
بسم الله الرحمن الرحيم

- <u>Software Engineering (CSC291)</u>

Lecture 07

Design Patterns

Overview

- Motivation
- Design Patterns
- Benefits of Design Patterns
- Elements of Design Patterns
- Patterns you have already seen
- Types of Design Pattern
- Examples

Motivation & Concept

- OOD methods emphasize design notations
  - Fine for specification, documentation
- But OOD is more than just drawing diagrams
  - Good draftsmen ? good designers
- Good OO designers rely on lots of experience
  - At least as important as syntax
- Most powerful reuse is *design* reuse
  - Match problem to design experience

Motivation & Concept (cont'd)
Recurring Design Structures

- OO systems exhibit recurring structures that promote
  - abstraction
  - flexibility
  - modularity
  - elegance
- Therein lies valuable design knowledge

Problem:

capturing, communicating, & applying this knowledge

What is a Design Pattern?

- A (Problem, Solution) pair.

- A technique to repeat designer success.

- Borrowed from Civil and Electrical Engineering domains.

Patterns in engineering

- How do other engineers find and use patterns?
  - Mature engineering disciplines have handbooks describing successful solutions to known problems
  - Automobile designers don't design cars from scratch using the laws of physics
  - Instead, they reuse standard designs with successful track records, learning from experience
  - *Should software engineers make use of patterns? Why?*
  - “Be sure that you make everything according to the pattern
- Developing software from scratch is also expensive
  - Patterns support reuse of software architecture and design

How Patterns are used?

Designer

- Design Problem.
- Solution.
- Implementation details.

Programmer

Design

Reduce gap

Implementation

Design Patterns

- Design patterns represent the best practices used by experienced object-oriented software developers.
- Design patterns are solutions to general problems that software developers faced during software development.
- These solutions were obtained by trial and error by numerous software developers over quite a substantial period of time.

Definitions

- A *pattern* is a recurring solution to a standard problem, in a context.
- Christopher Alexander, a professor of architecture…
  - “A pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice.”

OK

What is Gang of Four (GOF)?

- In 1994, four authors Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides published a book titled *Design Patterns - Elements of Reusable Object-Oriented Software* which initiated the concept of Design Pattern in Software development.
- These authors are collectively known as Gang of Four (GOF).
- According to these authors design patterns are primarily based on the following principles of object orientated design.
  - Program to an interface not an implementation
  - Favor object composition over inheritance

Benefits of Design Patterns

- Design patterns enable large-scale reuse of software architectures and also help document systems
- Patterns explicitly capture expert knowledge and design tradeoffs and make it more widely available
- Patterns help improve developer communication
- Pattern names form a common vocabulary

Elements of Design Patterns

- Design patterns have 4 essential elements:
  - Pattern name: increases vocabulary of designers
  - Problem: intent, context, when to apply
  - Solution: UML-like structure, abstract code
  - Consequences: results and tradeoffs

Name of Design Pattern

- Describe a design problems and its solutions in a word or two
- Used to talk about design pattern with our colleagues
- Used in the documentation
- Increase our design vocabulary
- Have to be coherent

Problem

- Describes when to apply the patterns
- Explains the problem and its context
- Sometimes include a list of conditions that must be met before it makes sense to apply the pattern
- Have to occurs over and over again in our environment

Solution

- Describes the elements that make up the design, their relationships, responsibilities and collaborations

- Does not describe a concrete design or implementation

- Has to be well proven in some projects

Consequences

- Results and trade-offs of applying the pattern
- Helpful for describing design decisions, for evaluating design alternatives
- Benefits of applying a pattern
- Impacts on a system’s flexibility, extensibility or portability

Design patterns you have already seen

- Encapsulation (Data Hiding)
- Subclassing (Inheritance)
- Iteration
- Exceptions

OK

Encapsulation pattern

- **Problem:** Exposed fields are directly manipulated from outside, leading to undesirable dependences and inconsistencies in data.

- **Solution:** Hide some components, permitting only stylized access to the object.

Subclassing pattern

- **Problem:** Similar abstractions have similar members (fields and methods). Repeating these is tedious, error-prone, and a maintenance headache.

- **Solution:** Inherit default members from a superclass; select the correct implementation via run-time dispatching.

Iteration pattern

- **Problem:** Clients that wish to access all members of a collection must perform a specialized traversal for each data structure.

- **Solution:** Implementations perform traversals. The results are communicated to clients via a standard interface.

Exception pattern

- **Problem:** Code is cluttered with error-handling code.

- **Solution:** Errors occurring in one part of the code should often be handled elsewhere. Use language structures for throwing and catching exceptions.

OK

Types of Design Patterns

- Creational Patterns
- Structural Patterns
- Behavioral Patterns

Creational Patterns

- These design patterns provide a way to create objects while hiding the creation logic, rather than instantiating objects directly using new operator.
- Deal with initializing and configuring classes and objects
- This gives more flexibility to the program in deciding which objects need to be created for a given use case.

Creational Patterns

- Factory
- Factory Method
- Abstract Factory
- Singleton
- Builder
- Prototype
- Object Pool

Structural Patterns

- These design patterns concern class and object composition.
- Concept of inheritance is used to compose interfaces and define ways to compose objects to obtain new functionalities.
- Deal with decoupling interface and implementation of classes and objects

Structural Patterns

- Adapter
- Bridge
- Filter
- Composite
- Decorator
- Façade
- Flyweight
- Proxy

Behavioral Pattern

- These design patterns are specifically concerned with communication between objects.
- Deal with dynamic interactions among societies of classes and objects
- How they distribute responsibility

