MEMORIA TÉCNICA DEL PROYECTO: SISTEMA DE GESTIÓN Y VENTA DE CALZADO

Código del Proyecto: DAM2-PROY-202X-XX
Versión del Documento: 1.0
Fecha: 01/02/2026

---

1. Introducción y Objetivo

Esta memoria describe el proyecto de una aplicación de escritorio que hemos desarrollado para el módulo de Desarrollo de Interfaces de 2º DAM. La idea era crear un sistema completo tipo tienda online de zapatillas, con dos partes claras: una zona de administración (para gestionar productos y usuarios) y una zona de cliente (para comprar). El objetivo era poner en práctica todo lo aprendido sobre Java, interfaces gráficas y bases de datos.

2. Qué Tiene que Hacer la Aplicación (Requisitos)

2.1. Funcionalidades Principales

- RF-01: Dos tipos de usuario. Tiene que haber Administradores y Clientes, cada uno con sus pantallas y permisos.
- RF-02: Login seguro. Tienes que logearte con tu usuario y contraseña, y la app te lleva a tu sitio (admin o cliente).
- RF-03: CRUD de zapatillas (solo admin). Como administrador, puedes:
    - Añadir nuevas zapatillas (con su nombre, marca, precio, etc.).
    - Ver todas las zapatillas.
    - Modificar cualquier dato de una zapatilla.
    - Borrar zapatillas.
- RF-04: Gestión de usuarios (solo admin). El admin puede ver la lista de usuarios, editar sus datos y borrarlos. (Los usuarios normales se registran ellos solos).
- RF-05: Registro. Cualquiera puede crearse una cuenta de cliente.
- RF-06: Comprar (cliente). Como cliente puedes:
    - Ver el catálogo de zapatillas disponibles.
    - Meter y sacar cosas del carrito.
    - "Pagar" (simularlo), metiendo una tarjeta y generando un pedido.

- RF-07: Tarjetas (cliente). Puedes guardar varias tarjetas de crédito/débito en tu perfil.

2.2. Cómo tiene que funcionar (Requisitos Técnicos)

- RNF-01: Tipo de aplicación. Es una app de escritorio Java que se conecta a una base de datos.
- RNF-02: Fácil de usar. La interfaz tiene que ser clara, ir rápido y que no se atasque.
- RNF-03: Código ordenado. El código está separado en partes (ventanas, lógica, base de datos) para poder arreglar o añadir cosas fácilmente.
- RNF-04: Que guarde todo. Toda la información importante (usuarios, productos, pedidos) se guarda en una base de datos MySQL.
- RNF-05: Seguridad básica. Las contraseñas no se guardan en claro (se hashean) y no puedes acceder a sitios de admin si eres cliente.
- RNF-06: Que funcione en muchos sitios. Con tener Java instalado, debería funcionar en cualquier ordenador.

3. Tecnologías que hemos usado

- Lenguaje: Java 11. Lo usamos porque es el que hemos dado en clase, es potente y sirve perfectamente para este tipo de programas.
- Interfaz gráfica: JavaFX y FXML. Usamos Scene Builder para diseñar las ventanas (archivos `.fxml`) de forma visual, y luego Java para programar lo que hacen. Así separamos el diseño del código, que es mucho más limpio.
- Base de datos: Hibernate para conectarse a MySQL. Hibernate nos ahorra escribir mucho SQL a mano. Nos permite trabajar con objetos Java (como un objeto `Zapatilla`) y él solo lo guarda o lo lee de la base de datos. Muy útil.
- Estructura del código: Seguimos un estilo MVC (Modelo-Vista-Controlador):
    - Modelo: Las clases Java que representan las tablas de la BD (Usuario, Zapatilla...).
    - Vista: Los archivos FXML (el diseño de las ventanas).
    - Controlador: El código Java que hace de puente: coge los datos de la vista, aplica la lógica y usa Hibernate para guardar o cargar cosas del modelo.
- Control de versiones: Git con GitHub, para no perder nada y trabajar mejor en equipo.

4. Cómo está hecho por dentro

4.1. Diagrama de Clases

Aquí tienes un esquema de las clases principales y cómo se relacionan. Lo dibujaría con cajas y flechas, pero en texto sería algo así:

Explicación:

- `Perfil` tiene es `Usuario` o `Admin`
- `Administrador` gestiona `Usuario`.
- Un `Usuario` puede tener una `Tarjetas`.
- Un `Usuario` hace `Pedidos`.
- Un `Pedido` es efectuado por un `Usuario` .
- Cada `Línea de Pedido` se refiere a una `Zapatilla` y dice cuántas unidades.

4.2. Diseño de las Pantallas

- Navegación: Hay ventanas separadas para: Login, Registro, Panel de Admin (con pestañas para Zapatillas y Usuarios), Catálogo para clientes y Carrito. Se van abriendo y cerrando, es fácil de seguir.
- Colores: Usamos azul claro (#4A90E2) y blanco (#FFFFFF).
    - ¿Por qué azul? Porque transmite confianza y seriedad (importante cuando la gente mete sus datos y tarjetas). Además, al ser claro, no cansa la vista.
    - ¿Por qué blanco? Porque queda limpio, deja ver bien el texto y hace que los colores azules destaquen. Juntos tienen buen contraste, lo que ayuda a que sea más accesible.
- Controles: Usamos lo que ofrece JavaFX: `TableView` para mostrar listas (de zapatillas, usuarios...), `TextField` para rellenar datos, `Button` con el mismo estilo en toda la app, y `ImageView` para mostrar las fotos de las zapatillas. Los botones de "Borrar" siempre piden confirmación para evitar accidentes.

