

CSE201: Monsoon 2020  
Advanced Programming

# **Lecture 24: Template, Prototype, Factory & Façade Design Patterns**

Vivek Kumar

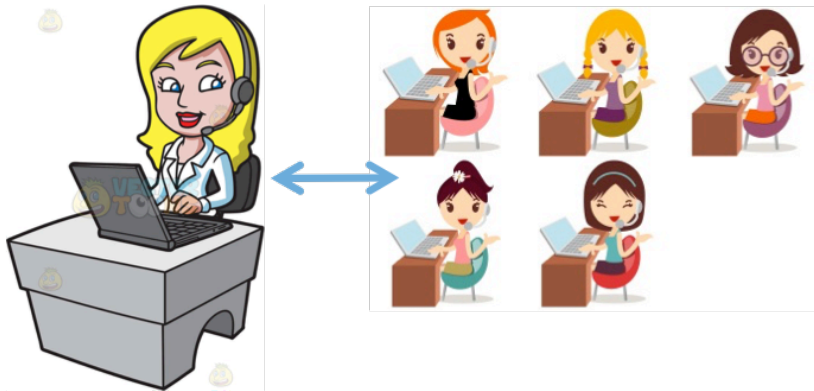
Computer Science and Engineering

IIT Delhi

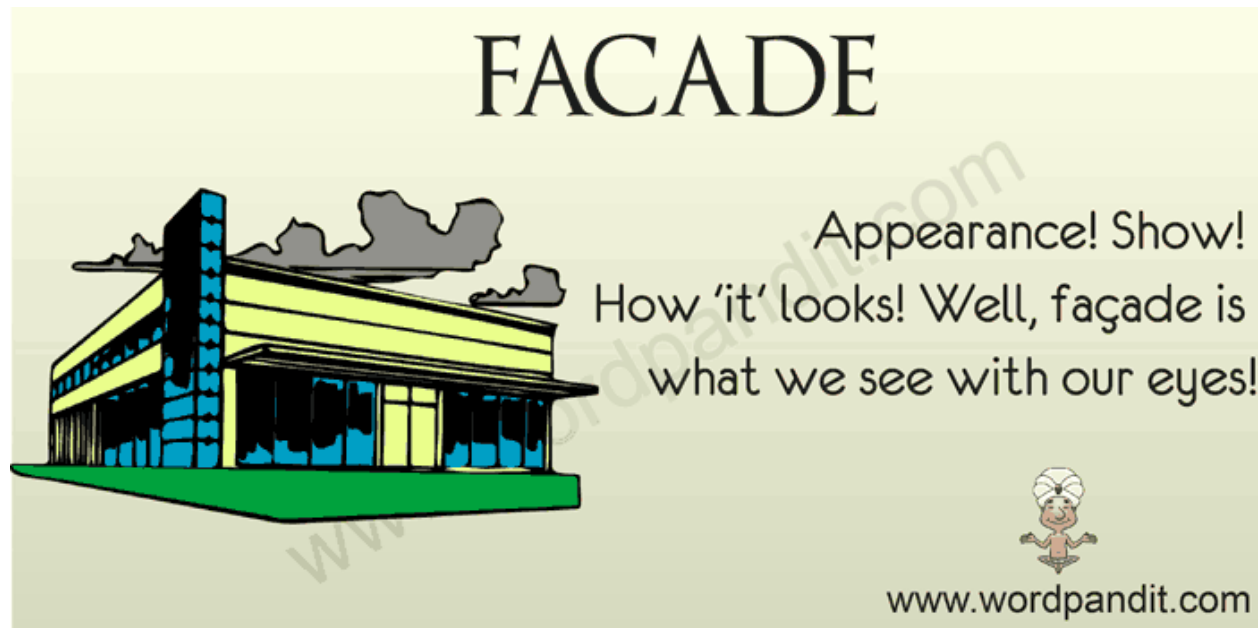
[vivekk@iiitd.ac.in](mailto:vivekk@iiitd.ac.in)

# Today's Lecture

- Some more design patterns
  - Template (DP # 6)
  - Prototype (DP # 7)
  - Factory (DP # 8)
    - Abstract Factory (DP # 9)
  - Façade (DP # 10)



# 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();  
    }  
}
```

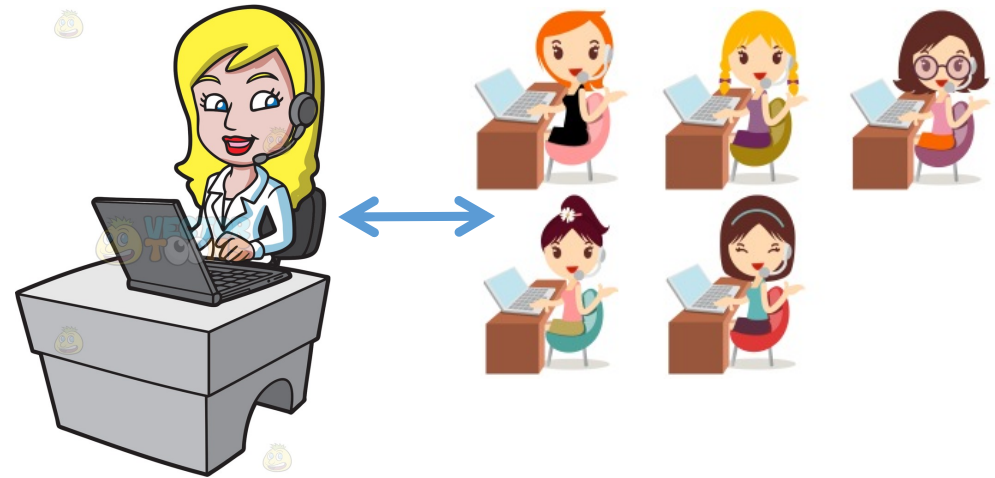


- Call center wants cost cutting and employees only one agent for handling all customer issues
  - Result?
    - Overloaded employee and bad customer satisfaction!

# 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*

# 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: Prototype

*An object that serves as a basis for creation of others*



Source of this slide: CSE331, Washington University

# Let's Build a Cloning Laboratory Simulator

- We are going to **clone** following Animals in our lab
  - Sheep
    - “Is an” Animal but has wool
  - Chicken
    - “Is an” Animal but lay eggs
- Which concepts we will be using?
  - Inheritance
  - Object cloning



# Cloning Lab Simulator

```
public class Animal {  
    private String name;  
    public Animal(String n) { name=n; }  
    public void sayHello() {  
        System.out.println("I am a " + name);  
    }  
}
```

```
public class Lab1 {  
    public static Sheep getClone(Sheep s)  
        throws CloneNotSupportedException {  
        return s.clone();  
    }  
}
```

```
public class Lab2 {  
    public static Chicken getClone(Chicken s)  
        throws CloneNotSupportedException {  
        return s.clone();  
    }  
}
```

```
public class Sheep extends Animal implements Cloneable {  
    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 Chicken extends Animal implements Cloneable {  
    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 class Client {  
    public static void main(String[] args) throws CloneNotSupportedException{  
        Sheep s1 = new Sheep(); Chicken c1 = new Chicken();  
        Sheep s2 = Lab1.getClone(s1);  
        Chicken c2 = Lab2.getClone(c2);  
    }  
}
```

# What are the Issues?

- Instead of having just one laboratory for all Animal types, we ended up creating individual Animal specific laboratory
  - No use of polymorphism!
- Client has to ensure he requests the laboratory suited for his Animal type
- Code duplication!
  - More serious when we need to code some more Animal types (Cow, Dog, etc.)



# Prototype Pattern

- **Problem:** Client wants another object similar to an existing one, but doesn't care about the details of the state of that object
  - Creating an instance of a class is time-consuming or complex in some way
- **Solution**
  - Decouple product creation from system behavior
  - Avoid subclasses of an object creator in the client application



# The Fixed Version

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

Sheep and Chicken also requires clone() implementation to enable deep copy (if any such fields are there in class)

```
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 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();
    }
}
```

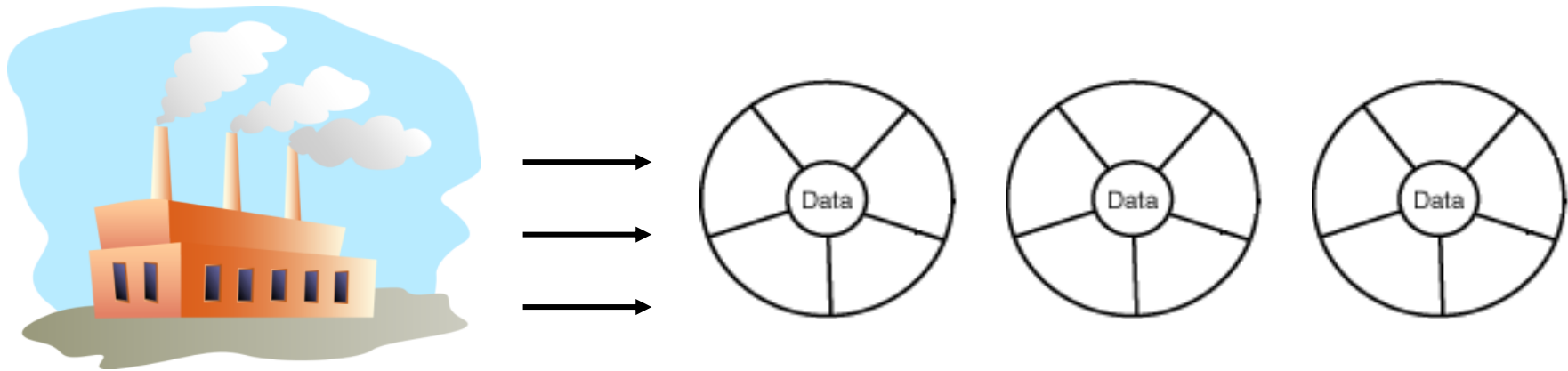
```
public class Client {
    public static void main(String[] args) throws CloneNotSupportedException{
        Animal s1 = new Sheep(); Animal c1 = new Chicken();
        Animal s2 = Lab.getClone(s1);
        Animal c2 = Lab.getClone(c1);
    }
}
```

# Drawback of Prototype Pattern

- It is built on the method `clone()`, which could be complicated sometimes in terms of shallow copy and deep copy

# Pattern: Factory

*A method or object that creates other objects*



# Let's Revisit our Client from Cloning Laboratory

```
public class Client {  
    public static void main(String[] args) throws  
CloneNotSupportedException{  
        String need = args[0];  
        Animal animal;  
        if(need.equals("wool") {  
            animal = new Sheep();  
        }  
        else if(need.equals("eggs") {  
            animal = new Chicken();  
        }  
        else if(need.equals("milk") {  
            animal = new Cow();  
        }  
        else System.exit(-1);  
        // Our client is too greedy  
        Animal[] cloned = new Animal[100];  
        for(int i=0; i<cloned.length; i++) {  
            cloned[i] = Lab.getClone(animal);  
        }  
    }  
}
```

- We have got more funding and our lab now support some more Animals!
  - Our client now has options to choose Animals based on his requirements
- What is the issue here?
  - Mixing two events in same place (or method)
    - Animal creation
    - Cloning of Animal

# The Issue with “new”

```
public class Client {  
    public static void main(String[] args) throws  
CloneNotSupportedException{  
        String need = args[0];  
        Animal animal;  
        if(need.equals("wool") {  
            animal = new Sheep();  
        }  
        else if(need.equals("eggs") {  
            animal = new Chicken();  
        }  
        else if(need.equals("milk") {  
            animal = new Cow();  
        }  
        else if(need.equals("protection") {  
            animal = new Dog();  
        }  
        else if(need.equals("riding") {  
            animal = new Horse();  
        }  
        .....  
    }  
}
```

- When we have several related classes, that's an indication that they might change in future
  - We might expand our Lab to support cloning of several other Animals...
- What is the issue?
  - Client code needs to be recompiled:
    - Every time we **add** the support for a new Animal in our Lab
    - Every time if we **remove** the support for an existing Animal in our Lab

# Factory Pattern

- **Factory:** A method or object whose primary purpose is to manage the creation of other objects (usually of a different type)
- **Problem:** Object creation is cumbersome or heavily coupled for a given client. Client needs to create but doesn't want the details.
- **Solution:** A helper method that creates and returns the object(s)

# The Fix: Encapsulate Creation Code

```
public class AnimalFactory {  
    public Animal createAnimal(String need) {  
        if(need.equals("wool") {  
            return new Sheep();  
        }  
        else if(need.equals("eggs") {  
            return new Chicken();  
        }  
        .....  
    }  
}
```

```
public class Client {  
    public static void main(String[] args) throws  
CloneNotSupportedException{  
        String need = args[0];  
        AnimalFactory factory = new AnimalFactory();  
        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);  
        }  
    }  
}
```

- What are the benefits?
  - Client need not recompile if support for Animals are added or removed in our Lab
  - Easy to serve some other Client class
  - Ensure consistent object initialization

# We Have Another Problem Now...

```
public class AnimalFactory {  
    public Animal createAnimal(String need) {  
        if(need.equals("wool") {  
            return new Sheep();  
        }  
        else if(need.equals("eggs") {  
            return new Chicken();  
        }  
        .....  
    }  
}
```

```
public class Client {  
    public static void main(String[] args) throws  
CloneNotSupportedException{  
        String need = args[0];  
        AnimalFactory factory = new AnimalFactory();  
        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);  
        }  
    }  
}
```

- Our cloning Lab is in very high demand and we have started cloning almost every Animal (*except ourselves...*)
- Supporting creation of so many Animals in just AnimalFactory class is becoming a bottleneck



# Abstract Factory Pattern

- A superclass factory that can be extended to provide different sub-factories, each with different features
- Used when we have **multiple families of object components**



www.shutterstock.com · 157937117



Cat Family



Dog Family

© Vivek Kumar

# The Fix: Abstract Factory Pattern

```
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);  
        }  
    }  
}
```

# Next Lecture

- Remaining 6 more design patterns