Factory Method Pattern

Abstract Factory Pattern

Contents

- Motivation of creational patterns
- Intro. to the Pizza Business
- Simple Factory (Not official)
- Factory Method Pattern
- Abstract Factory Method Pattern

Creating Objects

- Coding to an interface!

```
Duck duck = new MallardDuck();
```

  - Duck – interface
  - MallardDuck – concrete class

- Needs different ducks depending on the condition!

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

- Nothing wrong with “new”

Factory Patterns

- Creational patterns
  - Every object-oriented language has an idiom for object creation. (e.g. new operator in C++ and Java)
  - Creational patterns allow programmers to write methods creating new objects without explicitly using the new operator
  - Enable us to write methods that can instantiate different objects and that can be extended to instantiate other newly-developed objects, all without modifying the client code!
  - Examples: Factory Method, Abstract Factory, Singleton, Builder, Prototype
- Factory Method
  - Uses inheritance to decide the object to be instantiated
- Abstract Factory
  - Delegates object creation to a factory object

Original Code

```
Pizza orderPizza(String type) {
    Pizza pizza;

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

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

Pizza types are changing!

```
Pizza orderPizza(String type) {
      Pizza pizza;

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

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

      return pizza;
}
```

```
Pizza orderPizza(String type) {
      Pizza pizza;

      if (type.equals("cheese"))
          pizza = new CheesePizza();
      else if (type.equals(“clam"))
          pizza = new ClamPizza();
      else if (type.equals("pepperoni"))
          pizza = new PepperoniPizza();

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

      return pizza;
}
```

Building a simple factory

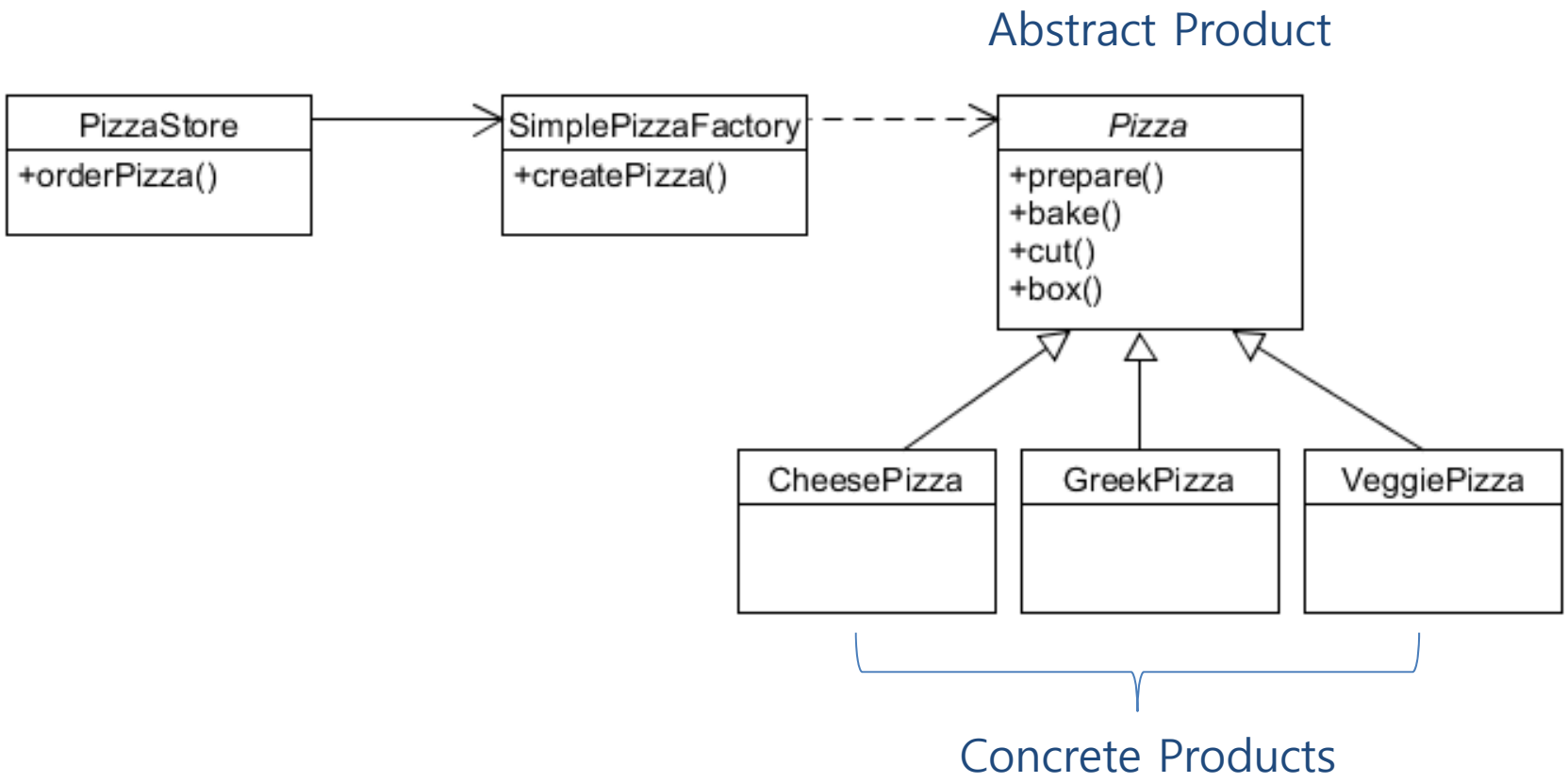
```
public class SimplePizzaFactory {
    public Pizza createPizza(String type) {
        Pizza pizza;

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

        return pizza;
    }
}
```

Revising PizzaStore class using the simple factory

```
pubilc 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;
    }
}
```

Q) It looks like we are just pushing the problem off to another object

Simple Factory (not an official pattern)



Franchising the pizza store

- Different styles
  - NY, Chicago, California, etc
- Different pizza factories
  - NYPizzaFactory, ChicagoPizzaFactory, CaliforniaPizzaFactory

```
NYPizzaFactory nyFactory = new NYPizzaFactory();
PizzaStore nyStore = new PizzaStore(nyFactory);
nyStore.order("Veggie");

