**Procesos del negocio para Cine Colombia**

**Camilo Andres López Contreras, Miguel Fernández de la Cruz y Juan José Márquez**

**Universidad Sergio Arboleda**

**Patrones de diseño de software**

**1 de abril de 2025**

**Adapter:**

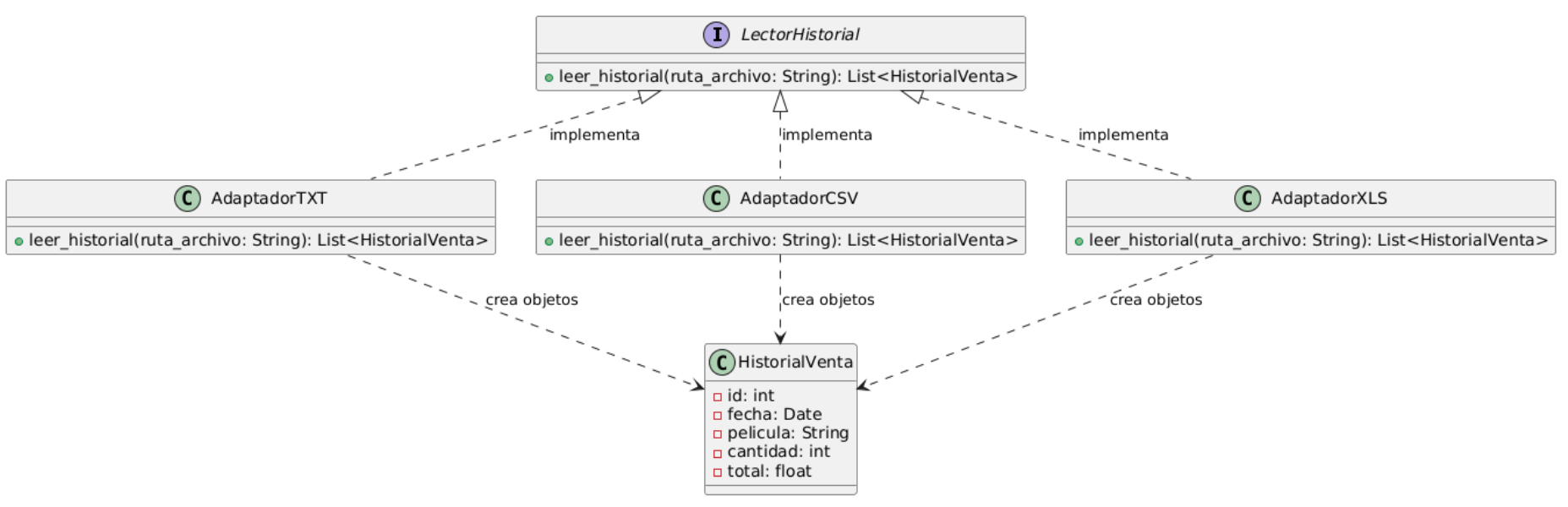
**Usar el sistema antiguo y vincularlo con el moderno.**

* **Problema:** CineColombia tiene históricos de ventas guardados en distintos formatos (por ejemplo, .txt, .xls, .csv) de sistemas anteriores. El nuevo sistema de análisis o reporte solo sabe leer objetos del tipo ***HistorialVenta***. Estos archivos no pueden cambiarse, pero necesitamos leerlos y convertirlos al nuevo formato compatible con el sistema actual.
* **Solución con adapter:** Creamos una interfaz común ***LectorHistorial*** y adaptadores concretos para cada tipo de archivo antiguo. Así, el nuevo sistema puede trabajar con todos ellos de forma unificada, sin importar el origen.

**¿Por qué este patron y no otro?**

"El patrón Adapter permite reutilizar los archivos históricos sin modificar su formato ni comprometer la arquitectura moderna del sistema. Es una solución elegante para integrar sistemas legados a través de una interfaz común."

