**Semana 2: Patrones Creacionales**

**Temas de la sesión**

**1. Singleton**

El patrón **Singleton** asegura que una clase tenga solo una instancia y proporciona un punto de acceso global a esa instancia. Esto es útil cuando quieres controlar el acceso a un recurso compartido, como una base de datos o un archivo de configuración.

* **Características clave**:
  + La clase tiene un constructor privado para evitar la creación de instancias desde fuera.
  + Proporciona un método estático para obtener la única instancia de la clase.
  + La instancia se crea de manera perezosa (lazy initialization) o en el momento de la primera solicitud.
* **Caso de uso**:
  + Conexiones a bases de datos.
  + Configuraciones globales.
  + Loggers.
* **Patron en la practica**:

public class Banco {

private static Banco instance;

private String nombreBanco;

private Banco() {

nombreBanco = "Banco Central";

}

public static Banco getInstance() {

if (instance == null) {

instance = new Banco();

}

return instance;

}

public void mostrarNombre() {

System.out.println("Bienvenido al " + nombreBanco);

}

}

// Uso

public class Main {

public static void main(String[] args) {

Banco banco = Banco.getInstance();

banco.mostrarNombre();

}

}

**2. Factory Method**

El patrón **Factory Method** define una interfaz para crear un objeto, pero permite a las subclases decidir qué clase instanciar. Este patrón delega la responsabilidad de la creación de objetos a las subclases.

* **Características clave**:
  + Define un método abstracto para crear objetos (el "factory method").
  + Las subclases implementan este método para crear instancias específicas.
  + Permite que una clase delegue la creación de objetos a sus subclases.
* **Caso de uso**:
  + Frameworks de aplicaciones que necesitan crear objetos de diferentes tipos dependiendo del contexto.
  + Cuando una clase no puede anticipar el tipo de objetos que debe crear.
* **Patron en la practica**:

// Producto abstracto

interface Medicamento {

void administrar();

}

// Producto concreto: Analgésico

class Analgesico implements Medicamento {

@Override

public void administrar() {

System.out.println("Administrando analgésico para el dolor.");

}

}

// Producto concreto: Antibiótico

class Antibiotico implements Medicamento {

@Override

public void administrar() {

System.out.println("Administrando antibiótico para la infección.");

}

}

// Creador abstracto

abstract class Farmacia {

public abstract Medicamento crearMedicamento();

public void recetarMedicamento() {

Medicamento medicamento = crearMedicamento();

medicamento.administrar();

}

}

// Creador concreto para Analgésico

class FarmaciaAnalgesico extends Farmacia {

@Override

public Medicamento crearMedicamento() {

return new Analgesico();

}

}

// Creador concreto para Antibiótico

class FarmaciaAntibiotico extends Farmacia {

@Override

public Medicamento crearMedicamento() {

return new Antibiotico();

}

}

// Uso

public class Main {

public static void main(String[] args) {

Farmacia farmaciaAnalgesico = new FarmaciaAnalgesico();

farmaciaAnalgesico.recetarMedicamento();

Farmacia farmaciaAntibiotico = new FarmaciaAntibiotico();

farmaciaAntibiotico.recetarMedicamento();

}

}

**3. Abstract Factory**

El patrón **Abstract Factory** proporciona una interfaz para crear familias de objetos relacionados o dependientes sin especificar sus clases concretas. Este patrón es útil cuando necesitas crear grupos de objetos que están diseñados para trabajar juntos.

* **Características clave**:
  + Define una interfaz abstracta para crear familias de objetos.
  + Cada fábrica concreta implementa esta interfaz para crear objetos específicos.
  + Permite cambiar fácilmente la familia de objetos que se están utilizando.
* **Caso de uso**:
  + Creación de interfaces gráficas de usuario (GUI) que deben ser consistentes en diferentes sistemas operativos.
  + Sistemas que necesitan ser independientes de cómo se crean, componen o representan sus productos.
* **Patron en la practica**:

// Producto abstracto: Cuenta

interface Cuenta {

void abrir();

}

// Producto concreto: Cuenta Corriente

class CuentaCorriente implements Cuenta {

@Override

public void abrir() {

System.out.println("Abriendo cuenta corriente.");

}

}

// Producto concreto: Cuenta Ahorros

class CuentaAhorros implements Cuenta {

@Override

public void abrir() {

System.out.println("Abriendo cuenta de ahorros.");

}

}

// Producto abstracto: Tarjeta

interface Tarjeta {

void emitir();

