DP1 2020-2021

Documento de Diseño del Sistema

Proyecto ***Aerolíneas AAAFC***

https://github.com/gii-is-DP1/dp1-2020-g3-3

- CAROLINA CARRASCO DIAZ
- FELIPE ESCALERA GONZALEZ
- ANTONIO JAVIER SÁNCHEZ SORIA
- ÁNGEL TORREGROSA DOMÍNGUEZ
- ANTONIO VIÑUELAS PERALES

Tutor: JOSÉ ANTONIO PAREJO MAESTRE

GRUPO G3-03

Versión 3

09/02/2021

Historial de versiones

Estos son ejemplo del contenido que debería tener el historial de cambios del documento a entregar a lo largo de los sprints del proyecto

Fecha	**Versión**	**Descripción de los cambios**	**Sprint**
13/12/2020	V1	● Creación del documento	2
5/01/2021	V2	● Incluimos primeros diagramas	3
10/01/2021	v2.1	● Fix de diagrama de diagramas ● Añadidos los primeros 3 patrones	3
09/01/2021	V3	● Añadidos el resto de patrones	4

Contents

Introducción

Aerolíneas AAAFC es una web diseñada para integrar y gestionar las áreas operativa y administrativa de la compañía. Pretendemos crear un software de gestión de aerolíneas, abarcando desde el control de vuelos y clientes hasta la administración del personal que trabaja en la misma.

Con nuestro software creado para las aerolíneas AAAFC, pretendemos brindar un control bastante completo sobre su sector. Para ello, pretendemos implantar funciones tales como las siguientes:

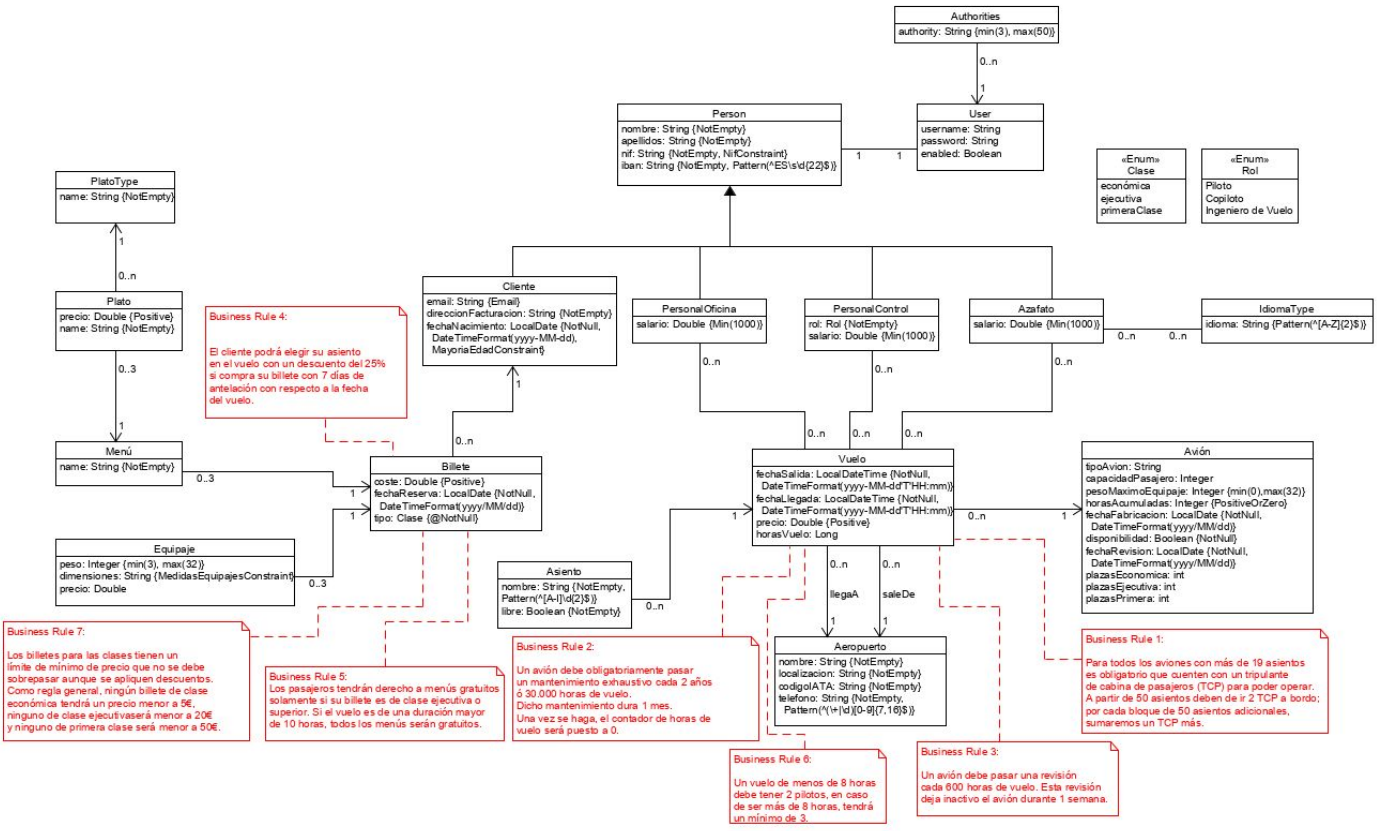
- Cálculo estimado de gastos de vuelos.
- Comunicación de los vuelos programados a los clientes.
- Gestión de la facturación y contabilidad.
- Generación de informes de cada vuelo para garantizar la seguridad.
- Gestión RRHH de todos los trabajadores de la aerolínea.
- Parte de horas o reporte de gastos, entre otras muchas funciones.
- Etc**.**

