Software Design Patterns

Email: Walaagad@cis.asu.edu.eg

Design Patterns Space

SCOPE	PURPOSE		
	CREATIONAL	STRUCTURAL	BEHAVIORAL
CLASS	FACTORY METHOD	ADAPTER	INTERPRETER
			TEMPLATE METHOD
OBJECT	ABSTRACT FACTORY	ADAPTER (OBJECT)	CHAIN OF RESPONSIBILITY
	BUILDER	BRIDGE	COMMAND
	PROTOTYPE	COMPOSITE	ITERATOR
	SINGELTON	DECORATOR	MEDIATOR
		FACADE	MEMENTO
		FLYWEIGHT	OBSERVER
		PROXY	STATE
			STRATEGY
			VISITOR

Creational Patterns

- Singleton
- Prototype
- Builder
- Abstract Factory

Creational Patterns

- Singleton
- Prototype
- Builder
- Abstract Factory

String builder I Java

Builder Pattern

Separate the construction of a complex object from its representation so that the same construction processes can create different representations

Build complex step by step

The boss tells Joe what he wants

Director

Builder

Joe tells the furniture maker to build a chair. Note that the furniture maker could have also built a beach furniture or a skateboard.

Concrete Builder1

Concrete Builder2

Concrete Builder1

The boss wants some new furniture. He calls his direct aid.

Client

Product

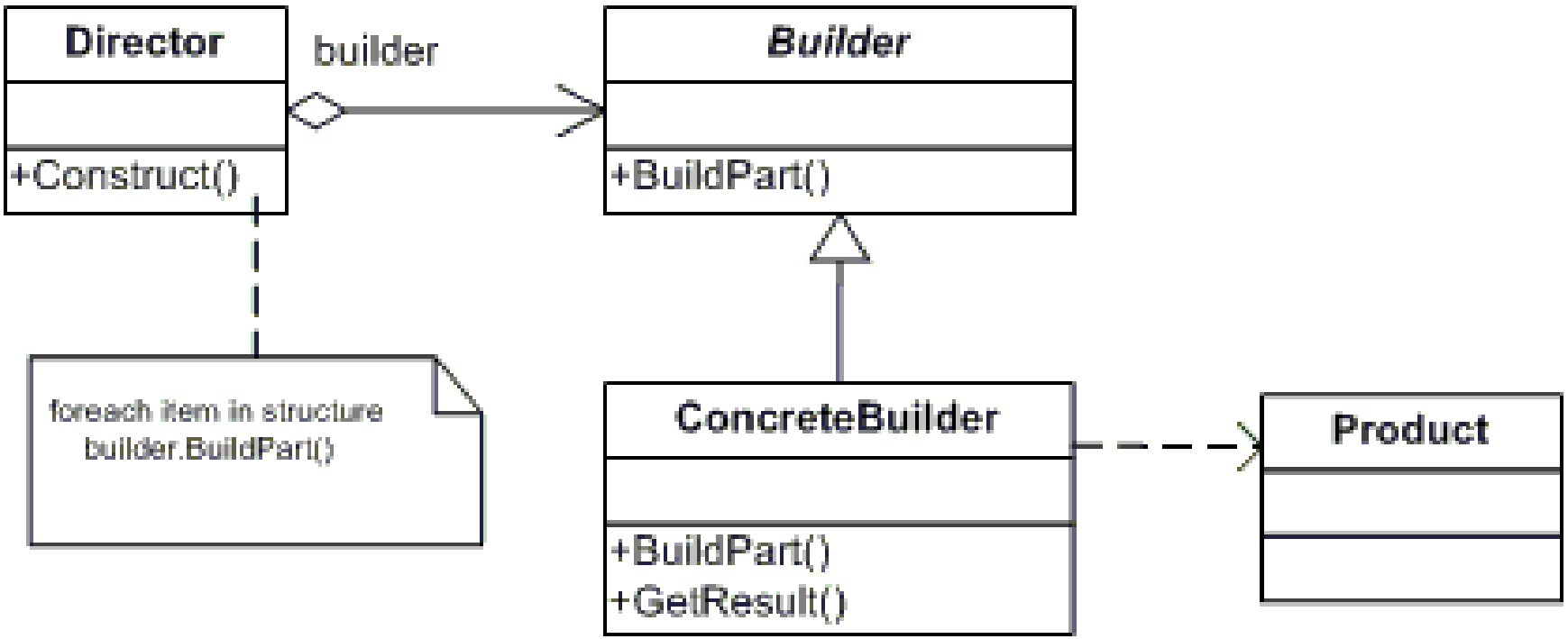
The boss gets the armchair produced by the furniture maker and his tools

