Planificación y Gestión del proyecto Alquiler de Viviendas

Realizado por:

- Daniel Salas Conejo
- Saul Corbelle Hernandez
- Laura Rodriguez Velasco

INDICE

1-Diseños del Sistema

1.1 Requisitos

1.1.1 Requisitos Funcionales

Los requisitos funcionales se pueden clasificar en cuatro grupos, requisitos de usuario, que son los que definen las operaciones que se deben realizar con los usuarios, de propiedades, que son aquellas operaciones que puede realizase hacia las propiedades, de reservas, que son las operaciones que se realizan para la gestión de las reservas de propiedades, y de pago, que definen las operaciones de pago de las reservas.

1.1.1.1 Requisitos de Usuario

Los requisitos de usuario son los siguientes:

- ☐ **RF-01:** El sistema debe permitir el registro de nuevos usuarios como inquilinos o propietarios.
- ☐ **RF-02**: El sistema debe permitir el inicio de sesión para ambos tipos de usuarios.
- ☐ **RF-03**: El sistema debe permitir que inquilinos y propietarios gestionen su perfil (datos personales, medios de pago, etc.).
- ☐ **RF-04:** Un usuario no registrado debe poder buscar alojamientos sin necesidad de iniciar sesión.

1.1.1.2 Requisitos de Propiedades

Los requisitos sobre las propiedades son los siguientes:

- ☐ **RF-05**: El propietario, una vez autenticado, puede registrar una propiedad (vivienda completa o habitación individual).
- ☐ **RF-06:** El propietario puede definir si la propiedad admite reserva inmediata o si requiere confirmación manual.
- ☐ **RF-07**: El propietario puede confirmar o rechazar solicitudes de reserva recibidas.
- ☐ **RF-08**: El propietario puede confirmar la disponibilidad de una propiedad.
- ☐ **RF-09**: El sistema debe notificar al propietario cuando recibe una solicitud de reserva.

1.1.1.3 Requisitos de Reservas

Los requisitos de las reservas son las siguientes.

- ☐ **RF-10**: El sistema debe ofrecer **filtros avanzados** de búsqueda, reserva inmediata, comodidades y política de cancelación.
- ☐ **RF-11**: El inquilino autenticado puede añadir alojamientos a su lista de deseos.
- ☐ **RF-12**: El inquilino autenticado puede **realizar una reserva** de un alojamiento.

- ☐ **RF-13**: Si el alojamiento permite reserva inmediata, el sistema debe permitir completar la reserva directamente (con pago incluido).
- ☐ **RF-14**: Si el alojamiento no permite reserva inmediata, el sistema debe enviar una solicitud de reserva al propietario.
- ☐ **RF-15**: El sistema debe notificar al inquilino si su reserva es confirmada.
- ☐ **RF-16**: En caso de denegación, el sistema debe reembolsar automáticamente el pago al inquilino.
- ☐ **RF-17:** Las reservas solo podrán ser canceladas hasta dos días antes de la fecha de inicio de la reserva
- ☐ **RF-18:** Si una reserva abarca distintas disponibilidades se tomaerá los valores peores de esta (politica cancelacion, etc.)

1.1.1.4 Requisitos de Pago

Los requisitos del método de pago son:

- ☐ **RF-18**: El sistema debe permitir realizar pagos mediante tarjeta de crédito, tarjeta de débito o PayPal.

1.1.2 Requisitos no Funcionales

Los requisitos no funcionales se clasifican en diferentes grupos según el tipo de restricción o característica que aportan al sistema: de rendimiento, seguridad, usabilidad, fiabilidad, mantenibilidad, portabilidad, escalabilidad y compatibilidad.

1.1.2.1 Requisitos de Rendimiento

RNF-01: El sistema debe ser capaz de atender al menos 100 usuarios concurrentes sin afectar el rendimiento.

RNF-02: El tiempo de respuesta ante una búsqueda no debe superar los 3 segundos en condiciones normales de carga.

RNF-03: El proceso de reserva y pago debe completarse en menos de 10 segundos tras la validación.

1.1.2.2 Requisitos de Seguridad

RNF-04: Todas las comunicaciones entre el cliente y el servidor deben realizarse bajo el protocolo HTTPS.

RNF-05: Las contraseñas de los usuarios deben almacenarse cifradas mediante algoritmos seguros.

RNF-06: El sistema no almacenará información de pago sensible; las transacciones se realizarán a través de servicios externos certificados.

1.1.2.3 Requisitos de Usabilidad

RNF-08: La interfaz del sistema debe ser clara, intuitiva y coherente, permitiendo una fácil navegación entre las secciones.

RNF-09: El sistema debe ser accesible desde dispositivos móviles, tablets y ordenadores mediante un diseño responsivo.

RNF-10: Las acciones críticas (como eliminar una propiedad o cancelar una reserva) deben requerir confirmación del usuario.

1.1.2.5 Requisitos de Mantenibilidad

RNF-14: El sistema debe desarrollarse con una arquitectura en 3 capas(capas de presentación, negocio y persistencia).

RNF-15: La base de datos debe diseñarse normalizada, facilitando su mantenimiento y posibles ampliaciones

1.1.2.6 Requisitos de Escalabilidad

RNF-16: El sistema debe permitir la incorporación de nuevas funcionalidades sin afectar las existentes.

RNF-17: La arquitectura debe soportar un aumento en el número de usuarios y propiedades sin comprometer el rendimiento.

1.2-Arquitectura en 3 capas

1.2.1-Capa de Presentación (controladores webs)

La Capa de Presentación es el punto de entrada de la aplicación.
Su función principal es gestionar la interacción entre el usuario y el sistema, recibiendo las solicitudes HTTP y retornando las vistas adecuadas.