**Diagrama de clases:**

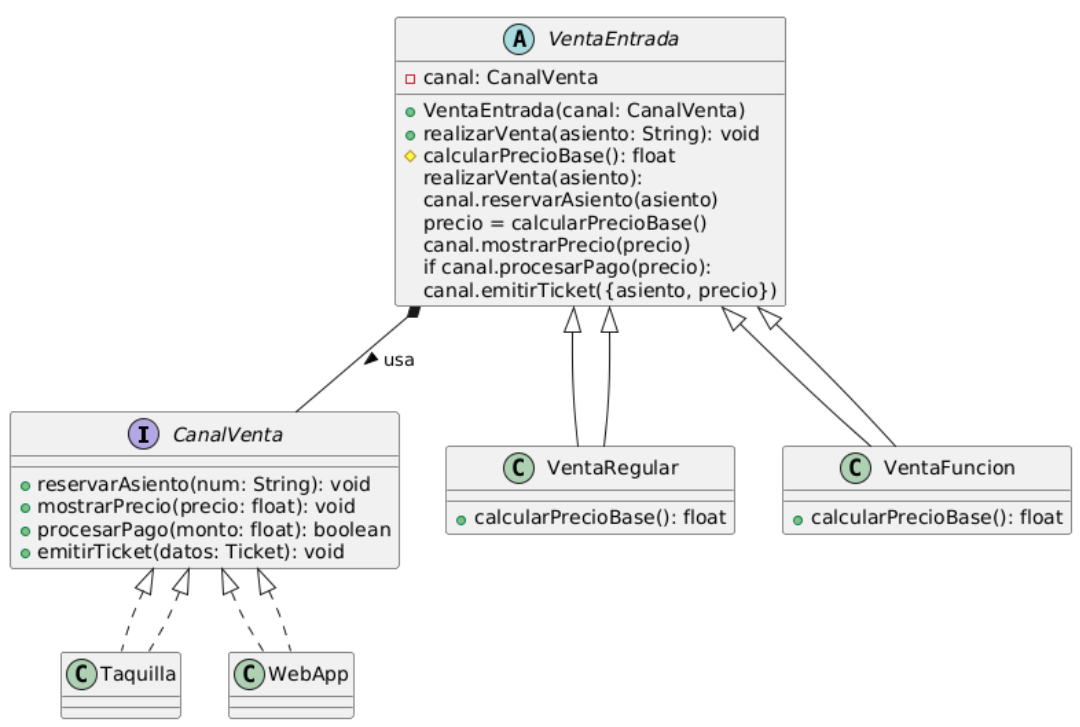


El uso del patrón Adapter permite integrar eficientemente los datos históricos de CineColombia en el sistema moderno sin comprometer la estabilidad ni la escalabilidad del sistema actual."

**Bridge:**

* **Problema.**
  + Hay **múltiples canales de venta** (taquilla física, kioscos de autoservicio, web/app).
  + Cada venta implica **selección de sala y asiento**, **cálculo de precio** (tarifa base más posibles descuentos/promociones), **procesamiento de pago** (efectivo, tarjeta, monedero digital) y **emisión del ticket** (QR o impreso).
  + Además, al finalizar la venta tenemos que **notificar** distintos subsistemas (control de acceso, CRM, estadísticas, finanzas).
  + Si acoplamos cada canal, método de pago, regla de precio y estrategia de notificación, acabaríamos con una explosión de clases.
* **Solución:**
  + **Abstracción**: VentaEntrada define el flujo genérico de una venta (selección de asiento, cálculo de precio, pago, emisión).
  + **Implementor**: CanalVenta define la interfaz de “cómo” se lleva a cabo cada paso (mostrando la UI de web, imprimiendo ticket en la taquilla, kiosco físico).
  + **Refinamientos de la abstracción**: subclases de VentaEntrada para cada **tipo de venta** (VentaRegular, VentaFuncion, VentaCombo, etc...). Sólo cambian la parte de “cálculo de precio” o reglas específicas.
  + **Implementadores concretos** de CanalVenta: Taquilla, WebApp. Se encargan de la interacción con el usuario y de emitir el ticket (QR impreso, código en pantalla, etc.).
  + De este modo puedes combinar **cualquier** tipo de venta con **cualquier** canal sin crear nuevas clases

**Diagrama de clases:**



Bridge es el más indicado en este contexto ya que tenemos dos dimensiones de variación (tipos de venta vs canales de venta) donde por separado, reusamos al máximo el código común y manteniendo la aplicación abierta a nuevas extensiones sin proliferación o extensión abrupta de clases ni acoplamientos rígidos.

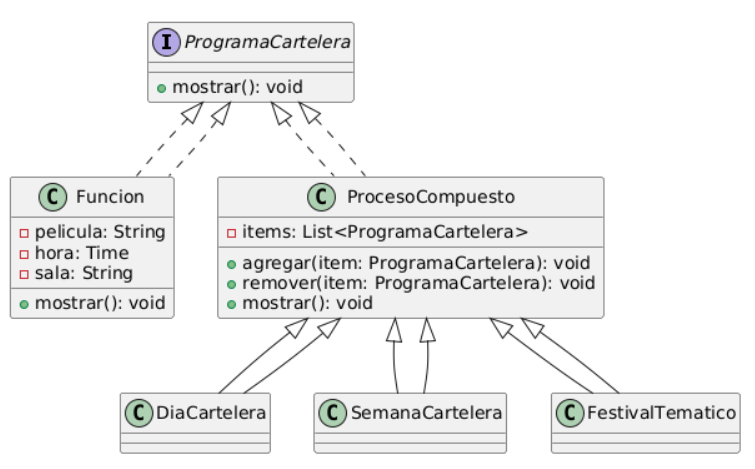
**Composite:**

**Programación de Funciones**

CineColombia debe organizar y mostrar su cartelera en distintos niveles jerárquicos: funciones individuales, programación diaria, semanal o incluso temáticas como festivales. Cada nivel puede contener uno o más elementos del mismo tipo o de tipo compuesto, por lo que se requiere un sistema que trate de manera uniforme funciones simples y grupos de funciones, facilitando así su gestión y visualización.

El patrón **Composite** resuelve el problema definiendo una interfaz común (ProgramaCartelera) para funciones individuales (Funcion) y para agrupaciones de funciones (DiaCartelera, SemanaCartelera, FestivalTematico). Estas agrupaciones heredan de una clase compuesta (ProcesoCompuesto) que permite añadir, remover y mostrar funciones o subgrupos. Así, el sistema puede recorrer y operar sobre estructuras anidadas de forma recursiva y uniforme, sin importar si se trata de una sola función o una colección jerárquica.

