University of Calgary
Department of Electrical and Computer Engineering
Principles of Software Design - ENSF480

Lab 7 – Friday Oct 30, 2020

M. Moussavi, PhD, P.Eng

Note: Lab is a group assignment and you can work with a partner. Groups of 3 or more are not allowed.

Introduction:

This lab is also on design patterns. The main objective of this lab is to give you an opportunity to practice a few more important design patterns: Observer, Decorator, and Singleton pattern.

Marking Scheme: (45 marks total)

- Exercise A: 15 marks
- Exercise B: 16 marks
- Exercise C: 4 marks
- Exercise D: 10 marks

Due Date: Friday Nov 6 before 5:00 PM.

Exercise A (15 marks):

The purpose of this exercise is to give you an opportunity to practice using Observer design pattern in a simple Java program.

Read This First – A Quick Note on Observer Pattern

The Observer pattern is also one of the **behavioural** patterns - This pattern is also used to form relationships between objects at the runtime.

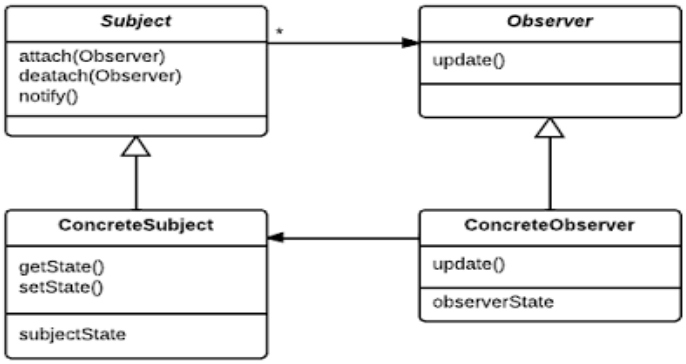


Figure 1

Figure 1: shows that the concrete subjects can add any observers, and when any changes happen to the data, all observers will be notified. The following figures may help you to better understand how this pattern works.

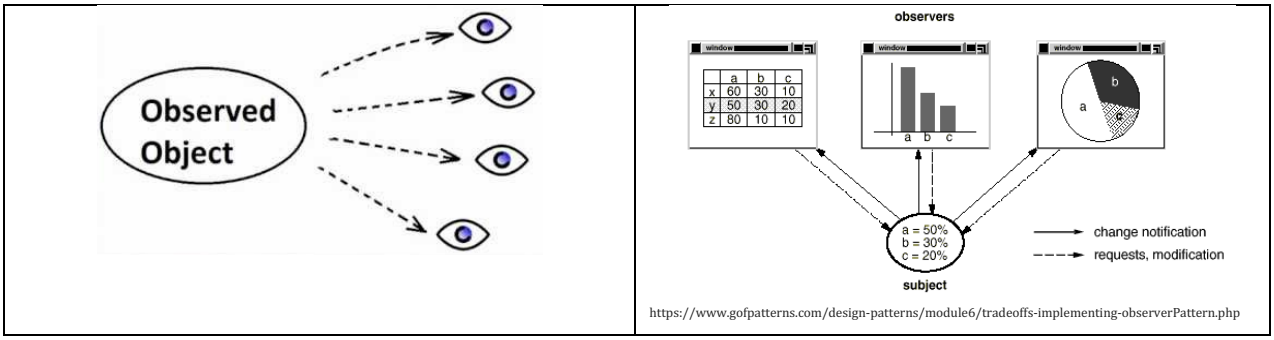
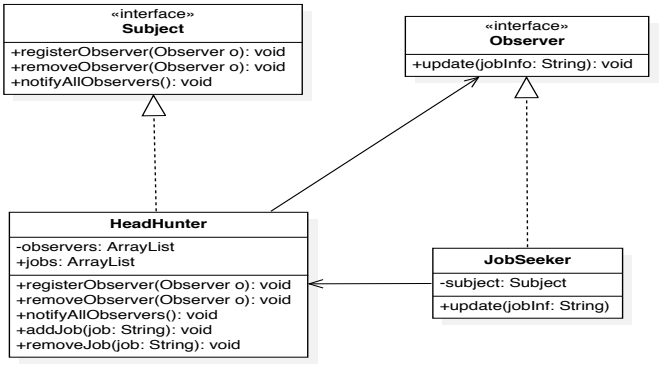