ChicagoPizzaFactory chicagoFactory = new ChicagoPizzaFactory();
PizzaStore chicagoStore = new PizzaStore(chicagoFactory);
chicagoStore.order("Veggie");
```

Factory Method Pattern

- Purpose
  - Exposes a method for creating objects, allowing subclasses to control the actual creation process.
- Use When
  - A class will not know what classes it will be required to create.
  - Subclasses may specify what objects should be created.
  - Parent classes wish to defer creation to their subclasses.

Requirement Change

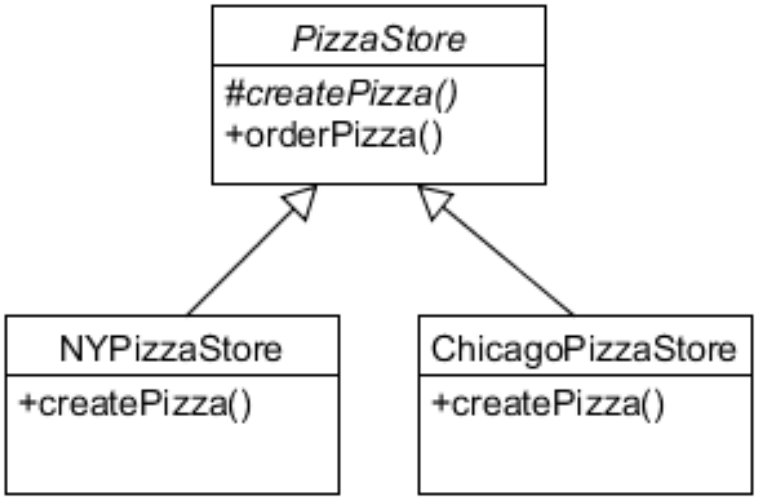
- Now we need want to make a framework which ties the store and the pizza creation

```
public abstract class PizzaStore {
    public Pizza orderPizza(String type) {
        Pizza pizza;
        pizza = createPizza(type);

        pizza.prepare();
        pizza.bake();
        pizza.cut();
        pizza.box();
        return pizza;
    }
    protected abstract Pizza createPizza(String type);
}
```

Factory Method

- This class is a (very simple) OO framework. The framework provides one service “prepare pizza”.
- The framework invokes the createPizza() factory method to create a pizza that it can prepare using a well-defined, consistent process.
- A “client” of the framework will subclass this class and provide an implementation of the createPizza() method.
- Any dependencies on concrete “product” classes are encapsulated in the subclass.

Allowing the subclasses to decide




```
public Pizza createPizza(String type) {
    Pizza pizza;
    if (type.equals(“cheese”)) {
        pizza = new NYStyleCheesePizza();
    } else if (type.equals(“veggie”)) {
        pizza = new NYStyleVeggiePizza();
    }
    return pizza;
}
```

```
public Pizza createPizza(String type) {
    Pizza pizza;
    if (type.equals(“cheese”)) {
        pizza = new ChicagoStyleCheesePizza();
    } else if (type.equals(“veggie”)) {
        pizza = new ChicagoStyleVeggiePizza();
    }
    return pizza;
}
```

Ordering a pizza using the Factory Method

1. Need an instance of PizzaStore

```
PizzaStore nyPizzaStore = new NYPizzaStore();
```

2. Call orderPizza()

```
nyPizzaStore.orderPizza("cheese");
```

3. orderPizza() calls createPizza()

```
Pizza pizza = createPizza("cheese");
```

4. orderPizza() now prepare, bake, cut and box the pizza

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

Implementing Pizza class

```
public abstract class Pizza {
    String name;
    String dough;
    String sauce;
    ArrayList toppings = new ArrayList();
    void prepare(){
        System.out.println(“Preparing “ + name);
        System.out.println(“Tossing dough...”);
        System.out.println(“Adding sauce...”);
        System.out.println(“Adding toppings: “);
        for (int i = 0; i  < toppings.size(); i++)
            System.out.println(“  “ + toppings.get(i));
    }
    void bake(){
        System.out.println(“Bake for 25 minutes at 350”);
    }
    void cut(){
        System.out.println(“Cutting the pizza into diagonal slices”);
    }
    void box(){
        System.out.println(“Place pizza in official PizzaStore box”);
    }
}
```

Subclasses of Pizza

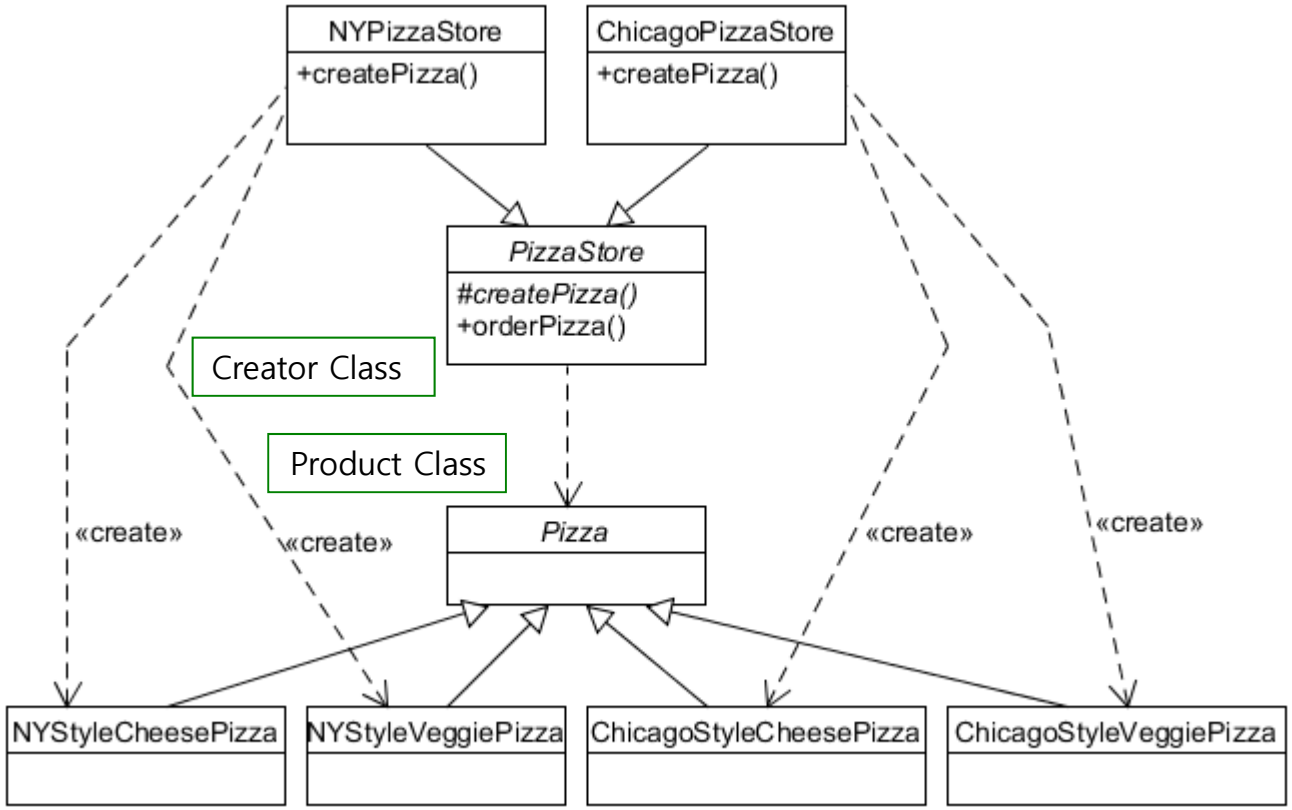
```
public class NYStyleCheesePizza extends Pizza {
    public NYStyleCheesePizza(){
        name = “NY Style Sauce and Cheese Pizza”;
        dough = “Thin Crust Dough”;
        sauce = “Marinara Sauce”;
        toppings.add(“Grated Reggiano Cheese”);
    }
}

