War Games Project Report

Trym Hamer Gudvangen

10029

24 May 2022

NTNU IDATT 2001

Table of Contents

Table of Figures

Introduction

The War Games project was provided through NTNU's programming course IDATT2001. In this course, a student learns the fundamentals of inheritance, functional programming, design patterns, version control, and unit testing. The War Games project is a major assignment which challenges students to utilize the knowledge and skills gained throughout the course. As such, the students have worked on this project alongside lectures and independent studies. At the end of the project, students are expected to reflect on the outcome in reference to mastery of the learning goals.

Assignment Description

The assignment undertaken was the creation of an application which simulates a war between two armies. To accommodate the learning stages of the course, the project was broken down into three separate parts. In each part, supplementary features were added, eventually building it into a more complex, realistic simulation of war.

The first part of the project focused on logical aspects through establishing the classes necessary for the simulation to function; this included an army, battle, and unit classes. The second part added sorting features to the army class, as well as created a class for handling army files. When both tasks were done, a rough sketch of a GUI was to be made. The last part of the project entailed a factory class for producing units, an implementation of different terrains and their consequent benefits to units, and an easy-to-use graphical user interface (GUI).

Fulfillment of Requirements

When it came to exception handling, the backend was extensively checked for exceptions. Whenever an error or invalid argument could be sent as a parameter, a suitable exception with a descriptive message was thrown. These reports were then translated to alert boxes in the GUI, helping the user navigate the bounds of the application.

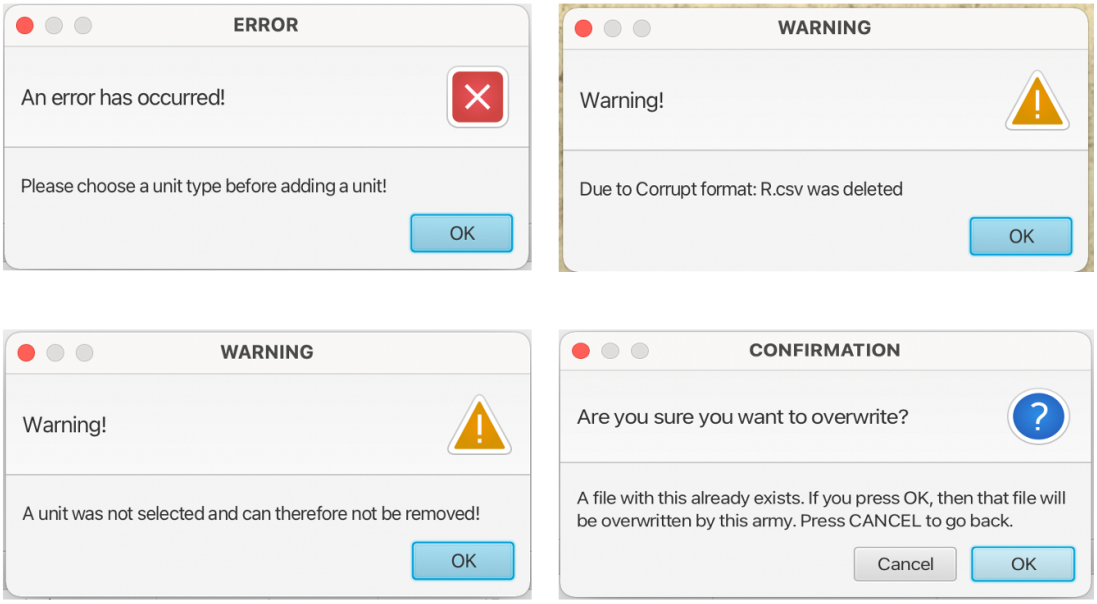

Figure 1. Exception Handling in GUI

Unit tests were used to tracked down places where errors may occur, as well as automized the process of checking for bugs. Through both positive and negative tests of the methods in classes, the chances of exceptions were significantly reduced. This may be seen in the source code; however, the coverage percentage of the main classes provides an idea of how extensively the functionality is tested.

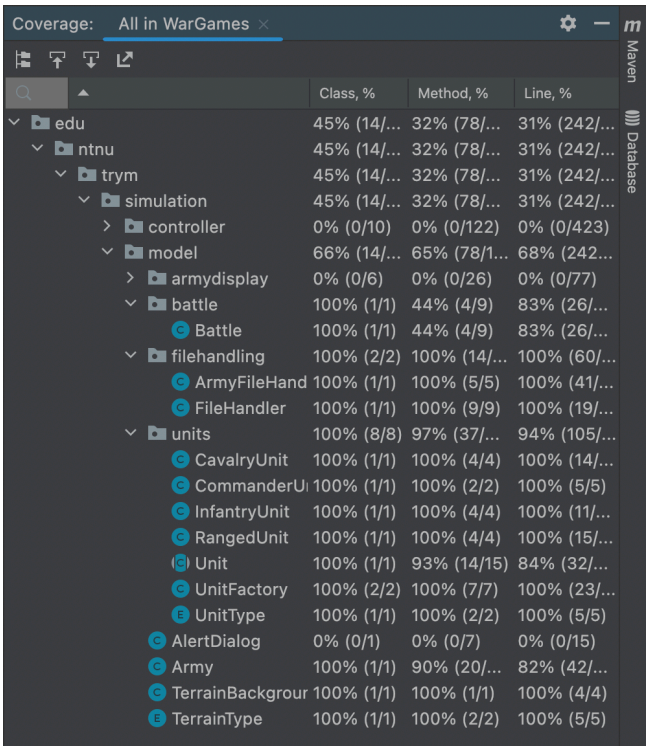

Figure 2. Unit Test Coverage

Moreover, using an enumeration, terrains were added as a feature of a unit's stats. However, it is also a crucial part of the user experience, where the background changes based on the terrain.

