Deliverable 3

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

27 september 2014

1 Introduction

2048 is a very popular game created by by Gabriele Cirulli, based on 1024 by Veewo Studio and conceptually similar to Threes by Asher Vollmer.

We are five Computer Science students at the Delft University of Technology, The Netherlands. We created a clone of the original 2048 game for the course Software Engineering Methods.

We didn't just make a clone of the original 2048 game, we also added multiplayer functionality. Now you can challenge your neighbours, your coworkers, your kids or even the Queen of England to play a game of 2048!

This document tells you everything you need to know to get started. Technical aspects of the game are covered in here as well.

Setting up a multiplayer game is really straightforward, however. Just make sure incoming connections to port TCP/2048 reach the hosting machine! Yes, port 2048! Who would have thunk* it? More details are up ahead!

Good luck and have fun!

P.S.: We, the developers, are not reliable in case of frustration. ;)

***** See: http://english.stackexchange.com/questions/55577/proper-usage-of-the-word-thunk

2 How to play 2048

This section briefly describes how to play 2048 and provides information on the functionality it has, such as playing the game alone, with friends and how to use the logging features.

2.1 Singleplayer game

After starting the application you will see the main menu. In the main menu, click the `Singleplayer` button to start your singleplayer game.

You move the tiles with the arrow keys. Each time two tiles with the same number collide, the numbers are added and the two tiles merge. Your goal is to reach the 2048 tile!

To return to the menu, press the `Escape` button any time. Don't worry, your current game will be saved for you! (This also applies to closing the game!).

2.2 Multiplayer game

The multiplayer version is identical to the singleplayer, except here you will compete against a friend, colleague, coworker or your worst enemy over LAN or the internet. Your opponent does not have to be in the same room with you; they can even be on the other side of the planet and you can still kick their asses!

Your goal is to reach the 2048 tile *before* your opponent does. In case you are unable to reach the 2048 tile (e.g., because you died), your opponent automatically wins.

We will now briefly explain how to connect to eachother. Please refer to the documentation of your networking equipment or software in case you experience networking problems.

2.2.1 Joining a game

To connect to another player, choose the `Join a game` button in the main menu. The application will try to connect to the remote address you entered, on port 2048, using TCP.

2.2.2 Hosting a game

To have another player connect to you, choose the `Host a game` button in the main menu. The application will bind to port 2048/TCP on all the system's network interfaces. In case you wish to play over the internet, please make sure connections on this port are forwarded to your local address on your NAT device.

2.3 Logging

The game supports several commandline arguments for logging.

By default, the application will log to the standard output, using the `INFO` logging level. If enabled, however, errors will be logged to `stderr`. The logging level can also be adjusted.

The supported arguments are:

```
$ jarfile.jar [logLevel] [file]
```

or, otherwise:

```
$ Launcher.java [logLevel] [file]
```

Both of these fields are parsed case-insensitively.

Two examples:

```
$ Launcher.java debug
```

will run the game and log all debug and info messages.

```
$ Launcher.java error file
```

will run the game and log all debug, error and info messages to the system's output streams (`stdout` and `stderr`) and will write them to a new file as well.

Please see the corresponding section below for more information on the possible arguments:

logLevel can be one of the following:

all logs all messages;

info logs info messages only;

error log error messages and info messages;

debug log debug, error and info messages;

none disables logging.

file
Setting the `file` flag will write all messages of the previously set logging level to a file. By default, a new file with the format `2048_YYYYMMDD_HHmmss.log` will be created, where `YYYMMMDDD_HHmmss` is the time of application start.

3 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.

3.1 Test frequency

In this section we will discuss how frequently we tested our game. Due to the design patters we implemented, testing was essential. We tested using our unit tests (locally and on Devhub as well) and visual tests. Implementing iterators caused some problems that only arised during our visual tests. These were, however, resolved immediately.

3.2 Testing methods

Visual tests involved actually playing the game and analyzing logging output manually. Unit tests simply check object properties with certain input.

3.3 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: 72.3%

Branch coverage: 62.0%

Instruction coverage: 68.8%

As with previous deliverables, we faced the same issues with code that requires graphically rendering our game.