public class ChicagoStyleCheesePizza extends Pizza {
    public ChicagoStyleCheesePizza(){
        name = “Chicago Style Deep Dish and Cheese Pizza”;
        dough = “Extra Thick Crust Dough”;
        sauce = “Plum Tomato Sauce”;
        toppings.add(“Shredded Mozzarella Cheese”);
    }
    void cut(){
        System.out.println(“Cutting the pizza into square slices”);
    }
}
```

Test Drive

```
public class PizzaTestDrive {
    public static void main(String[] args){
        PizzaStore nyStore = new NYPizzaStore();
        PizzaStore chicagoStore = new ChicagoPizzaStore();

        Pizza pizza = nyStore.orderPizza(“cheese”);
        System.out.println(“Ethan ordered a “ + pizza.getName());
        Pizza pizza = chicagoStore.orderPizza(“cheese”);
        System.out.println(“Joel ordered a “ + pizza.getName());
    }
}
```

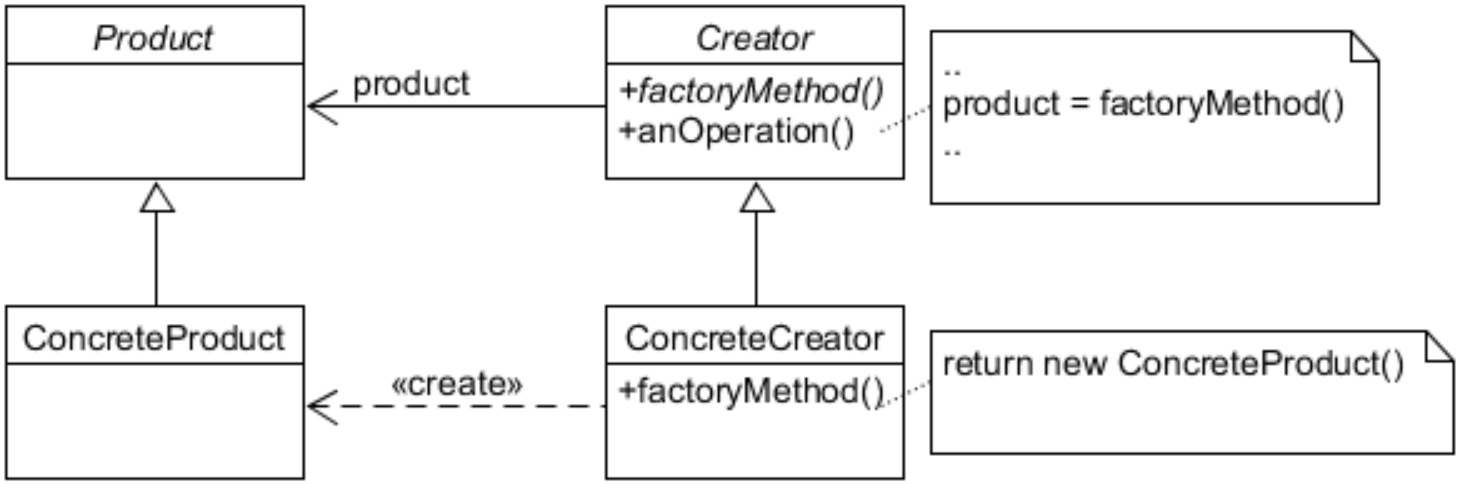
Applying Factory Method Pattern

- The Factory Method pattern encapsulates object creation by letting subclasses decide what objects to create



Factory Method Pattern

- Defines an interface for creating an object, but lets subclasses decide which class to instantiate.
- Factory Method lets a class defer instantiation to subclasses.



A very dependent PizzaStore

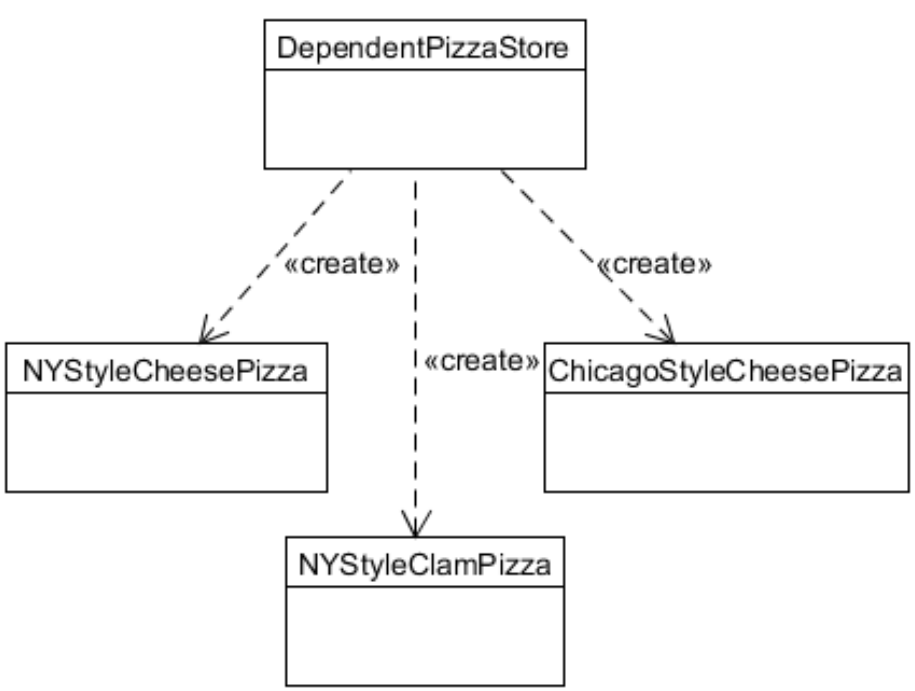
```
public class DependentPizzaStore {
    public Pizza createPizza(String style, String type) {
        Pizza pizza = null;
        if (style.equals("NY")) {
            if (type.equals("cheese")) {
                pizza = new NYStyleCheesePizza();
            } else if (type.equals("veggie")) {
                pizza = new NYStyleVeggiePizza();
            }
        } else if (style.equals("Chicago")) {
