



DEPARTMENT OF
COMPUTER TECHNOLOGY AND INFORMATICS

IDATG2001 - PROGRAMMING 2

WarGames - Final Report

Author:
Kjetil Karstensen Indrehus

May, 2022

Table of Contents

1	Introduction	1
1.1	Overview	1
1.2	Procedure	1
1.2.1	Part 1	1
1.2.2	Part 2	1
1.2.3	Part 3	2
2	Requirements	3
2.1	Description	3
2.2	Use Case Diagram(s)	3
2.2.1	Setup page	3
2.2.2	Simulation	4
3	Design Pattern(s)	5
3.1	Inheritance	5
3.2	Facade	7
3.3	Singleton	8
3.4	Factory	9
3.5	Template Method	10
4	Graphic Design & Usability	11
4.1	Wireframe	11
4.2	Pixel art	13
4.2.1	Tools	13
4.2.2	Result	13
4.3	Don Norman's Design Principles	13
4.3.1	Visibility	14
4.3.2	Feedback	14
4.3.3	Affordance	14
4.3.4	Mapping	14
4.3.5	Constraints	14
4.3.6	Consistency	14
5	Technical Details	15
5.1	Packages/Dependencies	15

5.2	FXML	16
5.3	Extra: Python Script	16
6	Process	18
6.1	Version Control	18
6.2	Management and Scheduling	18
6.2.1	Deadlines	18
6.2.2	Managing Task(s)	18
7	Reflection	22
7.1	Problem: How to show the simulation?	22
7.2	Problem: How to position the Units?	23
7.3	Problem: Terrain Bonus	24
7.4	Maven and IntelliJ Issues	24
7.4.1	No Tests Found	24
7.4.2	Forked VM	25
7.4.3	IntelliJ Issue	25
7.4.4	Maven Issue: Create Jar	26
7.4.5	Solution	26
7.5	Implement an Observer?	27
8	Conclusion	28
	List of Figures	29
	References	30

1 Introduction

1.1 Overview

This is the main report for the final software that has been developed in IDATG2001. The following task description were given:

You will develop a software called Wargames, which simulates a battle between two armies in a war. In this first part, you will focus on the code that deals with devices and simulation logic. In part 2, you will expand the program with i.a. file manager and a graphical user interface. In Part 3, apply design patterns and add more functionality. The complete program will be delivered to your folder at the end.

1.2 Procedure

The following section will cover the different parts of the software. It will not include all the technical details, but will give a brief overview over the different task given in each stage of the development. For more details, see the different PDF files given by the professor.

1.2.1 Part 1

Tasks

1. Create an empty Maven Project and make sure to have version control with Git.
2. Create the *Super Class* Unit. The abstract methods should be:
 - getAttackBonus()
 - getResistBonus()
3. Create the following classes that implements the Unit class:
 - InfantryUnit
 - CavalryUnit
 - RangedUnit
 - CommanderUnit
4. Create Army Class for storing a list of Units.
5. Create Battle Class that should take two armies and simulate attack.
6. Test all classes by creating a program in the terminal (Voluntary task).

1.2.2 Part 2

Tasks

1. Create methods using lambda code, to create methods to get all units of the same type. (For example; getInfantryUnits() should return a list of all infantry units.)
2. Make the program use File to store information. The requirements are:
 - It should be possible to save an army in a file
 - It should be possible to read an army from a file
 - The file must be a text file and follow a specific format (see the full requirements in *Wargames Del 2*)
3. Make a wireframe for the software.

1.2.3 Part 3

Tasks

1. Use the *Factory* design pattern to implement a Unit Factory
2. Add a Terrain for the Battle, and make the units stronger or weaker based on the terrain.
3. Create GUI: Use either clean Java code or *FXML* files to create pages.
4. Implement any changes that makes the software better (Freedom to improve)

2 Requirements

2.1 Description

For a software to work as intended, the relation between user and software, need to be clarified. To show the relation and how each feature is used see the added use case diagrams.

A *use case diagram* shows how a actor (the user) interacts with the system (the software). It does not show any technical detail on how the system achieves a goal. However, it shows the scope of the system. For more details on *Use Case Diagram* (what they show, the goal, how to interpret and description of professional concepts) , see: [12].

2.2 Use Case Diagram(s)

2.2.1 Setup page

For the setup page, the actor has a lot of control of how the armies are setup. Each use case that has a association (drawn as a line) with the actor is a button on the GUI. The internal tasks are handled by the software. For example, the user does not need to create a new *.csv* file.

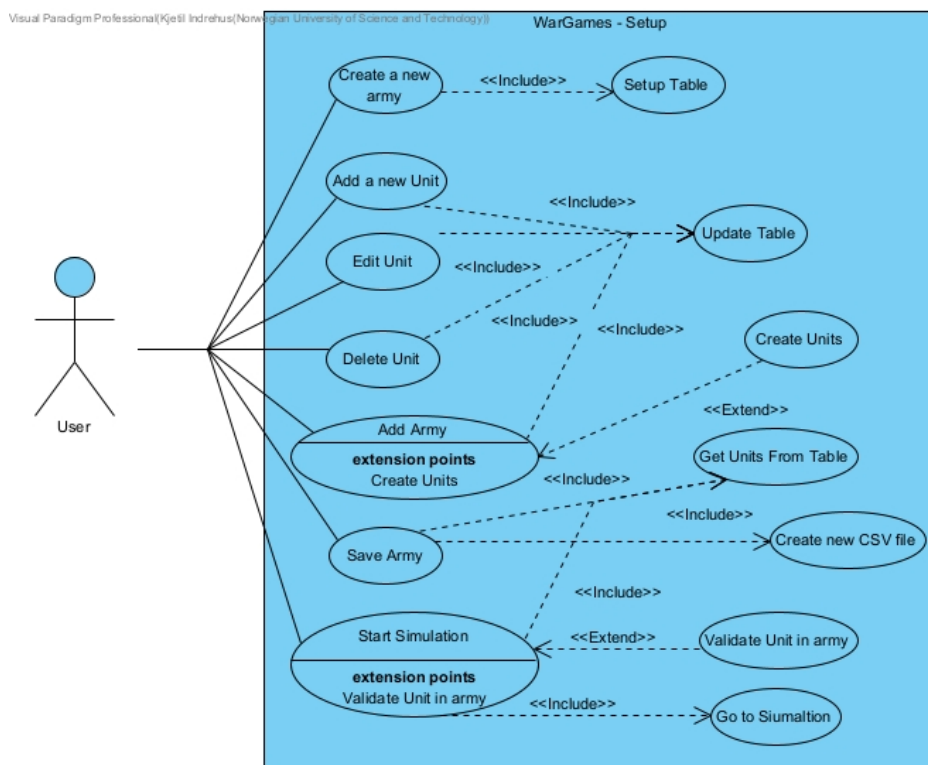


Figure 1: Setup Page

2.2.2 Simulation

The simulation page has different tabs. Each tab display different type of information. There are four buttons that are relevant to show. They all control a timeline in the application. This timeline calls a function that execute one step. Before each step we have to check if we have enough units in each army to have a single attack, then the attack function is called. The last step is to gather new information after the attack, and update all ways of showing information. Also the attack removes any units in the army that has zero health. This is not very technical detailed, but the goal of the user case diagram is to show what functionality is handled by the system and what the user can do.

