Factory Method and Abstract Factory Patterns

Factory Method: Defines an interface for creating an object, but let **subclasses** decide which class to instantiate.

Abstract Factory: Provides an interface for creating families of related or dependent objects without specifying their concrete classes. Creation is done through object composition

Factory

- Creational
- Too many dependencies to concrete classes makes your software difficult to maintain and modify
  - **Remember**: program to an interface, not an implementation

Example

```
Duck duck;

if (picnic) {
      duck = new Mallarduck();
}
else if (hunting) {
      duck = new DecoyDuck();
}
else if (inBathTub) {
      duck = new RubberDuck();
}
```

Adding a new Duck: Classes should be open for extension, but closed for modification

```
Duck duck;

if (picnic) {
      duck = new Mallarduck();
}
else if (hunting) {
      duck = new DecoyDuck();
}
else if (inBathTub) {
      duck = new RubberDuck();
}
else if (ginger) {
      duck = new RedHeadDuck();
}
```

Dealing with **Change**

Recall some of our fundamental Design Principles:

- Code to an interface and insulate yourself from changes (Strategy)
- Identify aspects that vary and separate them from what stays the same (Strategy)
- Close your code for modification (Decorator)

Pizza Example (partly from text)

```java
public 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 Example

```java
public 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;
}
```

VARIES

SAME

Factor Out

```
public 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;
    }
```

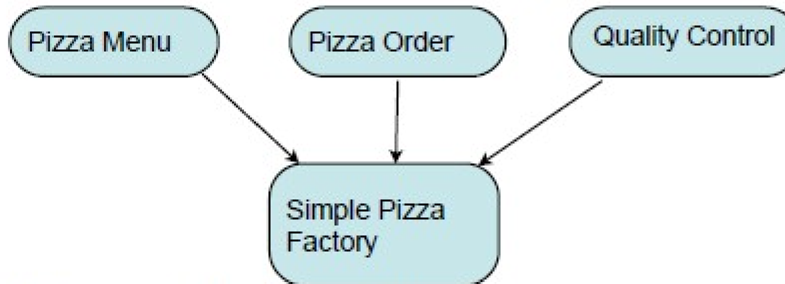
factories handle the detail of object creation

Pizza Factory

```java
public class SimplePizzaFactory {

   public Pizza createPizza(String type) {
        Pizza pizza = null;

