UNIVERSIDAD DEL VALLE DE GUATEMALA

Facultad de Ingeniería

Proyecto: GS Stock & Facturación.
Patrones de diseño

Genser Catalán 23401, Fernando Ruíz 23065, Hugo Barillas 23306, Jose López 23773

Link repositorio GITHUB: https://github.com/JosFer720/GS-Stock-y-Facturacion

Link CANVA: https://www.canva.com/design/DAGg5XCjlzU/drZ1P0ARxOFg5-mf8q6QAQ/edit?utm_content=DAGg5XCjlzU&utm_campaign=designshare&utm_medium=link2&utm_source=sharebutton

Link: https://uvggt-my.sharepoint.com/:w:/g/personal/lop23773_uvg_edu_gt/EWb1wQacSqpCpu3xAGYbj7EB46Dg9Xd92RX4tD4n6VrV9g?e=DXrhJX

Erick Marroquín

Ingeniería en Software 1 -- CC3090

Guatemala, 2025

Nota: Los tres patrones de diseño seleccionados por el grupo son los siguientes: MVC, Bridge y State.

1. Bridge:

INTENCIÓN
El patrón Bridge tiene como objetivo desacoplar una abstracción de su implementación, permitiendo que ambas puedan desarrollarse de forma independiente. Esto facilita la extensibilidad y el mantenimiento del código. (*Bridge*, s. f.)

CONOCIDO COMO
Patrón Puente (*Bridge*, s. f.)

MOTIVO
Este patrón se considera necesario cuando se quiere evitar la rigidez en el código debido a dependencias fuertes entre la abstracción y su implementación. Permite la evolución de ambas partes de forma independiente, promoviendo la reutilización de código y una mejor organización. (*Bridge*, s. f.)

APLICACIONES
- Uso en sistemas en los que se requiere cambiar implementaciones sin afectar a las abstracciones.
- Interfaces de usuario que se conectan a diferentes lógicas de negocio.

ESTRUCTURA
Se compone de los siguientes elementos:

- Abstracción: Define una interfaz para el control de la implementación.

- Implementador: Define la interfaz para las implementaciones concretas.

- Abstracción refinada: Extiende la abstracción y delega la implementación a un objeto de la interfaz de implementador.

- Implementación concreta: Proporciona la implementación específica de la interfaz de implementador.

PARTICIPANTES

- Abstracción: Interfaz general que utiliza el cliente.

- Implementador: Define las operaciones básicas que las implementaciones deben cumplir.

- Implementaciones concretas: Diferentes versiones de la implementación con funcionalidades específicas.

COLABORACIONES
La abstracción interactúa con la implementación a través de una interfaz común, lo que permite cambiar la implementación sin alterar la abstracción.

CONSECUENCIAS (*Bridge*, s. f.)
Beneficios:

- Fomenta la reutilización de código.
- Reduce la dependencia entre las capas de abstracción e implementación.
- Facilita la extensibilidad y la escalabilidad del sistema.

Riesgos:

Puede aumentar la complejidad del diseño debido a la introducción de nuevas interfaces.

IMPLEMENTACIÓN (*Bridge*, s. f.)
Se deben seguir los siguientes pasos para implementar el patrón Bridge:

- Crear la interfaz de la abstracción.
- Definir una implementación base para la abstracción.
- Crear la interfaz del implementador.
- Definir diferentes implementaciones concretas.
- Asociar la abstracción con la implementación concreta mediante una relación de agregación o composición.

CÓDIGO DE EJEMPLO (Tech, s. f.)

```
class Abstraction {
 constructor(implementation) {
  this.implementation = implementation;
 }

 operation() {
  return this.implementation.operationImplementation();
 }
}
```

```
class ConcreteImplementationA {
  operationImplementation() {
    return "Implementación A: Operación realizada";
  }
}

class ConcreteImplementationB {
  operationImplementation() {
    return "Implementación B: Operación realizada";
  }
}

// Uso del Patrón Bridge
const implementationA = new ConcreteImplementationA();
const abstractionA = new Abstraction(implementationA);
console.log(abstractionA.operation());

const implementationB = new ConcreteImplementationB();
const abstractionB = new Abstraction(implementationB);
console.log(abstractionB.operation());
```

USOS CONOCIDOS

- Bibliotecas de interfaces de usuario con diferentes renderizadores.
- Sistemas de almacenamiento que permiten cambiar entre distintas bases de datos sin modificar la capa de negocio.
- Frameworks de desarrollo de videojuegos que separan la lógica del juego de los motores gráficos.