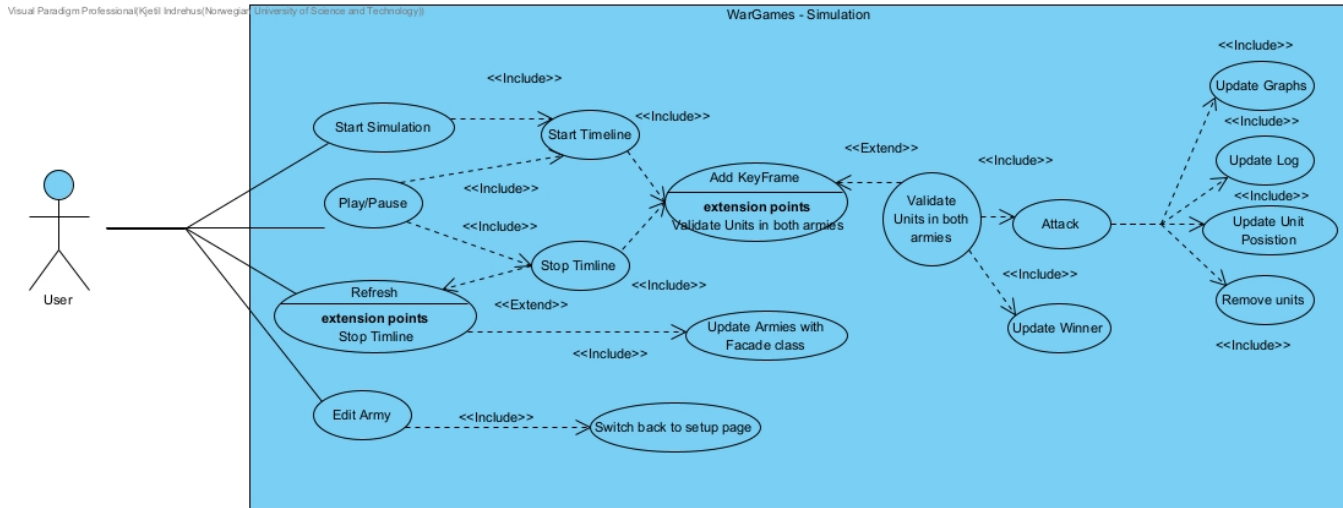


Figure 2: Simulation page

3 Design Pattern(s)

In this section will there be some overview of the theory behind some of the design patterns that has been used in the software. Furthermore, a brief explanation on how each design pattern was implemented.

3.1 Inheritance

One key design pattern was also a requirement was to use inheritance. Here are some reasons why we use inheritance;

- Minimize duplicate code.
- Allows polymorphism.
- More organised and efficient code.

This concept is used in the *Unit* class. Since all units will have some key functionality, we say that a specific unit type is a subclass of the *Unit* class. All unit types will have name, health and armor. They will also have attack and defence bonus. Also they will have a terrain bonus. They may be different values but they all act the same.

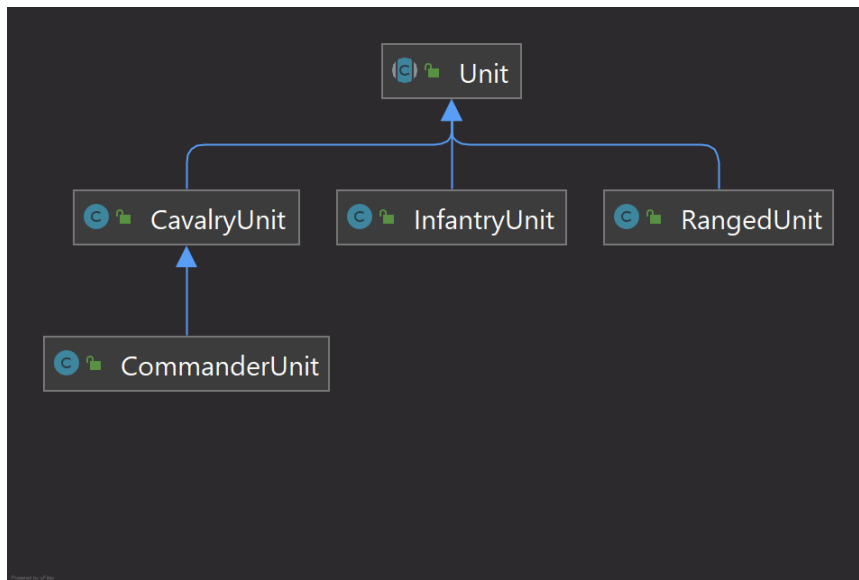


Figure 3: Class diagram of backend

Above is a class diagram that shows how the superclass *Unit* is used by the other subclasses. Below are the specific methods that are in the *Unit* class. The abstract methods are **getAttackBonus()**, **getDefenceBonus()**, **getTerrainBonusAttackDefence()** and **getUnitType()**. They need to be implemented when creating a new class that extends the *Unit* class.

