14 - Design patterns

TS. Trịnh Tuấn Đạt
Bộ môn CNPM, Viện CNTT,
ĐHBK Hà Nội

1

Content

1. Introduction
2. Singleton
3. Factory Method

2

Content

1. **Introduction**
2. Singleton
3. Factory

3

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

4

Introduction

- Design patterns represent the best practices used by experienced object-oriented software developers
- Design patterns are solutions to general problems that software developers faced during software development. These solutions were obtained by trial and error by numerous software developers over quite a substantial period of time.

5

5

What is Gang of Four (GOF)

- In 1994, four authors Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides published a book titled **Design Patterns - Elements of Reusable Object-Oriented Software** which initiated the concept of Design Pattern in Software development

6

6

GoF patterns: three categories

- *Creational Patterns* – these abstract the object-instantiation process
  - `Factory Method, Abstract Factory, Singleton, Builder, Prototype`
- *Structural Patterns* – these abstract how objects/classes can be combined
  - `Adapter, Bridge, Composite, Decorator, Façade, Flyweight, Proxy`
- *Behavioral Patterns* – these abstract communication between objects
  - `Command, Interpreter, Iterator, Mediator, Observer, State, Strategy, Chain of Responsibility, Visitor, Template Method`

7

7

Main elements of a design pattern

- Pattern Name:
  - A common name to talk about
- Problem:
  - Context: when to apply the pattern
  - May include a list of conditions for applying the pattern
- Solution:
  - Abstract description of a design problem and how a general arrangement of elements solves it
  - Elements making up the design, their relationships/responsibilities and collaborations
  - Like a template, language-neutral
- Consequences:
  - Results and tradeoff of applying patterns
  - Impacts on system's flexibility, extensibility or portability

8

8

Content

1. Introduction
2. **Singleton**
3. Factory Method

9

9

Motivation

- Only one instance for a class?
- Centralized management of internal or external resources: provide a global point of access to themselves
- Only one class:
  - responsible to instantiate itself, to make sure it creates not more than one instance;
  - provides a global point of access to that instance

10

10

Intent

- Ensure that only one instance of a class is created.
- Provide a global point of access to the object

11

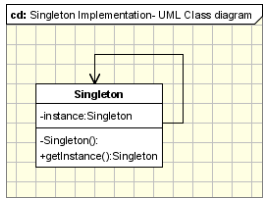
11

Implementation

```
class Singleton {
   private static Singleton instance;
   private Singleton() {
         ...
   }

   public static synchronized Singleton getInstance() {
         if (instance == null)
               instance = new Singleton();

         return instance;
   }
   ...
   public void doSomething() {
         ...
   }
}
```



12

Lazy instantiation using double locking mechanism

```
class Singleton {
    private static Singleton instance;

    private Singleton(){
        System.out.println("Singleton(): Initializing Instance");
    }

    public static Singleton getInstance(){
        if (instance == null){
            synchronized(Singleton.class){
                if (instance == null){
                    instance = new Singleton();
                }
            }
        }

        return instance;
    }

    public void doSomething(){
        System.out.println("doSomething(): Singleton does something!");
    }
}
```

13

13

Early instantiation using implementation with static field

```
class Singleton{
    private static Singleton instance = new Singleton();

    private Singleton(){
        System.out.println("Singleton(): Initializing Instance");
    }

    public static Singleton getInstance(){
        return instance;
    }

    public void doSomething(){
        // …
    }
}
```

14

14

Content

1. Introduction
2. Singleton
3. **Factory Method**

15

15

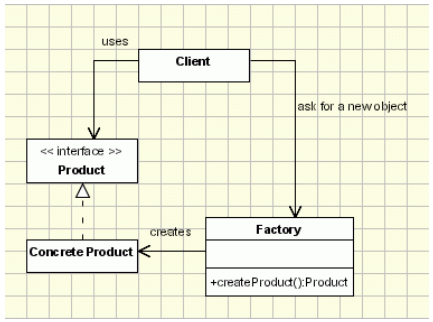
Intent

- creates objects without exposing the instantiation logic to the client.
- refers to the newly created object through a common interface

16

16

Implementation

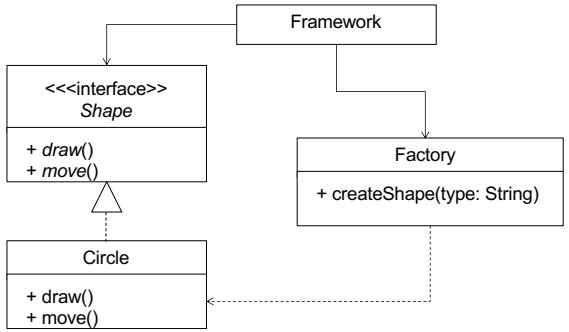


17

17

Example: a graphical framework works with shapes

- Client: drawing framework
- Product: Shape with 2 methods draw() and move()



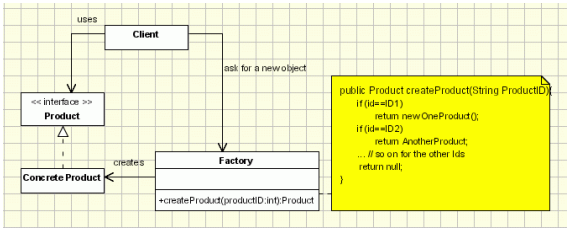
18

18

Switch/case noob instantiation

```
public class ProductFactory{
    public Product createProduct(String ProductID){
        if (id==ID1)
            return new OneProduct();
        if (id==ID2) return
            return new AnotherProduct();
        ... // so on for the other Ids

        return null; //if the id doesn't have any of the expected values
    }
    ...
}
```



19

Class Registration - using reflection

```
class ProductFactory {
    private HashMap m_RegisteredProducts = new HashMap();
    public void registerProduct (String productID, Class productClass){
        m_RegisteredProducts.put(productID, productClass);
    }
    public Product createProduct(String productID){
        Class productClass = (Class)m_RegisteredProducts.get(productID);
        Constructor productConstructor = productClass.
            getDeclaredConstructor(new Class[] {String.class});
        return (Product)productConstructor.newInstance(new Object[] { });
    }
}

public static void main(String args[]){
    Factory.instance().registerProduct("ID1", OneProduct.class);
}

class OneProduct implements Product{
    static {
        Factory.instance().registerProduct("ID1",OneProduct.class);
    }
    ...
}
```

20

20

To ensure correct class loading

```
class Main {
   static {
        try{
                Class.forName("OneProduct");
                Class.forName("AnotherProduct");
        }
        catch (ClassNotFoundException any){
                any.printStackTrace();
        }
   }
   public static void main(String args[]) {
        ...
   }
}
```

21

21

Class Registration – avoiding reflection

```
abstract class Product {
    public abstract Product createProduct();
    ...
}

class OneProduct extends Product {
    ...
    static  {
            ProductFactory.instance().registerProduct("ID1", new OneProduct());
    }
    public OneProduct createProduct() {
            return new OneProduct();
    }
    ...
}

class ProductFactory {
    private HashMap m_RegisteredProducts = new HashMap();

    public void registerProduct(String productID, Product p){
            m_RegisteredProducts.put(productID, p);
    }
    public Product createProduct(String productID){
            ((Product)m_RegisteredProducts.get(productID)).createProduct();
    }
}
```

22

22