3.4 Conclusion

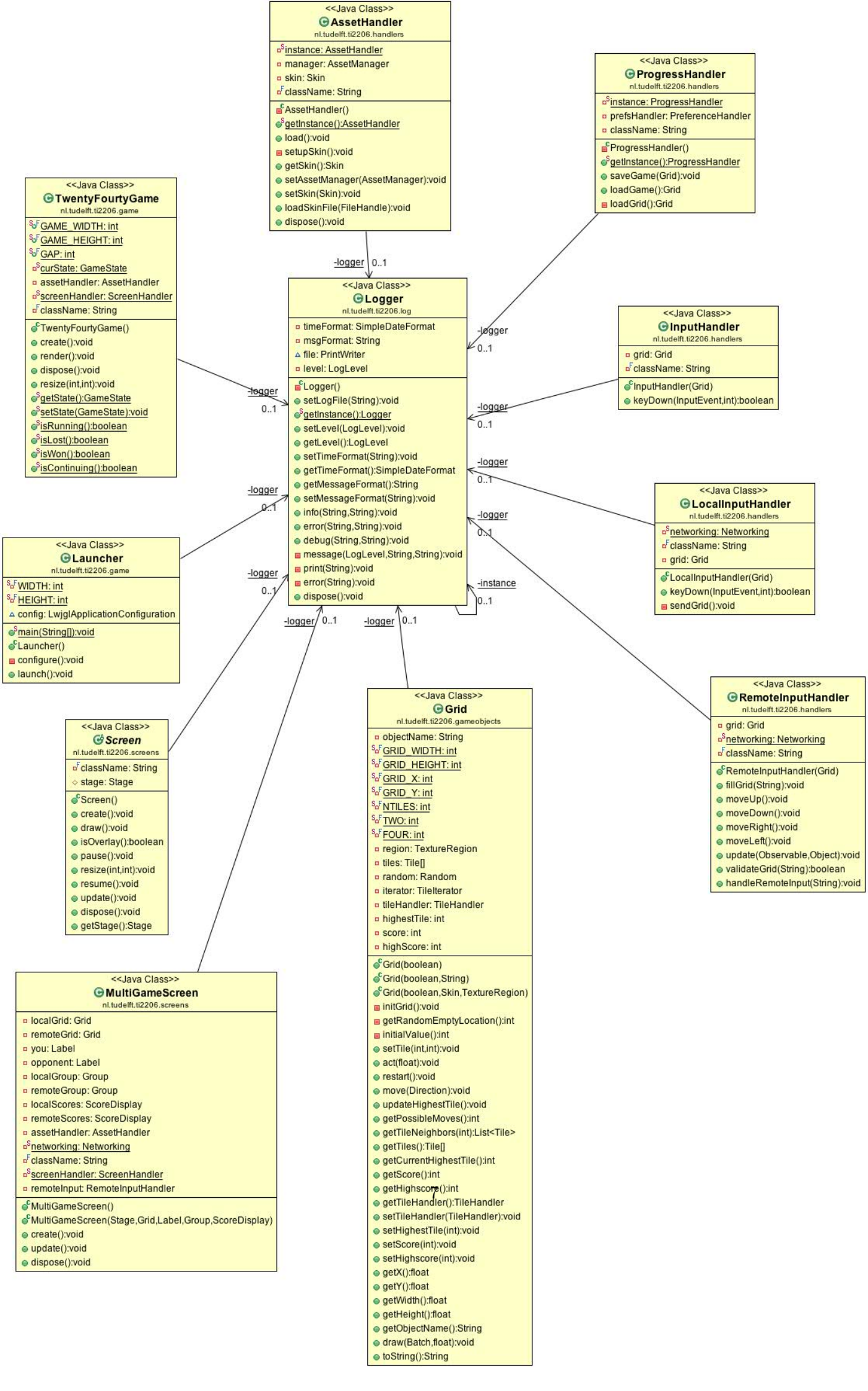
Although our test results are lower than we had planned to achieve, we believe our project has again been tested sufficiently.

4 Exercise 1 - Simple logging

The logging functionality has been implementer per our requirements (please see the `RequirementsLogger` file for these). The following CRC card was made during the design phase of this extension:

Logger	
Superclass: none	
Subclasses: none	
Purpose:	***Collaborator:***
Logging the input of the player	InputHandler
Logging the input of the local player	LocalInputHandler
Logging the input of the remote player	RemoteInputHandler
Logging the creating of screens	Screen
Logging the loading of a grid	ProgressHandler
Logging the saving of a grid	ProgressHandler
Logging of loading the assets	AssetHandler
Logging the spawning of a new tile	Grid
Logging the current score	Grid
Logging if the move is conducted	Grid
Logging the state of the game	TwentyFourtyGame
Logging if a player as won	MultiGameScreen
Logging if the server is listining	Networking
Logging if a connection is established	Networking
Logging if there is a network error	Networking

We have also made a small class diagram:


