PROGRAMACIÓN WEB – JAVA

Patrones de diseño

Dr. José Raúl Romero Salguero
jrromero@uco.es

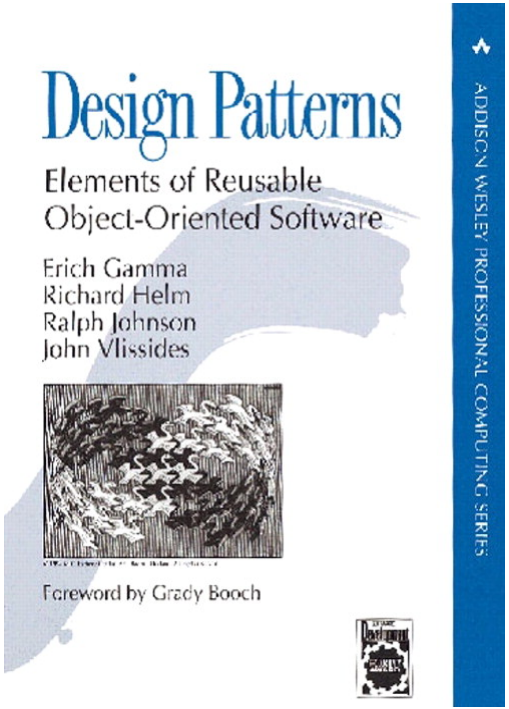
Dra. Aurora Ramírez Quesada

Contenidos

1. Introducción
2. Patrón de diseño: *Singleton*
3. Patrón de diseño: *Abstract factory*
4. Patrón de diseño: *Chain of responsibility*

1.
Introducción

Libro de referencia



Design Patterns

Elements of Reusable Object-Oriented Software

E. Gamma, R. Helm, T. Johnson, J. Vissides
Addison-Wesley, 1995
(Versión española de 2003)

Concepto

- Describen un problema recurrente y una **solución**.
- Cada patrón nombra, explica y evalúa un **diseño recurrente** en sistemas orientados a objetos
- **Elementos principales:**
  - Nombre
  - Problema (motivación)
  - Aplicabilidad
  - Solución (descripción abstracta y opciones de implementación)
  - Consecuencias

Clasificación

- **Patrones creacionales:**
  - Abstraen el proceso de creación de objetos
  - Ayudan a crear sistemas independientes de la forma en que los objetos son creados, compuestos y representados
  - El sistema conoce las clases abstractas
  - Flexibilidad en **qué**/**cómo**/**cuándo** se crea y **quién** lo hace

Clasificación

- **Patrones estructurales:**
  - Proponen cómo las clases y objetos se combinan para crear estructuras más complejas
  - Basados en **herencia**
  - Basados en **composición**
    - Describen formas de combinar objetos para obtener nueva funcionalidad
    - Posibilidad de cambiar la composición en tiempo de ejecución

Clasificación

- **Patrones de comportamiento:**
  - Relacionados con la asignación de **responsabilidades** entre clases
  - Enfatizan la colaboración entre objetos
  - Caracterizan un flujo de control más o menos complejo que será transparente al programador que utilice el patrón

Clasificación

		Propósito		
		Creacional	**Estructural**	**De comportamiento**
Ámbito	**Herencia**	Factory method	Adapter	Interpreter Template method
	Composición	**Singleton** **Abstract factory** Builder Prototype	Adapter Bridge Composite Decorator Facade Flyweight Proxy	**Chain of responsibility** Command Iterator Mediator Memento Observer State Strategy Visitor

2.

Patrón de diseño: *Singleton*

*Singleton_* Definición

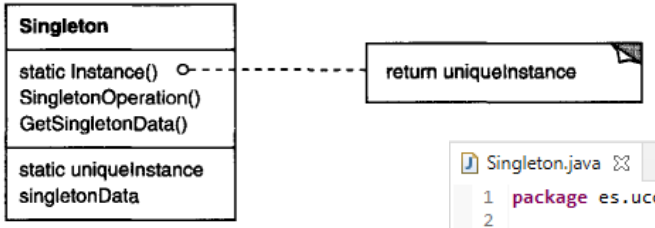
- **Problema:** impedir que se puedan crear múltiples objetos de una clase
- **Aplicabilidad:** es útil cuando…
  - Debe existir una **única instancia** de la clase, que sea accesible a sus clientes desde un punto de acceso conocido.
  - La instancia única debe ser extensible (subclases), y los clientes deben poder utilizar las subclases sin modificar el código.
- **Consecuencias:**
  - ✓ Acceso controlado, permitiendo más instancias si es necesario
  - ✓ Reducción del espacio de nombres (no es una variable global)
  - ✓ Su comportamiento puede refinarse en subclases (es extensible)

