
# Portafolio de Programación Orientada a Objetos

## Portafolio Final

Integrantes:

Amanda Padilla Leiva

Institución Educativa:

Universidad Cenfotec

Curso:

Programación Orientada a objetos

Profesor: Álvaro Cordero

Fecha de entrega: 12 de Diciembre de 2025

# Tabla de Contenidos

## Justificación

El problema seleccionado es el diseño e implementación de un sistema tipo biblioteca de películas o “Tienda de alquiler de películas”, pensado para la gestión, organización y registro de películas. La razón de esta elección surge de la necesidad común de administrar colecciones de películas de forma organizada, la cual permite que se puedan clasificar por género, director, actores, duración, si son apropiadas o no para todo público. Este tipo de sistema ha estado presente en varios contextos a lo largo de la historia. En el pasado , se empleaba de manera extensa en establecimientos que alquilaban películas en VHS, DVDs o Blue-Ray, donde la información de cada una debía registrarse en una computadora, o incluso, en registros físicos dentro de cada local. En la actualidad, ese concepto se mantiene tanto en plataformas de streaming que tienen catálogos de series y películas organizados como en las plataformas sociales como Letterbox, donde los usuarios pueden calificar, clasificar y hacer reseñas sobre las películas según sus gustos.

Como este sistema incluye una variedad de entidades interconectadas que pueden ser modeladas de forma natural como clases y objetos, el contexto es perfecto para implementar la Programación Orientada a Objetos (POO). La POO permite una representación clara y ordenada de elementos como Película, Director, Género, Sinopsis y Biblioteca, así como las relaciones que existen entre ellos. Asimismo, posibilita una solución escalable y adaptable, que en el futuro, podría incorporar nuevas funcionalidades, como recomendaciones personalizadas o hasta la combinación con una base de datos.

## Conceptos aplicados en el sistema.

El sistema se diseñó identificando las entidades clave del dominio:

- Pelicula
- Actor
- Director
- User
- Administrador
- Cuenta (clase abstracta)

Estas entidades se relacionan entre sí mediante diferentes tipos de relaciones orientadas a objetos: asociación, agregación, composición y dependencia. Los módulos se dividen en capas siguiendo la arquitectura recomendada para sistemas mantenibles y extensibles.

### Abstracción

La abstracción se emplea al modelar entidades del mundo real como clases: Película, Actor, Director, Usuario, Administrador. Cada clase expone solo lo esencial, ocultando detalles internos como validaciones, listas internas o relaciones complejas.

### Encapsulamiento

Todos los atributos son privados y se accede a ellos mediante getters y setters. Esto evita accesos indebidos y protege el estado del objeto. Ejemplo:

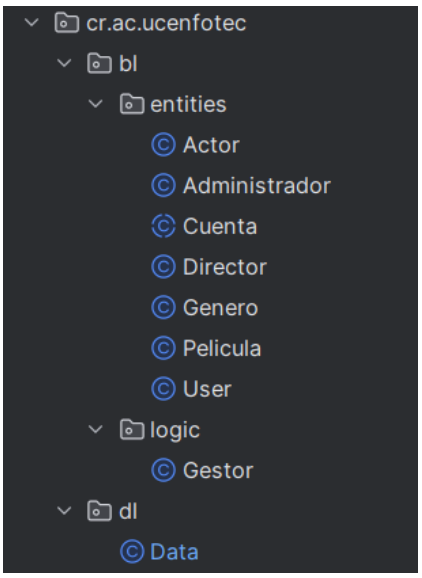
```java
// GETTERS & SETTERS
public String getId() { return id; }  Amanda Padilla

public void setId(String id) { this.id = id; }  Amanda Padilla

public String getTitulo() { return titulo; }  6 usages  Amanda Padilla

public void setTitulo(String titulo) { this.titulo = titulo; }  2 usages  Amanda Padilla

public int getAnio() { return anio; }  6 usages  Amanda Padilla

public void setAnio(int anio) { this.anio = anio; }  2 usages  Amanda Padilla
```

**Modularidad**

El programa se divide en paquetes:

- ui → Interfaz de usuario
- tl → Controlador
- bl.entities → Entidades
- bl.logic → Lógica del negocio (Gestor)
- dl → Acceso a datos

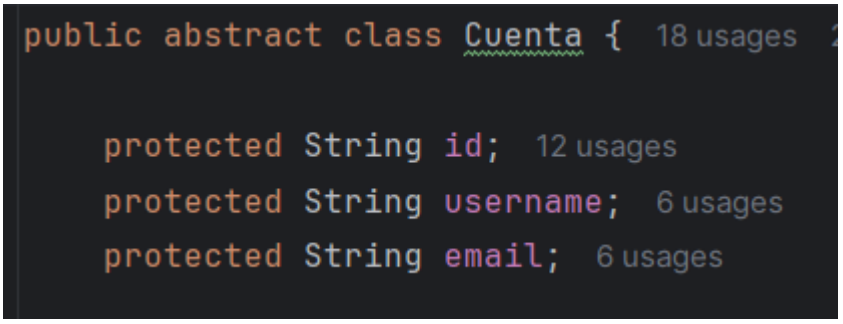
Esta separación permite escalar el proyecto sin afectar módulos ajenos.

