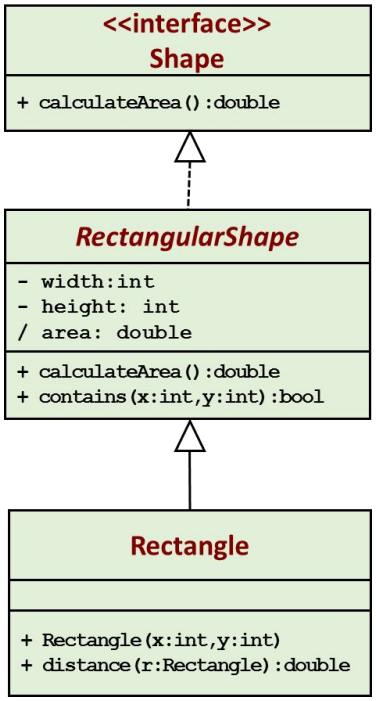
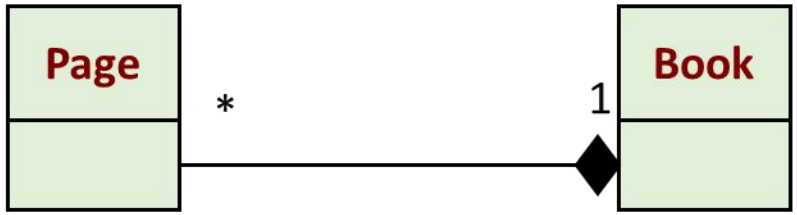
Creational Patterns

UML Revisited



Generalization

Aggregation: “is part of”



Composition: “is entirely made of”

Dependency: “uses”

Creational Patterns to be Covered

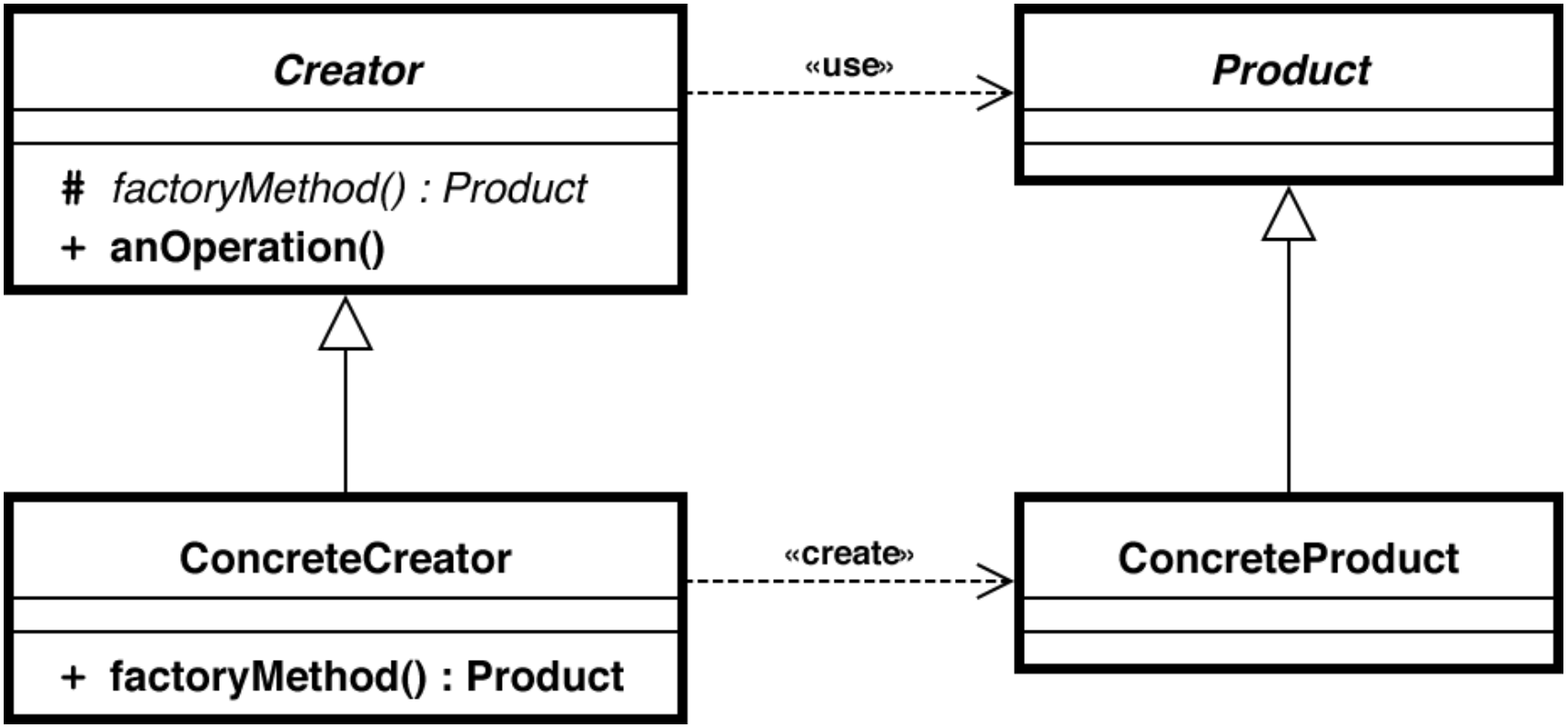
- Factory method
- Abstract factory
- Builder
- Singleton

Examples taken from:
https://www.tutorialspoint.com/design_pattern/
https://www.javatpoint.com/design-patterns-in-java

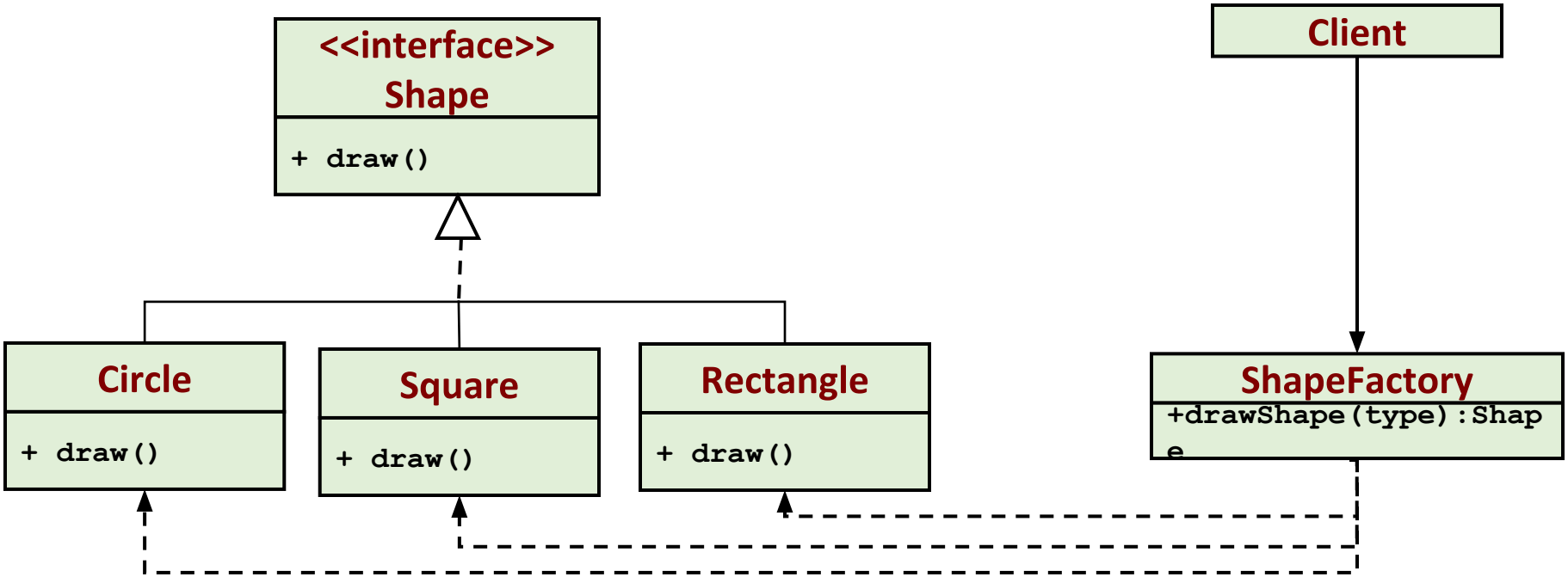
Factory Method

Intent	• Define an interface for creating an object, but let subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses.
Problem	• A framework needs to standardize the architectural model for a range of applications, but allow for individual applications to define their own domain objects and provide for their instantiation. • Enable the creator to defer product creation to sub-class.