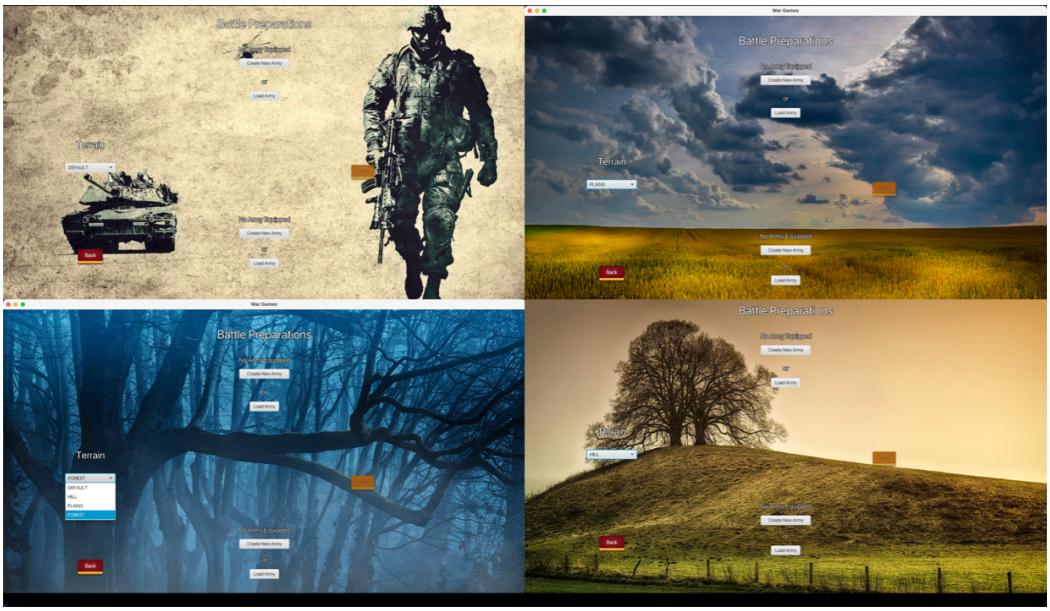

Figure 3. Terrain Backgrounds

With user-friendliness in mind, whenever a user performs an action, such as selecting a saved army, there is an appropriate response (the load button becomes green). Furthermore, at the end of the simulation, the user receives the results of the two armies and can quickly simulate another war, since the same armies are reloaded.

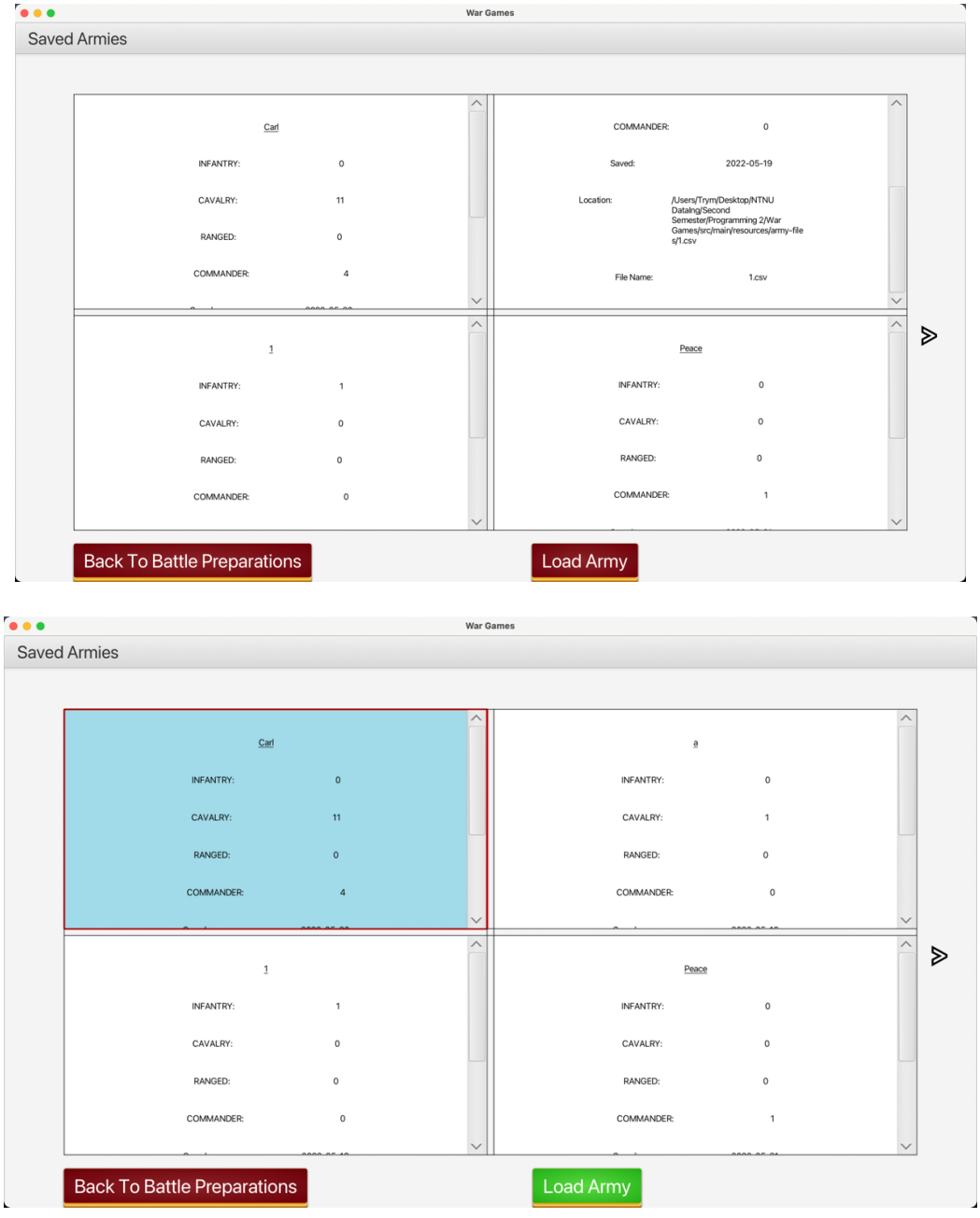

Figure 4. Load Army Screen

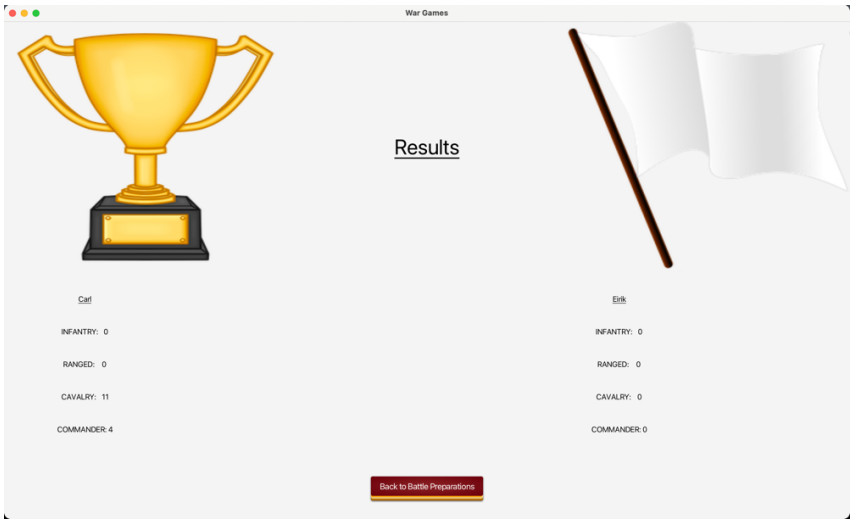

Figure 5. End Results Screen

Beyond the requirements, the program allows for file handling with multiple formats for default and special units. For default units, the user doesn't need to specify the attack or armor value, instead pre-set values are used. However, the user also has the opportunity to customize the unit's attack and armor value, if desired.