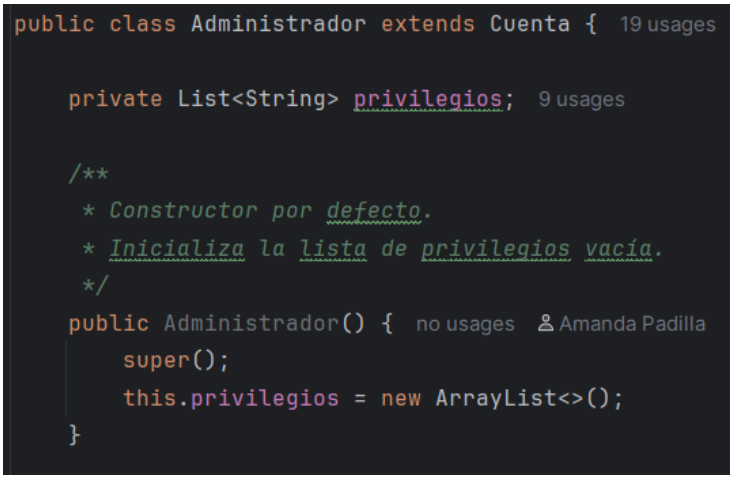


**Herencia**

El diseño propone el uso de herencia para organizar y reutilizar comportamientos comunes entre las clases. En este enfoque, creamos una nueva clase llamada Cuenta que funciona como una clase abstracta que encapsula aquellos atributos y métodos generales que comparten todos los tipos de cuentas dentro del sistema. A partir de esta base, las clases User y Administrador extienden dicha funcionalidad, heredando sus características y especializándose según las necesidades particulares de cada tipo de usuario.

Gracias a esta estructura, el sistema puede manejar tanto a un User como a un Administrador bajo la referencia genérica de Cuenta, lo que facilita el uso de polimorfismo, permitiendo que distintos tipos concretos se comporten de manera uniforme en contextos donde solo importa la funcionalidad común.



```java
public class User extends Cuenta {  22 usages  Amanda Padilla +1

    private ArrayList<Pelicula> favoritos;  8 usages

    /**
     * Constructor por defecto.
     * Inicializa la lista de favoritos vacía.
     */
    public User() {  no usages  Amanda Padilla
        super();
        this.favoritos = new ArrayList<>();
    }
```

**Polimorfismo**

La lista global de cuentas almacena objetos de tipo Cuenta:

- private List<Cuenta> cuentas;

Pero puede contener tanto User como Administrador. Esto permite tratarlos de forma homogénea, demostrando polimorfismo.

```java
public boolean agregarUsuario(User u) {  1 usage  Amanda Padilla
    if (u == null || u.getId() == null) return false;
    if (buscarUsuarioPorId(u.getId()) != null) return false;

    boolean agregado = usuarios.add(u);
    if (agregado) {
        cuentas.add(u); // User es una Cuenta (polimorfismo)
    }
    return agregado;
}

/**
 * Busca un usuario por su id.
 *
 * @param id identificador del usuario
 * @return el usuario encontrado o null si no existe
 */
public User buscarUsuarioPorId(String id) {  2 usages  Amanda Padilla
    if (id == null) return null;
    for (User u : usuarios) {
        if (u.getId() != null && u.getId().equalsIgnoreCase(id)) {
            return u;
        }
    }
    return null;
```

**Asociación**

En el diseño, la asociación se utiliza para describir relaciones entre clases cuyos objetos pueden existir de manera totalmente independiente, sin que una entidad dependa estructuralmente de la otra. Este tipo de vínculo refleja interacciones naturales dentro del dominio, pero sin implicar pertenencia o dependencia estricta.

En este contexto, una película puede estar asociada a un director, lo que indica una relación funcional, pero no una dependencia: la película puede existir incluso sin que se haya asignado un director específico, y el director continúa existiendo y trabajando independientemente de cualquier película concreta. De la misma forma, una película puede estar asociada a varios actores, quienes mantienen su existencia y trayectoria profesional sin importar si participan o no en una producción específica.