Behavioral Pattern

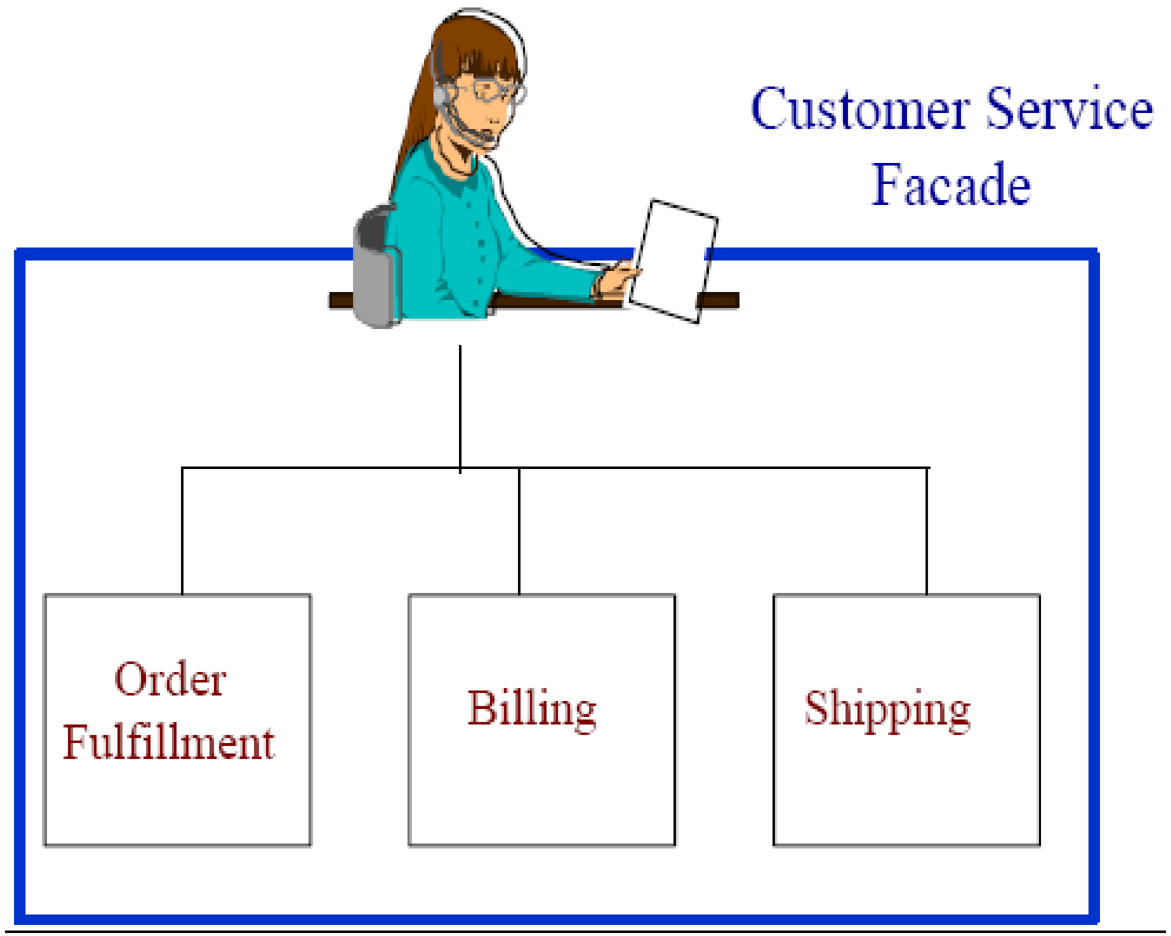
- Chain of Responsibility
- Command
- Interpreter
- Iterator
- Mediator
- Memento
- Observer
- State
- NULL Object Pattern
- Strategy
- Template Method
- Visitor

Some examples of Design Patterns

Template for discussion

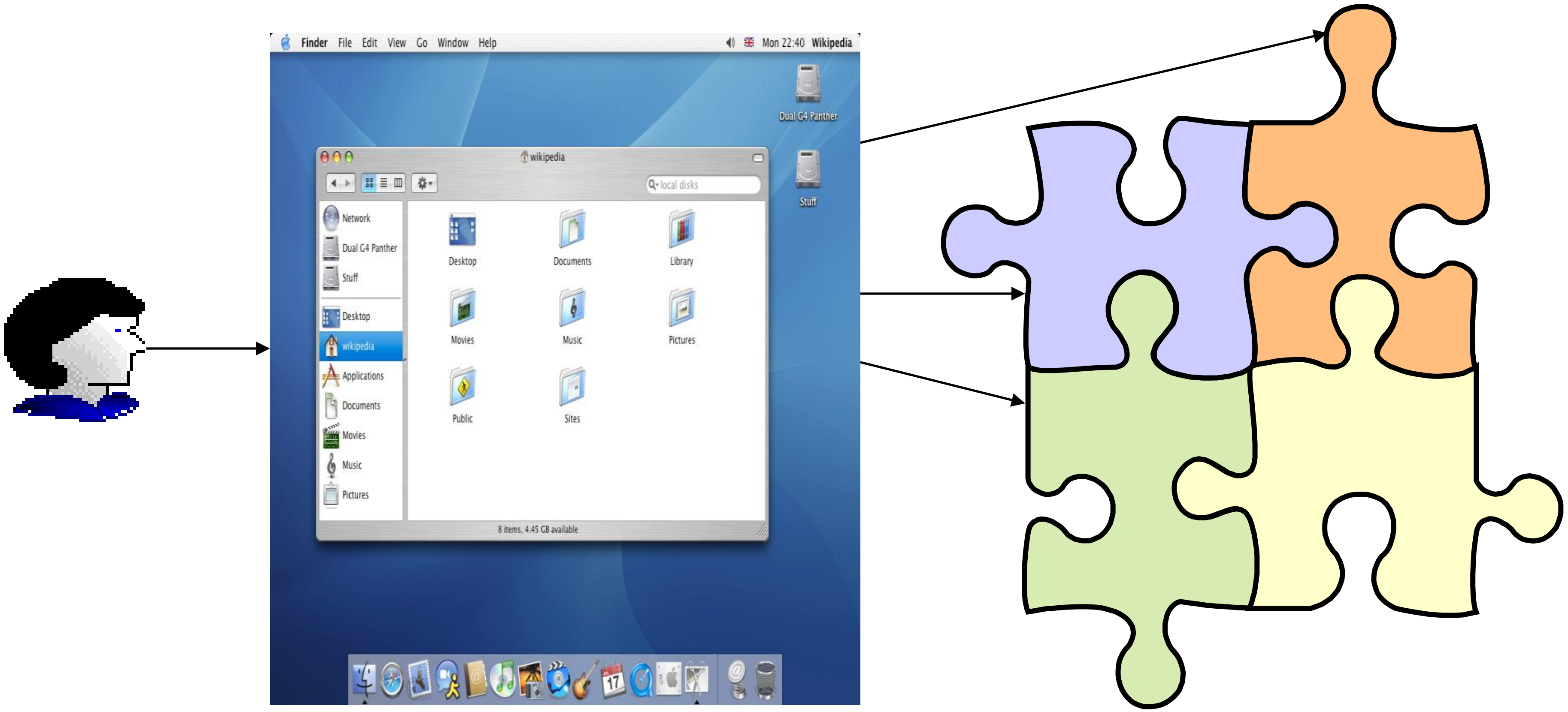
- Non-software example.
- Software counterpart example.

Facade (Non software example)