<<Java Class>>
AssetHandler
nl.tudelft.ti2206.handlers
instance: AssetHandler
manager: AssetManager
skin: Skin
className: String
AssetHandler()
getInstance():AssetHandler
load():void
setupSkin():void
getSkin():Skin
setAssetManager(AssetManager):void
setSkin(Skin):void
loadSkinFile(FileHandle):void
dispose():void
<<Java Class>>
ProgressHandler
nl.tudelft.ti2206.handlers
instance: ProgressHandler
prefsHandler: PreferenceHandler
className: String
ProgressHandler()
getInstance():ProgressHandler
saveGame(Grid):void
loadGame():Grid
loadGrid():Grid
<<Java Class>>
TwentyFourtyGame
nl.tudelft.ti2206.game
GAME_WIDTH: int
GAME_HEIGHT: int
GAP: int
curState: GameState
assetHandler: AssetHandler
screenHandler: ScreenHandler
className: String
TwentyFourtyGame()
create():void
render():void
dispose():void
resize(int,int):void
getState():GameState
setState(GameState):void
isRunning():boolean
isLost():boolean
isWon():boolean
isContinuing():boolean
-logger
0..1
<<Java Class>>
Logger
nl.tudelft.ti2206.log
timeFormat: SimpleDateFormat
msgFormat: String
file: PrintWriter
level: LogLevel
Logger()
setLogFile(String):void
getInstance():Logger
setLevel(LogLevel):void
getLevel():LogLevel
setTimeFormat(String):void
getTimeFormat():SimpleDateFormat
getMessageFormat():String
setMessageFormat(String):void
info(String,String):void
error(String,String):void
debug(String,String):void
message(LogLevel,String,String):void
print(String):void
error(String):void
dispose():void
<<Java Class>>
InputHandler
nl.tudelft.ti2206.handlers
grid: Grid
className: String
InputHandler(Grid)
keyDown(InputEvent,int):boolean
<<Java Class>>
LocalInputHandler
nl.tudelft.ti2206.handlers
networking: Networking
className: String
grid: Grid
LocalInputHandler(Grid)
keyDown(InputEvent,int):boolean
sendGrid():void
-instance
0..1
<<Java Class>>
Launcher
nl.tudelft.ti2206.game
WIDTH: int
HEIGHT: int
config: LwjglApplicationConfiguration
main(String[]):void
Launcher()
configure():void
launch():void
<<Java Class>>
Screen
nl.tudelft.ti2206.screens
className: String
stage: Stage
Screen()
create():void
draw():void
isOverlay():boolean
pause():void
resize(int,int):void
resume():void
update():void
dispose():void
getStage():Stage
<<Java Class>>
Grid
nl.tudelft.ti2206.gameobjects
objectName: String
GRID_WIDTH: int
GRID_HEIGHT: int
GRID_X: int
GRID_Y: int
NTILES: int
TWO: int
FOUR: int
region: TextureRegion
tiles: Tile[]
random: Random
iterator: TileIterator
tileHandler: TileHandler
highestTile: int
score: int
highScore: int
Grid(boolean)
Grid(boolean,String)
Grid(boolean,Skin,TextureRegion)
initGrid():void
getRandomEmptyLocation():int
initialValue():int
setTile(int,int):void
act(float):void
restart():void
move(Direction):void
updateHighestTile():void
getPossibleMoves():int
getTileNeighbors(int):List<Tile>
getTiles():Tile[]
getCurrentHighestTile():int
getScore():int
getHighscore():int
getTileHandler():TileHandler
setTileHandler(TileHandler):void
setHighestTile(int):void
setScore(int):void
setHighscore(int):void
getX():float
getY():float
getWidth():float
getHeight():float
getObjectName():String
draw(Batch,float):void
toString():String
<<Java Class>>
RemoteInputHandler
nl.tudelft.ti2206.handlers
grid: Grid
networking: Networking
className: String
RemoteInputHandler(Grid)
fillGrid(String):void
moveUp():void
moveDown():void
moveRight():void
moveLeft():void
update(Observable,Object):void
validateGrid(String):boolean
handleRemoteInput(String):void
<<Java Class>>
MultiGameScreen
nl.tudelft.ti2206.screens
localGrid: Grid
remoteGrid: Grid
you: Label
opponent: Label
localGroup: Group
remoteGroup: Group
localScores: ScoreDisplay
remoteScores: ScoreDisplay
assetHandler: AssetHandler
networking: Networking
className: String
screenHandler: ScreenHandler
remoteInput: RemoteInputHandler
MultiGameScreen()
MultiGameScreen(Stage,Grid,Label,Group,ScoreDisplay)
create():void
update():void
dispose():void

5 Exercise 2 - Design patterns

In this section we will discuss the three chosen design patterns. Per pattern, we will provide (in natural language) a description of why and how the pattern is implemented, together with a class- and sequence diagram.

