Relatório do Guião II

Laboratórios de Informática III

Grupo 73

Guilherme Sampaio, a96766

Tiago Oliveira, a97254

Miguel Neiva, a92945

Licenciatura em Engenharia Informática

14 de dezembro de 2021

Resumo

Este relatório vem em seguimento do relatório referente ao Guião I no âmbito da unidade curricular de Laboratórios de Informática III visando relatar o desenvolvimento deste novo exercício.

Foi sem dúvida um guião mais desafiante que precisa de bons conhecimentos da linguagem C assim como de estruturas de dados e algoritmos de procura, inserção e ordenação. Uma das maiores dificuldades foi a gestão de memória e como estruturar o trabalho, isto é, que tipo de estruturas de dados utilizar, como as organizar e qual a melhor estrutura a usar de modo a aumentar a eficiência do programa. Devido às falhas do guião anterior a gestão de memória, desta vez, já foi mais fácil de tratar em consequência dos cuidados tomados.

Outro aspeto que tivemos de ter em conta foi o encapsulamento dos tipos de dados, que por si só não complicou muito o trabalho, mas dificultou a escrita dos módulos e algumas funções (tiveram de ser utilizados bastantes *getters*). No geral, a resolução deste guião foi mais "fácil"mas também foi tomado outro tipo de atenção a aspetos importantes que outrora não teríamos.

Conteúdo

1 Introdução

No âmbito da unidade curricular Laboratórios de Informática III, este relatório tem como objetivo apresentar as estratégias e métodos empregues na realização do Guião 2 do projeto e revelar complicações encontradas na execução deste. Deste modo, este guião visa a criar uma base da arquitetura da aplicação final. Sendo assim, as finalidades são: fazer o *parsing* dos dados e a interpretação de comandos, executando várias *queries* (conjunto de questões sobre os dados) com os argumentos indicados (se existirem) e, desta forma, usando catálogos para organizar e armazenar todos os dados contidos nos documentos relativos aos *users*, *commits* e *repositories*. Para elaborar esta parte do projeto necessitamos de considerar os princípios da modularidade e encapsulamento.

2 Exercício

Contrariamente ao guião 1, desta vez apenas temos um exercício. Este, visa o armazenamento de todos os dados relativos a *users*, *commits* e *repositories*, dados estes que posteriormente serão submetidos a *queries*. Um dos grandes objetivos deste exercício é a consolidação de termos como modularidade e encapsulamento do código. Para a resolução do exercício foi dada uma estrutura abstrata da arquitetura do programa (Fig.1) de onde inferimos que o código terá de ser dividido em alguns módulos como, por exemplo, um módulo para cada catálogo de dados ou um módulo que apenas faz o tratamento do *input* das *queries*.

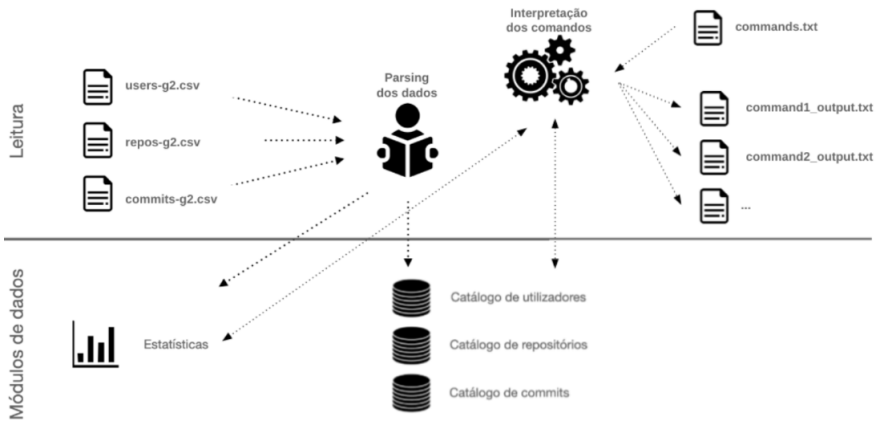

Figura 1: Arquitetura da Aplicação

O encapsulamento foi uma novidade porque nenhum dos nossos trabalhos anteriores exigiam esta preocupação, como tal, como o objetivo do encapsulamento é restringir o acesso direto a membros de estruturas de dados foi preciso utilizar *setters* e *getters*, ou seja, funções que definem valores na estrutura e funções que retiram valores das estruturas. Finalmente, este relatório está dividido em três partes: o *parsing* dos dados e o método

do seu armazenamento, a interpretação e execução de *queries* e os desafios encontrados no que toca aos conceitos de modularidade e encapsulamento.

