Informe Final

Proyecto SIA Gestor Inmobiliario

Benjamín Contreras
Anselmo Díaz
Ignacio Ruiz

27 de septiembre de 2025

ÍNDICE GENERAL

INTRODUCCIÓN Y OBJETIVOS

Introducción

El presente documento describe el proceso de desarrollo e implementación de un sistema de software para la gestión de ventas en el sector inmobiliario. El proyecto consistió en la creación de una aplicación de escritorio, utilizando **Java** con la biblioteca **SWING** para la interfaz gráfica y **SQLite** para la persistencia de datos.

El sistema final centraliza la administración de múltiples proyectos de departamentos, incorporando una lógica de negocio que permite adaptar los precios de las propiedades a las condiciones del mercado, ofreciendo una solución integral y funcional.

Objetivo del Proyecto

El objetivo principal del proyecto fue el desarrollo de un sistema para el manejo integral del ciclo de venta de departamentos, implementando un modelo de precios dinámico.

Este objetivo se cumplió exitosamente mediante la creación de una aplicación de escritorio que integra las siguientes capacidades:

- Un sistema de persistencia de datos que gestiona la información de proyectos, edificios y departamentos.
- Una interfaz gráfica de usuario intuitiva para que los vendedores administren la cartera de propiedades y realicen operaciones de venta.
- La implementación de un modelo de precios dinámico en la clase `GestorPrecios`, que ajusta el valor de los inmuebles basándose en sus características.

SISTEMA DE INFORMACIÓN (SIA)

SIA1.1: Análisis de Datos y Funcionalidades

El análisis inicial del proyecto estableció las bases para un sistema robusto de gestión inmobiliaria, centrado en un **modelo de precios dinámico**. Aunque la visión original se mantuvo, la implementación final evolucionó para incorporar una arquitectura más técnica y detallada, migrando de un concepto puramente teórico a una aplicación funcional con persistencia de datos y una interfaz gráfica.

A continuación, se describe cómo se materializaron los conceptos iniciales en la versión actual del proyecto:

- **Datos Principales:** El dominio del problema, originalmente planteado en términos generales, se implementó con una estructura de clases específica y bien definida.
  - **Actores del Sistema:** La conceptualización de los roles evolucionó durante el desarrollo. El diseño inicial contemplaba una jerarquía de herencia con una superclase abstracta `Usuario` y dos subclases que representaban los roles específicos: `Vendedor` y `Comprador`. Para la versión final, el modelo se simplificó al máximo, eliminando dicha jerarquía y definiendo una única clase `Comprador` para representar al actor principal del sistema.
  - **Activos Inmobiliarios:** La estructura jerárquica se implementó fielmente. La clase `ProyectoInmobiliario` contiene una colección de `Edificio`, y a su vez, cada `Edificio` gestiona una colección de `Departamento`, modelando así la relación del mundo real.
  - **Transacciones y Estados:** El modelo de transacciones se simplificó eliminando la clase `Reserva`. En el diseño final, la relación entre un comprador y una propiedad se maneja de forma directa: la clase `Departamento` ahora posee un atributo para almacenar al objeto `Comprador` que lo adquiere. El ciclo de vida de la propiedad sigue siendo gestionado por el enumerador `EstadoDepartamento`, cuyos estados (`DISPONIBLE`, `RESERVADO`, `VENDIDO`) se actualizan en correspondencia con la asignación de un comprador al departamento.
- **Funcionalidades Clave:** Las capacidades esenciales del sistema, inicialmente descritas como objetivos, fueron implementadas a través de componentes de software específicos.
  - **Gestión de Cartera de Propiedades:** Esta funcionalidad, originalmente un concepto, se materializó en las ventanas de "Registrar Proyecto" y "Modificar Proyecto" de la interfaz gráfica, gestionadas por la clase `VisualDisplayer`.
  - **Búsqueda y Reserva:** La búsqueda se implementó a través de una ventana de filtrado avanzado. Los criterios seleccionados por el usuario se encapsulan en un objeto de la clase `FiltroBusqueda`, que se utiliza para construir una consulta SQL dinámica contra la base de datos.
  - **Algoritmo de Precios:** El modelo de precios dinámico se implementó en la clase `GestorPrecios`. Esta clase centraliza la lógica para calcular el valor de los inmuebles, aplicando fórmulas que consideran diversas características como los

metros cuadrados, el número de baños y el piso de la unidad, permitiendo una valoración flexible y adaptable.

SIA1.2: Diseño Conceptual y Arquitectura Final

El diseño conceptual original, basado en el paradigma de Orientación a Objetos, se materializó en una arquitectura de software robusta y bien definida. La estructura final del sistema no solo preserva las jerarquías de clases del avance, sino que las integra con las capas de servicio, persistencia y vista, formando un sistema cohesivo y funcional.