            if (type.equals("cheese")) {
                pizza = new ChicagoStyleCheesePizza();
            } else if (type.equals("veggie")) {
                pizza = new ChicagoStyleVeggiePizza();
            }
        } else {
            System.out.println("Error: invalid type of pizza");
            return null;
        }
        pizza.prepare();
        pizza.bake();
        pizza.cut();
        pizza.box();
        return pizza;
    }
}
```

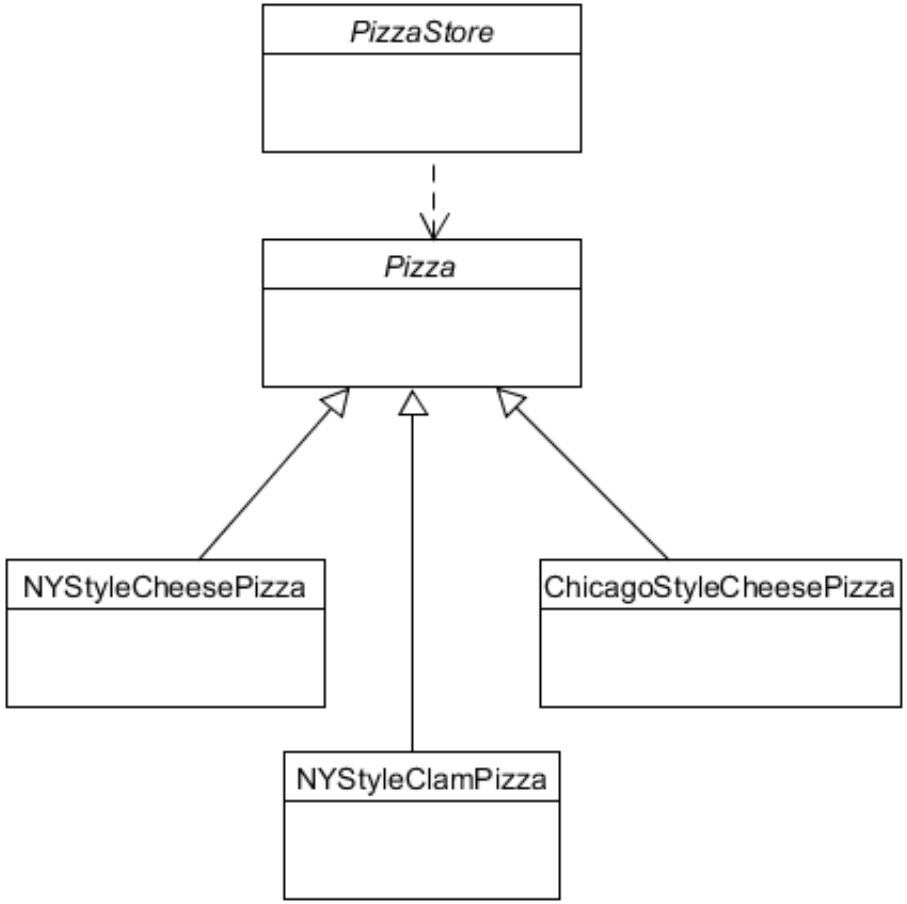
Design Principle: Dependency Inversion Principle

- Dependency Inversion Principle
  - Depend upon abstractions. Do not depend upon concrete classes.
  - High-level components should not depend on low-level components; rather, they should both depend on abstractions

- Factory Method is one way of following the dependency inversion principle

Dependent PizzaStore VS Factory Method





DIP: How to?

- To achieve the dependency inversion principle in your own designs, follow these GUIDELINES
  - No variable should hold a reference to a concrete class
  - No class should derive from a concrete class
  - No method should override an implemented method of any of its base class

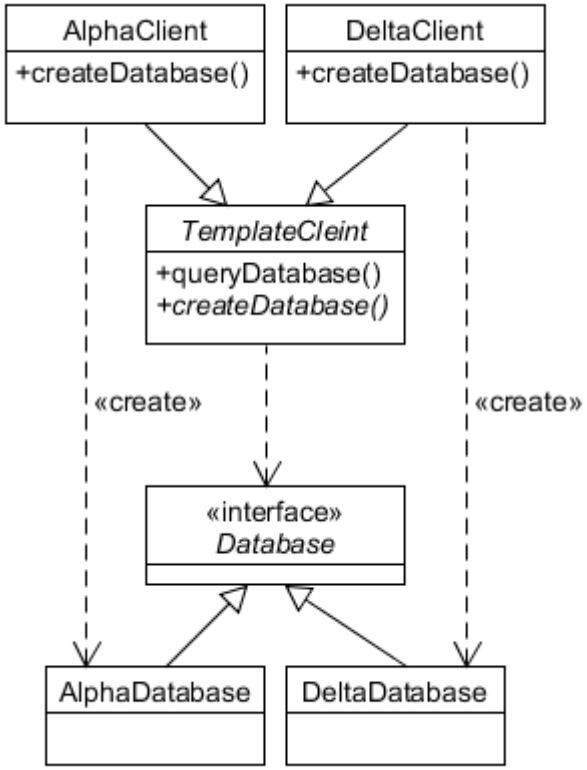
Exercise

```
public interface Database {}
public class AlphaDatabase implements Database {}
public class DeltaDatabase implements Database {}

public class BadClient {
    public Map queryDatabase(String queryString) {
        Database db = new AlphaDatabase();
        checkQueryIsValid(queryString);
        return db.find(queryString);
    }
}
```

- Apply the Factory Method Pattern and revise the above code
- Hint: Database is a product

Solution: Applying Factory Method Pattern

