Design Patterns

Again ☺

Elements of a Design Pattern

- A pattern has four essential elements (GoF)
  - Name
  - Describes the pattern
  - Adds to common terminology for facilitating communication (i.e. not just sentence enhancers)
  - Problem
  - Describes when to apply the pattern
  - Answers - What is the pattern trying to solve?

Elements of a Design Pattern (cont.)

- Solution
- Describes elements, relationships, responsibilities, and collaborations which make up the design
- Consequences
- Results of applying the pattern
- Benefits and Costs
- Subjective depending on concrete scenarios

Design Patterns Classification

- Creational
  - Factory Pattern
  - Abstract Factory Pattern
  - Singleton Pattern
  - Prototype Pattern
  - Builder Pattern.
- Structural
  - Adapter Pattern
  - Bridge Pattern
  - Composite Pattern
  - Decorator Pattern
  - Facade Pattern
  - Flyweight Pattern
  - Proxy Pattern
- Behavioral
  - Chain Of Responsibility Pattern
  - Command Pattern
  - Interpreter Pattern
  - Iterator Pattern
  - Mediator Pattern
  - Memento Pattern
  - Observer Pattern
  - State Pattern
  - Strategy Pattern
  - Template Pattern
  - Visitor Pattern

Pros/Cons of Design Patterns

- Pros
  - Add **consistency** to designs by solving similar problems the same way, independent of language
  - Add **clarity** to design and design communication by enabling a common vocabulary
  - Improve **time** to solution by providing templates which serve as foundations for good design
  - Improve **reuse** through composition

Pros/Cons of Design Patterns

- Cons
  - Some patterns come with negative consequences (i.e. object proliferation, performance hits, additional layers)
  - Consequences are subjective depending on concrete scenarios
  - Patterns are subject to different interpretations, misinterpretations, and philosophies
  - Patterns can be overused and abused → Anti-Patterns

Popular Design Patterns

- We will look at following patterns;
  - Factory
  - Abstract Factory
  - Singleton
  - Decorator
  - Façade
  - Adapter
  - And more….

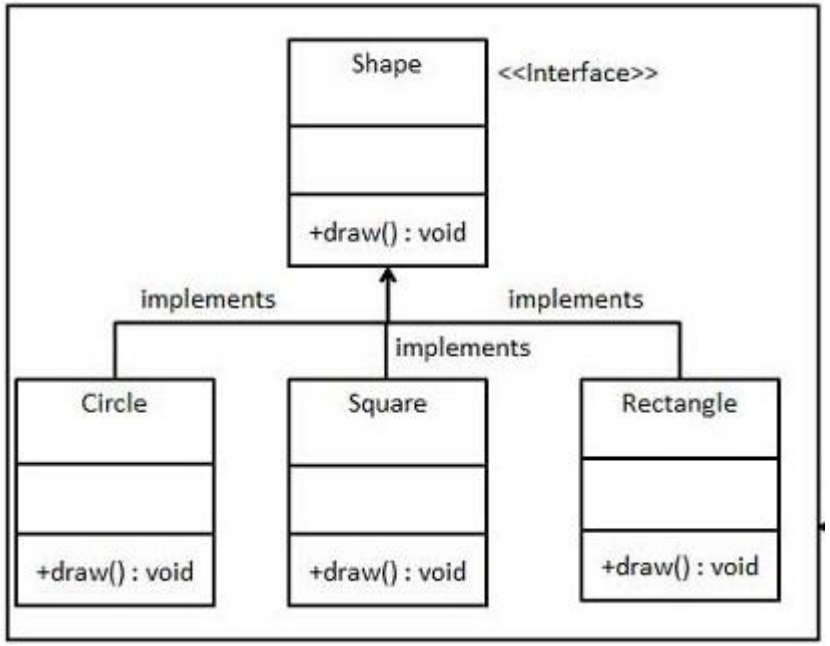
Factory Pattern

DESIGN PATTERNS

Factory Pattern

- Factory pattern is one of the most used design patterns in Java.
- we create object without exposing the creation logic to the client
- We refer to newly created object using a common interface.

Example

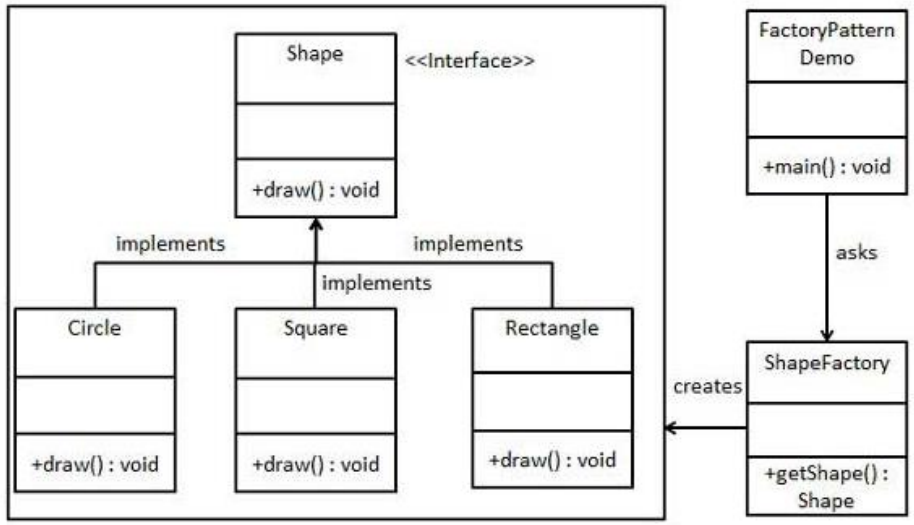



```java
public interface Shape {
   void draw();
}
```

```java
public class Rectangle implements Shape {

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

```java
public class Square implements Shape {

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

```java
public class Circle implements Shape {

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




```java
public class ShapeFactory {

   //use getShape method to get object of type shape
   public 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;
   }
}
```