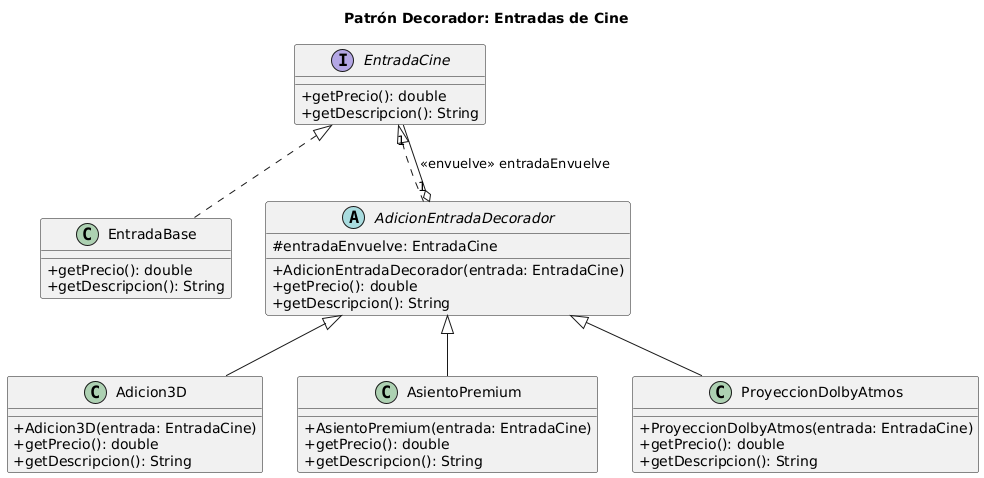
**Diagrama de clases:**



El patrón Composite permite tratar de forma uniforme objetos individuales (como una función) y composiciones de objetos (como un día, semana o festival), facilitando la gestión jerárquica de la cartelera. A diferencia de otros patrones como Decorator o Strategy, que modifican comportamiento o intercambian algoritmos, Composite se enfoca en estructuras jerárquicas, lo cual es ideal para representar una cartelera compuesta por múltiples niveles de programación.

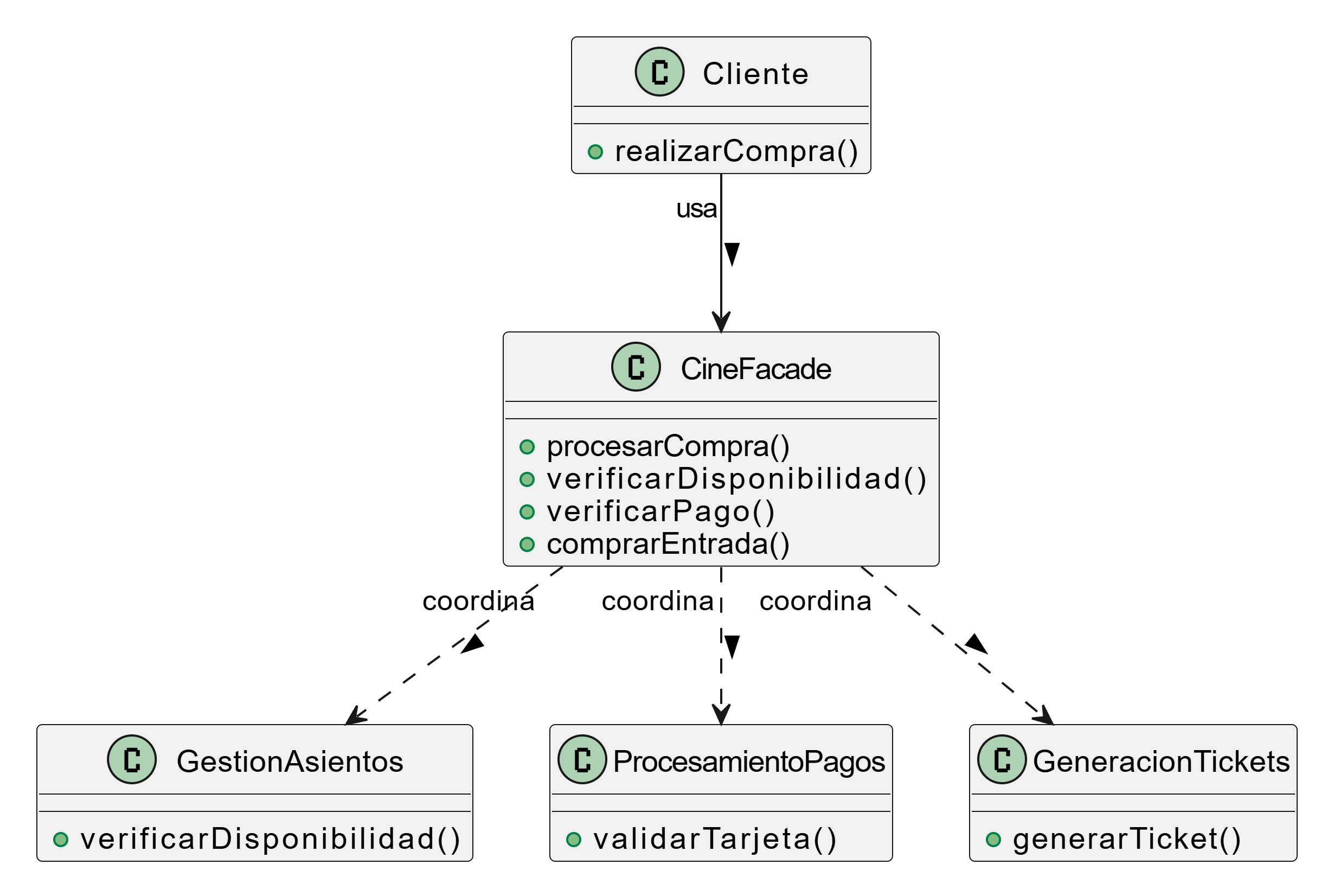
**Decorator:** En un sistema de cine, una EntradaCine puede tener múltiples características adicionales (ej. 3D, asiento premium, sonido Dolby Atmos). Usar la herencia directa para cada combinación de estas características (ej. Entrada3DAsientoPremium) llevaría a una "explosión de subclases" inmanejable y poco mantenible.

