Java Patterns

Sang Shin
Java Technology Architect
Sun Microsystems, Inc.
sang.shin@sun.com
www.javapassion.com

Topics

- What is a pattern?
- Why patterns?
- Singleton pattern
- FactoryMethod pattern
- AbstractFactory pattern
- MVC pattern
- Observer pattern

What is a Pattern?

What is a Pattern?

- Patterns are a software engineering problem-solving discipline that emerged from the object-oriented community

Why Patterns?

- It solves a problem: Patterns capture solutions, not just abstract principles or strategies.
- It is a proven concept: Patterns capture solutions with a track record, not theories or speculation.
- It describes a relationship: Patterns don't just describe modules, but describe deeper system structures and mechanisms.

Elements Of a Pattern

- Pattern name
- Problem
- Solution
- Consequences

List of Common Patterns

Common Patterns

- Singleton
- Factorymethod
- Abstract Factory
- MVC (Model-View-Controller)
- Observer
- Facade

Singleton Pattern

Singleton Pattern

- Used when there is a need to have exactly one instance of a class
  - Example: Window manager, Print spooler
- Enforced by having a global point of access
- Allow multiple instances in the future without affecting a singleton class's clients

Singleton Class

```
public class Singleton {

    private static Singleton singleton = new Singleton();

    /** A private Constructor prevents any other class from instantiating. */
    private Singleton() {
    }

    /** Static 'instance' method */
    public static Singleton getInstance() {
        return singleton;
    }

    // other methods protected by singleton-ness would be here...

    /** A simple demo method */
    public String demoMethod() {
        return "demo";
    }

}
```

Singleton Pattern Example

```
public class SingletonPattern {

    public static void main(String[] args) {
        Singleton s1 = Singleton.getInstance();
        System.out.println("Calling a method of a Singleton: " + s1.demoMethod());

        Singleton s2 = Singleton.getInstance();
        System.out.println("Calling a method of a Singleton: " + s2.demoMethod());

        boolean b1 = (s1 == s2);
        System.out.println("Object instance s1 and s2 are the same object: " + b1);

    }

}
```

