Universidade do Minho
Escola de Engenharia

Análise e Teste de Software

Refabricação Verde de Software

Mestrado Integrado em Engenharia Informática

1º Semestre
2017-2018

A74166 - Pedro Daniel Gomes Fonseca
A74260 - Luís Miguel da Cunha Lima

26 de Janeiro de 2017
Braga

Resumo

Este documento descreve o trabalho prático desenvolvido no âmbito da unidade curricular de **Análise e Teste de Software**, tendo como objetivo aplicar as técnicas e conhecimentos aprendidos e adquiridos na Unidade Curricular a um software disponiblizado pelos docentes.

Neste estudo foram aplicadas técnicas de teste de software, de modo a verificar se este cumpre todos os requisitos especificados, bem como técnicas de análise a sua qualidade, técnicas de refrabicação de código e finalmente técnicas de monitorização do consumo de energia utilizando a framework de teste em JAVA - RAPL e JProfiler.

Conteúdo

Lista de Figuras

Lista de Tabelas

1 Introdução

Na origem da computação a obrigação pela qualidade do software era exclusivamente do programador, por volta das décadas de 1950 e 1960. Apenas pela década de 1970, padrões que garantiam a qualidade foram inseridos no desenvolvimento, e logo se dispersou pelo mundo comercial. A qualidade de software, com o passar do tempo passou a ser vista de maneira diferente, existe uma vigorosa pressão por parte dos consumidores, pela qualidade de desenvolvimento, preço do mercado e tempo de entrega final.

A produção de software é uma atividade bastante complexa que envolve pessoas, técnicas e ferramentas, e quando se trata de projetos vastos a complexidade destes cresce gigantescamente e a presença de falhas torna-se inevitável. Fatores como a especificação errada ou requisitos impossíveis de ser realizados são exemplos de motivos que provocam falhas.

A Garantia de Qualidade de Software(SQA) é um fator bastante importante para qualquer empresa, é o padrão de planeamento e ações estabelecidas para garantir a qualidade do produto. A atividade de teste, uma das funções do grupo SQA, agrupa um imenso pacote de casos de testes, que auxiliam a detetar erros, de modo a revelar a maioria dos erros após o desenvolvimento.

Teste de Qualidade de Software é parte da engenharia de software determinada a analisar detalhadamente todo o código em busca do maior nível de qualidade possível. O principal objetivo dos testes é naturalmente o de encontrar possíveis incorreções para que consigam ser retificados antes da entrega do produto ao consumidor final.

A atividade de teste é de extrema importância no desenvolvimento de software, chave que pode garantir o sucesso. Estando todos os requisitos funcionais corretos, devidamente testados, o cliente e consumidor final ficará satisfeito.

2 Especificação do Software

Neste tópico iremos, de uma forma breve e sucinta pois este não é o objetivo deste estudo, analisar o enunciado do software fornecido pelos docentes desta unidade curricular no qual posteriormente será realizada um conjunto de testes apresentados nos próximos tópicos deste relatório.

O software fornecido tinha como objetivo a criação de um sistema que criasse um serviço de transportes de passageiros através de táxis, e que possibilite ao utilizador realizar uma viagem num destes táxis.

Pedia-se então para desenvolver um programa capaz de auxiliar numa empresa de transporte de pessoas sendo capaz de:

- Registar um utilizador (cliente ou motorista);
- Implementar login no sistema;
- Criar viaturas;
- Associar motoristas a viaturas;
- Solicitar uma viagem escolhendo uma viatura específica ou a mais próxima de si;
- Classificar o motorista,após a viagem;
- Possiblidade de utilizador (cliente ou motorista) ver as viagens que já fez;
- Indicar o total facturado pela viatura ou pela empresa;
- Listar os 10 clientes que mais gastam;
- Listar os 5 motoristas que apresentam mais desvios entre valores previstos para a viagem e o valor final faturado;
- Gravar o estado do programa em ficheiro.

Havia ainda uma lista de funcionalidades propostas avançadas como: gestão de fatores de aleatoriedade, na duraçao da viagem e fiabilidade das viaturas, e criação de viaturas em fila de espera.

3 Estruturas de Dados

Os dados manipulados pelo software em causa são armazenados em estruturas que consistem em Maps. Estas estruturas cumprem todos os requisitos propostos, no entanto dependendo do objetivo podem levar a algumas ineficiências.

Relativamente à performance de operações básicas, o HashMap executa em tempo constante, ou seja, O(1). Posto isto, o HashMap pode ser considerado mais eficiente em geral, sabendo que o HashMap não mantém nenhuma ordem, ou seja, este não fornece qualquer garantia de que o elemento inserido primeiro será imprimido primeiro. Já a complexidade de "get"(obter), "put"(adicionar) e "remove"(remover) operações no TreeMap, é respetivamente O(log(n)). No entanto, os elementos do TreeMap também são ordenados de acordo com a ordenação natural dos seus elementos. Se os objetos deste não puderem ser ordenados conforme a ordem natural, é necessário usar o método compareTo() para classificar os elementos do objeto TreeMap.