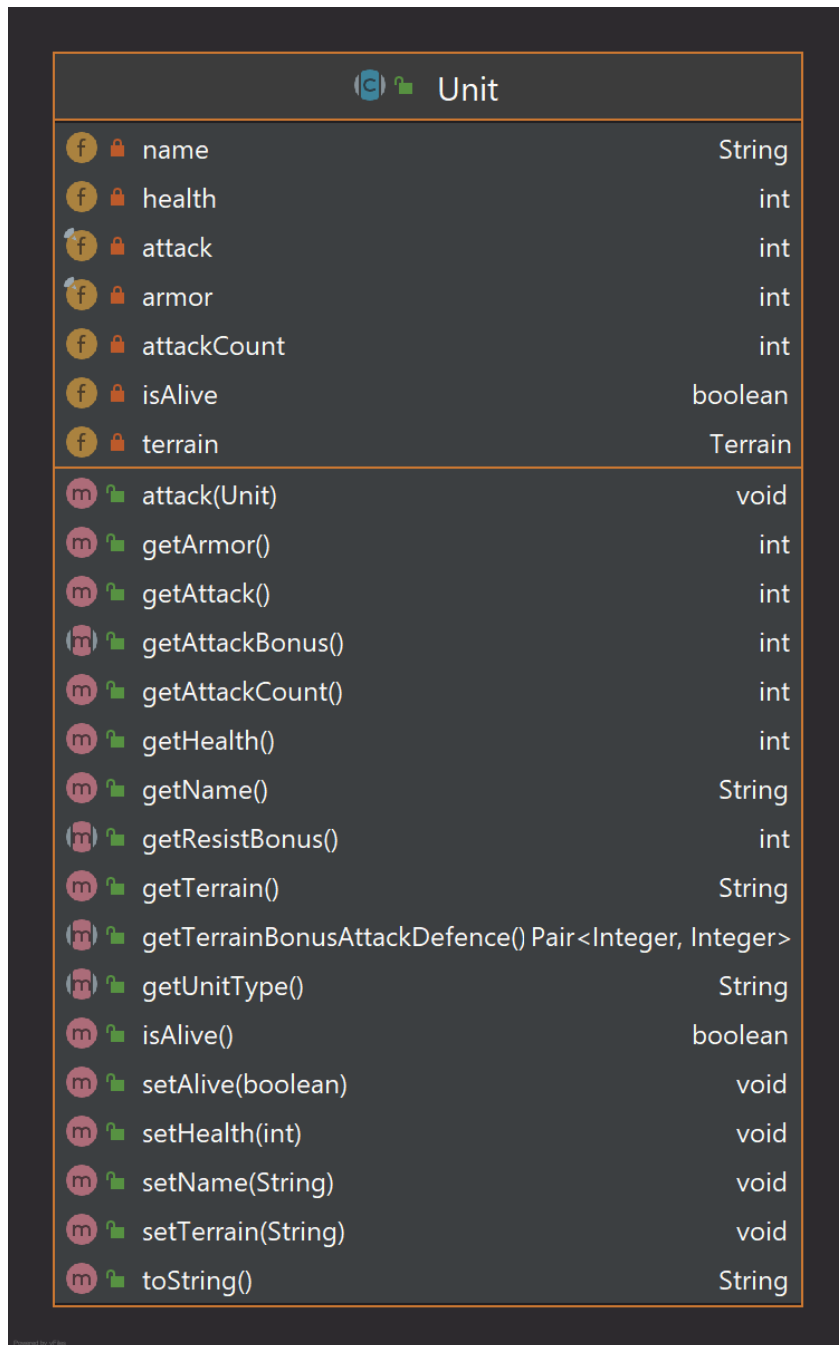


Figure 4: Unit Class Diagram

3.2 Facade

Facade is a pattern that is necessary to better simplicity and overall structure. It is a own class that holds information from the other classes. The private constructor creates new instances of other classes. When calling the class, it creates a new instance of the class. Here is the code:

```
1
2  /**
3   * Method that gets the instance of the facade class.
4   * If the facade has not been called before, it creates a new instance.
5   *
6   * @return returns the instance of the facade class.
7   */
8
9  public static Facade getInstance() {
10     if(instance == null){
11         synchronized (Facade.class){
12             instance = new Facade();
13         }
14     }
15     return instance;
16 }
```

The reason to have a facade is because it can store information about the two armies in one class. Then when I need to switch to the simulation page;

1. Get information for both armies, and add them to the Facade armies.
2. Get the name of the armies and update the Facade
3. Use the Facade class to get the two different armies.

It might be hard to understand why there is necessary to use a class to hold instances of other classes. The main reason is to make it easier to transfer all the information from one controller to another. What if there was not 3 instances of classes, but 25! Each instance could be stored in a single class. To illustrate this better, see the how Refactoring Guru shows visually [3] ;

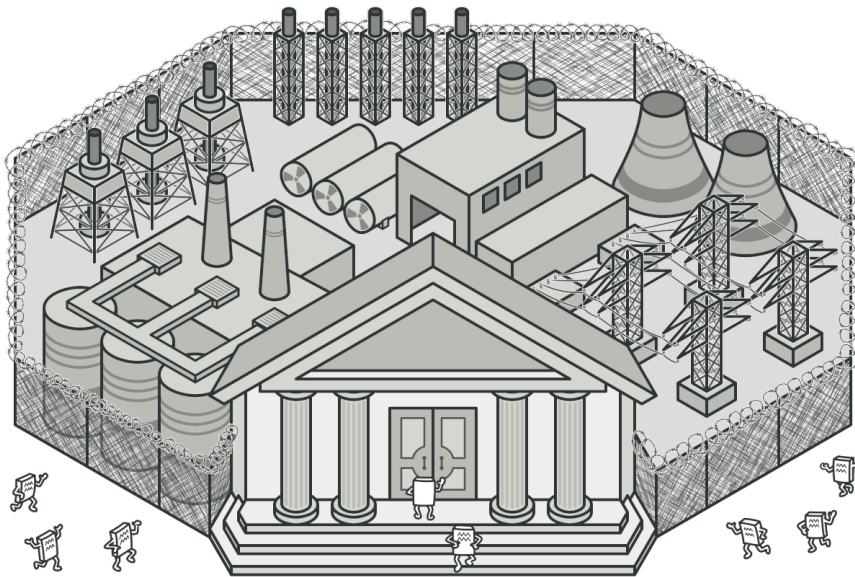


Figure 5: A facade hides other information in a single class.

3.3 Singleton

A singleton is a design pattern that insures that only one instance of the class has been made. The code is almost the same as the *Facade design Pattern* [5]. The key difference is that you only have a single class.

Why would you separate a single instance in its own class? Why not add it to the Facade? The reason why is because of *java.util.Random* class (see java doc here: [9]). When a new Random object is created, then a new Random number generator is generated. This instance should only be created once to make the Random truly random. The generator (Random object) is based on linear recurrences. For more information on Pseudorandom number generator, see [15].

Random is used by a lot of different classes. Therefor a singleton insures that only one instance of random is created, and all classes uses the Singleton class. Refactoring Guru [5], uses this image, to show how all classes are dependent on one instance of the Singleton;

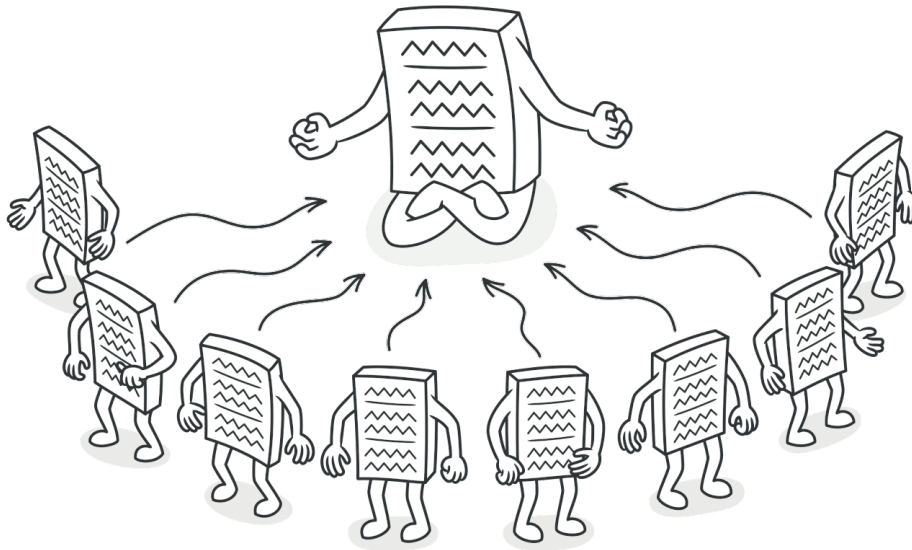


Figure 6: Random Instance is used by other classes.

3.4 Factory

A factory is a creational design pattern that creates an interface for creating new object.

In the application, the user should be able to add random units of the same name. Name of the unit might not be relevant. We want the user to be able to quickly add units to a army. The factory is perfect for returning a list of the same units.

Some of the pros are:

- You avoid tight coupling between the creator and the concrete products.
- Single Responsibility Principle. You can move the product creation code into one place in the program, making the code easier to support.
- Open/Closed Principle. You can introduce new types of products into the program without breaking existing client code.

The dialog box that uses the Factory looks like this;

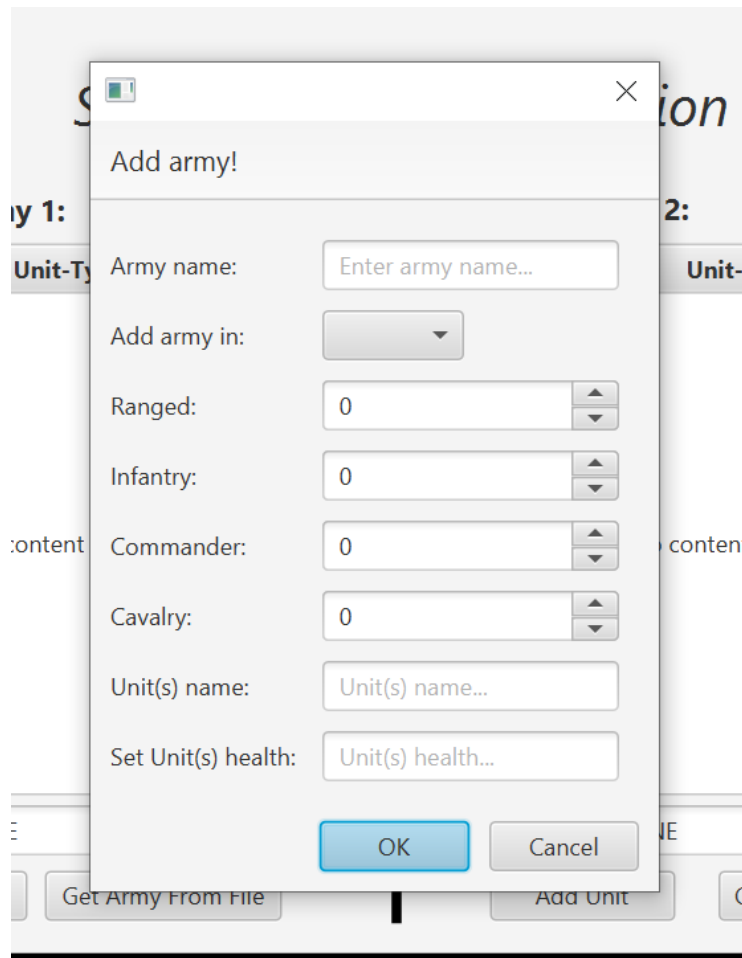


Figure 7: "Add Army Dialog" needs to create multiple instances of unit.

