University of Calgary
Department of Electrical and Software Engineering
ENSF 614
Lab 6 - Fall 2023

M. Moussavi, PhD, PEng.

Notes:

- **This a group assignment and you can work with a partner. Groups 3 or more are not allowed.**
- The first exercise is related to C++ topics overloading operators and defining template class types. The next few exercises are related to the material that will be discussed in the second chapter of the course, "Design Pattens". In this new part of the course some of the lab exercise will be in Java and some in C++.

Due Dates: Friday Nov. 10, before 11:59 PM

Objectives:

The purpose of this lab is:

1. Understand the concepts of overloading operators and templates in C++.
2. Understanding and developing a simple program in Java that uses one of the important design pattern models called "Strategy Pattern", and "Observer Pattern".

Marking scheme:

The total mark for the exercises in this lab is: **59 marks**

- Exercise A: 10 marks
- Exercise B: 12 marks
- Exercise C: 2 marks
- Exercise D:15 mark
- Exercise E: 16 marks
- Exercise F: 4 marks
- Exercise G: (not marked). **Exercise G will not be marked, but it is as important as other exercises, and you are strongly recommended to complete the exercise and learn how Singleton pattern works.**

Exercise A: Templates in C++ (10 marks)

Read This First:

What is a Template Function?

A template function is a function that one or more of its argument types, and/or its return type are defined in a generic format that can be substituted with almost any actual built-in or user-defined type. These substitutions and creation of an instance of function is called instantiation of a template function. In the following example, function swap has been declared as a template function:

```
// function prototype
template  <class T > void swap (T* a, T*  b);

//function definition
template <class T>  void swap (T* a, T* b){
  T temp ;
  temp = *a;
  *a = *b;
  *b = temp;
}
```

The keywords in this code are bold. The remaining parts of the code are similar to regular C++ functions. As mentioned earlier, this generic or template definition, can be instantiated with different types. For example, the following code shows how the template function swap has been instantiated, and how its first call receives two integer pointers; while its second instance receives two float pointers. For more details about template function please refer to your lecture notes (slides) or a textbook such as C++ Primer, the latest version.

```
void main() {
   int n = 8, m = 6;
   float x = 4.5, y =5.5;
   swap (&n,&m);  //  instantiation for swapping two integer
   swap (&x, &y); //  instantiation for swapping two float
}
```

What is a Template Class?

Templates classes are usually used to design and develop container data structures, such as queues, stacks, linked lists, vectors and etc. The template and non-template classes behave essentially the same. Once the template class has been made known to the program, it can be used as a type specifier, the same as a non-template class. However, the only difference is that the use of a template class name must always include its parameter list enclosed by angle brackets (except within its own class definition). The following examples show the definition of a template class called Vector, and one of its member functions, getValue():

```
template <class T>
class Vector {
  public:
    Vector(int s);
    ~Vector();
    T getValue(int elem);
  private:
    T *array;
     int size;
};

template <class T>
T Vector<T>::getValue (int elem) {
return array[elem];
}
```

Sometimes a template function cannot provide a correct instance of the functions. In these cases, you should write a specialization of the function. For more details about the specialization of a template function, please refer to your lecture notes.

If there is a need to a specialization of a member function, it must be defined in the same file, after the definition of the template function.

What is Template Class Instantiation?

A template class can be instantiated by appending the full list of actual parameters enclosed by angle brackets to the template class name. The following example shows how the template class Vector can be instantiated.

```
int main() {
   Vector <char> x (3);
   Vector <double> y (40);
    …
    return 0;
}
```

What is an Iterator?

In general, an iterator is an object that provides a general method of successively accessing each element of a container type such as vectors or lists. In other words, it's an object that enables a programmer to traverse linear containers such as arrays, vectors and lists.