Como podemos ver na figura 1, outra das diferenças importantes é que os valores nulos no HashMap são permitidos tanto para as Keys como para os Values (pares Key-Value), já o TreeMap apenas permite que sejam usados valores nulos unicamente para os Values. Resumindo se o objetivo do problema necessitar da ordenação dos dados para que este seja resolvido de uma melhor forma, então deve-se usar o TreeMap, no entanto se o objetivo não precisar de ordenação a melhor solução é usar HashMap.

Map String,Veiculo $veiculos = newHashMap <> ();$
Map String,Ator $atores = newHashMap <> ();$

Os objetos principais do programa são os Veículos e os Atores(Clientes ou Motoristas) sendo que a escolha efetuada para as estruturas da versão original foram dois HashMaps. Um dos HashMap armazena todos os Veiculos e o outro armazena todos os Atores. Podemos ver logo à partida que estas estruturas possibilitam um acesso eficiente aos objetos pretendidos num dado momento, visto que permitem a referência através de, por exemplo, um ID único.

Property \ Map	HashMap	TreeMap
Ordering	not guaranteed	sorted, natural ordering
get / put / remove complexity	O(1)	O(log(n))
Inherited interfaces	Map	Map NavigableMap SortedMap
NULL values / keys	allowed	only values

Figura 1: Diferenças entre tipos de Maps

4 Teste e refabricação do Código

Inicialmente, como referido no enunciado, era necessário proceder à realização de testes unitários, testes de integração e testes de sistema.

Para a realização dos testes unitários implementamos testes às classes consideradas de maior importância e de seguida, de forma a completar a cobertura desses mesmos testes, utilizamos uma ferramenta de geração de testes automaticamente.

Para tal, recorremos à ferramenta Randoop que gera testes unitários assim como testes de regressão, para a posterior refabricação do código realizada numa etapa posterior. Através do uso desta ferramenta garantimos uma maior cobertura de testes às nossas classes, assim como uma maior certeza na eficiência dos mesmos.

No que toca ao funcionamento desta ferramenta, para cada classe que selecionarmos serão geradas duas classes com uma grande variedade de testes aos métodos de cada uma delas, sendo que numa estarão contidos testes unitários, e na outra serão os ditos testes de regressão.

Relativamente aos testes unitários iremos apresentar de seguida alguns dos testes que implementamos de forma a cobrir as funcionalidades desejadas.

Para a classe *Ator* inicialmente decidimos criar várias instâncias das classes *Cliente* e *Viagem* de modo a auxiliar nos vários testes implementados aos métodos desta classe conforme podemos verificar na Figura 2.

```
public void setUp() {
c1 = new Cliente("bruno_1_dantas@hotmail.com","Bruno","123456","Ponte de Lima","15-10-1996", histc1, 60);
c2 = new Cliente("luis.soccer5@gmail.com","Luis","123456","Arcos de Valdevez","23-07-1997", histc2, 60);
v1 = new Viagem(new Coordenada(1,1),new Coordenada(17,6),60,"bruno_1_dantas@hotmail.com",new GregorianCalendar(2017,12,17),5
v2 = new Viagem(new Coordenada(152,151),new Coordenada(153,15),69,"luis.soccer5@gmail.com",new GregorianCalendar(2017,9,15),
v3 = new Viagem(new Coordenada(169,169),new Coordenada(145,1023),100,"pedrofonseca@gmail.com",new GregorianCalendar(2017,12,
v4 = new Viagem(new Coordenada(123,153),new Coordenada(14,314),200,"pedrofonseca@gmail.com",new GregorianCalendar(2017,12,01
}
```

Figura 2: Criação de instâncias na classe de teste AtorTeste()

De seguida, apresentamos três testes à classe ator usando as instâncias criadas anteriormente. São eles, *registaViagemTeste()*, *maiorDesvioTest()* e *viagemEntreDatasTest()*. Testes estes que evidenciam as funções mais importantes da classe *Ator* e por isso é importante ser verificada a sua correta correção. No teste *registaViagemTest* foi criada uma coleção auxiliar onde serão guardadas todas as viagens efetuadas para posteriormente se poder comparar com o Historico de viagens que os clientes efetuaram até então.

