Design Patterns

By Võ Văn Hải
Faculty of Information Technologies - HUI

Behavioral Patterns

Session objectives

Strategy

Observer

Behavioral Patterns

Strategy Pattern

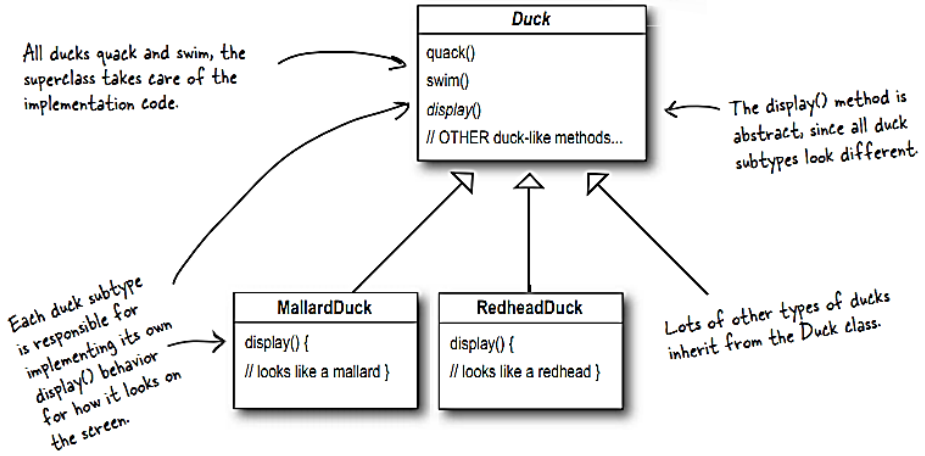
Strategy Pattern

Motivating example - SimpleUDuck

- Joe works for a company that makes a highly successful duck pond simulation game called SimUDuck.
- The game can show a large variety of duck species swimming and making quacking sounds.
- Initial designers used standard OO techniques and created one Duck super-class from which other duck types inherit.

Strategy Pattern

Motivating example – First approach: Using Inheritance



Duck

quack()
swim()
display()
// OTHER duck-like methods...

MallardDuck

display() {
// looks like a mallard }

RedheadDuck

display() {
// looks like a redhead }

All ducks quack and swim, the superclass takes care of the implementation code.

The display() method is abstract, since all duck subtypes look different.

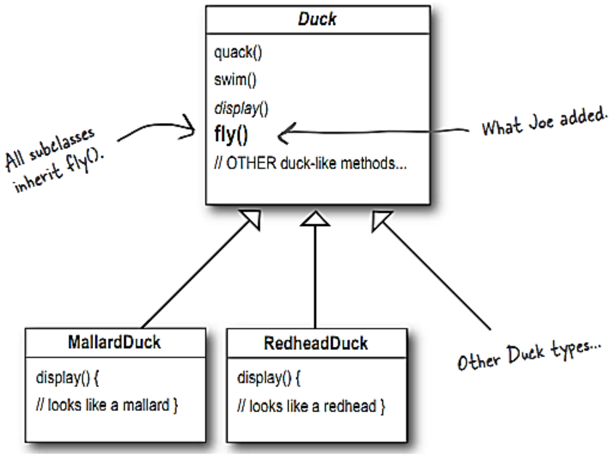
Each duck subtype is responsible for implementing its own display() behavior for how it looks on the screen.

Lots of other types of ducks inherit from the Duck class.

Strategy Pattern

Motivating example – First approach

Now we need them to fly…



Duck

quack()
swim()
display()
fly()
// OTHER duck-like methods...

MallardDuck

display() {
// looks like a mallard }

RedheadDuck

display() {
// looks like a redhead }

All subclasses inherit fly().

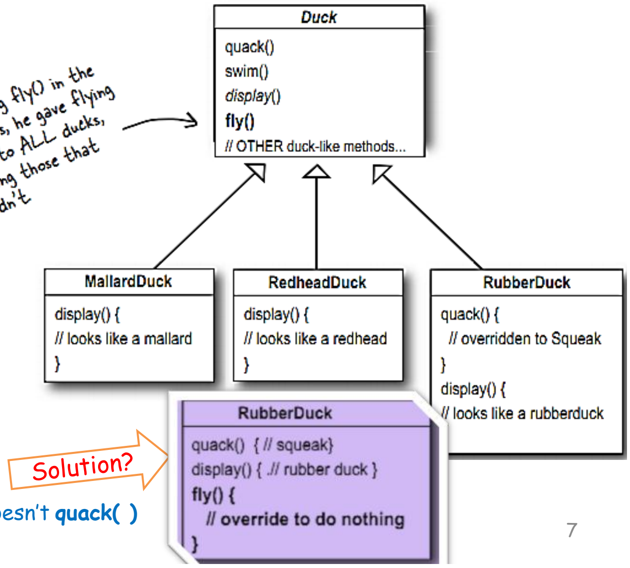
What Joe added.

