Practical 3

COS214

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

Department of Computer Science

Deadline: 26/08/2024 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 any library but it can't do the pattern for you:
- 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

Background

Prologue: The Call of Destiny

In the 2nd century AD, Emperor Marcus Aurelius, philosopher and leader, faces continuous threats along the Roman frontiers. Your role is to manage and optimize the deployment of Roman legions — you will use Abstract Factory, Strategy, Memento, and Composite.

4 Abstract Factory

Goal: Dynamically produce various types of military units tailored to specific battlefield environments influenced by strategic decisions.

These specific types of military units should no be mixed, in other words, if the current terrain requires units specialised for the Woodland, there should not also be units specialised for the Riverbank. The Abstract Factory helps us achieve this.

4.1 Classes and Attributes

4.1.1 LegionFactory

(Abstract Base Class)

- *Methods:*
  - createInfantry(): Infantry *
  - createCavalry(): Cavalry *
  - createArtillery(): Artillery *

4.1.2 RiverbankFactory, WoodlandFactory, OpenFieldFactory

(Concrete Factory Classes)

- *Methods for each factory:*
  - Adapted implementations for creating units best suited to the terrain type.
  - deployArtillery(): Specific siege engines for terrain challenges.

4.1.3 LegionUnit

(Abstract Class - not a participant):
Description: Serves as the common interface for all military units produced by the factories, ensuring standard functionality across different unit types.

Methods:

- move() - Command to move the unit on the battlefield.
- attack() - Command to engage in combat.

4.1.4 Infantry, Cavalry, Artillery

(Abstract Product Classes):

Description: These classes inherit from **LegionUnit** and are tailored with specialised behaviors and attributes that reflect their respective roles on the battlefield.

Details:

- **Infantry** - Optimized for close combat and defense operations.
- **Cavalry** - Provides mobility and impact in flanking maneuvers.
- **Artillery** - Delivers long-range support and bombardment capabilities.

4.1.5 Woodland, Riverbank and OpenField specialisations per Abstract Product

(Concrete Product Classes):

Description: These classes inherit from the abstract products and are tailored with specialised behaviors that reflect their respective terrain on the battlefield.

Details:

- **Woodland** - Trained and optimised for woodlands terrain.
- **Riverbank** - Trained and Optimised for riverbank terrain.
- **OpenField** - Trained and Optimised for open fiedls terrain.

4.1.6 Implementation Details:

- Each concrete factory produces units with attributes suited for specific terrains and strategic needs, influencing the composition and capabilities of each legion.
- These product classes also function as leaves in the Composite pattern, representing the basic operational units within larger military formations (details to follow).

5 Strategy

Goal: Implement varying military tactics that can be adapted based on enemy movements and battlefield conditions.

5.1 Classes and Methods

5.1.1 TacticalCommand

(Context Class):

- *Attributes:*
  - strategy: BattleStrategy *
- *Methods:*
  - setStrategy(s: BattleStrategy *): void
  - executeStrategy(): Executes the current strategy.
  - chooseBestStrategy(): void - chooses an appropraite strategy based on previous results (using the Memento pattern, explaination to follow)

5.1.2 BattleStrategy

(Interface)

- *Methods:*
  - engage(): Defines the combat engagement protocol.

5.1.3 Flanking, Fortification, Ambush

(Concrete Strategy Classes)

- Specific implementations for different military tactics.

5.1.4 Implementation Tasks:

- Strategies influence how factories produce and deploy units, integrating with the Abstract Factory pattern to adapt production rates and unit types in real-time.

6 Memento

Goal: Utilise tactical memories to learn from and adapt to evolving combat scenarios, storing successful strategies for later use.

6.1 Classes and Responsibilities

6.1.1 TacticalPlanner

(Originator Class)

Responsibility: Manages the strategic decisions and state changes in battle operations, using the Memento pattern to save and restore previous states.

Attributes:

- currentStrategy: BattleStrategy *- Holds the currently active military strategy.

Methods:

- createMemento(): TacticalMemento *- Saves the current state into a memento.
- restoreMemento(memento: TacticalMemento*) - Restores the state from a memento.

6.1.2 TacticalMemento

(Memento Class):

- *Methods:*
  - storeStrategy(strategy: BattleStrategy *): Captures and stores the current strategy.

6.1.3 WarArchives

(Caretaker)

- *Methods:*
  - addTacticalMemento(memento: TacticalMemento *, label: string)
  - removeTacticalMemento(label: string)
  - Any other methods you might need

6.1.4 TacticalCommand

(Memento Client)

- Client for the Memento (as well as the Context for Strategy)
- The idea is to use the saved mementos (from caretaker) to make informed decisions of the strategies to use

6.1.5 Implementation Tasks:

- The Caretaker interacts with the Memento to store and retrieve strategies. The Strategy patterns Context should use the stored mementos (as client) to programatically choose the best strategy

7 Composite

Goal: Manage groups of units or legion compositions as a single entity, allowing for unified command and control over mixed-type forces.

7.1 Classes and Structure

UnitComponent (Interface):

- *Methods:*
  - move()
  - fight()
  - add(component: UnitComponent *)
  - remove(component: UnitComponent *)

Infantry, Cavalry, Artillery (Leaf Classes):

- Basic leaf units without children.

Legion (Composite Class):

- Manages collections of UnitComponent, treating individual and composite units uniformly.
- Remember that a legion can contain sublegions etc.

8 Task 1: UML Diagrams (30 marks)

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

8.1 Class Diagram (15 marks)

- Create a single 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.

8.2 Object Diagram (5 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 all patterns should be represented

8.3 State Diagram (10 marks)

- Draw a UML state diagram showing how the memento and strategy patterns interact to choose the best strategy based on the "war archives" containing past strategies

9 Task 2: Implementation (60 marks)

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.
- 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. If you really need a specific library please email u21689432@tuks.co.za at least a week before the due date.
- Tutors may require you to explain specific functions to ensure you understand the pattern being implemented.

10 Task 3: FitchFork Testing Coverage (10 marks)

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** (6/10) 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`.

11 Task 4: Demo Preparation (20 marks)

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.

12 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.