}

// Producto concreto: Tarjeta Débito

class TarjetaDebito implements Tarjeta {

@Override

public void emitir() {

System.out.println("Emitiendo tarjeta de débito.");

}

}

// Producto concreto: Tarjeta Crédito

class TarjetaCredito implements Tarjeta {

@Override

public void emitir() {

System.out.println("Emitiendo tarjeta de crédito.");

}

}

// Fábrica abstracta

interface BancoFactory {

Cuenta crearCuenta();

Tarjeta crearTarjeta();

}

// Fábrica concreta para Cuenta Corriente y Tarjeta Débito

class BancoCorrienteFactory implements BancoFactory {

@Override

public Cuenta crearCuenta() {

return new CuentaCorriente();

}

@Override

public Tarjeta crearTarjeta() {

return new TarjetaDebito();

}

}

// Fábrica concreta para Cuenta Ahorros y Tarjeta Crédito

class BancoAhorrosFactory implements BancoFactory {

@Override

public Cuenta crearCuenta() {

return new CuentaAhorros();

}

@Override

public Tarjeta crearTarjeta() {

return new TarjetaCredito();

}

}

// Uso

public class Main {

public static void main(String[] args) {

BancoFactory bancoCorriente = new BancoCorrienteFactory();

Cuenta cuentaCorriente = bancoCorriente.crearCuenta();

Tarjeta tarjetaDebito = bancoCorriente.crearTarjeta();

cuentaCorriente.abrir();

tarjetaDebito.emitir();

BancoFactory bancoAhorros = new BancoAhorrosFactory();

Cuenta cuentaAhorros = bancoAhorros.crearCuenta();

Tarjeta tarjetaCredito = bancoAhorros.crearTarjeta();

cuentaAhorros.abrir();

tarjetaCredito.emitir();

}

}

**4. Builder**

El patrón **Builder** separa la construcción de un objeto complejo de su representación, permitiendo que el mismo proceso de construcción cree diferentes representaciones. Este patrón es útil cuando un objeto necesita ser creado con muchos parámetros opcionales o configuraciones.

* **Características clave**:
  + El proceso de construcción se divide en pasos.
  + Un director controla el proceso de construcción.
  + Diferentes constructores pueden crear diferentes representaciones del objeto.
* **Caso de uso**:
  + Creación de objetos complejos como documentos XML o JSON.
  + Construcción de objetos con muchos parámetros opcionales (por ejemplo, una pizza con diferentes ingredientes).
* **Patron en la practica**:

// Producto: Paciente

class Paciente {

private String nombre;

private int edad;

private String diagnostico;

public void setNombre(String nombre) {

this.nombre = nombre;

}

public void setEdad(int edad) {

this.edad = edad;

}

public void setDiagnostico(String diagnostico) {

this.diagnostico = diagnostico;

}

public void mostrarInfo() {

System.out.println("Paciente: " + nombre + ", Edad: " + edad + ", Diagnóstico: " + diagnostico);

}

}

// Builder abstracto

interface PacienteBuilder {

void construirNombre();

void construirEdad();

void construirDiagnostico();

Paciente getPaciente();

}

// Builder concreto para Paciente con Gripe

class PacienteGripeBuilder implements PacienteBuilder {

private Paciente paciente;

public PacienteGripeBuilder() {

this.paciente = new Paciente();

}

@Override

public void construirNombre() {

paciente.setNombre("Juan Pérez");

}

@Override

public void construirEdad() {

paciente.setEdad(30);

}

@Override

public void construirDiagnostico() {

paciente.setDiagnostico("Gripe");

}

@Override

public Paciente getPaciente() {

return paciente;

}

}

// Director

class Medico {

private PacienteBuilder pacienteBuilder;

public void setPacienteBuilder(PacienteBuilder pacienteBuilder) {

this.pacienteBuilder = pacienteBuilder;

}

public Paciente getPaciente() {

return pacienteBuilder.getPaciente();

}

public void construirPaciente() {

pacienteBuilder.construirNombre();

pacienteBuilder.construirEdad();

pacienteBuilder.construirDiagnostico();

}

}

// Uso

public class Main {

public static void main(String[] args) {

Medico medico = new Medico();

PacienteBuilder pacienteGripeBuilder = new PacienteGripeBuilder();

medico.setPacienteBuilder(pacienteGripeBuilder);

medico.construirPaciente();

Paciente paciente = medico.getPaciente();

paciente.mostrarInfo();

}

}

**5. Prototype**

El patrón **Prototype** permite la creación de nuevos objetos copiando un objeto existente (prototipo) en lugar de crear nuevos objetos desde cero. Este patrón es útil cuando la creación de un objeto es más costosa en términos de recursos o tiempo.

