Entrega final: Documentación adicional

Marcelo Fort Muñoz

20 de diciembre de 2024

Resumen

Este documento es una documentación adicional a la entrega final del proyecto de la asignatura de Técnicas de Programación Avanzada.

Índice general

Capítulo 1

Introducción

1.1. Descripción del proyecto

En este proyecto se ha desarrollado un front-end para un quiosco de venta de billetes de tren para la empresa *TrainGo*. El quiosco contempla cambio en los datos del usuario, búsqueda de trenes, compra de billetes y visualización de billetes comprados. Además, se puede cambiar el idioma de la interfaz.

Aparte de esto, se ha desarrollado un sistema de persistencia de datos basado en el patrón DAO (*Data Access Object*) que se sirve de archivos XML para almacenar los datos.

1.2. Objetivos

1. Desarrollar una aplicación de escritorio en Java.
2. Usar Herencia y Polimorfismo.
3. Usar Interfaces y Clases Abstractas.
4. Usar Excepciones.
5. Usar anotaciones.
6. Usar patrones de diseño.
7. Implementar una interfaz gráfica (GUI).

1.3. Motivaciones y alcance

El proyecto se ha desarrollado con el objetivo de aprender a usar manualmente java swing y poner en práctica patrones de diseño y buenas prácticas de programación.

El alcance del proyecto es limitado, ya que no solo se procesa la interacción por parte del usuario, dejando de lado la interacción con el resto de la infraestructura de la empresa.

Por otro lado, se ha implementado un sistema de persistencia de datos basado en archivos XML.

