RuntimeCounterTerrorists

COS214 Project

Google Docs
📄 RuntimeCounterTerrorists
Github
https://github.com/DieSeeKat/RuntimeCounterTerrorists

u21434345 - Lukas Anthonissen (DieSeeKat)
u21446459 - Daniël van Zyl (DvZ02)
u21429121 - Stefan van der Merwe (Sniperlyf3)
u21529583 - Adam Osborne (AdamOsb)
u21637386 - Dharshan Gopaul (FuryOfD)
u20572337 - Willem Prinsloo (Liam-Prinsloo)

Planning and Initial Design

The manner in which we interpreted the project specifications, led us to designing a war simulation where multiple computer controlled empires wage war against one another on a battlefield.

We picked the Roman conquests as the milieu around which the simulation will be modeled.

During the Roman conquests, the Romans waged war against many nations, many of whom formed alliances to attempt to withstand the Roman conquest. In the end, the Roman empire was one of if not the most powerful empire in history.

When Rome annexed a settlement or nation, it forced its population to either join its empire as citizens or slaves, boosting the immense political and economic growth of the empire by sheer manpower and feeding its ever growing armies with fresh soldiers.

Rome's armies were formed out of mainly the legionaries. These were soldiers specially trained by the Roman authority and on the Roman authority's dime. This was both expensive and effective, making the control and distribution of resources crucial.

Cavalry and archers were also integrated into the army to fulfill important roles on the battlefield and during the later parts of the conquests, Balearic slingers were employed as they were more effective in ranged combat than archers.

This composition of the army paired with the immense skill and motivation of the Roman legionaries, made for a ferocious foe and not many nations could defend itself effectively against the Roman war machine, leading to the widespread Roman empire.

The Roman empire, during its conquests in the middle east and asia, also faced threats at home, with the Germanic tribes and settlers regularly attacking the outskirts of the empire, which was regularly left undefended by the conquesting legions.

One big issue of the large legions of the Roman empire was resources. The cost of upkeep for an army of thousands of soldiers was immense and if these armies did not get the required resources, they would regularly lose many soldiers en route to the battlefields, especially while marching through rough terrain, like deserts and mountains.

Each of these elements will be implemented in the simulation, by use of design patterns mentioned and explained below.

This milieu was chosen due to its simplicity in military entities. Here we needn't worry about airborne or oceanic warfare nor politics, as the states of those times were almost entirely controlled by the monarch or ruling entity.
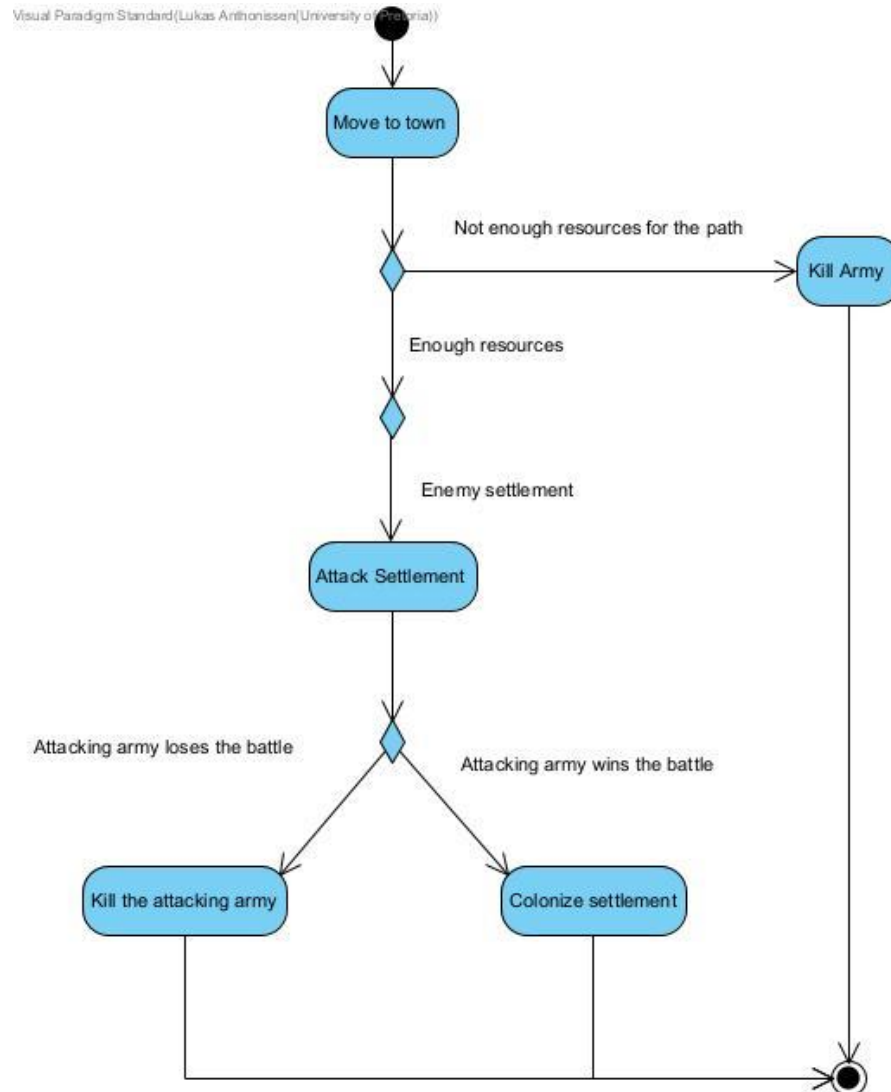
Settlements were also spread further apart, allowing us to use graphs to model the battlefield, rather than grids.
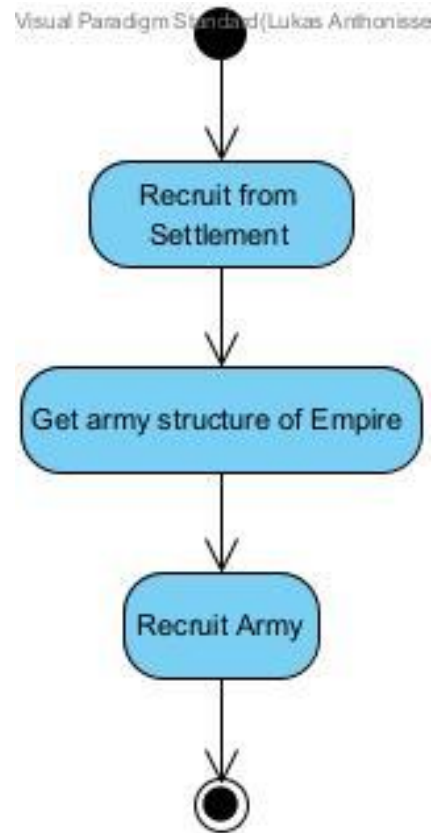
Functional Requirements

- A way for Armies to capture enemy settlements.
- A way for Armies to travel between towns and die along the way if they do not have sufficient resources.
- A way for Empires to form alliances.
- A way for Empires to differ in the way they structure their armies
- A way for armies to be informed if their immediate allied towns are attacked.
- A way for settlements to inform surrounding settlements if they are attacked
- A way for settlements to recruit soldiers from their population.
- A way to switch between different stages of war for each Empire.

