UNIVERSIDAD PRIVADA DE TACNA
INGENIERIA DE SISTEMAS

TITULO:

PATRONES DE DISEÑO CREACIONALES

CURSO:

CALIDAD Y PRUEBAS DE SOFTWARE

DOCENTE:

Ing. Patrick Cuadros Quiroga

Integrantes:

Sivirichi Falcón, Ricardo Alonso	(2018060905)
Chambilla Maquera, Araceli Noemi	(2018060897)
Arenas Paz Soldan, Miguel Jesus	(2017059282)
Cotrado Marino, Ana Luz	(2018060907)

Tacna - Perú
2020

Resumen

Este trabajo describe lo que es un patrón de diseño, sus objetivos y clasificaciones aportando ejemplos de cada uno de ellos. El trabajo también ofrece información sobre algunos proyectos que sobre patrones de diseño aplicados a objetos de aprendizaje. Los patrones creacionales proporcionan varios mecanismos de creación de objetos que incrementan la flexibilidad y la reutilización del código existente. Proporciona una interfaz para la creación de objetos en una superclase, mientras permite a las subclases alterar el tipo de objetos que se crearán.

Palabras Clave: patrones de diseño, patrones creacionales.

Abstract

This work describes what a design pattern is, its objectives and classifications, providing examples of each of them. The work also offers information on some projects that on design patterns applied to learning objects. Creational patterns provide various object creation mechanisms that increase flexibility and reuse of existing code. Provides an interface for creating objects in a superclass, while allowing subclasses to alter the type of objects that will be created.

Keywords: design patterns, creational patterns.

---

1 Introducción

Como programadores seguramente nos habremos dado cuenta de que muchas veces resolvemos un mismo problema de manera diferente.

Esto al principio de nuestra carrera puede ser algo positivo: experimentamos diferentes formas de enfocar un problema y podemos comparar los pros y los contras de nuestras anteriores soluciones, pero con el paso del tiempo nos gusta ir directamente al grano, y aplicar la solución más escalable, más estable y más reutilizable.

1.1 Evolución histórica

En 1979, el arquitecto Chistopher Alexander escribe el libro "The Timeless Way of Building" sobre el uso de patrones en la construcción de edificios, lo que contribuyó a que años más tarde se escribiese otro libro "A Pattern Language" que fue el primer intento por formalizar los conocimientos arquitectónicos.

Más tarde, en 1987, Ward Cunningham y Kent Beck orientaron esos patrones hacia la informática en su libro "Using Pattern Language for OO Programs" donde desarrollaron cinco patrones orientados a la interacción hombre máquina.

A principios de los 90's es cuando con el libro "Design Patterns" escrito por el GoF (Gang of Four) donde los patrones de diseño alcanzan su auge.

Este libro recogía 23 patrones de diseño aplicados a la programación informática. Es a partir de principios de los 90 cuando los patrones alcanzan su auge, utilizándose para aportar soluciones a los proyectos informáticos, con la publicación del libro Design Patterns escrito por el GoF (Gang of Four, nombre que reciben comúnmente los autores de este libro).

En este caso el patrón se presenta como la solución a un problema que ocurre infinidad de veces en el entorno. Este libro recogía 23 patrones de diseño aplicados a la programación informática.

De igual manera, mencionan que un patrón de diseño se compone de un nombre (que debe ser muy claro, usualmente dice lo que hace), describe de manera general el problema a trabajar y la solución debe estar definida en función del problema. Por último, señalan tres categorías para agrupar los patrones de diseño: creacionales, estructurales y de comportamiento.

2 Desarrollo

A menudo los patrones se confunden con algoritmos porque ambos conceptos describen soluciones típicas a problemas conocidos. Mientras que un algoritmo siempre define un grupo claro de acciones para lograr un objetivo, un patrón es una descripción de más alto nivel de una solución.

Los patrones de diseño creacionales abstraen el proceso de instanciación, encapsulan el conocimiento acerca de qué clase concreta se usa y esconde como las instancias de estas clases son creadas y unidas.

Un patrón de clase creacional usa la herencia para variar la clase que es instanciada. Un patrón de objeto creacional delega la instanciación a otro objeto, estos nos ayudan a definir la forma en la que los objetos interactúan entre ellos.

2.1 Tipos de Patrones de Diseño Creacionales

2.1.1 Singleton

GoF Definición: Asegura que la clase sólo tiene una instancia, y proporcionar un punto de acceso global a ella.

Utiliza el patrón Singleton cuando una clase de tu programa tan solo deba tener una instancia disponible para todos los clientes; por ejemplo, un único objeto de base de datos compartido por distintas partes del programa.

