Outline

Review of patterns
Factory pattern
Decorator pattern
Talk only to your immediate friends.

Reduce interactions between objects

Avoid tight coupling between client and subsystem

Better structured and easier to read code

Applies to both OOP and SOA
One to many dependency between objects

- When source/subject/publisher state changes, all dependents/observer/subscribers are notified and updated automatically.
- There are many different ways to implement.
Defines a family of algorithms, encapsulates each one, and makes them interchangeable. Strategy lets the algorithm vary independently from clients that use it.
“Buzz words” that we have learned

Keep in mind those when you design your class

- Least knowledge
- Loosely coupled classes
- Program to an interface, not implementation
- Identify and encapsulate behaviors that vary (fly, quack)

What if the instantiation behaviors vary?

- Introducing the Factory pattern!
Pizza ordering application

```
Pizza orderPizza(String type) {
    Pizza pizza = new Pizza();
    pizza.prepare();
    pizza.bake();
    pizza.cut();
    pizza.box();
    return pizza;
}
```

- Programming to concrete classes, hard to change (add, remove pizza types)
- Based on the string parameter, create different types of pizza object
- All subclasses have the same “prepare”, “bake”, “cut”, “box” method
The object creation: factory

Factories handle the details of object creation
Are we just moving coconuts around? What’s the benefit of this?
Simple Factory

Simple factory isn’t a real pattern.

And it’s not “Factory pattern”
Your pizza business is expanding

At the same time, you want to make sure the stores follow the same procedure in the process: bake, cut, box...

You need a framework to scaffold the process while allow flexibility
Pizza store framework

What do we do with an abstract class? We extend it! Now pay attention to “orderPizza” method. It’s defined in superclass, but doesn’t know which subclass will run it.
Once a NYPizzaStore or ChicagoPizzaStore object is instantiated, the “orderPizza” will be executed the same without knowing about the sub-class
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);
}

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;
    }
}

We have talked about a lot about store, what about pizza?

Actually this is the beauty of using abstraction... Remember “Least Knowledge” Principle?
PizzaStore class isn’t aware of NYStyleCheesePizza or ChicagoStyleCheesePizza, but that doesn’t prevent it from doing its job.
Test the code

```java
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");
    }
}
```

*java PizzaTestDrive

Preparing NY Style Sauce and Cheese Pizza
Tossing dough...
Adding sauce...
Adding toppings:
  - Grated Regiano cheese
  - Bake for 25 minutes at 350
  - Cutting the pizza into diagonal slices
  - Place pizza in official PizzaStore box
Ethan ordered a NY Style Sauce and Cheese Pizza

Preparing Chicago Style Deep Dish Cheese Pizza
Tossing dough...
Adding sauce...
Adding toppings:
  - Shredded Mozzarella Cheese
  - Bake for 25 minutes at 350
  - Cutting the pizza into square slices
  - Place pizza in official PizzaStore box
Joel ordered a Chicago Style Deep Dish Cheese Pizza
Factory (creator) classes

This is our abstract creator class. It defines an abstract factory method that the subclasses implement to produce products.

Often the creator contains code that depends on an abstract product, which is produced by a subclass. The creator never really knows which concrete product was produced.

The `createPizza()` method produces products. Classes that produce products are called concrete creators.

Since each franchise gets its own subclass of PizzaStore, it's free to create its own style of pizza by implementing `createPizza()`.

**PizzaStore**
- `createPizza()`
- `orderPizza()`

**NYPizzaStore**
- `createPizza()`

**ChicagoPizzaStore**
- `createPizza()`
Product class

The Product classes

Factories produce products, and in the PizzaStore, our product is a Pizza.

These are the concrete products - all the pizzas that are produced by our stores.
Notice how these class hierarchies are parallel: both have abstract classes that are extended by concrete classes, which know about specific implementations for NY and Chicago.

The Product classes

Pizza

NYStyleCheesePizza
NYStylePepperoniPizza
NYStyleClamPizza
NYStyleVeggiePizza

ChicagoStyleCheesePizza
ChicagoStylePepperoniPizza
ChicagoStyleClamPizza
ChicagoStyleVeggiePizza

The Creator classes

PizzaStore
createPizza()
orderPizza()

NYPizzaStore
createPizza()

ChicagoPizzaStore
createPizza()
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.
“Straightforward” implementation

```java
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;
    }
}
```
The Dependency Inversion Principle

Depend upon abstractions. Do not depend upon concrete classes.

- Further step beyond “Program to an interface, not to an implementation”
- High-level components should not depend on low-level components
- Both high level and low level components should depend on abstractions
- Both PizzaStore and concrete pizza classes depend on Pizza abstraction
Homework: Add ingredients to pizza store

New York
- FreshClams
- MarinaraSauce
- ThinCrustDough
- ReggianoCheese

Chicago
- FrozenClams
- PlumTomatoSauce
- ThickCrustDough
- MozzarellaCheese

California
- Calamari
- BruschettaSauce
- VeryThinCrust
- GoatCheese
public interface PizzaIngredientFactory {
    public Dough createDough();
    public Sauce createSauce();
    public Cheese createCheese();
    public Veggies[] createVeggies();
    public Pepperoni createPepperoni();
    public Clams createClam();
}

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();
    }
}
Now the prepare step will be different, so make it abstract...

```java
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");
    }
    //...
}

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 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;
    }
}
Abstract Factory Pattern: an interface for creating families of related or dependent objects without specifying their concrete classes.