Solution


Creator
factoryMethod() : Product
+ anOperation()
«use»
Product
ConcreteCreator
+ factoryMethod() : Product
«create»
ConcreteProduct

Example: Drawing



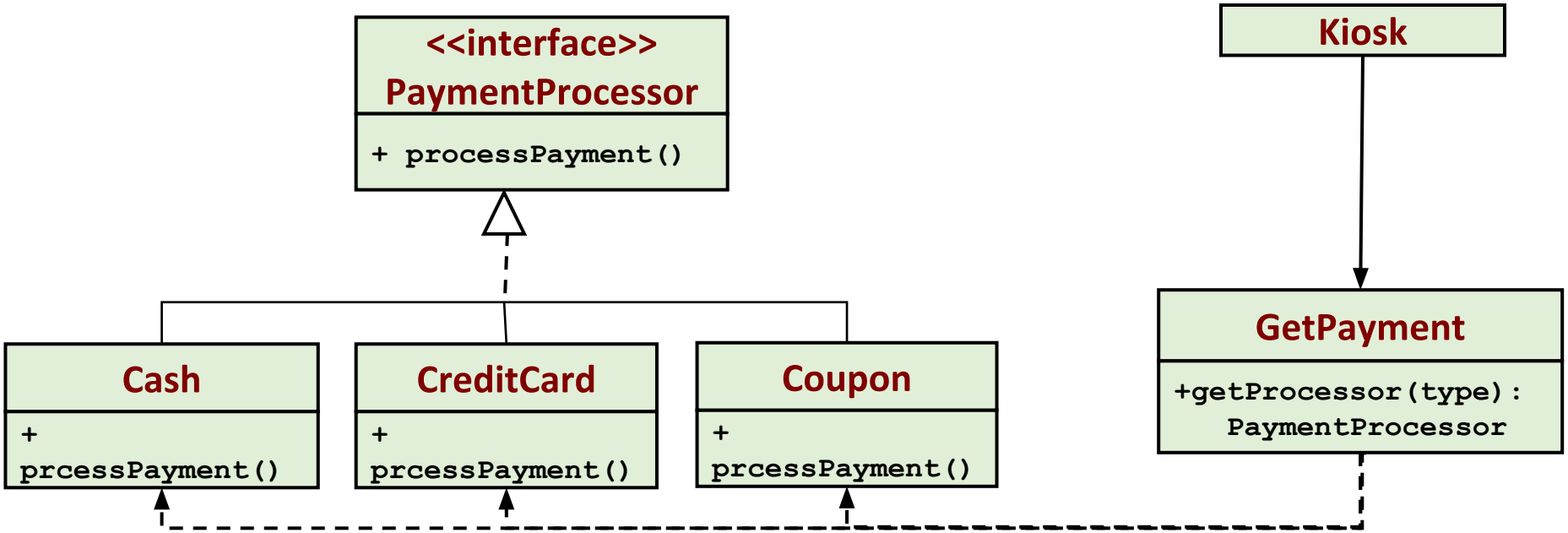
```
public interface Shape {
      void draw();
}
public class Rectangle implements Shape {
     @Override
      public void draw() {
       System.out.println("Inside Rectangle::draw() method.");
      }
 }
```

Similar implementation for Square and Circle

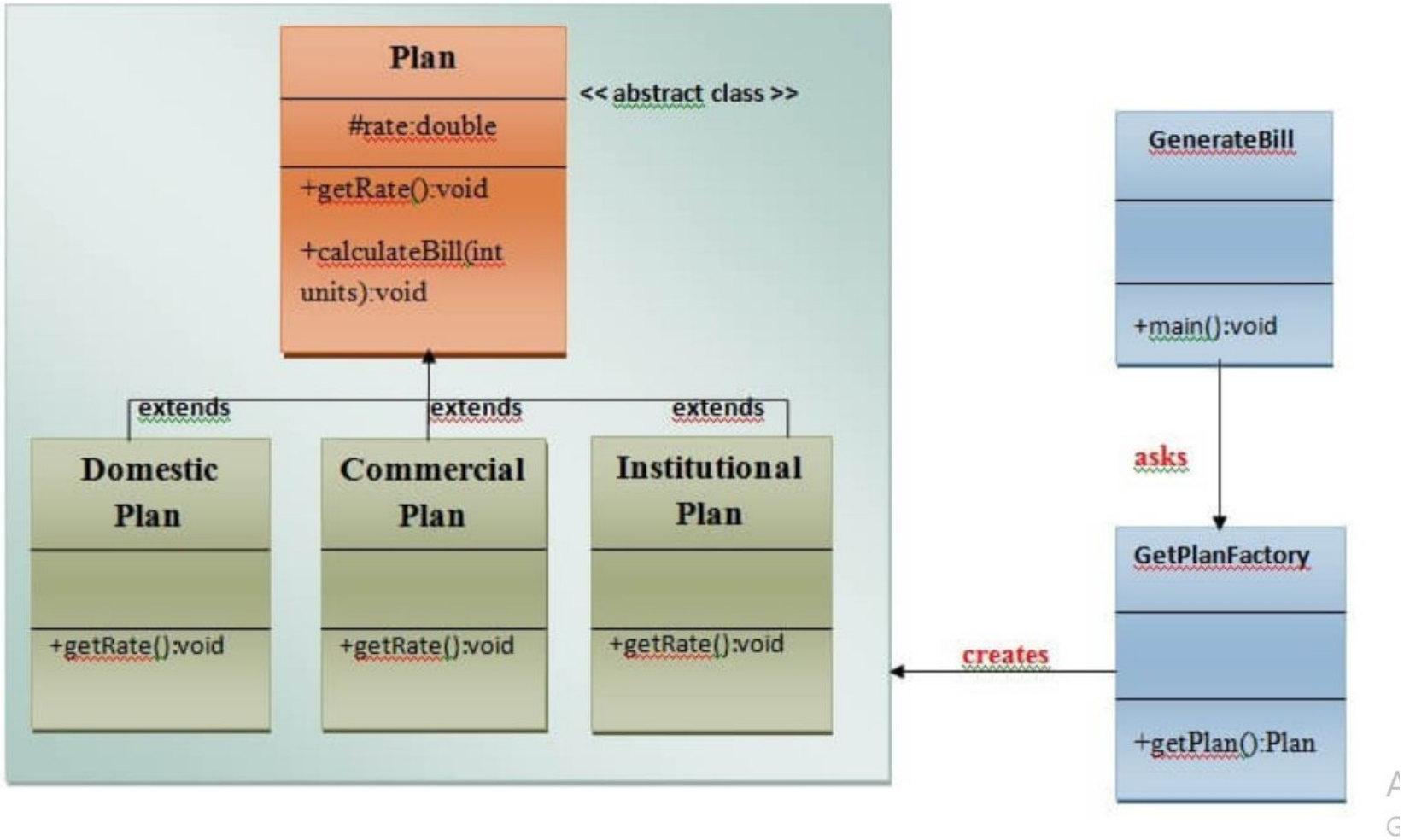
```
public class ShapeFactory {

   //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;
   }
}
```

```
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();
```

