COS 214 PROJECT

World War II Engine

Team FML:
Shashin Gounden
Jordan Timberlake
Tyrone Sutherland-MacLeod
Andile Ngwenya
Jonel Albuquerque

2022

Table of Contents

Task 1: Practical Assignment 3

Task 2: Design 4
2.1 Identify the functional requirements: 4
2.8 Provide at least two object diagrams showing the state of the objects active in the war simulation at a specific point in time: 9

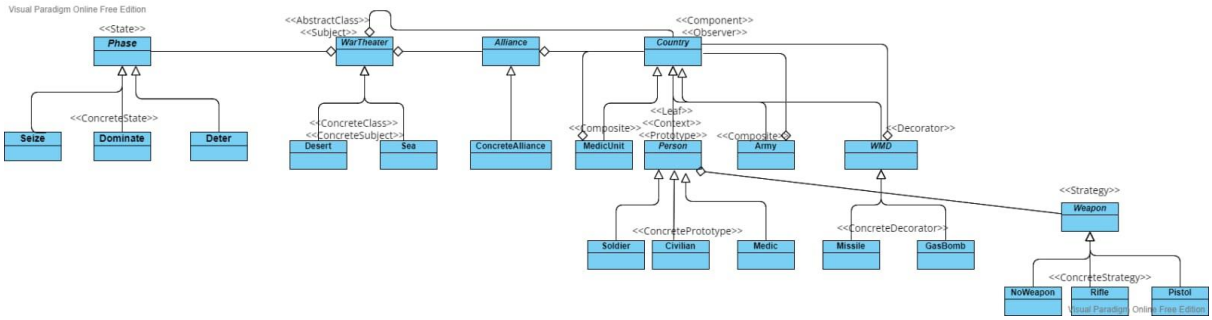
Task 3: Implementation 10
3.1 Implement your war simulator including at least a text-based interface: 10
3.2 Implementation of a graphical interface is not required, however will provide you with bonus marks: 10

Task 4: Report 10
4.1 You are required to write a brief about your research, ensuring that you reference all sources. Your brief should include what you understand about warfare as well as associated entities, phases, etc. that you will be using as part of your warfare engine. Make sure to document any further assumptions or decisions you made. Lastly ensure to include relevant definitions and explanations where required: 10
4.2 *A report stating how you applied the design patterns to address the functionality required by the system. This report should include UML diagrams to augment the explanation. This Task goes hand-in-hand with the Design task. Much of the design must be reported on in this task.* 12

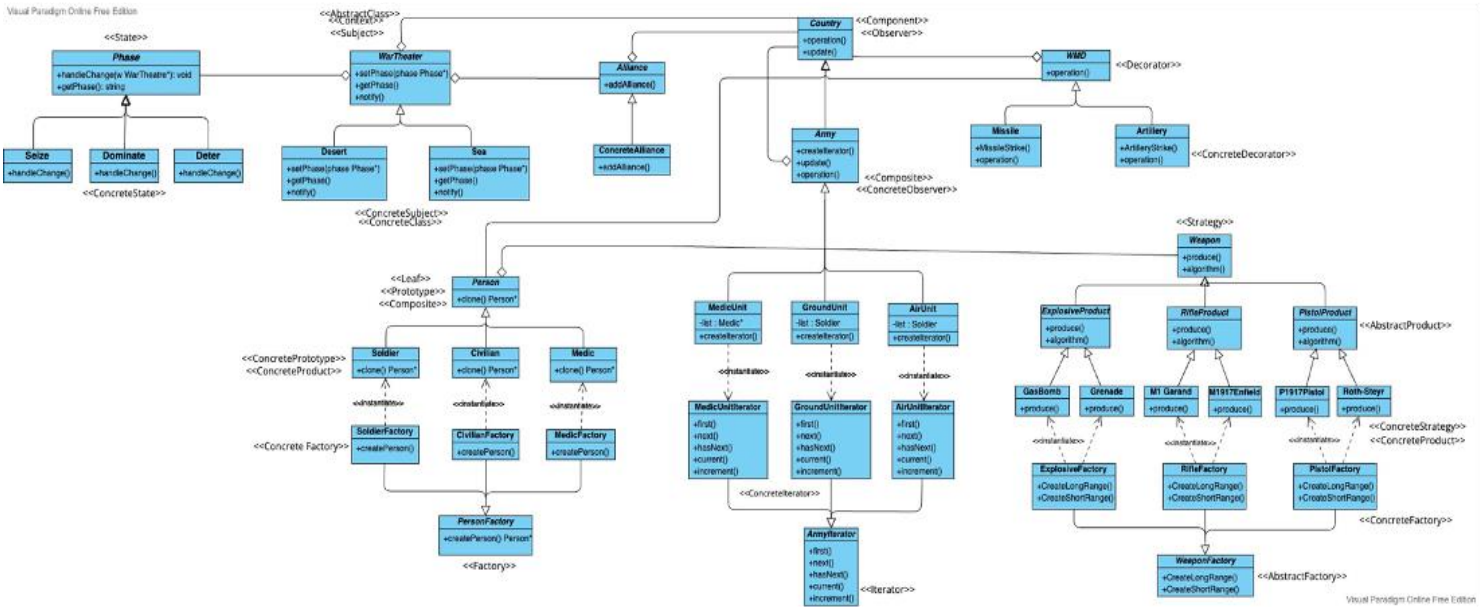
Task 5: Development Practices 19
5.1 Use git as the Version Control System (VCS). Every member of your group MUST make at least 10 (ten) commits: 19
5.3 Generate documentation for your system using Doxygen: 19
5.4 Develop your code according to best practices using a contract first design: 20
5.5 Use an automated unit testing framework to unit test your system. Full coverage is not required; however, every member of your team MUST implement various unit tests: 20

Task 1: Practical Assignment

This was our first and pre-initial UML diagram, designed using *Visual Paradigm* (Online Version). We make a note that the names of the participants of the various design patterns are enclosed using “<<ParticipantName>>” to allow for easy identification of the different design patterns. Below we have our very first design, followed by our pre initial design that we began coding with.



UML Diagram V1



UML Diagram V2

Task 2: Design

2.1 Identify the functional requirements:

We have identified that we are required to construct a generic war simulator that models the various elements of wars. This will be done using the C++ programming language. The design of the system will be conducted using the design patterns as identified by Gamma et. al. From these design patterns, at least 10 of them must be included in the design and the implementation thereof. The war engine will include various components of war including *War Theatres, Transportation, Entities, Phases of War, Changes to War Engine and Weaponry.* The system will also be required to provide an interface to set up war simulations and run a war simulation in a design mode or real mode.
Our particular system will be used to simulate World War II, which was a major global war that lasted from 1939 to 1945. Regarding our functional requirements, we have decided on these implementations for each:

The *War Theatre* component will comprise of two theatres the Battle of Dunkirk and Pearl Harbour, which will be the Sea and Airspace classes respectively.

The *Transportation* will be modelled as an attribute of an army's unit. Each army has 3 units that require different types of transport to carry out their different requirements, and these different vehicles will be created by builders that also belong to the unit.

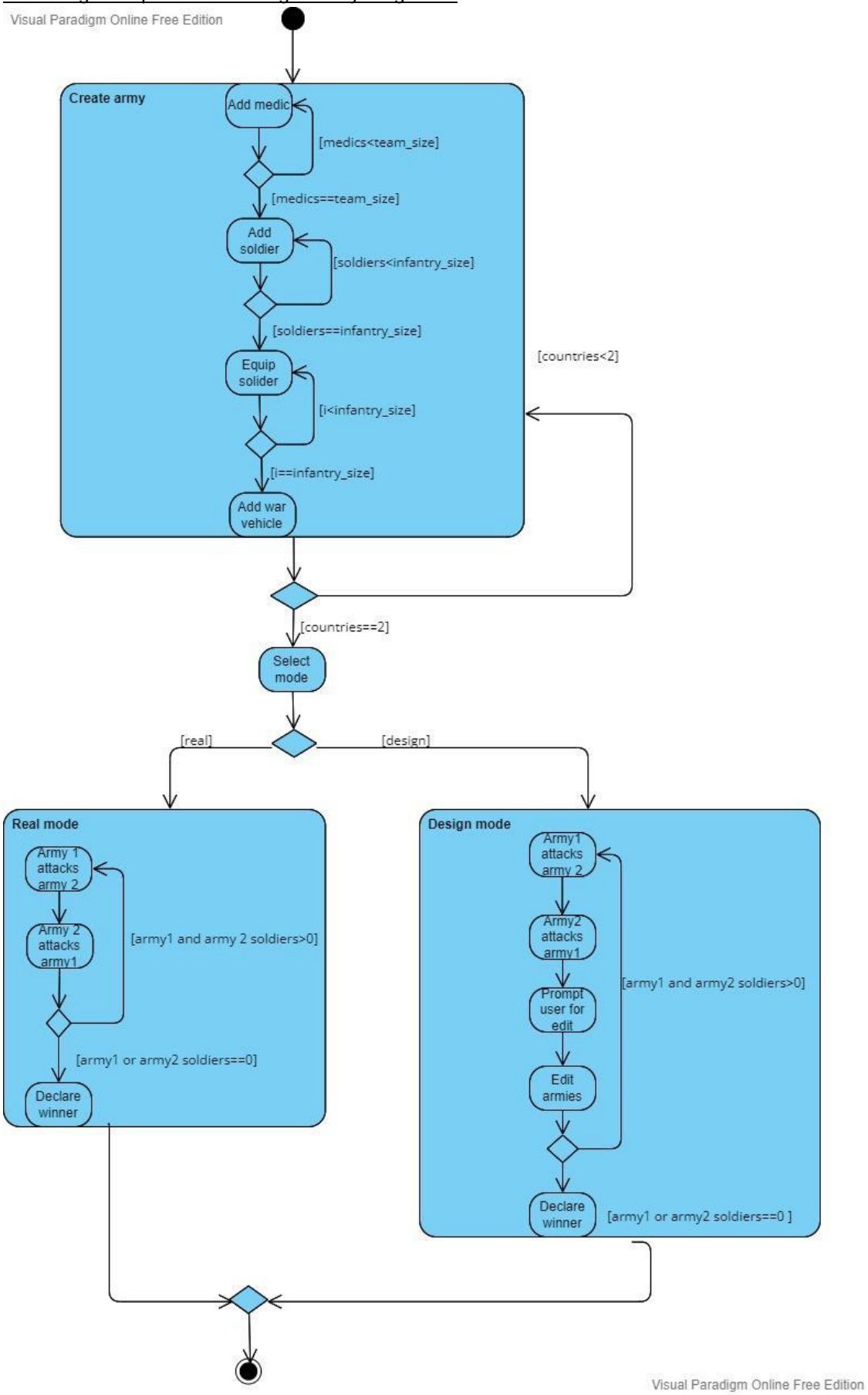
The *Entities* component is specific to the *War Theatre* component, mentioned above, that shall be implemented. The Entities consist of countries including, United States, The United Kingdom, France and of course Germany. These countries will each have their own army and units that will be used within the war engine.

