**Universidade do Minho**
Escola de Engenharia
Licenciatura em Engenharia Informática

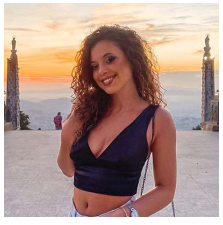
## Unidade Curricular de Sistemas Distribuídos

Ano Letivo de 2021/2022

# Sistema de gestão de reservas de voos

**Grupo 12**


62608 - Marco Sousa


93198 - Mariana Marques


93271 - José Malheiro


94269 - Miguel Fernandes

SD

14 de Janeiro de 2022

# Resumo

A crescente procura por serviços cliente - servidor, obrigada a que um sistema seja capaz de responder a múltiplos pedidos de forma concorrente. Como consequência, surge a necessidade de moldar o sistema para que este tenha um comportamento determinístico, independentemente da circunstância de utilização. Assim, na sequência da Unidade Curricular de **Sistemas Distribuídos**, surge a oportunidade de desenvolver uma aplicação com múltiplos clientes e um servidor, ambos *multi-threaded*.

O projeto proposto corresponde a um sistema de reservas de voos, denominado *Flight Manager*. Este, conforme referido, será capaz de ter um servidor que responde a múltiplos clientes, de forma concorrente. A linguagem de programação utilização foi ***Java***, com recurso a *Socket* TCP para efetuar a comunicação entre cliente-servidor.

**Área de Aplicação**: Sistemas Distribuídos

**Palavras-Chave**: Sistemas Distribuídos, *Threads*, *Multithreaded*, *Middleware*, Servidor, Cliente, Serialização, *Data Transfer Object*, *Sockets*, TCP, *Tagged Connection*

# 1 Introdução

O presente relatório foi escrito no âmbito da Unidade Curricular de Sistemas Distribuídos (SD), sendo o tendo como principal objetivo apresentar uma possível solução para um Sistema de Gestão de Reservas, que iremos designar *Flight Manager*, capaz de responder às funcionalidades propostas pela equipa docente.

Assim, será apresentado um sistema *cliente-servidor* que, do lado do servidor, será capaz de atender múltiplos clientes e, do lado do cliente, será capaz de comunicar com o servidor e executar métodos assíncronos e síncronos.

Para dar resposta ao proposto, utilizou-se um componente comum a ambos os sistemas, um **Middleware**, que servirá de ponte de ligação entre os *hosts*. O objeto de comunicação, será um *Data Transfer Object* e, a partir deste, será derivado múltiplos objetos relacionados com pedidos e respostas.

Com o desenvolvimento deste projeto, pretende-se apromirar vários conceitos de Sistemas Distribuídos, tais como:

- Programação com *Sockets*
- Programação com múltiplas *Threads* (*MultiThread*)
- Programação Concorrente (estado partilhado)
- Exclusão Mútua
- Serialização de Objetos
- Arquitetura Cliente-Servidor
- Desenvolvimento de *Middleware*

# 2 Análise de Requisitos

O enunciado proposto, remete à conceção e implementação de uma plataforma para gestão de reservas de voos, apresentando um conjunto de requisitos funcionais que o sistema deve responder.

A partir do enunciado, identificou-se dois tipos de utilizadores: cliente e adminstrador. Cada um terá um conjunto de funcionalidades associadas, tendo sido captadas num Diagrama de Use Case (ver 3.1).

Assim, a nível estrutural, definiu-se três componentes principais:

**Cliente** Interface de interação com o utilizador para utilizar as funcionalidades definidas

**Middleware** Ponte de comunicação entre o cliente e o servidor

**Servidor** Responsável pelo tratamento dos pedidos do cliente

Definida a estrutura e as funcionalidades, passou-se para a modelação comportamental do sistema.