Podemos considerar como módulos interesantes del proyecto aquellos que están relacionados directamente con Vuelo o Billete. Se puede destacar también la gestión de datos que se hace sobre los usuarios (ya que necesita una validación bastante densa).

Diagrama(s) UML:

Diagrama de Dominio/Diseño

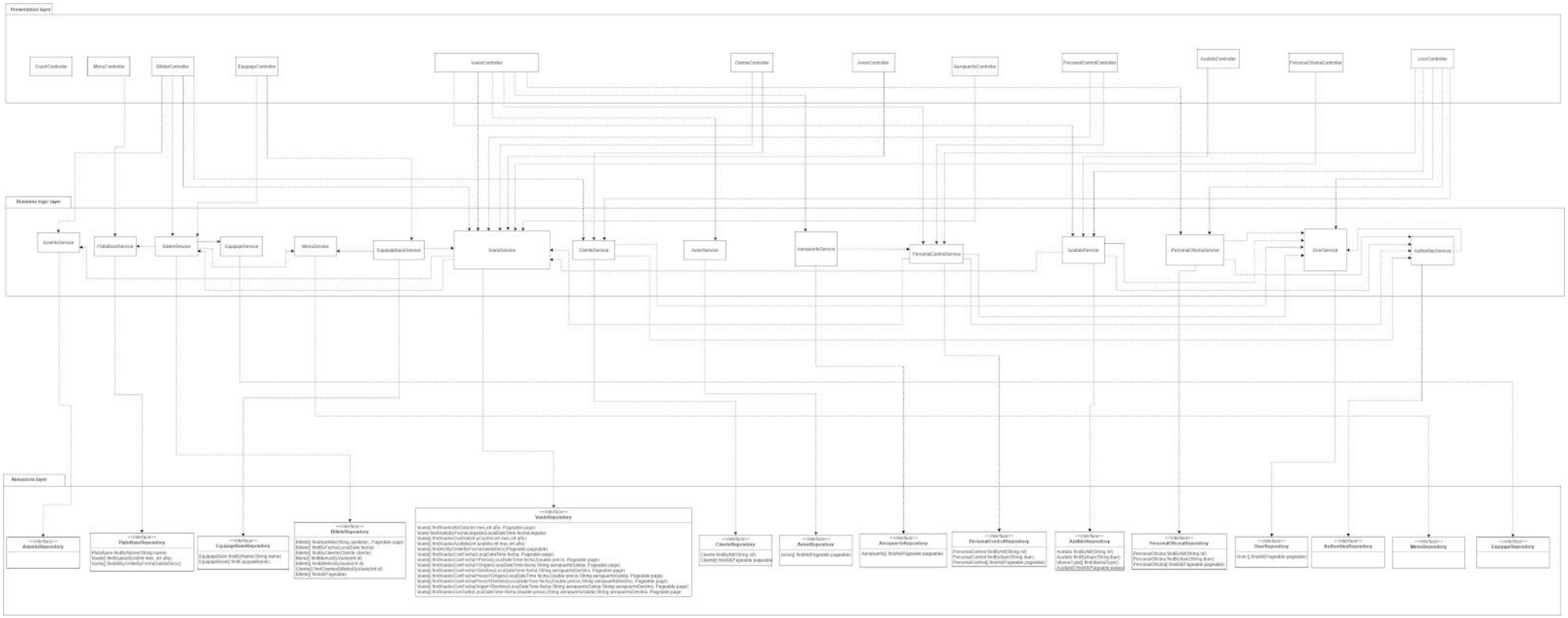
En esta sección proporcionamos el diagrama UML de diseño y de capas de nuestra aplicación (hemos omitido las generalizaciones hacia BaseEntity para simplificar el diagrama).



