Structural vs Behavioral

- "Structural patterns are concerned with how classes and objects are composed to form larger structures."
- "Behavior patterns are concerted with algorithms and the assignment of responsibilities between objects. Behavioral patterns describe not just the patterns of objects or classes but also the patterns of communication between them."

Summary

Structural Patterns	
Adapter	Match interfaces of different classes
Bridge	Separates an object's interface from its implementation
Composite	A tree structure of simple and composite objects
Decorator	Add responsibilities to objects dynamically
Facade	A single class that represents an entire subsystem
Flyweight	A fine-grained instance used for efficient sharing
Proxy	An object representing another object

Behavioral Patterns

- Deal with the details of assigning responsibilities between different objects.
- Describe the communication mechanism between objects.
- Define the mechanism for choosing different algorithms by different objects at runtime.

Observer Pattern

Object Behavioral

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

Intent

Observer Pattern

- **Also Known as: Dependents, Publish-Subscribe**
- A common side-effect of partitioning a system into a collection of cooperating classes is the need to maintain consistency between related objects.
- The key objects in the Observer pattern are Subject and Observer.
- A subject may have any number of dependent observers.
- All observers are notified whenever the subject undergoes a change in state.
- In response, each observer will query the subject to synchronize its state with the subject´s state.

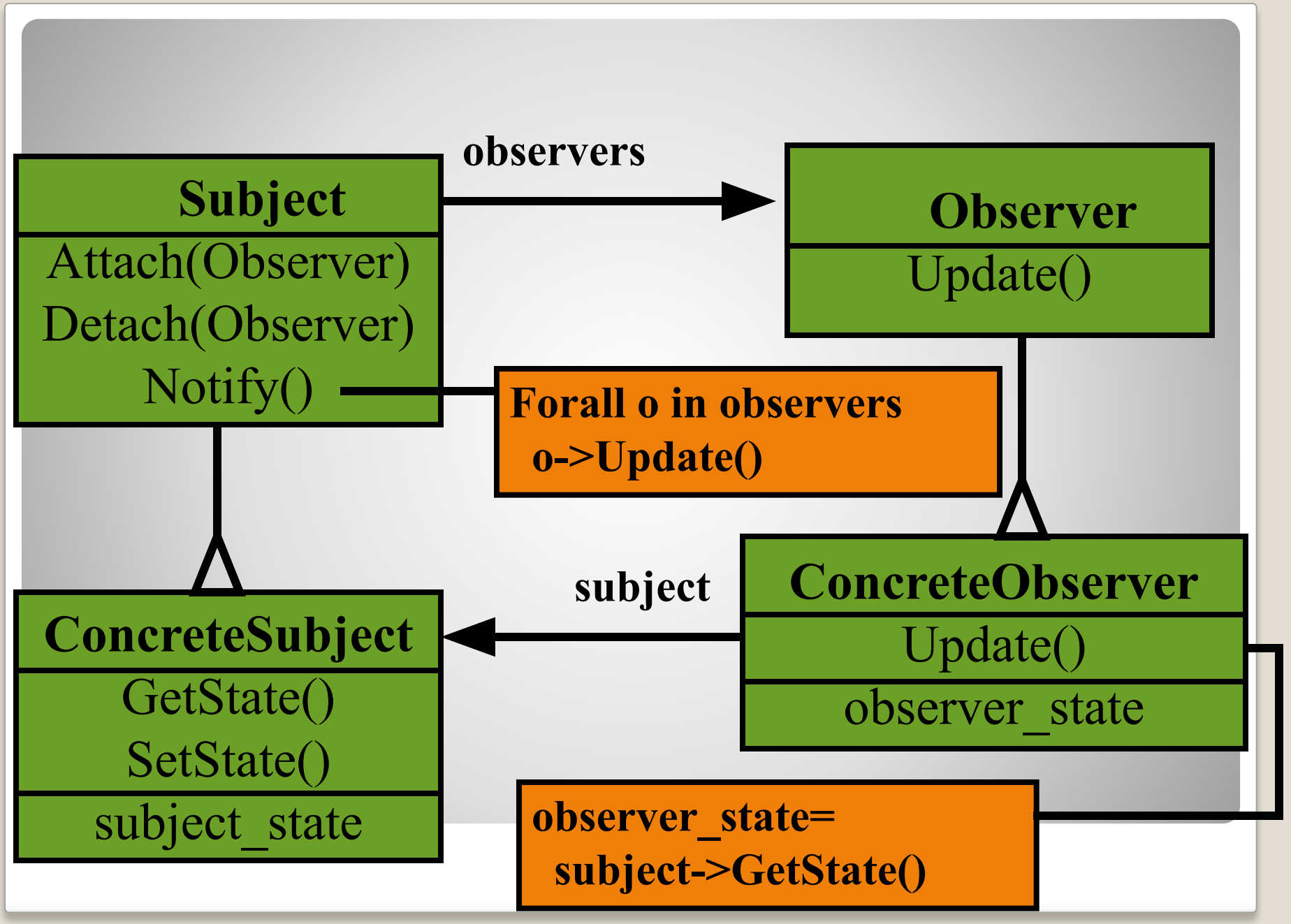
Strive for loosely coupled designs between objects that interact.