Para o método *maiorDesvio* da classe *Ator* o objetivo deste teste era simplesmente criar duas viagens efetuadas pelo mesmo cliente e verificar qual delas terá o percurso maior. Por último mas não menos importante, o teste relativo ao método *viagemEntreDatas()* segue o mesmo raciocínio do teste da figura 3 explicado anteriormente, onde é criada uma lista auxiliar onde são também guardadas as viagens realizadas por um cliente e de seguida criadas duas datas para teste verificando se existem viagens entre estas mesmas datas e se as coleções auxiliares e viagens do respetivo cliente são iguais.

```
@Test
public void registaViagemTest() {
    Set<Viagem> aux = new TreeSet<Viagem>();
    c1.registaViagem(v1);
    aux.add(v1);
    assertEquals(aux,c1.getHistorico());
    c1.registaViagem(v2); c1.registaViagem(v4);
    aux.add(v2); aux.add(v4);
    assertEquals(aux,c1.getHistorico());
}
```

Figura 3: Teste a classe ator - método RegistaViagem()

```
// Teste ao mÃ©todo maiorDesvio
@Test
public void maiorDesvioTest() throws NenhumaViagemException {
    c1.registaViagem(v1);
    c1.registaViagem(v3);
    assertEquals(v3,c1.maiorDesvio());
}
```

Figura 4: Teste a classe ator - método MaiorDesvio()

```java
@Test
public void viagemEntreDatasTest() throws InvalidIntervalException {
    List<Viagem> aux = new ArrayList<Viagem>();
    c1.registaViagem(v3);
    c1.registaViagem(v2);
    aux.add(v3);
    GregorianCalendar data1 = new GregorianCalendar(2017,11,10);
    GregorianCalendar data2 = new GregorianCalendar(2018,01,01);
    assertNotNull(c1.viagensEntreDatas(data1, data2));
    assertEquals(aux,c1.viagensEntreDatas(data1,data2));
}
```

Figura 5: Teste a classe ator - método ViagemEntreDatas()

O imagem seguinte corresponde a um exemplo de um teste relativo a classe utilizador ao método adiciona em que o objetivo é adicionar um utilizador ao sistema. Para verificar a correta resolução deste método o objetivo passava por criar um *Map* auxiliar e adicionar utilizador pretendido ao sistema e a este mesmo *Map* e verificar se estes são iguais, ou seja, se contém os mesmos utilizadores anteriormente adicionados. Uma possibilidade também testada e mostrada na imagem, é quando o cliente/motorista tenta adicionar um utilizador já existente (com email idêntico), o sistema não deve deixar este ser adicionado ao *Map* com todos os utilizadores já existentes, uma vez que é perdida toda a coerência e integridade da aplicação.

```java
@Test
public void adicionaTest() {
    Utilizadores aux;
    // NÃ£o existe
        aux = new Utilizadores();
        assertNotNull(aux);
        try { aux.adiciona(c); aux.adiciona(m); }
        catch (EmailAlreadyInUseException e) {}
        assertEquals(u, aux);
    // JÃ¡ existe (nÃ£o deve adicionar)
        aux = new Utilizadores(atores);
        assertNotNull(aux);
        try { u.adiciona(c); }
        catch (EmailAlreadyInUseException e) {}
        assertEquals(u, aux);
}
```

Figura 6: Teste a classe utilizador - método adiciona()

De forma a eliminar maus cheiros no código e tornar o nosso código mais legível e mais organizado recorremos a uma ferramenta para realizar a refabricação do código.

A refabricação do código pretende melhorar o design do nosso código, isto é, pretende torná-lo mais legível para outros programadores que possam ter de desenvolver código baseado no nosso, ou simplesmente, para um eventual update às funcionalidades da aplicação desenvolvida, de forma a permitir uma melhor compreensão do código que já se encontra feito. Este processo tem também o propósito de facilitar a descoberta de bugs no nosso código, assim como uma melhoria significativa na nossa produtividade como programadores, pois quanto mais fácil for a nossa compreensão do programa que estamos a construir, maior será a nossa capacidade de desenvolver bom código para o restante.

De seguida, apresentamos um exemplo de como o código refabricado permite melhorar todos os aspetos referidos anteriormente. Nestas porções de código temos uma função que se torna muito mais legível e compreensível depois de refabricada.

Como podemos observar um método que estava talvez demasiado longo, através de uma ferramenta de refactoring foi tranformado num método bastante diminuto pois algumas das operações que ali eram feitas foram delegadas para outros métodos, posteriormente criados, de forma a melhorar o nosso código e a sua execução.

```java
public void funcionalidades() {
    Scanner sc = new Scanner(System.in);

    System.out.println("\n-----------Extra-----------");
    System.out.println("1 - Listar Utilizadores");
    System.out.println("2 - Listar os 10 clientes que mais gastam");
    System.out.println("3 - Listar viaturas");
    System.out.println(
            "4 - Listar os 5 motoristas com maior desvio entre os valores previstos para as viagens e
    System.out.print("Opção: ");
    int op = sc.nextInt();

    System.out.println("\n-----------Extra-----------");
    switch (op) {
    case 1:
        trataListarUtilizadores();
        break;
    case 2:
        top10ClientesGastadores();
        break;
    case 3:
        listarVeiculos();
        break;
    case 4:
        top5MotoristasComMaiorDesvio();
        break;
    default:
        System.out.println("Opção Inválida!");
        break;
    }

    sc.close();
}
```

