Sección de menús:

MenuAth:

***menuAuth.ts:***

**📂 Archivo: src/app/menus/menuLogin.ts**

**📌 Descripción del archivo**

Este archivo implementa el **menú de inicio de sesión y registro de usuarios** en la aplicación.  
Se encarga de interactuar con el usuario a través de la consola utilizando la librería @inquirer/prompts.

**Funcionalidades principales:**

* Mostrar un menú principal con tres opciones:
  + 📝 **Registrarse**
  + 🔑 **Iniciar sesión**
  + ❌ **Salir**
* Permitir el registro de un nuevo usuario solicitando **nombre, correo, contraseña y rol**.
* Permitir el inicio de sesión solicitando **correo y contraseña**.
* Finalizar el programa de manera controlada al seleccionar "Salir".
* Utiliza el **UsuarioService** para gestionar la lógica de negocio relacionada con usuarios.

**⚖️ Principios SOLID aplicados**

**✅ S — Single Responsibility Principle (SRP)**

El archivo tiene una **única responsabilidad**: mostrar el menú de login y gestionar la interacción con el usuario.  
La lógica de negocio (registro, login, validaciones) se delega correctamente a UsuarioService.

**✅ O — Open/Closed Principle (OCP)**

El menú está abierto a **extensión** (se pueden agregar nuevas opciones al menú sin modificar la lógica existente).  
Ejemplo: se puede añadir una opción "Recuperar contraseña" fácilmente agregando un choice.

**✅ L — Liskov Substitution Principle (LSP)**

Se cumple indirectamente, ya que se utilizan entidades (Usuario, Rol) y servicios (UsuarioService) que pueden ser reemplazados por implementaciones compatibles sin romper la funcionalidad del menú.

**✅ I — Interface Segregation Principle (ISP)**

El archivo no depende de interfaces innecesarias. Solo interactúa con UsuarioService y Usuario, lo que mantiene el acoplamiento bajo.

**✅ D — Dependency Inversion Principle (DIP)**

El menú depende de la **abstracción de un servicio (UsuarioService)**, no de una implementación concreta.  
Esto permite cambiar la implementación de la gestión de usuarios sin modificar el menú.

**📊 Resumen de por qué se adecúa a SOLID**

El archivo respeta los principios de separación de responsabilidades y bajo acoplamiento.  
Su rol es **interfaz de interacción con el usuario**, mientras que la lógica se mantiene en capas de dominio. Esto asegura que el código sea **extensible, mantenible y testeable**.

***menuComida.ts:***

**📂 Archivo: src/app/menus/menuComida.ts**

**📌 Descripción del archivo**

Este archivo implementa el **menú de gestión y venta de comida** en la aplicación.  
Se encarga de manejar la interacción con el usuario para registrar productos, combos y realizar ventas, diferenciando las opciones según el **rol del usuario** (admin, vendedor, cliente).

**Funcionalidades principales:**

* **Administrador**:
  + ➕ Registrar productos de comida.
  + ➕ Crear combos de productos (económico, familiar, premium).
  + 📄 Listar productos y combos disponibles.
  + 📄 Listar todas las ventas registradas.
* **Vendedor**:
  + ➕ Realizar ventas de productos o combos.
  + 📄 Ver todas las ventas registradas.
* **Cliente**:
  + ➕ Comprar productos o combos.
  + 🧾 Consultar su historial de compras.

**Servicios utilizados:**

* **ComidaService** → Maneja productos y combos.
* **ComidaVentaService** → Registra y consulta ventas de comida.
* **ComidaFormatter** → Da formato a la información (listados, facturas, tablas).
* **UsuarioService** → Permite validar clientes y buscar usuarios por correo.
* **ValidationError** → Maneja errores de validación (ejemplo: nombre vacío, precio inválido).

**⚖️ Principios SOLID aplicados**

**✅ S — Single Responsibility Principle (SRP)**

El archivo se centra únicamente en **mostrar y gestionar el menú de comida**.  
La lógica de negocio (productos, combos, ventas, validaciones) se delega a sus servicios correspondientes. Esto mantiene el código modular y fácil de mantener.

**✅ O — Open/Closed Principle (OCP)**

El menú está **abierto a la extensión**: se pueden añadir nuevas opciones de productos, tipos de combos o reportes sin modificar las funcionalidades ya existentes.  
Ejemplo: agregar un nuevo tipo de combo solo requiere modificar la capa de dominio y añadir una nueva opción en el menú.

**✅ L — Liskov Substitution Principle (LSP)**

El archivo trabaja con **entidades (Usuario, Comida) y servicios abstractos (ComidaService, ComidaVentaService)** que podrían ser sustituidos por otras implementaciones compatibles sin romper el funcionamiento del menú.

**✅ I — Interface Segregation Principle (ISP)**

El menú depende únicamente de los servicios y métodos necesarios (listarProductos, crearCombo, realizarVenta, etc.).  
No se expone a métodos innecesarios de los servicios, lo que evita dependencias forzadas.

**✅ D — Dependency Inversion Principle (DIP)**

El menú depende de **abstracciones (servicios y entidades de dominio)**, no de implementaciones concretas.  
Esto permite cambiar cómo se gestionan productos o ventas (ejemplo: persistencia en BD o en memoria) sin alterar el código del menú.

**📊 Resumen de por qué se adecúa a SOLID**

Este archivo cumple con SOLID porque:

* Centraliza solo la lógica de interacción del usuario con el sistema de comida.
* Usa servicios especializados para delegar operaciones.
* Permite extensibilidad (nuevos roles, opciones de menú, tipos de combo).
* Mantiene bajo acoplamiento con la lógica de negocio, asegurando mantenibilidad y escalabilidad.

***menuFunciones:***

**📂 Archivo: src/app/menus/menuFunciones.ts**

**📌 Descripción del archivo**

Este archivo implementa el **menú de gestión de funciones de cine** en la aplicación.  
Se encarga de mostrar las opciones necesarias para administrar las funciones (horarios de películas en salas específicas) a través de la consola con la librería @inquirer/prompts.

**Funcionalidades principales:**

* **📄 Listar funciones** → Muestra todas las funciones programadas con ayuda de FuncionFormatter.
* **➕ Crear función** → Permite crear una nueva función seleccionando:
  + Película disponible (PeliculaService).
  + Sala disponible (SalaService).
  + Fecha y hora (con validación de formato YYYY-MM-DD HH:mm).
  + Precio de la entrada.
* **🗑️ Eliminar función** → Permite eliminar una función existente a partir de su ID.
* **🔙 Volver** → Cierra el menú y regresa al menú principal.

**Servicios utilizados:**

* **FuncionService** → Administra las funciones (crear, listar, eliminar).
* **PeliculaService** → Gestiona las películas disponibles para asignar a funciones.
* **SalaService** → Maneja las salas disponibles.
* **FuncionFormatter** → Da formato legible a la información de funciones.
* **date-fns (parse)** → Convierte la entrada de texto de fecha en un objeto Date.

**⚖️ Principios SOLID aplicados**

**✅ S — Single Responsibility Principle (SRP)**

El archivo se encarga únicamente de la **interacción con el usuario para gestionar funciones**.  
Toda la lógica de negocio (crear, listar, eliminar funciones) está delegada a servicios especializados.

**✅ O — Open/Closed Principle (OCP)**

El menú puede **extenderse con nuevas opciones** (ejemplo: editar función, filtrar funciones por fecha) sin modificar las funcionalidades ya existentes.

**✅ L — Liskov Substitution Principle (LSP)**

Este menú utiliza **servicios y entidades** que cumplen contratos claros (FuncionService, PeliculaService, SalaService).  
Cualquier implementación que respete la misma interfaz puede sustituir a la actual sin romper la lógica.

**✅ I — Interface Segregation Principle (ISP)**

El archivo depende solo de los métodos que realmente necesita (listarFunciones, crearFuncion, eliminarFuncion, etc.).  
No fuerza dependencias innecesarias en otras funcionalidades de los servicios.

**✅ D — Dependency Inversion Principle (DIP)**

El menú **depende de abstracciones (servicios)** en lugar de implementaciones concretas.  
Esto hace posible cambiar la forma en que se persisten las funciones (por ejemplo, de memoria a base de datos) sin modificar el código del menú.

**📊 Resumen de por qué se adecúa a SOLID**

Este archivo cumple con SOLID porque:

* Se enfoca únicamente en mostrar y gestionar el menú de funciones.
* Delegación clara de responsabilidades a servicios especializados.
* Abierto a extensión y adaptable a nuevas funcionalidades.
* Bajo acoplamiento con las capas de negocio, facilitando pruebas y mantenimiento.

***menuPeliculas:***

**📂 Archivo: src/app/menus/menuPeliculas.ts**

**📌 Descripción del archivo**

Este archivo implementa el **menú de gestión de películas** en la aplicación.  
A través de la interacción en consola (usando @inquirer/prompts), permite a los administradores listar, crear y eliminar películas.