Example: Kiosk



Factory


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

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

        public void calculateBill(int units){
            System.out.println(units*rate);
         }
}//end of Plan class.
```

Step 2: Create the concrete classes that extends Plan

```java
class  DomesticPlan extends Plan{
       //@override
        public void getRate(){
            rate=3.50;
       }
  }//end of DomesticPlan class.
```

```java
class  CommercialPlan extends Plan{
  //@override
   public void getRate(){
      rate=7.50;
  }
/end of CommercialPlan class.
```

```java
class  InstitutionalPlan extends Plan{
  //@override
   public void getRate(){
      rate=5.50;
  }
/end of InstitutionalPlan class.
```

```
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;
  }
}//end of GetPlanFactory class.
```

```
import java.io.*;
class GenerateBill{
   public static void main(String args[])throws IOException{
     GetPlanFactory planFactory = new GetPlanFactory();

     System.out.print("Enter the name of plan for which the bill will be generated: ");
     BufferedReader br=new BufferedReader(new InputStreamReader(System.in));

     String planName=br.readLine();
     System.out.print("Enter the number of units for bill will be calculated: ");
     int units=Integer.parseInt(br.readLine());

     Plan p = planFactory.getPlan(planName);
     //call getRate() method and calculateBill()method of DomesticPaln.