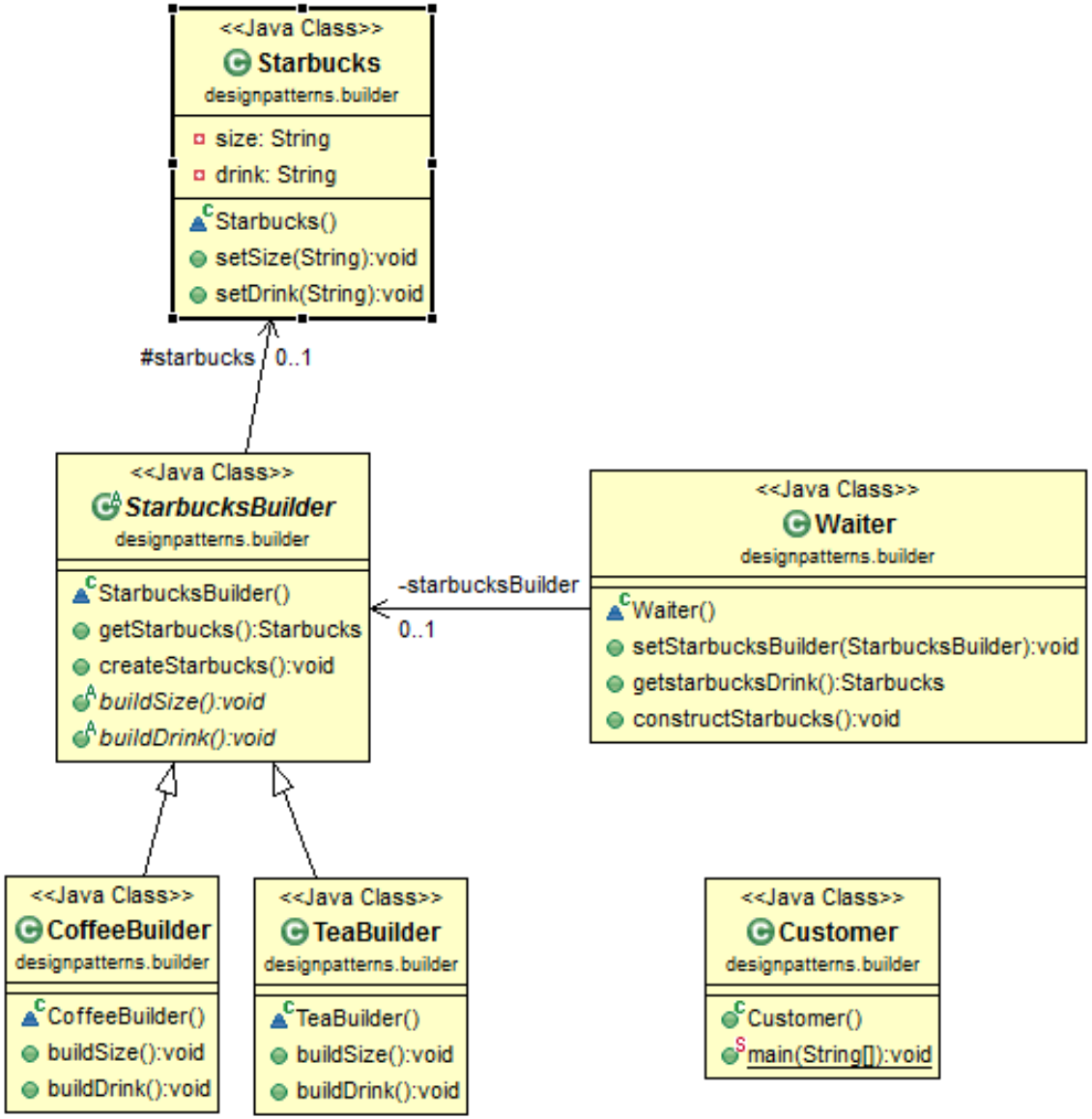
Builder Pattern

The pattern is used to create objects made from a bunch of other objects.

- When you want to build an object made up of other objects.
  - When you want the creation of these parts to be independent of the main object.
  - Hide the creation of the parts from the clients so both aren't dependent.
  - The builder knows the specifics and nobody else does.

Builder Pattern UML




<<Java Class>>
Starbucks
designpatterns.builder
size: String
drink: String
Starbucks()
setSize(String):void
setDrink(String):void
#starbucks
0..1
<<Java Class>>
StarbucksBuilder
designpatterns.builder
StarbucksBuilder()
getStarbucks():Starbucks
createStarbucks():void
buildSize():void
buildDrink():void
-starbucksBuilder
0..1
<<Java Class>>
Waiter
designpatterns.builder
Waiter()
setStarbucksBuilder(StarbucksBuilder):void
getstarbucksDrink():Starbucks
constructStarbucks():void
<<Java Class>>
CoffeeBuilder
designpatterns.builder
CoffeeBuilder()
buildSize():void
buildDrink():void
<<Java Class>>
TeaBuilder
designpatterns.builder
TeaBuilder()
buildSize():void
buildDrink():void
<<Java Class>>
Customer
designpatterns.builder
Customer()
main(String[]):void