**Funcionalidades principales:**

* **📄 Listar películas** → Muestra todas las películas registradas en formato tabla (PeliculaFormatter).
* **➕ Crear película** → Permite registrar una película ingresando:
  + 🎬 **Título**.
  + ⏱️ **Duración en minutos**.
  + 🎟️ **Clasificación** (G, PG, PG-13, R, NC-17).
  + 🌍 **Idioma** (Español, Inglés, Subtitulada, Doblada).
  + 🎭 **Género** (Acción, Comedia, Drama, Terror, Animación).
* **🗑️ Eliminar película** → Borra una película de la lista a partir de su ID.
* **🔙 Volver** → Cierra el menú y regresa al menú principal.

**Servicios y entidades utilizadas:**

* **PeliculaService** → Gestiona la lógica de negocio para crear, listar y eliminar películas.
* **PeliculaFormatter** → Da formato a los listados en forma de tabla.
* **Clasificacion, Genero, Idioma** → Tipos de la entidad Pelicula que aseguran consistencia en los atributos de cada película.

**⚖️ Principios SOLID aplicados**

**✅ S — Single Responsibility Principle (SRP)**

El archivo se encarga únicamente de **gestionar el menú de películas** y la interacción con el usuario.  
La lógica de negocio (crear, listar, eliminar películas) está separada y delegada al PeliculaService.

**✅ O — Open/Closed Principle (OCP)**

El menú puede **extenderse fácilmente** para admitir nuevas clasificaciones, géneros o idiomas sin modificar las funcionalidades existentes.  
Ejemplo: agregar un nuevo género de película implica solo añadir una opción al menú.

**✅ L — Liskov Substitution Principle (LSP)**

Este menú trabaja con entidades (Pelicula) y servicios (PeliculaService) que podrían reemplazarse por otras implementaciones (ej. conexión a base de datos en lugar de memoria) sin alterar la lógica.

**✅ I — Interface Segregation Principle (ISP)**

El archivo utiliza únicamente los métodos que necesita del servicio (crearPelicula, listarPeliculas, eliminarPelicula), sin depender de métodos innecesarios.

**✅ D — Dependency Inversion Principle (DIP)**

El menú **depende de abstracciones (servicios y entidades)** en lugar de implementaciones concretas.  
Esto permite modificar la forma en que se almacenan o gestionan las películas sin necesidad de cambiar el menú.

**📊 Resumen de por qué se adecúa a SOLID**

Este archivo cumple con SOLID porque:

* Centraliza únicamente la interacción del usuario con la gestión de películas.
* Usa servicios especializados para separar la lógica de negocio.
* Es extensible y fácil de mantener gracias al uso de tipos y servicios independientes.
* Mantiene bajo acoplamiento, lo que facilita su escalabilidad y testeo.

***menuSalas:***

**📂 Archivo: menuSalas.ts**

**📌 Descripción General**

Este archivo define el menú interactivo para la **gestión de salas de cine**.  
Permite listar, crear y eliminar salas, trabajando directamente con el SalaService (lógica de negocio) y mostrando resultados con el SalaFormatter (formateador de salida).

**🛠️ Funcionalidad Principal**

* **Listar salas:**  
  Muestra todas las salas registradas en formato de tabla.
* **Crear sala:**  
  Solicita al usuario nombre, tipo de sala (2D, 3D, VIP), filas y columnas.  
  Calcula automáticamente la **capacidad** (filas × columnas) y crea la sala.
* **Eliminar sala:**  
  Elimina una sala a partir de su ID ingresado.
* **Salir:**  
  Regresa al menú principal.

**🎯 Relación con Principios SOLID**

* **S - Single Responsibility Principle (SRP):**  
  Este módulo se limita únicamente a **interacciones del usuario para gestionar salas**, sin mezclarse con la lógica de negocio ni el almacenamiento.  
  La lógica está delegada al SalaService y la presentación al SalaFormatter.
* **O - Open/Closed Principle (OCP):**  
  El menú se puede extender fácilmente (ej. agregar opción de actualizar sala) sin modificar la lógica existente.
* **L - Liskov Substitution Principle (LSP):**  
  El SalaService y SalaFormatter cumplen contratos esperados: listar, crear y eliminar.  
  Cualquier implementación alternativa de estos servicios sería intercambiable.
* **I - Interface Segregation Principle (ISP):**  
  El archivo solo usa las interfaces y métodos que realmente necesita (listarSalas, crearSala, eliminarSala), sin depender de funcionalidades adicionales innecesarias.
* **D - Dependency Inversion Principle (DIP):**  
  El menú no crea ni gestiona datos por sí mismo; depende de **abstracciones** (SalaService, SalaFormatter).  
  Esto permite reutilización, testeo aislado y flexibilidad en futuras integraciones.

**✅ Justificación Técnica**

* El uso de prompts mantiene una experiencia interactiva en consola.
* La **capacidad de la sala** se calcula dinámicamente, reforzando la separación de responsabilidades:
  + Entrada de datos → prompts.
  + Lógica de negocio → SalaService.
  + Presentación → SalaFormatter.
* El diseño modular facilita futuras expansiones como actualizar salas, buscar por tipo o generar reportes.

***menuUser.ts:***

**📂 Archivo: menuPrincipal.ts**

**📌 Descripción General**

Este archivo implementa el **menú principal del sistema de cine**, que actúa como **punto central de navegación** para los distintos módulos:

* Películas 🎬
* Salas 🏟️
* Funciones 📅
* Ventas 🎟️
* Comida 🍿

El menú se adapta dinámicamente según el **rol del usuario** (admin, vendedor, cliente) y permite cerrar sesión o salir del sistema.

**🛠️ Funcionalidad Principal**

* **Administradores (admin)**  
  Pueden gestionar películas, salas, funciones, ventas y productos de comida.
* **Vendedores (vendedor)**  
  Pueden vender boletos y comida.
* **Clientes (cliente)**  
  Pueden comprar entradas y comida.
* **Opciones comunes a todos los roles**
  + Cerrar sesión (volver al login).
  + Salir del sistema por completo.

**🎯 Relación con Principios SOLID**

* **S - Single Responsibility Principle (SRP):**  
  El archivo se encarga únicamente de **mostrar el menú principal y redirigir a los submenús correspondientes**.  
  No contiene lógica de negocio, que está en los servicios (Service) ni lógica de presentación avanzada, que recae en los Formatter.
* **O - Open/Closed Principle (OCP):**  
  El menú es fácilmente extensible: agregar un nuevo rol o nueva opción no implica modificar toda la estructura, solo añadir nuevas elecciones en el switch.
* **L - Liskov Substitution Principle (LSP):**  
  El menú depende de **servicios y formatters genéricos**.  
  Si se reemplaza una implementación de VentaService, PeliculaService, etc., el menú seguirá funcionando sin cambios.
* **I - Interface Segregation Principle (ISP):**  
  El menú solo consume las interfaces necesarias (listar, crear, eliminar, etc.), sin obligar a depender de funcionalidades innecesarias.
* **D - Dependency Inversion Principle (DIP):**  
  La función recibe los servicios como dependencias externas (Servicios).  
  Esto evita acoplamiento y permite inyectar diferentes implementaciones (ej. mock para testing, o servicios con base de datos real).

**✅ Justificación Técnica**

* Uso de **prompts** para interacción clara en consola.
* Centralización de navegación → mantiene una arquitectura modular y clara.
* Control de flujo con retorno:
  + "logout" → redirige al login.
  + "exit" → termina el programa.
* La gestión basada en roles permite escalabilidad futura (ej. añadir un rol de "supervisor" con permisos intermedios).

***menuVentas:***

**📂 Archivo: menuVentas.ts**

**📌 Descripción General**

Este archivo implementa el **menú de ventas del sistema de cine**, adaptando las opciones disponibles según el **rol del usuario**:

* **Administrador (admin)**: puede listar todas las ventas.
* **Vendedor (vendedor)**: puede realizar ventas a clientes y listar todas las ventas.
* **Cliente (cliente)**: puede comprar entradas y consultar sus compras personales.

Además, el módulo gestiona la selección de funciones, asientos disponibles y genera facturas a través del VentaFormatter.