1.2.2-Capa de Negocio (Lógica de la aplicación)

La Capa de Negocio representa el núcleo funcional del sistema, donde se concentra toda la lógica y las reglas del negocio.
Dentro de esta capa, se encuentran dos carpetas principales:

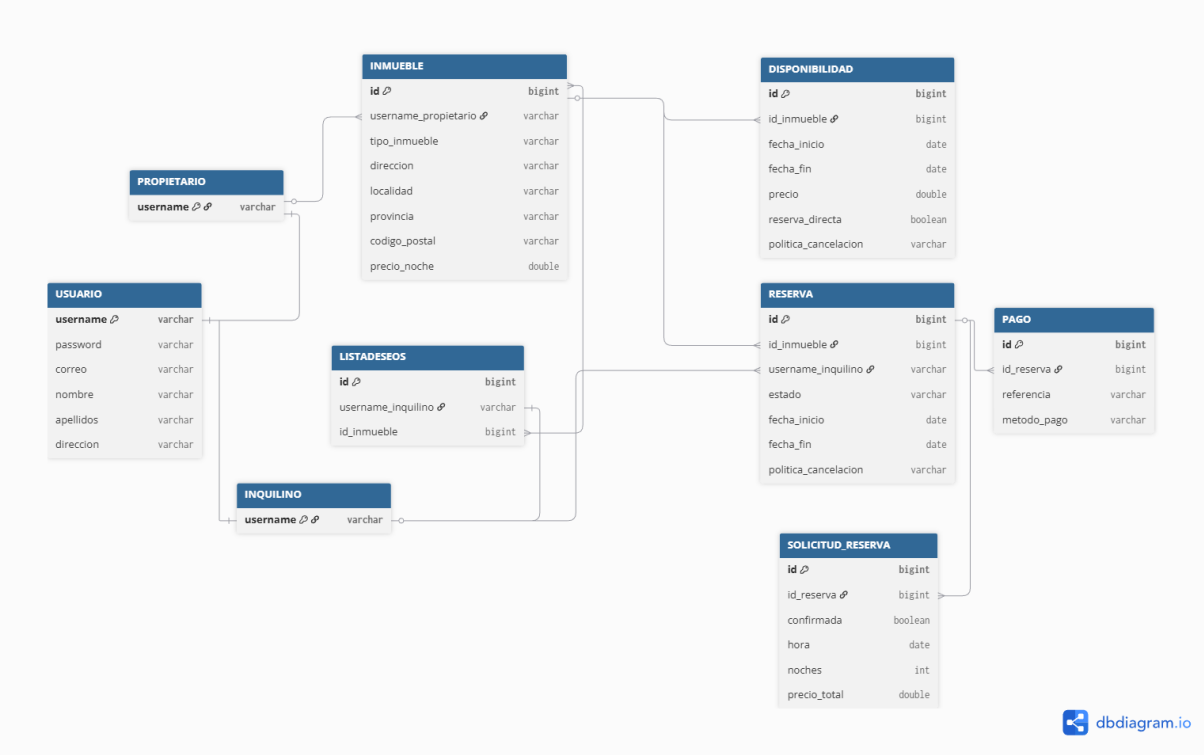
- Entidades: donde se definen las clases que representan los objetos del dominio del sistema (por ejemplo, `Usuario`, `Producto`, `Pedido`, etc.). Estas clases suelen estar anotadas con `@Entity` y mapeadas a tablas de la base de datos.
- Controladores o Gestores de Negocio: donde se encuentran las clases encargadas de aplicar la lógica del sistema y coordinar las operaciones con la capa de persistencia

1.2.3-Capa de Persistencia (Datos)

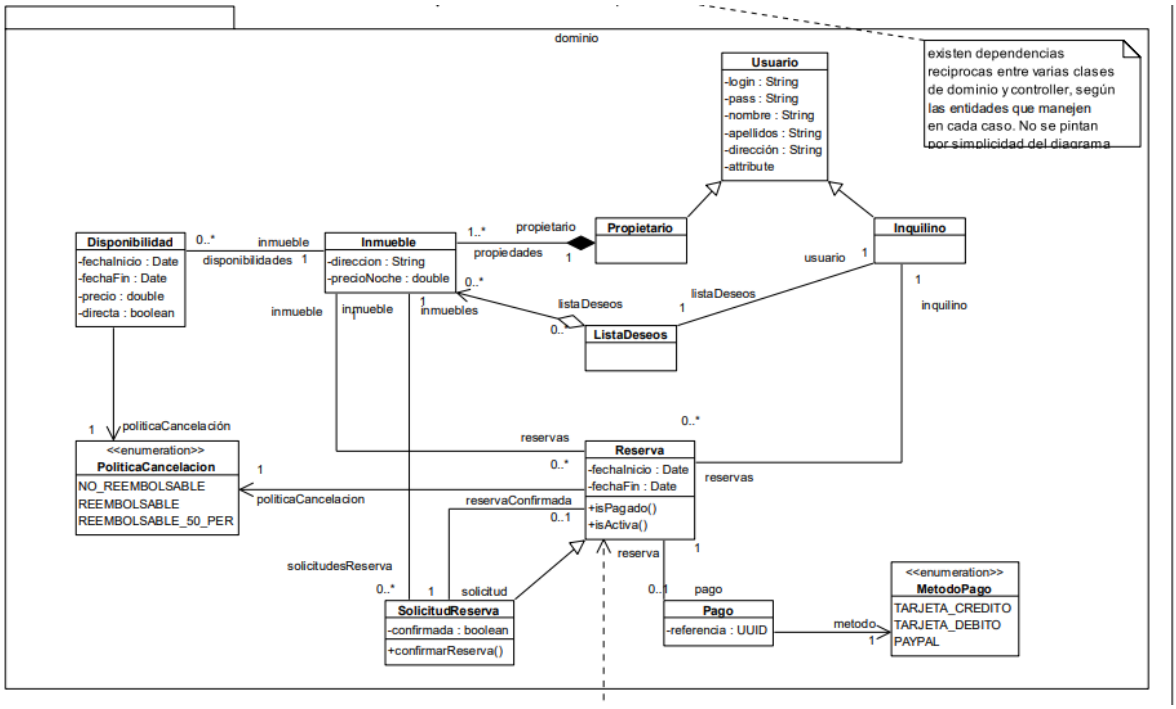
La Capa de Persistencia es responsable de gestionar el acceso y almacenamiento de los datos.
Su función es abstraer la comunicación con la base de datos, de modo que las demás capas no necesiten conocer detalles sobre el lenguaje SQL o las conexiones.

En este proyecto, la persistencia se implementa mediante el uso de Spring Data JPA, que permite interactuar con las entidades del modelo de datos a través de interfaces `@Repository`.