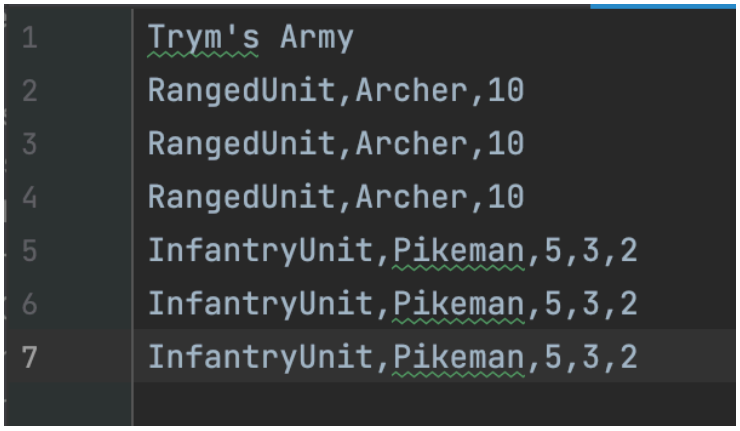

Figure 6. CSV Formatting

The application also provides further functionality with the changing of a saved army. When the user loads an already saved army, a button appears allowing them to change the units within that army and then to overwrite the file again. The application's extensive functionalities are shown below.

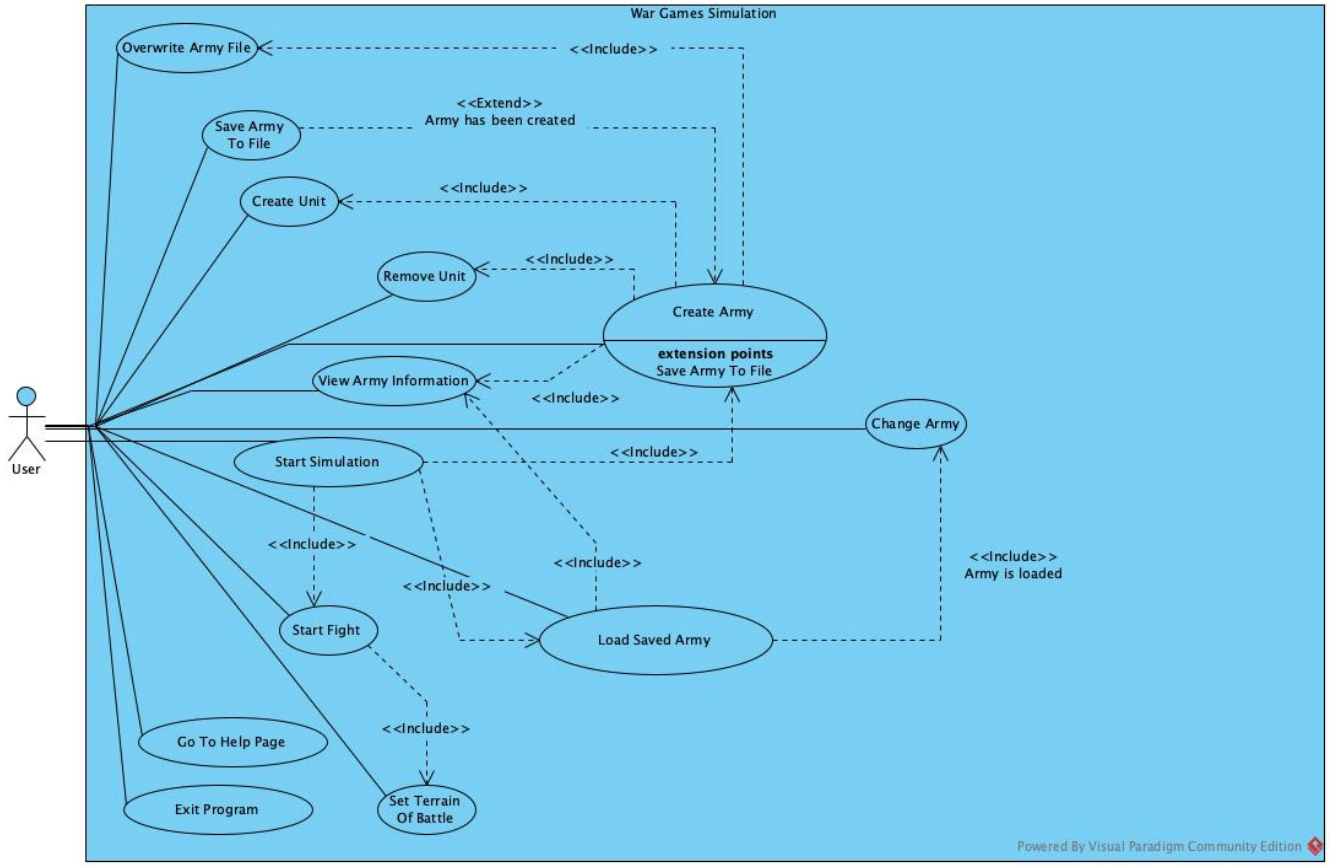

Figure 7. Use Case of War Games Application

Design

Single Responsibility Principle

Single Responsibility Principle (SRP) is a programming paradigm where classes and methods serve one purpose. This concept cannot be understood without the ideas of cohesion and coupling.

High Cohesion

Cohesion looks at the interconnectedness of a class's functionality. High cohesion is desirable as it means the functionality of a class serves a single purpose. In this project, high cohesion was achieved through precise, well-defined methods. Moreover, within a class, functions were reused, and redundant methods removed. All these aspects were complimented with appropriately named variables and methods.

Loose Coupling

While cohesion looks at the intraclass interactions, coupling focuses on the relationship between classes. Throughout this project, there has been a focus on an application with low coupling, allowing for a more adaptable and dynamic program. This can be seen through the use of primitive parameters and reduced direct interaction between classes.

Builder Design Pattern

Similar to the abstract factory design pattern, the builder design pattern aids in the construction of intricate objects with potentially many attributes. This design pattern tackles a problem commonly known as the telescoping constructor anti-pattern. This anti-pattern occurs when an object needs to be instantiated with different amounts of parameters defined. Instead of creating a new constructor for each different variation, the builder allows for a programmer to, as the name suggests, build upon an object by adding the desired variables. Due to the focus on lowering redundant methods, this method also promotes SRP.