We identified that *Phases of War* component could be modelled in 3 phases for our system, namely, phase I (deter), phase II (seize initiative) and phase III (dominate). Put simply, a country shall deter if their funds decrease dramatically, seize initiative when they are close to finishing all their funds, and will dominate when they have the most funds.

Changes to the War Engine goes hand-in-hand with the requirement of having a design mode and real mode. In design mode, a user will be able to make changes to the war engine by allowing countries to change alliances. Countries will further be able to add Weapons of Mass Destruction to the war engine, thereby facilitating a change to the war engine.

The system will also have different types of weapons that deal different damages. Each soldier will have their own weapon.

2.2 Design the processes using Activity diagrams:



2.3 Decide on the patterns to address the functionality defined by the functional requirements and processes:

A combination of 10 different design patterns shall be used to address the functionality defined by the functional requirements. These design patterns include:

1. State
2. Observer
3. Builder
4. Prototype
5. Decorator
6. Strategy
7. Template Method
8. Factory Method
9. Abstract Factory Method
10. Iterator

A description of what each design pattern is responsible for is provided below:

The *State* design pattern is used to model the *Phases of War* component. The phase of the war will change its state from either seizing, dominating, or deterring depending on the current war state.

The *Observer* design pattern is used by a Country to observe the phase of war that it's in. Hence changes to the phase will result in a country being notified so that the country can be updated and attack according to the relevant changes of its phase of war.

The *Builder* design pattern is used to build different modes of transport which pertain to different types of army units, to carry out transportation and or carry out attacks.

The *Prototype* design pattern will be used to create and clone People in this system, where people include soldiers, civilians, and medics.

The *Decorator* design pattern will be used to model Weapons of Mass Destruction in our system. A weapon of mass destruction can be decorated and become a Missile or Artillery.

The *Strategy* design pattern will be used to carry out a different type of fire, depending on the weapon carrying out its execution.

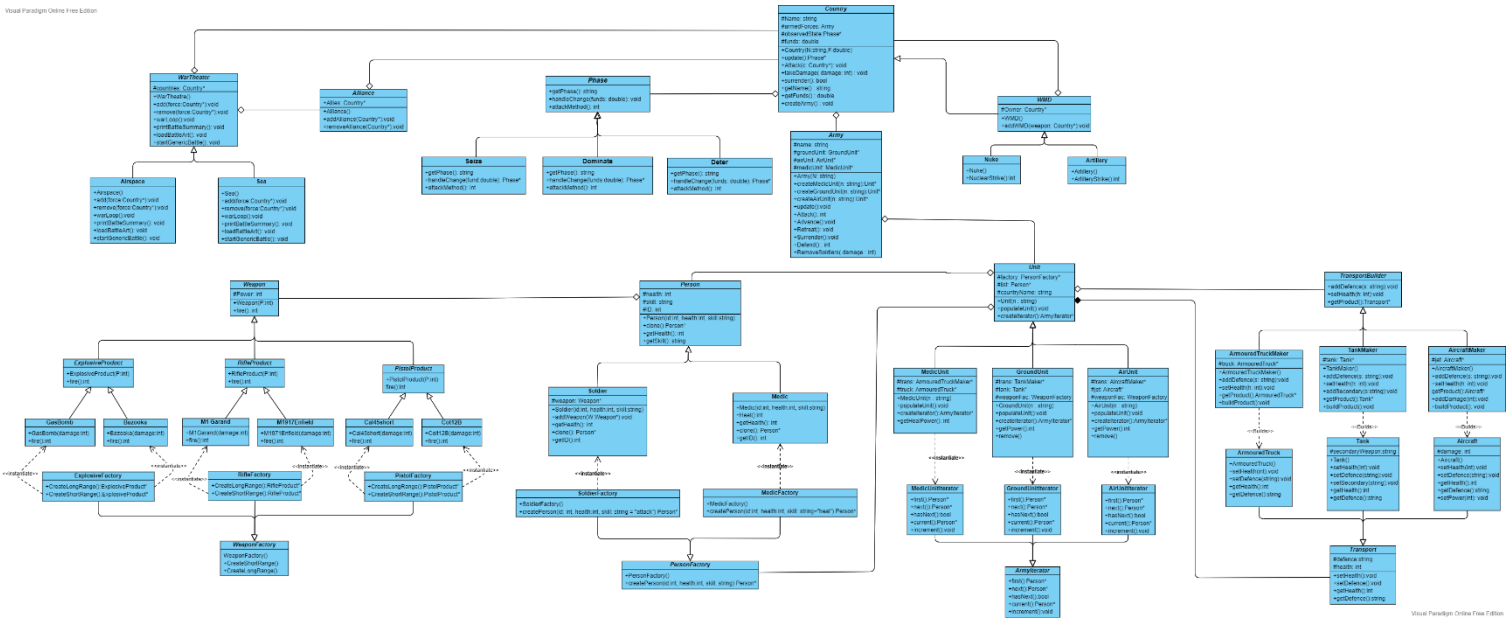
The *Template Method* design pattern will be used to model the inheritance between war theatre and the two types of "children classes" being the airspace or sea type of war theatre that countries will fight in. The Template Method will be used to start a generic battle sequence which will be common to all War Theatres.

The *Factory Method* design pattern will be used to create a Person entity, which means it shall include a SoliderFactory, CivilianFactory and MedicFactory.

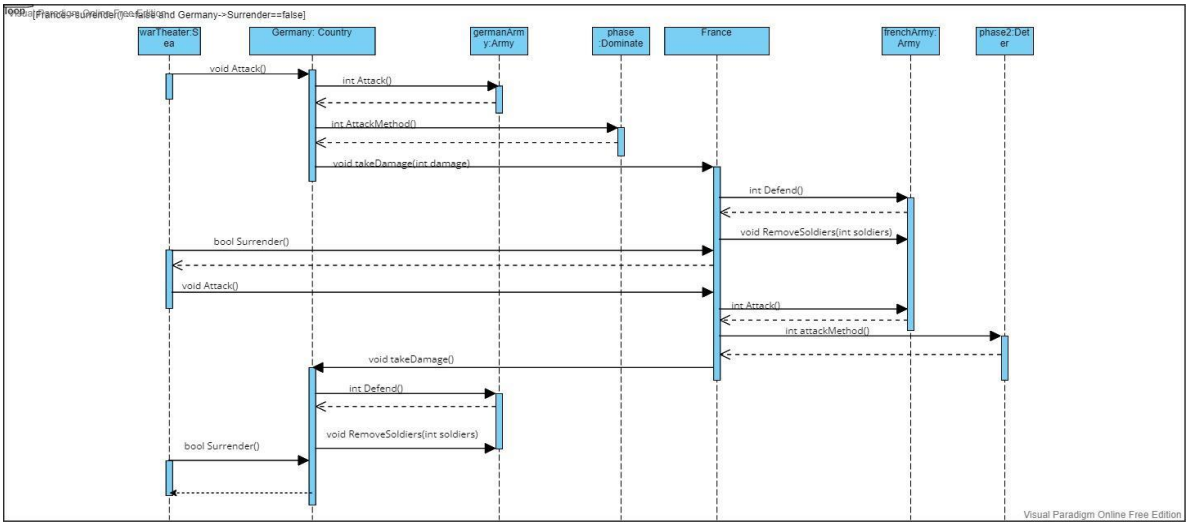
The *Abstract Factory Method* will be used to create different types of long range and short-range weapons. Hence it will comprise of an ExplosiveFactory, RifleFactory and PistolFactory, each of which can be a long- or short-range weapon.

