University of Calgary
Department of Electrical and Computer Engineering
Principles of Software Design - ENSF480
Lab 5 – Friday October 22, 2021

M. Moussavi, PhD, PEng

Marking Scheme: (34 marks total)

- Exercise A: 15 marks
- Exercise B: 4 marks
- Exercise C: 15 marks

Due Date:

Due to upcoming Quiz (on Wed October 27), to give you more time to concentrate on the material in this quiz, this lab will be due in two weeks.
Due date is Friday Nov 5, before 2 PM.

Exercise A - Design Pattern (15 marks)

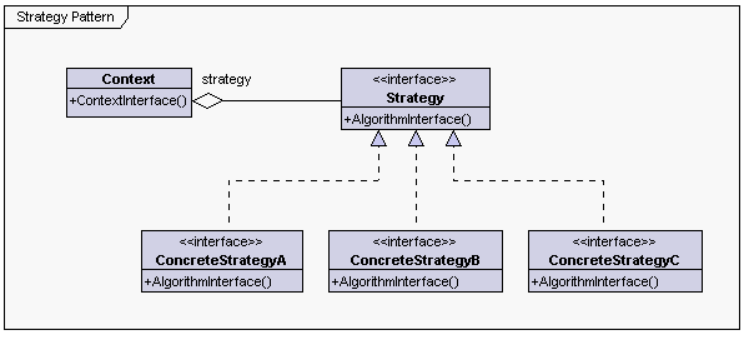
The purpose of this exercise is to give you an opportunity to practice using Strategy Design Pattern in you program.

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 **behavioural** 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 generic collection class called `MyVector<E>`. 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` or its possible decedent classes.

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 a generic class that is bound only to `Numeric` objects. It means its private data member `item` can be any Java Numeric type and its decedents, which includes: Integers, Doubles, Floats, etc.

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

- Class `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<E>>` that 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 class `Sorter.`

  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 that are used in the client class `DemoStrategyPattern:`

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

  `public void` `setSortStrategy(Sorter <E> 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 B (4 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<E>`.

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 into the D2L Dropbox.
2. Create and submit a zip file that contains your source code file (.java files) and submit it into D2L Dropbox.

Exercise C (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.

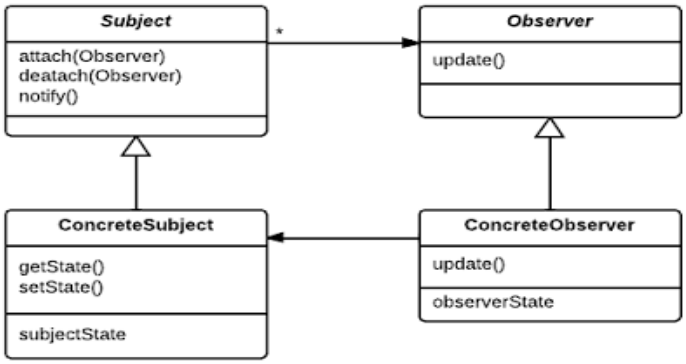


Figure 1

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.

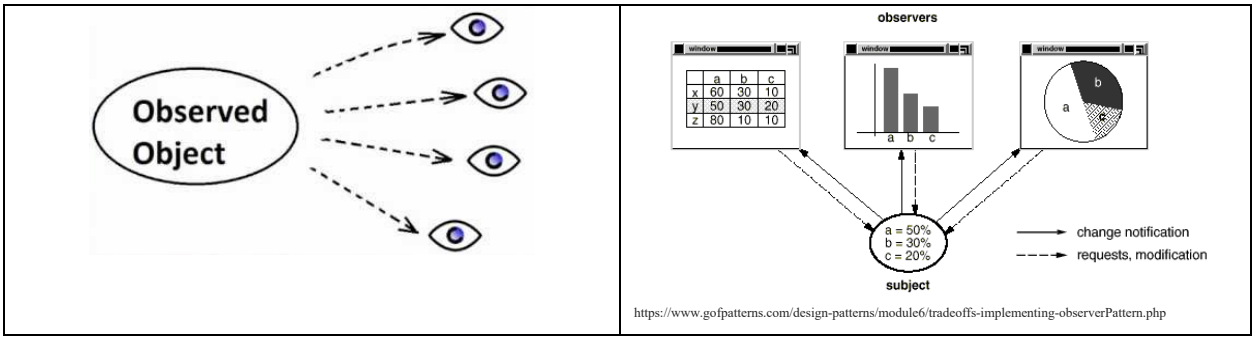
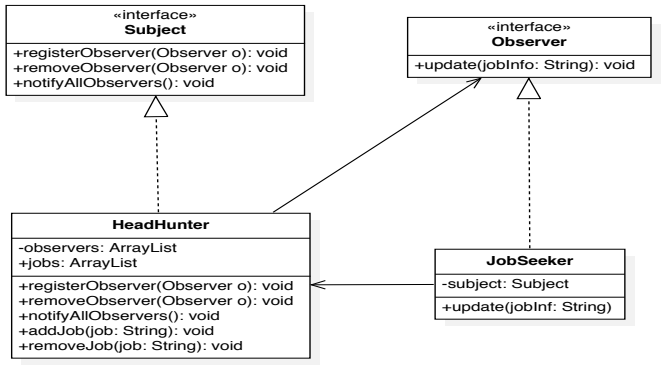


Figure 2

Figure 2 in this exercise 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 C?

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))