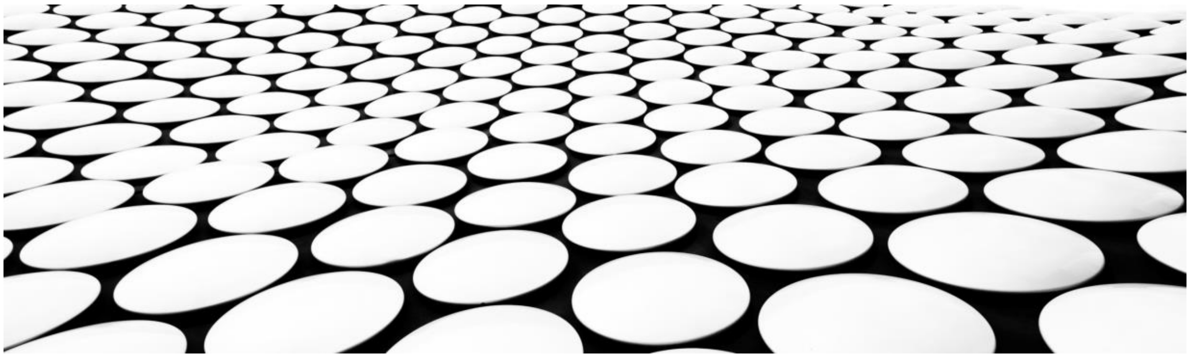
SOFTWARE DESIGN PATTERNS

CREATIONAL PATTERN



ABOUT THE COURSE

- This course is all about Design Patterns. In this course, we will present to you, the most useful and famous design patterns.
- In this lesson, first we will see what really are the Design Patterns. What is their use? Why one should really use them, and how to use them?
- Later, we will also see how patterns are organized, and categorized into different groups according to their behavior and structure.
- In the next several lessons, we will discuss about the different design patterns one by one. We will go into depth and analyze each and every design pattern, and will also see how to implement them in Java.

INTRODUCTION

- In the late 70's, an architect named Christopher Alexander started the concept of patterns. Alexander's work focused on finding patterns of solutions to particular sets of forces within particular contexts.
- Christopher Alexander was a civil engineer and an architect, his patterns were related to architects of buildings, but the work done by him inspired an interest in the object-oriented (OO) community.
- GOF (Gang of four) : *Design Patterns: Elements of Reusable Object-Oriented Software* (1994) is a software engineering book describing software design patterns. The book was written by Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides

WISH LIST AND OOP-PRINCIPLES

- loose coupling: 1 change = ceteris paribus
- code reuse (is not the same as copy/paste)
- open for extension, closed for modification
- encapsulate what varies
- single responsibility principle
- prefer composition over inheritance

SOFTWARE MODULE CHARACTERISTICS

- Coupling
  - Degree with which methods of different modules are dependent on each other
  - A loose coupling is good quality
- The open-closed principle states that a software module should be:
  - Open for extension — It should be possible to alter the behavior of a module or add new features to the module functionality.
  - Closed for modification — Such a module should not allow its code to be modified.

BECOMING A CHESS MASTER

- First learn rules and physical requirements
  - e.g., names of pieces, legal movements, chess board
- geometry and orientation, etc.
  - Then learn principles
  - e.g., relative value of certain pieces, strategic value of
- center squares, power of a threat, etc.
  - However, to become a master of chess, one must
- study the games of other masters
  - These games contain patterns that must be understood,
- memorized, and applied repeatedly
  - There are hundreds of these patterns

BECOMING A SOFTWARE DESIGNER MASTER

- First learn the rules
  - e.g., the algorithms, data structures and languages of software
- Then learn the principles
  - e.g., structured programming, modular programming, object oriented programming, generic programming, etc.
- However, to truly master software design, one must study the designs of other masters
  - These designs contain patterns must be understood, memorized, and applied repeatedly
- There are hundreds of these patterns

ECOSYSTEM OF PATTERNS

- Programming Patterns (idioms)
  - low-level pattern specific to a programming language. An idiom describes how to implement particular aspects of components or the relationships between them using the features of the given language. e.g. singleton, string copy in C (while (*d++=*s++);
- Design Patterns
  - A design pattern provides a scheme for refining the subsystems or components of a software system, or the relation ships between them. It describes a commonly-recurring structure of communicating components that solves a general design problem within a particular context.
- Architectural Patterns
  - fundamental structural organization schema for software systems. It provides a set of predefined subsystems, their responsibilities, and includes rules and guidelines for organizing the relationships between them.
  - e.g. layers, distribution, security, MVC...