```
public abstract class TemplateClient {
    public Map queryDatabase(String queryString) {
        Database db = createDatabase ();
        checkQueryIsValid(queryString);
        return db.find(queryString);
    }
    protected abstract Database createDatabase ();
}


public class AlphaClient extends TemplateClient {
    public Database createDatabase () {
        return new AlphaDatabase();
    }
}

public class DeltaClient extends TemplateClient {
    public Database createDatabase () {
        return new DeltaDatabase();
    }
}
```



Factory Method Pattern

- What exactly does it mean by saying that "the Factory Method Pattern lets subclasses decide which class to instantiate?“

  - **Creator class** is written in such a fashion that it does not know what actual Concrete Product class will be instantiated. The **Concrete Product** class to be instantiated is determined solely by which **Concrete Creator** subclass is instantiated and used by the application.

  - **It does *not* mean that the subclass decides at runtime which ConcreteProduct class to create**

Abstract Factory Pattern

- Purpose
  - Provide an interface that delegates creation calls to one or more concrete classes in order to deliver specific objects.
- Use When
  - The creation of objects should be independent of the system utilizing them.
  - Systems should be capable of using multiple families of objects.
  - Families of objects must be used together.
  - Libraries must be published without exposing implementation details.
  - Concrete classes should be decoupled from clients.

Factory Patterns

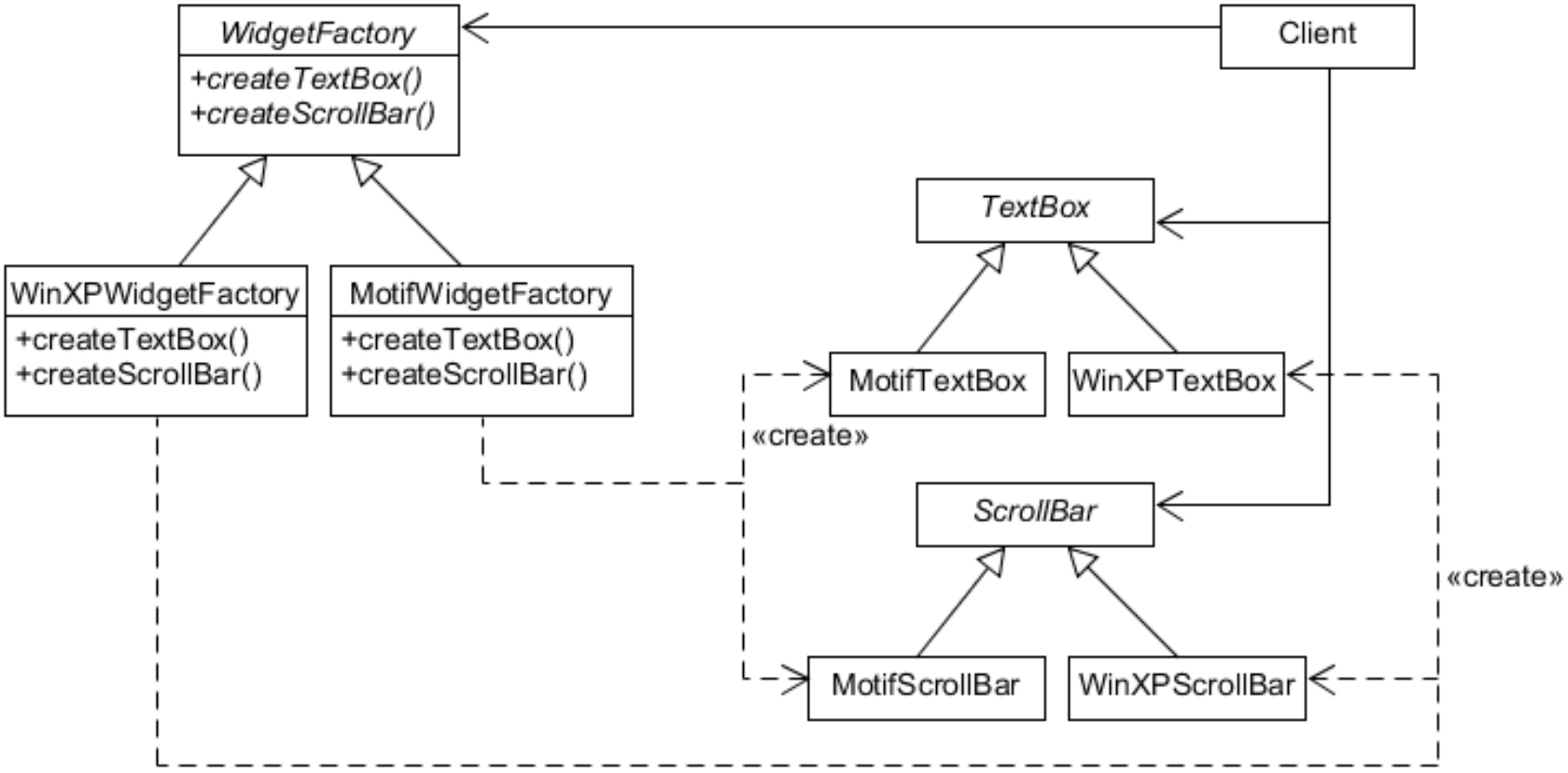
- Abstract factory pattern is one level of abstraction higher than the factory method pattern

- To delegate or defer the responsibility of object instantiation to another object or class
  - **Abstract Factory** pattern uses **composition**
  - **Factory Method** pattern uses **inheritance**

Example

- Building a user interface toolkit that supports multiple look and feel standards
  - Windows XP, MAC OS X, Motif, X Window
- Have different appearances and behavior for a large set of subclasses
  - scroll bars, windows, buttons, etc.

Abstract Factory Example

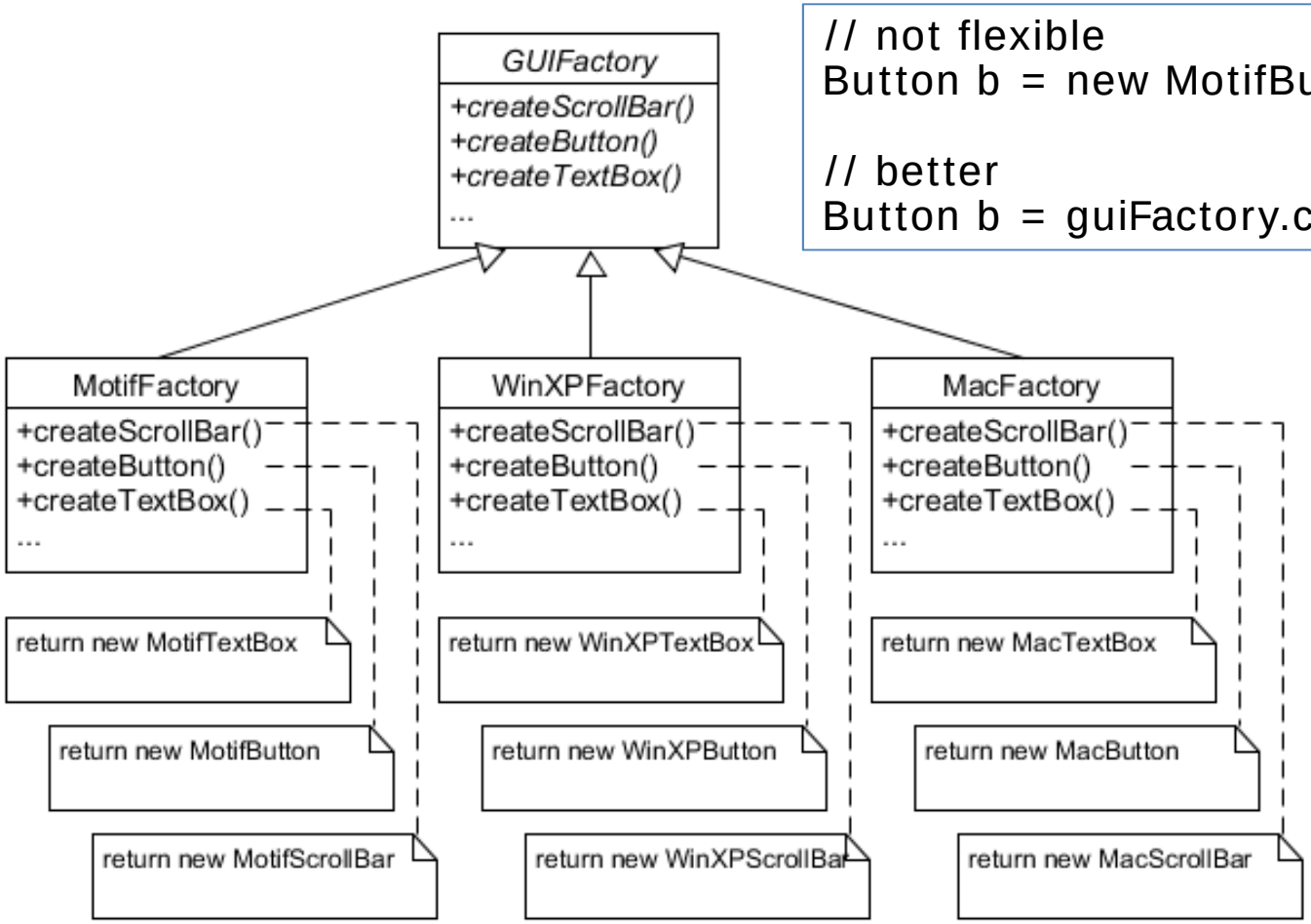


Factory Object Set-up and Delegation

