CSE201: Monsoon 2024
Advanced Programming

Lecture 13: Introduction to Design Patterns

Dr. Arun Balaji Buduru

Head, Center of Technology in Policing

Founding Head, Usable Security Group (USG)

Associate Professor, Dept. of CSE | HCD

IIIT-Delhi, India

What is Design Pattern

- It is a solution for a repeatable problem in the software design
- This is not a complete design for a software system that can be directly transformed into code
- It is a description or template for how to solve the problem that can be used in many different situations

Why Study Patterns

- Reuse tried, proven solutions
    - Provides a head start
    - Avoids gotchas later (unanticipated things)
    - No need to reinvent the wheel
- Establish common terminology
    - Design patterns provide a common point of reference
    - Easier to say, “We could use Strategy here.”
- Provide a higher level prospective
    - Frees us from dealing with the details too early

“GoF” (Gang of Four) 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”

Pattern: Iterator

objects that traverse collections

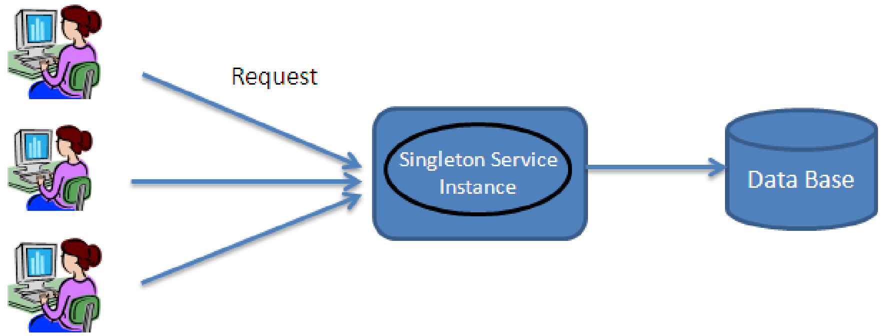
Pattern: Iterator

- Recurring Problem
  - How can you loop over all objects in any collection. You don’t want to change client code when the collection changes. Want the same methods
- Solution
  1. Provide a standard *iterator* object supplied by all data structures
  2. The implementation performs traversals, does bookkeeping
  3. The implementation has knowledge about the representation
  4. Results are communicated to clients via a standard interface
- Consequences
  - Can change collection class details without changing code to traverse the collection

Pattern: Singleton

A class that has only a single instance

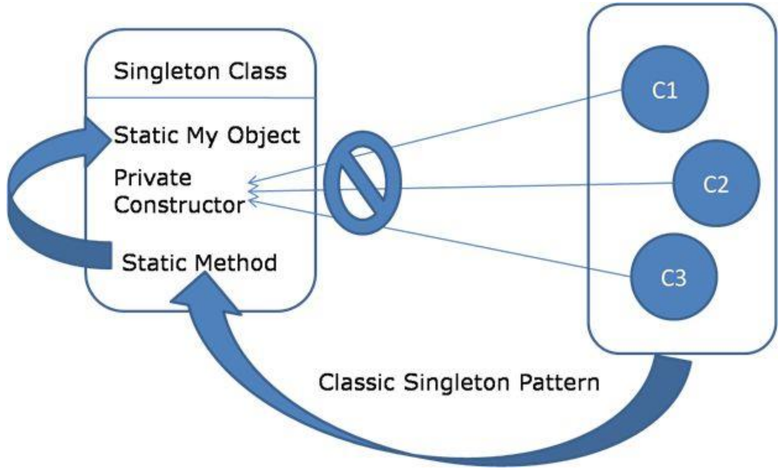
Pattern: Singleton



- Recurring problem
  - Sometimes we only ever need one instance of a particular class
  - It should be illegal to have another instance of the same class
- Solution
  - Singleton pattern – ensuring that a class has at most one instance
  - Providing global access to that instance

Fig. source: https://rajneekanth.wordpress.com/2014/04/11/what-are-design-patterns/

Implementing Singleton



1. Make constructor private so that no client is able to call it from outside
2. Declare a single private static instance of the class
3. Write a getInstance() method (or similar) that allows access to the single instance
    - Ensure thread safety in case multiple threads can access this method

Singleton Example

```
public class RandomGenerator {

    private static RandomGenerator gen = null;

    public static RandomGenerator getInstance()
    {
        if (gen == null) {
            gen = new RandomGenerator();
        }

        return gen;
    }

    private RandomGenerator() {}

    ...
}
```

- Creates a new random generator
- Clients will not use the constructor directly but will instead call getInstance to obtain
  a RandomGenerator obect that is shared by all classes in the application
- Lazy initialization
  - Can wait until client asks for the instance to create it
  - How to ensure thread safety?

Pattern: Flyweight

a class that has only one instance for each unique state

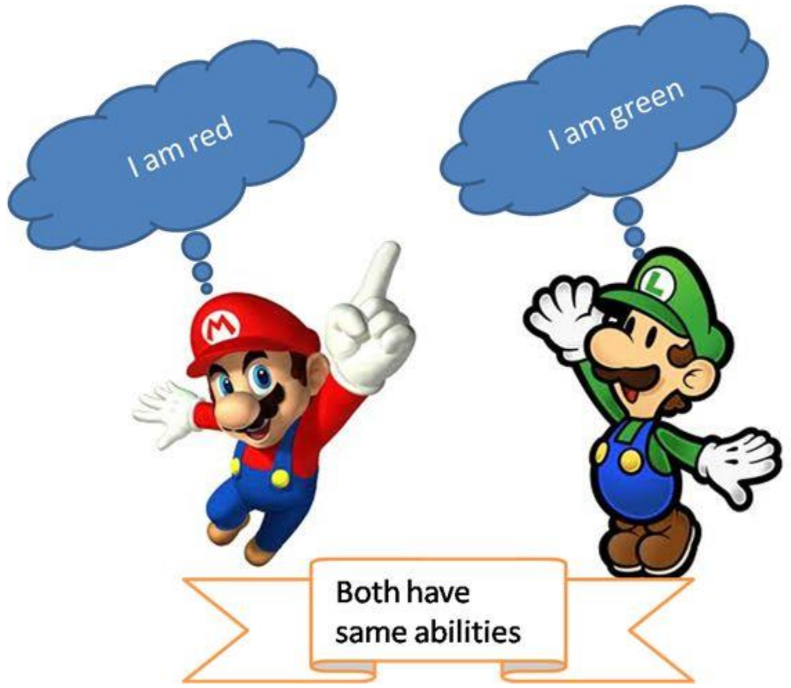
Pattern: Flyweight

- Problem
  - **Redundant objects** can bog down the system
    - Many objects have the same state

