

CS2013: Programación III

Teoría: Patrones de Diseño I

José Chávez

Agenda

- Introducción
- Elementos
- Categorías
- Preliminares
- Patrones de Creación

1. Introducción

Retos del Diseño de Software

El diseño de software es un proceso complejo que enfrenta varias dificultades y retos. Algunos de ellos incluyen:

- **Cambios y Evolución del Software:**
 - El software está en constante evolución y debe adaptarse a los cambios en los requisitos del negocio, avances tecnológicos y demandas del mercado.
 - Los cambios en el diseño pueden ser difíciles de manejar y pueden afectar la estructura y la funcionalidad del software.

Retos del Diseño de Software

El diseño de software es un proceso complejo que enfrenta varias dificultades y retos. Algunos de ellos incluyen:

- **Complejidad del Software:**
 - El software puede volverse muy complejo a medida que crece en tamaño y funcionalidad.
 - La complejidad puede dificultar el entendimiento, el mantenimiento y la extensión del software.

Retos del Diseño de Software

El diseño de software es un proceso complejo que enfrenta varias dificultades y retos. Algunos de ellos incluyen:

- **Diseño Flexible y Extensible:**
 - El diseño de software debe ser flexible y extensible para adaptarse a futuros cambios y permitir la adición de nuevas funcionalidades.
 - La falta de flexibilidad y extensibilidad puede llevar a un diseño rígido y difícil de modificar.

Retos del Diseño de Software

El diseño de software es un proceso complejo que enfrenta varias dificultades y retos. Algunos de ellos incluyen:

- **Reutilización de Código:**
 - La reutilización de código es un desafío en el diseño de software, ya que es necesario identificar y encapsular componentes reutilizables.
 - La falta de reutilización de código puede conducir a la duplicación, el acoplamiento excesivo y la dificultad para mantener y actualizar el software.

¿En que nos pueden ayudar los Patrones de Diseño ?

- **Cambios y Evolución del Software:**
 - Los Patrones de Diseño proporcionan soluciones **flexibles** y **modulares** que facilitan la incorporación de cambios y la evolución del software.
 - Los patrones como el **Patrón de Comportamiento Observer** permiten establecer relaciones dinámicas entre objetos, lo que facilita la adaptación a cambios y actualizaciones.

¿En que nos pueden ayudar los Patrones de Diseño ?

- **Complejidad del Software:**

- Los Patrones de Diseño ayudan a gestionar la complejidad del software al proporcionar estructuras **claras** y **organizadas**.
- Los patrones como el **Patrón de Estructura Composite** permite tratar objetos individuales y colecciones de objetos de manera uniforme, reduciendo la complejidad en el diseño y la interacción entre componentes.

¿En que nos pueden ayudar los Patrones de Diseño ?

- **Diseño Flexible y Extensible:**

- Los Patrones de Diseño fomentan un diseño flexible y extensible al **separar responsabilidades y componentes** en el software.
- Los patrones como el **Patrón de Creación Abstract Factory** permiten encapsular la creación de familias de objetos relacionados, lo que facilita la adición de nuevas variantes sin modificar el código existente.

¿En que nos pueden ayudar los Patrones de Diseño ?

- **Reutilización de Código:**

- Los Patrones de Diseño promueven la reutilización de código al **encapsular soluciones probadas** en estructuras y componentes reutilizables. Esto significa que no es necesario reinventar la rueda cada vez que se enfrenta a un problema similar, ya que se puede aprovechar la solución preexistente y adaptarla a las necesidades actuales.
- Los patrones como el **Patrón de Estructura Adapter** permiten adaptar interfaces incompatibles entre componentes existentes y nuevos, facilitando la reutilización y la integración.

¿Qué son los Patrones de Diseño?

- Son soluciones estructurales probadas y documentadas para problemas comunes en el diseño de software.
- Estos patrones encapsulan las mejores prácticas y conocimientos acumulados por expertos en la industria del software a lo largo de los años.
- Proporcionan una forma común y estandarizada de abordar problemas específicos, lo que resulta en un diseño más eficiente y modular.
- Los Patrones de Diseño no son código listo para usar, sino más bien guías que nos permiten resolver problemas de diseño de manera efectiva.