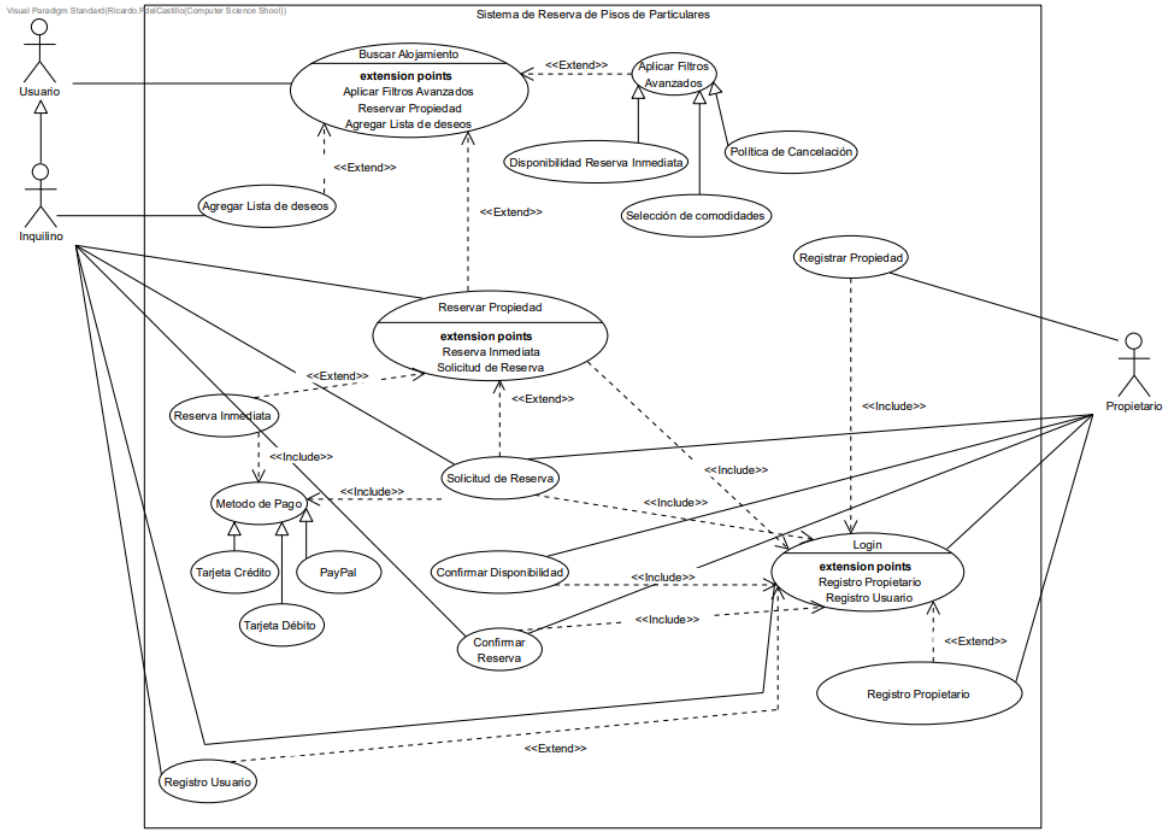
1.3-Diseño de la Base de Datos



1.4-Diagrama de Clases



1.5-Diagrama de Casos de Uso



2-Metodologia de Desarrollo

Para el desarrollo del proyecto se ha optado por utilizar la metodología ágil Kanban, debido a su enfoque flexible y visual en la gestión del trabajo. Kanban permite **organizar y controlar las tareas del equipo** mediante un tablero dividido en columnas que representan los distintos estados del flujo de trabajo

2.1-Estructura del flujo de trabajo

El tablero Kanban se divide en las siguientes columnas:

- **Backlog** → tareas pendientes.
- **To Do** → tareas pendientes de iniciar en el sprint actual.
- **In Progress** → tareas en desarrollo.
- **Testing** → tareas completadas pendientes de revisión o pruebas.
- **Done** → tareas finalizadas e integradas en el repositorio principal.

Cada tarea incluye una breve descripción, y una etiqueta que identifica su categoría (backend, frontend, base de datos, documentación, etc.)

3-Metodología de Gestión de Proyectos

Para la gestión del proyecto se ha adoptado la **metodología ágil Scrum**, debido a su enfoque iterativo, colaborativo y orientado a la mejora continua. Esta metodología permite adaptarse a cambios de manera rápida y mantener una comunicación constante entre los miembros del equipo.

3.1-Roles del equipo

Cada rol clave del equipo tiene un responsable asignado, pero la toma de decisiones se realiza de manera colaborativa entre todos los miembros para garantizar una visión integral del proyecto.

3.1.1 Product Owner

Responsable: Daniel Salas

Funciones principales:

- Definir y comunicar la visión general del proyecto.
- Priorizar las funcionalidades y los objetivos del sistema según su valor para el negocio.
- Revisar y validar las historias de usuario antes de considerarlas completadas.
- Actuar como enlace entre el equipo de desarrollo y los stakeholders, recogiendo feedback constante.

3.2.1 Scrum Master

Responsable: Saul Corbelle

Funciones principales:

- Facilitar la comunicación y colaboración entre los miembros del equipo.
- Garantizar que el equipo siga las buenas prácticas de Scrum y Kanban.
- Identificar obstáculos o bloqueos que puedan afectar el progreso y ayudar a resolverlos.
- Supervisar el flujo de trabajo y la gestión del tablero Kanban para mantener un ritmo equilibrado.

3.2-Planificacion de Sprints

El desarrollo se divide en **iteraciones quincenales** que agrupan tareas afines.
Cada sprint representa una evolución del producto, avanzando desde la base del sistema hasta la versión final.

Sprint	Contenidos principales
Sprint 1 – Inicialización del proyecto	Configuración del entorno de trabajo, creación del repositorio GitHub, configuración de Maven y Spring Boot, definición de estructura de paquetes, diseño inicial de la base de datos Derby.
Sprint 2 – Entidades base y Login	Implementación de entidades `Usuario`, `Inquilino` y `Propietario`, creación de DAOs y servicios, sistema de autenticación y registro.
Sprint 3 – Gestión de propiedades	Desarrollo de la funcionalidad para que los propietarios puedan dar de alta, modificar y eliminar propiedades; conexión con la base de datos.
Sprint 4 – Búsqueda y reservas	Implementación de la búsqueda de alojamientos con filtros; sistema de reservas con validaciones y control de disponibilidad.
Sprint 5 – Interfaz de usuario	Creación de vistas con Thymeleaf, navegación entre páginas, formularios, plantillas y estilos.
Sprint 6 – Integración, pruebas y documentación final	Pruebas integradas del sistema, revisión del código, corrección de errores, generación de documentación técnica y entrega final del proyecto.

4-Gestion de la configuración

4.1 Control de Versiones

Para el manejo del código fuente se ha utilizado Git (https://git-scm.com/downloads), una herramienta de control de versiones distribuido que permite registrar el historial de cambios, trabajar en diferentes ramas de desarrollo y facilitar la colaboración entre los integrantes del equipo.

El repositorio del proyecto se encuentra alojado en GitHub (https://github.com/Danielol04/ISO2-LAB.git), lo que permite acceder de manera remota, compartir el código, gestionar incidencias (issues), realizar revisiones de código mediante *pull requests* y mantener un flujo de integración continua.

Gracias a GitHub, se garantiza la trazabilidad de todas las versiones del sistema y se evita la pérdida de información ante posibles errores o conflictos durante el desarrollo.

