Observer Design Pattern

Eric McCreath
Overview

In this lecture we will:

- motivate and define the Observer design pattern,
- give some examples of where this design pattern is used,
- provide a template for implementing your own classes that use this pattern, and
- a "live code" example.
The Context

- Normally when your program uses a library (or another class) it constructs an object that references that library. To then get the library to do something you call methods with respect to this object.
- The great thing about this approach is the library is not coupled to your program and it can be reused for other programs.

Your Program

```java
TheLibrary lib = new TheLibrary();
lib.doSomething();
```
The Context

- In some cases you want the library to inform your program of an event that occurred in the library.
- Now we could provide the library a reference to your program and then when the event occurs in the library it simply calls a method in your program. The down side of this approach is that the library is now tightly coupled with your program (it only works with your program and you would need to modify the Library to use it in another situation).

```java
Your Program

TheLibrary lib = new TheLibrary(this);

update() {
    :
}

TheLibrary

YourProgram yp;

constructor(YourProgram yp) {
    this.yp = yp;
}

yp.update()
```
The observer design pattern has the code using the library, register observers with the library. When the library wishes to notify the code using the library of an event, it simply calls the notification method in all the registered observers (in the diagram below it is the update method).

The advantage is this library is no longer tightly coupled to your program. Also different parts of your program can register a number of observers which can all be notified of an event.

Your Program

```java
TheLibrary lib = new TheLibrary();
lib.registerObserver(this);
update();
```

```java
YourProgram
```

```
TheLibrary
```

```
ArrayList<Observer> observers;
constructor() {
    observers = new ArrayList<Observer>
}
registerObserver(Observer o) {
    observers.add(o);
}
```

```
call update() on all the observers
```
The Observer Design Pattern

- Normally you would also have a way of de-registering observers that no longer wish to be informed of an "update".
- java.util has a standard Observer interface and a class that is called Observable which may be used. Also you can just create your own interface and observable class.
- We also see this design pattern in languages which are not object-oriented. The approach is often described as "call-back" where a function or reference to a function is provided to a library via a function call. The provided function or function reference is later "called-back" from the library. Once again the library can initiate the program's code without being coupled to the program.
Examples of Use

We see this design pattern used in situations like:

- GUI libraries (like Swing, Android),
- network IO,
- parsing, and
- error handling.
We need an interface for the observers:

```java
public interface MyObserver {
    void update();
}
```

The class that receives the events must implement the MyObserver interface:

```java
public MyProgram implements MyObserver {
    public void update() {
        // do what is required when informed of an event.
    }
}
```

Your program code would also construct an instance of the library:

```java
lib = new TheLibrary();
```

and then register itself as an observer of this library:

```java
lib.registerObserver(this);
```
The library code would keep track of a list of all the observers:

```java
ArrayList<MyObserver> observers;
```

This would need to be initialized in the library's constructor:

```java
observers = new ArrayList<MyObserver>();
```

Observers would be registered via:

```java
public void registerObserver(MyObserver o) {
    observers.add(o);
}
```

and when the library wishes to inform all the observers of an event it would call a `notifyObservers` method:

```java
private void notifyObservers() {
    for (MyObserver o : observers) o.update();
}
```
• Explain the observer design pattern. What key design issues does this pattern address?

• Suppose you are implementing a library that uses the observer design pattern to inform the users of the library of an event that has happened. However only part of this code has been completed in the library. Fill in the "....." to complete this implementation.

```java
class TheLibrary {
    ArrayList<MyObserver> observers;
    public TheLibrary() { observers = ..... ; }
    private void informObserversOfEvent() {
        ..... 
    }
    public void registerObserver(MyObserver o) {
        ..... 
    }
}
```

We also have:

```java
public interface MyObserver {
    void update();
}
```