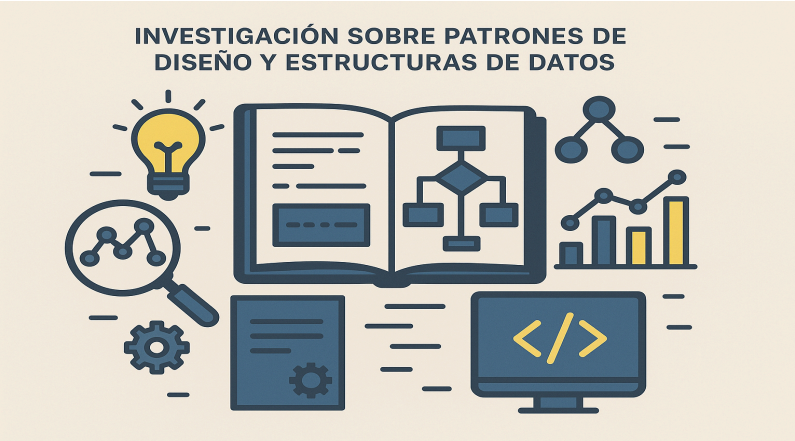
IES CASTELAR

Frameworks Aplicados a la Web



UT2-TAREA 1 Investigación sobre patrones de diseño y estructuras de datos

Nombre del alumno: Borja De La Cruz Lucio

Desarrollo de Aplicaciones Web

Módulo: Frameworks Aplicados a la Web

Fecha: 15/10/2025

1. INTRODUCCIÓN

La tarea consistió en seleccionar tres patrones de diseño de software y analizar en detalle cada uno, cumpliendo con los requisitos de definición, estructura, ejemplo de uso, implementación en *frameworks* y análisis de ventajas y limitaciones.

Los patrones seleccionados son:

- Singleton
- Observer
- Composite
- Decorator

2. DEFINICIÓN DEL PATRÓN Y EL PROBLEMA QUE RESUELVE

A) Singleton

Es un patrón creacional que asegura que una clase tenga una sola instancia y proporciona un punto de acceso global a ella.

Con respecto al problema que resuelve, podemos decir que sería la necesidad de tener un control estricto sobre el número de instancias de una clase, impidiendo la creación de múltiples objetos. Esto es crucial para gestionar recursos compartidos o globales, como un registro (loguearse), una conexión de configuración o un “thread pool”, donde múltiples instancias podrían causar inconsistencia de estado o consumo.

B) Observer

Define una dependencia de uno a muchos entre objetos. Cuando el estado de un objeto cambia, todos sus dependientes son notificados y actualizados automáticamente.

Con respecto al problema que resuelve, afronta el alto acoplamiento entre componentes. Permite que el sujeto envíe notificaciones a múltiples objetos sin necesidad de conocer sus clases concretas. Promueve el bajo acoplamiento y la coherencia de estado al garantizar que los componentes dependientes reaccionen de manera sincrónica al cambio del componente principal.

C) Decorator

Permite añadir nuevas responsabilidades o comportamientos a un objeto dinámicamente (en tiempo de ejecución) sin alterar su código base. Lo hace envolviendo el objeto original en un objeto *Decorator* que mantiene la misma interfaz.

Con respecto al problema que afronta, resuelve la limitación de la herencia estática y la "explosión de subclases". En lugar de crear subclases para cada posible combinación de funcionalidad, el Decorator permite componer funcionalidades apilando wrappers sobre el objeto base, logrando gran flexibilidad y adhiriéndose al Principio Abierto/Cerrado.

D) Composite

Organiza objetos en estructuras de árbol para representar jerarquías de parte-todo. El objetivo clave es permitir que el código cliente trate tanto a los objetos individuales (Hojas) como a las colecciones de objetos (Compuestos) de forma uniforme a través de una interfaz común.

Con respecto al problema que resuelve, afronta el desafío de manejar estructuras jerárquicas recursivas (como sistemas de archivos, menús o estructuras organizativas). Simplifica el código del cliente al eliminar la necesidad de usar lógica condicional para distinguir entre un elemento simple y un grupo de elementos.

