HACETTEPE UNIVERSITY
DEPARTMENT OF COMPUTER ENGINEERING

BBM 486 – DESIGN PATTERNS

5. THE FACTORY PATTERN

The 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.

Here is more to making objects than just using the new operator. You will learn that instantiation is an activity that should not always be done in public and can often lead to coupling problems. Factory pattern can help save you from embarrasing dependencies. We will illustrate the factory pattern with as example.

1. Let's say we have a pizza shop, and the pressure is on to add more pizza types. Initially, we might end up writing some code like this:

```
Pizza orderPizza() {
    Pizza pizza = new Pizza();

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

2. But you need more than one type of pizza… So, the you'd add some code that determines the appropriate type of pizza and then goes about making the pizza:

```
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;
    }
```

3. Assume that competitors have added a couple of trendy pizzas to their menus: the Clam Pizza and the Veggie Pizza. To keep up with the competition, you add these items to your menu. And you have not been selling many Greek Pizzas lately, so you decide to take that off the menu.

```
    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();
        } else if (type.equals("clam")) {
                pizza = new ClamPizza();
        } else if (type.equals("veggie")) {
                pizza = new VeggiePizza();
        }

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

   This requires more modifications on orderPizza(). Clearly, dealing with which concrete class is initiated is really messing up our orderPizza() method, and preventing it from being closed for modification. But now that we know what is varying and what is not, it is probably time to encapsulate it. We have got a name for this new object: we call it a Factoty.

4. Encapsulating object creation: We know we would be better off moving the object creation out of the orderPizza() method into another object that is only going to be concerned with creating pizzas. Define a class that encapsulates the object creation for all pizzas:

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

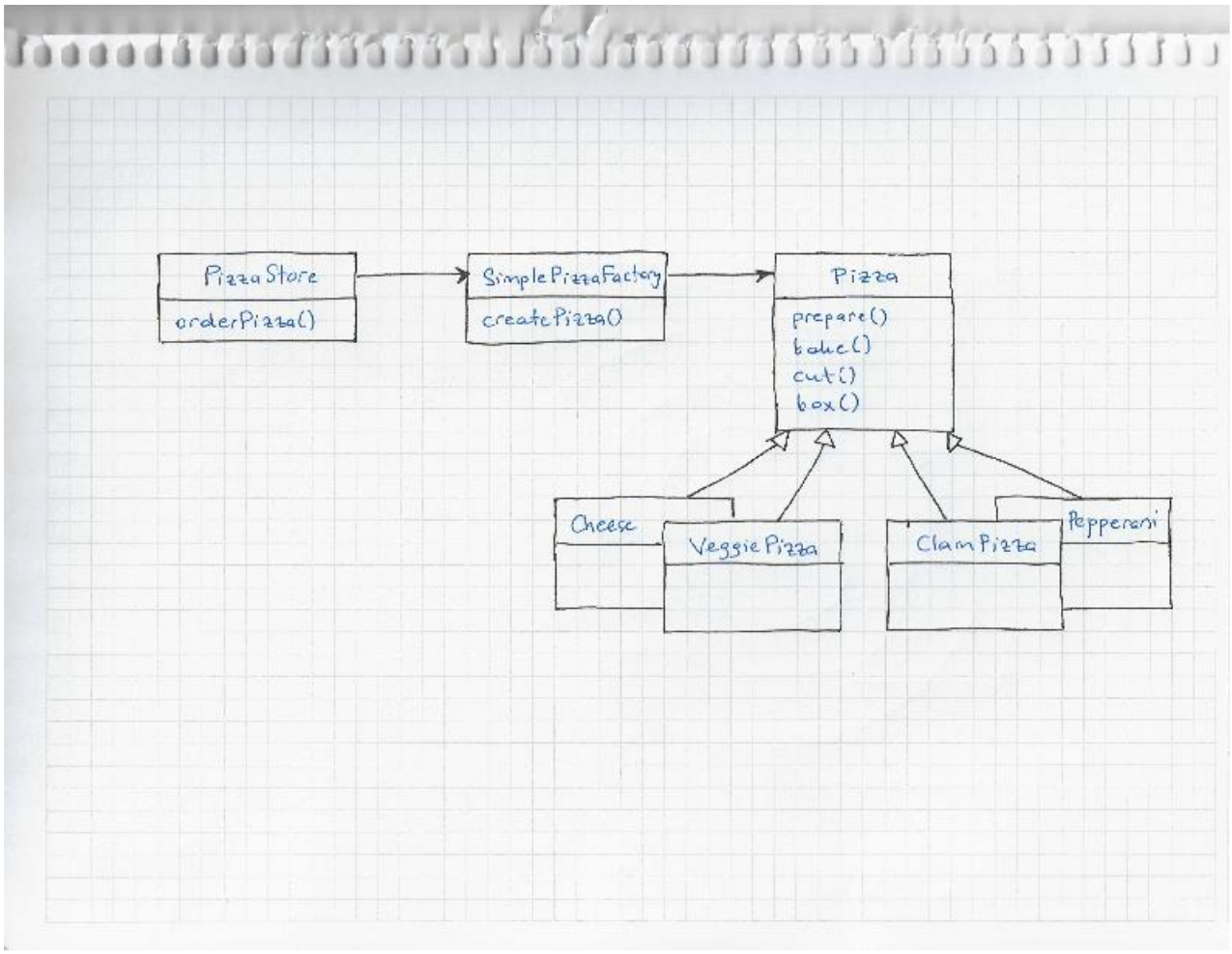
```
        if (type.equals("cheese")) {
            pizza = new CheesePizza();
        } else if (type.equals("pepperoni")) {
            pizza = new PepperoniPizza();
        } else if (type.equals("clam")) {
            pizza = new ClamPizza();
        } else if (type.equals("veggie")) {
        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

5. Reworking the PizzaStore class:

```
public 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;
    }

    // other methods here

}
```

6. **The Simple Factory defined**

   The Simple Factory is not actually a Design Pattern; it is more of a programming idiom.



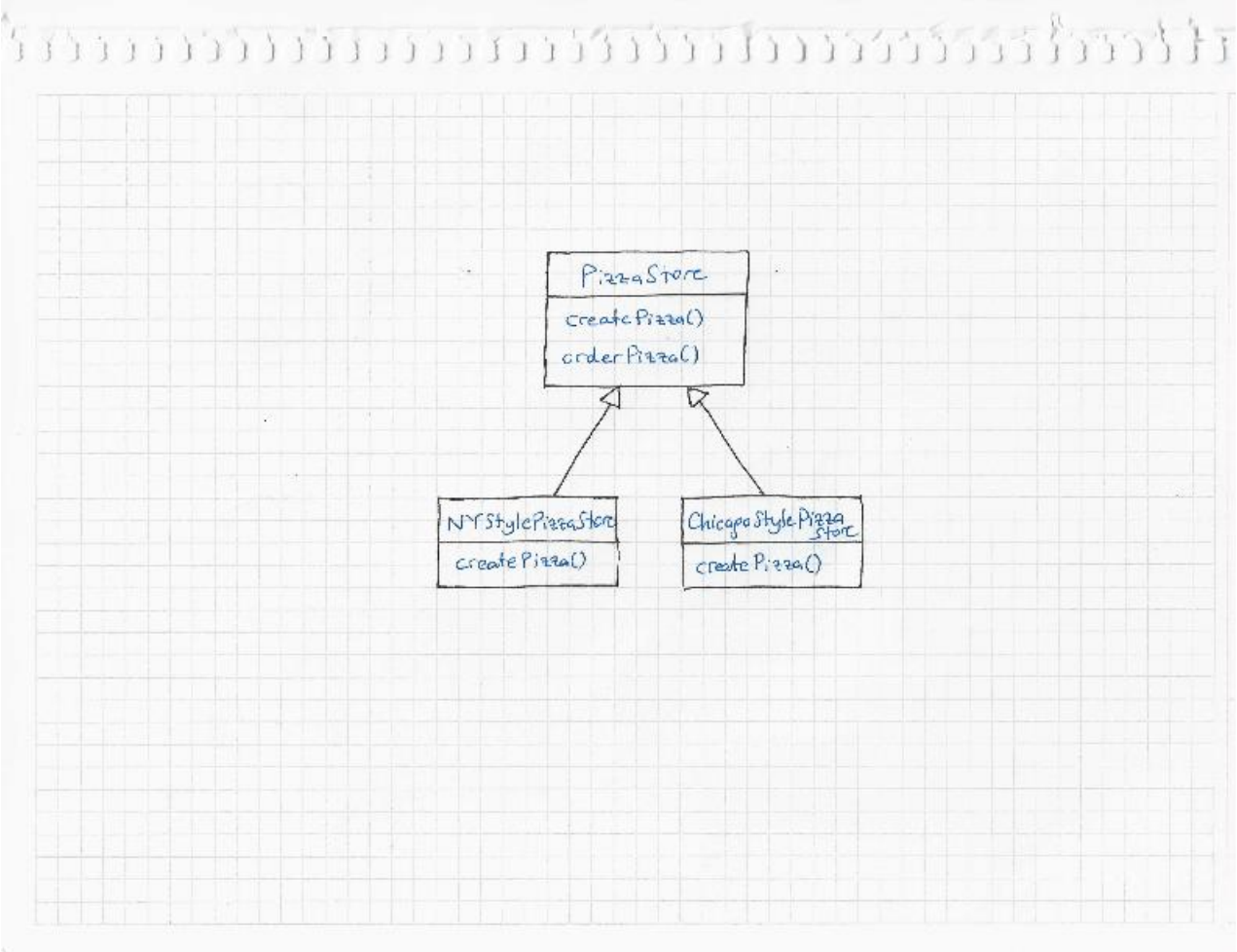
7. Franchising the pizza store: Your PizzaStore has done so well that everybody wants a PizzaStore in their own neighborhood. As the franchiser, you want to ensure the quality of the franchise operations and so you want them to use your time-tested code. But what about regional differences? As one approach, we can take SimplePizzaFactory and create different factories for each region:

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

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

8. The franchises were using the factory to create pizzas, but starting to employ their own home grown procedures for the rest of the process. Rethinking the problem, you decided to create a framework for the pizza store:

```
public abstract class PizzaStore {

    public Pizza orderPizza(String type) {
        Pizza pizza;

        pizza = createPizza(type);

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

        return pizza;
    }

    abstract Pizza createPizza(String type);

}
```

9. Allowing the subclasses to decide: What varies among the regional PizzaStores is the style of pizza they make. We are going to push all these variations into the createPizza() method, and make it responsible for creating the right kind of pizza.



10. Let's make a PizzaStore: All the regional stores need to do is subclass PizzaStore and supply a createPizza() method that implements their style of pizza.

```
public class NYPizzaStore extends PizzaStore {
    Pizza createPizza(String type) {
        if (item.equals("cheese")) {
            return new NYStyleCheesePizza();
        } else if (item.equals("veggie")) {
            return new NYStyleVeggiePizza();
        } else if (item.equals("clam")) {
            return new NYStyleClamPizza();
        } else if (item.equals("pepperoni")) {
            return new NYStylePepperoniPizza();
        } else return null;
    }
}
```

11. Declaring a factory method: We have gone from having an object handle the instantiation of our concrete classes to a set of subclasses that are now taking on that responsibility.

```
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);

    // other methods here

}
```

A factory method handles object creation and encapsulates it in a subclass. This decouples the client code in the superclass from the object creation code in the subclass.

abstract Product factoryMethod (String type)

12. Let's check out how these (NY syle and Chicago style) pizzas are really made to order…

First, we need a NY PizzaStore:

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

Then, we can take an order:

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

The orderPizza() method then calls the createPizza() method:

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

Finally, we have the unprepared pizza in hand and the orderPizza() method finishes preparing it:

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

Let’s implement the Pizza abstract 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");
    }

    public String getName() {
        return name;
    }
}
```

Now, we need some concrete subclasses:

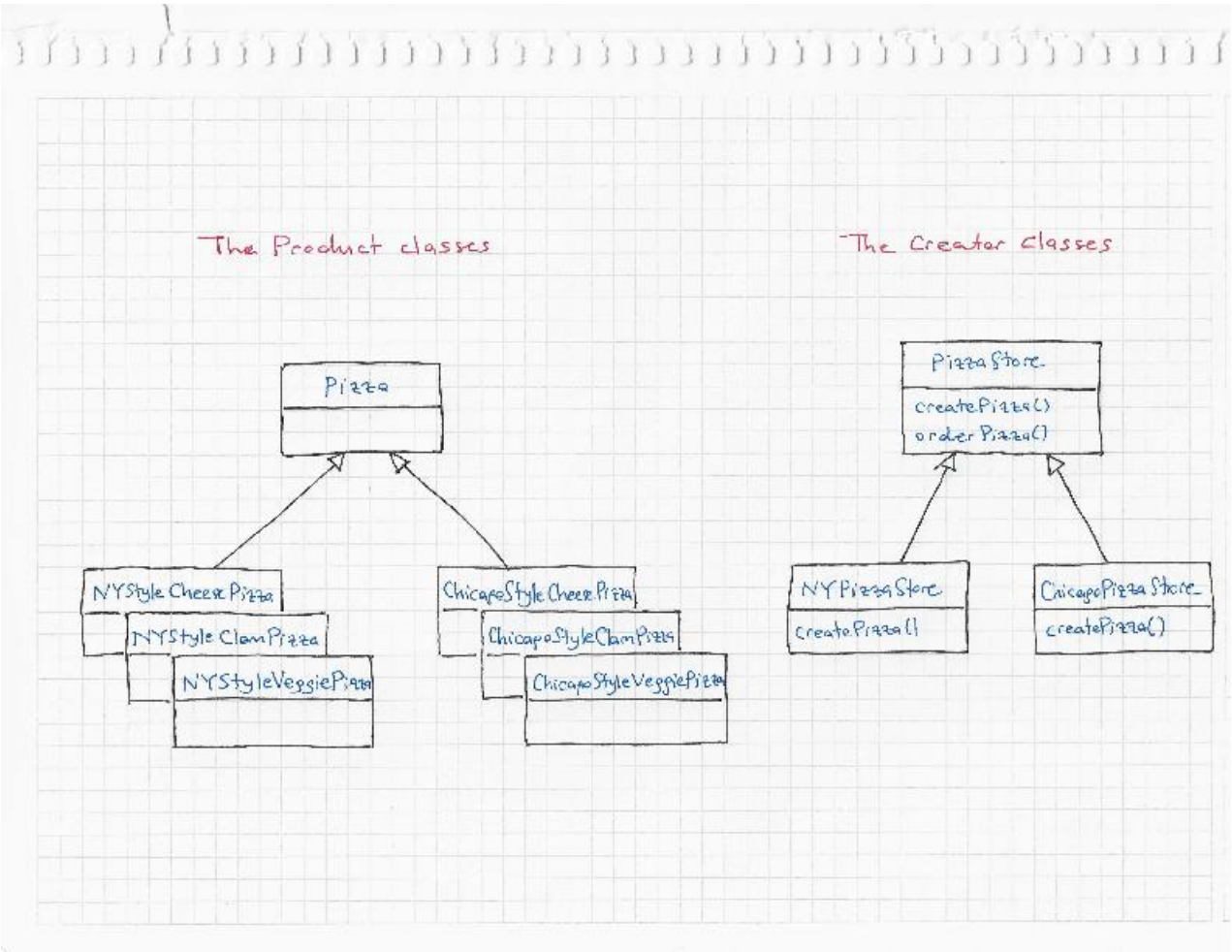
```
public class NYStyleCheesePizza extends Pizza {
    public NYStyleCheesePizza() {
        name = "NY Style Sauce and Cheese Pizza";
        dough = "Thin Crust Dough";
        sauce = "Marinara Sauce";

        toppings.add("Grated Reggiano Cheese");
```

```
        }
    }

    public class ChicagoStyleCheesePizza extends Pizza {
        public ChicagoStyleCheesePizza() {
            name = "Chicago Style Deep Dish 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");
        }
    }
```

And here is how we can test this:

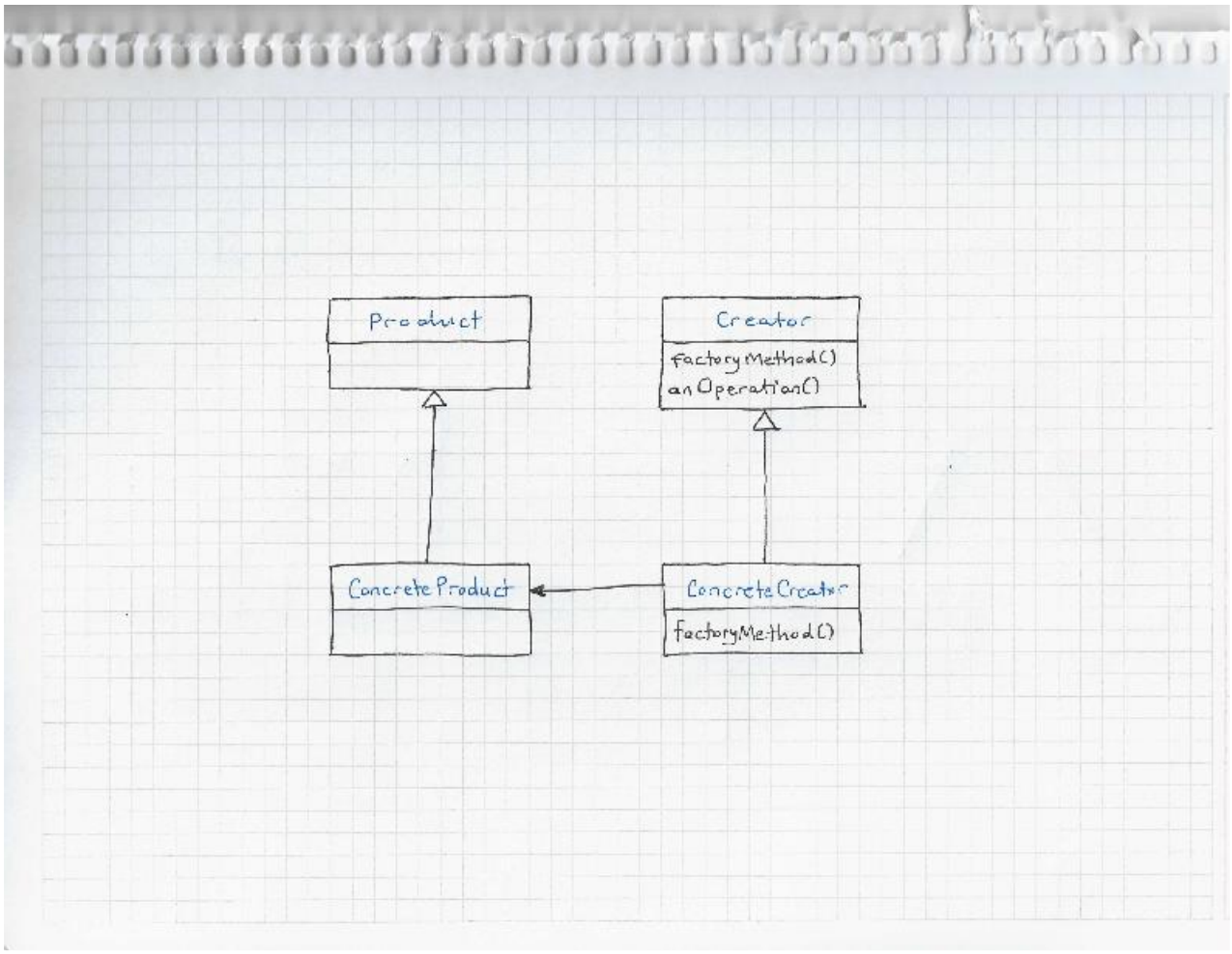
```
    public class PizzaTestDrive {

        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() + "\n");

            pizza = chicagoStore.orderPizza("cheese");
            System.out.println("Joel ordered a " + pizza.getName() + "\n");
        }
    }
```

13. It’s finally time to meet the Factory Method Pattern:



14. Factory Method Pattern defines an interface for creating an object, but lets the subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses.

    As with every factory, the Factory Method Pattern gives us a way to encapsulate the instantiations of concrete types.