4.2 Entornos de Desarrollo

Para la implementación y despliegue de los diferentes módulos del sistema se han empleado dos entornos principales:

- **Eclipse IDE** (https://www.eclipse.org/downloads/packages/): utilizado principalmente para el desarrollo de componentes Java del sistema, integrando de forma nativa el uso de **Maven** y proporcionando herramientas de depuración, compilación y empaquetado del proyecto.
- **Visual Studio Code** (https://code.visualstudio.com/download): empleado como entorno complementario para la edición de código, scripts y archivos de configuración. Su flexibilidad y extensibilidad mediante plugins facilita el trabajo con diferentes lenguajes y tecnologías del proyecto.

4.3 Gestión de Datos

Como herramienta de administración de bases de datos se ha utilizado **DBeaver** (https://dbeaver.io/download/), un cliente universal que permite la conexión y gestión de distintos motores de bases de datos. DBeaver facilita la visualización de tablas, ejecución de consultas SQL y verificación de la integridad de los datos empleados durante las pruebas del sistema.

4.4 Gestión de Dependencias

Para la gestión de dependencias y la construcción del proyecto se ha empleado **Maven** (https://maven.apache.org/), una herramienta de automatización de compilaciones y empaquetado ampliamente utilizada en el entorno Java. Maven permite definir las bibliotecas y componentes requeridos en un archivo de configuración centralizado (*pom.xml*), garantizando la correcta instalación y compatibilidad de las versiones necesarias para la ejecución del sistema.
Además, Maven facilita la integración con repositorios remotos y contribuye a la reproducibilidad del entorno de desarrollo en distintos equipos.

4.5 Control y Mantenimiento de la Configuración

El proceso de gestión de configuración se complementa con prácticas de mantenimiento continuo, como la documentación de versiones, la verificación de dependencias actualizadas y la estandarización de los entornos de desarrollo. Estas medidas aseguran que el sistema pueda ser desplegado y mantenido de manera eficiente a lo largo del tiempo, reduciendo la posibilidad de errores y facilitando futuras ampliaciones o correcciones.

4.6 Entorno de Ejecución: Java 21

El sistema ha sido desarrollado y probado utilizando Java 21 (https://www.oracle.com/es/java/technologies/downloads/), la versión más reciente del lenguaje y entorno de ejecución de Java. Esta versión incorpora mejoras en

rendimiento, seguridad y soporte de nuevas características del lenguaje, lo que permite aprovechar las ventajas de un entorno moderno, estable y compatible con las herramientas de desarrollo empleadas.
El uso de Java 21 garantiza la compatibilidad con Maven, Eclipse y los demás componentes del proyecto, asegurando así un entorno unificado y actualizado para todo el ciclo de vida del software

5- Plan de Gestión de Pruebas

5.1. Objetivo del plan

El objetivo de este plan es definir la estrategia, alcance, técnicas y criterios necesarios para garantizar la calidad del sistema mediante la ejecución de pruebas unitarias y funcionales sobre todas las capas relevantes del proyecto:

- Presentación (ventanas)
- Negocio
    - Controladores
    - Entidades
- Persistencia (DAOs y GestorBD)

5.2. Alcance de las pruebas

5.2.1. Capa de Presentación

Incluye:

- Validación de entradas de usuario
- Manejo de errores (NumberFormatException)
- Comportamiento básico de los componentes
- Interacción con controladores simulados mediante mocks

5.2.2. Capa de Negocio

- **Controladores:**

  Incluye:

  - Validación de permisos
  - Validación de reglas de negocio

- Procesamiento de datos y lógica interna

- **Entidades**

  Incluye:

  - Integridad básica de los objetos
  - Validación de relaciones entre entidades
  - Estados y enumeraciones
  - Comportamiento en flujos reales

5.2.3. Capa de Persistencia

Incluye:

- Generación correcta de JPQL
- Parámetros en consultas
- Manejo de NoResultException
- Persistencia simulada mediante mocks
- Comportamiento de merge, persist y remove
- Consultas con múltiples parámetros

5.3. Estrategia de Pruebas

5.3.1. Pruebas Unitarias

Se han implementado pruebas unitarias exhaustivas utilizando:

- JUnit 5 como framework principal
- Mockito para simular dependencias
- Aserciones claras (assertEquals, assertThrows, assertTrue)
- Verificación de interacciones (verify())

5.3.2 Pruebas Funcionales Ligeras

Aplicadas en la capa de presentación:

- Simulación de eventos de usuario
- Validación de entradas
- Comportamiento de la interfaz sin abrir ventanas reales

5.4 Criterios de Aceptación

Funcionales

- Cada método crítico debe tener al menos un test positivo y uno negativo.
- Todas las rutas de ejecución deben estar cubiertas.
- Las excepciones esperadas deben ser verificadas.

5.5. Casos de Prueba

En este parte se describen únicamente los casos de prueba correspondientes a las clases más relevantes del sistema, concretamente la capa de presentación (ventanas) y los controladores.

5.5.1 Ventana Presentación

Ventana Disponibilidades

Nº	Categoría	Caso de prueba	Descripción
1	Lógica interna	Generación de lista de fechas	Verifica que se genera correctamente una lista de fechas entre dos días (incluyendo ambos).
2	Sesión	Mostrar formulario sin sesión	Si el usuario no está logueado, se redirige a `/login`.
3	Limpieza automática	Disponibilidad expirada → eliminar	Si existen disponibilidades caducadas, el controlador las elimina automáticamente antes de mostrar la vista.
4	Limpieza automática	Reserva expirada/no pagada → cancelar	Si hay reservas expiradas o no pagadas, el controlador las cancela automáticamente.
5	Sesión	Crear disponibilidad sin sesión	Si el usuario no está logueado, se redirige a `/login`.
6	Validación	Crear disponibilidad → inmueble no existe	Si el inmueble no existe, se lanza `ResponseStatusException (404)`.
7	Permisos	Crear disponibilidad sin permiso	Si el usuario no es el propietario del inmueble, se lanza `ResponseStatusException (403)`.
8	Flujo principal	Crear disponibilidad correcta	Si el usuario es propietario y los datos son válidos, se registra la disponibilidad y se redirige correctamente.
9	Sesión	Eliminar disponibilidad sin sesión	Si el usuario no está logueado, se redirige a `/login`.