También ocurre algo similar con un usuario asociado a un conjunto de películas favoritas. El usuario no deja de existir si elimina todas sus películas preferidas, y las películas tampoco dependen del usuario para existir dentro del sistema.

```
public boolean asociarPeliculaConDirector(Pelicula peli, Director dir) {
    if (peli == null || dir == null) return false;

    peli.setDirector(dir);

    if (dir.getPeliculasDirigidas() == null) {
        dir.setPeliculasDirigidas(new ArrayList<>());
    }
    if (!dir.getPeliculasDirigidas().contains(peli)) {
        dir.getPeliculasDirigidas().add(peli);
    }
    return true;
}
```

**Agregación**

La agregación representa una relación “todo–parte” en la que las partes mantienen su independencia, aunque exista una organización conceptual entre ellas. En el proyecto, la relación Pelicula → Género se modela como una agregación: un género puede aplicarse a muchas películas y no depende de ninguna para existir. Por eso, la película simplemente mantiene una lista de géneros, reflejando esta vinculación flexible sin una dependencia fuerte entre las clases.

```
private ArrayList<Genero> generos;
```

```
 */
public void setGeneros(ArrayList<Genero> generos) {  1 usage  Amanda Pa
    this.generos = (generos != null) ? generos : new ArrayList<>();
}
```

**Composición**

La composición describe una relación “todo–parte” en la que la parte depende completamente del objeto principal. En el sistema, esto se refleja en la relación Pelicula → Ficha (sinopsis), donde la ficha no tiene sentido fuera de la película. Por esa razón, Ficha se declaró como una clase interna dentro de Pelicula, lo que refuerza su dependencia total y evita que sea utilizada por otros módulos.

Al ser una parte inseparable, la película controla totalmente su creación y eliminación: si una película deja de existir, su ficha también desaparece. Este comportamiento coincide perfectamente con la definición de composición. La implementación como record (por ejemplo, public record Ficha(String sinopsis) {}) aporta inmutabilidad y simplicidad, permitiendo representar la sinopsis como un dato limpio, estable y encapsulado dentro de la estructura de la película.

Modelarla como clase externa habría añadido complejidad innecesaria y habría debilitado la relación, convirtiéndola en una asociación o agregación que no representa la dependencia real. En cambio, mantener Ficha como clase interna logra cohesión, claridad conceptual y un diseño fiel a la idea de que la sinopsis forma parte esencial de la película y no puede existir sin ella.

```
public record Ficha(String sinopsis) {}

private Ficha ficha;  2 usages
```

**Dependencia**

El Gestor depende de Data para recuperar, modificar o eliminar elementos del sistema, el Controller depende del Gestor para ejecutar la lógica del negocio.

```
private void actualizarPelicula() throws IOException {  1 usage  Amanda Padill
    ui.mostrarMensaje( msg: "\n=== Actualizar película ===");
    String id = ui.leerId( label: "ID de la película a actualizar:");
    Pelicula p = gestor.buscarPeliculaPorId(id);
    if (p == null) {
        ui.mostrarMensaje( msg: "Película no encontrada.");
        return;
    }
```

## Explicación de diseño y correcciones de la primera versión del portafolio

### Arquitectura de Clases

El proyecto se encuentra organizado en paquetes, siguiendo una arquitectura por capas:

- **cr.ac.ucenfotec.ui**

Contiene la clase UI, responsable de todas las interacciones por consola. No implementa reglas de negocio, solo muestra información y captura datos del usuario.

- **cr.ac.ucenfotec.tl**

Contiene la clase Controller, que se encarga de orquestar el flujo del programa. Llama a los métodos del Gestor, gestiona el estado de sesión (sin sesión, usuario, administrador) y decide qué menú mostrar.

- **cr.ac.ucenfotec.bl.entities**

Contiene las entidades del dominio: Cuenta, User, Administrador, Pelicula, Actor, Director, Genero y la record Ficha dentro de Pelicula.

- **cr.ac.ucenfotec.bl.logic**

Contiene la clase Gestor, que implementa las operaciones del sistema (CRUD de películas, registro de actores y directores, asociación de entidades, manejo de favoritos y listado polimórfico de cuentas).

- **cr.ac.ucenfotec.dl**

Contiene la clase Data, que mantiene las listas de entidades en memoria (peliculas, usuarios, administradores, actores, directores, cuentas). Aquí se centraliza la “persistencia en memoria” del sistema.

Esta arquitectura por capas muestra una separación clara entre la interfaz de usuario, la lógica de control, la lógica de negocio y la capa de datos, lo cual responde de forma directa al objetivo del curso de trabajar con aplicaciones por capas.