En este caso el patrón Decorator permite envolver o añadir a una entrada de cine básica diferentes adiciones a esta en tiempo de ejecución, sumando sus costos y descripciones a medida se agregan



**Facade:**

1. **Problema:** El proceso de compra de boletos en un cine implica múltiples subsistemas interconectados: verificación de disponibilidad de asientos, validación de métodos de pago, generación de tickets, actualización de inventario de confitería y gestión de promociones para clientes fidelizados. Actualmente, estos subsistemas operan de forma desarticulada, lo que genera complejidad para los clientes y el personal, ya que deben interactuar con cada componente por separado. Esto deriva en errores de sincronización (ej: asientos vendidos que aparecen disponibles), dificultad para escalar funcionalidades y una experiencia de usuario fragmentada.
2. **Solución:** El patrón Facade soluciona este problema al proporcionar una interfaz unificada que encapsula la interacción con los subsistemas. En lugar de requerir que el cliente gestione manualmente cada etapa (disponibilidad, pago, tickets, etc.), el Facade centraliza toda la lógica detrás de un único método como procesarCompra(). Esto reduce la complejidad percibida por el usuario, garantiza un flujo secuencial ordenado (ej: no se procesan pagos sin antes verificar asientos), y permite modificar subsistemas internos sin afectar al cliente. Adicionalmente, centraliza el manejo de errores y la consistencia de datos entre componentes.

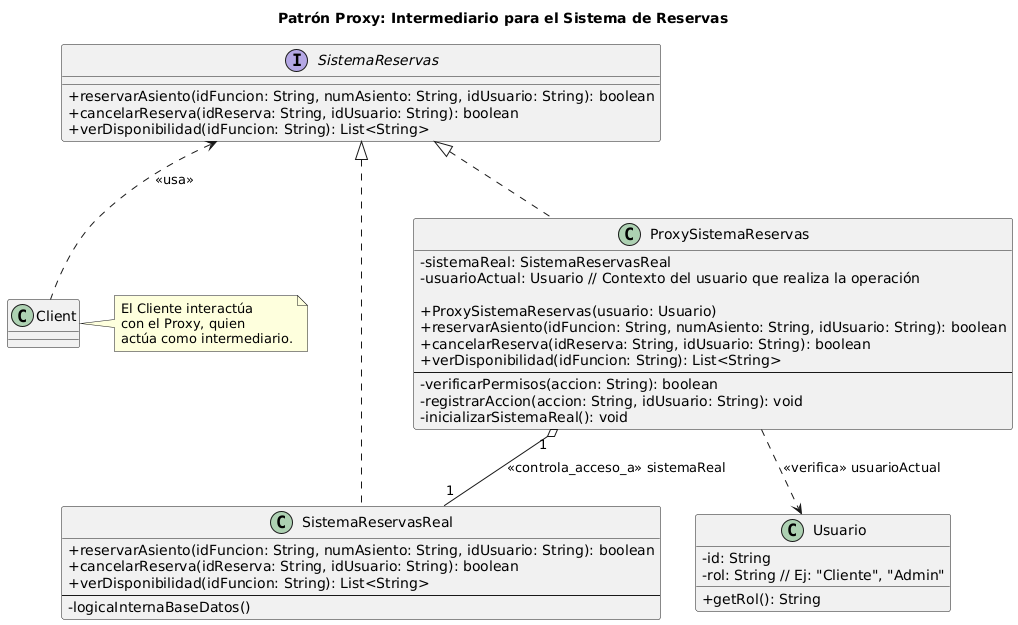


**Flyweight:**

**Proxy:** En un sistema de cine, las operaciones sensibles como **reservar o cancelar asientos** en el SistemaReservasReal requieren control. Acceder directamente a este sistema podría comprometer la seguridad o la integridad de los datos (por ejemplo, que un usuario sin autorización intente liberar un asiento).

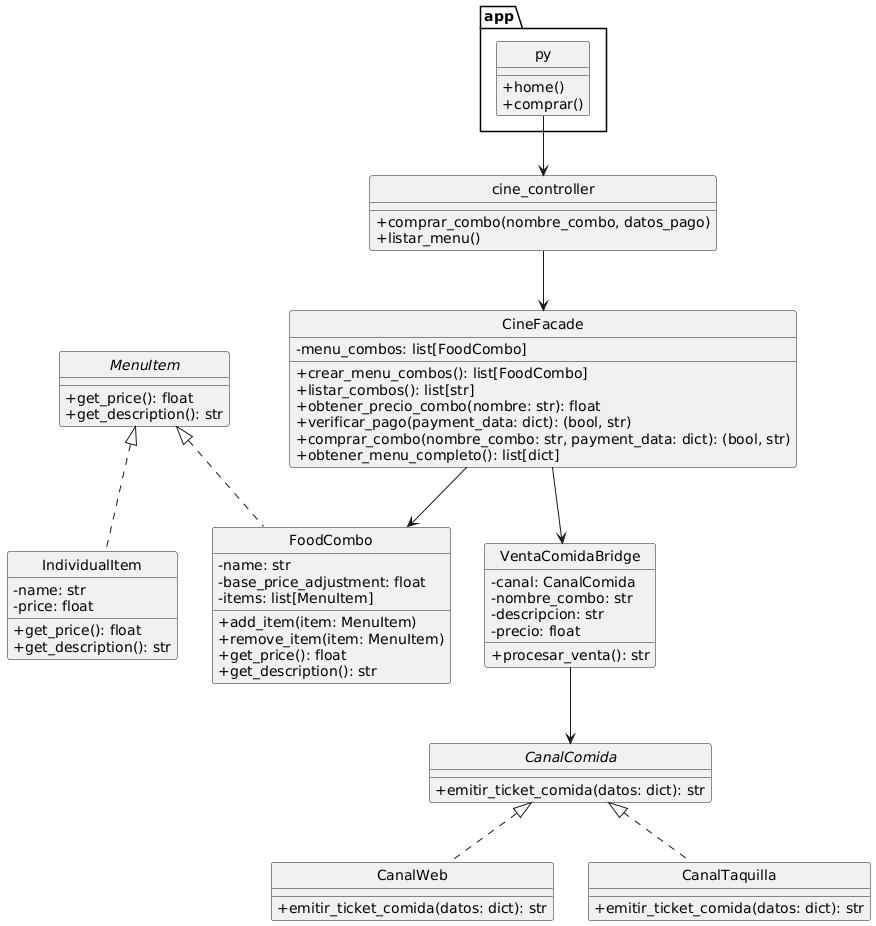
El patrón **Proxy** se usa como un **intermediario** entre el cliente y el SistemaReservasReal. Cuando un usuario intenta una acción de reserva, el Proxy intercepta la petición. Este Proxy verifica las credenciales del usuario (ej. si tiene permisos para reservar), registra la operación o gestiona recursos, y **solo si las condiciones se cumplen, delega la acción al SistemaReservasReal**.