10	Validación	Eliminar disponibilidad → no existe	Si la disponibilidad no existe, se lanza `ResponseStatusException (404)`.
11	Permisos	Eliminar disponibilidad sin permiso	Si el usuario no es propietario del inmueble asociado, se lanza `ResponseStatusException (403)`.
12	Flujo principal	Eliminación correcta	Si el usuario es propietario, la disponibilidad se elimina y se redirige

Ventana Home

Nº	Categoría	Caso de prueba	Descripción
1	Roles de usuario	Home propietario	Si el usuario es propietario, se muestra la vista `homePropietario` con sus inmuebles y su username en el modelo.
2	Roles de usuario	Home inquilino	Si el usuario es inquilino, se muestra la vista `homeInquilino` con la lista de inmuebles y su username
3	Roles de usuario	Home público (sin rol)	Si el usuario no es propietario ni inquilino, se muestra la vista pública `home` con los inmuebles disponibles
4	Búsqueda	Buscar como propietario	Si el usuario es propietario, la búsqueda devuelve resultados y se muestra `homePropietario`
5	Búsqueda	Buscar como inquilino	Si el usuario es inquilino, la búsqueda devuelve resultados y se muestra `homeInquilino`
6	Búsqueda	Buscar como público	Si el usuario no tiene rol, la búsqueda devuelve resultados y se muestra la vista pública `home`

Ventana AltaInmueble

Nº	Categoría	Caso de prueba	Descripción
1	Sesión / Vista	Mostrar formulario de alta	Si el usuario está logueado, se muestra el formulario de alta con el modelo correcto (`inmueble`, `username`) y la vista `AltaInmuebles`
2	Flujo principal	Registro exitoso de inmueble	Si el usuario es propietario y envía datos válidos (incluyendo imagen), se registra el inmueble y se redirige a `/home`
3	Permisos	Registro fallido → usuario no propietario	Si el usuario no es propietario, el alta se bloquea y se redirige a `/login`.
4	Recursos multimedia	Mostrar foto existente	Si el inmueble tiene foto, se devuelve correctamente el contenido binario con tipo `application/octet-stream`.
5	Listado	Listar inmuebles	La ruta `/inmuebles/listar` devuelve la vista `listarInmuebles` con el atributo `inmuebles` cargado.

6	Eliminación	Eliminar inmueble	Si el usuario es propietario y la eliminación es exitosa, se devuelve JSON con `{ exito: true, id: 5}`

Ventana ListaDeseos

Nº	Categoría	Caso de prueba	Descripción
1	Favoritos / Sesión	Toggle favorito con usuario logueado	Si el usuario está logueado, el controlador alterna el estado del inmueble en la lista de deseos y devuelve `"true"`.
2	Favoritos / Sesión	Toggle favorito sin sesión	Si el usuario no está logueado, la operación devuelve `401 Unauthorized`.
3	Obtener lista	Obtener favoritos → usuario con lista	Si el usuario tiene lista de deseos, se devuelve un JSON con los IDs de los inmuebles favoritos.
4	Obtener lista	Obtener favoritos → usuario sin lista	Si el usuario no tiene lista de deseos, se devuelve un JSON vacío `[]`.
5	Obtener lista	Obtener favoritos → no logueado	Si no hay sesión, se devuelve `[]` sin error.
6	Vista lista deseos	Mostrar lista deseos → usuario con lista	Si el usuario tiene lista, se carga el modelo con `propiedades` y se muestra la vista `lista-deseos`.
7	Vista lista deseos	Mostrar lista deseos → usuario sin lista	Si el usuario no tiene lista, igualmente se muestra la vista `lista-deseos` con `propiedades` vacío.
8	Vista lista deseos / Sesión	Mostrar lista deseos → no logueado	Si no hay sesión, se redirige a `/login`.

Ventana Login

Nº	Categoría	Caso de prueba	Descripción
1	Vista	Mostrar formulario de login	La ruta `GET /login` carga correctamente la vista `login` sin errores.
2	Autenticación	Login fallido	Si las credenciales son incorrectas, se vuelve a la vista `login` y se muestra un atributo `error` en el modelo.
3	Autenticación	Login exitoso	Si las credenciales son correctas, se redirige a `/home` y se almacena el `username` en la sesión.

4	Sesión	Cerrar sesión	La ruta `GET /logout` invalida la sesión y redirige a `/login`.

Ventana Pago

Nº	Categoría	Caso de prueba	Descripción
1	Validación / Estado	Mostrar pago correcto	Si la reserva existe y no está pagada, se muestra la vista `pago` con el modelo `reserva`.
2	Validación	Mostrar pago → reserva inexistente	Si la reserva no existe, se redirige a `/home`.
3	Validación / Estado	Mostrar pago → reserva ya pagada	Si la reserva ya está pagada, se redirige a `/home`.
4	Pago directo	Confirmar pago → reserva directa	Si la reserva es directa, se marca como **ACEPTADA**, se registra el pago, se actualiza la reserva y NO se genera solicitud. Redirige a `/reserva/crear/{id}?reservado=true`.
5	Solicitud de reserva	Confirmar pago → solicitud de reserva	Si la reserva NO es directa, se marca como **PAGADA**, se genera solicitud, se registra el pago y se actualiza la reserva. Redirige a `/reserva/crear/{id}?reservado=true`.

Ventana Registro

Nº	Categoría	Caso de prueba	Descripción
1	Vista	Mostrar formulario de registro	La ruta `GET /registro` carga correctamente la vista `registro`.
2	Validación	Registro fallido → usuario ya existe	Si el username ya está registrado, se vuelve a la vista `registro` y se muestra un atributo `error`.
3	Registro propietario	Registro exitoso de propietario	Si el usuario no existe y el tipo es PROPIETARIO, se registra correctamente y se muestra la vista `login` con un mensaje de éxito.
4	Registro inquilino	Registro exitoso de inquilino	Si el usuario no existe y el tipo es INQUILINO, se registra correctamente y se muestra la vista `login` con un mensaje de éxito.
5	Validación	Tipo de usuario inválido	Si el tipo de usuario no es válido (ej. ADMIN), se vuelve a la vista `registro` y se muestra un atributo `error`.

Ventana Reserva

Nº	Categoría	Caso de prueba	Descripción

1	Vista / Datos iniciales	Mostrar formulario de reserva	Carga la vista `reserva` mostrando fechas disponibles, fechas reservadas e información del inmueble.
2	Permisos	Crear reserva → usuario propietario	Si el usuario es propietario, no puede reservar y se redirige a `/login`.
3	Sesión	Crear reserva → sin sesión	Si no hay sesión activa, se redirige a `/login`.
4	Reserva directa	Crear reserva directa	Si la disponibilidad es directa, se registra la reserva y se redirige a `/pago/confirmarPago/{id}`.
5	Reserva no directa	Crear reserva no directa	Si la reserva no es directa, se calcula política y tipo, se registra y se redirige a `/pago/confirmarPago/{id}`.
6	Sesión	Ver mis reservas → sin sesión	Si no hay sesión, se redirige a `/login`.
7	Listado reservas	Ver mis reservas → inquilino	Muestra la vista `misReservas` con reservas del inquilino y atributo `esPropietario = false`.
8	Listado reservas	Ver mis reservas → propietario	Muestra la vista `misReservas` con reservas del propietario y atributo `esPropietario = true`.
9	Sesión	Cancelar reserva → sin sesión	Si no hay sesión, se redirige a `/login`.
10	Cancelación	Cancelar reserva correctamente	Se obtiene la solicitud, se elimina, se cancela la reserva y se redirige a `/reserva/misReservas/{username}`.