      System.out.print("Bill amount for "+planName+" of  "+units+" units is: ");
         p.getRate();
         p.calculateBill(units);
          }
   }//end of GenerateBill class.
```

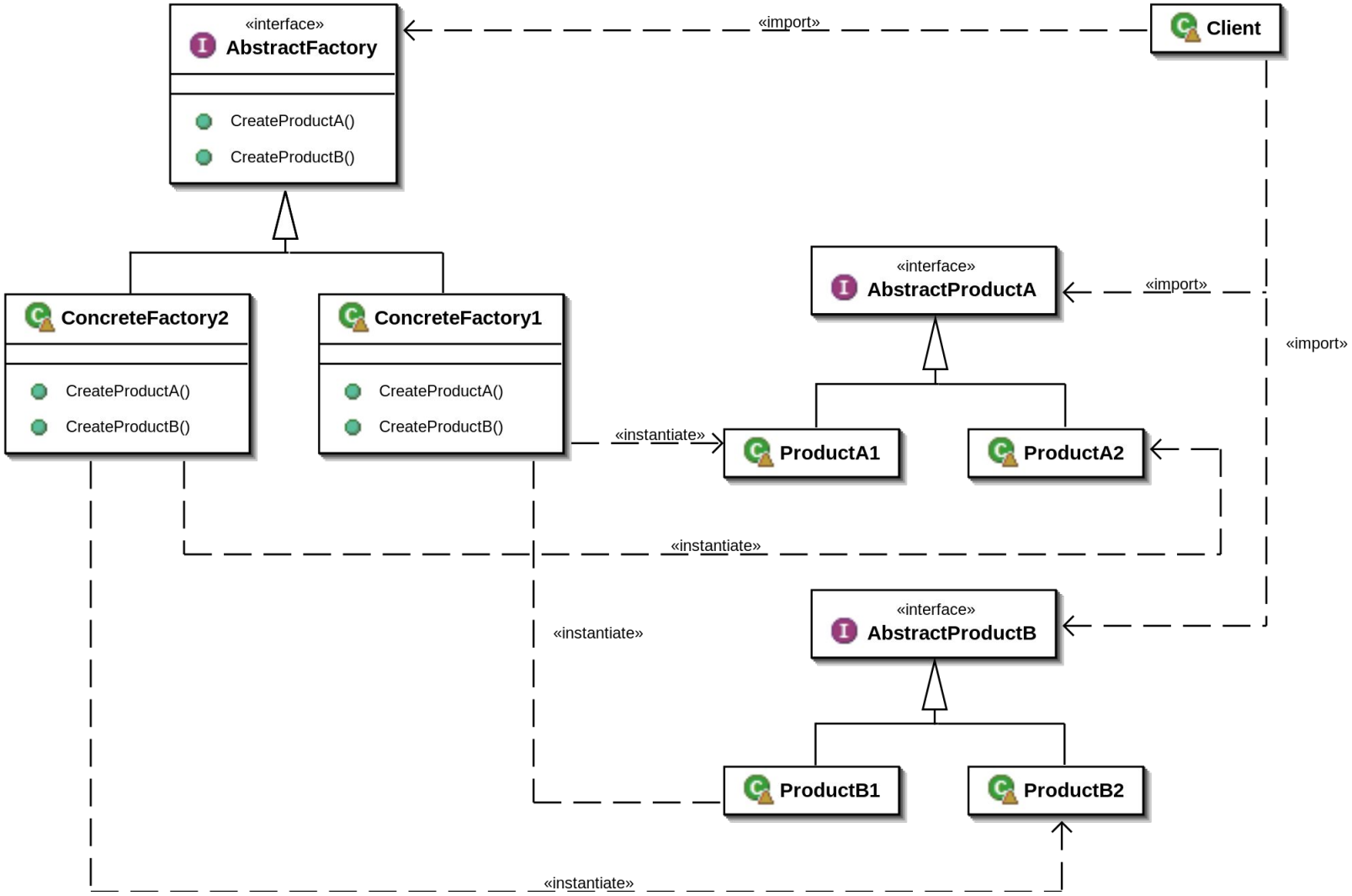
Consequences

- Factory design pattern provides approach to code for interface rather than implementation.
- Factory pattern removes the instantiation of actual implementation classes from client code. Factory pattern makes our code more robust, less coupled and easy to extend. For example, we can easily change lower class implementation because client program is unaware of this.
- Factory pattern provides abstraction between implementation and client classes through inheritance.

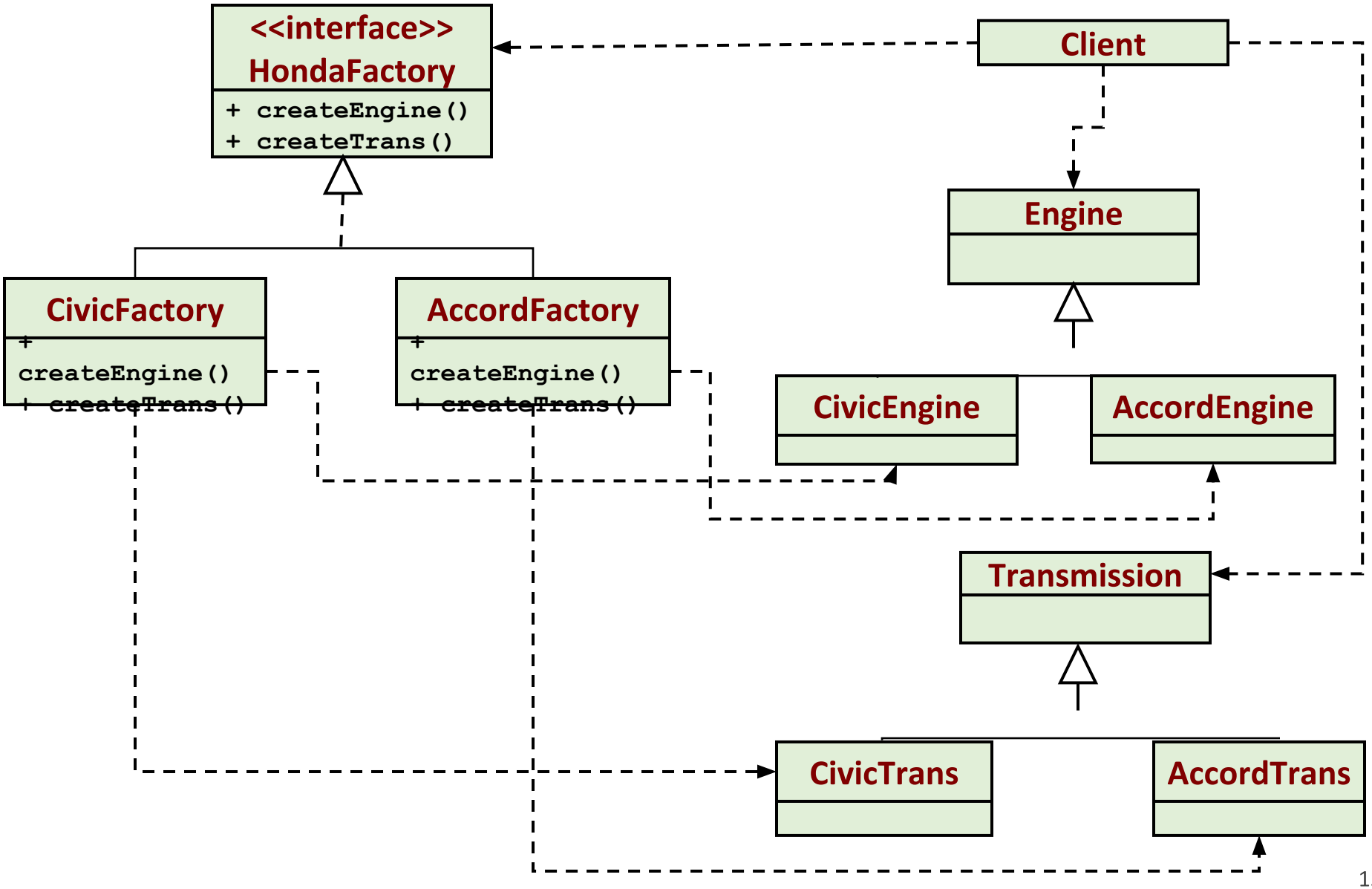
Abstract Factory

Intent	• Provide an interface for creating families of related or dependent objects without specifying their concrete classes.
Problem	• A portable application needs to encapsulate platform dependencies • Consider an application that support multiple look and feels. • An application need to work with multiple types of DBMS

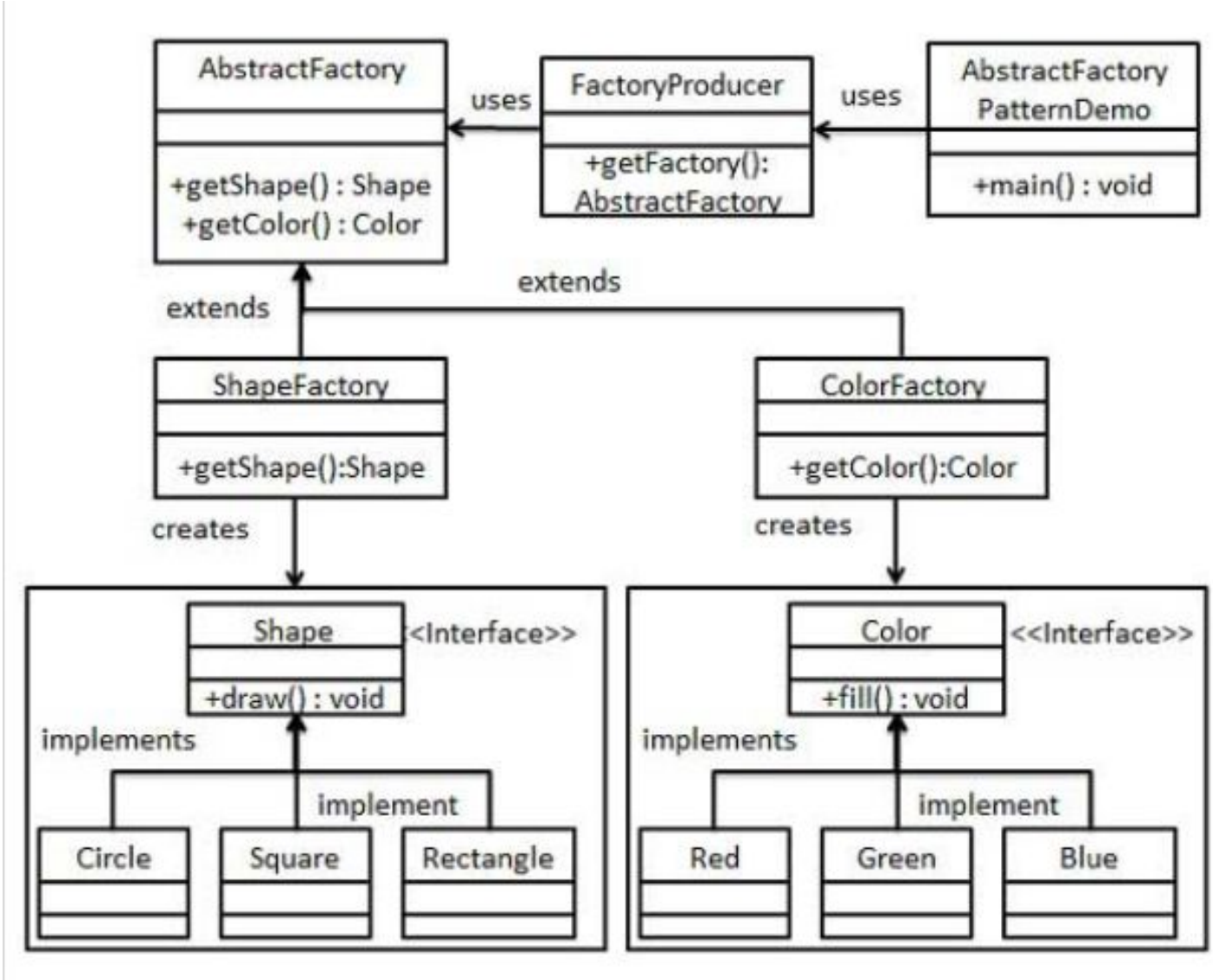
Solution


«interface»
AbstractFactory
CreateProductA()
CreateProductB()
«import»
Client
«interface»
AbstractProductA
«import»
«import»
ConcreteFactory2
CreateProductA()
CreateProductB()
ConcreteFactory1
CreateProductA()
CreateProductB()
«instantiate»
ProductA1
ProductA2
«instantiate»
«instantiate»
«interface»
AbstractProductB
ProductB1
ProductB2
«instantiate»

Example: Car Factory



https://www.tutorialspoint.com/design_pattern/design_pattern_quick_guide.htm



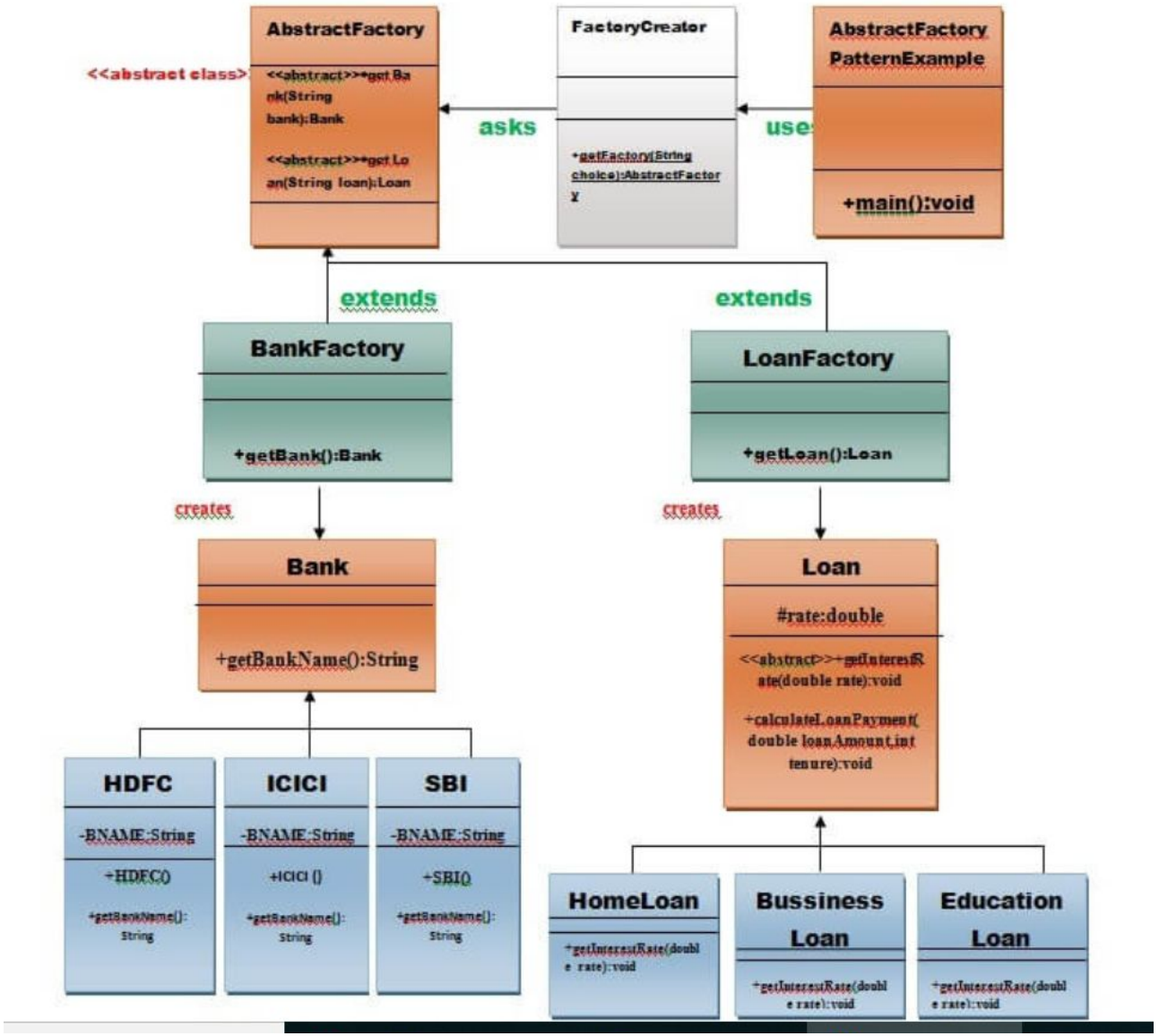
```java
public abstract class AbstractFactory {
   abstract Color getColor(String color);
   abstract Shape getShape(String shape) ;
}
```