```java
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 square
      shape3.draw();
   }
}
```

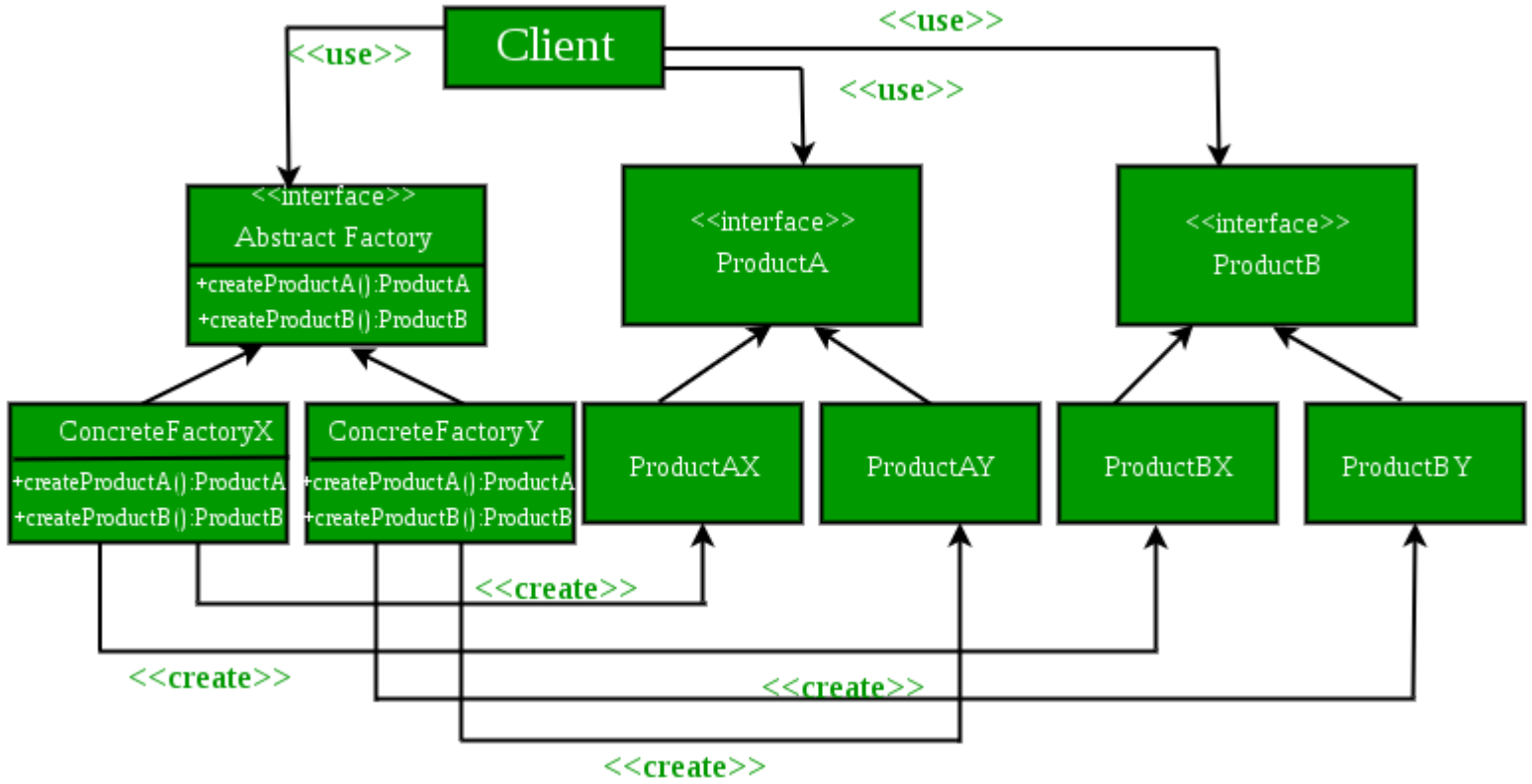
```
Inside Circle::draw() method.
Inside Rectangle::draw() method.
Inside Square::draw() method.
```

Abstract Factory Pattern

DESIGN PATTERNS

Abstract Factory Pattern

- Creational pattern
- almost similar to Factory Pattern
  - considered as another layer of abstraction over factory pattern.
- works around a super-factory which creates other factories.
- implementation provides us with a framework that allows us to create objects that follow a general pattern.
- at runtime, the abstract factory is coupled with any desired concrete factory which can create objects of the desired type.

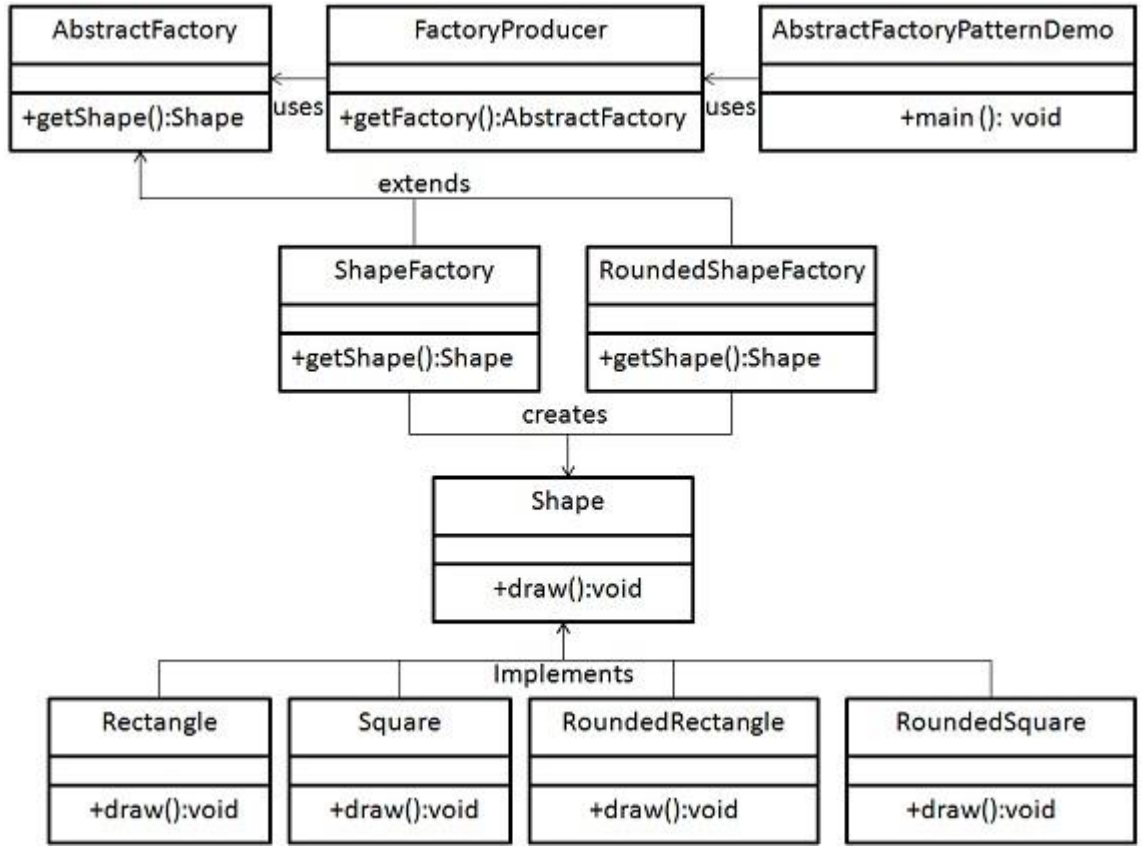


•**AbstractFactory**: Declares an interface for operations that create abstract product objects.

•**ConcreteFactory**: Implements the operations declared in the AbstractFactory to create concrete product objects.

•**Product**: Defines a product object to be created by the corresponding concrete factory and implements the AbstractProduct interface.

Abstract Factory Pattern



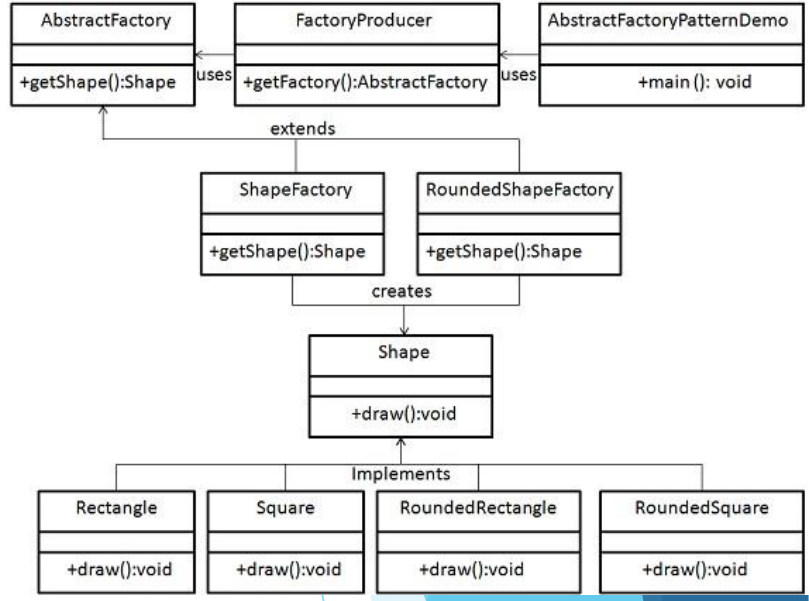
```java
public interface Shape {
   void draw();
}
```

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

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

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

```java
public abstract class AbstractFactory {
   abstract Shape getShape(String shapeType) ;
}
```

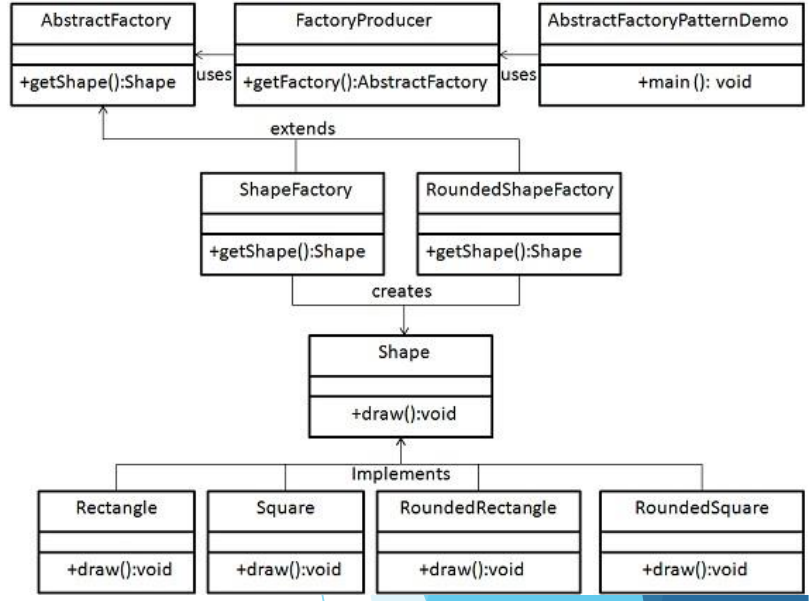


```java
public abstract class AbstractFactory {
   abstract Shape getShape(String shapeType) ;
}
```

```java
public class ShapeFactory extends AbstractFactory {
   @Override
   public Shape getShape(String shapeType){
      if(shapeType.equalsIgnoreCase("RECTANGLE")){
         return new Rectangle();
      }else if(shapeType.equalsIgnoreCase("SQUARE")){
         return new Square();
      }
      return null;
   }
}
```

```java
public class RoundedShapeFactory extends AbstractFactory {
   @Override
   public Shape getShape(String shapeType){
      if(shapeType.equalsIgnoreCase("RECTANGLE")){
         return new RoundedRectangle();
      }else if(shapeType.equalsIgnoreCase("SQUARE")){
         return new RoundedSquare();
      }
      return null;
   }
}
```

