Universidad Autónoma de Nuevo León

Facultad de Fisicomatemáticas

Lic. En Seguridad en Tecnologías de la Información

Diseño Orientado a Objetos

Tarea 6

Profesor: Miguel Salazar

Alumno: Rodolfo Heriberto Espinoza Gonzalez

Matricula: 1418935

Patrones de diseño

Singleton

Su intención consiste en garantizar que una clase sólo tenga una instancia y proporcionar un punto de acceso global a ella.

Este patrón se implementa creando en nuestra clase un método que crea una instancia del objeto sólo si todavía no existe alguna. Para asegurar que la clase no puede ser instanciada nuevamente se regula el alcance del constructor (con modificadores de acceso como protegido o privado).

La instrumentación del patrón puede ser delicada en programas con múltiples hilos de ejecución. Si dos hilos de ejecución intentan crear la instancia al mismo tiempo y esta no existe todavía, sólo uno de ellos debe lograr crear el objeto. La solución clásica para este problema es utilizar exclusión mutua en el método de creación de la clase que implementa el patrón.

Las situaciones más habituales de aplicación de este patrón son aquellas en las que dicha clase controla el acceso a un recurso físico único (como puede ser el ratón o un archivo abierto en modo exclusivo) o cuando cierto tipo de datos debe estar disponible para todos los demás objetos de la aplicación.

El patrón *singleton* provee una única instancia global gracias a que:

- La propia clase es responsable de crear la única instancia.
- Permite el acceso global a dicha instancia mediante un método de clase.
- Declara el constructor de clase como privado para que no sea instanciable directamente.
- Al estar internamente autoreferenciada, en lenguajes como Java, el recolector de basura no actúa.

Ejemplo

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

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

Factory

Consiste en utilizar una clase constructora (al estilo del Abstract Factory) abstracta con unos cuantos métodos definidos y otro(s) abstracto(s): el dedicado a la construcción de objetos de un subtipo de un tipo determinado. Es una simplificación del Abstract Factory, en la que la clase abstracta tiene métodos concretos que usan algunos de los abstractos; según usemos una u otra hija de esta clase abstracta, tendremos uno u otro comportamiento.

Ejemplo

```
abstract class Creator{
    // Definimos método abstracto
    public abstract Product factoryMethod()
}
```

Ahora definimos el creador concreto:

```
public class ConcreteCreator extends Creator{
    public Product factoryMethod() {
        return new ConcreteProduct();
    }
}
```

Definimos el producto y su implementación concreta:

```
public interface Product{
    public void operacion();
}

public class ConcreteProduct implements Product{
    public void operacion(){
        System.out.println("Una operación de este producto");
    }
}
```

Main:

```
public static void main(String args[]){
    Creator aCreator;
    aCreator = new ConcreteCreator();
    Product producto = aCreator.factoryMethod();
    producto.operacion();
```

```
}
```

Prototype

tiene como finalidad crear nuevos objetos clonando una instancia creada previamente. Este patrón especifica la clase de objetos a crear mediante la clonación de un prototipo que es una instancia ya creada. La clase de los objetos que servirán de prototipo deberá incluir en su interfaz la manera de solicitar una copia, que será desarrollada luego por las clases concretas de prototipos.

Este patrón resulta útil en escenarios donde es impreciso abstraer la lógica que decide qué tipos de objetos utilizará una aplicación, de la lógica que luego usarán esos objetos en su ejecución. Los motivos de esta separación pueden ser variados, por ejemplo, puede ser que la aplicación deba basarse en alguna configuración o parámetro en tiempo de ejecución para decidir el tipo de objetos que se debe crear. En ese caso, la aplicación necesitará crear nuevos objetos a partir de modelos. Estos modelos, o prototipos, son clonados y el nuevo objeto será una copia exacta de los mismos, con el mismo estado. Esto resulta interesante para crear, en tiempo de ejecución, copias de objetos concretos inicialmente fijados, o también cuando sólo existe un número pequeño de combinaciones diferentes de estado para las instancias de una clase.