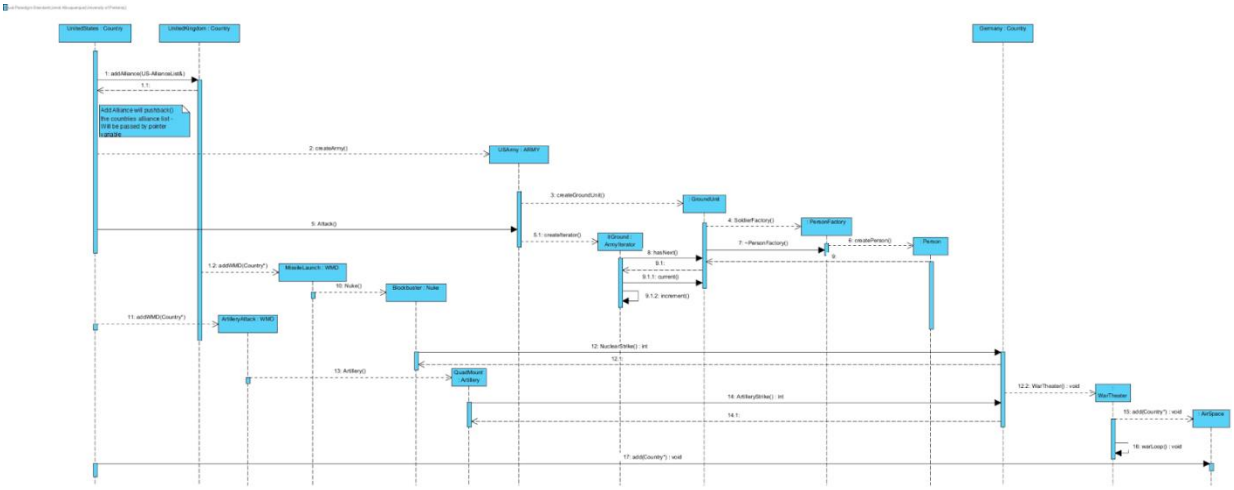
The *Iterator* design pattern is going to be used to traverse through army units in our system. This will allow us to iterate through soldiers and apply the relevant functions to all the soldier's part of a specific unit.

2.4 and 2.5 Design the classes for each of the identified patterns taking their interrelationships into account. Draw a class diagram of your system:

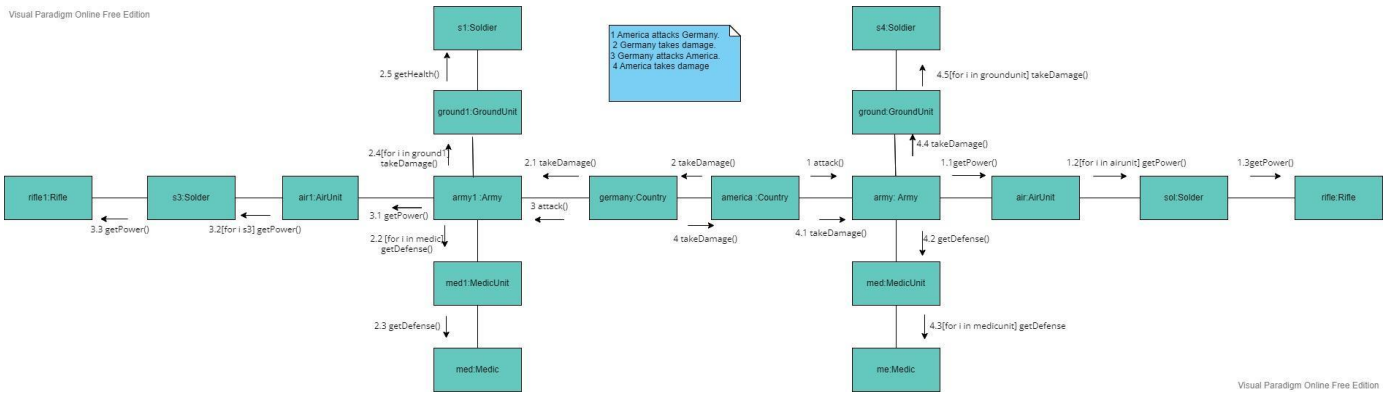


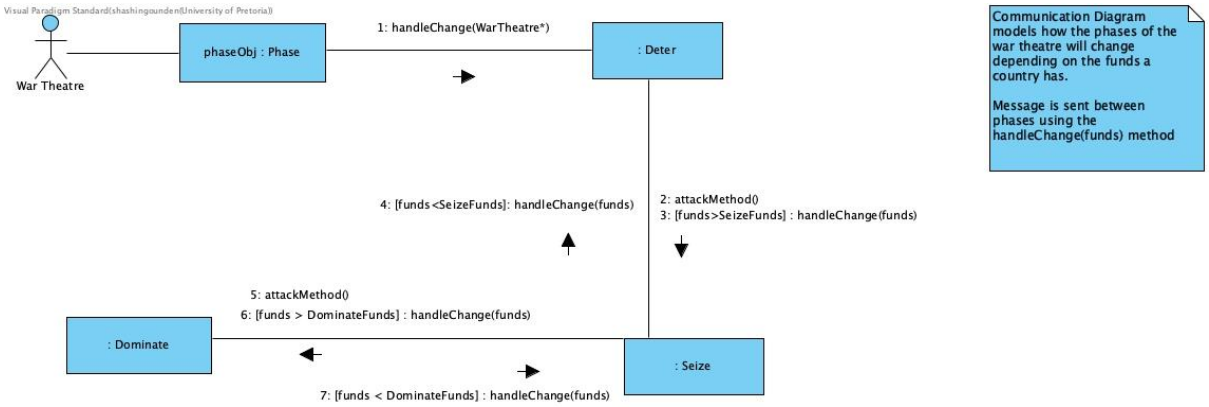
2.6 Draw Sequence and communication diagrams showing the message passing between objects:



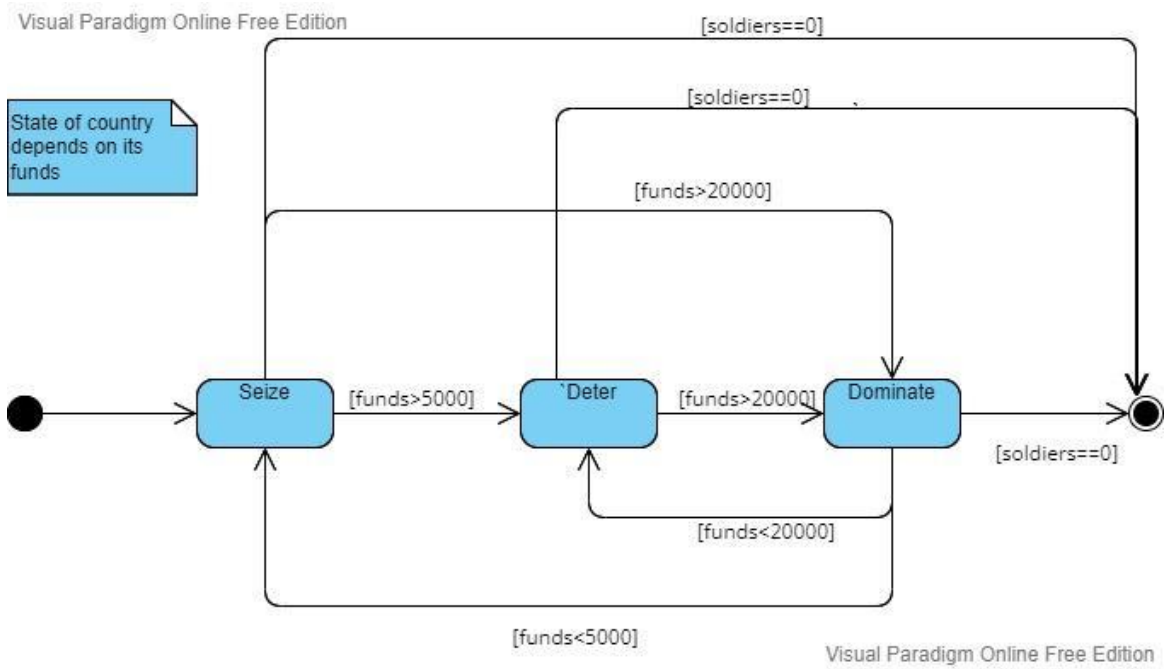


Communication diagrams:

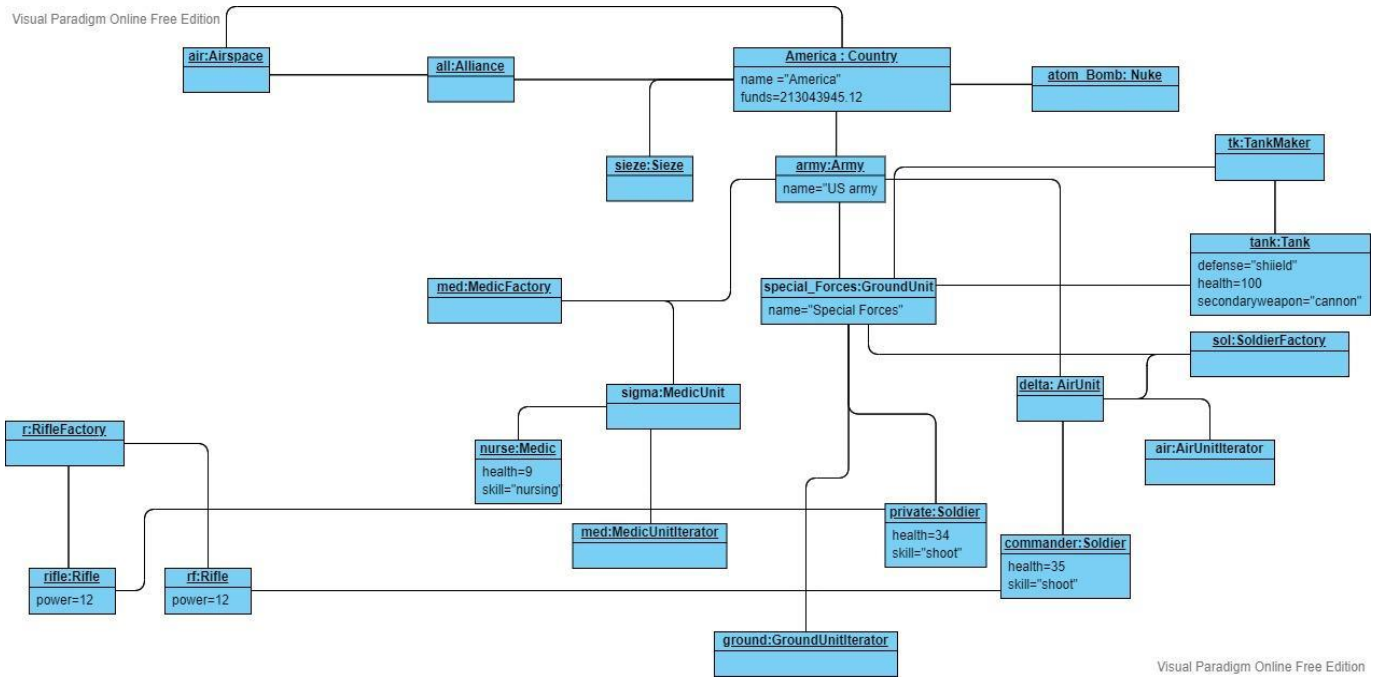


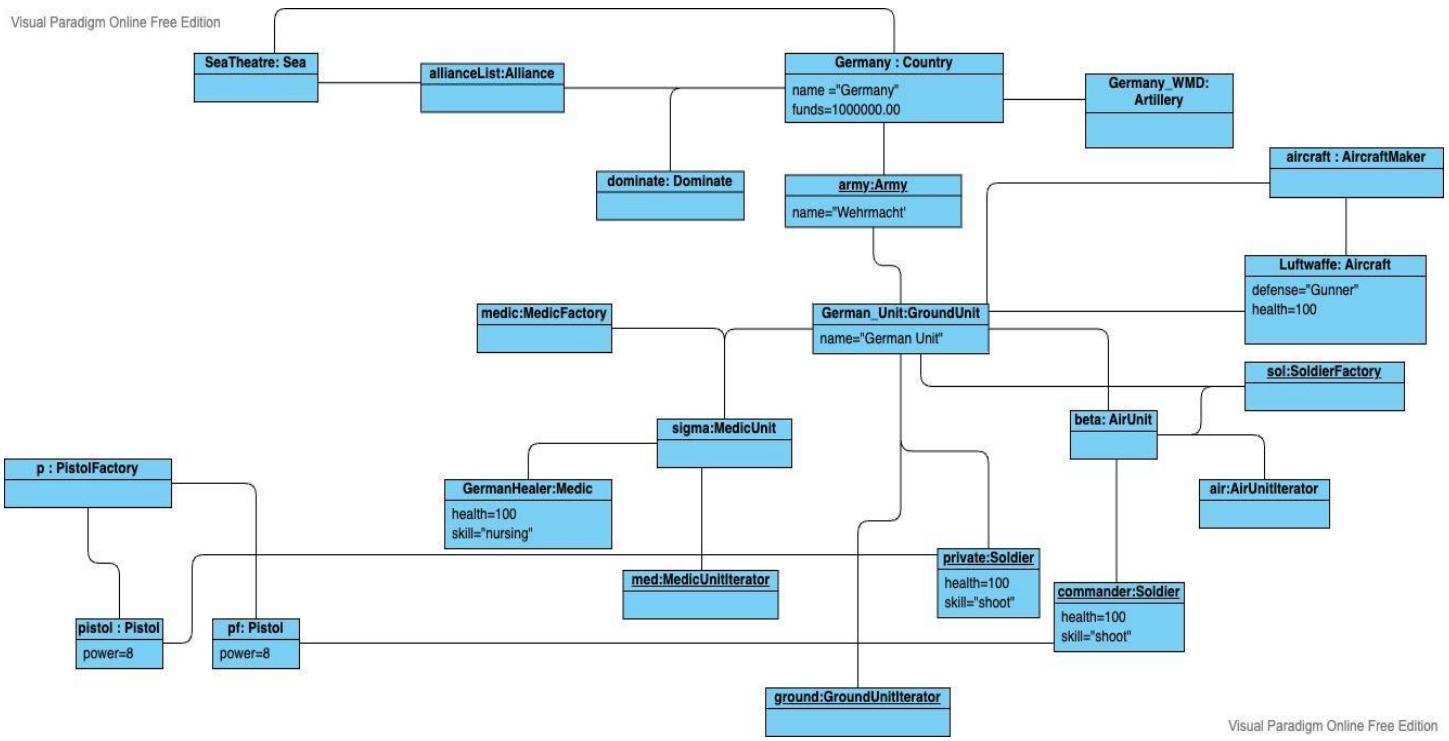


2.7 Design state diagrams showing how an object (which could also be a composite) changes state:



2.8 Provide at least two object diagrams showing the state of the objects active in the war simulation at a specific point in time:





Task 3: Implementation

3.1 Implement your war simulator including at least a text-based interface:

We note that the .h and .cpp files for our program have been put into separate folders, according to the hierarchy that they fall part of so that it is easier to find and manage the files during implementation and execution of the program. The makefile has been included and takes into account the separate hierarchy folders during compilation.

3.2 Implementation of a graphical interface is not required, however will provide you with bonus marks:

Our approach of a Graphical User Interface uses ASCII art, loaded from text files to make the game interactive. The ASCII art provides for a GUI that mimics a retro-styled video game to give the user a feeling of playing a game that was developed during the 80s.

Task 4: Report

4.1 You are required to write a brief about your research, ensuring that you reference all sources. Your brief should include what you understand about warfare as well as associated entities, phases, etc. that you will be using as part of your warfare engine. Make sure to document any further assumptions or decisions you made. Lastly ensure to include relevant definitions and explanations where required:

We define *Warfare* as the process of military struggle between two nations or groups, where this conflict usually involves armed forces. The nations may form alliances, which is a formal agreement between two or more countries for mutual support if a war was to arise.

Countries involved in warfare typically have armed forces that use *weaponry* which includes but isn't limited to rifles, pistols, and grenades. These armed forces will use different modes of *transportation* during an attack, to aid them in their progression as they march forward to take over an area of lands. Transportation such as tanks and armoured vehicles are usually used.

The aspects of war relating to the *Entities of Warfare* are simply the major countries and or states that were involved in a specific war. The Entities consist of countries including, United States, The United Kingdom, France and of course Germany.

The Entities will have battles in places that are called *War Theatres*, which is the entire land, sea or airspace that may become directly involved in war operations.

During our research we identified that *Phases of War* component could be modelled in 3 phases for our system, namely, phase I (deter), phase II (seize initiative) and phase III (dominate). Put simply, a country shall deter if their funds decrease dramatically, seize initiative when they are close to finishing all their funds, and will dominate when they have the most funds.

With the above assumptions in place, we modelled our system, a ***war engine***, to fulfil the above requirements. We have decided for our war engine to simulate **World War II**, with two major battles as the focus.

Our system comprises two types of *War Theatres*, namely a Sea and Airspace theatre. These theatres will correspond to specific battles that occurred during World War II, namely The Battle of Dunkirk and Pearl Harbour.

The *Entities* will be specific to a battle, and thus a war theatre. For example, France and Germany will be the Entities in The Battle of Dunkirk, which is the Sea war theatre.

We have decided to allow two countries in a War Theatre at a time during real mode, and these will be the major countries involved in the battle. However, more entities can be added in Design Mode, where an option to join alliances is given.