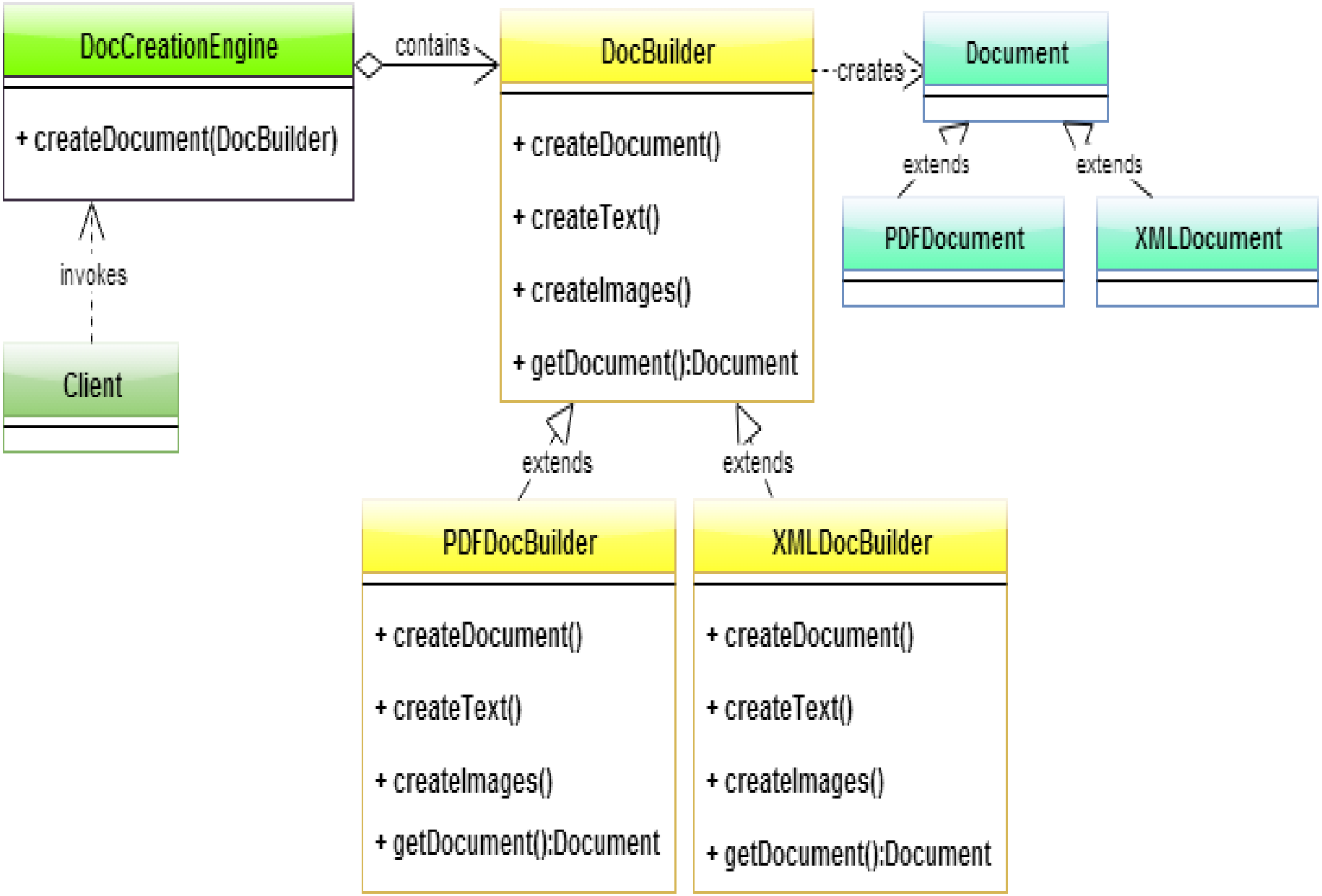
```java
// produce to be built
class Starbucks {
        private String size;
        private String drink;

        public void setSize(String size) {
                this.size = size;
        }

        public void setDrink(String drink) {
                this.drink = drink;
        }
}
```

```java
//abstract builder
abstract class StarbucksBuilder {
        protected Starbucks starbucks;

        public Starbucks getStarbucks() {
                return starbucks;
        }

        public void createStarbucks() {
                starbucks = new Starbucks();
                System.out.println("a drink is created");
        }

        public abstract void buildSize();
        public abstract void buildDrink();
}
```

```java
// Concrete Builder to build tea
class TeaBuilder extends StarbucksBuilder {
        public void buildSize() {
                starbucks.setSize("large");
                System.out.println("build large size");
        }

        public void buildDrink() {
                starbucks.setDrink("tea");
                System.out.println("build tea");
        }

}
```

```java
// Concrete builder to build coffee
class CoffeeBuilder extends StarbucksBuilder {
        public void buildSize() {
                starbucks.setSize("medium");
                System.out.println("build medium size");
        }

        public void buildDrink() {
                starbucks.setDrink("coffee");
                System.out.println("build coffee");
        }
}
```

```
//director to encapsulate the builder
class Waiter {
        private StarbucksBuilder starbucksBuilder;

        public void setStarbucksBuilder(StarbucksBuilder builder) {
                starbucksBuilder = builder;
        }

        public Starbucks getstarbucksDrink() {
                return starbucksBuilder.getStarbucks();
        }

        public void constructStarbucks() {
                starbucksBuilder.createStarbucks();
                starbucksBuilder.buildDrink();
                starbucksBuilder.buildSize();
        }
}
        //customer
        public class Customer {
                public static void main(String[] args) {
                        Waiter waiter = new Waiter();
                        StarbucksBuilder coffeeBuilder = new CoffeeBuilder();

                        //Alternatively you can use tea builder to build a tea
                        //StarbucksBuilder teaBuilder = new TeaBuilder();

                        waiter.setStarbucksBuilder(coffeeBuilder);
                        waiter.constructStarbucks();

                        //get the drink built
                        Starbucks drink = waiter.getstarbucksDrink();

                }
        }
```