Figura 7: Código da função 'funcionalidades' antes de ser refabricado

Apresentamos ainda outro exemplo das vantagens oferecidas pela refabricação.

```java
public void funcionalidades() {
    Scanner sc = new Scanner(System.in);

    displayOptions();
    int op = sc.nextInt();

    executeOption(op);

}
```

Figura 8: Código da função 'funcionalidades' refabricado

```java
private void run() {
    initData();

    do {
        menuPrincipal.executa();
        casesHandler();
    } while (menuPrincipal.getOpcao() != 0);

    saveData();
}
```

Figura 9: Código da função 'run' antes de ser refabricado

```java
private void run() {
    System.out.println("A carregar os dados...");
    try{
       carregaEstado("Appdadosbin");
    }
    catch (IOException e){
        System.out.println("Erro no ficheiro.");
    }
    catch (ClassNotFoundException e){
        System.out.println("Erro nas classes.");
    }
    do {
        this.menuPrincipal.executa();
        switch (menuPrincipal.getOpcao()) {
            case 1: this.menuLogin.executa(this.utilizadores,this.veiculos);
                    break;
            case 2: this.menuRegisto.executa(this.utilizadores,this.veiculos);
                    break;
            case 3: this.funcionalidades();
                    break;
        }
    } while (menuPrincipal.getOpcao()!=0);

    System.out.println("A guardar os dados...");
    try{
       guardaEstado("Appdadosbin");
       escreveEmFicheiroTxt("Appdadosbin.txt");
    }
    catch (IOException e){
        System.out.println("Erro no ficheiro.");
    }

    System.out.println("Até breve!...");
}
```

Figura 10: Código da função 'run' refabricado

Este processo por vezes pode causar alterações indesejadas ao código e como tal é necessário garantir que os diferentes métodos das diferentes classes mantêm o resultado esperado da implementação dos mesmos. É neste ponto que entram os testes de regressão que servem para esse mesmo propósito. Como referido anteriormente a geração destes testes foi delegada à ferramenta Randoop e o seu funcionamento explicado anteriormente.

5 Refabricação Verde e Análise de Performance

Vivendo nós numa era tecnológica onde a maioria dos utilitários que usamos no dia-a-dia são dispositivos eletrónicos que para o seu funcionamento necessitam de energia elétrica e com o latente declínio de recursos naturais no nosso planeta é necessário exigir aos desenvolvedores de código uma preocupação máxima em reduzir o consumo energético das aplicações por si criadas de forma a manter uma certa sustentabilidade de vida no planeta Terra.

Como tal, após a realização dos vários ajustes ao código original, teremos de proceder a uma análise extensa às repercussões dessas mesmas alterações, tanto tem termos de performance como de consumo energético mas também sem nunca esquecer, quanto ao consumo de memória por parte do programa.

Para tal, recorremos à ferramenta RAPL(Running Average Power Limit), que nos providencia com dados efetivos do consumo energético da nossa aplicação para nos ajudar a analisar as alterações efetuadas ao código original.

De forma a podermos tirar ilações sobre o referido anteriormente, procederemos inicialmente à análise dos dados recebidos da execução do programa original, comparando-os seguidamente com os dados relativos ao programa refabricado.

5.1 Testes de Performance

De forma a comparar o desempenho da otimização realizada com a versão original do programa, foram realizados testes de performance. As três vertentes exploradas foram:

- Tempo de execução
- Consumo de memória
- Consumo energético

Para tentar cobrir vários cenários, as duas versões do software foram expostas a vários conjuntos de casos de testes. Estes conjuntos consistiam em testes com:

- Muitos Clientes
- Muitos Veículos
- Muitas Viagens

Decidimos fazer inclusão de mais dados para além daqueles que os ficheiros de input forneciam pois constatamos que seria mais fácil notar as diferenças entre as versões do software se a carga a que este fosse submetido fosse maior, pois algumas diferenças não se faziam notar quando os testes decorriam apenas com os dados presentes nos ficheiros de input fornecidos. Desta forma, podemos verificar mais facilmente em que situações se encontram as maiores diferenças de performance entre as versões. Para assegurar a credibilidade e validade dos resultados obtidos, cada teste efetuado foi executado mais do que uma vez. O valor considerado para cada caso de teste foi a média entre os resultados obtidos com as suas execuções.