La arquitectura implementada se compone de las siguientes capas y jerarquías principales:

- **Modelo de Datos:** Se mantuvo la estructura jerárquica de los activos inmobiliarios... Paralelamente, el modelo de actores se representa directamente a través de la clase `Comprador`.
- **Capa de Servicio (Núcleo de Negocio):** El concepto de un núcleo de negocio se implementó en la clase `GestorInmobiliarioService`. Esta actúa como una fachada (*Facade*), sirviendo como el único punto de comunicación entre la interfaz de usuario y la lógica de negocio subyacente.
- **Capa de Persistencia:** Se añadió una capa de acceso a datos, encapsulada en la clase `DatabaseManager`, que gestiona toda la interacción con la base de datos SQLite.
- **Capa de Vista:** La interacción con el usuario se maneja a través de la clase `VisualDisplayer`, que es responsable de construir y gestionar toda la interfaz gráfica.

El siguiente diagrama UML, se muestra la arquitectura que se propuso inicialmente, la cual fue modificada con posterioridad de acuerdo a las nuevas necesidades del proyecto, lo que se detalla en profundidad en el **SIA2.1**.

SIA1.3: Encapsulamiento de Atributos

El principio de encapsulamiento fue un pilar fundamental en el diseño de todo el modelo de datos del sistema, manteniéndose desde el concepto inicial hasta la implementación final.

Todos los atributos de las clases del modelo, como `ProyectoInmobiliario`, `Edificio`, `Departamento` y `Comprador`, fueron declarados con el modificador de acceso `private`. Esta decisión de diseño protege la integridad del estado interno de los objetos, impidiendo su manipulación directa desde el exterior.

El acceso y la modificación de estos atributos se gestionan de manera controlada exclusivamente a través de métodos públicos `get` y `set`. Adicionalmente, para garantizar la inmutabilidad de datos críticos una vez que un objeto es creado, atributos como el `RUT` y el `NOMBRE` en la clase `Comprador`, o el `codigo` en la clase `Departamento`, fueron declarados como `final`. Este enfoque asegura la consistencia y la fiabilidad de los datos en toda la aplicación.

SIA1.4: Inclusión de Datos Iniciales

En la versión final del proyecto, el sistema de carga de datos iniciales (“seeding”) fue reemplazado por un robusto mecanismo de persistencia que utiliza una base de datos local **SQLite**.

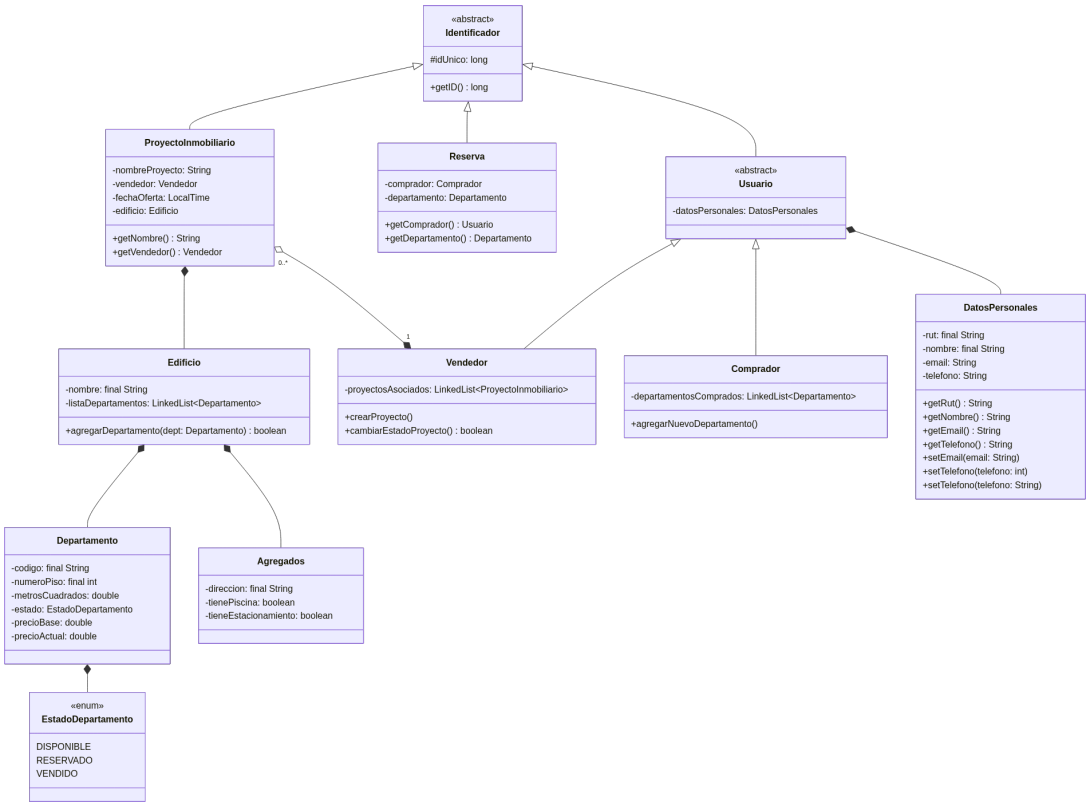

Figura 1: Diagrama de clases antiguo del sistema, integrando todas las capas.