**🛠️ Funcionalidad Principal**

1. **Realizar ventas (vendedor/cliente):**
   * Seleccionar una función disponible.
   * Mostrar mapa de asientos (visual y con IDs).
   * Validar cantidad de entradas y disponibilidad de asientos.
   * Permitir a los vendedores registrar ventas para clientes existentes o nuevos.
   * Generar factura y mostrarla en pantalla.
2. **Ver compras personales (cliente):**
   * Listar todas las compras asociadas al usuario logueado.
3. **Listar todas las ventas (admin y vendedor):**
   * Mostrar en formato de tabla todas las ventas registradas.
4. **Volver al menú principal.**

**🎯 Relación con Principios SOLID**

* **S - Single Responsibility Principle (SRP):**  
  El archivo se centra únicamente en **gestionar el flujo de ventas en consola** (compra, listado, factura).  
  La lógica de negocio está separada en VentaService, FuncionService y UsuarioService.
* **O - Open/Closed Principle (OCP):**  
  El menú se adapta fácilmente a nuevos roles o reglas de negocio, sin necesidad de modificar toda la estructura.
* **L - Liskov Substitution Principle (LSP):**  
  Usuario puede ser admin, vendedor o cliente sin que se rompa el funcionamiento.  
  El sistema funciona igual con cualquier subtipo de usuario válido.
* **I - Interface Segregation Principle (ISP):**  
  Cada servicio (VentaService, FuncionService, UsuarioService) expone solo lo necesario, y el menú consume únicamente esos métodos relevantes.
* **D - Dependency Inversion Principle (DIP):**  
  La función recibe como parámetros los servicios (VentaService, FuncionService, UsuarioService) y el VentaFormatter.  
  Esto permite cambiar las implementaciones (mock, base de datos real, etc.) sin modificar el menú.

**✅ Justificación Técnica**

* Se utiliza **prompts** para interacción clara con el usuario en consola.
* Control de asientos mediante dos representaciones:
  + Mapa visual (💺 disponible, ❌ ocupado).
  + Mapa con IDs para selección exacta.
* La lógica contempla casos de error:
  + Cantidad inválida.
  + Asientos insuficientes.
  + Cliente no registrado.
* Generación de **factura en consola** usando VentaFormatter.

***Archivo(main):***

**📂 Archivo: main.ts**

**📌 Descripción General**

Este archivo representa el **punto de entrada principal del sistema de gestión de cine**.  
Se encarga de:

1. Inicializar los **repositorios en memoria** (para usuarios, ventas, funciones, salas y películas).
2. Crear las instancias de los **servicios de dominio** que gestionan la lógica de negocio.
3. Inicializar los **formatters**, responsables de la presentación de datos (tablas, facturas, etc.).
4. Instanciar los módulos de **comida y ventas de comida**.
5. Ejecutar un ciclo principal:
   * **Login/registro de usuario.**
   * **Mostrar el menú principal según el rol del usuario.**
   * **Gestionar logout o salida definitiva del sistema.**

En otras palabras: este archivo **coordina todo el flujo de la aplicación**.

**🛠️ Funcionalidad Principal**

* **Inicio de repositorios en memoria:**  
  Se usan implementaciones InMemory para manejar datos durante la ejecución (ideal para pruebas o prototipos).
* **Servicios creados:**
  + UsuarioService, VentaService, FuncionService, SalaService, PeliculaService, ComidaService, ComidaVentaService.
* **Formatters:**
  + PeliculaFormatter, SalaFormatter, FuncionFormatter, VentaFormatter, ComidaFormatter.
* **Flujo de ejecución:**
  + Forzar login/registro hasta que el usuario sea válido.
  + Dependiendo del rol (admin, vendedor, cliente), se abre un menú específico.
  + Permite cerrar sesión y volver al login (logout) o salir completamente del sistema (exit).

**🎯 Relación con Principios SOLID**

* **S - Single Responsibility Principle (SRP):**  
  Este archivo solo se encarga de **orquestar la ejecución principal** del sistema.  
  No contiene lógica de negocio ni de presentación, que están delegadas a servicios y formatters.
* **O - Open/Closed Principle (OCP):**  
  Es extensible: pueden añadirse nuevos servicios o módulos sin necesidad de modificar la lógica central, solo agregando más dependencias.
* **L - Liskov Substitution Principle (LSP):**  
  Se usan repositorios en memoria (InMemory) que podrían sustituirse por repositorios reales (ej. base de datos SQL) sin alterar la lógica del flujo principal.
* **I - Interface Segregation Principle (ISP):**  
  Los servicios implementan interfaces claras (ej. UsuarioRepository), por lo que este archivo consume solo lo que necesita de cada servicio.
* **D - Dependency Inversion Principle (DIP):**  
  El main no depende de implementaciones concretas (ej. UsuarioRepositoryInMemory) sino de **abstracciones pasadas como dependencias**.  
  Esto hace que pueda cambiarse la implementación (ej. usar MongoDB o PostgreSQL) sin afectar el resto de la aplicación.

**✅ Justificación Técnica**

* Uso de repositorios **InMemory** → Permite simular datos sin depender de una base de datos real, lo cual es útil para pruebas o desarrollo rápido.
* El main controla un **ciclo principal de ejecución**, garantizando que el usuario esté autenticado antes de acceder a los menús.
* Gracias a la separación en **servicios, repositorios y formatters**, el sistema es escalable y fácilmente mantenible.
* Al centralizar la orquestación en un único punto (main.ts), el sistema sigue una arquitectura **limpia y organizada**.

👉 Este archivo podría documentarse en el README global como **"🟢 Punto de entrada del sistema"**, resaltando que es el **coordinador principal** del ciclo de vida de la aplicación.

***Carpeta (Domain):***

**Comida.ts:**

**📂 Archivo: src/domain/entidades/Comida.ts**

**📌 Descripción General**

La clase Comida representa un **producto de comida o combo** dentro del sistema de gestión del cine.  
Un objeto Comida puede ser:

* **Producto individual** (ej: gaseosa, crispetas).
* **Combo** (ej: combo económico, familiar, premium), compuesto por varios productos y con un **descuento automático aplicado** sobre el precio total.

También se maneja la validación de datos, evitando crear comidas inválidas.

**🛠️ Atributos principales**

* **id: string** → Identificador único generado automáticamente (Comida-1, Comida-2, etc.).
* **nombre: string** → Nombre del producto o combo.
* **precio: number** → Precio final (calculado automáticamente si es combo).
* **productos: Comida[]** → Lista de productos incluidos (solo si es combo).
* **tipoCombo: TipoCombo** → Puede ser "economico", "familiar", "premium" o null.
* **descuento: number** → Porcentaje aplicado según tipo de combo (10%, 20%, 30%).

**📌 Constructor**

constructor(

nombre: string,

precio: number,

productos: Comida[] = [],

tipoCombo: TipoCombo = null

)

**Validaciones:**

* El **nombre no puede estar vacío**.
* Si no es combo → el precio debe ser **mayor a 0**.
* Si es combo → el precio se **calcula automáticamente** sumando los precios de los productos y aplicando el descuento según el tipo de combo.

**🎯 Métodos Principales**

* **getId(): string** → Retorna el identificador único.
* **getNombre(): string** → Retorna el nombre.
* **getPrecio(): number** → Retorna el precio final (ya con descuento si es combo).
* **getProductos(): Comida[]** → Retorna los productos incluidos en el combo.
* **getTipoCombo(): TipoCombo** → Retorna el tipo de combo (economico, familiar, premium, null).
* **getDescuento(): number** → Retorna el porcentaje de descuento aplicado.
* **esCombo(): boolean** → Devuelve true si la comida es un combo.
* **toString(): string** → Devuelve una representación amigable del producto o combo con su precio, descuento y lista de productos incluidos.

Ejemplo de salida si es combo:

🍱 Combo: Familiar (familiar) — $20000 (base $25000, desc 20%)

Incluye:

- Gaseosa ($5000)

- Crispetas ($15000)

- Perro caliente ($5000)

Ejemplo si es producto:

🥤 Producto: Gaseosa — $5000

**🎯 Relación con Principios SOLID**

* **S (SRP - Responsabilidad Única):**  
  La clase Comida solo se encarga de **modelar y validar un producto/combo de comida**.  
  La lógica de negocio adicional (ventas, inventario, etc.) se maneja en otros servicios.
* **O (OCP - Abierto/Cerrado):**  
  La clase está abierta a extensión (se pueden crear nuevos tipos de combos agregando reglas de descuento) sin modificar el código base.
* **L (LSP - Sustitución de Liskov):**  
  Un producto individual puede ser tratado igual que un combo, gracias a que ambos son instancias de Comida.
* **I (ISP - Segregación de Interfaces):**  
  Solo expone getters necesarios, sin obligar a usar métodos irrelevantes.
* **D (DIP - Inversión de Dependencias):**  
  No depende de infraestructura, solo de la clase ValidationError, manteniendo la lógica **100% de dominio**.

**✅ Justificación Técnica**

* Representa de forma clara la dualidad entre **producto individual** y **combo de productos**.
* El cálculo automático de precios en combos asegura consistencia y evita errores manuales.
* La validación inicial garantiza que no existan comidas sin nombre ni precios inválidos.
* Su independencia lo hace fácil de **testear y reutilizar** dentro del sistema.

