Software Architecture

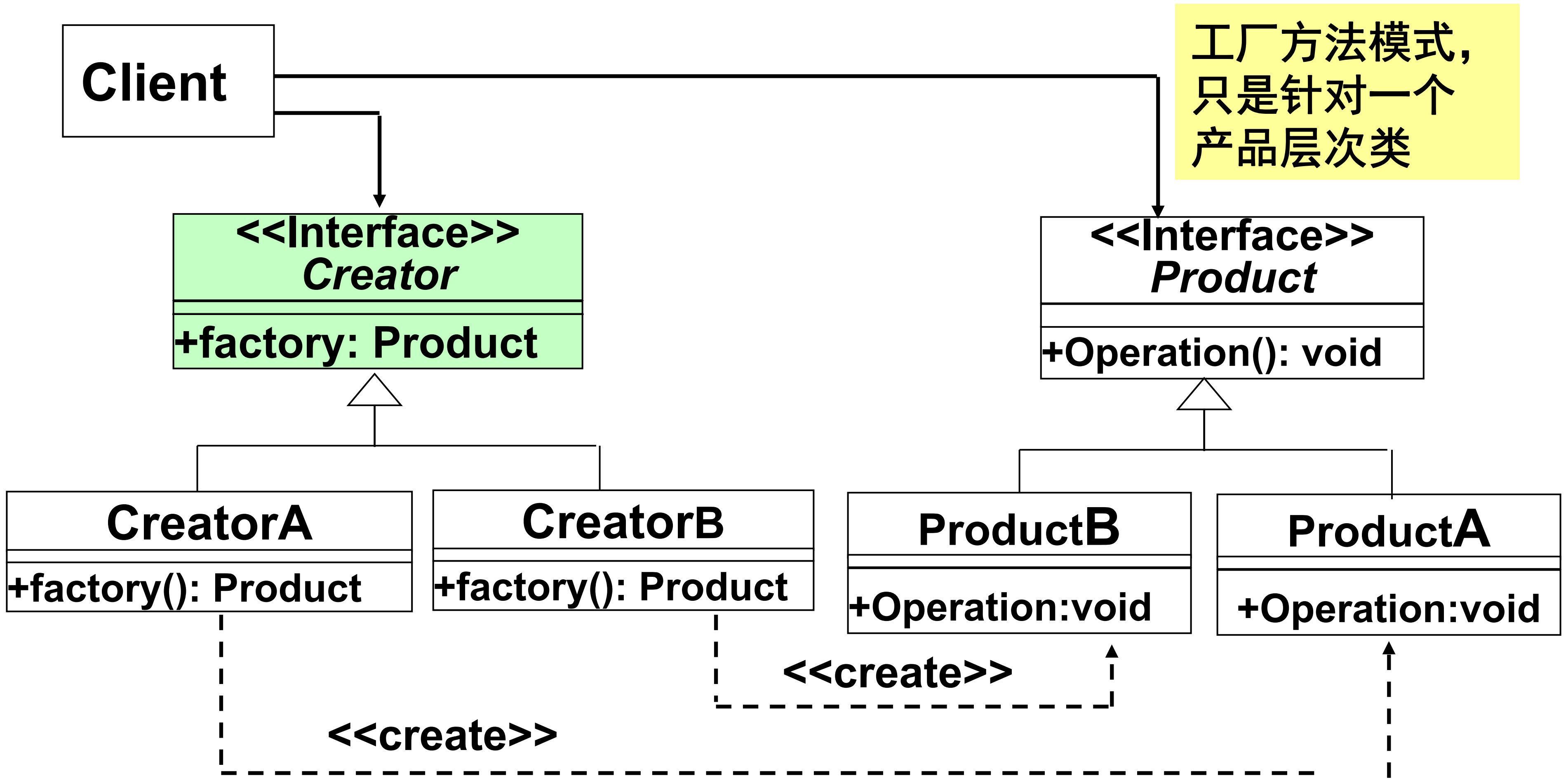
Lecture 2: Abstract Factory Pattern
抽象工厂模式

Fall 2020
Professor Yushan Sun

Contents of this lecture

Introduction of the abstract factory pattern
抽象工厂模式的引入

Review of Factory Method Pattern

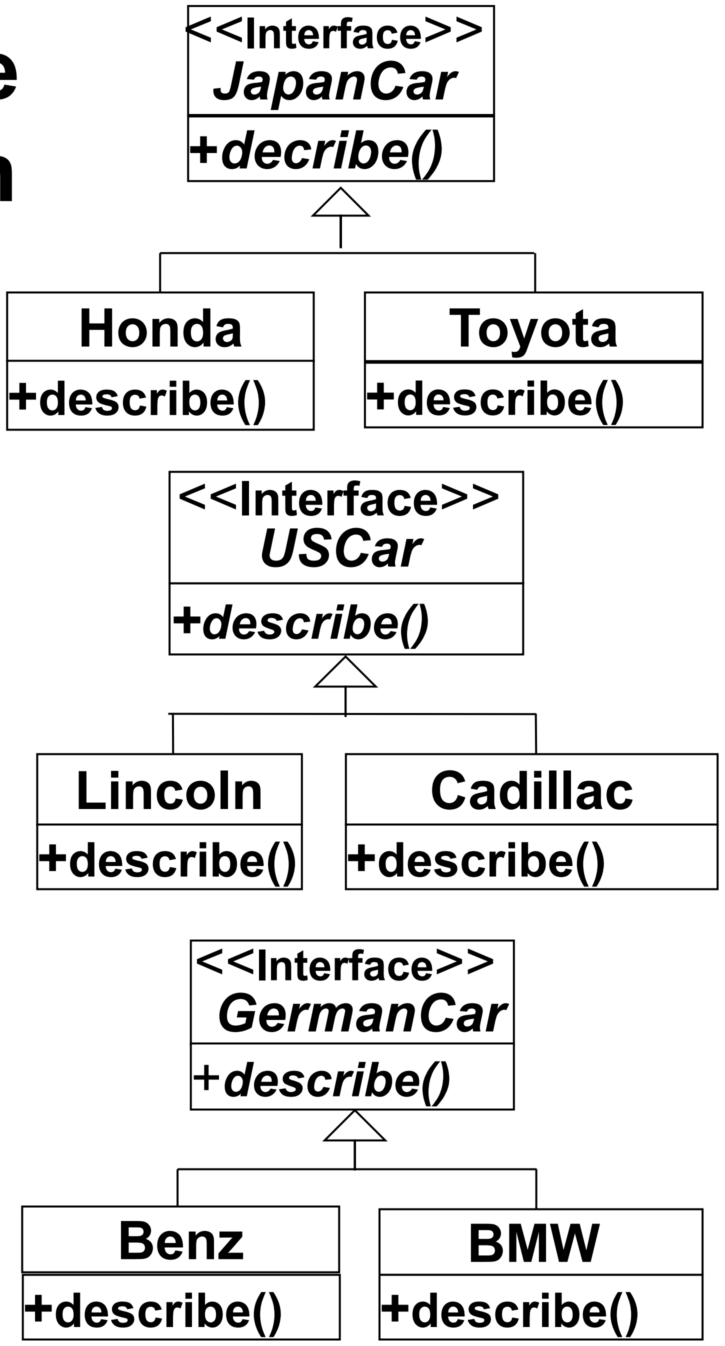


Factory method pattern: creator class hierarchy has the same structure as the product hierarchy does

Leading examples to the Abstract Factory pattern

- 现在假设有多个结构相同的产品层次类
- Suppose that we have a family of products with the same structure.