Toda la lógica de acceso a datos está centralizada en la clase `DatabaseManager`. Esta clase es responsable de la conexión con la base de datos y de la transferencia de información entre la base de datos y la aplicación.

El sistema opera en modo *batch* y utiliza una caché en memoria para optimizar el rendimiento:

- **Carga de Datos al Iniciar:** Al arrancar la aplicación, el método `inicializar()` del `DatabaseManager` establece la conexión con el archivo de la base de datos. Inmediatamente después, el método `cargarDatos()` lee todas las tablas (Proyectos, Edificios, Departamentos) y construye los objetos correspondientes, almacenándolos en mapas Hash para un acceso rápido durante la ejecución del programa.
- **Guardado de Datos al Salir:** Cuando el usuario decide guardar y salir, el método `actualizarDatosDatabase()` se encarga de procesar todos los cambios realizados en la caché (nuevos registros, modificaciones y eliminaciones) y los escribe en la base de datos dentro de una única transacción para garantizar la consistencia de los datos.

SIA1.5: Diseño de Colecciones Anidadas

El diseño original de utilizar **colecciones anidadas** se mantuvo y se implementó fielmente en la versión final del proyecto, ya que permite un modelado natural e intuitivo de la relación jerárquica entre los activos inmobiliarios.

La estructura de datos se implementó de la siguiente manera:

- **Colección de Primer Nivel:** La capa de persistencia, a través de la clase `DatabaseManager`, utiliza un `Map<Long, ProyectoInmobiliario>` como caché principal. Esta estructura de datos permite un acceso eficiente por ID a cualquier proyecto del sistema.
- **Colecciones Anidadas:** Cada objeto `ProyectoInmobiliario` contiene una colección `LinkedList<Edificio>`. A su vez, cada objeto `Edificio` contiene su propia colección anidada, una `LinkedList<Departamento>`.

Esta arquitectura de datos no es solo una decisión interna; es la base sobre la cual opera toda la interfaz gráfica. Las ventanas de "Ver Proyecto" y "Modificar Proyecto", por ejemplo, utilizan esta estructura para permitir al usuario navegar de manera lógica: al seleccionar un proyecto en una tabla, el sistema accede a su colección de edificios para poblar la siguiente tabla, y así sucesivamente con los departamentos.

SIA1.6: Sobrecarga de Métodos

Se empleó la **sobrecarga de métodos** como una técnica clave para proporcionar una interfaz de programación (API) más flexible e intuitiva, cumpliendo con las buenas prácticas de diseño de software al permitir que un mismo método realice tareas similares con diferentes tipos de entrada.

Un ejemplo representativo de esta técnica se encuentra en la clase `VisualDisplayer`, responsable de gestionar la interfaz gráfica. Esta clase ofrece dos versiones del método `cargarDepartamentosEnTabla.txt` para poblar la tabla de departamentos, cada una adaptada a un contexto de uso diferente:

- `cargarDepartamentosEnTabla(Edificio edificio)`: Esta versión recibe un objeto `Edificio` completo, extrae internamente su lista de departamentos y los muestra en la tabla. Es ideal para cuando el usuario selecciona un edificio en la interfaz para ver sus unidades.
- `cargarDepartamentosEnTabla(List<Departamento>listaDepartamentos)`: Esta versión recibe directamente una lista de departamentos, que puede provenir, por ejemplo, de un resultado de búsqueda filtrada. Permite reutilizar la misma lógica de llenado de tabla con una fuente de datos ya procesada.

Esta dualidad demuestra un uso práctico y efectivo de la sobrecarga para aumentar la flexibilidad del código y reutilizar la lógica de presentación de datos en diferentes partes de la aplicación.

SIA1.7: Utilización de Clases Mapa (JCF)

El concepto original del avance planteaba el uso de la estructura de datos `Map` en la clase `GestorInmobiliarioService` para almacenar todas las entidades principales. Si bien la idea de utilizar un `Map` por su eficiencia se mantuvo, en la implementación final esta responsabilidad se especializó y se trasladó a la capa de persistencia para un diseño más robusto.

En la versión actual del proyecto, la clase `DatabaseManager` es la que utiliza `Map` del Java Collections Framework (JCF) para implementar un sistema de **caché en memoria**. Específicamente, se utilizan los mapas `cacheProyectos` y `cacheEdificios`.

Esta arquitectura representa una mejora significativa sobre el diseño inicial:

- **Rendimiento:** Al cargar los datos de la base de datos en estos mapas al iniciar la aplicación, el sistema puede acceder a cualquier proyecto o edificio casi instantáneamente a través de su ID, con una complejidad de tiempo promedio de **O(1)**.
- **Experiencia de Usuario:** Este acceso de alto rendimiento es crucial para que la interfaz gráfica sea fluida y receptiva, evitando la latencia que supondría consultar la base de datos para cada pequeña interacción del usuario.

