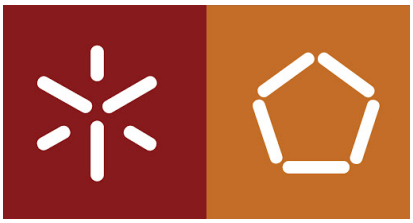

Universidade do Minho

Licenciatura em Engenharia Informática

Laboratórios de Informática III

Guião III

Grupo 28

Rui Monteiro (A93179) Rodrigo Rodrigues (A93201)
Daniel Azevedo (A93324)

Ano Letivo 2021/2022

Conteúdo

1 Introdução **3**

2 Solução técnica **4**
2.1 Mudanças em relação ao guião 2 4
2.2 Exercícios 5
2.2.1 Mecanismo de interação 5
2.2.2 Testes funcionais e de desempenho 6
2.2.3 Gestão de dados 7
2.3 Comparação de performance 7

3 Resultados e discussão **8**

4 Conclusão **9**

1 Introdução

Este documento é um relatório técnico acerca do Guião 3 do Trabalho Prático de Laboratórios de Informática III. Esta fase do trabalho tem como objetivos principais a consolidação dos conceitos de modularidade e encapsulamento, suporte a diferentes mecanismos de interação entre programa e utilizador, manipulação e otimização de consulta de grande volume de dados e criação de testes funcionais para avaliação de desempenho.

Este trabalho surge como um desafio na medida em que implica organização e utilização de conhecimentos lecionados nas aulas de LI3 de forma a desenvolver um programa com bom desempenho face a um grande volume de dados.

Ao longo da realização dos guiões desta UC foi fulcral respeitar e aplicar conceitos como **Modularidade e Encapsulamento**, **Criação de código reutilizável** e **Escolha otimizada de estruturas de dados**

Neste relatório apresentamos as estratégias utilizadas para desenvolver os exercícios propostos. Os objetivos são o desenvolvimento de um mecanismo de interação através de menu iterativo para os utilizadores do programa, desenvolvimento de testes para validação e avaliação do funcionamento do programa, e otimização dos mecanismos de gestão de grandes volumes de dados.

Palavras-Chave: **Modularidade**, **Encapsulamento**, **Interação**, **Programa**, **Utilizador**, **Otimização**, **Testes**, **Desempenho**, **Gestão de dados**

2 Solução técnica

2.1 Mudanças em relação ao guião 2

Neste guião foram feitas alterações em relação ao guião anterior de forma a corrigir falhas e melhorar soluções do guião anterior, especialmente no que ao **encapsulamento** diz respeito. Anteriormente, o programa apresentava diversas falhas de encapsulamento, nomeadamente na utilização de *getters*. Eis alguns exemplos de alterações feitas:

Guião2: Sem encapsulamento
Função do guião 2 sem encapsulamento.

```
int *getFollowersList(GH_USER u){
    return u->followers_list;
}
```

Versão atual: Com encapsulamento

```
int *getFollowersList(GH_USER u){
    int *followersList = (int*) malloc(sizeof(int*) * (u->followers));
    for(int i = 0; i < (u->followers); i++){
        followersList[i] = u->followers_list;
    }
    return followersList;
}
```

Guião2: Utilização de Getters para obter hashtables no catálogo das queries
No guião2, sempre que se pretendia consultar/realizar operações relativas às *Hash Tables* dos catálogos (*users,commits,repos*), utilizava-se um *getter*. A abordagem de retornar a referência para a hashtable quebrava o encapsulamento.

```
GHashTable * get_catalogoRepos(C_repos crepos) {
    return (crepos->catalogo_repos);
}
```

Versão atual: Com encapsulamento
De forma a garantir modularidade e encapsulamento, agora utiliza-se uma estrutura em vez de usar diretamente uma *Hash Table* tornando o nosso programa mais modular e extensível, estando o resto do programa alheio a mudanças na estrutura de dados de cada catálogo. Isto porque o resto do programa conhece a estrutura do catálogo e respetivas funções mas não o seu conteúdo ou representação interna. Neste guião, procurou-se implementar funções que "trabalham"com a estrutura de dados de um catálogo, nesse mesmo catálogo. Assim, garantimos acesso direto à

Hash Table, descartando a necessidade de utilizar *Getters* para obter uma cópia da estrutura.