3. DESCRIPCIÓN DE ESTRUCTURA Y COMPONENTES

3.1. DESCRIPCIÓN DE ESTRUCTURA

A) Singleton

La estructura es autorreferencial y restrictiva. La clase es responsable de crear y gestionar su propia y única instancia. Esto se logra haciendo que la creación externa sea imposible (constructor privado) y proporcionando un único punto de acceso estático para obtener el objeto.

B) Observer

La estructura es de desacoplamiento basado en interfaz y unidireccional (uno-a-muchos). El diseño se basa en dos jerarquías de clases independientes (Sujetos y Observadores) que solo se conocen a través de una interfaz común. El Sujeto mantiene una lista de referencias de la interfaz Observer, permitiéndole notificar a muchos sin saber quiénes son.

C) Decorator

La estructura es de envoltura (*wrapper*) y composición flexible. El Decorator crea una cadena de objetos que se envuelven mutuamente. Todos los elementos (el componente base y todos los decoradores) implementan la misma interfaz. Esto permite añadir responsabilidades dinámicamente mediante la composición, delegando la ejecución a la capa inferior de la envoltura.

D) Composite

La estructura es recursiva y transparente. Se construye una jerarquía en forma de árbol donde tanto los nodos simples (Hojas) como los nodos contenedores (Compuestos) implementan la misma interfaz común. Esta transparencia permite que el código cliente trate cualquier elemento del árbol de manera uniforme, aplicando operaciones a la estructura completa mediante recursividad o delegación.

3.2. COMPONENTES

A) Singleton

Componente	Rol principal
Instancia Estática Privada	Contenedor interno de la única instancia creada.
Constructor Privado	Bloquea la instanciación directa desde el exterior.
Método de Acceso Estático	Punto de entrada público para recuperar la instancia única de forma controlada.

B) Observer

Componente	Rol principal
Sujeto Concreto	Fuente de la verdad o datos. Contiene los métodos para registrar/eliminar observadores y el mecanismo para desencadenar la notificación.
Interfaz Observador	Define el contrato de respuesta que deben seguir todos los dependientes.
Observadores Concretos	Dependientes. Implementan la lógica específica de cómo reaccionar y procesar la información enviada por el sujeto.

C) Decorator

Componente	Rol principal
Componente (Interfaz)	Define el contrato común que el objeto base y sus decoradores deben cumplir.
Componente Concreto	El objeto base que proporciona la funcionalidad principal.
Decorador Abstracto	Una clase base que implementa la interfaz Componente y mantiene la referencia al objeto envuelto.
Decorador Concreto	Añade responsabilidades. Implementa la interfaz, llama al método del componente envuelto y añade la funcionalidad adicional antes o después de esa llamada.

D) Composite

Componente	Rol principal
Componente (Interfaz)	La interfaz base que garantiza la uniformidad. Declara los métodos que son comunes para Hojas y Compuestos.
Hoja (Leaf)	Objetos terminales de la jerarquía. No pueden tener hijos.
Compuesto (Composite)	Contenedores que pueden tener hijos. Implementa la operación del Componente delegando la llamada a sus propios hijos.

4. EJEMPLO DE USO

A) Singleton

El diagrama UML para Singleton es simple y se centra en las restricciones de accesibilidad:

- Estructura: Una única clase.
- Relaciones: Es autorreferencial (la clase contiene una instancia de sí misma).

Elemento Clave	Notación UML
Instancia estática	Atributo estático y privado (marcado con - o private) de la misma clase.
Constructor privado	Operación (método) privada (marcado con -)
Método de acceso	Operación estática y pública (marcado con + y subrayado o texto static)

B) Observer

Este diagrama muestra las interfaces clave y la relación de dependencia entre el Sujeto y los Observadores.

- Estructura: Dos jerarquías (Sujeto y Observador) que se unen mediante una interfaz.
- Relaciones: El SujetoConcreto tiene una relación de agregación con la interfaz Observador (representada por un rombo vacío y una línea)

Elemento Clave	**Notación UML**
Interfaz Sujeto/Observador	Etiqueta con <<interface>>
Relación Sujeto-Observador	El Sujeto mantiene una lista (agregación) de Observadores.
Notificación	El Sujeto llama al método actualizar() del Observador.

C) Decorator

El diagrama de Decorator muestra cómo se apilan los objetos sin modificar la interfaz.

- Estructura: Una jerarquía de herencia para el Decorador y una relación de composición que envuelve el Componente.
- Relaciones: El DecoradorAbstracto hereda de la interfaz Componente y a la vez contiene por composición una referencia a un Componente.