De este modo, se conservó la ventaja de rendimiento del `Map`, pero integrándola en una capa de persistencia más completa y escalable.

SIA1.8: Menú del Sistema para la Colección Anidada

La implementación inicial del proyecto, visible en la clase `ConsoleDisplayer`, cumplía con este requisito mediante un **menú de consola interactivo**. Este permitía al usuario navegar textualmente a través de las colecciones anidadas, listando proyectos y luego sus edificios y departamentos.

En la versión final del sistema, este concepto fue completamente rediseñado y mejorado, reemplazando el menú de texto por una **navegación visual e intuitiva** dentro de la interfaz gráfica.

La funcionalidad de navegar y manipular las colecciones anidadas ahora se implementa de la siguiente manera:

- La ventana principal muestra la lista de `ProyectosInmobiliarios` en una `JTable`.
- Al seleccionar un proyecto y acceder a las ventanas de "Ver" o "Modificar", la interfaz presenta dos tablas interconectadas: una para los `Edificios` del proyecto seleccionado y otra para los `Departamentos`.
- La selección de una fila en la tabla de edificios filtra y actualiza automáticamente el contenido de la tabla de departamentos.

Este enfoque gráfico no solo cumple con el requisito original, sino que lo hace de una manera mucho más eficiente y amigable. En la siguiente figura se puede observar la ventana de visualización que implementa esta navegación:

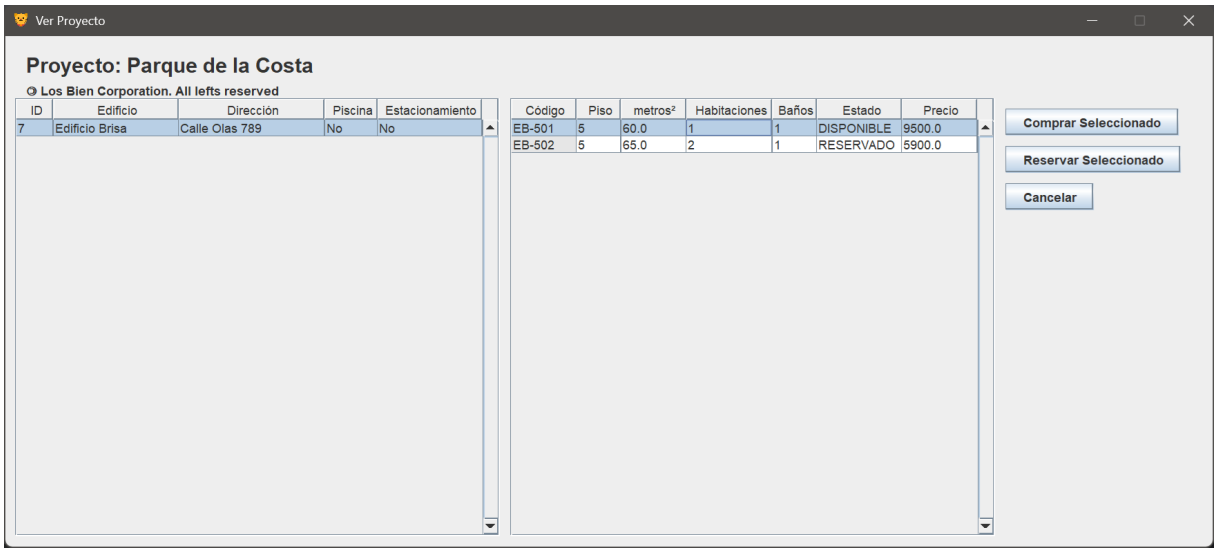

Figura 2: Ventana de visualización de un proyecto, donde el usuario puede navegar entre los edificios y sus departamentos.

SIA1.9: Evolución de la Interfaz de Usuario

La primera versión funcional del sistema, de acuerdo con los requisitos del avance, fue desarrollada con una interfaz de usuario operada íntegramente por **consola de texto**. Esta implementación inicial, encapsulada en la clase `ConsoleDisplayer`, permitió validar toda la lógica de negocio y el modelo de datos sin la complejidad añadida de una capa gráfica.

Para la entrega final, y en cumplimiento con el requisito **SIA2.3**, esta interfaz de consola fue completamente **reemplazada por una Interfaz Gráfica de Usuario (GUI)** desarrollada con la biblioteca **Java SWING**.

Esta transición representa una mejora fundamental en el proyecto, ya que la nueva interfaz gestionada por la clase `VisualDisplayer` ofrece mayor usabilidad, mejor visualización de datos e interacción eficiente. De este modo, el proyecto evolucionó de un prototipo funcional basado en texto a una aplicación de escritorio completa y robusta.

Figura 3: Ventana principal del Gestor Inmobiliario, mostrando la tabla de proyectos y las opciones de gestión.

SIA1.10 Utilización de GitHub

Desde el inicio y durante todo el ciclo de vida del desarrollo, el proyecto se gestionó bajo un sistema de control de versiones, utilizando **Git**. El código fuente se encuentra alojado en un repositorio centralizado en **GitHub**, lo que facilitó el trabajo colaborativo y un seguimiento riguroso de los cambios.