3.5 Template Method

Template Method [6] is a behavioral design pattern at fist glance looks like the inheritance. And it is! The key difference is that there is only one part of the code that is changed. When a superclass and a subclass has so much in common, the superclass becomes more like a template.

In our example we need to add pop-ups that all:

- Has the same Icon in the top left corner.
- Uses a Grid Pane to orginizes nodes.
- Uses checkboxes and/or textfields.
- Has nodes that need to be configured.

The abstarct class, *WarGamesDialog*, works as a template for more complex dialogues. The only abstract method is the **createContent()** method that sets the dialog main content:

```
1 public abstract class WarGamesDialog<T> extends Dialog<T> {
2
3     /**
4      * Abstract class to add content to the dialog.
5      */
6     protected abstract void createContent ();
7
8
9     //Deprecated code for report ....
10 }
```

It takes a generic type. And the subclass can therefor choose what type to return. See the class structure her:

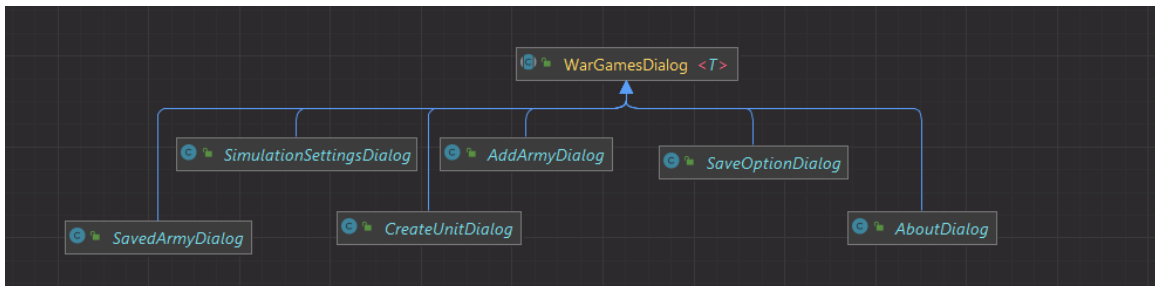


Figure 8: Template Design Pattern: Still uses inheritance

4 Graphic Design & Usability

This section will cover the visual element in the application. Then at the end see how the design looks compared to the design principles by Don Norman.

4.1 Wireframe

First I planed a starting page. Nothing special would be here, just a start button. This gives the option to make the user set a format before starting. Also a second idea would be to set the terrain here. However, it seemed unnecessary no go back to the main page just to change the terrain;

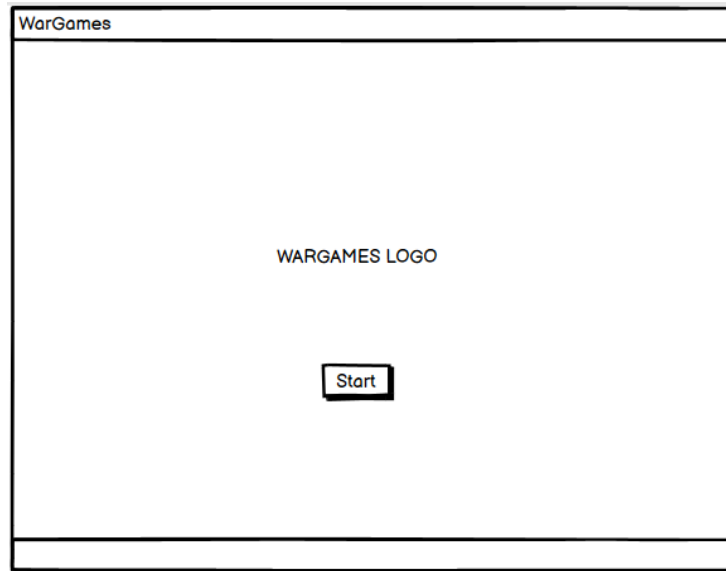


Figure 9: Start Page

The second page would be the setup page. There would be two different tables; one for each army. Each table would have the list of the units. The user would also be able to edit the army name. At this stage, I was unsure of what buttons to have. I ended up with more buttons. They could be structured differently, but the end result looked almost the same as the end result;

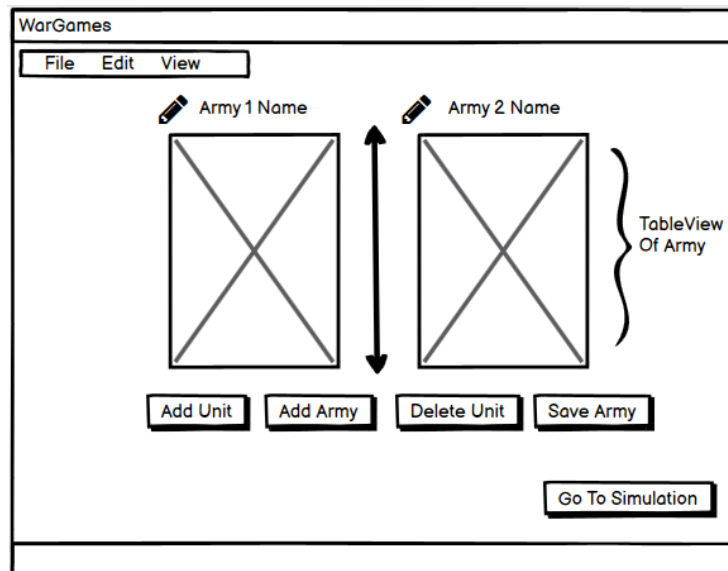


Figure 10: Set Up Page

The Simulation page was tricky. When I designed the simulation page, I wanted all information to be on that single page. This was hard. The battle field itself was too small and the graphs lacked information when minimized (bigger graphs in JavaFx have labels etc). Instead I made permanent tabs on the left side. The user can navigate to the relevant information. This was the first draft of the simulation page;

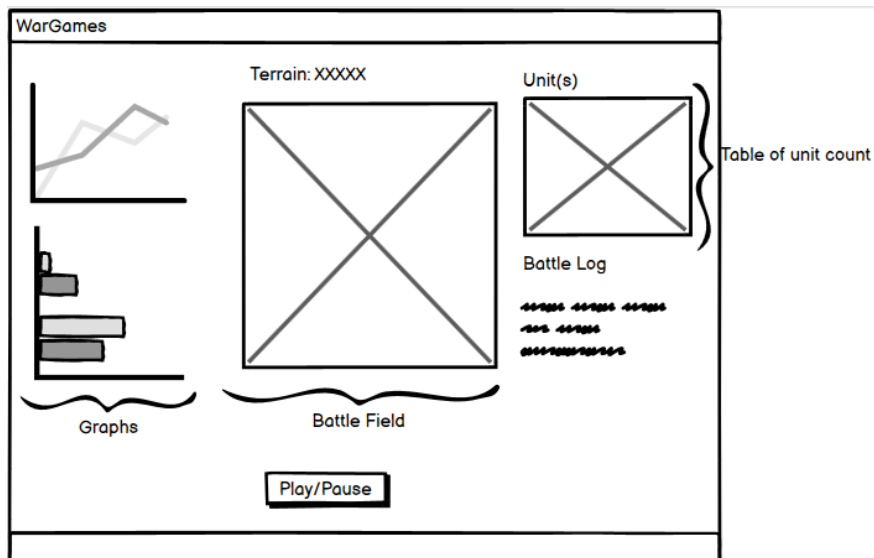


Figure 11: Simulation Page

4.2 Pixel art

4.2.1 Tools

All pixel art has been designed using Aseprite [1]. Aseprite is a tool that is used to design animation and pixel art. I could also use a free tool to design the Pixel Art. However, I am somewhat familiar with the software.

