TDDB84 Design Patterns
Lecture 07
Observer, Chain of Responsibility, Memento

Peter Bunus
Dept of Computer and Information Science
Linköping University, Sweden
petbu@ida.liu.se
Observer
Observer – A Non Software Example

Auctioneer (Subject)

1. Accept Bid
2. Broadcast New High Bid

Bidders (Observers)

486

501 319 127
A Weather Monitoring Application

What we need to implement:
- Weather forecast
- Current conditions
- Weather statistics

What we have:
- Weather station
- Temperature sensor device
- Pressure sensor device
- Humidity sensor device

Weather data object

A Weather Monitoring Application

What we need to implement

Weather forecast

Current conditions

Weather statistics
The Weather Data Object

These three methods return the most recent weather measurements for temperature, humidity and barometric pressure respectively.

We don’t care how this variables are set; the Weather data object know how to get updated info from the Weather station.

I guess I need to implement measurementsChanged so that it updates the three displays.

```java
public void measurementsChanged()
{
    // your code here
}
```

Weather forecast  Current conditions  Weather statistics
public class WeatherData{
    // instance variable declarations
    public void measurementsChanged(){
        float temp = getTemperature();
        float humidity = getHumidity();
        float pressure = getPressure();

        currentConditionsDisplay.update(temp, humidity, pressure);
        statisticsDisplay.update(temp, humidity, pressure);
        forecastDisplay.update(temp, humidity, pressure);

        // other WeatherData methods here
    }
}

• for every new display we need to alter the code
• we have no way to add (or remove) display elements at runtime
• the display elements don’t implement a common interface
• we haven’t encapsulated the part that changes

Grab the most recent measurements by calling the WeatherData’s getter methods (already implemented)
Call each display element to update its display passing it the most recent measurements
Publisher + Subscriber = Observer

When data in the WeatherData object changes the observers are notified.

New data values are communicated to the observers in some form when they change.

This object isn’t an observer so it doesn’t get notified when the subject data changes.

The displays have subscribed with the WeatherData to receive updates when the weather data changes.

Weather data object

Weather forecast

Current conditions

Observer objects

Weather statistics

The displays have subscribed with the WeatherData to receive updates when the weather data changes.
A TV station (SVT) comes along and tells the Weather data that it wants to become an observer.
Adding Observers

- The TV station (SVT) is now an official observer
  - It gets a notification when the Weather object has changed
One of the displays asks to be removed as an observer.
Removing Observers

- The WeatherData acknowledges the Mouse’s request and removes it from the set of observers.
- All the other observers can get another notification except the display that has been recently removed from the set of observers.
The Observer Pattern

Here is the Subject interface. Objects use this interface to register as observers and also to remove themselves as observers.

Each subject can have many observers.

All potential observers need to implement the Observer interface. This interface has just one method Update() that gets called when the Subject’s state changes.

Concrete observers can be any class that implements the Observer Interface. Each observer register with a concrete subject to receive updates.

Concrete subject may also have methods for setting and getting its state.
Observer Structure

Subject
- Attach(Observer)
- Detach(Observer)
- Notify()

ConcreteSubject
- GetState()
- SetState()
- subjectState

for all o in observers {
    o->Update()
}

Observer
- Update()

ConcreteObserver
- observerState = subject->GetState()

observerState

aConcreteSubject
- Notify()
- Update()
- SetState()
- GetState()

anotherConcreteObserver
- GetState()
Loose Coupling

The only things the subject knows about an observer is that it implements a certain interface.

- We can add new observers at any time.
- We never need to modify the subject to add new types of observers.
- We can reuse subjects or observers independently of each other.
- Changes to either the subject or an observer will not affect the other.
The Constitution of Software Architects

- Encapsulate what varies.
- Program to an interface not to an implementation.
- Favor Composition over Inheritance.
- Classes should be open extension but closed for modification.
- Don’t call us, we’ll call you.
- A Class should have only one reason to change.
- Depend upon abstractions. Do not depend upon concrete classes.
- Strive for loosely coupled designs between objects that interact.
- ???????????
The Weather Station

Here is the Subject interface. All our weather components implement the Observer interface. This gives the Subject a common interface to talk to when it comes time to update the observers.

Let’s also create an interface for all display elements to implement.

WeatherData now implements the interface.

This display element shows the current measurements from the WeatherData object.
Implementing the Weather Station

```java
public interface Subject {
    public void registerObserver(Observer o);
    public void removeObserver(Observer o);
    public void notifyObservers();
}

public interface DisplayElement {
    public void display();
}

public interface Observer {
    public void update (float temp, float humidity, float pressure);
}
```
public class WeatherData implements Subject {
    private ArrayList observers;
    private float temperature;
    private float humidity;
    private float pressure;

