Observer Design Pattern

In this segment, we introduce the concept of *behavioral* design pattern by describing the `Observer` pattern. We start by reviewing some behavioural patterns. Then, we completely define the functionality of the `Observer`. Subsequently, we outline a concrete implementation of an `Observer` pattern.

1 Behaviour design pattern

Behaviour patterns describe common communication strategies between objects. They show how objects should easily interact being loosely coupled. Hence, the behaviour patterns are concerned not only with channels of communication, but also with the assignment of responsibilities between objects. Some of the most important behaviour design patterns are:

1. *`Command`* - the pattern defines how an object should encapsulate state and behaviour to perform actions
2. *`Iterator`* - decouples algorithms from containers, thus providing a way to access elements of a container without exposing the underlying representation of the container.
3. *`Memento`* - provides the capability of an object to be restored to its previous state. It captures the object internal state without violating the principles of encapsulation.
4. *`Observer`* - it is a key element of model-view-controller (MVC) architecture pattern, where an object called subject is observed by a set of objects called observers.
5. *`Strategies`* - enables algorithms to be selected dynamically at runtime. It is also called policy pattern.
6. *Visitor* - defines operations that could be applied to a group of objects. The operations are defined and performed by the visitor rather than the classes being visited.

2 Observer intent

The `Observer` pattern allows an object called `Subject` to notify a set of objects called `Observers` about any change in its state. The `subject` is sometimes called `Observable` and `Observers` are called `Listeners.`

[1]3 Occurring problem

The problem arises when one wants to automatically update a group of objects based on the changes that occurred in a particular object. Therefore, when there is a one-to-many relationship between objects, one object must maintain a list of objects that should be notified when the change occurs in that particular object.

For example, let us suppose there is a class that defines the effects of climate change by storing temperature variations. Let us further suppose that there are a group of classes that can display the changes in temperature values. In order to design the one-to-many relationship, we can use the `Observer` pattern.

4 Observer pattern elements

To properly design and implement the `observer` pattern, one needs to define its elements. Here are the elements, which are depicted in the UML :

1. `Observable (Subject)` interface defines the behaviour of an `Observable` object that needs to keep a list of `Observer` objects and notify them.

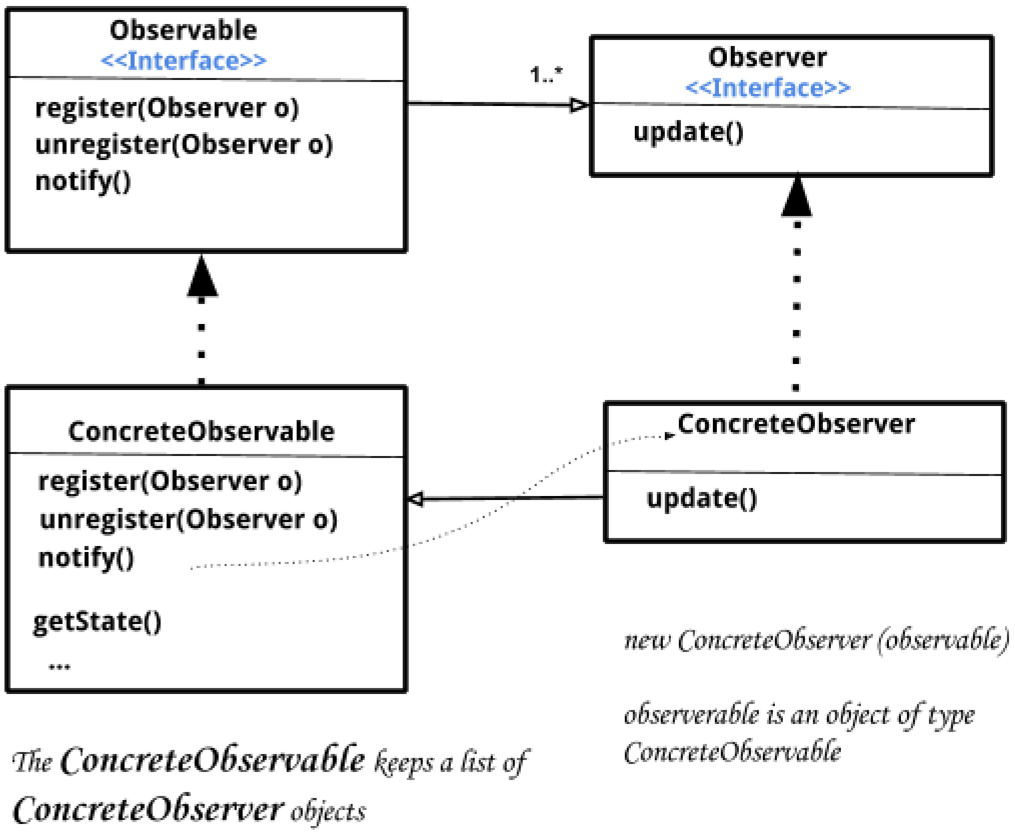
2. `Observer (Listener)` interface must define a method that can update the observer objects when the state of the `Observable` object has been changed. There is a one-to-many relationship between `Observable` object and `Observer` objects.

