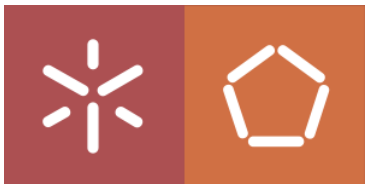

Universidade do Minho
Licenciatura em Engenharia Informática

# Laboratórios de Informática III

2022/2023

# Fase 1

***Realizado por:***

| | |
|---|---|
| Ana Alves | A95128 |
| Simão Antunes | A100597 |
| Gonçalo Brandão | A100663 |

22 de novembro de 2023

# Conteúdo

# Lista de Figuras

# 1 Introdução

Para a primeira parte deste projeto, foi necessário aplicar conhecimentos no que diz respeito à leitura e parsing de ficheiros CSV e, a partir desses ficheiros, ser capaz de criar coleções e estruturas de dados de forma a conseguir implementar estratégias eficientes e rápidas de realizar as nove queries propostas. Para este efeito, recorremos à biblioteca da linguagem C, GLib. Este relatório contem os passos e decisões tomados na realização da primeira fase do projeto. Inicialmente é apresentada uma descrição detalhada de como realizamos o encapsulamento e garantimos modularidade, seguida de uma explicação da implementação do parsing dos ficheiros, bem como a forma como fizemos o a validação dos mesmos. Posteriormente temos uma explicação das estratégias implementadas no desenvolvimento das queries. Além disso, concluímos de forma crítica a execução do projecto.

# 2 Arquitetura do projeto

A arquitetura do trabalho foi baseada na arquitetura sugerida no enunciado disponibilizado. Desta forma, conseguimos entender o conceito de encapsulamento e modularidade e destacar os principais componentes, interações e dependências, o que é crucial para tomar decisões informadas sobre a implementação.

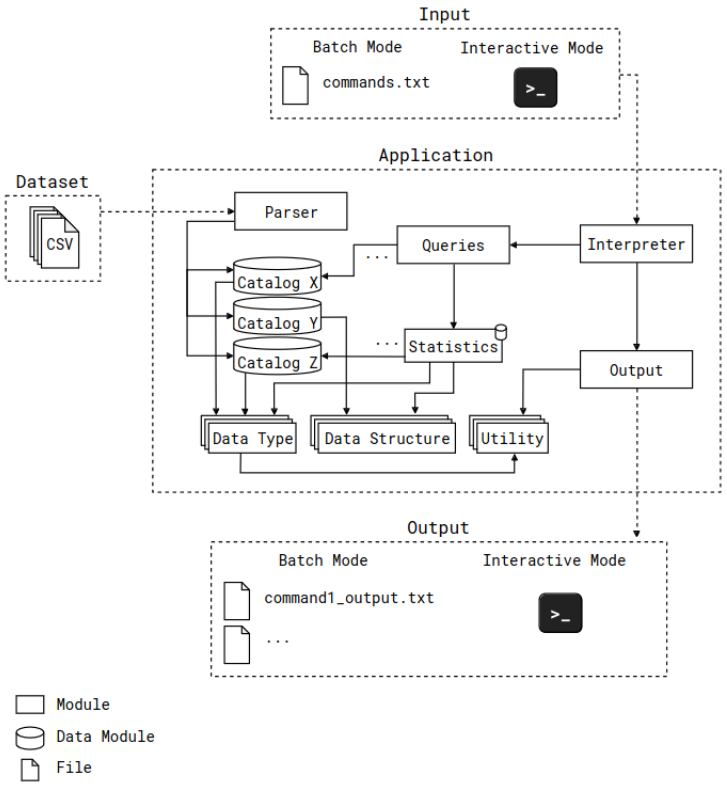


Figura 1: Sugestão da arquitetura para a aplicação a desenvolver

## 2.1 Encapsulamento e modularidade

De forma a garantir um bom encapsulamento e modularidade no trabalho, o mesmo foi estruturado para esse efeito, crinado módulos, onde as dependências apenas são dos níveis abaixo em relação aos mais acima,

nunca no mesmo "nível"ou de cima para baixo.