3 Resolução

Como já dito acima, este capítulo está divido em três partes principais que abrangem os assuntos mais importantes do problema, que dificuldades foram encontradas, como se resolveram e o que se poderia ter feito melhor de modo a melhorar a *performance* do programa.

3.1 *Parsing* de Dados e Armazenamento

Mais uma vez, a parte mais difícil desta resolução do problema de *parsing* de dados e o seu armazenamento foi a decisão acerca das estruturas de dados, nomeadamente, quando e como as utilizar de modo a criar uma base de dados eficiente que não ocupe muita memória.

Os ficheiros que os contêm dados a ser guardados são três, um para dados relativos a utilizadores (*users.csv*), dados relativos a *commits* (*commits.csv*) e dados relativos a repositórios (*repos.csv*). Decidimos, então, criar três módulos, um para cada ficheiro, deste modo, conseguimos cumprir com a modularidade do programa, pois quem quiser consegue utilizar cada módulo individualmente sem requerer outras funções quaisquer e com o encapsulamento das estruturas de dados, pois cada módulo contém estruturas opacas que só conseguem ser acedidas com a ajuda de *getters*. Para além destes módulos, teve ainda de ser criado um quarto onde são armazenadas estatísticas relativas aos módulos anteriores. Surgiu-nos então a questão: **Que estrutura de dados vamos adotar?**

No que toca a estruturas de dados, no mínimo, são precisas três por cada módulo (sem contar com as estatísticas), uma que representa um utilizador, uma que representa um catálogo (contêm utilizadores) e as estruturas de dados onde vão ser armazenados os utilizadores. Assim, cada módulo, fica com o seguinte formato abstrato:

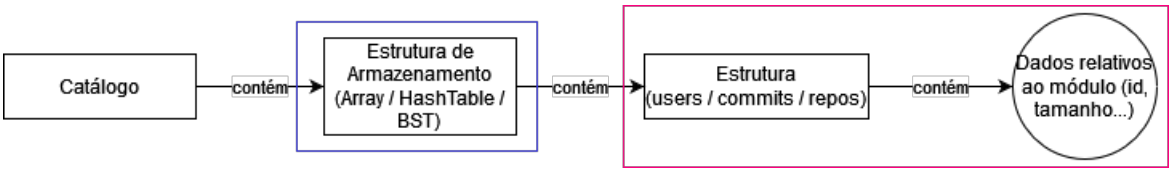

Figura 2: Estruturas de Dados

Destas estruturas todas, apenas houve uma que não foi trivial decidir o que utilizar, sendo esta, a(s) estrutura(s) de armazenamento, pois é a base do catálogo e é também onde os dados vão ser todos guardados. Depois de alguma discussão e pesquisa acabamos por escolher três concorrentes, *arrays* ordenados, *hash tables* e *binary search trees*, mas

Operation\Structure	**AVL Tree**	**Hash Table**	**Sorted Array**
Insertion	$O(\log n)$	$O(n)$	$O(n^2)$
Lookup	$O(\log n)$	$O(1)$	$O(\log n)$

Tabela 1: Complexidade para o pior caso de inserção e procura.

agora ficamos sem saber quando utilizar cada, por isso recorremos à análise assintótica de cada estrutura.

Após esta análise, cujos resultados podem ser observados na Tabela 1, decidimos optar por utilizar maioritariamente *arrays* ordenados.

Em primeiro lugar, decidimos comparar os tempos de inserção de elementos e, realmente, olhando para a tabela veríamos que a opção mais sábia seria escolher a *AVL Tree* pois tem uma complexidade para o pior caso de apenas $O(\log n)$, porém não iremos fazer nenhuma inserção após o *storing* dos dados, isto é, apenas vamos adicionar os dados uma vez na estrutura e para este, e só este, objetivo adicionar num *array* ordenado é bem mais rápido ($O(1)$, acessar a um índice). É claro que vai ser sempre perdido algum tempo ao ordenar o *array* mas como isso apenas será feito uma vez e com uma complexidade temporal de $O(n \cdot \log n)$ (*quick sort*) não terá muito impacto.