5.1 The Singleton design pattern

We implemented the singleton design pattern because we noticed the large amount of classes that only contained static methods. Nearly all of them are located in the `handlers` package. This is logical, since handlers are supposed to carry out one specific task, without the need of an instance of the class.

We noticed that the code did not really look clean with all those classnames in front of methods. Instead, we could declare an instance variable in each class using the static methods to access those. A requirement would be that only one instance is allowed to exist, as it would make no sense to have multiple instances when the class does not maintain any state. This is exactly what the Singleton design pattern is for.

5.1.1 The implementation

First of all, we started refactoring the classes that were going to follow the singleton design pattern. A static instance variable of the singleton class was created, initializing the unique instance of the class:

```
private static Singleton instance = new Singleton();
```

Since this instance is static, it already exists before a the class instance can exist. Therefore, this approach is thread safe.

A private constructor was added to override the default constructor:

```
private Singleton() {}
```

This constructor cannot be accessed from outside the singleton class because it has the private modifier, so no other class can even try to create a new instance of the singleton.

Then, a public static method was added for other classes to retrieve the singleton instance:

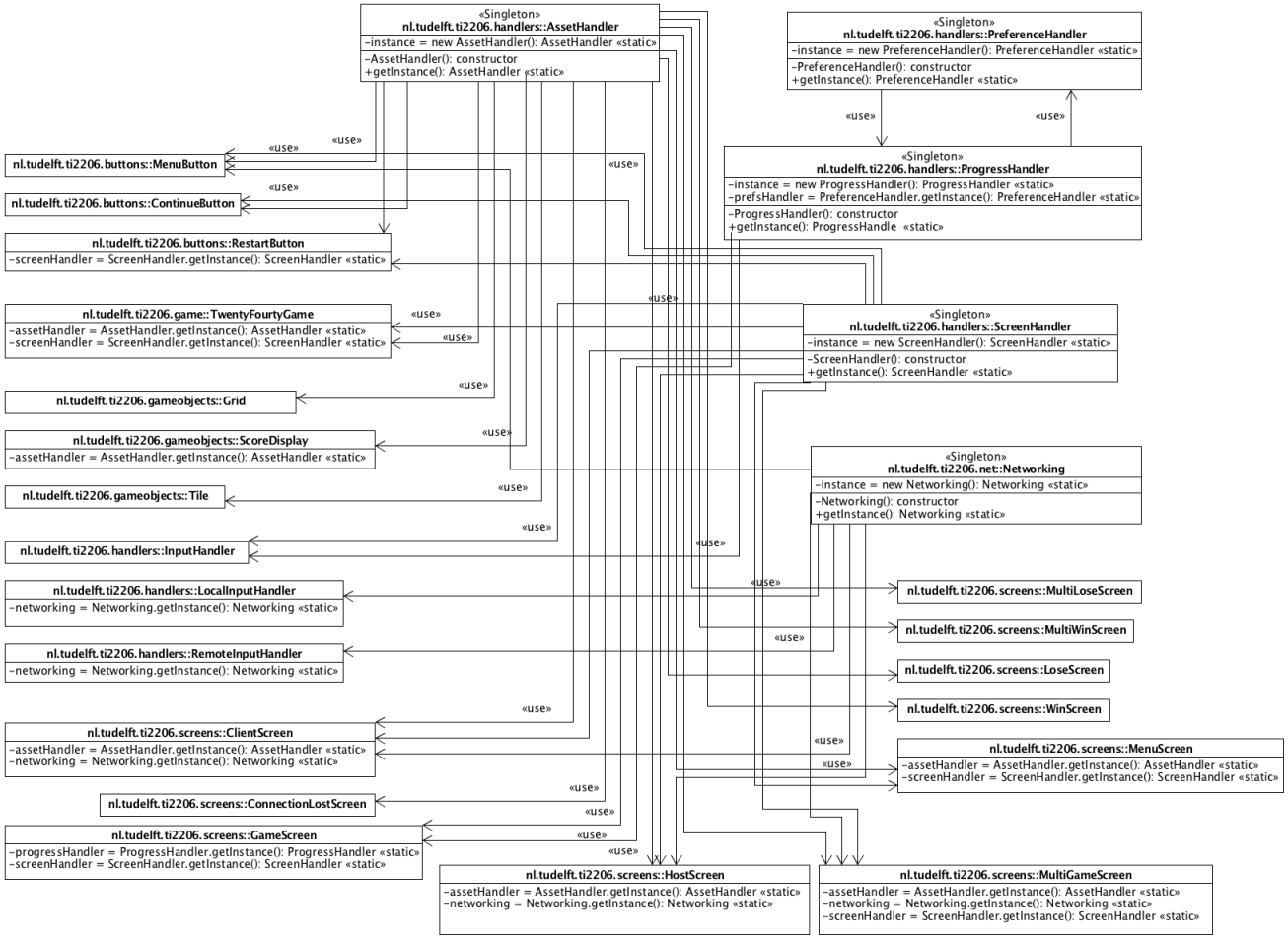
```
public static Singleton getInstance() {
    return instance;
}
```