```java
public class FactoryProducer {
   public static AbstractFactory getFactory(boolean rounded){
      if(rounded){
         return new RoundedShapeFactory();
      }else{
         return new ShapeFactory();
      }
   }
}
```




```java
public class AbstractFactoryPatternDemo {
   public static void main(String[] args) {
      //get shape factory
      AbstractFactory shapeFactory = FactoryProducer.getFactory(false);
      //get an object of Shape Rectangle
      Shape shape1 = shapeFactory.getShape("RECTANGLE");
      //call draw method of Shape Rectangle
      shape1.draw();
      //get an object of Shape Square
      Shape shape2 = shapeFactory.getShape("SQUARE");
      //call draw method of Shape Square
      shape2.draw();
      //get shape factory
      AbstractFactory shapeFactory1 = FactoryProducer.getFactory(true);
      //get an object of Shape Rectangle
      Shape shape3 = shapeFactory1.getShape("RECTANGLE");
      //call draw method of Shape Rectangle
      shape3.draw();
      //get an object of Shape Square
      Shape shape4 = shapeFactory1.getShape("SQUARE");
      //call draw method of Shape Square
      shape4.draw();

   }
}
```

```
Inside Rectangle::draw() method.
Inside Square::draw() method.
Inside RoundedRectangle::draw() method.
Inside RoundedSquare::draw() method.
```

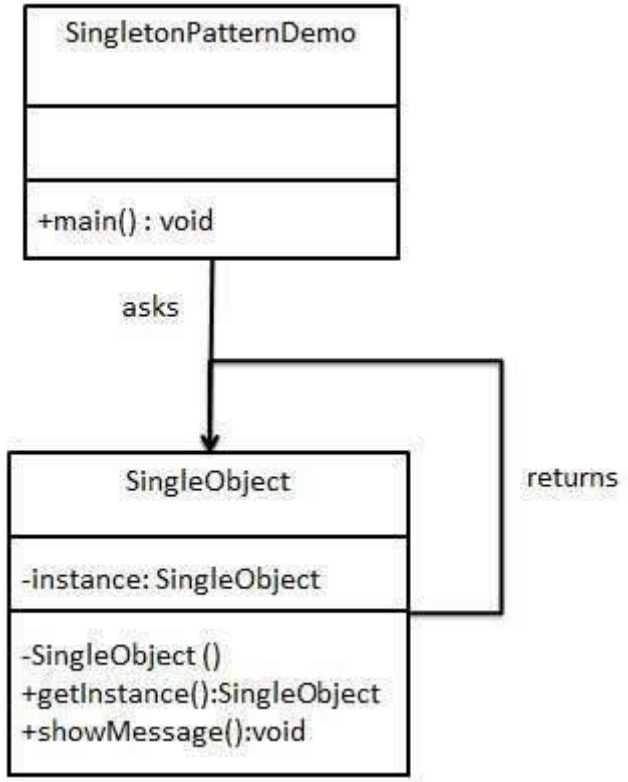
Singleton Pattern

DESIGN PATTERNS

Singleton Pattern

- one of the simplest design patterns in Java
- creational pattern
- involves a single class which is responsible to create an object while making sure that only single object gets created.
- This class provides a way to access its only object which can be accessed directly without need to instantiate the object of the class.

Singleton Pattern



Eager initialization

Static block initialization

Lazy Initialization

Thread Safe Singleton

Bill Pugh Singleton Implementation

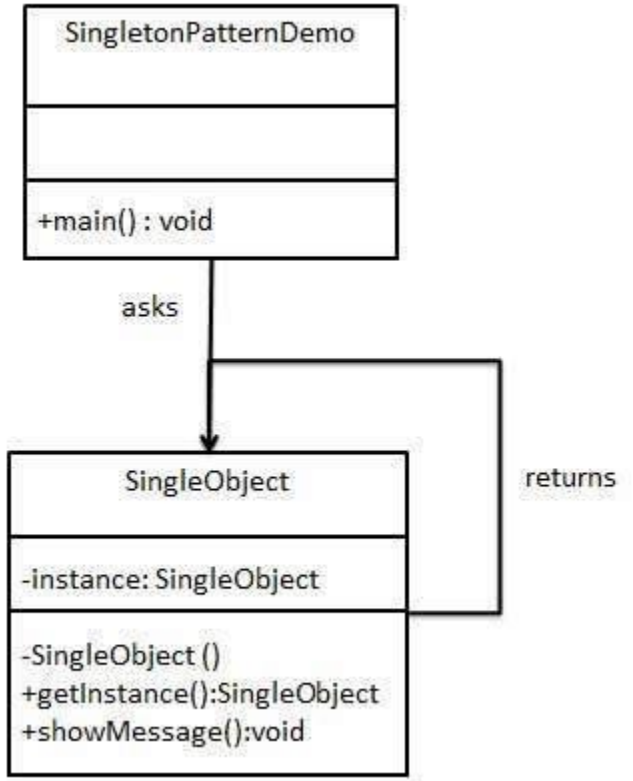
Using Reflection to destroy Singleton Pattern

Enum Singleton

1.Make constructor private.
2.Write a static method that has return type object of this singleton class. (Lazy initialization)

```java
public class SingleObject {

   //create an object of SingleObject
   private static SingleObject instance = new SingleObject();

   //make the constructor private so that this class cannot be
   //instantiated
   private SingleObject(){}

   //Get the only object available
   public static SingleObject getInstance(){
      return instance;
   }

   public void showMessage(){
      System.out.println("Hello World!");
   }
}
```

Eager initialization




```java
public class SingletonPatternDemo {
   public static void main(String[] args) {

      //illegal construct
      //Compile Time Error: The constructor SingleObject() is not visible
      //SingleObject object = new SingleObject();

      //Get the only object available
      SingleObject object = SingleObject.getInstance();

      //show the message
      object.showMessage();
   }
}
```

Lazy Initialization

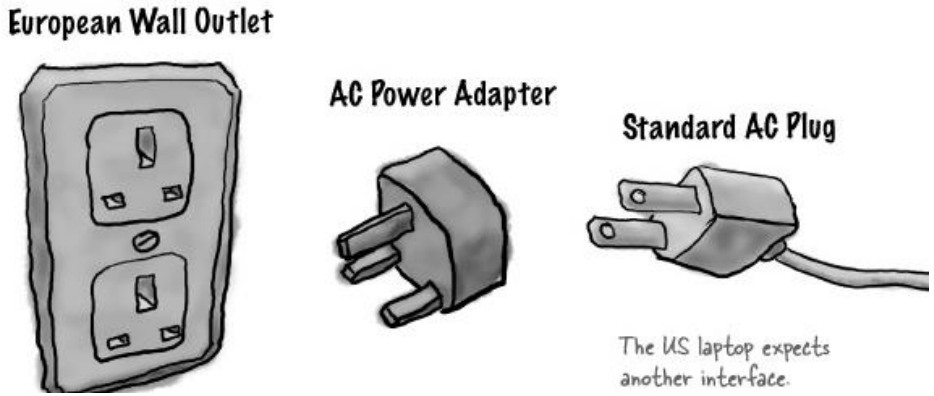
```
public class LazyInitializedSingleton {

    private static LazyInitializedSingleton instance;

