COS 214 Practical Assignment 3

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

- Date Issued: **8 September 2020**
- Date Due: **29 September 2020** at **8:00am**
- Submission Procedure: **Upload via the CS website**
- 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 and Valgrind for tracing memory leaks in your code.
- simulate a real-world application of the Observer design pattern, emphasizing the order of message exchanges between participants.
- implement the Iterator and Mediator design patterns with an understanding of when each design pattern is most suitable for use.

1.2 Outcomes

When you have completed this practical you should:

- have a basic practical knowledge of how to use GDB and Valgrind.
- be a 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.

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**
Programming Tools	20
Prison Break	30
Aggregate Traversal	20
Let's Socialize...	30
TOTAL	100

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.

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)

Copy all the commands that you use for this part onto a separate document and submit the documment with the rest of the assignment. Please include answers to the following questions in the document.

1. Compile and run the program on your computer 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: Prison Break ..... (30 marks)

In this task you will be using the Observer design pattern. You will create a prisoner, border guard and a prison area. The purpose of the observer is to prevent the prisoner from exiting the border of the prison area by blowing a whistle and pushing the prisoner one block back.

2.1 Create a subject called `Subject` with the following members: (20)

- `mObservers`: A vector of `Observers` pointers.
- `attach`: A function with a `Observers` pointer parameter which should be added to the vector.
- `notify`: A function that iterates over all the observers and notifies them by calling their `update` function.

Given the `Human` class, implement the subclass `Prisoner` that inherits from `Human` and `Subject` (multiple inheritance). `Prisoner` is therefore a concrete subject. Give it the following members:

- `Constructor`: The constructor takes the x and y coordinates as parameters and calls the parent constructor with the character 'X'.
- `setX` and `setY`: The functions should be overridden from the parent class `Human`. The `Human setX` and `setY` functions should be called and then `Subject`'s `notify` function. Hence all observers are notified immediately after the prisoner changes position. However, the observers will only react if the new position of the prisoner is on one of their borders.
- `runTowards`: Takes a parameter direction which specifies which direction to move (up, down, left, right). The prisoner can only move one block in any direction.

Implement a class `PrisonArea` with the following members:

- `mWidth and mHeight`: Member variables holding the width and height of the grid.
- `mprisoner`: A pointer to a `prisoner` object.
- `Constructor`: Takes the width and height as parameters and initializes the corresponding member variables. `mprisoner` should be allocated dynamically at the middle of the prison area. Create four border guards, one at each corner (according to the area's width and height). Add these border guards to the member vector. Since the border guards are the observers, they have to be attached to the subject, in this case the prisoner. Border guards are discussed below.
- `Destructor`: Should delete the dynamically allocated member variables including all the dynamically allocated objects in the `mBorderGuard` vector.
- `getWidth`, `getHeight` and `getprisoner`: Getter functions to access the member variables.
- `mBorderGuard`: Add a member variable that is a vector of `BorderGuard` pointers. Border guards are discussed below.

You can use the following `print` function in `PrisonArea` to test your code with the given function.

```
void PrisonArea::print()
{
        for (int y = 0; y < mHeight; ++y)
        {
                for (int x = 0; x < mWidth; ++x)
                {
                        char value = '␣';
                        if (x == mprisoner->getX() && y == mprisoner->getY())
                        {
                                value = mprisoner->getValue();
                        }
                        else
                        {
                                for(int i = 0; i < mBorderGuard.size(); ++i)
                                {
                                        if(x == mBorderGuard[i]->getX()
                                                && y == mBorderGuard[i]->getY())
                                        {
                                                value = mBorderGuard[i]->getValue();;
                                                break;
                                        }
                                        else if(x == mBorderGuard[i]->getX()
                                                || y == mBorderGuard[i]->getY())
                                        {
                                                value = '*';
                                        }
                                }
```

```
                }
                cout << value;
            }
            cout << endl;
        }
    }
```

Create an abstract observer class called `Observer` with the following members:

- `update`: A pure virtual function.

Create a concrete observer `BorderGuard` that inherits from `Observer` and `Human` with the following members:

- `mprisoner`: A pointer to a `prisoner` object.
- `Constructor`: Takes `x` and `y` coordinates and a `prisoner` pointer as parameters.
- `update`: This function is the reaction in the `Observer` pattern. The border guard should check if the prisoner is on the observed border. Each border guard only observes the borders directly connected to him/her. Hence the top-left border guard will only check the left and top borders, therefore checking if the prisoner's coordinates are on the border (x or y coordinates equal to zero). The other border guard will obviously have to check different borders, depending on the prison area's width and height. Since each border guard is positioned at a corner, you can get the x and y coordinates of the border guard and compare it to the coordinates of the prisoner. If the prisoner gets onto the border, the border guard should blow the whistle (print this to screen) and force the prisoner back into the area. Hence the prisoner will never be able to get onto the border ('*'). You can force the prisoner back by directly setting the coordinates of the prisoner one position back.