**Uso del Final**

El uso de final en el código se justifica porque garantiza que ciertas referencias esenciales no puedan modificarse después de ser inicializadas. Esto evita reemplazos accidentales y mantiene recursos críticos en un estado estable durante toda la ejecución. Un ejemplo claro es el lector de entrada:

Al declararlo como final, se asegura que el lector no pueda ser sustituido, lo que previene errores y refuerza el encapsulamiento. En conjunto, final se utiliza para proteger dependencias importantes y mantener la lógica interna del sistema consistente y segura.

```
private final BufferedReader in = new BufferedReader(new InputStreamReader(System.in));
```

**Jerarquía de cuentas y polimorfismo**

Para modelar las personas que utilizan el sistema se diseñó una jerarquía de clases basada en la abstracción Cuenta. La clase Cuenta es una clase abstracta que define los atributos y comportamientos comunes de cualquier cuenta en el sistema:

- id
- username
- email
- El método abstracto getTipoCuenta()

A partir de esta clase se derivan dos clases concretas:

- User: representa a una persona usuaria normal, con una lista de películas favoritas.
- Administrador: representa una cuenta con privilegios de administración, incluyendo una lista de permisos.

Esta jerarquía permite aplicar herencia y polimorfismo. Gracias a esto, en la clase Data se mantiene una lista polimórfica List<Cuenta> cuentas, donde se almacenan tanto instancias de User como de Administrador. La UI puede listar todas las cuentas tratándolas como Cuenta, pero al momento de mostrar la información se distingue el tipo concreto mediante getTipoCuenta() y el uso de instanceof. Esta decisión de diseño demuestra polimorfismo en la práctica y simplifica la gestión de cuentas al no tener que mantener estructuras separadas para cada subtipo.

**Constructores por defecto y constructores sobrecargados**

Otra decisión importante fue incluir tanto constructores por defecto como constructores completos o sobrecargados en las clases de entidades (Pelicula, Actor, Director, User, Administrador, Genero).

El constructor por defecto permite crear objetos en un estado inicial válido, inicializando listas con new ArrayList<>() para evitar valores nulos. Los constructores completos permiten crear instancias pasando todos los datos relevantes de una sola vez, lo cual mejora la legibilidad y facilita las pruebas. Este enfoque permite usar el sistema de forma flexible en diferentes contextos y, al mismo tiempo, garantiza que las colecciones nunca queden en null, lo que mejora la robustez del código.

Por ejemplo, en Pelicula se ofrece tanto un constructor sin parámetros como uno completo:

```java
 * Constructor por defecto.
 * Inicializa las listas de géneros y elenco vac
 */
public Pelicula() {  1 usage  Amanda Padilla
    this.generos = new ArrayList<>();
    this.elenco = new ArrayList<>();
}

/**
```

```java
public Pelicula(String id, String titulo, int anio, int duracionMinutos, String clasificacion,
                ArrayList<Genero> generos, ArrayList<Actor> elenco, Director director) {

    this.id = id;
    this.titulo = titulo;
    this.anio = anio;
    this.duracionMinutos = duracionMinutos;
    this.clasificacion = clasificacion;

    this.generos = (generos != null) ? generos : new ArrayList<>();
    this.elenco = (elenco != null) ? elenco : new ArrayList<>();
    this.director = director;
}
```

**Identidad de objetos**

En las clases de entidades principales (Pelicula, Actor, Director, User, Administrador) se sobrescribieron los métodos equals y hashCode, definiendo la identidad de los objetos a partir del atributo id. Esto significa que dos instancias se consideran iguales si comparten el mismo identificador, aunque sean objetos distintos en memoria.

Por ejemplo, en la clase Pelicula:

```java
@Override  Amanda Padilla
public boolean equals(Object o) {
    if (this == o) return true;
    if (!(o instanceof Director)) return false;
    Director director = (Director) o;
    return Objects.equals(id, director.id);
}

@Override  Amanda Padilla
public int hashCode() { return Objects.hash(id); }
```


Esta decisión de diseño es coherente con un sistema de catálogo, donde cada película, actor o usuario debe identificarse de manera única por su ID. Además, prepara el código para un posible uso futuro de colecciones como Set o estructuras basadas en hash, donde equals y hashCode son fundamentales para evitar duplicados.

**Eliminación de la clase Catálogo y justificación del rediseño**

En la primera versión del portafolio, el sistema incluía una clase denominada Catálogo, encargada de agrupar las colecciones de películas, actores y directores, además de ejecutar sobre ellas diversas operaciones como registros, búsquedas o modificaciones. Con el avance del proyecto y la necesidad de adoptar una arquitectura por capas más clara y profesional, se identificó que esta clase acumulaba demasiadas funciones y, como resultado, interfería con la correcta separación de responsabilidades dentro del diseño.