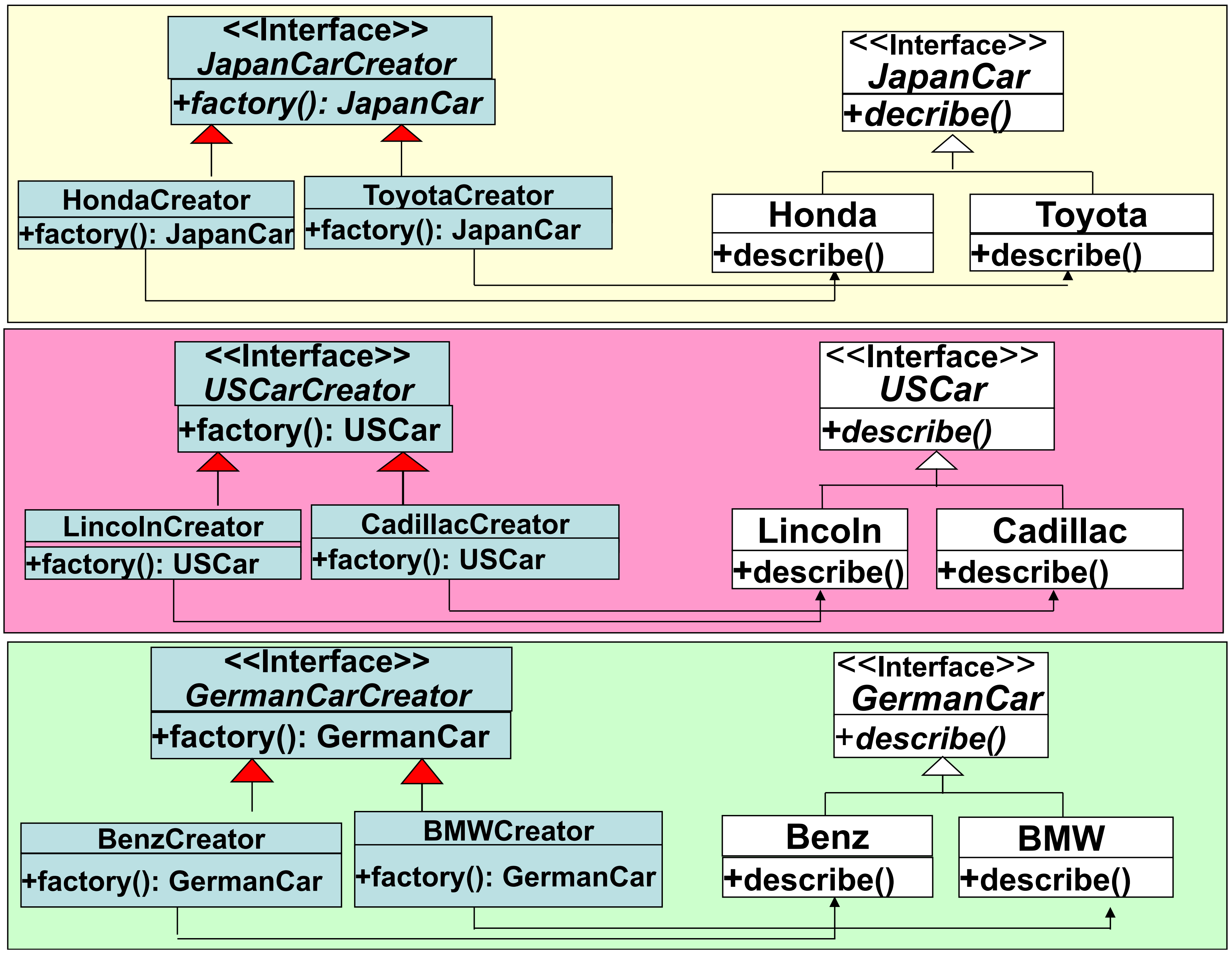
问题：我们是否仍然使用工厂方法模式？
Question:
Can we still use the factory method pattern?
How?



Leading examples to the Abstract Factory pattern

Somebody’s Answer:

- **for each product, we construct a Creator class.**
- **see the next page for the three creators used**



<<Interface>>
JapanCarCreator
+factory(): JapanCar
HondaCreator
+factory(): JapanCar
ToyotaCreator
+factory(): JapanCar
<<Interface>>
JapanCar
+decribe()
Honda
+describe()
Toyota
+describe()
<<Interface>>
USCarCreator
+factory(): USCar
LincolnCreator
+factory(): USCar
CadillacCreator
+factory(): USCar
<<Interface>>
USCar
+describe()
Lincoln
+describe()
Cadillac
+describe()
<<Interface>>
GermanCarCreator
+factory(): GermanCar
BenzCreator
+factory(): GermanCar
BMWCreator
+factory(): GermanCar
<<Interface>>
GermanCar
+describe()
Benz
+describe()
BMW
+describe()