Since this method is static, the singleton is accessed without having to create a new instance of the class. In classes that use the instance, we could just add an instance variable to retrieve it once and use it throughout the code.

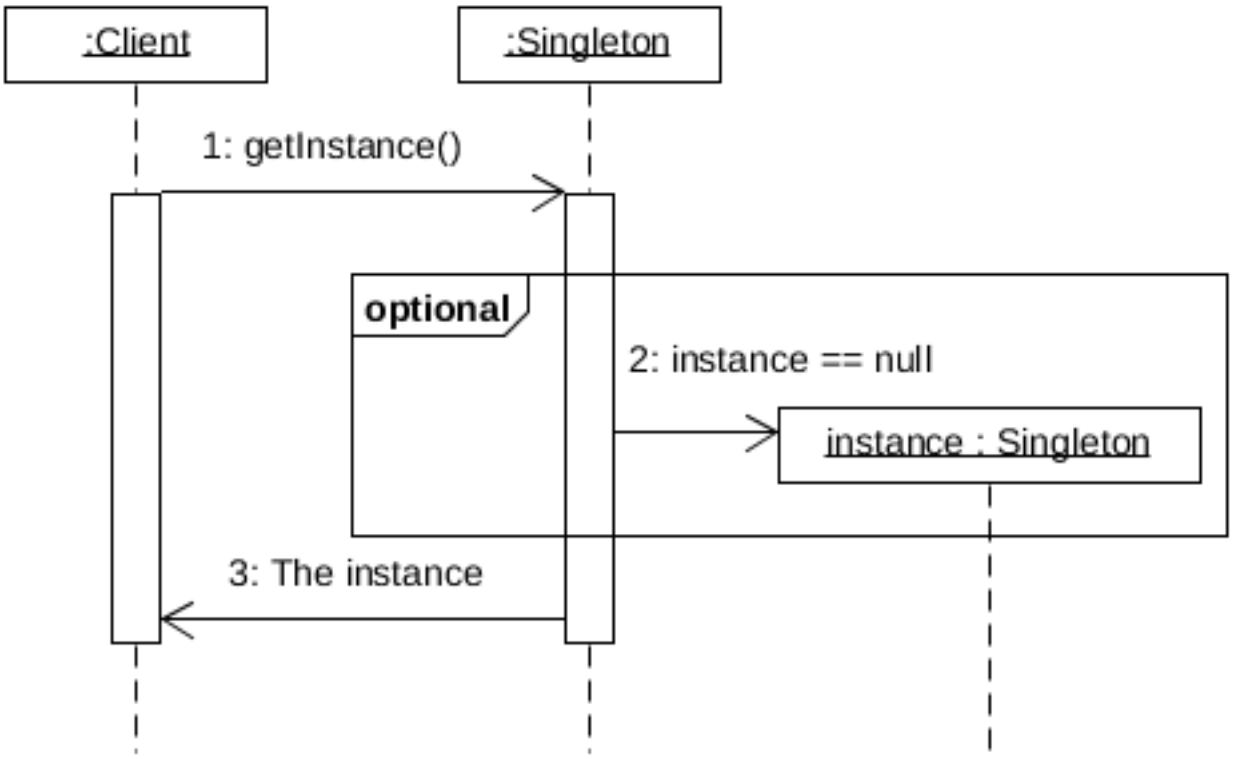
For this to actually work, the methods that used to be static were refactored to public methods in order to be able to access them through the instance of the singleton.

5.1.2 The class- and sequence diagram

The class diagram:



The sequence diagram:



5.2 The Iterator design pattern

We implemented the iterator design pattern because we have to iterate over our grid a lot. It seemed a natural choise to implement an iterator on top of this. The prospect of having to refactor our `TileHandler` class was daunting, though, so we never did it. This assignment, however, gave us the needed motivation to go ahead and do it.