  - Example: `File` objects that represent the same file on disk
    - `new File("chatlog.txt")`
    - `new File("chatlog.txt")`
    - `new File("chatlog.txt")`
      `...`
    - `new File("notes.txt")`

  - Example: Date objects that represent the same date of the year
    - `new Date(4, 18)`
    - `new Date(4, 18)`

Pattern: Flyweight



- An assurance that no more than one instance of a class will have identical state
    - Achieved by caching identical instances of objects.
    - Similar to singleton, but one instance for each unique object state
    - Useful when there are many instances, but many are equivalent

Fig source: http://www.c-sharpcorner.com/UploadFile/SukeshMarla/learn-design-pattern-flyweight-pattern/

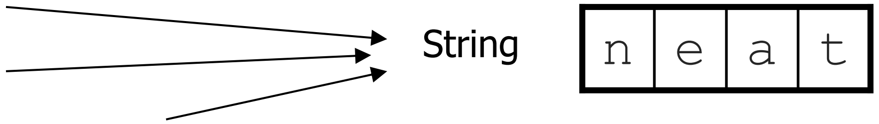
Implementing a Flyweight

```
public class Flyweighted {
    private static Map<KeyType, Flyweighted> instances
             = new HashMap<KeyType, Flyweighted>();

    private Flyweighted(...) { ... }

    public static Flyweighted getInstance(KeyType key) {
        if (!instances.contains(key)) {
            instances.put(key, new Flyweighted(key));
        }
        return instances.get(key);
    }
}
```

Flyweighting in String by JVM

- The possible combinations for Strings is close to infinite, hence JVM maintains a cache for strings, called the **string constant pool**
  - It is empty at startup and is filled constantly during the lifecycle of the JVM
- Java String objects are automatically flyweighted by the JVM **whenever** possible
  - If you declare two string variables that point to the same literal.
  - If you concatenate two string literals to match another literal

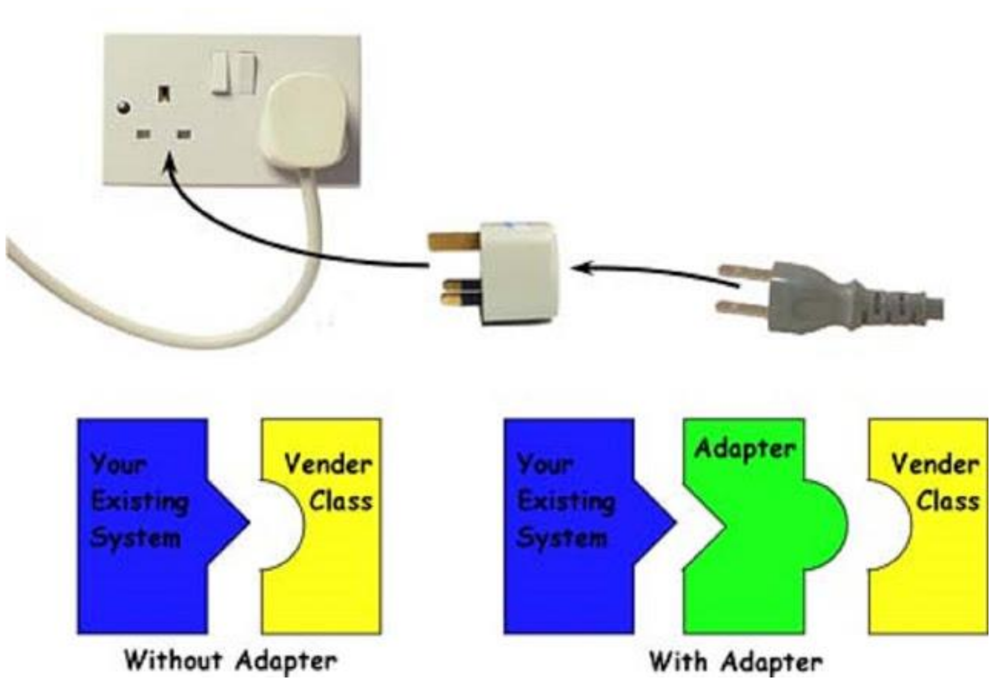
```
String a = "neat";
String b = "neat";
String c = "n" +
"eat";
```



Pattern: Adapter

an object that fits another object into a given interface

Pattern: Adapter



- Recurring problem
  - We have an object that contains the functionality we need, but not in the way we want to use it
- Solution
  - Create an **adapter object** that bridges the provided and desired functionality

Fig source: http://javarevisited.blogspot.in/2016/08/adapter-design-pattern-in-java-example.html

Adapter Pattern Example (1/2)

```
public interface Movable {
    public void move();
}

public class Car implements Movable {
    public void move() {
       System.out.println(“Car is moving”);
    }
}

public class Bike implements Movable {
    public void move() {
       System.out.println(“Bike is moving”);
    }
}
```

```
public class Vehicle {
  public static void main(String[] args) {
    List<Movable> mylist = new ArrayList<Movable>();

    mylist.add(new Car());
    mylist.add(new Bike());


    for(Movable obj: mylist) {
        obj.move();
    }
  }
}
```

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

public class Airplane implements Flyable {
   public void fly() {
      System.out.println(“Airplane is flying”);
   }
}

public class Drone implements Flyable {
   public void fly() {
      System.out.println(“Drone is flying”);
   }
}
```

- The **adaptee** interface “Flyable” only implements fly() method, although it is similar to move() in Movable inteface
- Client class, Vehicle, doesn’t understand Flyable and only use Movable
  - How to add Flyable type objects inside Movable type list in Vehicle?
  - We will code an adaptor that can serve this client by using this adaptee without any modifications

Adapter Pattern Example (2/2)