        if (type.equals("cheese")) {
            pizza = new CheesePizza();
        } else if (type.equals("pepperoni")) {
            pizza = new PepperoniPizza();
        } else if (type.equals("clam")) {
            pizza = new ClamPizza();
        } else if (type.equals("veggie")) {
            pizza = new VeggiePizza();
        }
        return pizza;
   }
}
```

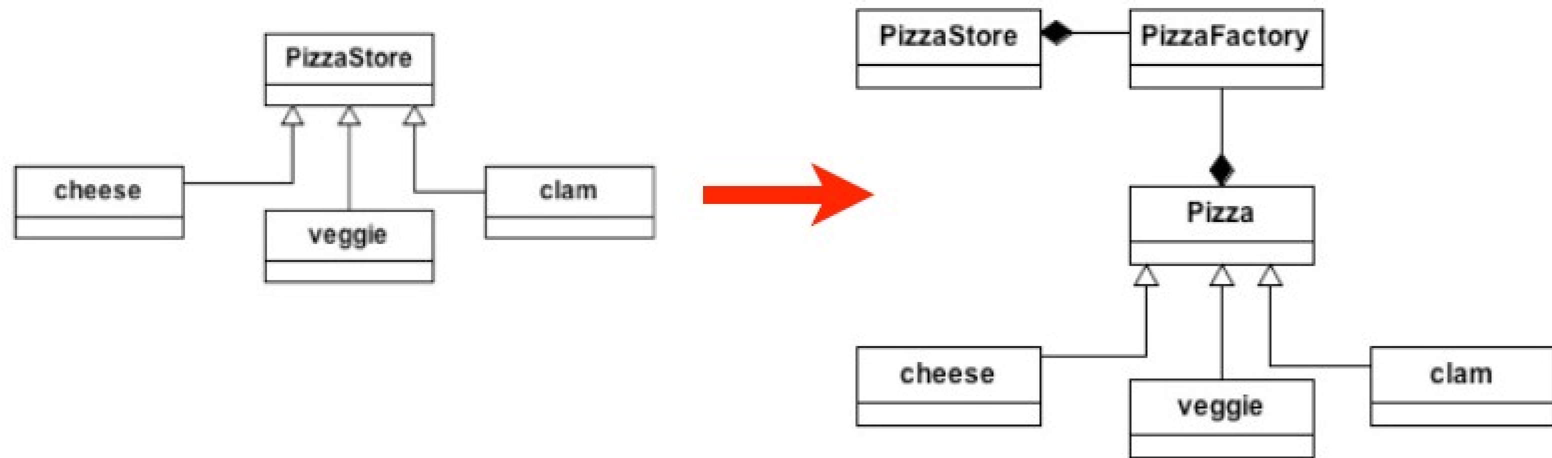
Benefits?

★ put object creation in an object?



★ "spaghetti-ness" of software

Simple Factory (not the same as Factory Method)



- ★ Subclass the object creation
- ★ Not a pattern but an idiom

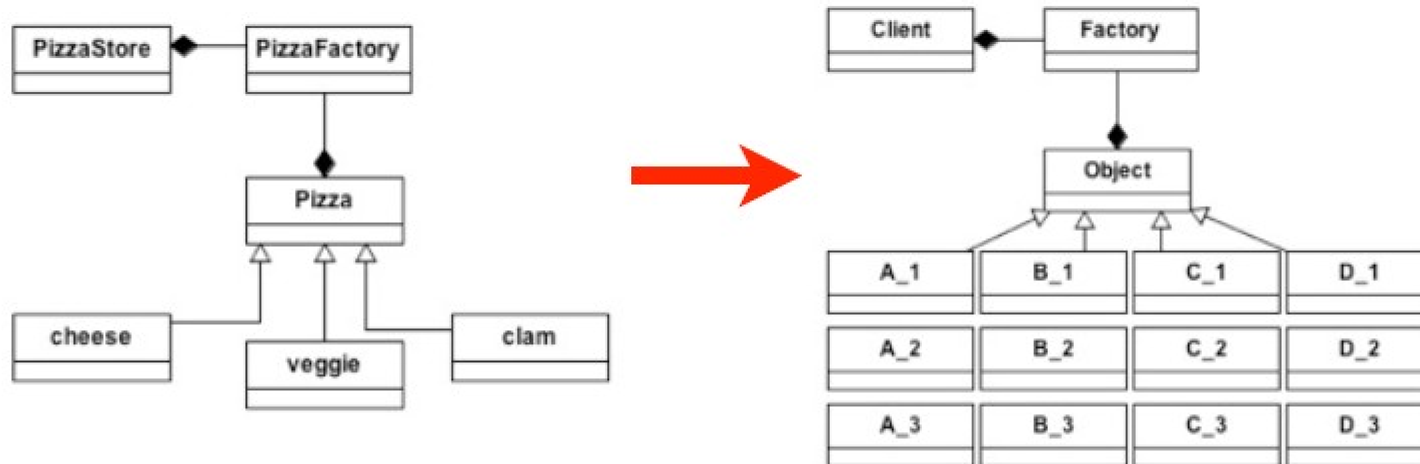
Simple Factory

- A Simple Factory may have many clients
- Each of those clients can ask the factory for an object – object creation is done in one place
- Allows decoupling of code
- Can have static versions of this, but then you can't subclass and change behavior of create method