Other Duck types...

Strategy Pattern

Motivating example – First approachPatterns

By putting fly() in the superclass, he gave flying ability to ALL ducks, including those that shouldn't.



Duck
quack()
swim()
display()
fly()
// OTHER duck-like methods...

MallardDuck
display() {
// looks like a mallard
}

RedheadDuck
display() {
// looks like a redhead
}

RubberDuck
quack() {
// overridden to Squeak
}
display() {
// looks like a rubberduck

RubberDuck
quack() { // squeak}
display() { .// rubber duck }
fly() {
// override to do nothing
}

Solution?

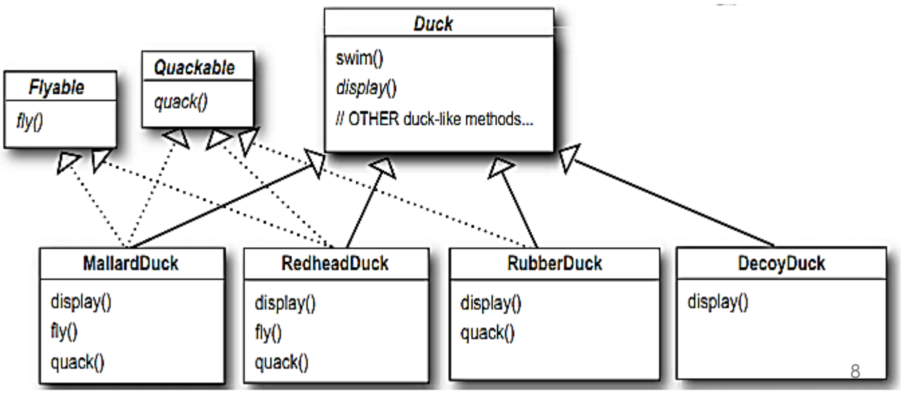
They just gave a demo and there were rubber duckies flying around the screen.

How about **DecoyDuck** ? It doesn't **quack( )** or **fly( )**

Strategy Pattern

Motivating example – Second Approach: Using Interface

- The aspects that change for each type of duck are the methods fly() and quack() ⇨ Take these methods out of the Duck class.



Flyable
fly()

Quackable
quack()

Duck
swim()
display()
// OTHER duck-like methods...

MallardDuck
display()
fly()
quack()

RedheadDuck
display()
fly()
quack()

RubberDuck
display()
quack()

DecoyDuck
display()

Strategy Pattern

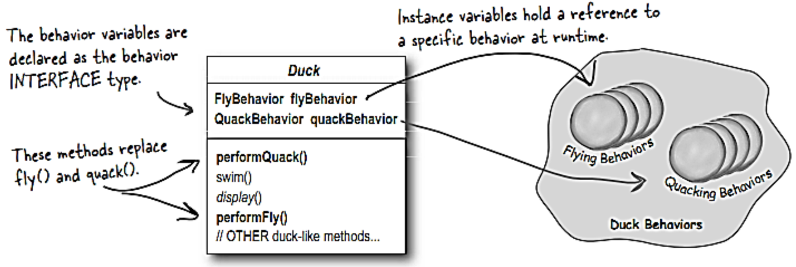
Motivating example – Second Approach

- Disadvantage of this solution:
  - All methods in Java interfaces are abstract.
  - Code has to be duplicated for classes.
  - Modification will have to be made to more than one class.
  - This will introduce bugs.
- Design principles:
  - Identify the aspects of the application that vary and separate them from what stays the same.
  - Encapsulate the parts that vary.
  - Future changes can be made without affecting parts that do not vary.
  - Results in fewer unexpected consequences from code change.

