Practical 2

COS214

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA
Denkleiers • Leading Minds • Dikgopolo tša Dihlalefi

Department of Computer Science

Deadline: 11/08/2024 at 23:59

Marks: 120

1 General Instructions

- This assignment should be completed in pairs.
- Be ready to upload your practical well before the deadline, as no extensions will be granted.
- You can use any version of C++.
- You can import the following libraries:
  - vector
  - string
  - iostream
  - map
  - list
- You will upload your code with your main to FitchFork as proof that you have a working system.
- **If you meet the minimum FitchFork requirements (working code with at least 60% testing coverage), you will be marked by tutors during your practical session. Please book in advance (Instructions will follow on ClickUp).**
- **You will not be allowed to demo if you do not meet the minimum FitchFork requirements.**

2 Plagiarism

The Department of Computer Science considers plagiarism a serious offence. Disciplinary action will be taken against students who commit plagiarism. Plagiarism includes copying someone else's work without consent, copying a friend's work (even with permission) and copying material from the Internet. Copying will not be tolerated in this course. For a formal definition of plagiarism, the student is referred to http://www.library.up.ac.za/plagiarism/index.htm. **If you have any questions regarding this, please ask one of the lecturers to avoid misunderstanding.**

3 Mark Distribution

Activity	**Mark**
Task 1: UML Diagrams	30
Task 2: Implementation of Patterns	60
Task 3: FitchFork Testing Coverage	10
Task 4: Demo Preparation	20
Total	**120**

Table 1: Mark Distribution

4 Background

Prologue: The Call of Destiny

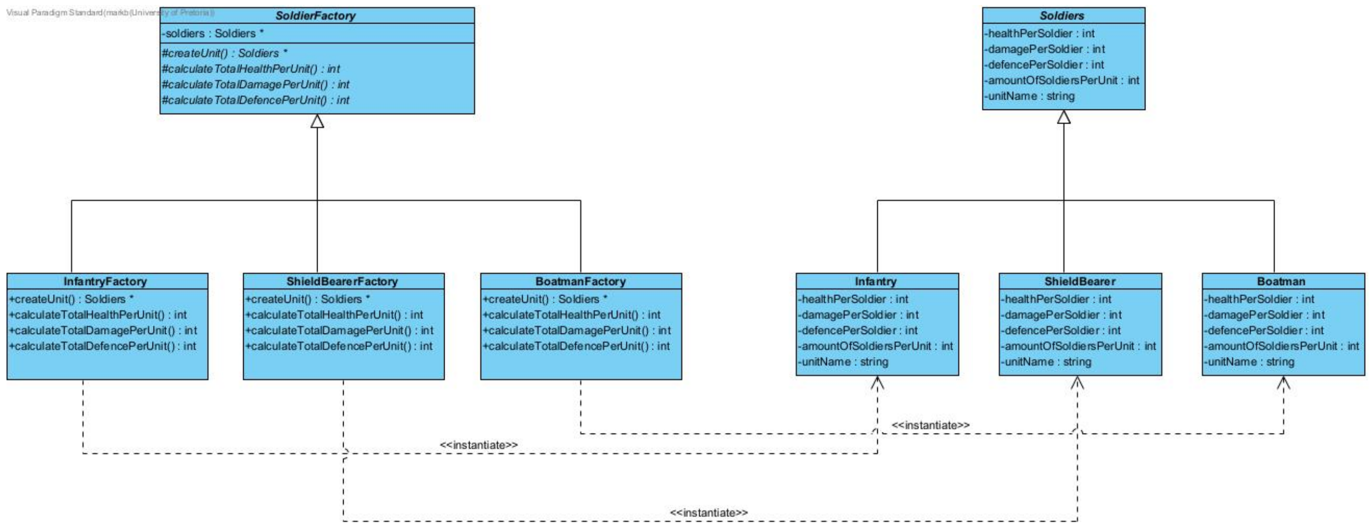
In the year 100 BC, Rome is a republic on the brink of empire. To the north, hostile barbarian tribes threaten the borders. To the east, the vast realms of Parthia eye Rome warily. Amidst political chaos, a young Roman legionnaire, Titus Flavius, rises through the ranks, destined to shape the future of the mightiest army the world has ever seen.

As Titus rises through the ranks in a time of war and expansion, you are tasked with strategically building and managing the Roman legions using advanced military systems. Your tools are the implementations of the Factory Method, Template Method, Memento, and Prototype design patterns.