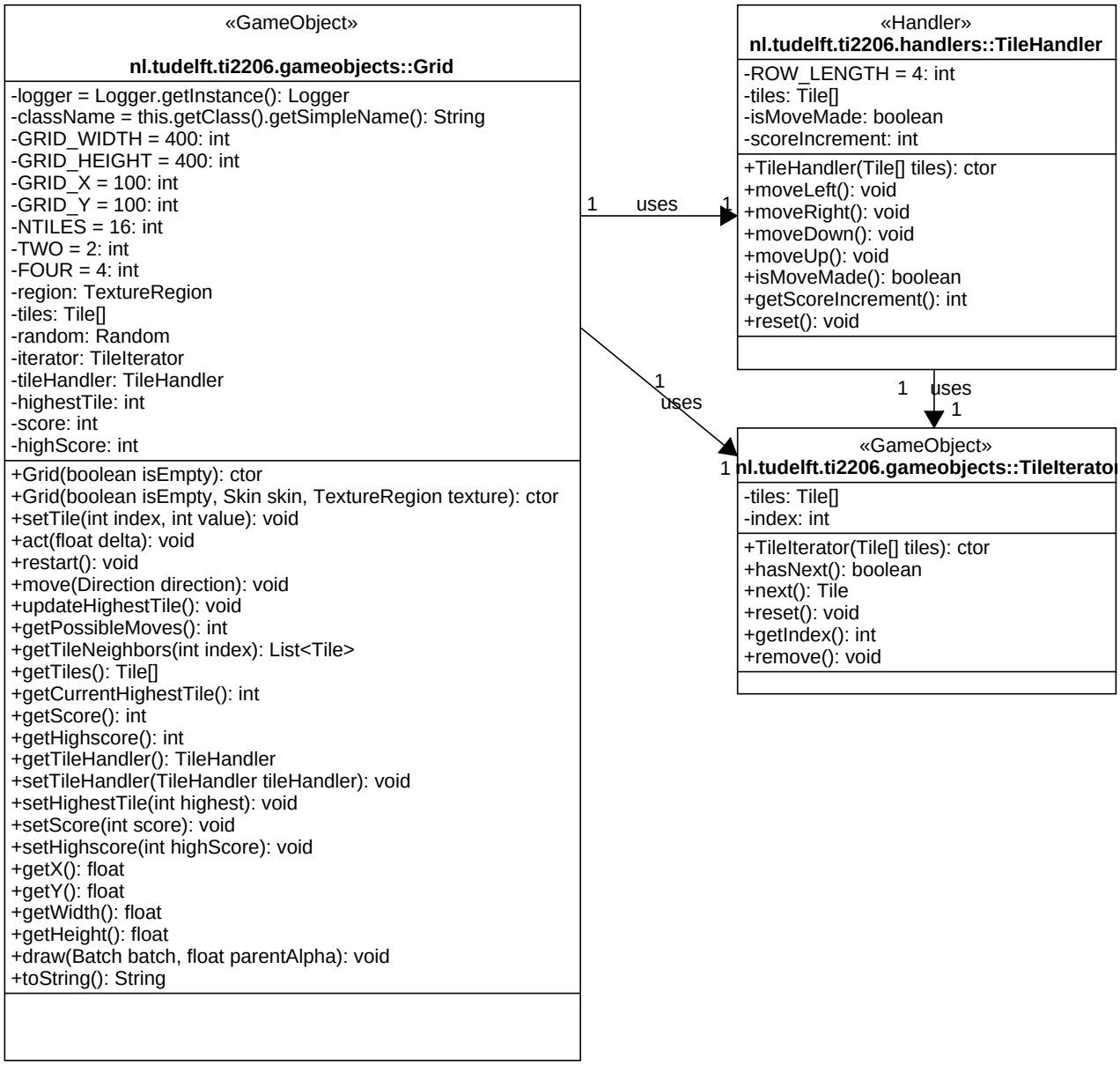
5.2.1 The implementation

A new class, the `TileIterator` class, has been created. This class implements the `Iterator` interface that is defined in `java.util`. The classes that have to iterate over the grid now make use of this `TileIterator`.

