NTNU
Norwegian University of
Science and Technology

Department of Computer Science

IDATG2001
Programmering 2

Project Report

By:
Ole Kristian Elnæs

May, 2022

Table of Contents

1 Introduction

This project report is in regard to the mandatory project assignment as part of the evaluation of the computer science course IDATG2001. The purpose of this project is to put what we've learnt throughout this course to the test; such as inheritance, design patterns, design principles, unit testing and so on. The program is called "WarGames" and allows the user to create armies, edit armies, import from and save armies to files, as well as simulate a battle between two armies in real-time.

2 Requirements Specification

In WarGames, the user can create an army, edit an army, load an army from file, and simulate a battle between two armies in real-time. Underneath is a simplified use-case diagram.

- Create new army
  - Import units into new army from other files.
  - View units currently in army.
  - View general statistics over the army being created.
- New War
  - Select two armies from the army register to fight each other.
  - Select speed of the simulation.
  - Select the terrain in which the battle will take place.
  - Set the name of the battle.
- Add Army From File
  - Opens up a filechooser to select an army file from.
- View Armies
  - Get an overview of the armies, and their total number of units, and the number of commander units, ranged units, infantry units, and cavalry units, and the total health of the units in the army (including the percentage of current health in relation to the original health), and lastly, the path to the file on which the army is saved.
  - Edit a selected army.
  - Replenish/reset an army.
- Start war/simulate war (must be accessed through the "new war" page)
  - Real-time information about the state of the warring armies, including which is currently in the lead, which army is currently attacking and what units are facing each other.
  - Play/pause the simulation.
  - Alter the speed of the simulation.

3 Design

3.1 Design patterns

3.1.1 Singleton

The program utilizes a class that adheres to the singleton design pattern, and is called "Facade.java". Simply put, the purpose of a singleton is to only have one instance of a class that holds instances of other classes, or other fields.

This way, the singleton class can be used to share information across multiple scenes in the graphical user interface.

3.1.2 Factory

The factory design pattern is used to create units in the program. The strength of the factory pattern is to delegate the creation of objects to the factory class instead of accessing the sub-classes of a super-class directly.

3.1.3 Inheritance

Inheritance is a key concept and design choice in object-oriented programming. In this program, inheritance is used to create a super-class with all the common fields and methods that all units have in common, while the sub-classes more clearly defines the abstract methods from the super-class.

3.2 Design Principles and Design Choices

3.2.1 Duplication of code

I have focused on limiting duplication of code as much I can in this project. Duplication of code increases the work needed to alter a program, especially when it grows large; it also decreases the readability of the code. When I noticed that snippets of the same code were used in several places, I created a method for that portion of code, instead.

3.2.2 Coupling and cohesion

Loose coupling between the classes of a program is desirable, and this is done by making sure that classes are, for the most part, independent of each other, and that each class has a clearly defined functionality that it represents. The WarGames program has for the most part low coupling as I have made sure that the classes aren't inherently reliant on each other.

3.2.3 Constants

It is good practice to use constants for Strings such as folder paths. It makes it simpler to alter the paths at a later point, whereas hard-coding them increases the chance of errors in the program as a result of outdated, hard-coded paths.

3.2.4 Handlers

In the program, I have created handler classes for routines or functionalities that may be used by several classes, as a way to increase the delegation of responsibility across the classes and to decrease duplication of code and complexity.

3.3 GUI

3.3.1 Resizability

In my project, I have focused on implementing a user-friendly and aesthetically pleasing solution for resizing the window, such that the content – as much it can – extends and adapts to the window when it is being resized. The figures below illustrate how the GUI of the new war page of the program adapts to the window being resized.