- Simple Factory is not an actual pattern, but rather an idiom

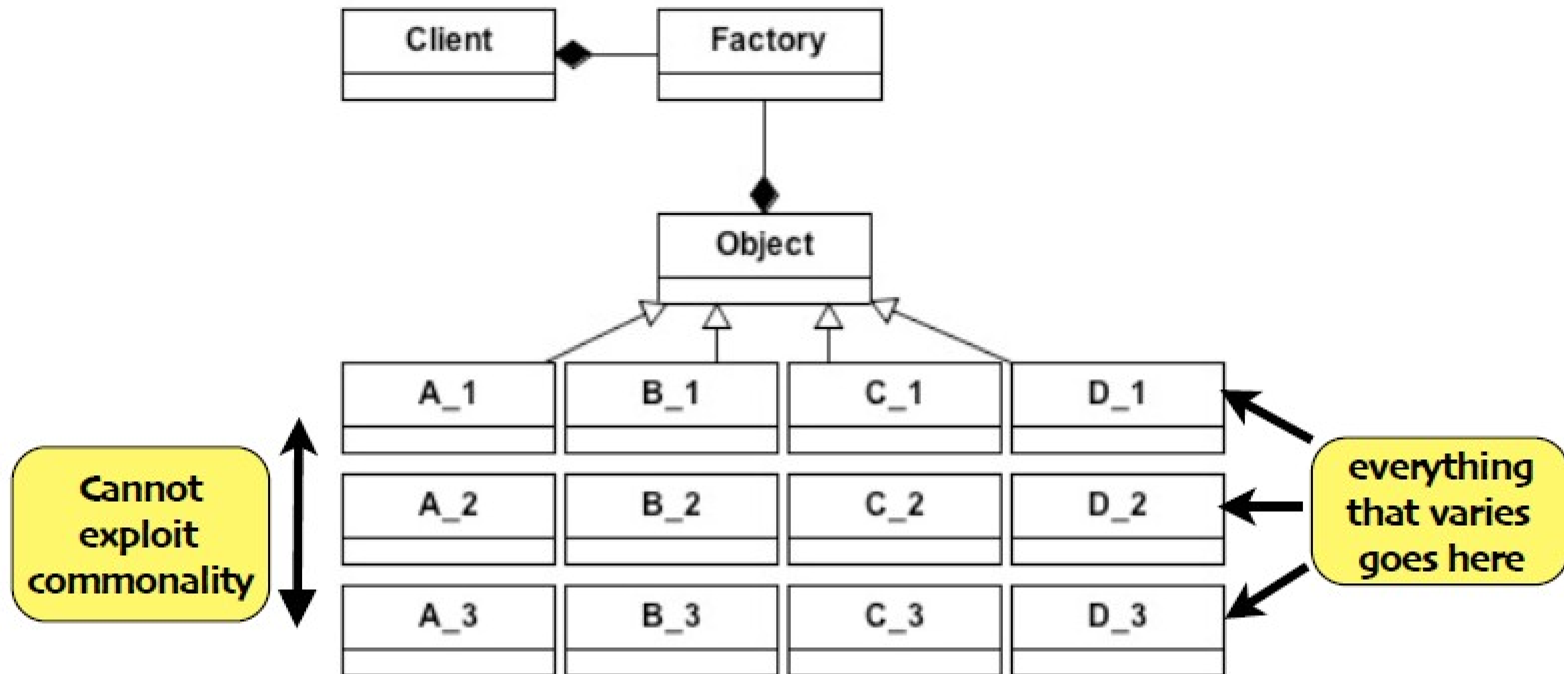
One Step Beyond

★ how can we support more variability?

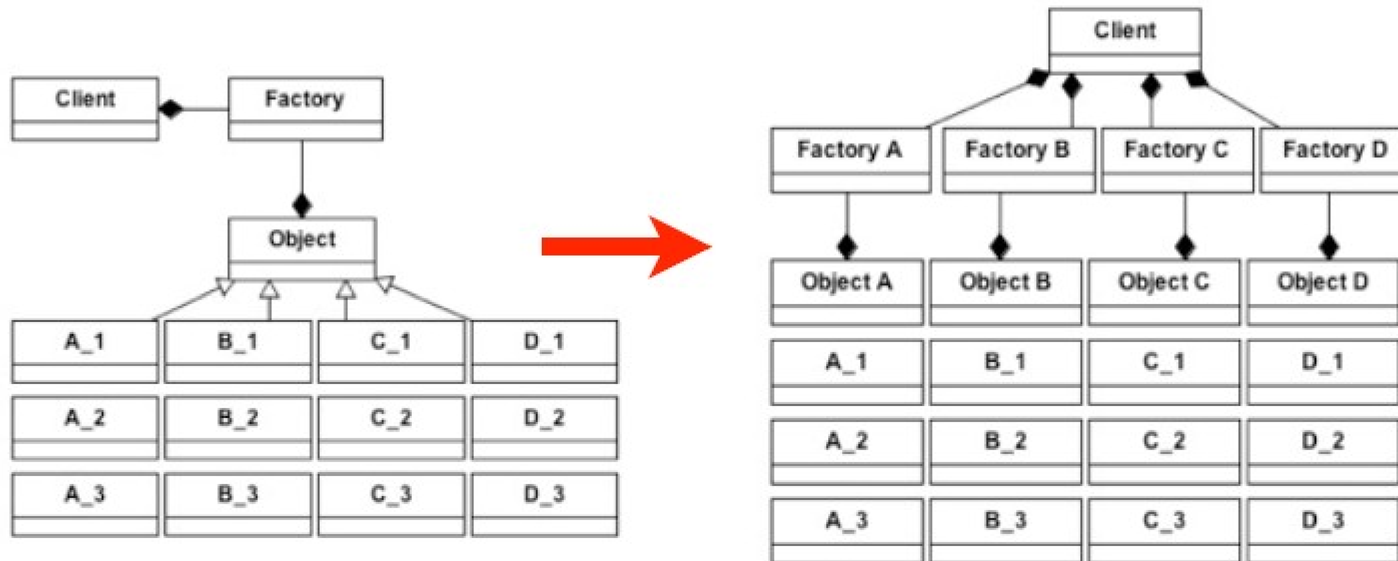
★ support "families" of products?



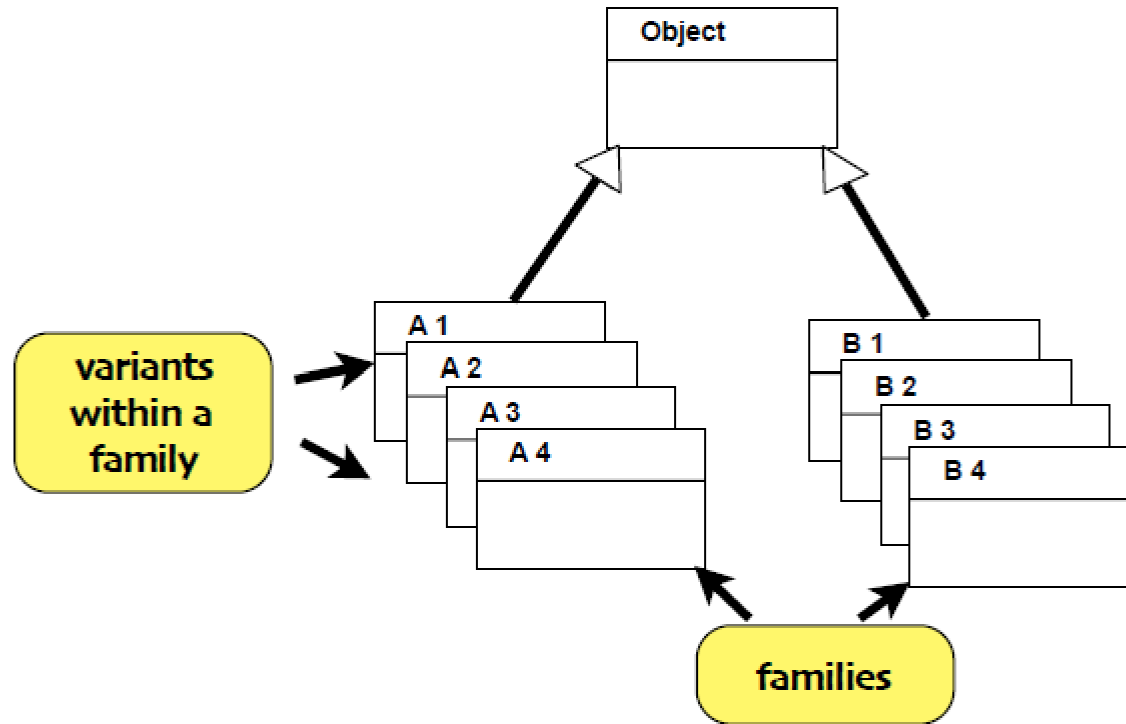
Problem



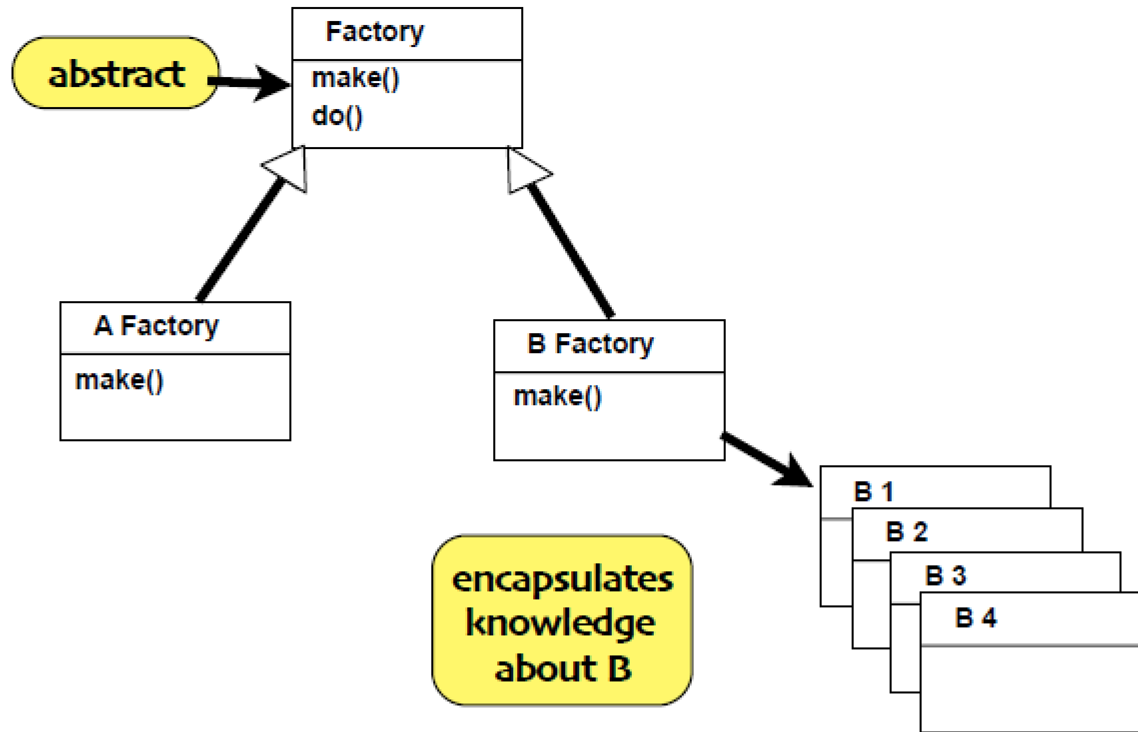
Subclass the Factory



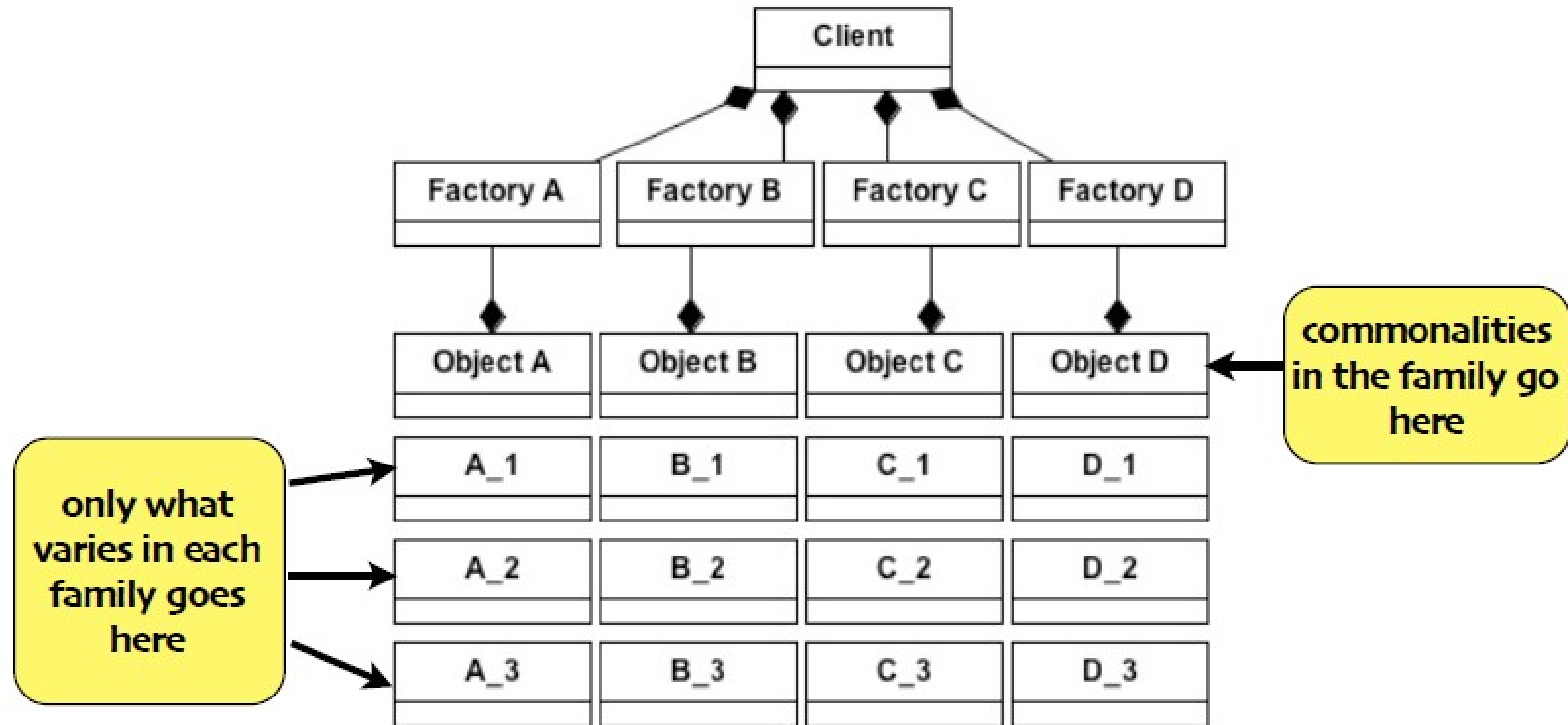
Example: Product Class



Creator Class



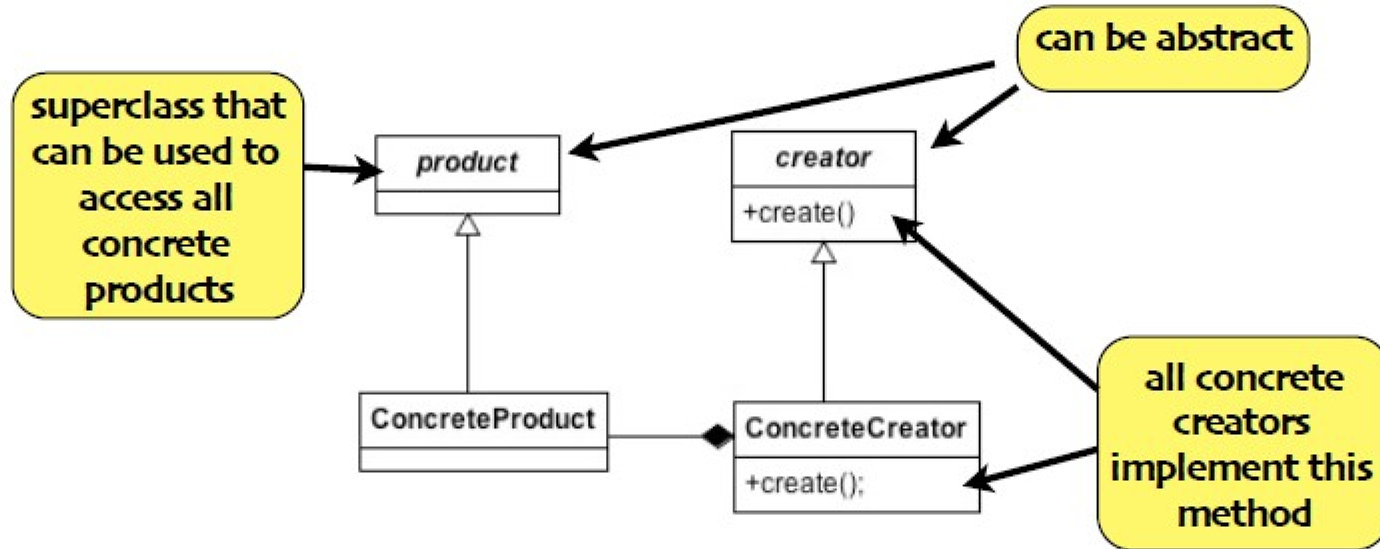
Benefits



Factory Method Pattern (from first slide)

Defines an interface for creating an object, but lets **subclasses** decide which class to instantiate. It lets a class defer instantiation to subclasses.

Factory Method Class Diagram



Example: Slot Machine

We'll follow the Factory Method to allow for creation of different kinds of slot machines

Main App