2.

Elementos

Nombre del Patrón de Diseño

- El nombre del Patrón de Diseño es una etiqueta que identifica y describe brevemente el problema y la solución que aborda.
- Los nombres suelen ser descriptivos y proporcionan una idea general del propósito y el contexto del Patrón de Diseño.
- Por ejemplo, algunos nombres comunes de Patrones de Diseño son "Singleton", "Observer" y "Factory Method".
- El nombre del Patrón de Diseño es una forma de comunicación efectiva entre los diseñadores y desarrolladores de software, ya que proporciona un lenguaje común para referirse a soluciones específicas.

Descripción del Problema

- La descripción del problema en un Patrón de Diseño define y explica el escenario o la situación en la que se aplica el patrón.
- Se describe el contexto en el que se enfrenta el problema y se presentan los requisitos y restricciones asociados.
- La descripción del problema permite comprender mejor el desafío que se aborda y ayuda a identificar cuándo es apropiado aplicar el Patrón de Diseño en una situación determinada

Solución Propuesta

- La solución propuesta en un Patrón de Diseño describe la estructura y las interacciones entre los elementos del diseño.
- Proporciona una guía o un conjunto de pautas para resolver el problema de manera efectiva.
- La solución puede incluir la organización de las clases, la asignación de responsabilidades y los mecanismos de comunicación entre los objetos involucrados.
- Al aplicar la solución propuesta, se espera obtener un diseño más flexible, mantenible y reutilizable en el contexto específico del problema abordado.

Ejemplos y Aplicaciones

- Los ejemplos y aplicaciones demuestran cómo se utiliza el Patrón de Diseño en situaciones reales.
- Se presentan casos prácticos donde el patrón ha sido aplicado con éxito y se ilustra su utilidad en diferentes escenarios.
- Los ejemplos ayudan a visualizar cómo el patrón resuelve problemas específicos y brindan inspiración para su implementación en proyectos futuros.

3. Categorías

Patrones de Creación

- Los Patrones de Creación se centran en cómo crear objetos de manera eficiente y flexible.
- Patrones importantes en esta categoría: Singleton, Abstract Factory, Builder, Factory Method.
- **Ejemplo: El Patrón de Diseño "Factory Method"**
 - Proporciona una interfaz para crear objetos, pero delega la responsabilidad de decidir qué tipo de objeto crear a las subclases.
 - Permite la creación de objetos sin especificar explícitamente la clase concreta, lo que facilita la flexibilidad y el mantenimiento del código.

Patrones de Estructura

- Los Patrones de Estructura se centran en cómo organizar las clases y objetos para formar estructuras más grandes y complejas.
- Patrones importantes en esta categoría: Decorator, Proxy, Composite, Adapter.
- **Ejemplo: El Patrón de Diseño "Adapter"**
 - Este patrón permite que dos interfaces incompatibles trabajen juntas al actuar como un puente entre ellas.
 - Convierte la interfaz de una clase en otra interfaz que los clientes esperan, lo que facilita la integración de componentes existentes.

Patrones de Comportamiento

- Los Patrones de Comportamiento se centran en cómo las clases y objetos interactúan y distribuyen las responsabilidades entre ellos.
- Patrones importantes en esta categoría: Strategy, Template Method, Iterator.
- **Ejemplo: Patrón de Diseño "Observer"**
 - Este patrón define una dependencia uno a muchos entre objetos, de modo que cuando un objeto cambia su estado, todos los objetos dependientes son notificados y actualizados automáticamente.
 - Permite una comunicación eficiente y separada entre objetos en un sistema.