I used Pinterest [13] to look for inspiration. Pinterest is a website where creators of any art/design can post their design. It is a great way to get ideas. I got tips on designing rocks, grass, trees and characters for the 2D display.

4.2.2 Result



Figure 12: Ranged unit from the blue army facing left

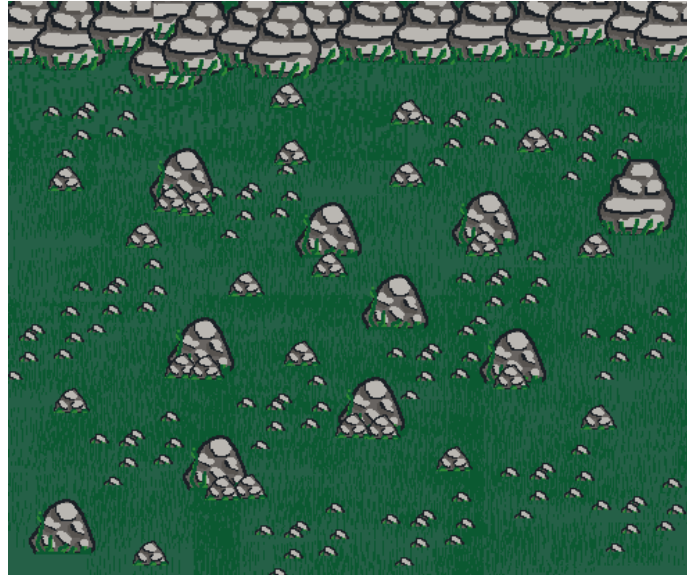


Figure 13: World map: Hill.

4.3 Don Norman's Design Principles

Don Norman [2] was one of the most important user design researches in the computer science department. Here is how the software is compared to the design principals;

4.3.1 Visibility

- Clear contrast between buttons and background. (Dark Blue and white)
- The menu bar is available so the user can easily find any help
- On simulation page, are information divided into tabs, to make the information not clutter.
- Setup page with focus on symmetry to make the page look less messy.

4.3.2 Feedback

- Dialog is opened after each button click (or something happens to the screen).
- Tooltip is added (Shows when hovering over buttons on setup page).
- Icon next to file path is changed based on correct or wrong file.
- Buttons makes the cursor to a hand (to indicate they are clickable).

4.3.3 Affordance

- Icons added make the intention of the buttons more clear.
- Army name is the same color as unit type.

4.3.4 Mapping

- Army is split by a line to indicate to different tables.
- Line between buttons that are only for one army, and for both. F.ex both have a save button, but they have a create unit button.

4.3.5 Constraints

- Buttons are disabled to constrain options.
- Hide options in menu bar.
- Add Dialog to make the options on setup page not too overwhelming (F.ex add a unit in its own popup window)

4.3.6 Consistency

- Consistency in design, both color palette for the software and dividing army the same way (Always army 1 on the left and army 2 on the right)

5 Technical Details

5.1 Packages/Dependencies

For the full list of dependencies, see POM.XML file. The key dependencies are:

- JavaFx Controls
- JavaFx FXML - For windows
- JUnit - For test code

```
<dependencies>
  <dependency>
    <groupId>org.openjfx</groupId>
    <artifactId>javafx-controls</artifactId>
    <version>${javafx.version}</version>
  </dependency>
  <dependency>
    <groupId>org.openjfx</groupId>
    <artifactId>javafx-fxml</artifactId>
    <version>${javafx.version}</version>
  </dependency>
  <dependency>
    <groupId>org.junit.jupiter</groupId>
    <artifactId>junit-jupiter</artifactId>
    <version>${junit-jupiter.version}</version>
    <scope>test</scope>
  </dependency>
</dependencies>
```

Figure 14: Dependencies found in POM.XML file. See source code

5.2 FXML

All pages have been created using *.fxml* files. These types of files has some similarities with *HTML* files. There are some options when it comes to how to create these files. A better way is to use scene builder [14]. Scene Builder lets you create application views in a GUI. You can drag and drop nodes to a pane, and generate the layout as FXML files.

All the pages can be found in resources folder:

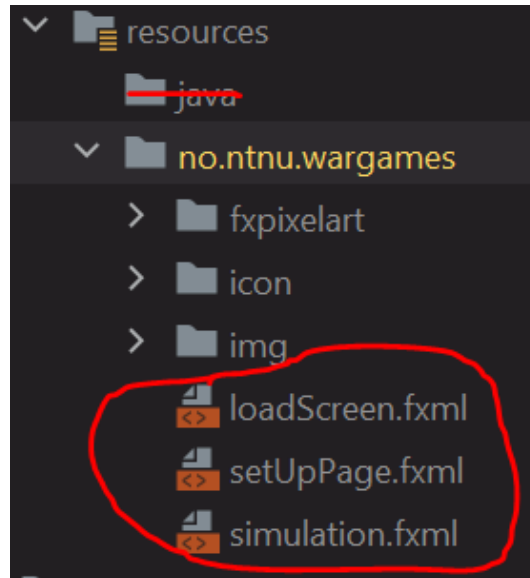


Figure 15: FXML files in Pages

5.3 Extra: Python Script

I used python for generating Army files. The python script is simple, and not a requirement for running the software. It is therefor not included in the source code. However, here is the code is below (Figure 16).

NB! You need have a file of random names, and set the location of this file in *getRandomName()* function.

```
armyFileGenerator.py X
armyFileGenerator.py > ...
1 import random
2 import math
3 from re import A
4
5 """
6 WarGames Script.
7
8 This is a script to create a random unit for the wargames project.
9 (See Github)
10 After running this script, you should be able to get a file with random units.
11
12 #Version-LOG:
13
14 - 0.0.1: returns line of unit information
15 - 0.0.2: generate a new army to a file
16
17 NB! needs a List of random names as file named "randomName.txt"
18
19 @author Kjetil Indrehus
20 @version 0.0.1
21
22 """
23
24
25
26 units = ["Infantry","Commander","Cavalry","Ranged"]
27
28 def getRandomHealth():
29     """Method that return a random integer between 1 and 100"""
30     return random.randint(1,100)
31
32 def getRadomName():
33     nameNumberIndex = random.randint(1,900) #random index in file
34
35     lines = []
36     with(open("dummy-data-generating/randomName.txt","r")) as file:
37         lines = file.readlines()
38
39
40     return lines[nameNumberIndex].rstrip()
41
42 def getRandomUnit():
43     return str(units[random.randint(0,len(units)-1)])
44
45
46 def createNewUnitFile(amount,name,armyname):
47     file = open(name+".csv","w")
48     file.write(armyname+", "+"\\n")
49
50     for x in range(amount):
51         line = getRandomUnit()+","+getRadomName()+","+str(getRandomHealth())+"\\n"
52         file.write(line)
53
54     file.close()
55
56
57 createNewUnitFile(200,"pythonUnit","human")
```

Figure 16: Python Script for generating the army files.

