COS 214 Practical Assignment 4

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

- Date Issued: **30 September 2021**
- Date Due: **14 October 2021** at **11:00am**
- Submission Procedure: **Upload via ClickUP**
- Submission Format: **archive (zip or tar.gz)**

1 Introduction

1.1 Objectives

In this practical you will:

- get comfortable using the programming tools: GDB for debugging your code, Valgrind for tracing memory leaks in your code, and GoogleTest to perform unit testing of your code.
- simulate a real-world application of the Observer Design pattern, emphasizing the order of message exchanges between participants.
- integrate the Iterator and the Mediator Design patterns into a scenario involving the Observer Design pattern.
- perform unit testing on your code and report both positive and negative test results.

1.2 Outcomes

When you have completed this practical you should:

- have a basic practical knowledge of how to use GDB, Valgrind, and GoogleTest.
- be able to use the Observer Design pattern to solve practical problems, while reducing coupling between classes and improving code reusability.
- have a solid understanding of how the Iterator and the Mediator design patterns work. You should contrast between how messages are exchanged in the Observer and in the Mediator.
- be able to design test cases and develop a test suite for parts of your code.

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 and any data files you may have created, in a single archive to ClickUP before the deadline. **If you did not upload you will receive a 0 for this practical assignment. Code that does not compile and run will not be marked.**

4 Mark Allocation

Task	**Marks**
Programming Tools	20
Telecommunications Network Maintenance Process	40
The List of Engineers	40
Faults Notifications	40
Unit Testing	10
TOTAL	150

5 Assignment Instructions

In this assignment you will learn how to use the GNU Debugger to debug your code in case it produces errors that are not obvious from looking at the source code itself. You will then use Valgrind to trace any memory leaks in your *C++* program.
The remainder of the assigment will guide you through the implementation of the Observer, Iterator, and Mediator Design patterns.
Finally, you will develop a unit test suite for parts of your program.

Task 1: Programming Tools ..... (20 marks)

A professor at 214 University is worried about the performance of his students in the module "Design Patterns in C++". He decides to develop a C++ program that will improve the students' performance by calculating each of their marks as a percentage of the highest mark per test. In one of the tests, the performance was so poor that no student obtained any marks. He was very eager to see how far his program would go in helping the students. To his amazement, his program also stopped working.

1.1 You are provided with the program `marks.cpp` which the professor was using to improve the students' marks. Follow the instructions below to find out what the cause of the error is that is tormenting the old professor: (10)

1. Compile and run the program on your linux machine with an option to request debugging information; note the behaviour of the program.
2. Use GDB to run the program in debug mode.
3. Use a command to run the program within the debugger. What is the error produced by the program? At which line did the error occur and what were the values of the function arguments at the time?
4. Use the list command to see the context of the crash, i.e., to list the lines around where the crash occured.
5. Use the `where` or `backtrace` command to generate a stack trace and interpret the output.
6. Move from the default level '0' of the stack trace up one level to level 1.
7. Run the `list` command again and note which lines are printed this time.
8. Print the value of the local (to main) variable *highest*
9. Why did the crash occur?

1.2 While waiting for you to debug his "marks" program, the professor decides to write another program that he will use to input marks for all his 10 students. This program also behaves unexpectedly and the professor gives it to you with a suspicion that there is a memory leak. (10)

1. Compile the program `capture.cpp` with an option to include debugging information.
2. Run your program in Valgrind with the `leak check` option enabled. There is no need to specify "Memcheck" for the tool, as it is the default option.
3. What is the process ID on your output?
4. What is the first error that you see from the output?
5. Below the first error is a stack trace, what does it tell you?
6. "definitely lost" shows that there is a memory leak in your program. Why has 40 bytes been lost?
7. How would you fix the memory leak?

Task 2: Telecommunications Network Maintenance Process ..... (40 marks)

As the number of devices connected to the internet explodes, and many economies rely heavily on this massive network, internet service providers need to maintain very high standards of reliability to keep abreast of the changing market demands. Many mobile devices are connected through radio interfaces to telecommunications networks such as GSM, 3G, LTE, and of late, 5G. In this task you will use the Observer Design pattern to design a robust process through which a telecommunication network can be maintained to minimize outages and promote proactive maintenance.