Factory Design Pattern

The factory design pattern handles the creation of objects using factory methods. In instances where a subclass of an abstract class needs to be initialized, without pre-cognition of which subclass, a factory design pattern provides a succinct and loosely coupled method for solving this issue.

Front Controller Design Pattern

The front controller design pattern is utilized to create a central class, commonly known as a dispatcher, in order for controllers to navigate the view package. Through this centralized handling class, the code becomes more readable and both looser coupling and high cohesion is promoted.

Singleton Design Pattern

The Singleton Design Pattern is used when a class should only produce one object. Furthermore, the pattern gives rise to a class which represents the functionality behind that one object. This is usually done through the use of a private constructor and a private static instance of the class. One of the dangers with this pattern, however, is that complications with threading may occur.

Enum Singleton

A thread-safe implementation of the singleton design pattern is the enumeration singleton. Instead of using a constructor, this class just uses an INSTANCE enumeration, still with the desired variables attached to it, as seen in the figure below.

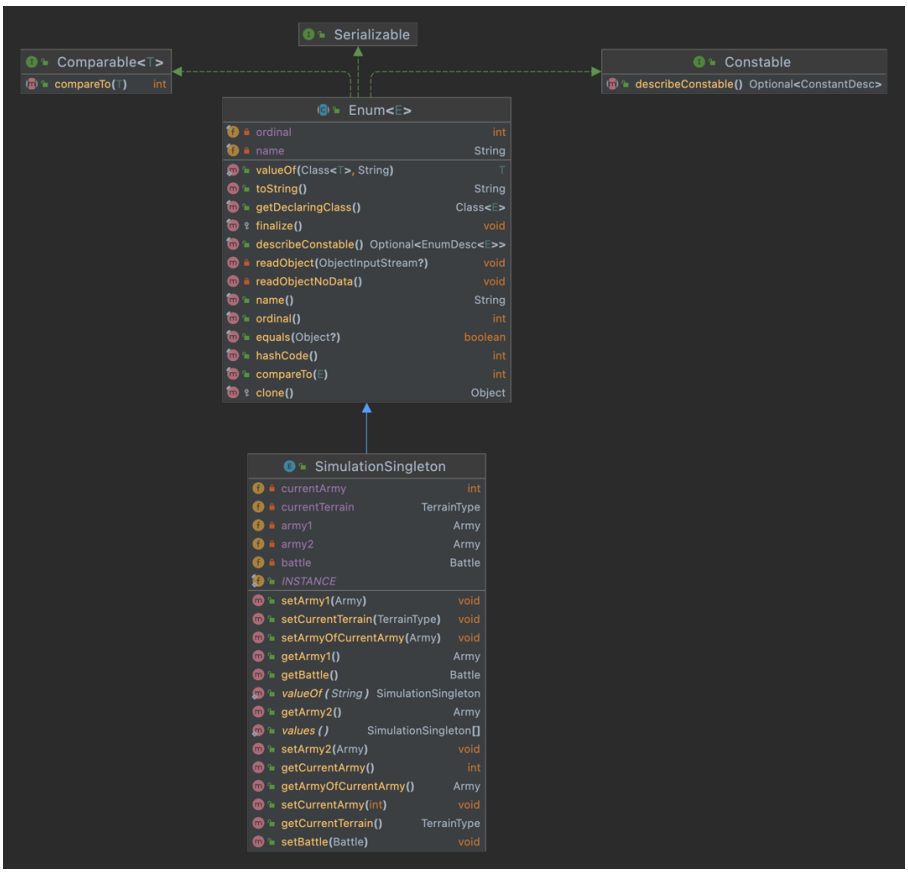

Figure 8. Simulation Enum Singleton

Model-View-Controller Design Pattern

The MVC design pattern is an architectural pattern for organizing a complex system into simple and easy to understand packages: model, view, and controller. The model package contains the logical, backend classes. The view package contains the scenes and other graphical aspects of the application associated with the frontend. The controller acts as the connection between the model and the view. This makes the code more readable as well as aids in designing a low coupling, high cohesion program.

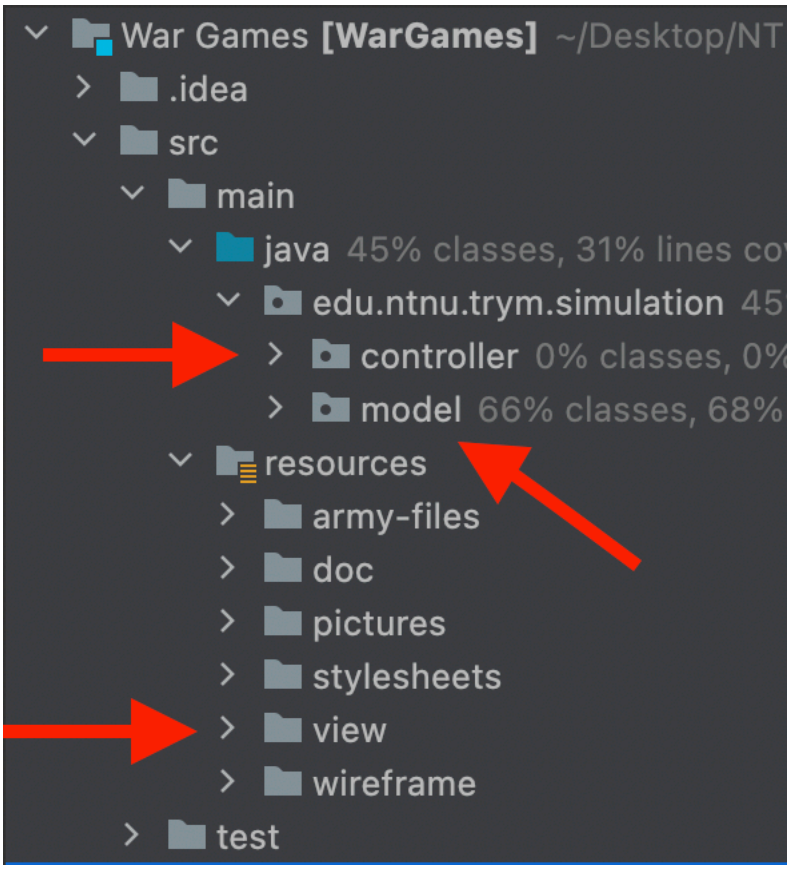

Figure 9. Model-View-Controller Structure

Implementation

War Games Architecture

Through the MVC architectural design pattern, the War Games project was compartmentalized into three main packages, as seen in figure 10. Furthermore, classes with similar functionality were divided into sub-packages for a clearer overview of the project.