## 2.1 Modelação Comportamental

Por forma a permitir uma implementação sustentada, optou-se por efetuar a modelação comportamental do sistema. Para isso, começou-se por definir a forma como o cliente e o servidor iriam comunicar. Esta revelou-se como uma das etapas fundamentais para o funcionamento da aplicação, tendo como perspetiva a sua escalabilidade e manutenção.

### 2.1.1 Comunicação

A troca de mensagens entre o cliente e o servidor será mediada por um **Middleware** (ver 3.3). Pode-se encontrar um esboço da estratégia analisando o diagrama 3.2.

#### Middleware

No *middleware* (ver 3.3), haverá especialização do lado do cliente, por forma a receber respostas e enviar pedidos e do lado do servidor, para receber pedidos e enviar respostas. Esta especialização, serve apenas para facilitar a implementação, pois, na realidade, o objeto de comunicação será sempre o mesmo: uma `Frame` que terá no seu corpo informação sobre o pedido, nomeadamente uma *tag* que a identifica e um *Data Transfer Object* (DTO). Este último, será um dos principais atores na troca de informação, pois a partir da sua especialização, será possível criar cada um dos pedidos e cada uma das respostas, conforme o conteúdo necessário. Com o objetivo de generalização do código, colocou-se que a classe *DTO* será abstrata e terá como método, também este abstrato, a serialização (*serialize*).

### 2.1.2 Servidor

Dotado de um *ServerSocket*, este será capaz de estabelecer ligação com cada cliente, sendo-lhe atribuída uma *Thread* própria, partilhando um estado comum com arquitetura **Facade**: `IFlightManager` (ver 3.4). Neste, poderá identificar-se dois subsistemas: *Booking Manager* e *User Manager*.

Utilizou-se os princípios *SOLID* [1] ao longo da implementação de todo o código, assim como o seu planeamento.

#### BookingManager

De modo a ter um desenvolvimento mais estruturado, aplicando recursos disponibilizados através de outras Unidades Curriculares, foram previamente construídos diagramas para os principais elementos do sistema.

A partir das funcionalidades mencionadas no enunciado do trabalho proposto, foram distribuídas as opções para os utilizadores da aplicação, seguindo a conceção do sistema.

Esta, divide-se em três grandes componentes:

Cliente Responsável pela apresentação e envio dos pedidos disponíveis do Cliente.

Servidor Responsável pela resposta por parte do Servidor.

Middleware Construtor das *Querys* e *Responses*, bem como gere o suporte para a conexão.

Considerou-se mais intuitivo previamente apresentar o modo como a conexão do Servidor/Cliente é geridas e o tipo de mensagens criadas, sendo a primeira grande componente da aplicação a ser apresentada o: *Middleware*.

### 2.1.3 Middleware

*Middleware*, como o nome indica, agrega toda a dinâmica que acontece entre o Servidor e o Cliente. Utilizamos no sentido de mover ou transportar informações e dados entre programas de diferentes protocolos de comunicação e dependências do sistema operacional.

Assim, podemos definir como a comunicação entre o Servidor e o Cliente vai ocorrer.

#### Gestão da Conexão

A gestão desta conexão para suporte de diferentes padrões de interação por processos *multithreaded* é do âmbito da *Tagged Connection* utilizada muito efetivamente durante as aulas.

```
public class TaggedConnection implements AutoCloseable {

    private Socket s;
        protected DataInputStream in;
        protected DataOutputStream out;
        protected Lock sendLock = new ReentrantLock();
        protected Lock rcvLock = new ReentrantLock();
}
```

Encontra-se definido o *socket* a usar, usando **TCP**, *streams* para permitir a serialização e deserialização das mensagens entre os programas, e *Locks*, para permitir vários pedidos e respostas serem efetuados concorrentemente, com o uso de *Threads*.