Elemento Clave	Notación UML
Herencia	Tanto el ComponenteConcreto como el DecoradorAbstracto heredan de la interfaz Componente.
Composición	El DecoradorAbstracto tiene una referencia (agregación o composición) al Componente envuelto.
Cadena de llamadas	La operación del decorador llama al método del objeto que envuelve.

D) Composite

El diagrama de Composite ilustra la recursividad y la uniformidad de las interfaces.

- Estructura: Todos los nodos (Hojas y Compuestos) heredan de la misma interfaz Componente.
- Relaciones: El Compuesto tiene una relación de agregación recursiva consigo mismo y con la Hoja (a través de la interfaz Componente).

Elemento Clave	Notación UML
Herencia	Línea con punta de flecha vacía y cerrada, apuntando hacia la interfaz Componente.
Relación Compuesto-Componente	El Compuesto almacena una colección de referencias de tipo Componente.
Uniformidad	Todos los componentes tienen la operación común (ej. calcularCosto()).

5. FRAMEWORKS QUE UTILIZAN LOS PATRONES DE DISEÑO

A) Singleton

Este patrón es implementado por contenedores de Inversión de Control (IoC) al gestionar el ciclo de vida de los servicios.

a) **Spring Framework** (Java): Utiliza el alcance **singleton** por defecto para sus *beans* (componentes), garantizando una única instancia compartida por toda la aplicación.
b) **Laravel / Symfony** (PHP): Implementan servicios únicos (como el *Container* o el gestor de **Configuración**) a través de sus contenedores de inyección de dependencias.
c) **NestJS** (Node.js): Sus servicios, módulos y controladores se gestionan con un alcance *singleton* por defecto.

B) Observer

Este patrón es la base de la programación reactiva y los sistemas de eventos.

a) **RxJS** (JavaScript/Frontend): Es una implementación directa y moderna del patrón, donde los **Observables** (Sujetos) emiten datos que son consumidos por **Suscriptores** (Observadores).
b) **Programación Reactiva (Reactor/Kotlin Flow):** Estos *frameworks* utilizan el patrón Observer para manejar flujos de datos asíncronos y eventos en aplicaciones *backend*.
c) **Frameworks MVC:** La relación entre el **Modelo** (Sujeto) y la **Vista** (Observador) en muchos frameworks de interfaz gráfica.

C) Decorator

Este patrón es clave para añadir funcionalidad a objetos o flujos de manera flexible sin usar herencia.

a) **Java I/O:** Las clases de flujo de entrada/salida como **BufferedInputStream** o **GZIPOutputStream** son decoradores que envuelven el flujo base para añadir funcionalidad (buffering, compresión).
b) **ASP.NET Core / Express.js (Middleware):** El concepto de **Middleware** en *frameworks* web actúa como un decorador. Cada capa envuelve la ejecución de la siguiente, añadiendo funciones como *logging* o autenticación a la solicitud antes de que llegue a su destino.
c) **Decoradores de Python (@):** El mecanismo de decoradores del lenguaje se utiliza para envolver funciones y clases, modificando o añadiendo comportamiento dinámicamente.

D) Composite

El Composite se usa en sistemas que representan estructuras jerárquicas recursivas.

a) **Document Object Model (DOM)** (Web): La estructura de los elementos HTML y XML es un Composite, donde los nodos (*div*, *body*) son **Compuestos** y los nodos de texto son **Hojas**.
b) **Java Swing / AWT:** Las librerías de interfaz gráfica utilizan el patrón para construir la jerarquía de *widgets*. Un JPanel (Compuesto) contiene otros componentes.
c) **Sistemas de Archivos:** Las estructuras de directorios de los sistemas operativos son Composites, donde un **Directorio** es un Compuesto que contiene **Archivos** (Hojas).