WHAT ARE DESIGN PATTERNS?

- As an Object Oriented developer, we may think that our code contains all the benefits provided by the Object Oriented language. (flexibility, reusability, maintainability).Unfortunately, these advantages do not come by its own.
- Christopher had said that “Each pattern describes a problem that 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”.

- we can think of patterns as a formal document which contains recurring design problems and its solutions

WHAT ARE DESIGN PATTERNS? CONT.....

- In software engineering, a design pattern is a general repeatable solution to a commonly occurring problem in software design.
- Design patterns are “descriptions of communicating objects and classes that are customized to solve a general design problem in a particular context.”
- In general, a pattern has four essential elements:
  - ✓ Pattern name,
  - ✓ The problem: describes when to apply the pattern.
  - ✓ The solution: describes the elements that make up the design, their relationships, responsibilities, and collaborations
  - ✓ The results and consequences of applying the pattern: Costs and benefits of applying the pattern.

WHY DESIGN PATTERN

- Flexibility: Using design patterns your code becomes flexible. It helps to provide the correct level of abstraction due to which objects become loosely coupled to each other which makes your code easy to change.
- Reusability: Loosely coupled and cohesive objects and classes can make your code more reusable. This kind of code becomes easy to be tested as compared to the highly coupled code.
- Design patterns offer a common dictionary between developers ,it allow developers to communicate using well-known, well understood names for software interactions.
- Capture best practices: Design patterns capture solutions which have been successfully applied to problems. By learning these patterns and the related problem, an inexperienced developer learns a lot about software design.
- simply,
- Design patterns help a designer get a design "right" faster.

HOW TO SELECT AND USE ONE

- Have a very deep understanding of them in order to implement the correct design pattern for the specific design problem.
- First, you need to identify the kind of design problem you are facing. A design problem can be categorized into creational, structural, or behavioral. Based to this category you can filter the patterns and selects the appropriate one.
- Recognizing when and where to use design patterns requires familiarity & experience

DESIGN PATTERNS CLASSIFICATION

Creational Patterns:
are concerned with the process of object creation

Structural Patterns:
are concerned with how classes and objects are composed to form larger structures

Behavioural Patterns:
are concerned with algorithms and the assignment of responsi-bilities between objects

Class Patterns deal with static relationships between classes and subclasses

Object Patterns deal with object relationships which can be changed at run time

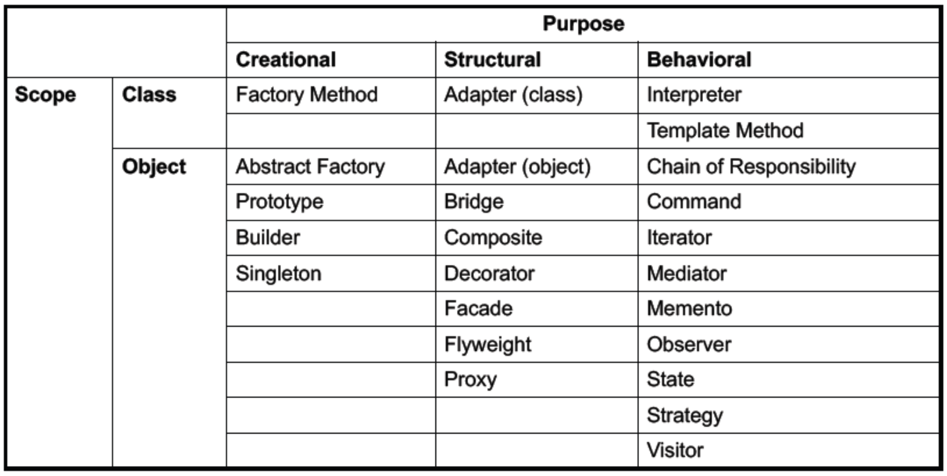
CLASSIFICATION OF PATTERNS BASED ON PURPOSE

The GOF book defines 3 major types of patterns

Creational patterns	• 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. • is particularly useful for making independently developed class libraries work together.
Behavior patterns	• help you define the communication between objects in your system and how the flow is controlled in a complex program.

