COS 214 Practical Assignment 4

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

- Date Issued: **30 September 2020**
- Date Due: **14 October 2020** at **19:00**
- Submission Procedure: **Upload via ClickUp**
- Submission Format: **zip or tar + gzip archive (tar.gz)**

1 Introduction

1.1 Objectives

During this practical assignment you will be required to implement the *Command*, *Adapter* and *Chain of Responsibility* Design Patterns.

1.2 Outcomes

After successful completion of this assignment you should be comfortable with the following:

- Understanding the Command Design Pattern
- Understanding the Adapter Design Pattern
- Understanding the Chain of Responsibility Design Pattern

2 Constraints

1. You must complete this assignment individually.
2. You may ask the Teaching Assistants for help but they will not be allowed to give you the solutions.

3 Submission Instructions

You are required to upload all your source files (that is `.h` and `.cpp`), your Makefile, UML diagrams as individual or a single PDF document, a text file labelled "readme" explaining how to run the program and any data files you may have created, in a single archive to Clickup before the deadline. Each section should be in a seperate folder and each section should have its own Makefile. Failure to follow any of these instructions may result in a mark penalty. **If you did not upload you will receive a 0 for this practical assignment.**

4 Mark Allocation

Task	**Marks**
The RemoteControlVehicle Hierarchy	12
The Command Hierarchy	12
The Button Class	3
The Adapter Pattern	8
Main	10
UML Activity Diagram	6
Who will obey the Command?	18
Tie Everything Together	11
TOTAL	80

5 Assignment Instructions

You have become very interested in remote control vehicles, and you ordered a universal remote to control them on XLO - an online store. Unfortunately, the delivery men moved your remote right past your extensive magnet collection, erasing its software completely. You decide to redo the software using the Command design pattern.

In this assignment, you must make use of the Command and Adapter design patterns to implement several different remote control vehicles and a universal remote that controls them. You will then integrate the Chain of Responsibility into your implementation.

This assignment consists of eight tasks that build on each other:

1. The RemoteControlVehicle hierarchy
2. The Command hierarchy
3. The Button class
4. The Adapter design pattern
5. The main program for testing
6. The UML Activity diagram
7. Who will obey the Command?
8. Tie everything together

Task 1: The RemoteControlVehicle Hierarchy .... (12 marks)

In this task you will create the classes for the virtual RemoteControlVehicle and concrete vehicles which will act as receivers in the pattern.

- A RemoteControlVehicle Class.
- Concrete RemoteControlCar and RemoteControlPlane classes that inherit from RemoteControlVehicle.

1.1 **The RemoteControlVehicle Class**
The RemoteControlVehicle class should have at least the following members:

- **Private:**
  - on - a boolean value representing the on/off state of the vehicle.
- **Public:**
  - A constructor that initialises the on member variable to false.
  - A pure virtual on method which takes no parameters.
  - A pure virtual off method which takes no parameters.
  - A pure virtual forward method which takes no parameters.
  - A pure virtual backward method which takes no parameters.

1.2 **The receiver classes**
The RemoteControlCar and RemoteControlPlane classes act as the receivers in the pattern, and also inherit from RemoteControlVehicle, and should have at least the following members:

- **Public:**
  - An on method which takes no parameters. This method prints the following messages respectively for the different classes, and then updates the on variable, if the vehicles are not already on:
    * RemoteControlCar - "RC car is turned on."
    * RemoteControlPlane - "RC plane is turned on"

    If the vehicle is already on, an appropriate message stating this must be printed.

  - An off method which takes no parameters. This method follows the same pattern as the On method, only replacing "turned on" with "turned off" and updating the appropriate variable.
  - A forward method which takes no parameters. This method prints the following messages respectively for the different classes, if the vehicles are turned on:
    - RemoteControlCar - "RC car drives forward quickly."
    - RemoteControlPlane - "RC plane pulls up into a vertical climb."

    If the vehicle is not turned on, an appropriate message stating this must be printed.
  - A backward method which takes no parameters. This method prints the following messages respectively for the different classes, if the vehicles are turned on:
    - RemoteControlCar - "RC car reverses backward carefully."
    - RemoteControlPlane - "RC plane points its nose down into a dive."

    If the vehicle is not turned on, an appropriate message stating this must be printed.

Task 2: The Command Hierarchy ....(12 marks)

In this task you will create the classes for the virtual Command and different concrete commands which will act as concreteCommand participants in the pattern.

- The Command Class.
- Concrete TurnOn, TurnOff, MoveForward and MoveBackward classes that inherit from Command.