    private LazyInitializedSingleton(){}

    public static LazyInitializedSingleton getInstance(){
        if(instance == null){
            instance = new LazyInitializedSingleton();
        }
        return instance;
    }
}
```

Adapter Pattern

DESIGN PATTERNS

Adapter Design Pattern

- Gang of Four state the intent of Adapter is to
  - *Convert the interface of a class into another interface that the clients expect. Adapter lets classes work together that could not otherwise because of incompatible interfaces.*
- Use it when you need a way to *create a new interface for an object that does the right stuff but has the wrong interface* *Alan Shalloway*



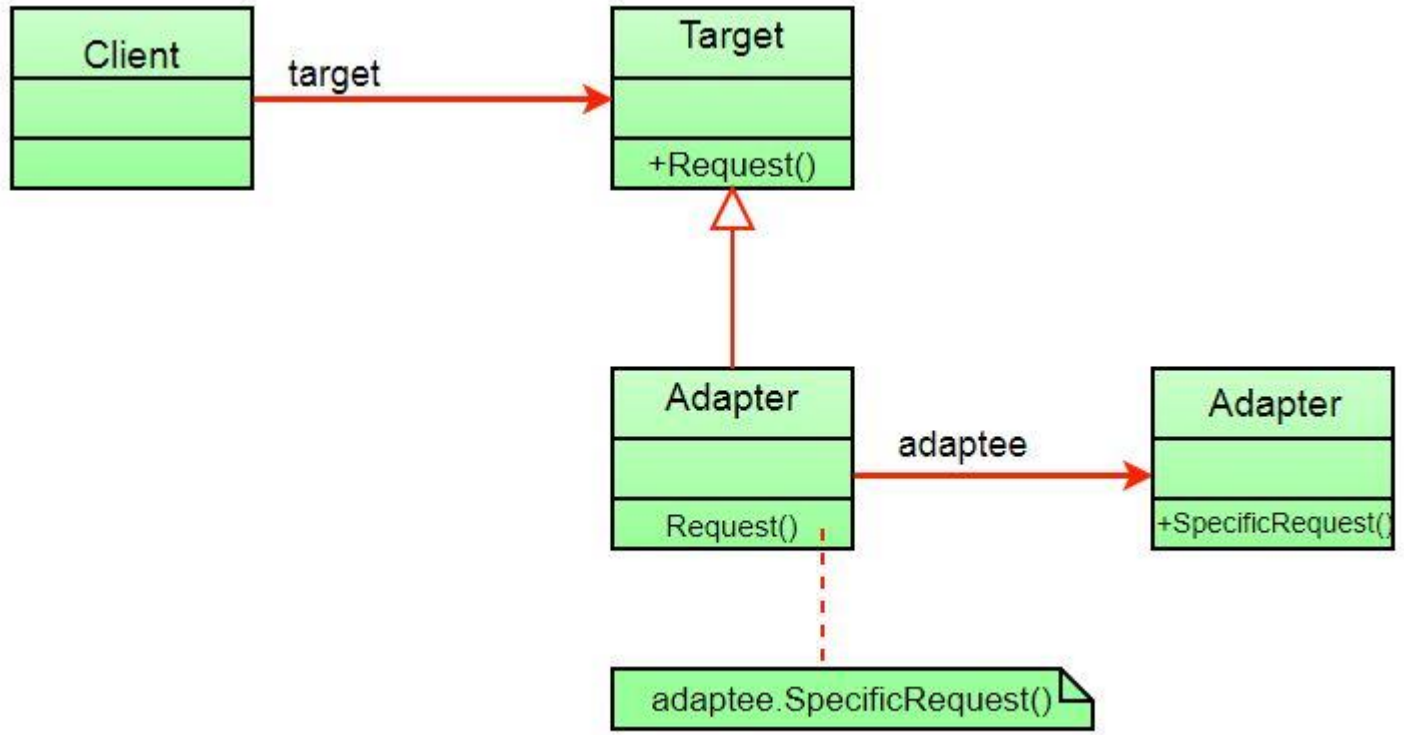
Adapter Design Pattern

- Adapter is a structural design pattern, which allows incompatible objects to cooperate.
- Adapter acts as a wrapper between two objects, It catches calls for one object and transforms them to format and interface recognizable by the second object.
- In other words, how we put a square peg in a round hole.

Definition & Applicability

- Adapters are used to enable objects with different interfaces to communicate with each other.
- The Adapter pattern is used to convert the programming interface of one class into that of another.
- We use adapters whenever we want unrelated classes to work together in a single program.
- Adapters come in two flavors,
  - object adapters and
  - class adapters.
- The concept of an adapter is thus pretty simple; we write a class that has the desired interface and then make it communicate with the class that has a different interface.
- Adapters in Java can be implemented in two ways:
  - by inheritance, and
  - by object composition.

(Object) Adapter Design Pattern



Object Adapters

- Object Adapters rely on one object (the adapting object) containing another (the adapted object)
- **Object Adapter** uses *composition* and can wrap classes or interfaces, or both. It can do this since it contains, as a private, encapsulated member, the class or interface **object instance** it wraps.
- The adapter inherits the target interface that the client expects to see, while it holds an instance of the adaptee.
- When the client calls the request() method on its target object (the adapter), the request is translated into the corresponding specific request on the adaptee.
- Object adapters enable the client and the adaptee to be completely decoupled from each other. Only the adapter knows about both of them.

Object Adapters

```
interface ClientInterface {
    void display();
}
```

```
class MyNewObjectAdapter implements ClientInterface {

    MyExistingServiceClass existingClassObject;

    void display() {
        existingClassObject.show();
    }
}
```

Class Adapters

- **Class Adapter** uses *inheritance* and can only wrap a **class**. It cannot wrap an interface since by definition it must derive from some base class.
- Class Adapters also come about by extending a class or implementing an interface used by the client code

```
class MyExistingServiceClass {
    public void show() {
        System.out.println("Inside Service method show()");
    }
}

interface ClientInterface {
    void display();
}

class MyNewClassAdapter extends MyExistingServiceClass implements ClientInterface {
    void display() {
        show();
    }
}
```

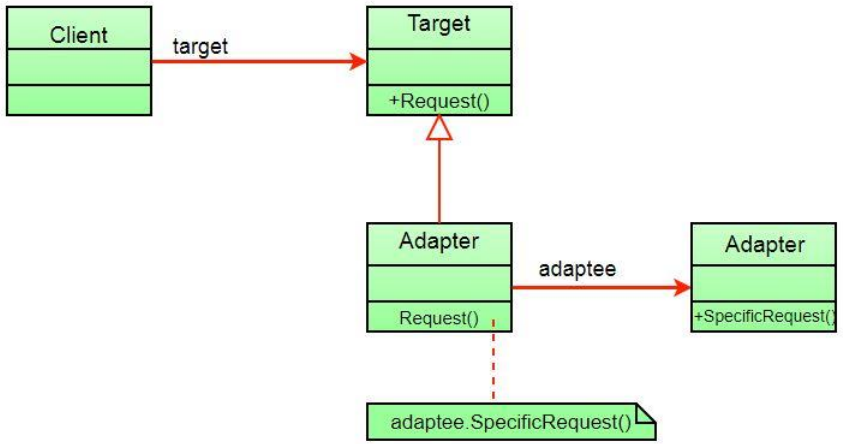
```java
interface Bird
{
    // birds implement Bird interface that allows
    // them to fly and make sounds adaptee interface
    public void fly();
    public void makeSound();
}

class Sparrow implements Bird
{
    // a concrete implementation of bird
    public void fly()
    {
        System.out.println("Flying");
    }
    public void makeSound()
    {
        System.out.println("Chirp Chirp");
    }
}
class Main
{
    public static void main(String args[])
    {
        Sparrow sparrow = new Sparrow();
        ToyDuck toyDuck = new PlasticToyDuck();

        // Wrap a bird in a birdAdapter so that it
        // behaves like toy duck
        ToyDuck birdAdapter = new BirdAdapter(sparrow);

        System.out.println("Sparrow...");
        sparrow.fly();
        sparrow.makeSound();

        System.out.println("ToyDuck...");
        toyDuck.squeak();

        // toy duck behaving like a bird
        System.out.println("BirdAdapter...");
        birdAdapter.squeak();
    }
}
```

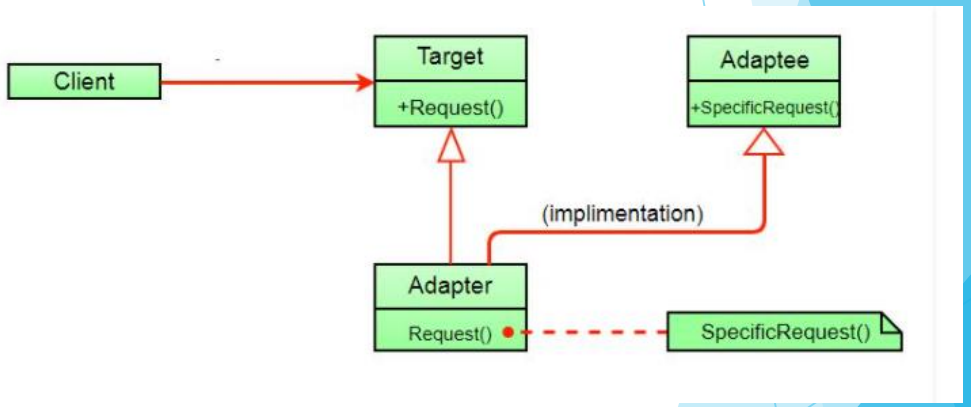
Output:

```
Sparrow...
Flying
Chirp Chirp
ToyDuck...
Squeak
BirdAdapter...
Chirp Chirp
```