2.1 On a high level, we can view a telecommunications network as being composed of network elements and engineers who maintain these network elements. (20)

1. Create a **NetworkElement** class that will represent the `Subject` participant in the Observer Design pattern with the following specification:
   - a virtual `destructor.`
   - an `attach()` function which takes a reference to an **Engineer** to allow a newly-employed engineer to register for maintaining the network element.
   - a `detach()` function to allow an engineer who is resigning to deregister for maintenance of the network element.
   - a `notify()` function which will inform all engineers tending to the network element that there is a fault or any other alarm condition on the network element. The function must print an appropriate message for each engineer being notified: e.g. networkElement.name + "changed status to" + networkElement.alarmState + "! Notifying" + engineer.name + "..."
   - a relevant `constructor.`
   - a list of engineers using any data structure of your choice.
2. Some of the most critical network elements in a telecommunications network are: BTS (Base Transceiver Station), BSC (Base Station Controller), MSC (Mobile Switching Centre), and GGSN (Gateway GPRS Support Node). Create the following classes as concrete subjects in the Observer Design pattern:
   - **BTS**
   - **BSC**
   - **MSC**
   - **GGSN**

   The concrete subjects must override the functions defined as virtual in the subject participant. In addition to this, each concrete subject must have the following:
   - a `name` attribute. For example: "BTS01", "BTS02", "MSC01"
   - The `constructor` must at least be parameterized with the `name` attribute or alternatively implement a `setter` and a `getter` for this attribute.
   - an `alarmState` attribute which can take any of the values: "critical", "major", "minor", or "clear".
   - a `getAlarm()` function that returns the alarm status of the node.
   - a `setAlarm()` function that sets the alarm status of the node.
3. Create an **Engineer** class which will be the observer interface within the Observer pattern with the following specification:
   - a virtual `destructor.`
   - a relevant `constructor.`
   - a pure virtual function `update()` which will update the observer object with the state of the concrete subject to which it is registered.
4. There are different types of engineers each of which monitors or observes a particular network element. Create the following classes as concrete observers:
   - **RadioEngineer**: monitors the **BTS** and the **BSC.**
   - **PowerEngineer**: monitors the **BTS.**
   - **TransmissionEngineer**: monitors the **BTS.**

- **CSCoreEngineer**: monitors the **MSC.**
- **PSCoreEngineer**: monitors the **GGSN.**

Each concrete observer must override the `update()` operation in the **Engineer** class. It must also have the attributes `alarmStatus` which holds the current state of the observed subject, a `_networkElement` attribute which is a reference to a subject to which it is registered, as well as a `name` of the engineer.

2.2 Consider the following scenario: a **RadioEngineer** is doing some routine maintenance on a **BTS** when he mistakenly trips a mains circuit breaker, thus setting the **BTS** `alarmState` to "critical". Following this change of state, the **BTS** notifies all 3 types of engineers registered to it, printing appropriate messages to this effect. In response, a **PowerEngineer** attends to the fault and clears the alarm; after which the **BTS** notifies all 3 engineers that the alarm is "clear." (10)

Write a `main()` function that simulates this scenario.

2.3 Draw a sequence diagram that shows how messages are exchanged between the involved objects in the scenario above. (10)

Task 3: The List of Engineers ..... (40 marks)

In the previous task, you were asked to use a data structure of your choice in the **NetworkElement** class to hold the list of **Engineers**. You probably opted for the C++ STL container classes such as `vector` or `list` and relied on their built-in iterators. In this task, we will drill down into the Iterator design pattern and implement our own aggregate and iterator.

3.1 1. Create an aggregate class called **EngineerCollection** with the following specification: (15)

- a factory method to create an instance of **EngineerIterator**; call it `createEngineerIterator()`.
- `addEngineer(Engineer):` `void`: adds a new engineer to the collection
- `removeEngineer():` `void`: removes an engineer from the collection.
- `isEmpty():` `bool`: checks whether the collection is empty.

Remember that these functions are defined as virtual.

2. All 3 types of engineers that maintain the **BTS** can be summarized as **OperationsEngineers.** Create the **VectorOfOperations** class as a concrete aggregate in the Iterator Design pattern. Include the following:

- a private `list` attribute of type $vector < Engineer >$.
- a `constructor` and a `destructor` for the class.
- override all the virtual functions in **EngineerCollection.**

3. The **PSCoreEngineer** and the **CSCoreEngineer** are collectively called **CoreEngineers**. Create the **ListOfCore** class as a concrete aggregate to hold a collection of **CoreEngineers**. Include the following:

- a protected `startingPoint` attribute which is a pointer to an **Engineer.**
- appropriate `constructor` and `destructor`.
- override all the virtual functions in **EngineerCollection.**

Note: The **ListOfCore** collection is a LinkedList while the **VectorOfOperations** relies on the vector structure.