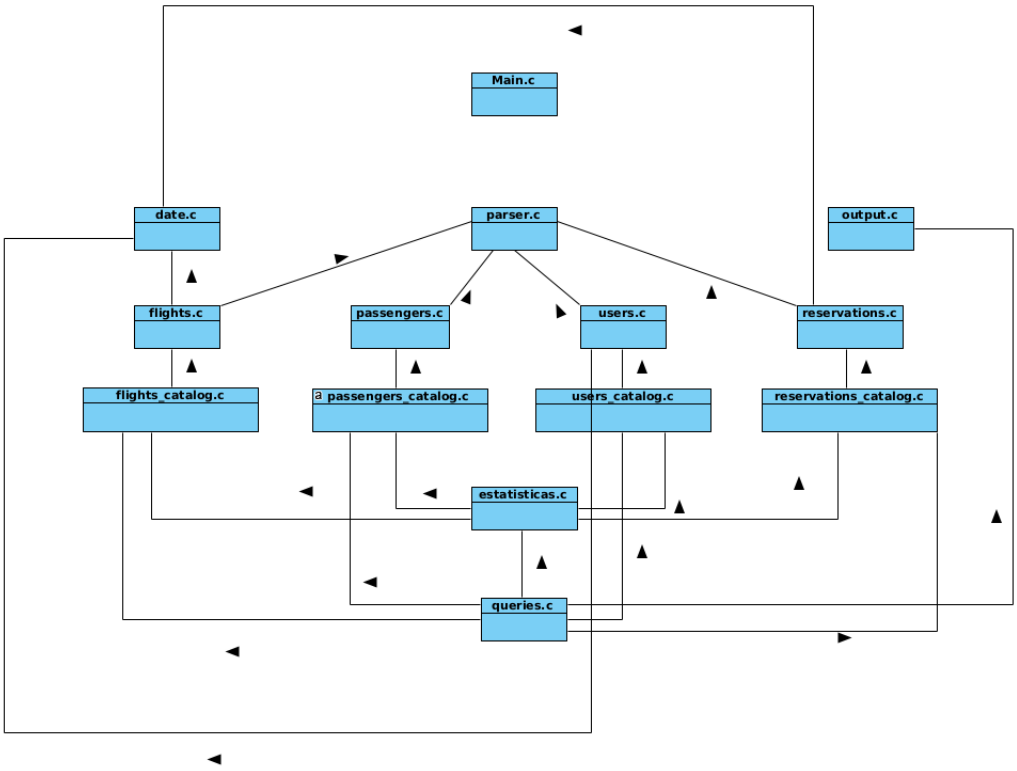


Figura 2: arquitetura para a aplicação a desenvolver

Posto isto, é nestes novos módulos que são implementadas coleções especificas, e todas as operações sobre estas, sendo estas estruturas privadas e as operações referidas públicas. Ou seja, apenas as funções no interior do módulo podem aceder a estas estruturas. No entanto, é sempre garantida a opacidade entre módulos através de cópias dos valores. Isto ocorre, por exemplo, com a estrutura User e as respetivas funções get que são as únicas com acesso à estrutura mencionada, devolvendo sempre cópias. Este encapsulamento está demonstrado na imagem que se segue. É de salientar que, no diagrama, estão apenas presentes os módulos gerais Deste modo, de forma a tornar a imagem mais percetível.

```
typedef struct users {
    char * id;
    char * name;
    Date * birth_date;
    char sex;
    char * passport;
    char * country_code;
    Date * account_creation;
    int account_status;
} User;
```

Figura 3: Estrutura Users

```
char * get_user_id(User * user);

char * get_user_name(User * user);

Date * get_user_birth_date(User * user);

char get_user_sex(User * user);

char * get_user_passport(User * user);

char * get_user_country_code(User * user);

Date * get_user_account_creation(User * user);

int get_user_account_status(User * user);
```

Figura 4: Gets da estrutura Users

## 2.2 Estrutura de dados

```
typedef struct flights {
    int id;
    char * airline;
    char * plane_model;
    int total_seats;
    char * origin;
    char * destination;
    Date * schedule_departure_date;
    Date * schedule_arrival_date;
    Date * real_departure_date;
    Date * real_arrival_date;
} Flight;
```

Figura 5: Gets da estrutura Flights

A nível das estruturas de dados foram tomadas decisões de modo a diminuir o tamanho que elas ocupam. Para tal, a estratégia que utilizamos passou por utlizar o tipo de dados com menor tamanho possivel. Como no caso do id e no total seats, foram definidos como int uma vez que consideramos mais eficiente do que um char.

## 2.3 Parsing dos dados

Para fazermos o parsing dos ficheiros CSV, em vez de termos uma função de parsing específica para cada ficheiro, de modo a reutilizarmos código, criámos uma função geral para parsing de ficheiros. Deste modo, a função recebe como argumento uma função que vai ser específica de cada ficheiro. Posto isto, optamos inicialmente por criar um array de strings, composto pelos tokens, obtidos a partir de cada linha de dados dos ficheiros, os quais são de seguida tratados pela função passada como argumento.