2.2 Exercícios

2.2.1 Mecanismo de interação

Além do mecanismo de interação do guião 2, o programa agora disponibiliza um menu sempre que não recebe um ficheiro de comandos como argumento, processando os ficheiros de dados e posteriormente apresentando a opção de executar cada uma das funcionalidades que o programa disponibiliza. Seguimos o formato apresentado no enunciado no projeto no que toca a estrutura dos menus e na forma de apresentar os resultados das funcionalidades no *stdout*.

Ao invocar o programa, é feito o carregamento e processamento dos ficheiros de dados, cujo progresso é apresentada à medida que se vai completando o processamento de determinado ficheiro. O menu é apresentado ao utilizador e, após haver a escolha de uma opção, o programa solicita ao utilizador os campos necessários para a invocação dessa funcionalidade e apresenta o seu resultado, oferecendo a opção de selecionar novas funcionalidades, abandonar o programa, ou avançar/retroceder nas páginas. Para implementar o mecanismo de páginas de um menu de uma funcionalidade desenvolveu-se a função **paginação** que nos permite apresentar esse mesmo resultado de forma estruturada e, ao mesmo tempo, habilitando a navegação entre as várias páginas usando uma função de seleção de página chamada ***selector***. O processo é o seguinte. A função principal de cada *query* no módulo das *queries* têm o tipo de retorno **GArray*** que corresponde a um array de *Strings* onde cada *String* contém a informação de cada **Top N** *user/repo/commit* solicitados na *query* em questão. A função *paginaçao* tem como parâmetros o **GArray*** com a informação de todas as páginas necessária para exibir no *stdout*, o número de páginas (calculado com base no número de *users/commits/repos* solicitados e no tamanho fixo de cada página (6)) e a página que se pretende exibir. Assim, apenas tem de determinar o conteúdo da página solicitada e apresentá-lo. A função ***selector*** permite avançar, retroceder e saltar para uma determinada página, invocando a *paginacao* conforme o "input"(Ex: Estando na página atual 2/50 e recebendo o *input* "P", invoca a *paginacao* para a página 3).

Relativamente aos resultados dos testes de desempenho/performance e de coerência, é feita a impressão do *CPU Time* calculado para cada *query* e da verificação da igualdade dos ficheiros (se os ficheiros criados são exatamente iguais aos esperados).

Foi criado o módulo *View* que contém funções de paginacao/selecionar páginas, funções relativas a apresentação de menus, exibição de resultados, operações "I/O", *prints*, entre outros.