GOF DESIGN PATTERNS

- based on book of Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, Elements of Reusable Object-Oriented Software



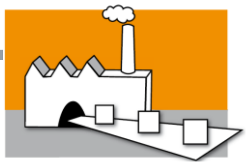
		Purpose		
		Creational	**Structural**	**Behavioral**
Scope	**Class**	Factory Method	Adapter (class)	Interpreter
				Template Method
	Object	Abstract Factory	Adapter (object)	Chain of Responsibility
		Prototype	Bridge	Command
		Builder	Composite	Iterator
		Singleton	Decorator	Mediator
			Facade	Memento
			Flyweight	Observer
			Proxy	State
				Strategy
				Visitor

CREATIONAL PATTERNS

- An important point of writing code in Object Oriented Programming is Object/Instance creation. Any program requires a considerable amount of object creation. Wouldn't it be nice if we had some standard ways to create objects? Hence, came the creational design patterns. These provide standardized pathways to create instances.
- These patterns control the way we define and design the objects, as well as how we instantiate them. Some encapsulate the creation logic away from users and handles creation (Factory and Abstract Factory), some focus on the process of building the objects themselves (Builder), some minimize the cost of creation (Prototype) and some control the number of instances on the whole JVM (Singleton).

CREATIONAL PATTERNS

- 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 - Encapsulate a set of analogous factories that produce families of objects.
- The Builder Pattern - Encapsulate the construction of complex objects from their representation; so, the same building process can create various representations by specifying only type and content.
- The Prototype Pattern - A fully initialized instance to be copied or cloned
- The Singleton Pattern - Ensure that only a single instance of a class exists and provide a single method for gaining access to it

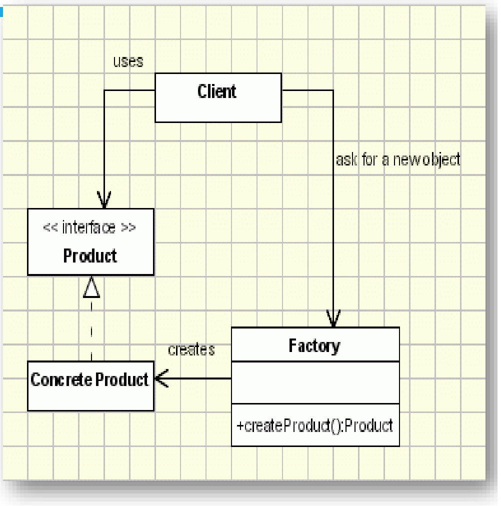


FACTORY METHOD

- A Factory design pattern also known as the “Factory Method pattern” is a type of Creational design patterns. By the name we can guess it produces or creates something, in our case objects.
- Advantage of Factory Design Pattern
  - Factory Method Pattern allows the sub-classes to choose the type of objects to create.
- Where to use Factory Pattern
  - We need to create different type of objects
  - Object creation is dynamic – at run time we need to decide which object to be created
  - Different methods contain same object creation code

UML CLASS DIAGRAM

- Participants
- The classes and objects participating in this pattern are:
- **Product**
  - defines the interface of objects the factory method creates
- **Concrete Product**
  - implements the Product interface
- **factory**
  - declares the factory method, which returns an object of type Product.

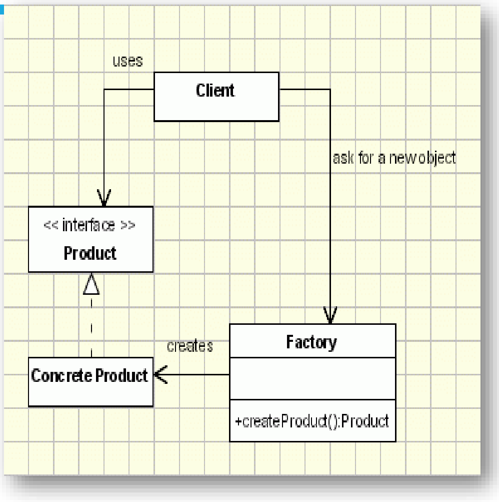


THE IMPLEMENTATION:

❑ **The client** needs a product, but instead of creating it directly using the new operator, it asks the factory object for a new product, providing the information about the type of object it needs.

