University of Calgary
Department of Electrical and Computer Engineering
Principles of Software Design - ENSF480
Lab 6 – Fall 2023
M. Moussavi, PhD, P.Eng

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: Decorator, and Singleton pattern.

Marking Scheme: (20 marks total):

- Exercise A: 16 marks
- Exercise B: 4 marks
- Exercise C: Not marked.

Note: Exercise C will not be marked but it is as important as other exercises and you are strongly urged to complete this exercise.

Due Dates:
Lab section 1 (B01) Monday Nov 6, before 11:59 PM.
Lab section 2 (B02): Wed, Nov 8, before 11:59 PM
Lab section 3 (B03): Friday, Nov 10 20, before 11:59 PM

Exercise A – Decorator Pattern (16 marks)

Read This First:

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

The concept of a decorator focuses on adding dynamically new futures/attributes to an object and particularly to add the new feature the original code and other added code for other features must remain 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

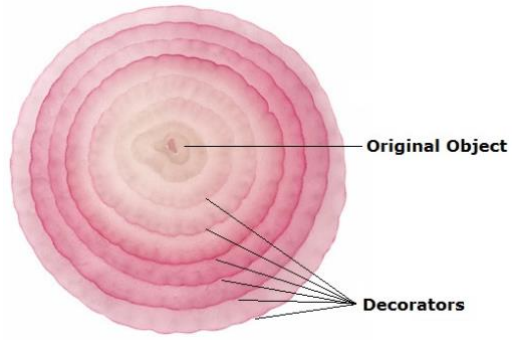


Figure from https://www.codeproject.com/Articles/176815/The-Decorator-Pattern-Learning-with-Shapes

Figure 2

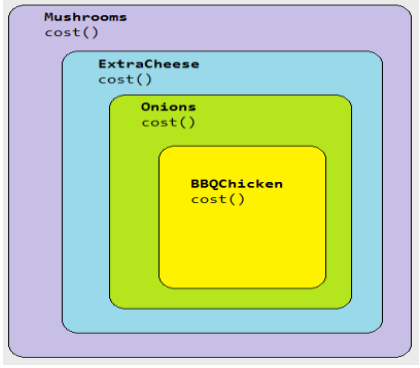


Figure from: http://conceptf1.blogspot.ca/2016/01/decorator-design-pattern.html

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 is 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:

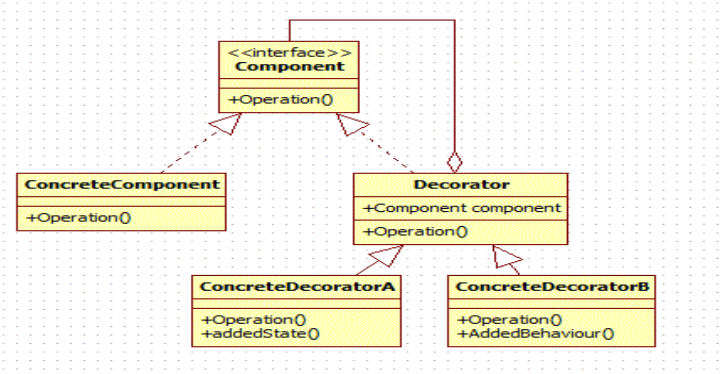


Figure 3

This diagram has three main elements:

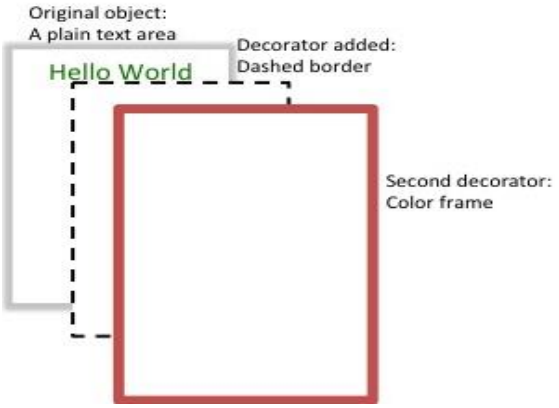
- The Component Interface defines the interface for objects that need their features to be added dynamically.
- The Concreter Component, implementing interface Component
- The Decorator implements the Component interface and aggregates a reference to the component. This is the important thing to remember,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