DETALLES DE LA IMPLEMENTACIÓN FINAL

A continuación, se presentan los detalles técnicos de la implementación final del proyecto, consolidando y evolucionando los conceptos presentados en el informe de avance.

SIA2.1: Diseño de Diagrama de Clases UML

Partiendo de los diagramas conceptuales presentados en el avance (sección **SIA1.2**), el diseño final del sistema se consolidó en una única arquitectura cohesiva. El siguiente diagrama UML representa la estructura final y completa del proyecto, integrando las clases del modelo, la vista (`VisualDisplayer`), el servicio (`GestorInmobiliarioService`) y la capa de persistencia (`DatabaseManager`), que en el avance se modelaron de forma separada.

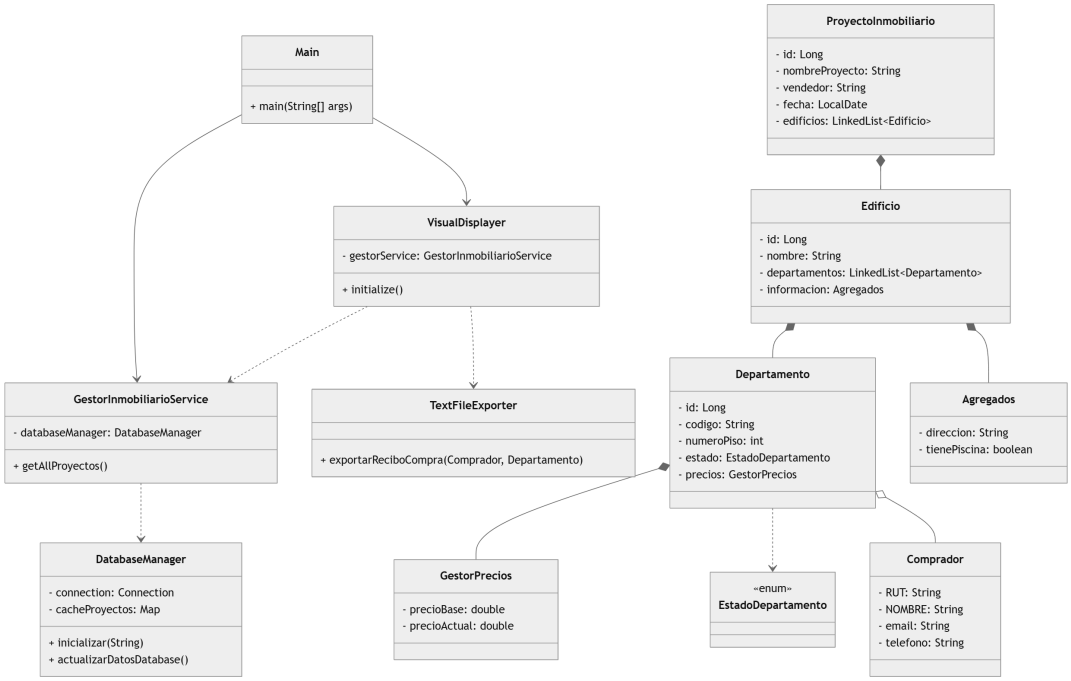

Figura 4: Diagrama de clases final del sistema, integrando todas las capas.

SIA2.2: Persistencia de Datos

Este componente reemplaza la funcionalidad de “Inclusión de Datos Iniciales” descrita en el punto **SIA1.4** del avance. Inicialmente se planteó un sistema de “seeding” para pruebas. Sin embargo, durante el desarrollo se identificó la necesidad de una solución robusta, por lo que se implementó un sistema de persistencia utilizando una base de datos local **SQLite**. Toda la lógica de acceso a datos está encapsulada en la clase `DatabaseManager`, que opera en modo *batch*, cargando los datos al iniciar y guardando los cambios al salir.

SIA2.3: Implementación de Interfaces Gráficas (SWING)

Esta implementación reemplaza por completo el enfoque de “Implementación en Consola” descrito en el punto **SIA1.9** del avance. Toda la interacción con el usuario se realiza a través de

una Interfaz Gráfica de Usuario (GUI) desarrollada íntegramente con la biblioteca **Java SWING**. La clase principal de la vista, `VisualDisplayer`, es la responsable de construir y gestionar todas las ventanas y componentes, ofreciendo una experiencia de usuario mucho más intuitiva.

SIA2.4: Edición y Eliminación (Colección Anidada)

Esta funcionalidad es la evolución directa del "Menú del Sistema para la Colección Anidada" (punto **SIA1.8**). En la versión final, la ventana de "Modificar Proyecto" se convierte en el centro de gestión integral de un proyecto.

Esta interfaz no solo permite editar los datos del proyecto en sí, sino que también facilita la manipulación de sus colecciones anidadas. Como se aprecia en la siguiente figura, mediante dos tablas interconectadas, el usuario puede agregar, modificar y eliminar tanto los `Edificios` como los `Departamentos` asociados a ese proyecto.