3. `ConcreteObservable` class implements the `Observable` interface and also implements a method to retrieve the state of the `ConcreteObservable` object. It implements the `Target` interface. It modifies the client request and uses the `Adaptee` class to invoke its functionality. It obtains the results from the `Adaptee` class, but with a request specified by the `Target` interface.

4. `ConcreteObserver` class implements the `Observer` interface. Sometimes, this class violates the “single responsibility principle” by using a reference to `ConcreteObservable` object to obtain directly the data from the `ConcreteObservable` object.

[1] Observer pattern in networking: each client is registered as an observer once it is connected to the server

Observer Design Pattern



5 Observer pattern example

Let us suppose that we want inform learners that are enrolled in a course about the course content updates. Therefore, we want to use the `Observer` pattern, where the `Observable` is the course and the `Observer` objects are the various type of learners.

6 Example implementation

Our implementation defines the interface **`Observable`** and the implementation class called **`Course.`** Furthermore, the **`Observer`** interface is implemented in the various observers, such as **`OnlineLearner`** and **`InclassLearner.`**

The `Course (Observable)` type keeps track of the list of objects of type `Observer,` the listeners. Anytime the content of the course is changed, the observers (learners) are notified. Thus, learners are able to read the new course content, regardless of the learner type.

Observer Design Pattern

```
/*********************************************************************
 *  Compilation:  javac Observable.java
 *
 *  The Observable contract of the Observer Design Pattern
 *
 *********************************************************************/
public interface Observable<T> {
    /**
     * add new observer to the collection of observers
     */
    public void registerObserver(Observer<T> o);

    /**
     * remove an observer from the collection of observers
     */
    public void unregisterObserver(Observer<T> o);

    /**
     * method to inform Observer about its new state
     */
    public void notifyObserver();
}
```

```
/*********************************************************************
 *  Compilation:  javac Observer.java
 *
 *  The Observer contract of the Observer Design Pattern
 *
 *********************************************************************/
public interface Observer<T> {
    /**
     * method invoked by Observable when its state changed
     */
    public void update();
}
```

Observer Design Pattern

In our example, the `ConcreteObservable` class that implements the `Observable` interface is the `Course` class.

```java
import java.util.ArrayList;
import java.util.List;
/*************************************************************************
 *  Compilation:  javac Course.java
 *
 *  ConcreteObservable - Course class implements Observable
 *
 *************************************************************************/
public class Course<T> implements Observable<T> {

    /**
     * the collection of observers objects
     **/
    private List<Observer<T>> observers;
    /**
     * the observable data
     **/
    private T content;

    /**
     * Constructor - it builds an object of type ConcreteObservable
     * with the initial data
     * @param content an object of type T
     */
    public Course(T content) {
        this.observers = new ArrayList<Observer<T>>();
        this.content = content;
    }

    /**
     * Add an object of type Observer to the collection of observers
     * @param o the object of type Observer
     */
    @Override
    public void registerObserver(Observer<T> o) {
       observers.add(o);
    }

    /**
     * Remove an object of type Observer from the collection of observers
     * @param o the object of type Observer
     */
    @Override
    public void unregisterObserver(Observer<T> o) {
        observers.remove(o);
    }


    /**
     * Notify all objects of type Observer when
     * the state of Observable change
     * - the state of Course is changed
     */
    @Override
    public void notifyObserver() {
        for(Observer<T> observer : observers)
            observer.update();
    }

    /**
     * Get the data of the Observable
     * @return the data of the Observable - an object of type T
     */
    public T getContent() {
        return content;
    }


    /**
     * Set the data of the Observable
     * @param content an object of type T- the Observable data
     */
    public void setContent(T content) {
        this.content = content;
    }
}
```