5.1.1 Tempo de Execução

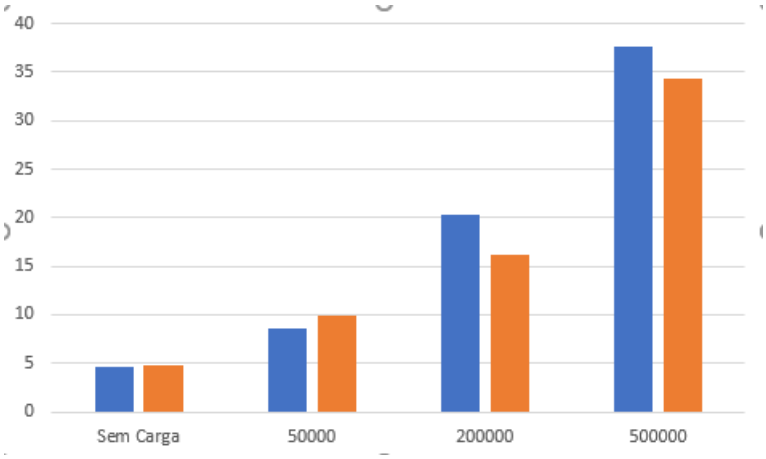

Figura 11: Comparação dos tempos de execução para casos de teste com muitos clientes/motoristas

As refabricações efetuadas revelaram ganhos de performance ao nível de tempo de execução do programa para este caso de teste, apesar das melhorias não serem muito significativas pois estas só são notadas quando á medida que se aumenta o número de clientes/motoristas na aplicação, tendo mesmo testes onde o tempo de execução do original é inferior ao do refabricado. Á medida que se aumenta o número de utilizadores a diferença torna-se cada vez mais substancial entre os tempos de execuções das duas versões da aplicação, chegando-se a verificar uma descida de aproximadamente 9% no caso de teste com 500000 utilizadores - diferença de 3,33s para a versão original comparando com a versão otimizada. Analisando a relação entre o aumento do tempo de execução e o aumento do número de clientes/motoristas os tempos não aumentam significativamente uma vez que estamos a lidar com estruturas (neste caso HashMaps) em que operações de inserção e de procura apresentam uma complexidade temporal de O(1), tempo constante, uma vez que não necessita de atravessar uma grande quantidade de elementos sequencialmente até localizar o objetivo, sendo isto feito diretamente e em pouco tempo.

- 50000 utilizadores: 8,53 segundos – 9,93 segundos - subida de 12,89%
- 200000 utilizadores: 20,2 segundos – 18,2 segundos - descida de 9,9%
- 500000 utilizadores: 37,6 segundos – 34,2 segundos - descida de 8,8%

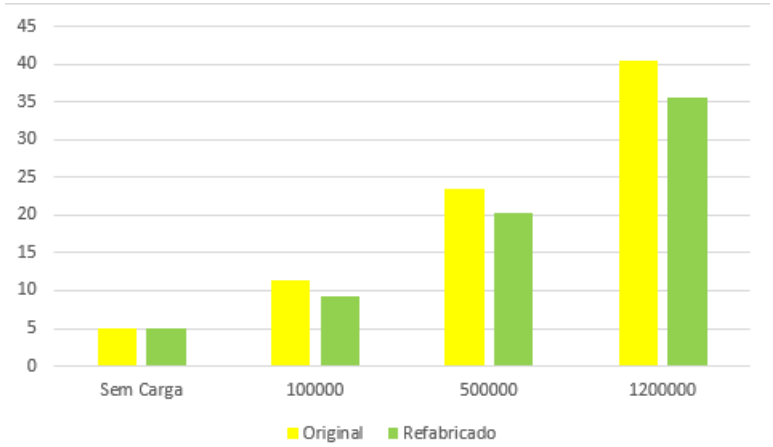

Figura 12: Comparação dos tempos de execução para casos de teste com muitos veiculos

Como no caso de teste anterior, também com muitos veículos se verificam diferenças substanciais nos tempos de execução das duas versões do software. Uma vez que estes conjuntos de testes carregam grandes quantidades de dados no sistema da aplicação e as operações efetuadas sobre este tipo de dados não implicam ordenações (só para determinadas tarefas extras é que será necessário ordenação) faz com que apresente vantagens, dadas as circunstâncias do problema, face a, por exemplo, *ArrayLists* se fosse o caso da escolha das estruturas por parte dos programadores do trabalho original.