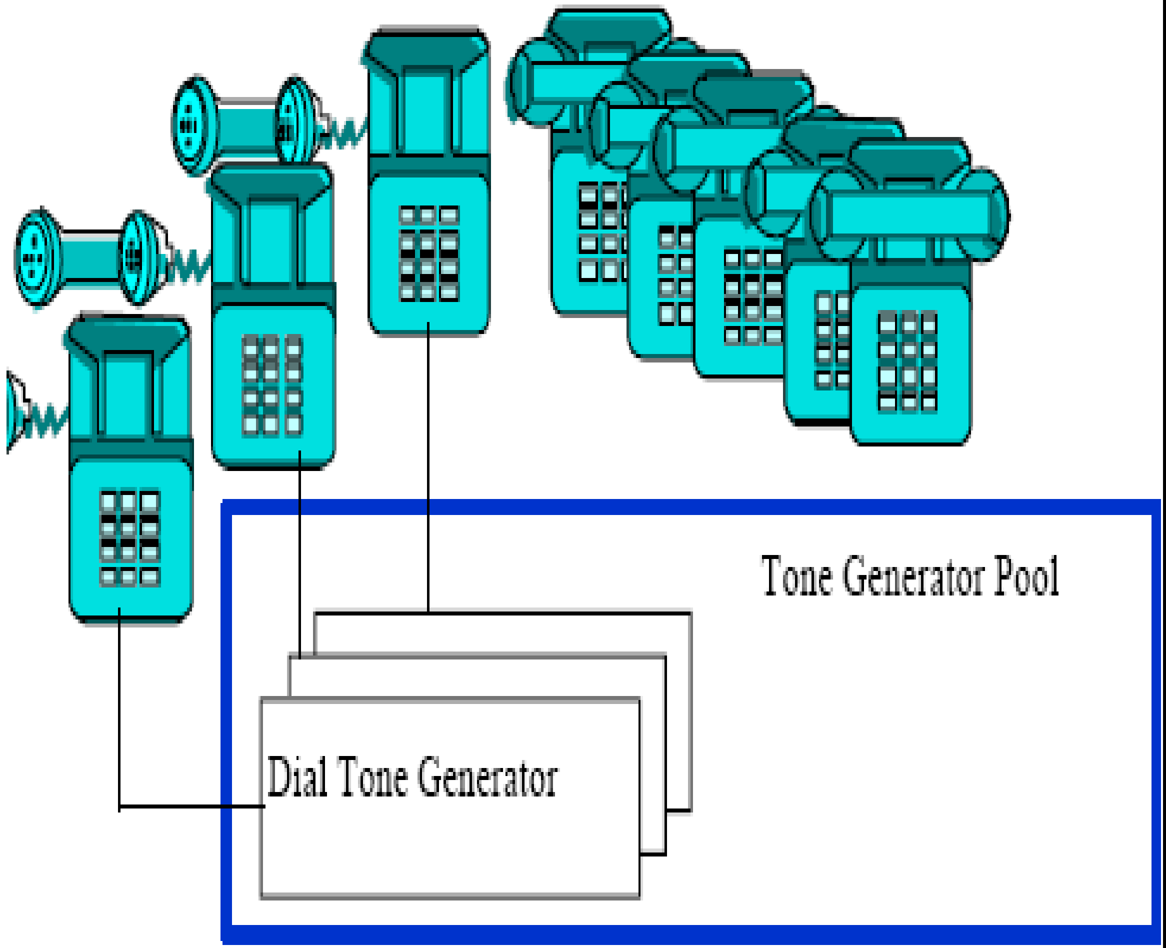


Provide a unified interface to a set of interfaces in a subsystem.

Facade (Software counterpart)



Flyweight (Non software example)



Use sharing to support large numbers of fine-grained objects efficiently

OK

Object Creation

- Get ready to bake some loosely coupled OO designs.
- There is more to making objects than just using the **new** operator.
- You'll learn that instantiation is an activity that shouldn't always be done in public and can often lead to ***coupling problems***.

Object Creation (Cont)

- When you use **new** you are certainly instantiating a concrete class, so that's definitely an implementation, not an interface.
- GoF: Program to an interface not an implementation.
- Tying your code to a concrete class can make it more fragile and less flexible.
- Duck duck = new MallardDuck();

Object Creation (Cont)

- When you have a whole set of related concrete classes, often you’re forced to write code like this:

```
Duck duck;
if (picnic) {
    duck = new MallardDuck();
} else if (hunting) {
    duck = new DecoyDuck();
} else if (inBathTub) {
    duck = new RubberDuck();
}
```

- When you see code like this, you know that when it comes time for changes or extensions,
- you’ll have to reopen this code and examine what needs to be added (or deleted).
- Violates Open Closed Principle.

What’s wrong with “new”?

- Technically there’s nothing wrong with **new**, after all, it’s a fundamental part of Java. The real culprit is our old friend CHANGE and how change impacts our use of **new**.
- By coding to an interface, you know you can insulate yourself from a lot of changes that might happen to a system down the road.
- If your code is written to an interface, then it will work with any new classes implementing that interface through polymorphism.
- When you have code that makes use of lots of concrete classes, you’re looking for trouble because that code may have to be changed as new concrete classes are added.
- **So, in other words, your code will not be “closed for modification.” To extend it with new concrete types, you’ll have to reopen it.**

Pizza Store

- We need a pizza store that can create pizza.
- The customer will order a specific type of pizza:
  - Cheese
  - Veggie
  - Greek
  - Pepperoni
  - etc.
- Each order request is for one type of pizza only.

Pizza Store

- During the ordering of a Pizza, we need to perform certain actions on it:
  - Prepare
  - Bake
  - Cut
  - Box

- We know that all Pizzas *must* perform these behaviors. In addition, we know that these behaviors will not change during runtime. ***(i.e. the Baking time for a Cheese Pizza will never change!)***

- ***Question:*** Should these behaviors (prepare, bake, etc) be represented using Inheritance or Composition?

PizzaStore (Cont)

```
public Pizza orderPizza(String type) {

    Pizza pizza = new Pizza();

    pizza.prepare();
    pizza.bake();
    pizza.cut();
    pizza.box();

    return pizza;
}
```

This method is responsible for creating the pizza.

It calls methods to prepare, bake, etc.

Pizza is returned to caller.

- Creating an instance of Pizza() doesn't make sense here because we know there are different types of Pizza.

PizzaStore (Cont)

```java
public Pizza orderPizza(String type) {

    if (type.equals("cheese")) {
          pizza = new CheesePizza();
    } else if (type.equals("pepperoni")) {
          pizza = new PepperoniPizza();

    pizza.prepare();
     pizza.bake();
     pizza.cut();
     pizza.box();
     return pizza;
}
```

A parameter indicating type

Code that varies

Code that stays the same

PizzaStore (Cont)

- Pressure is on for change…
- Now we get some new types of Pizza (Clam, Veggie)
- Every time there is a change, we would need to break into this code and update the If/Else statement. (and possibly introduce bugs in our existing code).

Solution: Factory Pattern

Move the creation of Pizzas into a separate class!

```
public class SimplePizzaFactory {

   public Pizza createPizza(String type) {
        Pizza pizza = null;

        if (type.equals("cheese")) {
               pizza = new CheesePizza();
        } else if (type.equals("pepperoni")) {
               pizza = new PepperoniPizza();
        } else if (type.equals("clam")) {
               pizza = new ClamPizza();
        } else if (type.equals("veggie")) {
               pizza = new VeggiePizza();
        }
        return pizza;
   }
}
```

SimplePizzaFactory

- Advantage: We have one place to go to add a new pizza.
- Disadvantage: Whenever there is a change, we need to break into this code and add a new line. (but at least it is in one place!!)

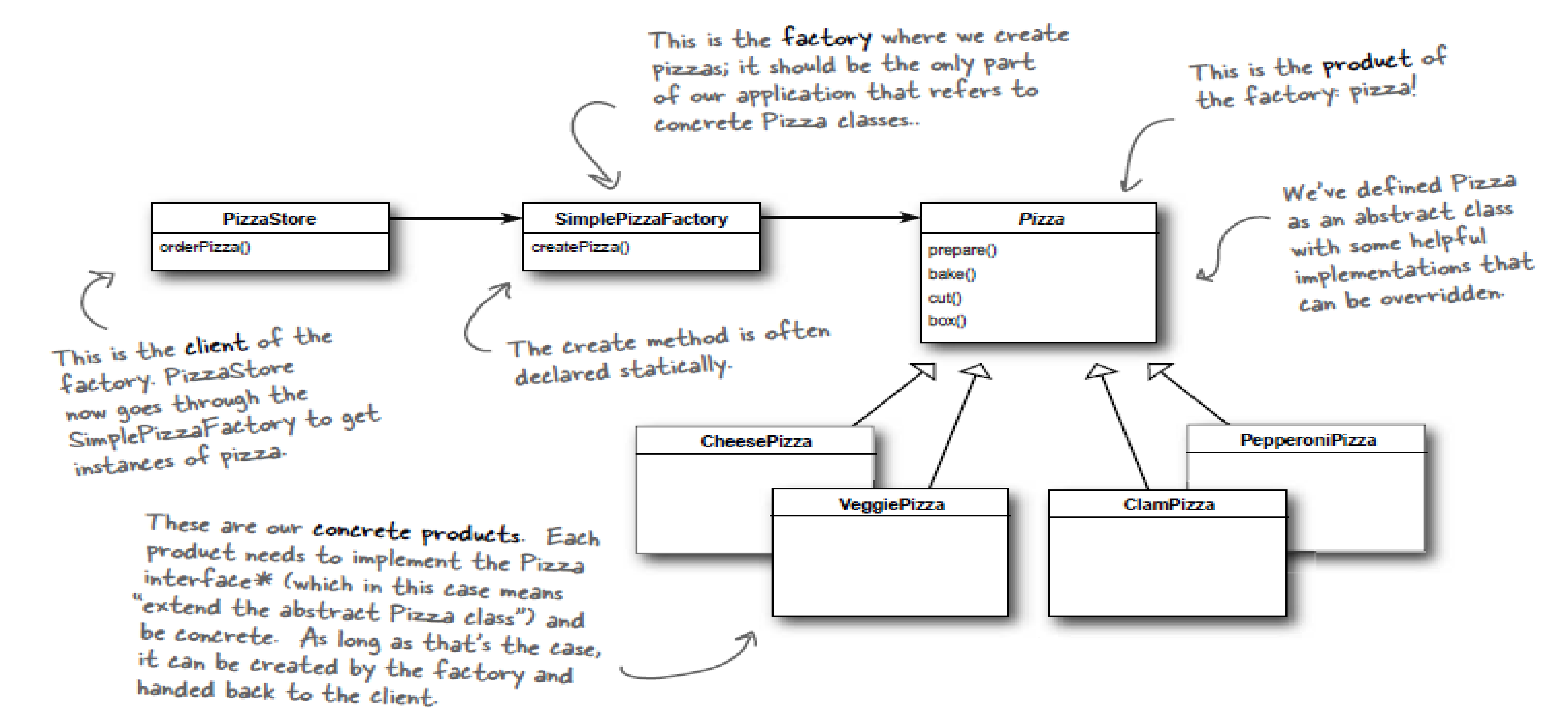
Rework of PizzaStore

Store is *composed* of a factory.

Creation of pizza is delegated to factory.

```
public class PizzaStore {
    SimplePizzaFactory factory;
    public PizzaStore(SimplePizzaFactory factory) {
        this.factory = factory;}
    public Pizza orderPizza(String type) {
        Pizza pizza;
        pizza = factory.createPizza(type);
        pizza.prepare();
        pizza.bake();
        pizza.cut();
        pizza.box();
        return pizza;
    }
    public static void main(String[] args) {
        SimplePizzaFactory factory = new SimplePizzaFactory();
        PizzaStore store = new PizzaStore(factory);
        Pizza pizza = store.orderPizza("cheese");
        pizza = store.orderPizza("veggie");
        }
}
```

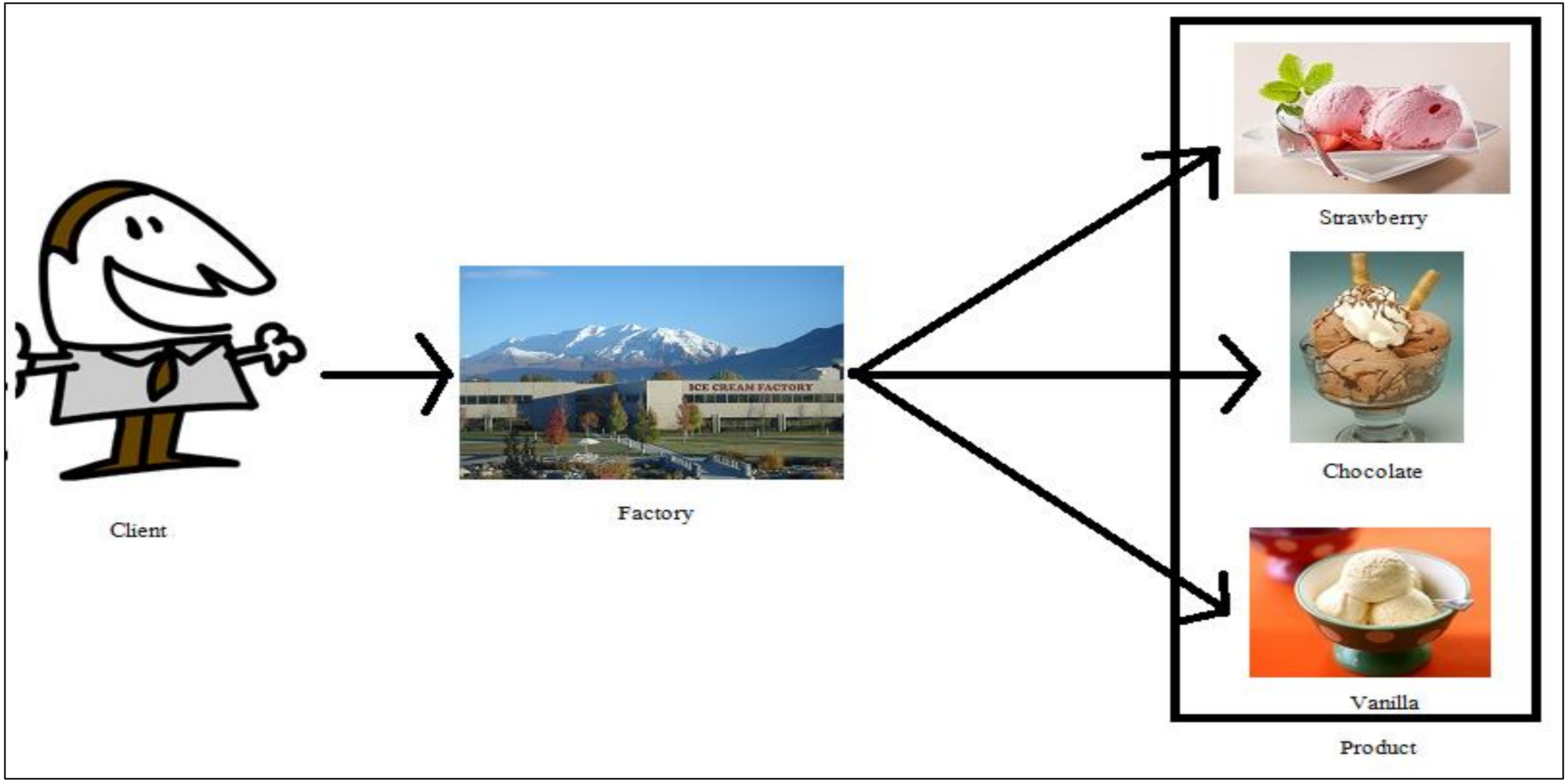
SimplePizzaFactory



OK

Factory Pattern

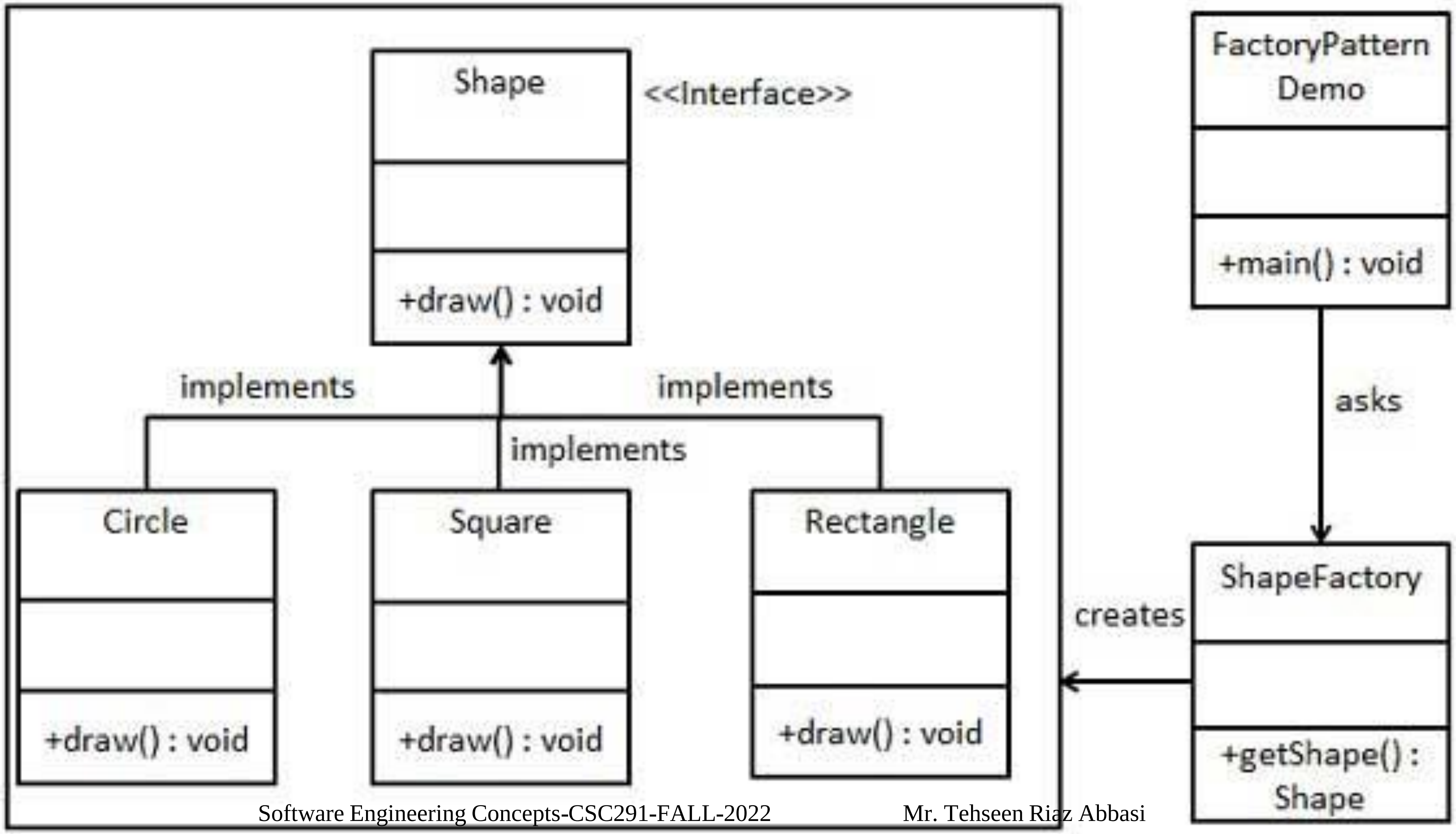
Factory Pattern



Intent

- Creates objects without exposing the instantiation logic to the client.

Factory Pattern - Example



Participants

- Create a Shape interface and concrete classes that implement the Shape interface.
- A factory class ShapeFactory is defined as a next step.
- A client class FactoryPatternDemo will use ShapeFactory to get a Shape object. It will pass information (CIRCLE / RECTANGLE /SQUARE) to ShapeFactory to get the type of object it needs.

Interface (Step 1)

```
public interface Shape {
        void draw();
}
```

Concrete Classes (Step 2)

```
public class Rectangle implements Shape {
        @Override
        public void draw() {
                System.out.println("Inside Rectangle::draw()method.");
        }
}
public class Square implements Shape {
        @Override
        public void draw() {
                System.out.println("Inside Square::draw()     method.");
        }
}
public class Circle implements Shape {
        @Override
        public void draw() {
                System.out.println("Inside Circle::draw() method.");
        }
}
```

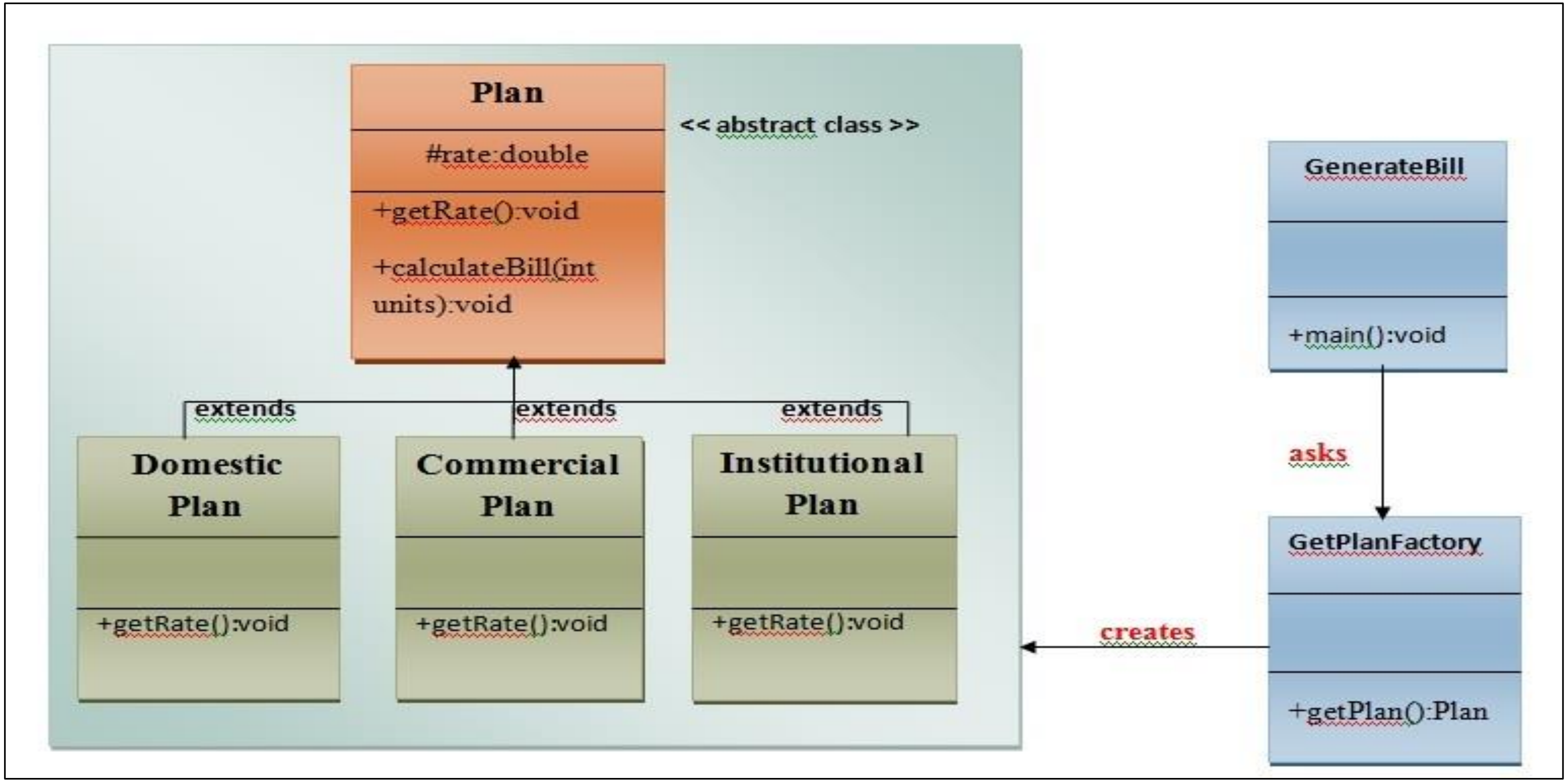
Factory Class (Step 3)

```
public class ShapeFactory {
Shape getShape(String shapeType){
        if(shapeType == null){
                return null;
        }
        if(shapeType.equalsIgnoreCase("CIRCLE")){                return new Circle();
        } else if(shapeType.equalsIgnoreCase("RECTANGLE")){                return new
Rectangle();
        } else if(shapeType.equalsIgnoreCase("SQUARE")){                return new Square();
        }
        return null;
        }
}
```

Client Class (Step 4)

```
public class FactoryPatternDemo {
public static void main(String[] args) {
            ShapeFactory shapeFactory = new ShapeFactory();
//get an object of Circle and call its draw method.
            Shape shape1 = shapeFactory.getShape("CIRCLE");
//call draw method of Circle
            shape1.draw();
//get an object of Rectangle and call its draw method.
            Shape shape2 = shapeFactory.getShape("RECTANGLE");
//call draw method of Rectangle
            shape2.draw();
//get an object of Square and call its draw method.
            Shape shape3 = shapeFactory.getShape("SQUARE");
//call draw method of circle
            shape3.draw();
}
}
```

Example



Plan
<< abstract class >>
#rate:double
+getRate():void
+calculateBill(int units):void
extends
extends
extends
Domestic Plan
+getRate():void
Commercial Plan
+getRate():void
Institutional Plan
+getRate():void
GenerateBill
+main():void
asks
GetPlanFactory
+getPlan():Plan
creates

Participants

- Create a Plan abstract class and concrete classes that extends the Plan abstract class.
- A factory class GetPlanFactory is defined as a next step.
- GenerateBill class will use GetPlanFactory to get a Plan object. It will pass information (DOMESTICPLAN / COMMERCIALPLAN / INSTITUTIONALPLAN) to GetPalnFactory to get the type of object it needs.

Abstract Class (Step 1)

```
import java.io.*;
abstract class Plan{
        protected double rate;
        abstract void getRate();

        public void calculateBill(int units){
            System.out.println(units*rate);
         }
}
```

Concrete Classes (Step 2)

```
class  DomesticPlan extends Plan{
       //@override
        public void getRate(){
            rate=3.50;
       }
   }
class  CommercialPlan extends Plan{
   //@override
    public void getRate(){
        rate=7.50;
   }
class  InstitutionalPlan extends Plan{
   //@override
    public void getRate(){
        rate=5.50;
   }
```

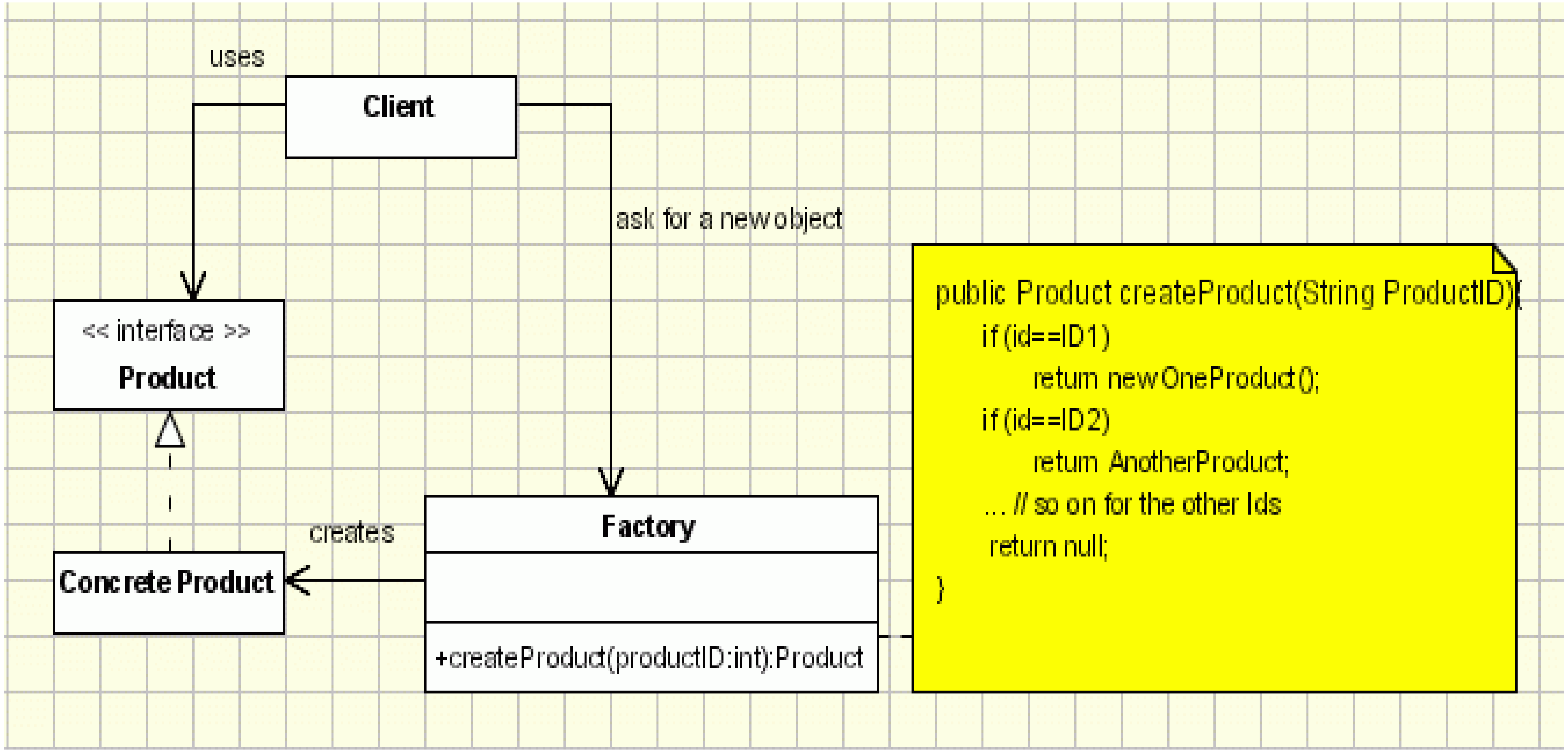
Factory Class (Step 3)

```
class GetPlanFactory{
      //use getPlan method to get object of type Plan
     public Plan getPlan(String planType){
         if(planType == null){
          return null;
         }
       if(planType.equalsIgnoreCase("DOMESTICPLAN")) {
              return new DomesticPlan();
            }
        else if(planType.equalsIgnoreCase("COMMERCIALPLAN")){
             return new CommercialPlan();
         }
       else if(planType.equalsIgnoreCase("INSTITUTIONALPLAN")) {
             return new InstitutionalPlan();
       }
    return null;
  }
}
```

Client Class (Step 4)

```
import java.io.*;
// Client class that uses factory class.
class GenerateBill{
    public static void main(String args[])throws IOException{
            GetPlanFactory planFactory = new GetPlanFactory();
            System.out.print("Enter the name of plan");
            BufferedReader br =
            new BufferedReader(new InputStreamReader(System.in));
            String planName=br.readLine();
            System.out.print("Enter the number of units for bill");
            int units=Integer.parseInt(br.readLine());
            Plan p = planFactory.getPlan(planName);
            System.out.print("Bill amount for "+planName+" of  "+units+"
            units is: ");
            p.getRate();
            p.calculateBill(units);
        }
    }
```

Example


uses
Client
ask for a new object
<< interface >>
Product
Concrete Product
creates
Factory
+createProduct(productID:int):Product
public Product createProduct(String ProductID){
if (id==ID1)
return new OneProduct();
if (id==ID2)
return AnotherProduct;
... // so on for the other Ids
return null;
}

Advantages

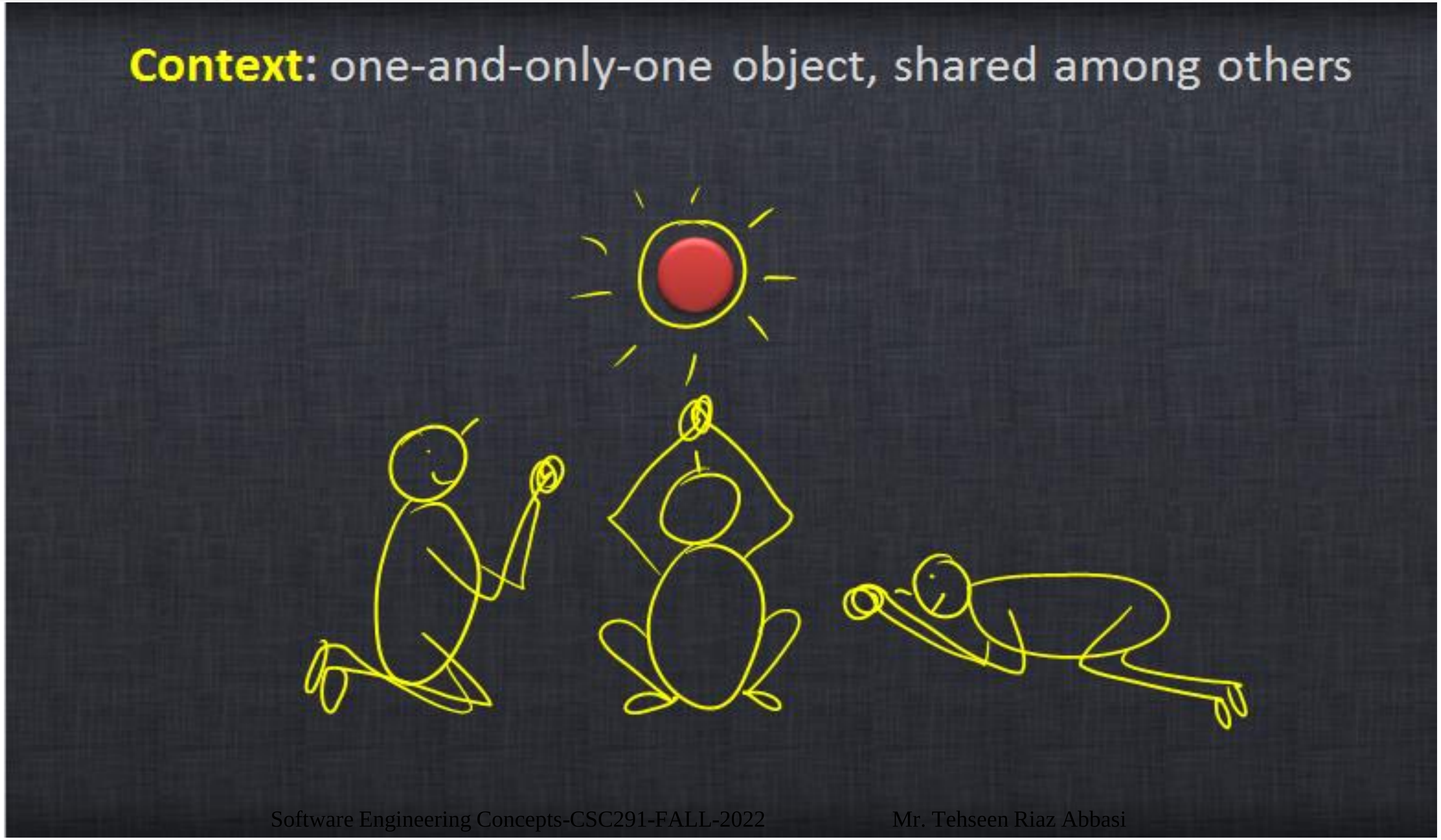
- Factory classes are often implemented because they allow the project to follow the SOLID principles more closely.
- Factories allow for a lot more long term flexibility. It allows for a more decoupled - and therefore more testable - design.
- It gives you a lot more flexibility when it comes time to change the application

Singleton Pattern

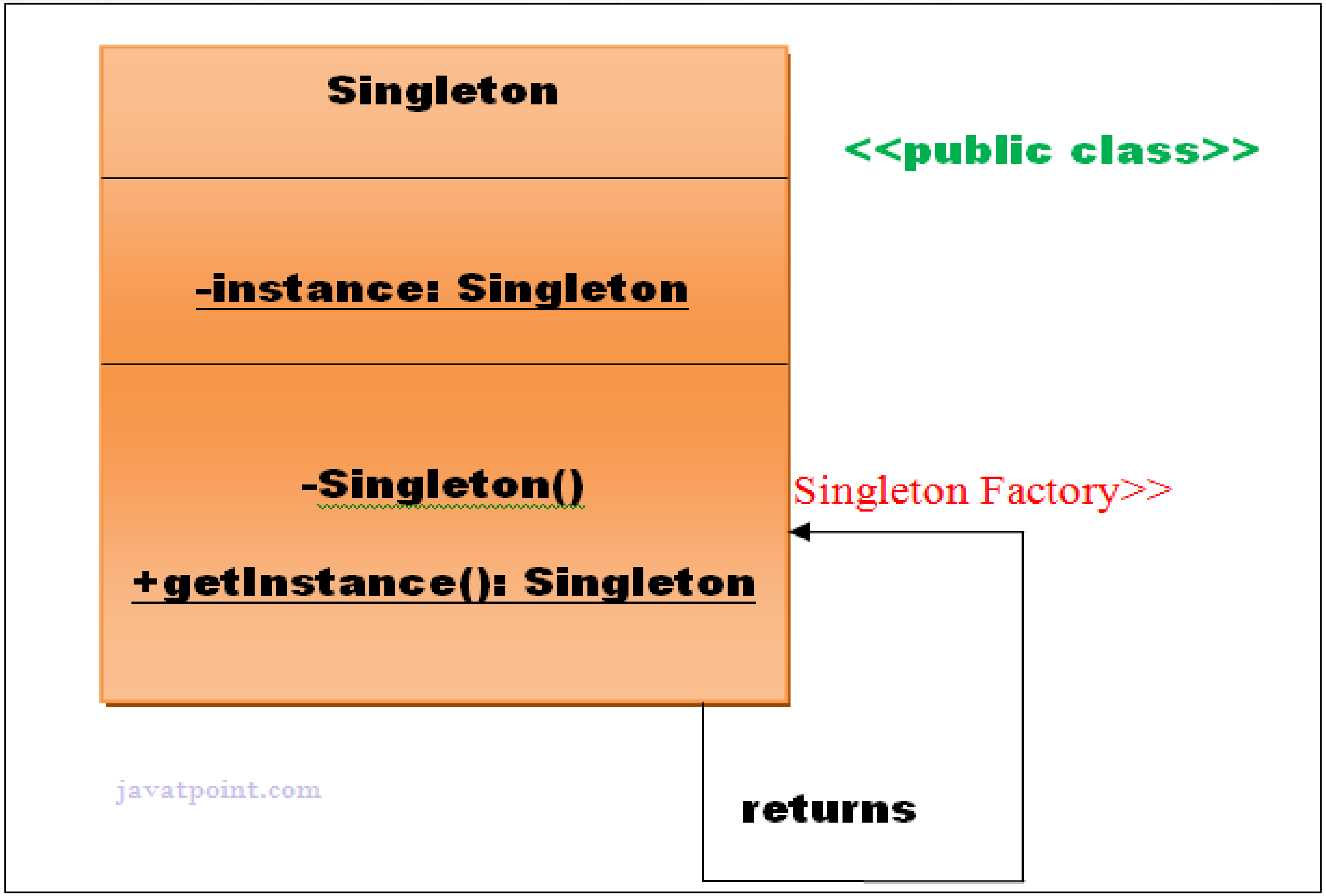
Singleton Pattern

- Singleton Pattern says that just **"define a class that has only one instance and provides a global point of access to it".**
- In other words, a class must ensure that only single instance should be created and single object can be used by all other classes.
- There are two forms of singleton design pattern
  - **Early Instantiation:** creation of instance at load time.
  - **Lazy Instantiation:** creation of instance when required.

Singleton Pattern



UML of Singleton Pattern



How to create Singleton design pattern?

- **Static member:**
  - It gets memory only once because of static, it contains the instance of the Singleton class.
- **Private constructor:**
  - It will prevent to instantiate the Singleton class from outside the class.
- **Static factory method:**
  - This provides the global point of access to the Singleton object and returns the instance to the caller.

Early Instantiation of Singleton Pattern

- In such case, we create the instance of the class at the time of declaring the static data member, so instance of the class is created at the time of classloading.

```
class Singleton{
    private static  Singleton  uniqueinstance=new  Singleton();
   //Early, instance will be created at load time
    private  Singleton(){}

    public static  Singleton  get Singleton(){
     return uniqueinstance;
    }
   }
```

Lazy Instantiation of Singleton Pattern

- **In such case, we create the instance of the class in synchronized method or synchronized block, so instance of the class is created when required.**

Check for an instance and if there isn't one, enter a synchronized block.

Once in the block, check again and if still null, create an instance.

```
class  Singleton{
        private static  Singleton  uniqueinstance;
        private  Singleton(){}

        public static Singleton  get Singleton(){
            if (uniqueinstance == null){

                synchronized (Singleton.class){

                        if (uniqueinstance == null){
                                System.out.println("First time getInstance was invoked!");
                                 uniqueinstance = new Singleton();
                        }
                }
                return uniqueinstance;
        }}}
        public void doSomething()
        {
        ... } }
Public class client{
Public static void main (String arg[]{
                Singleton.getSingleton().doSomething();
}}
```

Singleton Pattern

- **Advantage of Singleton design pattern**
  - Saves memory because object is not created at each request.
  - Only single instance is reused again and again.
- **Usage of Singleton design pattern**
  - It is used in logging, caching, thread pools, configuration settings, device drivers etc.
  - If we were to instantiate more than one we'd run into all sorts of problems like incorrect program behavior, overuse of resources, or inconsistent results.

Applicability & Examples

- Logger Classes
  - **The Singleton pattern** is used in the design of logger classes.
  - These classes are usually implemented as singletons, and provide a global logging access point in all the application components without being necessary to create an object each time a logging operation is performed.
- Configuration Classes
  - **The Singleton pattern is used to design the classes** which provide the configuration settings for an application.
  - **By implementing configuration** classes as Singleton not only that we provide a global access point, but we also keep the instance we use as a cache object.
  - **When the class is instantiated**(or when a value is read) the singleton will keep the values in its internal structure.
  - **If the values are read from the database** or from files this avoids the reloading the values each time the configuration parameters are used.

Applicability & Examples

- Accessing resources in shared mode
  - It can be used in the design of an application that needs to work with the serial port.
  - Let's say that there are many classes in the application, working in a multi-threading environment, which needs to operate actions on the serial port.
  - In this case a singleton with synchronized methods could be used to manage all the operations on the serial port.
- Factories implemented as Singletons
  - Let's assume that we design an application with a factory to generate new objects(Acount, Customer, Site, Address objects) with their ids, in a multithreading environment.
  - If the factory is instantiated twice in 2 different threads then is possible to have 2 overlapping ids for 2 different objects.
  - If we implement the Factory as a singleton we avoid this problem. Combining Abstract Factory or Factory Method and Singleton design patterns is a common practice.

Specific problems and Implementation

- Thread-safe implementation for multi-threading use.
  - A singleton implementation should work in any conditions.
  - This is why we need to ensure it works when multiple threads uses it.
- Early instantiation using implementation with static field
  - Singleton object is instantiated when the class is loaded and not when it is first used, due to the fact that the instance member is declared static.
  - We don't need to synchronize any portion of the code in this case. The class is loaded once this guarantee the uniqueness of the object.

Specific problems and Implementation

- Serialization
  - If the Singleton class implements the java.io.Serializable interface, when a singleton is serialized and then deserialized more than once, there will be multiple instances of Singleton created.
  - In order to avoid this the readResolve method should be implemented.
  - Serialization -: Turn object into a stream of bytes
  - Deserialization - Turn a stream of bytes back into a copy of the original object.

```
public class Singleton implements Serializable {
    protected Object readResolve() {
        return getInstance();
    }
}
```

Questions?

- Does Singleton violate SRP?
  - Yes. Class has two responsibilities
    - Manage its own instance
    - Main role of the class
  - But advantage is obvious, and it is widely used.
- Can we subclass Singleton?
  - You can't extend a class with a private constructor. So, the first thing you'll have to do is change your constructor so that it's public or protected.
  - But then, it's not *really* a Singleton anymore, because other classes can instantiate it.

Questions

- Why global variables are worse than a Singleton?
  - Intent of the pattern: to ensure only one instance of a class exists and to provide global access.
  - A global variable can provide the global access, but can not ensure only one instance.


Thank you