Figure 2

Figure 2 is a simple demonstration of the observers' notification concept and figure 3 illustrates the most common form of using this pattern. Any changes to the subject (data a, b or c) will be immediately translated in three observer views (tabular format, bar chart, or pie chart).
Using observer pattern is not limited to GUI presentation; it can be used for any notification system. Here is another example:



What to Do:

Download file `ObserverPatternController.java` form D2L. This file provides a client class that demonstrates how your observer pattern works. For the purpose of this exercise you just need to have three observers, and your design must be very flexible for change. In other words at anytime you should be able to add a new observer or remove an observer without any changes to the subject or observer classes. Your program must have the following interfaces and classes:

- Interface `Observer` with the method `update` that receive a parameter of type `ArrayList<Double>`
- Interface `Subject` with the required methods.
- Class `DoublArrayListSubject,` with a data list of type ArrayList<Double> , called `data` that is supposed to be visible to the observers. Consider other data members as shown in the Observer Pattern Design Model. This class should also have at least the following methods:
    - A default constructor that initializes its data members as needed. For example should create an empty list for its member called `data`.
    - Method `addData` that allows a new Double data to be added to the list

- Method `setData` that allows changing the data at any element in the list
- Method `populate` that populates the list with the data supplied by its argument of the function, which is an array of `double`.
- Other methods as needed

- Three concrete Observer classes as follows:
  - Class `FiveRowsTable_Observer` This class should have a function display that shows the date in 5 rows as illustrated in following example (any number of columns, as needed):

    ```
    10      30      11
    20      60      23
    33      70      34
    44      80      55
    50      10
    ```

    This class should also have a constructor to initialize it data member(s) as needed and to register the object as an observer.

  - Class `ThreeColumnTable_Observer` that displays the same list of data in tabular format as illustrated in the following example (3 columns and any number of rows as needed):

    ```
    10    20    33
    44    50    30
    60    70    80
    10    11    23
    34    55
    ```

    This class should also have a constructor to initialize it data member(s) as needed and to register the object as an observer.

  - Class `OneRow_Observer` that displays the same vector of data in single line as follows:

    ```
    10 20 33 44 50 30 60 70 80 10 11 23 34 55
    ```

    This class should also have a constructor to initialize its data member(s) as needed and to register the object as an observer.

If you have all the classes and methods defined properly, your program with the given client class `ObserverPatternController` should produce the following output:

```
Creating object mydata with an empty list -- no data:
Expected to print: Empty List ...
Empty List ...
mydata object is populated with: 10, 20, 33, 44, 50, 30, 60, 70, 80, 10, 11, 23, 34, 55
Now, creating three observer objects: ht, vt, and hl
which are immediately notified of existing data with different views.

Notification to Three-Column Table Observer: Data Changed:
10.0 20.0 33.0
44.0 50.0 30.0
60.0 70.0 80.0
10.0 11.0 23.0
34.0 55.0

Notification to Five-Rows Table Observer: Data Changed:
10.0 30.0 11.0
20.0 60.0 23.0
33.0 70.0 34.0
44.0 80.0 55.0
50.0 10.0

Notification to One-Row Observer: Data Changed:
10.0 20.0 33.0 44.0 50.0 30.0 60.0 70.0 80.0 10.0 11.0 23.0 34.0 55.0

Changing the third value from 33, to 66 -- (All views must show this change):
```

```
Notification to Three-Column Table Observer: Data Changed:
10.0 20.0 66.0
44.0 50.0 30.0
60.0 70.0 80.0
10.0 11.0 23.0
34.0 55.0

Notification to Five-Rows Table Observer: Data Changed:
10.0 30.0 11.0
20.0 60.0 23.0
66.0 70.0 34.0
44.0 80.0 55.0
50.0 10.0

Notification to One-Row Observer: Data Changed:
10.0 20.0 66.0 44.0 50.0 30.0 60.0 70.0 80.0 10.0 11.0 23.0 34.0 55.0

Adding a new value to the end of the list -- (All views must show this change)

Notification to Three-Column Table Observer: Data Changed:
10.0 20.0 66.0
44.0 50.0 30.0
60.0 70.0 80.0
10.0 11.0 23.0
34.0 55.0 1000.0

Notification to Five-Rows Table Observer: Data Changed:
10.0 30.0 11.0
20.0 60.0 23.0
66.0 70.0 34.0
44.0 80.0 55.0
50.0 10.0 1000.0

Notification to One-Row Observer: Data Changed:
10.0 20.0 66.0 44.0 50.0 30.0 60.0 70.0 80.0 10.0 11.0 23.0 34.0 55.0 1000.0

Now removing two observers from the list:
Only the remained observer (One Row ), is notified.

Notification to One-Row Observer: Data Changed:
10.0 20.0 66.0 44.0 50.0 30.0 60.0 70.0 80.0 10.0 11.0 23.0 34.0 55.0 1000.0 2000.0

Now removing the last observer from the list:

Adding a new value the end of the list:
Since there is no observer -- nothing is displayed ...

Now, creating a new Three-Column observer that will be notified of existing data:
Notification to Three-Column Table Observer: Data Changed:
10.0 20.0 66.0
44.0 50.0 30.0
60.0 70.0 80.0
10.0 11.0 23.0
34.0 55.0 1000.0
2000.0 3000.0
```