**comidaVenta:**

**📂 Archivo: src/domain/entidades/ComidaVenta.ts**

**📌 Descripción General**

La clase ComidaVenta modela una **transacción de venta de productos o combos de comida**.  
Incluye los datos del **comprador**, la **lista de ítems adquiridos** y el **total de la compra**.

Se asegura de que toda venta tenga **al menos un producto válido** y calcula automáticamente el precio total.

**🛠️ Atributos Principales**

* **id: string** → Identificador único autogenerado (ComidaVenta-1, ComidaVenta-2, etc.).
* **compradorId?: string** → Identificador del usuario si el comprador es un Usuario registrado.
* **compradorNombre: string** → Nombre del comprador (puede ser un usuario o un cliente genérico).
* **compradorCorreo: string** → Correo electrónico del comprador.
* **items: Comida[]** → Lista de productos o combos comprados.
* **total: number** → Monto total de la venta, calculado como la suma de precios de los ítems.

**📌 Constructor**

constructor(

comprador: Usuario | { getNombre: () => string; getCorreo: () => string },

items: Comida[]

)

**Validaciones:**

* La venta debe tener **al menos un producto o combo**.
* Si el comprador es un Usuario → se guarda también su id.
* Calcula automáticamente el **total de la compra** sumando los precios de todos los ítems.

**🎯 Métodos Principales**

* **getId(): string** → Retorna el identificador único de la venta.
* **getCompradorId(): string | undefined** → Retorna el ID del comprador si es un usuario registrado.
* **getCompradorNombre(): string** → Retorna el nombre del comprador.
* **getCompradorCorreo(): string** → Retorna el correo del comprador.
* **getItems(): Comida[]** → Retorna los productos/combos adquiridos.
* **getTotal(): number** → Retorna el monto total de la venta.

**🧾 Ejemplo de Uso**

const gaseosa = new Comida("Gaseosa", 5000);

const crispetas = new Comida("Crispetas", 12000);

const cliente = {

getNombre: () => "Juan Pérez",

getCorreo: () => "juanperez@mail.com"

};

const venta = new ComidaVenta(cliente, [gaseosa, crispetas]);

console.log(venta.getId()); // ComidaVenta-1

console.log(venta.getTotal()); // 17000

console.log(venta.getCompradorNombre()); // Juan Pérez

**🎯 Relación con Principios SOLID**

* **S (SRP - Responsabilidad Única):**  
  La clase ComidaVenta se encarga únicamente de modelar una venta de comida y calcular su total.  
  No gestiona persistencia ni lógica de negocio avanzada.
* **O (OCP - Abierto/Cerrado):**  
  Permite extender la lógica de venta (ej. cupones de descuento, puntos de fidelidad) sin modificar su base.
* **L (LSP - Sustitución de Liskov):**  
  El comprador puede ser tanto un Usuario como un objeto que implemente getNombre() y getCorreo(), manteniendo polimorfismo.
* **I (ISP - Segregación de Interfaces):**  
  No obliga a usar más métodos de comprador que los estrictamente necesarios (getNombre, getCorreo).
* **D (DIP - Inversión de Dependencias):**  
  No depende de una implementación concreta de comprador, solo de que tenga los métodos requeridos.

**✅ Justificación Técnica**

* Separa la lógica de **productos (Comida)** de la de **ventas (ComidaVenta)**.
* Flexibilidad al permitir que el comprador sea un **usuario registrado** o un **cliente externo**.
* Cálculo automático del total reduce errores manuales.
* Clase independiente y fácil de testear.

***Funcion.ts:***

**📂 Archivo: src/domain/enditades/Funcion.ts**

**📌 Descripción General**

La clase Funcion modela una **proyección de película en una sala determinada**.  
Cada función tiene su propio **mapa de asientos**, el cual es independiente de la sala, de modo que los estados de ocupación se gestionan de forma particular para cada función.

**🛠️ Atributos Principales**

* **id: string** → Identificador único de la función.
* **pelicula: Pelicula** → Película que se proyectará.
* **sala: Sala** → Sala en la que se realiza la función.
* **fecha: Date** → Fecha y hora de la proyección.
* **precio: number** → Precio de entrada para esta función.
* **seatIds: string[][]** → Matriz con los IDs de los asientos (ej: A1, A2, …).
* **ocupados: Set<string>** → Conjunto de asientos ocupados.

**📌 Constructor**

constructor(id: string, pelicula: Pelicula, sala: Sala, fecha: Date, precio: number)

**Validaciones:**

* La película y la sala son obligatorias.
* La fecha debe ser válida.
* El precio debe ser mayor que **0**.

Además, inicializa el **mapa de asientos** según la sala.

**🎯 Métodos Principales**

**Gestión de asientos**

* **getAsientosDisponibles(): number**  
  Retorna el número de asientos libres.
* **getMapaVisual(): string**  
  Retorna una representación visual en texto:
  + 💺 disponible
  + ❌ ocupado
* **getMapaConIds(): string**  
  Retorna una representación con **ID + estado**.  
  Ejemplo: A1(💺) A2(❌)
* **listarAsientosDisponibles(): string[]**  
  Lista en plano todos los asientos libres.
* **getAsientoPorId(id: string)**  
  Retorna información de un asiento por ID.
* **ocuparAsiento(id: string): boolean**  
  Marca un asiento como ocupado. Retorna true si se ocupó correctamente, false si ya estaba ocupado o no existe.
* **restarAsientos(cantidad: number): void**  
  Método de compatibilidad para ocupar automáticamente n primeros asientos disponibles.

**Getters generales**

* **getId(): string**
* **getPelicula(): Pelicula**
* **getSala(): Sala**
* **getFecha(): Date**
* **getPrecio(): number**
* **getAsientosVendidos(): number** → Retorna cuántos asientos están ocupados.

**Representación textual**

* **toString(): string**  
  Devuelve información de la función en formato legible:
* Titanic | Sala: VIP 1 | 12/10/2025 20:30 | $25 | Disp: 50

**🧾 Ejemplo de Uso**

const sala = new Sala("Sala 1", "2D", 30, 5, 6);

const pelicula = new Pelicula("Inception", "Sci-Fi", 148);

const funcion = new Funcion("F1", pelicula, sala, new Date("2025-10-15T20:00:00"), 20000);

console.log(funcion.getMapaVisual());

console.log(funcion.getAsientosDisponibles()); // 30

funcion.ocuparAsiento("A1");

funcion.ocuparAsiento("B2");

console.log(funcion.getMapaConIds());

console.log(funcion.getAsientosDisponibles()); // 28

**🎯 Relación con Principios SOLID**

* **S (SRP):**  
  La clase gestiona exclusivamente los datos y lógica de una **función de cine**.
* **O (OCP):**  
  Puede extenderse para incluir descuentos por horario, promociones, etc., sin modificar su estructura base.
* **L (LSP):**  
  Se puede sustituir en cualquier contexto donde se use una proyección sin romper la lógica.
* **I (ISP):**  
  Ofrece solo los métodos necesarios para manejar la función y asientos.
* **D (DIP):**  
  Depende de abstracciones como Pelicula y Sala, no de implementaciones de bajo nivel.

**✅ Justificación Técnica**

* El **estado de ocupación** se gestiona por función, no por sala → lo que refleja la realidad.
* Separa responsabilidades:
  + Sala define estructura (filas, columnas).
  + Funcion administra disponibilidad dinámica.
* Permite renderizar **planos visuales de asientos** de forma clara en consola.
* Implementa validaciones robustas para garantizar consistencia en los datos.

***Peliculas.ts:***

**📄 Documentación Pelicula.ts**

**📌 Descripción del archivo**

El archivo **Pelicula.ts** define la entidad **Pelicula** dentro del dominio del sistema de cine.  
Esta clase modela una película con atributos fundamentales como:

* id: Identificador único.
* titulo: Nombre de la película.
* duracion: Duración en minutos.
* clasificacion: Restricción de edad (G, PG, PG-13, etc.).
* idioma: Idioma de la función (Español, Inglés, Subtitulada, Doblada).
* genero: Género de la película (Acción, Comedia, etc.).
* disponible: Estado booleano para indicar si puede ser programada en cartelera.

Además, incluye validaciones en su **constructor** para garantizar integridad (no permite títulos vacíos ni duraciones inválidas).

La clase proporciona **getters** para acceder de forma controlada a sus atributos y un **setter restringido** únicamente para modificar el campo disponible.

**✅ Cumplimiento de Principios SOLID**

**🔹 S - Single Responsibility Principle (SRP)**

✔ Cumple.  
La clase **Pelicula** tiene **una única responsabilidad**: representar el modelo de una película en el dominio.  
No gestiona lógica de negocio ajena (como funciones, ventas o reservas), solamente encapsula atributos y validaciones de la película.