```
public class SlotMachineApp {
    public static void main(String[] args) {
        // TODO Auto-generated method stub
        SlotFactory njfactory = new NJSlotFactory();
        SlotFactory nvfactory = new NVSlotFactory();
        System.out.println("\n Trump Taj Mahal orders a:");
        Slot slot = njfactory.orderSlot("bonus");
        System.out.println("price:" + slot.getPrice());
        System.out.println("\n The Peppermill orders a:");
        Slot slot2 = nvfactory.orderSlot("progressive");
        System.out.println("price:" + slot.getPrice());
    }
}
```

SlotFactory

```java
public abstract class SlotFactory {

    public  Slot orderSlot(String type) {
        Slot slot = makeSlot(type);
        System.out.println("--- Making a " + slot.getName() + " ---");
        slot.collectParts();
        slot.assembleParts();
        slot.test("hardware");
        slot.uploadSoftware();
        slot.test("software");
        slot.ship();
        return slot;
    }
    abstract Slot makeSlot(String type);
}
```

Concrete Slot Factory

```
public class NJSlotFactory extends SlotFactory {
    Slot makeSlot(String type) {
    if (type.equals("progressive")) {
        return new NJStyleProgressiveSlot();
    }
    else if (type.equals("bonus")) {
        return new NJStyleBonusSlot();
    }
    else return null;
    }
}
```

Slot Class

```
import java.util.ArrayList;
public abstract class Slot {
    String name;
    String software = "linux";
    ArrayList components = new ArrayList();
    void makeSlot(String type) {
    }
    void collectParts() {
        System.out.print("fetching components:");
        for (int i=0; i< components.size(); i++) {
            System.out.print("   " + components.get(i));
            if (i==0) System.out.print(" Cabinet,");}
        System.out.println();
    }
    void assembleParts()
    void test(String type)
    void uploadSoftware()
    void ship()
    public String getName()
}
```

Concrete Slot

```
public class NVStyleBonusSlot extends Slot {
    public NVStyleBonusSlot() {
        name = "Nevada style Bonus Slot Machine";
        software = "OSX";
        components.add("Medium Coin CRT X86");
        price = 6000;
    }