6 Process

6.1 Version Control

This project has been using version control with Git. Each commit has a prefix and some information on what type of commit it is. After the prefix comes the commit message. It include details on what was changed. The message should also clarify why the prefix was used. The prefixes that has been used are;

- Add: Add new code.
- Refactor: Code has been modified.
- Fix: Issues was found and fixed.
- Docs: Documentation for code was added.
- Test: Added Test(s) for code (Test methods and/or classes).
- Feat: A feature was added (See commit message for details).
- Del: Deleted some part of the code that was unnecessary or redundant.

6.2 Management and Scheduling

6.2.1 Deadlines

Type	Deadline
Wargames Part 1	26.Jan 2022
Wargames Part 2	02.Mar 2022
Wargames Part 3	06.Apr 2022
Wargames FINAL	24.May 2022

6.2.2 Managing Task(s)

I decided to use Notion [11] to manage all the tasks that had to be done for each deadline. The main reason why is because I don't need to work with anyone. So a small structured to-do list works better for managing tasks. Below are the different task lists;

WarGames Part 1:

- ✓ Maven and Git Setup
- ✓ Classes:
 - ✓ Unit
 - ✓ Infantry
 - ✓ Cavalry
 - ✓ Ranged
 - ✓ Commander
 - ✓ Army
 - ✓ Battle
- ✓ Text-Based Client
 - ✓ Single Battle
 - ✓ Multiple Battles
- ✓ Add Tests

Figure 17: The To-do list for the first deadline

WarGames Part 2:

- Army Methods:
 - GetInfantry
 - GetCavalry
 - GetRanged
 - GetCommander
 - Add new Tests
- File
 - Save Army
 - Load Army
 - Add Tests (DO LATER)
- GUI Wireframe

Figure 18: The To-do list for the second deadline

-
- Wargames Part 3**
- UnitFactory
 - Terrain
 - Set Terrain
 - GUI
 - + Windows:
 - + Intro Window (Splash Screen (?))
 - Set Up Window
 - Simulation Window
 - Pop-up
 - Error
 - Add Army
 - Save army
 - Information
 - Choose Terrain and delay time
 - Icons for buttons
 - Info Buttons
 - Tests (?)

Figure 19: The To-do list for the last deadline

7 Reflection

Not all solutions are great. They may have better solutions. However, in this next section will there be some reflection on the code and what some key problems where.

7.1 Problem: How to show the simulation?

What does a user want from the simulation? Is it the data? Or maybe purely visuals (to see units move around)?

A battle contains of armies moving around and units being wounded until they are dead. This does not happen instantly. Therefor it makes sense that you see how the war develops over time.

There is probably a better solution for this, but I decided to use Timeline [10]. This is a class that is used in JavaFx application. The class holds KeyFrames [7]. A single keyframe "records" the state of the GUI and saves it. Each keyframe can be added after a function is called. The animation is then stored in the timeline. This timeline can be paused and played. The user can use this during the simulation to pause, and look at the data.

The timeline itself has also a cycle count. In this case, the cycle count is INDEFINITE. This can lead to memory leak (timeline keeps adding keyframes and the animation never stops properly). However, the timeline stops when either one of the armies don't have any units. Also the timeline is reset, when the user refresh the animation.

Here is the code for the timeline:

```
1  /* Button Event */
2
3  @FXML
4  public void onSimulate() {
5
6      setupGraphsBeforeSim(); // Method that sets up the graph
7      buttonPausePlay.setDisable(false);
8      buttonStart.setDisable(true);
9
10     /*Set the timeline that uses given delay*/
11     timeline = new Timeline(new
12         KeyFrame(Duration.millis(getDelay()), this::simStep));
13     timeline.setCycleCount(Animation.INDEFINITE);
14     timeline.play();
15     isPlaying = true;
16 }
```

As seen on line 11, the KeyFrame [7] is added after the method **simStep()** is called. This method carries out the following tasks:

- Check if both army has units, else stops the timeline and announce a winner.
- Chooses an attacking army and attacks the other army.
- Updates graphs with the new state of each army.
- Adds the attack (and any relevant detail) to the battle log.
- Sets new positions for all of the units in each army.

7.2 Problem: How to position the Units?

I have chosen to have a 2D display of the battle. This is not a requirement, nevertheless it illustrates the state of each army in a more visual way. It also given an idea of who is winning.

The key problem is; How are you going to position each unit? My solution is to give each unit a random position. This is done in a for loop for each unit in each army. The "position" method gives a random position. This could lead to a unit been painted over (there is a chance for all units not to be displayed).

The delay time between each battle is recommended to be between 100-200 ms. The result of each attack will be shown in less of a second. So why add a lot of code to display a unique position for each unit?

I also decided to make the units **not** attack each other based on position. This means that two units that appear far away from each other, can still attack each other. I do not see a reason why position also relevant. It only makes the code 10x more complicated. A lot of positioning problems occur.

We can calculate the chance that a single unit is overwrites one of 50 other units.

We know that each unit (see figure 12), is 16x16 pixels and the canvas is 416x496 pixels. This means that there are a total of:

$$\begin{aligned} \text{Width} &= \frac{496}{16} = 31 \\ \text{Height} &= \frac{416}{16} = 26 \end{aligned}$$

$$\text{Tiles} = \text{Width} \cdot \text{Height} = 31 \cdot 26 = \underline{\underline{806}} \quad (1)$$

There is a total of 806 tiles.

Given that 50 units are already drawn on unique tiles and we want to draw a new unit:

$$\text{C: Chance of painting over a unit} \quad (2)$$

$$\begin{aligned} C &= \frac{\text{Amount of units on canvas}}{\text{Total amount of tiles}} \\ &= \frac{50}{806} \\ &= 0,0620347394540943 \end{aligned}$$

$$\underline{\underline{\text{There is a 6,2\% chance of painting over a unit.}}} \quad (3)$$

Even though we assumed that 50 units was already painted was the chance to paint over some units very low. This is the main reason why I did not implement position to the 2D display.

7.3 Problem: Terrain Bonus

In the assignment deception of part 3 we need to implement Terrain (see the terrain task on figure 19). Each unit type could get a bonus for either attack or defence, based on the terrain of the battle field.

For example, a ranged unit would not be as strong in a forest. It would struggle to attack, because a ranged weapon is most likely to hit a tree. The unit would struggle to defend against infantry units. However, a ranged unit would be dominant in an open field.

So we have the problem. We need to give each unit attack and defence bonus based on terrain. There are two main options;

1. Create two methods for attack and defence bonus of terrain.
2. Create one method that returns attack and defence bonus of terrain.

I chose to use the Pair [8] class for returning two values; a key and value. To access the attack bonus, we use `getKey()` method, and for defence we use `getValue()` (both get methods are found in the Pair documentation [8]). The abstract method looks like this;

```
201     */
202     public abstract Pair<Integer,Integer> getTerrainBonusAttackDefence();
203
```

Figure 20: Screenshot of the abstract method in the Unit class.

This could have been done differently. A more clean solution would be two methods. However, then there would be two methods to override in each subclass. Also the Pair [8] class works well.

With many other programming languages, you can return two values. This is not possible with Java (it is not native function). The Pair class seems like a good practice for returning two values in Java.

7.4 Maven and IntelliJ Issues

I had some problems with Maven. Here are some of the issues.

