CPOA - Subject TD 5

PreReq	1. I know how to code in Java. 2. I know I need to think before I start coding. 3. I know several design patterns already.
ObjTD	Learn the **Observer** pattern.
Duration	**1** TD and **2** TPs

1. Motivations

In this exercise, we will explore the need to maintain consistency between related objects without making classes tightly coupled.

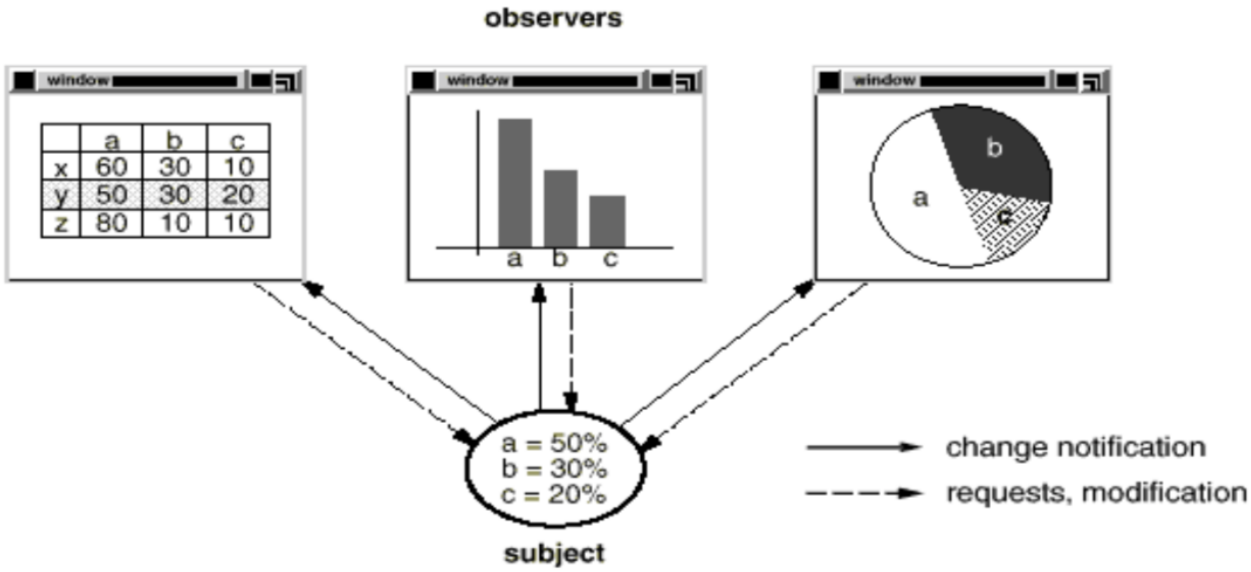

Figure 1. The classical Observer illustration

2. The Observer design pattern

2.1. Definition (sorry for the French)

Observateur définit une relation entre objets de type un-à-plusieurs, de façon que, lorsqu'un objet change d'état, tous ceux qui en dépendent en soient notifiés et soient mis à jour automatiquement.