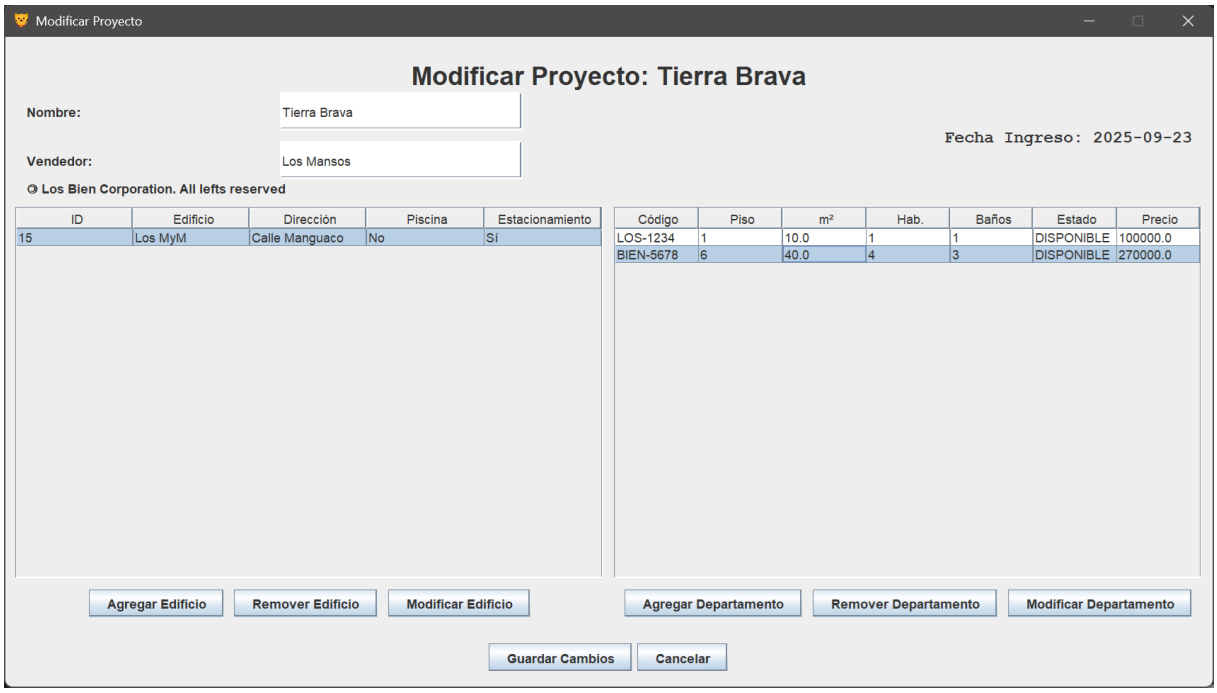

Figura 5: Interfaz para la modificación de un proyecto, mostrando las tablas interconectadas para la gestión de edificios y departamentos.

SIA2.5: Funcionalidad Propia (Filtrado por Criterio)

En esta versión se consolidó la "Búsqueda y Reserva" propuesta en el punto **SIA1.1** del avance. Se desarrolló una ventana de búsqueda avanzada que permite la selección de un subconjunto de departamentos basado en múltiples criterios. Los criterios se encapsulan en un objeto `FiltroBusqueda` que se utiliza para construir una consulta SQL dinámica y eficiente.

Esta funcionalidad representa una herramienta de gran utilidad para el negocio, permitiendo a los usuarios realizar consultas complejas para encontrar propiedades que se ajusten a sus necesidades específicas. Los detalles de las opciones de filtrado disponibles se profundizarán en la sección **SIA2.13**.

SIA2.6: Modularización del Código Fuente

Basado en el "Modelado del Núcleo de Negocio" (**SIA1.2**), la arquitectura final del proyecto se organizó en paquetes lógicos con responsabilidades bien definidas para asegurar una alta cohesión y bajo acoplamiento. La estructura incluye los paquetes `modelo`, `gestion`, `servicios`, `validaciones` y `excepciones`, siguiendo las buenas prácticas de la ingeniería de software.

SIA2.7: Sobrescritura de Métodos

Como una mejora sobre los principios de POO aplicados en el avance (que se centraban en la sobrecarga, **SIA1.6**), se implementó la **sobrescritura de métodos** para modificar el comportamiento de componentes heredados de la biblioteca Java SWING y adaptarlos a las necesidades de la aplicación. Esto se realizó a través de clases anónimas internas.