De seguida, decidimos comparar os tempos de procura de *keys* de elementos (porque trabalhamos com estruturas de dados com vários parâmetros), mais uma vez, olhando para a tabela vemos que a melhor opção seria escolher a *hash table* pois tem tempo de *lookup* constante. Relativamente a *AVL Tree* e *array* ordenado vemos que, para o pior caso, ambas têm um tempo semelhante (procurando no *array* com procura binária). Com isto, podemos concluir que a melhor opção para escolha de estrutura de dados seria utilizar *ALV Trees*, porém como a diferença de complexidade temporal entre estas e o *array* ordenado não é muito significativa e o custo de implementação é maior, decidimos optar por *arrays* ordenados como estrutura principal para a base de dados.

Com as estruturas de dados definidas podemos, finalmente, ler os ficheiros de entrada, fazer o seu *parsing* e guardar em memória. De um modo geral, cada ficheiro é lido linha a linha onde cada uma é passada por um *parser* que inicializa a respetiva estrutura de dados e guarda-a no *array* ordenado e, após isto, caso dê jeito, o *array* pode ainda ser ordenado. Ou seja, no final do *parsing* e do armazenamento dos dados ficamos com, por exemplo:

```
struct user_reg // Estrutura que representa um utilizador.
{
    long id, public_repos, followers, following, gists;
    char* username;
    datetime* created_at; // Vem no formato YYYY-MM-DD HH:mm:ss
    type type; // Bot | User | Organization
    int *following_list, *follower_list; // Array de ids de user_reg.
};
```

```
struct user_cat // Catalogo de users.
{
    user_reg **user_sorted_array; // Array de users (user_reg*).
    int user_sorted_array_size;   // Tamanho do array acima.
};
```

Listing 1: Exemplo de um Catálogo de Users

3.1.1 Dificuldades Encontradas

Ao longo da resolução deste problema de armazenamento surgiram dois problemas mais significativos.

O primeiro tem a ver com a *scope* dos utilizadores, isto é, inicialmente os utilizadores (representados pela estrutura *user-reg*) eram adicionados à base de dados (representada por um *array ordenado*) pela variável e não pelo apontador para essa estrutura. Isto resultou na perda de todos os dados lá contidos assim que a função acabava a sua execução. Este problema foi resolvido facilmente ao mudar o tipo da base de dados de `user_reg *array` para `user_reg **array` e adicionar os *user pointers*.

O segundo problema está relacionado com corrupção de memória devido a *memory leaks* que resultaram da tentativa de uso de *hash tables* da biblioteca *GLib2.0* de modo a tentar reduzir o custo de algumas funções. Esta corrupção causou bastantes problemas, pois corrompia algumas *strings* guardadas na base de dados. Até hoje, não sabemos bem o porquê deste problema e apenas o conseguimos detetar com o uso da ferramenta *Valgrind* e ao testar o programa com e sem o uso da estrutura problemática. Devido a isto, acabamos por não utilizar nenhuma *hash table* nas bases de dados.

```
==11454== HEAP SUMMARY:
==11454==     in use at exit: 136 bytes in 1 blocks
==11454==   total heap usage: 10 allocs, 9 frees, 224 bytes allocated
==11454==
==11454== LEAK SUMMARY:
==11454==    definitely lost: 136 bytes in 1 blocks
==11454==    indirectly lost: 0 bytes in 0 blocks
==11454==      possibly lost: 0 bytes in 0 blocks
==11454==    still reachable: 0 bytes in 0 blocks
==11454==         suppressed: 0 bytes in 0 blocks
```

Listing 2: Output simplificado do Valgrind

3.2 Interpretação e Execução de *Queries*

Agora que todos os nossos dados (dos ficheiros de entrada) já estão disponíveis, podemos começar a trabalhar no processamento e execução das *queries*. No guião do trabalho foram descritas dez *queries* diferentes, quatro delas são estatísticas e não precisam de receber qualquer tipo de parâmetro e as restantes são não parametrizáveis e relativas a

dados mais específicos de cada categoria de dados (*Users, Commits, Repositories*). Assim, este capítulo está divido em dois tópicos, sendo estes, *queries* não parametrizáveis e parametrizáveis, onde vai ser discutido o método de resolução e os problemas encontrados.