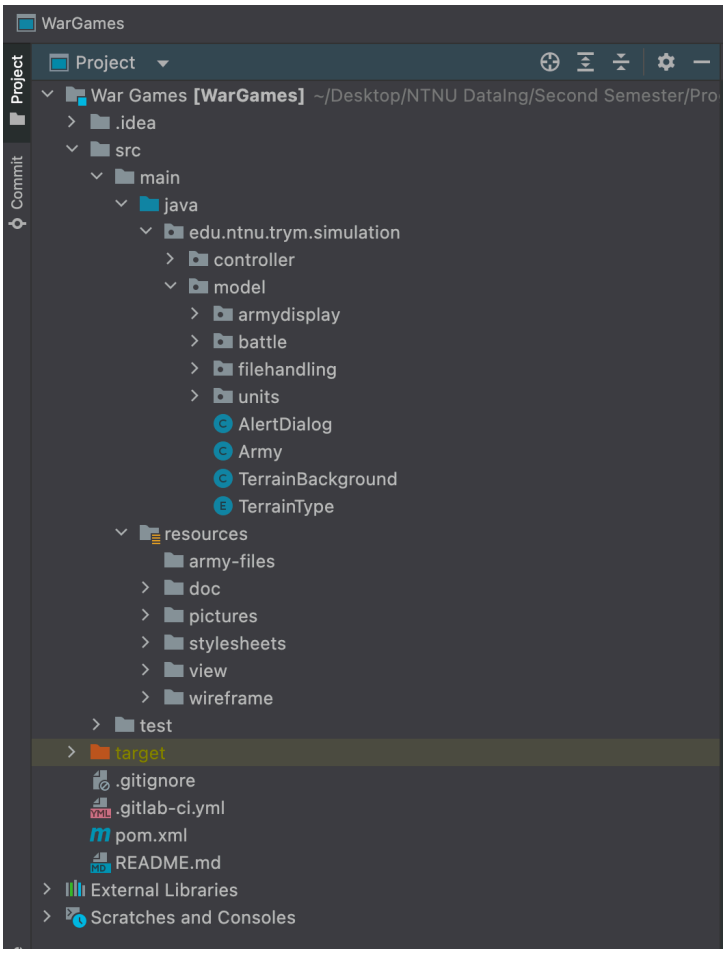

Figure 10. Project Structure

Unit Factory

To mass produce the different units stemming from the abstract class unit, the factory design pattern was applied. A unit factory, as it was called, has a method for getting a single instance of a unit based on the information provided and a method for getting multiple of that instance.

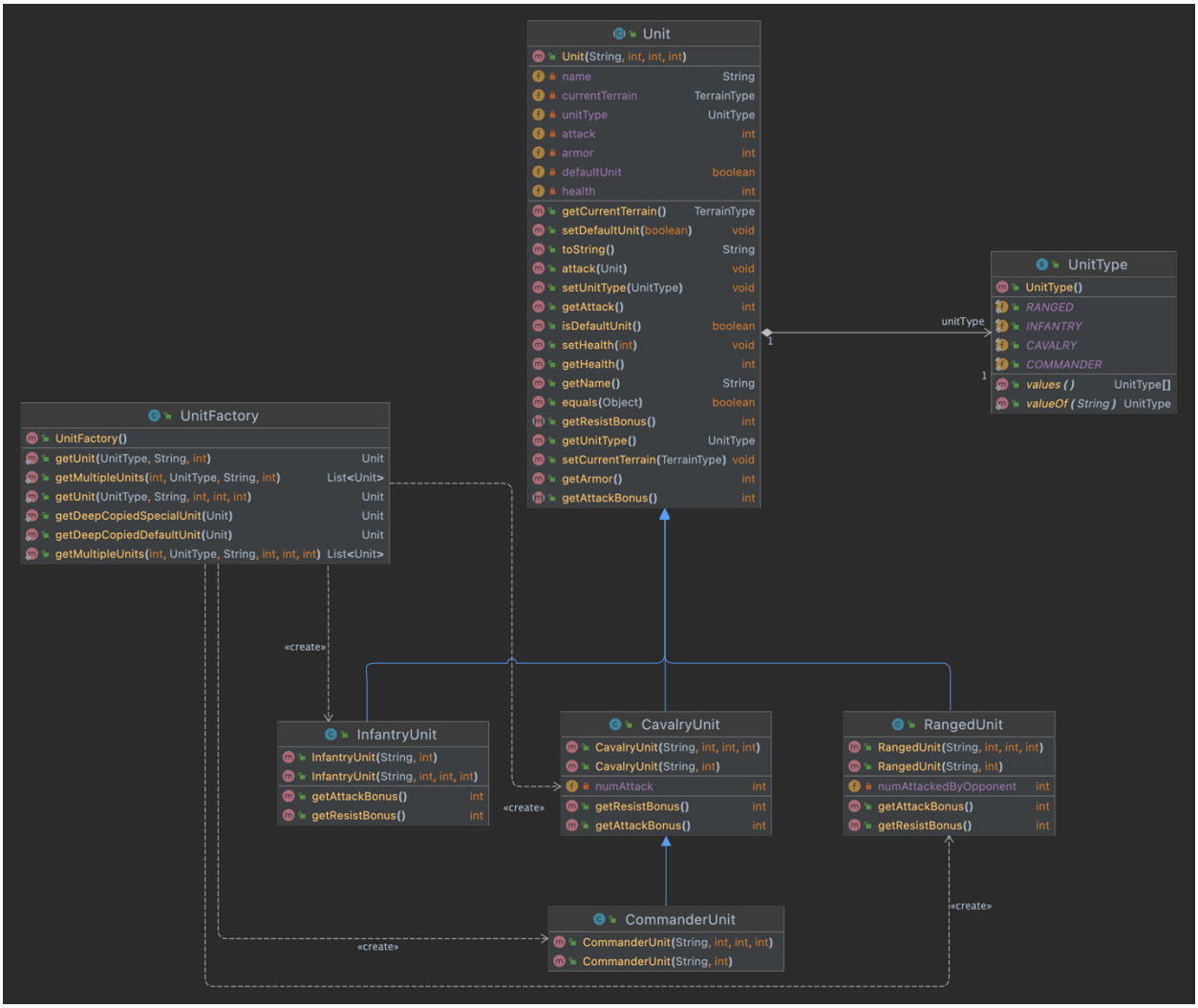

Figure 11. Class Diagram of Units Package

Builder Classes

When it came to the GUI, it was complicated and time-consuming to construct text fields, vboxes and hboxes to display an army's information. The code would also become extremely convoluted with node variables, making the readability of the program plummet. Similarly, if a tableview was to present unit information from an army, then one of the easiest ways was to create the columns using Scene Builder and fill them later. However, through both of these methods, the freedom within the program and overall cohesion would suffer severely.