Behavioral Patterns

Motivating example – Third Approach

- Separate what varies - separate what changes from what stays constant.
  1. We know that fly ( ) and quack ( ) are the parts of the Duck class that vary across ducks.
  2. To separate these behaviors from the Duck class, we'll pull both methods out of the Duck class and create a new set of classes to represent each

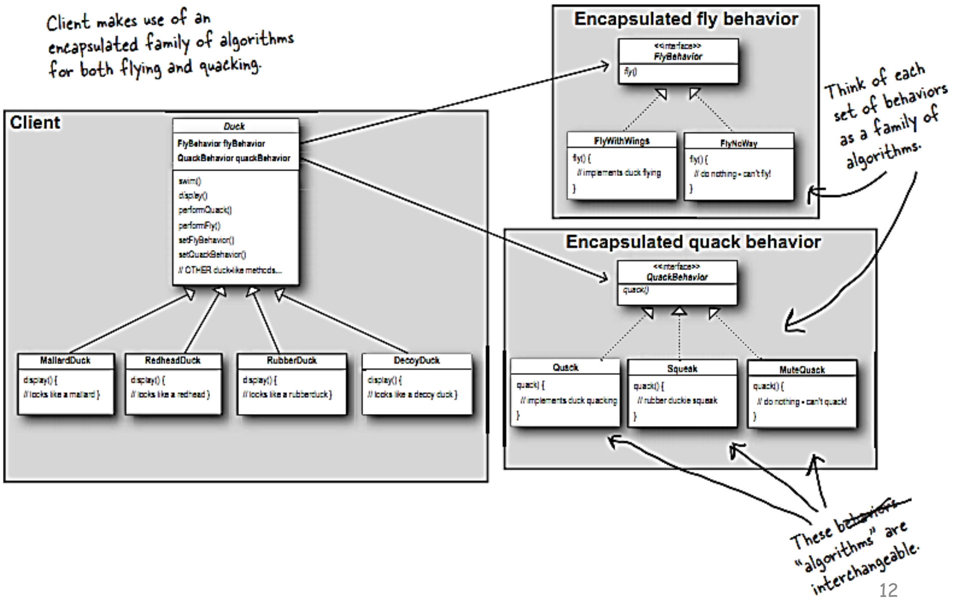


Duck

FlyBehavior flyBehavior
QuackBehavior quackBehavior

performQuack()
swim()
display()
performFly()
// OTHER duck-like methods...

The behavior variables are declared as the behavior INTERFACE type.
These methods replace fly() and quack().
Instance variables hold a reference to a specific behavior at runtime.
Flying Behaviors
Quacking Behaviors
Duck Behaviors

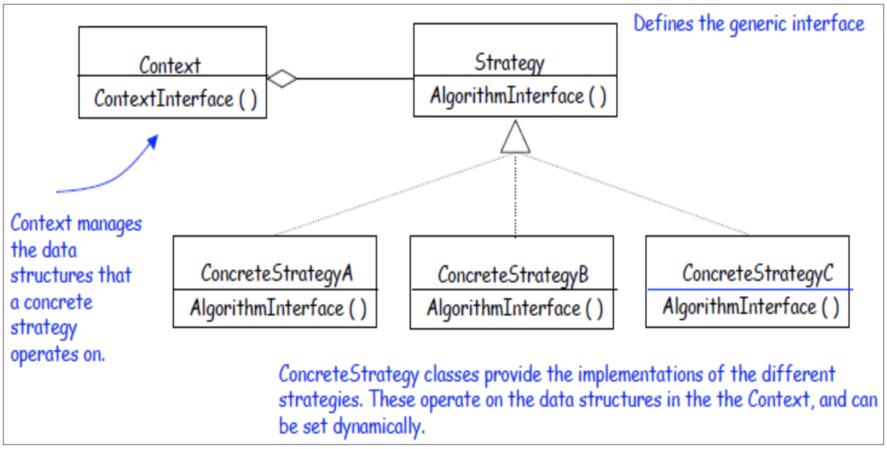

Client makes use of an encapsulated family of algorithms for both flying and quacking.
Client
Duck
FlyBehavior flyBehavior
QuackBehavior quackBehavior
swim()
display()
performQuack()
performFly()
setFlyBehavior()
setQuackBehavior()
// OTHER duck-like methods...
MallardDuck
display() {
// looks like a mallard }
RedheadDuck
display() {
// looks like a redhead }
RubberDuck
display() {
// looks like a rubberduck }
DecoyDuck
display() {
// looks like a decoy duck }
Encapsulated fly behavior
<<interface>>
FlyBehavior
fly()
FlyWithWings
fly() {
// implements duck flying
}
FlyNoWay
fly() {
// do nothing - can't fly!
}
Think of each set of behaviors as a family of algorithms.
Encapsulated quack behavior
<<interface>>
QuackBehavior
quack()
Quack
quack() {
// implements duck quacking
}
Squeak
quack() {
// rubber duckie squeak
}
MuteQuack
quack() {
// do nothing - can't quack!
}
These ~~behaviors~~ "algorithms" are interchangeable.

Strategy Pattern

Definition

- Definition
  - Defines a family of algorithms, encapsulates each one, and makes them interchangeable. Strategy lets the algorithm vary independently from clients that use it.
- Use the Strategy pattern when
  - many related classes differ only in their behavior. Strategies provide a way to configure a class with one of many behaviors
  - you need different variants of algorithm
  - an algorithms uses data that client shouldn't know about
  - a class defines many behaviors, and these appear as multiple conditional statements in its operations

Strategy Pattern
UML

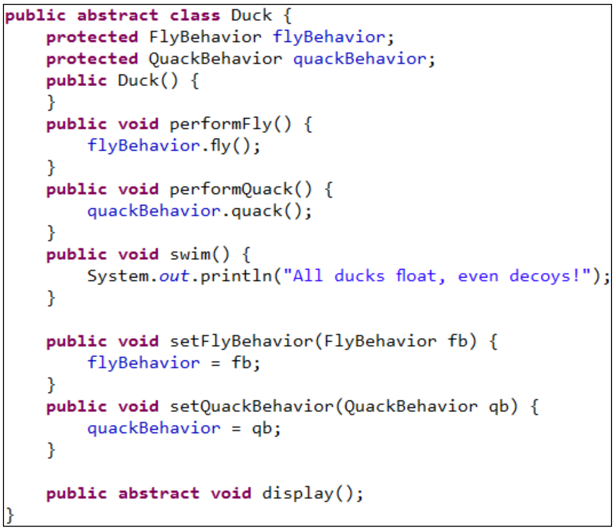


Defines the generic interface

Context manages the data structures that a concrete strategy operates on.

ConcreteStrategy classes provide the implementations of the different strategies. These operate on the data structures in the the Context, and can be set dynamically.

Strategy Pattern
Code sample



```java
public abstract class Duck {
    protected FlyBehavior flyBehavior;
    protected QuackBehavior quackBehavior;
    public Duck() {
    }
    public void performFly() {
        flyBehavior.fly();
    }
    public void performQuack() {
        quackBehavior.quack();
    }
    public void swim() {
        System.out.println("All ducks float, even decoys!");
    }

    public void setFlyBehavior(FlyBehavior fb) {
        flyBehavior = fb;
    }
    public void setQuackBehavior(QuackBehavior qb) {
        quackBehavior = qb;
    }

    public abstract void display();
}
```

Strategy Pattern

Code sample

```
public interface FlyBehavior {
    void fly();
}
```

```
public interface QuackBehavior {
    void quack();
}
```

```
public class FlyWithWings implements FlyBehavior {
    @Override
    public void fly() {
        System.out.println("I’m flying!!");
    }
}
```

```
public class FlyNoWay implements FlyBehavior {
    @Override
    public void fly() {
        System.out.println("I can’t fly");
    }
}
```

16

Strategy Pattern

Code sample

```
public class RedheadDuck extends Duck{
    public RedheadDuck() {
        quackBehavior=new Quack();
        flyBehavior=new FlyWithWings();
    }
    @Override
    public void display() {
        System.out.println("I’m a real Redhead duck");
    }
}
```

```
public class RubberDuck extends Duck{
    public RubberDuck() {
        flyBehavior=new FlyNoWay();//cannot fly
        quackBehavior=new Squeak();//squeak
    }
    @Override
    public void display() {
        System.out.println("I’m a rubber duck");
    }
}
```

17

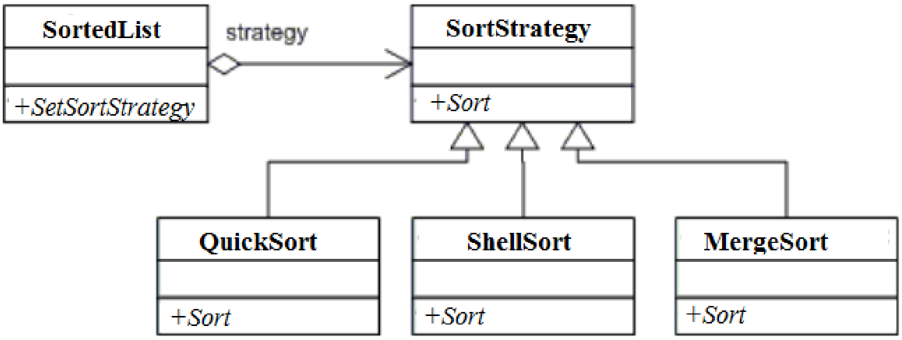
Strategy Pattern

Code sample - excercise

- Add a new type of duck to the simulator, namely, *DodelDuck*. A model duck does not fly and quacks.
- Add a new fly behavior to the simulator, namely, *FlyRocketPowered*, which represents flight via a rocket.
- Create an instance of a *ModelDuck* and change its behavior at runtime to be flight via a rocket.

Strategy Pattern

Other Example

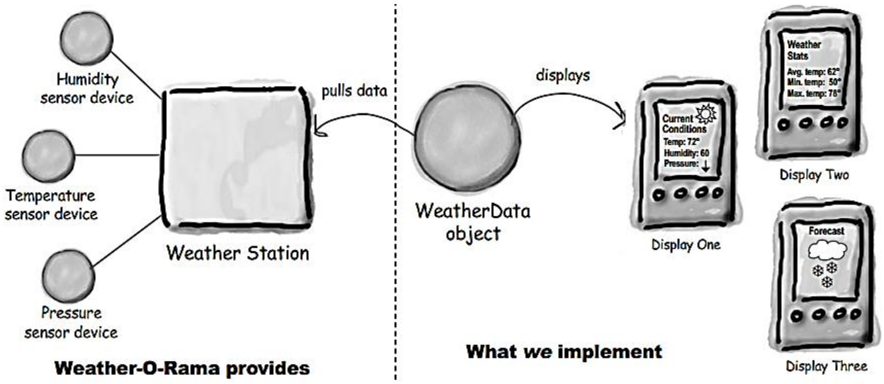


Behavioral Patterns

Observer Pattern

Observer Pattern

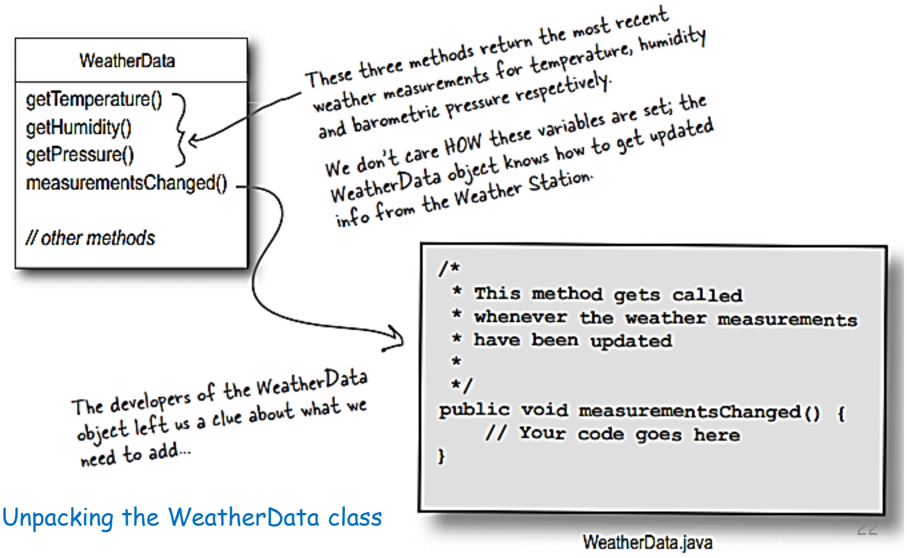
Motivation Example: Weather Forecast



Our work: create an application that uses the WeatherData object to update three displays for current conditions, weather stats and the forecast.

Observer Pattern

Motivation Example: Weather Forecast



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

We don't care HOW these variables are set; the WeatherData object knows how to get updated info from the Weather Station.

The developers of the WeatherData object left us a clue about what we need to add...

```
/*
 * This method gets called
 * whenever the weather measurements
 * have been updated
 *
 */
public void measurementsChanged() {
    // Your code goes here
}
```

Unpacking the WeatherData class

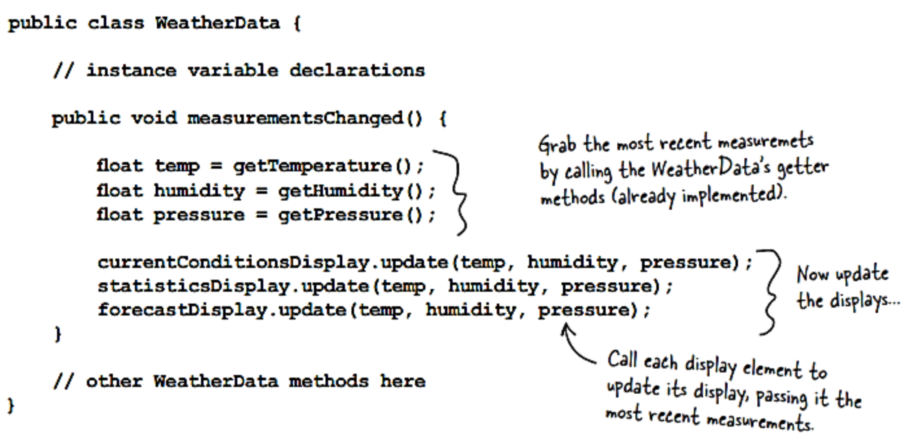
Observer Pattern

Motivation Example: Weather Forecast

- The WeatherData class has getter methods that obtain measurement values from temperature, humidity and pressure.
- The class has a **`measurementsChanged()`** method that updates the three values.
- Three displays must be implemented: current conditions, statistics and forecast display.
- System must be expandable – other display elements maybe added or removed.

Observer Pattern

Motivation Example: First Approach



```
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
}
```

Grab the most recent measuremets by calling the WeatherData's getter methods (already implemented).

Now update the displays...

Call each display element to update its display, passing it the most recent measurements.

How to add or remove other display elements without making changes to the program?

Observer Pattern

Real world example: newspaper/magazine subscriptions

- A newspaper publisher goes into business and begins publishing newspapers.
- You subscribe to a particular publisher, and every time there's a new edition it gets delivered to you. As long as you remain a subscriber, you get new newspapers.
- You unsubscribe when you don't want papers anymore, and they stop being delivered.
- While the publisher remains in business, people, hotels, airlines and other businesses constantly subscribe and unsubscribe to the newspaper.

Publishers + Subscribers = Observer Pattern

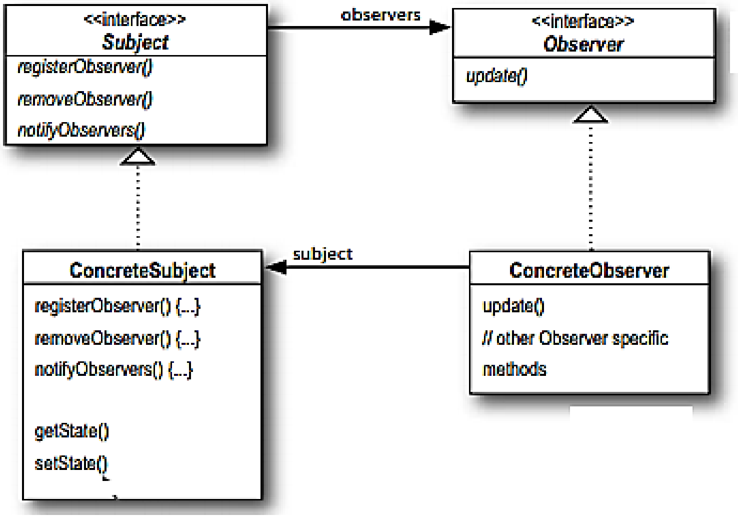
Observer Pattern

Introduction

- Definition:
  - 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
- Features:
  - The object that changes state is called the subject and the other objects are the observers.