DocCreationEngine
+ createDocument(DocBuilder)
contains
DocBuilder
+ createDocument()
+ createText()
+ createImages()
+ getDocument():Document
creates
Document
extends
extends
PDFDocument
XMLDocument
invokes
Client
extends
extends
PDFDocBuilder
+ createDocument()
+ createText()
+ createImages()
+ getDocument():Document
XMLDocBuilder
+ createDocument()
+ createText()
+ createImages()
+ getDocument():Document

```java
//PDFDocBuilder.java
public class PDFDocBuilder extends DocBuilder{
  private PDFDocument pdfDoc;
  public void createDocument(){
    this.pdfDoc=new PDFDocument();
  }
  public void createText(){
    System.out.println("Creating text for PDF Document.");
  }
  public  void createImages(){
    System.out.println("Creating images for PDF Document.");
  }
  public  Document getDocument(){
   System.out.println("Fetching PDF Document.");
   return this.pdfDoc;
  }
}

//XMLDocBuilder.java
public class XMLDocBuilder extends DocBuilder{
  private XMLDocument xmlDoc;
  public void createDocument(){
    this.xmlDoc=new XMLDocument();
  }
  public void createText(){
    System.out.println("Creating text for XML Document.");
  }
  public  void createImages(){
    System.out.println("Creating images for XML Document.");
  }
  public  Document getDocument(){
    System.out.println("Fetching PDF Document.");
    return this.xmlDoc;
   }
}
```

```java
//Interface - Document.java
public interface Document{
}

//Class PDFDocument.java
public class PDFDocument implements Document{
    //attributes for holding the PDFDocument
}

//Class XMLDocument.java
public class XMLDocument implements Document{
  //attributes for holding the XMLDocument
}
```

```java
//DocCreationEngine.java
public class DocCreationEngine{
  public void generateDocument(DocBuilder builder){
    builder.createDocument();
    builder.createText();
    builder.createImages();
  }
 }

//Client.java
public class Client{
  public static void main(String args[]){
   DocCreationEngine engine=new DocCreationEngine();
   //Creating PDF Document
   PDFDocBuilder pdfDocBuilder=new PDFDocBuilder();
   engine.generateDocument(pdfDocBuilder);
   PDFDocument pdfDocument=(PDFDocument)pdfDocBuilder.getDocument();
   //Creating XML Document
   XMLDocBuilder xmlDocBuilder=new XMLDocBuilder();
   engine.generateDocument(xmlDocBuilder);
   XMLDocument xmlDocument=(XMLDocument)xmlDocBuilder.getDocument();
  }
}
```

Builder Pattern Advantages

- It provides a clear separation between the construction and representation of an object.
- It provides better control over the construction process.
- It supports changing the internal representation of objects.