DP1 2020/21

Aerolíneas AAAFC **Grupo: G3-03**

Diagrama de Capas



Patrones de diseño y arquitectónicos aplicados

En esta sección especificamos el conjunto de patrones de diseño y arquitectónicos aplicados durante el proyecto:

Patrón: Capas

Tipo: Arquitectónico

Contexto de Aplicación

Este patrón ha sido aplicado en la aplicación de manera general. Se ha integrado de forma que esté dividida en diferentes capas, todas ellas conectadas entre sí.

Clases o paquetes creados

Podemos ver que en nuestro proyecto, esta división se ve en los paquetes con esta estructura:

org.springframwork.samples.aerolineasAAAFC.{web/service/service/repository/model} estos paquetes referencian a las diferentes capas del patrón (presentación, lógica de negocio y recursos respectivamente).

Ventajas alcanzadas al aplicar el patrón

En casos en los que se hace una aplicación web, es bastante interesante implementarlo. Principalmente porque hay una división cohesiva entre los diferentes módulos, cada uno se dedicará a hacer una única cosa. También es de gran comodidad porque permite el desarrollo concurrente de capas, un reemplazo más sencillo, una separación clara de responsabilidades... Es por ello que es ampliamente escogido para el desarrollo de aplicaciones web.

Patrón: MVC

Tipo: Diseño

Contexto de Aplicación

El patrón de diseño MVC ha permitido que nuestra aplicación separe por capas de vista, controlador y modelo nuestra aplicación, a través de Spring.

Clases o paquetes creados

Estas capas (como extensión al patrón arquitectónico anterior), se ven integradas en la parte web de la aplicación en /src/main/webapp/WEB-INF/jsp. Esta es la parte que verían los usuarios al entrar en la aplicación. Para gestionar todas estas vistas, habría una capa de controlador en el paquete org.springframwork.samples.aerolineasAAAFC.web, que redireccionará según la necesidad del cliente. También guardamos una capa de modelo que nos permite consultar los datos de las clases en el siguiente paquete: org.springframwork.samples.aerolineasAAAFC.model.

Ventajas alcanzadas al aplicar el patrón

El patrón fue conveniente integrarlo (aparte de las facilidades obvias que aporta) por el hecho de que actúa como una extensión de las capas, esto nos permite dar soporte a vistas diferentes; además de que favorece la alta cohesión y el bajo acoplamiento.

Patrón: Front controller

Tipo: Diseño

Contexto de Aplicación

El patrón de diseño front controller, nos permite adoptar un mecanismo centralizado para el manejo de solicitudes por un único controlador. Todo ello nos permitirá además, incluir una forma de incluir las autorizaciones de usuarios.

Clases o paquetes creados

Este diseño viene integrado con el framework de Spring a través de un gestor de dispatchers.

Ventajas alcanzadas al aplicar el patrón

Es una forma de centralizar las solicitudes que se hagan en las diferentes vistas de la aplicación. Pudiendo gestionar así la seguridad de una forma más sencilla.

Patrón: Dependency injection

Tipo: Diseño

Contexto de Aplicación

El patrón de diseño dependency injection es un patrón el cual nos permite implementar el principio de Inversión de Control. A través del patrón dependency injection se suministran objetos a una clase en lugar de ser la propia clase la que cree dichos objetos.

Clases o paquetes creados

Este diseño viene integrado con el framework de Spring a través de anotaciones como @AutoWired, @Bean...