Observer Pattern

UML



<<interface>>
Subject
registerObserver()
removeObserver()
notifyObservers()

observers

<<interface>>
Observer
update()

subject

ConcreteSubject
registerObserver() {...}
removeObserver() {...}
notifyObservers() {...}

getState()
setState()

ConcreteObserver
update()
// other Observer specific methods

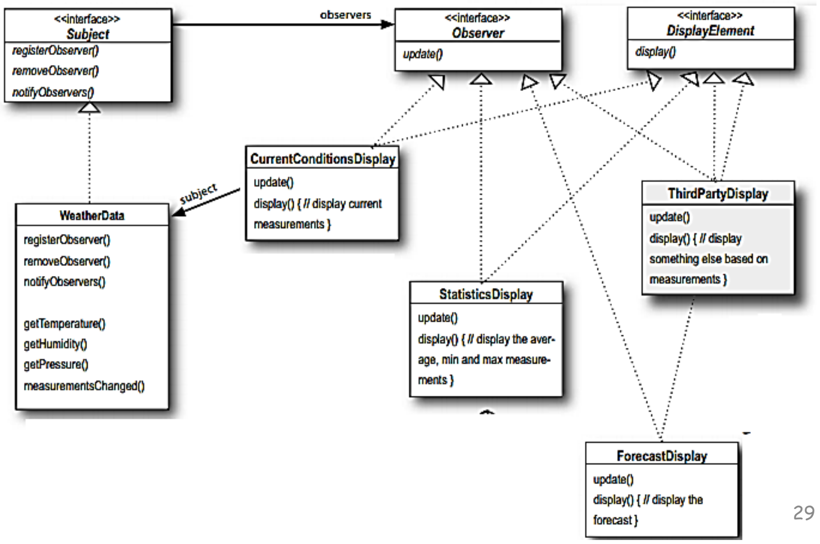
Observer Pattern

UML - components

- Subject
  - knows its observers. Any number of Observer objects may observe a subject.
  - provides an interface for attaching and detaching Observer objects.
- ConcreteSubject
  - stores state of interest to ConcreteObserver
  - sends a notification to its observers when its state changes
- Observer
  - defines an updating interface for objects that should be notified of changes in a subject.
- ConcreteObserver
  - maintains a reference to a ConcreteSubject object
  - stores state that should stay consistent with the subject's
  - implements the Observer updating interface to keep its state consistent with the subject's

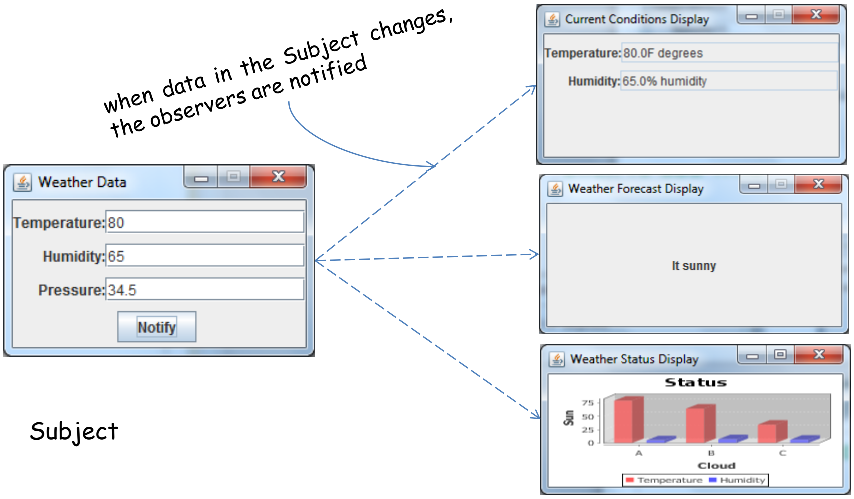
Observer Pattern

Motivation Example: applied Observer Pattern to my approach



Observer Pattern

Motivation Example: results



when data in the Subject changes, the observers are notified

Weather Data

Temperature: 80

Humidity: 65

Pressure: 34.5

Notify

Current Conditions Display

Temperature: 80.0F degrees

Humidity: 65.0% humidity

Weather Forecast Display

It sunny

Weather Status Display

Subject

Observer objects

Summary

- Strategy
  - Defines a family of algorithms, encapsulates each one, and makes them interchangeable. Strategy lets the algorithm vary independently from clients that use it.
- Observer
  - Defines a one-to-many dependency between objects so that when one object changes state, all of its dependents are notified and updated automatically

©Jiri Moucka * illustrationsOf.com/60293