Capítulo 2

Diseño del programa

2.1. Prototipado de la interfaz gráfica

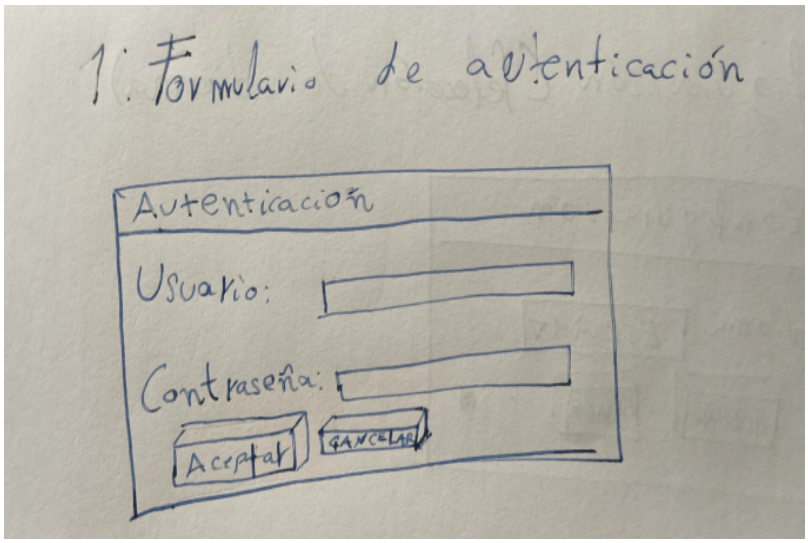

Figura 2.1: Pantalla 1: Formulario de autenticación

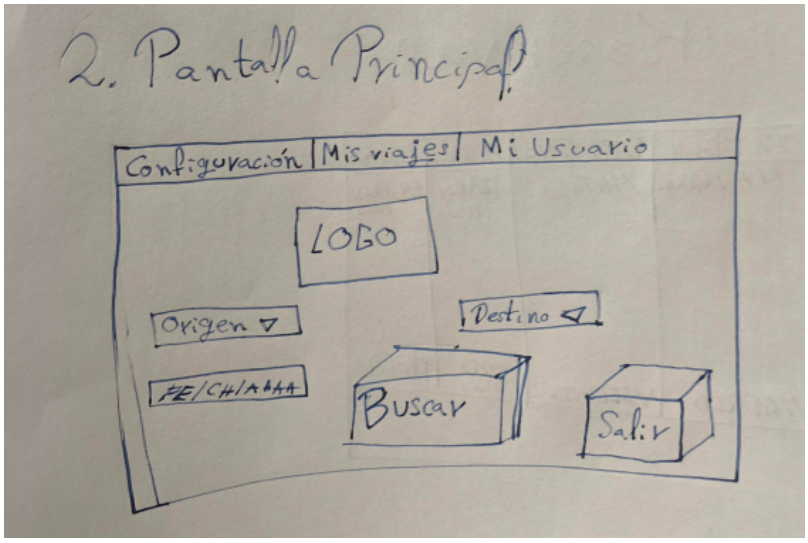

Figura 2.2: Pantalla 2: Pantalla principal

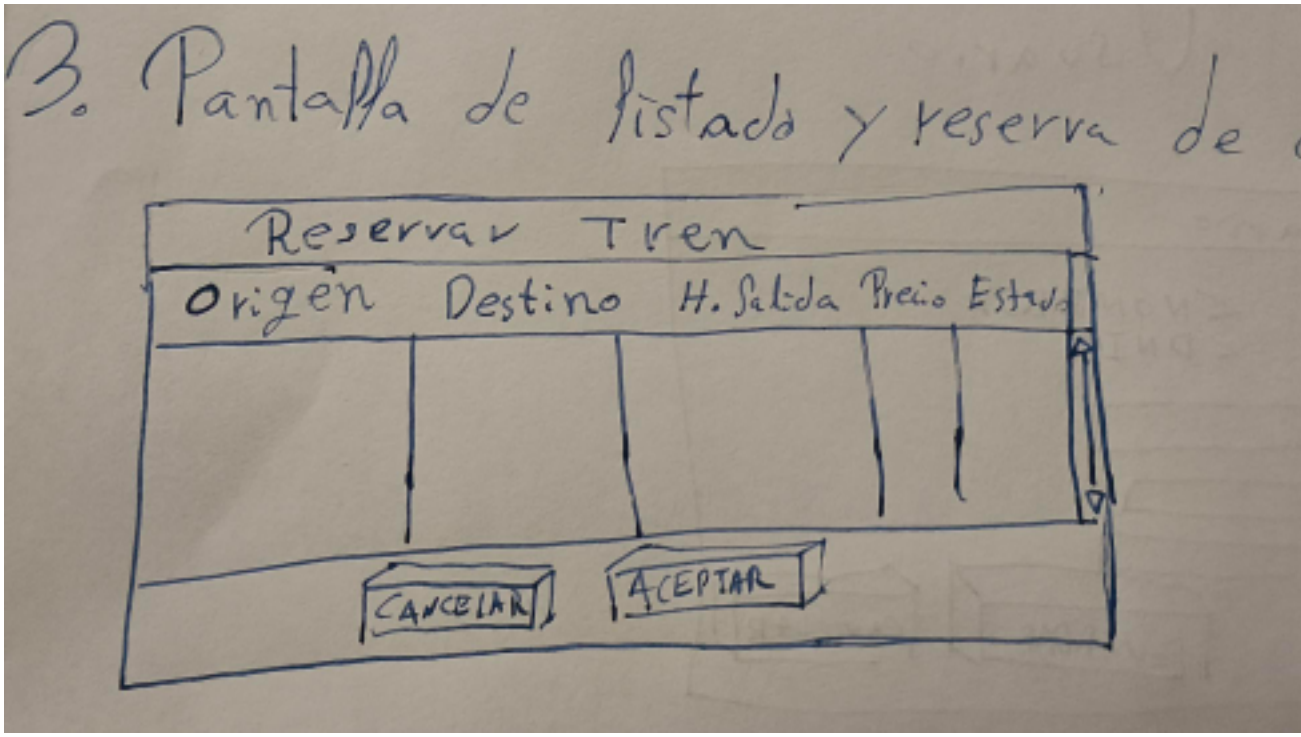

Figura 2.3: Pantalla 3: Lista de trenes y reserva de billetes

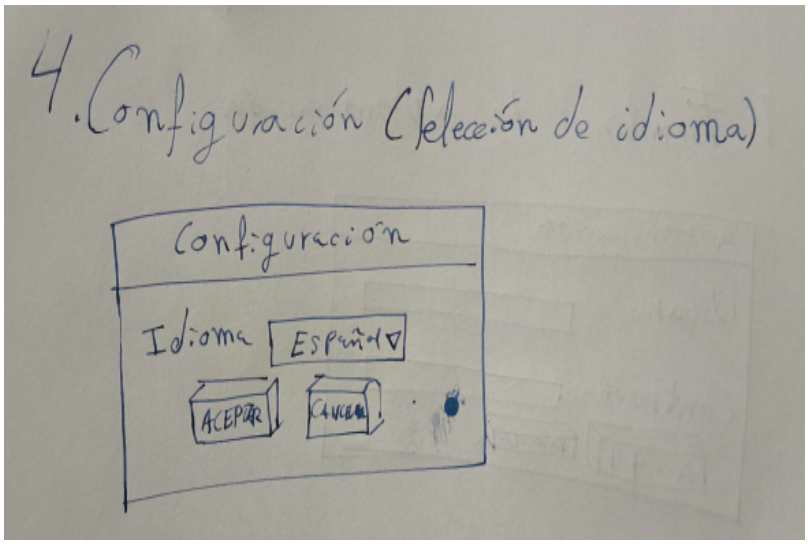

Figura 2.4: Pantalla 4: Configuración

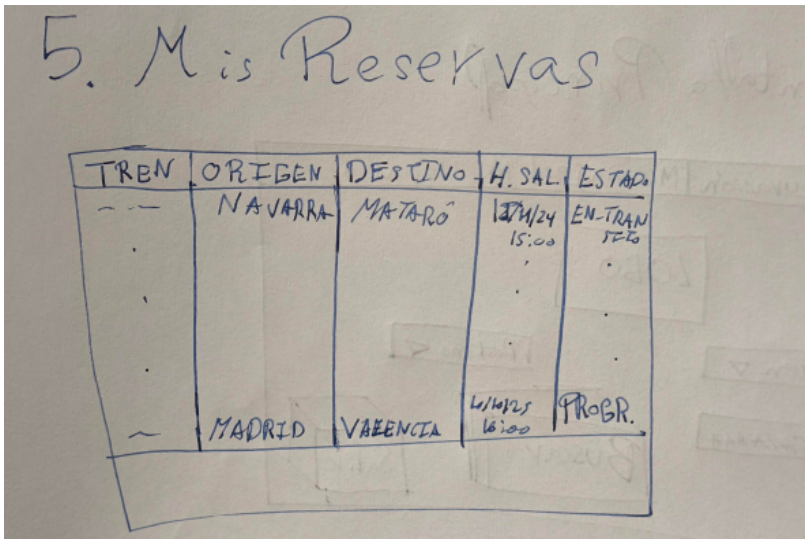

Figura 2.5: Pantalla 5: Mis Reservas

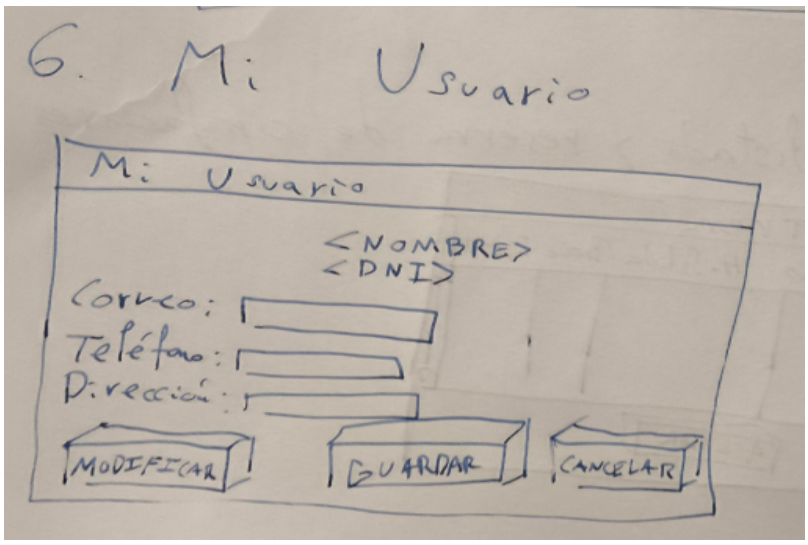

Figura 2.6: Pantalla 6: Mi usuario

2.2. Diagramas UML

2.2.1. Diagrama de clases

Para comenzar, en la Figura 2.7, se muestran las clases que representan a la información que se maneja en la aplicación.

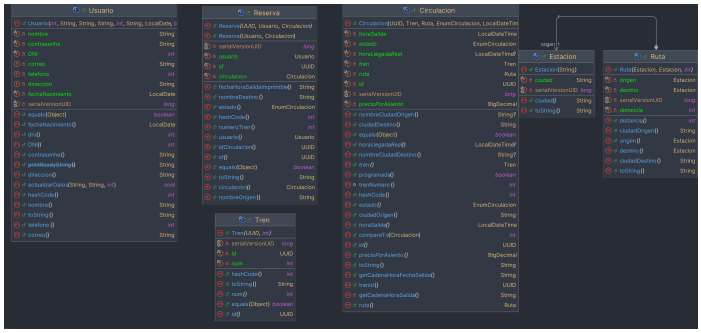

Figura 2.7: Modelo de datos

2.2.2. Diagrama de clases DAO

Relacionado con el modelo de datos, para implementar una persistencia o «serialización» de los datos, se ha implementado un patrón DAO (*Data Access Object*). El patrón DAO se encarga de abstraer la lógica de acceso a los datos (ver Sección 2.3). En la Figura 2.8 se muestra el diagrama de clases de este patrón.

En la implementación de este patrón, se han usado varias funcionalidades clave de Java. Por un lado, la clase *AbstractDAO* es una clase abstracta que implementa la interfaz *IDAO*. La clase AbstractDAO a su vez tiene una relación de herencia con los DAOs individuales.

2.2.3. Diagrama de clases GUI

Llegados a este punto, se ha implementado la interfaz gráfica de usuario (GUI). Se ha usado una estructura formulario-modelo para las clases GUI con tablas y elementos con información. En la Figura 2.9 se muestra el diagrama de clases de la GUI.