To counteract these two approaches, the builder pattern was utilized. With the builder class, a table or display could be constructed by choosing what information was desired and attaching it to the builder object. Since a node (TableView or Pane object) was needed once the information was displayed, the classes containing the builders extend the desired node. Through the use of polymorphism, an ArmyTable extending TableView<Unit> and an ArmyDisplay extending Pane can be treated the same as super classes. Therefore, using a static inner-class Builder with the necessary build methods, the construction of an army table view and an army display became more readable and dynamic.

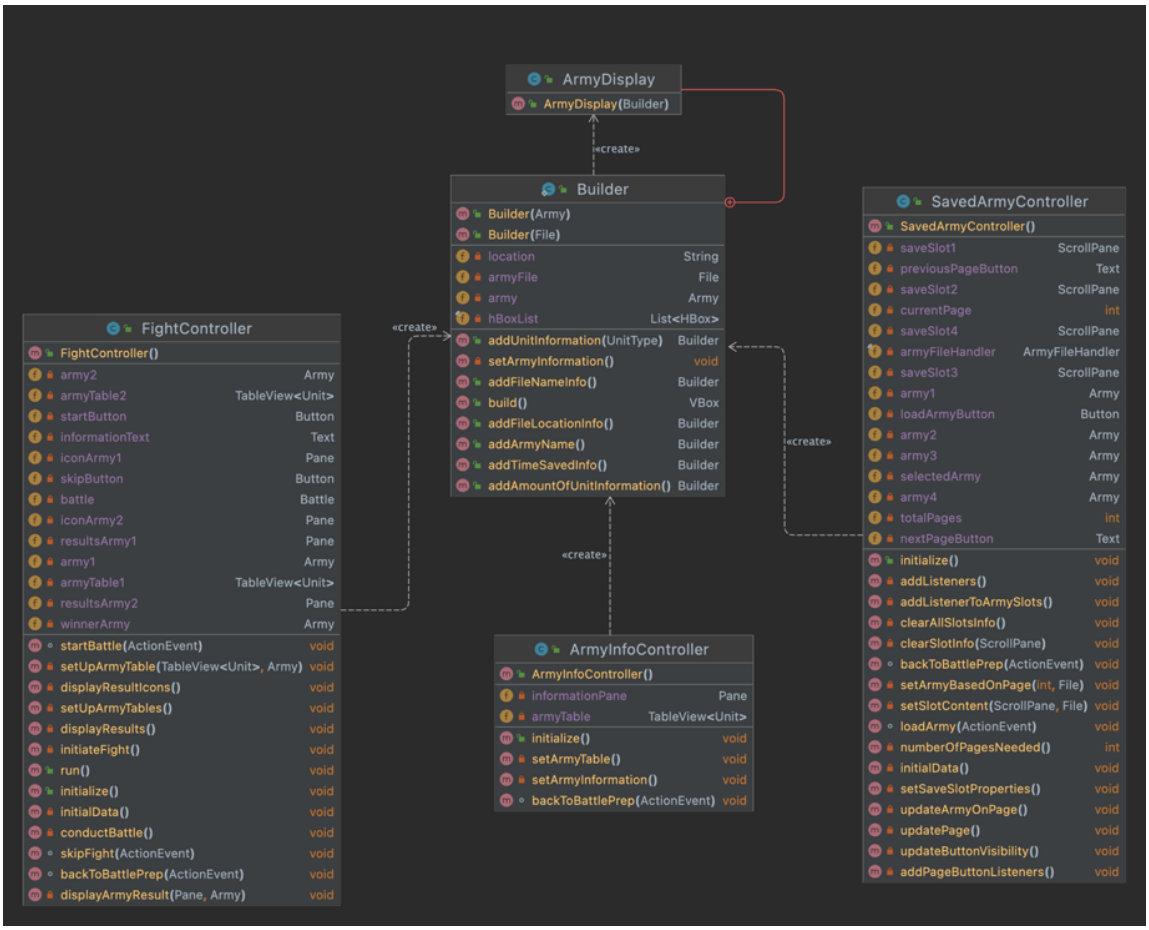

Figure 12. ArmyDisplay Class's Builder

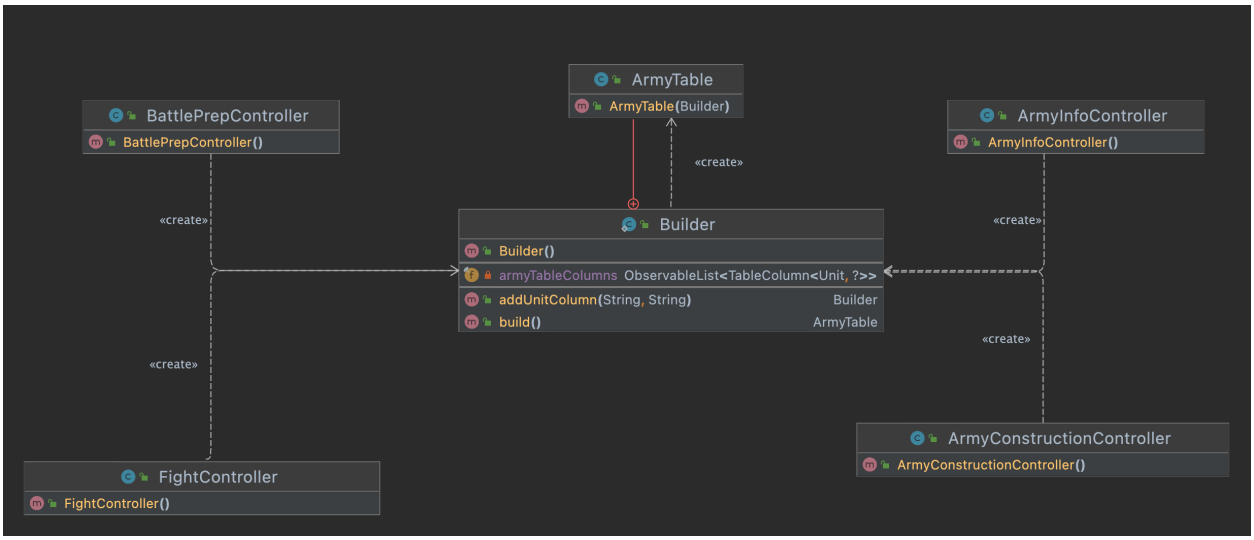

Figure 13. ArmyTable Class's Builder

Scene Handler

In order to display different scenes, the corresponding FXML files need to be loaded, the stage set, and the scene switched. Since this functionality was required in every controller, it was logical to gather the methods into a handler class. Later on, it became apparent that this centralized class for changing scenes utilizes the front controller design pattern. As such, the advantages of higher cohesion and looser coupling are provided.