Catálogo combinaba simultáneamente la gestión de datos, la lógica de negocio y parte del flujo del sistema, lo cual generaba un alto nivel de acoplamiento. Esta mezcla de responsabilidades hacía que creciera rápidamente y dificultaba su mantenimiento, especialmente cuando se pretendía extender el programa hacia conceptos más avanzados como persistencia, patrón DAO o almacenamiento en archivos y bases de datos. Por estas razones, resultaba necesario replantear su función dentro de la arquitectura general.

En la versión final del sistema, la funcionalidad de Catálogo fue redistribuida de forma más coherente. La clase Data asumió el rol de contenedor de datos, limitándose a mantener en memoria las listas de películas, actores, directores y cuentas, sin incluir ninguna operación compleja. Este cambio permite que la capa de datos esté completamente aislada, lo cual es indispensable si en el futuro se implementa persistencia real mediante archivos o bases de datos.

Por otro lado, todas las operaciones que antes ejecutaba Catálogo pasaron a la clase Gestor, que ahora concentra la lógica del negocio y las reglas del sistema: registrar, modificar y eliminar películas; asociar actores y directores; gestionar favoritos; realizar validaciones; y asegurar la consistencia del catálogo. De esta forma, Gestor funciona como un intermediario formal entre la interfaz y los datos, cumpliendo con los principios de diseño orientado a objetos y con la estructura por capas solicitada en la consigna.

Finalmente, el Controller quedó como el encargado de coordinar el flujo general del programa, decidir qué menú mostrar y recibir las peticiones de la UI para enviarlas al Gestor. Esta reorganización permitió obtener una arquitectura mucho más limpia, clara y sostenible, en la que cada clase cumple un propósito bien definido y no invade las funciones de otras capas.

La eliminación de Catálogo no fue un recorte, sino una mejora fundamental que permitió al sistema alinearse con buenas prácticas de programación, facilitar su comprensión y dejarlo preparado para futuras extensiones. Con esta reestructuración, el proyecto refleja un diseño más profesional y adecuado a los objetivos del curso.

**Cómo funciona el programa**

El sistema se ejecuta por consola y utiliza una arquitectura por capas (UI → Controller → Gestor → Data). El flujo inicia siempre en el menú para usuario invitado, y a partir del tipo de cuenta que inicia sesión, se habilitan diferentes funcionalidades.

**Menú inicial (Usuario invitado)**

Cuando el programa inicia, se muestra un menú básico con opciones del 0 al 5, donde el usuario puede:

- Registrar un Usuario normal
- Registrar un Administrador
- Iniciar sesión
- Listar todas las cuentas registradas (muestra tanto usuarios como administradores)
- Salir del programa

Este menú existe porque, antes de iniciar sesión, el sistema no puede determinar qué permisos tendrá el usuario. Por eso, aquí únicamente se permiten tareas de creación de cuentas e inicio de sesión.

**Menú de Administrador**

Cuando un Administrador inicia sesión correctamente, el sistema despliega un menú administrativo con opciones del 0 al 10.

Este menú otorga acceso total a las funciones CRUD y al registro de entidades.

Opciones disponibles para el Administrador:

0. Cerrar sesión

Retorna al menú de usuario invitado.

**1. Crear película**

Esta opción permite registrar una nueva película solicitando la siguiente información:

ID de película

- Título
- Año
- Duración en minutos
- Clasificación
- Lista de géneros
- Opción de agregar una sinopsis (Ficha)
- Opción de agregar un director: Registro de director dentro de esta opción. Si el administrador indica:
  - “Sí” → se despliega una lista de directores registrados para seleccionar uno.
  - “No” → el sistema abre automáticamente el menú de registro de un nuevo director y lo asocia de inmediato a la película.

Si el usuario indica que desea elegir un director existente pero no hay ninguno registrado, el sistema detecta esto y obliga al registro de uno nuevo antes de continuar.

**2. Listar películas**

Muestra cada película con su información general, incluyendo elenco y director.

**3. Buscar película por ID**

Permite localizar rápidamente una película específica.

**4. Actualizar película**

Permite modificar atributos individuales: título, año, duración, clasificación, géneros, director o sinopsis.

**5. Eliminar película**

Permite eliminarla del catálogo de manera definitiva.

**6. Registrar director**

El administrador ingresa:

- ID
- Nombre

Se crea el director y se agrega al catálogo global.

**7. Listar directores y sus películas**

Muestra cada director junto con la lista de películas que ha dirigido.

**8. Registrar actor**

Solicita:

- ID
- Nombre

Crea el actor y lo agrega al sistema.

**9. Listar actores y su filmografía**

Muestra qué actores existen en el sistema y en qué películas han participado.

**10. Asociar actor a película**

**El sistema pide:**

- ID del actor
- ID de la película

Si ambos existen, el actor se agrega al elenco, y la película se agrega automáticamente a la filmografía del actor.