Foi necessária a distinção entre a conexão por parte do Cliente e do Servidor, sendo criadas a:

- *Server Connection*
- *Client Connection*

A razão para esta divisão foi devido ao método usado para o controlar as funcionalidades disponíveis aos utilizadores **administrativos** e **comuns**, auxiliando na autenticação destes no sistema. Cada cliente após se registar, obtém um *Token* próprio e único a sí, *encoded*. Usando a biblioteca **JWT**, foi possível facilmente manusear este processo, e em cada query efetuada pelo cliente, é inserida à cabeça o *token* respetivo, para parametrizar os utilizadores que estão a utilizar o sistema, e impedir que ocorram erros ao nível de funcionalidade administrativas erradamente disponíveis a utilizadores comuns.

Neste sentido, o *Client Connection*:

```
public class ClientConnection extends TaggedConnection {
  private String token;
}
```

### Serialização

Com a conexão entre o Servidor e o Cliente estabelecida, é de maior importância definir o tipo de mensagens a serem trocadas entre ambos, como um **protocolo**.

Antes do estado atual do sistema, encontrava-se definido que todas as mensagens a serem transmitidas eram do tipo *String*, sendo que facilitava a construção do programa, dado à sua simplicidade de manuseamento.

Contudo, eram claras as limitações causadas por esta construção. Neste sentido, foi estabelecido uma dinâmica usando **DTOs**, ou *Data-Transfer Objects*, trazendo um alto nível de flexibilidade ao sistema.

#### →DTO

Para o transporte de dados entre diferentes componentes, instâncias ou processos de um sistema distribuído ou diferentes sistemas via serialização, usa-se, caso da linguagem *Java*, o conceito de um *Data-Transfer Object*.

Neste caso, para serializar e deserializar as *Queries* e *Responses* do cliente e servidor, respetivamente, foi criado para cada uma o seu próprio DTO.

Com o tipo primitivo **DTO**,

```
public abstract class DTO {
    public abstract void serialize(DataOutputStream out) throws IOException;
}
```

foram criados, adionalmente, nas suas *packages* respetivas (*QueryDTO* e *ResponseDTO*), ficheiros que controlam esta dinâmica de serialização/deserialização para cada um. Permitindo que, no futuro, com a adição de novas funcionalidades, seja apenas necessária a adição dos seus *DTOs* e não haja

H

Figura 2.2: Diagrama de comuniações para o *Middleware*

uma reestruturação do código.

→**Frame**

Encontra-se na classe *Frame*, o protótipo das mensagens enviadas.

```
public class Frame {

    public final int tag;
    private final String className;
    private DTO dto;
}
```

Com esta definição, explica-se o nome atribuído à conexão do Servidor/Cliente - *Tagged Connection*, dado às mensagens serem etiquetadas, neste caso, através da variável *Tag*. Possui o nome da classe (*QueryDTO* e *ResponseDTO*) que está a enviar e o respetivo DTO.

#### Outros aspetos

Encontram-se, também, dentro do *Middleware*, várias exceções, para permitir que sejam lidadas ao nível do *UI*.

Assim, como mencionado na estratégia utilizada (Ver 2.1), a conceção deste componente partiu de um diagrama, sendo este um **Diagrama de Classes**(Ver 2.1.3).

Figura 2.1: Diagrama de classes para o *Middleware*

Adicionalmente, foi construído um **Diagrama de Comunicações**, para definir a dinâmica (Ver 2.1.3).

## 2.2 Cliente

Apresentar o Client class diagram do VPP.

### 2.2.1 Interface com o Utilizador

Apresentar os menus com os utilizadores.

## 2.3 Servidor

Apresentar o Server class diagram do VPP.

### 2.3.1 FlightManager

#### Booking Manager

#### User Manager

### 2.3.2 ServerWorker

### 2.3.3 Apresentação do servidor

## 2.4 Funcionalidades Adicionais

No seguimento da realização das funcionalidade obrigatórias pré-definidas pela equipa docente, foi considerada a adição de aspetos adicionais, de modo a enriquecer a resposta previamente construída.