7.4.1 No Tests Found

When I try to run all test using Maven Surefire Plugin, I get an error. The Plugin does not find all the tests. I have tried to;

- Compiling and Clean before using the plugin.
- Using older version of the the plugin.
- Deleting Module-Info.java. (This leads to forked VM issue, see figure 23)

There also seems to be something wrong with module info.

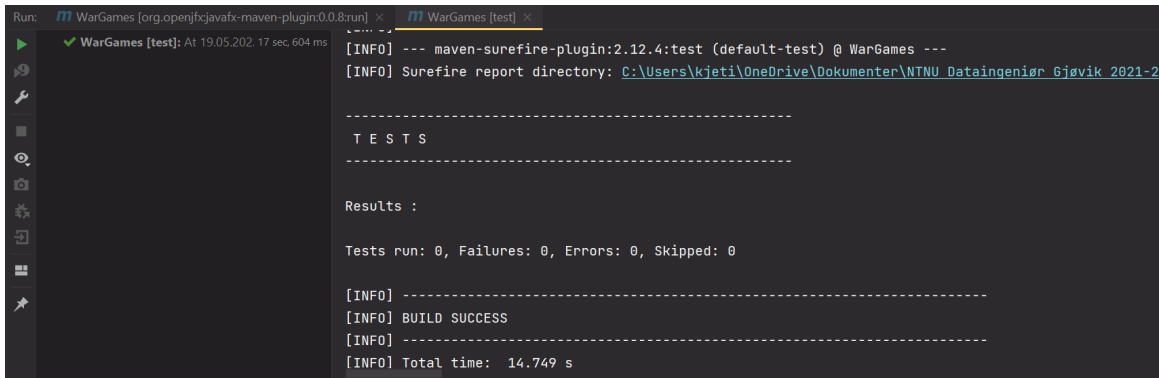


Figure 21: Screenshot of the results.

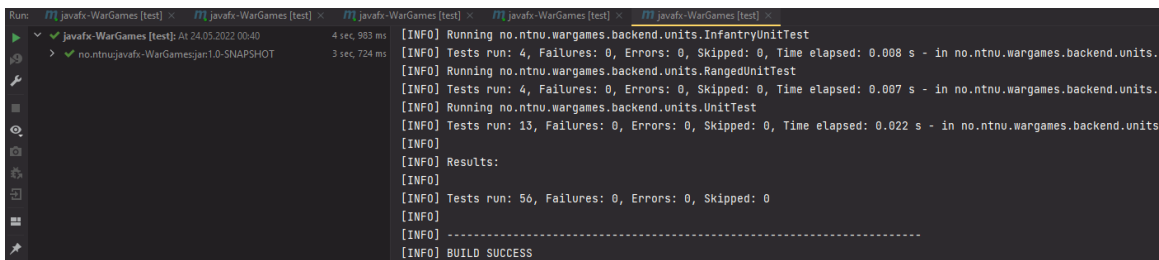


Figure 22: Runs test with errors: No mudule name @7997dhf found.

7.4.2 Forked VM

Sometimes I get a forked VM issue. A reason to this could be Windows (a operating system error). Meaning for a Linux user, this might not be an issue.

Nb! This error only happens if I use the Maven plugin to run tests.

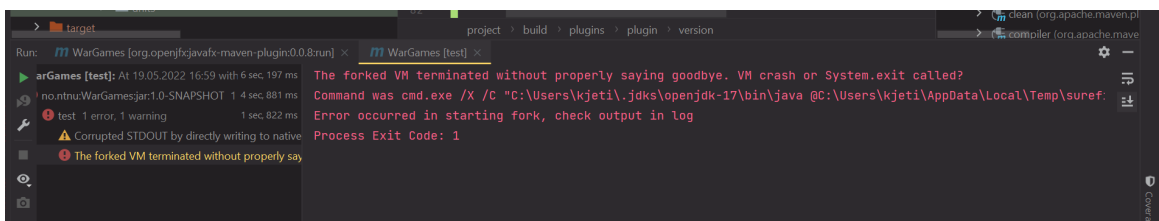


Figure 23: Forked VM issue.

7.4.3 IntelliJ Issue

I also sometimes had a weird IntelliJ issue that fixes itself. It happens when I run test with JUnit. However if I try again, it works. This is the message;

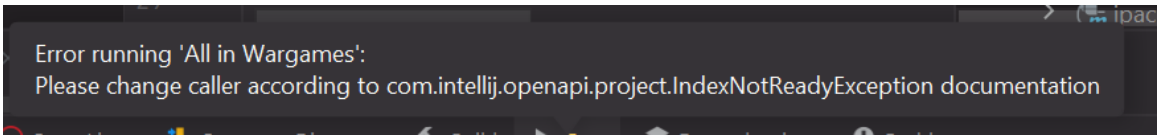


Figure 24: IntelliJ Error: Index error.

7.4.4 Maven Issue: Create Jar

I was not able to create Jar with either jpackage or regular jar plugin. This was the output.

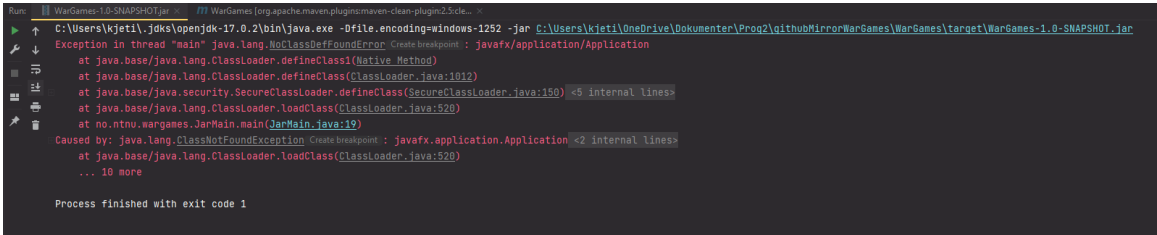


Figure 25: Compiling using Jar Plugin

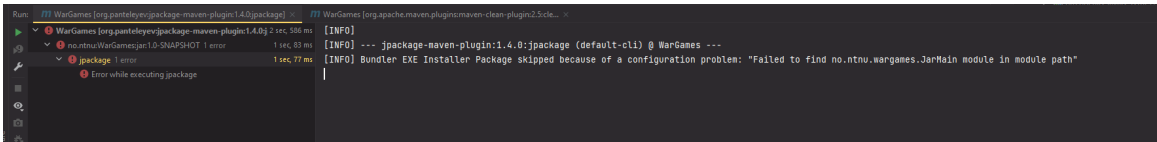


Figure 26: Using JPackage

7.4.5 Solution

All problems mentioned above is a error related to Maven. No solution has been found. Hopefully it is only a OS Issue. I have discussed this problem with Ivar Farup. So instead of running the test with maven, use JUnit! It works well. I don't feel like this ruins the software, since the app works (Also more concepts like git and inheritance are more important than Maven).