    public WeatherData() {
        observers = new ArrayList();
    }
    public void registerObserver(Observer o) {
        observers.add(o);
    }
    public void removeObserver(Observer o) {
        int i = observers.indexOf(o);
        if (i >= 0) { observers.remove(i); }
    }
    public void notifyObservers() {
        for (int i = 0; i < observers.size(); i++) {
            Observer observer = (Observer)observers.get(i);
            observer.update(temperature, humidity, pressure);
        }
    }
    public void measurementsChanged() {
        notifyObservers();
    }
    public void setMeasurements(float temperature, float humidity, float pressure) {
        this.temperature = temperature;
        this.humidity = humidity;
        this.pressure = pressure;
        measurementsChanged();
    }
    // other WeatherData methods here
}
public class CurrentConditionsDisplay implements Observer, DisplayElement {
    private float temperature;
    private float humidity;
    private Subject weatherData;

    public CurrentConditionsDisplay(Subject weatherData) {
        this.weatherData = weatherData;
        weatherData.registerObserver(this);
    }

    public void update(float temperature, float humidity, float pressure) {
        this.temperature = temperature;
        this.humidity = humidity;
        display();
    }

    public void display() {
        System.out.println("Current conditions: " + temperature
                         + "F degrees and " + humidity + 
                         
                         "% humidity");
    }
}
public class WeatherStation {

    public static void main(String[] args) {
        WeatherData weatherData = new WeatherData();

        CurrentConditionsDisplay currentDisplay =
                new CurrentConditionsDisplay(weatherData);
        StatisticsDisplay statisticsDisplay =
                new StatisticsDisplay(weatherData);
        ForecastDisplay forecastDisplay =
                new ForecastDisplay(weatherData);

        weatherData.setMeasurements(80, 65, 30.4f);
        weatherData.setMeasurements(82, 70, 29.2f);
        weatherData.setMeasurements(78, 90, 29.2f);
    }
}
Exercise

- Did you know that Java has built-in support for the **Observer** pattern. Check out the **Observer** interface and the **Observable** class in the **java.util** package.
- Download the Weather application from the course web page and rework the application by using the **Observer** interface from the **java.util** package.
- The first one who is ready will have the honour of posting his/her solution on the course blogging page.
Mediator versus Observer

Observer does so through an abstract mechanism. Allows source of notification to be independent of its observers.

Observer

Multiple observers

Each observer knows how to query the state change in its subject

In Mediator, the source must know its mediator. This makes it possible for mediator to define reactions to each stimulus.

Mediator

One mediator per pattern

All the response actions are stored in the mediator. The mediator may modify states of the concrete colleagues.
Chain of Responsibility
Chain of Responsibility – Non Software Example
Chain of Responsibility – Software Example 1
Avoid coupling the sender of a request to its receiver by giving more than one object a chance to handle the request. Chain the receiving objects and pass the request along the chain until an object handles it.
Chain of Responsibility - Properties

- Multiple handlers may be able to handle a request
- Only one handler actually handles the request
- The requester knows only a reference to one handler
- The requester doesn't know how many handlers are able to handle its request
- The requester doesn't know which handler handled its request
- The requester doesn't have any control over the handlers
- The handlers could be specified dynamically
- Changing the handlers list will not affect the requester's code
You’ve got an email !!!!

FROM THE DESK OF DR OLU PHILIPS
MANAGER 11 BUGET AND PLANING
WEB FAX NO: 0871 242 0927
THE MANAGING DIRECTOR/CEO

WE ARE SENDIND THIS LETTER TO BASED ON THE INFORMATION GARTHED FROM THE FOREIGN TRADE OF THE NIGERIAN CHAMBERS OF COMMERCE AND INDUSTRY.WE BELIEVE THAT YOU WOULD BE IN THE POSITION TO HELP US IN OUR BID TO TRANSFER THE SUM OF THIRTY ONE MILLION FIVE HUNDRED DOLLARS ($31.5MUSD)INTO A FORIEGHN ACCOUNT.

WE ARE MEMBERS OF THE SPECIAL COMMITTEE OF BUGET AND PLANING OF THE MINISTRY OF PERTROLEUM, THIS COMMITTEE IS SPECIALY CONCERNED WITH CONTRACT APPRAISALS AND APROVAL OF CONTRACT IN ORDER OF PRIORITIES AS REGARDS CAPITAL PROJECTS OF THE FEDERAL GOVERNMENT OF NIGERIA.WITH OUR POSITION WE HAVE CAREFULLY SECURED THE SUM ($31.5MUSD) THIS AMOUNT WAS ACCUMULATED THROUGH UNDELARED WINDFALL FROM THE SALE OF CRUDE OIL DURING THE GOLF WAR.

WHAT WE NEED FROM YOU IS TO PROVE A SAFE ACCOUNT WERE THIS FUND TRANSFERRED SINCE GOVERNMENT OFFICIALS ARE NOT ALLOWED TO BY OUR LAWS TO OPERATE FORIEGHN ACCOUNT.
Each object in the chain acts like a handler and has a successor object. If it can handle the request it does; otherwise it forwards the request to its successor.

As email is received, it is passed to the first handler: the Spam handler. If the spam handler cannot handle the request it is passed to the FanHandler.

Each email is passed to the first handler.