These countries will each have their own army and units that will be used within the war engine. These armies are equipped with weapons such as rifles, pistols, and explosives when they are created. In addition to weaponry, the units are equipped with certain transportation like tanks when they are created. *Changes to the War Engine* goes hand-in-hand with the requirement of having a design mode and real mode. In design mode, a user will be able to make changes to the war engine by allowing countries to change alliances. Countries will further be able to add Weapons of Mass Destruction to the war engine, thereby facilitating a change to the war engine.

Sources from research [Referenced using Harvard Referencing]:

Warfare, Warfare Definition and Meaning. Available at: https://www.dictionary.com/browse/warfare (Accessed: October 20, 2022).

Theater of war definition & meaning (no date) Merriam-Webster. Merriam-Webster. Available at: https://www.merriam-webster.com/dictionary/theater of war (Accessed: October 23, 2022).

Pearl Harbor (article) | World War II (no date) Khan Academy. Khan Academy. Available at: https://www.khanacademy.org/humanities/us-history/rise-to-world-power/us-wwii/a/pearl-harbor (Accessed: October 30, 2022).

History.com Editors (2018) Battle of Dunkirk, History.com. A&E Television Networks. Available at: https://www.history.com/topics/world-war-ii/dunkirk (Accessed: November 1, 2022).

List of World War II Infantry Weapons (2022) Wikipedia. Wikimedia Foundation. Available at: https://en.wikipedia.org/wiki/List_of_World_War_II_infantry_weapons (Accessed: November 3, 2022).

Car, land and air vehicles during the Second World War. Available at: https://www.pacific-rentals.com/useful-information/car-war.asp (Accessed: November 2, 2022).

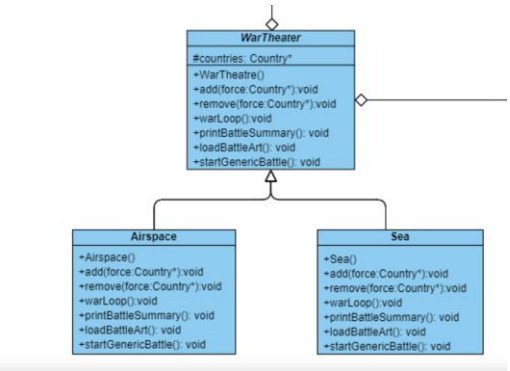
Alliance (no date) Encyclopædia Britannica. Encyclopædia Britannica, inc. Available at: https://www.britannica.com/topic/alliance-politics (Accessed: October 19, 2022).

American strategy and the six phases of grief (2016) War on the Rocks. Available at: https://warontherocks.com/2016/10/american-strategy-and-the-six-phases-of-grief/ (Accessed: October 20, 2022).

4.2 A report stating how you applied the design patterns to address the functionality required by the system. This report should include UML diagrams to augment the explanation. This Task goes hand-in-hand with the Design task. Much of the design must be reported on in this task.

1. **Template Method**: *Define the skeleton of an algorithm in an operation, deferring some steps to subclasses. TemplateMethod lets subclasses redefine certain steps of an algorithm without changing the algorithm structure*. (GOF:325)

   An integral part of our war simulator is that a battle takes place in a war theatre. This is where two (or more) countries are instantiated and launched into battle. The WarTheater is the AbstractClass, and the Airspace and Sea are the ConcreteClasses. For our implementation we wanted to be able to simulate battles in different types of war theatres, and so the **Template Method** Design Pattern is used to model a generic loading sequence when a War Theatre is selected. This is a



   suitable choice for modelling this aspect of our system because the two war theatres have significant similarities but demonstrate no reuse of common interface or implementation. The system required the functionality of having multiple war theatres, and by using this we simplify the process of starting a battle in the specific war theatre. The figure alongside is a snapshot of the class hierarchy of this pattern, where the function startGenericBattle() is the template method function which loads the relevant ASCII text before a battle starts, prints out a summary of the battle using a type-writer effect and also begins the relevant war loop. The participants are identified as follows, WarTheater is the AbstractClass participant and Airspace in conjunction with Sea are the Concrete class participants.

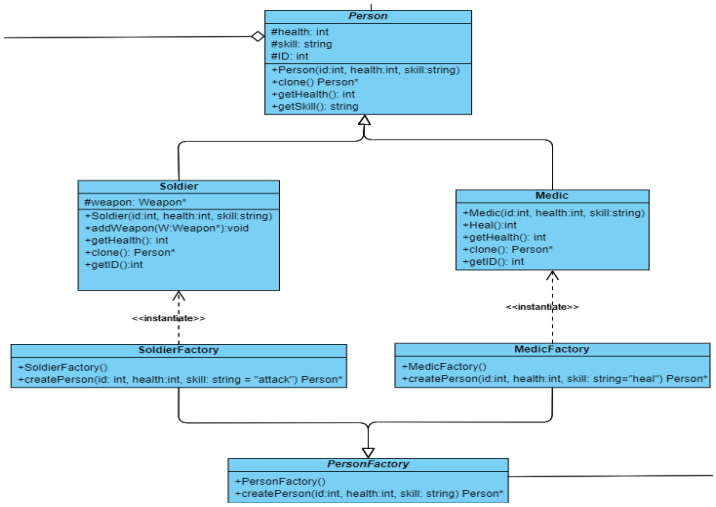
   After a war theatre is created, the participants of battle (being countries) are instantiated before beginning to battle. Upon instantiation, a country creates its army which is composed of three different units (ground unit, medic unit and air unit) each of which play different roles in a country's attack. Furthermore, these units are composed of many soldiers, each of which have their own weapon which can be of many types, and the units also need to be able to create and maintain different vehicles for their different purposes. This army hierarchy entails the biggest creational components of the system, the following are the design patterns we used to tackle this massive army creation:

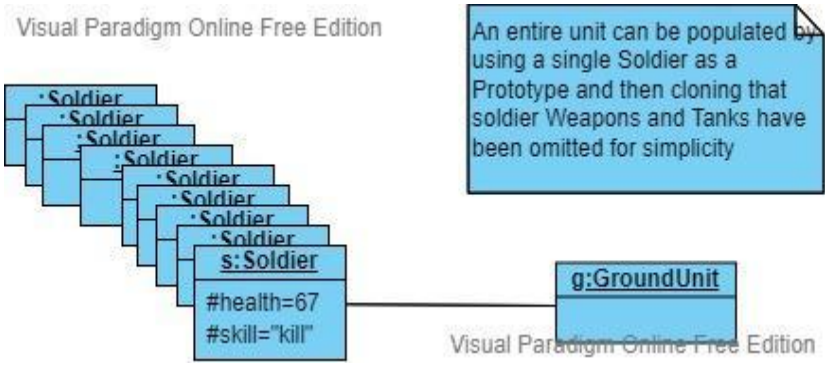
2. **Factory Method**: *Define an interface for creating an object, but let subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses*
(GOF: 107)

   Given that our system has two types of persons, namely soldiers and medics, we used the **Factory Method** to create the different persons. Each unit has a person factory of the type of person that they need to populate their unit, and so the medic unit has a factory to populate their unit with medics, and the other two units have soldier factories to populate their units with soldiers. PersonFactory is the Creator participant and MedicFactory and SoldierFactory are the ConcreteCreators. Person is the Product participant and Soldier and Medic are the ConcreteProduct participants. Using factories, we can ensure that the correct Person objects are added to correct unit objects. This is important since the number of soldiers or medics is not known beforehand and soldiers and medics can also be created during the fighting so the creation process must be managed. Factories are a convenient way of doing this. In our implementation each unit is associated with its corresponding factory making it easy to populate units or even add to them.

3. **Prototype**: *Specify the kinds of objects to create using a prototypical instance and create new objects by copying this prototype.* (GOF:117)

   Our Person hierarchy also employs the **Prototype** method, giving us the option to clone persons without the use of factories in case we need another way to generate more soldiers or medics. The Prototype participant is the Person class and the ConcretePrototypes are Medic and Soldier. The Prototype pattern is especially suitable since soldiers are all similar or have little variation in state so cloning them is an easy way to increase their numbers if need be.