    void ship() {
        System.out.println("having uncle vinnie drop it off");
    }
}
```

This is a "family" property
what is it doing here?

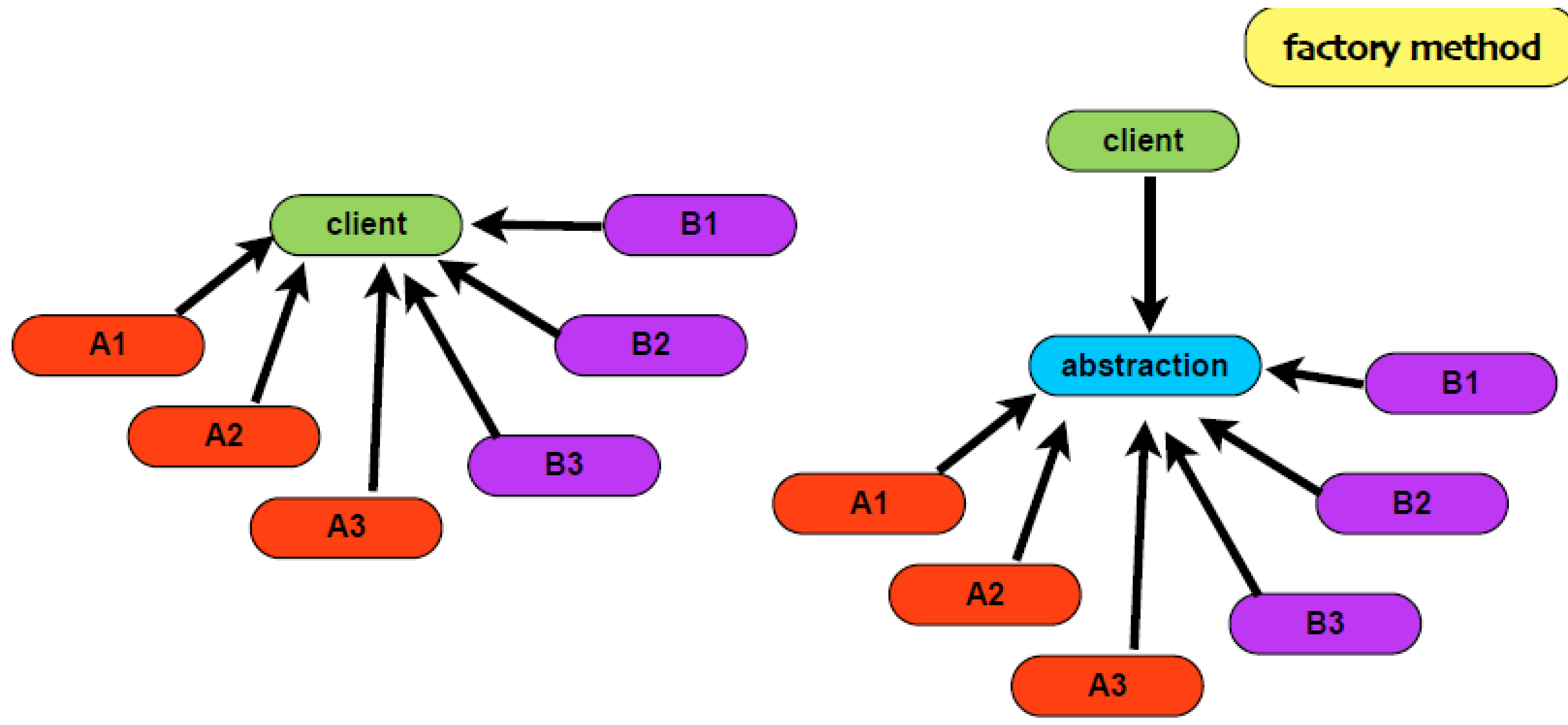
Design Principle

Dependency Inversion Principle:

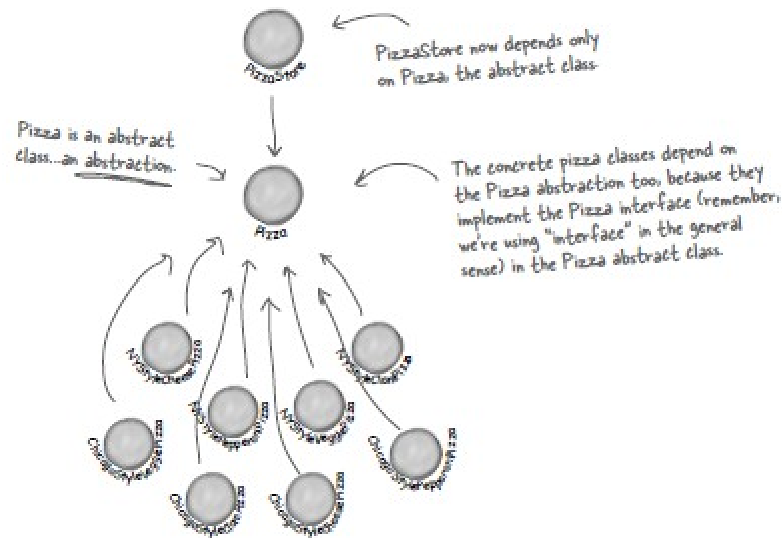
Depend on Abstractions, do not depend on Concrete Classes

Start at bottom (concrete objects that will be created) and find an abstraction that relates them all – use that abstraction to separate those objects from the client

Example



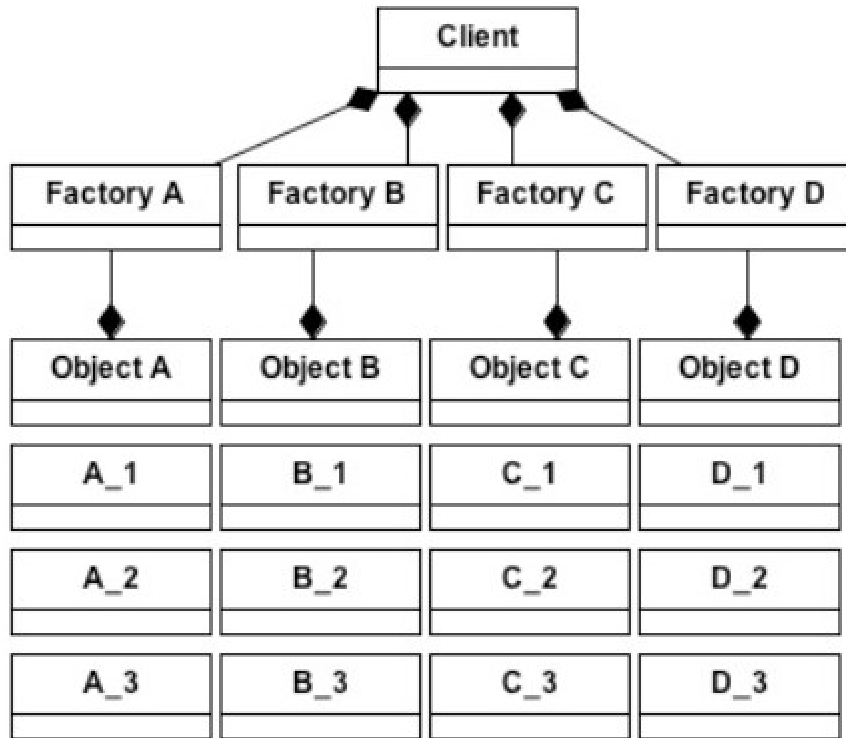
From book

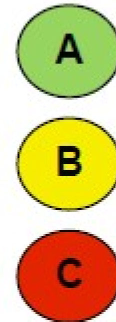


The OO Code (...they're more like Guidelines, see...)

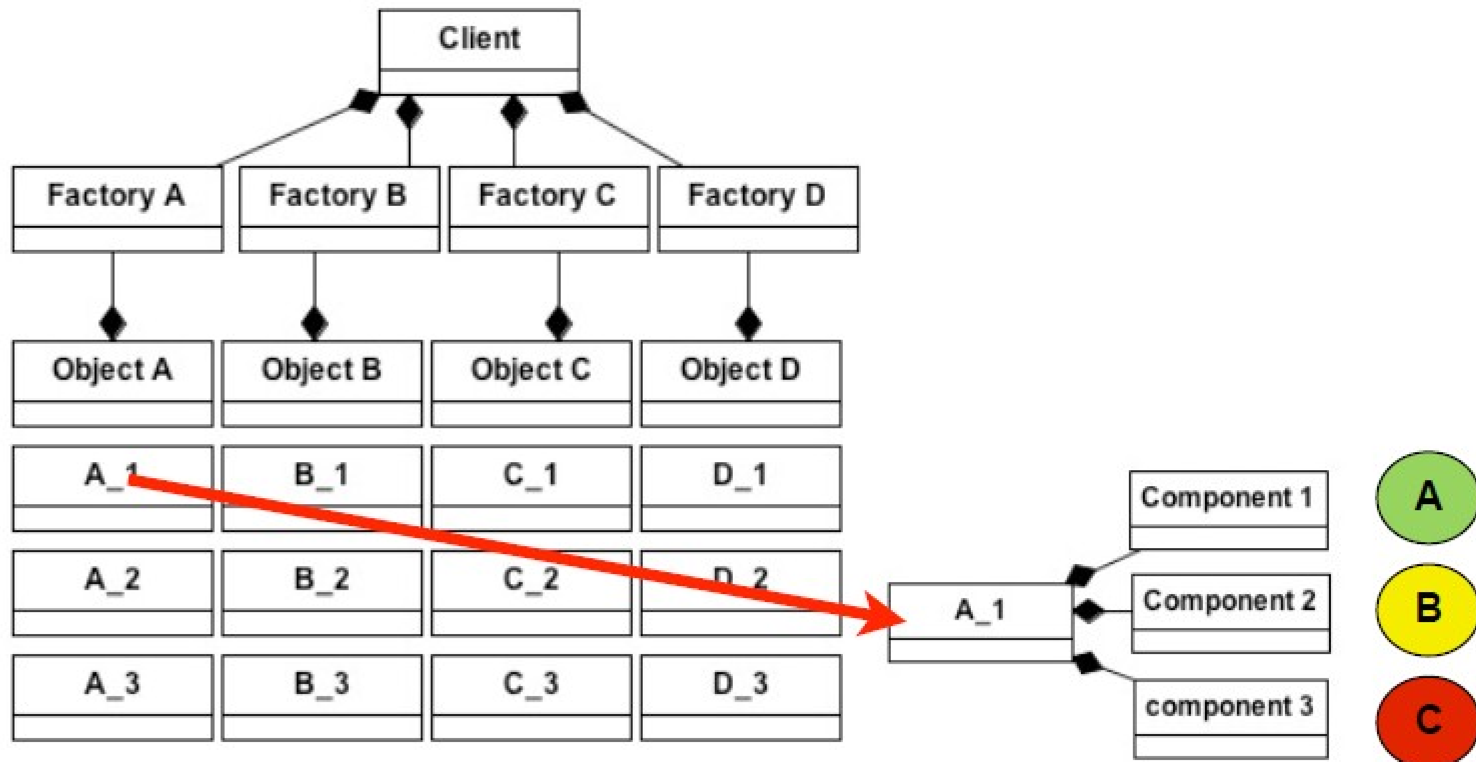
- No variable should have a reference to a concrete class
- No class should derive from a concrete class
- No method should override an implemented method of any of its base classes
