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 2.1

10/01/2021

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

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).

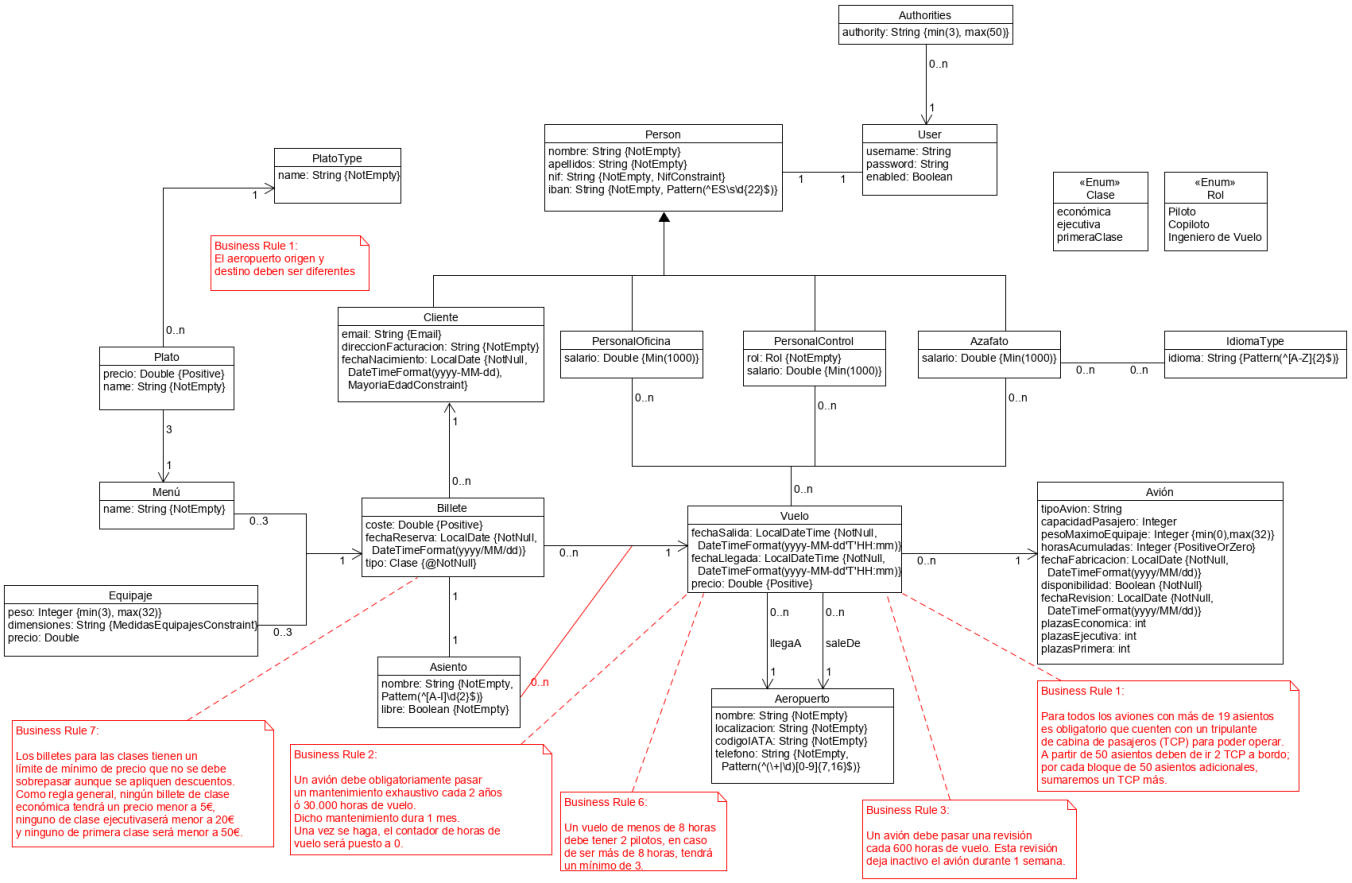
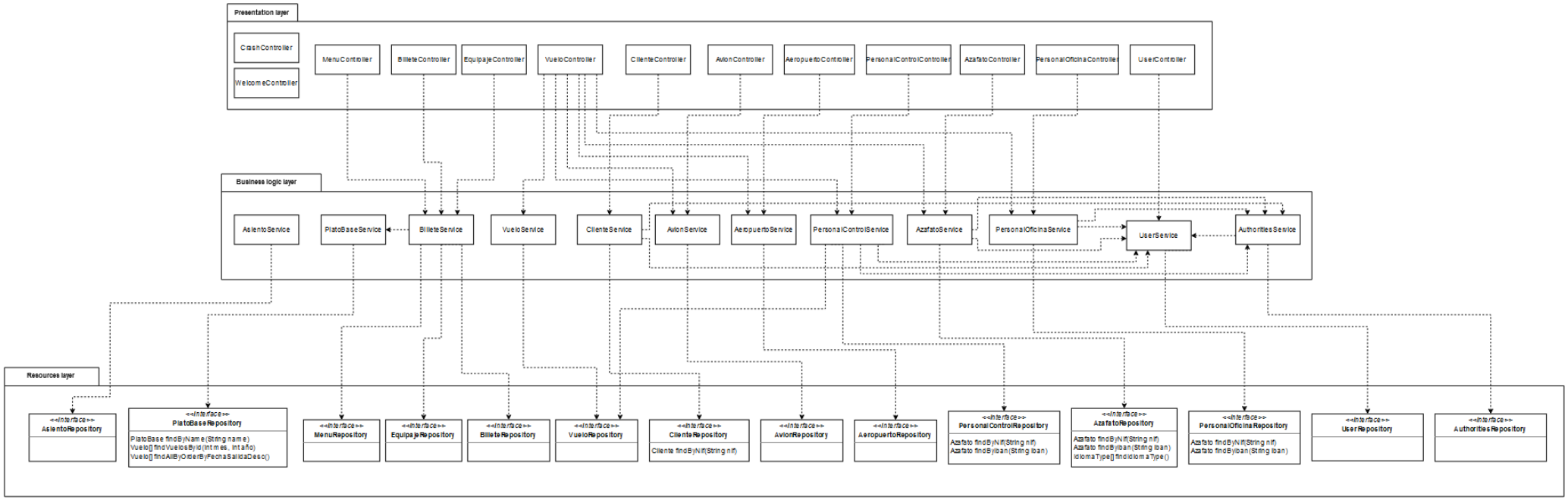


Diagrama de Capas



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.

Decisiones de diseño (de forma individual)

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.

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 4.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.