Ejemplo:

```
class Database is
private static field instance: Database
private constructor Database() is
public static method getInstance() is
    if (Database.instance == null) then
        acquireThreadLock() and then
        if(Database.instance == null) then
            Database.instance=new Database()
    return Database.instance

public method query(sql) is

class Application is
    method main() is
        Database foo = Database.getInstance()
        foo.query("SELECT ...")

        Database bar = Database.getInstance()
        bar.query("SELECT ...")
```

2.1.2 Abstract Factory

Definición de GoF: Proporciona una interfaz para crear familias de objetos relacionados o dependientes sin especificar sus clases concretas.

Utiliza el patrón Abstract Factory cuando tu código deba funcionar con varias familias de productos relacionados, pero no desees que dependa de las clases concretas de esos productos, ya que puede ser que no los conozcas de antemano o sencillamente quieras permitir una futura extensibilidad.

Ejemplo:

```
interface GUIFactory is
  method createButton():Button
  method createCheckbox():Checkbox

class WinFactory implements GUIFactory is
  method createButton():Button is
    return new WinButton()
  method createCheckbox():Checkbox is
    return new WinCheckbox()

class MacFactory implements GUIFactory is
  method createButton():Button is
    return new MacButton()
  method createCheckbox():Checkbox is
    return new MacCheckbox()

interface Button is
  method paint()

class WinButton implements Button is
  method paint() is

class MacButton implements Button is
  method paint() is

interface Checkbox is
  method paint()

class WinCheckbox implements Checkbox is
  method paint() is

class MacCheckbox implements Checkbox is
  method paint() is

class Application is
  private field factory: GUIFactory
  private field button: Button
  constructor Application(factory:GUIFactory)is
    this.factory = factory
  method createUI() is
    this.button = factory.createButton()
  method paint() is
    button.paint()

class ApplicationConfigurator is
  method main() is
    config = readApplicationConfigFile()

    if (config.OS == "Windows") then
      factory = new WinFactory()
    else if (config.OS == "Mac") then
      factory = new MacFactory()
    else
      throw new Exception
      ("Error! Unknown operating system.")

  Application app = new Application(factory)
```

2.1.3 Factory Method

Definición de GoF: Define una interfaz para crear un objeto, pero deja que las subclases decidan qué clase instanciar.

- Utiliza el Factory Method cuando no conozcas de antemano las dependencias y los tipos exac-

tos de los objetos con los que deba funcionar tu código.

- Utiliza el Factory Method cuando quieras ofrecer a los usuarios de tu biblioteca o framework, una forma de extender sus componentes internos.

Ejemplo:

```
class Dialog is
    abstract method createButton():Button
    method render() is
        Button okButton = createButton()
        // Ahora utiliza el producto.
        okButton.onClick(closeDialog)
        okButton.render()

class WindowsDialog extends Dialog is
    method createButton():Button is
        return new WindowsButton()

class WebDialog extends Dialog is
    method createButton():Button is
        return new HTMLButton()

interface Button is
    method render()
    method onClick(f)

class WindowsButton implements Button is
    method render(a, b) is
    method onClick(f) is

class HTMLButton implements Button is
    method render(a, b) is
    method onClick(f) is

class Application is
  field dialog: Dialog
  method initialize() is
    config = readApplicationConfigFile()

    if (config.OS == "Windows") then
      dialog = new WindowsDialog()
    else if (config.OS == "Web") then
      dialog = new WebDialog()
    else
      throw new Exception
        ("Error! Unknown operating system.")

  method main() is
    this.initialize()
    dialog.render()
```

2.1.4 Builder

Definición de GoF: Separa la construcción de un objeto complejo de su representación para que los mismos procesos de construcción puedan crear diferentes representaciones.

Utiliza el patrón Builder para evitar un "constructor telescópico". Digamos que tenemos un constructor con diez parámetros opcionales. Invocar a semejante bestia es poco práctico, por lo que sobrecargamos el constructor y creamos varias versiones más cortas con menos parámetros. Estos constructores siguen recurriendo al principal, pasando algunos valores por defecto a cualquier parámetro omitido.

```
class Pizza {
  Pizza(int size) { ... }
  Pizza(int size, boolean cheese) { ... }
  // ...
```

El patrón Builder permite construir objetos paso a paso, utilizando tan solo aquellos pasos que realmente necesitamos. Una vez implementado el patrón, ya no hará falta apiñar decenas de parámetros dentro de los constructores.

- Utiliza el patrón Builder cuando quieras que el código sea capaz de crear distintas representaciones de ciertos productos (por ejemplo, casas de piedra y madera).