```java
interface ToyDuck
{
    // target interface
    // toyducks dont fly they just make
    // squeaking sound
    public void squeak();
}

class PlasticToyDuck implements ToyDuck
{
    public void squeak()
    {
        System.out.println("Squeak");
    }
}

class BirdAdapter implements ToyDuck
{
    // You need to implement the interface your
    // client expects to use.
    Bird bird;
    public BirdAdapter(Bird bird)
    {
        // we need reference to the object we
        // are adapting
        this.bird = bird;
    }

    public void squeak()
    {
        // translate the methods appropriately
        bird.makeSound();
    }
}
```

Object Adapter vs Class Adapter





Summary

- When you need to use an existing class and its interface is not the one you need, use an adapter: allows collaboration between classes with incompatible interfaces.
- An adapter changes an interface into one a client expects.
- Implementing an adapter may require little work or a great deal of work depending on the size and complexity of the target interface.
- There are two forms of adapter patterns: object and class adapters.
- Class adapters require multiple inheritance.

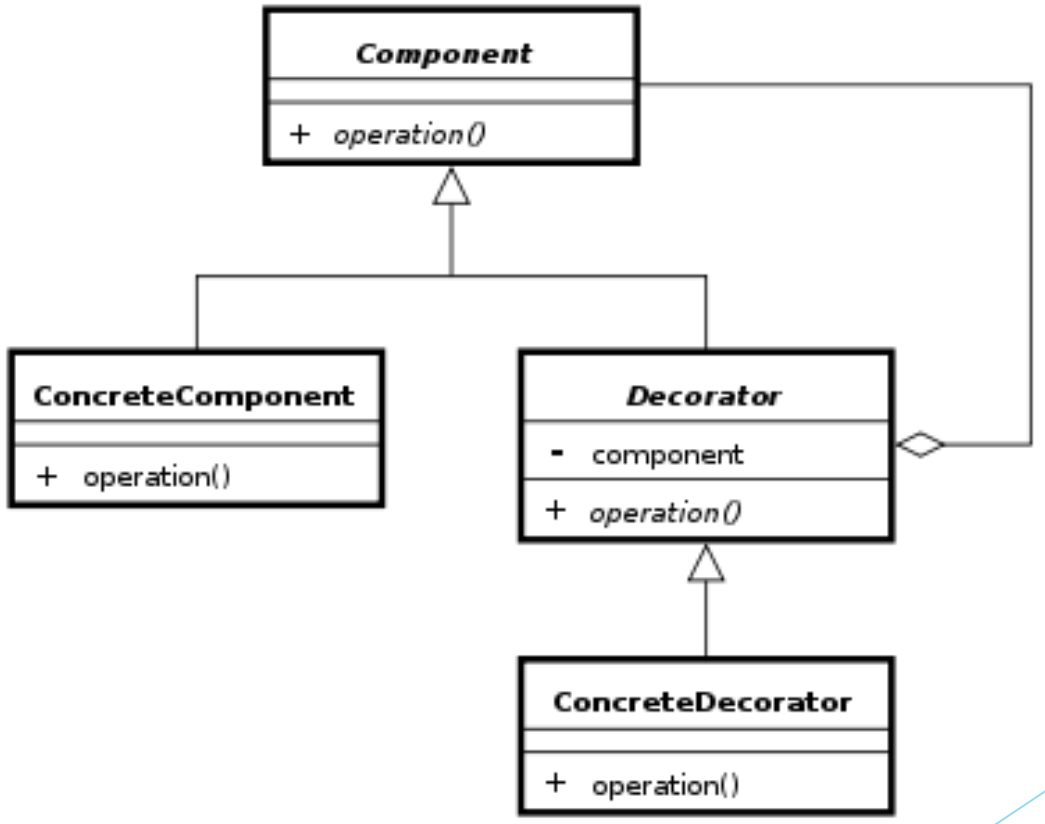
Decorator Pattern

Design Patterns

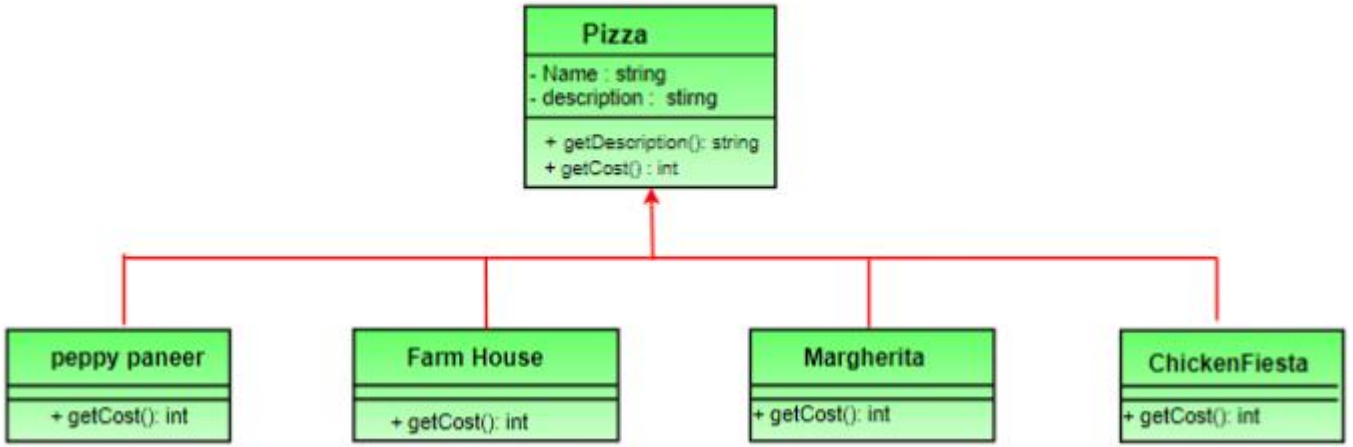
Decorator Pattern

- The Decorator Pattern attaches additional responsibilities to an object dynamically.
- Decorators provide a flexible alternative to subclassing for extending functionality.

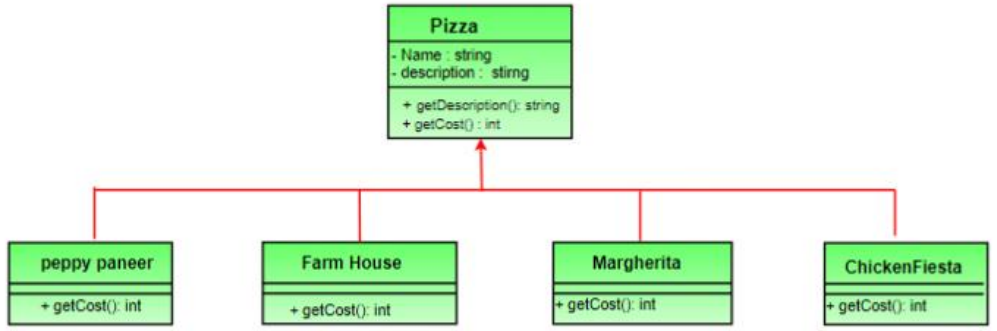
Decorator Pattern



Example

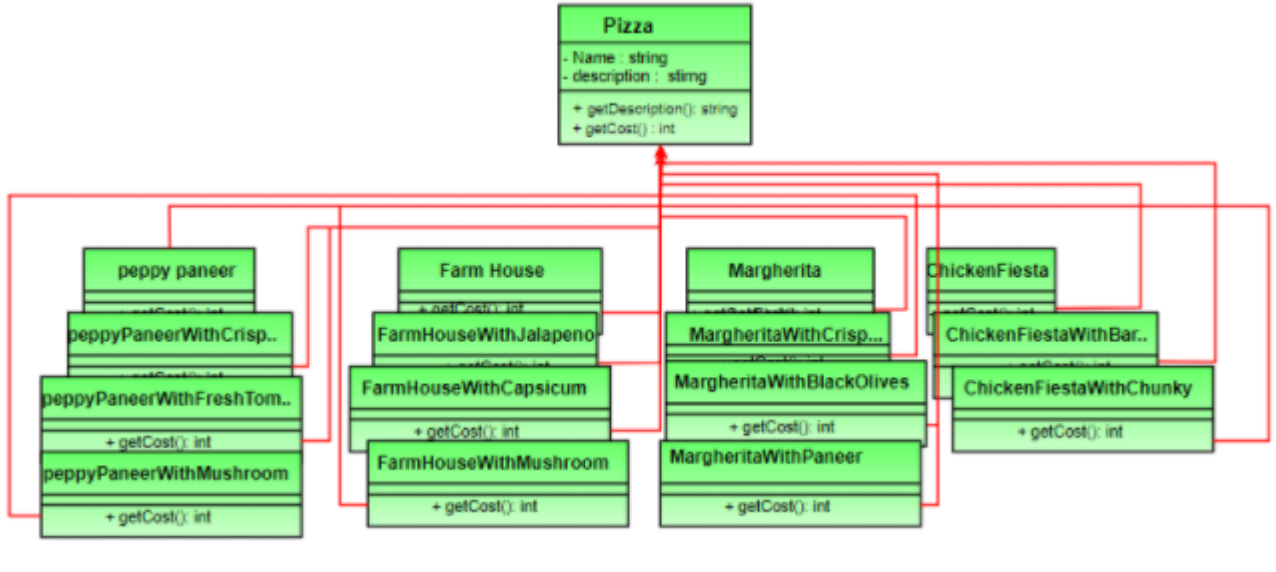

Pizza
- Name : string
- description : stirng
+ getDescription(): string
+ getCost() : int
peppy paneer
+ getCost(): int
Farm House
+ getCost(): int
Margherita
+ getCost(): int
ChickenFiesta
+ getCost(): int

New Requirement

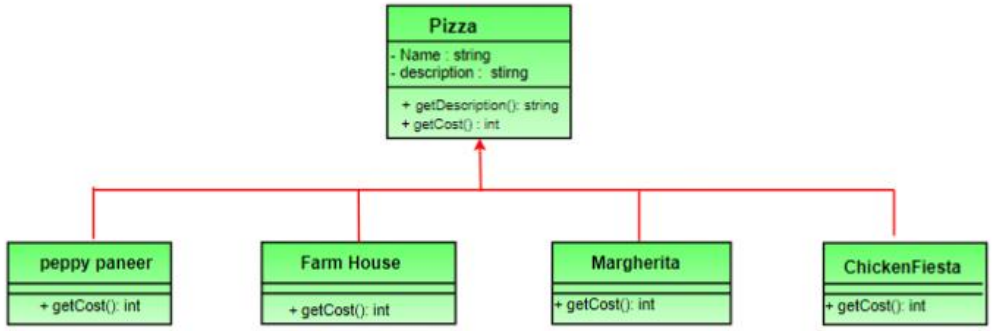


Option 1

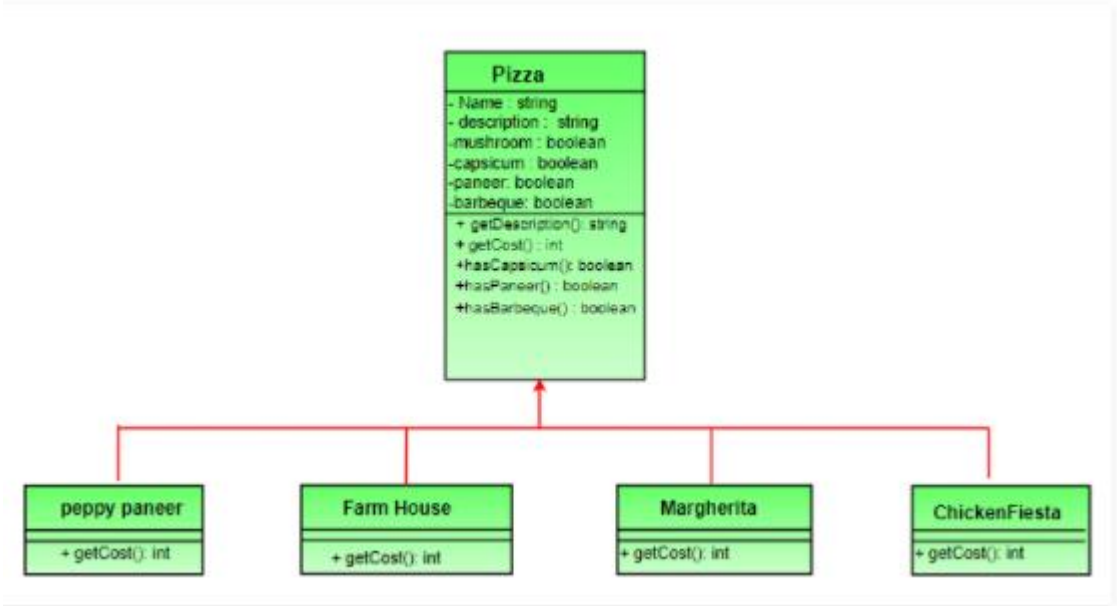
Create a new subclass for every topping with a pizza. The class diagram would look like:



New Requirement



Option 2:
Let's add instance variables to pizza base class to represent whether or not each pizza has a topping.

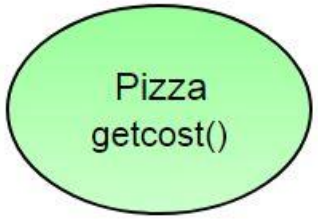


The getCost() of superclass calculates the costs for all the toppings while the one in the subclass adds the cost of that specific pizza.

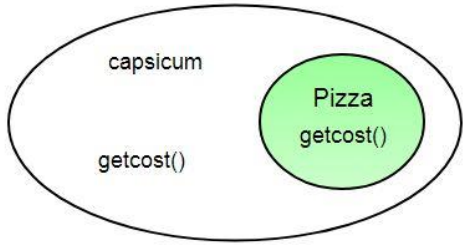
This design looks good at first but lets take a look at the problems associated with it.

- Price changes in toppings will lead to alteration in the existing code.
