Java Design Patterns

Outline

- Introduction to Design Patterns
  - Pattern's Elements
  - Types of Design Patterns
- Java Design Patterns
  - The Factory Pattern
  - The Abstract Factory Pattern
  - The Builder Pattern
  - The Prototype Pattern
  - The Singleton Pattern
  - The Adapter Pattern
  - The Bridge Pattern
  - The Composite Pattern
  - Java BluePrints Patterns Catalog

Design Patterns
:: What is a Design Pattern?

“Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice.” [1]

[Christopher Alexander]

Design patterns capture the *best practices of experienced object-oriented software developers.*

Design patterns are solutions to general software development problems.

Design Patterns
:: Pattern’s Elements – Pattern Name

In general, a pattern has four essential elements.

- The pattern name
- The problem
- The solution
- The consequences

Design Patterns

:: Pattern's Elements – The Pattern Name

The **pattern name** is a handle we can use to describe a design problem, its solutions, and consequences in a word or two.

- Naming a pattern immediately increases the design vocabulary. It lets us design at a higher level of abstraction.
- Having a vocabulary for patterns lets us talk about them.
- It makes it easier to think about designs and to communicate them and their trade-offs to others.

Design Patterns

:: Pattern's Elements – The Problem

The **problem** describes when to apply the pattern.

- It explains the problem and its context.
- It might describe specific design problems such as how to represent algorithms as objects.
- It might describe class or object structures that are symptomatic of an inflexible design.
- Sometimes the problem will include a list of conditions that must be met before it makes sense to apply the pattern.

Design Patterns
:: Pattern's Elements – The Solution

The **solution** describes the elements that make up the design, their relationships, responsibilities, and collaborations.

- The solution doesn't describe a particular concrete design or implementation, because a pattern is like a template that can be applied in many different situations.
- Instead, the pattern provides an abstract description of a design problem and how a general arrangement of elements (classes and objects in our case) solves it.

Design Patterns
:: Pattern's Elements – The Consequences

The **consequences** are the results and trade-offs of applying the pattern.

- The consequences for software often concern space and time trade-offs.
- They may address language and implementation issues as well.
- Since reuse is often a factor in object-oriented design, the consequences of a pattern include its impact on a system's flexibility, extensibility, or portability.
- Listing these consequences explicitly helps you understand and evaluate them

Design Patterns
:: Types

Erich Gamma, Richard Helm, Ralph Johnson and John Vlisides in their **Design Patterns** book define 23 design patterns divided into three types:

- ***Creational patterns*** are ones that create objects for you, rather than having you instantiate objects directly. This gives your program more flexibility in deciding which objects need to be created for a given case.
- ***Structural patterns*** help you compose groups of objects into larger structures, such as complex user interfaces or accounting data.
- ***Behavioral patterns*** help you define the communication between objects in your system and how the flow is controlled in a complex program.

Java Design Patterns
:: Why Use Patterns with Java?

- They have been proven. Patterns reflect the experience, knowledge and insights of developers who have successfully used these patterns in their own work.
- They are reusable. Patterns provide a ready-made solution that can be adapted to different problems as necessary.
- They are expressive. Patterns provide a common vocabulary of solutions that can express large solutions succinctly.
- J2EE provides built in patterns.

Java Design Patterns
:: Creational Patterns and Java I

- The creational patterns deal with the best way to create instances of objects.
- In Java, the simplest way to create an instance of an object is by using the **new** operator.

```
Fred = new Fred();  //instance of Fred class
```

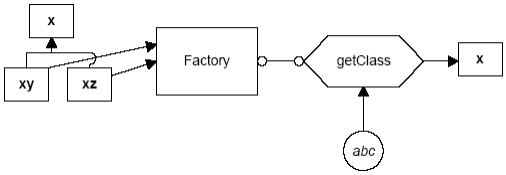
- This amounts to hard coding, depending on how you create the object within your program.
- In many cases, the exact nature of the object that is created could vary with the needs of the program and abstracting the creation process into a special "creator" class can make your program more flexible and general.

Java Design Patterns
:: Creational Patterns and Java II

- **The Factory Pattern** provides a simple decision making class that returns one of several possible subclasses of an abstract base class depending on the data that are provided.
- **The Abstract Factory Pattern** provides an interface to create and return one of several families of related objects.
- **The Builder Pattern** separates the construction of a complex object from its representation.
- **The Prototype Pattern** starts with an initialized and instantiated class and copies or clones it to make new instances rather than creating new instances.
- **The Singleton Pattern** is a class of which there can be no more than one instance. It provides a single global point of access to that instance.

The Factory Pattern
:: How does it Work?

The Factory pattern returns an instance of one of several possible classes depending on the data provided to it.



- Here, **x** is a base class and classes **xy** and **xz** are derived from it.
- The **Factory** is a class that decides which of these subclasses to return depending on the arguments you give it.
- The ***getClass()*** method passes in some value *abc*, and returns some instance of the class **x**. Which one it returns doesn't matter to the programmer since they all have the same methods, but different implementations.