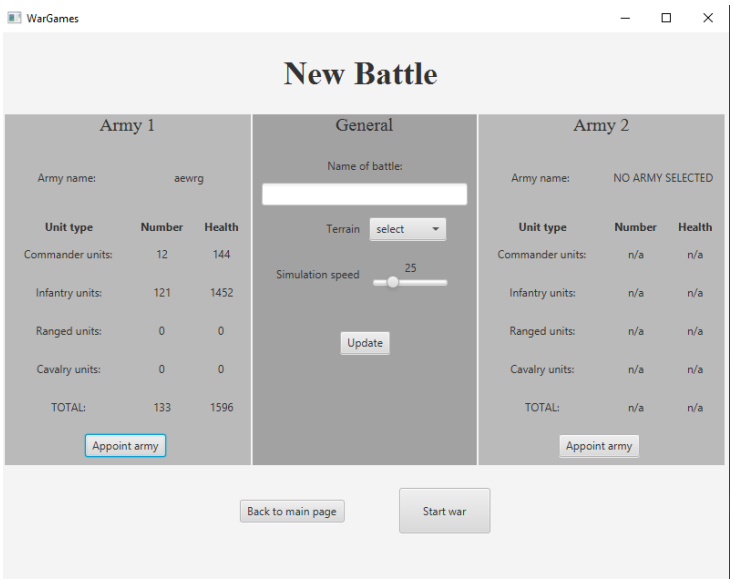

Figure 1: The new war page before resizing.

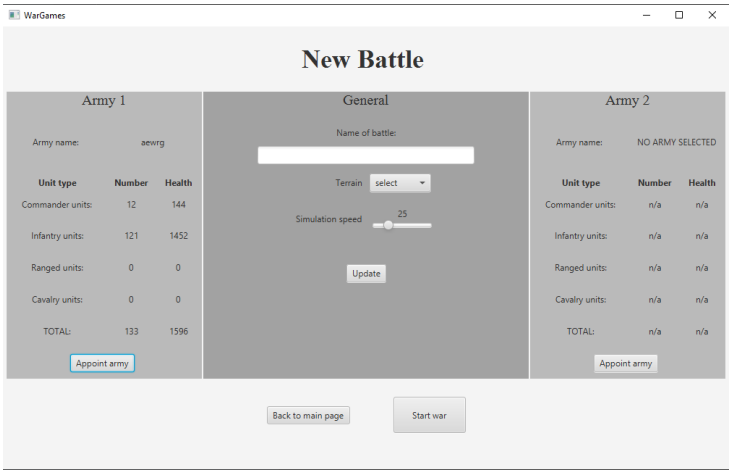

Figure 2: The new war page after resizing horizontally.

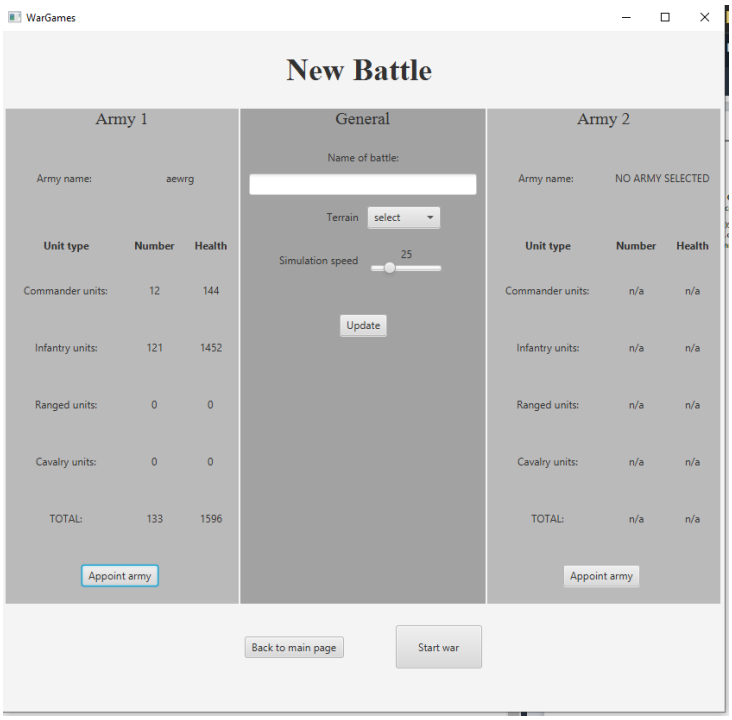

Figure 3: The new war page after resizing vertically.