6. VENTAJAS Y DESVENTAJAS DE LOS PATRONES

A) Singleton

Aspecto	Ventajas (Escenarios Apropiados)	Limitaciones (Anti-patrón)
Ventajas	**Control de Instancia:** Garantiza que solo exista una instancia, ideal para recursos únicos como configuraciones de sistema o *loggers* simples. **Acceso Controlado:** Proporciona un punto de acceso global, pero es un punto de acceso conocido y único.	**Acoplamiento Global:** Introduce un estado global, haciendo que el código sea difícil de mantener y de **probar unitariamente**, ya que no se pueden aislar las dependencias (lo que lo convierte en un anti-patrón en arquitecturas modernas).
Desventajas	Puede usarse legítimamente en *scripts* pequeños o sistemas *legacy* sin inyección de dependencias.	**No es *Thread-Safe* (por defecto):** En entornos multihilo, una implementación descuidada puede resultar en la creación accidental de múltiples instancias. **Violación del SRP:** La clase es responsable de su lógica de negocio *y* de su propia gestión de ciclo de vida.

B) Observer

Aspecto	Ventajas (Escenarios Apropiados)	Limitaciones (Riesgos)
Ventajas	**Bajo Acoplamiento:** El Sujeto y el Observador son independientes, facilitando la extensión (puedes añadir Observadores sin modificar el Sujeto). **Coherencia de Estado:** Ideal para sistemas de interfaz (MVC) o datos reactivos (RxJS) donde muchos componentes deben reflejar el mismo estado en tiempo real.	**Problemas de Dependencia Oculta:** Los Observadores pueden depender de detalles del Sujeto que no se revelan en la interfaz, dificultando la comprensión del flujo de datos.
Desventajas	**Orden de Notificación:** No garantiza el orden en que se notifican los Observadores, lo cual es problemático si la acción de un Observador depende de que otro ya haya actuado. **Sobrecarga de Actualizaciones:** Si hay muchos Observadores o si la lógica de `update()` es pesada, la notificación puede convertirse en un cuello de botella de rendimiento.	

C) Decorator

Aspecto	**Ventajas (Escenarios Apropiados)**	**Limitaciones (Riesgos)**
Ventajas	**Flexibilidad Dinámica:** Permite añadir o quitar responsabilidades en **tiempo de ejecución** mediante la composición, sin modificar el objeto base. **Principio Abierto/Cerrado:** Extiende la funcionalidad de un componente sin modificar su código existente (abierto para extensión, cerrado para modificación).	**Complejidad de Debugging:** Una cadena larga de decoradores anidados puede dificultar el *debugging* y el seguimiento de dónde se está aplicando exactamente la funcionalidad.
Riesgos	**Problemas de Identidad:** Como un objeto está envuelto por otro, las comprobaciones de identidad o tipo (ej. `instanceof`) pueden volverse engañosas o requerir lógica adicional.	Puede resultar en un diseño **excesivamente atomizado** si cada pequeña responsabilidad es un decorador separado.

D) Composite

Aspecto	Ventajas (Escenarios Apropiados)	Limitaciones (Riesgos)
Ventajas	**Transparencia/Uniformidad:** Permite a los clientes ignorar la diferencia entre elementos simples y compuestos, simplificando enormemente el código. **Recursividad Natural:** Facilita la implementación de jerarquías complejas (ej. DOM, sistemas de archivos) y la aplicación de operaciones a todos los elementos.	**Exceso de Generalización:** Obliga a que la interfaz Componente sea muy genérica. Esto puede hacer que las **Hojas** hereden o tengan que implementar métodos irrelevantes (ej. agregar_hijo() en una hoja).
Riesgos	**Restricción de Tipos:** El sistema de tipos no puede impedir que un cliente intente añadir una Hoja a otra Hoja, ya que ambas cumplen con la interfaz Componente (se requiere verificación en tiempo de ejecución).	Puede ser una **sobrecarga de diseño** para estructuras jerárquicas estáticas o muy simples.

7. BIBLIOGRAFÍA

- Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1994). *Design patterns: Elements of reusable object-oriented software*. Addison-Wesley Professional. Recuperado de https://books.google.com/books/about/Design_Patterns.html?id=vRfxugEACAAJ
- Refactoring.Guru. (s. f.). *Design Patterns*. Recuperado el 15 de octubre de 2025, de https://refactoring.guru/design-patterns
- Spring Framework. (s. f.). *Bean Scopes* (documentación oficial). Recuperado el 15 de octubre de 2025, de https://docs.spring.io/spring-framework/reference/core/beans/factory-scopes.html
- Mozilla Developer Network (MDN). (s. f.). *Document Object Model (DOM)*. Recuperado el 15 de octubre de 2025, de https://developer.mozilla.org/en-US/docs/Web/API/Document_Object_Model