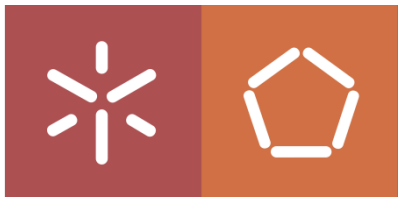
Universidade do Minho

Licenciatura em Engenharia Informática



Laboratórios de Informática III

2024/2025

Relatório 1ªFase Projeto

Grupo 73

a106804, Alice Soares

a106853, Ana Beatriz Freitas

a106894, Francisco Barros

Novembro 2024

Índice

Capítulo 1

Introdução

Este relatório aborda o desenvolvimento do trabalho prático da disciplina de Laboratórios de Informática III, implementado em linguagem *C*, inserido no 2ºano da licenciatura em Engenharia Informática.

Sendo o tema centrado num sistema de *streaming* de música, a partir do qual exploramos um conjunto de dados relativos a músicas, artistas, utilizadores e estatísticas de uso do sistema, carregamos os dados para estruturas de dados e utilizamos essa informação para responder a um conjunto de *queries*. Pretendemos fornecer uma visão geral do contexto, abordagem e métodos adotados, recorrendo à aplicação prática de conceitos como modularização e encapsulamento.

Nesta primeira fase, a estrutura do projeto enfatiza a organização modular. Para garantir a modularidade, o projeto é dividido em diversos módulos, cada um com diferentes responsabilidades, como *parse* de dados, execução de *queries* e gestão de estruturas de dados. Esta abordagem visa facilitar a manutenção e a evolução do sistema ao longo das diferentes fases do projeto.

Aliado a estes, a implementação de estratégias eficientes e o uso de bibliotecas, como a *GLib*, e ferramentas auxiliares, como o GDB para *debugging*, o *Valgrind* para análise de uso de memória, o *Doxygen* para documentação e o *Docker* para simularmos um ambiente idêntico à plataforma de testes e aí desenvolver todo o projeto.

O desenvolvimento deste trabalho prático inclui a criação de um programa principal que processa dados a partir de arquivos CSV, lidando com validações de dados e gerando respostas para *queries*. Inclui também a criação de um programa de testes e, a par disso, criamos também um programa de testes unitários.

Capítulo 2

Metodologia utilizada e Descrição do sistema

2.1 Arquitetura Geral

Dividimos o projeto em diferentes módulos e à medida que fomos desenvolvendo o código, fomos notando alguns módulos com código repetido e procedemos à generalização dos mesmos. O *parser* e o *output* são exemplos desses módulos que recebem apontadores para funções para executar as tarefas específicas.