Design Principle

The power of loose coupling

The Observer Pattern provides an object design where subjects and observers are loosely coupled

- We can add new observers at any time
- We never need to modify the subject to add new types of observers
- We can reuse subject and observers independently of each other
- Changes to either the subject or an observers will not affect the other


observers
Subject
Attach(Observer)
Detach(Observer)
Notify()
Observer
Update()
Forall o in observers
o->Update()
subject
ConcreteSubject
GetState()
SetState()
subject_state
ConcreteObserver
Update()
observer_state
observer_state=
subject->GetState()

Applicability

- 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, and you do not know how many objects need to be changed.
- When an object should be able to notify other objects without making assumptions about who these objects are.

Participants

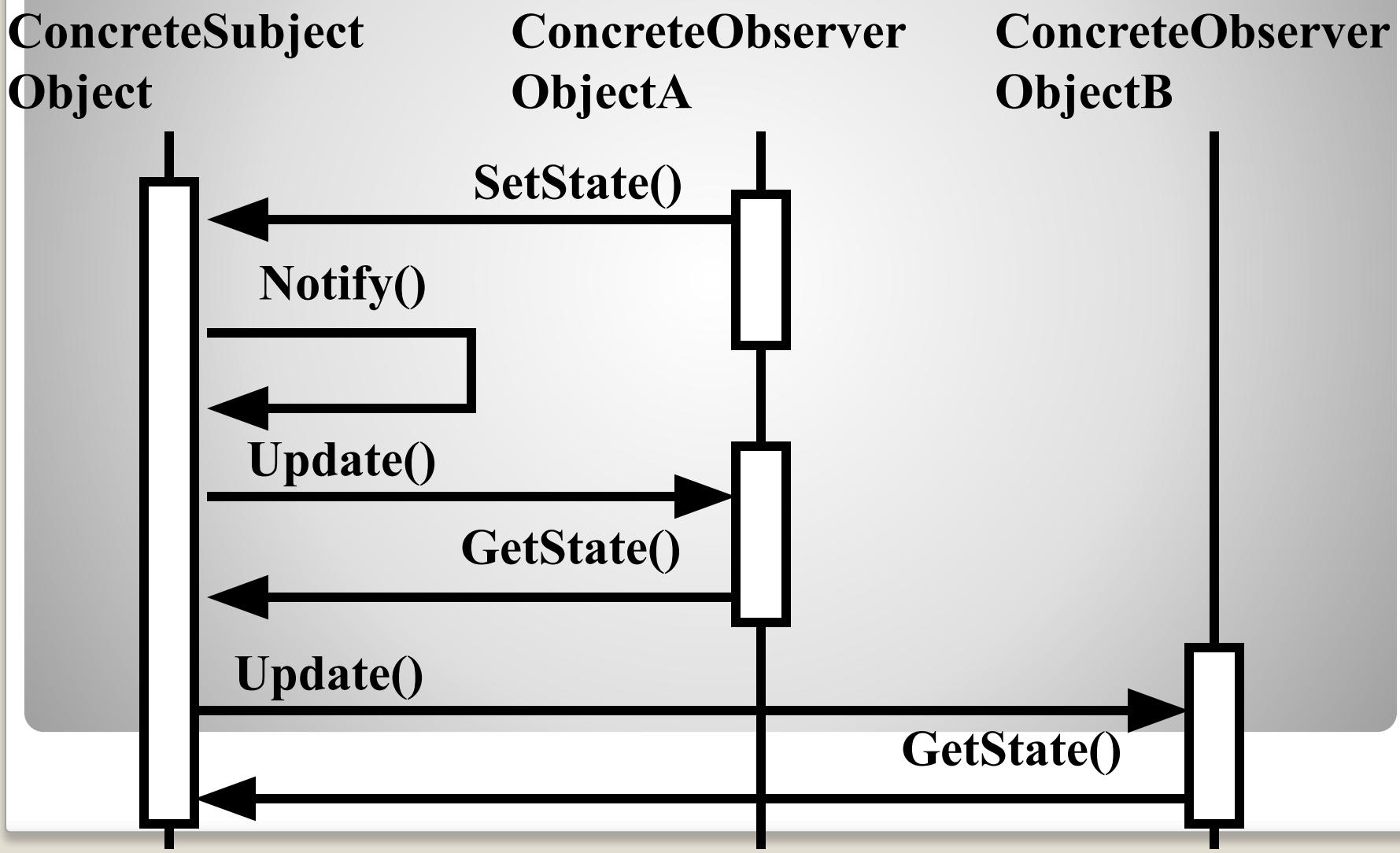
- Subject
  - Knows its observers. Any number of Observer objects may observe a subject.
  - Provides an interface for attaching and detaching Observer Objects.
- Observer
  - Defines an updating interface for objects that should be notified of changes in a subject