Read the following sections carefully and complete the tasks that follow.

4.1 Factory Method

As Titus encounters different war zones, choosing the right factory allows for tailored army compositions that meet specific battlefield needs.


Visual Paradigm Standard(markb(University of Pretoria))
SoldierFactory
-soldiers : Soldiers *
#createUnit() : Soldiers *
#calculateTotalHealthPerUnit() : int
#calculateTotalDamagePerUnit() : int
#calculateTotalDefencePerUnit() : int
Soldiers
-healthPerSoldier : int
-damagePerSoldier : int
-defencePerSoldier : int
-amountOfSoldiersPerUnit : int
-unitName : string
InfantryFactory
+createUnit() : Soldiers *
+calculateTotalHealthPerUnit() : int
+calculateTotalDamagePerUnit() : int
+calculateTotalDefencePerUnit() : int
ShieldBearerFactory
+createUnit() : Soldiers *
+calculateTotalHealthPerUnit() : int
+calculateTotalDamagePerUnit() : int
+calculateTotalDefencePerUnit() : int
BoatmanFactory
+createUnit() : Soldiers *
+calculateTotalHealthPerUnit() : int
+calculateTotalDamagePerUnit() : int
+calculateTotalDefencePerUnit() : int
Infantry
-healthPerSoldier : int
-damagePerSoldier : int
-defencePerSoldier : int
-amountOfSoldiersPerUnit : int
-unitName : string
ShieldBearer
-healthPerSoldier : int
-damagePerSoldier : int
-defencePerSoldier : int
-amountOfSoldiersPerUnit : int
-unitName : string
Boatman
-healthPerSoldier : int
-damagePerSoldier : int
-defencePerSoldier : int
-amountOfSoldiersPerUnit : int
-unitName : string
<<instantiate>>
<<instantiate>>
<<instantiate>>

4.1.1 Classes and Attributes:

SoldierFactory (Abstract Base Class):

Attributes:

- soldiers: Soldiers*

Methods:

- createUnit() : Soldiers* - Abstract method to be implemented by concrete factories.

InfantryFactory, ShieldBearerFactory, BoatmanFactory (Concrete Factory Classes):

Methods:

- createUnit() : Soldiers* - Concrete implementation that instantiates and returns an object of the respective soldier type (Infantry, ShieldBearer, Boatman).
- calculateTotalHealthPerUnit() : int
- calculateTotalDamagePerUnit() : int
- calculateTotalDefencePerUnit() : int

These methods should calculate the respective totals based on the number of soldiers in a unit and their individual health, damage, and defence attributes.

Soldiers (Abstract Product Class):

Attributes:

- healthPerSoldier : int
- damagePerSoldier : int

- defencePerSoldier : int
- amountOfSoldiersPerUnit : int
- unitName : string

Attributes to hold the general characteristics of a soldier.

Infantry, ShieldBearer, Boatman (Concrete Product Classes):
Inherit from Soldiers No additional methods or attributes are specified in the diagram beyond what is inherited from Soldiers.

4.1.2 Implementation Details:

Factories:
Each factory class should have methods to instantiate its corresponding soldier type. For instance, InfantryFactory's createUnit() would create and return an instance of Infantry.