```java
public class ShapeFactory extends AbstractFactory {

   @Override
   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 FactoryProducer {
   public static AbstractFactory getFactory(String choice){
      if(choice.equalsIgnoreCase("SHAPE")){
         return new ShapeFactory();
      } else if(choice.equalsIgnoreCase("COLOR")){
         return new ColorFactory();
      }
      return null;
   }
}
```

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

      //get shape factory
      AbstractFactory shapeFactory = FactoryProducer.getFactory("SHAPE"

      //get an object of Shape Circle
      Shape shape1 = shapeFactory.getShape("CIRCLE");

      //call draw method of Shape Circle
      shape1.draw();

      //get an object of Shape Rectangle
      Shape shape2 = shapeFactory.getShape("RECTANGLE");

      //call draw method of Shape Rectangle
      shape2.draw();
```

Abstract Factory



```java
import java.io.*;
interface Bank{
        String getBankName();
}
```

Step 2: Create concrete classes that implement the I

```java
class HDFC implements Bank{
        private final String BNAME;
        public HDFC(){
                BNAME="HDFC BANK";
        }
        public String getBankName() {
                return BNAME;
        }
}
```

```java
class ICICI implements Bank{
        private final String BNAME;
        ICICI(){
                BNAME="ICICI BANK";
        }
        public String getBankName() {
                return BNAME;
        }
}
```

```java
class SBI implements Bank{
        private final String BNAME;
        public SBI(){
                BNAME="SBI BANK";
        }
        public String getBankName(){
                return BNAME;
        }
}
```

```
abstract class Loan{
  protected double rate;
  abstract void getInterestRate(double rate);
  public void calculateLoanPayment(double loanamount, int years)
  {
      /*
          to calculate the monthly loan payment i.e. EMI

          rate=annual interest rate/12*100;
          n=number of monthly installments;
          1year=12 months.
          so, n=years*12;

        */

       double EMI;
       int n;

       n=years*12;
       rate=rate/1200;
       EMI=((rate*Math.pow((1+rate),n))/((Math.pow((1+rate),n))-1))*loanamount;

System.out.println("your monthly EMI is "+ EMI +" for the amount"+loanamount+" you have borrowed");
 }
}// end of the Loan abstract class.
```

```
class HomeLoan extends Loan{
    public void getInterestRate(double r){
        rate=r;
    }
}//End of the HomeLoan class.
```

```
class BussinessLoan extends Loan{
    public void getInterestRate(double r){
        rate=r;
    }

}//End of the BusssinessLoan class.
```

```
class EducationLoan extends Loan{
    public void getInterestRate(double r){
      rate=r;
 }
}//End of the EducationLoan class.
```

Step 5: Create an abstract class (i.e AbstractFactor

```
abstract class AbstractFactory{
  public abstract Bank getBank(String bank);
  public abstract Loan getLoan(String loan);
}
```

```
class BankFactory extends AbstractFactory{
   public Bank getBank(String bank){
      if(bank == null){
         return null;
      }
      if(bank.equalsIgnoreCase("HDFC")){
         return new HDFC();
      } else if(bank.equalsIgnoreCase("ICICI")){
         return new ICICI();
      } else if(bank.equalsIgnoreCase("SBI")){
         return new SBI();
      }
      return null;
   }
  public Loan getLoan(String loan) {
      return null;
   }
}//End of the BankFactory class.
```

```
class LoanFactory extends AbstractFactory{
        public Bank getBank(String bank){
            return null;
        }

    public Loan getLoan(String loan){
     if(loan == null){
       return null;
     }
     if(loan.equalsIgnoreCase("Home")){
       return new HomeLoan();
     } else if(loan.equalsIgnoreCase("Business")){
       return new BussinessLoan();
     } else if(loan.equalsIgnoreCase("Education")){
       return new EducationLoan();
     }
     return null;
   }

}
```

```
class FactoryCreator {
    public static AbstractFactory getFactory(String choice){
     if(choice.equalsIgnoreCase("Bank")){
       return new BankFactory();
     } else if(choice.equalsIgnoreCase("Loan")){
       return new LoanFactory();
     }
     return null;
   }
}//End of the FactoryCreator.
```

```
import java.io.*;
class AbstractFactoryPatternExample {
    public static void main(String args[])throws IOException {

    BufferedReader br=new BufferedReader(new InputStreamReader(System.in));

    System.out.print("Enter the name of Bank from where you want to take loan amount: ");
    String bankName=br.readLine();

System.out.print("\n");
System.out.print("Enter the type of loan e.g. home loan or business loan or education loan : ");

String loanName=br.readLine();
AbstractFactory bankFactory = FactoryCreator.getFactory("Bank");
Bank b=bankFactory.getBank(bankName);

System.out.print("\n");
System.out.print("Enter the interest rate for "+b.getBankName()+ ": ");

double rate=Double.parseDouble(br.readLine());
System.out.print("\n");
System.out.print("Enter the loan amount you want to take: ");

double loanAmount=Double.parseDouble(br.readLine());
System.out.print("\n");
System.out.print("Enter the number of years to pay your entire loan amount: ");
int years=Integer.parseInt(br.readLine());

System.out.print("\n");
System.out.println("you are taking the loan from "+ b.getBankName());

AbstractFactory loanFactory = FactoryCreator.getFactory("Loan");
        Loan l=loanFactory.getLoan(loanName);
        l.getInterestRate(rate);
        l.calculateLoanPayment(loanAmount,years);
 }
```

