CSE201: Monsoon 2020
Advanced Programming

Lecture 24: Design Pattern Part-4

Raghava Mutharaju (Section-B)
Vivek Kumar (Section-A)
CSE, IIIT-Delhi
raghava.mutharaju@iiitd.ac.in

Last Lecture

- Five design patterns
  - **Template**
    - Define the skeleton of an algorithm in an operation, deferring some steps to client subclasses
  - **Prototype**
    - An object that serves as a basis for creation of others
  - **Factory**
    - A method or object that creates other objects
    - **Abstract factory**
      - A superclass factory that can be extended to provide different sub-factories, each with different features
  - **Facade**
    - Hiding the complexities of a large body of code by providing a simplified interface

```
public class Animal implements Cloneable {
    private String name;
    public Animal(String n) { name=n; }
    public void sayHello() {
        System.out.println("I am a " + name);
    }
    public Animal clone() throws CloneNotSupportedException {
        return (Animal) super.clone();
    }
}
```

```
public class Sheep extends Animal {
    private String wool;
    public Sheep() { super("Sheep"); wool ="10KG"; }
    public void sayHello() {
        super.sayHello();
        System.out.println("I have "+wool+" wool");
    }
    public Sheep clone() throws CloneNotSupportedException {
        return (Sheep) super.clone();
    }
}
```

```
public class Lab {
    public static Animal getClone(Animal s) {
        return s.clone();
    }
}
```

```
public class Client {
  public static void main(String[] args)
                        throws CloneNotSupportedException{
        Animal s1 = new Sheep(); Animal c1 = new Chicken();
        Animal s2 = Lab.getClone(s1);
        Animal c2 = Lab.getClone(c2);
    }
}
```

```
public class Chicken extends Animal {
    private int eggs;
    public Chicken() { super("Chicken"); eggs=3; }
    public void sayHello() {
        super.sayHello();
        System.out.println("I have "+eggs+" eggs");
    }
   public Chicken clone() throws CloneNotSupportedException{
        return (Chicken) super.clone();
    }
}
```

```
public abstract class Cafe {
    public void boilWater() {
        System.out.println("Boil Water");
    }
    public void pourInCup() {
        System.out.println("Pour in Cup");
    }
    // "final" ensures that the person preparing
    // the beverage sticks to the recipe of this
    // Café instead of generating his own
    public final void prepare() {
        boilWater();
        brew();
        pourInCup();
        addCondiments();
    }
    public abstract void brew();
    public abstract void addCondiments();
}
```

```
public class Coffee extends Cafe {
    private void brew() {
        System.out.println("Brew Coffee");
    }
    private void addCondiments() {
        System.out.println("Add Sugar and Milk");
    }
}
```

```
public class Tea extends Cafe {
    private void brew() {
        System.out.println("Steep Tea Bag");
    }
    private void addCondiments() {
        System.out.println("Add Sugar and Lemon");
    }
}
```

```
public abstract class AnimalFactory {
    public abstract Animal createAnimal(String need);
}
```

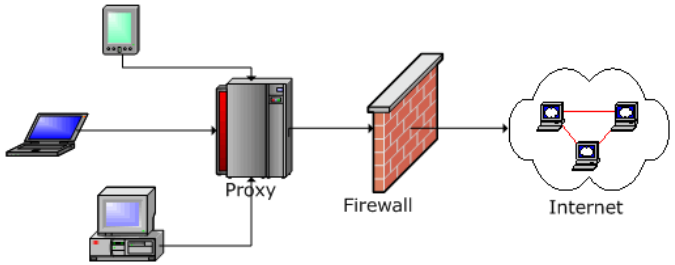
```
public class CatFactory extends AnimalFactory {
    public Animal createAnimal(String need) {
        if(need.equals("pet") {
            return new HouseCat();
        }
        else if(need.equals("zoo") {
            return new Lion();
        }
    }
}
```

```
public class DogFactory extends AnimalFactory {
    public Animal createAnimal(String need) {
        if(need.equals("kids") {
            return new Poodle();
        }
        else if(need.equals("hunting") {
            return new Greyhound();
        }
    }
}
```

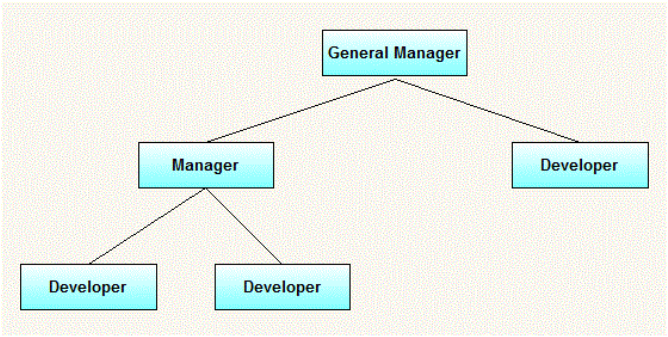
```
public class ClientForCats {
  public static void main(String[] args) throws
CloneNotSupportedException{
        String need = args[0];
        AnimalFactory factory = new CatFactory();
        Animal animal = factory.createAnimal(need);
        // Our client is too greedy
        Animal[] cloned = new Animal[100];
        for(int i=0; i<cloned.length; i++) {
            cloned[i] = Lab.getClone(animal);
        }
    }
}
```

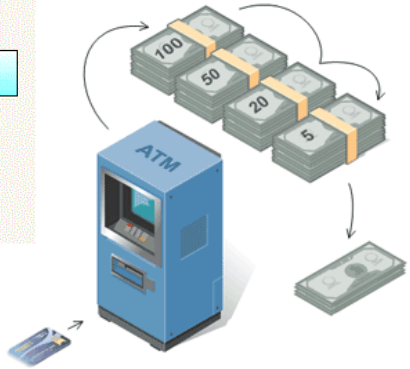
```
public class ClientForDogs {
  public static void main(String[] args) throws
CloneNotSupportedException{
        String need = args[0];
        AnimalFactory factory = new DogFactory();
        Animal animal = factory.createAnimal(need);
        // Our client is too greedy
        Animal[] cloned = new Animal[100];
        for(int i=0; i<cloned.length; i++) {
            cloned[i] = Lab.getClone(animal);
        }
    }
}
```

Today's Lecture



- Six more design patterns
  - Decorator (DP # 11)
  - Composite (DP # 12)
  - Proxy (DP # 13)
  - Chain of responsibility (DP # 14)
  - Observer (DP # 15)
  - State (DP # 16)

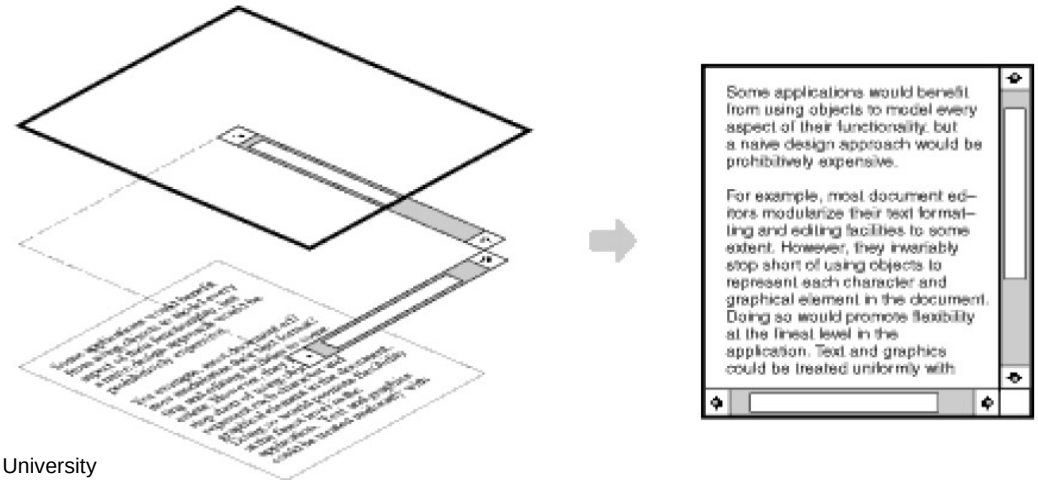




All images that appears in this slide are from https://images.google.com/

Pattern: Decorator

objects that wrap around other objects to add useful features



Source of this slide: CSE331, Washington University

Remember this from IO Streams?

```
public static void main(String args[])
            throws IOException
{
    Scanner in = null;
    PrintWriter out = null;
    try {
        in = new Scanner( new BufferedReader( new
                FileReader("input.txt")));
        out = new PrintWriter( new
                FileWriter("output.txt"));
        while (in.hasNext()) {
            out.println(in.next());
        }
    } finally {
        if (in != null)
            in.close();
        if (out != null)
            out.close();
        }
}
```

- We saw this example in Lecture 13 of combining three classes for breaking input into tokens:
  - Scanner
  - BufferedReader
  - FileReader
- Normal InputStream class has only public int read() method to read one letter at a time
- BufferedReader or Scanner add additional functionality to read the stream more easily
  - Here, BufferedReader and Scanner are examples of Decorator objects

Decorator pattern

- **Decorator**: an object that modifies behavior of, or adds features to, another object
  - Helps in adding features to an existing simple object without needing to disrupt the interface that client code expects when using the simple object

Decorator Pattern: Vehicle Paint Shop

```
interface Vehicle {
    public void paint();
}
```

```
class Bike implements Vehicle {
    public void paint() {
        System.out.println(“White color Bike”);
    }
}
```

```
class Car implements Vehicle {
    public void paint() {
        System.out.println(“White color Car”);
    }
}
```

```
//Abstract to disallow clients to instantiate
it
abstract class VehicleDecorator implements
Vehicle {
    private Vehicle decoratedVehicle;
    public VehicleDecorator(Vehicle v) {
        this.decoratedVehicle = v;
    }
    public void paint() {
        decoratedVehicle.paint();
```

```
class BlueVehicleDecorator extends VehicleDecorator
{
    public BlueVehicleDecorator(Vehicle v) {
        super(v);
    }
    public void paint() {
        super.paint();
        System.out.println(“Now painted in Blue
color”);
    }
}
```

```
public class Client {
    public static void main(String[] args) {
        Vehicle c1 = new Car();
        c1.paint(); // default white paint
        Vehicle c2 = new BlueVehicleDecorator(new
Car));
        c2.paint(); // painted in blue color
        ......
    }
```

Pattern: Composite

objects that can contain their own type

Source of this slide: CSE331, Washington University

Composite Pattern

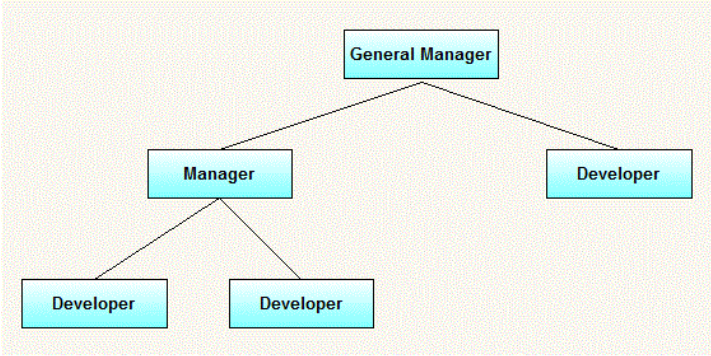
- An object that can be either an individual item or a collection of many items
  - Can be composed of individual items or other composites
  - Recursive definition: Objects that can hold themselves

Employee Hierarchy

```
interface Employee {
    public void print();
}
```

```
class Manager implements Employee {
    List<Employee> emp = new
ArrayList<Employee>();
    public void add(Employee e) { emp.add(e); };
    public void remove(Employee e)
{ emp.remove(e); }
    public void print() {
        System.out.println("Manager");
        for(Employee e : emp) {
            e.print();
        }
    }
}
```

```
class Developer implements Employee {
    public void print() {
        System.out.println("Employee");
    }
}
```



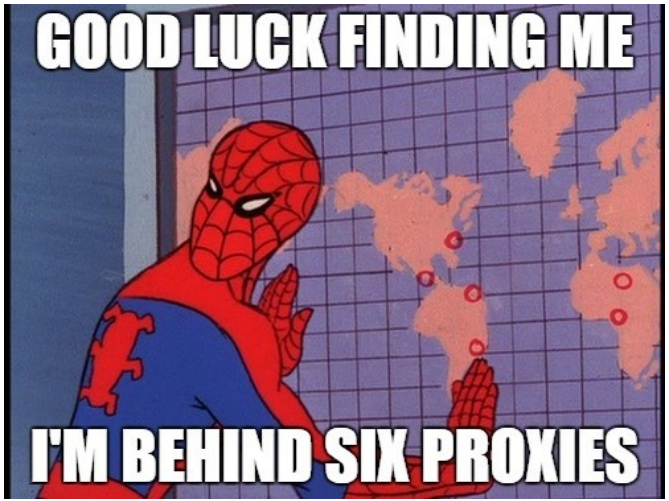

```
public class Client {
    public static void main(String[] args) {
        Employee gm = new Manager();
        Employee emp1 = new Developer();
        Employee manager = new Manager();
        Employee emp2 = new Developer();
        Employee emp3 = new Developer();
        gm.add(emp1); gm.add(manager);
        manager.add(emp2); manager.add(emp3);
        gm.print(); // print all nodes in tree
above
    }
}
```

- Composite pattern helps client to ignore the difference between individual objects and allow him to treat all objects in the composite structure uniformly

Source of this figure: https://dzone.com/articles/composite-design-pattern-java-0

Pattern: Proxy

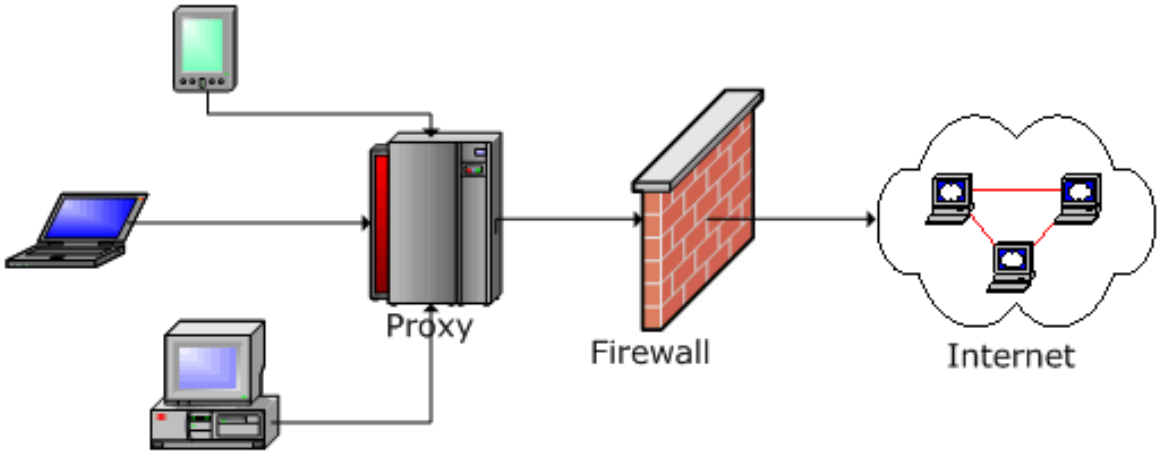
Controls and manages access to objects they are protecting



Proxy Pattern

- **Proxy** – provides a surrogate or placeholder for another object to control access to it
- **Examples**
  - A cheque or credit card is a proxy for what is in our bank account and provides a means of accessing that cash
  - Sometimes real subject is not available, then proxy can behave as real subject and allow simple operations (avoiding compilation errors, emulation of real subject, etc.)
  - Using a proxy to query a database but without having the ability to modify it

Implementing Proxy Firewall for Intranet



- Users who want to login to company's intranet have to first authenticate themselves with the **proxy** firewall
- How to implement this software using proxy design pattern?

Implementing Proxy Firewall for Intranet

```
interface IntranetAccess {
    public void getAccess(String name);
}
```

```
class Intranet implements IntranetAccess {
    public void getAccess(String name) {
        System.out.println("Unrestricted access
                                    granted to "+
name);
    }
}
```

```
public class Client {
    public static void main(String [] args) {
        String name = args[0];
        IntranetAccess proxy = new ProxyFirewall();
        proxy.getAccess(name);
    }
}
```

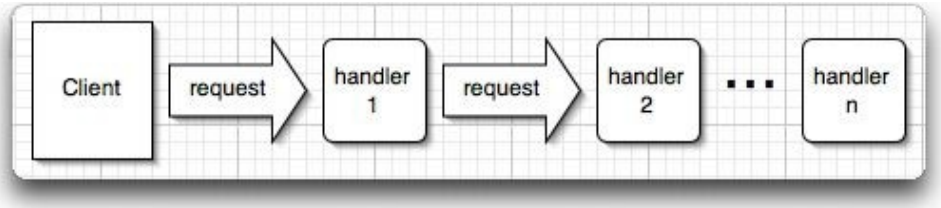
```
import java.util.*;
class ProxyFirewall implements IntranetAccess {
    private static List<String> db = new
ArrayList<String>();

    public void getAccess(String name) {
        if(db.contains(name)) {
            (new Intranet()).getAccess(name);
        }
        else {
            System.out.println("Access denied to "+
name);
        }
    }

    public void add(String name) {
        db.add(name);
    }
    // Some more code that is elided
}
```

Pattern: Chain of Responsibility

Gives more than one object an opportunity to handle a request by linking receiving objects together



Chain of Responsibility Pattern

- Avoid coupling sender of request to its receiver by giving more than one object chance to handle request. Chain receiving objects and pass request along until an object handles it
- Scenario for usage
  - When more than one object may handle a particular request and the handler isn’t known ahead of time
  - When you want to issue a request to one of several objects without specifying the receiver explicitly
- Example
  - Pipeline assembly for car manufacturing

Example: Implementing Bank ATM Software

- An ATM machine contains notes in fixed denominations, e.g., INR 2000, 500, 200 and 100
- Withdrawing an amount that is not in multiples of 100 will not work
- Withdrawing amount less than INR 2000 could dispense notes of 500, 200 and 100 denominations
- How to implement the note dispensing software for this ATM in an object-oriented fashion?

Bank ATM Software

```
abstract class NoteDispenser {
    private NoteDispenser chain;
    private int denom;
    public NoteDispenser(int d) { denom = d; }
    public void setNextChain(NoteDispenser c) {
        chain = c;
    }
    public void dispense(int amount) {
        if(amount >= denom) {
            int bills = amount / denom;
            amount = amount % denom;
            System.out.println(denom+" Bills =
"+bills);
        }
        if(amount > 0) { chain.dispense(amount); }
    }
}
```

```
class INR2000Dispenser extends NoteDispenser {
    public INR2000Dispenser() { super(2000); }
}
```

```
class INR500Dispenser extends NoteDispenser {
    public INR500Dispenser() { super(500); }
}
```

```
class INR200Dispenser extends NoteDispenser {
    public INR200Dispenser() { super(200); }
}
```

```
class INR100Dispenser extends NoteDispenser {
    public INR100Dispenser() { super(100); }
}
```

```
public class ATMMachine {
    private NoteDispenser chain1;
    public ATMMachine() {
        chain1 = new INR2000Dispenser();
        NoteDispenser chain2 = new
INR500Dispenser();
        NoteDispenser chain3 = new
INR200Dispenser();
        NoteDispenser chain4 = new
INR100Dispenser();
        chain1.setNextChain(chain2);
        chain2.setNextChain(chain3);
        chain3.setNextChain(chain4);
    }
    public void withdraw(int amount) {
        chain1.dispense(amount);
    }
    public static void main(String[] args) {
        ATMMachine atm = new ATMMachine();
        int amount = Integer.parseInt(args[0]);
        if(amount % 100 == 0)
```

Pattern: Observer

objects that listen for updates to the state of others

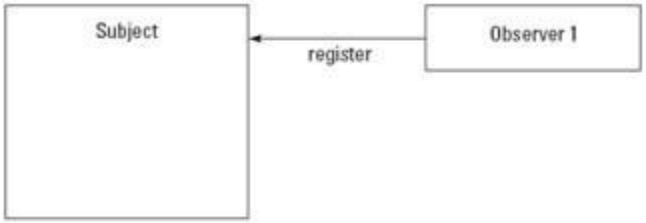
Observer Pattern

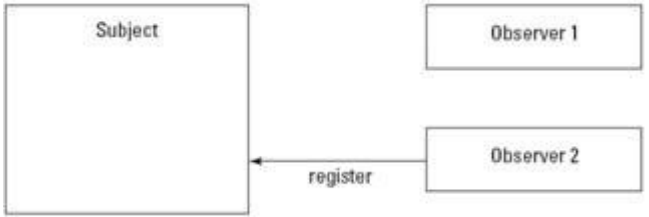
- Defines a “one-to-many” dependency between objects so that when one object changes state, all its dependents are notified and updated automatically
  - Dependence mechanism
  - Publish-subscribe
  - Broadcast
  - Change-update
- Subject
  - the object which will frequently change its state and upon which other objects depend
- Observer
  - the object which depends on a subject and updates according to its subject's state

Source: http://slideplayer.com/slide/10731275/

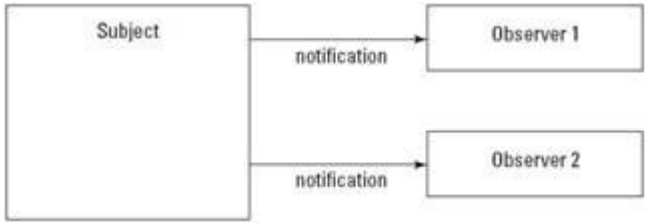
Observer Pattern - Working

- A number of Observers “register” to receive notifications of changes to the Subject. Observers are not aware of the presence of each other



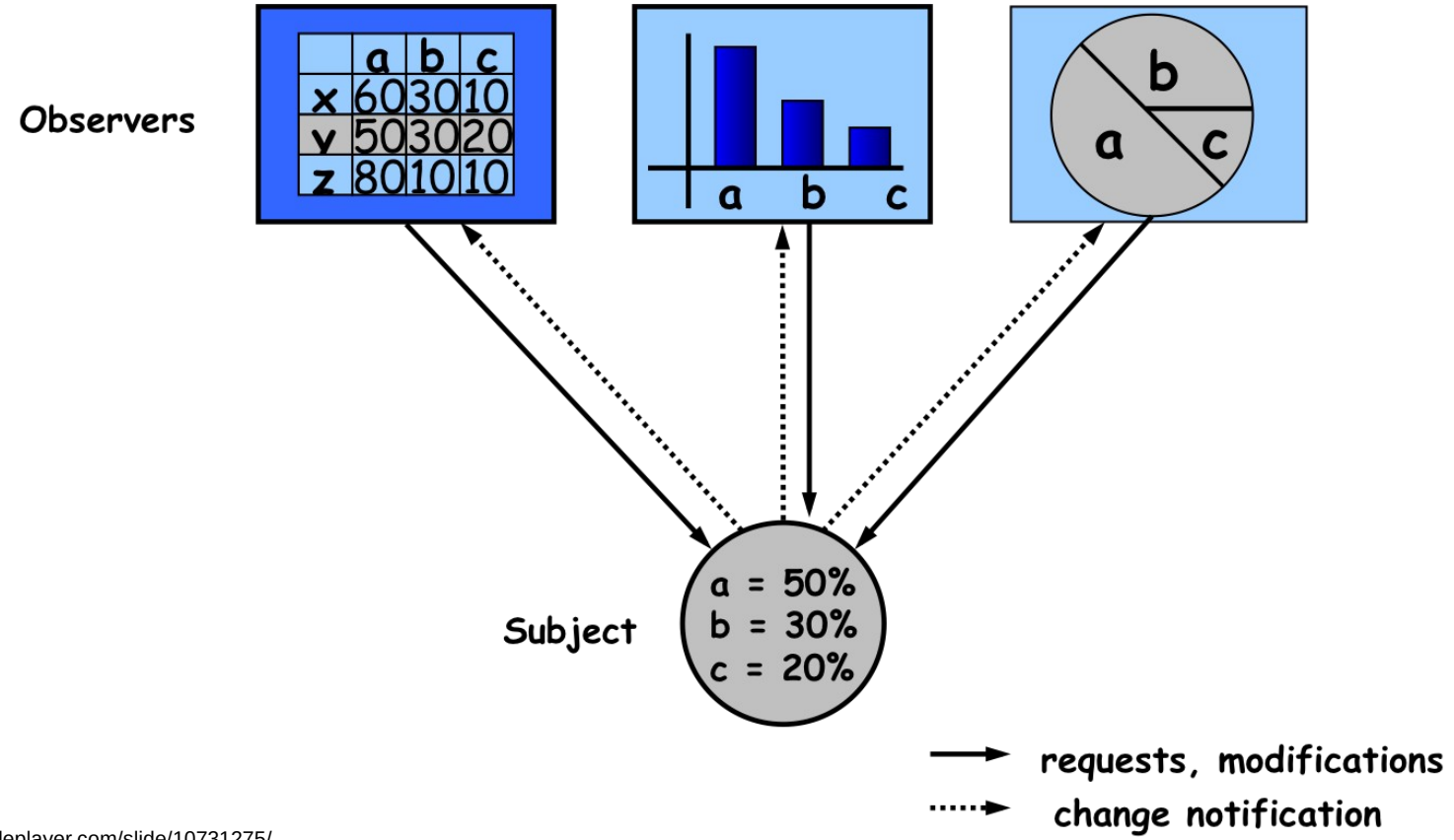


- When a certain event or “change” in Subject occurs, all Observers are “notified‘



Source: http://slideplayer.com/slide/10731275/

Observer Pattern Example



Source: http://slideplayer.com/slide/10731275/

Observer Pattern Example

- We saw the code for this example in Lecture 20
  - Marge and Simpson acts as both Observer and Subject

Let’s Implement Backpack Poll

```
interface Subject {
    public void add(Observer o);
    public void remove(Observer o);
    public void announce();
    public String getUpdate();
    public void startPoll(String msg);
}
```

```
class Backpack implements Subject {
    private List<Observer> obsvs = new
ArrayList<Observer>();
    private String discussion;
    public String getUpdate() { return discussion; }

    public void add(Observer o) {
        if(!obsvs.contains(o)) obsvs.add(o);
    }
    public void remove(Observer o) { obsvs.remove(o); }
    public void startPoll(String msg) {
        discussion = msg;
        announce();
    }
    public void announce() {
        for (Observer obj : obsvs) {
            obj.update();
        }
    }
}
```

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

```
class Student implements Observer {
    private Subject course;
    public Student(Subject s) { course = s; }
    public void update() {
        String msg = course.getUpdate();
        System.out.println("New message: "+msg);
    }
}
```

```
public class CSE201 {
    public static void main(String[] args) {
        Subject cse201 = new Backpack();
        for(int i=0; i<5; i++) {
            Observer student = new
Student(cse201);
            cse201.add(student);
        }
        cse201.startPoll("Do you want a bonus
quiz?");
    }
}
```

- Be careful about thread safety if you are using multithreading to implement this design pattern

Pattern: State

Changing behavior based on state

State Pattern

- Allows an object to alter its behavior when its internal state changes
- Uses Polymorphism to define different behaviors for different states of an object

Acknowledgement: all slides on state pattern from http://slideplayer.com/slide/3152957/

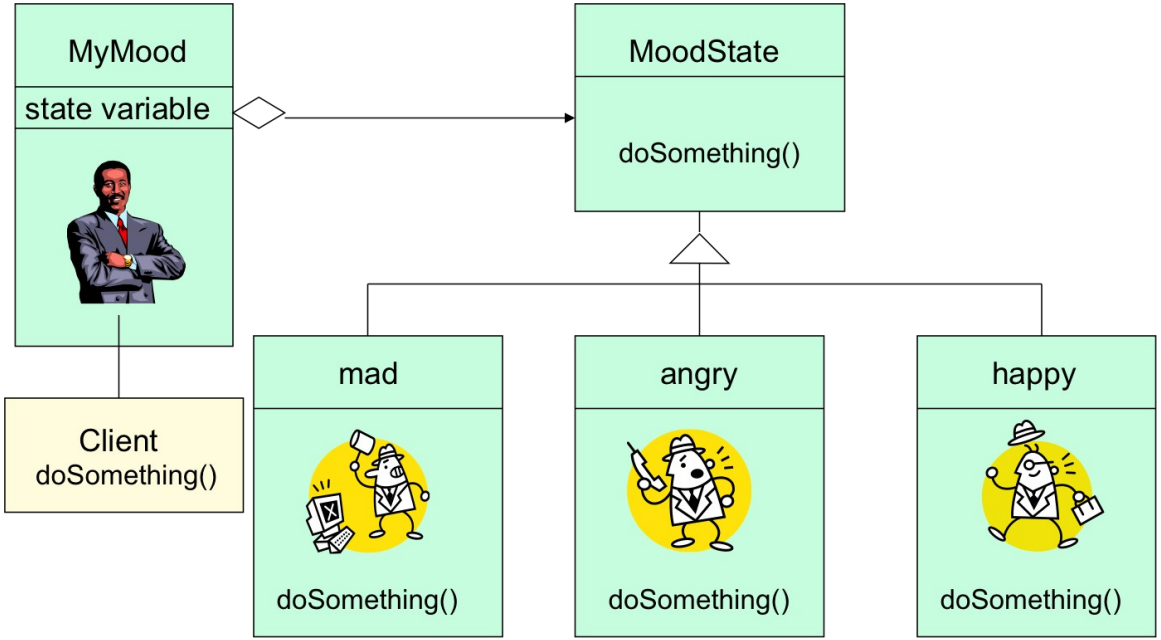
When to Use State Pattern

```
if (myself = bored) then
{
    watchMovie();
    ….
}
else if (myself = sad) then
{
    goOnDrive();
    ….
}
else if (myself = happy) then
{
    ….
```

- State pattern is useful when there is an object that can be in one of several states, with different behavior in each state
- To simplify operations that have large conditional statements that depend on the object's state

Acknowledgement: all slides on state pattern from http://slideplayer.com/slide/3152957/

How is STATE Pattern Implemented ?



- "Context" class
  - Represents the interface to the outside world
- "State" abstract class
  - Base class which defines the different states of the "state machine"
- "Derived" classes from State class
  - Defines the true nature of the state that the state machine can be in
- Context class maintains a pointer to the current state. To change the state of the state machine, the pointer needs to be changed

Acknowledgement: all slides on state pattern from http://slideplayer.com/slide/3152957/

What we Covered in GoF Patterns

- **Creational Patterns** *(abstracting the object-instantiation process)*
  - **Factory Method** **Abstract Factory** **Singleton**
  - Builder **Prototype**

- **Structural Patterns** *(how objects/classes can be combined)*
  - **Adapter** Bridge **Composite**
  - **Decorator** **Facade** **Flyweight**
  - **Proxy**

- **Behavioral Patterns** *(communication between objects)*
  - Command Interpreter **Iterator**
  - Mediator **Observer** **State**
  - **Strategy** **Chain of Responsibility** Visitor
  - **Template Method**

In 1990 a group called the Gang of Four or "GoF" (Gamma, Helm, Johnson, Vlissides) compile a catalog of design patterns in the book "Design Patterns: Elements of Reusable Object-Oriented Software"

Our Current Status (We are done!!)

- **CSE201 Post Conditions**
  1. Students are able to demonstrate the knowledge of basic principles of Object Oriented Programming such as encapsulation (classes and objects), interfaces, polymorphism and inheritance; by implementing programs ranging over few hundreds lines of code
  2. Implement basic event driven programming, exception handling, and threading
     - Already covered little bit of event driven programming in refresher module (Day 3) but we will see more
  3. Students are able to analyze the problem in terms of use cases and create object oriented design for it. Students are able to present the design in UML
     - Already covered little bit of UML but we will see more
  4. Students are able to select and use a few key design pattern to solve a given problem in hand
  5. Students are able to use common tools for testing (e.g., JUnit), debugging, and source code control as an integral part of program development

Remaining Two Lectures

- Lecture 25
  - End semester review lecture part-1
    - Generic programming
    - I/O streams
    - UML
    - Event driven programming
- Lecture 26
  - End semester review lecture part-2
    - Multithreading
    - Mutual exclusion

No recap on design patterns as we just completed it.

No more recap on inheritance, interfaces, polymorphism as we went through it several times during lectures on design patterns