Ventajas alcanzadas al aplicar el patrón

Es una forma de asegurar que solo se cree una instancia de una clase y de que todas las clases que lo necesitan pueden acceder a él. Permite un bajo acoplamiento entre componentes.

Patrón: Proxy

Tipo: Diseño

Contexto de Aplicación

El patrón de diseño proxy maneja las relaciones entre clases y objetos de manera sencilla proveyendo un sustituto del objeto para controlar su acceso. El objetivo de un objeto proxy es controlar la creación y el acceso al objeto real que representa.

Clases o paquetes creados

Este diseño viene integrado con el framework de Spring a través de la creación de proxies dinámicos entre la aplicación principal y toda la implementación. Sirve para inyectar los diferentes atributos, métodos, servicios... a la aplicación.

Ventajas alcanzadas al aplicar el patrón

Es una forma de mantener el objeto real fuera del alcance de los usuarios a menos que se necesite hacer un cambio sobre dicho objeto.

Patrón: Domain Model

Tipo: Diseño

Contexto de Aplicación

El patrón de diseño Domain Model nos permite crear objetos que tiene un comportamiento el cual define una capa de modelo que permite inyectar las clases @Entity a la base de datos.

Clases o paquetes creados

Cualquier entidad de org.springframwork.samples.aerolineasAAAFC.model contendrá esta jerarquía de clases. Esto es una implementación que nos vendrá dada por JPA. Todas entidades serán desarrolladas con la anotación @Entity

Ventajas alcanzadas al aplicar el patrón

Nos permite implementar una lógica de negocio más compleja, que además, está soportada por un gran número de frameworks.

Patrón: Service Layer

Tipo: Diseño

Contexto de Aplicación

El patrón de diseño service layer define los límites de una aplicación con una capa de servicios que establece un conjunto de operaciones disponibles y coordina la respuesta de la aplicación en cada operación.

Clases o paquetes creados

Este diseño viene integrado con el framework de Spring a través de la creación de servicios en el proyecto.

Ventajas alcanzadas al aplicar el patrón

Nos permite dividir la lógica de negocio en lógica de aplicación y lógica de dominio, lo que promueve una mayor separación de responsabilidades y reducir el acoplamiento de la capa de lógica de negocio.

Patrón: Meta-data Mapper

Tipo: Diseño

Contexto de Aplicación

El patrón de diseño front controller, nos permite decidir cómo se mapean las entidades creadas en nuestro sistema en la base de datos. Garantiza que la base de datos no tenga que conocer cómo funciona la lógica de negocio del sistema y que la lógica de negocio no necesite saber cómo se almacenan los datos.

Clases o paquetes creados

Cualquier entidad de org.springframwork.samples.aerolineasAAAFC.model contendrá esta jerarquía de clases. Esto es una implementación que nos vendrá dada por JPA, vendrá dado en anotaciones como @OneToOne, @ManyToMany...

Ventajas alcanzadas al aplicar el patrón

Simplifica la implementación de las relaciones entre las entidades y sus operaciones con la base de datos.

Patrón: Layer Supertype

Tipo: Diseño

Contexto de Aplicación

El patrón de diseño de Layer Supertype, nos permitirá crear clases abstractas y relacionarlas al resto de entidades implementadas.

Clases o paquetes creados

Este patrón viene dado por JPA, en notaciones como @MappedSuperclass. En nuestro proyecto podremos encontrar un ejemplo en org.springframwork.samples.aerolineasAAAFC.model, donde encontramos a la clase BaseEntity, de las que heredan el resto.

Ventajas alcanzadas al aplicar el patrón

Una de las principales ventajas es la incitación a la separación de responsabilidades. Esto modulariza el código más y no afecta de forma significativa a la mantenibilidad. Ventajoso a la hora de usar atributos comunes para todas las clases.

Patrón: Repository

Tipo: Diseño

Contexto de Aplicación

El patrón de Repository nos permite encapsular la lógica requerida para acceder a la fuente de datos. Es un artefacto intermedio entre la base de datos y la capa de servicios.

Clases o paquetes creados

Spring nos da este diseño con la notación @Repository. Podemos encontrar todos los repositorios en la ruta org.springframwork.samples.aerolineasAAAFC.repository.