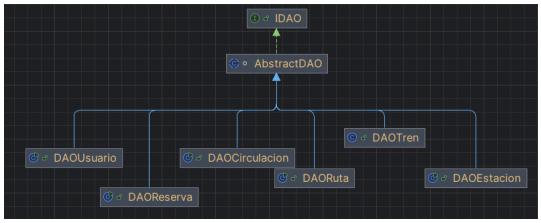

Figura 2.8: Diseño de la estructura de clases del DAO

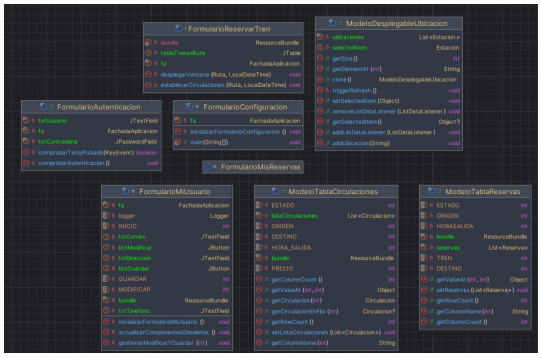

Figura 2.9: Diagrama de clases para la interfaz gráfica

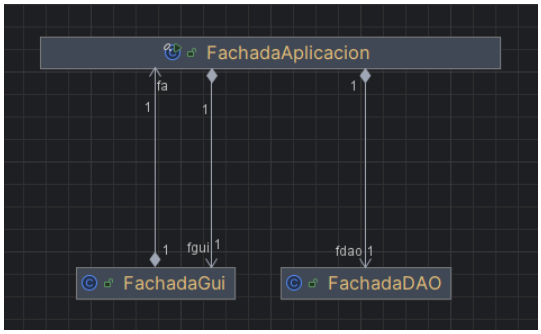

Figura 2.10: Jerarquia de clases para el patrón façade

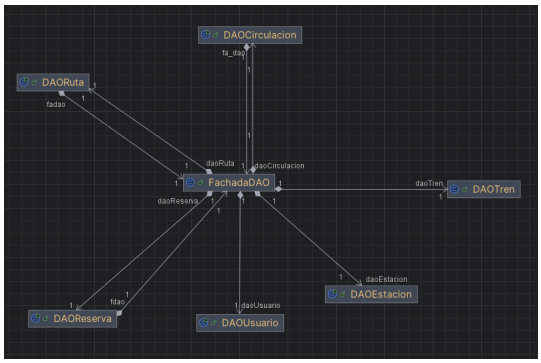

Figura 2.11: Relaciones entre la façada de los DAOs y los mismos DAOs

2.2.4. Façade

Para simplificar la interacción entre la GUI y el modelo de datos, se ha implementado un patrón Façade. No solo simplifica la interacción, sino que también permite una mayor flexibilidad en la implementación de la GUI respecto al modelo de datos. En la Figura 2.10 se muestra el diagrama de clases del Façade respecto a su jerarquía interna.

Para poner un ejemplo, se puede ver como interactúan los DAOs con el Façade en la Figura 2.11.

Estos son todos los diagramas UML más relevantes. Se hablará más en detalle de la implementación en la Capítulo 3. Respecto a la estructura del proyecto, se detalla a continuación en la Sección 2.3.

2.3. Estructura del proyecto

Para este apartado, seguiremos una lógica de aproximación top-down. Analizaremos primero la estructura general del proyecto, para luego ir descendiendo en detalle. [1]

De esta forma iremos viendo cómo se ha estructurado el proyecto, desde la raíz hasta las clases más concretas. A continuación, se muestra la estructura de directorios del proyecto en la Figura 2.12.

Figura 2.12: Estructura de directorios del proyecto

Dentro del directorio *datos*, habremos de hallar los archivos de datos que se usan en la aplicación. Estos están escritos en un formato XML.

En el directorio *src*, se encuentran los archivos fuente del proyecto, donde se encuentra la implementación de la aplicación.

El código fuente de esta aplicación sigue una modificación **patrón Modelo-Vista-Controlador** (MVC). En este caso, se ha modificado el patrón para que se ajuste a las necesidades de la aplicación. La estructura de directorios de la aplicación se muestra en la Figura 2.13.

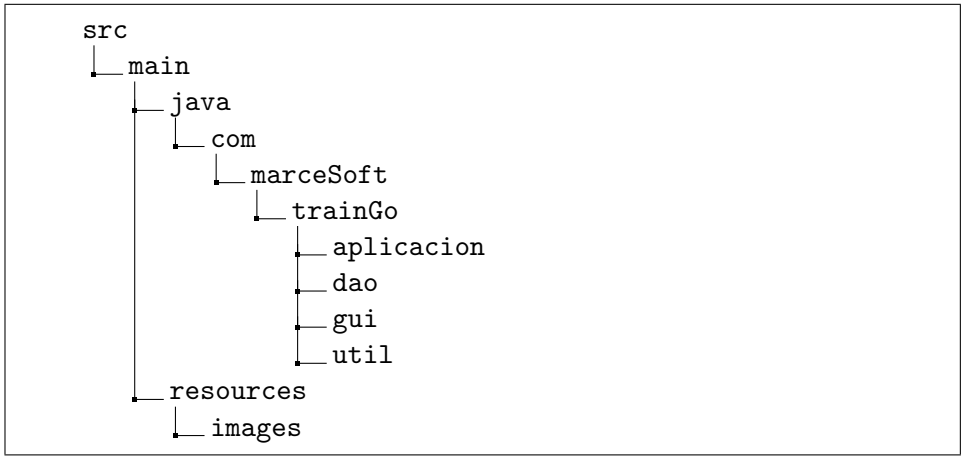

Figura 2.13: Estructura de directorios del código fuente

[1] Nota para el lector: Algunos patrones están repartidos entre este capítulo y el siguiente.

En el directorio *aplicacion*, se encuentran las clases que implementan la lógica de la aplicación y los modelos de datos. (es decir, modelo y controlador).

En el directorio *dao*, se encuentran las clases que implementan el patrón DAO. Es importante separar la lógica de acceso a los datos de la lógica de la aplicación, ya que al hacerlo, se facilita la implementación de la persistencia de los datos.

En el directorio *gui*, se encuentran las clases que implementan la interfaz gráfica de usuario. Estas clases se encargan de mostrar la información al usuario y de recibir la información del usuario (vista).

En el directorio *util*, se encuentran las clases que implementan utilidades para la aplicación.

En el directorio *resources*, hallamos los *bundles* de internacionalización, archivo de configuración del *logger* y las imágenes que se usan en la aplicación. Sin embargo, es mejor no adelantarse, ya trataremos estos aspectos en el Capítulo 3.

2.3.1. Estructura del paquete *aplicacion*

Volviendo a la estructura de paquetes, en el paquete *aplicacion* se encuentran las clases que implementan parte de la lógica de la aplicación.

Por un lado, tenemos la *FachadaAplicacion*, que se encarga de abstraer la lógica de la aplicación. [2] Al mismo nivel, se encuentran las clases «contenedoras» de los modelos de datos.

Bajando un nivel nos podemos encontrar con la anotación *NoNegativo* y su validador, los dos enums, las excepciones y formateadores personalizados. Esto se ve muy bien en la Figura 2.14.

Figura 2.14: Estructura de clases del paquete *aplicacion*

[2] En la Figura 2.10 se muestra la jerarquía de clases de la Fachada.