3.4 Class-diagram

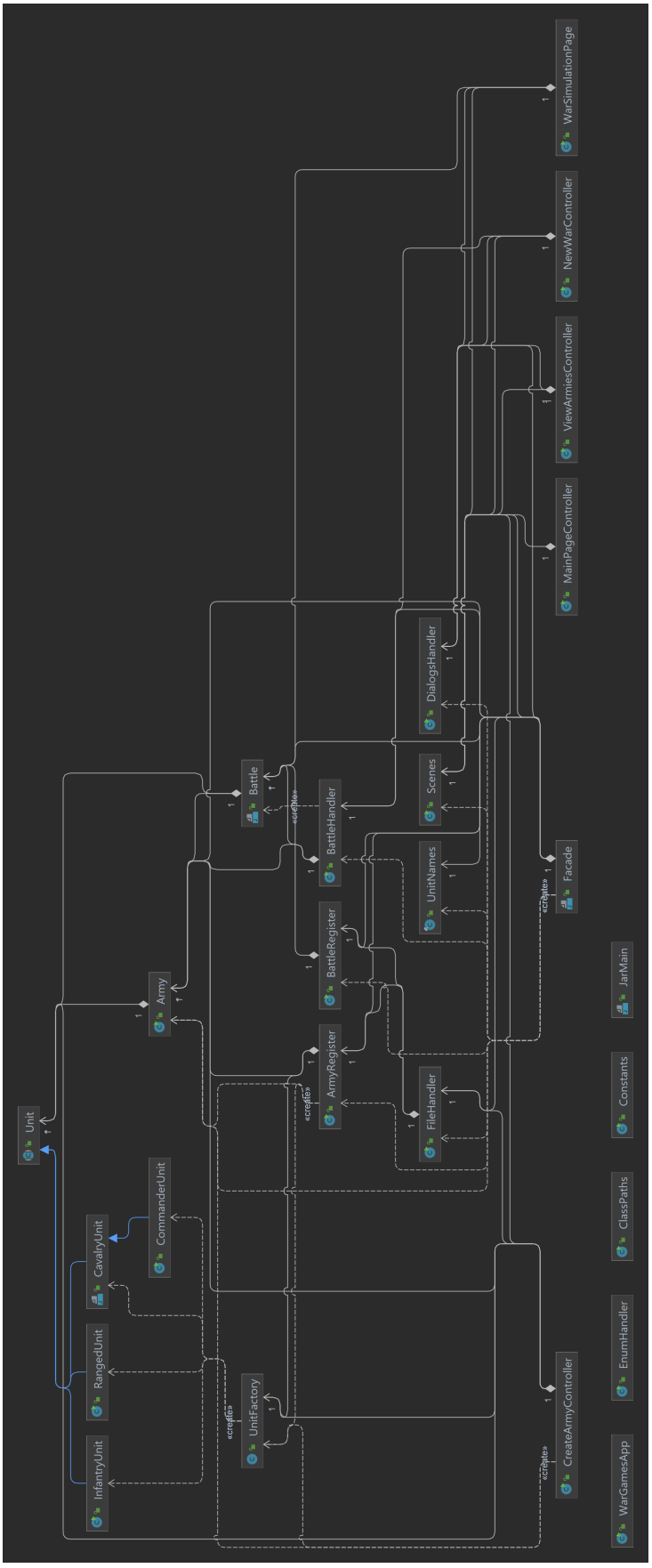

Figure 4: The class diagram.

3.4.1 Multi-threading

In order to simulate a battle between two armies, I needed to implement multi-threading. As JavaFX generally runs on one thread, using Thread.sleep() simply freezes the thread that the GUI runs on and doesn't allow for an animation effect. Instead, one has to use functionality provided by JavaFX in order to implement this.

4 Implementation

4.1 Design patterns

4.1.1 Singleton

The following two figures illustrate how I implemented the singleton design pattern into the Facade class of my program. Figure 5 displays the private constructor of the class. Figure 6 illustrates the getInstance() method for the Facade class that either creates an instance of the class, or returns the instance that already exists

```
/**
 * Private constructor.
 */
private Facade(){
    this.scenes = new Scenes();
    this.armyRegister = new ArmyRegister();
    this.battleRegister = new BattleRegister();
    this.fileHandler = new FileHandler(armyRegister, battleRegister);
    this.army = new Army( name: "Temporary");
    this.dialogs = new DialogsHandler();
    this.battleHandler = new BattleHandler();
    this.state = EnumHandler.State.NEW;
    unitNames = new UnitNames();
}
```

Figure 5: The private constructor.

```java
/**
 * Method that checks whether an instance of the class exists or not,
 * if not, it creates the instance. After this, it returns the instance.
 *
 * @return  The instance of the class.
 */
public static Facade getInstance() {
    if(instance == null) {
        synchronized (Facade.class) {
            instance = new Facade();
        }
    }
    return instance;
}
```

Figure 6: The getInstance() method.