P. RELACIONADOS

- Adapter: Se usa para convertir una interfaz en otra esperada por el cliente.
- Strategy: Permite definir una familia de algoritmos y encapsularlos de manera intercambiable.

2. State

INTENCION

Permite que un objeto cambie su comportamiento cuando cambia su estado interno, simulando una máquina de estados. (Refactoring.Guru, 2025)

CONOCIDO COMO

Máquina de estados, Estado. (Refactoring.Guru, 2025)

MOTIVO

El patrón de diseño State se caracteriza por modificar su comportamiento dependiendo del estado en el que se encuentra la aplicación. Para lograr esto, es necesario crear una serie de clases que representarán los distintos estados posibles. (Oscar Blancarte, 2021)

APLICACIONES

- Sistemas con múltiples estados y reglas de transición. (Oscar Blancarte, 2021)
- Máquinas expendedoras, reproductores de medios, autenticación de usuarios. (Refactoring.Guru, 2025)
- Servidores con distintos estados operativos que afectan su comportamiento. (Oscar Blancarte, 2021)

ESTRUCTURA

- Contexto: Componente que cambia de estado.
- AbstractState: Clase base para los estados.
- ConcreteState: Clases que representan cada estado específico. (Oscar Blancarte, 2021)

PARTICIPANTES

- Contexto: Mantiene una referencia al estado actual.
- AbstractState: Clase base que define comportamientos por defecto.
- Estados Concretos: Implementan comportamientos según el estado. (Oscar Blancarte, 2021)

COLABORACIONES

El Contexto delega su comportamiento al Estado, y este puede cambiar dinámicamente modificando la referencia del estado actual. (Refactoring.Guru, 2025)

CONSECUENCIAS

- Facilita la expansión del código sin modificar el contexto.

- Mejora la organización del código.
- Puede aumentar el número de clases. (Refactoring.Guru, 2025)

IMPLEMENTACION

1. Crear una clase AbstractState para definir comportamientos base.
2. Implementar múltiples Estados Concretos heredando de AbstractState.
3. El Contexto delega la lógica al estado actual y puede cambiarlo dinámicamente. (Oscar Blancarte, 2021)

CODIGO DE EJEMPLO

```
abstract class Estado {
   void manejar(Contexto contexto) {}
}

class EstadoA extends Estado {
   @Override
   void manejar(Contexto contexto) {
      System.out.println("Estado A activo, cambiando a B");
      contexto.setEstado(new EstadoB());
   }
}

class EstadoB extends Estado {
   @Override
   void manejar(Contexto contexto) {
      System.out.println("Estado B activo, cambiando a C");
      contexto.setEstado(new EstadoC());
   }
}

class EstadoC extends Estado {
   @Override
   void manejar(Contexto contexto) {
      System.out.println("Estado C activo");
   }
}

class Contexto {
   private Estado estado;
   public Contexto() { this.estado = new EstadoA(); }
   public void setEstado(Estado estado) { this.estado = estado; }
   public void solicitar() { estado.manejar(this); }
```

}

USOS CONOCIDOS

- TCP/IP: Estados de conexión.
- Máquinas expendedoras: Pago, selección, entrega.
- Edición de documentos: Modos de edición y solo lectura. (Refactoring.Guru, 2025)
- Servidores: Responden de forma distinta según su estado actual. (Oscar Blancarte, 2021)

P. RELACIONADOS

- Strategy: Similar, pero los estados pueden ser intercambiables.
- Singleton: Puede usarse para instancias únicas de estados.
- (Refactoring.Guru, 2025)

3. MVC (Model-View-Controller)

INTENCIÓN
El patrón MVC separa la aplicación en tres componentes principales: Modelo, Vista y Controlador. Esto permite una organización clara del código, facilitando el mantenimiento y la escalabilidad del sistema. (MDN Web Docs, s. f.)