Ventana Solicitudes

No	Categoría	Caso de prueba	Descripción
1	Sesión	Ver solicitudes → usuario no logueado	Si no hay sesión activa, la ruta `/solicitudes/confirmacionReserva/{username}` redirige a `/login`.
2	Vista / Datos	Ver solicitudes correctamente	Si el usuario está logueado, se cargan las solicitudes del propietario, se muestra la vista `solicitudes` y se inicializa `solicitudSeleccionada` como `null`.
3	Aceptación	Aceptar solicitud	La ruta POST `/solicitudes/solicitud/{id}/aceptar` ejecuta `aceptarSolicitudReserva(id)` y redirige a la lista de solicitudes del propietario.
4	Rechazo	Rechazar solicitud	La ruta POST `/solicitudes/solicitud/{id}/rechazar`

			obtiene la solicitud, elimina la solicitud, cancela la reserva asociada y redirige a la lista de solicitudes.

5.5.2 Capa de Negocio (Controllers)

Gestor Busquedas

Nº	Categoría	Caso de prueba	Descripción
1	Llamada al DAO	Buscar → parámetros correctos	Verifica que el método `buscar()` llama al DAO `buscarFiltrado()` con los parámetros exactos proporcionados por el usuario. Comprueba además que la lista devuelta no es nula y contiene los elementos esperados.
2	Resultados	Buscar → lista vacía	Si el DAO devuelve una lista vacía, el método `buscar()` debe devolver también una lista vacía sin errores. Se verifica además que la llamada al DAO se realiza con los parámetros

Gestor Disponibilidad

Nº	Categoría	Caso de prueba	Descripción
1	Política de cancelación	Política más restrictiva	Si hay varias disponibilidades, se devuelve la política más restrictiva (NO_REEMBOLSABLE).
2	Política de cancelación	Lista vacía	Si no hay disponibilidades, la política por defecto es REEMBOLSABLE.
3	Política de cancelación	Todas iguales	Si todas las disponibilidades tienen la misma política, se devuelve esa misma política.
4	Tipo de reserva	Todas directas	Si todas las disponibilidades son directas, el tipo de reserva es directa (`true`).
5	Tipo de reserva	Una no directa	Si alguna disponibilidad no es directa, el tipo de reserva es no directa (`false`).
6	Disponibilidad para reserva	Intervalo solapa	Si el rango solicitado solapa con alguna disponibilidad, se devuelve la lista correspondiente.
7	Disponibilidad para reserva	Intervalo no solapa	Si el rango solicitado no solapa con ninguna disponibilidad, se devuelve una lista vacía.
8	Registrar disponibilidad	Sin adyacentes	Si no existen disponibilidades adyacentes, se guarda la nueva disponibilidad tal cual.
9	Registrar disponibilidad	Con adyacentes	Si existen disponibilidades adyacentes, se fusionan correctamente (elimina la antigua y guarda la nueva con fechas extendidas).

Gestor Inmuebles

Nº	Categoría	Caso de prueba	Descripción
1	Registro	Registrar inmueble	Verifica que el gestor llama al DAO para guardar un inmueble mediante `save()`.

2	Consulta	Obtener inmueble por ID	Comprueba que el gestor devuelve el inmueble obtenido desde el DAO mediante `findById()`.
3	Consulta	Listar todos los inmuebles	Verifica que el gestor devuelve la lista completa proporcionada por `findAll()`.
4	Consulta filtrada	Listar inmuebles por propietario	Comprueba que el gestor devuelve los inmuebles asociados a un propietario usando `findByPropietario()`.
5	Eliminación	Eliminar inmueble (caso exitoso)	Si el usuario es propietario y no hay reservas activas, el inmueble se elimina correctamente mediante `delete()`.
6	Eliminación / Validación	No eliminar si tiene reservas activas	Si el inmueble tiene reservas activas, el gestor **no** lo elimina y no se llama a `delete()`.

Gestor ListaDeseos

Nº	Categoría	Caso de prueba	Descripción
1	Añadir favorito	Añadir inmueble cuando la lista no existe	Si el inquilino no tiene lista de deseos, se crea una nueva, se añade el inmueble y se guarda en la base de datos.
2	Quitar favorito	Quitar inmueble si ya estaba en la lista	Si el inmueble ya estaba en la lista, se elimina y se guarda la lista actualizada.
3	Validación	Excepción si el inmueble no existe	Si el inmueble no se encuentra en el sistema, se lanza `IllegalArgumentException` y no se modifica la lista.
4	Añadir favorito	Añadir inmueble cuando la lista ya existe	Si la lista ya existe pero no contiene el inmueble, se añade correctamente y se guarda.

Gestor Reservas

Nº	Categoría	Caso de prueba	Descripción
1	Existencia	Existe reserva → true	Si el DAO devuelve una reserva, `existeReserva()` debe retornar `true`.
2	Existencia	Existe reserva → false	Si el DAO devuelve `null`, `existeReserva()` debe retornar `false`.
3	Registro	Registrar reserva	Verifica que el gestor llama a `save()` del DAO para persistir la reserva.
4	Actualización	Actualizar reserva	Comprueba que el gestor llama a `update()` del DAO para actualizar la reserva.
5	Autenticación	Autenticar reserva → usuario correcto	Si el username coincide con el del inquilino asociado a la reserva, devuelve `true`.
6	Autenticación	Autenticar reserva → usuario incorrecto	Si el username no coincide, devuelve `false`.

7	Autenticación	Autenticar reserva → reserva no existe	Si la reserva no existe, devuelve `false`.
8	Cancelación	Cancelar reserva	Si la reserva existe, se elimina mediante `delete()`.
9	Consulta	Obtener reserva por ID	Devuelve la reserva proporcionada por el DAO.
10	Consulta	Obtener reservas por inmueble	Devuelve la lista de reservas asociadas a un inmueble.
11	Consulta	Obtener reservas por inquilino	Devuelve la lista de reservas asociadas a un inquilino.
12	Consulta	Obtener reservas por propietario	Devuelve la lista de reservas asociadas a un propietario.