Leading examples to the Abstract Factory pattern

- **Comment: This is not good because there are too many Creator classes.**

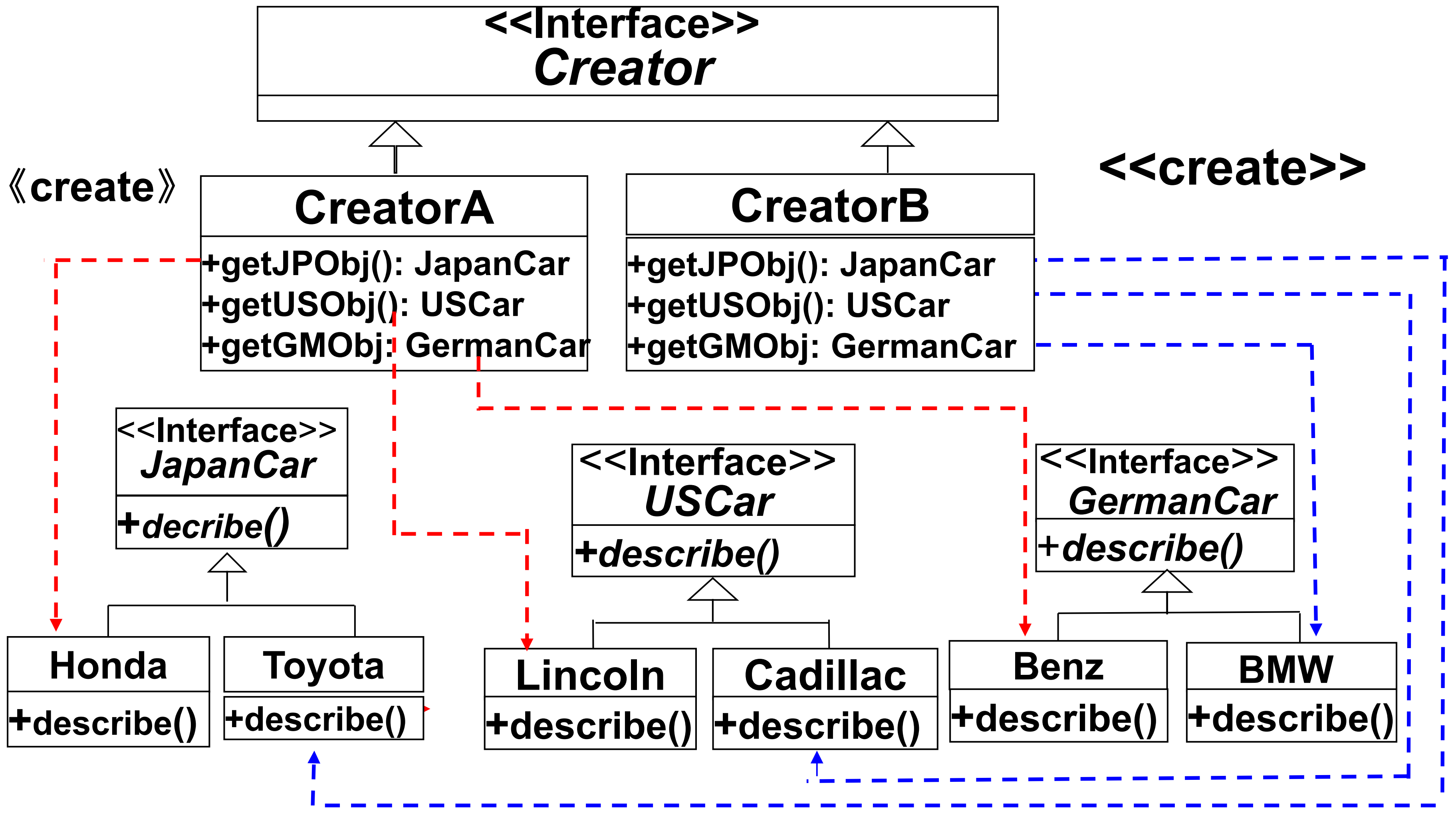
- **Any better solutions?**

Leading examples to the Abstract Factory pattern

Pay attention to the fact that products below have the same structure,

We get the solution as below in the next page.

Leading examples to the Abstract Factory pattern

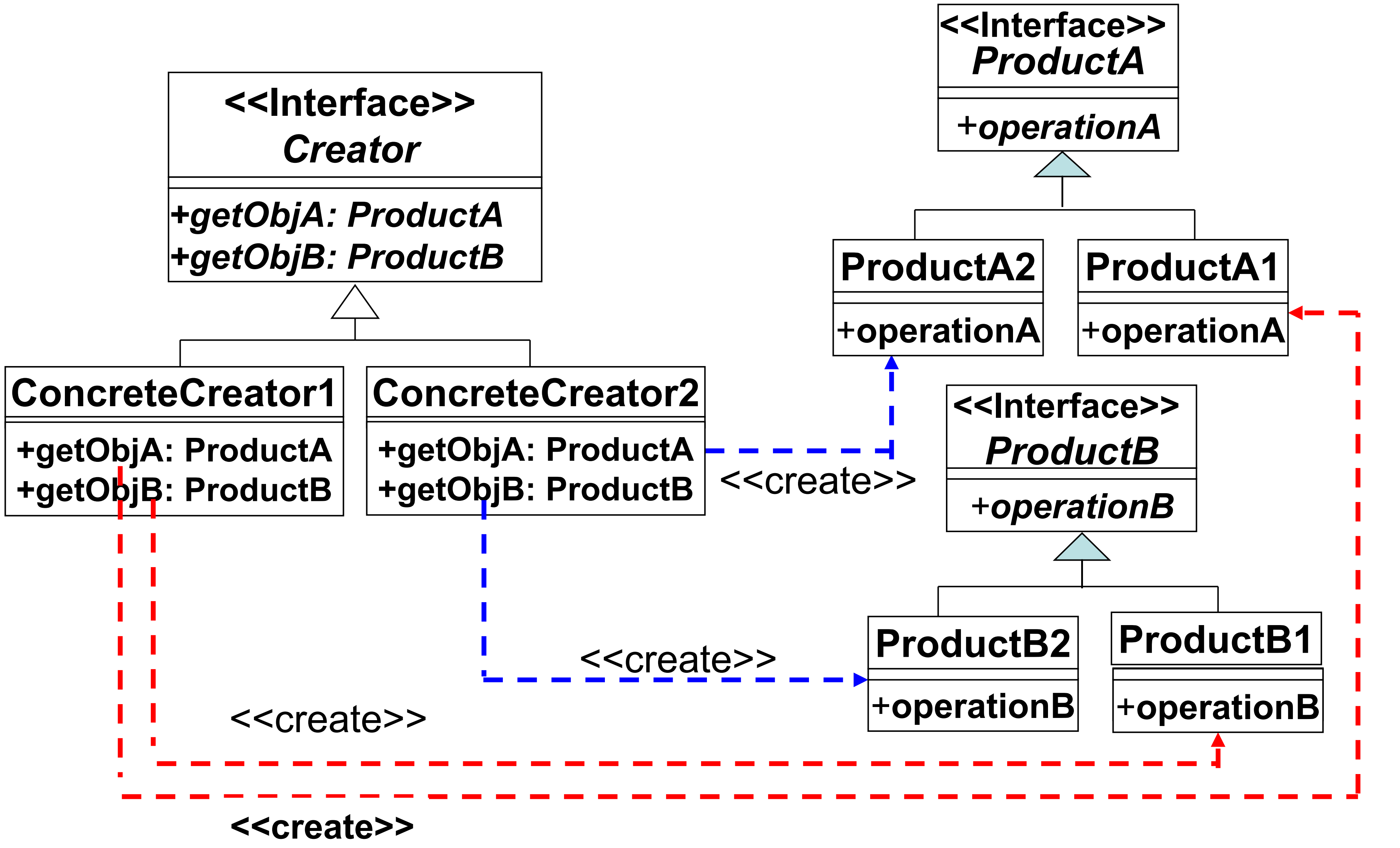


Two concrete factory classes, each is responsible for creating 3

Back

Theory of the abstract factory pattern
抽象工厂模式理论

Abstract Factory Pattern



Abstract Factory Pattern

Abstract Factory Pattern

- The Abstract Factory pattern takes the same concept as the Factory method pattern does.
- An *abstract factory* *Creator* is a class that provides an interface to produce a family of objects.
- In the Java programming language, it can be implemented either as
  - an interface or as
  - an abstract class.

