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
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

```java
Duck duck;

if (picnic) {
    duck = new MallardDuck();
}
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;
}
```
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;
}
Factor Out

```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;
}
```

factories handle the detail of object creation
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

Cannot exploit commonality

everything that varies goes here
Subclass the Factory
Example: Product Class
Creator Class

abstract

Factory
- make()
- do()

A Factory
- make()

B Factory
- make()

encapsulates knowledge about B

B 1
B 2
B 3
B 4
Benefits

<table>
<thead>
<tr>
<th>Client</th>
<th>Factory A</th>
<th>Factory B</th>
<th>Factory C</th>
<th>Factory D</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Object A</td>
<td>Object B</td>
<td>Object C</td>
<td>Object D</td>
</tr>
<tr>
<td></td>
<td>A_1</td>
<td>B_1</td>
<td>C_1</td>
<td>D_1</td>
</tr>
<tr>
<td></td>
<td>A_2</td>
<td>B_2</td>
<td>C_2</td>
<td>D_2</td>
</tr>
<tr>
<td></td>
<td>A_3</td>
<td>B_3</td>
<td>C_3</td>
<td>D_3</td>
</tr>
</tbody>
</table>

- only what varies in each family goes here
- commonalities in the family go here
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

- **Product**: Superclass that can be used to access all concrete products.
- **Creator**: Can be abstract. All concrete creators implement this method.
- **ConcreteProduct**
- **ConcreteCreator**
Example: Slot Machine

We'll follow the Factory Method to allow for creation of different kinds of slot machines.
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: " + slot2.getPrice());
    }
}
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

```java
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

```java
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()
}
```
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

Pizza is an abstract class... an abstraction.

PizzaStore now depends only on Pizza, the abstract class.

The concrete pizza classes depend on the Pizza abstraction too, because they implement the Pizza interface (remember, we're using "interface" in the general sense) in the Pizza abstract class.
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

```java
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

```java
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();
}

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

PizzaStore is implemented as a Factory Method because we want to be able to create a product that varies by region. With the Factory Method, each region gets its own concrete factory that knows how to make pizzas which are appropriate for the area.

Each subclass decides which concrete class to instantiate.

The NYPizzaStore subclass only instantiates NY style pizzas.

The createPizza() method is parameterized by pizza type, so we can return many types of pizza products.

Subclasses are instantiated by the Factory Methods.
Abstract Factory from HFDP

PizzaioloIngredientFactory is implemented as an Abstract Factory because we need to create families of products (the ingredients). Each subclass implements the ingredients using its own regional suppliers.

Each concrete subclass creates a family of products.

Methods to create products in an Abstract Factory are often implemented with a Factory Method.

For instance, the subclass decides the type of dough.

Or the type of clam.

Each ingredient represents a product that is produced by a Factory Method in the Abstract Factory.

The product subclasses create parallel sets of product families. Here we have a New York ingredient family and a Chicago family.
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