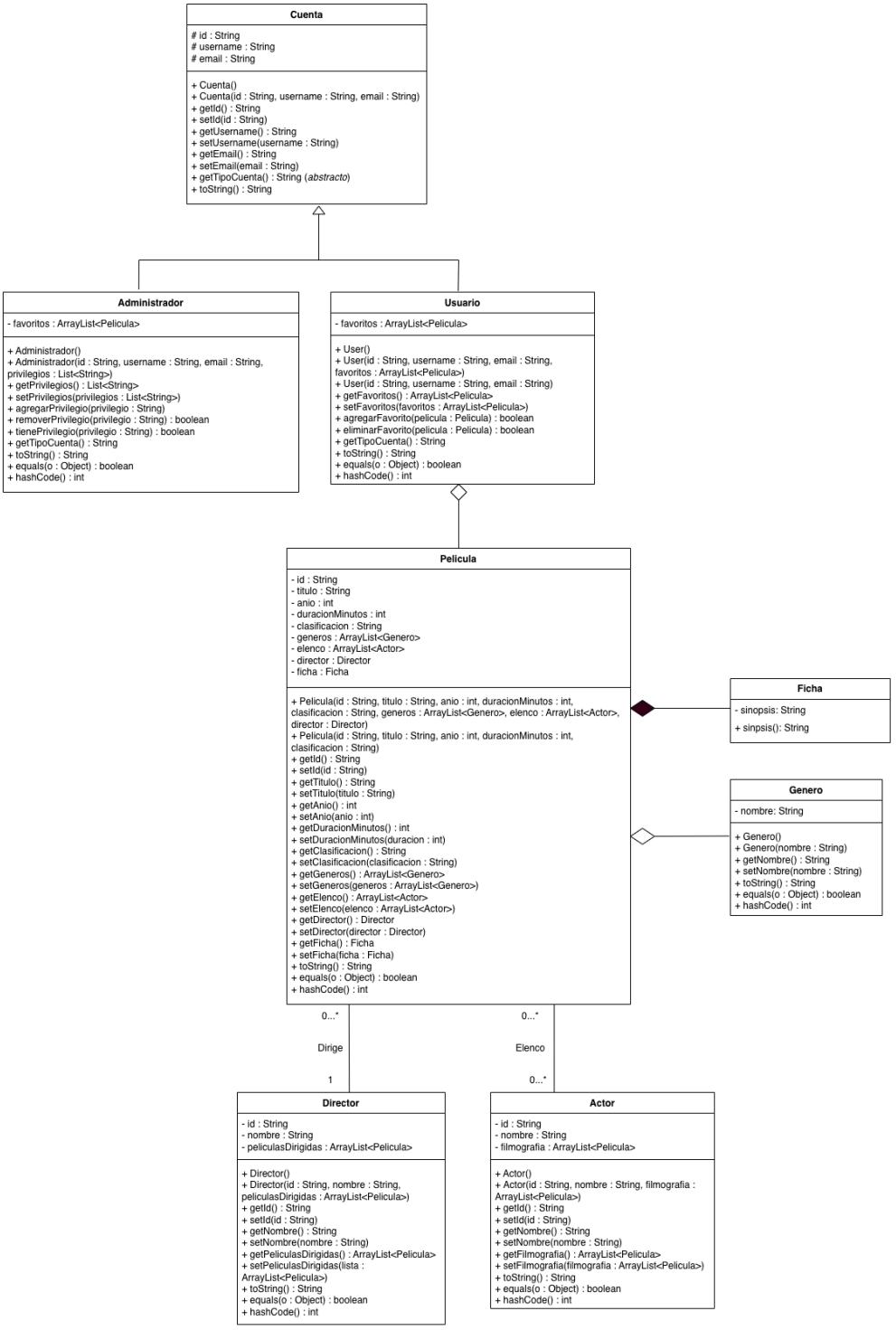
Esta relación es bidireccional, asegurando consistencia en todo el sistema.

**Menú de Usuario normal**

El usuario normal, una vez que inicia sesión, tiene acceso a:

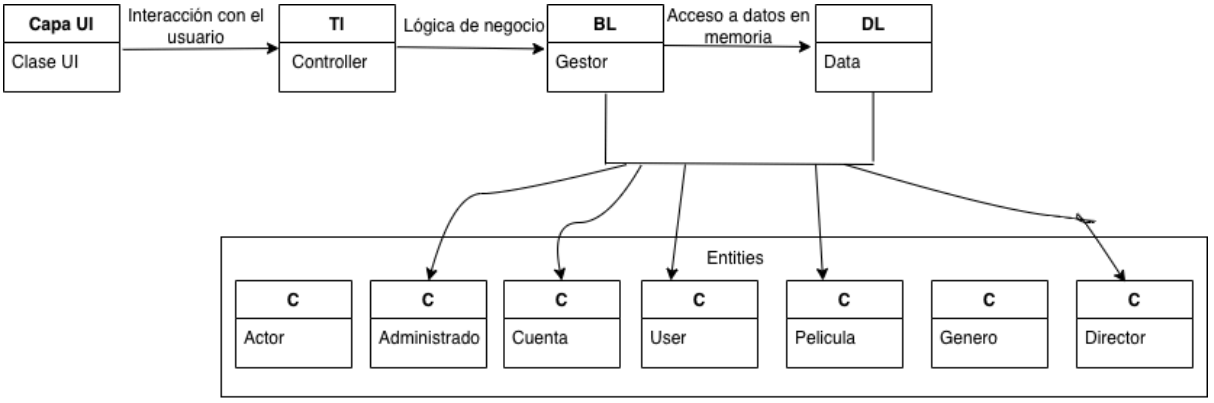
- Listar películas
- Agregar películas a favoritos
- Ver su lista de favoritos
- Listar directores y sus películas dirigidas
- Listar actores y películas en las que actuaron

## Diagramas

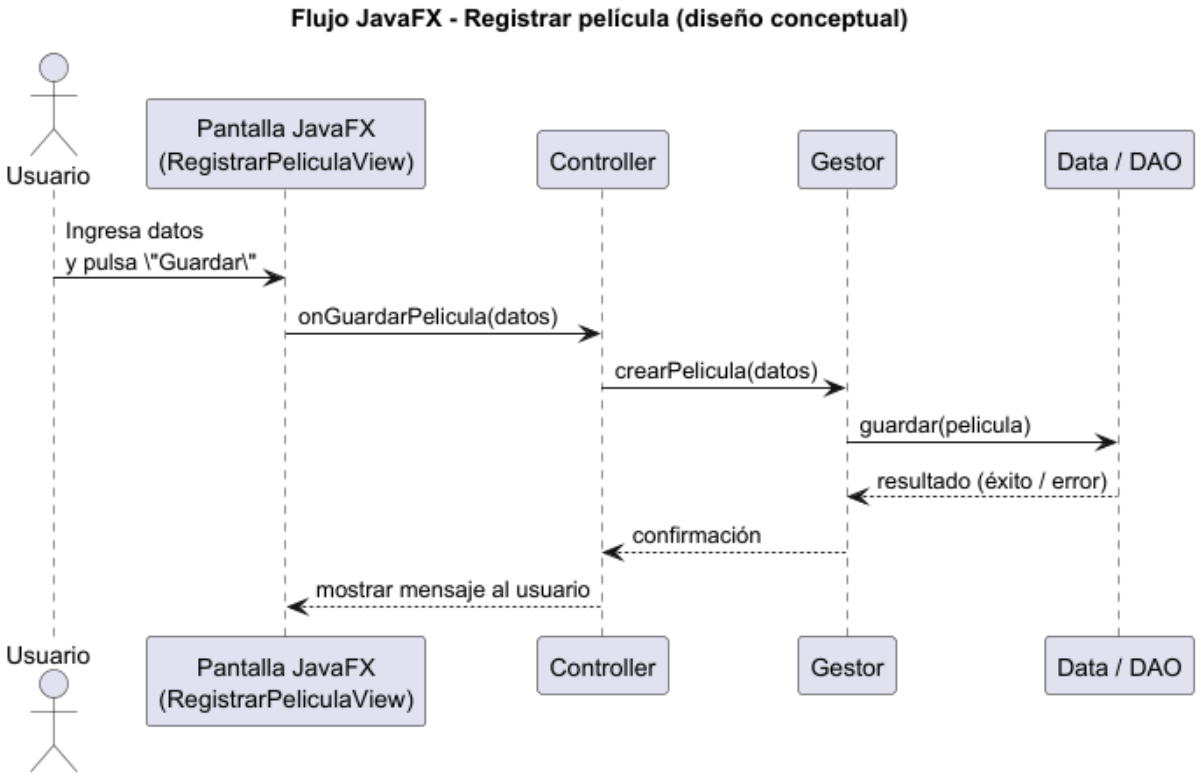



El diagrama utiliza correctamente los principales elementos de POO. La herencia se representa con una flecha de triángulo blanco apuntando a la superclase, donde Usuario y

Administrador extienden a la clase abstracta Cuenta. La composición, mostrada con un rombo negro, se aplica bien entre Película y Ficha, ya que la ficha no puede existir sin su película. La agregación, indicada con un rombo blanco, se usa adecuadamente entre Película y Género, porque un género puede existir de forma independiente. La asociación muchos a muchos entre Película y Actor está bien representada, ya que un actor puede participar en varias películas y una película tener varios actores. Finalmente, la asociación uno a muchos entre Director y Película es correcta: un director puede dirigir múltiples películas, mientras que cada película tiene exactamente un director.