What to Submit for Exercise A?

1. Copy and paste all your source codes and your program output as part of your lab report and submit it in PDF format.
2. Create and submit a zip file that contains your source code (.java file(s))

A Brief Note on Application Decorator Design Patten in Real World:

The concept of a decorator focuses on the dynamically adding new futures/attributes to an object and particularly

to add the new feature the original code and other added code for other features must remains unaffected. The Decorator pattern should be used when object responsibilities/feature should be dynamically changed and the concrete implementations should be decoupled from these features. To get a better idea the following figures can help:

Figure 1

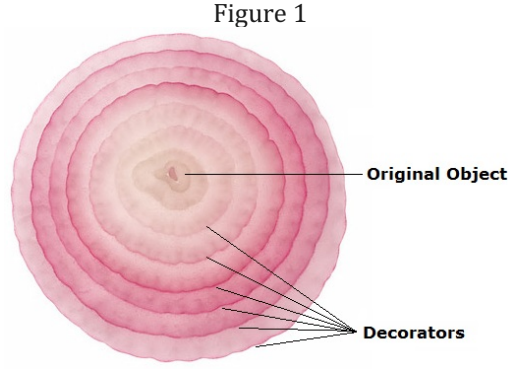


Figure from https://www.codeproject.com/Articles/176815/The-Decorator-Pattern-Learning-with-Shapes

Figure 2

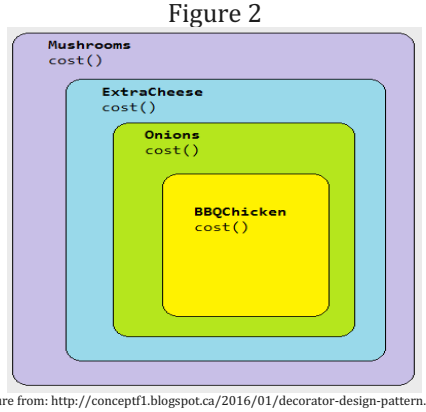


Figure from: http://conceptf1.blogspot.ca/2016/01/decorator-design-pattern.html

The left figure shows how an original object is furnished by additional attributes. A better real world example is the on the right that shows how the basic BBQ-chicken pizza is decorate by onion, extra-cheese, and mushrooms.

Official Definition of the Decorator Pattern:

The Decorator is a **structural** pattern, because it's used to form large object structures across many disparate objects. The official definition of this pattern is that:
It allows for the dynamic wrapping of objects in order to modify their existing responsibilities and behaviours.

Traditionally, you might consider subclassing to be the best way to approach this. However, not only subclassing isn't always a possible way, but the main issue with subclassing is that we will create objects that are strongly coupled, and adding any new feature to the program involves substantial changes to the existing code that is normally a desirable approach.
Let's take a look at the following class diagram that express the concept of Decorator Pattern:

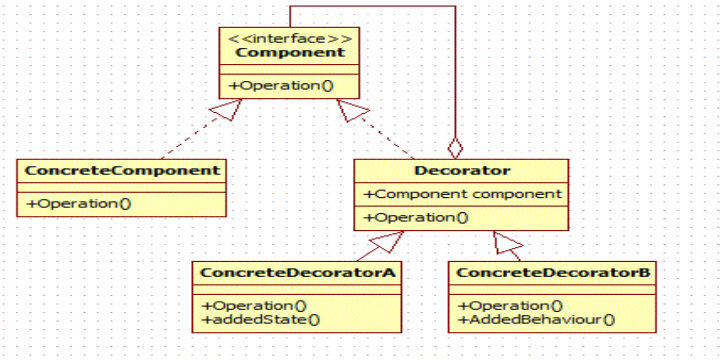


Figure 3

This diagram has three main elements:

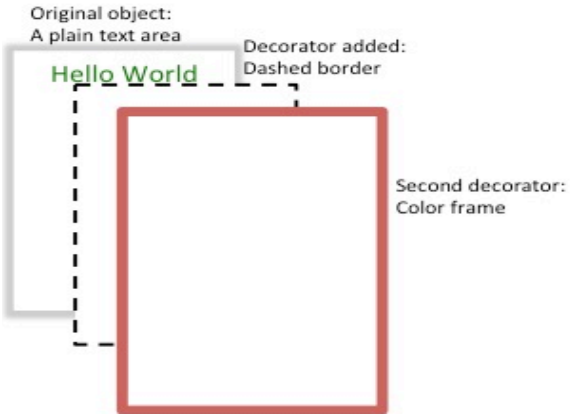
- The Component Interface, that defines the interface for objects that need their features to be added dynamically.
- The Concreter Component, implementing interface Component
- The Decorator, implementing the Component interface and aggregating a reference to the component. This is the important thing to remember, as the Decorator is essentially wrapping the Component.

And, one or more Concrete Decorators, extended from Decorator

Exercise B:

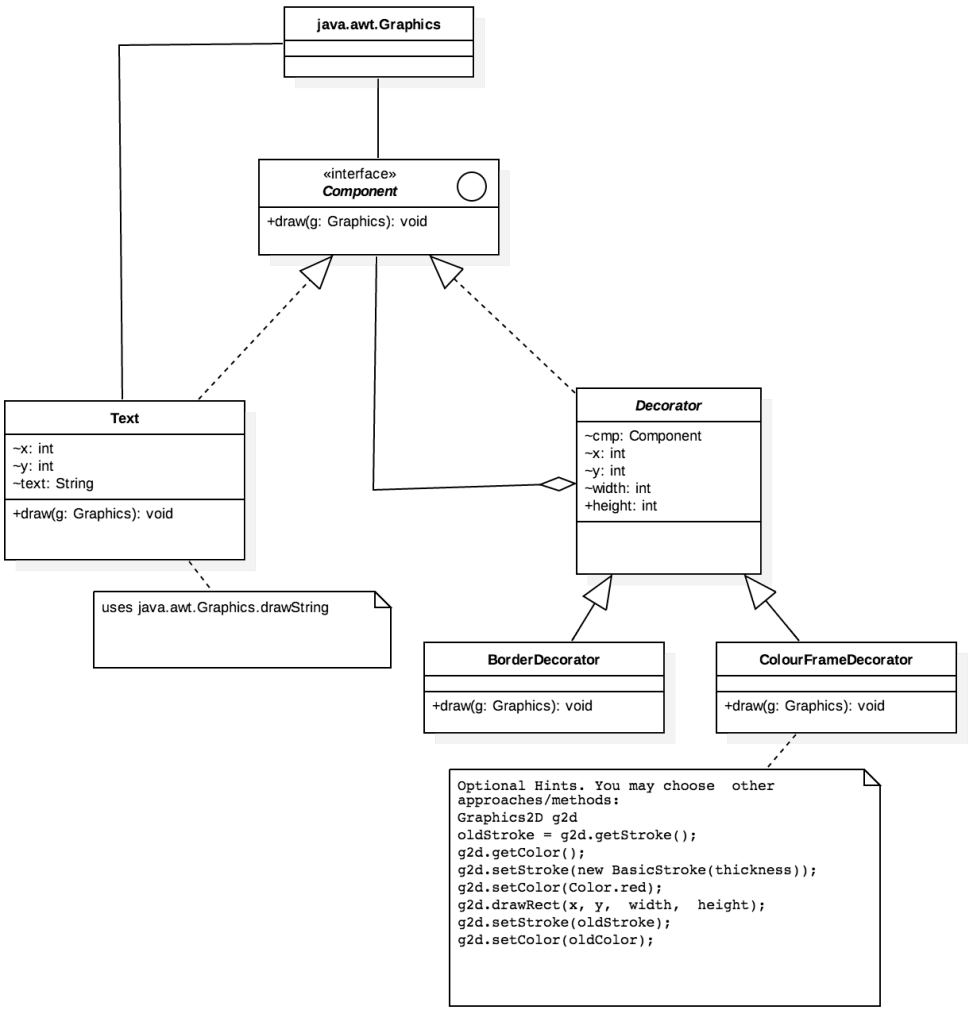
Lets assume you are working as part of a software development team that you are responsible to write the required code for implementing a simple graphics component that is supposed to look like:

But the details of this component is displayed in the following figure that consists of a main object, a text area with green color text, which is decorated with two added features: a black border that is a dashed line, and a thicker red color frame.

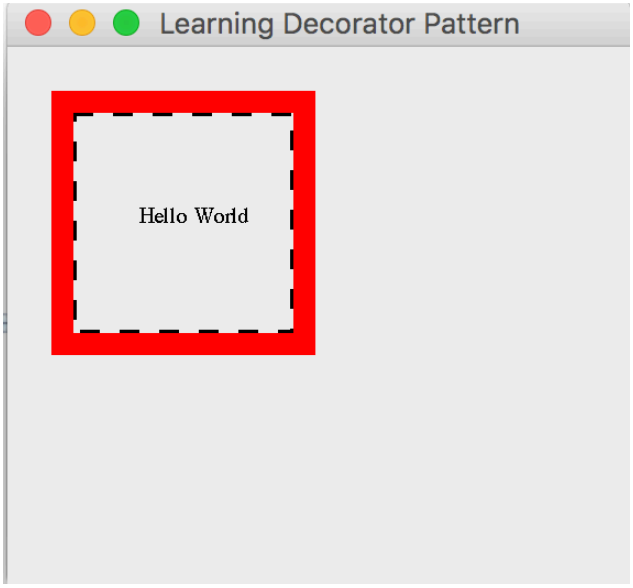


What to Do:

To implement this task refer to the following UML diagram. Also download file `DemoDecoratorPattern.java` that uses this pattern to test your work:

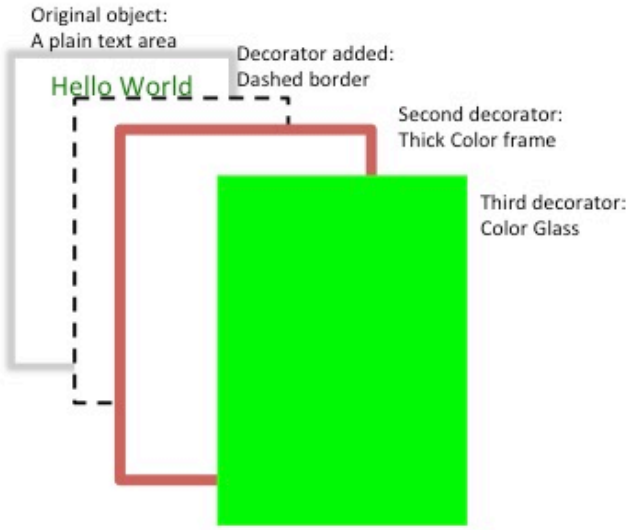


If your decorator pattern design is properly implemented the output of your program should look like this figure:



Exercise C

Now lets assume you need to add another decorator. But this time object text must be covered with transparent green-glass cover that it looks like:



What to Do:

You should add a new class called `ColouredFrameDecorator` that decorates the text area with the new decorating feature which a green glass. Now if you replace the current paintComponent method in the file DemoDecoratorPattern.java with the following code;