De esta zona, lo más reseñable a nivel de diseño, es, tal vez, es la decisión de poner Usuario como clase contenedora modificable, mientras que el resto de clases son inmutables (*Records*). Esto se debe a que los usuarios son los únicos datos que son susceptibles a cambios.

Otra vez, se me puede (con cierta razón) acusar de adelantarme. Permitidme, entonces, que me disculpe y que os remita al Capítulo 3 para más detalles. Sin embargo, considero que es una decisión de diseño importante, ya que aunque hay datos que pueden ser ≪renovados≫ (como las reservas), una reserva ≪modificada≫ es, en realidad, una reserva nueva (en nuestro caso). Por este motivo, se ha decidido que las reservas sean inmutables.

2.3.2. Estructura del paquete *dao*

En el paquete *dao*, se encuentran las clases que implementan el patrón DAO, que se encarga de abstraer la lógica de acceso a los datos (persistencia).

Este paquete está estructurado como se muestra en la Figura 2.15.

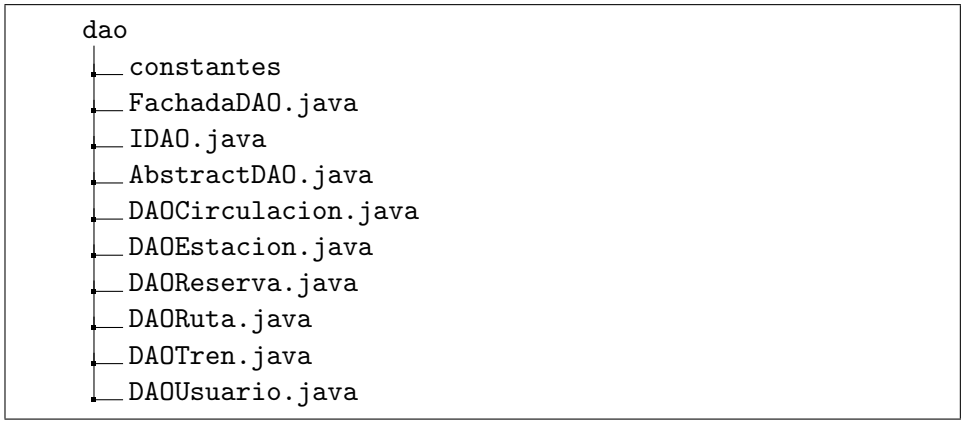

Figura 2.15: Estructura de clases del paquete *dao*

En este paquete, se ha implementado un patrón DAO para abstraer la lógica de acceso a los datos. Este ha sido implementado siguiendo una versión simplificada de las directrices oficiales de ORACLE [1], que también es usado por otros autores [2].

La toma de decisiones en este paquete ha sido más sencilla, ya que se ha seguido una estructura de clases más estándar.

Siguiendo el patrón previamente mencionado, se ha implementado una interfaz *IDAO* que define los métodos que deben implementar los DAOs. En este caso, se ha añadido una clase abstracta *AbstractDAO* que implementa

la interfaz *IDAO*. Todo esto se puede ver en la Figura 2.8.

Hay un paquete más dentro del paquete *dao*, que puede pasar fácilmente desapercibido. Este paquete es el de las constantes. Su distribución se muestra en la Figura 2.16.

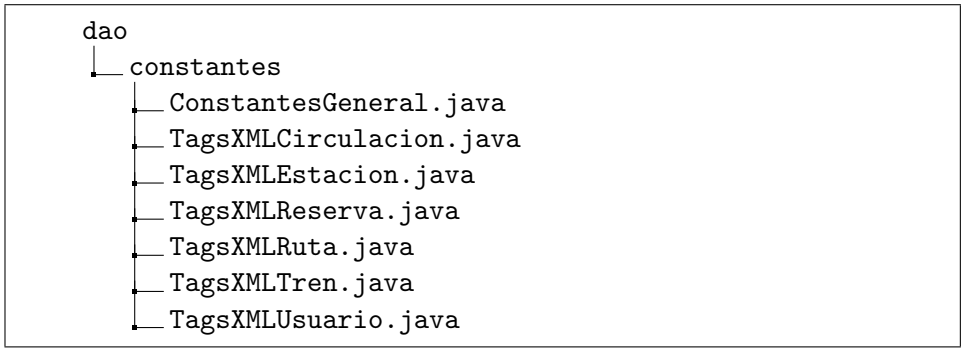

Figura 2.16: Estructura de clases del paquete *dao*

En este paquete, se encuentran las constantes que se usan en los DAOs.

2.3.3. Estructura del paquete *gui*

Saliendo de la espesura de los DAOs, nos encontramos con el paquete *gui*.

Este paquete contiene las clases que implementan la interfaz gráfica de usuario. La estructura de este paquete se muestra en la Figura 2.17.

Figura 2.17: Estructura de clases del paquete *gui*

Es un paquete relativamente sencillo con una estructura bastante intuitiva. Por un lado, tenemos la habitual façada, que se encarga de abstraer la lógica de la interfaz gráfica. Y, nos quedan los formularios y los modelos.

Si desviamos la mirada hacia los formularios, nos encontramos con todos los formularios que se usan en la aplicación. La estructura de este paquete se muestra en la Figura 2.18.

Pero, ¿qué es ese paquete *modelos* que se ve en la Figura 2.17? Pues, es un paquete que contiene los modelos de las tablas que se usan en la aplicación.