Ejemplo:

```
class Car is
class Manual is

interface Builder is
  method reset()
  method setSeats(...)
  method setEngine(...)
  method setTripComputer(...)
  method setGPS(...)

class CarBuilder implements Builder is
  private field car:Car
  constructor CarBuilder() is
    this.reset()
  method reset() is
    this.car = new Car()
  method setSeats(...) is
  method setEngine(...) is
  method setTripComputer(...) is
  method setGPS(...) is
  method getProduct():Car is
    product = this.car
    this.reset()
    return product

class CarManualBuilder implements Builder is
  private field manual:Manual
    constructor CarManualBuilder() is
    this.reset()
  method reset() is
    this.manual = new Manual()
```

```
  method setSeats(...) is
  method setEngine(...) is
  method setTripComputer(...) is
  method setGPS(...) is
  method getProduct():Manual is

class Director is
  private field builder:Builder

  method setBuilder(builder:Builder)
    this.builder = builder

  method constructSportsCar(builder:Builder)is
    builder.reset()
    builder.setSeats(2)
    builder.setEngine(new SportEngine())
    builder.setTripComputer(true)
    builder.setGPS(true)

  method constructSUV(builder:Builder)is
    // ...

class Application is

  method makeCar() is
    director = new Director()

    CarBuilder builder = new CarBuilder()
    director.constructSportsCar(builder)
    Car car = builder.getProduct()

    CarManualBuilder builder =
        new CarManualBuilder()
    director.constructSportsCar(builder)
    Manual manual = builder.getProduct()
```

2.1.5 Prototype

Definición de GoF: Especifique los tipos de objetos que se crearán utilizando una instancia prototípica y cree nuevos objetos copiando este prototipo.

- Utiliza el patrón Prototype cuando tu código no deba depender de las clases concretas de objetos que necesites copiar.
- Utiliza el patrón cuando quieras reducir la cantidad de subclases que solo se diferencian en la forma en que inicializan sus respectivos objetos. Puede ser que alguien haya creado estas subclases para poder crear objetos con una configuración específica.

Ejemplo:

```
abstract class Shape is
    field X: int
    field Y: int
    field color: string

    constructor Shape() is

    constructor Shape(source: Shape) is
        this()
        this.X = source.X
        this.Y = source.Y
        this.color = source.color

    abstract method clone():Shape

class Rectangle extends Shape is
    field width: int
    field height: int

    constructor Rectangle(source: Rectangle) is
        super(source)
        this.width = source.width
        this.height = source.height

    method clone():Shape is
        return new Rectangle(this)

class Circle extends Shape is
    field radius: int

    constructor Circle(source: Circle) is
        super(source)
        this.radius = source.radius

    method clone():Shape is
        return new Circle(this)

class Application is
    field shapes: array of Shape

    constructor Application() is
        Circle circle = new Circle()
        circle.X = 10
        circle.Y = 10
        circle.radius = 20
        shapes.add(circle)

        Circle anotherCircle = circle.clone()
        shapes.add(anotherCircle)

        Rectangle rectangle = new Rectangle()
        rectangle.width = 10
        rectangle.height = 20
        shapes.add(rectangle)

    method businessLogic() is
        Array shapesCopy = new ArrayOfShapes.
        foreach (s in shapes) do
            shapesCopy.add(s.clone())
```

3 Conclusión

No se necesita mucha experiencia en el campo laboral para darse cuenta de que todos codifican de manera diferente y conocer estas soluciones, evitará que debas reinventar la rueda en alguno de tus proyectos. Pero antes de implementar alguno de estos patrones, debes conocer muy bien las mejoras que traería el patrón a tu proyecto, si no lo haces, es mejor no aplicarlo. Es mejor saber cuándo no implementar un patrón.

4 Recomendaciones

Si queremos desarrollar aplicaciones robustas y fáciles de mantener, debemos cumplir ciertas reglas", ya que estas reglas de diseño son recomendables (muy recomendables), pero no son obligatorias. Siempre podemos decidir no aplicarlas. Aunque si no lo hacemos, hay que ser conscientes de la razón de no aplicarlas y de sus consecuencias.

Referencias

[1] Freeman, E., Robson, E., Bates, B., y Sierra, K. (2008). Head first design patterns. .ºʼ Reilly Media, Inc.".

[2] Dockins, K. (2017). Design Patterns in PHP and Laravel. Apress. https://allitbooks.net/web-development/2056-design-patterns-php-laravel.html

[3] Holzner, S. (2006). Design patterns for dummies. John Wiley y Sons. https://allitbooks.net/programming/1605-design-patterns-for-dummies.html

[4] Giridhar, C. (2016). Learning Python Design Patterns. Packt Publishing Ltd. https://allitbooks.net/programming/1777-learning-python-design-patterns-second-edition.html

[5] Cooper, J. W. (2000). Java design patterns: a tutorial. https://allitbooks.net/programming/2648-java-design-patterns.html

[6] Hunt, J. (2013). Gang of four design patterns. In Scala design patterns. Springer, Cham. http://www.w3sdesign.com/GoF-Design-Patterns-Reference0100.pdf

[7] Shvets, A. (2021). Dive into Design Patterns: Vol. 1.7 (v2021 ed.) [Libro electrónico]. https://refactoring.guru/es/design-patterns/book