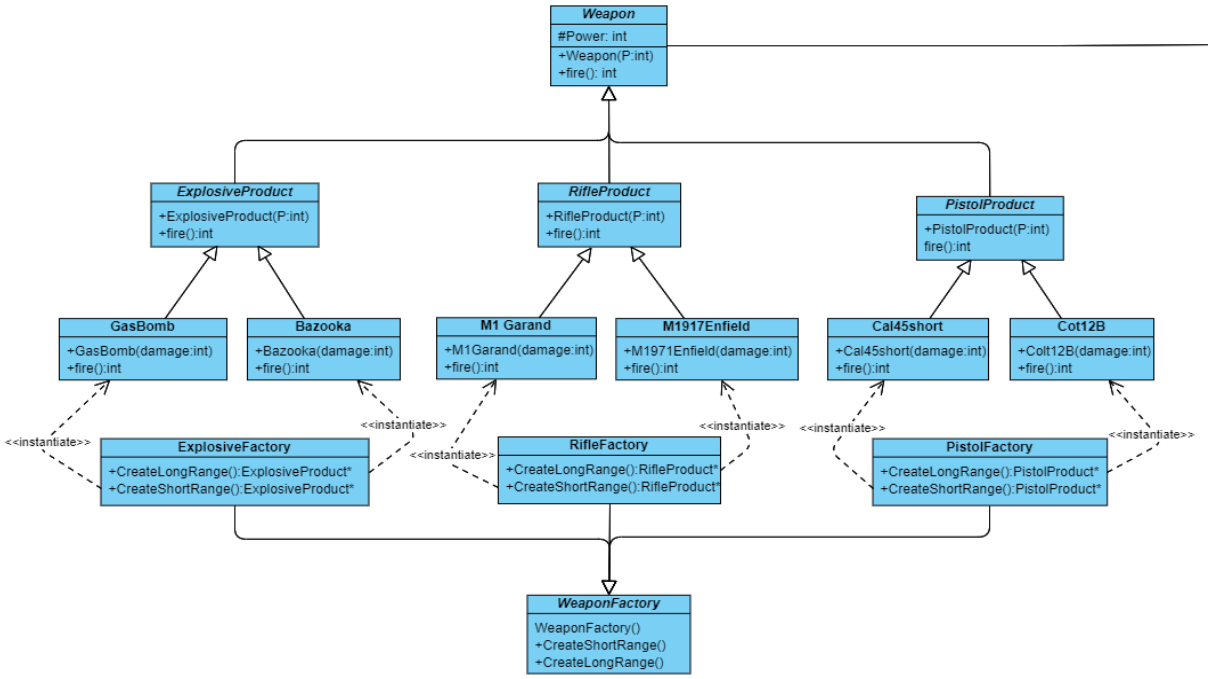


4. **Abstract Factory**: *Provide an interface for creating families of related or dependent objects without specifying their concrete classes.* (GOF:87)

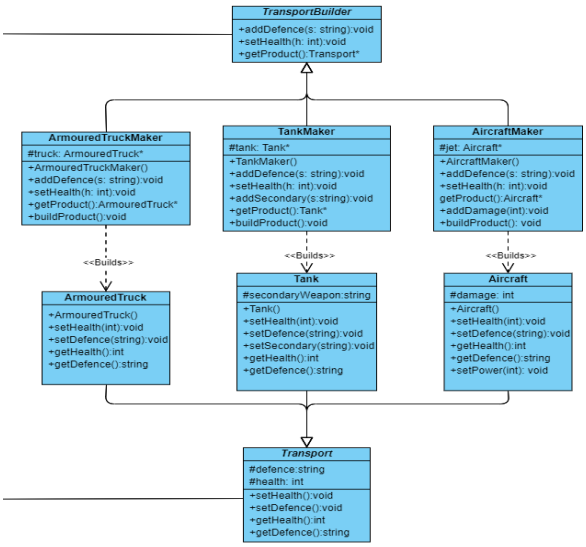
   After creating a new soldier, each one must be given a new weapon. We desired to have 3 types of weapons (explosives, rifles, and pistols), and each of these weapon types can be further categorised into long- and short-range variants (a total of 6 different possible weapons). To adequately create such a variety of weapons, we used the **Abstract Factory** method. Each unit has an abstract factory and based on fine-tuned probabilities a particular abstract factory is used to create either a short- or long-range variant of the required weapon type. The probabilities of weapon creation that we used are in place for fairness purposes, since explosive weapons do a lot of damage, rifles a medium amount and pistols, less. This ensures that a total army's attack damage is proportional, considering the size of units.

5. **Strategy**: *Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy Lets The algorithm varies independently from clients that use it*. (GOF:315)

   Our weapons make use of the **Strategy** pattern by each having a fire() method that yields different results depending on which type of weapon is fired. The Weapon class is the Strategy participant and the Soldier class is the Context. Their subclasses are the Concrete participants in the pattern.

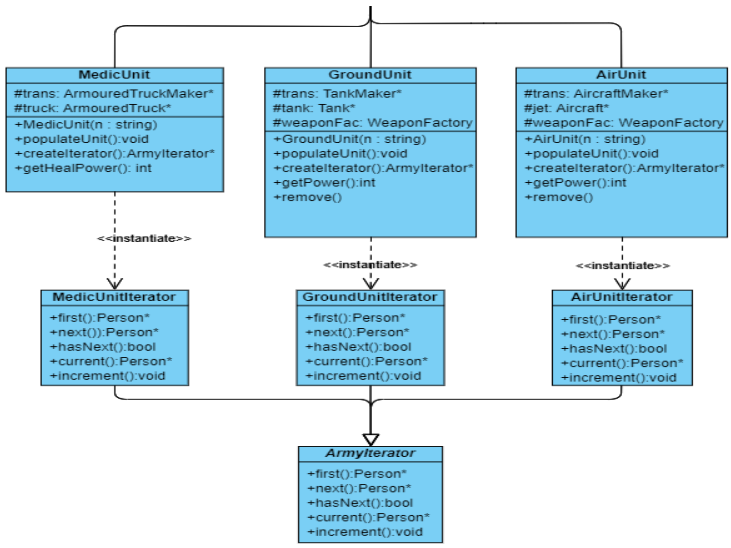


6. **Builder**: *Separate the construction of a complex object from its representation so that the same construction process can create different representations.* (GOF:97)

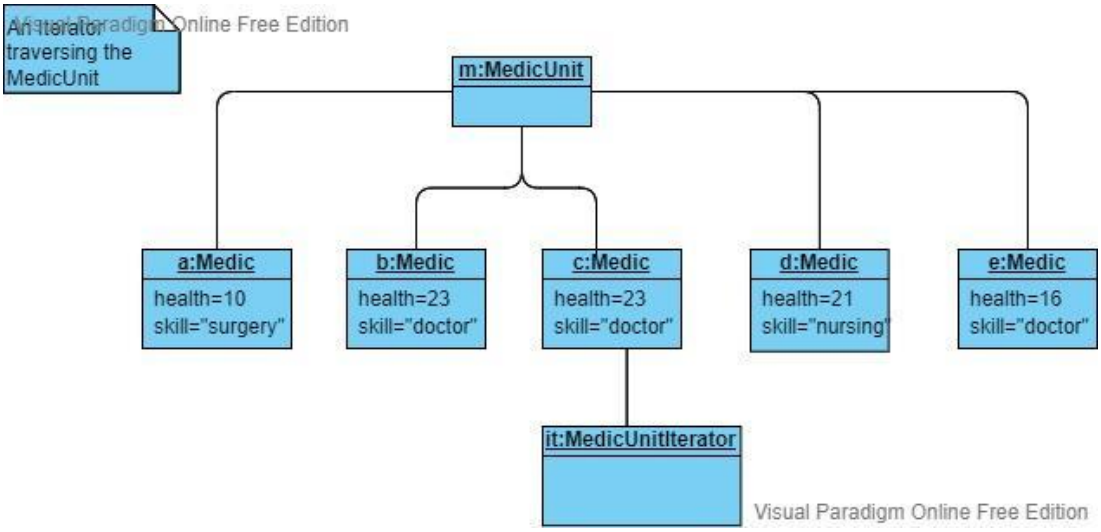


The last creational design pattern we used was the **Builder** pattern. This was chosen because the different units need different types of transport with different functionalities. Each unit has a builder that takes steps to assemble and return a fully built vehicle of the required type.

7. **Iterator**: *Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation.* (GOF:257)

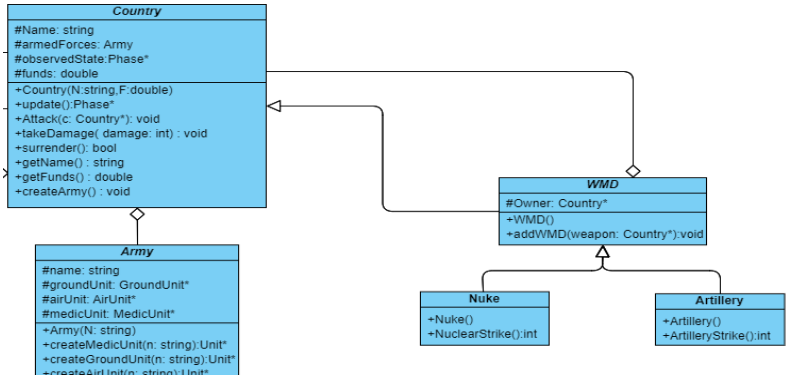


As highlighted above, our units have many (tens of thousands of) people, and so another major design pattern we needed to use was the **Iterator** pattern
The Aggregrate participant is Unit and the ConcreteAggregrates are MedicUnit, GroundUnit and AirUnit. The ArmyIterator is the Iterator participant and the ConcreteIterators are the MedicUnitIterator, GroundUnitIterator and the AirUnitIterator. Each unit instantiates an iterator (either a ground, medic or air unit iterator) of their respective type, and they are used for the traversal of the unit lists, in order to aid us with certain functions such as adding, removing and totalling up fire power and healing power values (which are used in a countries attack method). Iterators allow us to obtain information about units without exposing their internal structure or representation.