Gestor Solicitudes

Nº	Categoría	Caso de prueba	Descripción
1	Consulta	Obtener solicitud por ID	Verifica que el gestor devuelve la solicitud obtenida desde el DAO mediante `findById()`.
2	Consulta	Obtener solicitud por ID de reserva	Comprueba que el gestor devuelve la solicitud asociada a una reserva usando `findByIdReserva()`.
3	Generación	Generar solicitud de reserva	Crea una solicitud con noches calculadas, precio total, fecha de creación y la reserva asociada. Verifica que se guarda correctamente.
4	Aceptación	Aceptar solicitud existente	Si la solicitud existe, se marca la reserva como **ACEPTADA**, se actualiza la reserva y se elimina la solicitud.
5	Aceptación	Aceptar solicitud inexistente	Si la solicitud no existe, no se actualiza ninguna reserva ni se elimina nada.
6	Eliminación	Borrar solicitud	Verifica que el gestor llama a `delete()` del DAO para eliminar la solicitud.
7	Consulta por propietario	Obtener solicitudes por propietario	Obtiene los inmuebles del propietario, recupera las solicitudes asociadas a cada uno y devuelve la lista completa

Gestor Usuarios

Nº	Categoría	Caso de prueba	Descripción
1	Login	Login exitoso	Si el usuario existe y la contraseña coincide, `login()` devuelve `true`. Si la contraseña es incorrecta, devuelve `false`.
2	Login	Login usuario no existe	Si el usuario no está registrado, `login()` devuelve `false`.
3	Existencia	Existe usuario → true	Si el DAO devuelve un usuario, `existeUsuario()` devuelve `true`.

4	Existencia	Existe usuario → false	Si el DAO devuelve `null`, `existeUsuario()` devuelve `false`.
5	Registro	Registrar inquilino	Verifica que el gestor llama a `save()` del DAO de inquilinos.
6	Registro	Registrar propietario	Verifica que el gestor llama a `save()` del DAO de propietarios.
7	Consulta	Obtener propietario por username	Devuelve el propietario obtenido desde el DAO.
8	Consulta	Obtener inquilino por username	Devuelve el inquilino obtenido desde el DAO.
9	Roles	esPropietario / esInquilino → propietario	Si el usuario es un `Propietario`, `esPropietario()` devuelve `true` y `esInquilino()` devuelve `false`.
10	Roles	esPropietario / esInquilino → inquilino	Si el usuario es un `Inquilino`, `esInquilino()` devuelve `true` y `esPropietario()` devuelve `false`.
11	Roles	Usuario no existe	Si el usuario no existe, ambos métodos (`esPropietario`, `esInquilino`) devuelven `false`.

6- Plan de Gestión de Mantenimiento

1. Alcance y Objetivos del Mantenimiento

1.1 Alcance

Este plan de mantenimiento se aplica de forma integral al sistema de **Alquiler de Viviendas**. El alcance abarca los siguientes elementos técnicos y funcionales definidos en la documentación del proyecto:

- **Software y Código Fuente:** Incluye todas las clases de las tres capas (Presentación con Thymeleaf, Negocio con entidades y gestores, y Persistencia con Spring Data JPA).
- **Base de Datos:** Mantenimiento y optimización del esquema de base de datos Derby/DBeaver diseñado para el sistema.
- **Gestión de Dependencias:** Supervisión del archivo `pom.xml` de Maven para asegurar que las librerías externas se mantengan actualizadas y seguras.
- **Documentación:** Actualización constante de los requisitos funcionales/no funcionales, el Javadoc del código y el manual de usuario ante cualquier cambio en el sistema.
- **Entorno de Ejecución:** Garantizar la compatibilidad continua con el entorno Java 21.

1.2 Objetivos

El objetivo principal es asegurar que el sistema de alquiler opere de manera ininterrumpida y evolucione según las necesidades de inquilinos y propietarios. Los objetivos específicos son:

- **Garantizar la operatividad funcional:** Asegurar que procesos críticos como el inicio de sesión, la reserva inmediata y la gestión de pagos (vía PayPal o tarjetas) funcionen sin errores.
- **Preservar la Seguridad:** Mantener la integridad de los datos personales y el cifrado de contraseñas, además de asegurar el uso de protocolos HTTPS según el RNF-04.
- **Optimizar el Rendimiento:** Monitorizar que el tiempo de respuesta en búsquedas se mantenga por debajo de los 3 segundos, cumpliendo con el RNF-02.
- **Facilitar la Evolución (Escalabilidad):** Permitir la incorporación de nuevas funcionalidades (como nuevas reglas de negocio o filtros de búsqueda) sin degradar la calidad del producto existente, respetando el RNF-17.
- **Controlar la Deuda Técnica:** Asegurar que cualquier cambio respete los estándares de programación Java y la arquitectura en 3 capas para evitar que el código se vuelva inmanejable con el tiempo.

2. Clasificación de las Intervenciones de Mantenimiento

En el sistema de Alquiler de Viviendas, las actividades de mantenimiento se clasificarán según su objetivo en las siguientes categorías:

2.1 Mantenimiento Correctivo

Su objetivo es localizar y eliminar defectos que causen que el sistema no cumpla con sus requisitos especificados.

- **Ejemplos en el proyecto:**
    - Error en el proceso de pago (RF-18) donde el inquilino paga pero la reserva no se confirma.
    - El sistema no reembolsa automáticamente el pago al inquilino tras una denegación de reserva (RF-17).
    - Fallos en la visualización de fotos en el diseño responsivo (RNF-09).

2.2 Mantenimiento Adaptativo

Modificaciones realizadas para que el sistema siga funcionando correctamente ante cambios en su entorno externo.

- **Ejemplos en el proyecto:**
    - Actualización del código si se migra de la base de datos Derby a una base de datos en la nube como PostgreSQL.
    - Ajustes en la lógica de persistencia si se actualiza la versión de **Java 21** a una versión superior.
    - Modificación de los términos de servicio o gestión de datos para cumplir con nuevas leyes de protección de datos (RGPD).

2.3 Mantenimiento Perfectivo (Evolutivo)

Orientado a mejorar el rendimiento o añadir nuevas funcionalidades no previstas originalmente.

- **Ejemplos en el proyecto:**
    - Añadir un sistema de "Reseñas y Calificaciones" para que los inquilinos valoren a los propietarios.
    - Mejorar el algoritmo de búsqueda para ofrecer alojamientos recomendados basados en la "Lista de Deseos" (RF-12).
    - Optimizar la Capa de Negocio para que el tiempo de respuesta sea inferior a 1 segundo (superando el RNF-02 de 3 segundos).