❑ **The factory** instantiates a new concrete product and then returns to the client the newly created product.

❑ **The client** uses the products as abstract products without being aware about their concrete implementation.



FACTORY DESIGN PATTERN EXAMPLE WITH VEHICLES

- Step 1: Create an interface for Vehicle.

```
// Vehicle.java
public interface Vehicle {
    void drive();
}
```

- Step 2: Create concrete classes implementing the Vehicle interface

```
// Car.java
public class Car implements Vehicle {
    @Override
    public void drive() {
        System.out.println("Driving a car!");
    }
}

// Bike.java
public class Bike implements Vehicle {
    @Override
    public void drive() {
        System.out.println("Riding a bike!");
    }
}
```

- Step 3: Create the Factory class

```
// VehicleFactory.java
public class VehicleFactory {
    // This method returns an instance of Vehicle based on
the input type
    public Vehicle getVehicle(String vehicleType) {
        if (vehicleType == null) {
            return null;
        }
        if (vehicleType.equalsIgnoreCase("CAR")) {
            return new Car();
        } else if (vehicleType.equalsIgnoreCase("BIKE")) {
            return new Bike();
        }
        return null;
    }
}
```

STEP 4: USE THE FACTORY CLASS TO GET OBJECTS OF DIFFERENT TYPES OF VEHICLES

```
// FactoryPatternDemo.java
public class FactoryPatternDemo {
    public static void main(String[] args) {
        // Create the factory
        VehicleFactory vehicleFactory = new
VehicleFactory();

        // Get an object of Car and call its drive method
        Vehicle car = vehicleFactory.getVehicle("CAR");
        car.drive();  // Output: Driving a car!

        // Get an object of Bike and call its drive method
        Vehicle bike = vehicleFactory.getVehicle("BIKE");
        bike.drive();  // Output: Riding a bike!
    }
}
```

Output:

```
Driving a car!
Riding a bike!
```

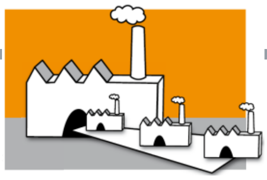
Explanation:

1. Vehicle interface: Declares the drive method.
2. Car and Bike are concrete classes that implement the drive method.
3. VehicleFactory is responsible for creating objects of Car or Bike based on the input string provided.
4. In the FactoryPatternDemo class, we use VehicleFactory to create objects of different vehicle types without needing to instantiate them directly in the client code.

BENEFITS AND LIMITATIONS OF FACTORY DESIGN PATTERN

- The client code doesn't directly instantiate objects using the `new` operator. Instead, the Factory takes the responsibility of determining which object to create based on the parameters it receives.
- This superclass defines the API the factory-produced objects will adhere to, maintaining uniformity.
- Benefits
  - Creation of different types of objects is possible at run time
  - It separates the object creation logic from the object usage logic
  - Removes duplicate code
  - Thus, makes changing or addition to object creation easier
- Limitations
  - The different types of objects created must have the same parent class
  - The addition of new classes and interfaces could increase the complexity of the code

