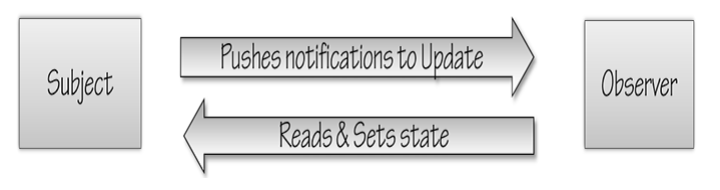
**Adapter Pattern**

A way of notifying change to a number of classes



**Problem**

A large monolithic design does not scale well as new graphing or monitoring requirements are levied.

**Pattern Idea**

* Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.
* The "View" part of Model-View-Controller.

Define an object that is the "keeper" of the data model and/or business logic (the Subject). Delegate all "view" functionality to decoupled and distinct Observer objects. Observers register themselves with the Subject as they are created. Whenever the Subject changes, it broadcasts to all registered Observers that it has changed, and each Observer queries the Subject for that subset of the Subject's state that it is responsible for monitoring.

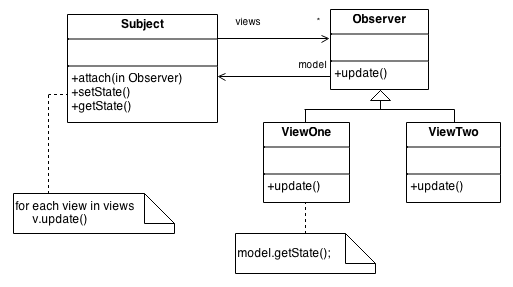
This allows the number and "type" of "view" objects to be configured dynamically, instead of being statically specified at compile-time.

The protocol described above specifies a "pull" interaction model. Instead of the Subject "pushing" what has changed to all Observers, each Observer is responsible for "pulling" its particular "window of interest" from the Subject. The "push" model compromises reuse, while the "pull" model is less efficient.

Issues that are discussed, but left to the discretion of the designer, include: implementing event compression (only sending a single change broadcast after a series of consecutive changes has occurred), having a single Observer monitoring multiple Subjects, and ensuring that a Subject notify its Observers when it is about to go away.

The Observer pattern captures the lion's share of the Model-View-Controller architecture that has been a part of the Smalltalk community for years.

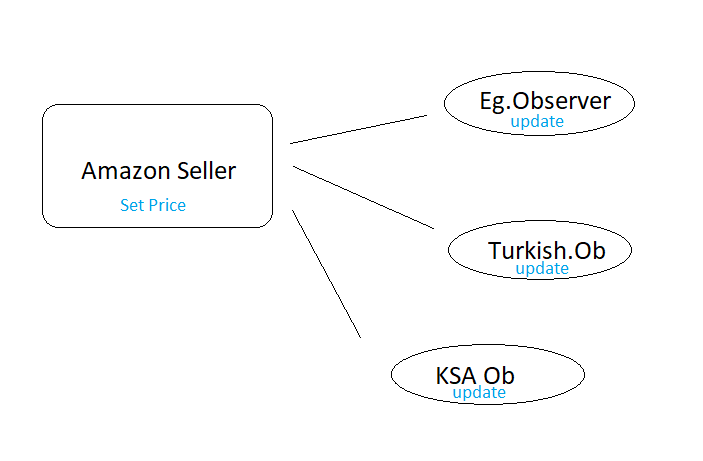
**Structure**



.

**Example**





**Usage**

The Observer pattern is typically used when

* When one object changes its state,then all other dependents object must automatically change their state to maintain consistency system is very complex or difficult to understand.
* When subject doesn't know about number of observers it has.

**Pattern implementation (Java)**

package tryObserver;

import java.util.ArrayList;

import java.util.List;

import java.util.Scanner;

abstract class Observer {

protected Subject subject;

public abstract void update();

}

class Subject {

private List<Observer> observers = new ArrayList<>();

private int state;

public void add(Observer o) {

observers.add(o);

}

public int getState() {

return state;

}

public void setState(int value) {

this.state = value;

execute();

}

private void execute() {

for (Observer observer : observers) {

observer.update();

}

}

}

class EgyptObserver extends Observer {

public EgyptObserver(Subject subject) {

this.subject = subject;

this.subject.add(this);

}

public void update() {

System.out.println("Price in Egypt : " + subject.getState()\*17);

}

}

class TurkishObserver extends Observer {

public TurkishObserver(Subject subject) {

this.subject = subject;

this.subject.add( this );

}

public void update() {

System.out.println("Price in Turky : " + subject.getState()\*2);

}

}

class KSAObserver extends Observer {

public KSAObserver(Subject subject) {

this.subject = subject;

this.subject.add(this);

}

public void update() {

System.out.println("price in USK : " + subject.getState()\*4);

}

}

/\*\*

\*

\* @author pcc

\*/

public class tryObserver {

public static void main( String[] args ) {

Subject sub = new Subject();

new EgyptObserver(sub);

new TurkishObserver(sub);

new KSAObserver(sub);

Scanner scan = new Scanner(System.in);

for (int i = 0; i < 5; i++) {

System.out.print("\nEnter a price: ");

sub.setState(scan.nextInt());

}

}

}

Drawbacks

* If not used carefully the observer pattern can add unecessary complexity
* Observable protects crucial methods which means you can’t even create an instance of the Observable class and compose it with your own objects, you have to subclass.