4. Preliminares

El modificador `static` en C++.

```
class Foo{  
    public:  
        static int x;  
};  
  
int Foo::x = 3;  
int main(){  
    Foo obj1;  
    cout << obj1.x << endl;  
  
    Foo obj2;  
    cout << obj2.x << endl;  
  
    return 0;  
}
```

El modificador `static` en C++.

```

class Foo{
    public:
        static int x;
};


int Foo::x = 3;
int main(){
    Foo obj1;
    cout << obj1.x << endl;

    Foo obj2;
    cout << obj2.x << endl;

    return 0;
}

```

Un dato común para todas las instancias



El modificador `static` en C++.

```
class Foo{
    public:
        static int x;
};
```

```
int Foo::x = 3;
```


```
int main(){
    Foo obj1;
    cout << obj1.x << endl;
```

```
    Foo obj2;
    cout << obj2.x << endl;
```

```
    return 0;
```

```
}
```

Se puede acceder sin tener siquiera una instancia



El modificador `static` en C++.

```

class Foo{
    public:
        static int x;
        int y;

        static void f1(){
            x = 2;
            y = 3;
        }
        void f2(){
            x = 2;
            y = 3;
        }
};
    
```



Un método estático sólo utilizará miembros estáticos de su clase (Foo).

El modificador `static` en C++.

```

class Foo{
    public:
        static int x;
        int y;

        static void f1(){
            x = 2;
            y = 3;
        }
        void f2(){
            x = 2;
            y = 3;
        }
};
    
```



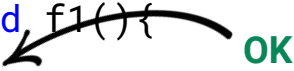
Un método estático sólo utilizará miembros estáticos de su clase (Foo).

El modificador `static` en C++.

```

class Foo{
    public:
        static int x;
        int y;

        static void f1(){
            x = 2;
            y = 3;
        }
        void f2(){
            x = 2;
            y = 3;
        }
};
  
```




El modificador `static` en C++.

```

class Foo{
    public:
        static int x;
        int y;

        static void f1(){
            x = 2;
            y = 3;
        }
        void f2(){
            x = 2;
            y = 3;
        }
};
  
```


ERROR!

5.

Patrones de Creación

Singleton

- **Funcionamiento:** El Patrón *Singleton* garantiza que solo exista una única instancia de una clase en todo el programa.
- **Problema que resuelve:** El Patrón Singleton resuelve el problema de tener múltiples instancias de una clase que deben compartir un estado común.
- **Aplicaciones reales:** Controladores de base de datos, registros de eventos, gestores de configuración.
- **Guía para su implementación:**
 1. Privatizar el constructor de la clase.
 2. Definir una variable estática privada que contenga la única instancia de la clase.
 3. Proporcionar un método público estático para acceder a la instancia única.

Singleton: Ejemplo en C++

```
class Singleton {
private:
    static Singleton* instance;

    // Constructor privado
    Singleton() {}

public:
    // Método estático para acceder a la instancia única
    static Singleton* getInstance() {
        if (instance == nullptr) {
            instance = new Singleton();
        }
        return instance;
    }
};
```

No se puede crear instancias
fuera de la clase

Singleton: Ejemplo en C++

```
int main() {  
    // Uso del Singleton  
    Singleton* instance = Singleton::getInstance();  
    // ...  
    return 0;  
}
```

Abstract Factory

- **Funcionamiento:** El Patrón *Abstract Factory* proporciona una interfaz para crear familias de objetos relacionados sin especificar sus clases concretas.
- **Problema que resuelve:** Permite la creación de objetos relacionados sin acoplar el código cliente a implementaciones concretas.
- **Aplicaciones reales:** Frameworks de UI, librerías de persistencia, sistemas de GUI multiplataforma.
- **Guía para su implementación:**
 1. Definir interfaces abstractas para los productos que se crearán.
 2. Implementar clases concretas que implementen las interfaces abstractas para cada familia de productos.
 3. Definir una interfaz abstracta para la fábrica que creará los productos.
 4. Implementar clases concretas de fábrica que producen objetos de una familia específica.