Un primer ejemplo se encuentra en la personalización de las tablas. Para garantizar la integridad de los datos y evitar que el usuario modifique las celdas, se diseñó una clase anónima que hereda de `DefaultTableModel` y sobrescribe su método `isCellEditable()`:

```
// ... new DefaultTableModel() {
    @Override
    public boolean isCellEditable(int row, int column) {
        return false; // Inhabilita la edición de celdas
    }
// };
```

Un segundo ejemplo se utilizó para mejorar la legibilidad de la interfaz. En este caso, se codificó otra clase anónima que hereda de `DefaultTableCellRenderer` para sobrescribir el método `getTableCellRendererComponent()` y así alterar el color de fondo de ciertas celdas, mejorando la experiencia visual del usuario.

SIA2.8: Implementación del Manejo de Excepciones (Try-Catch)

El manejo de errores potenciales se implementó a través de bloques `try-catch`, permitiendo capturar excepciones específicas sin interrumpir la ejecución del programa.

Las excepciones personalizadas son lanzadas por los métodos de las clases del paquete `validaciones`, como por ejemplo el método `validarRut()` de la clase `ValidadorRut`. En la capa de la vista, dentro de los `ActionListeners` de la clase `VisualDisplayer`, las llamadas a estos métodos de validación se envuelven en bloques `try`.

Si uno de estos métodos lanza una excepción, el bloque `catch` correspondiente la captura. Este bloque está diseñado para capturar un tipo de excepción específico, como `RutInvalidoException`. Al hacerlo, se extrae el mensaje de la excepción y se le presenta al usuario de forma controlada a través de un `JOptionPane`, evitando así que la aplicación se cierre inesperadamente y proporcionando retroalimentación útil. Este patrón se utiliza de manera consistente para todos los procesos de validación de datos en la interfaz.

SIA2.9: Creación de Excepciones Personalizadas

Para manejar los errores específicos de la lógica de negocio del sistema, se crearon múltiples clases de excepción personalizadas, superando el requisito mínimo de dos. Estas clases, como `RutIvalidoException`, `EmailInvalidoException` y `NombreInvalidoException`, se organizaron en un paquete dedicado (`excepciones`) para mantener la modularidad del código.

Cada una de estas clases hereda de la clase base `java.lang.Exception` y posee un constructor que recibe un `String` como parámetro. Este mensaje se pasa al constructor de la clase padre a través de la llamada `super(mensaje)`, permitiendo que cada excepción contenga una descripción clara y específica del error ocurrido.

SIA2.10: Reporte en Archivo TXT

Como una extensión a las funcionalidades básicas, se incorporó la capacidad de generar un **recibo de compra** en formato `.txt` al momento de realizar una transacción.

Esta funcionalidad está encapsulada en la clase de utilidad `TextFileExporter`, la cual contiene un método estático `exportarReciboCompra(comprador, departamento)`. Al ser invocado, este método genera un archivo de texto con un nombre único que incluye el RUT del comprador y una marca de tiempo para evitar sobreescrituras (ej: `recibo_11222333-4_20250927_182157.txt`).

El recibo generado contiene un resumen detallado y formateado de la transacción, incluyendo los datos del comprador, los detalles de la propiedad y el precio final pagado, como se ilustra en el siguiente ejemplo:

```
=======================================
      RECIBO DE COMPRA INMOBILIARIA
=======================================

Fecha de Compra: 27-09-2025 19:58:56

--- Datos del Comprador ---
Nombre: 20808082-2
RUT: Los Bien Incorporated
Email: example@gmail.cl
Tel fono: 912345678

--- Detalles de la Propiedad ---
Proyecto: Prueba
Edificio: cochayuyo
  - Direccion: jeje
  - Piscina: Si
  - Estacionamiento: No

--- Detalles del Departamento ---
Codigo: FDSC
Piso: 1
Habitaciones: 1
Banos: 1

---------------------------------------
PRECIO FINAL PAGADO: $1040,00
---------------------------------------

Gracias por su compra.
(c) Los Bien Corporation
```

Figura 6: Ejemplo del contenido de un recibo de compra generado en formato .txt.

SIA2.11: Continuidad en la Utilización de GitHub

Cumpliendo con el requisito de continuidad, se mantuvo una actividad constante en el repositorio de GitHub, realizando múltiples commits adicionales desde la entrega del avance. Esto permitió documentar la implementación de todas las nuevas funcionalidades, como la interfaz gráfica, la capa de persistencia y las validaciones, dejando un historial completo y trazable del desarrollo del software. El enlace al repositorio es: https://github.com/honaisu/GestorInmobiliario.

SIA2.12: Gestión de la Primera Colección

La gestión completa (CRUD) de los elementos de la primera colección, es decir, los `ProyectoInmobiliario`, se implementa a través de la interfaz gráfica, sirviendo como punto de partida para la administración de toda la jerarquía de datos.

- **Creación y Modificación:** La ventana principal de la aplicación centraliza estas acciones. El botón “Registrar un Proyecto” abre una nueva ventana dedicada a la creación de un nuevo proyecto, como se muestra en la siguiente figura. Por su parte, el botón “Modificar

un Proyecto” carga los datos de un proyecto existente en una ventana de edición.

Figura 7: Formulario para el registro de un nuevo proyecto inmobiliario.