- These be guidelines because if you adhered strictly to them, you wouldn't get much code written!

Pushing it further

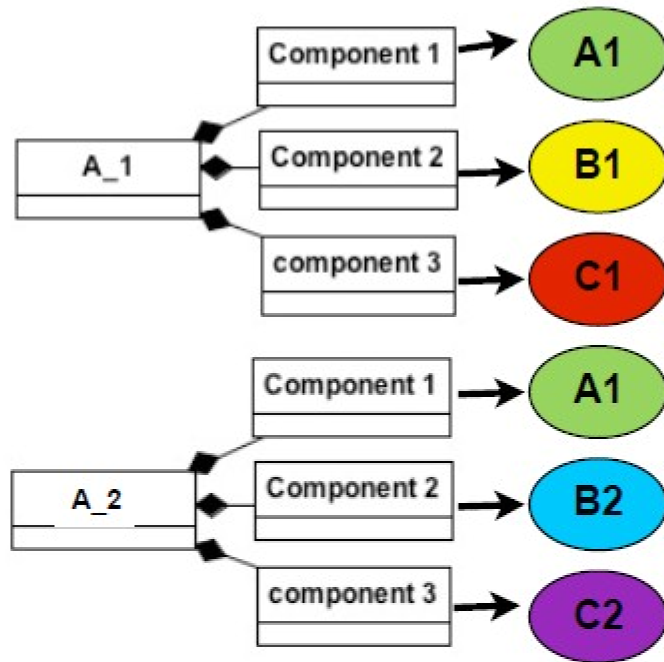


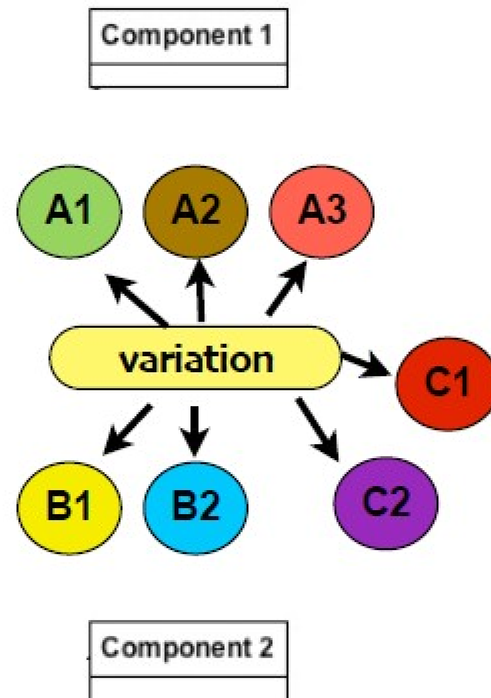


Pushing it further

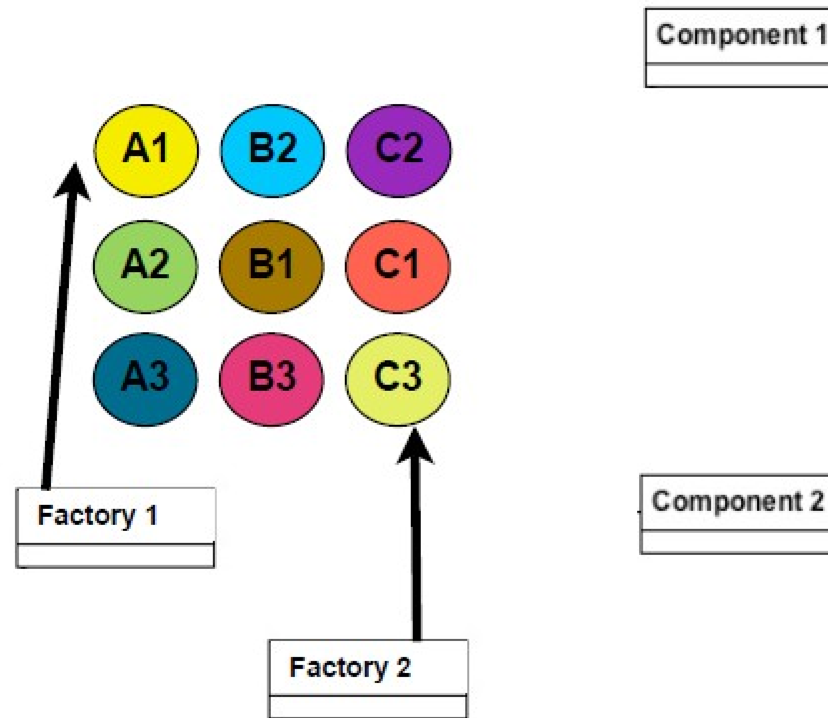


Products are more similar

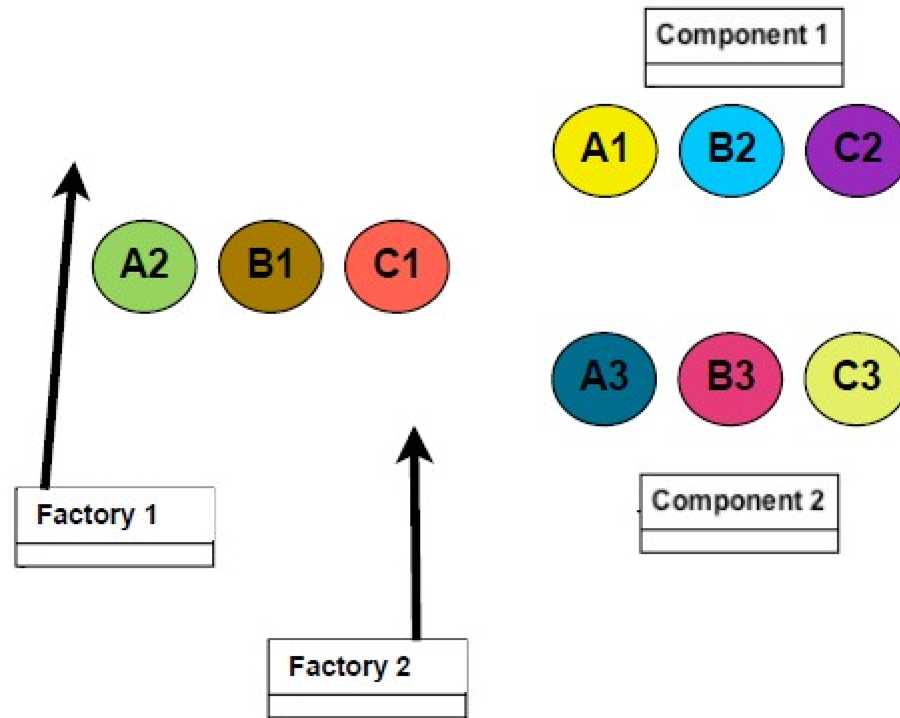




Alternative Approach



Alternative Approach



Observation

- Since only the components vary
  - Factor out this variation
  - We need 3 abstractions
    - One for product
    - Two for component factories

Abstract Factory Pattern (from slide 1)

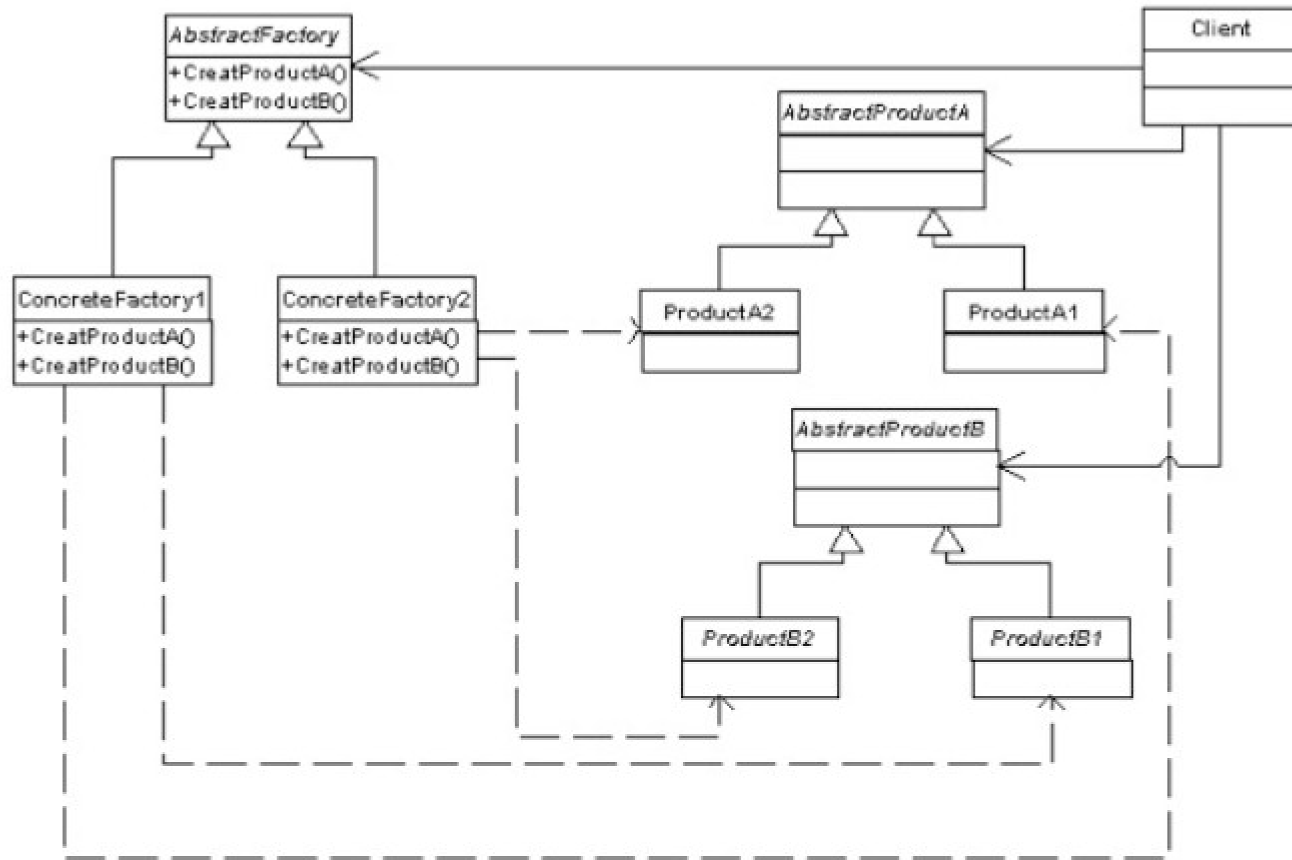
Provides an interface for creating families of related or dependent objects without specifying their concrete classes

This is done through object composition, rather than inheritance

Mental Note

Factory: Inheritance-based
Abstract Factory: Composition-based

Abstract Factory Class Diagram



Pizza Example Revisited: Abstract and Concrete Clients