Activity Diagrams

The turn of each army when it marches to a new node.

Move to town

Not enough resources for the path → Kill Army

Enough resources

Enemy settlement

Attack Settlement

Attacking army loses the battle

Attacking army wins the battle

Kill the attacking army

Colonize settlement

The recruitment process of an Empire

Recruit from Settlement

Get army structure of Empire

Recruit Army

Design Patterns and data-structures:

First off, we decided to use an undirected graph as a map, whereon our war will be unfolding. The nodes of the graph represent settlements and the edges represent paths between the edges

Example graph below:
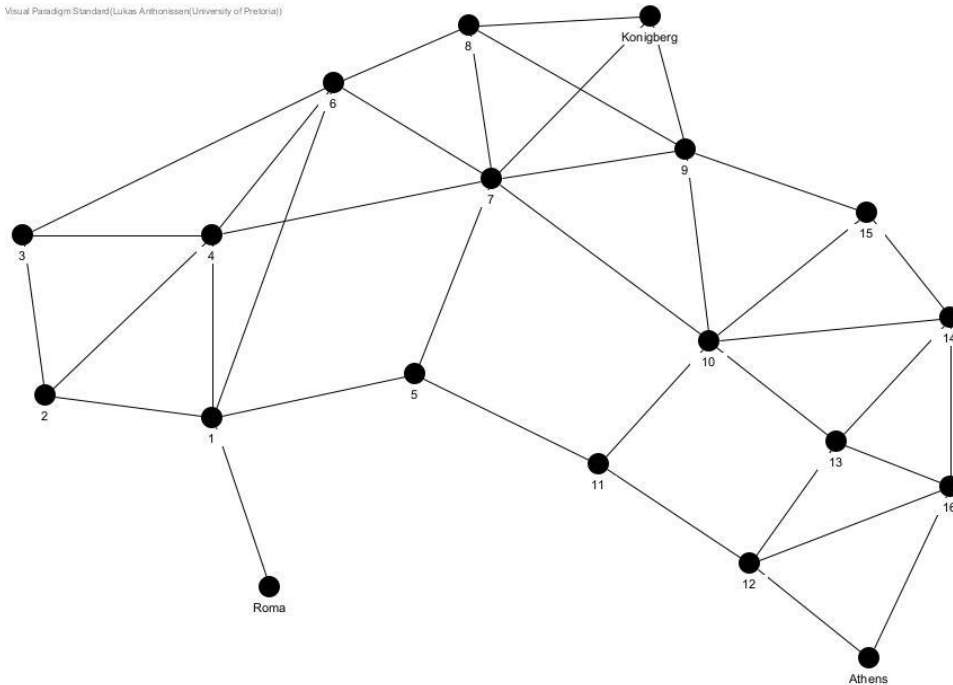
Here, Roma, Athens, and Konigsberg are the capitals of the 3 distinct empires on this map. Each number represents a settlement owned by an empire.

Template Method:

To have different terrain types and for them to function slightly differently, we will use the Template method. The purpose of the Template Method is to show the army size (soldier deaths) and resources changes that are happening while moving over a path. Each path goes through a specific terrain type and therefore each terrain type affects the army size and its resources differently.

Classes related to the Template Method:

1) Client: Path
2) AbstractClass: TerrainConcreteClass: Desert, Forest, Mountains, Plains, Snow

The Template Method used: calculate_losses(Army *army)
See Object diagram below:
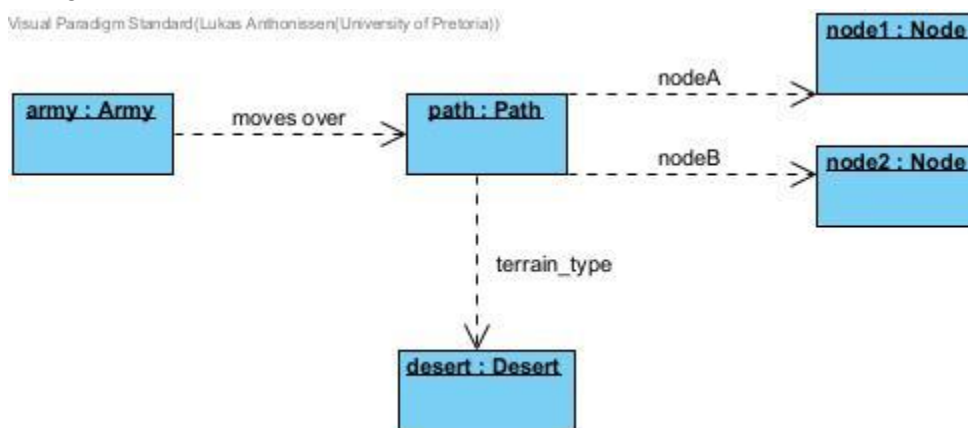
Iterator:

The iterator is used to iterate through the surrounding nodes of a certain town. Since the iterator has a unique goal it is the only pattern that we could have used for it and we did not consider any other design patterns. Towns (Nodes) are connected with paths. The iterator then gets all the endpoints of all the connected paths to a specific town (node). The endpoints will be all the adjacent towns of a specific town. Classes related to the iterator:

1) Aggregate: Aggregate
2) ConcreteAggregate: Node
3) Iterator: Iterator
4) ConcreteIterator: NodeIterator

The iterator has 4 methods: first, next, isDone, getCurrentNode. first() set the current node to the first adjacent element. next() set current node to the next adjacent node. isDone() check to see if all adjacent nodes were visited. getCurrentNode() return the current node. The iterator uses the paths connected to the node and it gets the opposite node of all the paths in the path vector.