- **Eliminación:** Por una decisión de diseño enfocada en la seguridad y en prevenir acciones accidentales, la funcionalidad para eliminar un proyecto no se encuentra como un botón directo en la ventana principal. En su lugar, esta operación se ha ubicado dentro de la ventana de “Modificar Proyecto”. De esta manera, el usuario primero debe entrar al contexto de edición de un proyecto específico antes de poder proceder con su eliminación, lo que añade un paso de confirmación implícito y reduce el riesgo de borrados no deseados.

SIA2.13: Funcionalidad de Búsqueda

La funcionalidad de búsqueda de elementos en uno o más niveles se implementa a través de la ventana de filtrado (descrita en **SIA2.5**). Esta interfaz, que se muestra en la siguiente figura, permite al usuario localizar departamentos específicos, mostrando información tanto del departamento como de su edificio y proyecto padre.

Para realizar una búsqueda detallada, la interfaz ofrece una combinación de múltiples criterios de filtrado que el usuario puede especificar. Las opciones disponibles son:

- **Rango de Precios:** Permite definir un precio mínimo y máximo.
- **Número de Habitaciones:** Seleccionar una cantidad mínima de habitaciones.
- **Número de Baños:** Seleccionar una cantidad mínima de baños.
- **Estado:** Filtrar por el estado actual del departamento (Disponible, Reservado o Vendido).
- **Dirección:** Búsqueda por texto que coincida parcial o totalmente con la dirección del edificio.

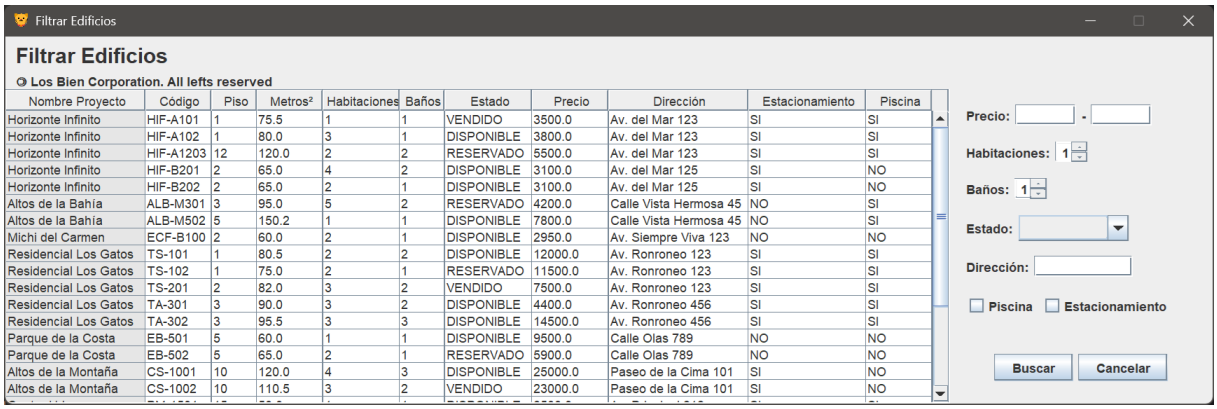

Figura 8: Ventana de búsqueda avanzada con sus criterios de filtrado y la tabla de resultados.

- **Características Adicionales:** Opciones para incluir solo propiedades que cuenten con piscina y/o estacionamiento.

Esta combinación de filtros permite al usuario realizar consultas complejas y precisas para encontrar las propiedades que se ajusten exactamente a sus necesidades.

CONCLUSIONES

El desarrollo del proyecto Gestor Inmobiliario ha culminado con la implementación exitosa de una aplicación de escritorio completamente funcional, que cumple con todos los requerimientos solicitados. Se logró transitar desde un diseño conceptual inicial a un sistema robusto que integra una interfaz gráfica de usuario, una capa de lógica de negocio y un sistema de persistencia de datos.

Los principales logros del proyecto incluyen:

- La implementación de una arquitectura modular y escalable, que separa claramente las responsabilidades del modelo, la vista y el acceso a datos.
- El desarrollo de una interfaz gráfica de usuario intuitiva con Java SWING, que reemplazó por completo la interacción por consola, mejorando significativamente la experiencia de usuario.
- La implementación exitosa de todas las funcionalidades de negocio clave, incluyendo la gestión completa (CRUD) de proyectos y departamentos, y el desarrollo de un modelo de precios dinámico.
- La integración de una base de datos SQLite para la persistencia de datos, asegurando que la información se mantenga de manera consistente entre sesiones.

El mayor desafío técnico fue la correcta sincronización entre la interfaz gráfica, la caché de datos en memoria y la base de datos, lo cual se resolvió con un sistema de guardado por lotes (batch). Este proyecto no solo permitió aplicar los conocimientos teóricos del curso en un caso práctico, sino que también proporcionó una valiosa experiencia en el ciclo de vida completo del desarrollo de software.

Como futura mejora, el sistema podría extenderse para incluir un sistema de roles de usuario más complejo o migrar a una arquitectura cliente-servidor.