Participants

- ConcreteSubject
  - Stores a state of interest to ConcreteObserver objects.
  - Sends a notification to its observers when its state changes.
- ConcreteObserver
  - Maintains a reference to a ConcreteSubject object
  - Stores state that should stay consistent with the subject state.
  - Implements the Observer updating interface to keep its state consistent with the subject state.

Collaborations

- ConcreteSubject notifies its observers whenever a change occurs that could make its observer's state inconsistent with its own.
- After being informed of a change in the ConcreteSubject, a ConcreteObserver object may query the subject for information. ConcreteObserver uses this information to reconcile its state with that of the object.


ConcreteSubject
Object
ConcreteObserver
ObjectA
ConcreteObserver
ObjectB
SetState()
Notify()
Update()
GetState()
Update()
GetState()

- Note that the Observer object that initiates the change request with SetState() postpones its update until it gets a notification from the subject.
- In this scenario Notify() is called by the subject, but it can be called by an observer or by another kind of object.

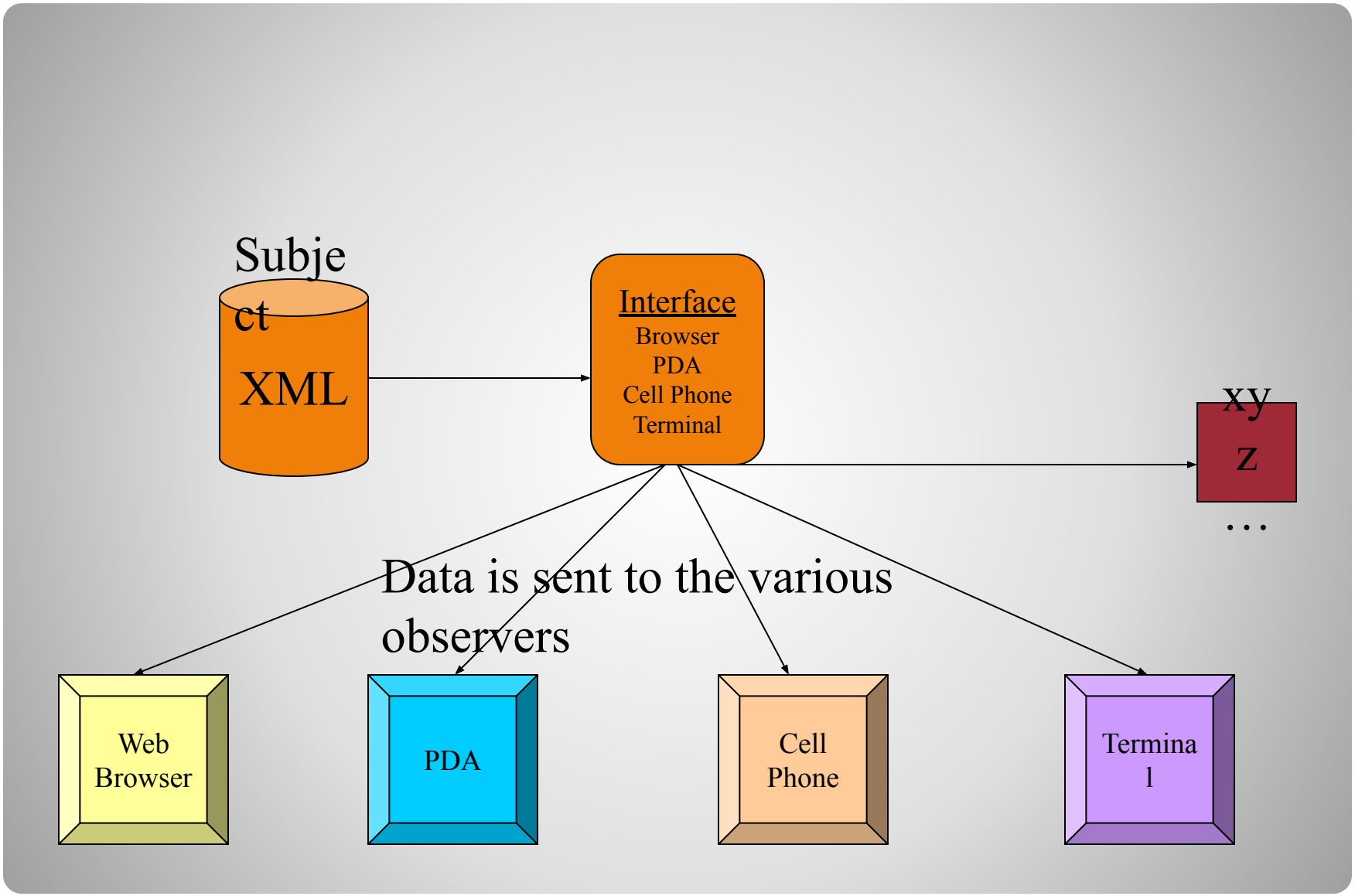
Sequence Diagram

Implementation

- Pull model:
  - The pull model emphasizes the subject's ignorance of its obervers. The pull model may be inefficient, because Observer classes must infer what changed without help from Subject.
- Push model:
  - The push model assumes that the subject knows something about its obervers' needs.