Implement any other required functions to make the prisoner move around the prison area. You can write a fixed script that moves the prisoner or you can allow for user input to move the prisoner. You will be required to demonstrate that the observer pattern works, so keep that in mind when you implement the prisoner's script.

You are required to create a main to demonstrate that your observer pattern works. If implemented correctly, the prisoner will never be able to go onto the border ('*') or beyond. Technically the prisoner can go onto the border, but will be pushed back into the area, so you will not notice it in the program output.

2.2 Draw a sequence diagram to demonstrate which messages are exchanged between the objects when a prisoner tries to escape and the guard blows a whistle while pushing the prisoner one block back. (10)

Task 3: Aggregate Traversal ..... (20 marks)

For this task, you are required to implement the Iterator design pattern for a dynamic array of type integer. You have been given a partial implementation of the aggregate and concrete aggregate classes, namely, LinearStructure and DynamicArray. You are required to create the iterator and concrete iterator class.

- Your iterator class should contain the following virtual methods:
  - `void first()`
  - `void next()`
  - `bool hasNext()`
  - `int current()`
- Your concrete iterator class should define these methods to work for the DynamicArray class.

NOTE: The provided aggregate classes are implemented as template classes, to create an iterator for ints, you will need to create a specialisation of the template class.
You may implement the iterators to work with the aggregate classes as templates.

Task 4: Let's Socialize... ..... (30 marks)

4.1 For this task, you should implement the Mediator design pattern. It has 2 additional classes combined with it which are not part of the pattern but are simply there to assist the program. You may create the following classes as specified: (30)

- `User`
  - It has `SocialSiteMediator* mediator` and `string username` as protected member variables
  - It has the following public member functions
    - `User(SocialSiteMediator* _mediator, string name)`
    - `void changed(string message)`
    - `string getName()`
    - `virtual void receiveMessage(string)=0`
- `SocialSiteMediator`
  - It has the following public member functions
    - `virtual void notify(User*, string message) = 0`
    - `virtual bool addUser(string username) = 0`
- `ConcreteSocialSiteMediator`
  - It inherits from SocialSiteMediator.
  - It has the following private member variables
    - `User** users` – to hold an array of pointers to Users
    - `int numusers`
  - It has the following public member functions:
    - `ConcreteSocialSiteMediator()`
    - `~ConcreteSocialSiteMediator()`
    - `virtual bool addUser(string username)` – implements parent function, adds new SocialSiteUser to users array, returns false if user is already in the array.
    - `virtual void notify(User* username, string message)` – implements parent function, send message to other users.
    - `User* getUser(string name)` – returns the user with the username given from users. If the user doesn't exist, it returns NULL
- `SocialSiteUser`
  - It inherits from `User`
  - It has the following private member variables:
    - `string status` – holds the user's status
    - `Inbox* myInbox` – holds messages received, you may decide the maximum number of messages to store.
    - `OldStatus* old` – holds previous statuses, you may decide the maximum number of statuses to store.
  - It has the following public member functions:
  - `SocialSiteUser(SocialSiteMediator* mediator, string name)` – should set the default status to "I am using SocialSite", and should save this status. It should also create a new Inbox object and OldStatus object for the user.
  - `void changeStatus(string newStatus)` – sets the status to `newStatus`

  - `string viewStatus()` – returns status
  - `virtual void receiveMessage(string message)` – implements parent function. Gets the message sent by the mediator, outputs that it?s been received and then calls the `saveMessage` function
  - `void postMessage(string message)` – sends a message to other users.
  - `void saveMessage(string message)` – saves the message to myInbox
  - `void viewInbox()` – prints out all the messages in myInbox
  - `void saveStatus()` – saves the current status into old
  - `void loadStatus()` – loads a status from old into status
- `Inbox`
  - This class is an extra class that holds the messages that the user received so that they may view them later on.
  - Each SocialSiteUser has a pointer to its own Inbox object.
  - It has the following private member variables:
    - `int numMessages`
    - `int MaxMessages` – the maximum number of messages that can be stored
    - `string* messages` – array of messages
  - It has the following public member functions:
    - `Inbox(int _MaxMessages)`
    - `~Inbox()`
    - `void save(string message)` – saves the message into the next available place in messages
    - `string load(int index)` – returns the message at the provided index
    - `int getNumMessages()`
- `OldStatus`
  - This is another extra class that holds previous statuses of the user if they choose to save it so that the user can reset their status to one of these later on.
  - Each SocialSiteUser has a pointer to its own OldStatus object.
  - It has the following private member variables:
    - `int numStatus` – the number of statuses in the array
    - `int MaxStatus` – the maximum number of statuses that can be saved
    - `string* status` – an array of statuses
    - It has the following public member functions:
    - `OldStatus(int _MaxStatus)`
    - `~OldStatus()`
    - `void save(string _status)` – saves the status to the next available position in the array
    - `load(int index)` – returns the status at the provided index
    - `getNumStatus()`

You may test your code with `test.cpp`. Example output can be found in `ExpectedOutput.txt`.