4. include a `nextEngineer` attribute in the **Engineer** class that points to the next engineer in the collection, as well as a `getter` for the attribute.

3.2 1. Create the **EngineerIterator** with the following specification. (15)

- a function `first()` which specifies the first engineer in the aggregate.
- a function `next()` which specifies the next engineer in the aggregate.
- `hasNext()`: checks whether the aggregate has a next engineer after the current engineer.

- `current()`: returnts the current engineer.

All these functions are virtual.

2. Create the concrete iterator **OperationsIterator** with the following specifications:
   - `currentPos`; an attribute which shows the index of the current engineer. It is an integer.
   - `itList`; an iterator list of type $vector < Engineer >$.
   - override the virtual methods in the **EngineerIterator** class.
   - provide a relevant `constructor` and `destructor.`
3. Create the concrete iterator **CoreIterator** with the following specification:
   - the attributes `firstEngineer` and `currentEngineer` which are pointers to **Engineer**.
   - override the virtual methods in the **EngineerIterator** class.
   - provide a relevant `constructor` and `destructor.`

3.3 In addition to the data structure you used in the **Engineer** class before, add the collections **VectorOfOperations** to hold the observers attached to the subjects **BTS** and **BSC**, as well as **ListOfCore** to hold the observers attached to **MSC** and **GGSN**. Use the variable names `operations` and `core` respectively. (10)

A CS Core enginner is doing some configurations on the MSC when he inadvertently removes a static route to another node, setting the alarm status of the MSC to "major". The MSC immediately notifies him of the change, upon which he re-configures the route and "clears" the alarm, thereby recieving confirmation that the alarm is clear. When attaching the CS Core engineer to the MSC, use the `core` collection described above.

Write a `main()` function to simulate this scenario. Note that the `notify()` function should now iterate on your new **ListOfCore** collection using your corresponding iterator object. You can attach more CS Core engineers to MSC to see the Iterator pattern in action.

Task 4: Faults Notifications ..... (40 marks)

Engineers have a "social media" system through which they notify each other of the current status of the network elements which they maintain to avoid duplication of work. When each engineer updates their status, from "ready" to "busy" for example, every other engineer in the network is notified. In this task we will use the Mediator design pattern to help build this system for the engineers.

4.1 1. The engineers rely on a Network Management System (NMS) which forms the interface between them and network elements. However, there is a "social" component of the NMS that forms an interface specifically for engineer interactions. Create the class **NMS**, the mediator participant in the Mediator Design pattern, with the following specification: (30)
   - a `constructor` and `destructor`.
   - a pure virtual `notify()` function, which will notify all other engineer when an engineer changes their status.
   - a pure virtual `createEngineer()` function to create a new engineer in the system.
2. Add the following to the `Engineer` class:
   - a `constructor` taking a pointer to `NMS` as a parameter.
   - a virtual `statusChanged()` function which is called when the attribute `status` is changed. This function will call the `notify()` function.
3. Each specific **Engineer** class is a concrete colleague. Add the following to each type of engineer class:
   - a string attribute `status` together with its setter and getter functions.
   - override the `statusChanged()` function. This function will be called in the setter for `status`.
   - a `constructor` taking a pointer to `NMS` as a parameter.

4. In this scenario, engineers are using the NMS to update each other specifically when they are working on faults. Create a **FaultsNMS** class which will be a concrete mediator between the engineers with the following specification:
   - a `constructor` and `destructor`.
   - override the `notify()` function. This function takes a pointer to **Engineer** as a parameter and prints a message showing the engineer's name. For example: engineer.name + "changed status to" + engineer.getStatus + "...notifying the others!"

4.2 A fault on the **BSC** occurs and a **RadioEngineer** changes their status to "busy". Write a main funtion to simulate how other engineers immediately see this status change. (10)

Task 5: Unit Testing ..... (10 marks)

Pick any function that you have implemented and develop a unit test suite using GoogleTest. You may perform a boundary value analysis, branch testing, or loop testing depending on the function you have chosen. Submit both positive and negative test results along with your unit test code.