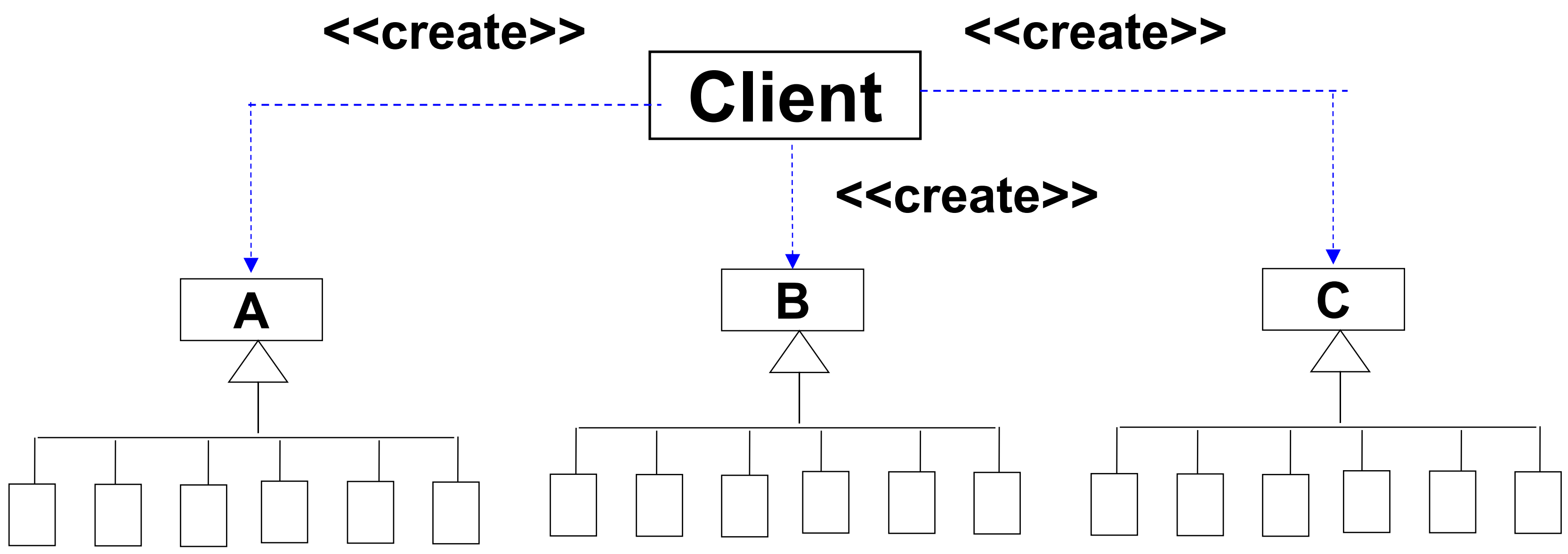
Abstract Factory Pattern

抽象工厂模式的组件
Components in abstract factory pattern

– Families of related, dependent product classes.
– A group of concrete factory classes that implement the interface provided by the abstract factory class.
– Each of these factories provides access to a particular suite of related, dependent objects and implements the abstract factory interface in a manner that is specific to the family of classes it controls.

When to use Abstract Factory Pattern?

- 何时使用抽象工厂模式？What is the situation to use abstract factory pattern?
- Use the Abstract Factory pattern when a client object wants to create an instance of one of a suite of related, dependent classes without having to know which specific concrete class is to be instantiated.



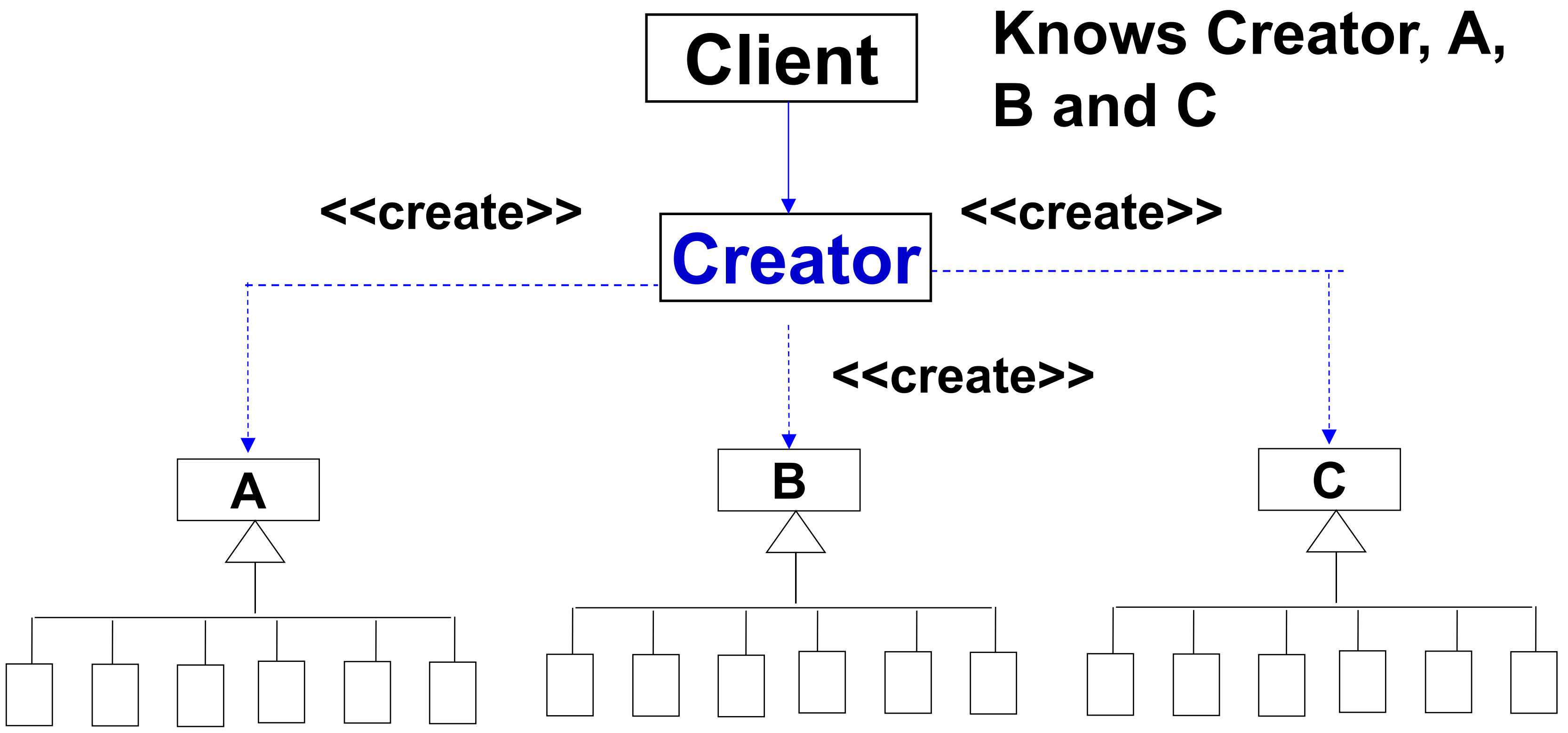
When to use Abstract Factory Pattern?