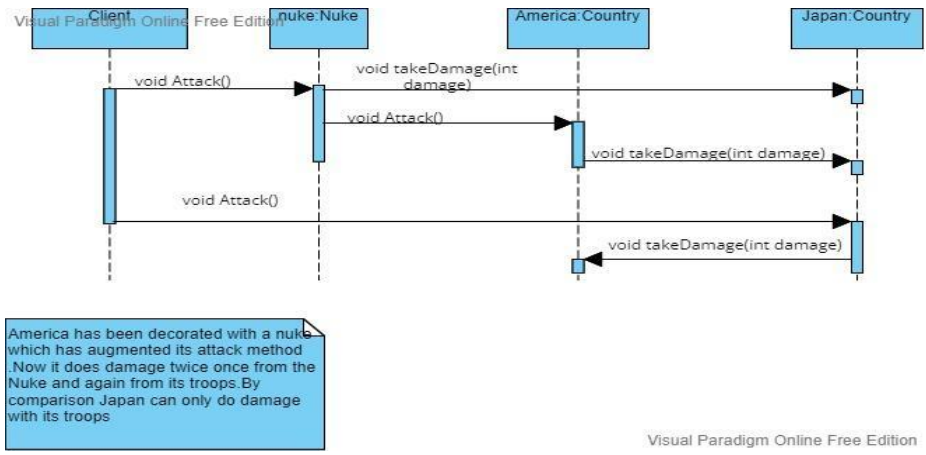
8. **Decorator**: *Attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality. (GOF:175)*

   The **Decorator** pattern was used in the WMD class (WMD means weapon of mass destruction). Country is the Component participant and WMD the Decorator. The ConcreteDecorators are Nuke and Artillery which give countries the ability to fire nukes and launch artillery strikes. In order to increase the damage a Country can do when attacking, our system makes it possible to decorate countries with WMDs like Nuke. This is especially useful in design mode which the user controls. A powerful weapon can help turn the tides of battle and it is possible to dynamically add these weapons using the Decorator pattern. It enhances the attack to also include deployment of weapons of mass destruction.



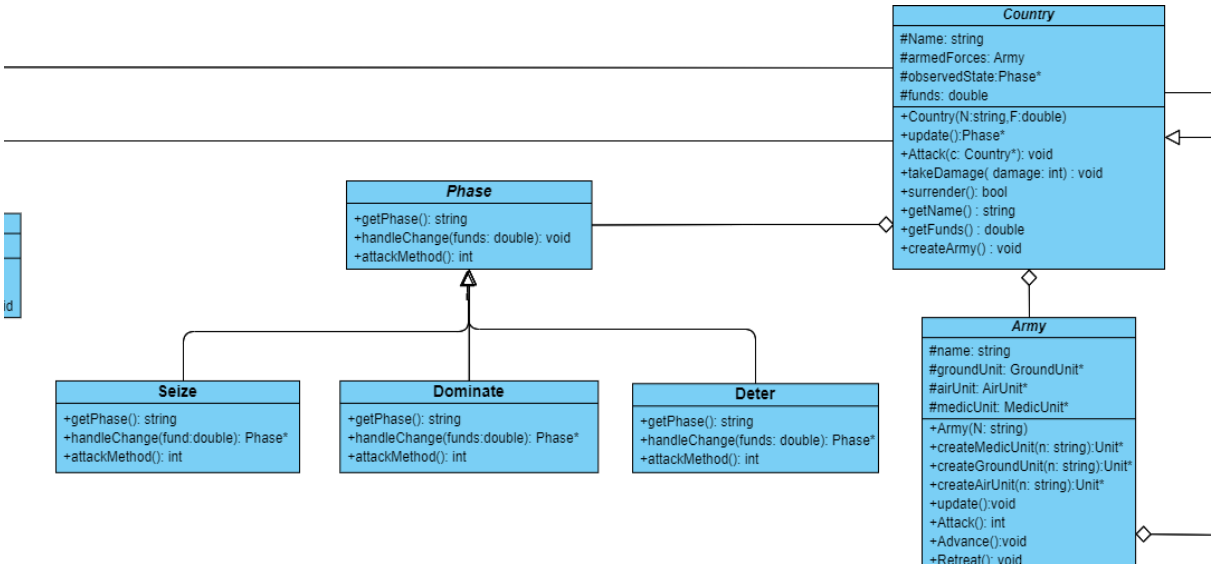
(Decorator class diagram to the left)

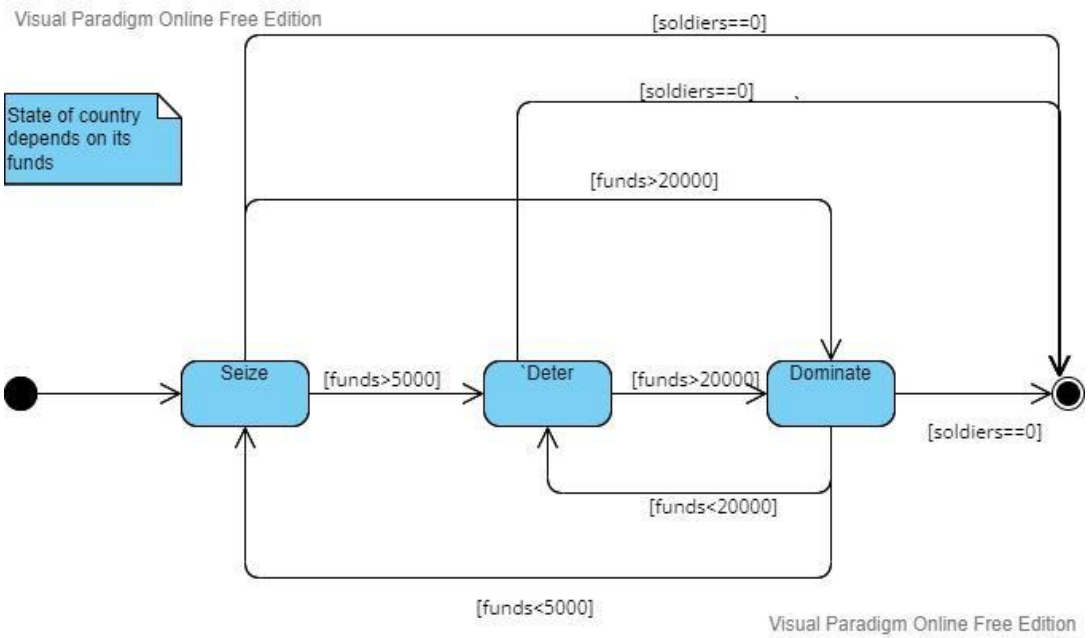
(Decorator sequence diagram to the right)



9. **State**: *Allow an object to alter its behaviour when its internal state changes. The object will appear to change its class.* (GOF:305)

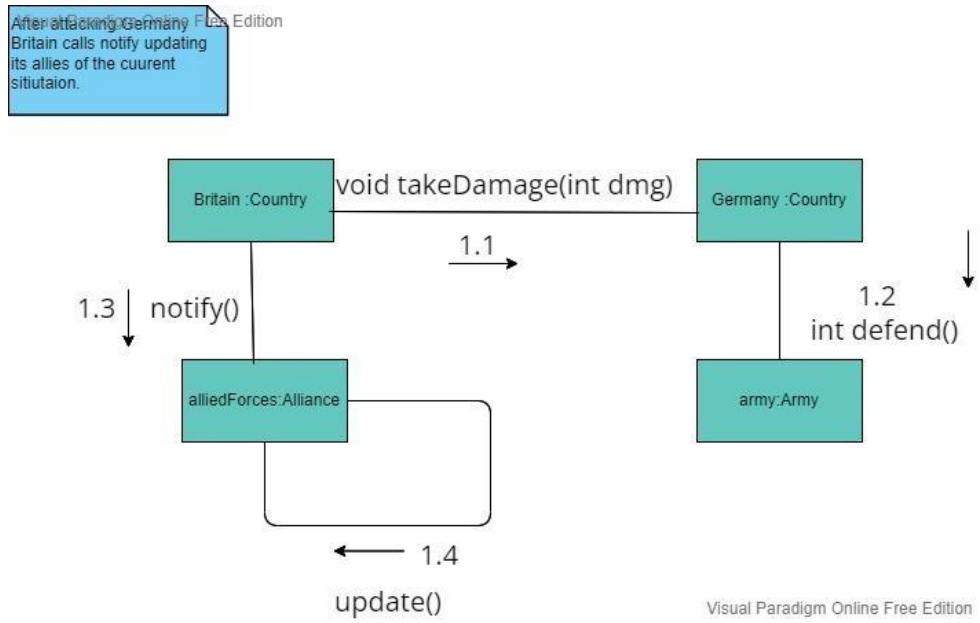
   The **State** pattern was used to model the phases of war that countries go through and to regulate the damage an attack could cause, depending on whether a country is in the deter, seize, or dominate state. State changes are based on the funds a country has. As the funds rise or fall, they could move Seize to Deter or to Dominate phases. This served to simplify the conditional logic related to attacking since this was self-contained in the states or rather the phases themselves. The Phase class is the State class, Dominate, Deter and Seize are the ConcreteStates. Country is the Context class.





10. **Observer**: *Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically* (GOF:293).