**🔹 O - Open/Closed Principle (OCP)**

✔ Cumple parcialmente.  
La clase está **abierta a extensión** porque se puede extender con subclases (ejemplo: Pelicula3D, PeliculaEstreno) o añadir nuevos comportamientos sin alterar la clase original.  
No obstante, al usar **tipos union** (Clasificacion, Idioma, Genero) se requiere modificar el archivo para agregar nuevas opciones. Esto puede romper el OCP si constantemente se editan.

**🔹 L - Liskov Substitution Principle (LSP)**

✔ Cumple.  
Cualquier posible subclase de Pelicula podría sustituirla sin romper el sistema, ya que la clase está diseñada con getters/setters claros y sin comportamiento dependiente de implementaciones internas.

**🔹 I - Interface Segregation Principle (ISP)**

✔ Aplica indirectamente.  
Aunque no se define una interfaz aquí, la clase está enfocada en un contrato simple: atributos de película y métodos de acceso.  
No obliga a implementar métodos innecesarios, lo que está alineado con ISP.

**🔹 D - Dependency Inversion Principle (DIP)**

✔ Cumple.  
La clase depende de una **abstracción** (ValidationError) en vez de implementar lógica de validación ad-hoc dentro de ella.  
Esto facilita mantener separada la validación y aplicar reutilización.

**🛠️ Conclusión técnica**

La clase **Pelicula**:

* Está bien estructurada y **orientada al dominio**.
* Cumple con **SRP, LSP, DIP** de manera clara.
* Puede mejorar su adherencia al **OCP** evitando enums rígidos y usando configuraciones dinámicas o catálogos externos.
* Es una entidad sólida y fácil de extender o mantener.

***Sala.ts:***

**📄 Documentación Sala.ts**

**📌 Descripción del archivo**

El archivo **Sala.ts** define la entidad **Sala** dentro del dominio del sistema de cine.  
Esta clase modela una sala de cine con atributos y métodos esenciales para la gestión de funciones:

* id: Identificador único de la sala.
* nombre: Nombre de la sala.
* tipo: Tipo de sala (2D, 3D, VIP).
* capacidad: Número total de asientos.
* filas y columnas: Distribución de asientos en forma de matriz.

Además, incluye validaciones en el **constructor** para garantizar que la capacidad coincida con el producto de filas x columnas y que se cumplan restricciones básicas de tipo y existencia de nombre.

La clase también incluye un método **generarMapaIds** que construye la matriz de IDs de asientos (ejemplo: A1, A2, B1), lo cual es fundamental para gestionar la disponibilidad de asientos en cada función.

**✅ Cumplimiento de Principios SOLID**

**🔹 S - Single Responsibility Principle (SRP)**

✔ Cumple.  
La clase **Sala** tiene una única responsabilidad: representar y gestionar la estructura de la sala.  
No gestiona funciones, ventas ni otros procesos; solamente encapsula los atributos y métodos relacionados con la sala y su distribución de asientos.

**🔹 O - Open/Closed Principle (OCP)**

✔ Cumple.  
La clase puede ser extendida para crear subtipos de salas (por ejemplo, SalaPremium) sin modificar la clase base.  
El constructor permite la extensión futura mediante parámetros opcionales para filas y columnas.

**🔹 L - Liskov Substitution Principle (LSP)**

✔ Cumple.  
Cualquier subclase que herede de Sala puede sustituirla sin romper la funcionalidad existente, ya que los métodos getFilas, getColumnas, generarMapaIds y toString permanecen consistentes.

**🔹 I - Interface Segregation Principle (ISP)**

✔ Cumple.  
La clase define un **contrato limitado**: atributos de sala y método para generar IDs de asientos.  
No obliga a implementar métodos que no sean necesarios, alineándose con ISP.

**🔹 D - Dependency Inversion Principle (DIP)**

✔ Cumple parcialmente.  
La clase depende de ValidationError para manejar errores de manera abstracta, evitando dependencias de lógica de validación ad-hoc.  
No hay dependencia de capas concretas de infraestructura.

**🛠️ Conclusión técnica**

La clase **Sala**:

* Está bien encapsulada y orientada al dominio de cine.
* Cumple con **SRP, LSP, ISP y DIP** claramente.
* Permite extensiones futuras respetando **OCP**, aunque las reglas de filas/columnas están rígidamente definidas en el constructor.
* Proporciona herramientas fundamentales para gestionar funciones y reservas de manera consistente.

***Usuario.ts:***

**📄 Documentación Usuario.ts**

**📌 Descripción del archivo**

El archivo **Usuario.ts** define la entidad **Usuario** dentro del dominio del sistema de cine.  
Esta clase modela un usuario registrado en el sistema, incluyendo clientes, administradores y vendedores.

Los atributos principales son:

* id: Identificador único del usuario.
* nombre: Nombre completo del usuario.
* correo: Correo electrónico, usado como identificador de login.
* password: Contraseña del usuario (mínimo 4 caracteres).
* rol: Rol dentro del sistema (cliente, admin o vendedor).

Incluye validaciones en el **constructor** para asegurar integridad de datos, como correo válido, contraseña mínima y rol permitido.  
También ofrece métodos para obtener información del usuario (getId, getNombre, getCorreo, getRol) y para validar la contraseña (validarPassword).

El método toString() devuelve una representación legible del usuario para logs o visualización en consola.

**✅ Cumplimiento de Principios SOLID**

**🔹 S - Single Responsibility Principle (SRP)**

✔ Cumple.  
La clase **Usuario** tiene una sola responsabilidad: representar un usuario y manejar la validación básica de sus datos.  
No se encarga de autenticación completa ni de gestión de ventas, manteniéndose enfocada en la entidad.

**🔹 O - Open/Closed Principle (OCP)**

✔ Cumple.  
Se pueden extender subclases de Usuario (por ejemplo Cliente, Administrador) sin modificar la clase base.  
El comportamiento básico permanece estable y extensible.

**🔹 L - Liskov Substitution Principle (LSP)**

✔ Cumple.  
Cualquier subclase que herede de Usuario puede reemplazarlo sin afectar la lógica de validación o uso de getters.

**🔹 I - Interface Segregation Principle (ISP)**

✔ Cumple.  
La clase define un conjunto mínimo de métodos necesarios (getId, getNombre, getCorreo, getRol, validarPassword).  
No obliga a implementar métodos innecesarios, cumpliendo con ISP.

**🔹 D - Dependency Inversion Principle (DIP)**

✔ Cumple parcialmente.  
La clase depende únicamente de ValidationError para manejo de errores de validación.  
No depende de capas concretas de servicios ni de infraestructura, manteniendo bajo acoplamiento.

**🛠️ Conclusión técnica**

La clase **Usuario**:

* Está bien encapsulada, enfocada en la gestión de datos de usuario.
* Cumple con **SRP, LSP, ISP y DIP** claramente.
* Permite extensiones respetando **OCP**, por ejemplo para roles con funcionalidades específicas.
* Es fundamental para la autenticación, gestión de roles y asociación de ventas dentro del sistema.

***Venta.ts:***

**📄 Documentación Venta.ts**

**📌 Descripción del archivo**

El archivo **Venta.ts** define la entidad **Venta** dentro del dominio del sistema de cine.  
Esta clase representa una transacción de venta de entradas asociada a una **función**, a un **usuario** y a los **asientos vendidos**.

Los atributos principales son:

* id: Identificador único de la venta.
* funcion: La función asociada a la venta (clase Funcion).
* cantidad: Número de entradas vendidas.
* precio: Precio unitario de la entrada.
* total: Total de la venta (precio \* cantidad).
* fecha: Fecha y hora en que se realiza la venta.
* asientosIds: Lista de los IDs de los asientos vendidos.
* usuarioId, usuarioNombre, usuarioCorreo: Información del usuario comprador.

Incluye **validaciones en el constructor** para asegurar que la función existe y la cantidad de entradas es mayor a 0.  
El método toString() devuelve una representación legible de la venta, incluyendo los asientos y el total.

**✅ Cumplimiento de Principios SOLID**

**🔹 S - Single Responsibility Principle (SRP)**

✔ Cumple.  
La clase **Venta** tiene una única responsabilidad: representar la venta de entradas y almacenar toda la información relevante de la transacción.  
No se encarga de procesar pagos ni gestionar usuarios o funciones fuera de la relación directa con la venta.

**🔹 O - Open/Closed Principle (OCP)**

✔ Cumple.  
La clase está abierta a extensión mediante herencia o composición (por ejemplo, ventas con descuentos, ventas de combos) sin modificar el código base.

**🔹 L - Liskov Substitution Principle (LSP)**

✔ Cumple.  
Cualquier subclase de Venta puede reemplazar la clase base sin romper la lógica de la venta, los getters o la representación textual.

**🔹 I - Interface Segregation Principle (ISP)**

✔ Cumple.  
Define únicamente métodos necesarios para acceder a información de la venta (getId, getCantidad, getTotal, getAsientosIds, getNombre, getCorreo) y no impone métodos innecesarios.