Así, el Proxy añade una capa de seguridad y control, asegurando que el acceso al sistema de reservas real sea validado y auditado transparentemente.



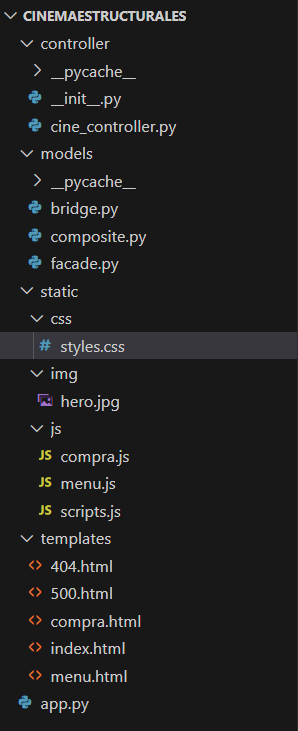
**APLICACION IMPLEMENTADA**

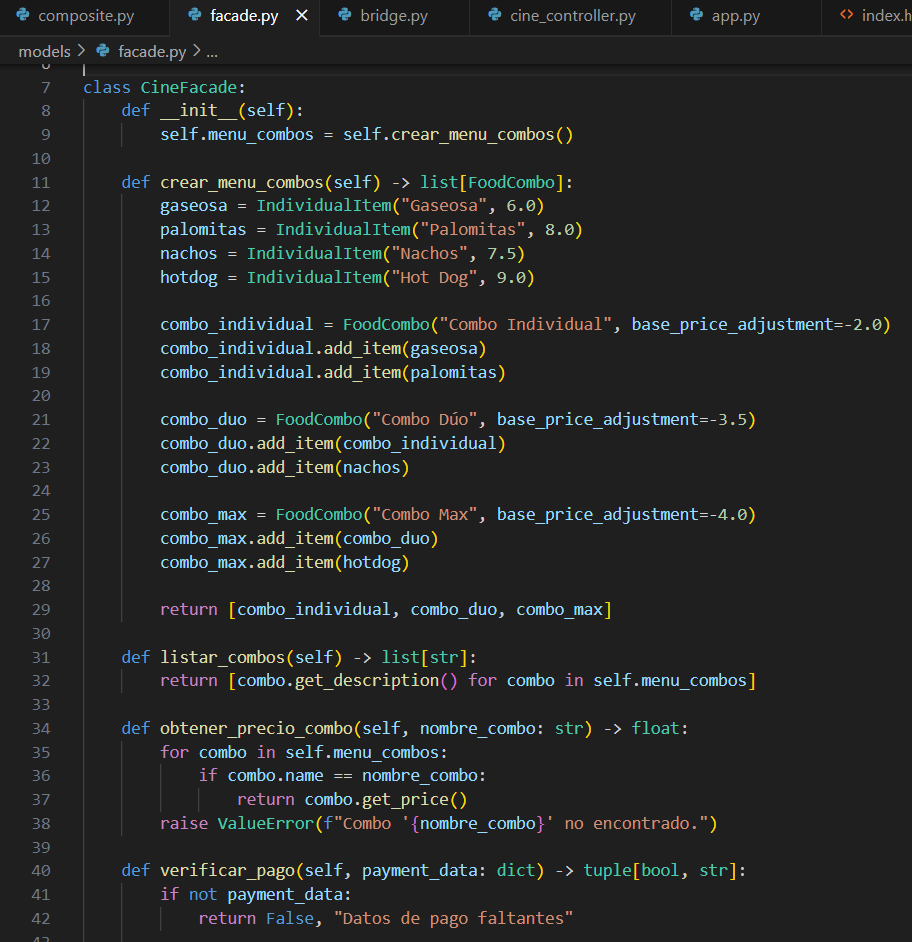
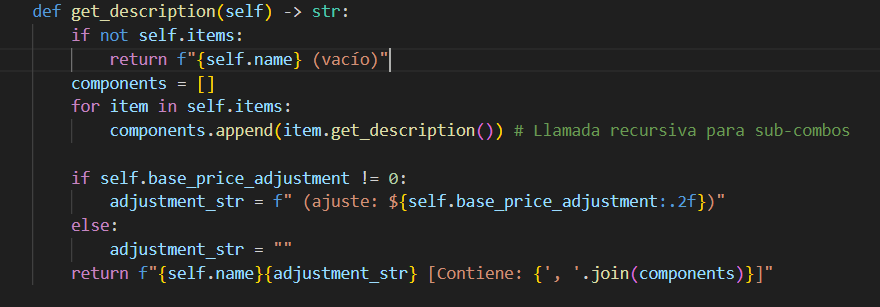
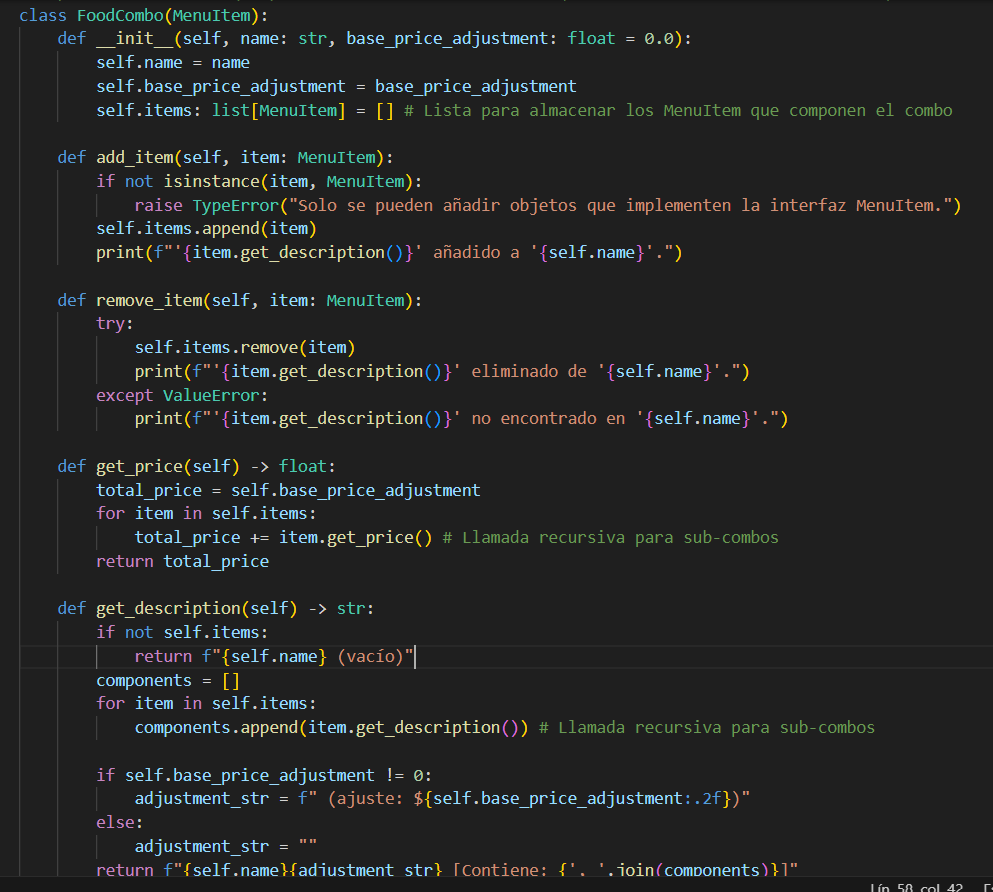
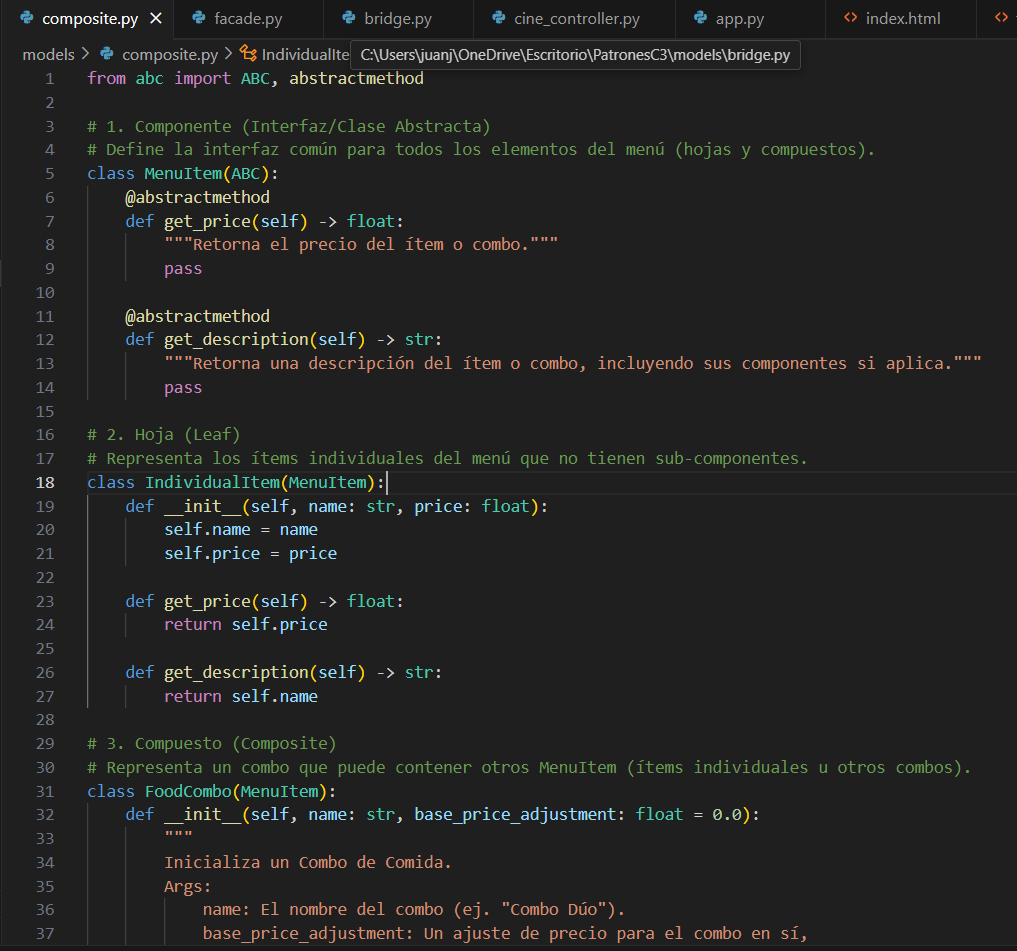
**Diseño de aplicación:**

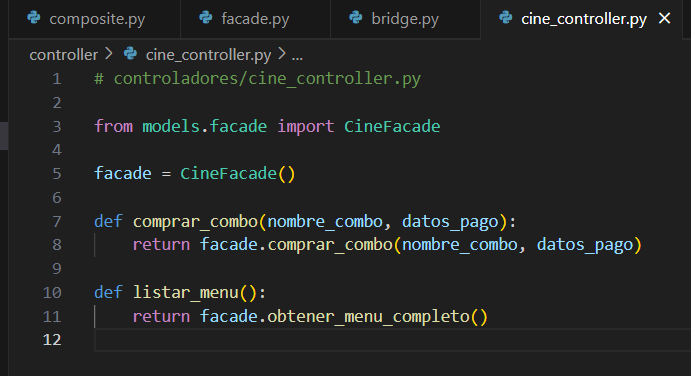
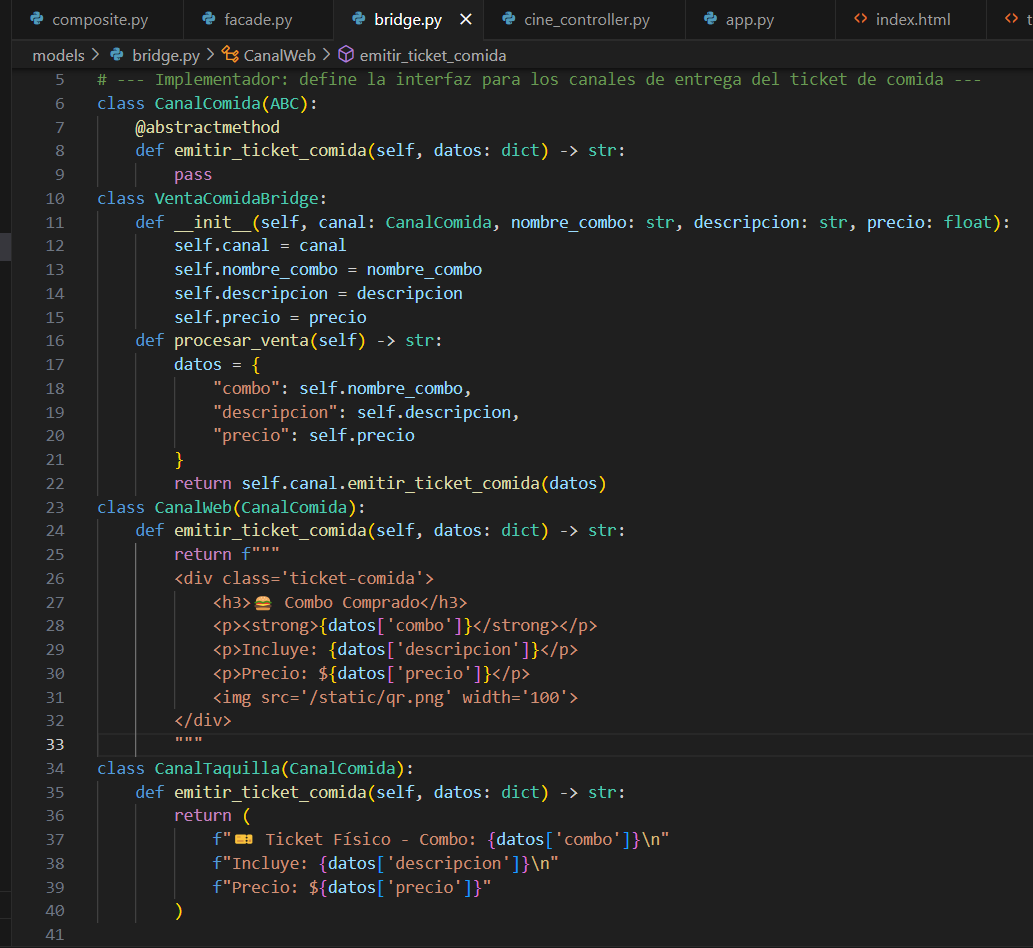
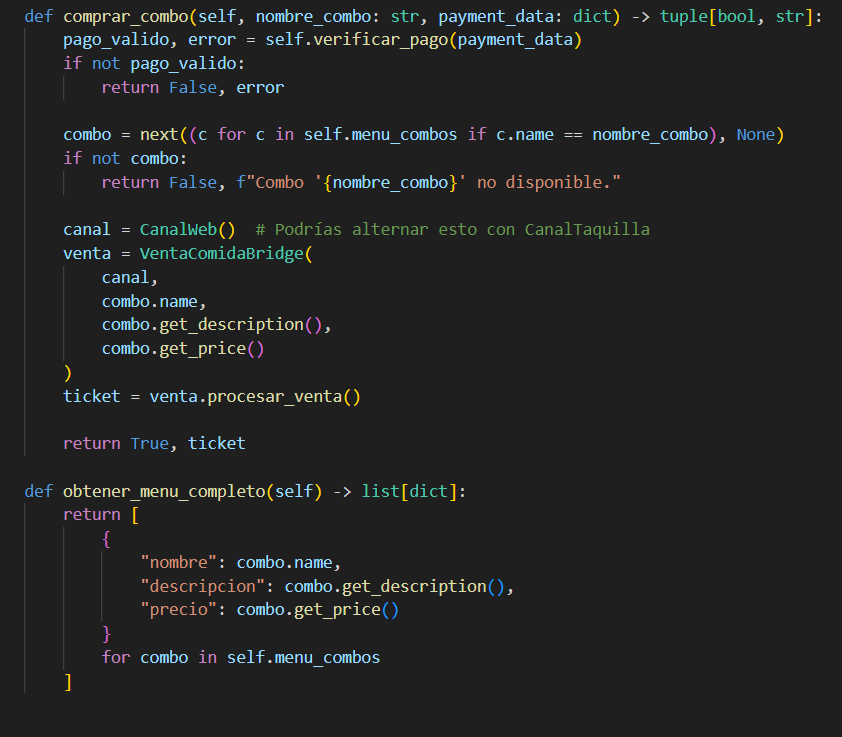
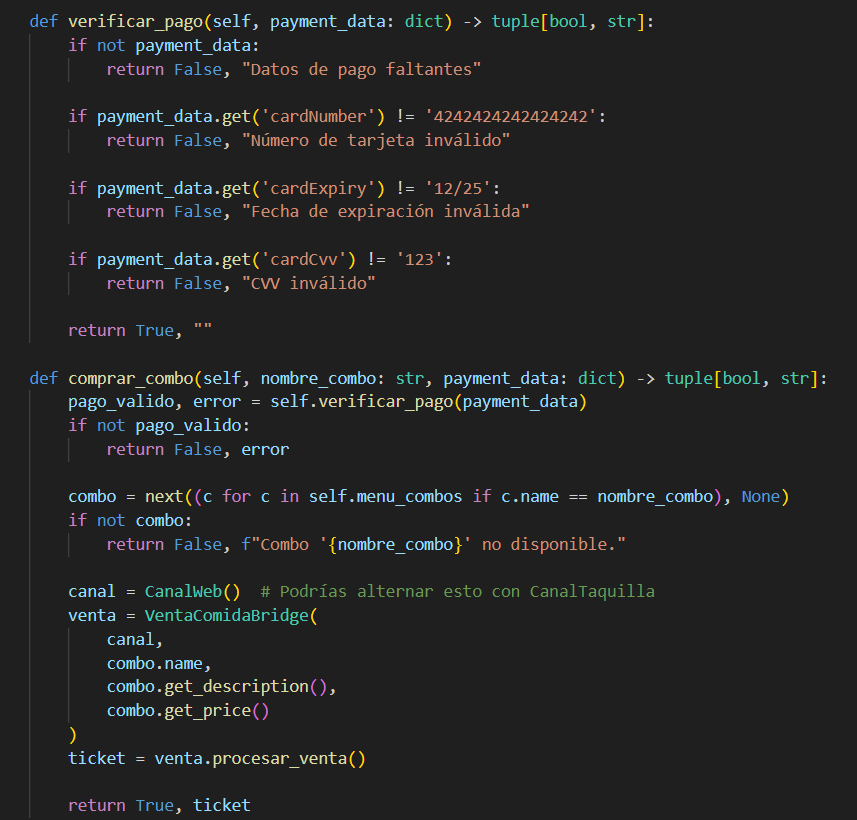


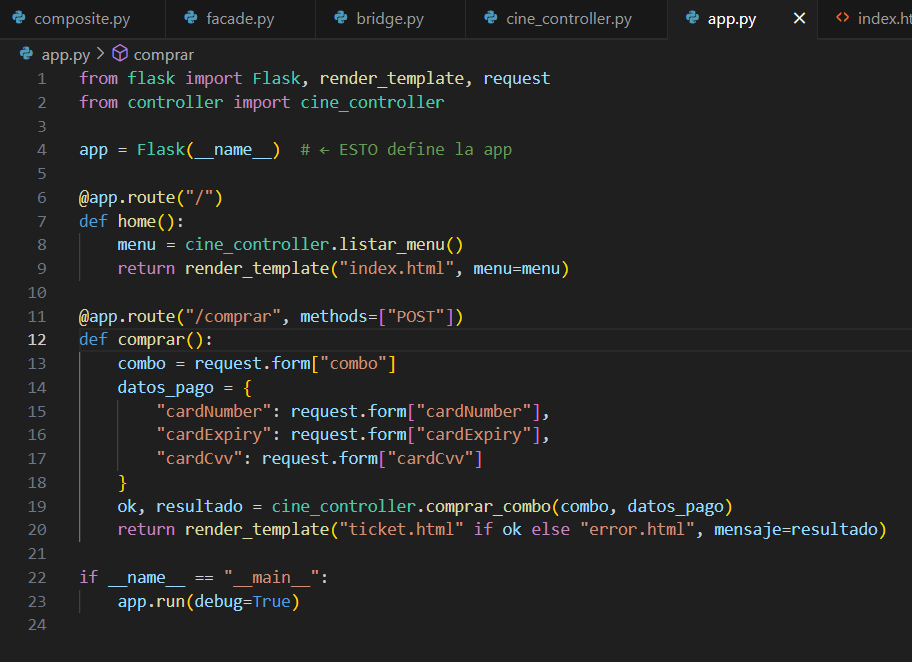
**Código de la aplicación:**

**Estructura:**









**Ejecución de la app:**