FactoryMethod Pattern

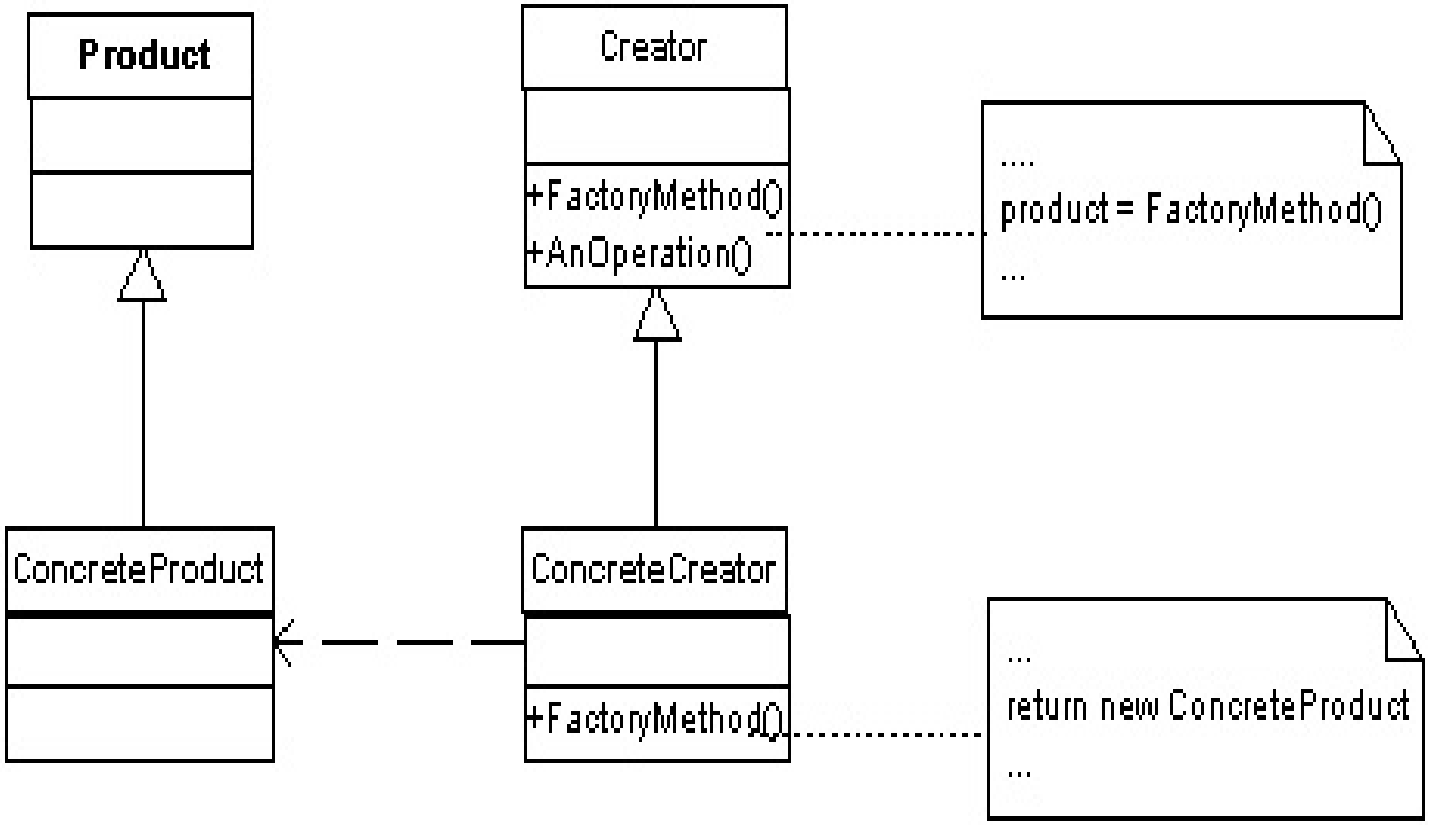
FactoryMethod Pattern

- Like other creational patterns, it deals with the problem of creating objects (products) without specifying the exact class of object that will be created
- Factory method handles this problem by defining a separate method for creating the objects, which subclasses can then override to specify the derived type of product that will be created
- More generally, the term factory method is often used to refer to any method whose main purpose is creation of objects.

FactoryMethod Pattern

- Common in toolkits and frameworks where library code needs to create objects of types which may be subclassed by applications using the framework
- Factory Method pattern is a simplified version of Abstract Factory pattern
  - Factory Method pattern is responsible of creating products that belong to one family, while Abstract Factory pattern deals with multiple families of products.

FactoryMethod Pattern



Abstract Factory Pattern

Abstract Factory Pattern

- Provides a way to encapsulate a group of individual factories that have a common theme
- In normal usage, the client software would create a concrete implementation of the abstract factory and then use the generic interfaces to create the concrete objects that are part of the theme
- The client does not know (nor care) about which concrete objects it gets from each of these internal factories since it uses only the generic interfaces of their products

Abstract Factory Pattern

- This pattern separates the details of implementation of a set of objects from its general usage
- Typically uses FactoryMethod pattern underneath

Abstract Factory Pattern Example

- An example of this would be an abstract factory class *DocumentCreator* that provides interfaces to create a number of products (eg. *createLetter*() and *createResume*())
- The system would have any number of derived concrete versions of the *DocumentCreator* class like *FancyDocumentCreator* or *ModernDocumentCreator*, each with a different implementation of *createLetter*() and *createResume*() that would create a corresponding object like *FancyLetter* or *ModernResume*