```
public abstract class PizzaStore {
   protected abstract Pizza createPizza(String item);
   public Pizza orderPizza(String type) {
        Pizza pizza = createPizza(type);
        System.out.println("--- Making a " + pizza.getName() + " ---");
        pizza.prepare();
        pizza.bake();
        return pizza; } }

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");
        }
```

Abstract and Concrete Products

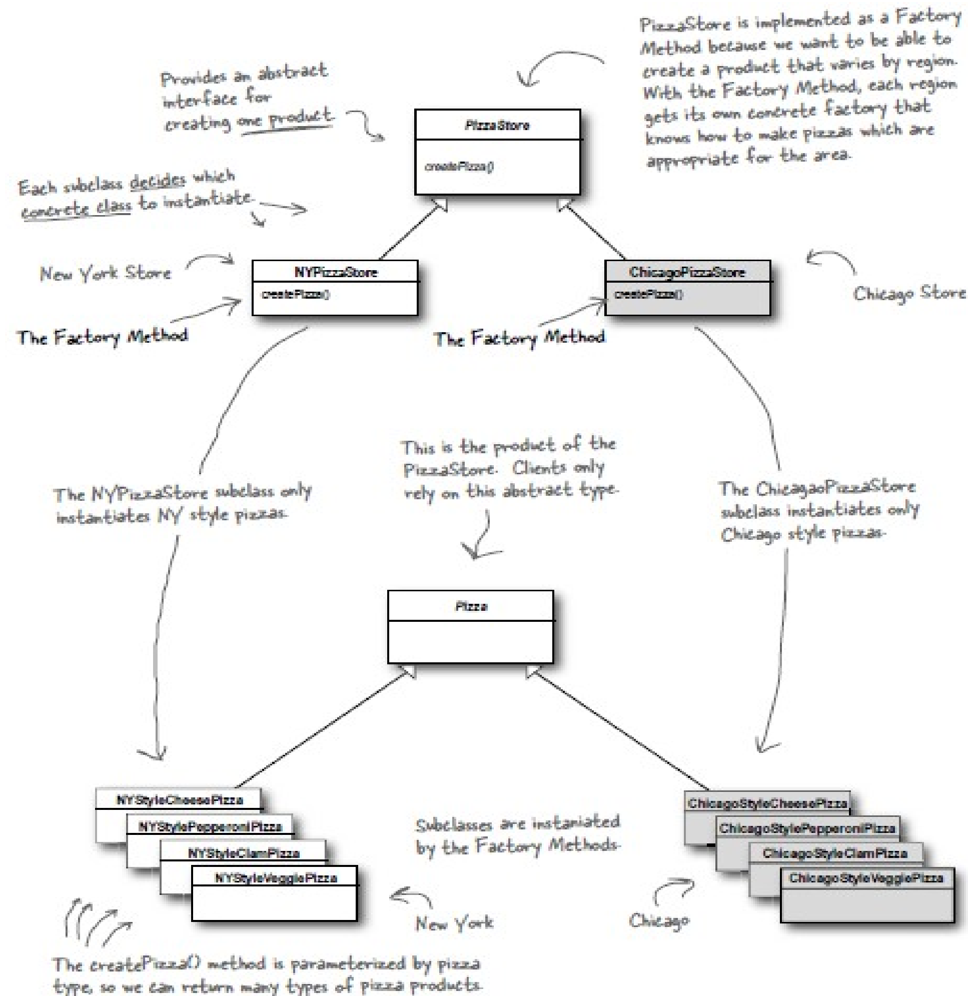
```
public abstract class Pizza {
   String name;
   Dough dough;
   Sauce sauce;
   abstract void prepare();
   void bake() {
        System.out.println("Bake for 25 minutes at 350");
   }