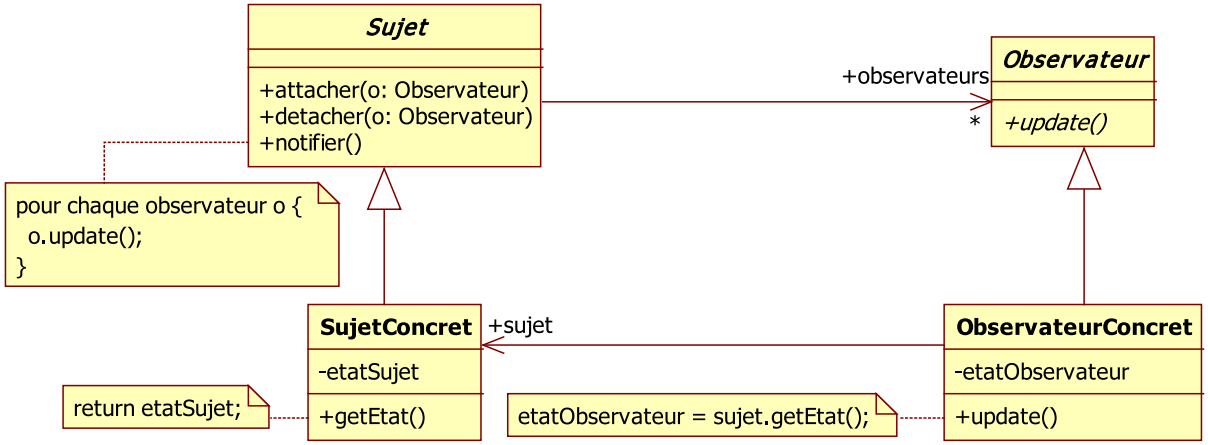

Figure 2. Modèle UML du patron Observer

Several precisions:

- the observers must be registered somehow
- the subject has to inform the observers (about changes).

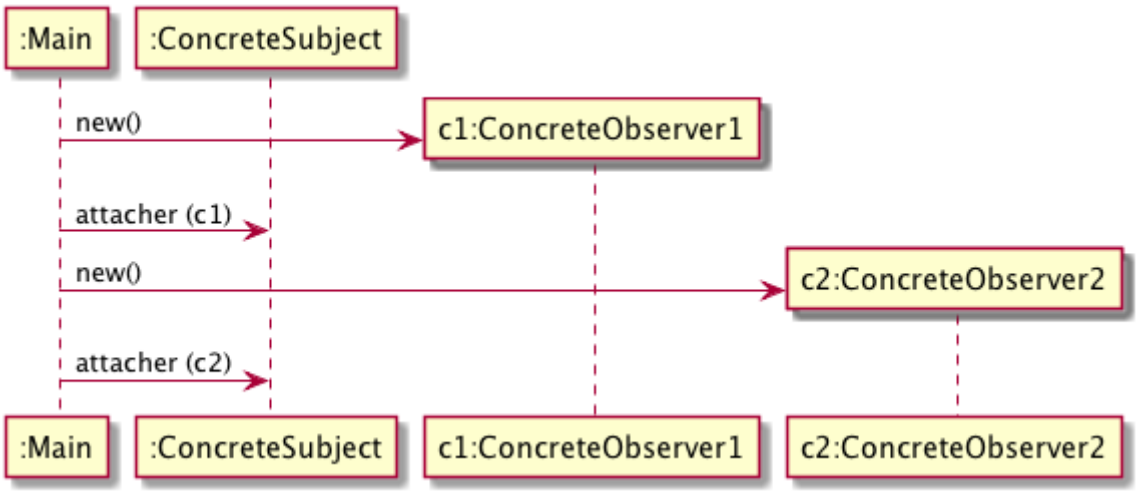

Figure 3. Example of use

There are several ways to implement the `attacher`, but in order to work properly it must be done:

- by an external object,
- by the `concreteObserver` itself.

2.2. Intent

- Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.

2.3. Applicability

Use the Observer pattern in any of the following situations:

- When an abstraction has two aspects, one dependent on the other. Encapsulating these aspects

in separate objects lets you vary and reuse them independently.

- When a change to one object requires changing others
- When an object should be able to notify other objects without making assumptions about those objects

2.4. Participants

- `Subject`
  - Keeps track of its `observers`
  - Provides an interface for attaching and detaching `Observer` objects
- `Observer`
  - Defines an interface for update notification
- `ConcreteSubject`
  - The object being observed
  - Stores state of interest to ConcreteObserver objects
  - Sends a notification to its observers when its state changes
- `ConcreteObserver`
  - The observing object
  - Stores state that should stay consistent with the subject's
  - Implements the Observer update interface to keep its state consistent with the subject's

2.5. Scenario

QUESTION

Let's see if you have undestood its utility. Make a sequence diagram that illustrate the use of the *Observer* pattern.