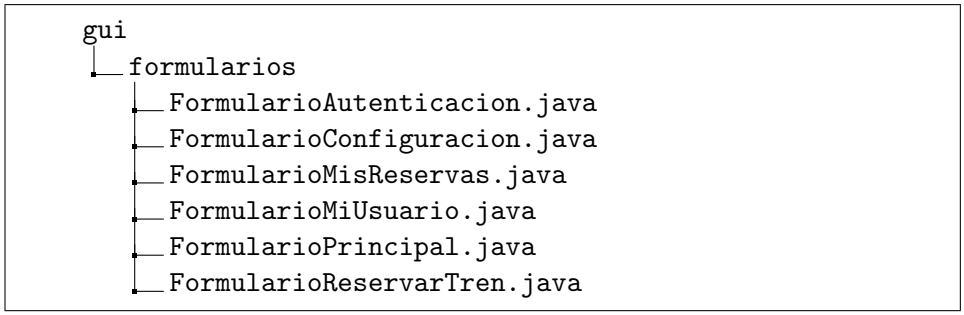

Figura 2.18: Estructura de clases del paquete *gui*

La estructura de este paquete se muestra en la Figura 2.19.

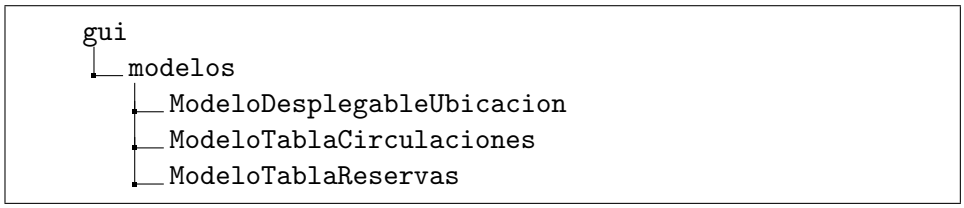

Figura 2.19: Estructura de clases del paquete *gui*

En este paquete, se encuentran los modelos de las tablas que se usan en la aplicación.

2.3.4. Estructura del paquete *util*

Por último, nos encontramos con el paquete *util*. Este paquete es trivial, ya que solo contiene clases de utilidades. Contamos tres clases en este paquete: *Criptograficos*, *Internacionalizacion* y *Ortograficos*.

Para no hacer un «look-ahead», ya hablamos ahora de sus propósitos en el capítulo de la implementación de la aplicación.

2.4. Herramientas, tecnologías y librerías

Para la implementación de la aplicación, se han usado varias herramientas, tecnologías y librerías. A continuación, se detallan las más relevantes (obviaré las más obvias, como Java 22).

Para generar los diagramas UML, JavaDocs y como IDE de desarrollo, se ha usado IntelliJ IDEA [3].

Para la gestión de dependencias, se ha usado Maven [4]. Maven es una herramienta de gestión de proyectos que se encarga de la gestión desde las dependencias hasta la compilación (y más allá).

Para la generación de datos «dummy», se ha utilizado la página web *Mockaroo* [5]. Mockaroo es una herramienta que permite generar datos de prueba de forma aleatoria.

Para la creación de este informe, se ha usado LaTeX.

Pasando a las librerías, se han usado las siguientes:

- logback [6] para el registro de eventos (logging).
- Jetbrains Annotations [7] para las anotaciones.

Finalmente, se ha hecho uso del estándar XML (eXtensible Markup Language) para el almacenamiento de los datos.

Capítulo 3

Implementación

Llegados a este punto, ya familiarizados con la estructura de clases de la aplicación, se procede a la implementación de la misma.

Como dijo el matemático Clive Humby, «Los datos son el nuevo petróleo»[8]. En este sentido, la aplicación se ha diseñado para que los datos sean el centro de la misma. Por ello, se ha implementado un patrón DAO (*Data Access Object*) para abstraer la lógica de acceso a los datos. Como ya hemos hablado de este patrón en la Subsección 2.2.2, no se repetirá la explicación aquí. Sin embargo, es un trozo de código interesante para abrir este capítulo.

3.1. Código de la aplicación

3.1.1. Implementación del DAO

Nada más empezar a analizar este paquete, resalta la existencia de una interfaz *IDAO* que define los métodos básicos de acceso a los datos. De entrar en ella, se verán dos métodos sobrecargados: *load* y *save*. Estos métodos son los que se encargan de cargar y guardar los datos, respectivamente.

Algo interesante de esta ≪sobrecarga≫ es que solo no se han implementado los métodos sin argumentos. Es interesante, porque nos permite establecer métodos obligatorios y otros métodos que se han de implementar si se desea. Por ejemplo, si se desea implementar un método *load(String file)* que cargue los datos de un archivo específico, se puede hacer. Pero si no se desea, no es necesario. He de puntualizar que, esta forma de usar las interfaces fue polémica cuando se implementó[9], pero es una forma de hacerlo que me gusta y ha llegado a ser aceptada en la comunidad[10].

Dicho esto, se puede ver que la clase *AbstractDAO* implementa la interfaz *IDAO*.

Clase AbstractDAO

Esta clase es el «esqueleto» de los DAOs. Llegando como llegamos de la interfaz, lo propio sería empezar por los métodos *load* y *save*. Estos métodos son los que se encargan de cargar y guardar los datos, respectivamente.

Para implementarlos, se ha hecho uso del patrón *Template Method*. Este patrón es muy útil para definir un esquema de algoritmo y dejar que las subclases implementen los detalles. Para ejemplificarlo, se puede ver el método *load* en la Listing 3.1.