  - The push model might make observers less reusable because Subject classes make assumptions about Observer classes that might not always be true.

Consequences

- The Observer pattern lets you vary subjects and observers independently.
- Abstract coupling between Subject and Object
- Support for broadcast communication
- Unexpected Updates


Subje
ct
XML
Interface
Browser
PDA
Cell Phone
Terminal
xy
z
. . .
Data is sent to the various observers
Web Browser
PDA
Cell Phone
Termina
l
Observe
rs

Related Patterns

- Mediator – an object in the middle intercepts all update requests and broadcasts them to other objects
- Singleton – A “single” subject object instance can be implemented with a Singleton

```
import java.util.Observable;
import java.util.Observer;

public class MessageBoard extends Observable {
  private String message;

  public String getMessage() {
    return message;
  }

  public void changeMessage(String message) {
    this.message = message;
    setChanged();
    notifyObservers(message);
  }
```

Java example

```
  public static void main(String[] args) {
    MessageBoard board = new MessageBoard();
    Student bob = new Student();
    Student joe = new Student();
    board.addObserver(bob);
    board.addObserver(joe);
    board.changeMessage("More Homework!");
  }
}

class Student implements Observer {
  public void update(Observable o, Object arg) {
    System.out.println("Message board changed: " + arg);
  }
}
```

Java example