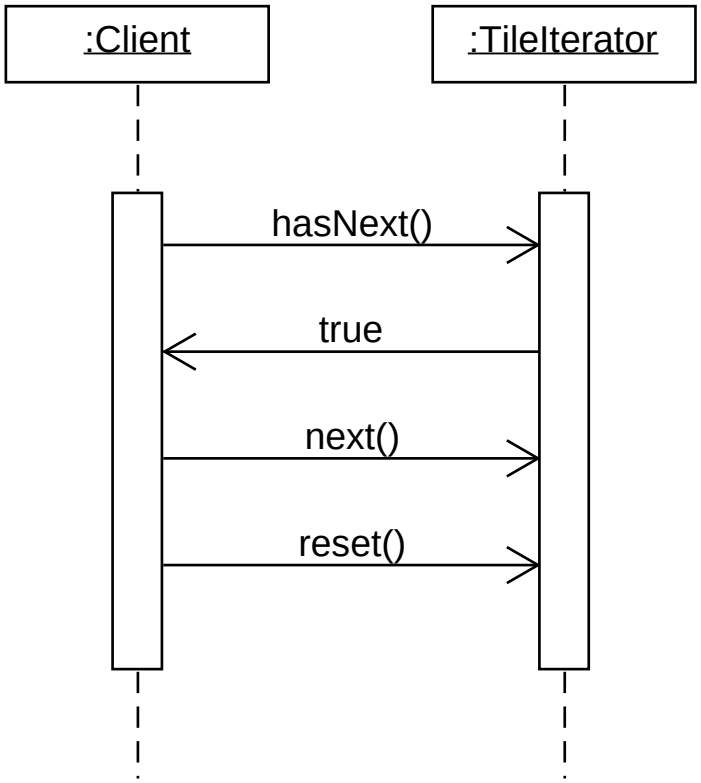
One detail we would like to point out, is that our `TileIterator` also has a reset method. This method will move the index back to the beginning of the array, so the iterator can be used again. This allows us to have only one instance of `TileIterator` inside the `Grid` class instead of every method creating an iterator when needed. This would mean that several new `TileIterator` instances would be created sixty times per second! The same trick is used inside `TileHandler`, although the impact here is less big.

5.2.2 The class- and sequence diagram

The class diagram:



The sequence diagram:



5.3 The Observer design pattern

The `Networking` class is required to synchronize movements and the grids.

In previous releases of 2048, we would give a handle to the `RemoteInputHandler` object to the `Networking` class. This way, however, attaches `RemoteInputHandler` entirely to `Networking`.

To circumvent this in our new version, the `RemoteInputHandler` registers itself to `Networking` as an `Observer` so that the `Observable` (the `Networking` class) can notify the `RemoteInputHandler` whenever it needs to be updated.

5.3.1 The implementation

The `RemoteInputHandler` class simply gets a singleton instance of the `Networking` class:

```
private static Networking networking = Networking.getInstance();
```

Then, in its constructor, it calls the `addObserver` method on the `Networking` object and it registers itself as a observer:

```
networking.addObserver(this);
```

The `Networking` class can now send objects to the `RemoteInputHandler` without them being fully attached to eachother. Whenever `Networking` needs to send an object to `RemoteInputHandler` it can simply set a flag to a changed state, using:

```
setChanged();
```

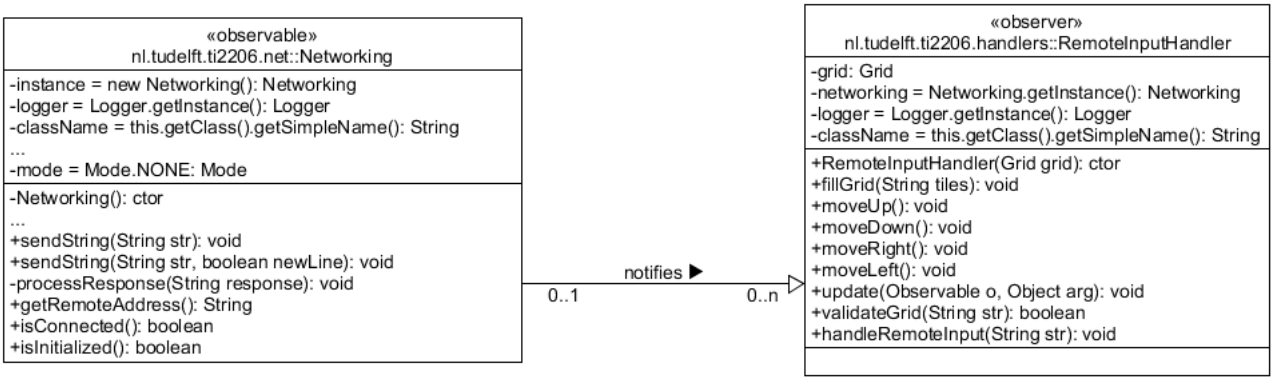
and then simply notify all observers that were registered in the past:

```
notifyObservers(response);
```