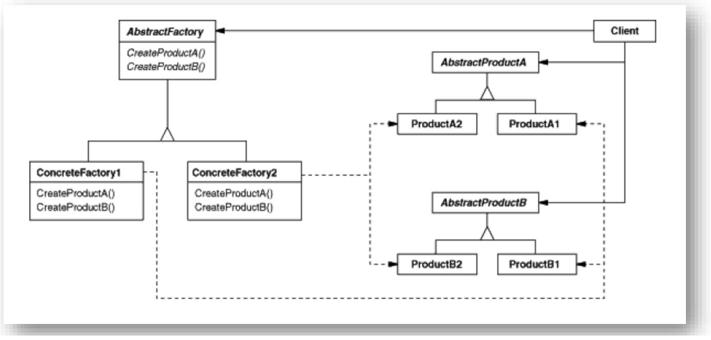
ABSTRACT FACTORY



- Abstract Factory pattern adds a "Factory" that brings together all these factories. Further, it decides at run time which factory should be invoked. This later creates an object of a specific family. Therefore, **this pattern is also known as "A Factory of Factories".**
- Abstract Factory Pattern in java encapsulates a group of factories in the process of object creation
- When can we use Abstract Factory Pattern?
- The system has multiple types (families) of objects
  - As a result, if we need different object or functions they should be from the same group
- The system needs to create or compose objects at run time according to the user input
- The system need different function but they should be in groups
  - If a(), b(), c() and p(),q(),r() two groups of functions, if we call a() then we can call either function b() or function c() only.

STRUCTURE

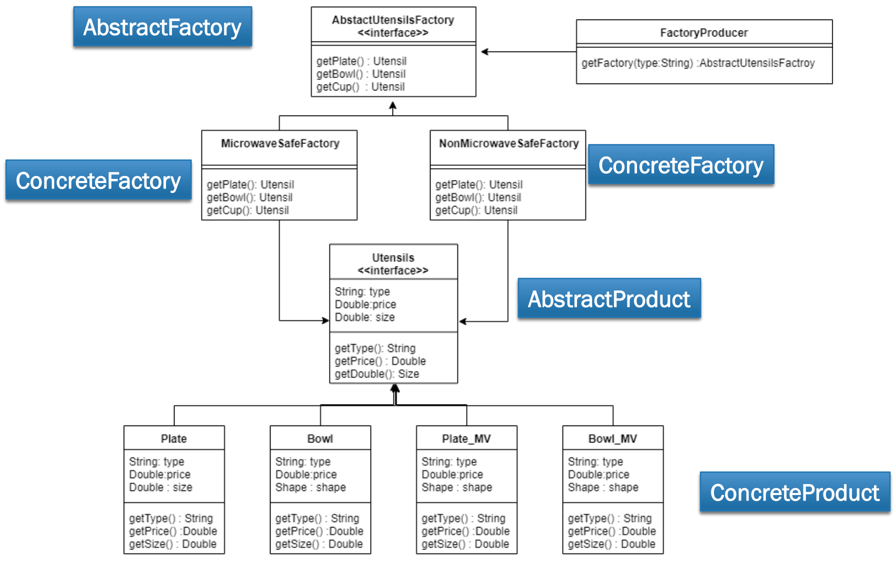
- AbstractFactory declares an interface for operations that create abstract product objects.
- ConcreteFactory implements the operations to create concrete product objects.
- AbstractProduct declares an interface for a type of product object.
- ConcreteProduct defines a product object to be created by the corresponding concrete factory. implements the *AbstractProduct* interface.
- Client uses only interfaces declared by *AbstractFactory* and *AbstractProduct* classes.



IMPLEMENT ABSTRACT FACTORY PATTERN

- For example we have two groups of utensils Microwave safe and non microwave safe products. If we need microwave safe products we should use microwave safe bowl, plate and cup. We can not mix microwave safe and non microwave safe. When we need to avoid mixing we can use abstract factory pattern by creating factories for each.
- Implement Abstract Factory Pattern:
  - Find out the different object types in the application
  - Create Interface and implementing classes for each type
    For eg, Utensils : Plate, Bowl, Cup, Plate_MW, Bowl_MW, Cup_MW (_MW is used for microwave safe products)
  - Create factory classes to group the classes
    - Microwave safe: Plate_MW, Bowl_MW, Cup_MW
    - Non Microwave safe: Plate, Bowl, Cup
  - Declare Abstract factory interface and declare all required methods from factory
  - Implement Abstract factory interface by created families
  - Create code which will use Abstract factory to get factory and then call the methods on that factory
  - Use the abstract factory in the code instate of objects directly