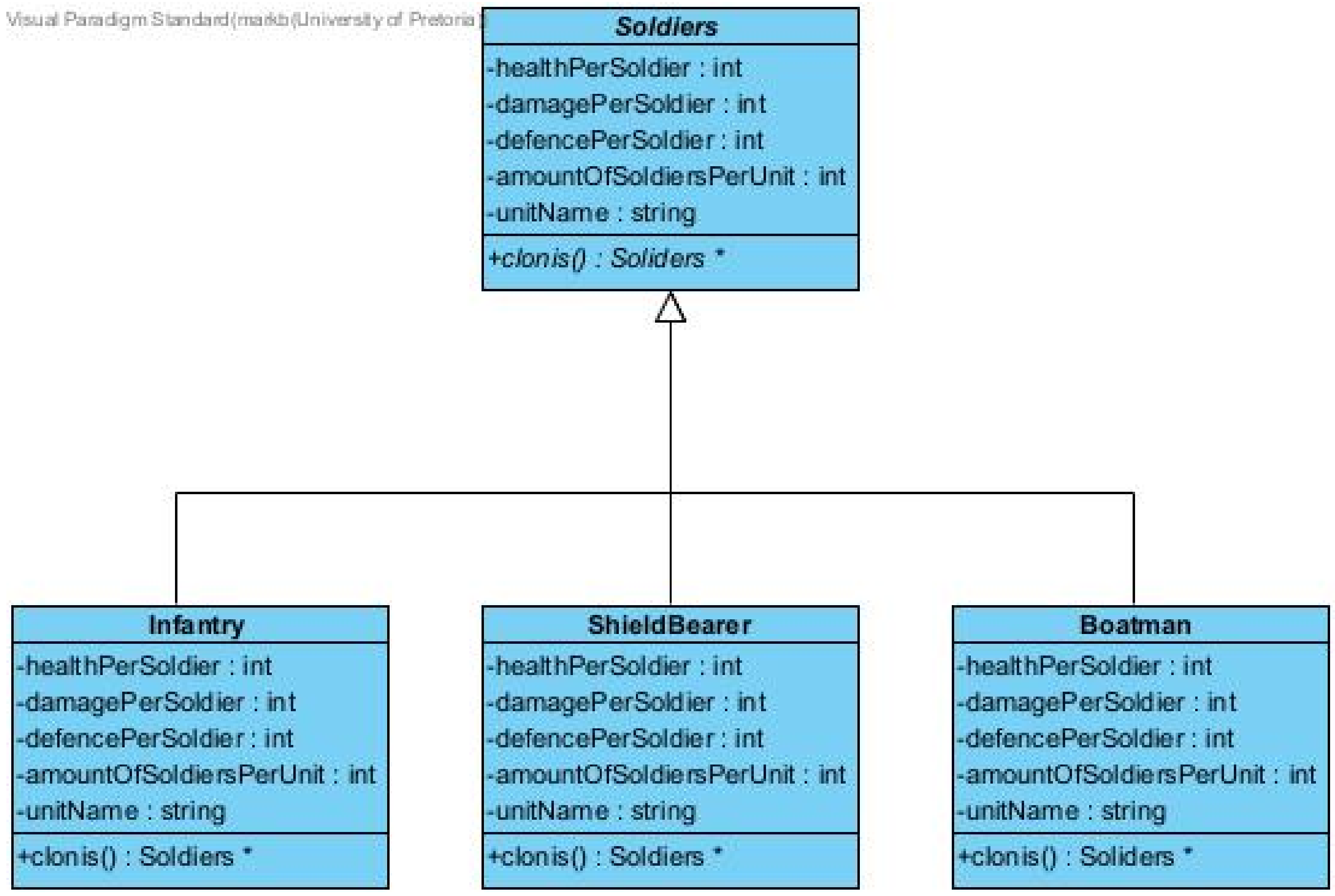
The methods for calculating total health, damage, and defense should sum up the respective attributes multiplied by the number of soldiers per unit. These could be used to display statistics or make calculations for battle simulations.

Soldier Types:

Implement the concrete soldier classes with their specific characteristics. For example, an Infantry unit might have high defense but lower damage compared to a Boatman which might have higher attack capabilities suited for different strategic needs.

4.2 Prototype

After a significant victory, Titus uses the spoils to clone elite units, swiftly bolstering his forces without waiting for new recruits.



4.2.1 Classes and Methods:

Soldiers (Abstract Class):

Attributes:

- healthPerSoldier : int
- damagePerSoldier : int
- defencePerSoldier : int
- amountOfSoldiersPerUnit : int
- unitName : string

Methods:

- clonis() : Soldiers* - Abstract method for cloning soldier objects.

Infantry, ShieldBearer, Boatman (Concrete Classes):

- Each class inherits all attributes from the Soldiers class.
- Each class implements the +clonis() : Soldiers* method, enabling the creation of new soldier instances with identical attributes.