For example, we can write an iterator class called ***VectIter***, that provides overloaded operators to allow access to the elements of the Vector. Assume we have an object of ***VectIter*** called ***iter*** that initially is associated with the first element of an object of class ***Vector*** called ***myVector***, then a statement such as:

```
cout  << iter++;
```

displays the value of the first element of the vector and then advances the iterator object to the next object in the second element of the vector. This means that you must overload operator ++ (postfix) in the class IntVectorIter for this purpose.

Or, a statement such as:

```
cout << ++iter;
```

should advance the iterator object to the next element of the vector and then display the value of the previous element. This means that you should overload operator ++ (prefix) in the class IntVectorIter for this purpose.

Similarly, the following statement:

```
cout  << iter--;
```

displays the value of the current element of the vector and then moves the iterator object to the previous object in the vector. It means that you should overload operator -- (postfix) in the class IntVectorIter for this purpose.

What to Do:

Download files `mystring2.h, mystring2.cpp,` and `iterator.cpp` from D2L. Read the `iterator.cpp` carefully to understand the detail of the code in this file.

Now, you should:

1. Write the definition of operator functions declared in class `Vector`, and class `VectIter` that is embedded in the class `Vector`.

2. Convert the `Vector` class to a Template class. A class that virtually can create vectors of different types.

3. The given main function is partially compiled by the conditional compilation directives. Means a major segment of the code is commented out by using conditional compilation directive `#if 0`. Once you are done with the conversion of the class vector to a template class, change the `#if 0` directive to `#if 1` to include this part for compilation, and test your template class Vector and its iterator operators for data type: `int, Mystring`, and `char*`.

4. Make sure your function `ascending_sort` in the file `iterator.cpp` works properly for all data types.

5. **Note:** you may need to make other changes to the given source code to get your template class to work perfectly.

What to Submit:

Submit your source file iterator.cpp, as part of your lab report (PDF file), the program output that shows that your program works, and a zipped file that contains ALL your source files for exercise A.

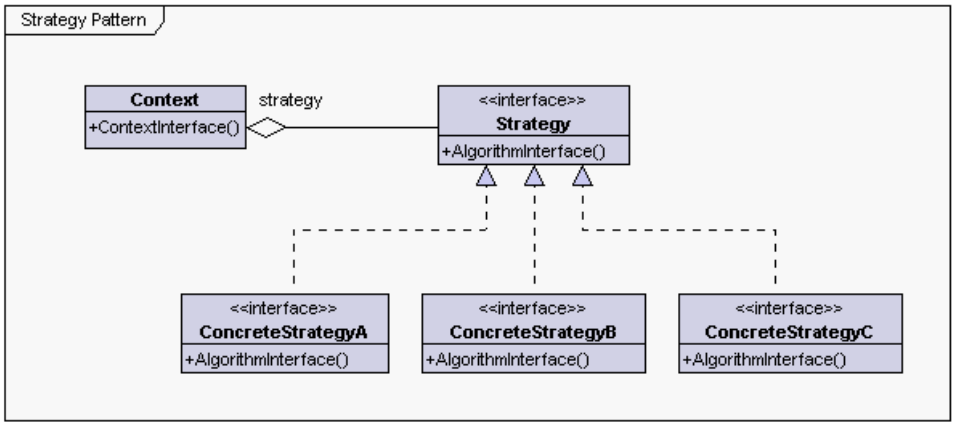
Exercise B - Strategy Design Pattern (12 marks)

The purpose of this exercise is to give you an opportunity to practice using Strategy Design Pattern in you program.

Read This First – Why and Where to Use Strategy Pattern

The subject of applying different strategies at different points of time (for good reasons) is a real demand in many real-world projects or processes. This is also a true requirement for many software applications. One of the most common applications of the Strategy Pattern is where you want to choose and use an algorithm at the runtime. A good example would be saving files in different formats, running various sorting algorithms, or file compression.
In summary, Strategy Pattern provides a method to define a family of algorithms, encapsulate each one as an object, and make them interchangeably used by a client.

