Deliverable 5

TI2206 Software Engineering Methods
Delft University of Technology
The Netherlands

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

19 oktober 2014

1 Requirements specification sprint 6

The goal is to develop a solver algorithm for the 2048 game and to clean up our code, including updating our javadoc.

All of the requirements have each been given a unique identifier followed by a number:

1. *AI* for the solver requirements;
2. *NF* for non-functional requirements.

1.1 Functional Requirements

Solver requirements are a leftover from last sprint, where we implemented another solver. This solver differs internally from the other in the sense that it uses a common algorithm, the expectimax algorithm, to solve the game.

1.1.1 Solver

AI1: The AI should be able to win at least 50% of the singleplayer-games it plays.
AI2: The AI should be able to look at the player's grid and add a tile to it that would hinder the player the most from winning.

1.2 Non-Functional Requirements

NF1: Javadoc should be updated to match with the functionality of the methods and parameters.

2 Sprint plan 6

Game: 2048
Group: 21

User Story	Task	Assigned to	Estimated Effort
Story 1	Exercise 1: Solver fix	Jente & Piet	25 hours (very hard)
	Exercise 1: code cleanup	Everyone	5 hours each (medium)
	Exercise 2: Flaw 1	Jochem	10 hours (medium)
	Exercise 2: Flaw 2	Arthur	10 hours (medium)
	Exercise 2: Flaw 3	Paul	10 hours (medium)

2.1 User Stories

2.1.1 Story 1

As a user, I want to be able to let the game be solved for me when I am stuck or when I want to see how it's done.

3 Exercise 1 - 20-Time

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

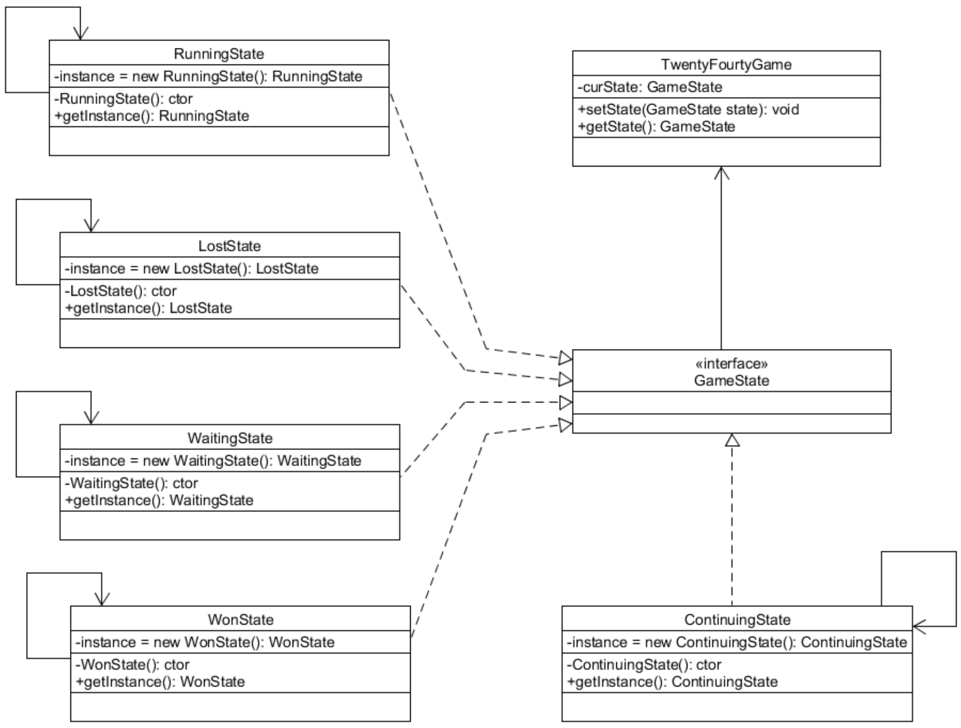
We implemented the following extra features:

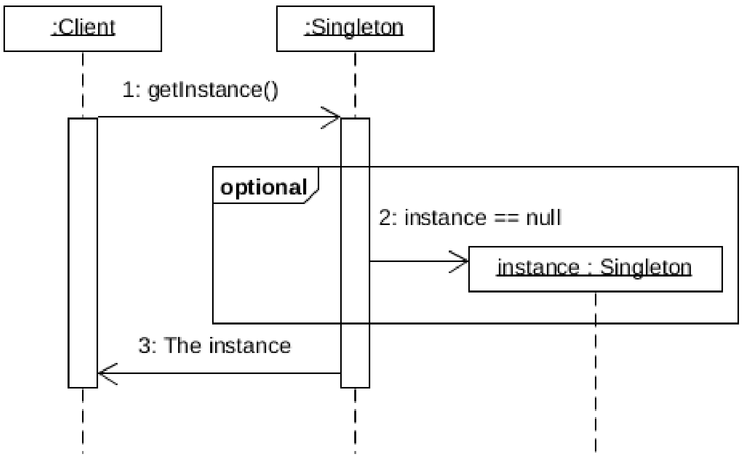
An AI that automatically solves the grid;

A settings menu, in which you can (obviously) change some settings.