Ventajas alcanzadas al aplicar el patrón

Nos permite crear una mayor centralización de la lógica de acceso a datos, de esta forma aumentamos la seguridad de la aplicación y promovemos la separación de responsabilidades. Facilita la inserción de pruebas y mantenibilidad de la aplicación.

Patrón: Eager/Lazy loading

Tipo: Diseño

Contexto de Aplicación

El patrón de diseño eager/lazy loading, nos permite decidir cómo queremos que se carguen ciertas relaciones de entidades que involucran una gran cantidad de datos. Eager loading nos permite cargar todos los datos de una relación de una entidad al principio, mientras que lazy loading sólo carga dichos datos cuando se dicha entidad los necesita.

Clases o paquetes creados

Este diseño viene integrado con el framework de Spring a través de las anotaciones @ManyToOne y @OneToOne para eager loading y @OneToMany y @ManyToMany para lazy loading por defecto.

Ventajas alcanzadas al aplicar el patrón

Es una forma de mejorar ampliamente el rendimiento de la aplicación. Usar este patrón nos permite evadir la carga de datos que no sean necesarios. Además de ello es beneficioso, porque permite generar un conjunto de datos en el momento de invocación para entidades con menos carga de memoria.

Patrón: Pagination

Tipo: Diseño

Contexto de Aplicación

El patrón de Pagination, nos permite distribuir la información de la BD en diferentes vistas, es una forma de reducir la carga de datos innecesaria. En nuestro caso, lo podremos ver para vistas como las de vuelo, cliente... que nos permite dividir los datos por páginas.

Clases o paquetes creados

En toda la extensión de la aplicación, podremos comprobar que en los repositorios de ciertas entidades como pueden ser clientes, vuelos... y en todas las listas de la aplicación (salvo las que incluyen “Details.jsp” y vueloSemana).

Ventajas alcanzadas al aplicar el patrón

Es una forma de mostrar más ordenadamente las vistas, de tal forma que, si hubiera una gran cantidad de datos, no saturaría el tiempo de carga de la página.

Decisiones de diseño

En esta sección describiremos las decisiones de diseño que se han tomado a lo largo del desarrollo de la aplicación que vayan más allá de la mera aplicación de patrones de diseño o arquitectónicos. Todas las decisiones tomadas sobre patrones venían en su mayoría, vienen en su gran mayoría integradas con la herramienta Spring Boot, es por ello que no lo incluiremos a continuación.

Decisión 1: Identidad entre nif y username

Descripción del problema:

A la hora de crear un nuevo usuario nos surgió la duda de si el atributo “username” debería de ser igual al atributo “nif” o no.

Alternativas de solución evaluadas:

Alternativa 1.a: Dejar que username no tenga que ser igual al nif.

Ventajas:

- No requiere implementación

Inconvenientes:

- Relación entre objetos menos intuitiva

Alternativa 1.b: Tener username igual al nif.

Ventajas:

- Relación entre objetos más intuitiva
- Datos de la creación autorrellenados

Inconvenientes:

- Requiere implementación
- Necesita crear notación

Justificación de la solución adoptada

Optamos por la alternativa 1.b porque es más simple a la hora de introducir los datos

Decisión 2: Creación de vuelos

Descripción del problema:

Una vez implementada de manera simple la creación y modificación de vuelos surgió la necesidad de enlazar la creación de billetes a la de vuelo.

Alternativas de solución evaluadas:

Alternativa 2.a: Utilizar el sistema tal cual para la creación de billetes.

Ventajas:

- No se precisa de otras clases.

Inconvenientes:

- Procedimiento complejo.
- Ralentiza el trabajo al tener que acceder a múltiples clases

Alternativa 2.b: Crear una clase auxiliar "asiento". A la hora de crear un vuelo se crearán todos los objetos asiento y cuando se compre un billete se podrá enlazar a este

Ventajas:

- No ralentiza el proceso de creación de billetes.
- Fácil de implementar.

Inconvenientes:

- Ralentiza la creación de vuelos.
- A la larga se saturara la base de datos con los datos de los asientos.

Alternativa 2.c: Crear una clase "asientos", en vez de crear varios objetos como en la alternativa "2.b" se creará un único objeto para cada vuelo.