Read This Second – A Quick Note on Strategy Pattern
The Strategy Pattern is known as a **behavioral** pattern - it's used to manage algorithms, relationships and responsibilities between objects. The definition of Strategy Pattern provided in the original Gang of Four book states:



The diagram shows that the objects of Context provide means to have access to objects that implement different strategy.

What to Do:

Assume as part of a team of the software designers you are working on an application that allows its clients to be able to use different sort methods for a class called `MyVector`. For the purpose of this exercise, you just need to implement two sort methods: `bubble-sort`, and `insertion-sort`. And of course, your design must be very flexible for possible future changes, in a way that at anytime the client objects should be able to add a new sort technique without any changes to the class `MyVector` (for the generic version will be MyVector<E>).

Please follow these steps:

Step 1: Download file `DemoStrategyPattern.java` form D2L. This file provides a client class in Java that must be able to use any sort techniques at the runtime.

Step 2: Download file called `Item.java` This is a class that represents data object. It means its private data member `item` can be used and sorted within the body of `MyVector` objects.

Step 3: Now your program must have the following classes:

- Class `MyVector` (or MyVector <E> which is also Bound to only Java Number type and its decedents). This class should have a private data member called `storageM` of type `ArrayList<Item>` (or ArrayList< Item<E> > for the generic version), which provides space for an array of certain size, and more data member as needed, and a second private data member called `sorter` that is a reference to an object of the Java interface Sorter (or interface Sorter <E> for the generic version).
  Class `MyVector` should also have at least two constructors as follows:

  - A constructor that receives only an integer argument, n, to allocate memory for an array with n elements.
  - A constructor that receives only an ArrayList object, `arr`, and makes `storageM` an exact copy of `arr`.

  Also must have at least the following methods, which are used in the client class `DemoStrategyPattern`:

  `public void` `add(Item value)`: That allows to add a new Item value to `storageM`

`public void setSortStrategy(Sorter s)`: That allows its private data member register with a an object that implements `Sorter`.

`public void performSort()`: That allows sort method of any sorter object to be called.

`public void display()`: That displays data values stored in `storage` on the screen in one line. For example: `1.0 2.0 3.0 4.0 5.0`

- Two Concrete classes called `BubbleSorter` and `InsertionSorter` that one implements a bubble sort algorithm and the other one implements insertion sort algorithm.

Exercise C (2 marks):

This is a smaller size exercise. The purpose of this exercise is to demonstrate how you can add a new algorithm to exercise A called `SelectionSorter` that uses selection-sort and can be used by the class client without making any changes to the class `MyVector`.

What to Submit for Exercise 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 into the D2L Dropbox.
2. Create and submit a zip file that contains your source code file (.java files) and submit it on the D2L Dropbox.

Exercise D (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.

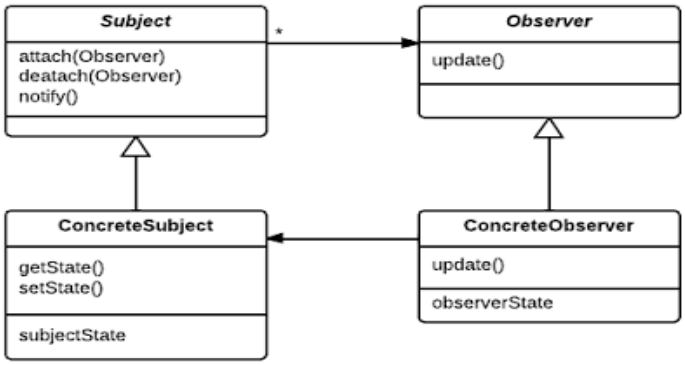


Figure 1

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.