Observer Design Pattern

In our example, the ConcreteObserver classes that implement the Observer interface are the OnlineLearner and InclassLearner classes. The classes are identical, so we could have designed a generic class Learner<L>, where type parameter L is replaced by the type of learner.

```
/*************************************************************************
 *  Compilation:  javac OnlineLearner.java
 *
 *  ConcreteObserver - OnlineLearner class implements Observer
 *
 *************************************************************************/
public class OnlineLearner<T> implements Observer<T> {

    /**
     * the observable object
     * is needed to obtain directly data from Observable
     **/
    private Course<T> course;

    /**
     * Constructor - it builds an object of type Observer
     *
     * @param course a Observable object
     */
    public OnlineLearner(Course<T> course) {
        this.course = course;
    }


    /**
     * Called by Observable to notify the Observer when its state was changed
     * The observer renders the new data (the new Observable state)
     */
    public void update() {
        System.out.println("Online learner reads: " + course.getContent() );
    }
}
```

The ObserverPatternExample is self-explanatory: we create the Observable (subject) object of type Course, then we create objects of type Observer, i.e., the learners. Next, we register them and notify them anytime a change in the course content occurs.

Observer Design Pattern

```java
/*************************************************************************
 *  Compilation:  ObserverPatternExample.java
 *  Execution:    java ObserverPatternExample
 *
 *  This class defines a Course and two types of Learners
 *  It uses the Observer pattern to notify learners
 *  when the course content is modified, updated.
 *
 *  1. create the Observable (subject) object of type Course
 *  2. create two Observer objects: inclassLerner and onlineLerner
 *  3. register learners with the course
 *  4. notify learners
 *  5. add new content (second lesson)
 *  6. notify learners
 *  7. unregister an Observer (the inclass lerner)
 *  8. add new content (third lesson)
 *  9. notify learners
 *
 *  Output:
 *  In class learner reads: First Lesson
 *  Online learner reads: First Lesson
 *  In class learner reads: Second Lesson
 *  Online learner reads: Second Lesson
 *  Online learner reads: Third Lesson
 *************************************************************************/
public class ObserverPatternExample {

    public static void main(String[] args) {

        // 1. create the Observable (subject) object of type Course
        Course<String> subject = new Course<>("First Lesson");

        // 2. create two Observer objects: inclassLerner and onlineLerner
        Observer<String> inclassLerner = new InclassLearner<String>(subject);
        Observer<String> onlineLerner = new OnlineLearner<String>(subject);

        // 3. register learners with the course
        subject.registerObserver(inclassLerner);
        subject.registerObserver(onlineLerner);

        //4. notify learners
        subject.notifyObserver();

        //5. add new content (second lesson)
        subject.setContent("Second Lesson");

        //6. notify learners
        subject.notifyObserver();

        //7. unregister an Observer (the inclass lerner)
        subject.unregisterObserver(inclassLerner);

        //8. add new content (third lesson)
        subject.setContent("Third Lesson");

        //9. notify learners
        subject.notifyObserver();

    }
}
```

7 Chapter resources

The basic online documentation for the design patterns is available at: https://en.wikipedia.org/wiki/Design_Patterns

More about the design patterns in Java, including examples and implementations, can be found at: https://www.tutorialspoint.com/design_pattern/observer_pattern.htm

8 Exercises

1. In the observer example provided, replace the **`OnlineLearner`** and **`InclassLearner`** class with the generic class

```
public class Learner<L> {
  ...
}
```

What else needs to be changed? Refactor the given example so that you can create three learner types.

2. Develop an Observer pattern implementation for solving the following problem:

The client wants to add two numbers in the `binary`, `decimal`, and `hexadecimal` format, so that anytime the numbers change, the result will be displayed in all formats.

9 A challenging exercise

1. Design and implement a solution for the following problem statement:

Let us suppose that the lectures of a course are stored in a file. You want to inform all students when the course content is changed.

Hint:
Apply the observer design pattern.