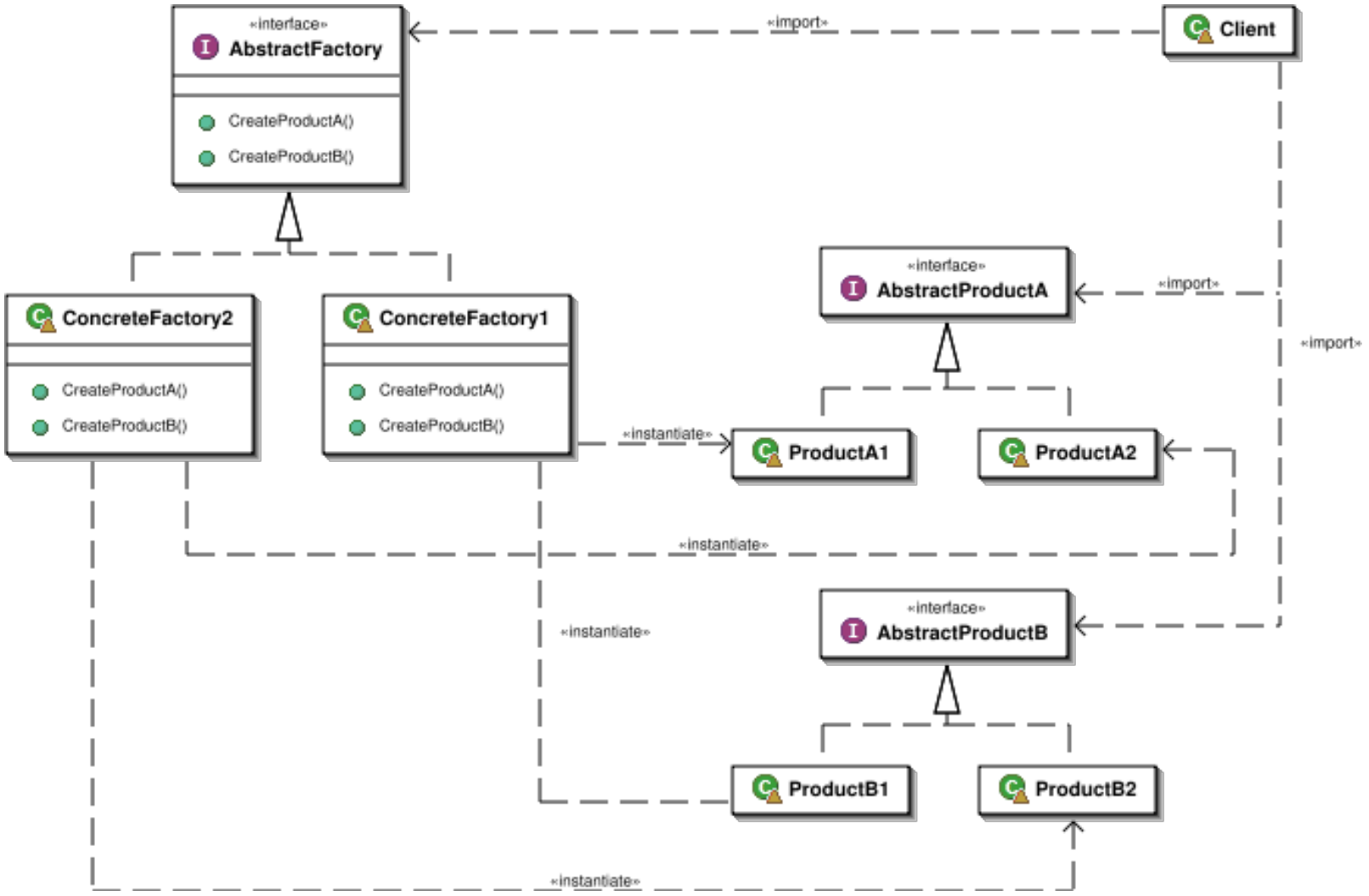
Abstract Factory Pattern Example

- Each of these products is derived from a simple abstract class like Letter or Resume of which the client is aware
- The client code would get an appropriate instantiation of the *DocumentCreator* and call its factory methods
- Each of the resulting objects would be created from the same *DocumentCreator* implementation and would share a common theme. (They would all be fancy or modern objects.)

Abstract Factory Pattern Example

- The client would need to know how to handle only the abstract *Letter* or *Resume* class, not the specific version that it got from the concrete factory

Abstract Factory Pattern Example



MVC Pattern

Facade Pattern

Facade Pattern

- Intent – Provide a simple interface to a subsystem
- Motivation – A complex system may have many pieces that need to be exposed. This could be confusing. Supply a simple interface on top of the complex system
- Participants – Facade, SubsystemClasses