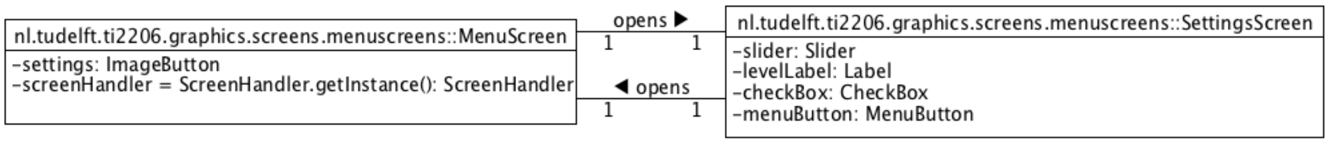
During this sprint we wanted to clean up our code ==, because we felt like it had gotten somewhat messy.

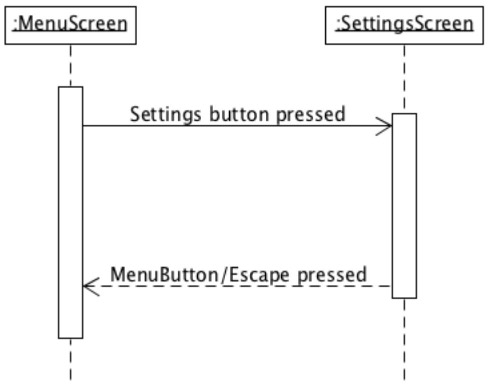
One of the changes we made to the code this sprint is that the states are now singletons. We made this change because semantically, it makes a lot of sense: there should only be one instance of every state. Besides this, there was the problem of maintainability: for every different state we have, we had to initialize it in the TwentyFourtyGame class and we had to make a getter method for it. This was making the class difficult to read and hard to maintain in the long run. With states now being singletons, we don't have to make changes to the TwentyFourtyGame class and we have improved its readability. To add a new state to the game, we can now just add a new class instead of being forced to make changes to the existing classes.


RunningState
-instance = new RunningState(): RunningState
-RunningState(): ctor
+getInstance(): RunningState
LostState
-instance = new LostState(): LostState
-LostState(): ctor
+getInstance(): LostState
WaitingState
-instance = new WaitingState(): WaitingState
-WaitingState(): ctor
+getInstance(): WaitingState
WonState
-instance = new WonState(): WonState
-WonState(): ctor
+getInstance(): WonState
TwentyFourtyGame
-curState: GameState
+setState(GameState state): void
+getState(): GameState
«interface»
GameState
ContinuingState
-instance = new ContinuingState(): ContinuingState
-ContinuingState(): ctor
+getInstance(): ContinuingState

