**DISEÑO DE SOFTWARE**

**PARALELO 3**

**TAREA 3- SISTEMA ENVIVOTICKETS**

**INTEGRANTES:**

**BARRIOS URETA ROBERTO CARLOS**

**MACIAS MENDOZA CHRISTIAN JAVIER**

**TAPIA LOOR PAULO MARCELO**

**PAO II**

**2024-2025**

INDICE

[**Sección A: Diagramas UML de los patrones de diseño corregidos** 3](#_Toc188049136)

[1. Observer: 3](#_Toc188049137)

[2. Decorator: 3](#_Toc188049138)

[3. Facade: 4](#_Toc188049139)

[**Sección B: Evaluación de los patrones y reflexión sobre sus beneficios y limitaciones** 4](#_Toc188049140)

[1. Observer: 4](#_Toc188049141)

[2.Decorator: 4](#_Toc188049142)

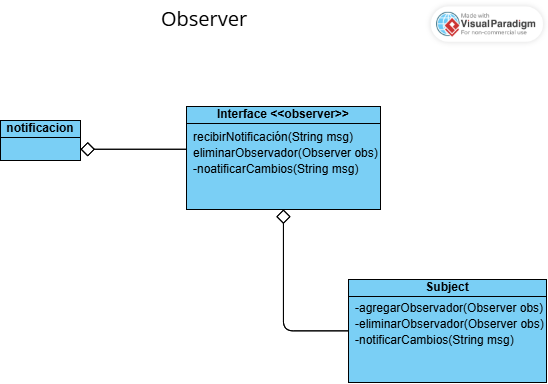
[3.Facade: 4](#_Toc188049143)

[4.Factory: 4](#_Toc188049144)

[**Sección C: Generación de Código en Java** 4](#_Toc188049145)

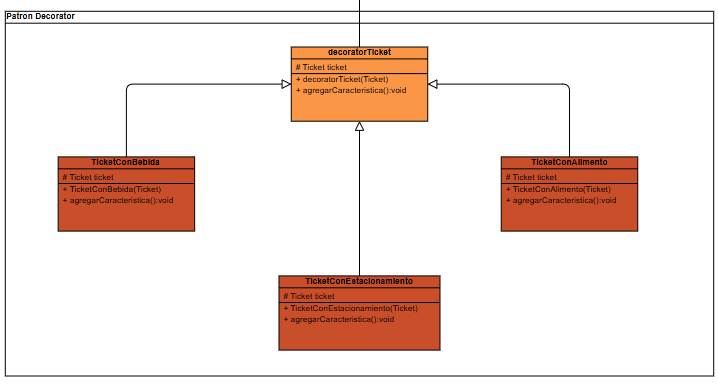
# **Sección A: Diagramas UML de los patrones de diseño corregidos**

## 1. Observer:



**LINK:** <https://online.visual-paradigm.com/share.jsp?id=333639363234322d36>

## 2. Decorator:



LINK: <https://online.visual-paradigm.com/share.jsp?id=333639383633352d34#diagram:workspace=xryfwbnc&proj=0&id=4>

## 3.Factory:

# **Sección B: Evaluación de los patrones y reflexión sobre sus beneficios y limitaciones**

## Observer:

**Evaluación y Beneficios:**

1. **Desacoplamiento:**
   * Las clases Notificacion y Subject separan la lógica de notificación de los cambios en el evento, facilitando su mantenimiento.
   * Los observadores pueden implementarse de forma independiente, lo que reduce el acoplamiento.
2. **Reutilización:**
   * La implementación de la interfaz Observer permite que distintas clases actúen como observadores sin necesidad de modificar el código del sujeto.
   * La clase Notificacion reutiliza el comportamiento de notificación del Subject.
3. **Extensibilidad:**
   * Es sencillo agregar nuevos observadores (Observer) o extender la funcionalidad del sujeto sin alterar las clases existentes.

**Limitaciones:**

1. **Coste computacional:**
   * Si el número de observadores crece significativamente, las notificaciones a cada uno de ellos pueden afectar el rendimiento.
2. **Gestión de errores:**
   * Si un observador falla al recibir notificaciones, puede impactar la propagación de los mensajes a otros observadores.
3. **Complejidad en cascada:**
   * En esta implementación, el método recibirNotificación de Subject llama a notificarCambios. Esto puede crear notificaciones redundantes o loops si no se gestiona adecuadamente.