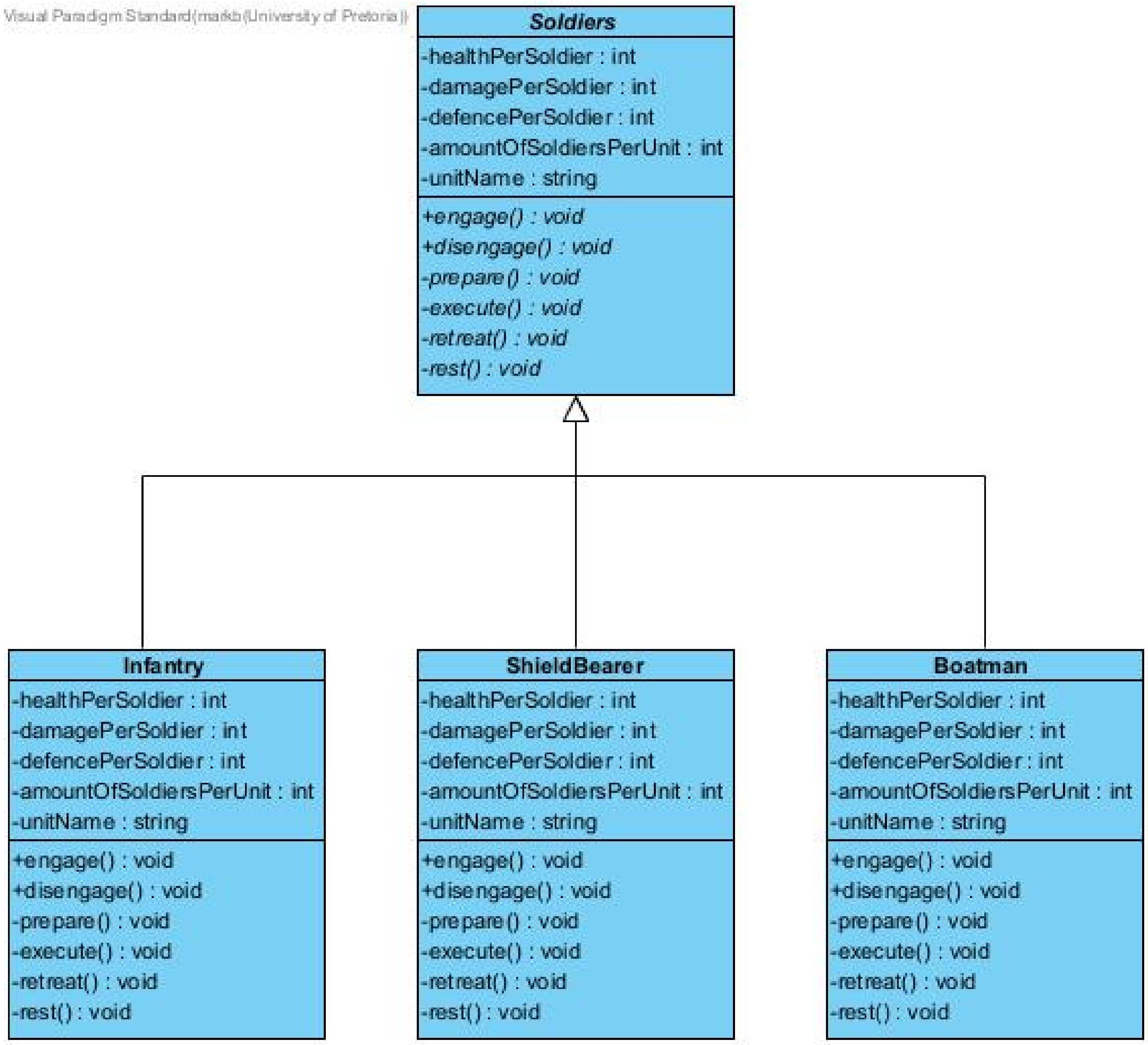
4.2.2 Implementation Tasks:

Clone Method Implementation:

Each concrete soldier class must implement the clonis() method. This method should create a new instance of the respective class and copy all attribute values from the this object to the new object, ensuring a deep copy if needed.

4.3 Template Method:

As Titus commands diverse legions, each type of soldier must respond uniquely to the commands of attack and retreat, yet follow a general pattern of behavior.



4.3.1 Classes and Methods:

Soldiers (Abstract Class):

Attributes:

- healthPerSoldier : int
- damagePerSoldier : int
- defencePerSoldier : int

- amountOfSoldiersPerUnit : int
- unitName : string

Methods:

- +engage(): void - A template method that outlines the sequence of engaging in battle. Calls abstract methods prepare() and execute().
- +disengage(): void - A template method for retreating. Calls abstract methods retreat() and rest().
- Abstract methods: +prepare(): void, +execute(): void, +retreat(): void, +rest(): void - These methods will be implemented differently by each concrete class.

Infantry, ShieldBearer, Boatman (Concrete Classes):

- Each class inherits from Soldiers.
- Implements all abstract methods defined in Soldiers to cater to their specific role in battle and retreat maneuvers.
- For instance, Infantry might implement prepare() to form tight defensive formations, while Boatman might prepare by maneuvering boats into strategic positions.