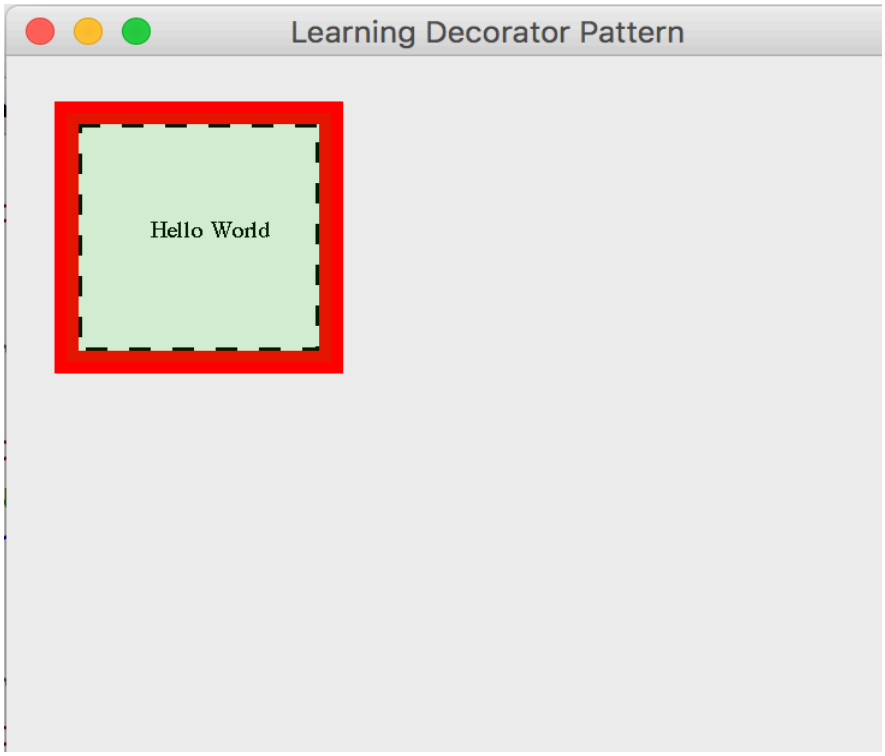
```
public void paintComponent(Graphics g){
 int fontSize = 10;
 g.setFont(new Font("TimesRoman", Font.PLAIN, fontSize));

  // GlassFrameDecorator info: x = 25, y = 25, width = 110, and height = 110

  t =  new ColouredGlassDecorator(new ColouredFrameDecorator(
       new BorderDecorator(t, 30, 30, 100, 100), 25, 25, 110, 110, 10), 25, 25,
       110, 110);

  t.draw(g);
}
```

The expected output will be:



Sample code that may help you for drawing graphics in java:

Sample Java code to create a rectangle at x and y coordinate of 30 and width and length of 100:

```
g.drawRect(30,  30,  100,  100);
```

Sample Java code to create dashed line:

```
Stroke dashed = new BasicStroke(3, BasicStroke.CAP_BUTT, BasicStroke.JOIN_BEVEL, 0, new
float[]{9}, 0);
Graphics2D g2d = (Graphics2D) g;
g2d.setStroke(dashed);
```

Sample Java code to set the font size

```
int fontSize = 10;
g.setFont(new Font("TimesRoman", Font.PLAIN, fontSize));
```

Sample Java code to fill a rectangle with some transparency level:

```
Graphics2D g2d = (Graphics2D) g;
g2d.setColor(Color.yellow);
g2d.setComposite(AlphaComposite.getInstance(AlphaComposite.SRC_OVER, 1 * 0.1f));
g2d.fillRect(25,  25,  110,  110);
```

What to Submit for Exercises B and C:

1. Copy and paste all your source codes and your program output as part of your lab report and submit it in PDF format.
2. Create and submit a zip file that contains your source code file (.java files)

Exercise D – Developing Singleton Pattern in C++

Objective:

The purpose of this simple exercise is to give you an opportunity to learn how to use Singleton Pattern in a C++ program.

When Do We Use It?

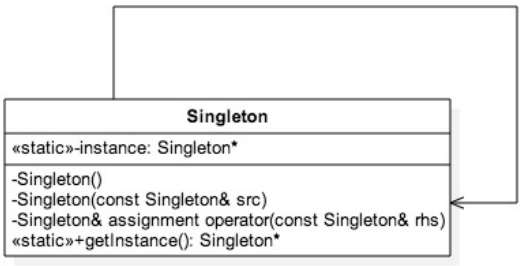
Sometimes it's important to have only one instance for a class. Usually singletons are used for centralized management of resources, where they provide a global point of access to the resources. Good examples include when you need to have a single:

- Window manger
- File system manager
- Login manager

The singleton pattern is one of the simplest design patterns: it involves only one class which is responsible to instantiate itself, to make sure it creates not more than one instance; in the same time it provides a global point of access to that instance. In this case the same instance can be used from everywhere, being impossible to invoke directly the constructor each time.