Listing 3.1: Método *load* de la clase *AbstractDAO*

```
@Override
public boolean load() {
    XMLEventReader reader = null;
    obtenerLogger(); //Obtiene el logger
    logger.info("Cargando archivo...");

    try {
        reader = obtenerXmlEventReader();
        cargarArchivo(reader);
    } catch (CargaArchivoFallidaException e) {
        logger.error("Error al cargar el archivo", e);
        return false;
    } finally {
        try {
            cerrarArchivo(reader);
        } catch (CierreArchivoXMLErroneoException e) {
            logger.error("Error al cerrar el archivo", e);
        }
    }

    logger.info("Archivo cargado de forma satisfactoria");
    return true;
}
```

Este método tiene una lógica muy normal hasta que se llega a la llamada a *obtenerXmlEventReader()*. Si se mira el código de esta función, se verá que es abstracta. Uno podría preguntarse, ¿cómo se implementa? Pues bien, se implementa en las subclases. De esta forma, se puede tener un método que se ejecuta siempre, pero que depende de la implementación de las subclases.

Luego, se llama a *cargarArchivo(reader)*. Como la estructura de cada ar-

chivo de datos es distinta, este método también es abstracto. Así permitimos que cada subclase implemente cómo cargar sus datos mientras mantenemos una estructura común, reducimos la duplicación de código y mantenemos la cohesión. Este patrón se llama *Template Method* y es muy útil en este tipo de situaciones.[11].

Si seguimos analizando la clase, (obviando el método save, que es similar y los métodos triviales como *obtenerLogger*) llegamos al final, que es una parte interesante.

Estos métodos como *getNextXMLEvent* o *escribirElemento* son resultado de un proceso de refactorización. En un principio, estos métodos estaban en las clases hijas, pero se detectó en el proceso de codificación que estos métodos eran comunes a todas las clases hijas. Por ello, se decidió moverlos a la clase madre.

Analizar una a una las clases hijas sería tedioso y no aportaría mucho. Por ello, vamos a hacer un análisis general de las clases hijas. Estas procesan los datos XML. Se ha usado la librería *javax.xml* para procesar los datos XML.

Todas las clases hijas implementan el patrón singleton y por ello tienen un método *getInstance* que devuelve la instancia de la clase. Se puede ver un ejemplo en *DAOCirculacion*:

Listing 3.2: Método *getInstance*

```
public static volatile DAOCirculacion getInstance(FachadaDAO fadao)//
{
    if (instance != null)
    {
        return instance;
    }
    synchronized (DAOCirculacion.class)
    {
        if (instance == null)
        {
            instance = new DAOCirculacion(fadao);
        }
    }
return instance;
}
```

Respecto al resto del código, es bastante trivial. En el caso de *guardarArchivo*, se ha usado un *BufferedWriter* para escribir los datos. Con él se itera sobre los datos y se escriben en el archivo.

El caso de *cargarArchivo* es más interesante. Se ha usado un *XMLEventReader* para leer los datos. Este itera por los datos con un (o varios) ≪whiles≫ y mediante un *enhanced-switch* se procesan los datos.

En resumen, el DAO es una parte fundamental de la aplicación. Se encarga de la persistencia de los datos y de abstraer la lógica de acceso a los mismos. En este caso, se ha implementado un patrón DAO con una estructura común y clases hijas que implementan la lógica específica de cada clase.

Las clases de paquetes externos al DAO mismo, acceden a los datos mediante el patrón Façade.

3.1.2. Implementación del Façade

Las ≪façadas≫ son el ≪hilo conductor≫ de la aplicación. Todo pasa por ellas. Gracias a ellas, se puede añadir una capa de abstracción entre la GUI y el modelo de datos. Además, aumenta la encapsulación proporcionada por el MVC. Las façadas implementan también el patrón singleton.

3.1.3. Contenedores de datos

Una vez visto el DAO y las façadas, es pertinente hablar de los contenedores de datos. Estas son las representaciones de como se estructura la información en la aplicación.

Podemos distinguir dos tipos de ≪contenedores≫: los inmutables y los mutables. En nuestro caso, JAVA nos proporciona un forma de diferenciarlos: los *Records* y las *Clases*.

Records

En nuestra aplicación, como hay muchos datos de infraestructura y fechas, hemos acabado teniendo muchos tipos que no admiten cambios. De esta forma, se ha decidido usar *Records* para representar estos datos. Estas clases además de ser una ≪señal≫ de inmutabilidad, nos dan alguna que otra ventaja como, por ejemplo, que los ≪getters≫, ≪equals≫, ≪hashCode≫ y ≪toString≫ se generan automáticamente.

Clases

Al otro lado del espectro, tenemos los usuarios. Estos pueden cambiar su correo, teléfono y dirección.

3.1.4. Implementación de la Interfaz Gráfica de Usuario (GUI)

La interfaz gráfica de usuario (GUI) es la parte de la aplicación que interactúa con el usuario. Esta construída con Java Swing. Hay dos asuntos que son interesantes de comentar: la interacción con el modelo de datos y la construcción de elementos dependientes de modelos.

Interacción con el modelo de datos

La interacción con el modelo de datos se hace mediante las façadas. Estas se encargan de abstraer la lógica de acceso a los datos.

Por ejemplo, para reservar un tren, se llama al método *reservarTren* de la clase *FachadaAplicacion*:

Listing 3.3: Interacción con el modelo de datos

```
FachadaAplicacion fachada = new FachadaAplicacion(); Usuario usuario =
Tren tren = new Tren(...);
fachada.reservarTren(usuario, tren);
```

Construcción de elementos dependientes de modelos

En algunos formularios de la gui, se han usado elementos que, como almacenan información de una forma específica, necesitan de un modelo específico.