- 100000 veiculos: 11,4 segundos – 9,89 segundos - descida de 13,2%
- 500000 veiculos: 23,6 segundos – 20,2 segundos - descida de 14,4%
- 1200000 veiculos: 40,8 segundos – 35,6 segundos - descida de 12,7%

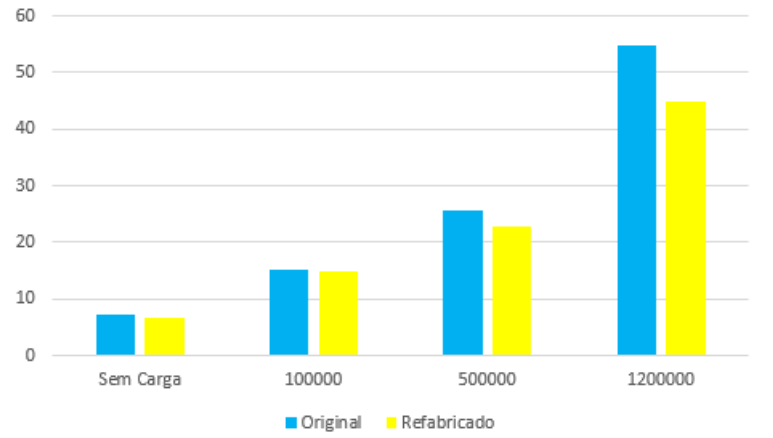

Figura 13: Comparação dos tempos de execução para casos de teste com muitas viagens

Neste terceiro caso de teste, em que o software é submetido a uma carga de muitas viagens efetuadas por um utilizador, as diferenças de tempo de execução entre as duas versões continuam a ser notadas, mas para além disso é neste terceiro conjunto de teste que as descidas entre os resultados são mais acentuadas.

Posto isto, as otimizações feitas ao nível das operações de viagens provam serem bastante eficazes uma vez que as diferenças percentuais calculadas comparando a versão original e a versão refabricada pelo grupo são superiores ás calculadas nos casos de teste com muitos utilizadores ou muitos veículos.

- 100000 viagens: 15,25 segundos – 14,8 segundos - descida de 2,9%
- 200000 viagens: 25,5 segundos – 22,6 segundos - descida de 11,37%
- 500000 viagens: 54,7 segundos – 44,93 segundos - descida de 17,9%

5.1.2 Consumo de Memória

Nesta tópico iremos descrever e comparar os consumos de mémoria da solução original e da solução refabricada testadas para diferentes instâncias de objetos adicionadas. Este consumo de memória foi medido com o auxílio de um *profiler*, o Jprofiler, um UI intuitivo que ajuda a resolver e compreender problemas de performance. Comparativamente com a solução inicial, o grupo esperava que a solução otimizada obtivesse um consumo de memória idêntico, uma vez que, que são usadas as mesmas estruturas de dados (mapeamentos que obrigam a um armazenamento adicional para a chave). De facto, é possível verificar observar esta igualdade, medindo os respetivos consumos. De seguida, sobrecarregou-se o software com 1000000 utilizadores e 1000000 de veículos, esperando algum tempo, após a inserção da carga para ter a certeza nas medições efetuadas.

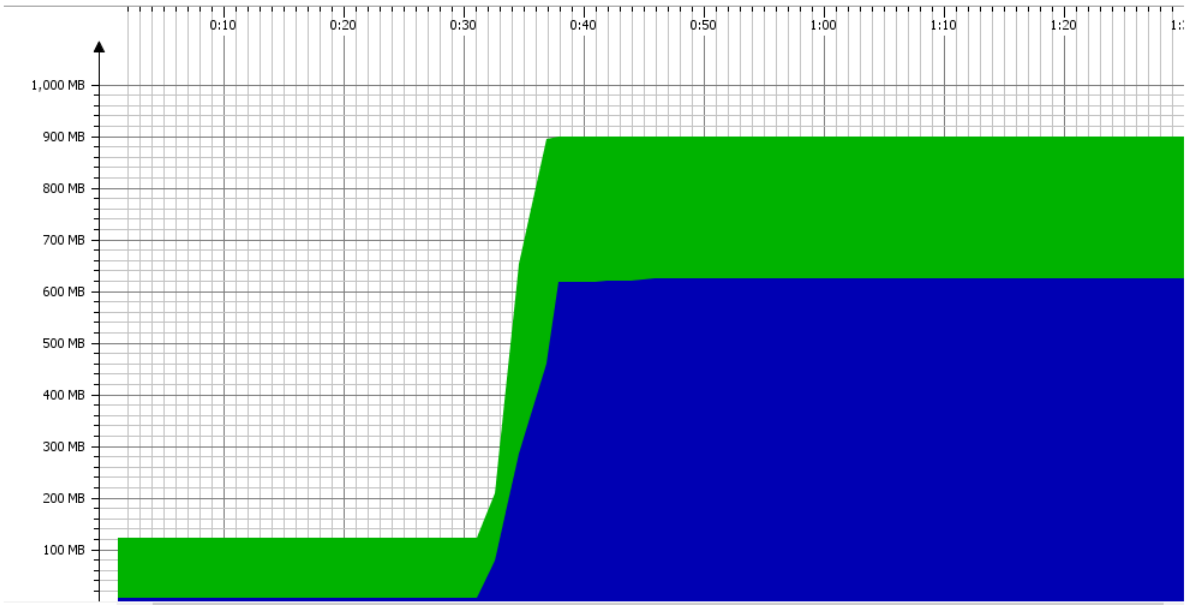

Figura 14: Consumo memória software original com carga

Na figura acima podemos ver inicialmente que o consumo de memória sobe muito pouco e se mantém, este crescimento deve-se a necessidade de criar novos objetos e estruturas auxiliares como por exemplo os *Maps* para o armazenamento dos utilizadores. O pico seguinte, por volta dos 35 segundos, deve-se ao momento em que é inserida a carga. Este pico a azul corresponde a memória usada ocupando cerca de 624 MB tendo ainda de espaço livre 279 MB por usar, pico a verde.