```
public class UIToolKit {
    WidgetFactory wfactory;

    public UIToolKit (WidgetFactory wf)
        this.wfactory = wf;
    }

    public void PopUpPrinterDialog() {
        DialogWindow d= wfactory.createDialogWindow("Printer Setup");
        d.add ( wfactory.createButton(OK_Button) );
        d.add ( wfactory.createButton(Cancel_Button) );
        d.showWindow();
    }

}
```

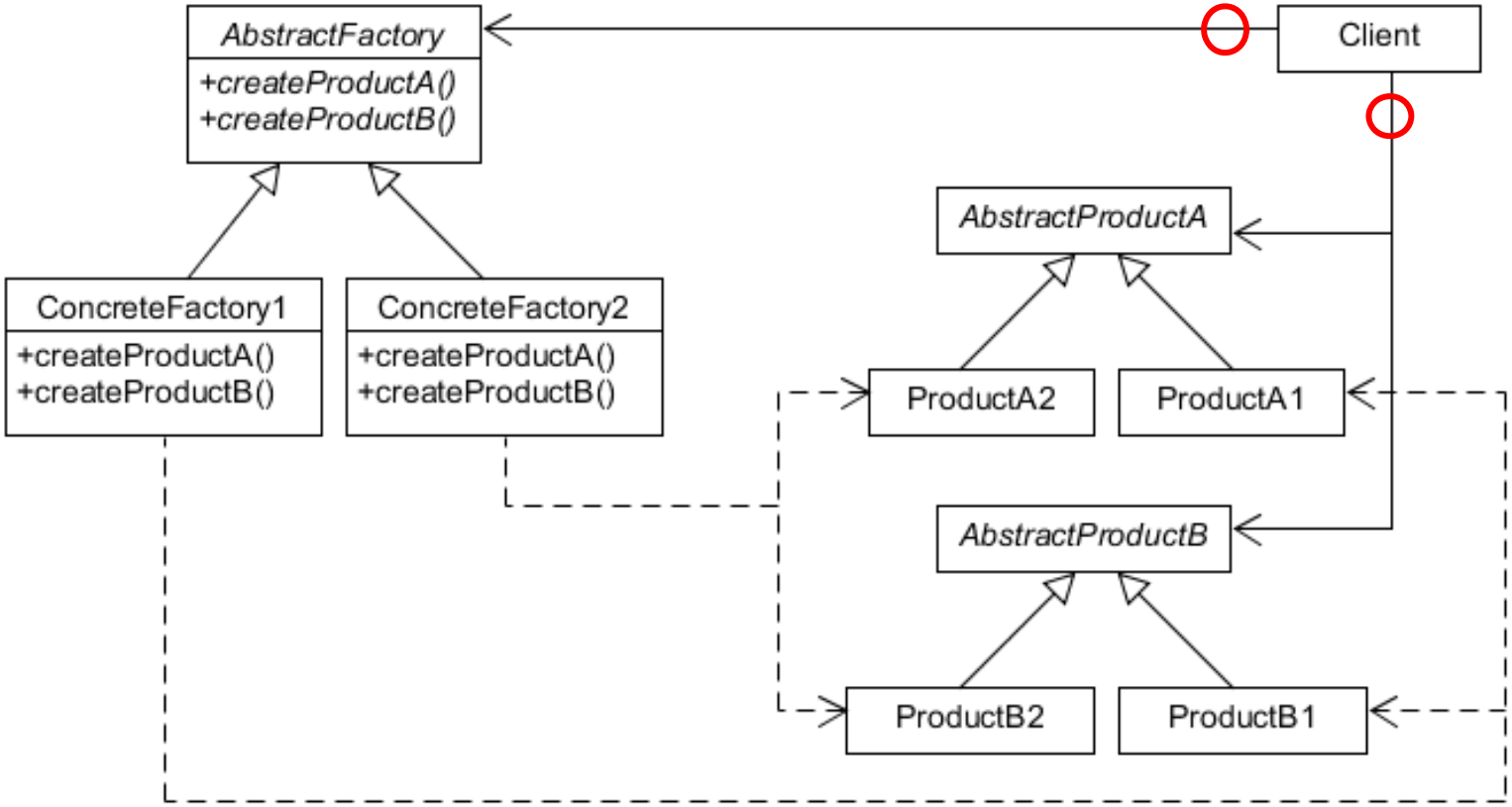
```
public class DriverForMotif {
    // setup a specific widget factory
    WidgetFactory wf = new WinXPWidgetFactory();

    // start a client program
    UIToolKit ui = new UIToolKiet(wf);

    // rest of code
}
```

Class Diagram for GUIFactory




```
// not flexible
Button b = new MotifButton();