Simulation Enum Singleton

Due to the necessity for the same data throughout controllers such as armies, terrain, etc., a consolidated class was created. An enumeration Singleton, named SimulationSingleton, was used instead of static variables in multiple classes for readability and higher cohesion due to less redundancy and a collected single purpose class.

Process

Since the project focused on exercising the learning outcomes, I worked steadily throughout the semester, using the lectures as pacing. A crucial part of the project process is the saving of various stages. Hence, the version control Git was utilized. Moreover, the local git repository was connected to a remote repository on GitLab. Commits were therefore saved locally and pushed to the remote branch. Using Git's conventions for writing a commit message – a short message potentially with a longer description below – combined with an angular style format of keywords, I created concise commit messages throughout the project.

Once connected to GitLab, the project was broken into different branches (each representing the different parts of the project). This acted as both a safety measure, as to not introduce any bugs to the master branch, and as to keep a record of checkpoints.

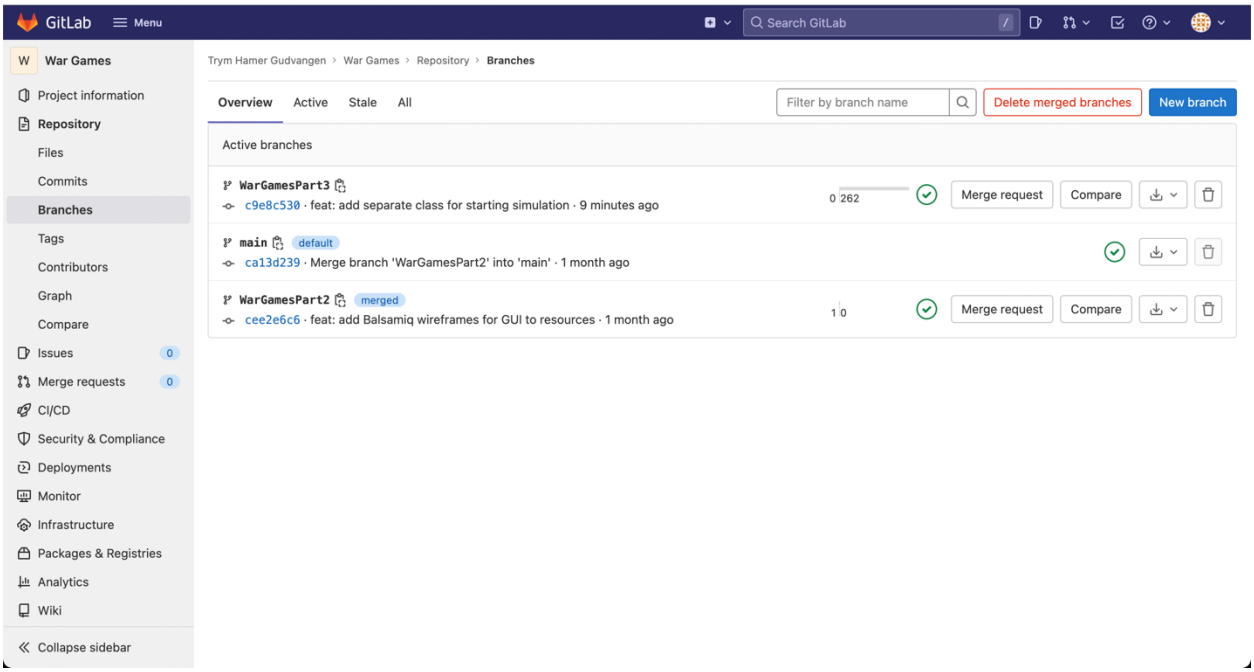

Figure 14. GitLab Branches

With a YAML file, GitLab also provides a pipeline service which automatically checks if builds, tests, and deployment was successful. I took advantage of this service in monitoring my project's state.

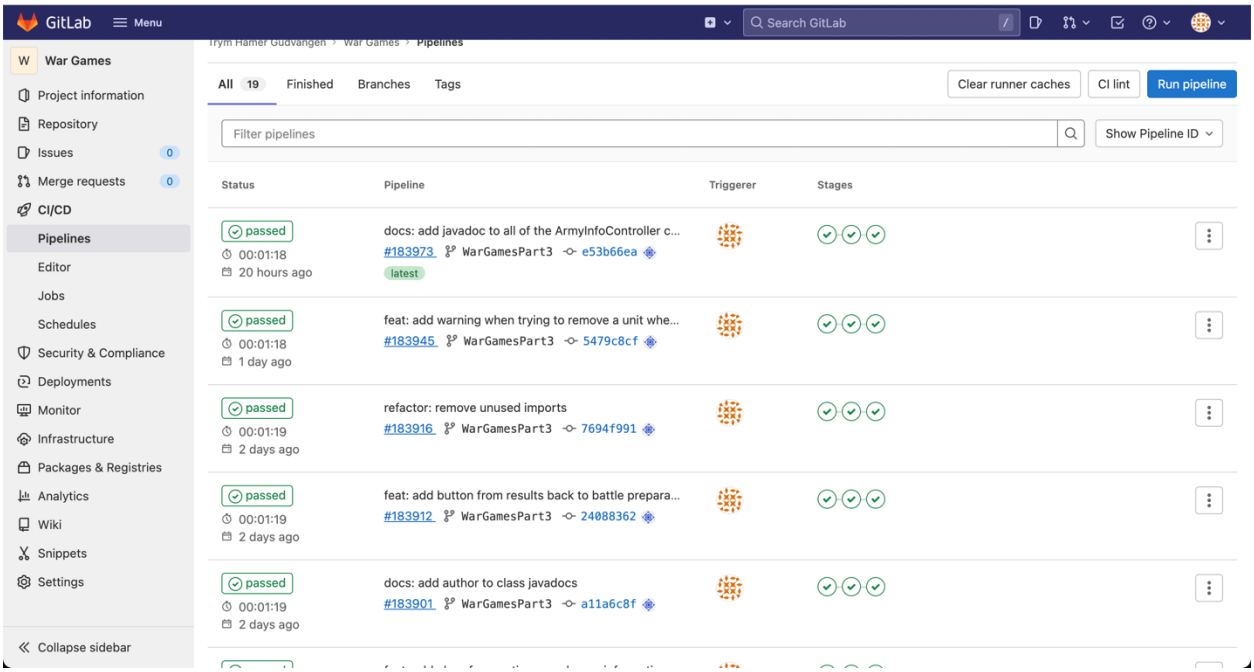

Figure 15. GitLab Pipeline

To maintain some overview of the deadlines, notes, and changes necessary for the project, I posted issues on GitLab's issueboard; however, since I was working on the project alone, this turned more into using TODO comments throughout the code.