# 3 Considerações Finais

## 3.1 Conclusões

Ao longo do desenvolvimento do projeto, e com o auxilio da equipa docente para qualquer questão relativa à matéria lecionada e à conceção do trabalho, o grupo foi munido com todos os conceitos e materiais necessários para o bom desenvolvimento do sistema, *Flight Manager*.

A partir de conceitos de outras unidades curriculares, o grupo foi possibilitado a resolver uma solução outrora complexa, a partir de um estruturamento planificado e modelado, usando para tal o sistema de modelação *Visual Paradigm*. Este permitiu, muito mais facilmente, fragmentar o que era necessário resolver para colocar o sistema a correr, poupando muito tempo que podia ser perdido, ao construir a aplicação "à mediada que fazia".

O grupo encontra-se orgulhoso por poder implementar um sistema que não só responde às funcionalidades básicas estabelcidades pela equipa docente, com uma plataforma bem construída do ponto de vista de encapsulamento e organização dos vários constituintes.

Escusado é dizer que muniu todos os participantes com experiência na realização de projetos que sigam os conceitos de **Sistemas Operativos**, nomeadamente a exclusão mútua, a programação com *sockets* e com várias *Threads*.

## 3.2 Trabalho Futuro

Apesar de todo o processo de desenvolvimento do sistema *Flight Manager* ter coincidido com as espectativas colocadas pelo grupo, e, até permitido adicionar funcionalidades numa tentativa de obter um melhor aproveitamento, existem alguns pontos que poderiam ser abordados numa iteração seguinte do **Flight Manager**.

Como qualquer sistema que visa a interação com utilizadores sem conhecimentos dentro da área de informática, o manuseamento de um terminal para utilizar a plataforma pode tornar-se incómodo. Dado a aplicação querer satisfazer as necessidades de utilizadores comuns, seria de extrema importância a conceção de uma interface com o utilizador intuitiva e simples de usar. Assim, um dos problemas seria a criação de uma *framework* conceptual para o **Flight Manager**.

Outra equívoco seria a adição de mais funcionalidades. O aperfeiçoamento da aplicação, de modo a chegar a um estado competitivo para o mercado atual, passaria pela implementação de várias e diferentes funcionalidades que o distingue dos restantes produtos análogos.

# Bibliografia

[1] *SOLID - Wikipedia*. URL: https://en.wikipedia.org/wiki/SOLID.