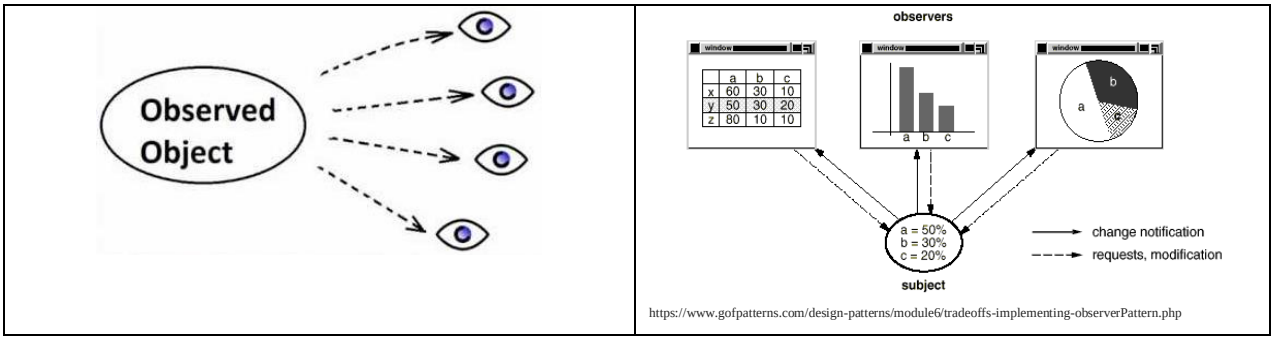
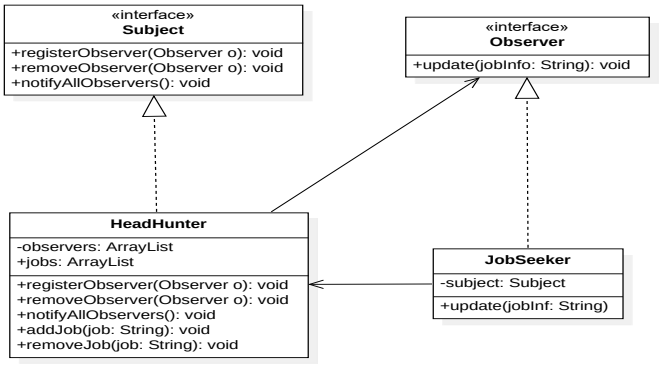


Figure 2

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 D?

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

Exercise E (15 marks):

Read This First:

A Brief Note on Application of 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:

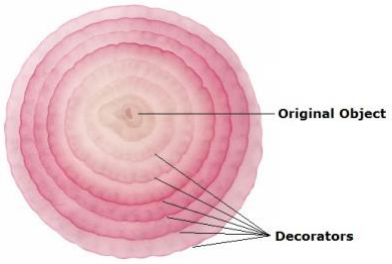


FIGURE 1

FIGURE FROM HTTPS://WWW.CODEPROJECT.COM/ARTICLES/176815/THE-DECORATOR-PATTERN-LEARNING-WITH-SHAPES

FIGURE 2

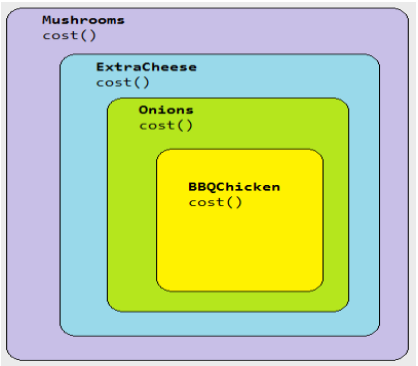


Figure from: http://conceptf1.blogspot.ca/2016/01/decorator-design-pattern.html

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:

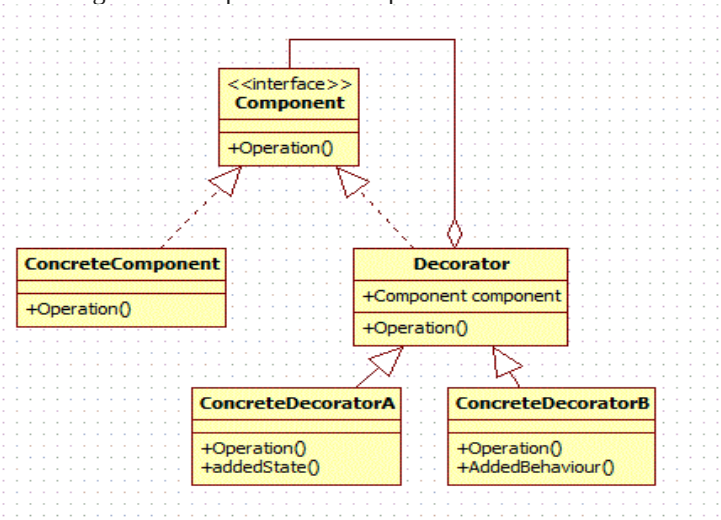


Figure 3

This diagram has three main elements:

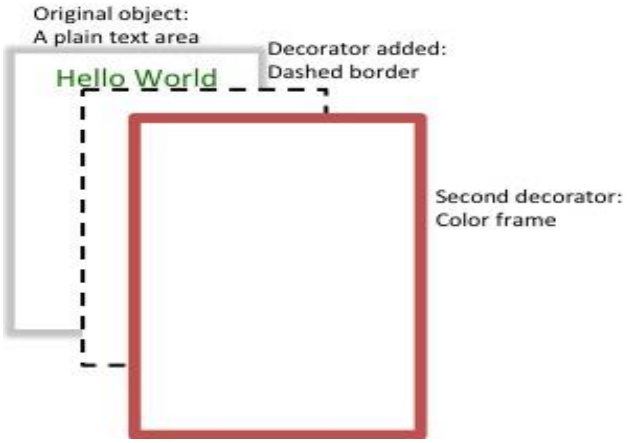
- The Component Interface, that defines the interface for objects that need their features to be added dynamically.
- The Concreter Component, implementing interface Component
- The Decorator, implementing the Component interface and aggregating a reference to the component. This is the important thing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

And, one or more Concrete Decorators, extended from Decorator

Read This Second:

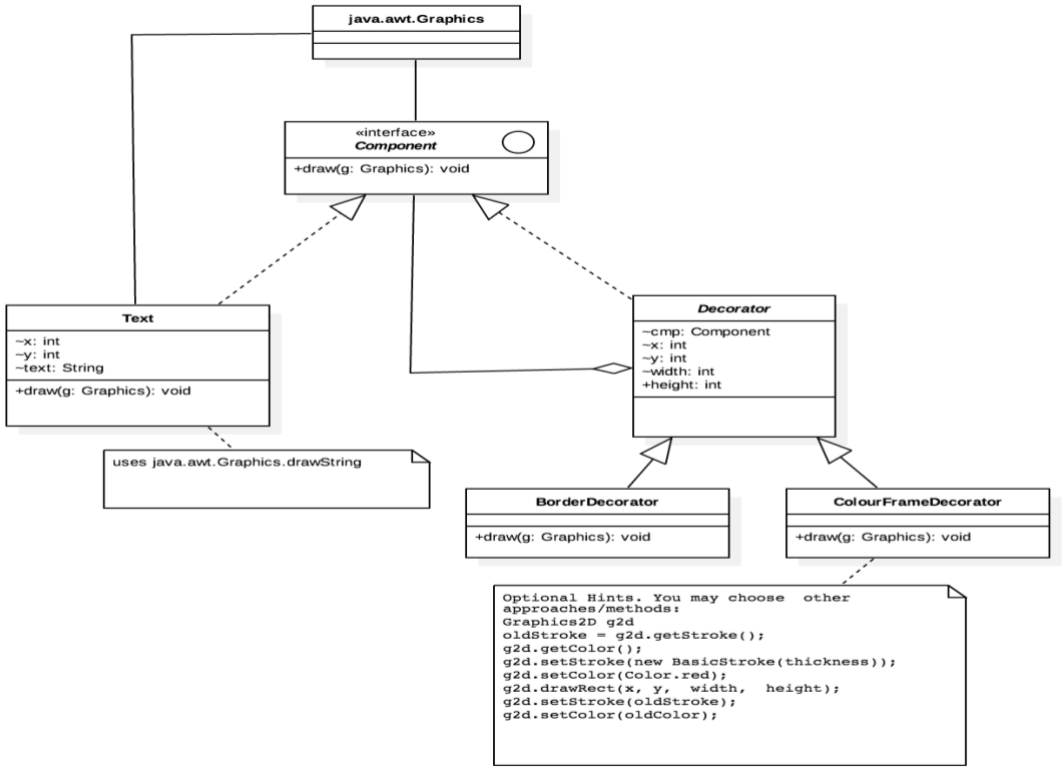
Lets, assume you are working as part of a software development team that you are responsible to write the required code for implementing a simple graphics component that is supposed to look like:

But the detail 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.

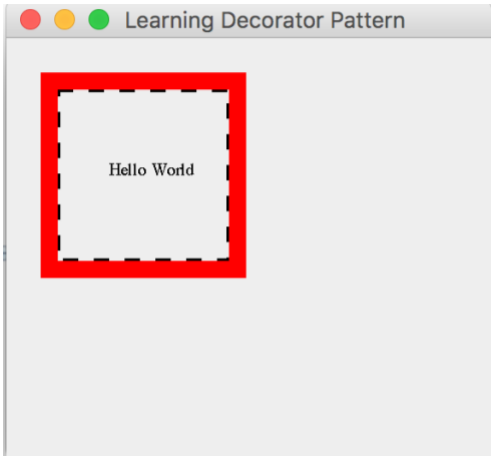


What to Do:

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

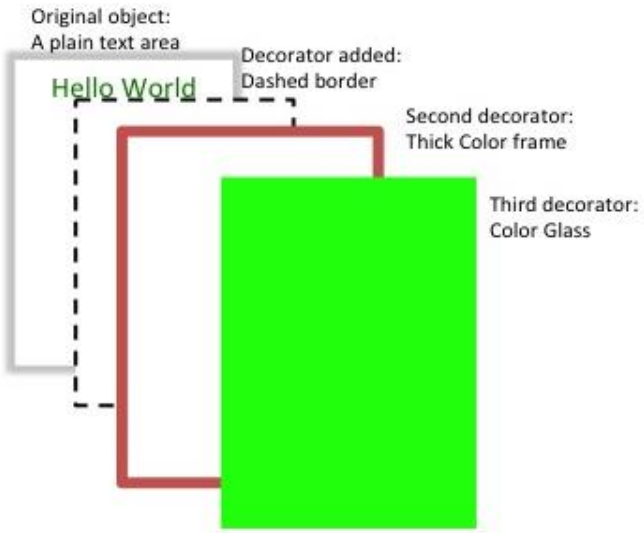


If your decorator pattern design is properly implemented the output of your program should look like this figure:



Exercise F (4 marks)

Now let's 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;

