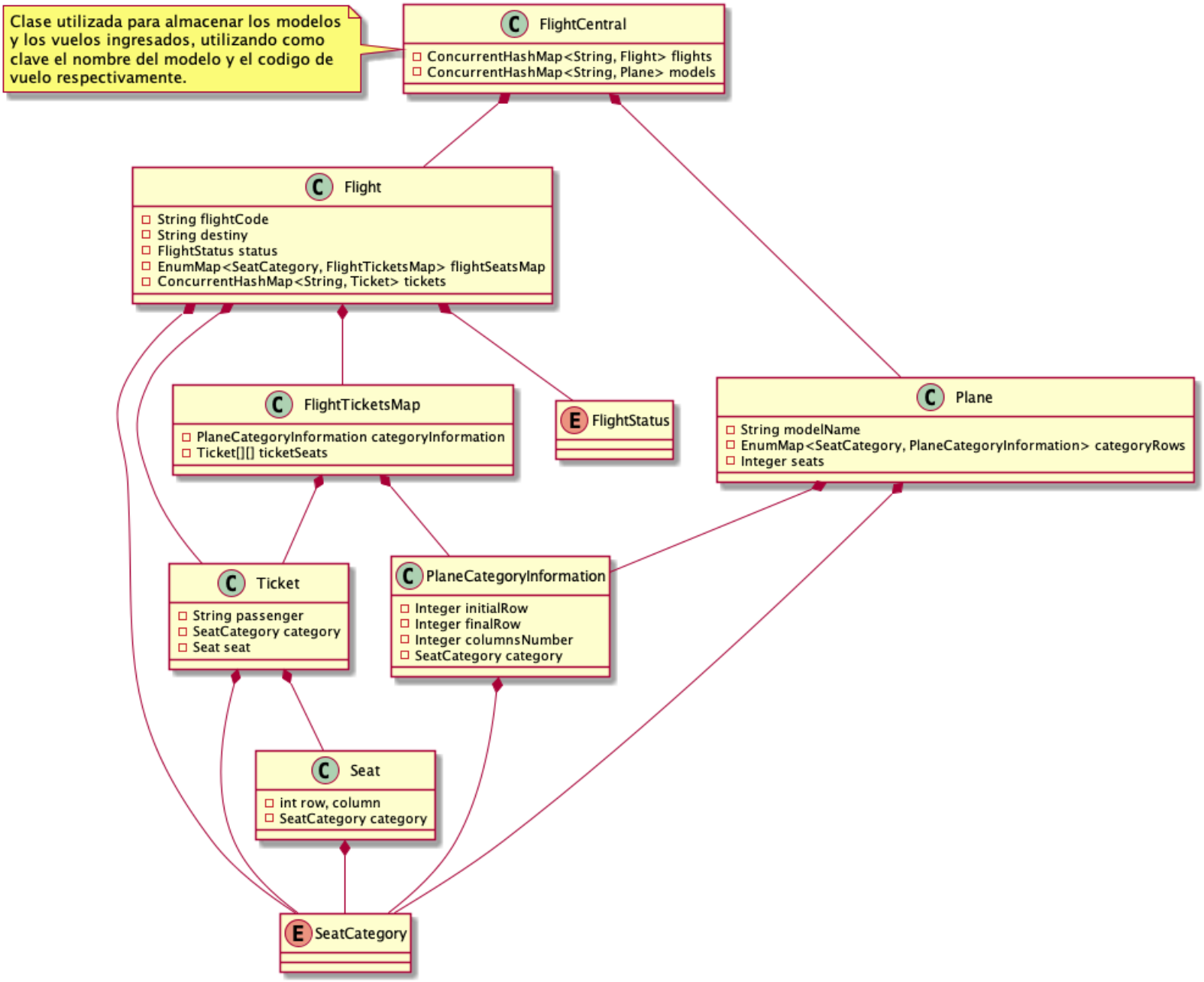
Grupo 14
Integrantes: Gaspar Budó Berra, Marcos Dedeu, Santiago Hadad, Bruno Squillari y Facundo Zimbimbakis

Decisiones de diseño e implementación de los servicios

Diseño de las clases utilizadas para llevar a cabo los servicios.