// better
Button b = guiFactory.createButton();
```

Class Diagram for Abstract Factory Pattern



Participants

- AbstractFactory
  - Declares an interface for operations that create abstract product objects
- ConcreteFactory
  - Implements the operations to create concrete product objects
- AbstractProduct
  - Declares an interface for a type of product object
- ConcreteProduct
  - Defines a product object to be created by the corresponding concrete factory
  - Implements the AbstractProduct interface
- Client
  - Uses only interfaces declared by AbstractFactory and AbstractProduct classes

Making factories for ingredients (Abstract Factory)

```
public interface PizzaIngredientFactory {
    public Dough createDough();
    public Sauce createSauce();
    public Cheese createCheese();
    public Veggies[] createVeggies();
    public Pepperoni createPepperoni();
    public Clams createClam();
}
```

Concrete PizzaIngredientFactory (Factory Object)

```
public class NYPizzaIngredientFactory implements PizzaIngredientFactory {
    public Dough createDough() {
        return new ThinCrustDough();
    }
    public Sauce createSauce() {
        return new MarinaraSauce();
    }
    public Cheese createCheese() {
        return new ReggianoCheese();
    }
    public Veggies[] createVeggies() {
        Veggies veggies[] = {new Garlic(), new Onion(), new Mushroom(),
              new RedPepper() };
        return veggies;
    }
    public Pepperoni createPepperoni() {
        return new SlicedPepperoni();
    }
    public Clams createClam() {
        return new FreshClams();
    }
}
```

Pizza product class

```
public abstract class Pizza {
    String name;
    Dough dough;
    Sauce sauce;
    Veggies veggies[];
    Cheese cheese;

    abstract public void prepare();
    public void bake() {
        System.out.println("Bake for 25 minutes at 350");
    }
    public void cut() {
        System.out.println("Cutting the pizza into diagonal slices");
    }
    public void box() {
        System.out.println("Place pizza in official PizzaStore box");
    }
    public void setName(String name) {
        this.name = name;
    }
    public String getName() {
        return name;
    }
    public String toString() {
        return this.getName();
    }
}
```

```
public class NYPizzaStore extends PizzaStore {
  protected Pizza createPizza(String item) {
    Pizza pizza = null;
    PizzaIngredientFactory ingredientFactory = new NYPizzaIngredientFactory();
    if (item.equals("cheese")) {
        pizza = new CheesePizza(ingredientFactory);
        pizza.setName("New York Style Cheese Pizza");
    } else if (item.equals("veggie")) {
        pizza = new VeggiePizza(ingredientFactory);
        pizza.setName("New York Style Veggie Pizza");
    } else if (item.equals("clam")) {
        pizza = new ClamPizza(ingredientFactory);
        pizza.setName("New York Style Clam Pizza");
    } else if (item.equals("pepperoni")) {
        pizza = new PepperoniPizza(ingredientFactory);
        pizza.setName("New York Style Pepperoni Pizza");
    }
    return pizza;
  }
}
```

```
public class CheesePizza extends Pizza {
    PizzaIngredientFactory ingredientFactory;
    public CheesePizza(PizzaIngredientFactory ingredientFactory) {
        this.ingredientFactory = ingredientFactory;
    }
    void prepare() {
        System.out.println("Preparing " + name);
        dough = ingredientFactory.createDough();
        sauce = ingredientFactory.createSauce();
        cheese = ingredientFactory.createCheese();
    }
}
```

What has changed?

1. Instantiating PizzaStore

```
PizzaStore nyPizzaStore = new NYPizzaStore();
```

2. Order

```
nyPizzaStore.orderPizza("cheese");
```

3. orderPizza() calls createPizza()

```
Pizza pizza = createPizza("cheese");
```

/** BEGIN of Changed Flow for Ingredient Factory **/

4. createPizza() now needs ingredientFactory

```
Pizza pizza = new CheesePizza(nyIngredientFactory);
```

5. prepare() creates ingredients using ingredient factory