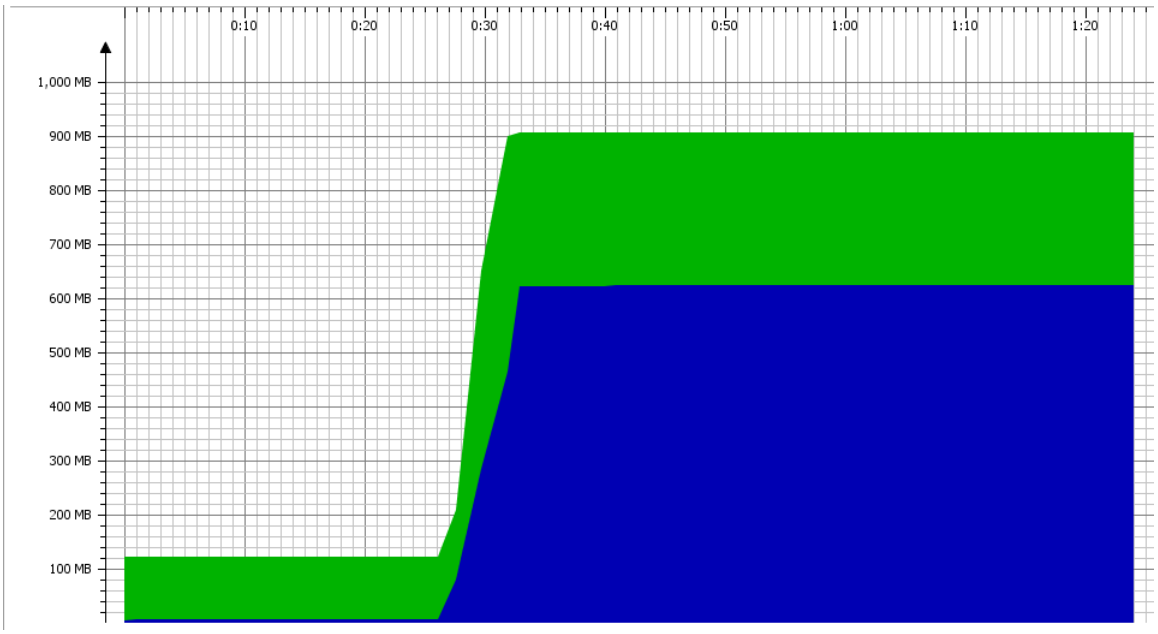

Figura 15: Consumo memória software otimizado com carga

Neste teste ao software otimizado encontramos praticamente a mesma solução, como já tínhamos referenciado em cima, uma vez que as estruturas principais não foram modificadas.

5.1.3 Consumo de energia

Neste tópico será calculado e observado o consumo energético da solução original e da solução apresentada pelo grupo, depois de trabalhada conforme explicada em tópicos anteriores. O consumo energético será monitorizado sobre 3 componentes: CPU, GPU e Package. Para efetuar estas medições, foi utilizado jRAPL, uma framework para fazer profiling de programas Java. Utilizando esta ferramenta, é possível medir o consumo no início do programa e no final, e obter o total gasto durante a execução do programa.