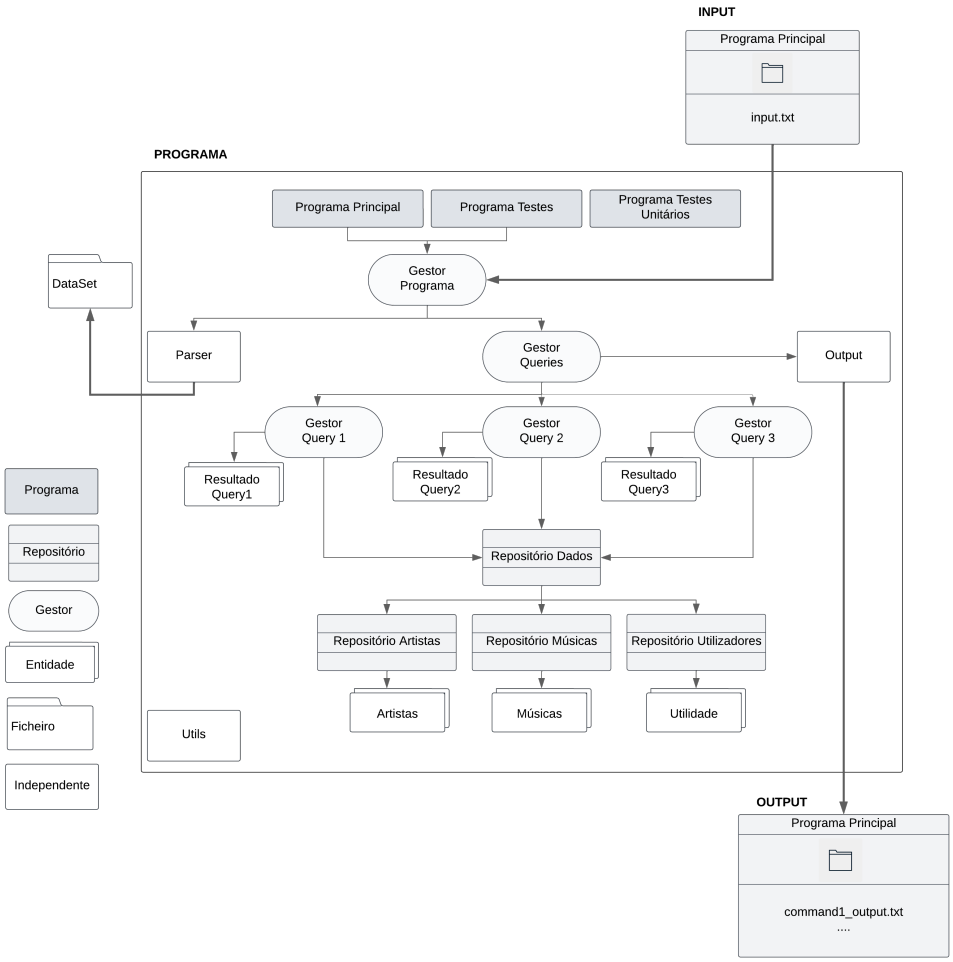


Figura 2.1: Arquitetura de referência para a aplicação a desenvolver

Ambos os programas (principal e teste) começam por executar o módulo de gestor de programa que está responsável por executar o *parser* (que é um módulo de *parse* genérico), ler o input e executar o gestor de *queries*. Optamos pelo módulo gestor de *queries* geral que inclui os módulos de gestão de cada uma das queries, para eventualmente avaliar se conseguiríamos fazer um gestor de *queries* preparado para receber qualquer tipo de *query*.

Por sua vez, cada gestor das *queries* utiliza os módulos de resultados respetivos para construir os resultados e passar ao módulo de *output* que é um módulo genérico. Decidimos ainda criar um *repositório-dados*, que engloba os repositórios das entidades, para evitar dependências pois, desta forma, os outros módulos apenas incluem o repositório dos dados e não cada um dos repositórios das entidades.

Temos ainda um módulo de *utils* constituído por funções de utilidades que podem ser usadas em qualquer um dos módulos.

2.2 Estrutura de dados

Todos os módulos possuem uma estrutura de dados própria e à medida que fomos avançando no projeto, estes foram mudando.

Relativamente às estruturas de dados das entidades utilizador, música e artista, inicialmente criamos *structs* com os campos apresentados no enunciado e acrescentamos-lhes informações úteis para a resolução das *queries*, como a idade do utilizador, a duração da música em segundos e a discografia em segundos, respetivamente.

Mais tarde notamos que algumas informações não estavam a ser usadas para responder às *queries*, nomeadamente o título, ano e letras das músicas e a descrição e as receitas dos artistas, por isso optamos por não carregar esses dados e, consequentemente, não os declarar na estrutura de dados.

Aquando da implementação das *queries*, percebemos que seria vantajoso também criar estruturas auxiliares para as *queries* 2 e 3, nomeadamente, uma lista auxiliar para os artistas e uma *hashTable* de géneros por *likes*.

2.3 Descrição dos módulos

A cada módulo está associada uma responsabilidade diferente e todos os módulos têm uma estrutura idêntica, como demonstramos nos arquétipos a seguir. Nos módulos genéricos que recebem funções anónimas temos os tipos das funções declarados no *ficheiro.h* para facilitar o *cast*. É também dentro destes módulos que encapsulamos as identidades.

Listing 2.1: Arquétipo do módulo.h

```
1
2  #ifndef MODULO_H
3  #define MODULO_H
4
5  // typedef estrutura incompleta para esconder os detalhes da implementacao
6  typedef struct Modulo Modulo;
7
8  // Declaracao da funcao para criar uma nova estrutura do Modulo
9  Modulo* Modulo_novo();
10
11 // Declaracao da funcao para libertar a memoria de uma estrutura Modulo
12 void Modulo_destroi(Modulo* modulo);
13
```

```
14 // Declaracao das funcoes gets e sets para definir um valor no objeto Modulo
15 void Modulo_set_...(Modulo* modulo, ...);
16 ... Modulo_get_...(Modulo* modulo);
17
18 // Declaracao de eventuais funcoes auxiliares e de execucao que sejam utilizadas por
       outros modulos
19 ... modulo_... (...);
20
21 #endif
```

