Java Design Patterns

Java Design Patterns

- Designing software for reuse is hard. One must find:
  - a good problem decomposition, and the right software
  - a design with flexibility, modularity and elegance
- designs often emerge from trial and error
- successful designs do exist
  - two designs they are almost never identical
  - they exhibit some recurring characteristics
- Can designs be described, codified or standardized?
  - this would short circuit the trial and error phase
  - produce "better" software faster

Java Design Patterns

- In 1994, ***Design Patterns: Elements of Reusable Object-Oriented Software*** by Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides explained the usefulness of patterns and resulted in the widespread popularity of design patterns.

- These four authors together are referred to as the Gang of Four (GoF).

- In this book, the authors documented the 23 patterns they found in their respective works.

Java Design Patterns

- Creational: address problems of creating an object in a flexible way. Separate creation, from operation/use.

- Structural: address problems of using O-O constructs like inheritance to organize classes and objects.

- Behavioral: address problems of assigning responsibilities to classes. Suggest both static relationships and patterns of communication

Java Design Patterns

CREATIONAL PATTERNS

1. Factory Method
2. Abstract Factory
3. Builder
4. Prototype
5. Singleton

STRUCTURAL PATTERNS

1. Adapter
2. Bridge
3. Composite
4. Decorator
5. Façade
6. Flyweight
7. Proxy

BEHAVIORAL PATTERNS

1.Chain of Responsibility
2.Command
3.Interpreter
4.Iterator
5.Mediator
6.Memento
7.Observer
8.State
9.Strategy
10.Template Method
11.Visitor

Java Design Patterns

- Program to an interface not an implementation.

- Favor object composition over inheritance.

Factory Pattern

- Word of the Day = Factory

- Different Flavors
  - Simple Factory
  - Factory Method
  - Abstract Factory

Factory Pattern

- We need a pizza store that can create pizza.
- The customer will order a specific type of pizza:
  - Cheese
  - Veggie
  - Greek
  - Pepperoni
  - etc
- Each order request is for one type of pizza only.

Factory Pattern

- During the ordering of a Pizza, we need to perform certain actions on it:
  - Prepare
  - Bake
  - Cut
  - Box

- We know that all Pizzas *must* perform these behaviors. In addition, we know that these behaviors will not change during runtime. (i.e. the Baking time for a Cheese Pizza will never change!)

- ***Question:*** Should these behaviors (prepare, bake, etc) be represented using Inheritance or Composition?

Factory Pattern

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

This method is responsible for creating the pizza.

It calls methods to prepare, bake, etc.

Pizza is returned to caller.

- Creating an instance of Pizza() doesn’t make sense here because we know there are different types of Pizza.

Factory Pattern

```
public Pizza orderPizza(String type) {
      Pizza pizza = null;
      if (type.equals("cheese")) {
            pizza = new CheesePizza();
      } else if (type.equals("pepperoni")) {
            pizza = new PepperoniPizza();

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

A parameter indicating type

Code that varies

Code that stays the same

Factory Pattern

- Pressure is on for change…
- Now we get some new types of Pizza (Clam, Veggie)
- Every time there is a change, we would need to break into this code and update the If/Else statement. (and possibly introduce bugs in our existing code).

Solution / Simple Factory

Move the creation of Pizzas into a separate object!

```
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;
   }
}
```

SimplePizzaFactory

- Advantage: We have one place to go to add a new pizza.

- Disadvantage: Whenever there is a change, we need to break into this code and add a new line. (but at least it is in one place!!)

Rework of PizzaStore

```
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;
   }
}
```

Store is *composed* of a factory.

Store is *composed* of a factory.

Simple Factory Defined

- This is not an official pattern, but it is commonly used.
- Not a bad place to start.
- When people think of “Factory”, they may actually be thinking of this.

Change Now Occurs…

- The PizzaStore is very popular and it needs to be franchised.
  - New York is interested
  - Chicago is interested
  - And perhaps one day Fairfax…

- Since PizzaStore is already composed of a Simple Factory, then this should be easy! Let's just create PizzaStore with a different SimpleFactory.

Example

```
//NY Pizza Factory has a different if/else logic
NYPizzaFactory nyFactory = new NYPizzaFactory();

//Create the Pizza Store, but use this Simple Factory
//instead

PizzaStore nyStore = new PizzaStore(nyFactory);

//Order pizza
nyStore.order(“Veggie”);
```

More change happens…

New York likes the PizzaStore, but they want to add more functionality, such as schedule a delivery.

New York attempts to extend PizzaStore…

```
public class NYPizzaStore extends PizzaStore {

  public void ScheduleDelivery() {
        …
  }
}
```

More change happens… (cont)

```
NYPizzaFactory nyFactory = new NYPizzaFactory();
NYPizzaStore nyStore = new NYPizzaStore(nyFactory);

//Order pizza
nyStore.order(“Veggie”);
myStore.ScheduleDelivery();
```

New York says the following:

- We only have one way to create pizzas; therefore, we don’t *need* to use composition for the pizza creation.

- We are not happy that we have to create our extended Pizza store *and* create a unique factory for creating pizzas. These two classes have a one-to-one relationship with each other. Can’t they be combined??

What New York wants

A framework so that NY can do the following:

- Create pizzas in a NY style
- Add additional functionality that is applicable to NY only.

A Framework for Pizza Store

```
public abstract class PizzaStore {

   abstract Pizza createPizza(String item);

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

        pizza.prepare();
       . . .
        return pizza;
   }
}
```

- NOTE: We are using inheritance here to create the pizzas, not composition. Also, the constructor for PizzaStore has been removed.

NYPizzaStore

```
public class NYPizzaStore extends PizzaStore {

      Pizza createPizza(String item) {
            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;
      }

      void ScheduleDelivery();
}
```

- The subclass is defining how the pizza is created….and it is also providing unique functionality that is applicable to New York.

Factory Method

```
public abstract class PizzaStore {

   abstract Pizza createPizza(String item);

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

       pizza.prepare();
       . . .
       return pizza;
   }
}
```

- Factory Method simply sets up an interface for creating a Product (in this case, a type of Pizza). Subclasses decide which specific Product to create.

Test Drive

1) PizzaStore nyStore = new NYPizzaStore();

2) nyStore.orderPizza("cheese");

3)Calls createPizza(“cheese”). The NYPizzaStore version of createPizza is called. It returns a NYStyleCheesePizza();

4) orderPizza continues to call the “parts that do not vary”, such as prepare, bake, cut, box.