Creational Patterns

- Singleton
- Prototype
- Builder
- Factory

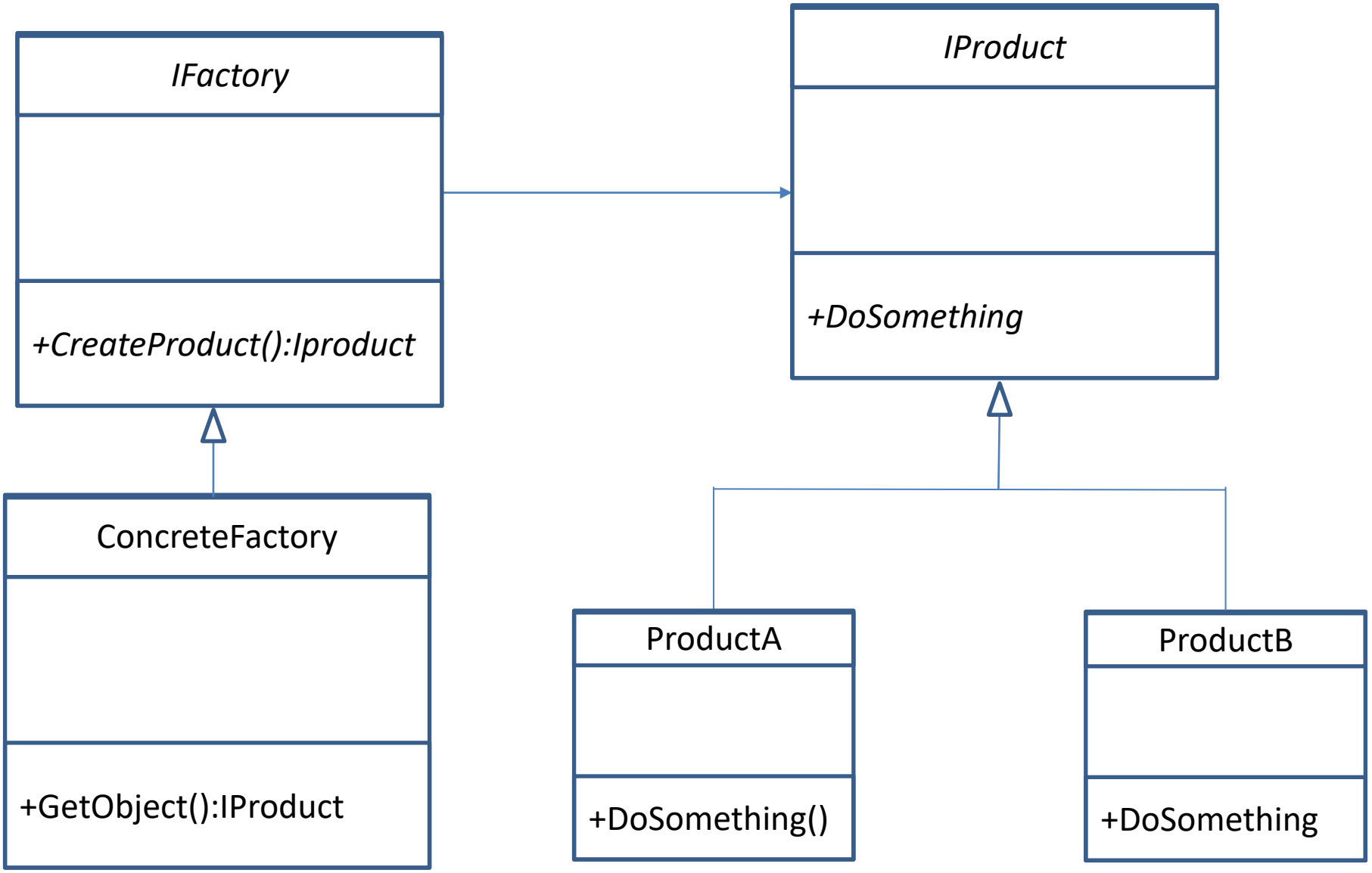
Creational Patterns

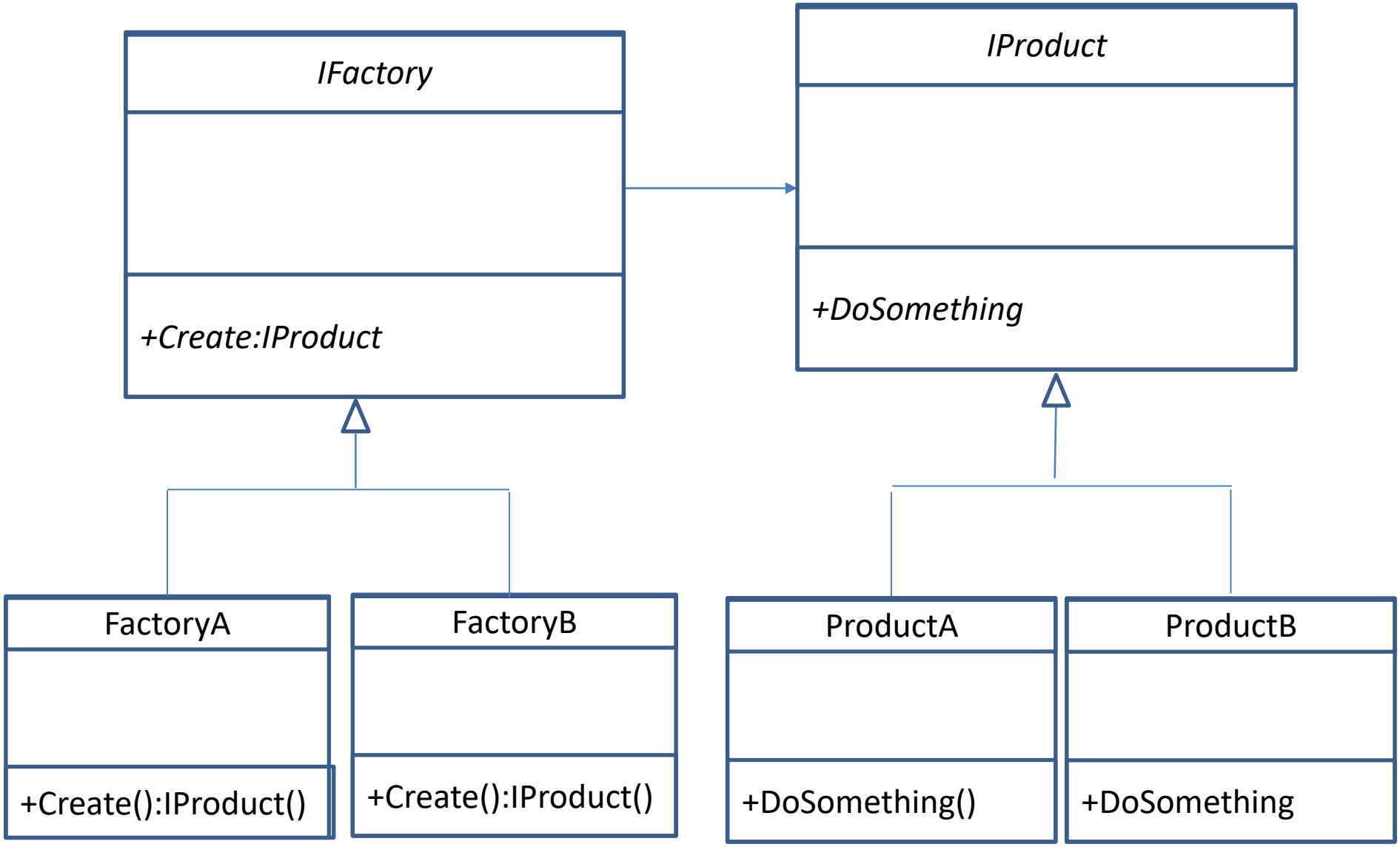
- Singleton
- Prototype
- Builder
- Factory