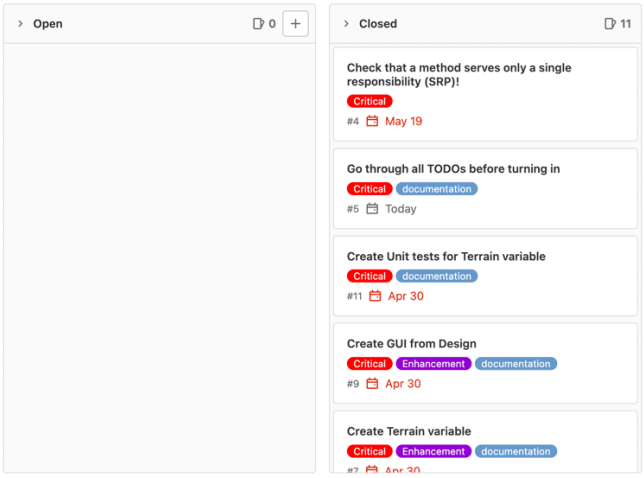

Figure 16. GitLab Issueboard

Reflection

Through this project, I have gotten to explore different scenarios where functional programming, inheritance, and various design patterns can be useful. The freedom to solve a problem in my own way has also given me the chance to evaluate diverse methods for tackling issues.

One of the challenges I faced was how to send data safely and cohesively between classes. This problem provided an opportunity to explore static variables and the singleton design pattern. Through independent research, I discovered the benefits of an enum singleton, and the end result paid off.

Another major obstacle was how to form unit tests. At first, I was unsure about how to properly structure such tests. Through the recommended videos, I learned unit tests serve dual purposes: examining a class and describing the functionality of the code to other programmers. Once I understood that unit tests tell the story of the class, I started structuring the code better. The arrange-act-assert structure also helped me to organize code in a legible manner.

An aspect of my approach I would change is the belief that user interfaces, such as SceneBuilder, allows for quicker and easier implementation. I believed visual features such as texts and tables would be easier if they were constructed in the FXML file. However, I did not consider how messy the controllers became, as well as the time consumption of creating those features. Eventually, I realized the use of design patterns such as the builder design pattern could help with the cleanness of the code, flexibility of the program, and time spent.

Conclusion

Overall, this project has provided students with the opportunity to understand the mechanisms taught in lectures on a deeper level. Through the different programming challenges faced, this project has successfully reinforced the learning outcomes of IDATT 2001. Looking into the future, the skills learned through this project will surely prove valuable.

Moreover, the final product not only fulfills the technical requirements but provides the user with additional functionality. Through an easy-to-use interface, a user may create, save, and load armies of their choosing in order to simulate a war in a given terrain.

References

8.1 customizing git - git configuration. Git. (n.d.). Retrieved May 24, 2022, from https://git-scm.com/book/en/v2/Customizing-Git-Git-Configuration

Conventional Commits. Conventional commits. (n.d.). Retrieved May 24, 2022, from https://www.conventionalcommits.org/en/v1.0.0/

GeeksForGeeks. (2020, July 16). *Front controller design pattern*. GeeksforGeeks. Retrieved May 24, 2022, from https://www.geeksforgeeks.org/front-controller-design-pattern/

Hombergs, T. (2021, February 22). *Writing meaningful commit messages*. Reflectoring. Retrieved May 24, 2022, from https://reflectoring.io/meaningful-commit-messages/#:~:text=The%20type%20of%20commit%20message,%E2%80%9D%2C%20or%20%E2%80%9Cdocs%E2%80%9D.

Jacobsen, D. H. (n.d.). *Builder design pattern*. INTEGU. Retrieved May 24, 2022, from https://integu.net/builder-pattern/

Kiwy, F. (2021, May 29). *Exploring Joshua Bloch's Builder design pattern in Java*. Blogs.oracle.com. Retrieved May 24, 2022, from https://blogs.oracle.com/javamagazine/post/exploring-joshua-blochs-builder-design-pattern-in-java

Pictures:

White flag:

PurePNG. (n.d.). *White Flag*. Pinterest. Retrieved from https://www.pinterest.com/pin/675610381577615583/.

Trophy icon:

Edurs34. (n.d.). *Trophy*. Pixabay. Retrieved from https://pixabay.com/no/illustrations/trof%C3%A9-kopp-gull-vinner-premie-5418654/.

Default terrain:

Best Military Wallpapers. (n.d.). WallpaperAccess. Retrieved from https://wallpaperaccess.com/best-military.

Forest terrain:

Darkmoon_Art. (n.d.). *Forest, foggy, tree ,mystical, landscape*. Pixabay. Retrieved from https://pixabay.com/no/photos/skog-t%c3%a5ke-tr%c3%a6r-mystisk-landskap-3394066/.

Hill terrain:

jplenio. (n.d.). *Tree, hill, landscape, nature*. Pixabay. Retrieved from https://pixabay.com/no/photos/tre-hill-eng-landskap-natur-3398108/.

Plains terrain:

Ivan. (2016). *Green Grass Field during Daytime*. Pexels. Retrieved May 24, 2022, from https://www.pexels.com/photo/green-grass-field-during-daytime-129539/.

Unit info:

Krystall, E. (2021). *Descargar el fondo de pantalla de 1 campo de batalla juego - bosque en llamas*. besthdwallpaper. Retrieved May 24, 2022, from https://www.besthdwallpaper.com/campo-de-batalla/1-campo-de-batalla-juego-bosque-en-llamas-dt_es-16533.html.

Main menu background:

dimarinski on DeviantArt. (n.d.). *Background02*. Pinterest. Retrieved May 24, 2022, from https://www.pinterest.com/pin/821695894482627079/.

Commander unit:

Pouaka. (2011). *Modern Great General Icon*. Civilization Fanatics Center. Retrieved May 24, 2022, from https://forums.civfanatics.com/media/modern-great-general-icon.3208/.

Infantry unit:

Pikeman. (2021). Civilization Wiki. Retrieved May 24, 2022, from https://civilization.fandom.com/wiki/Pikeman_(Civ5).

Ranged unit:

Archer. (2021). Civilization Wiki. Retrieved May 24, 2022, from https://civilization.fandom.com/wiki/Archer_(Civ5).

Cavalry unit:

Lancer. (2022). Civilization Wiki. Retrieved May 24, 2022, from https://civilization.fandom.com/wiki/Lancer_(Civ5)