The Factory Pattern
:: The Base Class

- Let's consider a simple case where we could use a Factory class. Suppose we have an entry form and we want to allow the user to enter his name either as *"firstname lastname"* or as *"lastname, firstname"*.
- Let's make the assumption that we will always be able to decide the name order by whether there is a comma between the last and first name.

```
class Namer { //a class to take a string apart into two names
        protected String last;  //store last name here
        protected String first; //store first name here
        public String getFirst() {
                return first;    //return first name
        }
        public String getLast() {
                return last;     //return last name
        }
}
```

The Factory Pattern
:: The First Derived Class

In the *FirstFirst* class, we assume that everything before the last space is part of the first name.

```
class FirstFirst extends Namer {
    public FirstFirst(String s) {
        int i = s.lastIndexOf(" ");   //find separating space
        if (i > 0) {
            first = s.substring(0, i).trim(); //left = first name
            last =s.substring(i+1).trim(); //right = last name
        } else {
            first = "" // put all in last name
            last = s; // if no space
        }
    }
}
```


The Factory Pattern
:: The Second Derived Class

In the *LastFirst* class, we assume that a comma delimits the last name.

```
class LastFirst extends Namer { //split last, first
   public LastFirst(String s) {
      int i = s.indexOf(","); //find comma
         if (i > 0) {
            last = s.substring(0, i).trim(); //left= last name
            first = s.substring(i + 1).trim(); //right= first name
         } else {
            last = s; // put all in last name
            first = ""; // if no comma
         }
    }
}
```

The Factory Pattern
:: Building the Factory

The Factory class is relatively simple. We just test for the existence of a comma and then return an instance of one class or the other.

```
class NameFactory {
    //returns an instance of LastFirst or FirstFirst
    //depending on whether a comma is found
    public Namer getNamer(String entry) {
        int i = entry.indexOf(","); //comma determines name order
        if (i>0)
            return new LastFirst(entry); //return one class
        else
            return new FirstFirst(entry); //or the other
    }
}
```


The Factory Pattern
:: Using the Factory

```
NameFactory nfactory = new NameFactory();
String sFirstName, sLastName;
….
private void computeName() {
  //send the text to the factory and get a class back
  namer = nfactory.getNamer(entryField.getText());
  //compute the first and last names using the returned class
  sFirstName = namer.getFirst();
  sLastName = namer.getLast();
}
```

The Factory Pattern

:: When to Use a Factory Pattern

You should consider using a Factory pattern when:

- A class can't anticipate which kind of class of objects it must create.
- A class uses its subclasses to specify which objects it creates.
- You want to localize the knowledge of which class gets created.

There are several similar variations on the factory pattern to recognize:

- The base class is abstract and the pattern must return a complete working class.
- The base class contains default methods and is only subclassed for cases where the default methods are insufficient.
- Parameters are passed to the factory telling it which of several class types to return. In this case the classes may share the same method names but may do something quite different.

The Abstract Factory Pattern

:: How does it Work?

The Abstract Factory pattern is one level of abstraction higher than the factory pattern. This pattern returns one of several related classes, each of which can return several different objects on request. In other words, **the Abstract Factory is a factory object that returns one of several factories.**

One classic application of the abstract factory is the case where your system needs to support multiple "look-and-feel" user interfaces, such as Windows, Motif or Macintosh:

- You tell the factory that you want your program to look like Windows and it returns a GUI factory which returns Windows-like objects.
- When you request specific objects such as buttons, check boxes and windows, the GUI factory returns Windows instances of these visual interface components.

The Abstract Factory Pattern
:: A Garden Maker Factory?

Suppose you are writing a program to plan the layout of gardens. These could be annual gardens, vegetable gardens or perennial gardens. However, no matter which kind of garden you are planning, you want to ask the same questions:

- What are good border plants?
- What are good center plants?
- What plants do well in partial shade?

We want a base *Garden* class that can answer these questions:

```
public abstract class Garden {
    public abstract Plant getCenter();
    public abstract Plant getBorder();
    public abstract Plant getShade();
}
```


The Abstract Factory Pattern
:: The Plant Class

The *Plant* class simply contains and returns the plant name:

```
public class Plant {
     String name;
     public Plant(String pname) {
       name = pname; //save name
     }
     public String getName() {
       return name;
     }
}
```

The Abstract Factory Pattern

:: A Garden Class

A Garden class simply returns one kind of each plant. So, for example, for the vegetable garden we simply write:

```
public class VegieGarden extends Garden {
    public Plant getShade() {
        return new Plant("Broccoli");
    }
    public Plant getCenter() {
        return new Plant("Corn");
    }
    public Plant getBorder() {
        return new Plant("Peas");
    }
}
```


The Abstract Factory Pattern

:: A Garden Maker Class – The Abstract Factory

We create a series of *Garden* classes - ***VegieGarden***, ***PerennialGarden, and AnnualGarden,*** each of which returns one of several Plant objects. Next, we construct our **abstract factory** to return an object instantiated from one of these *Garden* classes and based on the string it is given as an argument:

```
class GardenMaker {
    //Abstract Factory which returns one of three gardens
    private Garden gd;
    public Garden getGarden(String gtype) {
        gd = new VegieGarden(); //default
        if(gtype.equals("Perennial"))
                gd = new PerennialGarden();
        if(gtype.equals("Annual"))
                gd = new AnnualGarden();
        return gd;
    }
}
```

The Abstract Factory Pattern

:: Consequences of Abstract Factory

- One of the main purposes of the **Abstract Factory** is that it isolates the concrete classes that are generated.
- The actual class names of these classes are hidden in the factory and need not be known at the client level at all.
- Because of the isolation of classes, you can change or interchange these product class families freely.
- Since you generate only one kind of concrete class, this system keeps you for inadvertently using classes from different families of products.
- While all of the classes that the Abstract Factory generates have the same base class, there is nothing to prevent some derived classes from having additional methods that differ from the methods of other classes.

?

What is the difference between Factory pattern and abstract factory pattern ?

Abdalla.Moustafa@ejust.edu.eg