Observer:

For an Army to be able to return to its previous position once that position is attacked, without taking another turn, we will use the Observer pattern. The army will function as an observer on the node, which is the subject, notifying all observers to return to the node, this is achieved with the moveToTown method.

We used an Observer design pattern over the Mediator to avoid the Mediator becoming a "god object" where it references a large number of distinct types or has too many unrelated methods as this project is very complex with the different methods. Even though the mediator reduces coupling and you can use the individual components more easily the Observer is already used in conjunction with the Mediator with the onAttacked method and we need to notify all the armies at that moment of time which is made easy with the list of armies, making the observer the perfect design pattern for this problem.

Even though the observer notifies the armies in a random order it doesn't matter in this case as it doesn't interfere with the running of the war and achieves the goal of making them return to the node being attacked.
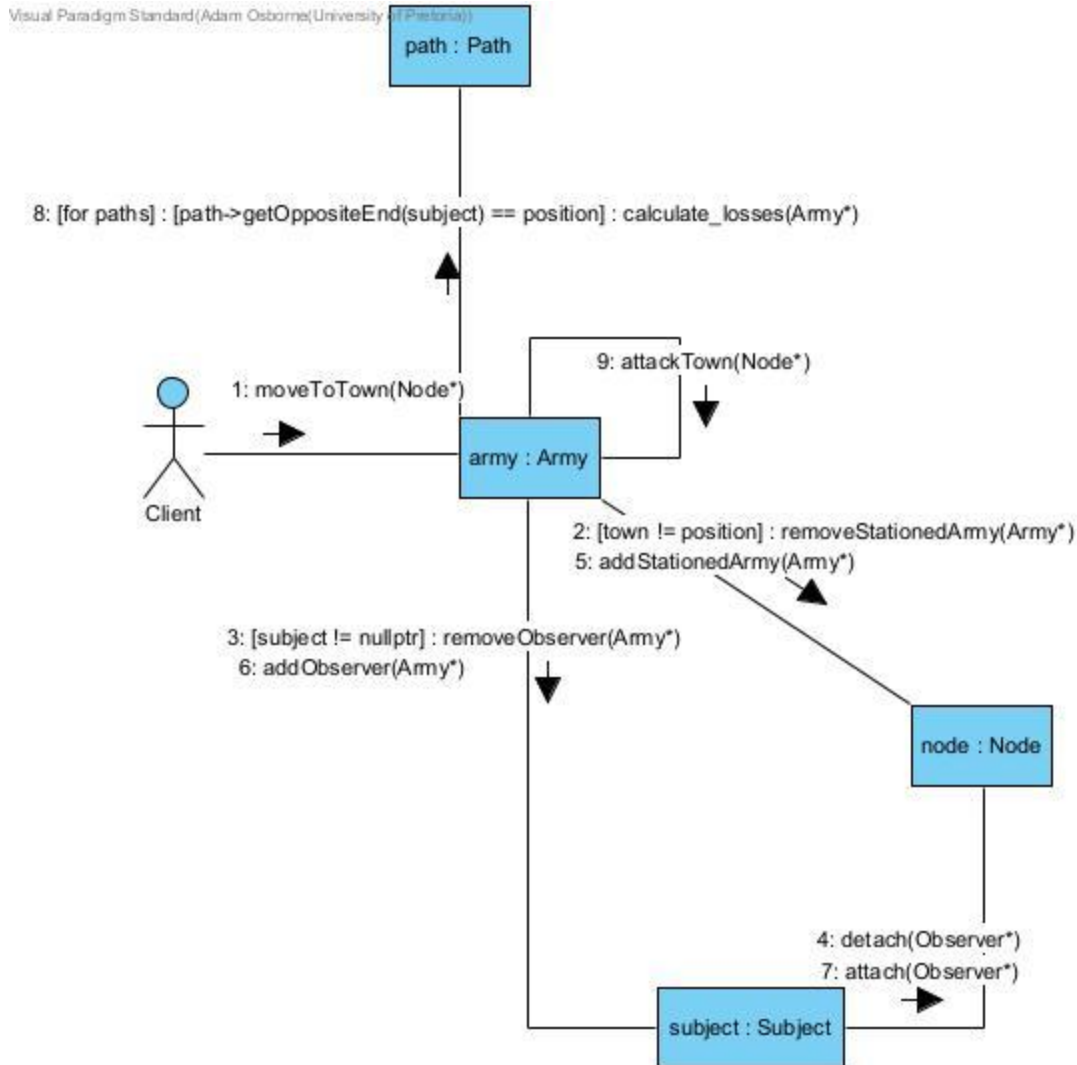
Participants:
Observer: Observer
Subject: Subject
Concrete Observer: Army
Concrete Subject: Node

Observer Communication Diagram:

path : Path

8: [for paths] : [path->getOppositeEnd(subject) == position] : calculate_losses(Army*)

9: attackTown(Node*)

1: moveToTown(Node*)

Client

army : Army

2: [town != position] : removeStationedArmy(Army*)
5: addStationedArmy(Army*)

3: [subject != nullptr] : removeObserver(Army*)
6: addObserver(Army*)

node : Node

4: detach(Observer*)
7: attach(Observer*)

subject : Subject

Mediator:

The mediator will be used to notify all the adjacent towns in the same empire that there is a town under attack. When a settlement is under attack a mediator object gets created which notifies all surrounding allied towns of an attack, so they get time to prepare their defenses. When a town gets notified it will call the onAttacked method, which will recruit new armies. Other patterns that we considered were the observer. The observer pattern distributes communication through Observer and Subject objects where the Mediator centralizes the communication between objects. We decided to use the mediator for notifying because the town is already connected by paths and it is not necessary to register observers and it will never have to deregister. Classes related to the mediator:

     1)   Mediator: Mediator
     2) ConcreteMediator: ConcreteMediator
     3) Colleague: Node
     4) ConcreteColleague: Node