**🔹 D - Dependency Inversion Principle (DIP)**

✔ Cumple parcialmente.  
La clase depende de ValidationError para la validación de entradas y de la clase Funcion para asociar la venta, pero no depende de servicios o repositorios concretos, manteniendo bajo acoplamiento.

**🛠️ Conclusión técnica**

La clase **Venta**:

* Está bien encapsulada y orientada a representar la transacción.
* Cumple **SRP, OCP, LSP e ISP**, asegurando mantenibilidad y extensibilidad.
* Permite asociar un usuario y los asientos vendidos de manera clara y consistente.
* Es fundamental para el registro de ventas y generación de facturas en el sistema.

***Tenemos la carpeta (errors):***

**📄 Documentación ValidationError.ts**

**📌 Descripción del archivo**

El archivo **ValidationError.ts** define una clase de error personalizada llamada **ValidationError**, que extiende de la clase nativa **Error** de JavaScript/TypeScript.

Esta clase se utiliza para **capturar y manejar errores de validación** dentro del dominio del sistema (por ejemplo, al crear películas, salas, funciones o ventas con datos inválidos).

* Constructor: recibe un mensaje de error message que describe la validación que falló.
* Propiedad name: se establece como "ValidationError" para identificar el tipo de error de manera consistente.

**✅ Cumplimiento de Principios SOLID**

**🔹 S - Single Responsibility Principle (SRP)**

✔ Cumple.  
Tiene una única responsabilidad: **representar errores de validación**, sin involucrarse en la lógica de negocio o persistencia.

**🔹 O - Open/Closed Principle (OCP)**

✔ Cumple.  
Se puede extender para crear otros tipos de errores de validación más específicos sin modificar la clase base.

**🔹 L - Liskov Substitution Principle (LSP)**

✔ Cumple.  
ValidationError puede ser utilizada donde se espera un objeto de tipo Error sin romper la funcionalidad existente.

**🔹 I - Interface Segregation Principle (ISP)**

✔ Cumple.  
No obliga a los clientes a implementar métodos que no necesitan, ya que hereda solo de Error y añade un name específico.

**🔹 D - Dependency Inversion Principle (DIP)**

✔ Cumple parcialmente.  
No depende de implementaciones concretas externas; solo depende de la clase base Error, lo cual mantiene bajo acoplamiento.

**🛠️ Conclusión técnica**

La clase **ValidationError**:

* Centraliza el manejo de errores de validación en el dominio.
* Facilita la captura y diferenciación de errores en toda la aplicación.
* Cumple los principios SOLID, asegurando claridad, mantenibilidad y extensibilidad.

***Sección de interface:***

**📄 Documentación: Interfaces de Repositorios**

**📌 Descripción general**

Estos archivos definen las interfaces de los **repositorios del dominio**, estableciendo los métodos que cualquier implementación concreta (por ejemplo, in-memory o base de datos) debe cumplir para manejar entidades como **Funciones, Películas, Salas, Usuarios y Ventas**.

Cada interfaz **separa la lógica de almacenamiento** de la lógica de negocio, siguiendo el patrón **Repository**, lo que facilita el cumplimiento de los principios SOLID y la testabilidad del sistema.

**1️⃣ IFuncionRepository.ts**

**Propósito:**

Define los métodos para gestionar **entidades Funcion**:

* guardar(funcion: Funcion): almacena una función.
* obtenerPorId(id: string): retorna una función por su ID.
* obtenerTodas(): retorna todas las funciones.
* eliminarPorId(id: string): elimina una función por su ID.

**SOLID:**

* **SRP:** Tiene una sola responsabilidad: interactuar con el almacenamiento de funciones.
* **OCP:** Se puede extender para usar distintos tipos de almacenamiento (BD, memoria) sin modificar la interfaz.
* **ISP:** Solo define métodos esenciales para la gestión de funciones.
* **DIP:** La capa de servicio depende de esta interfaz, no de implementaciones concretas.

**2️⃣ IPeliculaRepository.ts**

**Propósito:**

Gestiona **entidades Pelicula**, con métodos CRUD y actualización parcial (actualizar):

* guardar(pelicula: Pelicula), obtenerPorId, obtenerTodas, eliminarPorId, actualizar(id, data).

**SOLID:**

* **SRP:** Solo gestiona almacenamiento de películas.
* **OCP:** Se puede cambiar la implementación de persistencia sin alterar servicios que dependen de esta interfaz.
* **ISP:** No obliga a métodos innecesarios; define los esenciales para CRUD y actualización.
* **DIP:** Los servicios de películas dependen de esta interfaz, no de implementaciones concretas.

**3️⃣ ISalaRepository.ts**

**Propósito:**

Maneja **entidades Sala**, con CRUD y actualización parcial:

* guardar(sala: Sala), obtenerPorId, obtenerTodas, eliminarPorId, actualizar(id, data).

**SOLID:**

* **SRP:** Única responsabilidad: persistencia de salas.
* **OCP:** Permite distintos tipos de almacenamiento.
* **ISP:** Define solo métodos necesarios para la gestión de salas.
* **DIP:** Servicios dependen de esta interfaz y no de implementaciones específicas.

**4️⃣ IUsuarioRepository.ts**

**Propósito:**

Gestiona **entidades Usuario**, permitiendo:

* guardar(usuario: Usuario), obtenerPorId, obtenerPorCorreo, obtenerTodos, eliminarPorId.

**SOLID:**

* **SRP:** Solo gestiona almacenamiento de usuarios.
* **OCP:** Se puede extender o cambiar la persistencia sin afectar los servicios.
* **ISP:** Solo los métodos necesarios para gestionar usuarios.
* **DIP:** Los servicios que interactúan con usuarios dependen de esta interfaz.

**5️⃣ IVentaRepository.ts**

**Propósito:**

Maneja **entidades Venta**, incluyendo filtrado por función:

* guardar(venta: Venta), obtenerPorId, obtenerTodas, obtenerPorFuncionId, eliminarPorId.

**SOLID:**

* **SRP:** Solo persiste y consulta ventas.
* **OCP:** Puede cambiar la fuente de datos sin afectar servicios.
* **ISP:** No impone métodos adicionales; solo los esenciales para ventas.
* **DIP:** Los servicios de ventas dependen de esta interfaz.

**🛠️ Conclusión técnica**

* Estas interfaces **desacoplan la lógica de negocio del almacenamiento**, facilitando **tests unitarios** y cambios futuros en persistencia (InMemory, Base de Datos, API externa).
* Cumplen principios **SOLID**, especialmente **SRP, ISP y DIP**, y permiten extensibilidad sin modificar código existente (OCP).
* Cada repositorio se centra en una **entidad específica**, lo que mejora claridad y mantenibilidad.

***Services:***

**📄 Documentación: Servicios del Dominio**

**📌 Descripción general**

Los **servicios del dominio** encapsulan la **lógica de negocio** de cada entidad y coordinan operaciones entre las entidades y los repositorios.  
Se encargan de crear, listar, buscar, actualizar y validar entidades según las reglas del sistema, asegurando consistencia y seguridad de datos.

Cada servicio sigue los principios **SOLID**, separando responsabilidades y permitiendo extensibilidad sin modificar la lógica existente.

**1️⃣ ComidaService.ts**

**Propósito:**

* Gestiona productos y combos de comida.
* Permite:
  + crearProducto(nombre, precio) → crea un producto individual.
  + crearCombo(nombre, productosIds, tipoCombo) → crea un combo con productos existentes.
  + Listar productos, combos o todos (listarProductos, listarCombos, listarTodos).
  + Buscar por ID (buscarPorId).

**SOLID:**

* **SRP:** Solo maneja productos y combos de comida.
* **OCP:** Permite cambiar la implementación interna de almacenamiento sin afectar otros servicios.
* **ISP:** Define únicamente métodos necesarios para la gestión de comidas.
* **DIP:** Servicios que dependen de comida solo usan esta clase, no la estructura interna.

**2️⃣ ComidaVentaService.ts**

**Propósito:**

* Gestiona las **ventas de comida**.
* Permite:
  + realizarVenta(comprador, seleccionados, comidaService) → genera una venta y valida productos.
  + Listar todas las ventas (listarVentas) o por usuario (listarVentasPorUsuario).

**SOLID:**

* **SRP:** Única responsabilidad: gestionar ventas de comida.
* **OCP:** Puede cambiar la persistencia de ventas sin afectar servicios que usan esta clase.
* **ISP:** Solo define métodos necesarios para ventas de comida.
* **DIP:** Depende de ComidaService para obtener productos, pero no de su implementación interna.

**3️⃣ FuncionService.ts**

**Propósito:**

* Gestiona **Funciones de cine** (asignación de película, sala, fecha y precio).
* Permite:
  + Crear funciones (crearFuncion) validando duplicados y fechas.
  + Listar (listarFunciones) y buscar por ID (buscarFuncionPorId).
  + Eliminar funciones (eliminarFuncion).
* Utiliza IFuncionRepository y IdGenerator para crear identificadores únicos.