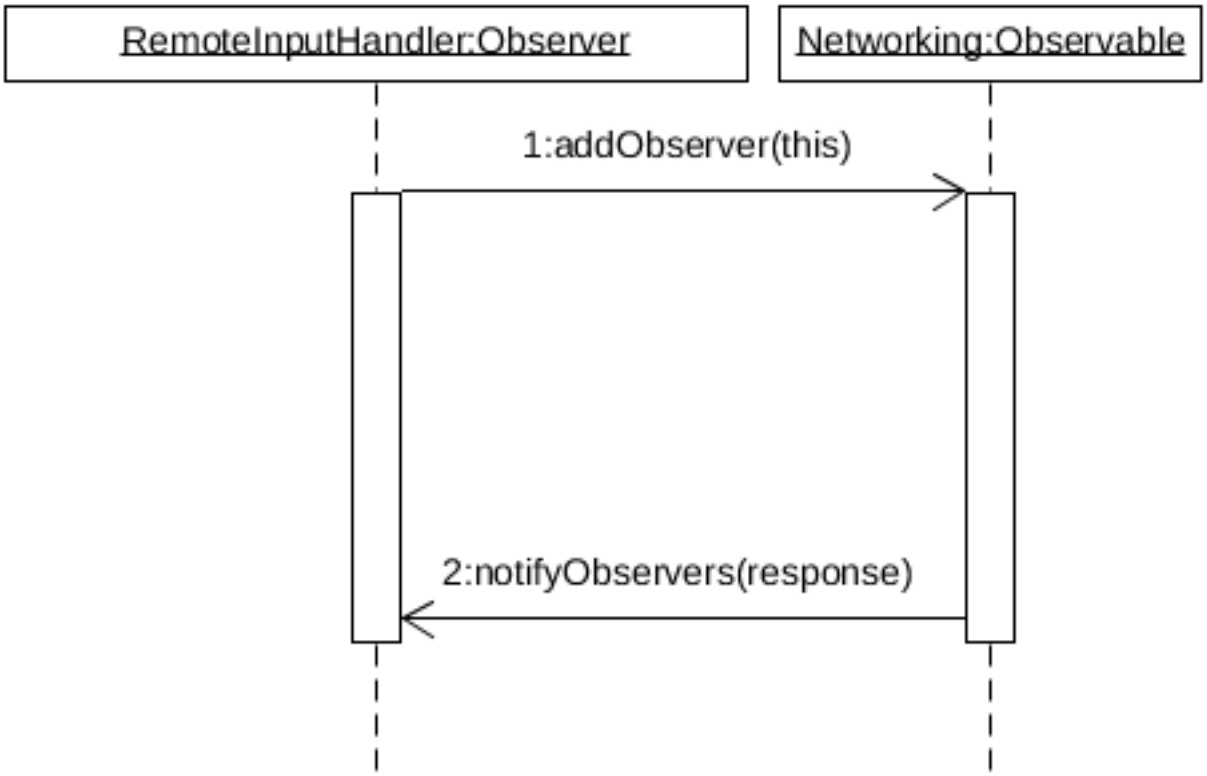
The benefits of this design are that `Networking` can communicate with other objects while not being fully attached to them and newly added objects can easily be registered to the list of observers, allowing the application to be maintained and extended in an easier way.

5.3.2 The class- and sequence diagram

The class diagram:



The sequence diagram:



6 Exercise 3 - One more design pattern

We use a `Screen` hierarchy to define some of the GUI aspects that we use in the game. The parent, `Screen`, has a draw method among others that are extended to the child classes. These child classes would use one of two implementations of the draw method: the implementation that was extended from the parent, or override into a different implementation. Because of this, the classes that override the existing method would have duplicate code.

By applying the Strategy design pattern to our existing hierarchy we can not only reduce duplicate code but also prepare our program to further changes. This has been done by encapsulating the different implementations in a family of algorithms. Every class can now add one of the objects to use an implementation dynamically (even in runtime), instead of implementing it by themselves. As the program grows and a new set of screens with a different implementation of the draw method is added, we can simply add a new class to the family in which define the implementation, without forcing any changes in the existing classes.

6.0.3 The implementation

The pattern has been implemented by moving the existing implementations of the draw method into separated classes that implement the `DrawBehavior` interface. This interface contains a draw method that the implementing classes override with their own implementation. By adding a local variable to the `Screen` class and a method to assign a new `DrawBehavior` object to it, a `DrawBehavior` object can be assigned to every screen object to determine its draw implementation.

6.0.4 The class- and sequence diagram

The class diagram:

The sequence diagram: