CS242
Advanced Programming Concepts in Java
Observer Design Pattern
10–14–2013
Outline

- Design Patterns
  - Iterator
  - Observer
- Model–View–Controller (MVC)

**Exam#1**: Wed, October 16th, 7:00 pm, SC162

**Project#1**: due Wed, October 30th
References

- **Head–First Design Patterns**

- [www.javaworld.com](http://www.javaworld.com)

- [cplusplus.com](http://www.cplusplus.com)
  - http://www.cplusplus.com/articles/SwTbqMoL/

- “Gang of Four” (GoF): Erich Gamma, Richard Helm, Ralph Johnson & John Vlissides
  - Design Patterns: Elements of Reusable Object-Oriented Software, Addison–Wesley, 1995
Categories of Design Patterns

- Creational
  - involves object instantiation and provides a way to decouple a client from the objects it needs to instantiate
  - e.g. **Factory**, AbstractFactory, Singleton, ...

- Behavioral
  - concerned with how classes and objects interact and distribute responsibility
  - e.g. **Strategy**, Iterator, **Observer**, Command, State, ...

- Structural
  - how to compose classes or objects into larger structures
  - e.g. **Decorator**, **Composite**, Adapter, Proxy, ...
Recap: Problem

Duck

- quack()
- swim()
- display()
- fly()

MallardDuck

display() { mallard }

WoodDuck

display() { wood duck }

RubberDuck

display() { rubber duck }
quack() { squeak }
fly() { no-op }

DecoyDuck

display() { decoy duck }
quack() { no-op }
fly() { no-op }
Recap: Problem

<<interface>>
Flyable
fly()

<<interface>>
Quackable
quack()

Duck
swim()

MallardDuck
display()
fly(){F}
quack(){Q}

WoodDuck
display()
fly(){F}
quack(){Q}

RubberDuck
display()
fly(){F}
quack(){S}

DecoyDuck
display()
The Strategy Pattern defines a family of algorithms, encapsulates each one in a concrete class, and makes them interchangeable via an interface. Strategy lets the algorithm vary independently from clients that use it.
Strategy – defines an interface common to all supported algorithms. Context uses this interface to call the algorithm defined by a ConcreteStrategy.

ConcreteStrategy – each concrete strategy implements an algorithm.
Strategy Pattern: Implementation

- Context contains a reference to a strategy object (i.e. Composition)
  - may define an interface that lets strategy access its data.
  - The Context objects contain a reference to the ConcreteStrategy that should be used. When an operation is required then the algorithm is run from the strategy object. The Context is not aware of the strategy implementation.
  - The context object receives requests from the client and delegates them to the strategy object. Usually the ConcreteStrategy is created by the client and passed to the context. From this point the clients interacts only with the context.
Duck UML

Client makes use of an encapsulated family of algorithms for both flying and quacking.

Context

Encapsulated fly behavior

Encapsulated quack behavior

Concrete Algorithms

iStrategy

Concrete Algorithms

iStrategy
Iterator Design Pattern

- Used to systematically move through a collection of data using a standard interface without having to know the internal representation; can use to process a collection of items (modify, filter, print, ...)

- Iterator Design pattern, in general, is of the form:

```java
public interface Iterator {
    public Object first(); // move to 1st item
    public Object next(); // advance to next item
    public boolean isDone(); // more elements?
    public Object currentItem(); // get current item
} // differs with each specific language (Java, C++, etc.)
```
Model–View–Controller
MVC Architecture

- **Model**: The core of the application. This maintains the state and data that the application represents. When significant changes occur in the **model**, it updates all of its **views**.

- **Controller**: The user interface presented to the user to manipulate the application.

- **View**: The user interface which displays information about the **model** to the user. Any object that needs information about the **model** needs to be a registered **view** with the **model**.
MVC

- key idea: minimize the coupling between the model, views and controllers
  - a controller detects user interaction that changes the data
  - that controller must tell model about the change
  - the model must notify all views of that change
  - all views must then update() themselves
  - each view asks the model for the current data to display

MVC:

- Model – only knows that views exist and they need to be notified
- Views – don’t know anything about the controllers (so easy to add more views to a model)
- Controller – easy to change the controller for a view
Example

- Bank Account Manager

- Three views of bank accounts
  - List view
  - Bar Graph view
  - Pie Chart view

Example shown in class
Observer Design Pattern

example: design an app that pulls data from a weather station to update three displays for current conditions, weather stats, and a forecast

The three players in the system are the weather station (the physical device that acquires the actual weather data), the WeatherData object (that tracks the data coming from the Weather Station and updates the displays), and the display that shows users the current weather conditions.