**SOLID:**

* **SRP:** Solo gestiona funciones y su validación.
* **OCP:** Repositorio puede cambiar sin afectar la lógica.
* **ISP:** Define métodos esenciales para funciones.
* **DIP:** Depende de la interfaz IFuncionRepository, no de implementaciones concretas.

**4️⃣ PeliculaService.ts**

**Propósito:**

* Gestiona **Películas**.
* Permite:
  + Crear películas (crearPelicula) validando título y duración.
  + Listar todas (listarPeliculas).
  + Buscar por ID (buscarPeliculaPorId) y eliminar (eliminarPelicula).
* Usa IPeliculaRepository y IdGenerator.

**SOLID:**

* **SRP:** Solo gestiona películas.
* **OCP:** Posibilidad de cambiar persistencia sin modificar la lógica.
* **ISP:** Métodos esenciales para películas.
* **DIP:** Servicios dependen de la interfaz, no de implementación.

**5️⃣ SalaService.ts**

**Propósito:**

* Gestiona **Salas de cine**, filas y asientos.
* Permite:
  + Crear sala (crearSala) con validaciones de filas, columnas y capacidad.
  + Listar (listarSalas) y buscar por ID (buscarSalaPorId).
  + Eliminar (eliminarSala).
* Usa ISalaRepository y IdGenerator.

**SOLID:**

* **SRP:** Solo gestiona salas.
* **OCP:** Permite cambiar almacenamiento sin afectar servicios.
* **ISP:** Métodos esenciales para salas.
* **DIP:** Depende de la interfaz ISalaRepository.

**6️⃣ UsuarioService.ts**

**Propósito:**

* Gestiona **usuarios** y roles.
* Permite:
  + Registro (registrar) con validación de correo único.
  + Login (login) verificando contraseña.
  + Buscar por correo (buscarPorCorreo) con creación automática de clientes si no existen.
* Usa IUsuarioRepository y IdGenerator.

**SOLID:**

* **SRP:** Solo gestiona usuarios y autenticación.
* **OCP:** Cambiar repositorio no afecta lógica de usuario.
* **ISP:** Define solo métodos necesarios para usuarios.
* **DIP:** Servicios dependen de la interfaz IUsuarioRepository.

**7️⃣ VentaService.ts**

**Propósito:**

* Gestiona **ventas de entradas de cine**.
* Permite:
  + Realizar ventas (realizarVenta) verificando disponibilidad de asientos y marcándolos como ocupados.
  + Listar todas las ventas (listarVentas) o por usuario (listarVentasPorUsuario).
* Usa IVentaRepository, FuncionService y IdGenerator.

**SOLID:**

* **SRP:** Solo gestiona ventas de entradas.
* **OCP:** Se puede cambiar persistencia de ventas sin afectar lógica.
* **ISP:** Define solo métodos necesarios para ventas.
* **DIP:** Depende de interfaces (IVentaRepository) y servicios (FuncionService) abstractos.

**🛠️ Conclusión técnica**

* Todos los servicios **siguen SRP**: cada uno gestiona una única entidad.
* Aplican **DIP** y **OCP** usando interfaces de repositorios y IdGenerator.
* **ISP** se cumple porque no se exponen métodos innecesarios.
* Los servicios garantizan **validación de reglas de negocio**, coordinación entre entidades y repositorios, y extensibilidad futura.

***Archivos (Formatter):***

**📄 Documentación: Formatter**

**📌 Descripción general**

Los **formatters** se encargan de **dar formato de salida** a los datos de las entidades y servicios para visualización en consola, tablas o facturas.  
No modifican la lógica de negocio ni las entidades, solo transforman la información para presentación.

* Cada clase se centra en una entidad específica (Comida, Película, Sala, Función, Venta).
* Uso de cli-table3 para formatear tablas legibles en terminal.
* Generación de facturas detalladas (Comida y Venta).

**1️⃣ ComidaFormatter.ts**

**Propósito:**

* Presentar **productos y combos de comida** en tabla y generar **facturas de comida**.
* Funciones principales:
  + listarProductos(productos: Comida[]) → tabla de productos individuales.
  + listarCombos(combos: Comida[]) → tabla de combos con detalle de productos y descuentos.
  + listarProductosYCombos(items: Comida[]) → combinación de productos y combos.
  + generarFactura(venta: ComidaVenta) → factura detallada con subtotal, descuento, IVA y total.
  + listarComoTabla(ventas: ComidaVenta[]) → tabla resumida de ventas.

**SOLID:**

* **SRP:** Solo formatea productos y ventas de comida.
* **OCP:** Se pueden agregar nuevas columnas o estilos sin afectar lógica de negocio.
* **ISP:** Define métodos esenciales para presentación de comida.
* **DIP:** Depende únicamente de las entidades (Comida, ComidaVenta), no de servicios internos.

**2️⃣ FuncionFormatter.ts**

**Propósito:**

* Mostrar **funciones de cine** en formato de tabla.
* Función principal:
  + listarComoTabla(funciones: Funcion[]) → tabla con ID, película, sala, fecha, precio por entrada y asientos disponibles.

**SOLID:**

* **SRP:** Único objetivo: formatear funciones.
* **OCP:** Se pueden cambiar columnas de salida sin modificar la lógica de función.
* **ISP:** Métodos mínimos necesarios para presentación.
* **DIP:** Depende de Funcion únicamente.

**3️⃣ PeliculaFormatter.ts**

**Propósito:**

* Mostrar **películas** en tabla.
* Función principal:
  + listarComoTabla(peliculas: Pelicula[]) → tabla con ID, título, duración, clasificación y género.

**SOLID:**

* **SRP:** Formateo de películas únicamente.
* **OCP:** Ampliable para agregar más columnas.
* **ISP:** Solo métodos necesarios para películas.
* **DIP:** Depende solo de Pelicula.

**4️⃣ SalaFormatter.ts**

**Propósito:**

* Mostrar **salas de cine** en tabla.
* Función principal:
  + listarComoTabla(salas: Sala[]) → tabla con ID, nombre, tipo y capacidad.

**SOLID:**

* **SRP:** Formateo exclusivo de salas.
* **OCP:** Se pueden agregar nuevas columnas sin afectar otros componentes.
* **ISP:** Métodos mínimos de salida.
* **DIP:** Solo depende de Sala.

**5️⃣ VentaFormatter.ts**

**Propósito:**

* Mostrar **ventas de entradas de cine** y generar **facturas de venta**.
* Funciones principales:
  + generarFactura(venta: Venta) → detalle de la venta con asientos, precio unitario, subtotal, IVA y total.
  + listarComoTabla(ventas: Venta[]) → tabla con ID, cliente, correo, película, entradas, asientos, precio unitario y total.

**SOLID:**

* **SRP:** Única responsabilidad: presentación de ventas y facturación.
* **OCP:** Se pueden añadir columnas o formatos de facturación sin modificar entidades o servicios.
* **ISP:** Define solo métodos de salida.
* **DIP:** Depende únicamente de la entidad Venta.

**🛠️ Conclusión técnica**

* Todos los **formatters cumplen SRP**: solo presentan datos.
* **OCP**: Permiten agregar más columnas, estilos o formatos sin tocar la lógica de negocio.
* **ISP**: Solo métodos necesarios para presentación de cada entidad.
* **DIP**: Dependen únicamente de entidades, no de servicios internos ni de persistencia.
* Separación clara entre **lógica de negocio** y **presentación**.

***Carpeta (Infra/inMemory):***

**📄 Documentación: Infraestructura In-Memory**

**📌 Descripción general**

Los repositorios **In-Memory** simulan la persistencia de datos en memoria para las entidades del proyecto.  
No dependen de bases de datos externas y permiten realizar pruebas y desarrollo rápido.

Cada repositorio implementa la **interfaz correspondiente** definida en domain/interFaces, asegurando **contratos claros** entre servicios y persistencia.

Se garantiza que:

* Se respeta la integridad de las entidades.
* Se cumplen los principios **SOLID**, especialmente DIP y SRP.
* Los métodos guardar, obtenerPorId, obtenerTodas y eliminarPorId son consistentes entre repositorios.

**1️⃣ FuncionRepositoryInMemory.ts**

**Propósito:**

* Persistir funciones de cine temporalmente en memoria.
* Métodos:
  + guardar(funcion: Funcion) → agrega una función.
  + obtenerPorId(id: string) → devuelve función según ID.
  + obtenerTodas() → devuelve todas las funciones.
  + eliminarPorId(id: string) → elimina función por ID.

**SOLID:**

* **SRP:** Solo gestiona la persistencia de funciones.
* **OCP:** Puede extenderse a nuevas estrategias de almacenamiento (archivo, DB) sin afectar servicios.
* **ISP:** Cumple solo la interfaz IFuncionRepository.
* **DIP:** Depende de la interfaz IFuncionRepository, no del servicio que lo utiliza.