Ventajas:

- No se ralentizará la creación de vuelos o billetes

Inconvenientes:

- Una implementación un poco más compleja

Justificación de la solución adoptada

Se intentó aplicar la alternativa 2.b pero nos parece más atractiva la idea de la alternativa 2.c y pues no vamos retrasados con la implementación del resto de historias de usuario

Decisión 3: Cambio de relaciones entre un vuelo y su tripulación

Descripción del problema:

Tras estructurar la lógica de nuestra aplicación y ponernos a trabajar sobre las historias de usuario, en aquellas en las que se debía acceder a un vuelo, se debían obtener los atributos de su tripulación a través de la entidad avión, y esto se traducía en una complicación.

Alternativas de solución evaluadas:

Alternativa 3.a: Crear una clase auxiliar que relacione los vuelos con su tripulación.

Ventajas:

- No hay que cambiar la lógica de relaciones ya implementada.

Inconvenientes:

- Mayor cantidad de clases.
- Ralentiza el trabajo al tener que acceder a múltiples clases.

Alternativa 3.b: Utilizar la lógica ya implementada y modificar las relaciones de tablas.

Ventajas:

- Facilita el acceso a las clases para la implementación de las historias de usuario.
- Fácil de implementar.

Inconvenientes:

- Se debe revisar el código ya producido y testeado.

Justificación de la solución adoptada

Se adoptó la segunda solución (3.b), ya que a pesar de tener que revisar el código ya implementado, la carga de trabajo era mucho menor y más asequible.

Decisión 4: Refactorización completa de menú

Descripción del problema:

En la segunda reunión con el tutor, sugirió la refactorización de la clase menú. Esta era totalmente funcional pero estaba implementada de forma que no seguía las buenas prácticas.

Alternativas de solución evaluadas:

Alternativa 4.a: *Dejar la implementación actual*.

Ventajas:

- No hacía falta cambiar más las clases.
- A corto plazo es altamente mantenible.

Inconvenientes:

- Muy alto acoplamiento y baja cohesión.
- Poco eficiente.
- Mantenimiento muy complejo a largo plazo.
- Dificulta la integración de otros módulos

Alternativa 4.b: Refactorizar el menú para crear subclases (PlatoBase, Plato, PlatoType) que ayuden a la gestión de sus datos. Menú ahora tendrá un atributo por cada plato.

Ventajas:

- Facilita la integración de platos y tipos futuros (sólo hace falta incluirlo en la base de datos).
- Algo más fácil de implementar que la alternativa 4.c.
- Más eficiente.
- Más legible.

Inconvenientes:

- A pesar de todo es bastante complicado de implementar (se debe manejar muchas subclases).
- No facilita la inserción de más platos por menú.

Alternativa 4.c: Refactorizar el menú para crear subclases que ayuden a la gestión de sus datos. Menú ahora tendrá un atributo que .

Ventajas:

- Facilita la integración de platos y tipos futuros (sólo hace falta incluirlo en la base de datos).
- Más eficiente.
- Más legible.

Inconvenientes:

- Implementación muy compleja.
- Facilita la inserción de más platos muy levemente.
- La inserción de más platos por menú no es necesaria.

Justificación de la solución adoptada

Se adoptó la segunda solución (4.b), ya que era la que mejor se adapta a una situación real y facilita los cambios futuros ampliamente.

Decisión 5: Refactorización completa de equipajes

Descripción del problema:

En la segunda reunión con el tutor, sugirió la refactorización de la clase menú. Esto llevó a cambiar también el planteamiento sobre la clase equipaje, ya que seguía el mismo modelo de implementación.

Alternativas de solución evaluadas:

Alternativa 5.a: *Dejar la implementación actual*.

Ventajas:

- No hacía falta cambiar más las clases.
- A corto plazo es altamente mantenible.

Inconvenientes:

- Muy alto acoplamiento y baja cohesión.
- Poco eficiente.
- Mantenimiento muy complejo a largo plazo.
- Dificulta la integración de otros módulos

Alternativa 5.b: Refactorizar el equipaje para que ahora incluya la referencia a un subtipo (EquipajeBase), facilitando así la inserción de nuevos tipos de equipaje.

Ventajas:

- Facilita la integración de diferentes equipajes (con una inserción en la base de datos serviría).
- Más eficiente.
- Más legible.

Inconvenientes:

- Es dificultoso implementarlo.

Justificación de la solución adoptada

Se adoptó la segunda solución (5.b), ya que las ventajas de implementarla, superaban ampliamente a las desventajas.