Porém, antes de poder executar *queries*, primeiro precisamos de as interpretar e, para isso, foi-nos pedido para fornecer um ficheiro de texto com as consultas que queremos realizar em cada linha, linhas estas que terão de ser analisadas de modo a entender **que *query* é suposto executar** e **quais os parâmetros da *query*** em questão. Segue um exemplo do ficheiro que contém *queries*:

```
1
5 100 2010-01-01 2015-01-01
6 5 Python
7 2014-04-25
8 3 2016-10-05
9 4
6 3 Haskell
10 50
```

Listing 3: Exemplo de um *query file*.

Como podemos observar, todas as linhas começam com um número (1 – 10), que indica o *id* da *query* a ser executada, e a muitas linhas seguem outros dados que são os parâmetros. Como dito acima, se $1 \leq id \leq 4$ então não podem existir dados adicionais e se $4 < id \leq 10$ têm de existir dados adicionais.

Não houve muitas dificuldades na realização desta etapa do projeto, pois, mais uma vez, leitura de ficheiros linha a linha e *parsing* de linhas num determinado *token* é algo que foi feito anteriormente no Guião 1, ou seja, um *loop* sobre as linhas dos ficheiros com um *parser* de linhas como a função *strtok* da *libc*. A parte relativamente difícil vem agora com a execução de cada linha.

3.2.1 *Queries* Estatísticas

Query 1

A *query 1* pede a quantidade de cada tipo de utilizador, isto é, a quantidade de utilizadores (*users*), de *bots* e de organizações *orgs* presentes nos registos dados para a resolução do trabalho. Esta *query* não ofereceu nenhuma dificuldade, pois o seu resultado pode ser facilmente obtido durante o *storing* dos dados, ou seja, enquanto guardamos os utilizadores verificamos sempre o seu tipo e incrementamos o valor correspondente que, finalmente, é guardado numa estrutura de dados estatísticos utilizando os tais *setters* mencionados no ponto 2.

Query 2

A *query 2* pretende calcular o número médio de colaboradores por repositório. Esta foi a primeira consulta que introduziu o conceito de colaborador, um utilizador é considerado colaborador quando teve algum *commit* num determinado repositório.

Sabendo que o número médio de colaboradores por repositório é dado por $\frac{Colaboradores}{Repositorio}$ e que o número total de repositórios pode ser dado pelo número de linhas do ficheiro dos repositórios - 1 só nos falta calcular o total de colaboradores. Para isso, precisamos de iterar pelos utilizadores e pelos *commits* procurando sempre por pelo menos um *commit* cujo *author-id* ou *committer-id* seja igual ao *id* do utilizador e ir guardando o total numa variável.

Esta *query* também foi relativamente fácil onde apenas houve umas dificuldades iniciais no cálculo do número total de repositórios. Em relação à escrita do ficheiro de output, para estas primeiro quatro *queries* foi sempre utilizada uma função que ao receber uma *string* com o *output* desejado, cria e escreve no ficheiro.

Query 3

A *query* 3 pede o número de repositórios contendo *bots* como colaboradores. Agora que já sabemos identificar um colaborador, esta *query* tornou-se bastante fácil, sendo apenas preciso criar uma função que verifique o tipo do utilizador, o que por si só também é simples, pois todos os utilizadores têm um parâmetro chamado *type* que indica o seu tipo.

Sabendo isto, bastou iterar pelos utilizadores, verificar o seu tipo, e se é ou não um colaborador, e guardar o número de colaboradores do tipo *bot* e escrever para o ficheiro de saída.

Query 4

A *query* 4 pretende calcular o número médio de *commits* por utilizador, dado por $\frac{Commits}{Utilizadores}$. Ambos estes dados podem ser obtidos sem necessidade de recorrer à base de dados, basta calcular o tamanho dos ficheiros dos utilizadores, dos *commits*, dividir um pelo outro e finalmente escrever para o ficheiro de saída.

É uma *query* muito simples assim como todas as anteriores e finaliza as *queries* não parametrizáveis/estatísticas.