**2️⃣ PeliculaRepositoryInMemory.ts**

**Propósito:**

* Persistir películas temporalmente en memoria.
* Métodos:
  + guardar(pelicula: Pelicula)
  + obtenerPorId(id: string)
  + obtenerTodas()
  + eliminarPorId(id: string)
  + actualizar(id, data) → no implementado por limitación de setters en la entidad.

**SOLID:**

* **SRP:** Persistencia exclusiva de películas.
* **OCP:** Se puede adaptar a otra forma de almacenamiento.
* **ISP:** Implementa solo métodos de IPeliculaRepository.
* **DIP:** Repositorio in-memory depende de la interfaz, desacoplado del servicio.

**3️⃣ SalaRepositoryInMemory.ts**

**Propósito:**

* Almacenar salas de cine temporalmente.
* Métodos:
  + guardar(sala: Sala)
  + obtenerPorId(id: string)
  + obtenerTodas()
  + eliminarPorId(id: string)
  + actualizar(id, data) → no implementado (similar a películas).

**SOLID:**

* **SRP:** Persistencia de salas únicamente.
* **OCP:** Compatible con futuras implementaciones persistentes.
* **ISP:** Implementa solo ISalaRepository.
* **DIP:** Depende de la interfaz ISalaRepository.

**4️⃣ UsuarioRepositoryInMemory.ts**

**Propósito:**

* Persistir usuarios temporalmente.
* Métodos:
  + guardar(usuario: Usuario)
  + obtenerPorId(id: string)
  + obtenerPorCorreo(correo: string)
  + obtenerTodos()
  + eliminarPorId(id: string)

**SOLID:**

* **SRP:** Gestiona solo la persistencia de usuarios.
* **OCP:** Se puede cambiar almacenamiento sin afectar servicios.
* **ISP:** Implementa solo la interfaz IUsuarioRepository.
* **DIP:** Repositorio depende de la interfaz, no de los servicios que lo usan.

**5️⃣ VentaRepositoryInMemory.ts**

**Propósito:**

* Persistir ventas de entradas de cine temporalmente.
* Métodos:
  + guardar(venta: Venta)
  + obtenerPorId(id: string)
  + obtenerTodas()
  + obtenerPorFuncionId(funcionId: string)
  + eliminarPorId(id: string)

**SOLID:**

* **SRP:** Única responsabilidad: persistencia de ventas.
* **OCP:** Puede reemplazarse por persistencia real sin cambios en servicios.
* **ISP:** Implementa únicamente IVentaRepository.
* **DIP:** Desacoplado de los servicios, depende de la interfaz.

**🛠️ Conclusión técnica**

* Todos los repositorios **siguen SRP y DIP**, manteniendo el desacoplamiento.
* Permiten pruebas unitarias sin necesidad de base de datos.
* Son fácilmente reemplazables por repositorios persistentes en DB u otras fuentes.
* Cumplen las interfaces de dominio, asegurando consistencia con servicios.

***Carpeta (utils):***

**📄 Documentación: Utilidades - Generar Reporte de Ventas**

**Archivo: generarReporteVentas.ts**

**Propósito:**

* Generar un **reporte en formato JSON** de todas las ventas realizadas.
* Exporta la función generarReporteVentasJSON que recibe un arreglo de objetos Venta y lo convierte en un archivo JSON legible.

**Funcionalidad:**

1. **Mapeo de datos de ventas:**
2. const reporte = ventas.map(v => ({
3. id: v.getId(),
4. fecha: v.getFuncion().getFecha().toISOString(),
5. pelicula: v.getFuncion().getPelicula().getTitulo(),
6. sala: v.getFuncion().getSala().getNombre(),
7. cantidad: v.getCantidad(),
8. total: v.getTotal(),
9. }));
   * Extrae información relevante de cada venta: ID, fecha, película, sala, cantidad de entradas y total.
   * Convierte la fecha a formato ISO para compatibilidad.
10. **Generación de archivo JSON:**
11. const ruta = path.join(\_\_dirname, "../../../output/reporte\_ventas.json");
12. fs.writeFileSync(ruta, JSON.stringify(reporte, null, 2), "utf-8");
    * Define la ruta de salida dentro de la carpeta output.
    * Escribe el archivo JSON con **indentación de 2 espacios** para legibilidad.
13. **Manejo de errores:**
14. try { ... } catch (error: any) { console.error(...) }
    * Captura errores de escritura en disco y los muestra en consola.
15. **Salida informativa:**
16. console.log(`\n✅ Reporte generado correctamente en: ${ruta}`);
    * Informa al usuario que el archivo se creó correctamente.

**SOLID aplicado:**

* **SRP (Single Responsibility Principle):** La función **solo se encarga de generar un reporte JSON**.
* **OCP (Open/Closed Principle):** Puede extenderse para generar otros formatos (CSV, Excel) sin modificar la función actual.
* **DIP (Dependency Inversion Principle):** Depende solo de la abstracción Venta y del módulo estándar fs, no del servicio de ventas.

**Uso:**

import { generarReporteVentasJSON } from "./utils/generarReporteVentas";

import { Venta } from "../domain/enditades/Venta";

const ventas: Venta[] = [...]; // Array de ventas

generarReporteVentasJSON(ventas);

* Generará un archivo reporte\_ventas.json en la carpeta output con toda la información de las ventas.

***Reporte general(conclusión):***

**🔹 Conclusión General del Proyecto**

Este proyecto implementa un sistema completo de gestión de **cines**, con manejo de **películas, salas, funciones, usuarios, ventas y productos de comida**, siguiendo principios de **buen diseño orientado a objetos** y patrones de arquitectura limpia.

**1. Diseño y organización**

* La estructura del proyecto sigue la separación en capas:
  + **Domain:** Contiene las entidades (Pelicula, Sala, Usuario, Funcion, Venta, Comida, ComidaVenta) y servicios de negocio (PeliculaService, SalaService, VentaService, etc.).
  + **Infra/InMemory:** Repositorios en memoria que implementan interfaces de persistencia (IPeliculaRepository, ISalaRepository, IVentaRepository, etc.).
  + **Formatter:** Clases para formatear y mostrar información en tablas o facturas en consola (PeliculaFormatter, SalaFormatter, VentaFormatter, ComidaFormatter, FuncionFormatter).
  + **Utils:** Funciones auxiliares, como generarReporteVentasJSON, para exportar datos a archivos JSON.
* Cada módulo tiene una **responsabilidad clara**, siguiendo el principio **SRP (Single Responsibility Principle)** de SOLID.

**2. Aplicación de principios SOLID**

* **SRP:** Cada clase o servicio realiza solo una función específica:
  + Entidades modelan datos.
  + Servicios gestionan la lógica de negocio.
  + Repositorios manejan la persistencia.
  + Formatters generan visualizaciones.
  + Utils realizan tareas de exportación de datos.
* **OCP:** Las clases son fácilmente extensibles, por ejemplo:
  + Nuevos formatos de reporte o formatos de tabla pueden agregarse sin modificar las clases existentes.
* **LSP:** Las interfaces de repositorios permiten intercambiar implementaciones (memoria, base de datos, API externa) sin afectar el resto del sistema.
* **ISP:** Cada servicio o repositorio implementa únicamente los métodos necesarios, evitando interfaces innecesarias.
* **DIP:** Los servicios dependen de abstracciones (interfaces de repositorio) en lugar de implementaciones concretas, facilitando pruebas y mantenimiento.

**3. Principales funcionalidades**

* **Gestión de entidades:**
  + Crear, listar, eliminar y buscar películas, salas, funciones y usuarios.
  + Control de disponibilidad de asientos por función.
* **Ventas:**
  + Registro de ventas de entradas y comida, con cálculo de totales e IVA.
  + Facturación en consola con formato legible.
* **Formatters:**
  + Visualización clara de datos en tablas para consola.
  + Facturas detalladas para clientes y administración.
* **Utils:**
  + Exportación de reportes de ventas a JSON para análisis externo.
* **Repositorios in-memory:**
  + Implementaciones temporales que facilitan pruebas y desarrollo sin base de datos.

**4. Conclusión final**

Este proyecto demuestra un enfoque **robusto y escalable** para un sistema de cine:

* Cumple con los principios de **programación orientada a objetos y SOLID**.
* La separación de responsabilidades facilita **mantenimiento, pruebas y futuras extensiones**.
* Su diseño modular permite integrar nuevas características, como base de datos real, APIs externas o nuevas formas de facturación, sin afectar la arquitectura existente.
* La capa de visualización con Formatter y la exportación de datos con Utils ofrecen una experiencia tanto para usuario como para administración.

En resumen, es un proyecto **bien estructurado y preparado para crecer**, con un código **claro, mantenible y orientado a buenas prácticas**.