*Singleton_* Solución

Solo puede haber una instancia en cada instante

Ver ejemplo `Singleton.java`

- **Estructura:**



- **Implementación:**
  - Atributo estático
  - Constructor privado
  - Función de acceso

Singleton.java

```
1  package es.uco.pw.singleton;
2
3  public class Singleton {
4
5      // 1 - One unique instance
6
7      private static Singleton instance = null;
8
9      // 2 - Private constructor
10
11     private Singleton() {
12
13     }
14
15     // 3 - Access point to the instance
16
17     public static Singleton getInstance() {
18         if(instance == null) {
19             instance = new Singleton();
20         }
21         return instance;
22     }
23 }
24
```

*Singleton_* Ejemplo

Ver paquete `es.uco.pw.singleton.example`

- **Generador de números aleatorios**
  - Se necesita un generador de secuencias de números aleatorios (por ejemplo, para un sistema de encriptación)
  - Se debe garantizar una única secuencia de valores, por lo que no debe permitirse cambiar la semilla inicial.

RandomSequence.java

```java
package es.uco.pw.singleton.example;

import java.util.Random;

/**
 * A random number generator implementing the singleton pattern
 * @author Aurora Ramirez
 * @author Jose Raul Romero
 * */

public class RandomSequence {

    // 1 - The singleton

    private static RandomSequence instance = null;

    // Other properties
    private Random rndObject;

    // 2 - Private constructor

    private RandomSequence() {
        this.rndObject = new Random(0);
    }

    // 3 - Access point to the instance

    public static RandomSequence getInstance() {
        if(instance == null) {
            instance = new RandomSequence();
        }
        return instance;
    }

    // Other operations within the class
    public Integer nextRandomValue() {
        if(instance != null) {
            Integer rndValue = Integer.valueOf(this.rndObject.nextInt());
            return rndValue;
        }
        return null;
    }
}
```

3.

Patrón de diseño: *Abstract factory*

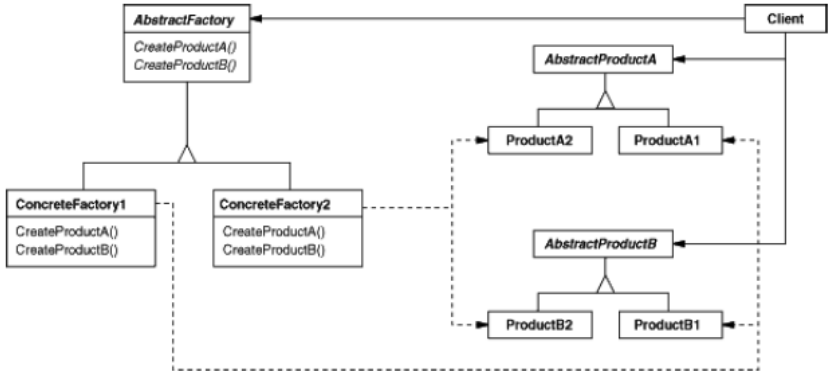
*Abstract factory_* Definición

- **Problema:** crear una familia de objetos relacionados
- **Aplicabilidad:** útil cuando…
  - El sistema debe ser **independiente** de cómo los productos son creados, compuestos y representados.
  - El sistema debe configurar una **familia de productos** diseñados para ser utilizados juntos
  - Se quiere ofrecer una interfaz de acceso a los productos, ocultando su implementación
- **Consecuencias:**
  - ✓ Las clases concretas quedan “aisladas” (encapsuladas)
  - ✓ Los productos pueden intercambiarse, pero el conjunto es fijo

*Abstract factory_* Solución

Ver paquete
`es.uco.pw.factory`

➢ Estructura:



➢ Implementación:

- ❑ Jerarquía de productos
- ❑ Interfaz de creación (o clase abstracta)
- ❑ Factorías concretas (puede ser un *Singleton*)

AbstractFactory.java

```java
package es.uco.pw.factory;

public abstract class AbstractFactory {

    // Factory methods for each product

    public abstract ProductA createProductA();

    public abstract ProductB createProductB();
}
```

ConcreteFactory1.java

```java
package es.uco.pw.factory;

public class ConcreteFactory1 extends AbstractFactory {

    // Implementation of creation methods

    @Override
    public ProductA createProductA() {
        ProductA product = new ProductA(0);
        return product;
    }

    @Override
    public ProductB createProductB() {
        ProductB product = new ProductB();
        return product;
    }
}
```

Ver paquete
`es.uco.pw.factory.example`

Abstract factory_ Ejemplo

Menú de comidas

- Servicio de comidas de un restaurante, con platos de menú o de temporada
- El catálogo de platos y precios varía en función de si el menú se crea para llevar o para consumir en el restaurante

```java
public abstract class MenuCreator {

    /**
     * Create a menu for a weekday with several dishes
     * @return A list of the dishes included in the menu
     * */
    public abstract ArrayList<DailyMeal> createWeekMenu();

    /**
     * Create a menu as a season meal
     * @return A season meal for the menu
     * */
    public abstract SeasonMeal createSeasonMenu();
}
```

```java
public class RestaurantMenuCreator extends MenuCreator{

    @Override
    public ArrayList<DailyMeal> createWeekMenu() {
        ArrayList<DailyMeal> weekMenu = new ArrayList<DailyMeal>();
        DailyMeal firstDish = new DailyMeal("salmorejo", 2.25);
        firstDish.setType(Type.entrante);

        DailyMeal secondDish = new DailyMeal("churrasco", 6);
        secondDish.setGarnish("ensalada");
        secondDish.setType(Type.principal);

        DailyMeal dessert = new DailyMeal("flan", 4.75);
        dessert.setType(Type.postre);

        weekMenu.add(firstDish);
        weekMenu.add(secondDish);
        weekMenu.add(dessert);
        return weekMenu;
    }

    @Override
    public SeasonMeal createSeasonMenu() {
        SeasonMeal dish = new SeasonMeal("alcachofas a la montillana", 3.75);
        dish.setSeason(Season.invierno);
        dish.setExtraCost(0);
        return dish;
    }
}
```

```java
public class TakeAwayMenuCreator extends MenuCreator{

    @Override
    public ArrayList<DailyMeal> createWeekMenu() {
        ArrayList<DailyMeal> takeAwayMenu = new ArrayList<DailyMeal>();
        DailyMeal firstDish = new DailyMeal("ensalada", 1.5);
        firstDish.setType(Type.entrante);

        DailyMeal secondDish = new DailyMeal("brocheta", 4);
        secondDish.setGarnish("patatas");
        secondDish.setType(Type.principal);

        takeAwayMenu.add(firstDish);
        takeAwayMenu.add(secondDish);

        return takeAwayMenu;
    }

    @Override
    public SeasonMeal createSeasonMenu() {
        SeasonMeal dish = new SeasonMeal("alcachofas a la montillana", 3.75);
        dish.setSeason(Season.invierno);
        dish.setExtraCost(1);
        return dish;
    }
}
```

4.

Patrón de diseño:
Chain of responsibility

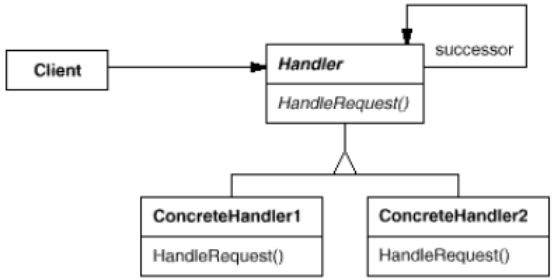
*Chain of responsibility_* Definición

- **Problema:** procesamiento de una petición centralizado en un receptor
- **Aplicabilidad:** útil cuando…
  - Una petición puede ser tratada de distintas maneras, sin conocer previamente cuál se va a aplicar.
  - Los objetos que manejan la petición (***handlers***) deben ser especificados dinámicamente (en función de su contenido o procesamiento anterior).
- **Consecuencias:**
  - ✓ Reduce el acoplamiento, pues el emisor no necesita conocer al *handler*
  - ✓ La responsabilidad se distribuye entre los *handlers*
  - ✓ Si no existe un *handler* adecuado, la petición no se procesará