3.2.2 *Queries* Parametrizáveis

Query 5

Esta *query* calcula o *top* N utilizadores mais ativos num determinado intervalo de datas, tanto o valor de N como duas datas que indicam o intervalo desejado. Todas as

queries parametrizáveis seguem uma ordem de "ações", em primeiro lugar realizamos o *parsing* dos argumentos (dividir os argumentos, por exemplo, separar números inteiros das datas), de seguida computamos o valor que a *query* pretende calcular e, finalmente escrevemos os resultados para o ficheiro de saída. Sendo tanto o primeiro como o último passo desta *query* trivial vamo-nos apenas focar no cálculo dos resultados.

Então, assumindo que o *parsing* dos parametros já foi feito, temos um número N e duas datas $D1, D2$ para trabalhar. O nosso primeiro passo foi descobrir como encontrar o intervalo de $D1$ e $D2$, para isso decidimos criar um *array* de *commits* ordenado pelas datas, desta forma, podemos calcular os dois índices que delimitam o que nos interessa. A seguir, tivemos de calcular o total de *commits* para cada utilizador, para isso, foi criado um *array* com uma nova estrutura que contém o *id* e total de *commits* do utilizador, a cada iteração pelos *commits* verificamos se o *user* já se encontra nos nossos registos e caso se encontre incrementamos o seu valor total, caso contrário adicionamos o *user* ao *array*. Finalmente, ordenamos (decrescentemente) o *array* pelo total de *commits* e escrevemos no ficheiro de saída os N primeiros utilizadores.

Query 6

A *query* 6 visa calcular o *top* N de utilizadores com mais *commits* em repositórios de uma determinada linguagem, ambos o N como a linguagem são dadas nos argumentos da *query*. Temos, então, mais uma *query* onde vamos ter de iterar sobre os *commits* e guardar os utilizadores já vistos num *array*, ou seja, decidimos adotar o mesmo método de *tracking* dos utilizadores da *query* anterior. Mas apenas calcular o número de *commits* de cada utilizador não chega, sendo que, teve de ser adicionada uma condição que verifica a linguagem do repositório correspondente ao *commit* atual, isto pode ser alcançado ao utilizar o `repo_id` do *commit* para procurar nesse repositório qual a linguagem usada (utilizando um *getter*).

Com base no que já foi dito podemos inferir que este trabalho teria sido menos complicado e provavelmente mais legível se não tivéssemos de ter atenção ao encapsulamento das estruturas de dados, pois não teríamos de criar sempre *getters* e *setters* para cada valor que pretendemos obter.

Query 7

A *query* 7 tem como função encontrar repositórios sem *commits* a partir de uma determinada data. Para isso usamos a data dada como parâmetro e comparamo-la com as datas em que cada repositório foi atualizado pela última vez (*updated-at*).

Deste modo, é necessário , para esta questão, procurar o *repo id* e a *description* destes mesmos repositórios sem novas atualizações para, posteriormente, escrevê-los no ficheiro de saída, sendo esse o resultado esperado por esta *query*.

Query 8

Esta *query*, calcula o *top N* de linguagens mais utilizadas a partir de uma determinada data, mais uma vez, tanto N como a data são fornecidos nos argumentos da consulta. Ou seja, vamos ter de iterar sob os *commits*, e manter um registo do número de cada linguagem utilizada a partir da data fornecida. Em primeiro lugar, já que iteramos sob um *array* de *commits* ordenado pela data podemos obter o índice na lista da data por onde pretendemos começar a procura utilizando procura binária no *array*. De seguida, podemos começar o *loop* a partir do índice da data passada como argumento e manter um registo da linguagem e do número de aparição com a ajuda de uma estrutura de dados (nova) chamada *triplet* (estrutura capaz de armazenar diferentes categorias de dados para diferentes utilidades). Após ter a estrutura populada podemos prosseguir à inserção da mesma num *array* e repetimos este processo para todos os *commits*. No final, apenas temos de ordenar por ordem decrescente o *array* que contêm vários apontadores para *triplets* e escrever no ficheiro os N primeiros.

Nota: Antes de adicionar um *triplet*, é sempre verificado se ele já existe no *array*, caso exista apenas incrementamos o valor.

Query 9

À parte : Entramos agora nas *queries* mais problemáticas do projeto inteiro, digo isto não por elas serem muito complexas ou difíceis, mas por relacionarem tanto os utilizadores como os *commits* e repositórios. Devido à nossa implementação da base de dados (o uso de *sorted arrays*) tanto esta *query* como a próxima executam num tempo mais demorado, pois cada iteração pode executar várias procuras noutros catálogos ou iterar duplamente sob catálogos.