2.4 Mantenimiento Preventivo

Modificación del software para mejorar su mantenibilidad o calidad interna sin cambiar sus funciones.

- **Ejemplos en el proyecto:**
    - Refactorizar clases de la **Capa de Persistencia** para reducir el acoplamiento y mejorar la legibilidad según estándares Java.
    - Actualizar y completar el **Javadoc** en métodos complejos de gestión de reservas (RNF-15).
    - Eliminar dependencias obsoletas en el archivo `pom.xml` de Maven para evitar conflictos futuros.

3. Organización y Roles del Mantenimiento

Para asegurar una gestión eficiente del sistema de Alquiler de Viviendas, se establecen los siguientes roles y responsabilidades entre los miembros del equipo:

3.1 Roles Definidos

- **Responsable de Gestión de Peticiones (Help Desk):** * **Función:** Recibir los informes de fallos de los usuarios (inquilinos/propietarios) o las solicitudes de nuevas funciones.
    - **Responsabilidad:** Clasificar la petición (Correctiva, Adaptativa, etc.) y asignarle una prioridad (Alta, Media, Baja).
- **Analista de Impacto:**
    - **Función:** Antes de tocar el código, evalúa qué clases de las capas de Negocio o Persistencia se verán afectadas.
    - **Responsabilidad:** Evitar que un cambio en la gestión de pagos (RF-18) rompa la lógica de reservas (RF-17).
- **Equipo de Desarrollo (Mantenedores):**
    - **Función:** Realizar la modificación física del código fuente y actualizar el archivo `pom.xml` si es necesario.
    - **Responsabilidad:** Aplicar estándares de **Clean Code** y actualizar el **Javadoc**.
- **Responsable de QA (Pruebas de Regresión):**
    - **Función:** Ejecutar el Plan de Gestión de Pruebas (sección 5 de vuestra documentación) tras cada cambio.
    - **Responsabilidad:** Asegurar que el sistema sigue siendo estable y que no han aparecido nuevos errores tras la intervención.

3.2 Asignación de Responsabilidades (Matriz simplificada)

Dado que somos tres integrantes, las asignaciones varían, y todos los miembros actuarán como "Mantenedores" en sus respectivas áreas, pero los cambios críticos deben ser revisados por un segundo integrante para garantizar la calidad.

Tarea	**Responsable**
Gestión de Base de Datos y Persistencia	**Daniel Salas**
Lógica de Negocio y Controladores	**Saul Corbelle**
Interfaz de Usuario (Thymeleaf) y Pruebas	**Laura Rodríguez**

4. Proceso de Actuación

Para realizar cualquier intervención de mantenimiento, el equipo seguirá un flujo de trabajo estructurado para evitar modificaciones precipitadas o sin documentar.

4.1. Recepción y Registro de la Petición

Toda solicitud, ya sea un error reportado por un inquilino o una mejora de un propietario, debe registrarse incluyendo:

- **Descripción del problema o mejora.**
- **Localización:** Módulos o capas afectadas (Presentación, Negocio, Persistencia).
- **Tipo de cambio:** Clasificación según la sección 2 de este plan.

4.2. Análisis de Impacto y Viabilidad

Antes de modificar el código, el responsable asignado debe:

- **Evaluar el impacto:** Determinar si el cambio en una entidad (ej. `Inmueble`) afecta a los controladores de negocio o a la base de datos **Derby**.
- **Estimar el esfuerzo:** Calcular el tiempo y recursos necesarios.

4.3. Implementación del Cambio

Se procede a la codificación siguiendo estas directrices:

- **Clean Code:** Aplicar el principio de "dejar el código más limpio de cómo se encontró" (Boy Scout Rule).
- **Estándares Java:** Cumplir con los estándares de programación y actualizar el **Javadoc** (RNF-15).
- **Control de Versiones:** Realizar los cambios en una rama específica de **GitHub** para mantener la trazabilidad.

4.4. Plan de Pruebas de Regresión

Es el paso más crítico para evitar efectos secundarios. Se deben re-ejecutar las pruebas definidas en nuestro **Plan de Gestión de Pruebas** :

- **Pruebas Unitarias:** Verificar las entidades y DAOs modificados.
- **Pruebas de Integración:** Asegurar que la comunicación entre capas (ej. Reserva -> Pago) sigue funcionando bajo HTTPS (RNF-04).

4.5. Documentación y Cierre

Una vez validado el cambio por el **Product Owner** o el **Scrum Master**:

- **Actualizar la documentación técnica:** Reflejar los cambios en el diagrama de clases o de base de datos si fuera necesario.

- **Registro de mantenimiento:** Anotar el esfuerzo real dedicado y las dificultades encontradas para mejorar futuros procesos

5. Gestión de la Deuda Técnica y Mantenibilidad

Para cumplir con el requisito **RNF-14** (Arquitectura en 3 capas) y **RNF-15** (Estándares Java y Javadoc) , el plan seguirá estas directrices para minimizar la deuda técnica y maximizar la facilidad de modificación.

5.1. Estrategias de Prevención (Clean Code)

Se aplicarán los principios de Clean Code para que el coste de mantenimiento no crezca de forma inmanejable:

- **Principio de Responsabilidad Única (SRP):** Cada clase (ej. `GestorPago` o `GestorReserva`) tendrá una sola razón para cambiar.
- **Principio del Boy Scout:** Todo miembro del equipo que realice un mantenimiento correctivo tiene la obligación de mejorar ligeramente la legibilidad del código circundante.
- **Principio KISS y YAGNI:** Se priorizará la simplicidad, evitando implementar funcionalidades complejas "por si acaso" se necesitan en el futuro.

5.2. Monitorización de la Deuda Técnica

La deuda técnica es inevitable, pero debe ser controlada para evitar que la capacidad de satisfacer necesidades caiga drásticamente. Se vigilarán los siguientes indicadores:

- **Código Duplicado:** Se utilizarán herramientas para detectar lógica repetida en (SonarCloud).
- **Complejidad Ciclomática:** Se evitarán métodos con demasiadas ramificaciones (`if/else` anidados) que dificulten las pruebas.
- **Comentarios y Javadoc:** Se asegurará que el código no contenga comentarios obsoletos o innecesarios, prefiriendo código que se explique por sí mismo.

5.3. Refactorización Regular

No toda la deuda debe pagarse de inmediato, pero se priorizarán las áreas críticas que afecten a la mantenibilidad:

- Se reservará un tiempo específico en cada ciclo de mantenimiento para la refactorización constante del código.

- Se realizarán revisiones de código cruzadas entre los integrantes para identificar patrones de diseño deficientes antes de que se acumulen.