Decisión 6: Soporte de Reglas de Negocio

Descripción del problema:

A la hora de aplicar las reglas de negocio en el sistema, aparecen varias alternativas las cuales tienen sus ventajas y desventajas.

Alternativas de solución evaluadas:

Alternativa 6.a: *Clases de validación personalizadas*

Ventajas:

- Fácil de desarrollar en Spring.
- Alta cohesión cuando sólo hay 1 regla de negocio por entidad.

Inconvenientes:

- Complejo si hay más de 1 regla de negocio por entidad.
- Dado que la validación no se hace en la entidad, deben ser añadidos en cada controlador que esté involucrado con dicha entidad.
- Todas las "*simple constraints*" deben de ser incluidas dentro de la clase de validación.

Alternativa 6.b: Excepciones

Ventajas:

- Manejo de errores sencillo.
- Mensajes más detallados en caso de error.

Inconvenientes:

- Si se usan varias excepciones, a la hora de mostrar los errores, solo se muestra una.
- Código más complejo al haber muchos bloques try-catch.
- Puede guardar datos inconsistentes en la base de datos.

Alternativa 6.c: Anotaciones de validación personalizadas

Ventajas:

- Fácil integración con Spring, ya que no requiere modificaciones en los controladores.
- Se verifica en la capa de presentación y de lógica de negocio, por lo que protege a la base de datos de guardar datos erróneos.
- Alta cohesión.

Inconvenientes:

- Más complejo.

Justificación de la solución adoptada

Se adoptó la tercera solución (6.c) y la segunda solución(), debido a que tenemos muchas entidades con varias reglas de negocio y había algunas reglas de negocio que era más sencillo hacerlo por excepciones. Reglas de negocio hechas con anotaciones personalizadas: RN1, RN6. Reglas de negocio hechas con excepciones: RN2, RN3, RN4, RN5, RN7.

Decisión 7: Soporte de logs

Descripción del problema:

A la hora de mantener un registro de lo que pasa en nuestro sistema, tuvimos varias ideas de dónde aplicar logs y su política de eliminación.

Alternativas de solución evaluadas:

Alternativa 7.a: *Logs solo para errores.*

Ventajas:

- Podemos ver los fallos del sistema fácilmente.
- Fácil de implementar.

Inconvenientes:

- No hay registro de nuevas entidades y sus respectivas modificaciones.

Alternativa 7.b: *Logs para creación, actualización y errores.*

Ventajas:

- Podemos ver todo lo que pasa en el sistema fácilmente, a través de los niveles INFO, WARN y ERROR.

Inconvenientes:

- Más difícil de implementar.
- Mucha en el archivo log.

Justificación de la solución adoptada

Se adoptó la segunda solución (7.b) debido a que establecimos una política de logs que al llegar el archivo a 80MB de tamaño cambie a otro archivo log. Así podemos tener constancia de todo lo que ocurre en el sistema.

Decisión 8: Refactorización de Idiomas en azafatos

Descripción del problema:

Al introducir los idiomas en azafatos no era óptimo introducir manualmente el idioma, ya que hay muchas formas de entenderlo (español se puede introducir como Español...) y podría generar atributos que expresan lo mismo pero de diferente forma, provocando una baja lógica en la BD.

Alternativas de solución evaluadas:

Alternativa 8.a: *Dejar la implementación actual*.

Ventajas:

- Dejar la relación como un Set<Idioma> facilitaba mucho las tareas de comprobación.
- La complejidad es mínima, sólo habría que comprobar que el set tiene un getSize() de 2 idiomas para validarlo.

Inconvenientes:

- Mantenimiento complicado.
- Introducción de datos inconsistentes en la base de datos.
- Rendimiento muy bajo, se introducen idiomas cada vez que se produce un alta.

Alternativa 8.b: Refactorizar los Idiomas para que este pase a ser un tipo (de formato ISO “XX”), habiendo sólo un número determinado de idiomas en la BD.

Ventajas:

- Consistencia de datos alta.
- Más eficiente.
- Más legible.
- Más mantenible

Inconvenientes:

- Requiere un Formatter.

Justificación de la solución adoptada

Se adoptó la segunda solución (8.b), ya que implementar esta alternativa, suponía mejorar el rendimiento y consistencia de nuestra BD, facilitando la información de manera más intuitiva.