4.1.2 Factory

The following three figures illustrate how I implemented the factory design pattern into my project. Figure 7 is the createUnit() method that creates and returns one unit based on the input given. Figure 8 is the createUnitList() method that creates and returns a unitList consisting of units as defined by the input passed. Figure 9 is the enum consisting of the different types that a unit can be; enum was used instead of String because it leads to less probability of error.

```java
/**
 * Method that creates and returns a unit of a specified type.
 *
 * @param unitType  The type of the unit.
 * @param name      The name of the unit.
 * @param health    The health of the unit.
 * @return          If unitType coincides with an existing unit type, it
 *                  returns a unit of that type, else it returns null.
 */
public Unit createUnit(Type unitType, String name, int health) {
  return switch (unitType) {
    case INFANTRY -> new InfantryUnit(name, health);
    case COMMANDER -> new CommanderUnit(name, health);
    case CAVALRY -> new CavalryUnit(name, health);
    case RANGED -> new RangedUnit(name, health);
  };
}
```

Figure 7: The createUnit() method.

```java
/**
 * Method that returns a list of units in the desired unit type.
 *
 * @param unitType  The type of the unit.
 * @param health    The health of the unit.
 * @param amount    The amount of units in the list to be returned.
 * @return          A list of units in the desired type. Returns null if invalid type.
 */
public List<Unit> createUnitList(Type unitType, int health, int amount) {
  List<Unit> units = new ArrayList<>();
  for (int i = 0; i < amount; i++) {
    String name = Facade.getInstance().getUnitNames().createUnitName();
    //checks whether the input is valid or not
    if (!checkCreateUnitListInput(unitType, name, health, amount))
      return Collections.emptyList();
    units.add(createUnit(unitType, name, health));
  }
  return units;
}
```

Figure 8: The createUnitList() method.

```java
public enum Type {
  COMMANDER,
  INFANTRY,
  CAVALRY,
  RANGED
}
```

Figure 9: The Type enum.

4.1.3 Inheritance

Inheritance was implemented by making Unit an abstract class and making the getType(), getAttackBonus() and getResistBonus() abstract too; and to be fully implemented by the subn-classes.

4.2 Design Principles and Design Choices

4.2.1 Constants

I have implemented the use of constants through making three dedicated (final) classes for this purpose. One class regards the various file paths, where each component of the path is its own constant. And then I have another class for the other constants, such as the version of the program, which has been used in the JavaDoc on the top of each class. I also have a fourth constants class which holds enum objects – this instead of having the enums inside the class where they are used. I also have a fourth, sort-of-constants class, whose only purpose is to provide randomly-generated names for units.

4.3 GUI

4.3.1 Resizability

The main way I have incorporated good resizability in the program is through using borderpane pane-type that javafx offers. The borderpane is good in that it lets you stick content to different parts of the window (left, right, top, bottom, center), and as such, makes the content within the window move along when the window is resized. It is also important to avoid using "preferred width/height" and instead use "minimum/maximum width/height" for the contents within the window, which allows the content to be resized instead of appearing in a fixed size.