15. 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 (type.equals("clam")) {
                pizza = new NYStyleClamPizza();
            } else if (type.equals("pepperoni")) {
                pizza = new NYStylePepperoniPizza();
            }
        } else if (style.equals("Chicago")) {
            if (type.equals("cheese")) {
```

```
                pizza = new ChicagoStyleCheesePizza();
            } else if (type.equals("veggie")) {
                pizza = new ChicagoStyleVeggiePizza();
            } else if (type.equals("clam")) {
                pizza = new ChicagoStyleClamPizza();
            } else if (type.equals("pepperoni")) {
                pizza = new ChicagoStylePepperoniPizza();
            }
        } else {
            System.out.println("Error: invalid type of pizza");
            return null;
        }
        pizza.prepare();
        pizza.bake();
        pizza.cut();
        pizza.box();
        return pizza;
    }
}
```

16. When you directly instantiate an object, you are depending on its concrete class. It should be pretty clear that reducing dependencies to concrete classes in our code is a good thing. We have got an OO design principle that formalizes this notion; *Dependency Inversion Principle*.

17. **Design Principle**: Depend upon abstractions. Do not depend upon concrete classes.

18. This is also called **dependency inversion principle**. This sound a lot like “Program to an interface, not an implementation,” but it makes an even stronger statement about abstraction. It suggests that out high-level components should not depend on our low-level components, rather, they should depend on abstractions.

19. The design for the PizzaStore is really shaping up: it has got a flexible framework and it does a good job of adhering to design principles. Now what you have discovered is that your franchises have been following your procedures, but a few franchises have been substituting inferior ingredients in their pies to lower costs. So how are you going to ensure each franchise is using quality ingredients?

20. Building the ingredient factories:

```
public interface PizzaIngredientsFactory {

    public Dough createDough();
    public Sauce createSauce();
    public Cheese createCheese();
    public Veggies[] createVeggies();
```

```
    public Pepperoni createPepperoni();
    public Clams createClam();
}
```

21. Building the New York and Chicago ingredient factories:

```
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();
    }
}
```

```
public class ChicagoPizzaIngredientFactory implements PizzaIngredientFactory
{

    public Dough createDough() {
        return new ThickCrustDough();
    }

    public Sauce createSauce() {
```

```
        return new PlumTomatoSauce();
    }

    public Cheese createCheese() {
        return new MozzarellaCheese();
    }

    public Veggies[] createVeggies() {
        Veggies veggies[] = { new BlackOlives(), new Spinach(), new Eggplant() };
        return veggies;
    }

    public Pepperoni createPepperoni() {
        return new SlicedPepperoni();
    }

    public Clams createClam() {
        return new FrozenClams();
    }
}
```

22. Reworking the pizzas…

```
public abstract class Pizza {
    String name;

    Dough dough;
    Sauce sauce;
    Veggies veggies[];
    Cheese cheese;
    Pepperoni pepperoni;
    Clams clam;

    abstract void prepare();

    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");
    }

    void setName(String name) {
        this.name = name;
    }

    String getName() {
        return name;
    }

    public String toString() {
            // code to print pizza here
    }
```

23. When we wrote the Factory Method code, we had a NYCheesePizza and a ChicagoCheesePizza class. If you look at the two classes, the only thing that differs is the use of regional differences. The pizzas are made just the same (dough + sauce + cheese). The same goes for the other pizzas: Veggie, Clam, and so on. They all follow the same preparation steps; they just have different ingredients. So, we really need two classes for each pizza; the ingredient factory is going to handle the regional differences for us.

```
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();
    }
}


public class ClamPizza extends Pizza {
    PizzaIngredientFactory ingredientFactory;

    public ClamPizza(PizzaIngredientFactory ingredientFactory) {
```

```
        this.ingredientFactory = ingredientFactory;
    }

    void prepare() {
        System.out.println("Preparing " + name);
        dough = ingredientFactory.createDough();
        sauce = ingredientFactory.createSauce();
        cheese = ingredientFactory.createCheese();
        clam = ingredientFactory.createClam();
    }
}
```

24. Revisiting our pizza stores

```
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;
    }
```

}

25. The Abstract Factory Pattern provides an interface for creating families of related or dependent objects without specifying their concrete classes. In this way, the client is decoupled from any of the specifics of the concrete products.