UML : BASIC UNDERSTANDING OF THE IMPLEMENTATION



```
public interface Utensil{
    public String getType();
    public Double getPrice();
    public Double getSize();
}
```

```
public class Bowl implements Utensil{
  ... code with method implementations
}

public class Bowl_MW implements Utensil{
  ... code with method implementations
}

public class Cup_MW implements Utensil{
  ... code with method implementations
}
```

```
public class Plate implements Utensil{
String type;
Double Prise;
Double size;
            public Plate() {
                        this.type = "PALTE";
                        this.Prise= 8.50;
                        this.size= 15.00;
            }
 @Override
       public String getType(){
         return type;
     }
   @Override
       public Double getPrise(){
         return Prise;
     }
   @Override
        public Double getSize(){
         return size;
     }}
```

- We have two different type of utensils microwave safe and non microwave safe, as a result, we create factory classes

```
public class MicrowaveSafeFactory {
public Utensil getPlate() {
return new Plate();
}
public Utensil getBowl() {
return new Bowl();
}
public Utensil getCup() {
return new Cup();
}
}
```

```
public class NonMicrowaveSafeFactory {
public Utensil getPlate() {
return new Plate_MW();
}
public Utensil getBowl() {
return new Bowl_MW();
}
public Utensil getCup() {
return new Cup_MW();
}
}
```

- we will create our AbstractUtensilFactory interface. It groups the other factories together.

```
public interface AbstractUtensilFactory {
public Utensil getPlate();
 public Utensil getBowl();
}
```

- Now we will implement the AbstractUtensilFactory interface by the factories created earlier.

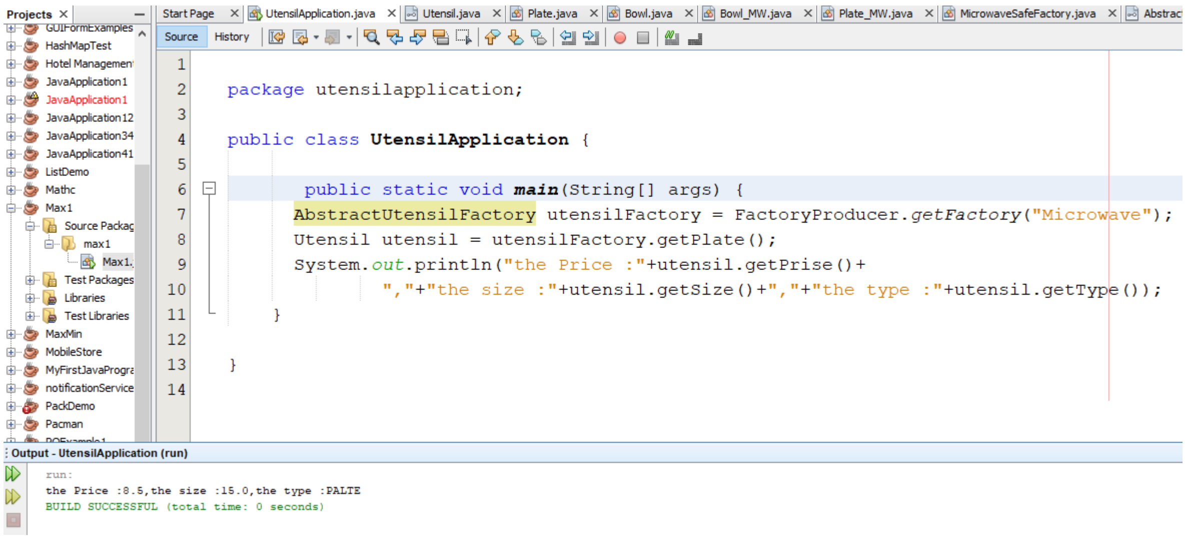
```
public class MicrowaveSafeFactory implements
AbstractUtensilFactory{
public Utensil getPlate() {
return new Plate();
}

public Utensil getBowl() {
return new Bowl();
}}
```

```
public class NonMicrowaveSafeFactory implements
AbstractUtensilFactory{
public Utensil getPlate() {
return new Plate_MW ();
}

public Utensil getBowl() {
return new Bowl_MW();
}}
```

- Finally, all code is in place and we need to use it. Because, we have two factories, we need to get the appropriate factory first. Once we get the factory we can call the require methods from the factory . Let's create a class FactoryProducer which will provide the factory instance as per the requirement.

```
public class FactoryProducer {
            public static AbstractUtensilFactory getFactory(String choice){

      if("Microwave".equalsIgnoreCase(choice)){
         return new MicrowaveSafeFactory();
      }
      else if("Non-Microwave".equalsIgnoreCase(choice)){
         return new NonMicrowaveSafeFactory();
      }
      return null;
   }
}
```