Dicho de otro modo, este patrón propone la creación de distintas variantes de objetos que la aplicación necesite, en el momento y contexto adecuado. Toda la lógica necesaria para la decisión sobre el tipo de objetos que usará la aplicación en su ejecución se hace independiente, de manera que el código que utiliza estos objetos solicitará una copia del objeto que necesite. En este contexto, una copia significa otra instancia del objeto. El único requisito que debe cumplir este objeto es suministrar la funcionalidad de clonarse.

En el caso de un editor gráfico, se pueden crear rectángulos, círculos u otros, como copias de prototipos. Estos objetos gráficos pertenecerán a una jerarquía cuyas clases derivadas implementarán el mecanismo de clonación.

Ejemplo

```
// Los productos deben implementar esta interface
public interface Producto implements Cloneable {
    Object clone();
    // Aquí van todas las operaciones comunes a los productos que
genera la factoría
}

// Un ejemplo básico de producto
public class UnProducto implements Producto {
    private int atributo;

    public UnProducto(int atributo) {
        this.atributo = atributo;
    }
```

```
    public Object clone() {
        return new UnProducto(this.atributo);
    }

    public String toString() {
        return ((Integer)atributo).toString();
    }
}

// La clase encargada de generar objetos a partir de los prototipos
public class FactoriaPrototipo {
    private HashMap mapaObjetos;
    private String nombrePorDefecto;

    public FactoriaPrototipo() {
        mapaObjetos = new HashMap();
        // Se incluyen en el mapa todos los productos prototipo
        mapaObjetos.put("producto 1", new UnProducto(1));
    }

    public Object create() {
        return create(nombrePorDefecto);
    }

    public Object create(String nombre) {
        nombrePorDefecto = nombre;
        Producto objeto = (Producto)mapaObjetos.get(nombre);
        return objeto != null ? objeto.clone() : null;
    }
}

public class PruebaFactoria {
    static public void main(String[] args) {
        FactoriaPrototipo factoria = new FactoriaPrototipo();
        Producto producto = (Producto) factoria.create("producto 1");
        System.out.println ("Este es el objeto creado: " + producto);
    }
}
```

Builder

el patrón builder (Constructor) es usado para permitir la creación de una variedad de objetos complejos desde un objeto fuente (Producto), el objeto fuente se compone de una variedad de partes que contribuyen individualmente a la creación de cada objeto complejo a través de un conjunto de llamadas a interfaces comunes de la clase Abstract Builder.

El patrón builder es creacional.

A menudo, el patrón builder construye el patrón Composite, un patrón estructural.

Su intención es abstraer el proceso de creación de un objeto complejo, centralizando dicho proceso en un único punto, de tal forma que el mismo proceso de construcción pueda crear representaciones diferentes.