Abstract Factory: Ejemplo en C++

```
// Productos abstractos
class AbstractProductA {
public:
    virtual void operation() = 0;
};

class AbstractProductB {
public:
    virtual void operation() = 0;
};
```

Definen las operaciones que los productos deben implementar

Abstract Factory: Ejemplo en C++

```
// Productos concretos
class ConcreteProductA : public AbstractProductA {
public:
    void operation() override {
        // Implementación específica para ConcreteProductA
    }
};

class ConcreteProductB : public AbstractProductB {
public:
    void operation() override {
        // Implementación específica para ConcreteProductB
    }
};
```

Abstract Factory: Ejemplo en C++

```
// Fábrica abstracta
class AbstractFactory {
public:
    virtual AbstractProductA* createProductA() = 0;
    virtual AbstractProductB* createProductB() = 0;
};
```

Define los métodos para
crear productos

```
// Fábrica concreta
class ConcreteFactory : public AbstractFactory {
public:
    AbstractProductA* createProductA() override {
        return new ConcreteProductA();
    }

    AbstractProductB* createProductB() override {
        return new ConcreteProductB();
    }
};
```

Implementa los métodos

Abstract Factory: Ejemplo en C++

```
int main() {  
    // Uso del Abstract Factory  
    AbstractFactory* factory = new ConcreteFactory();  
    AbstractProductA* productA = factory->createProductA();  
    AbstractProductB* productB = factory->createProductB();  
    // ...  
    return 0;  
}
```

Factory Method

- **Funcionamiento:** El patrón Factory Method proporciona una interfaz para crear objetos, pero delega la decisión de qué tipo de objeto crear a las subclases.
- **Problema que resuelve:** Resuelve el problema de crear objetos sin especificar la clase concreta, permitiendo la flexibilidad y extensibilidad en la creación de objetos.
- **Aplicaciones reales:** Frameworks de desarrollo web para personalizar la creación de controladores, frameworks de pruebas unitarias que necesitan crear objetos de prueba personalizados, entre otros.
- **Guía para su implementación:**
 1. Definir una interfaz o clase base para el producto.
 2. Crear una clase creadora abstracta con un método de fábrica abstracto para crear objetos del producto.
 3. Implementar subclases concretas de la clase creadora que proporcionen la implementación específica del método de fábrica para crear objetos del producto.

Factory Method: Ejemplo en C++

```
// Producto  
class Entrega {  
public:  
    virtual void entregar() = 0;  
};
```

Imaginemos que estamos desarrollando un sistema de logística donde necesitamos implementar diferentes servicios de entrega, como entrega en moto, entrega en carro y entrega en barco.

Factory Method: Ejemplo en C++

```
// Productos concretos
class Moto : public Entrega {
public:
    void entregar() override {
        // Implementación específica para la entrega en moto
    }
};

class Carro : public Entrega {
public:
    void entregar() override {
        // Implementación específica para la entrega en carro
    }
};

class Barco : public Entrega {
public:
    void entregar() override {
        // Implementación específica para la entrega en barco
    }
};
```

Factory Method: Ejemplo en C++

```
// Creador abstracto
class Creador {
public:
    virtual Entrega* crearEntrega() = 0;
};
```

/*

La clase **Creador** es la clase creadora abstracta con el método de fábrica abstracto para crear objetos del producto.

*/

Factory Method: Ejemplo en C++

```
// Creadores concretos
class MotoCreador : public Creador {
public:
    Entrega* crearEntrega() override {
        return new Moto();
    }
};

class CarroCreador : public Creador {
public:
    Entrega* crearEntrega() override {
        return new Carro();
    }
};

class BarcoCreador : public Creador {
public:
    Entrega* crearEntrega() override {
        return new Barco();
    }
};
```

Proporcionan **implementaciones** específicas del método de fábrica para crear objetos del producto.