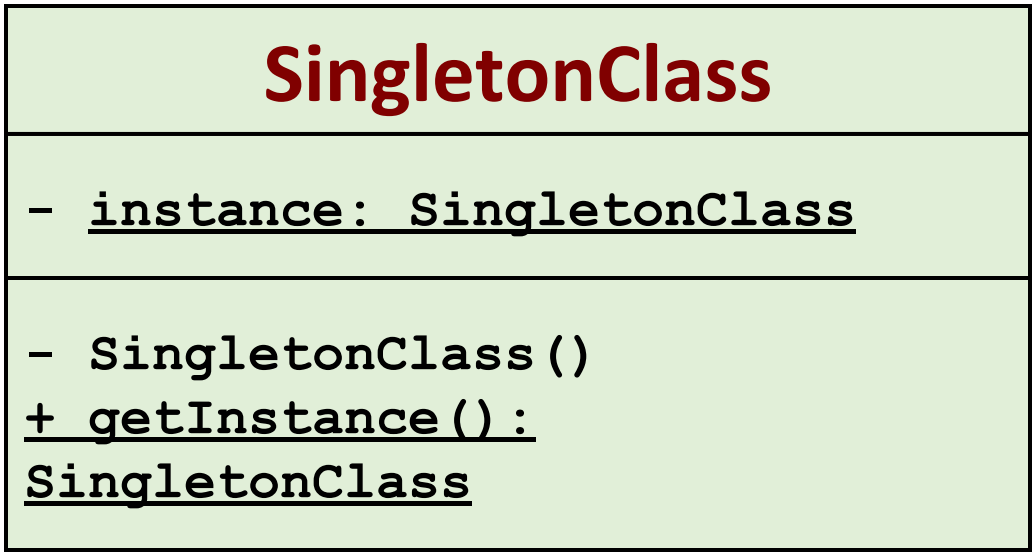
Consequences

- Isolates the concrete class
- Makes exchanging product family easy
- Promotes consistency among products
- Abstract Factory design pattern provides approach to code for interface rather than implementation.
- Abstract Factory pattern is “factory of factories” and can be easily extended to accommodate more products.
- Abstract Factory pattern is robust and avoid conditional logic of Factory pattern.
- Supporting new types of products is difficult

Singleton

Intent	• Ensure a class has only one instance, and provide a global point of access to it. • Encapsulated "just-in-time initialization" or "initialization on first use".
Problem	• Application needs one, and only one, instance of an object. Additionally, lazy initialization and global access are necessary.

Solution


SingletonClass
- instance: SingletonClass
- SingletonClass()
+ getInstance():
SingletonClass

Example: DB Connection Manger

```
public class DbConnection{

    private static DbConnection instance=null;
    private SQLConnection connection;

    private DbConnection() {
     connection = connectToDatabase(dbUser,dbPassword,dbName);
   }

    public static getDbConnection() {
      if (instance== null )
       instance = new DbConnection() ;

      return instance;
    }
}
```

```
DbConnection connection=DbConnection.getDbConnection();
```

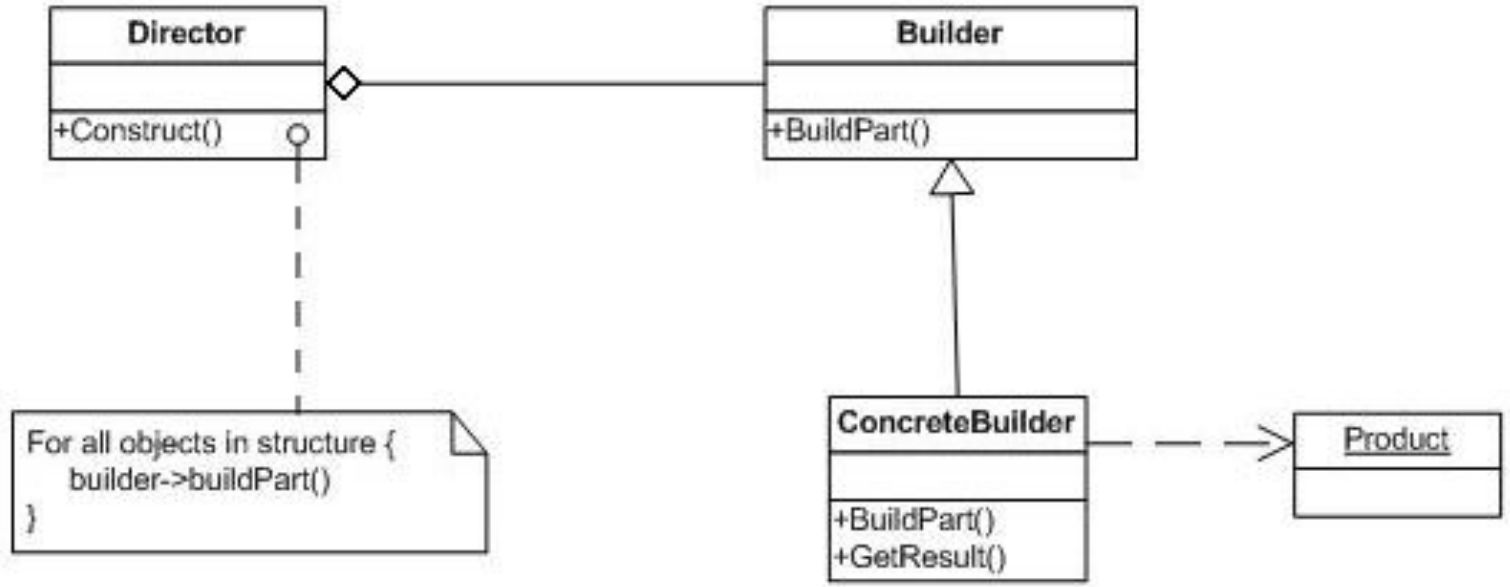
Consequences

- Controlled access to sole instance
- Reduced namespace
- Permits variable number of instances