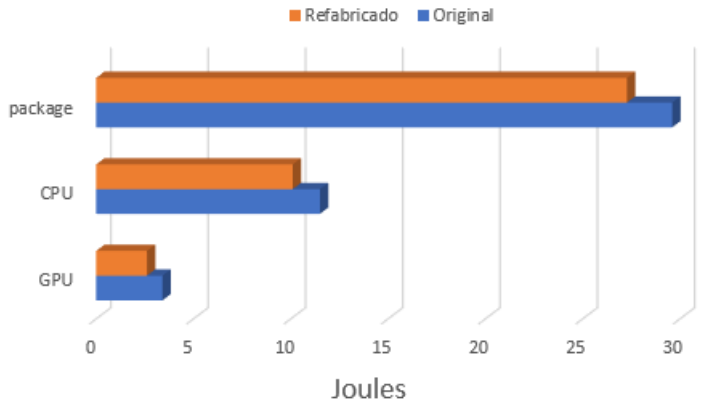

Figura 16: Comparação consumo energia do original e refabricado sem carga

Na figura acima, podemos observar a compararação dos consumos de energia entre a solução original e a nossa solução do problema. Em termos de GPU, os valores obtidos para ambas as versões do programa são bastante diminutos comparando com valores de outros parâmetros como CPU ou até mesmo o package. Visto isto, numa breve análise concluímos, sem carga no sistema, que o GPU praticamente não é utilizado, ou seja, não esta a ser atribuído trabalho a esta componente. Relativamente a componete de CPU, a solução refabricada consome menos 12% aproximadamente do que a solução original. Esta melhoria não é muito significativa, uma vez que operações de pesquisa/adição/remoção de elementos nas estruturas de dados não foram alteradas, apenas alterados simples detalhes que não chegam para realizar diferenças mais significativas na utilização do CPU entre as duas versões.

Como última componente em falta, package, este apresenta um comportamente semelhante ao do trabalho realizado pelo CPU descrito anteriormente. Uma vez que o package consiste na globalidade do processador, ou seja, engloba as outras componentes, é normal ter os valores de consumo energético maiores do que todas as outras componentes, como podemos ver nos resultados recolhidos através do RAPL, pois esta componente também os inclui. Como podemos ver na figura, pelos dados recolhidos temos uma redução pouco significativa relativamente a esta componente entre as duas versões testadas, tendo uma descida de 8% aproximadamente.

Mesmo com valores das componentes variando muito pouco, podemos afirmar que a versão otimizada realiza trabalho de uma forma mais eficiente do ponto de vista energético. Com o objetivo de ter uma maior perceção dos ganhos a nível do consumo o grupo decidiu realizar outro caso de teste em que é adicionada carga ao programa. Esta carga consiste em adicionar 200000 utilizadores e 1000000 veículos. Na imagem seguinte podemos ver os resultados obtidos.

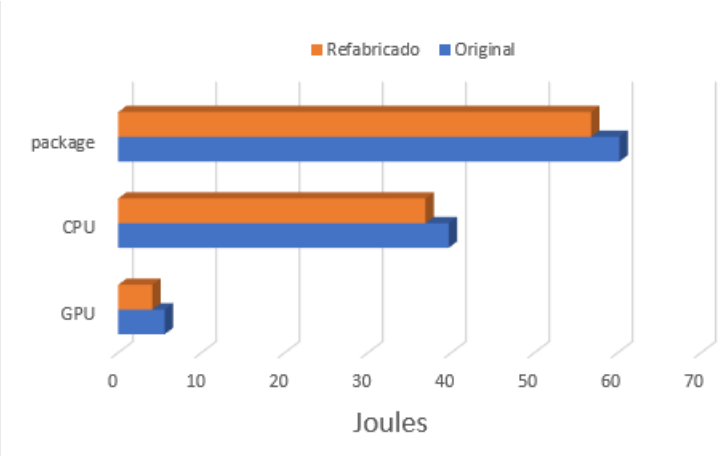

Figura 17: Comparação consumo energia do software original e refabricado com carga

Como podemos verificar, os valores dos resultados são superiores uma vez que é injetada carga no sistema como seria de esperar, mas apesar disso as conclusões verificadas são semelhantes. Resumindo, comparando com os resultados dos testes anteriores, sem carga adicionada, a versão produzida pelo grupo tem uma reação mais leve em termos energéticos do que a versão original.

6 Conclusão

A realização deste projeto proporcionou ao grupo a oportunidade de adquirir uma maior experiência na utilização de ferramentas tanto de realização de testes automáticos como também acima de tudo, ferramentas de medição de desempenho, quer energético quer ao nível de recursos computacionais, como o RAPL, Randoop, AutoRefactor e JProfiler.

Através da análise do código fonte do programa original, fomos capazes de melhorar certos aspetos de forma a colmatar falhas de implementação ao nível do design do programa utilizando algumas das ferramentas referidas anteriormente. Com o grande número de testes realizados conseguimos obter uma cobertura de todo o programa e com a sua consequente refabricação conseguimos obter melhores resultados em alguns aspetos, ou seja no que toca à performance e ao consumo de recursos computacionais.

Através dos testes de regressão realizados garantimos também, que as alterações realizadas ao código, feita a sua refabricação, não tiveram repercussão nos resultados anteriormente obtidos, mas surtiram efeito no que toca aos aspetos de consumo dos recursos.

Foi-nos também possível observar ao detalhe as alterações denotadas após a modificação de certos excertos de código, e ganhar a perceção do que, aparentemente pequenas alterações, podem causar no plano geral das coisas.

As melhorias quer ao nível do tempo de execução quer ao nível do consumo energético e computacional, cujas causas comuns foram as alterações realizadas na versão otimizada, provam latentemente a grande importância que a refabricação do código tem em todo o código desenvolvido hoje em dia.