See the sequence diagram below:
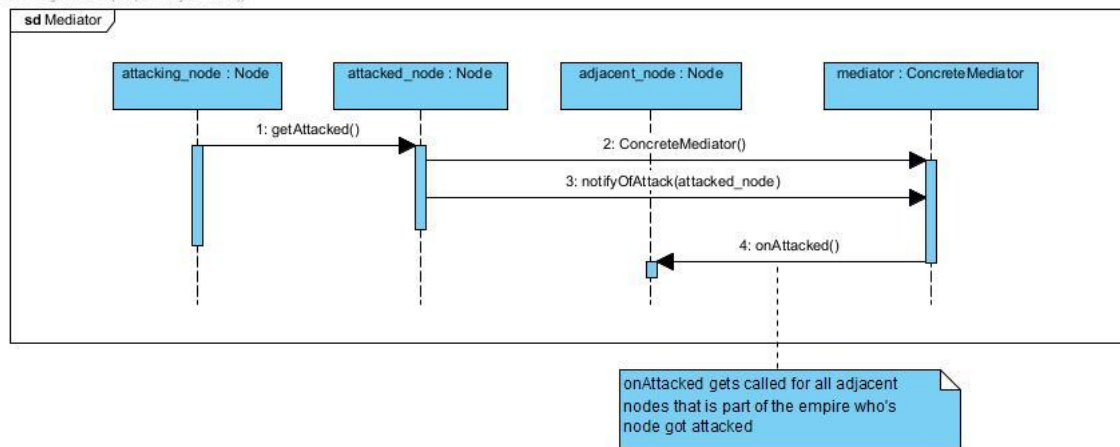
Factory:

To enable the settlements to create different types of soldiers and in the end an army, we will use the Factory method. The settlement will create a factory to create different units to add to an Army object. We used the Factory method instead of the Abstract Factory, since we don't need to create different types of each unit subtype. The Factory Method is just one method to override rather than a total object.
The different troops our Factory creates is:

1. Archer
2. Footmen
3. Slinger
4. Cavalry

Classes related:

1. Creator: Barracks
2. ConcreteCreator: SlingerBarracks
3. ConcreteCreator: ArcheryBarracks
4. ConcreteCreator: FootmenBarracks
5. ConcreteCreator: CavalryBarracks
6. ConcreteProduct: Slinger
7. ConcreteProduct: Archer
8. ConcreteProduct: Footmen
9. ConcreteProduct: Cavalry

In execution, a Node will create a factory object for whatever unit is needed and create units using said factory. Afterwards, the factories will be deleted and the units returned.
Each time the factory is called one unit of that type of troop is returned.

See sequence diagram below:
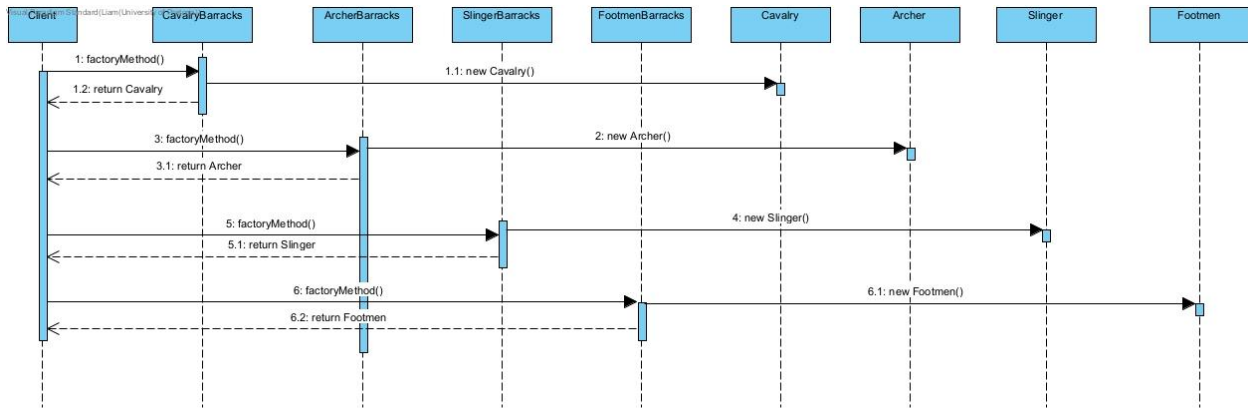
Command:

The Command pattern is used for the requesting and accepting of alliances in the war. We decided to use the Command for this because the Command's intent is encapsulating requests as objects. Because the Command pattern's intent is very unique we did not consider any other design patterns to use instead of the Command. Classes related to the Command:

1) Command: Communication
2) ConcreteCommand: RequestAlliance, AcceptAlliance
3) Invoker: Empire
4) Receiver: Empire

When an empire wants to form an alliance it will send a "request" command to the other empire. The empire that receives the alliance request will first check to see if they will benefit from the alliance, if they will then they send an "accept" command to the empire that requested to form the alliance, else they do not send anything. The following is checked when the empire is considering the alliance: Amount of current alliances, amount of resources, amount of armies, and the amount of resources per army. If the empire falls short in any of these, then they will accept the alliance.

See the sequence diagram below:
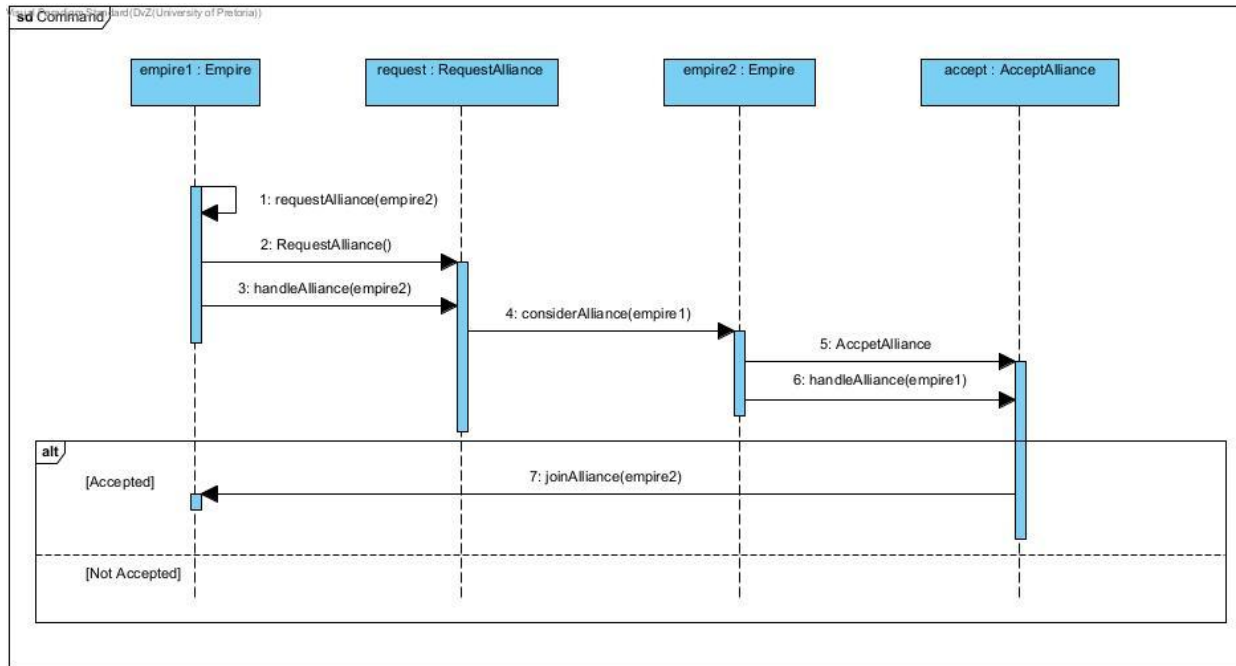
Strategy:

To enable different strategies, we will use the Strategy design pattern to enable the policies of different empires to differ regarding the number of recruits per settlement and the structure of the recruited armies. Two classes will have their own cases of the Strategy pattern. There is the RecruitmentPolicy which returns the number of recruits based on the army size and the WarStylePolicy returns a different composition of the army. Classes related to the Strategy:
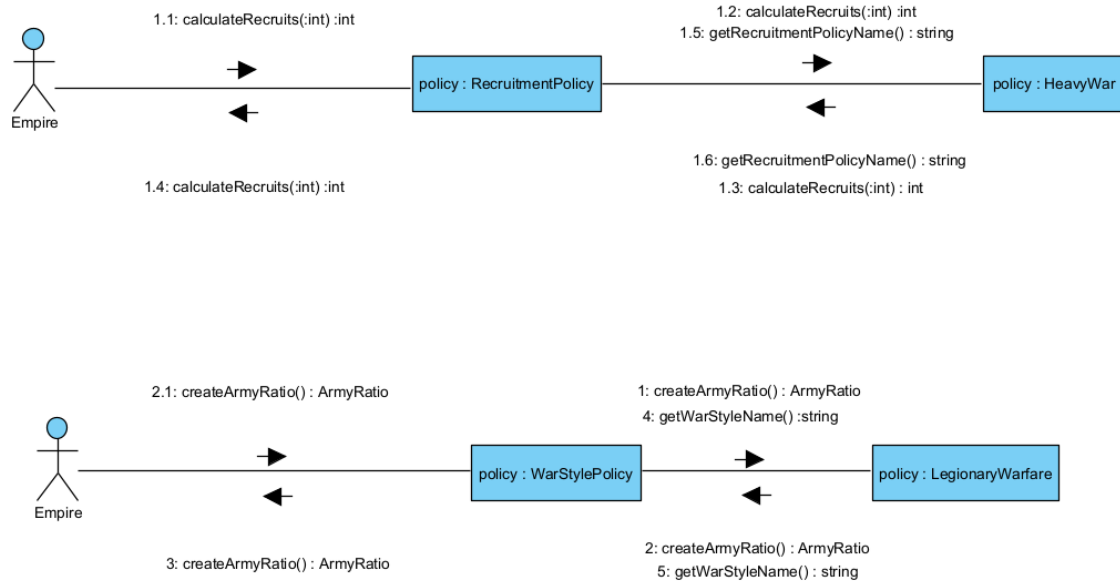Case 1:
    1) Context: Empire
    2) Strategy: RecruitmentPolicy
    3) ConcreteStrategy: LightWar, HeavyWar, NormalWar
Case 1:
    1) Context: Empire
    2) Strategy: WarStylePolicy
    3) ConcreteStrategy: HorsmenWarfare, LegionaryWarfare, ArcherWarfare, GuerillaWarfare

See Communication diagram below:

State Pattern:

In war, there are many stages of the war. In the start, the empires will try to annex or "colonize" as much land as possible. If put in a defensive position, the empire will try to defend its settlements instead of attacking the enemies.

To simulate these war stages [Attacking, Defending etc.], we will use the State design pattern. We used the State pattern over the Strategy, because the State holds a link to its context, while the Strategy doesn't. This allows the State to trigger certain methods of the Empire and, most importantly, to swap itself out for another State; e.g. an Attack state will notice it needs to retreat its armies, so it will swap to a Defend state.

See the state diagram below:

Not enough settlements lost

Attack

No settlements lost in the previous turn

Enough Settlements lost in the last turn

Defend

More than 0 settlements lost
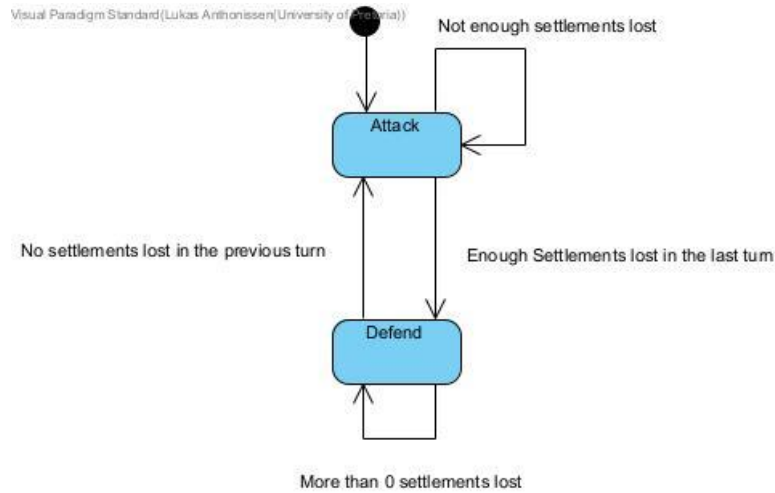
The WarStage will also function as the main controlling algorithm of the Empire. Meaning it will tell the Empire to advance its armies or recruit armies etc.