Builder

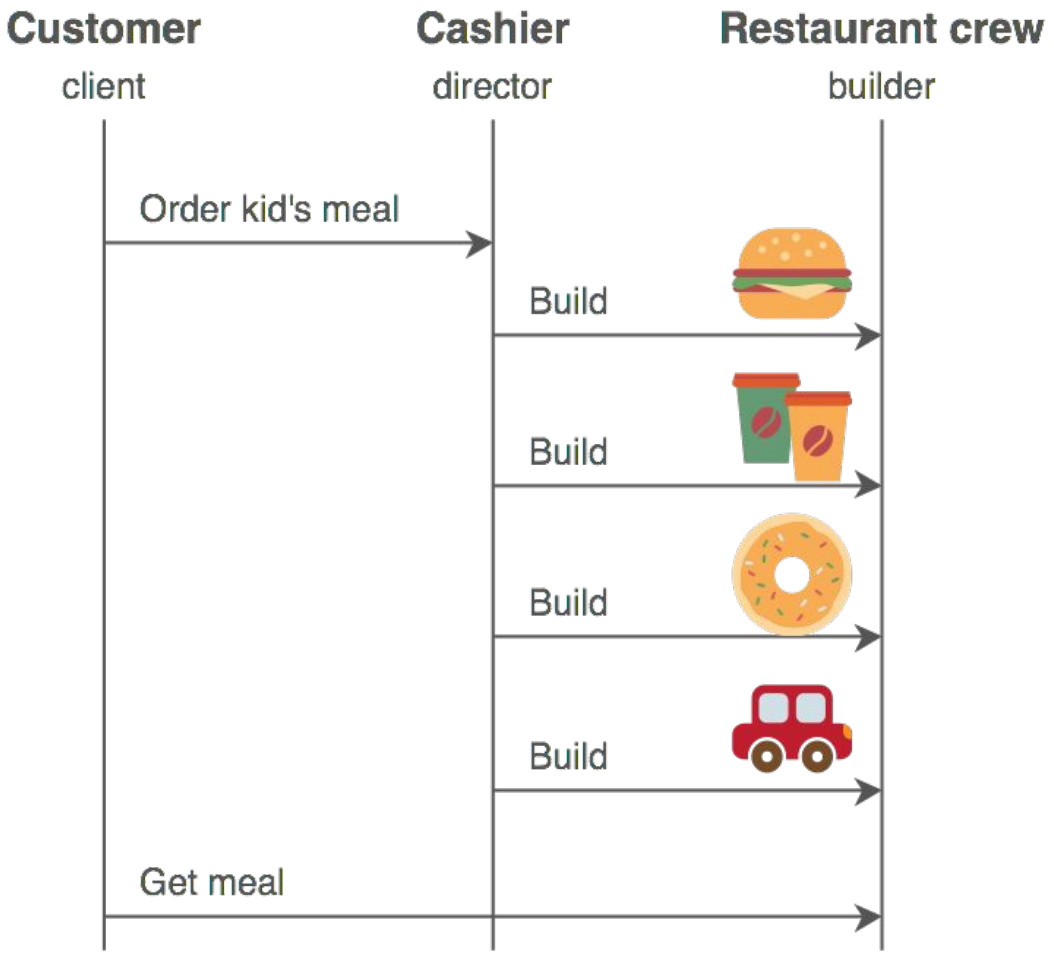
Intent	• Separate the construction of a complex object from its representation so that the same construction process can create different representations. • Parse a complex representation, create one of several targets.
Problem	• An application needs to create the elements of a complex aggregate. The specification for the aggregate exists on secondary storage and one of many representations needs to be built in primary storage.

Solution


Director
+Construct()
Builder
+BuildPart()
For all objects in structure {
builder->buildPart()
}
ConcreteBuilder
+BuildPart()
+GetResult()
Product

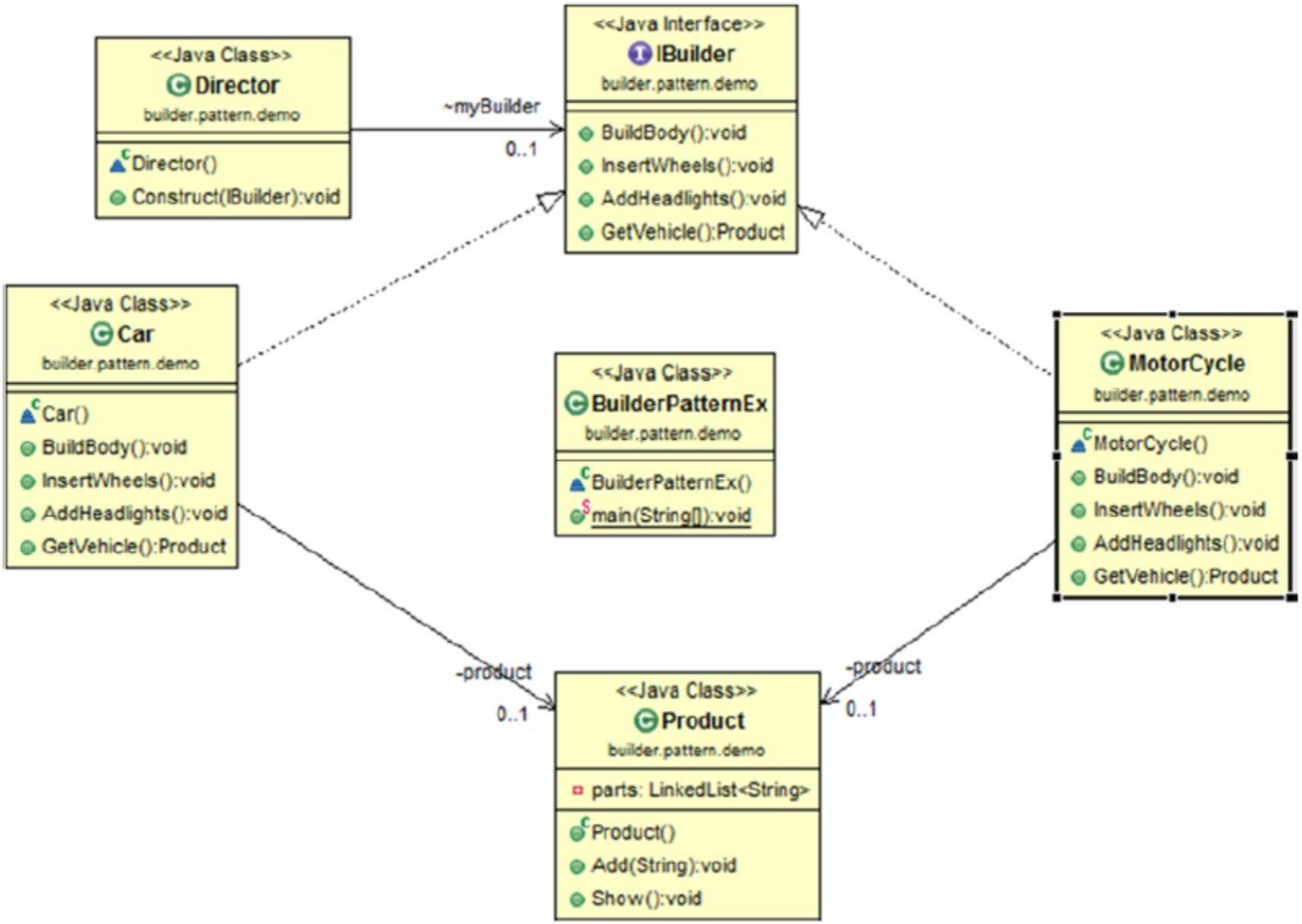
Examples

To create a computer, different parts are assembled depending upon the order received by the customer (e.g., a customer can demand a 500 GB hard disk with an Intel processor; another customer can choose a 250 GB hard disk with an AMD processor).