Este es el caso de, por ejemplo, las tablas. Para construir una tabla, se necesita un modelo de tabla. Los modelos de tabla son clases que extienden de *AbstractTableModel* y que se encargan de almacenar y gestionar los datos de la tabla.

3.1.5. Clases de utilidades

Las clases de utilidades son clases que no tienen estado y que contienen métodos estáticos. En nuestro caso, tenemos 3: *Criptograficos*, *Internacionalizacion* y *Ortograficos*.

En el caso de *Criptograficos*, se encarga de cifrar y descifrar contraseñas. En el caso de *Internacionalizacion*, se encarga de cargar los ficheros de internacionalización. En el caso de *Ortograficos*, se encarga de hacer operaciones de comparación de cadenas.

3.2. Problemas encontrados durante la implementación

Como en todo proyecto, se han encontrado problemas durante la implementación. Empezando por el principio, el primer problema fue la serialización y persistencia de los datos.

3.2.1. Serialización y persistencia de los datos

El primer enfoque que se adoptó para atajar este objetivo fue la serialización que java nos proporciona. Funcionaba bien, pero los archivos generados eran muy grandes y, no permitían una modificación fuera de la aplicación. Además, no son agnósticos respecto al lenguaje.

Por ello, se decidió cambiar a XML.

Lo que no se esperaba era la dificultad que iba a suponer la lectura y escritura de los datos. Como se espera trabajar con muchos datos, la carga en memoria del archivo XML no era una opción. Por ello, había que leer el archivo de forma secuencial y en streaming.

Sin embargo, tras uno o dos refactorings, se consiguió implementar la carga y guardado de los datos de forma eficiente y con un código más limpio que el de la primera implementación.

3.3. Resultados y Conclusiones

En resumen, la implementación de la aplicación ha sido un éxito. Se ha implementado un patrón DAO para abstraer la lógica de acceso a los datos. Se ha implementado un patrón Façade para simplificar la interacción entre la GUI y el modelo de datos. Se ha implementado un patrón Singleton para las clases DAO y Façade. Se ha implementado un patrón Template Method para la clase AbstractDAO y sus hijas. Se han implementado contenedores de datos inmutables y mutables. Se ha implementado la GUI con Java Swing. Se ha creado una clase para cifrar y descifrar contraseñas. Se han encontrado problemas durante la implementación, pero se han resuelto con éxito. Se ha implementado la persistencia de los datos en XML. Se ha permitido la internacionalización de la aplicación. Se ha implementado un sistema de loggin.

Toda esta funcionalidad no significa tampoco que el proyecto esté perfecto, quedan algunas cosas que habría sido interesante implementar.

3.4. Trabajo futuro

Como «deberes» hipotéticos para el futuro, se podrían implementar las siguientes funcionalidades:

- Permitir cambiar la contraseña de un usuario.
- Filtrar los trenes por fecha y hora.
- Filtrar los trenes por precio
- No mostrar los trenes que ya han pasado o que estén cancelados.
- Tal vez, migrar a una base de datos.
- De migrar a una base de datos, añadir protección contra conflictos de concurrencia.

Bibliografía

[1] Oracle, «Data Access Object,» 2024. visitado dic. de 2024. dirección: https://www.oracle.com/java/technologies/data-access-object.html.

[2] Baeldung, «Java DAO Pattern,» 2024. visitado dic. de 2024. dirección: https://www.baeldung.com/java-dao-pattern.

[3] JetBrains, *IntelliJ IDEA*, ver. 2024.3.1.1, Ultimate Edition, 2024. dirección: https://www.jetbrains.com/idea/.

[4] F. P. Miller, A. F. Vandome y J. McBrewster, *Apache Maven*. Alpha Press, 2010.

[5] Mockaroo, LLC, *Mockaroo*, Web Application, Herramienta de generación de datos de prueba, 2024. dirección: https://www.mockaroo.com.

[6] C. Gülcü y QOS.ch, *Logback*, Software Library, ver. 1.5.6, Framework de logging para Java, 2024. dirección: https://logback.qos.ch/.

[7] JetBrains, *JetBrains Java Annotations*, Software Library, ver. 26.0.1, Biblioteca de anotaciones para Java, 2024. dirección: https://github.com/JetBrains/java-annotations.

[8] N. Talagala, «Data as The New Oil Is Not Enough: Four Principles For Avoiding Data Fires,» *Forbes*, mar. de 2022. visitado 20 de dic. de 2024. dirección: https://www.forbes.com/sites/nishatalagala/2022/03/02/data-as-the-new-oil-is-not-enough-four-principles-for-avoiding-data-fires/.

[9] user3624295 et al., *Java 8: Why is it forbidden to define a default method for a method from java.lang.Object?* https://stackoverflow.com/questions/24016962/java8-why-is-it-forbidden-to-define-a-default-method-for-a-method-from-java-lan, jun. de 2014.

[10] MikeMyBytes, *No need to hate Java default methods*, https://mikemybytes.com/2021/08/05/no-need-to-hate-java-default-methods/, ago. de 2021.

[11] DigitalOcean, *Template Method Design Pattern in Java*, https://www.digitalocean.com/community/tutorials/template-method-design-pattern-in-java, Accessed: 2024-06-20, jun. de 2024.