## 2.4 Validação dos ficheiros de dados

De forma a fazer a validação de entradas de dados durante o parsing dos ficheiros CSV é passada como argumento uma função específica que tem, no seu corpo, uma função responsável pela validação dos dados de cada ficheiro. Assim, verifica-se token a token se o seu campo é ou não nulo e, no caso de estar preenchido, se está conforme o pressuposto. Caso seja válido, é então criada a coleção específica do ficheiro,

i.e Users, Flights, Passengers e Reservations, e adicionada à sua própria estrutura de dados, contida num catálogo exclusivo.

### 2.5 Queries

As estruturas de dados definidas no catálogo das estatísticas são apenas a GHashTable e o GPtrArray. A GHashTable foi escolhida para recolher informação nas queries, pois como para uma dada key a informação tem de ir sendo atualizada, é a estrutura mais rápida para detetar e devolver o campo que queremos atualizar. Já o GPtrArray foi a estrutura escolhida para que se pudesse ordenar as coleções que lhe são inseridas. Todas as funções que podem aceder ao catalogo das estatísticas são implementadas em estatisticas.c respeitando sempre a opacidade e encapsulamento

## 3 Leitura e Tratamento de Dados

De modo a conseguir responder às queries propostas de forma eficiente, primeiro organizamos o grande volume de informação presente nos flights,users, passengers e reservations. Desta forma, criamos catalogos diferentes para cada tipo de dados com a ajuda da biblioteca GLib. Para efectuar o parser, utilizamos uma função geral capaz de trabalhar qualquer estrutura de dados.

```c
void parser(char * path, void (* buildStruct)(char **, void *, void *, void *), void * structure1, void * structure2, void * structure3) {
    FILE * file = fopen(path, "r");
    if (!file) {
        perror("Erro ao abrir o ficheiro");
        return;
    }
    char * line = NULL;
    ssize_t read;
    size_t len;
    while ((read = getline(&line, &len, file)) != -1) {
        char * line_copy = strdup(line);
        char * formated_line = strsep(&line_copy, "\n");
        char ** tokens = parseLine(formated_line);
        buildStruct(tokens, structure1, structure2, structure3);
        free(formated_line);
        free(tokens);
    }
    free(line);
    fclose(file);
}
```

Figura 6: função para o parser

# 4 Implementação

## 4.1 Query 1

A query 1 é responsável por listar o resumo de um utilizador, voo ou reserva com base no identificador recebido como argumento. optamos pela crialão de HashTables visto que estas possuem a vantagem de associar valores a keys. Ou seja, a partir de uma key, é possível desenvolver uma pesquisa rápida afim de obter todas as informações sobre a pessoa, reserva ou voo desejada no que toca à informação presente. Algumas informações são retiradas diretamente das estruturas mencionadas, através de get's, como por exemplo, a get user name que indica o nome de um dado utilizador. Foram necessários fazer alguns calculos auxiliares no que toca aos utilizadores, como a idade, número de voos, número de reservas e total gasto. Para o voo, foi necessário calcular o número de passageiros e o tempo de atraso. Para a reserva, foram feitaas funções auxiliares para o calculo de noites de uma reserva e para o total price de uma reserva.

## 4.2 Query 2

O objetivo da query 2 passa por listar os voos ou reservas de um utilizador, considerando que se o segundo argumento for flights ou reservations, respetivamente, ordena por data (da mais recente para a mais antiga). Caso não seja fornecido um segundo argumento, apresentar voos e reservas, juntamente com o tipo (flight ou reservation). Para tal implementamos a seguinte estratégia, primeiro defenir duas fuções para os voos e reserva, de forma a quando não houvesse um segundo argumento utilizarmos as funções já existentes. Assim, após termos estas duas funções bem implementadas, no caso especifico de não existir segundo argumento, simplesmente chamamos as funções, danndo merge e reeordenando os resultados da mesma.

## 4.3 Query 3

A função "calcular classificacao media"Apresentar a classificação média de um hotel, a partir do seu identificador. Para tal acontecer, a hashtable das reservations é iterada e cada vez que encontra uma reserva cujo hotel id é igual ao passado como argumento, soma a rating e divide pelo numero de reservas contabilizadas até então.

## 4.4 Query 5

A query5 lista os voos com origem num dado aeroporto, entre duas datas, ordenados por data de partida estimada (da mais antiga para a mais recente). Um voo está entre begin date e end date caso a sua respetiva data estimada de partida esteja entre begin date e end date (ambos inclusivos). Caso dois voos tenham a mesma data, o identificador do voo deverá ser usado como critério de desempate (de forma crescente). Para a excecução da mesma, o catalogo dos flights é iterado de forma a analisar a data de partida estimada de cada voo, entre as dadas fornecidas, se essa condição se verificar os voos são armazenados e posteriormente ordenados como mencionado em cima.