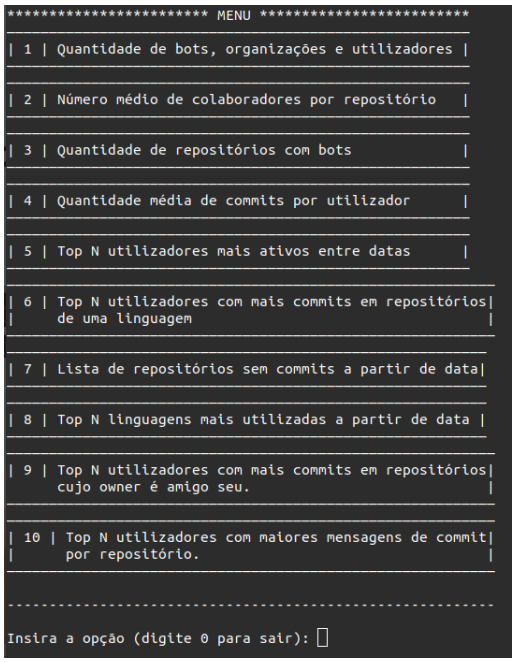

Figura 1: Menu inicial

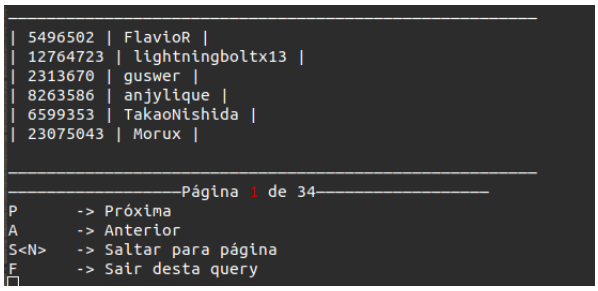

Figura 2: Páginas/resultados de queries

2.2.2 Testes funcionais e de desempenho

Foram desenvolvidos testes para avaliar o desempenho das *queries* (avaliar se a query é executada em tempo útil) e a sua funcionalidade/correção (verificar que o seu resultado é correto e que cada uma das queries funciona conforme o especificado). A Makefile do projeto foi adaptada para gerar um executável adicional de testes, capaz de invocar todos os testes e apresentar os seus resultados. Foi criado um novo módulo ***Tests*** que possui funções para comparar ficheiros (função *compare* e assegurar que os ficheiros produzidos são iguais aos ficheiros esperados e funções para testar as queries, calculando o seu tempo de execução.

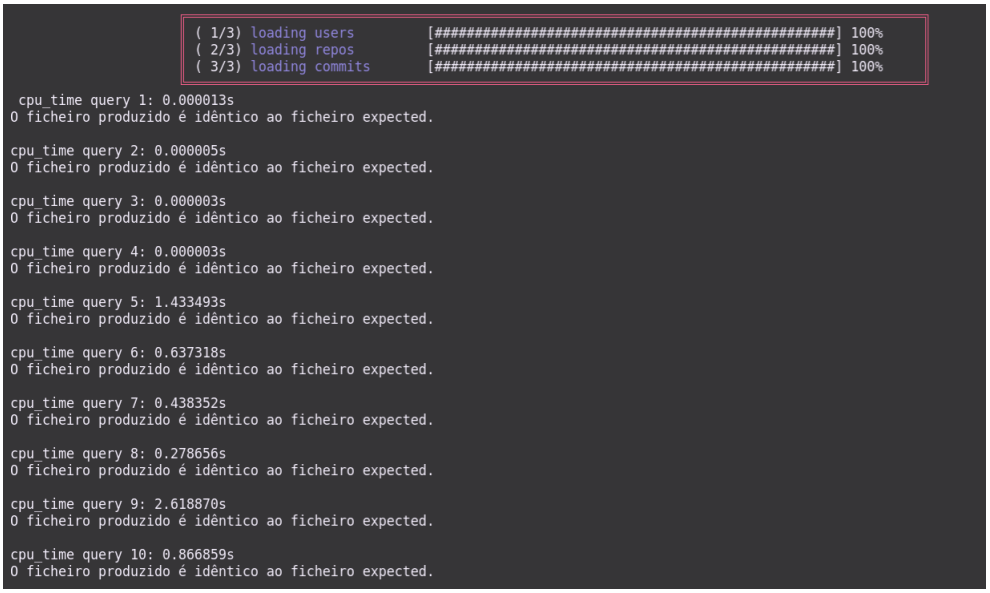

Figura 3: Testes de performance e funcionalidade

2.2.3 Gestão de dados

Para se realizar a gestão e manipulação de dados, foram criados 3 catálogos (commits, users, repos), que são carregados após a validação dos dados que se encontram nos ficheiros de *input*. Cada catálogo contém uma estrutura de dados *Hash Table*. No caso dos utilizadores e dos repositórios, visto que têm um identificador próprio, este é usado como chave para armazenar um elemento na *Hash Table*. Por sua vez o catálogo de *commits* conta com uma *Hash Table* cujas *keys* são o identificador do repositório e os *values* são um *GArray* de *commits* referentes a esse mesmo repositório. A escolha desta organização do catálogo de *commits* foi feita de forma a responder à necessidade de aceder aos *commits* de um repositório na maioria das *queries*.

2.3 Comparação de performance

No que diz respeito a otimizações, a equipa de desenvolvimento viu a necessidade de melhorar o campo das mensagens dos *commits*, visto que estava a ser armazenado todo o seu conteúdo quando apenas era necessário saber o seu comprimento (para determinar os *top N* utilizadores com maiores mensagens de commit). Sendo assim, atualmente quando um *commit* é carregado na memória passa a ser armazenado o tamanho da mensagem em vez da mensagem em si. Através desta alteração foi possível reduzir o uso de memória por parte do programa. Com o objetivo de melhorar o encapsulamento foram removidos todos os *getters* de *Hash Tables* dos catálogos, melhorando também o tempo de execução das diversas funcionalidades,

pois ao executar uma dada funcionalidade são executadas menos instruções pelo *CPU* e deixa-se de necessitar criar uma cópia de uma *Hash Table* que pode conter um número muito elevado de elementos, reduzindo assim o tempo de execução.