* **Características clave**:
  + Los objetos que se pueden copiar implementan una interfaz o método clone.
  + El proceso de clonación puede ser superficial (shallow) o profunda (deep), dependiendo de las necesidades.
  + Permite la creación dinámica de objetos sin conocer su tipo exacto.
* **Caso de uso**:
  + Cuando la creación de un objeto es más costosa que copiar uno existente.
  + Sistemas que necesitan crear múltiples instancias de objetos con estados similares.
* **Patron en la practica**:

// Prototipo

interface CuentaBancaria extends Cloneable {

CuentaBancaria clone() throws CloneNotSupportedException;

void mostrarInfo();

}

// Producto concreto: Cuenta de Ahorros

class CuentaAhorros implements CuentaBancaria {

private String titular;

private double saldo;

public CuentaAhorros(String titular, double saldo) {

this.titular = titular;

this.saldo = saldo;

}

@Override

public CuentaBancaria clone() throws CloneNotSupportedException {

return (CuentaAhorros) super.clone();

}

@Override

public void mostrarInfo() {

System.out.println("Cuenta de Ahorros - Titular: " + titular + ", Saldo: " + saldo);

}

}

// Uso

public class Main {

public static void main(String[] args) {

try {

CuentaAhorros cuentaOriginal = new CuentaAhorros("Carlos Gómez", 1000.0);

CuentaAhorros cuentaClonada = (CuentaAhorros) cuentaOriginal.clone();

cuentaOriginal.mostrarInfo();

cuentaClonada.mostrarInfo();

cuentaClonada.mostrarInfo();

} catch (CloneNotSupportedException e) {

e.printStackTrace();

}

}

}

# Tarea en Clase

**Ejercicios Conceptuales**

¿Cuál es el propósito principal del patrón Singleton?

A) Permitir la creación de múltiples instancias de una clase.

B) Asegurar que una clase tenga solo una instancia y proporcionar un punto de acceso global a ella.

C) Delegar la creación de objetos a subclases.

D) Separar la construcción de un objeto complejo de su representación.

¿Qué problema resuelve el patrón Factory Method?

A) La necesidad de crear familias de objetos relacionados.

B) La dificultad de crear objetos complejos paso a paso.

C) La necesidad de clonar objetos existentes.

D) La delegación de la creación de objetos a subclases.

¿Cuál es la principal diferencia entre Factory Method y Abstract Factory?

A) Factory Method crea un solo objeto, mientras que Abstract Factory crea familias de objetos relacionados.

B) Factory Method usa herencia, mientras que Abstract Factory usa composición.

C) Factory Method es más eficiente que Abstract Factory.

D) Factory Method no permite la creación de objetos complejos.

¿Cuál es la ventaja principal del patrón Builder?

A) Permite clonar objetos existentes de manera eficiente.

B) Separa la construcción de un objeto complejo de su representación, permitiendo el mismo proceso de construcción para diferentes representaciones.

C) Asegura que una clase tenga solo una instancia.

D) Delega la creación de objetos a subclases.

¿En qué situaciones es más útil el patrón Prototype?

A) Cuando se necesita crear objetos complejos paso a paso.

B) Cuando se quiere evitar la creación de objetos desde cero y en su lugar clonar un objeto existente.

C) Cuando se necesita asegurar que una clase tenga solo una instancia.

D) Cuando se quiere delegar la creación de objetos a subclases.

# Taller

**Ejercicios Prácticos**

1. Sector: Recreación

Historia de Usuario:

Como usuario, quiero poder reservar una cancha de fútbol para un día y hora específicos, para poder jugar con mis amigos.

Solución:

java

import java.util.HashMap;

import java.util.Map;

class CanchaReserva {

private Map<String, String> reservas = new HashMap<>();

public boolean reservarCancha(String fecha, String hora, String usuario) {

String clave = fecha + " " + hora;

if (reservas.containsKey(clave)) {

return false; // Ya está reservada

}

reservas.put(clave, usuario);

return true;

}

public void mostrarReservas() {

for (Map.Entry<String, String> entry : reservas.entrySet()) {

System.out.println("Fecha y Hora: " + entry.getKey() + ", Usuario: " + entry.getValue());

}

}

}

public class Main {

public static void main(String[] args) {

CanchaReserva cancha = new CanchaReserva();

cancha.reservarCancha("2023-10-15", "18:00", "Juan Pérez");

cancha.reservarCancha("2023-10-15", "20:00", "Carlos Gómez");

cancha.mostrarReservas();

}

}

2. Sector: Manufactura de Acero

Historia de Usuario:

Como supervisor de producción, quiero poder registrar la cantidad de acero producido por cada máquina en un turno, para poder llevar un control de la producción.