See the sequence diagram below:

gameLoop | empire | war_stage

1: takeTurn()

2: takeTurn()

3: recruitArmies()

alt

[Attack]

4: advanceArmies()

alt

[Defend]

5: retreatArmies()

6: requestAlliances()
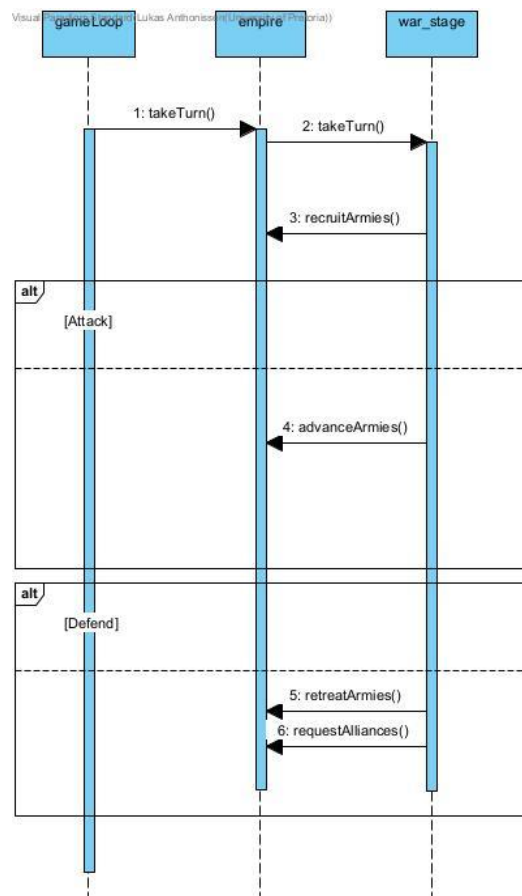
The Empire will be the context and classes like Attack and Defend will be ConcreteStates.

A state is also used in the NodeType class. A NodeType may be a Town or Capital. A Node will own such a class to determine how it will react when colonized. If a capital gets colonized, its owning empire will collapse and will be deleted from the war.

If a town gets colonized, it will merely be owned by the colonizing empire.

Memento: (See object diagram on next page)

The memento design pattern is being used in order to store the state of the war at a certain point in time, providing the capability to restore the state of the war to one of the stored states. (Undoable operations).
The memento makes use of the prototype design pattern in order to clone objects. This design choice was made because of the complexity and many relationships between the objects that are the state of the war.
This includes many references between capitals, towns, empires, paths etc. Since the state of the war also consists of graphs that have to be deep-copied, some modifications had to be made to the clone() methods of the prototype design pattern. (See Prototype for a more in-depth explanation).

Participants:
Memento: WarRollback
Client: main.cpp
Caretaker: WarCaretaker
Originator: War

Prototype: (See object diagram on next page)

We decided to use the prototype because of the complexity of the objects being cloned.
This makes the prototype pattern much more feasible compared to the factory method design pattern. Using a different design pattern would result in an immense and unnecessary increase in complexity.
All entities/objects that are a part of the war have a clone() method. The clone method had to be modified slightly, the function signature has been changed to:
clone(std::map<void*, void*> objmap)
The new parameter is a map of pointers to pointers, specifically the pointer to the objects being cloned and the corresponding new object created in its stead. This has been done in order to support the deep-copying of graphs and maintain the structure of all original objects exactly.
When the clone method gets called on the object, it firstly checks if the object being cloned is already contained in the map. If the object is already contained in the map, it simply returns the corresponding cloned object from the map. If it is not contained in the map, it creates a new object of the same type before calling clone() on all objects that this object has reference to, and finally adding it to the map.
Clone is recursive and will call .clone on all objects contained in itself.