2.1 The Command Class

The Command class should have at least the following members:

- **Private:**
  - receiver - a RemoteControlVehicle pointer, which will take the specific vehicle to which a command must be applied.
- **Public:**
  - A constructor that takes a parameter to initialise the receiver member variable.
  - A pure virtual execute method which takes no parameters.
  - A pure virtual undo method which takes no parameters.

2.2 The concrete command classes

The TurnOn, TurnOff, MoveForward and MoveBackward classes act as the concrete command participants in the pattern, which inherit from Command and should have at least the following members:

- **Public:**
  - An execute method which takes no parameters. The execute method will call the receiver member variable's corresponding function based on the concrete command you are working with, e.g. TurnOn's execute function will call its receiver's on function.
  - An undo method which takes no parameters. The undo method will call the receiver member variable's corresponding function based on the concrete command you are working with in order to reverse what the execute function would do, e.g. TurnOn's execute function will call its receiver's off function.

Task 3: The Button Class ....(3 marks)

In this task you will create the Button class, which will act as the Invoker participant in the pattern.

3.1 The Button Class

The Button class should have at least the following members:

- **Private:**

  - command - a Command pointer, which will take the specific command to be executed when the button is pressed.
- **Public:**
  - A constructor that takes a parameter to initialise the command member variable.
  - A press method which takes no parameters. This method executes the command it is associated with.

Task 4: The Adapter Pattern .... (8 marks)

In this task you will add an additional concrete remote control vehicle, using the Adapter pattern to make sure the vehicle is compatible with your current implementation.

- The RemoteControlRocket Class.
- The RocketAdapter Class.

4.1 The RemoteControlRocket Class

The RemoteControlRocket class will act as an Adaptee in the Adapter pattern, and should have the following members:

- **Public:**
  - A takeOff method which takes no parameters. This method prints the following message: "The RC rocket blasts off its take-off platform."
  - A stopLaunch method which takes no parameters. This method prints the following message: "The RC rocket deploys its parachute and floats down to the ground."
  - An increaseThrottle method which takes no parameters. This method prints the following message: "The RC rocket increases its throttle and accelerates."
  - A decreaseThrottle method which takes no parameters. This method prints the following message: "The RC rocket decreases its throttle and ascends at a slower rate."

4.2 The RocketAdapter Class

The RocketAdapter class acts as the Adapter participant in the Adapter class, as well as allowing interaction with the RemoteControlRocket as if it were a Receiver participant in the Command pattern and should have at least the following members:

- **Private:**
  - adaptee - a RemoteControlRocket pointer that contains the rocket to be adapted for use.
- **Public:**
  - A constructor, that takes at least an appropriate object to initialise the adaptee member variable.
  - An on method which takes no parameters. This method calls the adaptee's takeOff function, and then updates the on variable, if the vehicle is not already on. If the vehicle is already on, an appropriate message stating this must be printed.
  - An off method which takes no parameters. This method follows the same pattern as the on method, only calling the adaptee's stopLaunch function instead, and updating the appropriate variable.
  - A forward method which takes no parameters. This method calls the adaptee's increaseThrottle function, if the vehicle is already on. If the vehicle is not already on, an appropriate message stating this must be printed.
  - A backward method which takes no parameters. This method calls the adaptee's decreaseThrottle function, if the vehicle is already on. If the vehicle is not already on, an appropriate message stating this must be printed.

Task 5: Main .... (10 marks)

Create a main function that creates different receivers, commands and buttons relating to them, and demonstrate that your implementation works according to the design pattern. You must also explicitly show where you demonstrate the Adapter pattern.

Task 6: UML Activity Diagram .... (6 marks)

Create an activity diagram demonstrating the different activities that are associated with a specific receiver of your choice in your Command pattern implementation.

Export the activity diagram to a PNG image and upload it with your code.

Task 7: Who will obey the Command? .... (18 marks)

Make the following modifications in your code to simulate the Chain of Responsibility:

- Add the classes: *Fly* and *Speed* as concrete commands. Follow a similar design as implemented in the other concrete command participants.
- Adapt the *RemoteControlVehicle* class to function as a Handler participant in the Chain of Responsibility Design Pattern. Be sure to include a virtual *handleRequest()* function in this class.
- Only a plane can handle the "fly" command, and the car a "speed" command when a button is pressed on the remote. Adapt the *RemoteControlCar* and *RemoteControlPlane* classes to become concrete handlers, which implement *handleRequest()*.

Task 8: Tie Everything Together .... (11 marks)

Include a simulation of the Chain of Responsibility Design Pattern described in Task 7 in the main function you implemented in Task 5.