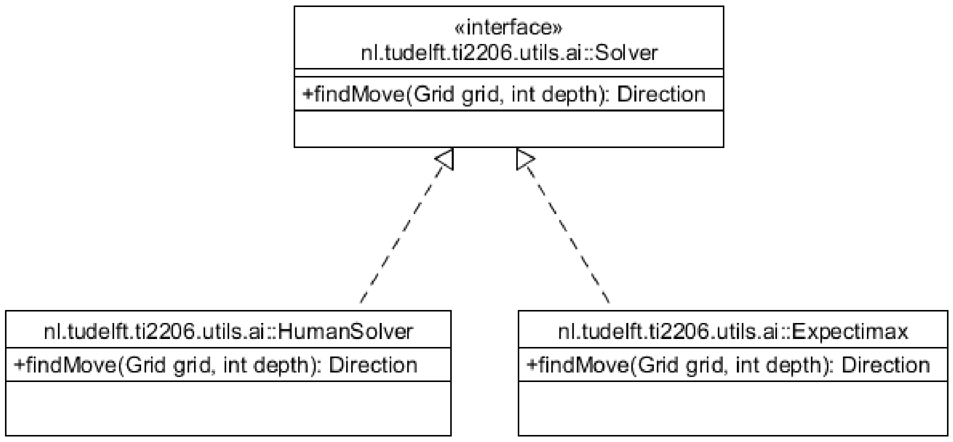


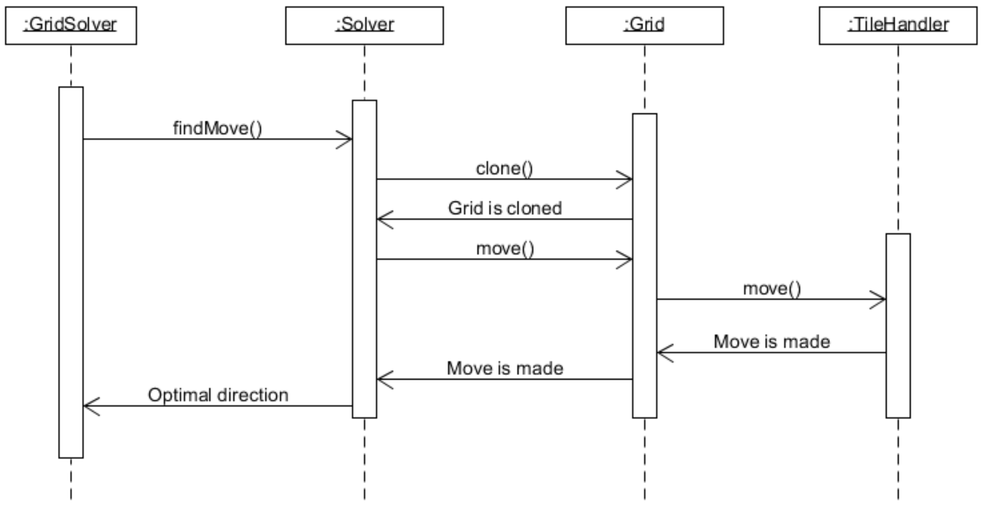
We have also dropped our commandline parsing in favor of a settings menu. The commandline parsing was flagged by inCode, plus it was hard to test. In the new settings menu, you can drag a slider to indicate which log level you want. There is also a check box to enable logging to a file. Of course these settings are saved! In the future, we would like to also list an option here to have you pick the solver you want to use (right now this is hardcoded to the `Expectimax` solver).


nl.tudelft.ti2206.graphics.screens.menuscreens::MenuScreen
-settings: ImageButton
-screenHandler = ScreenHandler.getInstance(): ScreenHandler
opens ▶
1
1
◀ opens
1
1
nl.tudelft.ti2206.graphics.screens.menuscreens::SettingsScreen
-slider: Slider
-levelLabel: Label
-checkBox: CheckBox
-menuButton: MenuButton



The solvers have also gotten some attention again. The solver that wasn't performing well last time has been improved and wins around 50% of its games. Both solvers now get called from the `GridSolver` class, which schedules the task of finding and making a move. To handle two different implementations, we have created a `Solver` interface, which defines one method: `public Direction findmove(Grid grid, int depth);`. Each solver implements this method, which is the starting point for finding and making a move. From here on, the behaviour differs per used algorithm. Using this interface enables us to use polymorphism in the `GridSolver` and thus avoiding the use of heavy branching should we ever add more solvers.


«interface»
nl.tudelft.ti2206.utils.ai::Solver
+findMove(Grid grid, int depth): Direction
nl.tudelft.ti2206.utils.ai::HumanSolver
+findMove(Grid grid, int depth): Direction
nl.tudelft.ti2206.utils.ai::Expectimax
+findMove(Grid grid, int depth): Direction



Lastly, we have cleaned up and updated our Javadoc. One requirement we did not make was the implementation of an AI that will place the new tiles on hard places. This has become quite cumbersome, as the new expectimax implementation does not take this heuristic into consideration anymore. Before using expectimax, we tried implementing the alphabeta algorithm which, amongst others, used this ännoying tile placementäs a heuristic. If we were to implement it now, it would most likely not be part of a solver but of the `Grid` class itself. We could still implement this, but we lost much time on all our other tasks so we decided to let it go.

4 Exercise 2 - Software Metrics

We used inCode to compute software metrics of our project. According to inCode there were no design flaws in the release tagged with v2.1.1. We did find two design flaws in v1.0.0: they are explained below. To complete this exercise, we will describe a third flaw that inCode did not find.

4.1 inCode snapshot

We generated an inCode snapshot of our release tagged with v2.2.1. You may find this snapshot file in `group-21/doc/inCode_snapshot_v2.2.1.result`.

4.2 Design flaws

The most severe design flaw in version 1.0.0 was the Data Class design flaw, therefore this will be described first. The Data clump design flaw will follow.