Current Conditions is one of three different displays. The user can also get weather stats and a forecast.

Weather-O-Rama provides

What we implement

- Temperature sensor device
- Humidity sensor device
- Pressure sensor device
public void measurementsChanged() {
  // implementation
}

- the implementation needs to update the three displays for current conditions, weather stats and forecast
The Observer Pattern defines a one-to-many dependency between objects so that when one object changes state, all of its dependents are notified and updated automatically.
Observer Design Pattern

Objects use the Subject interface to register as observers (add), to remove themselves from being observers (remove), and to update the observer whenever state changes (notify).
A concrete subject always implements the Subject interface. The concrete subject may also have methods for setting and getting its state.
Observer Design Pattern

Each subject can have many observers

- Subject
  - add()
  - remove()
  - notify()

- Observer
  - update()

- ConcreteSubject
  - theConcreteSubject

- ConcreteObserver
  - update()

notify each observer of subject's state change
update() obtains state information from the subject and acts on that state
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.

`update()` obtains state information from the subject and acts on that state.
Concrete observers can be any class that implements the Observer interface. Each observer registers with a concrete subject to receive updates.

update() obtains state information from the subject and acts on that state.
Loose Coupling

- When two objects are loosely coupled, they can interact, but have very little knowledge of each other.
- Design Principle: Strive for loosely coupled designs between objects that interact.
- Loosely couples designs allow us to build flexible OO systems that can handle change because they minimize the interdependency between objects.
In the Observer Pattern, subjects and observers are loosely coupled

- the only thing the subject knows about an observer is that it implements a certain interface; it doesn’t know the concrete class of the observer
- 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
Weather station app

Here's our subject interface, this should look familiar.

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. The display elements just need to implement a display() method.

Here's the Subject interface:

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

Here's the Observer interface:

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

Here's the WeatherData class:

```java
public class WeatherData implements Subject {
    // Methods to get temperature, humidity, etc.
    // methods to register and remove observers
    // method to notify observers
    // implementation of the Subject interface
}
```

Here's the CurrentConditions class:

```java
public class CurrentConditions implements Observer {
    @Override
    public void update() {
        // Display current conditions
    }
}
```

Here's the DateElement class:

```java
public class DateElement implements DisplayElement {
    @Override
    public void display() {
        // Display current conditions
    }
}
```

Here's the StatisticsDisplay class:

```java
public class StatisticsDisplay implements DisplayElement {
    @Override
    public void display() {
        // Display average, min and max measurements
    }
}
```

Here's the ForecastDisplay class:

```java
public class ForecastDisplay implements DisplayElement {
    @Override
    public void display() {
        // Display the forecast
    }
}
```

These three display elements should have a pointer to WeatherData labeled “subject” too, but boy would this diagram start to look like spaghetti if they did.

This display shows the weather forecast based on the barometer.

Developers can implement the Observer and Display interfaces to create their own display elements.

WeatherData now implements the Subject interface.
public interface Subject {
    public void registerObserver(Observer o);
    public void removeObserver(Observer o);
    public void notifyObservers();
}

public interface Observer {
    public void update(float temp, float humidity, float pressure);
}

public interface DisplayElement {
    public void display();
}
public class WeatherData implements Subject {
    private List observers;
    private float temperature;
    private float humidity;
    private float pressure;

    public WeatherData() {
        observers = new ArrayList();
    }
    public void registerObservers(Observer o) {
        observers.add(o);
    }
    public void removeObservers(Observer o) {
        int i = observers.indexOf(o);
        if (i >= 0) observers.remove(o);
    }
public void notifyObservers() {
    for (Observer o : observers)
        o.update(temperature, humidity, pressure);
}

public void measurementsChanged() {
    notifyObservers();
}

public void setMeasurements(float t, float h, float p) {
    this.temperature = t;
    this.humidity = h;
    this.pressure = p;
    measurementsChanged();
}
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 t, float h, float p) {
        this.temperature = t;
        this.humidity = h;
        display();
    }
    public void display() { /* display code goes here */ }
}