4.3.2 Multi-threading

The way multi-threading has been implemented was by creating a Task within the warSimulationPageController which freezes the program for given amount of time (Class Task, 2022). By also using Platform.runLater within the task, the GUI will get updated after the delay is over (Class Platform, 2022). By nesting a while-loop inside the Task (and inside the overridden Call() method) that loops for as long as one of the battling armies has units left, this will provide the appearance of an animation. It was also necessary to use JavaFX's StringProperty and IntegerProperty to assign values with the Platform.runLater methods, and to then bind these values to the labels of the scene. The figure below illustrates this simulation method (note that some code had to be taken out as the method was too long).

```java
/**
 * This method performs a real-time simulation of the war, by creating a background thread through which
 * labels and other content regarding the warring armies are updated in the GUI as the simulation
 * progresses.
 */
private void realtimeSimulation(){
  try {
    Task<Void> task = new Task<>() {
      @Override
      protected Void call() {
        //This is the main loop that runs while it is true that both armies have units left.
        while (battle.getArmyOne().hasUnits() && battle.getArmyTwo().hasUnits() && isSimulationRunning) {
          // Checks whether the boolean pause variable is true.
          if (pause) {
            // While pause is true, it loops a 10 milliseconds delay.
            while (pause) {
              delay( millisec: 10);
            }
            //sets the delay between battles.
          } else delay(battle.getSimulationSpeed());
          //tasks to be done by the GUI in-between delays.
          if (battle.getRounds() > 0) {...}
          //Simulate one round.
          Platform.runLater(battle::simulateOneRound);
        }
        return null;
      }
    };
    //Thread creation and initialization
    Thread th = new Thread(task);
    th.setDaemon(true); //sets the thread as a daemon thread (low-priority/background thread)
    th.start();
    //Binding values as set inside the while-loop and the Task to their appropriate labels/fields.
    totalUnitsLeftA1.textProperty().bind(unitsLeftA1Value.asString());
    //the remaining bindings here...
  } catch (Exception e) {
    e.printStackTrace();
  }
}
```

Figure 10: Real-time simulation method.

4.4 Testing

I have implemented testing for the classes that handle integral backend/background logic, such as the units classes and most of the other classes that are part of the core folder. The fxml controller classes do not have any tests as these are front-end and connected to their respective fxml-files.

4.5 Maven setup

For the maven setup, I based the pom file on the pom-file from the newsstand-project in the IDATG2001 git repository, and changed the necessary variables so that it fit with my project.

5 Process

In terms of work distribution, I have mostly worked in batches. That is, I have worked until the requirements for the delivery were done. For the first iteration, this was close to the deadline, but for the second iteration, this was a couple of weeks before it was due. However, for this final iteration of the project, I have taken to a better practice and distributed it more evenly across these final three weeks of the project.

I have used version control through git heavily, especially during this final iteration. I have 130 commits, of which only 35 were from the first two iterations. In terms of other tools used, I used balsamiq for the wireframe of the GUI.

6 Reflection

Throughout this project, I have learned about how to identify what problems can be solved with conventional design patterns. I have also learned about how to structure big projects, and how to implement helper/handler classes that provide common functionality across classes. But what I have learned the most about, is GUI programming: how to communicate between a controller class and the fxml-file; how to add listeners/observers to JavaFX objects; how to go about structuring the user interface with focus on resizability; and lastly, how to create a Task in JavaFX in order to allow multithreading.

If I were to start anew on this project, I would have focused on planning out the final structure of the program ahead of time, instead of figuring it out as I go. I would also have created tests for new, testable methods immediately, instead of often waiting several hours or even days.

There was some functionality that I was unable to implement due to time restriction. One of these functionalities was having a battle register scene where the user could view ongoing battles (that had been momentarily stopped). As part of this functionality, there was also supposed to be a column in the "view armies" scene where the user could see what battle and army was partaking in (if it was).

Initially, I wanted to implement the terrain function in the fashion of a decorator pattern. However, I found it to be too inconvenient and complex to alter the structure of the program that much. Instead, I implemented it through the use of another abstract method that the sub-classes take care of.

It is arguably the GUI that is the strong point of my solution; while also the use of alerts and the file handling is good. I am not too sure whether the overall structure of the files and the program in general is optimal. It might be that there are better ways to categorize files – for example: having all handler classes in their own folder called "Handlers."

7 Conclusion

Over all, I'm satisfied with the result of this project. I was able to implement all the required functionalities, and most importantly, I learned a lot from doing this project. The part that was the most rewarding was figuring out multithreading in JavaFX in order to allow Thread.sleep() to cause a delay while still not freezing the program.

8 Sources

Class Task. (2022) Class Task. Retrieved from:
https://docs.oracle.com/javafx/2/api/javafx/concurrent/Task.html

Class Platform. (2022) Class Platform. Retrieved from:
https://docs.oracle.com/javafx/2/api/javafx/application/Platform.html#runLater%28java.lang.Runnable%29