Email is not handled if it falls off the end of the chain, although you can always implement a catch all handler.
Chain of Responsability – Benefits/Drawbacks

**Benefits:**
- Decouples the sender of the request and its receivers
- Simplifies your object because it doesn’t have to know the chain’s structure and keep direct references to its members
- Allows you to add or remove responsibilities dynamically by changing the members or order of the chain
- Currently used in Windows systems to handle events like mouse clicks or keyboard events.

**Drawbacks:**
- Execution of the request isn’t guaranteed; it may fail at the end of the chain if no object handles it (this can be an advantage or a disadvantage)
- Can be hard to observe the runtime characteristics and debug
Memento
The Memento – Non Software Example

- This pattern is common among do-it-yourself mechanics repairing drum brakes on their cars. The drums are removed from both sides, exposing both the right and left brakes.
- Only one side is disassembled, and the other side serves as a *Memento* of how the brake parts fit together.
- Only after the job has been completed on one side is the other side disassembled. When the second side is disassembled, the first side acts as the *Memento*.

The *Memento* captures and externalizes an object's internal state, so the object can be restored to that state later.
Memento

```java
 Originator o = new Originator();
o.State = "On";

// Store internal state
Caretaker c = new Caretaker();
c.memento = o.CreateMemento();

// Continue changing originator
o.State = "Off";

// Restore saved state
o.SetMemento( c.Memento );
```
Joe is Getting Married
Memento – Another Non-Software Example

Memento

Caretaker

Originator
Memento Example

```
<table>
<thead>
<tr>
<th>Originator</th>
<th>Memento</th>
<th>Caretaker</th>
</tr>
</thead>
<tbody>
<tr>
<td>-state</td>
<td>-state</td>
<td></td>
</tr>
<tr>
<td>+setMemento(in m : Memento)</td>
<td>+getState() +setState()</td>
<td></td>
</tr>
<tr>
<td>+createMemento()</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

1

1

state = m->getState

return new Memento(state)

<table>
<thead>
<tr>
<th>Owner</th>
<th>SafetyDeposit</th>
<th>SecurityGuard</th>
</tr>
</thead>
<tbody>
<tr>
<td>-state</td>
<td>-state</td>
<td>-safetyDeposit</td>
</tr>
<tr>
<td>+writeCombination(in m : SafetyDeposit)</td>
<td>+getCombination() +setCombination()</td>
<td></td>
</tr>
<tr>
<td>+depositCombination()</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

state = m->getCombination()

return new SafetyDeposit(state)
```
Memento Example

```
Owner o = new Owner();
o.state = "123-45-78-45";

// Store code
SecurityGuard c = new SecurityGuard();
c.safetyDeposit = o.depositCombination();

// Continue changing code
o.state = "4444444444";

// Restore saved code
o.writeCombination( c.SafetyDeposit );
```
Memento – Rules of Thumb

Rules of thumb

If you only need one Memento, combine the Originator and Caretaker into one object (Brown, 1998).

If you need many Mementos, store only incremental changes (Brown, 1998). This will help to save space.

Memento often used in conjunction with Command, Iterator and Singleton.

Implementation of the Memento design pattern varies depending on the programming language. Implement the Originator as a friend class to the Memento in C++. Implement the Memento as an inner-class of the Originator in Java (Achtziger, 1999).
public class Originator {
    private T state;
    private class Memento { // value object
        private T mstate;
        private Memento(T state) { mstate = copy_of(state); }
        private T getState() { return mstate; }
    }
    public Memento createMemento() {
        return new Memento(state);
    }
    // continued...
}
class Originator {
    public:
        Memento* CreateMemento();
        void SetMemento(const Memento*);
    private:
        State* _state; // internal data structures
};

class Memento {
    public:
        virtual ~Memento();
    private:
        // private members accessible only to Originator
        friend class Originator;
        Memento();
        void SetState(State*);
        State* GetState();
        // ...
    private:
        State* _state; // ...
};
The Constitution of Software Architects

- Encapsulate that vary.
- Program to an interface not to an implementation.
- Favor Composition over Inheritance.
- Classes should be open extension but closed for modification.
- Don’t call us, we’ll call you
- A Class should have only one reason to change
- Depend upon abstractions. Do not depend upon concrete classes.
- Strive for loosely coupled designs between objects that interact
- **Only talk to your friends**
Memento Advantages and Disadvantages

+ + + + +

- Since OO programming dictates that objects should encapsulate their state it would violate this law if objects’ internal variables were accessible to external objects. The memento pattern provides a way of recording the internal state of an object in a separate object without violating this law.

- It eliminates the need for multiple creation of the same object (i.e. Originator) for the sole purpose of saving its state. Since a scaled down version of the Originator is saved instead of the full Originator object, space is saved.

- It simplifies the Originator since the responsibility of managing Memento storage is no longer centralized at the Originator but rather distributed among the Caretakers.

- The Memento object must provide two types of interfaces: a narrow interface to the Caretaker and a wide interface to the Originator. That is, it must acts like a black box to everything except for the class that created it.

- Using Mementos might be expensive if the Originator must store a large portion of its state information in the Memento or if the Caretakers constantly request and return the Mementos to the Originator. Therefore, this pattern should only be used if the benefit of using the pattern is greater than the cost of encapsulation and restoration.