Este diagrama representa la arquitectura en capas del sistema: la capa UI solo se comunica con el Controller; el Controller orquesta y llama al Gestor; el Gestor encapsula la lógica de negocio y delega el acceso a datos en la clase Data; y las entidades se comparten entre BL y DL.




Este diagrama de secuencia ilustra cómo funcionaría el sistema si se implementara una interfaz gráfica con JavaFX. El usuario interactúa con una ventana (RegistrarPeliculaView), que delega la acción al Controller. El Controller envía los datos al Gestor, que crea el objeto Pelicula y lo persiste a través de la capa de datos (Data o un DAO). Finalmente, la respuesta viaja de regreso hasta la vista, donde se muestra un mensaje al usuario. Aunque en esta versión del portafolio la interfaz se implementa por consola, la arquitectura actual permite integrar JavaFX en el futuro sin cambiar la lógica de negocio.

## Conclusiones y Reflexion Personal

El desarrollo de este portafolio permitió consolidar de manera práctica los principios de la Programación Orientada a Objetos aplicados a un sistema realista: una biblioteca o catálogo de películas. A través del análisis del problema, el modelado del dominio y la implementación del código, se integraron conceptos como abstracción, encapsulamiento, herencia, polimorfismo, modularidad y distintos tipos de relaciones entre clases (asociación, agregación, composición y dependencia). El uso de una arquitectura por capas facilitó la separación de responsabilidades, permitiendo que cada módulo cumpliera una función clara y manteniendo el sistema organizado, escalable y fácil de mantener.

Las correcciones realizadas a partir de la primera versión fortalecieron significativamente el diseño, incluyendo la implementación adecuada de constructores, el uso de final para asegurar la estabilidad de dependencias internas, la correcta jerarquía de clases y un uso más explícito del polimorfismo. Asimismo, la formalización de equals y hashCode según el ID de cada entidad mejoró la coherencia y fiabilidad del sistema. También se logró implementar de manera más concisa las clases director y actor, siendo una parte integral y funcional en el código.

En conjunto, el proyecto no solo cumple con los requisitos del curso, sino que constituye una base sólida para futuras mejoras, como persistencia en archivo o base de datos, interfaz gráfica con JavaFX o funcionalidades avanzadas dentro del catálogo. El proceso permitió comprender cómo los principios teóricos de POO se integran dentro de una solución estructurada, realista y aplicable al desarrollo de software profesional.

Al realizar este portafolio, se presentó un proceso de aprendizaje profundo y progresivo, al inicio muchos conceptos como la agregación, composición, incluso la herencia y las clases abstractas me resultaban complejos, sin embargo, conforme avance

en el curso y se implementaron estos temas en el portafolio, pude visualizar su propósito y como cada concepto encajaba dentro del sistema y cómo las decisiones de diseño influyen directamente en la claridad, escalabilidad y correctitud del programa.

Uno de los aprendizajes más valiosos fue entender la importancia de una buena arquitectura y un buen orden. Separar el proyecto en capas me permitió comprender aún más el propósito de cada clase y cada parte del código, organizarlo de esa forma mucho más clara y profesional, evitando la mezcla de responsabilidades y confusiones a la hora de seguir codificando. También comprendí la utilidad de los constructores por defecto, la necesidad de inicializar colecciones para evitar errores de null, y la razón detrás de utilizar final en ciertas dependencias esenciales, cosa que al incio crei que solo se hacía por defecto y no por un propósito más significativo.

Además, este proyecto me ayudó a fortalecer mi capacidad para justificar decisiones técnicas y para analizar críticamente el código, especialmente después de los comentarios del primer avance. Aprendí a mejorar mis clases, a agregar documentación adecuada y a representar correctamente las relaciones en UML, algo que al inicio resultaba confuso.

En general, este portafolio no solo reforzó mis conocimientos en Programación Orientada a Objetos, sino que también me permitió desarrollar una forma más estructurada de pensar y diseñar software. Aunque aún pueden haber muchísimas mejoras a nivel de código, ya que sigo aprendiendo, una implementación que se puede hacer a futuro es listas, no solo listar favoritas sino poder crear listas para películas. También a futuro una implementación de frontend con html y css, también el uso de bases de datos para guardar los datos.

Fue un proceso retador, pero realmente disfruté realizarlo, marcó un antes y un después en mi forma de abordar problemas de programación y arquitectura. Sin duda, son aprendizajes adquiridos que serán fundamentales para futuros cursos y para mi desarrollo profesional en el área de software.

Repositorio de Github: https://github.com/apadillalei/PRIMERPORTAFOLIO