Listing 2.2: Arquétipo do módulo.c

```
1  #include "modulo.h"
2
3  // Definicao da estrutura Modulo, "privada" para o arquivo de implementacao
4  struct Modulo {
5  };
6
7  // Implementacao da funcao para criar uma nova estrutura do Modulo
8  Modulo* Modulo_novo() {
9  };
10
11 // Implementacao da funcaoo para libertar a memoria de uma estrutura do Modulo
12 void Modulo_destruir(Modulo* modulo) {
13 };
14
15 // Implementacao das funcoes get e set da estrutura do Modulo
16 ... Modulo_get_...(Modulo* modulo, ...) {
17 };
18 void Modulo_set_...(Modulo* modulo, ...) {
19 };
20
21 //eventuais funcoes de auxiliares e de execucao
22 ... modulo_...(...){
23 };
```

2.4 Parsing dos dados

A função *parser_carrega* é um componente essencial do sistema de leitura e processamento de dados. O nosso *parser* lê um ficheiro, converte as linhas em objetos e adiciona-os a uma estrutura de dados definida pelo contexto, registando possíveis erros num arquivo separado.

Inicialmente tínhamos um *parser* para cada entidade, no entanto, para evitar repetição de código e para termos um *parser* capaz de receber qualquer tipo de entidade reestruturamos obtendo um *parser* genérico.

Capítulo 3

Implementação das funcionalidades

3.1 Validação dos Dados

Inicialmente fizemos as validações em ficheiros apenas para tal, porém com a modularização achamos melhor mudar as validações sintáticas para o módulo da entidade referente e as validações lógicas para o módulo do repositório dos dados, evitando assim dependências entre entidades e ficando mais coerente.

3.2 Queries

As *queries* são executadas a partir do *gestor-queries*, estando a implementação destas no seu próprio gestor que, por sua vez, usam os respetivos módulos de resultados.

3.2.1 Query 1

Listar o resumo de um utilizador, consoante o identificador recebido por argumento

Como já tínhamos os utilizadores carregados anteriormente para o seu repositório, apenas executamos uma pesquisa rápida por um utilizador específico no repositório de utilizadores (*HashTable*) utilizando o identificador fornecido e posteriormente construimos um *ResultadoQuery1* e preenchemos com as informações necessárias. É no *GestorQuery1* que o tempo de execução é registado para análise de desempenho.

3.2.2 Query 2

Quais são os top N artistas com maior discografia?

Para esta *query* decidimos usar um *GArray* como lista auxiliar, onde era adicionado o resumo do artista e, posteriormente, ordenado por ordem decrescente de discografia. Inicialmente isto era feito após o carregamento de todos os artistas mas percebemos que seria mais eficiente fazer aquando o carregamento dos artistas a partir do CSV, pelo que mudamos para tal. Ao executar a *query*2 acedemos a esta lista auxiliar ordenada e, filtrando ou não pelo país, o resultado é construído.

Para a *query* implementar esta estratégia colocamos um parâmetro *gboolean* para averiguar se o *array* já estava ordenado ou não, evitando fazer ordenações a mais, e, para o caso de, por exemplo, ser possível adicionar mais artistas, voltar a ordenar.

A estratégia utilizada pareceu-nos eficiente, porém é a query com maior tempo de execução.

3.2.3 Query 3

Quais são os géneros de música mais populares numa determinada faixa etária?

Esta última *query* foi a mais desafiante a nível da escolha da melhor estratégia que usaríamos para que fosse mais eficiente. A estratégia que optamos por usar foi, mais uma vez, criar uma estrutura de dados auxiliar, desta vez uma *HashTable* em que a *key* são os géneros de músicas e o valor é uma lista de inteiros. Esta lista de inteiros guarda os *likes* por idades, podendo o *índice* da lista ser interpretado como a idade, armazenando aí o número total de *likes* dessa idade para o respetivo género.

Assim como na *query* 2, esta estrutura estava inicialmente a ser criada apenas depois do carregamento de todos os dados pelo que alteramos, mais tarde, para ser carregado aquando o carregamento dos dados.

Esta abordagem oferece um processo de execução eficiente e organizado, com foco em tempos de resposta e uso de memória adequados.

3.3 Gestão de memória

Nesta primeira fase, tivemos sempre bastante cuidado com *memory leaks* criando, por isso, funções para fazer *free* para as estruturas de dados e também utilizando as funções de *free* predefinidas nas bibliotecas porém alguns erros foram-nos escapando, revelando-se assim difíceis de libertar.

Para percebermos onde havia falta de libertação de memória, usamos o *Valgrind* como auxílio. No momento de entrega apresentamos 0,019 MB de *memory leaks* associados à utilização da biblioteca GLib.