5. Mejoras implementadas y herramientas de desarrollo

5.1. Sistema de Logging personalizado

Hemos creado una clase `Logger` personalizada para tener un mejor control sobre lo que sucede en la aplicación:

- ¿Para qué sirve? Registra automáticamente eventos importantes, como errores, inicios de sesión, operaciones CRUD críticas (borrado de productos, creación de pedidos) o intentos de acceso no autorizados.
- ¿Cómo funciona? Genera un fichero de errores o de mensajes informativos del uso de la app en una carpeta dedicada. Este fichero sigue un formato claro.

5.2. Documentación Javadoc

Hemos documentado todo el código clave del proyecto usando Javadoc:

- ¿Qué hemos documentado? Todas las clases principales (entidades, controladores, DAOs, utilidades como `PersonalLogger`), sus atributos importantes y, sobre todo, los métodos públicos.
- ¿Qué incluye cada comentario? Para cada método, explicamos qué hace, qué parámetros recibe, qué devuelve y qué excepciones puede lanzar.
- Beneficio: Esto no solo sirve para nosotros (para recordar cómo funciona algo semanas después), sino que si otro desarrollador tuviera que trabajar en nuestro código, lo entendería mucho más rápido. Además, podemos generar automáticamente una página web con toda la documentación usando la herramienta `javadoc` de JDK.

5.3. Pruebas de Interfaz con `java.awt.Robot`

Para asegurarnos de que la interfaz de usuario funcionaba correctamente de principio a fin, implementamos pruebas automatizadas de flujo completo usando la clase `java.awt.Robot`:

- ¿Cómo funciona? Creamos una clase para cada ventana a testear que simula las acciones de un usuario real: mover el ratón, hacer clic en botones específicos, escribir texto en campos, pulsar teclas, etc. El robot sigue un “guión” que programamos para probar escenarios completos.

- Escenarios probados:
    - Flujo de registro: Navega a la ventana de registro, rellena los campos con datos de prueba, hace clic en "Registrar" y verifica que aparece el mensaje de éxito.
    - Flujo de login y navegación: Introduce credenciales válidas, verifica que se abre la ventana correcta según el perfil, y prueba la navegación entre pestañas en el panel de administración.
    - CRUD de productos: Verifica que se muestran zapatillas, añade una zapatilla nueva, busca en la tabla para verificar que aparece, la edita y finalmente la elimina, confirmando los diálogos de borrado.
- Retos y soluciones:
    - Sincronización: A veces el robot iba más rápido que la aplicación. Solucionamos esto añadiendo `Thread.sleep()` o esperando a que aparecieran elementos específicos en pantalla.
    - Programación del test: No usamos coordenadas de pantalla absolutas, porque puede fallar si mueves la ventana. Hacemos el test buscando componentes por su id dentro de la ventana.
- Beneficio: Nos permitió detectar errores que no habíamos visto en pruebas manuales, como botones que no se activaban correctamente o secuencias de navegación que podían dejar la aplicación en un estado inconsistente. Aunque es más rudimentario que frameworks como TestFX, `Robot` es una herramienta potente y útil que ya viene con Java.

6. Estrategia de Pruebas

Nuestra estrategia se ha basado en 3 puntos:

1. Pruebas manuales sistemáticas: Probamos cada funcionalidad de forma manual según los casos de uso.
2. Pruebas automatizadas de interfaz: Con `java.awt.Robot` verificamos los flujos de usuario más importantes.
3. Monitoreo mediante logging: El sistema de logs nos proporcionó un historial de ejecución que nos ayudó a identificar y reproducir errores.

7. Conclusión y qué se podría mejorar

Hemos conseguido hacer una aplicación completa y funcional que cumple con lo que se pedía. Con este proyecto hemos aprendido a integrar varias tecnologías: JavaFX para la interfaz, Hibernate para la base de datos, y además hemos aplicado buenas prácticas profesionales como la implementación de un sistema de Logger, la documentación exhaustiva con Javadoc, y pruebas automatizadas de interfaz usando `Robot`, lo que da mucha más calidad y robustez al proyecto.

Para el futuro, se podría mejorar con:

1. Pruebas unitarias automatizadas. El siguiente paso sería implementar JUnit para probar de forma aislada las clases de lógica de negocio, servicios y DAOs. Esto complementaría perfectamente nuestras pruebas de interfaz con `Robot`.
2. Framework de pruebas de UI más avanzado. Investigar el uso de TestFX, que está específicamente diseñado para JavaFX y permite localizar componentes de forma más fiable (por ID, texto, etc.) en lugar de usar coordenadas.
3. Mejorar la estructura. Separar mejor la lógica de negocio en una capa de "Servicios", para que los controladores solo se encarguen de la vista. Podríamos usar un framework como Spring para la inyección de dependencias y gestión de transacciones.
4. Pasarla a web. Convertir la parte de atrás (la lógica y la BD) en una API REST con Spring Boot, y hacer una página web con React o Angular como interfaz. Nuestra actual capa de modelo y lógica podría reutilizarse en gran medida.

Elaborado por: Kevin Sanz, Alicia Martinez y Pablo Velas
Para: Módulo de Desarrollo de Interfaces - 2º DAM
Curso: 2025-2026