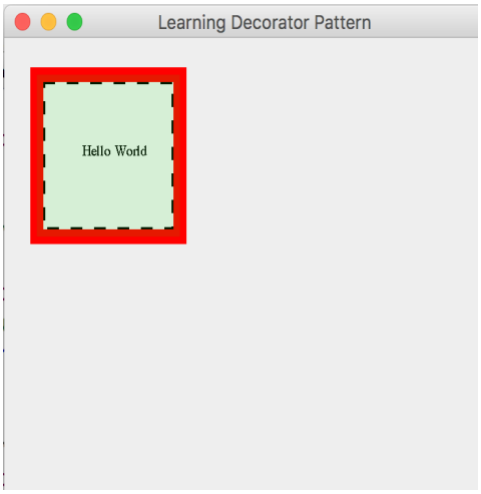
```
public void paintComponent(Graphics g){
 int fontSize = 10;
 g.setFont(new Font("TimesRoman", Font.PLAIN, fontSize));

  // 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 E and F:

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 G – 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.

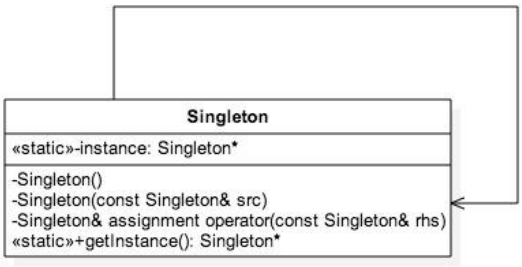
When Do We Use It?

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

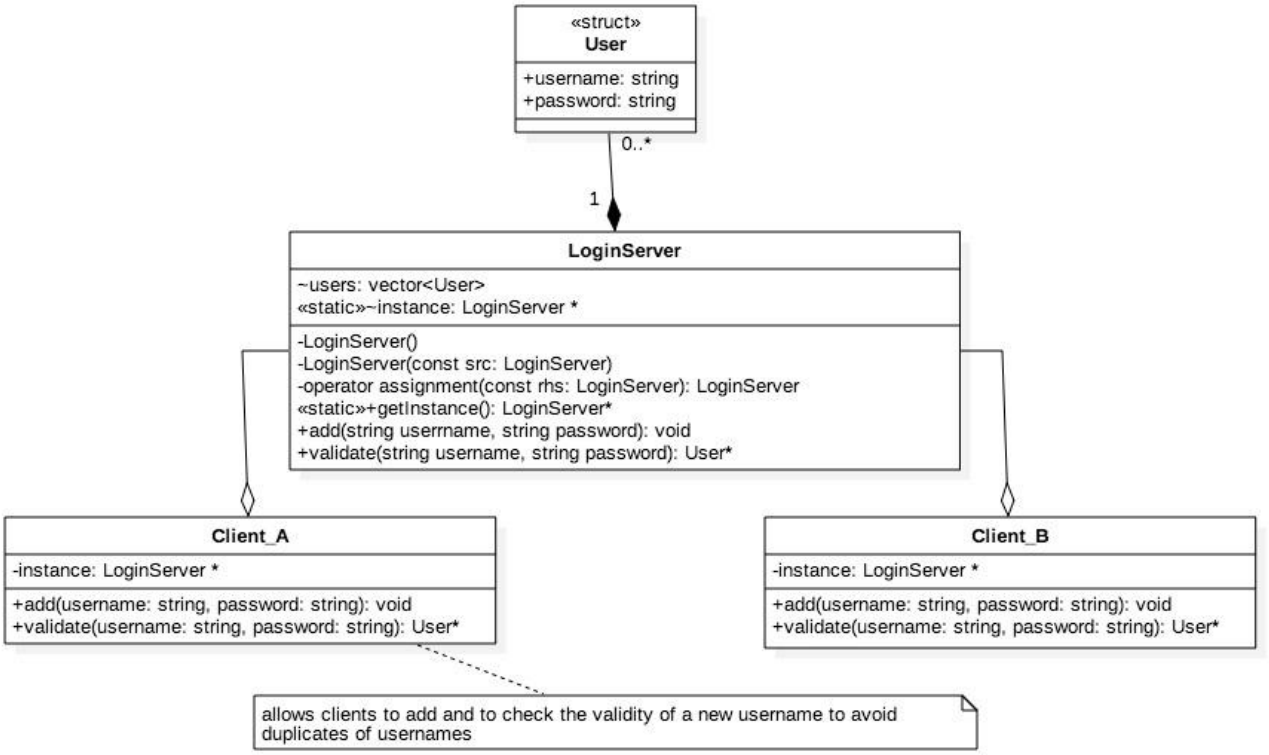
The singleton pattern is one of the simplest design patterns: it involves only one class which is responsible to instantiate itself, to make sure it creates not more than one instance; in the same time it provides a global point of access to that instance. In this case the same instance can be used from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