Implementation:

The implementation involves a static member in the "Singleton" class, a private constructor and a static public method that returns a reference to the static member. A class diagram that represents the concept of this pattern is as follows:

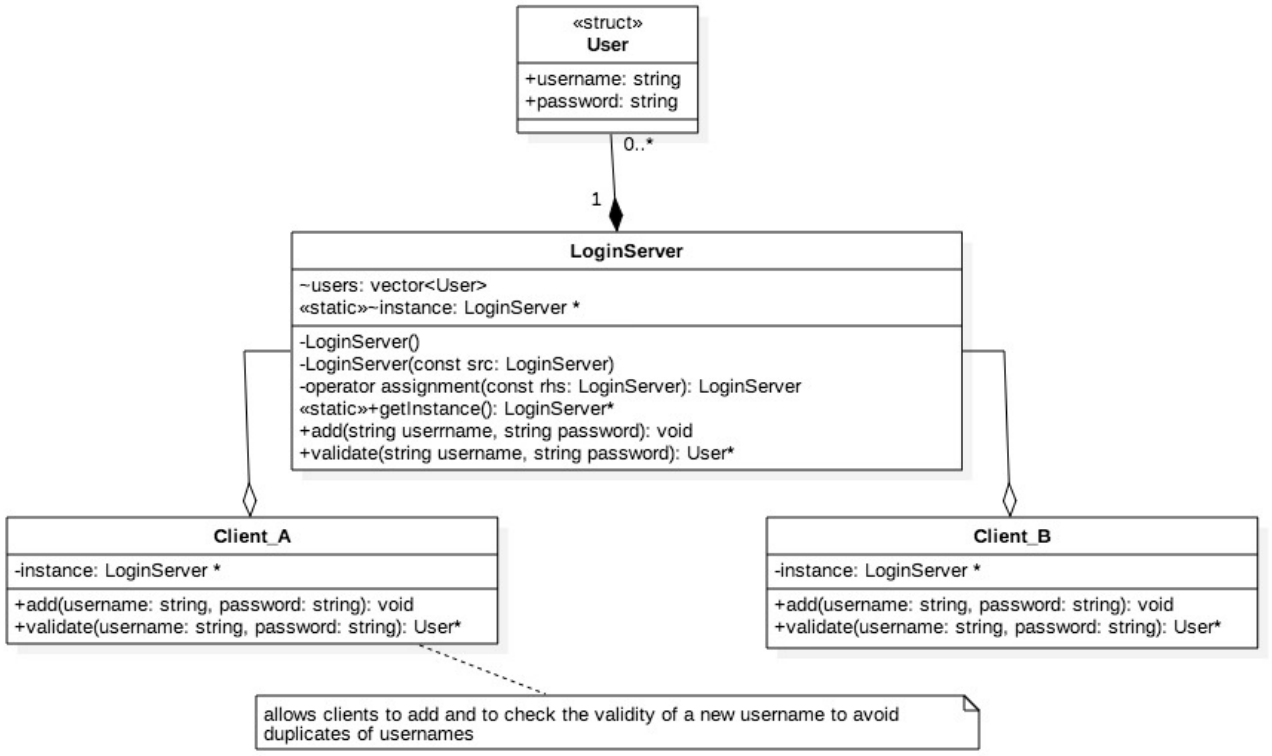


What to Do – Part I:

Step 1: download file main.cpp from D2L.

Step 2: write the class definitions as indicated in the following UML diagram: class `LoginServer`, `class Client_A`, `class Client_B`, and `struct User`.

Step 3: compile and run your classes with the given `main.cpp` to find out if your Singleton Pattern works.



What to Do – Part II:

Now you should test your code for an important fact about Singleton Pattern. At the end of the given file `main.cpp` there is a conditional compilation directive, `#if 0`. Change it to `#if 1` and report what happens:

- Does your program allow creating objects of LoginServer?
- If yes, is an object of LoginServer able to find user "Tim"?
- If no, why?

What to Submit:

1. Copy and paste all your source codes and your program output as part of your lab report and submit it in PDF format.
2. 2 answers to the question in Exercise D, part II
3. Create and submit a zip file that contains your source code file (.java files)