Todo comienza con la creación de los Planes y los Flights cuyos responsables son los métodos de agregar un modelo y un vuelo respectivamente. Si estos objetos son creados exitosamente, se agregan a su respectivo mapa de la clase FlightCentral.

La estructura de los modelos se almacena en un mapa utilizando como clave la categoría de los asientos y como valor, la clase llamada PlaneCategoryInformation. Esta clase nos fue útil para identificar desde que fila iniciaba y terminaba cada categoría presente en un modelo.

Por otro lado, los vuelos almacenan su estructura de forma similar a los modelos. Se tiene una clase llamada FlightTicketsMap, la cual contiene la información de la categoría sumada a una matriz de Ticket compuesta por la cantidad de filas y columnas que tiene esa categoría en ese modelo de avión. Esta matriz nos fue útil a la hora de consultar qué asientos se encontraban ocupados o libres.

En referencia a las notificaciones, decidimos almacenar los usuarios registrados en el siguiente mapa:

```
ConcurrentHashMap<Flight, List< Pair<String, Notifier> >>registeredUsers
```

Así tenemos cada pasajero registrado en una lista vinculada a un vuelo, para poder facilitar cada método que involucre a un vuelo completo, como por ejemplo: la confirmación o cancelación del mismo. También, cada pasajero tiene vinculada la implementación del Notifier para poder recibir las notificaciones de forma personalizada.

Criterios aplicados para el trabajo concurrente

A la hora de hacer que el servidor sea thread-safe, decidimos utilizar la clase ConcurrentHashMap para todo el manejo de mapas ya que esta clase se encarga por sí sola el manejo sincronizado en cada acción realizada.

Por otro lado, a la hora de hacer modificaciones en un vuelo ya sea, agregar tickets, cambiar su estado, asignar un asiento o mover un pasajero de asiento, tuvimos que tomar la decisión de realizar Locks utilizando como mutex el código de vuelo para garantizar que los datos sean consistentes en todos los métodos.

Otro problema que surgió fue a la hora de cambiar el ticket de un pasajero a un vuelo alternativo debido a que estaban involucrados dos vuelos distintos, por ende, como mencionamos anteriormente para cada acción en un vuelo determinado se utiliza un Lock por su código de vuelo. Es decir, debíamos usar dos Locks anidados y es así como se presentó la posibilidad de que ocurra un abrazo mortal. En consecuencia, la mejor solución presentada fue ordenar ambos códigos de vuelo alfabéticamente para que siempre se realicen ambos locks en el mismo orden y así evitar un potencial abrazo mortal.

Una de las soluciones que nos propusimos para garantizar el trabajo concurrente en este método implicaba realizar un Lock completo de todo el servidor a la hora de ejecutar el re-ticketing. Esto se debía a que podría ser un problema que durante el movimiento de pasajeros un vuelo se quede sin asientos disponibles (cambiando entonces el orden de prioridad del mismo para la reasignación) o incluso podría cancelarse/confirmarse.

Potenciales puntos de mejora y/o expansión

Para el futuro nos gustaría poder agregar distintas aerolíneas las cuales contendrían sus propios modelos de avión y sus propios vuelos.

Otra funcionalidad que nos gustaría que tenga nuestro servidor es la posibilidad de agregar un nuevo ticket de un nuevo pasajero a un vuelo (en este momento los tickets se agregan solamente a la hora de cargar un vuelo). También estaría bueno poder garantizar la imposibilidad de una sobreventa de un vuelo si es que agregamos la funcionalidad mencionada anteriormente.

Por último, queremos mencionar una posible mejora para el futuro la cual consiste en poder agregar dos pasajeros con el mismo nombre a un mismo vuelo. Esto podría manejarse por número de DNI o algo por el estilo.