The last design flaw has never been present in our project, but to complete this exercise we will discuss the Feature Envy design flaw.

4.2.1 Data Class design flaw

In our release tagged with v1.0.0, the `AssetHandler` class was tagged by inCode as affected by the Data Class design flaw.

A Data Class is a class that exposes its data to other classes instead of providing any additional functionality. Because other classes can directly access the data, they contribute to a design that is fragile and hard to maintain.

The `AssetHandler` class was responsible for loading images, sprites and fonts to be used throughout the game. Because we weren't aware of the features LibGDX provides for loading assets we decided this was the most convenient way to load them.

The `AssetHandler` class consisted of a high number of calls to `manager.load()` and had no other responsibilities than loading the files. One issue was that this class was not at all testable, due to Devhub not supporting OpenGL calls. After the first deliverable, it became clear that to easily extend our game we had to severily rewrite it. Whilst doing this, we stumbled upon LibGDX's own implementation for loading and distributing assets: the Skin object.
We incorporated this object into our `AssetHandler` and our two problems were solved: it was now testable, and there was no more design flaw.

After release v1.0.0, this design flaw is not present anymore in our code.

4.2.2 Data Clump design flaw

The Data Clumps design flaw is a concept where methods contain a set of parameters that reappear in multiple classes. There are several conditions that the methods must satisfy to be flagged as data clumps:

There must be different operations that use the same or very similair set of parameters.

The amount of parameters must be equal or higher to the `NOPAR` metric, meaning that there must be five or more parameters to satisfy this condition.

If all these conditions are met, the operations can be considered data clumps. These flaws are usually fixable by encapsulating the parameters into seperate objects and passing them on to the clumping operations. Refactoring these methods into a system where the clumping parameters become an object will not only increase the encapsulation of the code but also decrease the complexity.

To find data clumps in our system we have to revert to version 1.0.0 of our code. In our implementation of the buttons we would construct a button by passing every position attribute it could have. This set of attributes appears in every button class we used, namely the abstract class `SimpleButton` and its derived classes `ContinueButton` and `RestartButton`. Since the constructors of the classes share the same major set of parameter, the methods are classified as data clumps

This design flaw is fixed because of two reasons. The first reason is, again, the major rewrite we have done. We dropped our abstract `SimpleButton` class and instead made our buttons extend LibGDX's `TextButton`. We needed to do this because we were going to need several buttons now and it would be cumbersome having to create a sprite for all of them.
The second reason is the fact that the buttons were now being called from several places. To have consistent looks and to avoid duplicated code, we decided to "statically"set each button's position in the class itself. The result is that we simple have to instantiate the required button and add it to the screen's `Stage`. The button will do the rest and it will always be placed on the same coordinates!

4.2.3 Feature Envy design flaw

This design flaw describes a method or an object that gets the attributes of another class to perform a certain action or calculation. It's better in this case to let the class in which this data is stored perform this action or calculation. Feature envy can be detected as follows:

A method uses far more attributes from other classes than from its own. This means that the LAA metric is less than one third.

A method uses more than a few attributes of other classes directly. This means that the AFTD metric is more than a few.

The used foreign attributes belong to very few other classes. This means that the FDP metric is less than a few.

We have been able to avoid this design flaw in our project. There is a chance, however, that at some point it was indeed present in our game. This will have been in the `Command` class and its subclasses. The `Command` class is an abstract class with only two methods: one for setting a string as the current grid and one for updating the grid.

By not letting the `Grid` class perform the actions on the grid but instead letting a method in the subclass of `Command` do most of the actions, the design flaw could easily have slipped into this class. That is because in this case, the methods in the subclass use far more attributes from `Grid` than its own. This could also happen between the `Command` subclasses and the `TileHandler` class, by not letting the `TileHandler` do the move actions on the grid but instead having the subclass perform these actions.

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

In this section we will discuss how frequently we tested our game. During our code cleanup and small refactoring, our tests came in very handy again to test for reggressions. We did not add much new functionality this sprint, so we did not write much new tests. Tests were run after every change, to verify that there have not been any reggressions.

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 trying to fix the second solver. This is, again, partly because of their randomness, but also because visual testing is just plain easier here: when the static evaluation function changed, it's just easier to run the game and look at the output rather than having to change the unit test and then discovering the random factor is causing your test to fail. Also, a complex piece of code such as the solver is not really testable with unit testing and therefore, we mostly resorted to visual testing.

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: 78.6%

Branch coverage: 74.0%

Instruction coverage: 75.8%

Thanks to our cleanup, our coverage percentages have risen again slightly. As usual, we feel like our game has been sufficiently tested.