CONOCIDO COMO
Modelo-Vista-Controlador

MOTIVO
Este patrón permite una separación de preocupaciones en el desarrollo de software. Cada componente maneja una responsabilidad específica: el Modelo gestiona los datos, la Vista maneja la interfaz de usuario y el Controlador actúa como intermediario. Esto facilita la reutilización de código y la prueba independiente de cada componente. (MDN Web Docs, s. f.)

APLICACIONES

- Desarrollo de aplicaciones web y de escritorio con interfaces de usuario estructuradas.
- Aplicaciones con una lógica de negocio compleja separada de la interfaz de usuario.
- Sistemas modulares que requieren mantenimiento y escalabilidad.

ESTRUCTURA
El patrón MVC se compone de los siguientes elementos:

- **Modelo:** Representa los datos y la lógica de negocio.

- **Vista:** Se encarga de la representación visual de los datos.
- **Controlador:** Actúa como intermediario entre el Modelo y la Vista.

PARTICIPANTES

- **Modelo:** Maneja la información y la lógica de negocio.
- **Vista:** Presenta los datos al usuario.
- **Controlador:** Responde a la entrada del usuario y actualiza el Modelo y la Vista.

COLABORACIONES
El Controlador recibe las entradas del usuario, las procesa y actualiza el Modelo. La Vista observa los cambios en el Modelo y se actualiza automáticamente. (MDN Web Docs, s. f.)

CONSECUENCIAS
Beneficios:

- Fomenta la separación de responsabilidades.
- Facilita la reutilización y mantenimiento del código.
- Permite pruebas unitarias más efectivas.

Riesgos:

- Puede aumentar la complejidad en aplicaciones pequeñas.
- Requiere una correcta implementación para evitar dependencias innecesarias.

IMPLEMENTACIÓN
Los pasos para implementar el patrón MVC incluyen:

1. Crear el Modelo que maneje los datos y la lógica de negocio.
2. Diseñar la Vista para mostrar la información al usuario.
3. Implementar el Controlador para manejar la interacción entre la Vista y el Modelo.

CÓDIGO DE EJEMPLO

```
class Modelo {
   private String dato;


   public Modelo() {
      this.dato = "Hola, mundo";
   }


   public String obtenerDato() {
```

```
        return this.dato;
    }
}

class Vista {
    public void mostrarDato(String dato) {
        System.out.println("El dato es: " + dato);
    }
}

class Controlador {
    private Modelo modelo;
    private Vista vista;

    public Controlador() {
        this.modelo = new Modelo();
        this.vista = new Vista();
    }

    public void actualizarVista() {
        String dato = modelo.obtenerDato();
        vista.mostrarDato(dato);
    }
}

// Uso del patrón MVC
public class Main {
    public static void main(String[] args) {
        Controlador controlador = new Controlador();
        controlador.actualizarVista();
```

```
    }
}
```

USOS CONOCIDOS

- Frameworks web como Django, Spring MVC y Ruby on Rails.
- Aplicaciones de escritorio en Java, C# y Python.
- Juegos y simulaciones que requieren separación entre lógica y presentación.

PATRONES RELACIONADOS

- **Observer:** Permite a la Vista reaccionar a cambios en el Modelo.
- **Front Controller:** Centraliza el manejo de solicitudes en una aplicación web.

Tabla de horarios:

Gestión de tiempo.xlsx

Referencias:

- *Bridge*. (s. f.). https://refactoring.guru/es/design-patterns/bridge

Tech, M. (s. f.). *Mentores Tech: Impulsa tu carrera tecnológica con cursos, recursos y pruebas técnicas*. Mentores Tech. https://www.mentorestech.com/resource-design-pattern-bridge.php

- State. (n.d.). https://reactiveprogramming.io/blog/es/patrones-de-diseno/state
- *State*. (n.d.). https://refactoring.guru/es/design-patterns/state
- MDN Web Docs. (s. f.). Modelo-Vista-Controlador (MVC). https://developer.mozilla.org/es/docs/Glossary/MVC