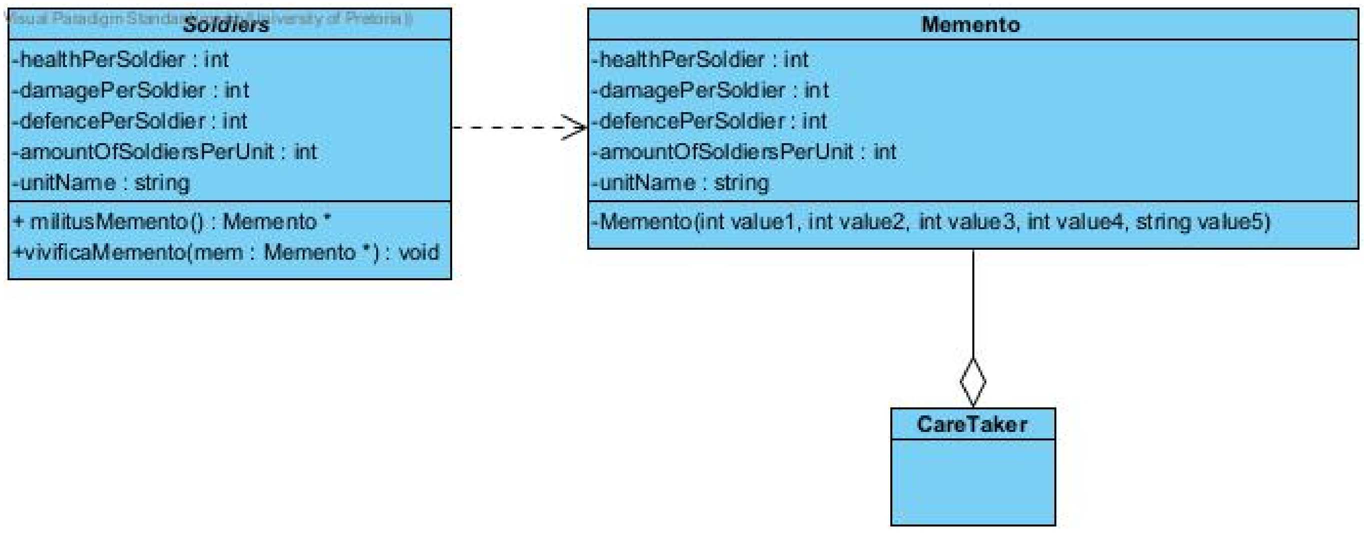
4.3.2 Implementation Tasks:

Each concrete soldier class (Infantry, ShieldBearer, Boatman) needs to provide their own implementation for prepare(), execute(), retreat(), and rest(). These implementations should reflect tactical maneuvers relevant to each type of soldier.

The engage() and disengage() methods in the Soldiers class should orchestrate the sequence of calling these abstract methods. The actual sequence (i.e., which methods are called and in what order during engagement and disengagement) should be common across all types but execute differently due to the overridden methods.

4.4 Memento:

Titus uses this system to refine his strategies, learning from past encounters to optimize future engagements.


Visual Paradigm Standard (University of Pretoria)
Soldiers
-healthPerSoldier : int
-damagePerSoldier : int
-defencePerSoldier : int
-amountOfSoldiersPerUnit : int
-unitName : string
+ militusMemento() : Memento *
+vivificaMemento(mem : Memento *) : void
Memento
-healthPerSoldier : int
-damagePerSoldier : int
-defencePerSoldier : int
-amountOfSoldiersPerUnit : int
-unitName : string
-Memento(int value1, int value2, int value3, int value4, string value5)
CareTaker

4.4.1 Classes and Responsibilities:

Soldiers (Abstract Class):

Attributes:

- healthPerSoldier : int
- damagePerSoldier : int
- defencePerSoldier : int
- amountOfSoldiersPerUnit : int
- unitName : string

Methods:

- +militusMemento() : Memento* - Creates a memento containing a snapshot of its current state.
- +vivificaMemento(mem : Memento*) : void - Restores its state from the memento object.

Memento:

Attributes: Mimics the attributes of Soldiers to hold the state snapshot. Constructor:

- -Memento(int value1, int value2, int value3, int value4, string value5) - Initializes the memento with the state of the Soldiers.

CareTaker:

Responsibility: Manages saving and restoring of Memento objects. This class can hold a stack, list, or another collection of mementos allowing the game to revert to any previous state or trace back multiple steps.