- Now, we can start using the produces and factory classes where ever we need.



```
package utensilapplication;

public class UtensilApplication {

      public static void main(String[] args) {
     AbstractUtensilFactory utensilFactory = FactoryProducer.getFactory("Microwave");
     Utensil utensil = utensilFactory.getPlate();
     System.out.println("the Price :"+utensil.getPrise()+
             ","+"the size :"+utensil.getSize()+","+"the type :"+utensil.getType());
    }

}
```

Output - UtensilApplication (run)

```
run:
the Price :8.5,the size :15.0,the type :PALTE
BUILD SUCCESSFUL (total time: 0 seconds)
```

BENEFITS AND LIMITATIONS OF ABSTRACT FACTORY DESIGN PATTERN

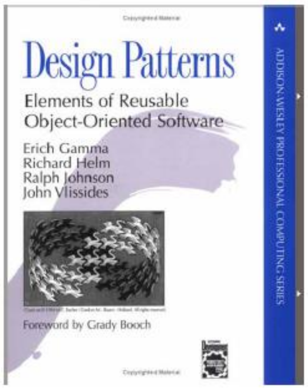
- Benefits of using abstract factory pattern
  - Firstly, it helps to group related objects or functions
  - Also, reduces errors of mixing of objects or functions from different groups
  - Helps to abstract code so that user don’t need to worry about object creations
- Limitations
  - Only useful when we have to group processes or objects
  - Before getting object or calling the function we need to get the factory which adds one more processes
  - Adds more classes and abstraction hence code could become complex

CONCLUSION

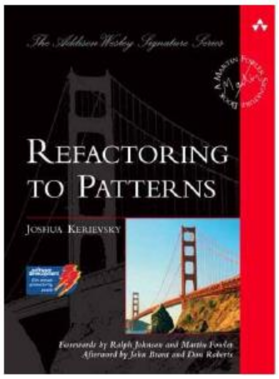
- Software Design Pattern Descriptions of reusable solutions to common software design problems.
- Classification of design Pattern (Creational, Structural, behavioral).
- Factory Design Pattern
  - Also, called as **Factory Method pattern**
  - Most commonly used design pattern
  - Further, enables to create objects by deciding the type of object at run time
  - Then, helps to isolate the object creation logic to a single method
  - This method can be further overridden by different classes
  - Hence avoids repetition of code
- Abstract Factory Design Pattern
  - Most important point, it is a type of “Creational design pattern”
  - Mange different object types of same family
  - Also known as “**Factory of factories**”
  - It is different from the Factory pattern because it has multiple functions

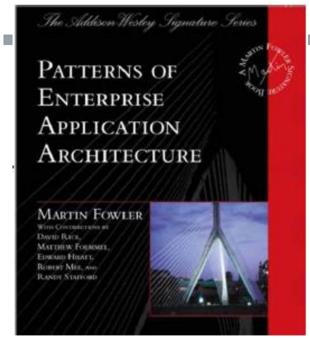
SOME BOOKS

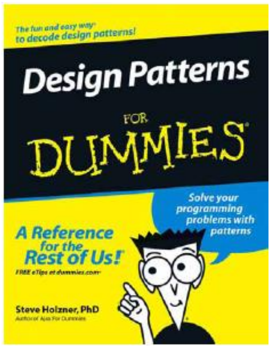
- GOF: 23 "classical" patterns:



classic,
The Book







QUESTIONS......

Good read about Design Patterns

- http://en.wikipedia.org/wiki/Design_pattern_(computer_science)
- http://sourcemaking.com/design_patterns
- http://java.sun.com/blueprints/patterns/index.html
- http://www.codeproject.com/KB/architecture/#Design Patterns
- http://msdn.microsoft.com/en-us/magazine/cc301852.aspx
- http://www.javacamp.org/designPattern/
- http://www.javaworld.com/channel_content/jw-patterns-index.html
- http://www.ibm.com/developerworks/java/tutorials/j-patterns201