- 如果不使用抽象工厂模式，将会发生什么?
- What will happen if we don’t use the abstract factory pattern?
- If we don’t use abstract factory, the required implementation to select an appropriate class needs to be present everywhere such an instance is created.
- Need to use a lot of conditional statements.

Abstract Factory Pattern

How a client program uses the abstract factory pattern?

- Client objects make use of these concrete factories to create objects and, therefore, do not need to know which concrete class is actually instantiated.



Difference between factory method pattern and abstract factory pattern

工厂方法模式与抽象工厂模式的区别
Differences between factory method pattern and abstract factory pattern

1. 产品不同 (The “Products” are different)
   - For factory method pattern, the product is a single product class hierarchy;
   - For abstract factory pattern, the product is a group of product class hierarchies.

Difference between factory method pattern and abstract factory pattern

2. 有关可扩展性 (About extensibility)
   - The factory method pattern follows open-closed principle, while
   - the abstract factory pattern only partially follow the open-closed principle (see next page)

Difference between factory method pattern and abstract factory pattern

两种情况 (Two situations):

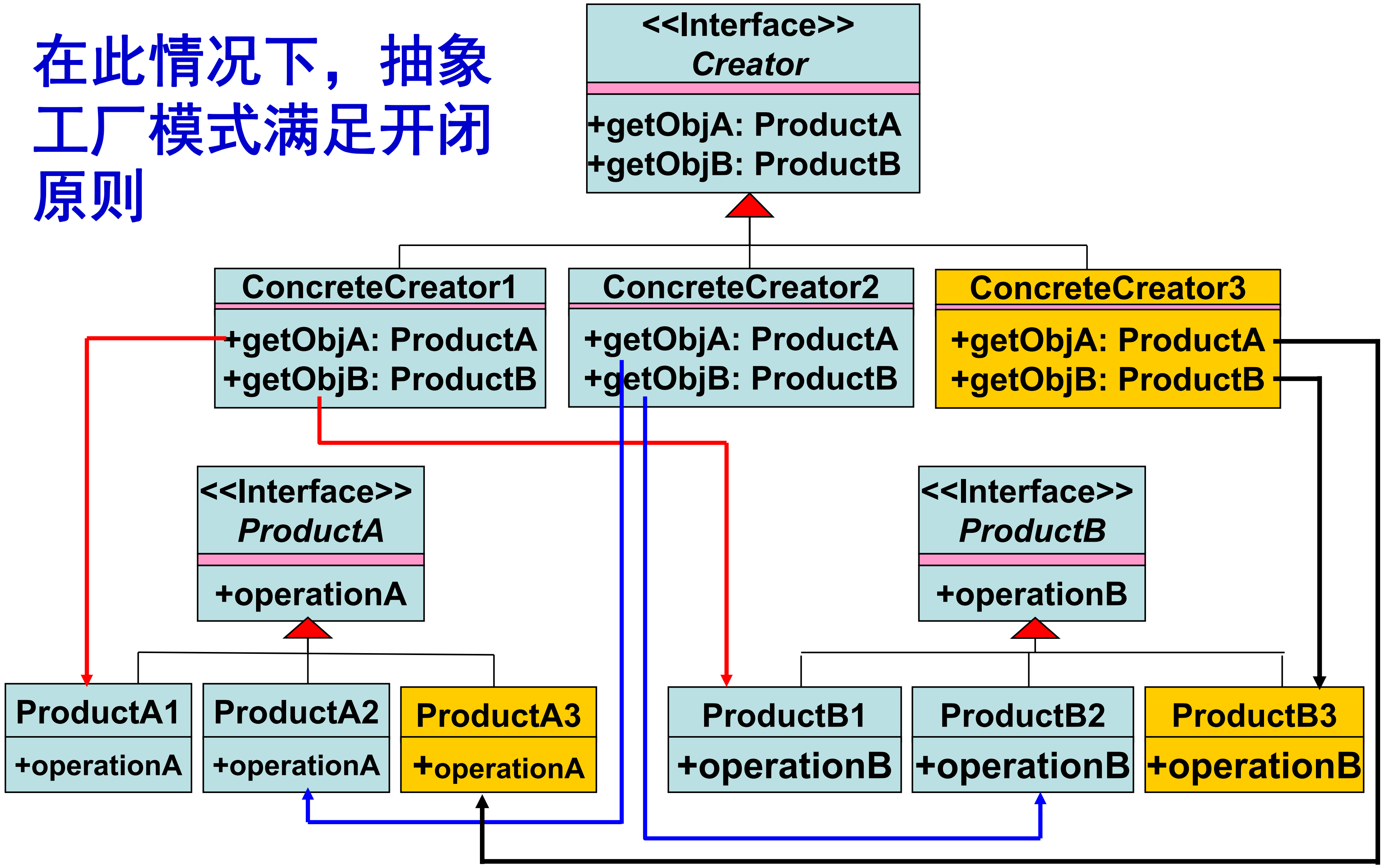
Case 1: In case If you add a ProductA3 and ProductB3, the *Creator class hierarchy also needs to add a class* CreatorC

在此情况下，抽象工厂模式满足开闭原则

In this case, the abstract factory pattern follows the open-closed principle.