Solución:

java

import java.util.ArrayList;

import java.util.List;

class Maquina {

private String nombre;

private double aceroProducido;

public Maquina(String nombre) {

this.nombre = nombre;

this.aceroProducido = 0;

}

public void producirAcero(double cantidad) {

this.aceroProducido += cantidad;

}

public String getNombre() {

return nombre;

}

public double getAceroProducido() {

return aceroProducido;

}

}

class Turno {

private List<Maquina> maquinas = new ArrayList<>();

public void agregarMaquina(Maquina maquina) {

maquinas.add(maquina);

}

public void mostrarProduccion() {

for (Maquina maquina : maquinas) {

System.out.println("Máquina: " + maquina.getNombre() + ", Acero Producido: " + maquina.getAceroProducido() + " toneladas");

}

}

}

public class Main {

public static void main(String[] args) {

Turno turno = new Turno();

Maquina maquina1 = new Maquina("Máquina 1");

Maquina maquina2 = new Maquina("Máquina 2");

maquina1.producirAcero(5.3);

maquina2.producirAcero(7.8);

turno.agregarMaquina(maquina1);

turno.agregarMaquina(maquina2);

turno.mostrarProduccion();

}

}

3. Sector: Bancos

Historia de Usuario:

Como cliente del banco, quiero poder transferir dinero de mi cuenta a otra cuenta, para poder realizar pagos o enviar dinero a familiares.

Solución:

java

class CuentaBancaria {

private String numeroCuenta;

private double saldo;

public CuentaBancaria(String numeroCuenta, double saldoInicial) {

this.numeroCuenta = numeroCuenta;

this.saldo = saldoInicial;

}

public boolean transferir(CuentaBancaria destino, double monto) {

if (this.saldo >= monto) {

this.saldo -= monto;

destino.saldo += monto;

return true;

}

return false;

}

public void mostrarSaldo() {

System.out.println("Cuenta: " + numeroCuenta + ", Saldo: " + saldo);

}

}

public class Main {

public static void main(String[] args) {

CuentaBancaria cuenta1 = new CuentaBancaria("123456", 1000.0);

CuentaBancaria cuenta2 = new CuentaBancaria("654321", 500.0);

cuenta1.transferir(cuenta2, 200.0);

cuenta1.mostrarSaldo();

cuenta2.mostrarSaldo();

}

}

4. Sector: Transporte

Historia de Usuario:

Como pasajero, quiero poder comprar un boleto de autobús para un viaje específico, para poder viajar a mi destino.

Solución:

java

class Boleto {

private String viaje;

private String pasajero;

public Boleto(String viaje, String pasajero) {

this.viaje = viaje;

this.pasajero = pasajero;

}

public void mostrarBoleto() {

System.out.println("Viaje: " + viaje + ", Pasajero: " + pasajero);

}

}

class Autobus {

public Boleto comprarBoleto(String viaje, String pasajero) {

return new Boleto(viaje, pasajero);

}

}

public class Main {

public static void main(String[] args) {

Autobus autobus = new Autobus();

Boleto boleto = autobus.comprarBoleto("Ciudad A - Ciudad B", "Juan Pérez");

boleto.mostrarBoleto();

}

}

5. Sector: Manufactura de Acero

Historia de Usuario:

Como gerente de planta, quiero poder calcular el costo total de producción de un lote de acero, para poder planificar el presupuesto.

Solución:

java

class LoteAcero {

private double cantidad; // en toneladas

private double costoPorTonelada;

public LoteAcero(double cantidad, double costoPorTonelada) {

this.cantidad = cantidad;

this.costoPorTonelada = costoPorTonelada;

}

public double calcularCostoTotal() {

return cantidad \* costoPorTonelada;

}

}

public class Main {

public static void main(String[] args) {

LoteAcero lote = new LoteAcero(50.0, 1200.0);

System.out.println("Costo total del lote: $" + lote.calcularCostoTotal());

}

}