```
void prepare(){
  dough = ingredientFactory.createDough();
  sauce = ingredientFactory.createSauce();
  cheese = ingredientFactory.createCheese();
}
```

/** END of Changed Flow for Ingredient Factory **/

6. Now it is ready. orderPizza() calls bake(), cut(), box()

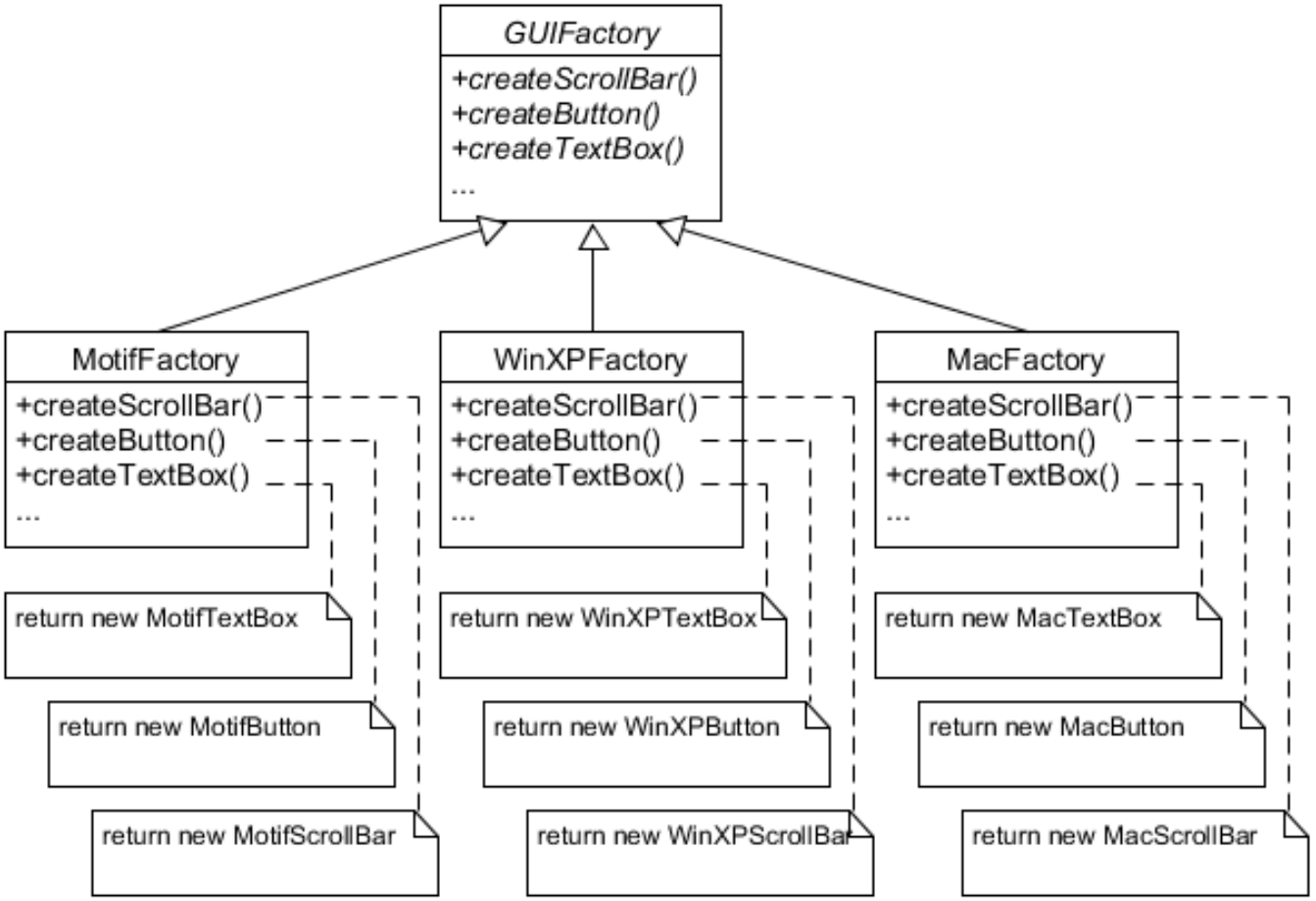
When to Use Abstract Factory Pattern?

- A system must be independent of how its products are created, composed, and represented
- A class cannot anticipate the class of objects it must create
- A system should use one of a set of families of products
- A family of related product objects is designed to be used together, and the constraint needs to be enforced

Abstract Factory Pattern

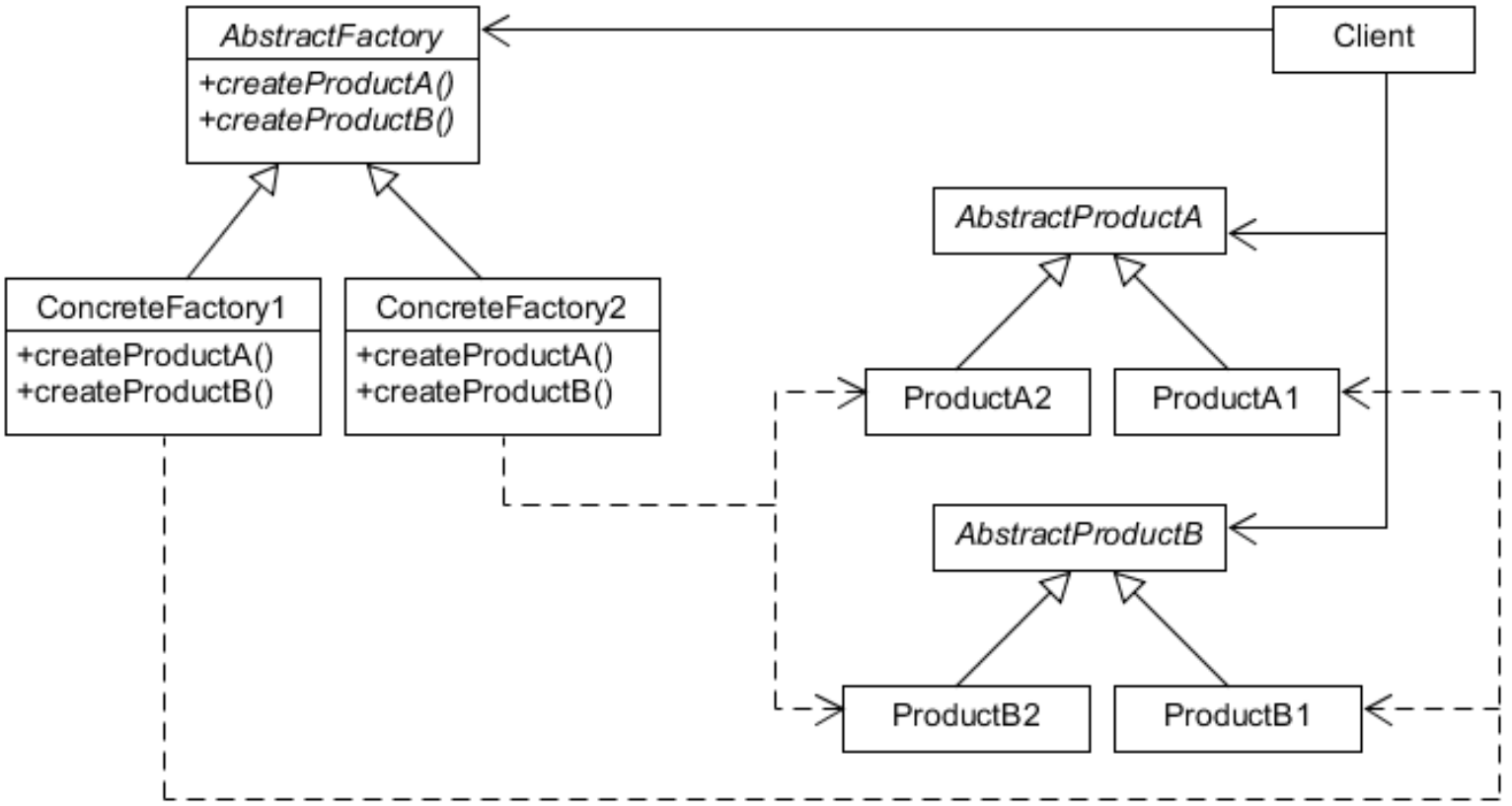
- Consequences
  - Isolates concrete classes
    - Factory encapsulates responsibility and process of creating parts
    - Isolates clients from implementation classes
  - Exchanging product families easy
    - Concrete factory appears once where it is instantiated
  - Promotes consistency among products
  - Supporting new kinds of products is difficult
    - Fixes set of parts to be created

Abstract Factory Pattern



- Supporting new kinds of products is difficult
  - Fixes set of parts to be created

Abstract Factory Pattern



- Supporting additional factory object is easy

Related Patterns

- Abstract Factory classes are often implemented with Factory Methods, but they can also be implemented using Prototype.

- Often, designs start out using Factory Method (less complicated, more customizable, subclasses proliferate) and evolve toward Abstract Factory, Prototype, or Builder (more flexible, more complex) as the designer discovers where more flexibility is needed.

- Dependency Inversion Principle (DIP)
  - Depend on abstractions. Do not depend on concrete classes
- Abstract Factory
  - Provide an interface for creating families of related or dependent objects without specifying their concrete classes
- Factory Method
  - Define an interface for creating an object, but let subclasses decide which class to instantiate.
  - Factory Method lets a class defer instantiation to the subclasses