3 Resultados e discussão

Para a realização dos testes automáticos utilizamos como ficheiros esperados, o resultado da utilização da validação, através dos ficheiros de *input* do guião 3, do guião 1, posteriormente usando o código do guião 2 para executar as funcionalidades pedidas.

Número da *query*	Parâmetros	Tempo CPU 1(s)	Tempo CPU 2(s)
1	-	0.000007	0.000015
2	-	0.000003	0.000005
3	-	0.000002	0.000001
4	-	0.000002	0.000002
5	100 2010-01-01 2015-01-01	1.380273	1.353008
6	100 Python	0.625183	0.629658
7	2014-04-25	0.414188	0.416483
8	100 2010-10-05	0.262154s	0.268255
9	100	2.610404	2.612706
10	3	0.843222	0.864235

Computador 1: SO - Ubuntu 20.04.3 LTS 64-bit, **CPU** - i7 9750H (6 cores, 12 threads, 2.4GHz), **RAM** - 16Gb DDR4 2666MHz, **Disco** - SSD SATA 500Gb 560MB/s Leitura 510MB/s Escrita

Computador 2: SO - Ubuntu 20.04.3 LTS 64-bit, **CPU** - i7 8750H (6 cores, 12 threads, 2.2GHz), **RAM** - 16Gb DDR4 2666Mhz, **Disco** - SSD SATA 480Gb 500MB/s Leitura 350MB/s Escrita

Todas as queries apresentam tempo de CPU (s) abaixo de 5s pelo que se pode dizer que executam em tempo útil. As *queries* estatísticas (1-4) apresentam tempo de CPU muito baixo, como era de esperar, devido ao mecanismo utilizado, i.e, os valores são calculados aquando do carregamento dos ficheiros e ficam guardados, sendo obtidos através de um *Getter* quando a query é invocada.

A query 5 demora à volta de 1.36s a executar nos computadores dos membros do grupo, ou seja, ligeiramente mais do que as restantes queries (exceto query 9). Tal pode se justificar pelo facto de fazer muitas comparações de datas (*struct tm*), e também pelo facto da organização do catálogo de commits não lhe ser favorável (RepoID $->$ GArray* de commits). Para que fosse favorável deveríamos ter como chave o identificador do "commiter"e associado à chave um GArray que continha todos os seus commits (CommiterID $->$ GArray* de commits). As queries 6,7,8,10 demonstram ser rápidas, fruto da boa escolha de estruturas de dados e

organização das mesmas e do tratamento de operações relativas aos catálogos nos próprios módulos, tendo acesso direto à *Hash Table*. No que toca à query 9, foi a única que teve pior *performance* após as alterações realizadas no presente guião, mais especificamente no que diz respeito ao encapsulamento, pois necessita de aceder às estruturas de dados dos *users*, *commits* e *repos* para calcular o seu resultado. Ainda assim, executa confortavelmente em tempo útil.

No que toca à coerência dos ficheiros esperados com os ficheiros obtidos, não há nada a assinalar visto que os resultados são sempre os mesmos.

4 Conclusão

Concluída a implementação do guião 3, consideramos que conseguimos alcançar os principais objetivos a que nos propusemos: suportar diferentes mecanismos de interação entre o programa e o utilizador, a manipulação e otimização de consulta de grande volume de dados, a realização de testes funcionais e avaliação de desempenho e, por último, a consolidação dos conceitos de modularidade e encapsulamento aplicados no guião anterior. Consideramos ter feito progresso positivo em relação ao último guião, sempre procurando respeitar os conceitos principais da UC de Laboratórios de Informática III e fazendo bom uso de técnicas de encapsulamento. Apesar disso, poderia-se melhorar a organização/estruturação do código, deixando-o mais claro e melhorando significativamente o tempo de execução de algumas funcionalidades. A realização deste projeto foi fulcral para consolidar os conceitos lecionados nesta UC, dando novas ferramentas e capacidades aos elementos do grupo acerca de desenvolvimento de software.