# Anexos

## Anexo 1 - Diagrama de Use Case

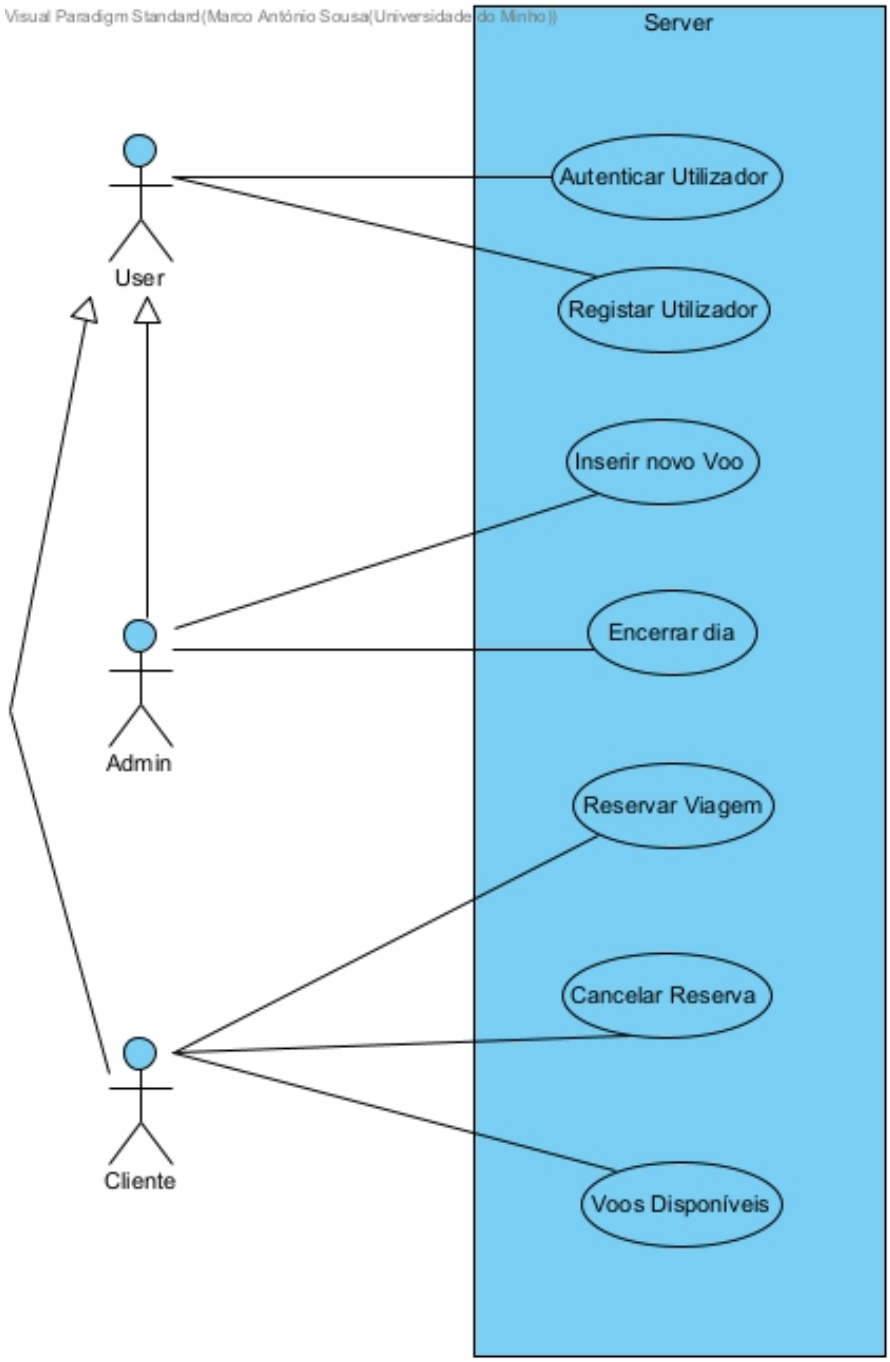



Figura 3.1: Diagrama de Use Case

# Anexo 2 - Diagrama de Comunicação

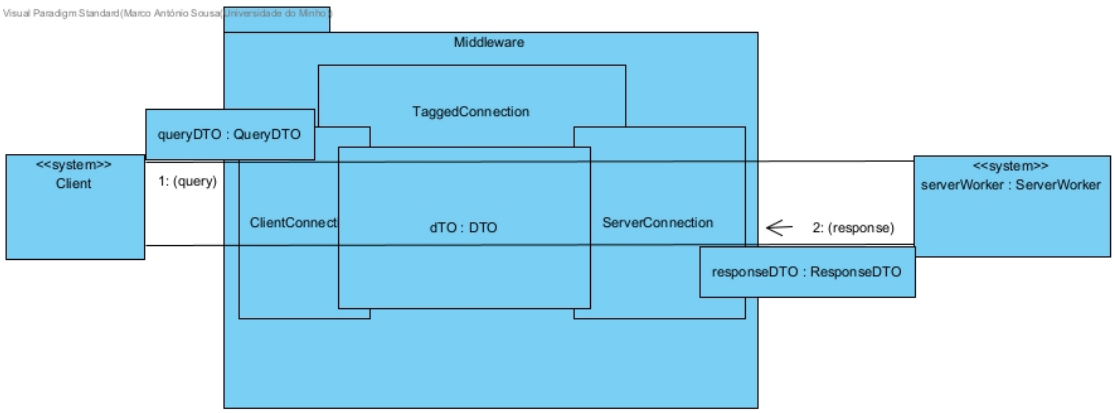