## 4.5 Query 8

A função "calculate total revenue"calcula a receita total de um hotel com base nas reservas dentro de um determinado intervalo de datas. Para isso utilizamos um iterador para percorrer as reservas no catálogo (Reservations catalog). De seguida, compara o identificador do hotel com o identificador fornecido e se a reserva pertencer ao hotel e estiver dentro do intervalo de datas, calcula o número de noites (nights), preço por noite (price) e a receita total para a reserva (total). Acumula a receita total do hotel (total revenue) e returna-a como resultado.

### 4.6 Query 9

A função "list users with prefix"tem como objetivo listar os utilizadores ativos cujo nome começa com um determinado prefixo. Inicialmente a função recebe o prefixo como uma combinação de dois tokens, uma vez que assumimos que um prefixo do tipo "J"representa um token e um "João M"representa dois tokens. Se tivermos dois tokens são combinados em um único token (combined token). Caso contrário, combined token recebe a cópia de token1.Iteramos sobre todos os utilizadores no catálogo, verifica se cada utilizador é ativo e se o nome começa com o prefixo especificado. Os utilizadores que atendem a esses critérios são adicionados a uma lista chamada activeUsersWithPrefix. A lista activeUsersWithPrefix é ordenada usando a função de comparação compare users. Esta função compara os utilizadores primeiro pelo nome e, em caso de empate, pelo identificador. Essa estratégia oferece uma maneira eficiente de listar e ordenar os utilizadores ativos com base em critérios específicos, facilitando a apresentação estruturada dos resultados.

# 5 Teste

De forma geral, a plataforma disponibilizada pelos docentes para testar o código tornou-se bastante útil na detenção e correção de erros ou código incompleto. Com base nos resultados obtidos, o código foi restruturado diversas vezes e com o auxílio do Valgrind estes foram resolvidos.

## 5.1 Memory Leaks

Na primeira fase do trabalho, o método utilizado gerava imensos erros e memory leaks devido às dependências de dados entre estruturas de dados que tornavam difícil libertar a memória, problema que foi corrigido através de uma reestruturação do mesmo. Ao longo do trabalho fomos sempre tentando prevenir a ocorrência das mesmas através da criação de funções para fazer free tanto das estruturas de dados como das coleções. Para além disso, sempre que foi necessária a utilização de memoria dinâmica, a implementação do código foi pensada de modo a poder libertar a memória alocada assim que possível.

Contudo, ao realizar as queries 1 e 2, apercebemo-nos que a estrutura de dados para o ficheiro csv passengers, não era a melhor na realização do nosso trabalho. Principalmente por não conseguirmos dar um free correto na mesma, ocurrendo sempre o double free em algum ponto da estrutura. Com isto, após termos passado grande parte do desenvolvimento com apenas as memory leaks associadas à utilização da biblioteca GLib, que são 0,019 MB, ou com pequenas memory leaks das ultimas queries a serem implementadas. No momento de entrega encontramo-nos com 1.021 MB de memory leaks que está muito longe do nosso objetivo.

# 6 Conclusão

Por outro lado, consideramos que conseguimos implementar um código que contempla boas práticas de encapsulamento, modularidade e opacidade. No primeiro, alcançamos um bom encapsulamento tendo sempre em conta uma hierarquia entre módulos, em que apenas existem dependências de baixo para cima e nunca no sentido inverso ou no mesmo "nível". Em relação à modularidade, garantimos que, apenas é possível gerir um determinado módulo com as funções do mesmo e, para além disso, tornámos os detalhes de implementação dos módulos privados e públicas apenas as funções necessárias a outros módulos. Já no que toca à opacidade, entendemos que esteja assegurada uma vez que nunca são fornecidos os valores reais dos dados, mas sim cópias ou clones dos mesmos. No entanto, mesmo com todas as otimizações efetuadas, reparámos que o nosso tempo de execução, em comparação com outros grupos, é superior, o que acreditamos que seria um pequeno desafio conseguir diminuir. Talvez isto fosse possível através de uma possível reutilização de estruturas de dados para as queries. Por fim, consideramos que, com a realização deste trabalho conseguimos melhorar o nosso método de programação, tomando conhecimento de novas bibliotecas, neste caso a GLib. Além disso, foi ainda possível consolidar melhor os conceitos de encapsulamento, modularidade e opacidade, que serão certamente úteis em trabalhos futuros.