

Patrones de Diseño

Patrones Creacionales.

Se encargan de la creación de objetos.

- Factory Method: Permite la creación de objetos de una clase sin especificar el tipo exacto del objeto.
- Abstract Factory: Crea familias de objetos relacionados sin especificar las clases concretas.
- Builder: Separa la construcción de un objeto complejo de su representación.

- Prototype: Crea nuevos objetos clonando una instancia existente.
- Singleton: Asegura que una clase tenga solo una instancia y proporciona un punto de acceso global a ella.

Ejemplo:

```
class Singleton {
    private static Singleton instance;

    private Singleton() {}

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

// Uso
public class Main {
    public static void main(String[] args) {
        Singleton s1 = Singleton.getInstance();
        Singleton s2 = Singleton.getInstance();
        System.out.println(s1 == s2); // Output: true
    }
}
```

PATRONES ESTRUCTURALES.

Se enfocan en la composición de clases u objetos.

- Adapter: Convierte la interfaz de una clase en otra interfaz que el cliente espera.
- Bridge: Desacopla una abstracción de su implementación.
- Composite: Permite tratar objetos individuales y compuestos de la misma manera.
- Decorator: Añade responsabilidades adicionales a un objeto de manera dinámica.

- Facade: Proporciona una interfaz simplificada para un subsistema complejo.
- Flyweight: Minimiza el uso de memoria compartiendo datos entre objetos similares.
- Proxy: Proporciona un sustituto o marcador de posición para controlar el acceso a otro objeto.

PATRONES DE COMPORTAMIENTO

Se centran en la interacción y comunicación entre objetos.

- Chain of Responsibility: Pasa una solicitud a lo largo de una cadena de handlers.
- Command: Convierte una solicitud en un objeto, permitiendo parametrizar clientes con diferentes solicitudes.
- Interpreter: Implementa una gramática para interpretar frases en un lenguaje.
- Iterator: Proporciona una forma de acceder a los elementos de una colección secuencialmente.
- Mediator: Define un objeto que centraliza la comunicación entre los demás objetos.

- Memento: Permite capturar y restaurar el estado interno de un objeto.
- Observer: Permite que los objetos observen y reaccionen a cambios en otros objetos.
- State: Permite a un objeto cambiar su comportamiento cuando cambia su estado interno.
- Strategy: Define una familia de algoritmos, permitiendo a los clientes seleccionar uno en tiempo de ejecución.
- Template Method: Define el esqueleto de un algoritmo en una operación, permitiendo que las subclases completen los pasos.
- Visitor: Permite definir una operación sobre elementos de una estructura de objetos.

```

Ejemplo:
interface EnglishSpeaker {
    String greet();
}

class EnglishGreeter implements EnglishSpeaker {
    public String greet() {
        return "Hello!";
    }
}

class SpanishSpeaker {
    public String saludar() {
        return "¡Hola!";
    }
}

class SpanishAdapter implements EnglishSpeaker {
    private SpanishSpeaker spanishSpeaker;

    public SpanishAdapter(SpanishSpeaker spanishSpeaker) {
        this.spanishSpeaker = spanishSpeaker;
    }

    public String greet() {
        return spanishSpeaker.saludar();
    }
}

// Uso
public class Main {
    public static void main(String[] args) {
        EnglishSpeaker spanishGreeter = new SpanishAdapter(new
        SpanishSpeaker());
        System.out.println(spanishGreeter.greet()); // Output: "¡Hola!"
    }
}

```

```

import java.util.ArrayList;
import java.util.List;

// Interfaz Observer
interface Observer {
    void update(String message);
}

// Clase concreta que implementa Observer
class ConcreteObserver implements Observer {
    @Override
    public void update(String message) {
        System.out.println("Received: " + message);
    }
}

// Clase Subject que mantiene una lista de observadores
class Subject {
    private List<Observer> observers = new ArrayList<>();

    public void attach(Observer observer) {
        observers.add(observer);
    }

    public void detach(Observer observer) {
        observers.remove(observer);
    }

    public void notifyObservers(String message) {
        for (Observer observer : observers) {
            observer.update(message);
        }
    }
}

// Uso
public class Main {
    public static void main(String[] args) {
        Subject subject = new Subject();
        Observer obs1 = new ConcreteObserver();
        Observer obs2 = new ConcreteObserver();

        subject.attach(obs1);
        subject.attach(obs2);

        subject.notifyObservers("Hello Observers"); // Output:
        "Received: Hello Observers" (x2)
    }
}

```