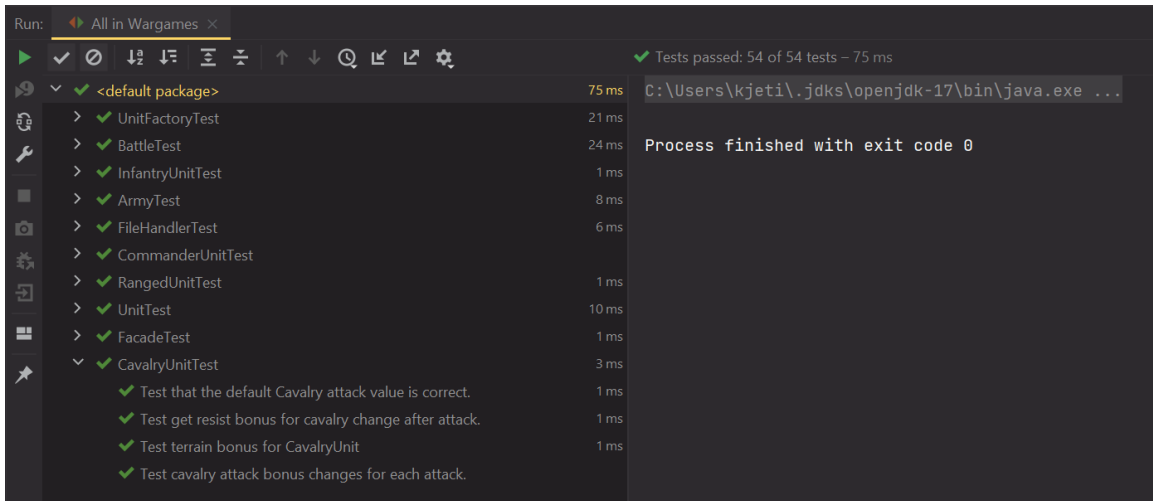


Figure 27: All test passed using JUnit

Also using javafx plugin, you are able to compile and run the project:

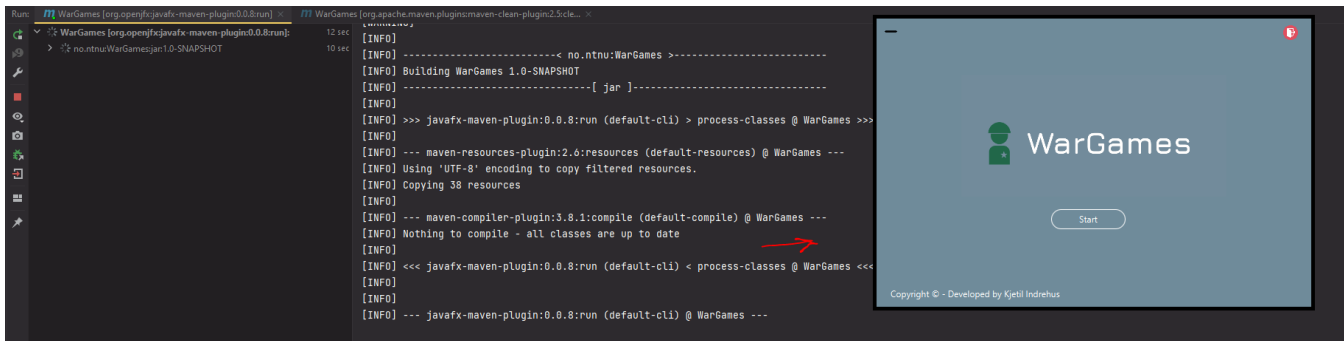


Figure 28: Able to run the application with javafx.

7.5 Implement an Observer?

A well known design pattern is called Observer [4]. The key feature is that it allows the user to make a subscription mechanism to notify object that change has happened.

This looks perfect for the simulation controller. By making all the different type of information be subscription based. If there is a change in the unit count, we only update all information that has anything to do with the unit count. This sounds like a solution that will improve memory of the application.

I did not implement this design pattern. After each attack, all information needs to be updated:

- **Graphs:** If no data change on the Y-axis, it is still relevant to show that the Y-axis data stays the same. For example if the unit count is the same after an attack, we show the user that the units have survived the attack. If we only updated the graph when the Y-axis data has changed, then the battle seems shorter than it is.
- **Attacklog:** Updated all the time (An attack is always called)
- **Unit Count:** The only viable data relevant for observer. However, this is too little information to make it worth implementing the unit count simulation.
- **Battle Field:** Unit position is updated after each attack.

As explained above, the observer would ask almost all subscribers to update. Therefore I see no reason to implement an observer here. (There might be a way to implement an observer for something else, but the easiest solution is often the best).

8 Conclusion

We were asked to simulate a battle between two armies. The user was able create a army. We saw that since the user could create a army in a billion different ways, we might want to know how the army did in a battle. Relevant information like health, unit count and who attack who, needed to be displayed to the user.

The one thing that did not work as well was Maven. I had trouble running the test through Maven. Neither any help on stackoverflow or by co-students and lecturer, was able to fix the POM file. However, javafx and test with JUnit still works.

To conclude; the software achieves all of the requirements (with the exception of Maven build) and is able to display relevant information in a structured way.

List of Figures

1	Setup Page	3
2	Simulation page	4
3	Class diagram of backend	5
4	Unit Class Diagram	6
5	A facade hides other information in a single class.	7
6	Random Instance is used by other classes.	8
7	"Add Army Dialog" needs to create multiple instances of unit.	9
8	Template Design Pattern: Still uses inheritance	10
9	Start Page	11
10	Set Up Page	12
11	Simulation Page	12
12	Ranged unit from the blue army facing left	13
13	World map: Hill.	13
14	Dependencies found in POM.XML file. See source code	15
15	FXML files in Pages	16
16	Python Script for generating the army files.	17
17	The To-do list for the first deadline	19
18	The To-do list for the second deadline	20
19	The To-do list for the last deadline	21
20	Screenshot of the abstract method in the Unit class.	24
21	Screenshot of the results.	25
22	Runs test with errors: No mudule name @7997dhf found.	25
23	Forked VM issue.	25
24	Intellij Error: Index error.	26
25	Compiling using Jar Plugin	26
26	Using JPackage	26
27	All test passed using JUnit	27
28	Able to run the application with javafx.	27

References

- [1] *Aseprite*. URL: <https://www.aseprite.org/>.
- [2] educative. *What are Norman's design principles?* URL: <https://www.educative.io/edpresso/what-are-normans-design-principles>. (accessed: 20.04.2022).
- [3] Refactoring Guru. *Facade*. URL: <https://refactoring.guru/design-patterns/facade>. (accessed: 12.05.2022).
- [4] Refactoring Guru. *Observer*. URL: <https://refactoring.guru/design-patterns/observer>. (accessed: 20.05.2022).
- [5] Refactoring Guru. *Singleton*. URL: <https://refactoring.guru/design-patterns/singleton>. (accessed: 12.05.2022).
- [6] Refactoring Guru. *Template Method*. URL: <https://refactoring.guru/design-patterns/template-method>. (accessed: 20.05.2022).
- [7] Oracle JavaDoc. *Class KeyFrame*. URL: <https://docs.oracle.com/javase/8/javafx/api/javafx/animation/KeyFrame.html>. (accessed: 05.04.2022).
- [8] Oracle JavaDoc. *Class Pair*. URL: <https://docs.oracle.com/javase/10/docs/api/javafx/util/Pair.html>. (accessed: 05.04.2022).
- [9] Oracle JavaDoc. *Class Random*. URL: <https://docs.oracle.com/javase/8/docs/api/java/util/Random.html>. (accessed: 05.04.2022).
- [10] Oracle JavaDoc. *Class Timeline*. URL: <https://docs.oracle.com/javase/8/javafx/api/javafx/animation/Timeline.html>. (accessed: 05.04.2022).
- [11] *Notion*. URL: <https://www.notion.so/product>.
- [12] Visual Paradigm. *What is a Use Case Diagram?* URL: <https://www.visual-paradigm.com/guide/uml-unified-modeling-language/what-is-use-case-diagram/>. (accessed: 12.05.2022).
- [13] *Pinterest*. URL: <https://no.pinterest.com/>.
- [14] *Scene Builder*. URL: <https://gluonhq.com/products/scene-builder/>.
- [15] Wikipedia. *Pseudorandom number generator*. URL: https://en.wikipedia.org/wiki/Pseudorandom_number_generator. (accessed: 12.05.2022).