- New toppings will force us to add new methods and alter getCost() method in superclass.
- For some pizzas, some toppings may not be appropriate yet the subclass inherits them.
- What if customer wants double capsicum or double cheeseburst?

1.Take a pizza object.

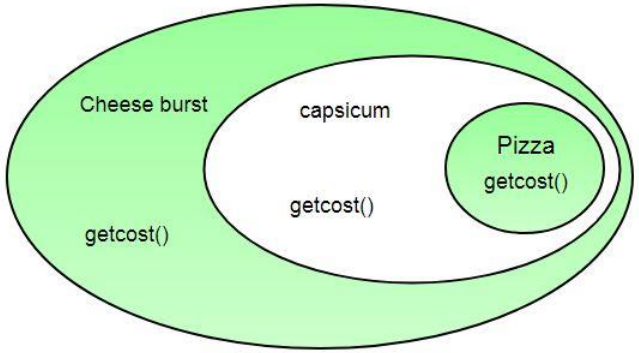


2.“Decorate” it with a Capsicum object.

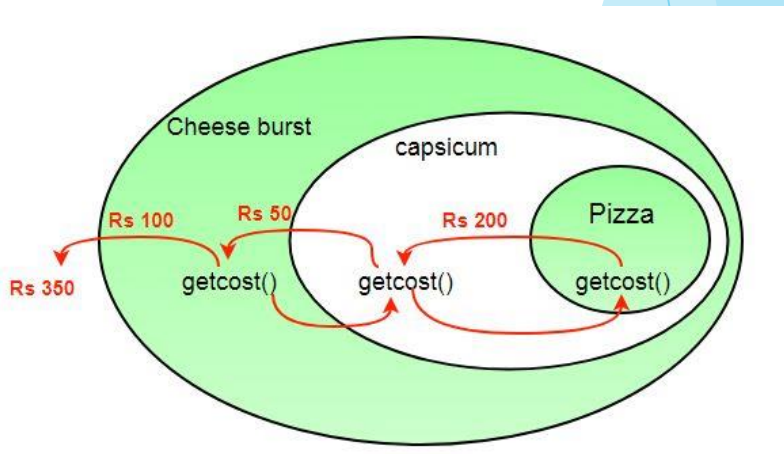


What we get in the end is a pizza with cheeseburst and capsicum toppings.

3. “Decorate” it with a CheeseBurst object

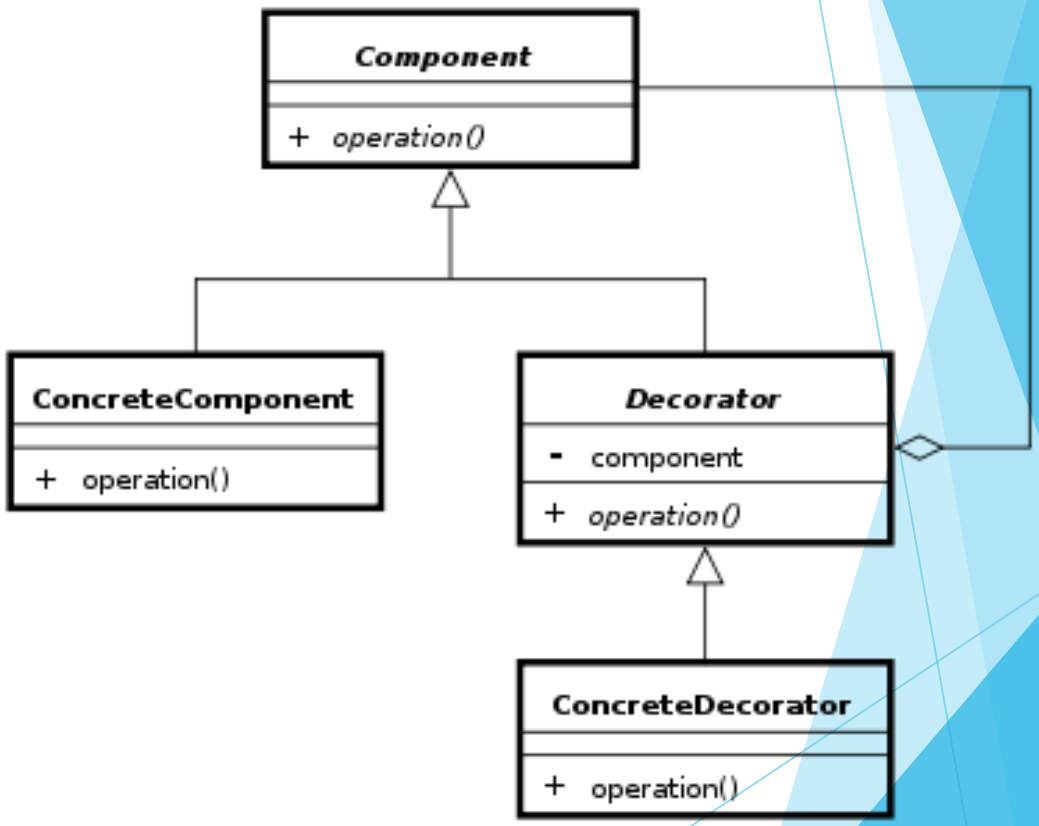


4. Call getCost() and use delegation instead of inheritance to calculate the toppings cost.



Decorator Pattern

- The Decorator Pattern attaches additional responsibilities to an object dynamically.
- Decorators provide a flexible alternative to subclassing for extending functionality.
- Decorators have the same super type as the object they decorate.
- You can use multiple decorators to wrap an object.
- Since decorators have same type as object, we can pass around decorated object instead of original.
- We can decorate objects at runtime.

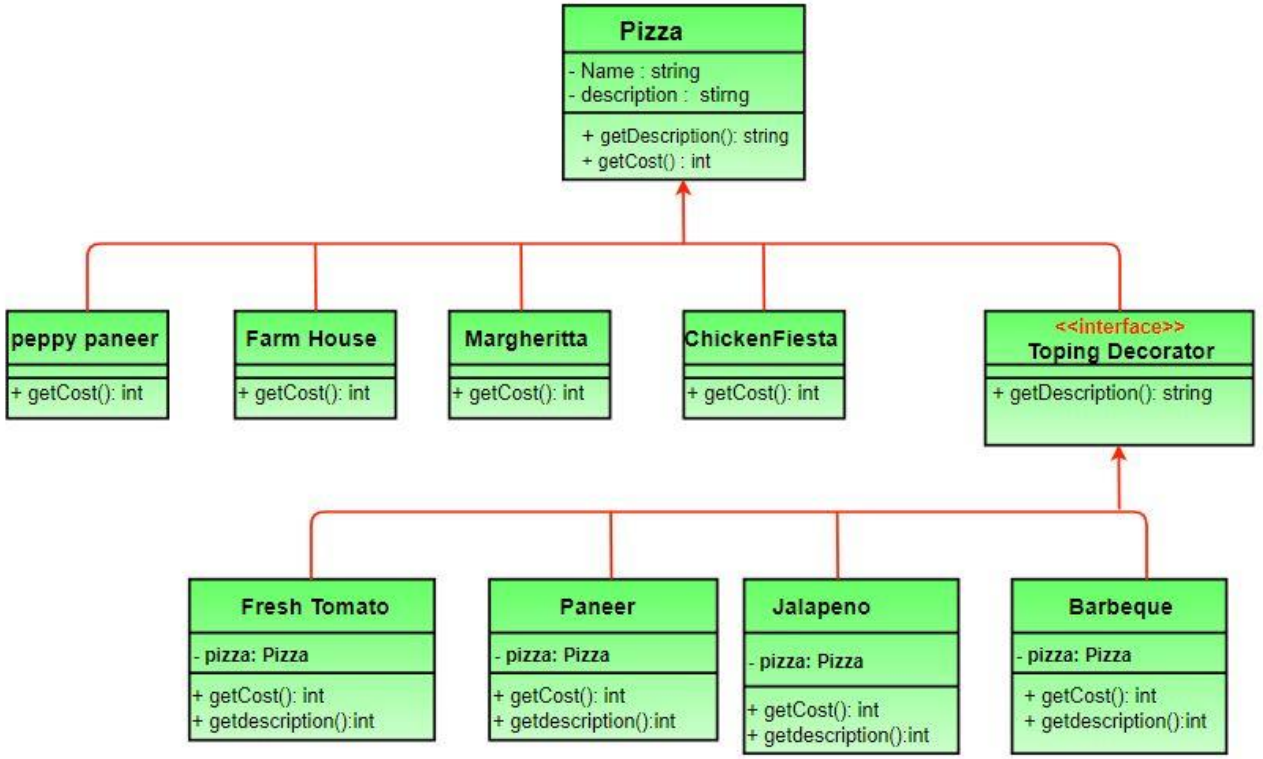


Decorator Pattern

- **Advantages:**
- The decorator pattern can be used to make it possible to extend (decorate) the functionality of a certain object at runtime.
- The decorator pattern is an alternative to subclassing. Subclassing adds behavior at compile time, and the change affects all instances of the original class; decorating can provide new behavior at runtime for individual objects.
- Decorator offers a pay-as-you-go approach to adding responsibilities. Instead of trying to support all foreseeable features in a complex, customizable class, you can define a simple class and add functionality incrementally with Decorator objects.

- **Disadvantages:**
- Decorators can complicate the process of instantiating the component because you not only have to instantiate the component, but wrap it in a number of decorators.
- It can be complicated to have decorators keep track of other decorators, because to look back into multiple layers of the decorator chain starts to push the decorator pattern beyond its true intent.

Example



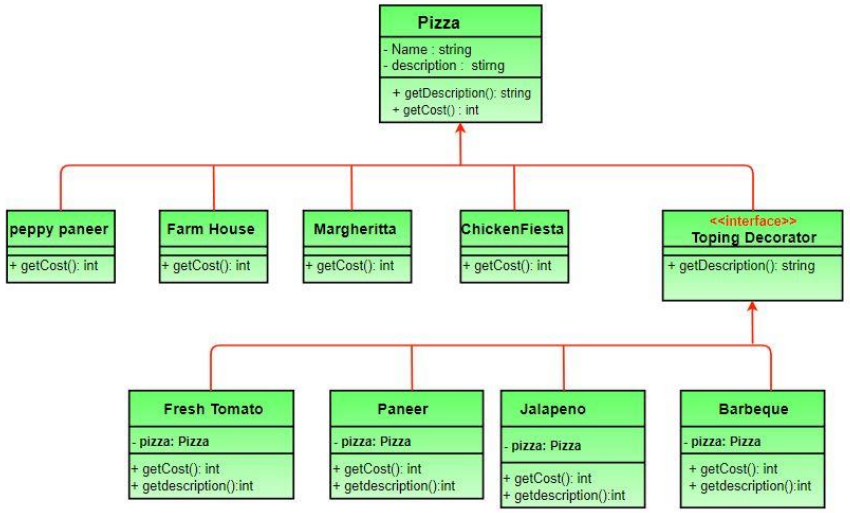
- **Pizza** acts as our abstract component class.
- There are four concrete components namely;
  - **PeppyPaneer** , **FarmHouse**, **Margherita**, **ChickenFiesta**.
- **ToppingsDecorator** is our abstract decorator
- **FreshTomato** , **Paneer**, **Jalapeno**, **Barbeque** are concrete decorators.