Ver paquete
`es.uco.pw.chain`

*Chain of responsibility_* Implementación

Estructura:



Implementación:

- Un *handler* abstracto, que declara al sucesor
- Varios *handlers* concretos:
  1. Si puede, procesa la petición
  2. Invoca al siguiente *handler* (sucesor) para continuar la cadena

AbstractHandler.java

```java
package es.uco.pw.chain;

public abstract class AbstractHandler {

    // The next handler in the chain
    protected AbstractHandler successor;

    // 1 - A method to assign the successor (build the chain in the client)
    public void setSuccessor(AbstractHandler sucesor) {
        this.successor = sucesor;
    }

    // 2 - An abstract method to handle the request
    public abstract void handleRequest(Request request);

}
```

DefaultHandler.java

```java
package es.uco.pw.chain;

public class DefaultHandler extends AbstractHandler {

    public DefaultHandler() {
        this.successor = new SecondHandler(); // or use setSucessor in the client
    }

    @Override
    public void handleRequest(Request request) {
        if(request.getType() == 0) {
            // do something
        }
        else if(this.successor != null){
            this.successor.handleRequest(request);
        }
    }
}
```

Ver paquete
`es.uco.pw.chain.example`

*Chain of responsibility_* Ejemplo

➢ Tienda de ropa

- ❑ Una cadena de tiendas de ropa dispone de un servicio de consulta de *stock*
- ❑ Si la prenda solicitada no está disponible en la tienda local, se envía la petición a la tienda nacional, que dispone de dos almacenes.

StockProvider.java

```java
package es.uco.pw.chain.example;

/**
 * The abstract class to build the chain of shops and find products in stock
 * @author Aurora Ramirez
 * @author Jose Raul Romero
 * */

public abstract class StockProvider {

    /** Another stock provider to ask for
     * a product when it is not available  */
    protected StockProvider nextProvider;

    /**
     * Set successor
     * */
    public void setSuccessor(StockProvider succesor) {
            this.nextProvider = succesor;
    }

    /**
     * Handle the purchase
     * */
    public abstract boolean findProduct(String productName);

}

```

LocalShop.java

```java

public class LocalShop extends StockProvider {

    /** The list of currently available products */
    private ArrayList<ShopProduct> productsInStock;

    public LocalShop() {
        this.productsInStock = new ArrayList<ShopProduct>();
        this.createStock();
    }

    @Override
    public boolean findProduct(String productName) {

        boolean inStock = false;
        int size = this.productsInStock.size();
        for(int i=0; !inStock && i<size; i++) {
            ShopProduct p = this.productsInStock.get(i);
            if(p.getName().equals(productName) && p.isAvailable()) {
                inStock = true;
                p.sellUnit();
                System.out.println("Product found in local shop");
            }
        }
        if(!inStock) {
            System.out.println("Product not found in local shop, ask next supplier");
            inStock = this.nextProvider.findProduct(productName);
        }
        return inStock;
    }
```

NationalShop.java

```java
public class NationalShop extends StockProvider {

    /** The national provider has two stocks */
    private ArrayList<ShopProduct> productsInStockMadrid;

    private ArrayList<ShopProduct> productsInStockBarcelona;

    public NationalShop() {
        this.productsInStockMadrid = new ArrayList<ShopProduct>();
        this.productsInStockBarcelona = new ArrayList<ShopProduct>();
        this.createStocks();
    }

    @Override
    public boolean findProduct(String productName) {
        boolean inStock = false;

        // Firstly, try in Madrid
        inStock = findProductInStock(this.productsInStockMadrid, productName);

        if(inStock) {
            System.out.println("Product found in Madrid");
        }
        else{
            // Try in Barcelona
            inStock = findProductInStock(this.productsInStockBarcelona, productName);
            if(inStock) {
                System.out.println("Product found in Barcelona");
            }
            else { // end of the chain
                System.out.println("Product not found");
            }
        }

        return inStock;
    }
```

Programación Web

Patrones de diseño__ **Curso 2020/21**