4.4.2 Implementation Tasks:

Define Memento and Caretaker Classes:

- Memento Class: Implement the constructor to take the current state of the Soldiers and store it. Ensure that all the attributes from Soldiers are accurately copied to preserve the state.
- CareTaker Class: Manage a collection of Memento objects. Implement methods to add new mementos to the collection and retrieve them based on the game's need (like undo functionality).

Implement Methods in Soldiers Class:

- militusMemento(): This method should create a new Memento object initialized with the current state of the Soldiers object.
- *vivificaMemento(mem : Memento)**: This method should take a Memento object and use it to restore the state of the Soldiers object.

5 Task 1: UML Diagrams

In this task, you need to construct UML Class and Object diagrams for the given scenario.

5.1 Class Diagram (20 marks)

- Create a comprehensive UML class diagram that shows how the classes in the scenario interact and participate in various patterns.
- Ensure it includes all relevant classes, their attributes, and methods.
- Identify and indicate the design patterns each class is involved in, as well as their roles within these patterns. (Remember a class can participate in more than one pattern)
- Display all relationships between classes.
- Add any additional classes that are necessary for a complete representation.

5.2 Object Diagram (10 marks)

- Develop a UML object diagram representing your system at a specific point in time (of your choosing).
- Include the relationships and attribute values of the objects.

- There should be objects of at least 3 different classes and they cannot all be part of the soldier hierarchy.

6 Task 2: Implementation

In this task, you need to implement the design patterns described in the scenario.

- Use your UML class diagram as a guide to integrate the design patterns. Note that a single class can be part of multiple patterns.
- The provided scenario outlines the minimum requirements for this practical. Feel free to add any additional classes or functionalities to better demonstrate your understanding of the patterns and tasks. You are allowed to add "enemy" legions, an "accounting system," or any other extensions for potential bonus marks.
- Thoroughly test your code (more details will be provided in the next task). Tutors will only mark code that runs.
- You may use arrays, vectors, or other relevant containers. Avoid using any libraries not mentioned in the general instructions, as your code must be compatible with FitchFork.
- Tutors may require you to explain specific functions to ensure you understand the pattern being implemented.

7 Task 3: FitchFork Testing Coverage

In this task, you are required to upload your completed practical to FitchFork. The FitchFork submission slot will open on **Friday, 2 August 2024**.

- Ensure that your code has at least **60% coverage** in your testing main. This is necessary for demonstrating your work to the tutors.
- You will need to show your FitchFork submission with sufficient coverage to the tutor before being allowed to demo.
- You will be required to download your code from FitchFork for demonstration purposes.
- It might be useful to have two main files: a testing main for FitchFork and a demo main with a more user-friendly interface (e.g., a terminal menu) for demoing. This is just a suggestion, not a requirement. **You are required to have at least a testing main.** If you choose to have a demo main, upload it to FitchFork as well, but ensure it does not compile and run with `make run`.
- Name your testing main file `TestingMain.cpp`.
- If you create a demo main, name it `DemoMain.cpp` so it can be excluded from the coverage percentage. Note that you do not have to test your demo main.

- Additionally, upload a makefile that will compile and run your code with the command `make run`.

8 Task 4: Demo Preparation

You will have 5-7 minutes to demo your system and answer any questions from the tutors. Proper preparation is crucial.

- Ensure you have all relevant files open, such as UML diagrams and source code.
- Set up your environment so that you can easily run the code during the demo after downloading it.
- Be ready to demonstrate specific function implementations upon request.
- Practice your demo to make sure it fits within the allotted time and leaves time for questions.
- Prepare a brief overview (30 seconds) of your system to quickly explain its functionality and design.
- Make sure your testing and demo mains (if applicable) are ready to show their respective functionalities.
- Have a clear understanding of the design patterns used and be prepared to discuss how they are implemented in your code. Also, think about other potential use cases for the patterns.
- Be prepared to answer questions. If you do not know the answer, inform the marker and offer to return to the question later to avoid running out of time.

9 Submission Instructions

You will submit on both ClickUp and FitchFork.

- FitchFork submission:
  - Zip all files.
  - Ensure you are zipping the files and not the folder containing the files.
  - Upload to the appropriate slot on FitchFork well before the deadline, as no extensions will be granted.
  - **Ensure that you have at least 60% coverage.**
- ClickUp submission. You should submit the following in an archived folder:
  - Your UML diagrams (both as images and Visual Paradigm projects).
  - Your source code.