Factory Pattern

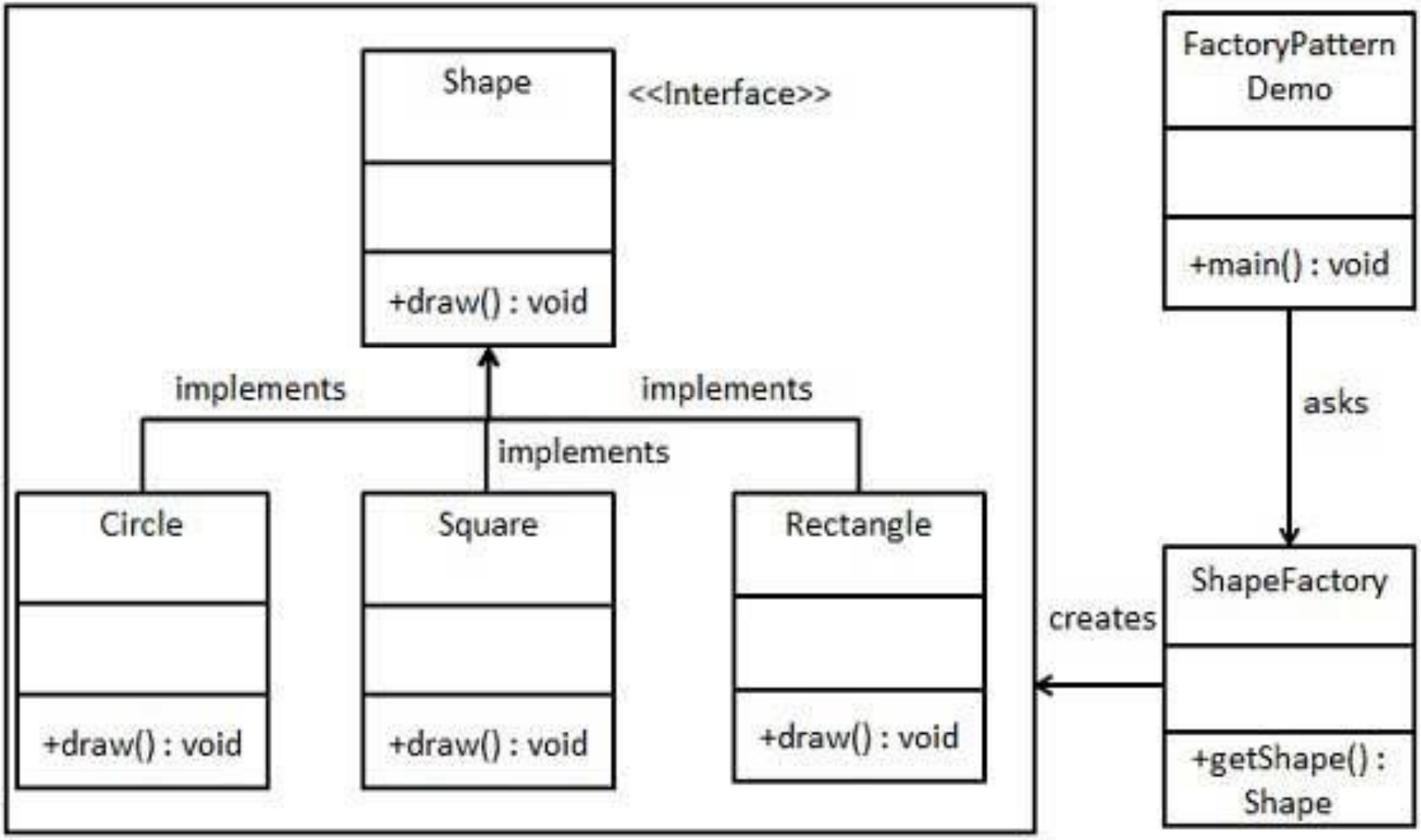
Define an interface for creating an object, but let subclasses decide which class to instantiate

Create an object without exposing the creation logic to the client and refer to newly created object using a common interface


IFactory
+CreateProduct():Iproduct
IProduct
+DoSomething
ConcreteFactory
+GetObject():IProduct
ProductA
+DoSomething()
ProductB
+DoSomething


IFactory
+Create:IProduct
IProduct
+DoSomething
FactoryA
+Create():IProduct()
FactoryB
+Create():IProduct()
ProductA
+DoSomething()
ProductB
+DoSomething

Factory Example



Step 1->

Create an interface.