Difference between factory method pattern and abstract factory pattern

在此情况下，抽象工厂模式满足开闭原则



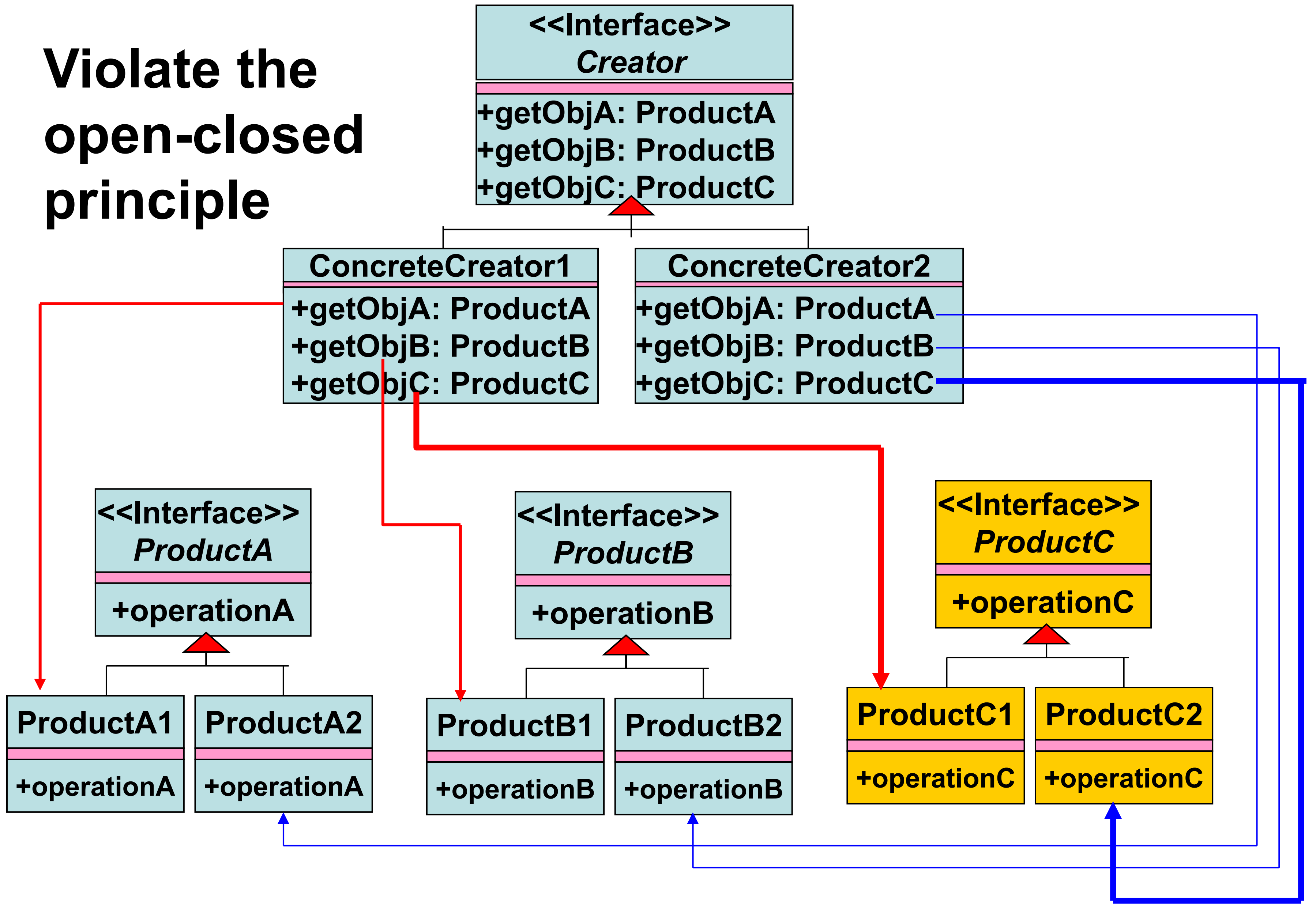
If you don’t want to add a product class, but only add a concrete product in the existing product hierarchies, this pattern follows open-closed

Case 2: In case If you add a new product hierarchy *ProductC*, then you need to add a new method “**+getObjC: ProductC**” in

- **ConcreteCreator1, and**
- **ConcreteCreator2**

- This means that you need to modify the existing factory class hierarchy, and so **it doesn’t follow the open-closed principle**

Violate the open-closed principle



When you want to add a whole new class hierarchy productC, this pattern doesn’ t follow open-closed principle, since in both Concrete creator classes, you need to add getObjC method

Back

Examples with Java code

Design example using the abstract factory pattern

Query Software of Building Features

- **Example of Abstract Factory Pattern**
- Let us design a web application program to query the features of different types of buildings .
- Consider two types of buildings: house and, condo