```java
// Abstract Pizza class (All classes extend
// from this)
abstract class Pizza
{
    // it is an abstract pizza
    String description = "Unkknown Pizza";

    public String getDescription()
    {
        return description;
    }

    public abstract int getCost();
}
```

```java
// Concrete pizza classes
class PeppyPaneer extends Pizza
{
    public PeppyPaneer() { description = "PeppyPaneer"; }
    public int getCost() {  return 100; }
}
class FarmHouse extends Pizza
{
    public FarmHouse() {  description = "FarmHouse"; }
    public int getCost() { return 200; }
}
class Margherita extends Pizza
{
    public Margherita()  { description = "Margherita"; }
    public int getCost() { return 100;  }
}
class ChickenFiesta extends Pizza
{
    public ChickenFiesta() { description = "ChickenFiesta";}
    public int getCost() { return 200; }
}
class SimplePizza extends Pizza
{
public SimplePizza() { description = "SimplePizza"; }
public int getCost() {  return 50;  }
}
```

```java
// The decorator class :  It extends Pizza to be
// interchangable with it topings decorator can
// also be implemented as an interface
abstract class ToppingsDecorator extends Pizza
{
    public abstract String getDescription();
}
```

```java
class FreshTomato extends ToppingsDecorator
{
    // we need a reference to obj we are decorating
    Pizza pizza;

    public FreshTomato(Pizza pizza) { this.pizza = pizza; }
    public String getDescription() {
        return pizza.getDescription() + ", Fresh Tomato ";
    }
    public int getCost() { return 40 + pizza.getCost(); }
}
class Barbeque extends ToppingsDecorator
{
    Pizza pizza;
    public Barbeque(Pizza pizza) {  this.pizza = pizza;  }
    public String getDescription() {
        return pizza.getDescription() + ", Barbeque ";
    }
    public int getCost() { return 90 + pizza.getCost(); }
}
class Paneer extends ToppingsDecorator
{
    Pizza pizza;
    public Paneer(Pizza pizza)  {  this.pizza = pizza; }
    public String getDescription() {
        return pizza.getDescription() + ", Paneer ";
    }
    public int getCost()  {  return 70 + pizza.getCost(); }
}
```




```java
// Driver class and method
class PizzaStore
{
    public static void main(String args[])
    {
        // create new margherita pizza
        Pizza pizza = new Margherita();
        System.out.println( pizza.getDescription() +
                        " Cost :" + pizza.getCost());

        // create new FarmHouse pizza
        Pizza pizza2 = new FarmHouse();