A *query* 9 centra-se no conceito de utilizadores amigos, ou seja, de certeza que vamos ter de verificar ambos os *arrays* de seguidores e seguindo do utilizador em questão. Esta provavelmente foi a parte mais desafiante da *query* inteira, pois inicialmente a função que converte *string* em *arrays* de inteiros funcionava incorretamente por isso teve de ser refeita. A maneira com que obtivemos o resultado esperado no que toca ao ser ou não ser um amigo, foi ao procurar o *id* do amigo em ambas listas de seguidores e seguindo, caso estivesse lá, o *id* pertence a um amigo.

Com isto, calcular o *top N* de utilizadores com mais *commits* em repositórios cujo *owner* é um amigo seu (objetivo real da *query* 9) torna-se trivial. Seguindo o mesmo método utilizado na *query* acima de "rastreamento", conseguimos obter um *array* ordenado de estruturas auxiliares que contêm os dados necessários para o *output* cujos N primeiros elementos podem ser "despejados"para o ficheiro de saída. No fundo, esta *query* é igual à número 6, mas sem a condição da linguagem e com a de o utilizador e o dono do repositório terem de ser amigos.

Query 10

Finalmente chegamos à última *query*, esta visa calcular o *top* N de utilizadores com as maiores mensagens de *commit* por repositório, isto, lido à primeira vista parece uma tarefa fácil, porém o facto de ser "por repositório"complica a resolução um pouco. Assim, o que realmente é pedido é que seja calculado para cada repositório os N utilizadores com maior mensagem, isto pode ser interpretado de diferentes formas, mas, ao saber o *output* desejado tudo fica claro:

```
ID1;Login1;Commit_msg_size1;repo_id1
ID2;Login2;Commit_msg_size2;repo_id1
...
ID...;Login...;Commit_msg_size...;repo_id...
```

Listing 4: *Output* da *query* 10.

Assim, podemos concluir que vão ser precisos quatro parâmetros que relacionam os três ficheiros de entrada. Isto é mau, especialmente devido à nossa implementação da base de dados, pois vamos ter de iterar sob os repositórios e sob os *commits* em simultâneo, para verificar se o *commit* pertence ou não ao repositório (comparando o `repo_id` como *id* do *repo*). Além disso, vamos ter de executar a cada iteração nos *commits* uma ou mais procuras binárias nos utilizadores.

Ora todas estas operações são custosas ao longo do tempo, e com *inputs* grandes o tempo de execução é afetado drasticamente (devido à complexidade graças aos *loops* aninhados e constantes procuras), no nosso caso chegando a demorar cerca de 3 minutos a executar. Essa foi a nossa maior dificuldade na execução da *query* 10 e a razão para a qual ela não estar completa.

4 Conclusão

Em suma, apesar dos novos conceitos adotados neste projeto como o encapsulamento e modularidade, conseguimos obter os resultados esperados em relação a essas noções.

No entanto, surgiram vários problemas com diferentes soluções, como, por exemplo, após o parsing, optamos pelo uso de arrays ordenados (ao invés de *AVL Trees* e *hash tables*) para criar os catálogos cuja função é armazenar os dados. Além disso, existiram complicações relacionadas com *memory leaks*, alusivas à tentativa do uso de *hash tables*, e a *scope* dos utilizadores, cuja resolução, apesar de relativamente simples, exigiu algum esforço.

Em relação às *queries*, por um lado, as estatísticas foram de realização acessível, sendo que, apenas precisamos do uso dados com acessos facilitados. Por outro lado, nas *queries* parametrizáveis necessitavamos primeiramente de fazer o *parsing* dos argumentos e, apenas após isso, escrever os resultados do que é pedido pela *query* no ficheiro de saída. A parte mais importante neste tipo de problemas é o cálculo desses mesmos resultados, que, de uma forma geral, são de mais difícil obtenção do que nos casos anteriores.

Ainda assim, existem algumas contrariedades que persistem como o tempo de execução de certas funções devido à necessidade da procura de dados de forma constante.

Deste modo, apesar da imperfeição do trabalho, todos estes recursos usados e soluções encontradas nas adversidades, permitiram-nos diminuir o impacto em termos de complexidade no programa e aprimorar a eficiência deste.