This allows for the clone method to be invoked multiple times on the same object and only clone the object the first time that clone() is invoked. If it gets "cloned" again in the future, with the same objectmap being used, it will simply return the existing clone.
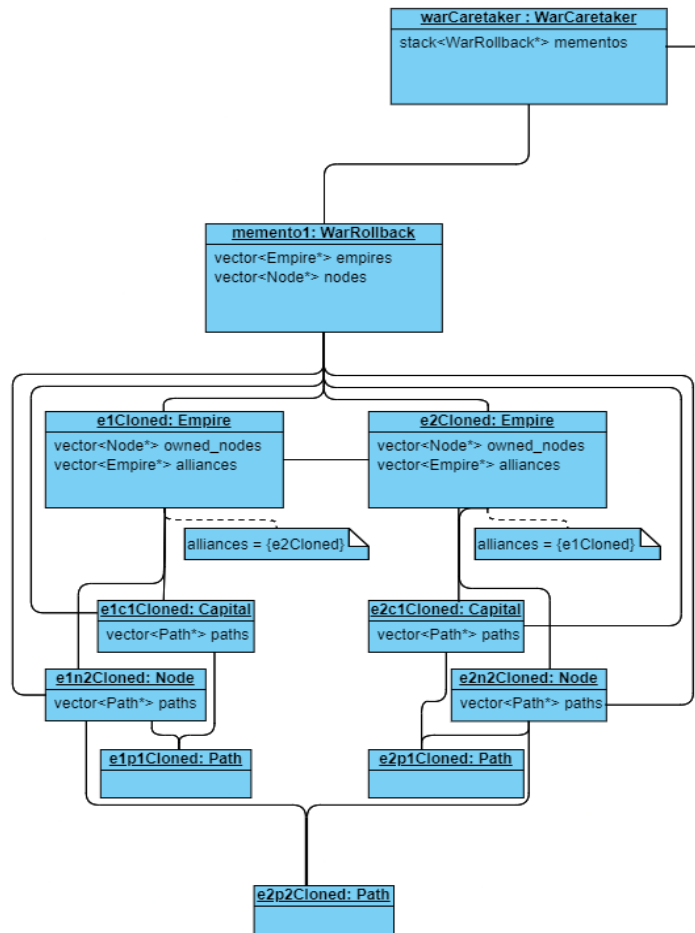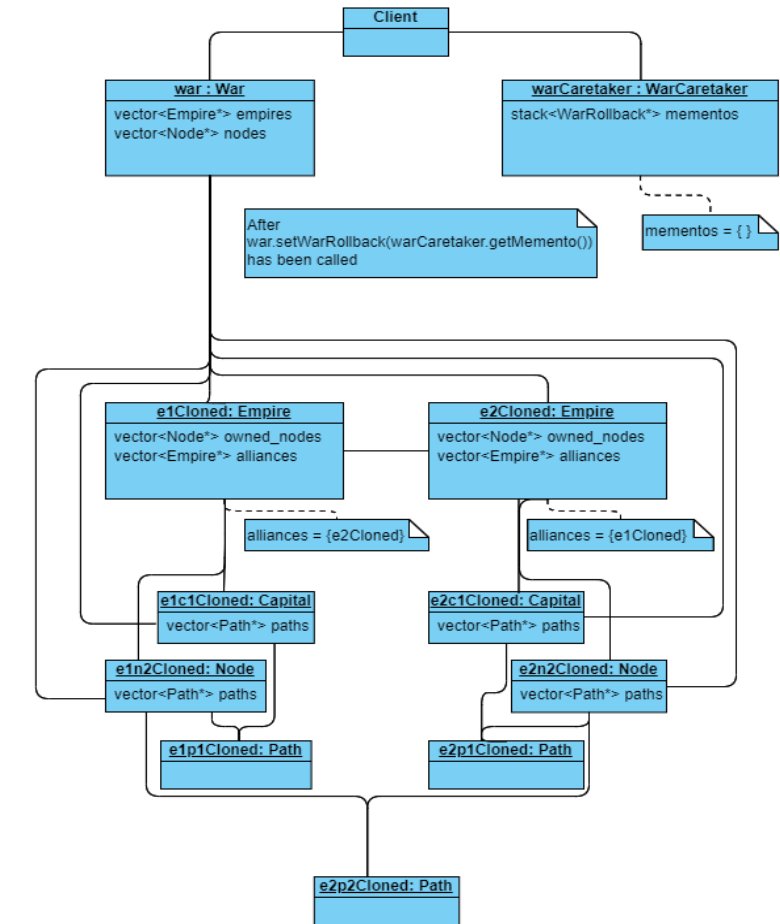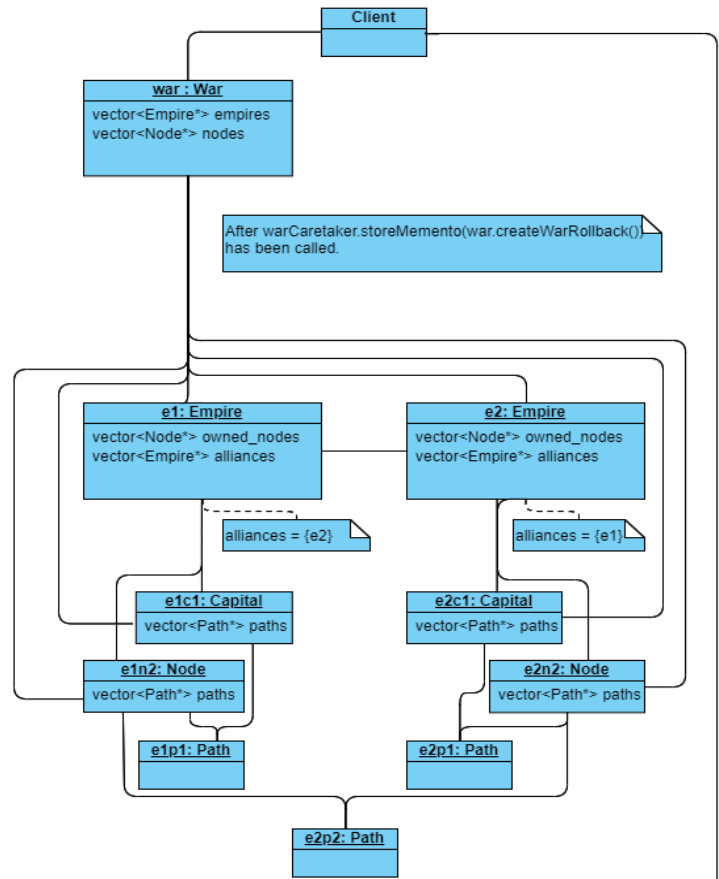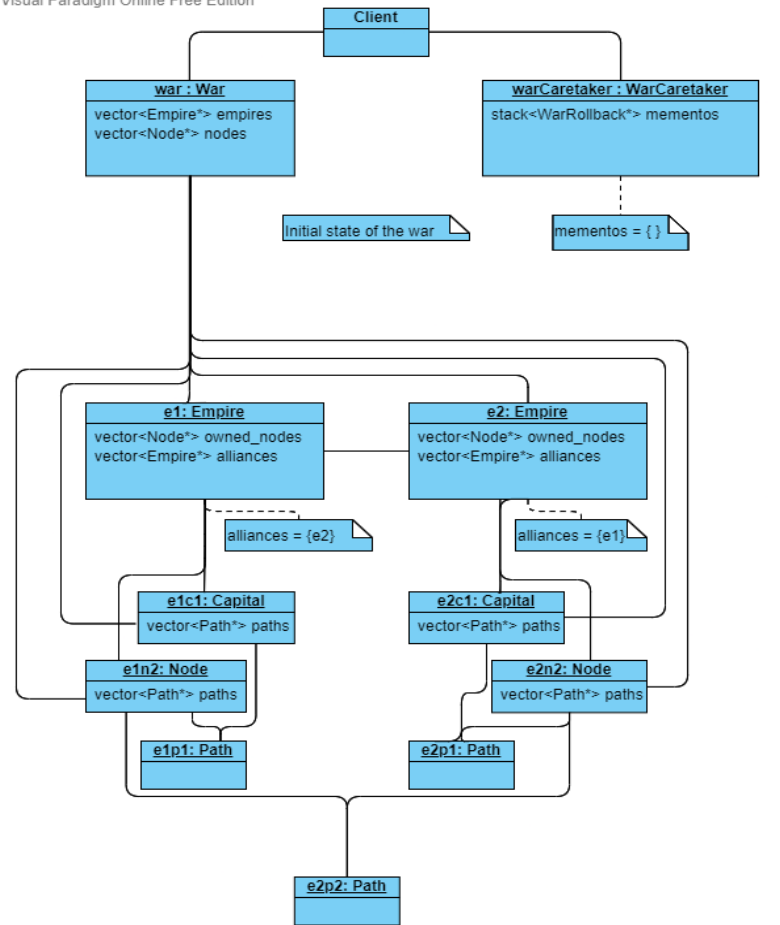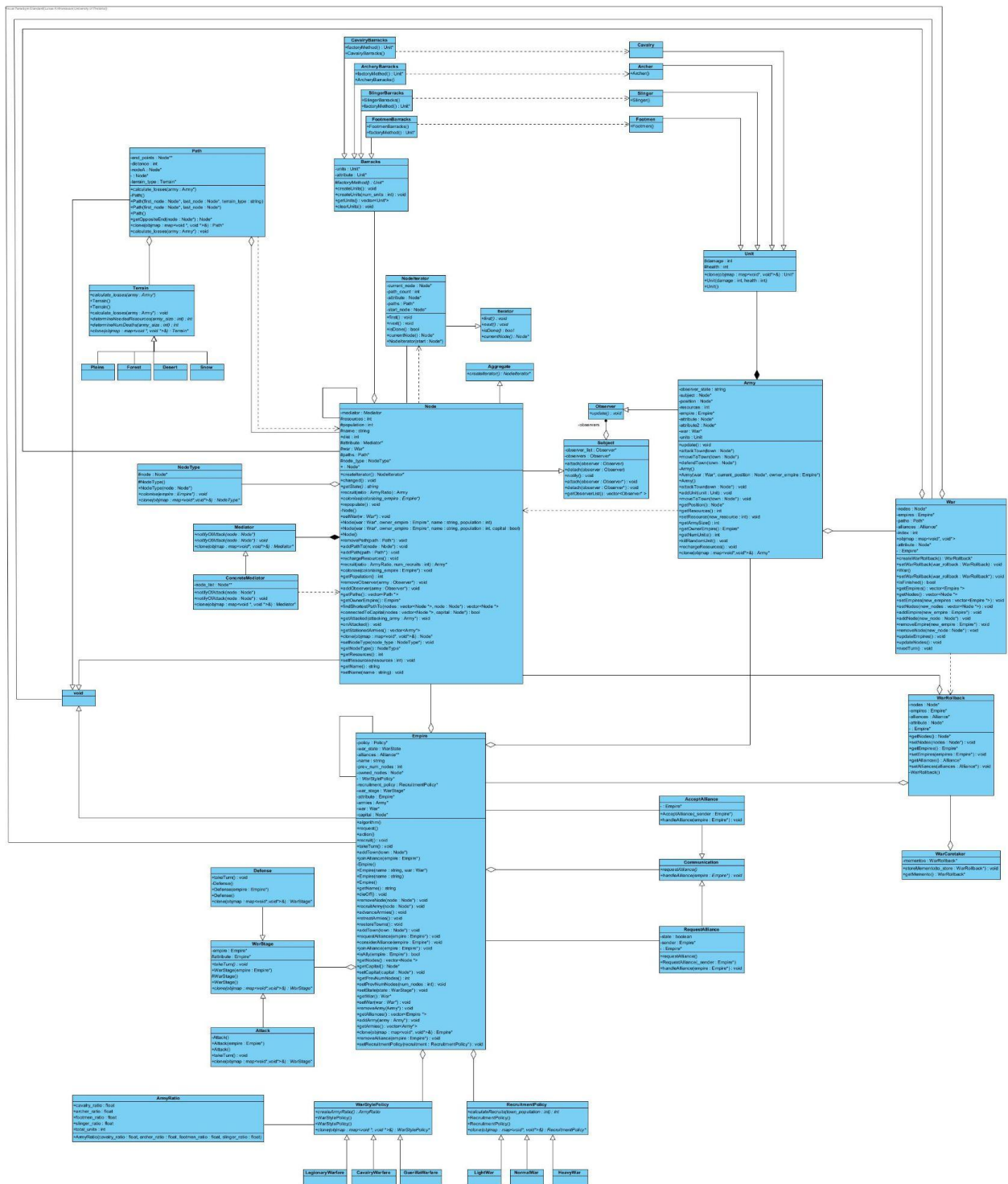
Participants:
Client: WarRollback
Prototype: Terrain
ConcretePrototypes: Snow, Mountains, Desert, Forest, Plains