        // decorate it with freshtomato topping
        pizza2 = new FreshTomato(pizza2);

        //decorate it with paneer topping
        pizza2 = new Paneer(pizza2);

        System.out.println( pizza2.getDescription() +
                        " Cost :" + pizza2.getCost());
        Pizza pizza3 = new Barbeque(null);    //no specific pizza
        System.out.println( pizza3.getDescription() + "  Cost :" + pizza3.getCost());
    }
}
```

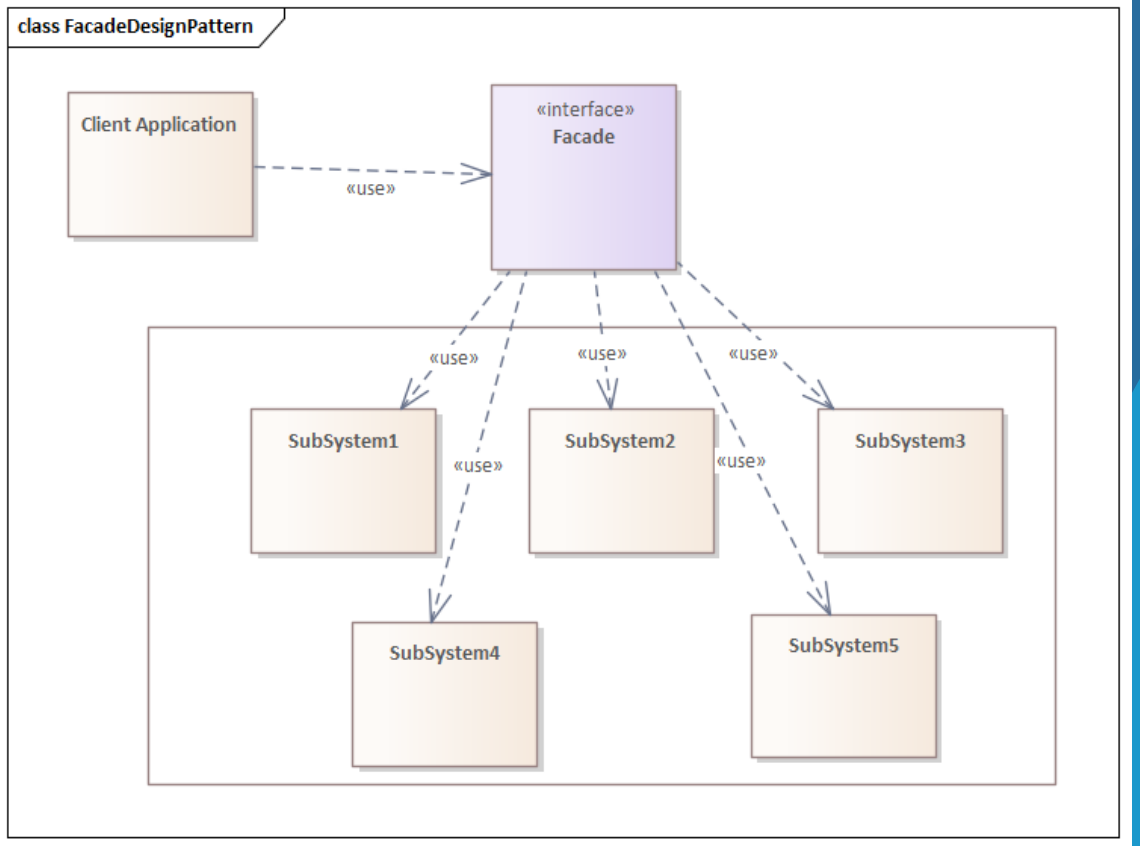
```
Margherita Cost :100
FarmHouse, Fresh Tomato , Paneer Cost :310
```

Facade Pattern

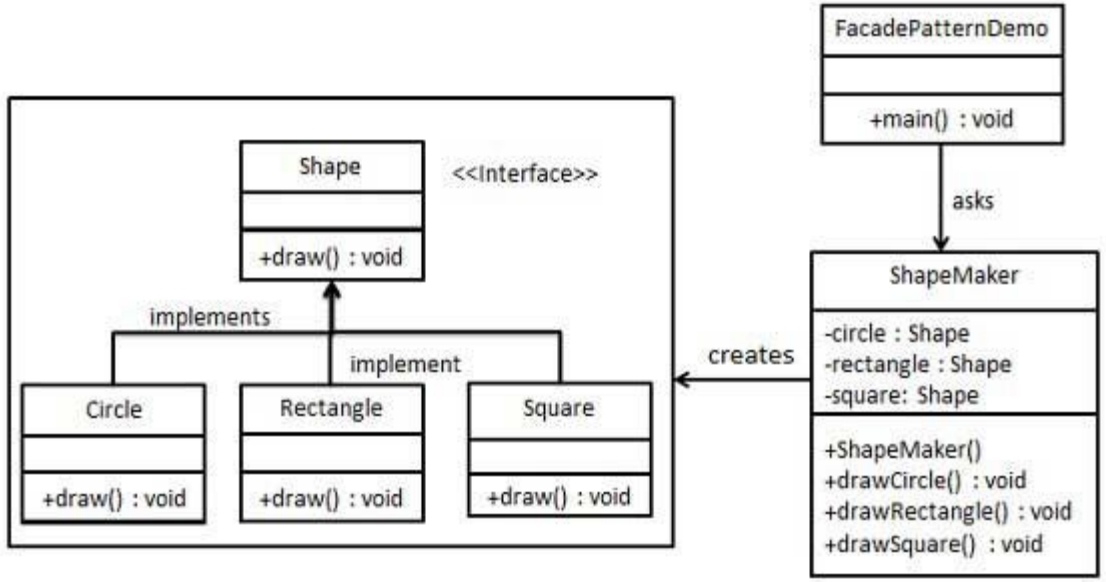
DESIGN PATTERNS

Facade pattern

- hides the complexities of the system.
- provides an interface to the client using which the client can access the system
- involves a single class which provides simplified methods required by client
- delegates calls to methods of existing system classes



Example


FacadePatternDemo
+main() : void
asks
ShapeMaker
-circle : Shape
-rectangle : Shape
-square: Shape
+ShapeMaker()
+drawCircle() : void
+drawRectangle() : void
+drawSquare() : void
creates
Shape
<<Interface>>
+draw() : void
implements
implement
Circle
+draw() : void
Rectangle
+draw() : void
Square
+draw() : void

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

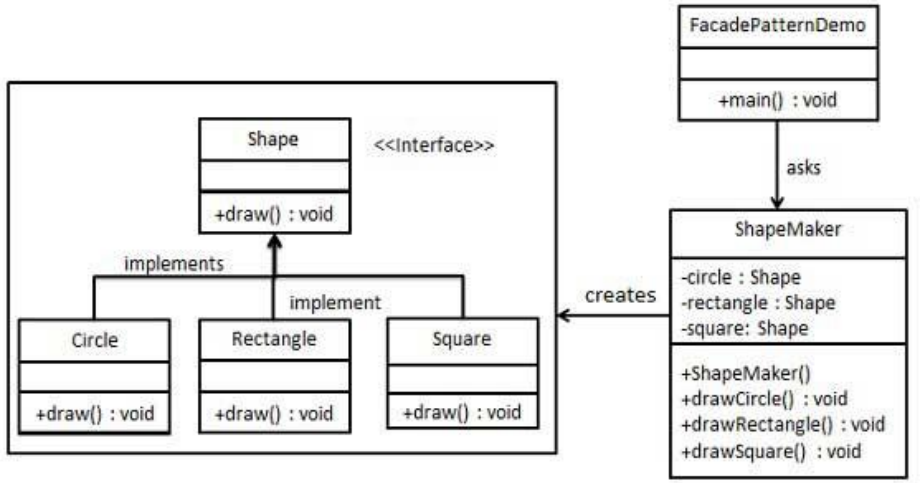
```java
public class Rectangle implements Shape {

   @Override
   public void draw() {
      System.out.println("Rectangle::draw()");
   }
}
```

```java
public class Square implements Shape {

   @Override
   public void draw() {
      System.out.println("Square::draw()");
   }
}
```

```java
public class Circle implements Shape {

   @Override
   public void draw() {
      System.out.println("Circle::draw()");
   }
}
```

```java
public class ShapeMaker {
   private Shape circle;
   private Shape rectangle;
   private Shape square;

   public ShapeMaker() {
      circle = new Circle();
      rectangle = new Rectangle();
      square = new Square();
   }

   public void drawCircle(){
      circle.draw();
   }
   public void drawRectangle(){
      rectangle.draw();
   }
   public void drawSquare(){
      square.draw();
   }
}
```

```java
public class FacadePatternDemo {
   public static void main(String[] args) {
      ShapeMaker shapeMaker = new ShapeMaker();

      shapeMaker.drawCircle();
      shapeMaker.drawRectangle();
      shapeMaker.drawSquare();
   }
}
```




```
Circle::draw()
Rectangle::draw()
Square::draw()
```