Figura 3.2: Diagrama de Comunicação - versão $\beta$

# Anexo 3 - Diagrama de Classes do Middleware

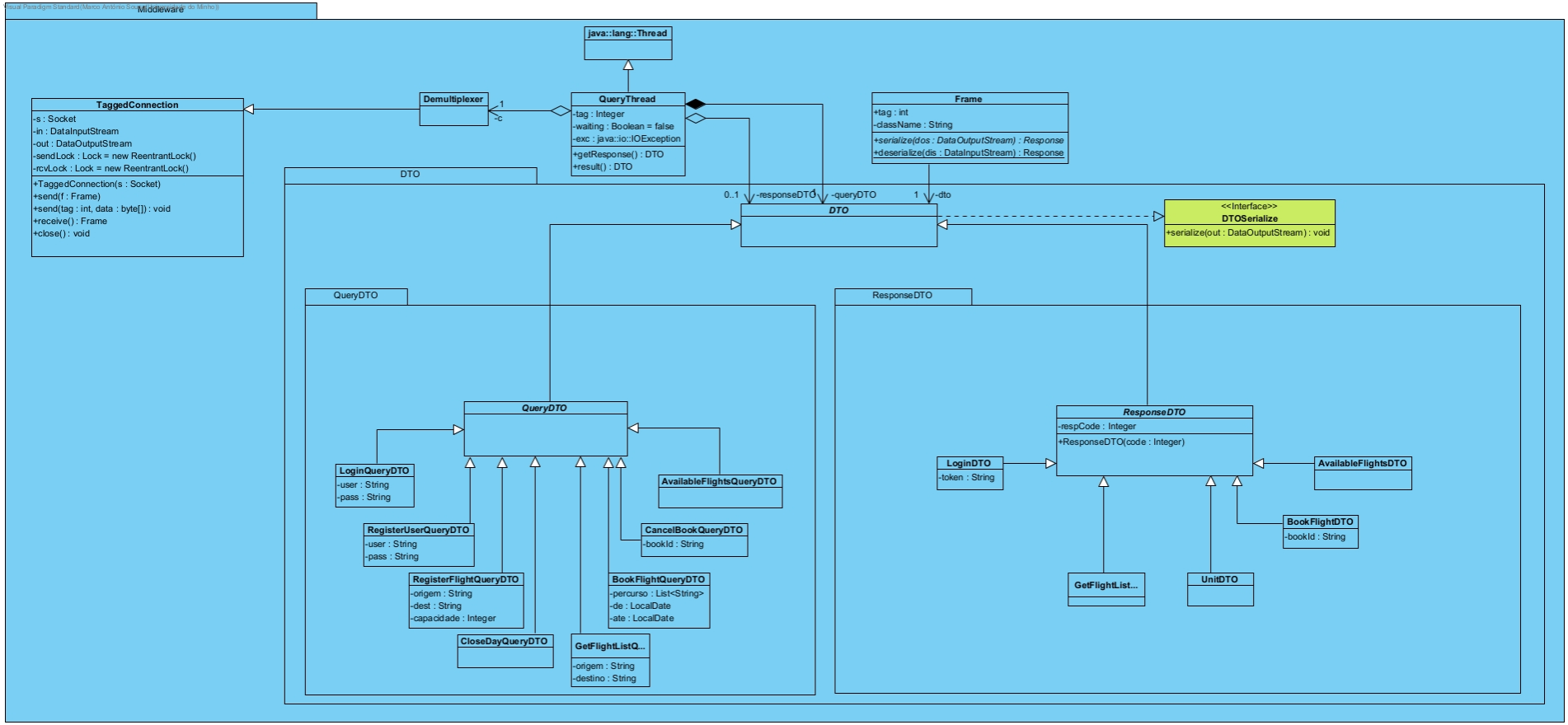