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();
   }
}
```

Abstract and Concrete Component Factory

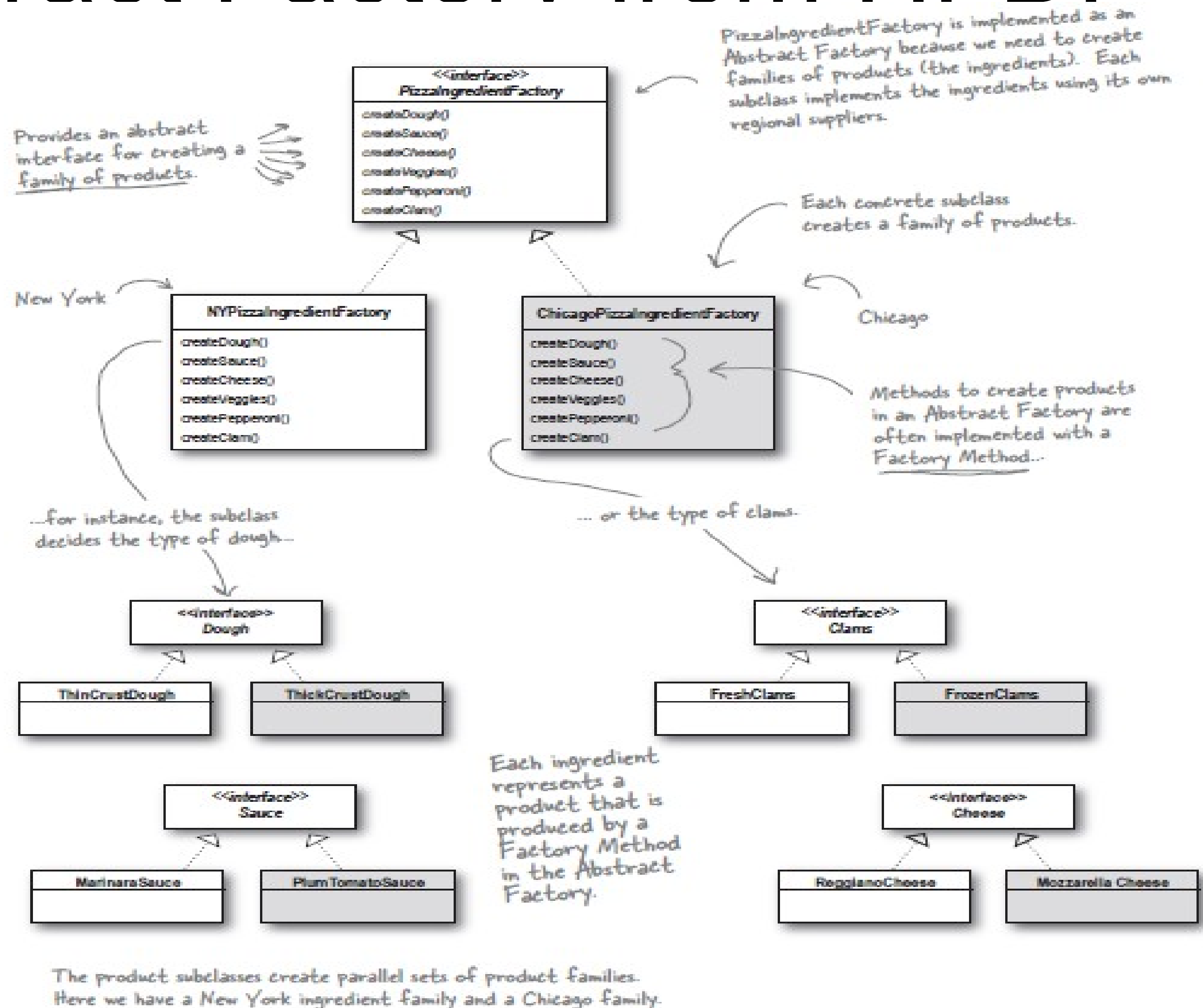
```java
public interface PizzaIngredientFactory {

   public Dough createDough();
   public Sauce createSauce();

}
```

```java
public class ChicagoPizzaIngredientFactory
   implements PizzaIngredientFactory
{
   public Dough createDough() {
       return new ThickCrustDough();
   }
```

Factory from HFDP



Abstract Factorv from HFDP



Summary

- All factories encapsulate object creation
- Simple Factory lets you decouple clients from concrete classes
- Factory Method relies on inheritance: object creation is delegated to subclasses
- Abstract Factory relies on object composition: object creation is implemented in methods exposed in the factory interface
- All factory patterns promote loose coupling
- The Dependency Inversion Principle guides us to avoid dependencies on concrete types and strive for abstractions