2.6. Benefits

- Minimal coupling between the Subject and the Observer
  - Can reuse subjects without reusing their observers and vice versa
  - Observers can be added without modifying the subject
  - All subject knows is its list of observers
  - Subject does not need to know the concrete class of an observer, just that each observer implements the update interface
  - Subject and observer can belong to different abstraction layers
- Support for event broadcasting
  - Subject sends notification to all subscribed observers

- Observers can be added/removed at any time

2.7. Liabilities

- Possible cascading of notifications
  - Observers are not necessarily aware of each other and must be careful about triggering updates
- Simple update interface requires observers to deduce changed item

3. Implementation Issues

3.1. Problems

Let's try to define an implementation.

If you already know or have used the `Observer` Java implementation, try to forget it!

QUESTION

Here are a list of possible issues. For each of them try to provide some answers:

1. How does the subject keep track of its observers?
2. What if an observer wants to observe more than one subject?
3. Who triggers the update?
4. Make sure the subject updates its state before sending out notifications
5. How much info about the change should the subject send to the observers?
6. Can the observers subscribe to specific events of interest?
7. Can an observer also be a subject?
8. What if an observer wants to be notified only after several subjects have changed state?

3.2. *Observer* in Java

Java provides the *Observable/Observer* classes as built-in support for the *Observer* pattern. The `java.util.Observable` class is the base `Subject` class. Any class that wants to be observed extends this class which:

- Provides methods to add/delete observers
- Provides methods to notify all observers
- A subclass only needs to ensure that its observers are notified in the appropriate mutators
- Uses a `Vector` for storing the observer references

The `java.util.Observer` interface is the `Observer` interface. It must be implemented by any observer class.

3.2.1. The java.util.Observable Class

QUESTION

Here is a list of methods of the `java.util.Observable` class:

```
1 public Observable()
2 public synchronized void addObserver(Observer o)
3 protected synchronized void setChanged()
4 public synchronized void deleteObserver(Observer o)
5 protected synchronized void clearChanged()
6 public synchronized boolean hasChanged()
7 public void notifyObservers(Object arg)
8 public void notifyObservers()
```

Match them with their corresponding comments:

1. Adds an observer to the set of observers of this object
2. If this object has changed, as indicated by the hasChanged() method, then notify all of its observers and then call the clearChanged() method to indicate that this object has no longer changed. Each observer has its update() method called with two arguments: this observable object and the arg argument. The arg argument can be used to indicate which attribute of the observable object has changed.
3. Indicates that this object has changed
4. Indicates that this object has no longer changed, or that it has already notified all of its observers of its most recent change. This method is called automatically by `notifyObservers()`.
5. Same as above, but the arg argument is set to null. That is, the observer is given no indication what attribute of the observable object has changed.
6. Tests if this object has changed. Returns true if setChanged() has been called more recently than clearChanged() on this object; false otherwise.
7. Construct an `Observable` with zero observers
8. Deletes an observer from the set of observers of this object

3.2.2. The java.util.Observer Interface

java.util.Observer

```
/**
* This method is called whenever the observed object is changed. An
* application calls an observable object's notifyObservers method to have all
* the object's observers notified of the change.
*
* Parameters:
* o - the observable object
* arg - an argument passed to the notifyObservers method
*/
public abstract void update(Observable o, Object arg)
```

3.3. Observable/Observer Example

ConcreteSubject.java

```
/**
* A subject to observe!
*/
public class ConcreteSubject extends Observable {

  private String name;
  private float price;
  public ConcreteSubject(String name, float price) {
    this.name = name;
    this.price = price;
    System.out.println("ConcreteSubject created: " + name + " at " + price);
  }
  public String getName() {return name;}
  public float getPrice() {return price;}
  public void setName(String name) {
    this.name = name;
    setChanged();
    notifyObservers(name);
  }
  public void setPrice(float price) {
    this.price = price;
    setChanged();
    notifyObservers(new Float(price));
  }
}
```