**Reflexión General:**

El patrón Observer es una solución eficaz para sistemas que requieren notificación automática de cambios, como la reprogramación de eventos. Sin embargo, su implementación debe considerar estrategias para mitigar problemas de rendimiento y gestionar la propagación de errores entre observadores. En este caso, la estructura jerárquica y el uso de interfaces garantizan una buena modularidad y reutilización.

## 2.Decorator:

**Evaluación y Beneficios:**

1. **Flexibilidad:**
   * Los decoradores permiten agregar dinámicamente características a un boleto, como alimentos, bebidas o estacionamiento, sin modificar la clase base Ticket.
   * Los usuarios pueden combinar múltiples decoradores (e.g., un boleto con bebida y estacionamiento), proporcionando configuraciones altamente personalizables.
2. **Cumplimiento del principio abierto-cerrado:**
   * Se pueden introducir nuevas funcionalidades adicionales mediante nuevos decoradores sin modificar las clases existentes.
   * Esto facilita la extensión del sistema sin riesgos de alterar el comportamiento de las clases base.
3. **Reutilización:**
   * Los decoradores como TicketConBebida y TicketConEstacionamiento pueden aplicarse a distintos tipos de boletos, promoviendo la reutilización de código.

**Limitaciones:**

1. **Complejidad estructural:**
   * La composición de múltiples decoradores puede generar una estructura en cascada que dificulta la comprensión y mantenimiento del sistema.
2. **Rendimiento:**
   * Cada decorador añade un nivel de abstracción. Si se aplican en exceso, el sistema puede volverse menos eficiente.
3. **Duplicidad potencial:**
   * Sin un control explícito, un usuario podría agregar decoradores redundantes (e.g., dos veces TicketConBebida), lo que podría requerir validaciones adicionales.

**Reflexión General:**

El patrón Decorator es una solución efectiva para gestionar boletos con características adicionales en un sistema flexible y extensible. En este caso, la estructura modular de los decoradores permite escalar la funcionalidad del sistema sin necesidad de alterar las clases existentes. Sin embargo, es esencial implementar buenas prácticas de validación y manejo de dependencias para evitar problemas de redundancia y garantizar un diseño limpio y mantenible.

## 3.Factory:

**Evaluación y Beneficios:**

1. **Modularidad:**
   * La clase Ticket centraliza la lógica relacionada con los boletos, facilitando la reutilización y mantenimiento del código.
   * Los métodos como reservar y cancelarReserva encapsulan comportamientos específicos del ticket, manteniendo una clara separación de responsabilidades.
2. **Flexibilidad con el patrón Factory:**
   * Aunque no se muestra una implementación directa del patrón Factory en este fragmento, el uso de esta clase en combinación con una fábrica permitiría crear instancias de Ticket con configuraciones específicas (e.g., diferentes precios, secciones, o estados).
   * Esto evita la necesidad de que el cliente gestione directamente la complejidad de inicializar objetos.
3. **Extensibilidad:**
   * Es fácil extender la clase Ticket para añadir nuevas funcionalidades o tipos de boletos sin modificar el código existente.

**Limitaciones:**

1. **Falta de especialización:**
   * Actualmente, la clase Ticket parece estar diseñada para un propósito general. Si se requieren diferentes tipos de boletos (e.g., VIP, estándar, descuento), puede ser necesario extender la clase o introducir más lógica en la fábrica.
2. **Dependencia del cliente en la creación:**
   * Sin una fábrica implementada, la creación de objetos Ticket queda a cargo del cliente, lo que podría introducir errores si no se gestionan correctamente las configuraciones.

**Reflexión General:**

El patrón Factory, en combinación con esta clase, sería útil para crear instancias específicas de Ticket según las necesidades del sistema (e.g., precio, sección, estado). Esto mejora la consistencia y reduce la duplicación de código en la inicialización. Sin embargo, es crucial implementar una fábrica que gestione esta creación de manera centralizada, reduciendo el acoplamiento y asegurando que cada Ticket se cree correctamente según las políticas del sistema.

# **Sección C: Generación de Código en Java**

**LINK:** [**https://github.com/RoberB1/Tarea2--EnVivoTickets**](https://github.com/RoberB1/Tarea2--EnVivoTickets)