Figura 3.3: Diagrama de Classes - Middleware

# Anexo 4 - Diagrama de Classes do FlightManager (Servidor)

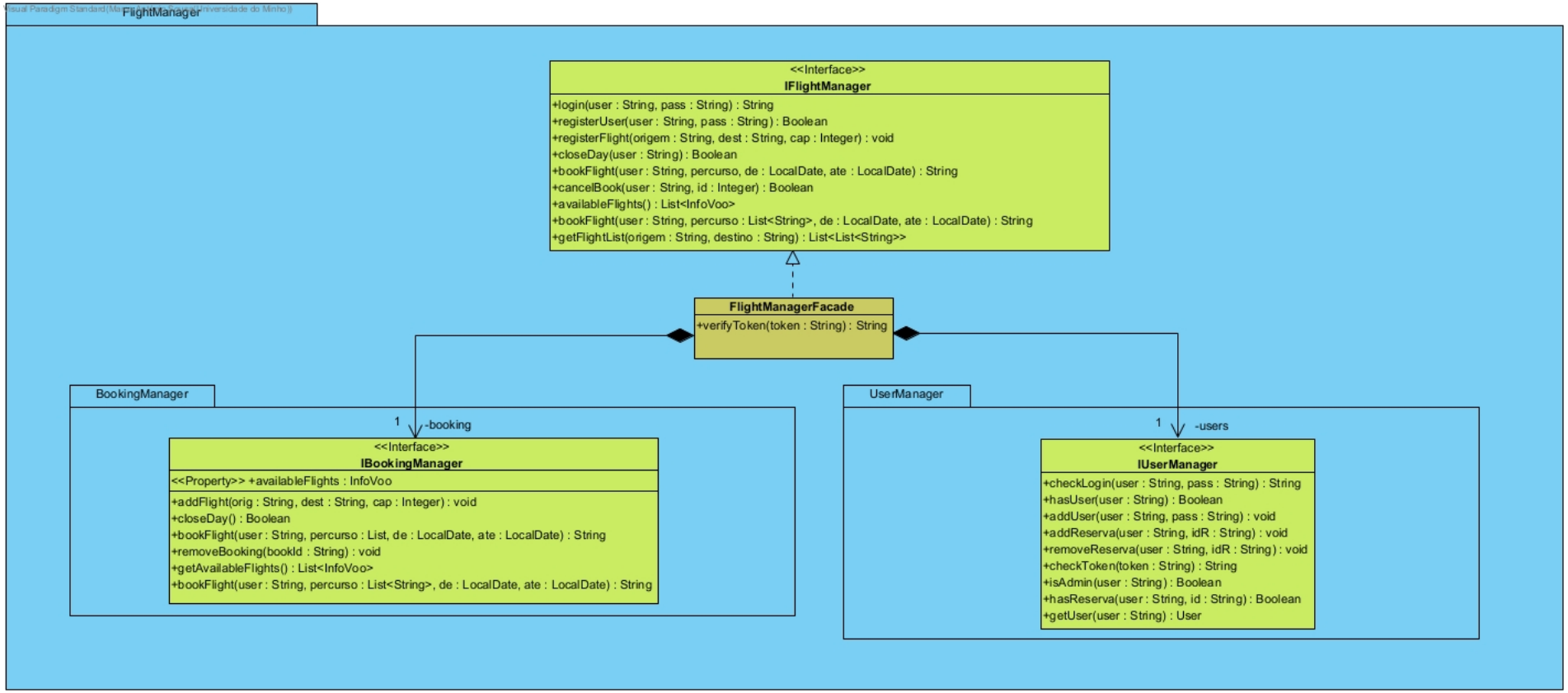



Figura 3.4: Diagrama de Classes - FlightManager