NameObserver.java

```
// An observer of name changes.
public class NameObserver implements Observer {
  private String name;
  public NameObserver() {
    name = null;
    System.out.println("NameObserver created: Name is " + name);
  }
  public void update(Observable obj, Object arg) {
    if (arg instanceof String) {
      name = (String)arg;
      System.out.println("NameObserver: Name changed to " + name);
    } else {
      System.out.println("NameObserver: Some other change to
      subject!");
    }
  }
}
```

PriceObserver.java

```
// An observer of price changes.
public class PriceObserver implements Observer {
  private float price;
  public PriceObserver() {
    price = 0;
    System.out.println("PriceObserver created: Price is " + price);
  }
  public void update(Observable obj, Object arg) {
    if (arg instanceof Float) {
      price = ((Float)arg).floatValue();
      System.out.println("PriceObserver: Price changed to " +
      price);
    } else {
      System.out.println("PriceObserver: Some other change to
      subject!");
    }
  }
}
```

TestObservers.java

```
1 // Test program for ConcreteSubject, NameObserver and PriceObserver
2 public class TestObservers {
3   public static void main(String args[]) {
4     // Create the Subject and Observers.
5     ConcreteSubject s = new ConcreteSubject("Corn Pops", 1.29f);
6     NameObserver nameObs = new NameObserver();
7     PriceObserver priceObs = new PriceObserver();
8     // Add those Observers!
9     s.addObserver(nameObs);
10    s.addObserver(priceObs);
11    // Make changes to the Subject.
12    s.setName("Frosted Flakes");
13    s.setPrice(4.57f);
14    s.setPrice(9.22f);
15    s.setName("Sugar Crispies");
16  }
17 }
```

QUESTION

Donnez la trace d'exécution du code ci-dessus.

3.4. Limitations of Observable/Observer in Java

Problem

Suppose the class which we want to be an observable is already part of an inheritance hierarchy, e.g.,:

```
1 class SpecialSubject extends ParentClass
```

QUESTION

Since Java does not support multiple inheritance, how can we have `ConcreteSubject` extend both `Observable` and `ParentClass` ?

1. Propose a solution
2. Define its corresponding class diagram
3. Write the Java implementations (mainly `SpecialSubject`)

Two methods of `java.util.Observable` are **protected** methods: `setChanged()` and `clearChanged()`.

You can use the following test code:

```
// Test program for SpecialSubject with a Delegated Observable.
public class TestSpecial {
  public static void main(String args[]) {
    // Create the Subject and Observers.
    SpecialSubject s = new SpecialSubject("Corn Pops", 1.29f);
    NameObserver nameObs = new NameObserver();
    PriceObserver priceObs = new PriceObserver();
    // Add those Observers!
    s.getObservable().addObserver(nameObs);
    s.getObservable().addObserver(priceObs);
    // Make changes to the Subject.
    s.setName("Frosted Flakes");
    s.setPrice(4.57f);
    s.setPrice(9.22f);
    s.setName("Sugar Crispies");
  }
}
```

That should produce the following results:

```
ConcreteSubject created: Corn Pops at 1.29
NameObserver created: Name is null
PriceObserver created: Price is 0.0
PriceObserver: Some other change to subject!
NameObserver: Name changed to Frosted Flakes
PriceObserver: Price changed to 4.57
PriceObserver: Price changed to 9.22
PriceObserver: Some other change to subject!
NameObserver: Name changed to Sugar Crispies
```

QUESTION

After studying the solution with your favorite teacher:

1. Do you see a problem to this implementation?
2. Can you provide a solution?

QUESTION

Hence how to you modify the test?

To go further…

QUESTION

- For the `price` and `name` example, we could have decided that only the `PriceObserver` are notified of a price variation an that only the `NameObserver` are notified of a change of name. How would you do that?
- Make some connections with what you hav practiced today and the listeners used in Swing.