```
public interface Movable {
    public void move();
}

public class Car implements Movable {
    public void move() {
       System.out.println(“Car is moving”);
    }
}

public class Bike implements Movable {
    public void move() {
       System.out.println(“Bike is moving”);
    }
}
```

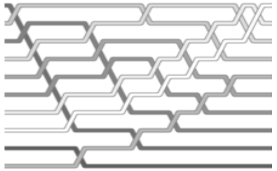
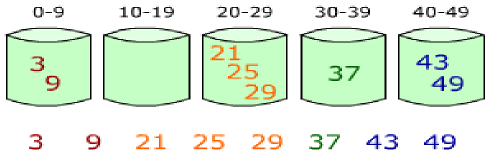
```
public interface Flyable {
   public void fly();
}

public class Airplane implements Flyable {
   public void fly() {
      System.out.println(“Airplane is flying”);
   }
}

public class Drone implements Flyable {
   public void fly() {
      System.out.println(“Drone is flying”);
   }
}
```

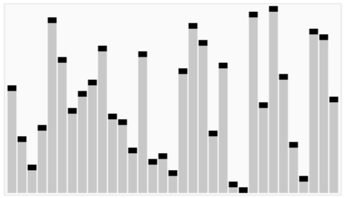
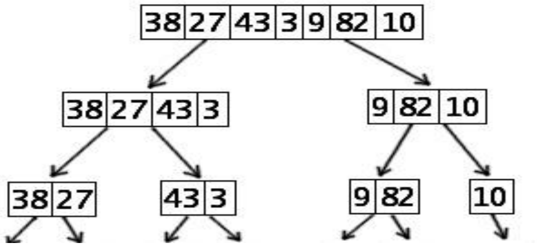
```
public class Vehicle {
  public static void main(String[] args) {
    List<Movable> mylist = new ArrayList<Movable>();

    mylist.add(new Car());
    mylist.add(new Bike());

    mylist.add(new FlyableAdapter(new Airplane()));
    mylist.add(new FlyableAdapter(new Drone()));

    for(Movable obj: mylist) {
        obj.move();
    }
  }

}
```

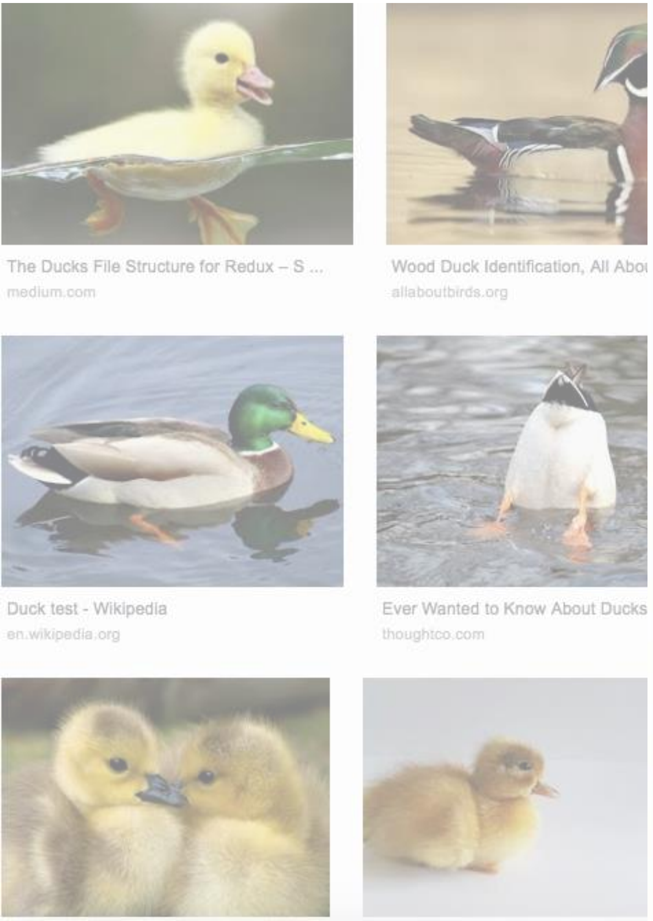
```
public class FlyableAdapter implements Movable {
    Flyable type;
    public FlyableAdapter(Flyable type) {
        this.type = type;
    }
    public void move() {
        type.fly();
    }
}
```

Pattern: Strategy

objects that hold different algorithms to solve a problem

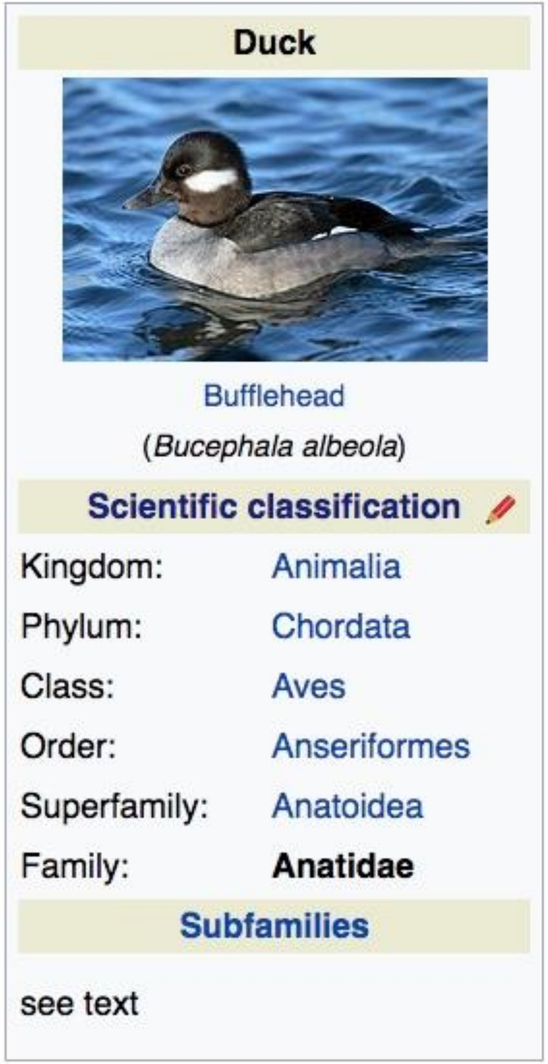


The Ducks File Structure for Redux – S ...
medium.com

Wood Duck Identification, All Abou
allaboutbirds.org

Duck test - Wikipedia
en.wikipedia.org

Ever Wanted to Know About Ducks
thoughtco.com



Duck

Bufflehead

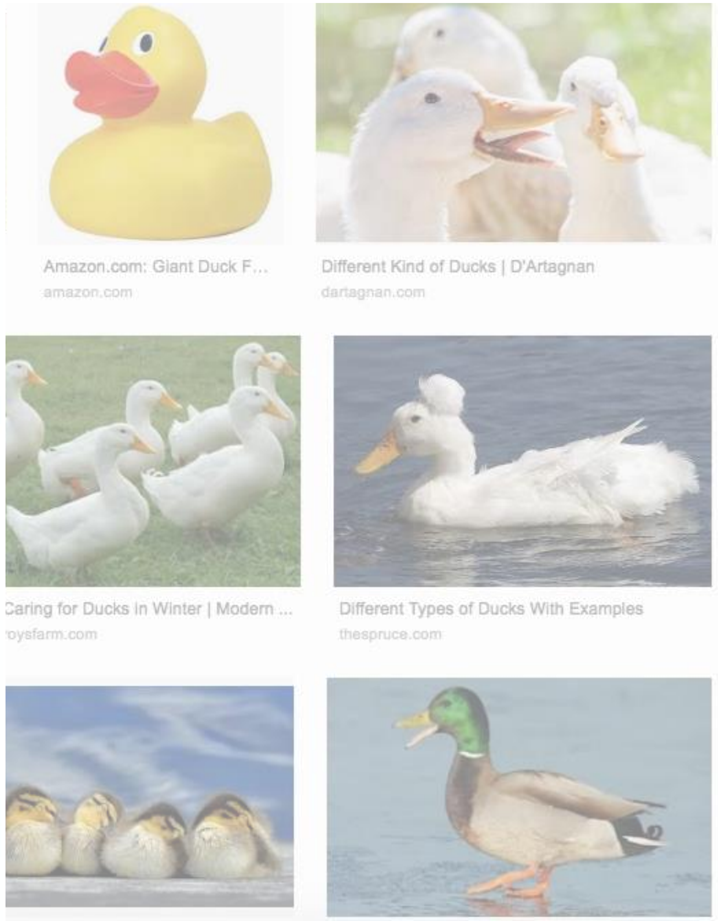
(*Bucephala albeola*)

Scientific classification

Kingdom:	Animalia
Phylum:	Chordata
Class:	Aves
Order:	Anseriformes
Superfamily:	Anatoidea
Family:	**Anatidae**

Subfamilies

see text



Amazon.com: Giant Duck F...
amazon.com

Different Kind of Ducks | D'Artagnan
dartagnan.com

Caring for Ducks in Winter | Modern ...
oysfarm.com

Different Types of Ducks With Examples
thespruce.com

Let’s Build a Duck Simulator!

- Concepts we will revisit
  - Inheritance
  - Interfaces
  - Polymorphism

What are their Characteristics?

- I’m Dabbler duck
- I can quack
- I can swim
- I can fly
- My home is on ground

- I’m Wood duck
- I can quack
- I can swim
- I can fly
- My home is on trees

How to Code a Duck Simulator?

- I’m Dabbler duck
- I can quack
- I can swim
- I can fly
- My home is on ground

- I’m Wood duck
- I can quack
- I can swim
- I can fly
- My home is on trees

Inheritance?

Lets See the Code

```
public abstract class Duck {
    private String name;
    public Duck(String n) { this.name = n; }

    public void type() {
        System.out.println("I am "+ name+" Duck");
    }
    public void speak() {
        System.out.println("I can quack");
    }
    public void swim() {
        System.out.println("I can swim");
    }
    public void fly() {
        System.out.println("I can fly");
    }
    public abstract void home();
    public void display() {
        this.type();
        this.speak();
        this.swim();
        this.fly();
        this.home();
    }
}
```

```
public class Dabbler extends Duck {
    public Dabbler() { super("Dabbler"); }

    public void home() {
        System.out.println("My home is on ground");
    }
}
```

```
public class Wood extends Duck {
    public Wood() { super("Wood"); }

    public void home() {
        System.out.println("My home is on trees");
    }
}
```

```
// Calling display on above two Duck type objects
I am Wood Duck
I can quack
I can swim
I can fly
My home is on trees
I am Dabbler Duck
I can quack
I can swim
I can fly
My home is on ground
```

Any Problems?

```
public abstract class Duck {
    private String name;
    public Duck(String n) { this.name = n; }

    public void type() {
        System.out.println("I am "+ name+" Duck");
    }
    public void speak() {
        System.out.println("I can quack");
    }
    public void swim() {
        System.out.println("I can swim");
    }
    public void fly() {
        System.out.println("I can fly");
    }
    public abstract void home();
    public void display() {
        this.type();
        this.speak();
        this.swim();
        this.fly();
        this.home();
    }
}
```

```
public class Dabbler extends Duck {
    public Dabbler() { super("Dabbler"); }

    public void home() {
        System.out.println("My home is on ground");
    }
}
```

```
public class Wood extends Duck {
    public Wood() { super("Wood"); }

    public void home() {
        System.out.println("My home is on trees");
    }
}
```

- I'm Rubber duck
- **I can squeak**
- I can swim
- **I don't fly**
- Your home is my home

What are the Issues?

- Applying inheritance for code reuse sometimes backfires
- Poor solution for maintenance
  - Our assumption that all Ducks can Fly is incorrect
  - Our assumption that all Ducks make quack-quack sound is incorrect
- How to fix this issue?
  - Overriding both the methods fly() and speak() in subclass Rubber Duck

Let’s Implement the Fix

```
public abstract class Duck {
    private String name;
    public Duck(String n) { this.name = n; }

    public void type() {
        System.out.println(“I am “+ name+” Duck”);
    }
    public void speak() {
        System.out.println(“I can quack”);
    }
    public void swim() {
        System.out.println(“I can swim”);
    }
    public void fly() {
        System.out.println(“I can fly”);
    }
    public abstract void home();
    public void display() {
        this.type();
        this.speak();
        this.swim();
        this.fly();
        this.home();
    }
}
```

```
public class Rubber extends Duck {
    public Rubber() { super(“Rubber”); }

    @Override
    public void speak() {
        System.out.println(“I can Squeak”);
    }
    @Override
    public void fly() {
        System.out.println(“I don’t Fly”);
    }
    public void home() {
        System.out.println(“Your home is my home”);
    }
}
```

```
// Calling display on Rubber Duck type object
I am Rubber Duck
I can Squeak
I can swim
I don’t Fly
Your home is my home
```

Wait.. What if we get other non-flyable Duck?

```
public abstract class Duck {
    private String name;
    public Duck(String n) { this.name = n; }

    public void type() {
        System.out.println("I am "+ name+" Duck");
    }
    public void speak() {
        System.out.println("I can quack");
    }
    public void swim() {
        System.out.println("I can swim");
    }
    public void fly() {
        System.out.println("I can fly");
    }
    public abstract void home();
    public void display() {
        this.type();
        this.speak();
        this.swim();
        this.fly();
        this.home();
    }
}
```

```
public class Rubber extends Duck {
    public Rubber() { super("Rubber"); }

    @Override
    public void speak() {
        System.out.println("I can Squeak");
    }
    @Override
    public void fly() {
        System.out.println("I don't Fly");
    }
    public void home() {
        System.out.println("Your home is my home");
    }
}
```

- If we have to code a **Domestic Duck** then they too don't fly
    - This means we need to Override the fly() method even inside Domestic Duck class

What are the Issues?

- Another Duck type could speak in a language other than “Quack” and “Squeak”
  - Examples:
    - **Decoy** Duck can’t speak
    - **Whistling** Duck make whistles
  - As there are **several possible** ways to speak, we don’t have any choice other than **Overriding** the speak() method
- However, the flying capability could be either true or false only. As the options for flying capability is limited, can we write a better code?
  - How about using an interface called Flyable that has fly() method?
    - Again there will be lot of duplicate code as each Duck type will have to implement this interface to show their flying capability

Recap: Design Principals

- Program to a supertype and not for an implementation
  - We used Duck as superclass in past
- Identify the aspects of the implementation that differs and separate them out from what stays the same
  - We took out similar functionality inside the superclass Duck and left the specialized implementation inside subclass

Using Strategy Pattern for Final Fix

1. We will still use **Flyable** interface BUT will limit its implementation in only **two** classes
2. Create a field of **Flyable** type in supertype (Duck)
3. Each subclass will simply instantiate this field inside their constructor with correct flying ability. The flying capability are defined inside the two classes mentioned in Step-1
4. display() method in Duck will use polymorphism to show the correct flying capability

Applying Strategy Pattern: The Final Fix!

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

```
public abstract class Duck {
    private String name;
    private Flyable flyStatus;
    public Duck(String n, Flyable f) {
        this.name = n;
        this.flyStatus = f;
    }
    ........
    ........
    public void tryFlying() {
        flyStatus.fly();
    }
    public void display() {
        this.type();
        this.speak();
        this.swim();
        this.tryFlying();
        this.home();
    }
}
```

```
public class CannotFly implements Flyable {
    public void fly() {
        System.out.println("I don't Fly");
    }
}
```

```
public class CanFly implements Flyable {
    public void fly() {
        System.out.println("I can Fly");
    }
}
```

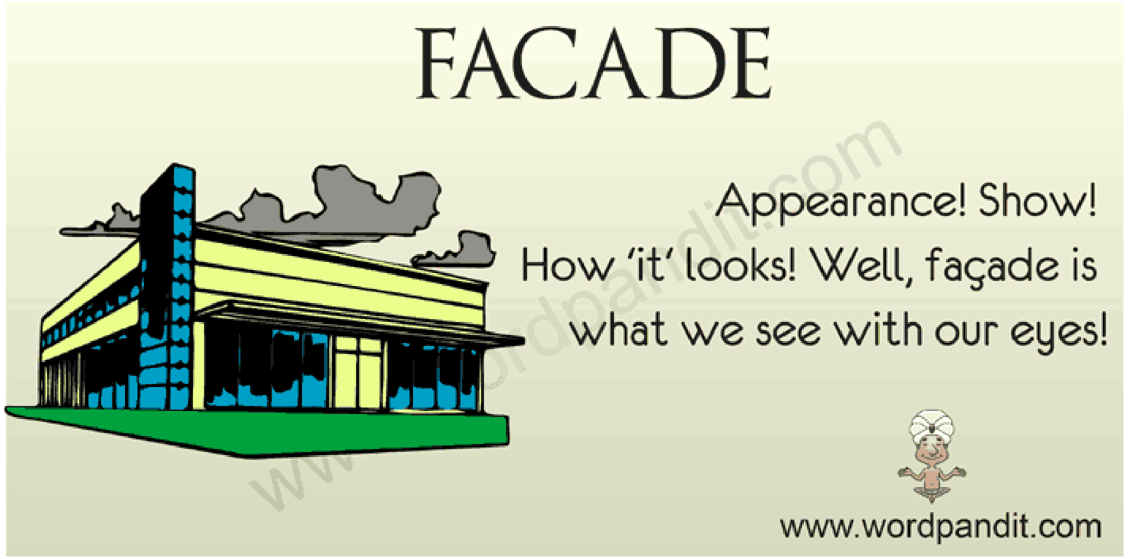
```
public class Dabbler extends Duck {
    public Dabbler() {
        super("Dabbler", new CanFly());
    }
    ........
}
```

```
public class Rubber extends Duck {
    public Rubber() {
        super("Rubber", new CannotFly());
    }
    @Override
    public void speak() {
        System.out.println("I can Squeak");
    }
    public void home() {
        System.out.println("Your home is my home");
    }
}
```

Summary: Strategy Pattern

- In Strategy pattern, a class behavior (or its algorithm) can be changed at run time
- In Strategy pattern, we create objects which represent various strategies and a context object whose behavior varies as per its strategy object
- The strategy object changes the executing algorithm of the context object
- This type of design pattern comes under behavior pattern

Pattern: Facade



Facade Pattern

- **Facade:** a structural design pattern used to identifying a simple way to realize relationships between entities
- Provide a unified “interface” to a set of interfaces in a subsystem. Facade defines a higher-level interface that makes the subsystem easier to use

The Tale of a Call Center

```
class CallCenter {
    public void handleNetwork() { / *Some code */ }
    public void handleBilling() { /* Some code */ }
    public void handleRoaming() { /* Some code */ }
    public void handleAccount() { /* Some code */ }
    ......
}
```

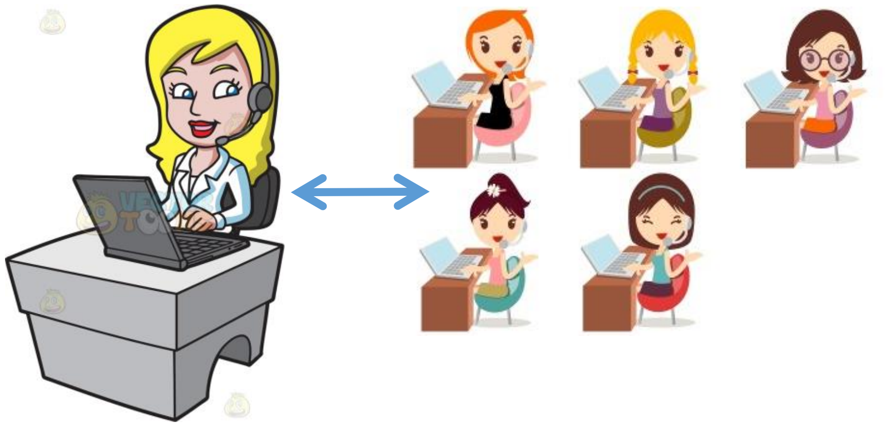
```
public class Client {
    public static void main(String[] args) {
        CallCenter c = new CallCenter();
        c.handleNetwork();
        c.handleBilling();
        c.handleRoaming();
        c.handleAccount();
    }
}
```

- Call center wants cost cutting and employees only one agent for handling all customer issues
  - Result?
    - Overloaded employee and bad customer satisfaction!

A Better Call Center Using Facade

```
class CallCenter {
    NetworkTeam net;
    BillingTeam bill;
    RoamingTeam roam;
    AccountTeam account;
    public CallCenter() { /* initializations */ }
    public void handleCalls(int option) {
        switch(option) {
            case 1:
                net.handleNetwork();
                break;
            case 2:
                bill.handleBilling();
                break;
            .......
        }
    }
}
```

```
public class Client {
    public static void main(String[] args) {
        CallCenter c = new CallCenter();
        c.handleCalls(1);
        ......
    }
}
```



- Facade design to the rescue
  - Hiding the complexities of a large body of code by providing a simplified interface

Pattern: Template

Define the skeleton of an algorithm in an operation, deferring some steps to client subclasses

Acknowledgements for slides on template pattern: Informatics 122 Software Design II, Bren School (UCI)

Let's Build a Café Simulator

- Coffee
  - Boil Water
  - Brew Coffee in boiling water
  - Pour in cup
  - Add sugar and milk

Inheritance?

- Tea
  - Boil Water
  - Steep tea in boiling water
  - Pour in cup
  - Add sugar and lemon

Let’s See the Code

```
public abstract class Cafe {
    public void boilWater() {
        System.out.println(“Boil Water”);
    }
    public void pourInCup() {
        System.out.println(“Pour in Cup”);
    }
    public abstract void prepare();
}
```

- Do you see any issues here?
  - Similar algorithms in `prepare` !!
    - How about doing the following?
      - Replace brewCoffee() and steepTeaBag() with brew()
      - Replace addSugarAndMilk() and addSugarAndLemon() with addCondiments()

```
public class Coffee extends Cafe {
    public void prepare() {
        boilWater();
        brewCoffee();
        pourInCup();
        addSugarAndMilk();
    }
    private void brewCoffee() {
        System.out.println(“Brew Coffee”);
    }
    private void addSugarAndMilk() {
        System.out.println(“Add Sugar and Milk”);
    }
}
```

```
public class Tea extends Cafe {
    public void prepare() {
        boilWater();
        steepTeaBag();
        pourInCup();
        addSugarAndLemon();
    }
    private void steepTeaBag() {
        System.out.println(“Steep Tea Bag”);
    }
    private void addSugarAndLemon() {
        System.out.println(“Add Sugar and Lemon”);
    }
}
```

Template Pattern

- The Template Method pattern defines the skeleton of an algorithm in a method, deferring some steps to subclasses
- Template Method lets subclasses redefine certain steps of an algorithm without changing the algorithm’s structure
- Usage
  1. Define the algorithm in superclass and ensure that subclasses cannot change the structure of this algorithm
  2. Each step of the algorithm is represented by a method
  3. Steps (methods) handled by subclasses are declared abstract
  4. Shared steps (concrete methods) are placed in the superclass

The Fixed Code

```
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”);
    }
}
```

Pattern: Composite

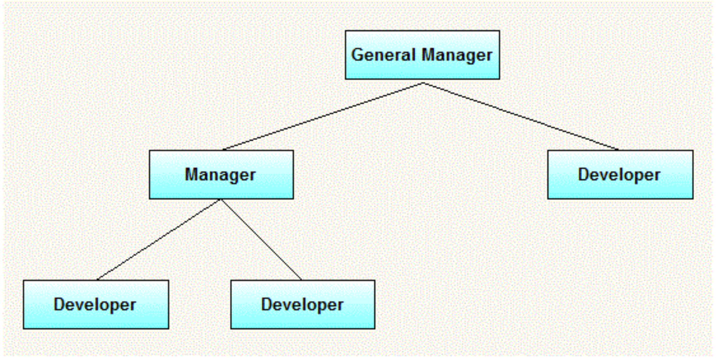
objects that can contain their own type

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”);
    }
}
```

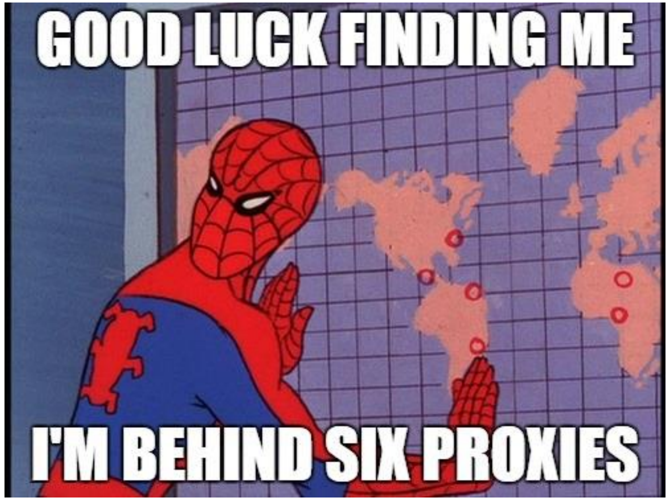
```
public class Client {
    public static void main(String[] args) {
       Employee gm = new Manager();
       Employee emp1 = new Developer();
       Employee manager = new Manager();
       Employee emp2 = new Developer();
       Employee emp3 = new Developer();
       gm.add(emp1); gm.add(manager);
       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

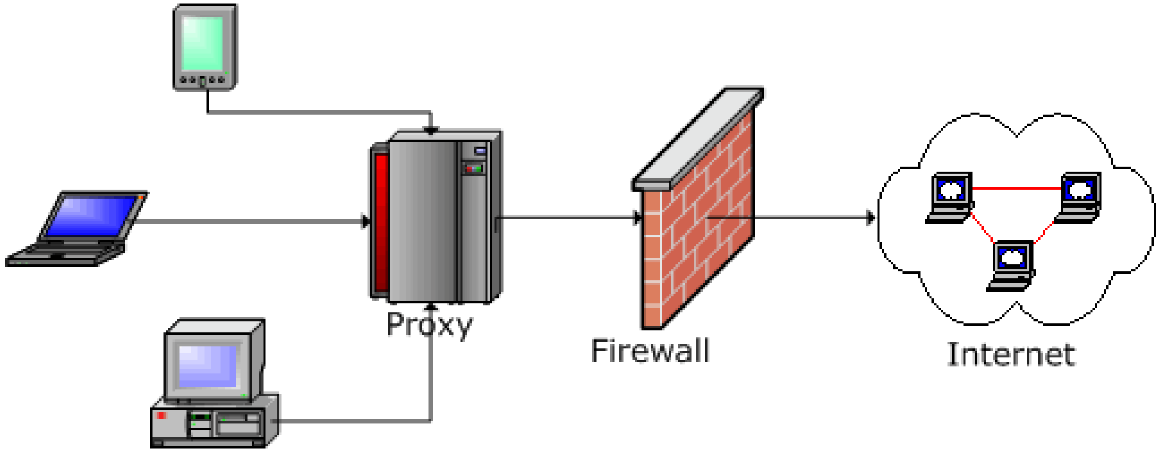
Controls and manages access to objects they are protecting



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);
}
```

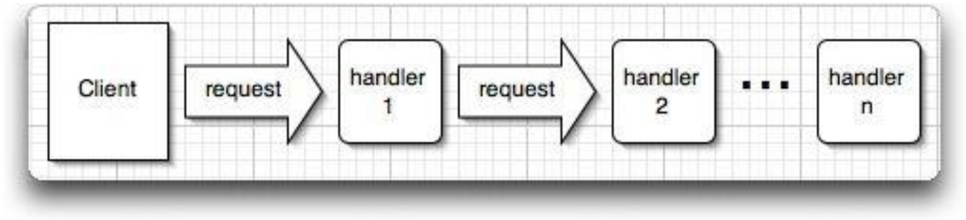
```
class Intranet implements IntranetAccess {
    public void getAccess(String name) {
        System.out.println(“Unrestricted access
                                    granted to “+
name);
    }
}
```

```
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

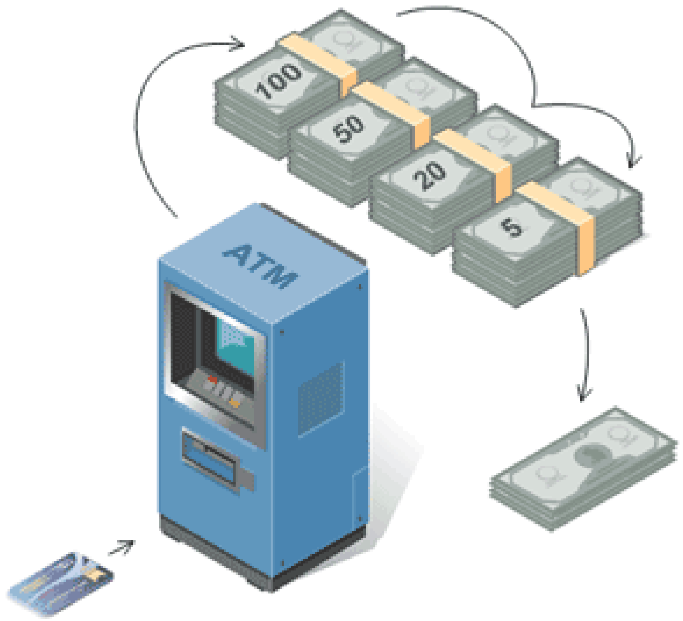
Gives more than one object an opportunity to handle a request by linking receiving objects together



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) { atm.withdraw(amount); }
    }
}
```

Pattern: Observer

objects that listen for updates to the state of others

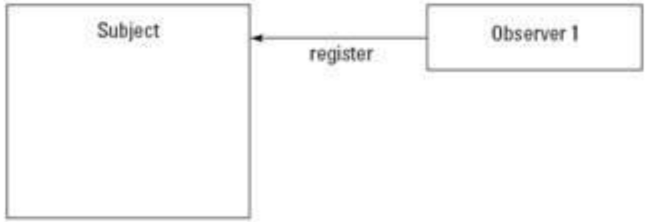
Observer Pattern

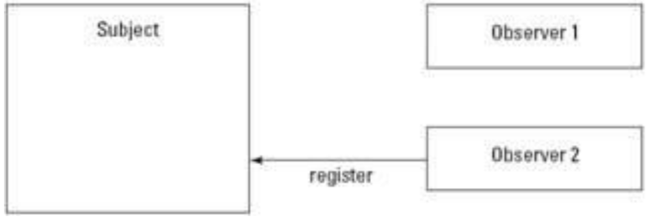
- Defines a “one-to-many” dependency between objects so that when one object changes state, all its dependents are notified and updated automatically
    - Dependence mechanism
    - Publish-subscribe
    - Broadcast
    - Change-update
- Subject
    - the object which will frequently change its state 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/