Factory Method: Ejemplo en C++

```
int main() {  
    // Uso del Factory Method  
    Creador* creador = new MotoCreador();  
    Entrega* entrega = creador->crearEntrega();  
    entrega->entregar();  
    // ...  
    return 0;  
}
```

Builder

- **Funcionamiento:** El Patrón Builder separa la construcción de un objeto complejo de su representación, permitiendo la creación paso a paso del objeto.
- **Problema que resuelve:** Resuelve el problema de construir objetos complejos paso a paso, permitiendo la variabilidad en la creación y evitando la explosión combinatoria de constructores.
- **Aplicaciones reales:** Creación de documentos, consultas SQL, y objetos gráficos.
- **Guía para su implementación:**
 1. Definir una interfaz para el builder que especifique los métodos para construir cada parte del objeto.
 2. Crear una clase concreta del builder que implemente los métodos para construir cada parte del objeto.
 3. Definir una clase director que reciba un builder y utilice sus métodos para construir el objeto paso a paso.
 4. Obtener el objeto construido del director.

Builder

```
int main() {
    // Creación del builder
    Builder* builder = new EmailBuilder();

    // Construcción del objeto usando el builder
    builder->construirDestinatario("destinatario@example.com");
    builder->construirRemitente("remitente@example.com");
    builder->construirAsunto("Asunto del email");
    builder->construirCuerpo("Cuerpo del email");

    // Obtención del resultado (Email)
    Email* email = builder->obtenerResultado();

    // Uso del objeto resultante
    email->enviar();

    delete email;
    delete builder;
}
```

Builder

```
class Email {  
private:  
    std::string destinatario_;  
    std::string remitente_;  
    std::string asunto_;  
    std::string cuerpo_;  
// Continuará. . .
```

Creamos una clase **Email** que representa el objeto de correo electrónico a construir. La clase **Email** tiene métodos para establecer cada parte del correo electrónico, como remitente, destinatario, asunto y contenido.

Builder

```
// . . .
public:
    void setDestinatario(const std::string& destinatario) {
        destinatario_ = destinatario;
    }
    void setRemitente(const std::string& remitente) { remitente_ = remitente; }
    void setAsunto(const std::string& asunto) { asunto_ = asunto; }
    void setCuerpo(const std::string& cuerpo) { cuerpo_ = cuerpo; }

    void enviar() {
        // Lógica para enviar el email
        std::cout << "Enviando email..." << std::endl;
        std::cout << "De: " << remitente_ << std::endl;
        std::cout << "Para: " << destinatario_ << std::endl;
        std::cout << "Asunto: " << asunto_ << std::endl;
        std::cout << "Cuerpo: " << cuerpo_ << std::endl;
    }
};
```


Builder

```
// Builder abstracto
class Builder {
public:
    virtual void construirDestinatario(
        const std::string& destinatario) = 0;
    virtual void construirRemitente(const std::string& remitente) = 0;
    virtual void construirAsunto(const std::string& asunto) = 0;
    virtual void construirCuerpo(const std::string& cuerpo) = 0;
    virtual Email* obtenerResultado() = 0;
};
```

Creamos una interfaz **Builder** que declara métodos abstractos para construir cada parte del correo electrónico

Builder

```
// Builder concreto
class EmailBuilder : public Builder {
public:
    void construirDestinatario(const std::string& destinatario) override {
        email_->setDestinatario(destinatario);
    }

    void construirRemitente(const std::string& remitente) override {
        email_->setRemitente(remitente);
    }

    void construirAsunto(const std::string& asunto) override {
        email_->setAsunto(asunto);
    }

    void construirCuerpo(const std::string& cuerpo) override {
        email_->setCuerpo(cuerpo);
    }
}

// Continuará. . .
```

Builder

```
// . . .
Email* obtenerResultado() override {
    return email_;
}

private:
    Email* email_ = new Email();
};
```

El **EmailBuilder** implementa las operaciones del **Builder** para construir las diferentes partes del email, como el destinatario, remitente, asunto y cuerpo. Finalmente, en la función principal, se obtiene el email resultante a través del método **obtenerResultado()** y se utiliza para enviar el email mediante el método **enviar()**.

Resumen

En esta sesión se trataron los tópicos siguientes:

- Patrones de Diseño
- Elementos
- Categorías
- Patrones de Creación