Capítulo 4

Testes e Resultados

4.1 Programa de testes

Inicialmente, criamos um conjunto de testes unitários para validar pequenas partes do código e funções individuais. Esses testes ajudaram a garantir que cada parte do código funcionava como esperado de forma isolada, antes de avançarmos para o desenvolvimento completo do programa.

O programa de testes foi desenvolvido com o objetivo de validar o funcionamento das *queries* do sistema, garantindo que os *outputs* gerados correspondem aos resultados esperados. O processo de teste foi dividido em várias etapas, incluindo a verificação da exatidão dos resultados de cada *query* e a medição de desempenho (tempo de execução e uso de memória). Este programa de testes teve um papel importante para o bom desenvolvimento do resto do código.

Durante a criação do programa de testes, enfrentamos algumas dificuldades, destacando-se exibir a linha específica do código em caso de discrepância nos resultados, mas mais tarde superada. Decidimos também implementar a contagem do tempo de carregamento de cada entidade.

4.2 Resultados dos testes e Análise de desempenho

A tabela compara o desempenho do programa nos três sistemas operacionais com diferentes configurações de hardware utilizados ao longo da execução do projeto.

Sistema Operativo	Processador	RAM	Memória	Query	Tempo de Exec (s)
Linux Ubuntu 24.04.1	i7-1165G	16GB	164.88	1	0.000011
				2	0.000726
				3	0.000083
Linux Mint 21.7	i7-10875H	32GB	164.88	1	0.000017
				2	0.000987
				3	0.000106
MacOS Ventura	i5 Quad Core	8GB	164.97	1	0.000067
				2	0.005452
				3	0.000602

Tabela 4.1: Desempenho do Programa

O segundo dispositivo presente na tabela apresentou os menores tempos de execução, sugerindo que maior capacidade de memória e processamento melhoram o desempenho. O primeiro dispositivo teve desempenho intermédio, enquanto o terceiro mostrou tempos de execução ligeiramente mais altos, especialmente na *Query*2, indicando que limitações de hardware aumentam o tempo de processamento.

A *Query*1 foi a mais rápida em todos os sistemas, enquanto a *Query*2 exigiu mais tempo, sugerindo que seria possível uma otimização. A análise revela que o programa opera de forma mais eficiente em sistemas com configurações robustas.

Capítulo 5

Conclusões

5.1 Análise Critica e Resultados alcançados

Notamos que o grande desafio deste projeto incidia na programação a grande escala, com base na grande quantidade de dados a analisar, e consideramos ter implementado um código que segue boas práticas de modularidade e encapsulamento. No entanto, o encapsulamento pode não estar completo, porque fomos encapsulando ao longo do desenvolver do projeto, não o priorizando, pois não seria avaliado nesta fase, ao contrário da modularização, onde nos focamos mais.

Desta forma, foi necessário manipular estruturas de dados e algoritmos que correspondessem tanto a nível de complexidade como em eficiência, aos desafios propostos concretizando, assim, os objetivos desta primeira fase.

Assim, consideramos que com o desenvolvimento desta parte do projeto conseguimos explorar e aprimorar os nossos métodos de programação, exploramos e conhecemos uma API nova para nós,*GLib*, assim como diversas ferramentas auxiliares e foi ainda possível consolidar os conceitos de modularidade e encapsulamento.

5.2 Otimizações e melhorias

Para tentar evitar que a *main* contivesse erros, criámos uma *git action*, para fazer *make* antes do *merge*, verificando assim se não tinha erros antes de fazer o *merge*. O ideal seria usar esta *action*, para criar um regra que impedisse o *merge*, apenas não a criamos porque não tinhamos permissões para tal no repositório do GitHub, porém seria uma mais valia se as tivessemos.

Sabemos também que temos várias melhorias a fazer que gostaríamos de implementar no futuro, nomeadamente um módulo para os comandos de *input* das *queries*, o *parser* permitir a divisão das linhas para a lista de colunas com a informação do CSV para evitar repetir código e, ainda, analisar a possibilidade de fazer um gestor de *queries* genérico pois sentimos que há partes repetitivas.

5.3 Limitações

Uma das limitações que enfrentamos durante o desenvolvimento do projeto foi termos percebido o impacto da modularização numa fase bastante avançada do mesmo, o que nos levou a reestruturar uma grande parte do código. Além disso, tivemos dificuldade em detetar e corrigir *memory leaks* muitos específicos e profundos nas definições de funções.