Ejemplo

```
/** "Producto" */
class Pizza {
    private String masa = "";
    private String salsa = "";
    private String relleno = "";

    public void setMasa(String masa)     { this.masa = masa; }
    public void setSalsa(String salsa)     { this.salsa = salsa; }
    public void setRelleno(String relleno) { this.relleno = relleno; }
}


/** "Abstract Builder" */
abstract class PizzaBuilder {
    protected Pizza pizza;

    public Pizza getPizza() { return pizza; }
    public void crearNuevaPizza()

    public abstract void buildMasa();
    public abstract void buildRelleno();
}


/** "ConcreteBuilder" */
class HawaiPizzaBuilder extends PizzaBuilder {
    public void buildMasa()   { pizza.setMasa("suave"); }
    public void buildSalsa()    { pizza.setSalsa("dulce"); }
```

```
    public void buildRelleno() {
pizza.setRelleno("chorizo+alcachofas"); }
}

/** "ConcreteBuilder" */
class PicantePizzaBuilder extends PizzaBuilder {
    public void buildMasa()    { pizza.setMasa("cocida"); }
    public void buildSalsa()    { pizza.setSalsa("picante"); }
    public void buildRelleno() {
pizza.setRelleno("pimienta+salchichón"); }
}


/** "Director" */
class Cocina {
    private PizzaBuilder pizzaBuilder;

    public void setPizzaBuilder(PizzaBuilder pb) { pizzaBuilder = pb;
}
    public Pizza getPizza() { return pizzaBuilder.getPizza(); }

    public void construirPizza() {

       pizzaBuilder.buildRelleno();
    }
}


/** Un cliente pidiendo una pizza. */
class BuilderExample {
    public static void main(String[] args) {
        Cocina cocina = new Cocina();
        PizzaBuilder hawai_pizzabuilder = new HawaiPizzaBuilder();
        PizzaBuilder picante_pizzabuilder = new PicantePizzaBuilder();

        cocina.setPizzaBuilder( hawai_pizzabuilder );
        cocina.construirPizza();

        Pizza pizza = cocina.getPizza();
    }
}
```

```java
/**
 *  2da opción para el abstract builder quizá más transparente para su uso.
 *  Dentro del crear se llaman los métodos build.
 *  Es válido siempre y cuando no se necesite alterar
 *  el orden del llamado a los "build's".
 */
abstract class OtroPizzaBuilder {
    protected Pizza pizza;

    public Pizza getPizza() { return pizza; }
    public void crearNuevaPizza() {
           pizza = new Pizza();
           buildMasa();
           buildSalsa();
           buildRelleno();
    }

    public abstract void buildMasa();
    public abstract void buildSalsasasa();
    public abstract void buildRelleno();
}

/** "Director" */
class OtraCocina {
    private OtroPizzaBuilder pizzaBuilder;



    public void construirPizza() {
       pizzaBuilder.crearNuevaPizza();
       //notar que no se necesita llamar a cada build.
    }
}
```

Bridge

También conocido como Handle/Body, es una técnica usada en programación para desacoplar una abstracción de su implementación, de manera que ambas puedan ser modificadas independientemente sin necesidad de alterar por ello la otra.

Esto es, se desacopla una abstracción de su implementación para que puedan variar independientemente.

Se usa el patrón *Bridge* cuando:

- Se desea evitar un enlace permanente entre la abstracción y su implementación. Esto puede ser debido a que la implementación debe ser seleccionada o cambiada en tiempo de ejecución.
- Tanto las abstracciones como sus implementaciones deben ser extensibles por medio de subclases. En este caso, el patrón *Bridge* permite combinar abstracciones e implementaciones diferentes y extenderlas independientemente.
- Cambios en la implementación de una abstracción no deben impactar en los clientes, es decir, su código no debe tener que ser recompilado.
- (En C++) Se desea esconder la implementación de una abstracción completamente a los clientes. En C++, la representación de una clase es visible en la interface de la clase.
- Se desea compartir una implementación entre múltiples objetos (quizá usando contadores), y este hecho debe ser escondido a los clientes.

Ejemplo

```
interface Implementador {
    public abstract void operacion();
}

/** primera implementacion de Implementador **/
class ImplementacionA implements Implementador{
    public void operacion() {
        System.out.println("Esta es la implementacion A");
    }
}
/** segunda implementacion de Implementador **/
class ImplementacionB implements Implementador{
    public void operacion() {
        System.out.println("Esta es una implementacion de B");
    }
}
/** interfaz de abstracción **/
interface Abstraccion {
    public void operacion();
}
```

```java
/** clase refinada que implementa la abstraccion **/
class AbstraccionRefinada implements Abstraccion{
    private Implementador implementador;
    public AbstraccionRefinada(Implementador implementador){
        this.implementador = implementador;
    }
    public void operacion(){
        implementador.operacion();
    }
}
/** aplicacion que usa el patrón Bridge **/
public class EjemploBridge {
    public static void main(String[] args) {
        Abstraccion[] abstracciones = new Abstraccion[2];
        abstracciones[0] = new AbstraccionRefinada(new
ImplementacionA());
        abstracciones[1] = new AbstraccionRefinada(new
ImplementacionB());
        for(Abstraccion abstraccion:abstracciones)
            abstraccion.operacion();
    }
}
```