COS 214 Practical Assignment 3

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

- Date Issued: **29 August 2023**
- Date Due: **26 September 2023** at **11:00am**
- Submission Procedure: **Upload to ClickUP**
- Submission Format: **archive (zip or tar.gz)**

1 Introduction

1.1 Objectives

In this practical you will:

- implement the Observer, Mediator, Iterator, and Command patterns in the context of a game
- Draft UML communication diagrams

1.2 Outcomes

When you have completed this practical you should:

- Be able to implement understand the patterns implemented
- Understand the function of UML communication diagram
- Be able to model and implement systems using the patterns of this practical to solve issues and model business logic

2 Constraints

1. You must complete this assignment individually or in groups of two.
2. You may ask the Teaching Assistants for help but they will not be allowed to give you the solutions.
3. You must demo the system in the Labs at the conclusion of the prac and both partners need to be present if working in pairs

3 Submission Instructions

You are required to upload all your source files (that is `.h` and `.cpp`), your Makefiles, and a single PDF document and any data files you may have created, in a single archive to ClickUP before the deadline. A demonstration of the implementation will be conducted in the labs, It is required to create a main that allows for the proper demonstration of the patterns.

4 Mark Allocation

Task	**Marks**
Observer pattern	15
Tower defense	20
TOTAL	35

5 Assignment Instructions

Task 1: Observer pattern .... (15 marks)

Observer pattern allows us to monitor the state of another object and react to updates as needed, eliminating the need for polling. This in and of itself can be very useful with regards to performance: Take for example the game Minecraft, when the creeper(an enemy that explodes causing damage to everyone and everything around it) was introduced it was planned that other enemies would run from the creeper so that it would not get hurt. As it was implemented using polling, ie every enemy would check if there is an exploding creeper nearby every frame, it had a major impact on performance and the idea was scrapped.

This demonstrates that observer allows us to only run code that should react to a state change only when the state changes rather than have code that runs periodically, possibly affecting performance, that would check for updates.

This is also simmilar to how we interact with smart contracts on a blockchain where we can setup an observer to notify when the contract changes state rather than grabbing every new block and scanning for changes which results in a large bandwidth and performance overhead.

For this exercise we will be extending the contract task from the last practical.

1.1 Add the subscriber pattern to send notifications whenever the state of the contract changes. (10)

1.2 Draft a uml communication diagram showing a user adding a condition to the contract and a notification being sent to the console (5)

Task 2: Tower defense .... (20 marks)

A major concern during the development of any application is the concept of highly coupled code and why we want to avoid this. Whenever code is "highly-coupled" any change in one piece of code can and probably will affect the manner in which other aspects of code perform. To alleviate this we concern ourselves with seperation of concerns meaning we reduce the duplicity of our code and we ensure that we do not have to care about what other code is doing to be able to perform our current function. For example we never care how the network library we imported send the request and performs the underlying protocol, we only care that the request comes back in a format that we can process.

Mediator allows us to clearly separate classes by controlling how communication is performed such that a specific object does not need to have references to a number of other objects and needing to know their exact interfaces. We rather have a mediator that any component of our application that needs to communicate can send a message to and the mediator takes responsibility to inform the necessary objects. Almost like a post office where we send letters then they go and deliver them to whom it concerns.

Similarly the iterator pattern allows us to traverse collections and data structures without us needing any knowledge of how the data structure is organised. The data structure can be as simple or complex as desired but we only care about if there is more data in the structure that still needs to be processed. We might still want the ability to traverse the structure in different ways in which case we can define differing concrete iterators that will again mean we do not need to care how, specifically, the traversal is implemented only that we get the data needed to perform our current task.

Lastly we have the command pattern, this allows us to address code duplicity and a few other problems in a very succinct manner. Whenever we write an application we might end up with the same action needing to be performed by various objects that are not related. Meaning we would end up with duplicated code littered across the code base. Or we can use the command pattern so that everywhere where we would execute the action we rather create a command object and send that off to the relevant receiver to process. We also end up with the ability to cue operations that might take some time and cannot be performed concurrently(Think about a webserver needing to process incoming requests from multiple clients that may outnumber the number of available threads) as well as the ability to keep a history of commands executed that can be used for various functions such as auditing and the reversal of operations by performing the inverse of the operation, essentially allowing us to 'Undo' actions.

For this exercise we will be implementing a dungeon crawler/tower defense game in the likeness of the Dungeons games. In this game we are in charge of building and protecting a lair from the heroes so that they cannot find and destroy your treasure hoard. We will simplify the game down to the following components:

- The storyteller class mediator
  The storyteller is a concept from tabletop games where one "player" takes the role of the storyteller/game master. They decide when things happen that aren't in the control of the player. In this case the storyteller will be responsible for creating waves of heroes that will attempt to assault your lair that you must then fight off.
- The lair class iterator
  Here we want to demonstrate the use of iterator by creating enemies that will search for the treasure. The player will be responsible for building a lair, they will create "tiles" that act as different rooms that the heroes will move through in search of the treasure. A tile can connect to another tile in each of the four cardinal directions allowing for the creation of a fully fledged lair. You will implement the iterator pattern to perform depth first and breadth first traversals which the hero classes can then use to navigate the lair as they should. stategy
- the hero classes stategy & iterator
  These should be very simple classes where you can define different unit archetypes that may have different health points or abilities. The crucial thing to demonstrate would be the use of different iterators based on the decision made by the storyteller, whether a breadth or depth traversal would be preferred for the current search. You can also decide rather than a single object being a hero, it could be a wave of heroes traversing the lair together.
- Trap classes iterator
  These should again be very simple as we want to have the ability to place traps inside tiles to create hazards for the heroes, dealing damage as they traverse the lair to try and eliminate the threat before they stumble onto your treasure leading to the player losing the game.
- player interaction (Menu class) command
  This class should present the player with a menu so that they can interact with the game, the player must be able to create or remove tiles from the lair, add traps to rooms, research new traps, manage the research queue. They must also have a way to view their current wealth which will be used in the next classes to manage resources used for research.
- The bank class mediator
  The bank class is responsible for keeping track of our current gold reserves. These gold reserves will be used to fund research and will also be used as the health of the treasure hoard. when you run out of gold the game end in defeat. If you manage to survive a certain number of waves you will be victorious.
- Research laboratory classes command
  The research class is where we will be using the command pattern. Research will allow us to create new traps that we can build in rooms to slow down or eliminate the heroes. Here the is some freedom in the manner in which the traps will function, straight damage to the heroes, or will it slow them down, teleport them to a different tile. the research will however cost money to conduct and will take some time to complete, this can be simulated by a turn counter that will count down until the research is concluded. Because the research takes time to complete we want the ability to queue up multiple research tasks to run after each other and to be able to remove tasks from the queue, hence the use of the command pattern.
- The engine class. observer
  This will be the heart of the game, whenever research is being queued by the player interface it will be responsible for notifying the bank how much gold has been used to fund the research. When research completes it will be notified by the research class and notify either the player interface about a new trap that has been added to their arsenal or when a wave of heroes interacts with a trap will communicate between the trap and the wave regarding damage or other effects inflicted on the wave.

2.1 implement the described system in such a manner that you can easily demonstrate the patterns at work during the demo. (20)