And, one or more Concrete Decorators, extended from Decorator

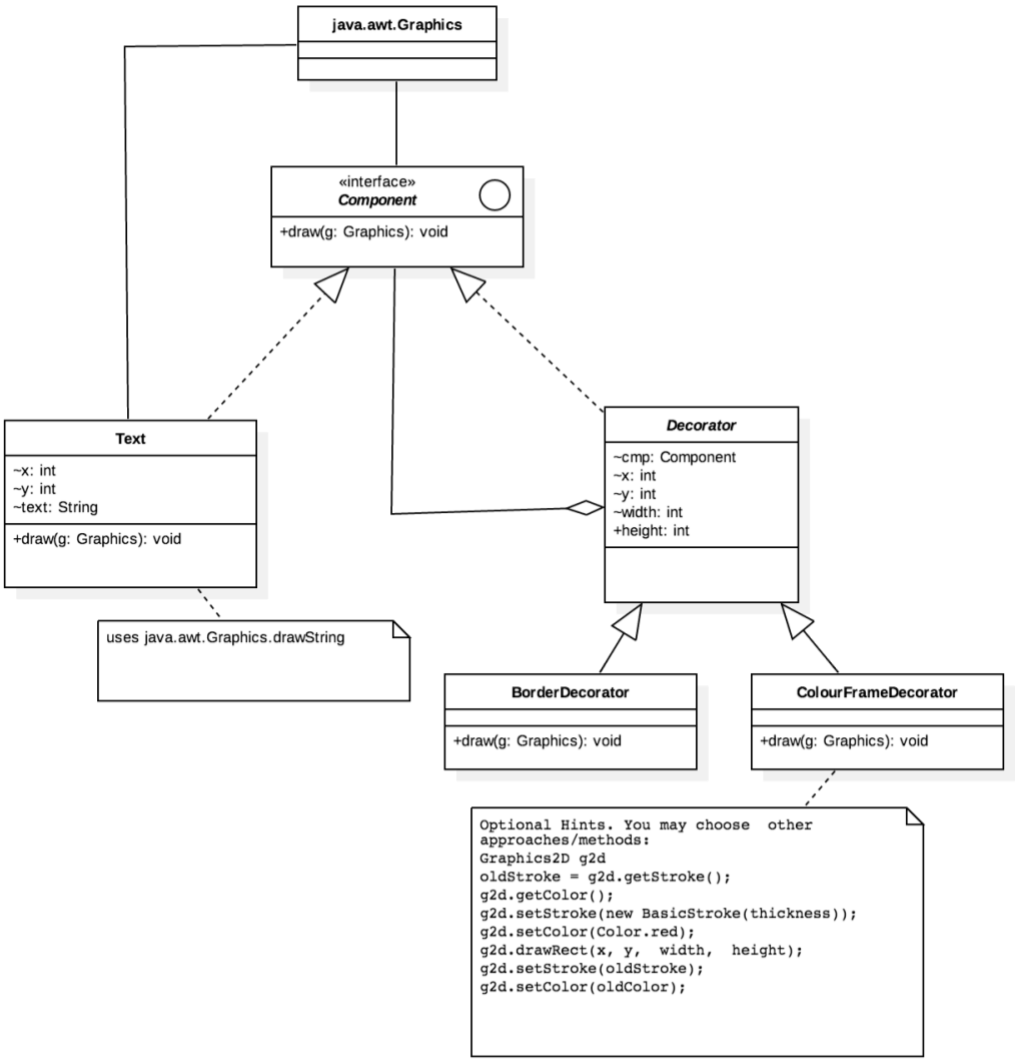
What to Do:

Let's 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:

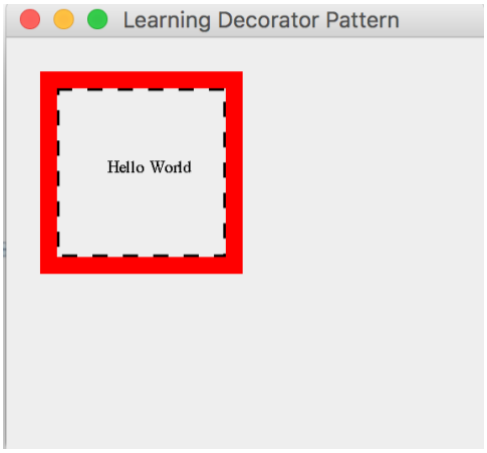
But the details of this component are displayed in the following figure that consists of a main object, a text area with green color text, which is decorated with two added features: a black border that is a dashed line, and a thicker red color frame.



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

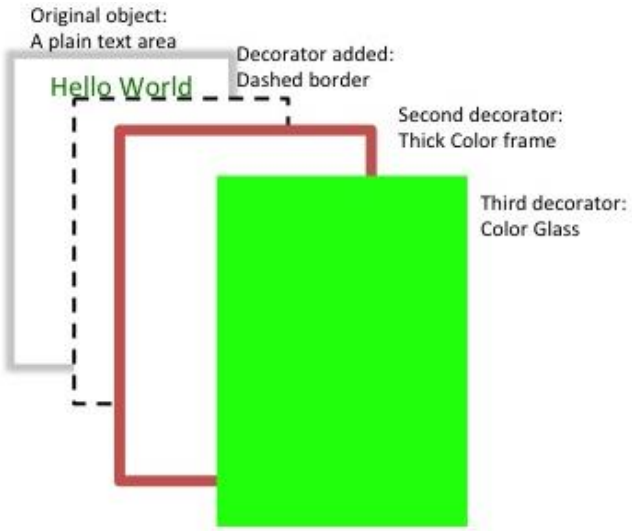


If your decorator pattern design is properly implemented the output of your program should look like this figure:



Exercise B (4 marks)