House

Condo

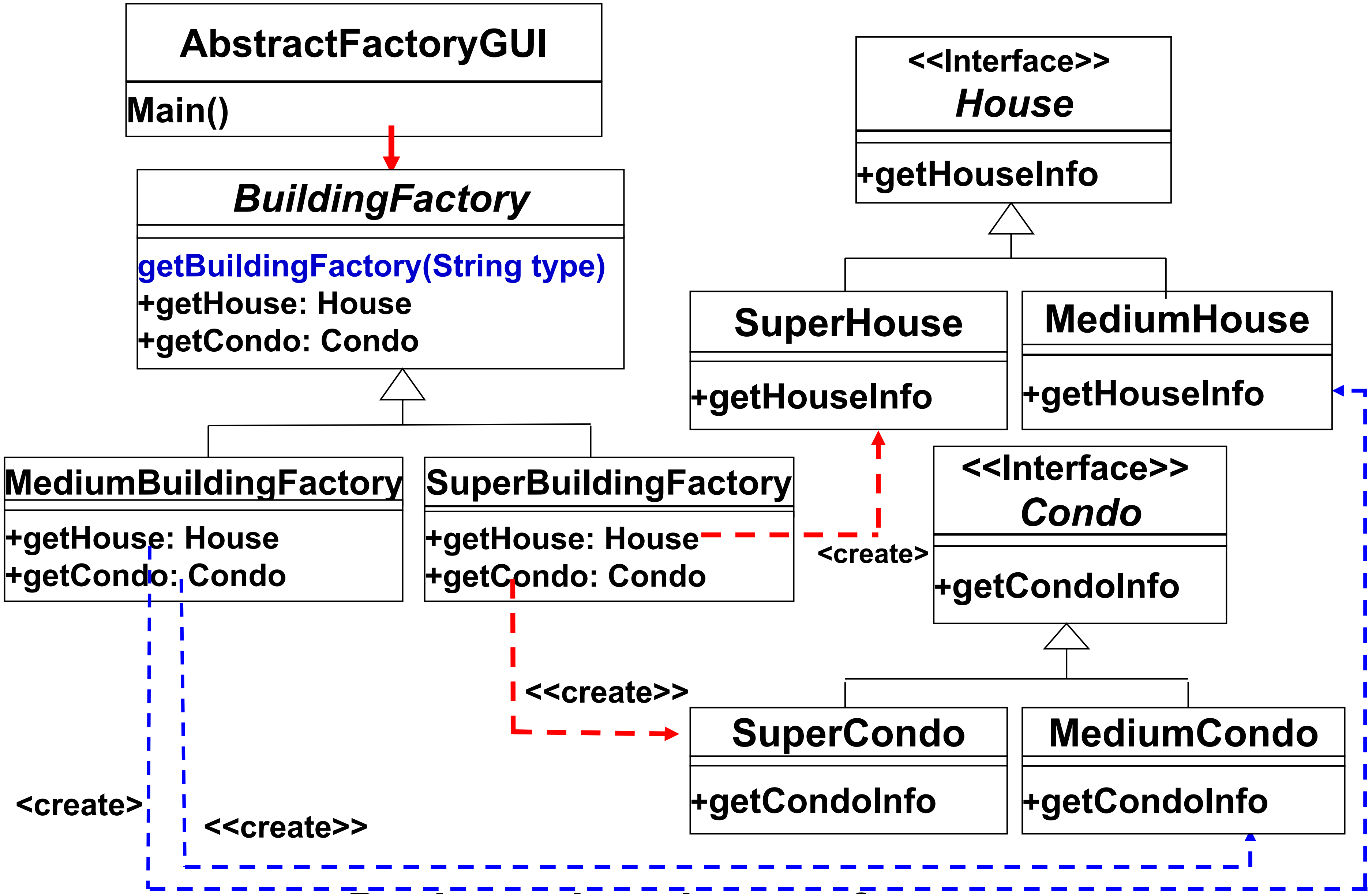
Semi detacher

Query Software of Building Features

- Further, a building can be of either
  - Super or
  - Medium

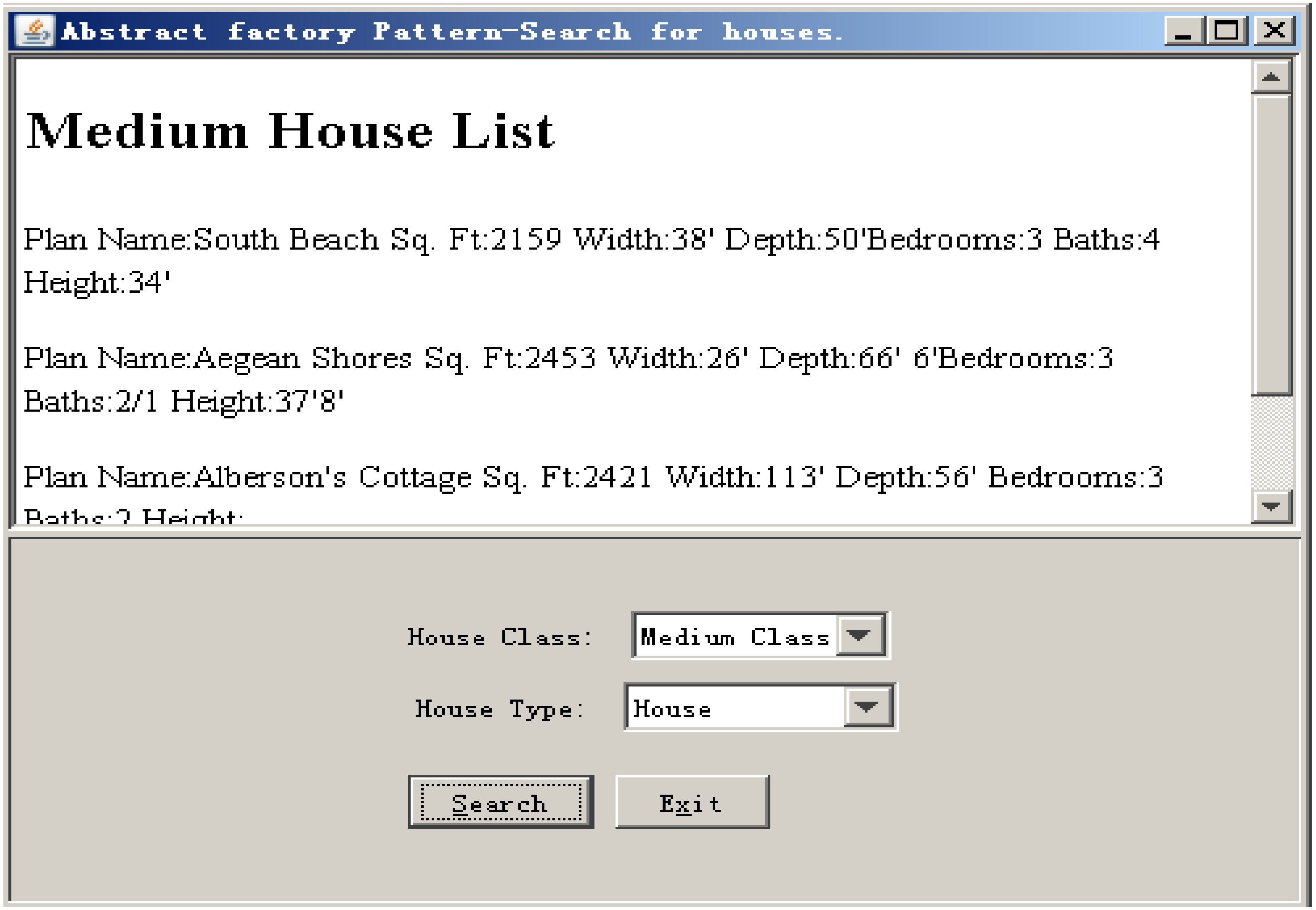
  category.
- Because House and Condo represent two kinds of products and each can be classified as super and medium, we can use abstract factory pattern as below.

Query Software of Building Features



Design using abstract factory pattern

Query Software of Building Features


Abstract factory Pattern-Search for houses.

Medium House List

Plan Name:South Beach Sq. Ft:2159 Width:38' Depth:50'Bedrooms:3 Baths:4 Height:34'

Plan Name:Aegean Shores Sq. Ft:2453 Width:26' Depth:66' 6'Bedrooms:3 Baths:2/1 Height:37'8'

Plan Name:Alberson's Cottage Sq. Ft:2421 Width:113' Depth:56' Bedrooms:3 Baths:2 Height:

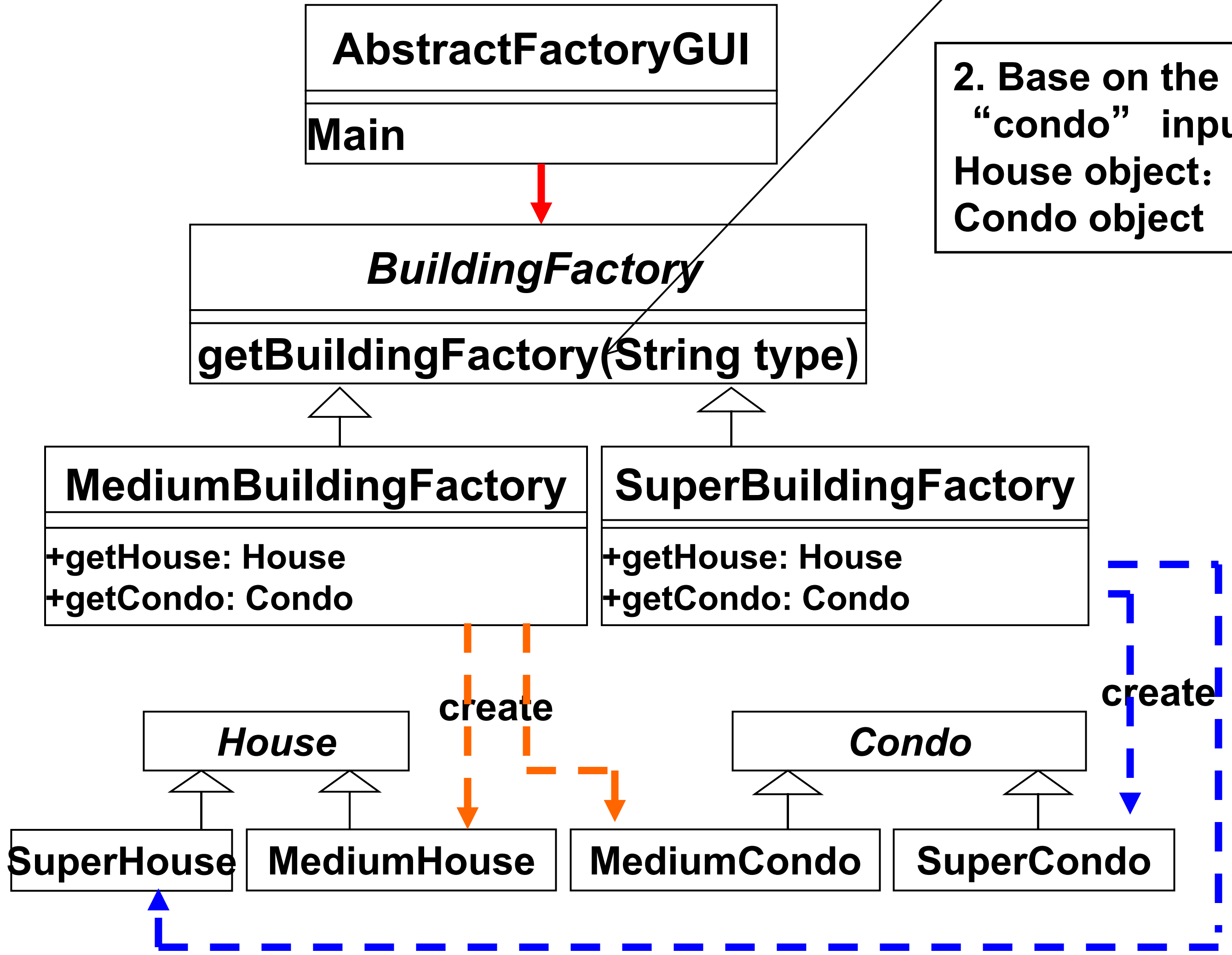
House Class: Medium Class

House Type: House

The graphical user interface of Query Software of Building Features

Query Software of Building Features

How the design works?



1. From GUI, input parameter " super" or " medium" , and call BuildingFactory to get a MediumBuildingFactory or SuperBuildingFactory object bf

2. Base on the parameter "house" or "condo" input from GUI, to get House object: h = bf.getHouse(), or Condo object c = bf. getCondo()

3. Using h or c to get h.getHouseInfo() c.getCondoInfo() information

Note: AbstractFactoryGUI only knows interface House or Condo but doesn't have to know their subclasses

Query Software of Building Features

Design reason:

- If we don’ t use the abstract factory pattern, and just call the methods in classes House and Condo directly,
- then in the ClientGUI, we need to write a lot of conditional statements and this is not good for extension and maintenance

Query Software of Building Features

- **Benefits of the design**:
- **Responsibility separation**:
- The responsibility of selecting and instantiating an appropriate House/Condo implementer can be moved out of application objects to a separate designated abstract factory class.

- **About BuildingFactory class hierarchy**:
- it simply declares the required interface, leaving the actual implementation details of class selection and instantiation to its implementers.

Query Software of Building Features

- **About client object AbstractFactoryGUI class**:
- It uses an instance of a concrete factory that implements the *BuildingFactory* interface to create objects representing houses of different types and categories without having to know the actual concrete class that needs to be instantiated.
- The client objects do not have to know about the existence of these concrete factory classes.

Query Software of Building Features

How AbstractFactoryGUI works?

1. When the Search button is clicked after a building category and type combination is selected, the client AbstractFactoryGUI invokes the static getBuildingFactory(String type) method on the abstract *BuildingFactory* class.
2. The getBuildingFactory(String type) method creates an appropriate factory object and returns it as an object of *BuildingFactory* type.

Query Software of Building Features

3. The factory object internally creates an instance of an appropriate class from among the family of buildings it controls (SuperHouse/SuperCondo or MediumHouse/MediumCondo) and returns it as an object of House/Condo type.
4. **Client class AbstractFactoryGUI even don’t know the existence of class** House/Condo

Query Software of Building Features

- The client AbstractFactoryGUI does not need to know the existence of different concrete House/Condo classes.
- It simply invokes methods declared by the House/Condo interface such as
  - **getHouseInfo or**
  - **getCondoInfo.**

Source Code for the Building Information System

```
public interface House {
     public String getHouseInfo();
}
```

```
public class SuperHouse implements House {
     public String getHouseInfo() {
        return "superHouse.html";
     }
}
```

```
public class MediumHouse implements House{
     public String getHouseInfo() {
        return "mediumHouse.html";
     }
}
```

Source Code for the Building Information System

```
public interface Condo {
      public String getCondoInfo();
}
```

```
public class SuperCondo implements Condo{
       public String getCondoInfo(){
          return "superCondo.html";
       }
}
```

```
public class MediumCondo implements Condo{
       public String getCondoInfo(){
          return "mediumCondo.html";
       }
}
```

Source Code for the Building Information System

```
public abstract class BuildingFactory{
   public static final String SUPER = "Super Class";
   public static final String MEDIUM = "Medium Class";
   public abstract House getHouse();
   public abstract Condo getCondo();

   public static BuildingFactory
   getBuildingFactory(String type){
         BuildingFactory bf = null;
         if (type.equals(BuildingFactory.SUPER)){
               bf = new SuperBuildingFactory();
         }
         else if (type.equals(BuildingFactory.MEDIUM)){
               bf = new MediumBuildingFactory();
         }
       return bf;
   }
}
```

Source Code for the Building Information System

```
public class MediumBuildingFactory extends
                                        BuildingFactory {

  public House getHouse(){
    return new MediumHouse();
  }
  public Condo getCondo(){
    return new MediumCondo();
  }
}
```

Source Code for the Building Information System

```
public class SuperBuildingFactory extends
                                    BuildingFactory {

  public House getHouse(){
    return new SuperHouse();
  }
  public Condo getCondo(){
    return new SuperCondo();
  }
}
```

Source Code for the Building Information System

```
public class AbstractFactoryGUI extends JFrame {
    class ButtonListener implements ActionListener {
    public void actionPerformed(ActionEvent ae) {
        if (ae.getActionCommand().equals(AbstractFactoryGUI.SEARCH))
   {

            String clas = (String) cmbHouseClass.getSelectedItem();
            String type = (String) cmbHouseType.getSelectedItem();

            BuildingFactory bf = BuildingFactory.getBuildingFactory(clas);

            if (type.equals(AbstractFactoryGUI.HOUSE)) {
                House hs = bf.getHouse();
                String fileNm = hs.getHouseInfo();
                putHouseInfoToScreen(fileNm);
            }
            if (type.equals(AbstractFactoryGUI.CONDO)) {
                Condo cd = bf.getCondo();
                String fileNm = cd.getCondoInfo();
                putHouseInfoToScreen(fileNm);
             }
          }
}}
```

Back