout looks like it should and that the new fonts really make a difference. During the cleanup and refactoring we made extensive use of our unit tests with the purpose of regression testing.

5.3 Test results

EclEmma is the tool we used for analyzing and measuring our test coverage. As before, we analyzed our entire project using three different metrics: line, branch and instruction coverage.

The results are as follows:

Line coverage: 80.0%

Branch coverage: 75.9%

Instruction coverage: 77.2%

As always, we are confident that our game has been sufficiently tested.