Consequences

- Lets you vary a product's internal representation
- Isolates construction and representation
- Gives you finer control over the construction process


<<Java Class>>
Director
builder.pattern.demo
Director()
Construct(IBuilder):void
~myBuilder
0..1
<<Java Interface>>
IBuilder
builder.pattern.demo
BuildBody():void
InsertWheels():void
AddHeadlights():void
GetVehicle():Product
<<Java Class>>
Car
builder.pattern.demo
Car()
BuildBody():void
InsertWheels():void
AddHeadlights():void
GetVehicle():Product
<<Java Class>>
BuilderPatternEx
builder.pattern.demo
BuilderPatternEx()
main(String[]):void
<<Java Class>>
MotorCycle
builder.pattern.demo
MotorCycle()
BuildBody():void
InsertWheels():void
AddHeadlights():void
GetVehicle():Product
-product
0..1
-product
0..1
<<Java Class>>
Product
builder.pattern.demo
parts: LinkedList<String>
Product()
Add(String):void
Show():void

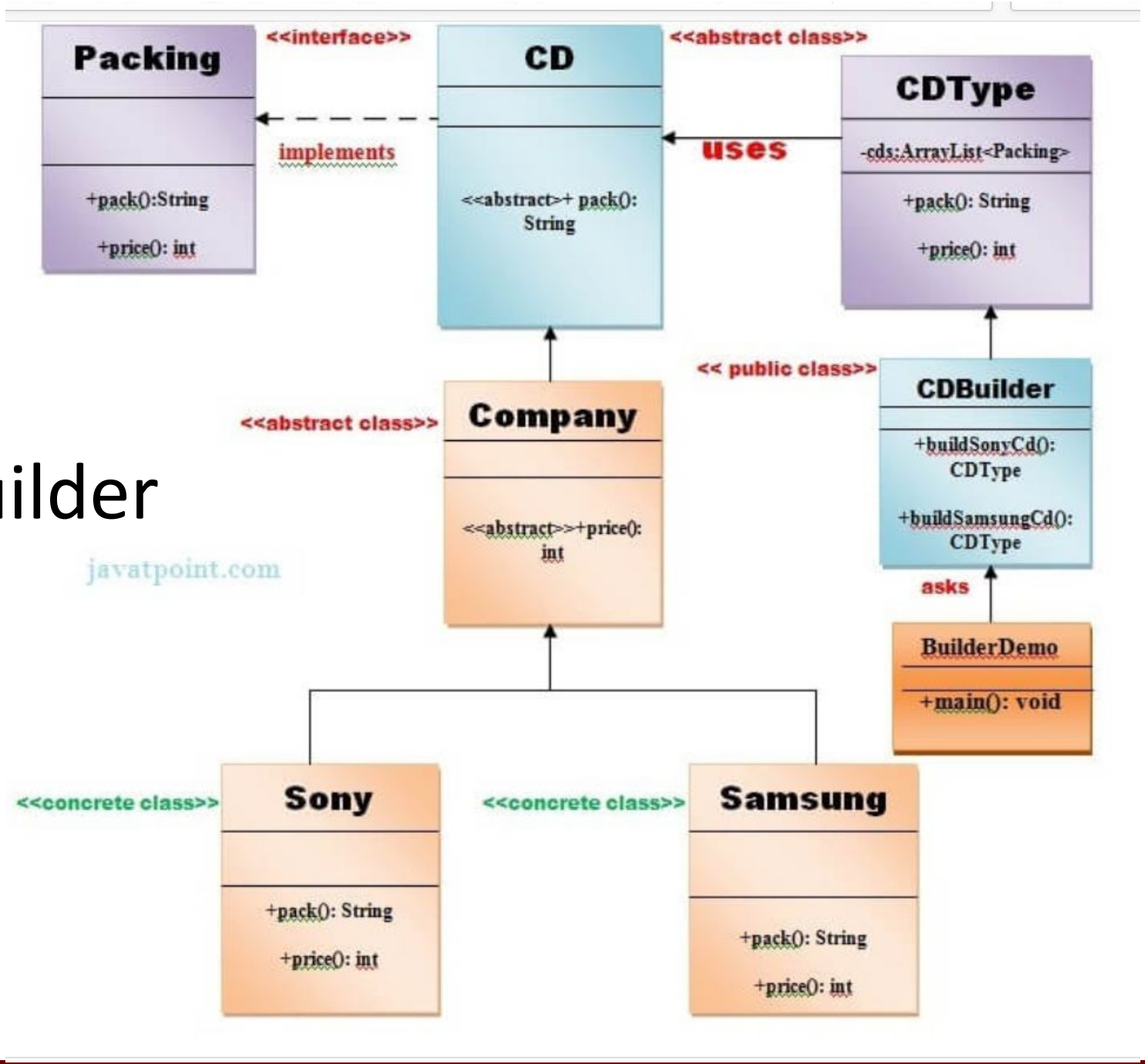
```
// Builders common interface
interface IBuilder
{
        void BuildBody();
        void InsertWheels();
        void AddHeadlights();
        Product GetVehicle();

}

// Car is ConcreteBuilder
class Car implements IBuilder
{
        private Product product = new Product();

        @Override
        public void BuildBody()
        {
                product.Add("This is a body of a Car");
        }
        @Override
        public void InsertWheels()
        {
                product.Add("4 wheels are added");
        }
        @Override
        public void AddHeadlights()
        {
                product.Add("2 Headlights are added");
        }
        public  Product GetVehicle()
        {
                return product;
        }

}
```

```
// "Product"
class Product
{
        // We can use any data structure that you prefer. We have used
        private LinkedList<String> parts;
        public Product()
        {
                parts = new LinkedList<String>();
        }

        public void Add(String part)
        {
                //Adding parts
                parts.addLast(part);
        }

        public void Show()
        {
                System.out.println("\n Product completed as below :");
                for(int i=0;i<parts.size();i++)
                {
                        System.out.println(parts.get(i));
                }
        }
}
```

```
// "Director"
class Director
{
        IBuilder myBuilder;

        // A series of steps-for the production
        public void Construct(IBuilder builder)
        {
                myBuilder=builder;
                myBuilder.BuildBody();
                myBuilder.InsertWheels();
                myBuilder.AddHeadlights();
        }
}
```

```
class BuilderPatternEx
{
        public static void main(String[] args)
        {
                System.out.println("***Builder Pattern Demo***\n");

                Director director = new Director();

                IBuilder carBuilder = new Car();
                IBuilder motorBuilder = new MotorCycle();

                // Making Car
                director.Construct(carBuilder);
                Product p1 = carBuilder.GetVehicle();
                p1.Show();

                //Making MotorCycle
                director.Construct(motorBuilder);
                Product p2 = motorBuilder.GetVehicle();
                p2.Show();
        }
}
```

Builder



File: Packing.java

```
public interface Packing {
        public String pack();
        public int price();
}
```

2) Create 2 abstract classes CD and Company

Create an abstract class CD which will implement Packing interface.

File: CD.java

```
public abstract class CD implements Packing{
public abstract String pack();
}
```

File: Company.java

```
public abstract class Company extends CD{
  public abstract int price();
}
```

File: Sony.java

```
public class Sony extends Company{
    @Override
        public int price(){
                return 20;
     }
    @Override
    public String pack(){
            return "Sony CD";
        }
}//End of the Sony class.
```

File: Samsung.java

```
public class Samsung extends Company {
    @Override
        public int price(){
                return 15;
    }
    @Override
    public String pack(){
            return "Samsung CD";
        }
}//End of the Samsung class.
```

File: CDType.java

```
import java.util.ArrayList;
import java.util.List;
public class CDType {
        private List<Packing> items=new ArrayList<Packing>();
        public void addItem(Packing packs) {
              items.add(packs);
        }
        public void getCost(){
         for (Packing packs : items) {
                packs.price();
         }
        }
        public void showItems(){
         for (Packing packing : items){
        System.out.print("CD name : "+packing.pack());
        System.out.println(", Price : "+packing.price());
       }
         }
}//End of the CDType class.
```

File: CDBuilder.java

```
public class CDBuilder {
                public CDType buildSonyCD(){
                  CDType cds=new CDType();
                  cds.addItem(new Sony());
                  return cds;
               }
               public CDType buildSamsungCD(){
              CDType cds=new CDType();
              cds.addItem(new Samsung());
              return cds;
               }
}// End of the CDBuilder class.
```

6) Create the BuilderDemo class

File: BuilderDemo.java

```
public class BuilderDemo{
 public static void main(String args[]){
   CDBuilder cdBuilder=new CDBuilder();
   CDType cdType1=cdBuilder.buildSonyCD();
   cdType1.showItems();

   CDType cdType2=cdBuilder.buildSamsungCD();
   cdType2.showItems();
 }
}
```