See Object Diagram below

Top-left diagram:

Client

war : War
vector<Empire*> empires
vector<Node*> nodes

warCaretaker : WarCaretaker
stack<WarRollback*> mementos

Initial state of the war

mementos = { }

e1: Empire
vector<Node*> owned_nodes
vector<Empire*> alliances

e2: Empire
vector<Node*> owned_nodes
vector<Empire*> alliances

alliances = {e2}

alliances = {e1}

e1c1: Capital
vector<Path*> paths

e2c1: Capital
vector<Path*> paths

e1n2: Node
vector<Path*> paths

e2n2: Node
vector<Path*> paths

e1p1: Path

e2p1: Path

e2p2: Path

Top-right diagram:

Client

war : War
vector<Empire*> empires
vector<Node*> nodes

After warCaretaker.storeMemento(war.createWarRollback()) has been called.

e1: Empire
vector<Node*> owned_nodes
vector<Empire*> alliances

e2: Empire
vector<Node*> owned_nodes
vector<Empire*> alliances

alliances = {e2}

alliances = {e1}

e1c1: Capital
vector<Path*> paths

e2c1: Capital
vector<Path*> paths

e1n2: Node
vector<Path*> paths

e2n2: Node
vector<Path*> paths

e1p1: Path

e2p1: Path

e2p2: Path

warCaretaker : WarCaretaker
stack<WarRollback*> mementos

memento1: WarRollback
vector<Empire*> empires
vector<Node*> nodes

e1Cloned: Empire
vector<Node*> owned_nodes
vector<Empire*> alliances

e2Cloned: Empire
vector<Node*> owned_nodes
vector<Empire*> alliances

alliances = {e2Cloned}

alliances = {e1Cloned}

e1c1Cloned: Capital
vector<Path*> paths

e2c1Cloned: Capital
vector<Path*> paths

e1n2Cloned: Node
vector<Path*> paths

e2n2Cloned: Node
vector<Path*> paths

e1p1Cloned: Path

e2p1Cloned: Path

e2p2Cloned: Path

Bottom-left diagram:

Client

war : War
vector<Empire*> empires
vector<Node*> nodes

warCaretaker : WarCaretaker
stack<WarRollback*> mementos

After war.setWarRollback(warCaretaker.getMemento()) has been called

mementos = { }

e1Cloned: Empire
vector<Node*> owned_nodes
vector<Empire*> alliances

e2Cloned: Empire
vector<Node*> owned_nodes
vector<Empire*> alliances

alliances = {e2Cloned}

alliances = {e1Cloned}

e1c1Cloned: Capital
vector<Path*> paths

e2c1Cloned: Capital
vector<Path*> paths

e1n2Cloned: Node
vector<Path*> paths

e2n2Cloned: Node
vector<Path*> paths

e1p1Cloned: Path

e2p1Cloned: Path

e2p2Cloned: Path

Final UML-diagram:

The above UML diagram is added as a pdf in the root folder.

Sources:

https://www.britannica.com/place/Roman-Empire

https://en.wikipedia.org/wiki/Roman_Empire

https://www.rome.net/roman-empire

https://www.khanacademy.org/humanities/world-history/ancient-medieval/roman-a/a/roman-empire