    Finally, we modelled the **Observer** pattern to work hand in hand with our state pattern. Country is the Subject participant and Alliance is the Observer. After attacks countries contact their allies by calling their notify method. The Alliance objects respond by calling their update method which prints a response. In addition, countries can also receive financial aid from their allies. A country with allies to communicate with will be at a major advantage compared to a country without.

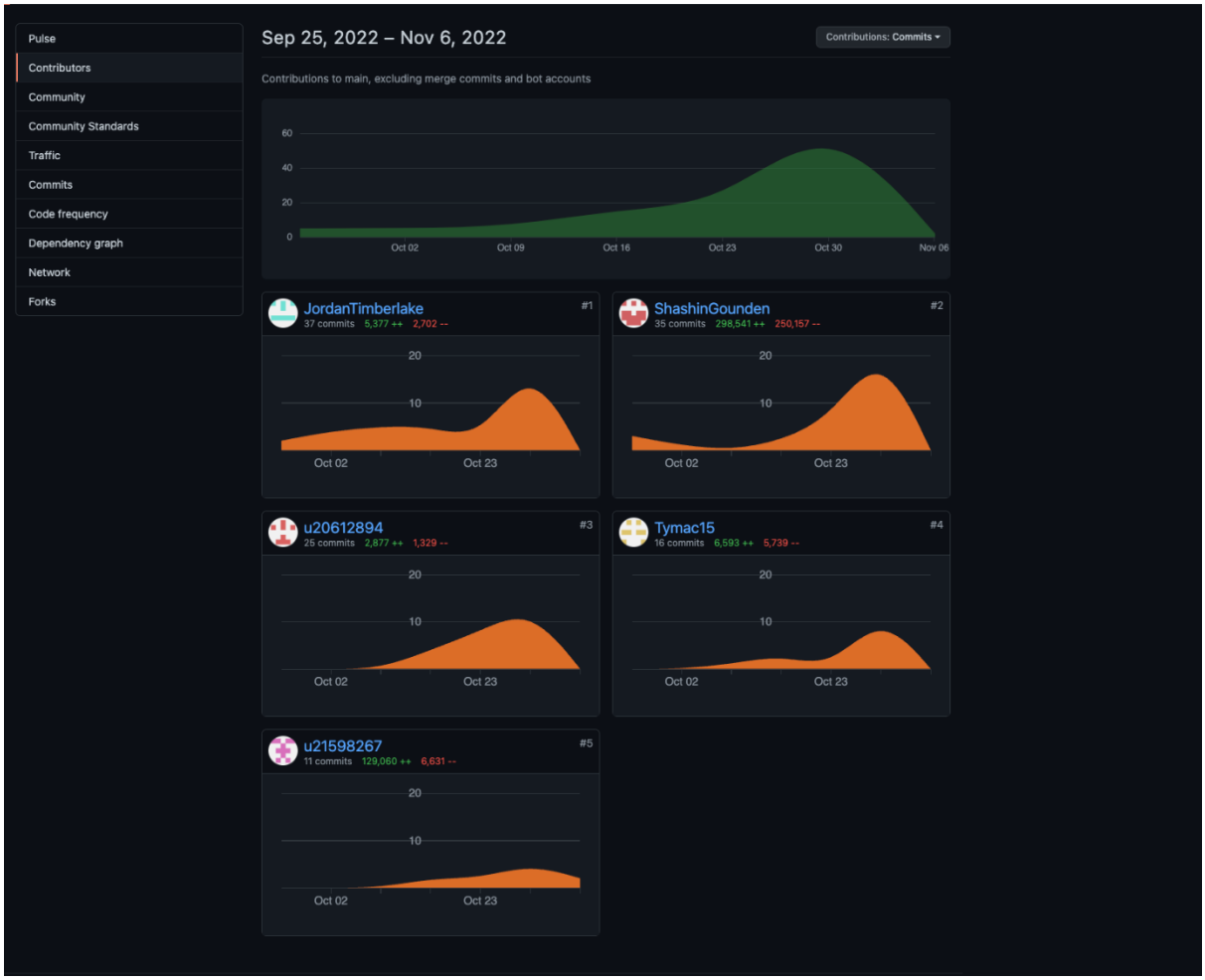


<u>Conclusion</u>: We used a variety of creational, behavioural, and structural design patterns to effectively set up a virtual war environment. We've given a user the ability to simulate real historical battles, as well as the option of simulating fictional ones that allow the user to play with different countries, starting funds and make different war choices that can decide the fate of a battle.

Task 5: Development Practices

5.1 Use git as the Version Control System (VCS). Every member of your group MUST make at least 10 (ten) commits:

Screenshot of the commits of each member:



5.2 Document your code using C++ documentation standards:

We note that we have followed C++ documentation standards by using header (.h) files and (.cpp) files for our implementation. We have also used the standard for creating comments where necessary, denoted by a double back slash in our source files. In conjunction with this, we used descriptive variable names and indentation as required.

5.3 Generate documentation for your system using Doxygen:

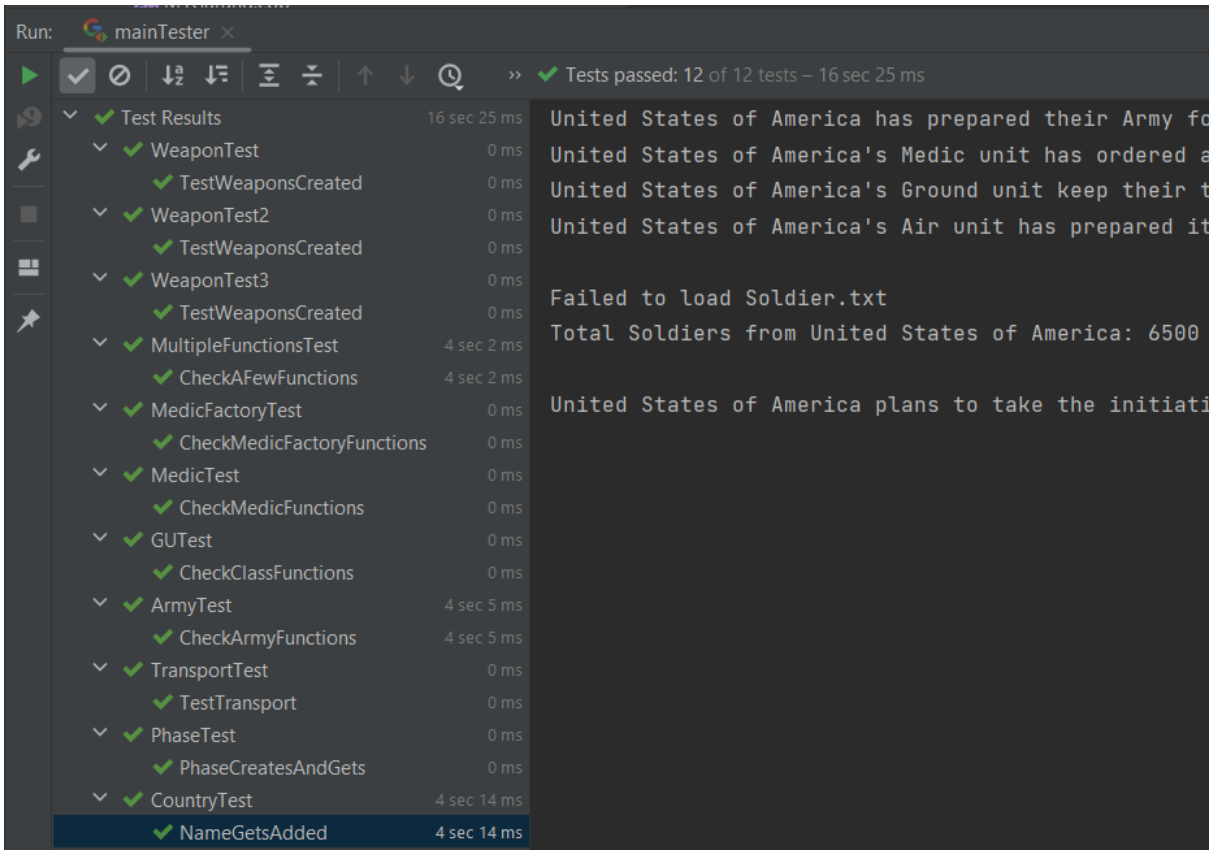
Our doxygen documentation can be found in the Data folder in the archive and can also be downloaded at: https://github.com/ShashinGounden/COS214-Project/tree/main/Data

5.4 Develop your code according to best practices using a contract first design:

We developed our system by planning accordingly using our UML Class, Sequence, Activity and Communication diagrams. Hence, we followed a contract first design approach by starting development after planning how the system will work.

5.5 Use an automated unit testing framework to unit test your system. Full coverage is not required; however, every member of your team MUST implement various unit tests:

We used GoogleTest (GTest suite) to run various unit tests in our program.



The original Google doc that all team members worked on can be found at: https://docs.google.com/document/d/1fIBljEvgrzkKuuCGGKqeGtKyDWfG38LYcaSMP7wZ9kw/edit?usp=sharing

Link to the Github repository: https://github.com/ShashinGounden/COS214-Project