Shape.java

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

Step 2->

Create concrete classes implementing the same interface.

1-Rectangle.java

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

2-Square.java

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

3-Circle.java

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

Step 3->

Create a Factory to generate object of concrete class based on given information.

ShapeFactory.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;
   }
}
```

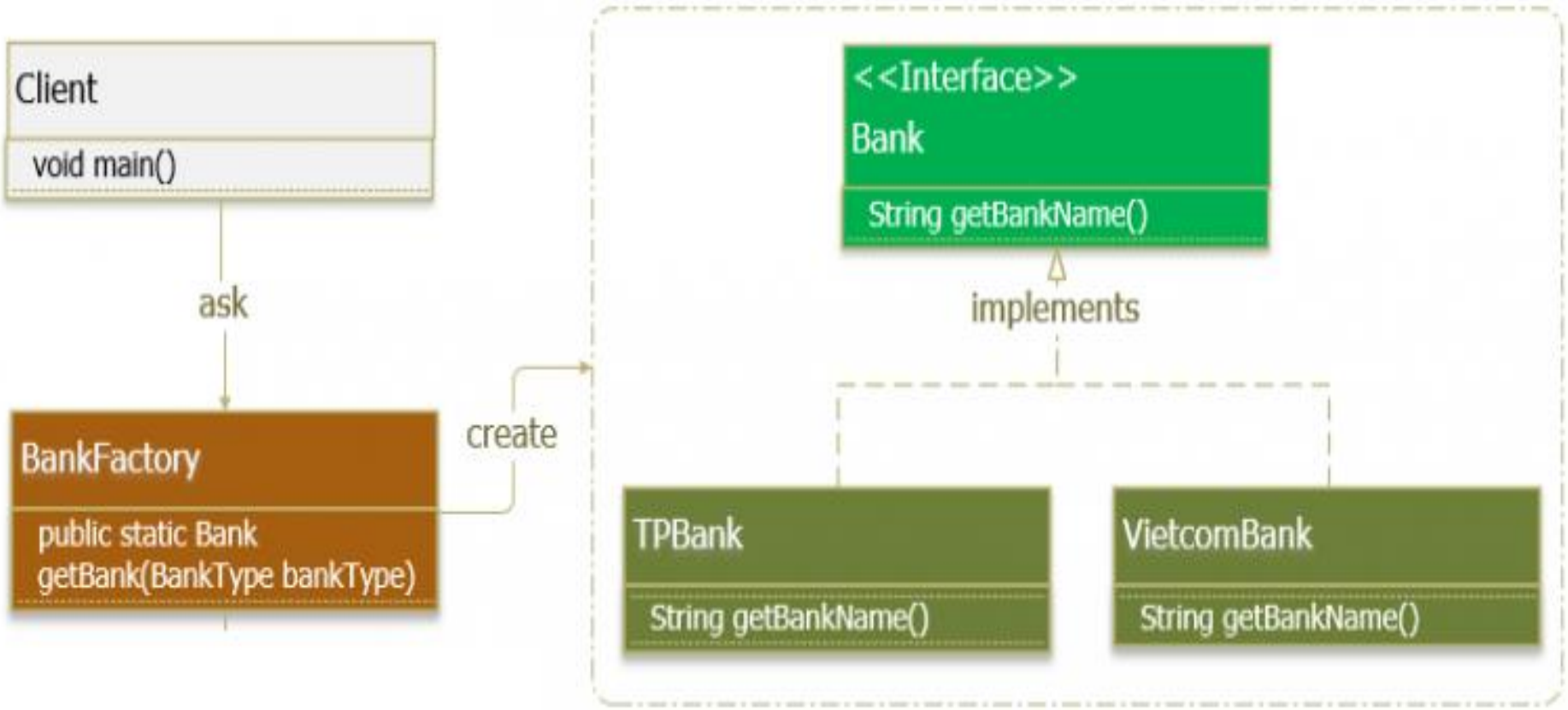
Step 4->

Use the Factory to get object of concrete class by passing an informatic

FactoryPatternDemo.java

```
public class FactoryPatternDemo
{
  public static void main(String[] args)
  {
     ShapeFactory shapeFactory = new ShapeFactory();

     Shape shape1 = shapeFactory.getShape("CIRCLE");
          shape1.draw();

    Shape shape2 = shapeFactory.getShape("RECTANGLE");
           shape2.draw();

     Shape shape3 = shapeFactory.getShape("SQUARE");
          shape3.draw();
   }
}
```

Factory Example



Supper Class:

```java
public interface Bank {
    String getBankName();
}
```

Sub Classes:

```java


public class TPBank implements Bank {

    @Override
    public String getBankName() {
        return "TPBank";
    }

}
```

```java


public class VietcomBank implements Bank {

    @Override
    public String getBankName() {
        return "VietcomBank";
    }

}
```

Factory class:

```
public class BankFactory {

    private BankFactory() {
    }

    public static final Bank getBank(BankType bankType) {
        switch (bankType) {

        case TPBANK:
            return new TPBank();

        case VIETCOMBANK:
            return new VietcomBank();

        default:
            throw new IllegalArgumentException("This bank type is un
        }
    }

}
```

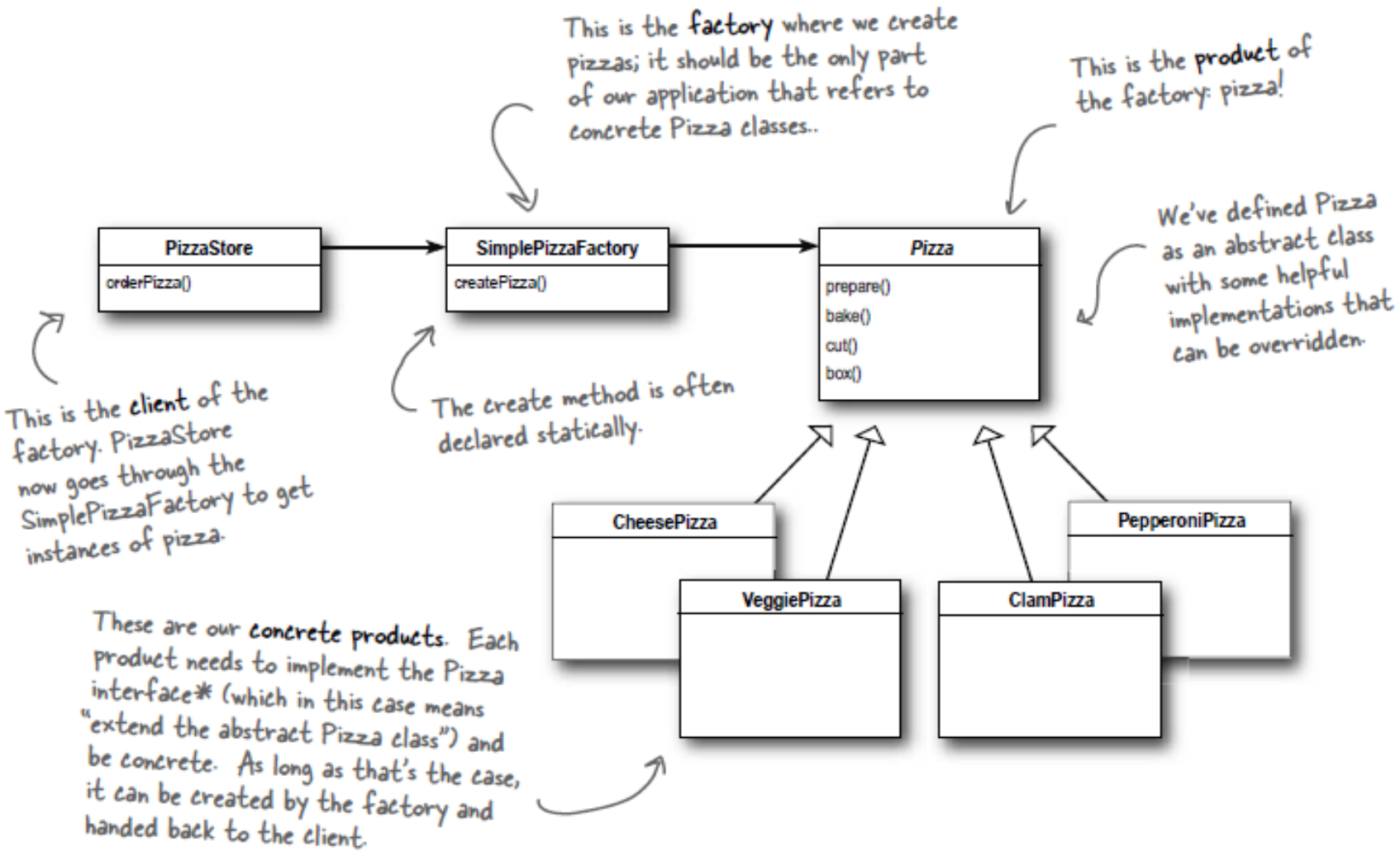
Bank type:

```
public enum BankType {

    VIETCOMBANK, TPBANK;

}
```

Client:

```
public class Client {

    public static void main(String[] args) {
        Bank bank = BankFactory.getBank(BankType.TPBANK);
        System.out.println(bank.getBankName()); // TPBank
    }
}
```

Factory Example



Factory Example

Joe, the Pizza store idea is really great. We are making a lot of money. I'm out of the duck business. I would like to have pizza shops everywhere: Paris, Rome, Stockholm..

Una pepperoni pizza semplice, ma saporita con il delicato prosciutto, per favore.

Je voudrais commander une pizza s'il vous plaît avec boeuf haché et sauce tomate. Je n'aime pas le bacon et le ananas.

Jag skulle vilja ha en pepperoni pizza med kötbullar

Factory Example