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



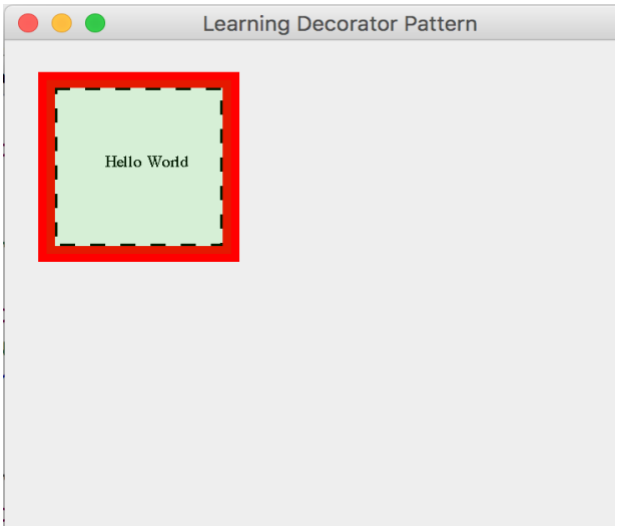
What to Do:

You should add a new class called `ColouredFrameDecorator` that decorates the text area with the new decorating feature which is 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);
```

Exercise C – 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

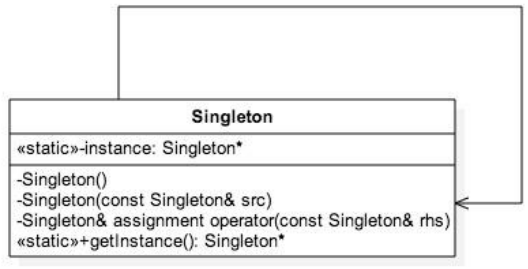
management of resources, where they provide a global point of access to the resources. Good examples include when you need to have a single:

- Window manager
- 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; at 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 the constructor directly 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:

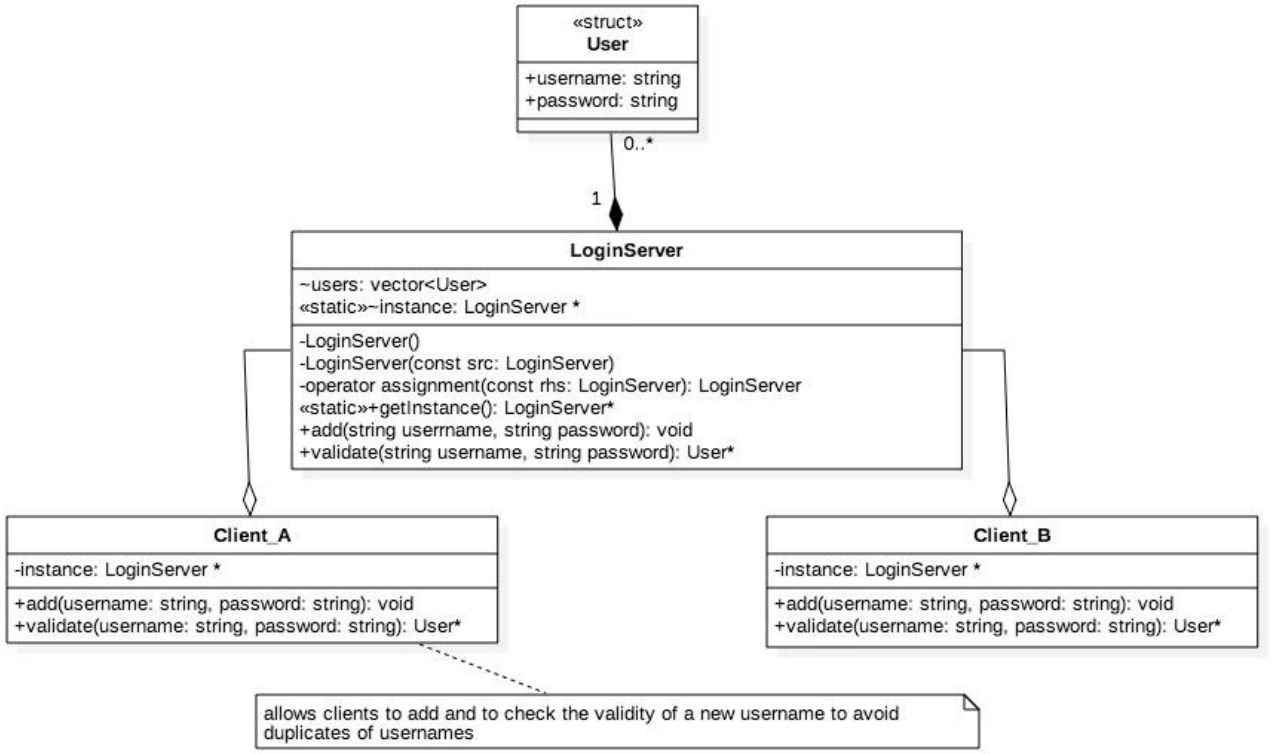


What to Do – Part I:

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 observe what happens:

- Does your program allow creating objects of LoginServer?
- If yes, is an object of LoginServer able to find user "Tim"?
- If not, why?