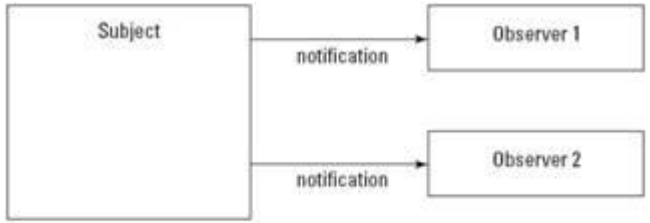
Observer Pattern - Working

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



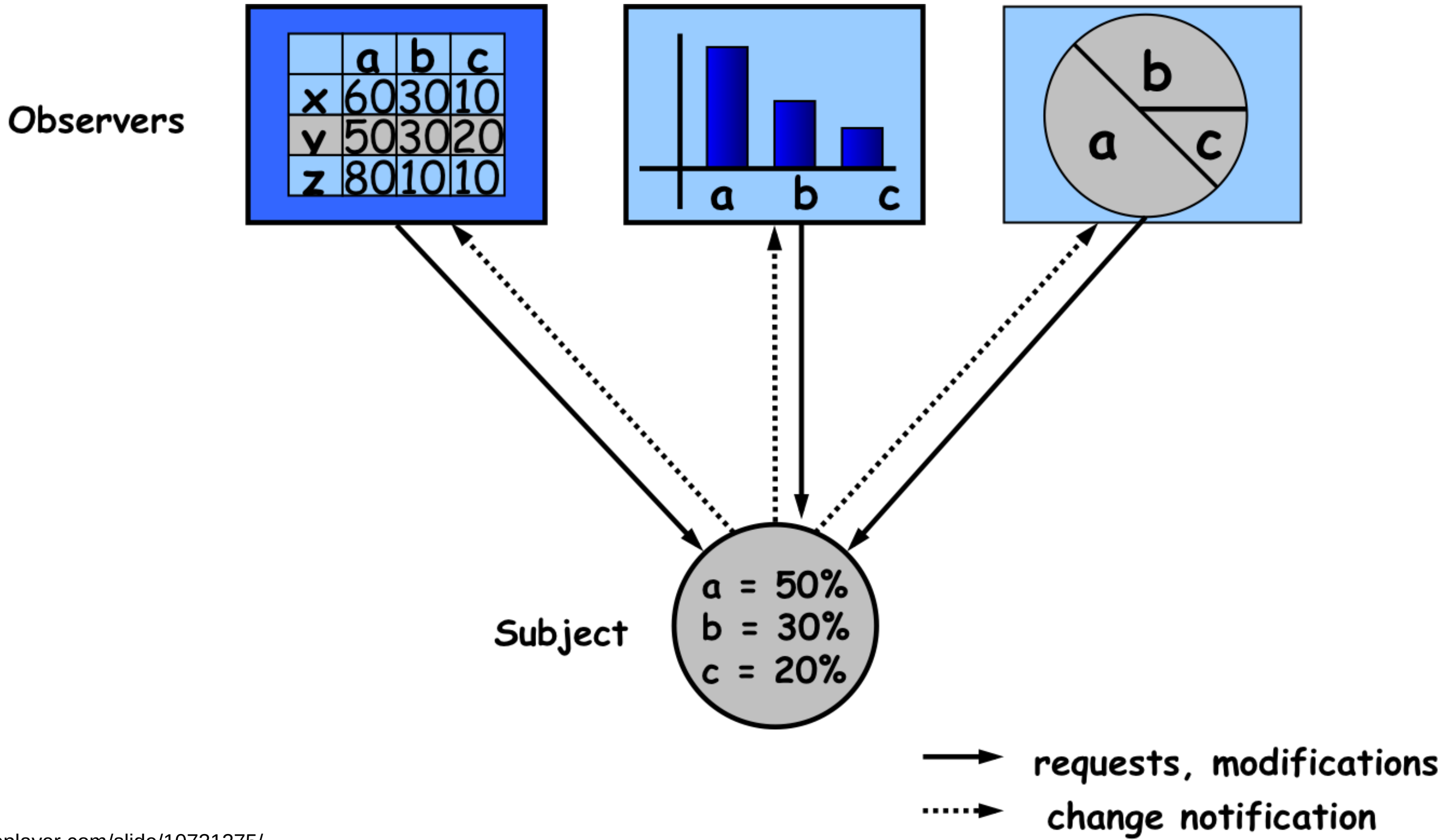


- When a certain event or “change” in Subject occurs, all Observers are “notified‘



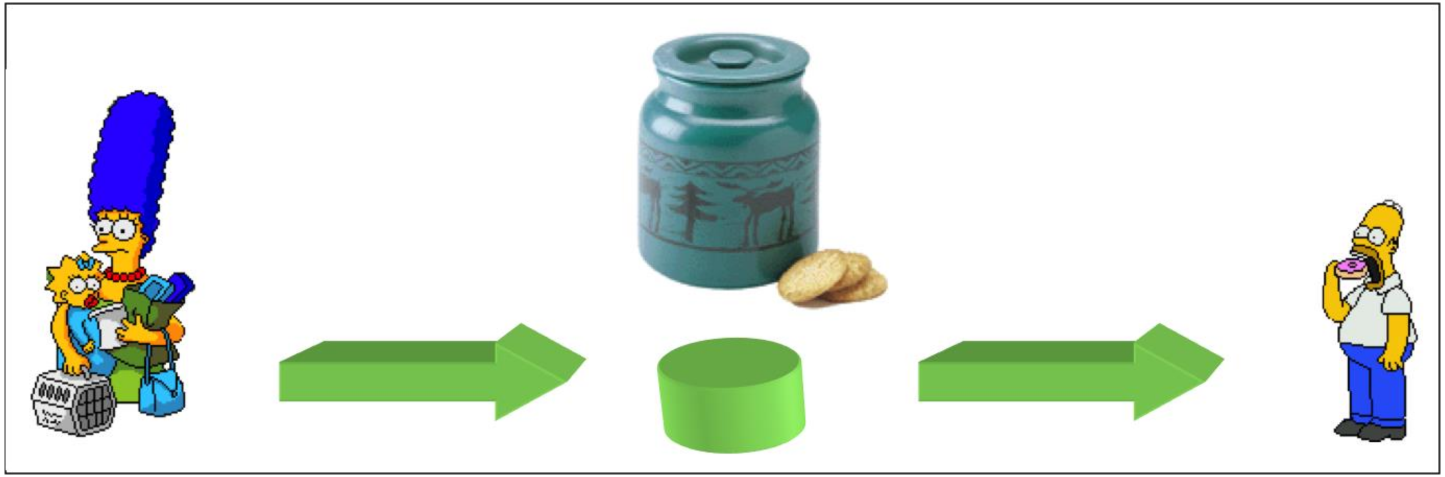
Source: http://slideplayer.com/slide/10731275/

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/

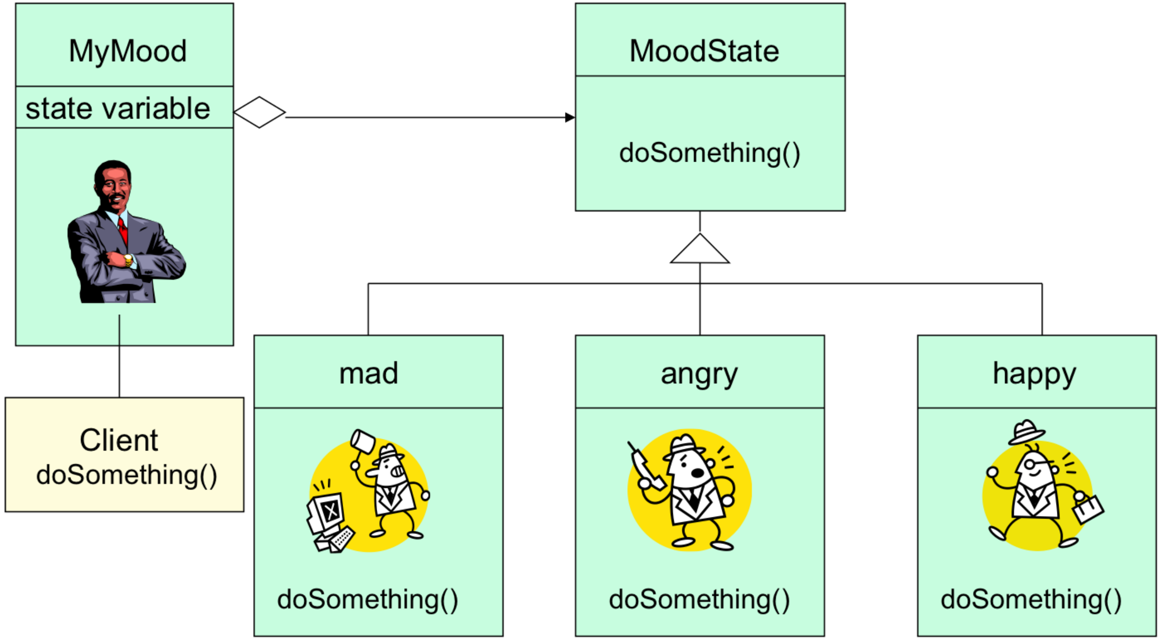
When to Use State Pattern

```
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"