Refabricação Verde de Software

Análise e Teste de Software

Lisandra Silva A73559
Rogério Moreira A74634
Hélder Sousa A58148

26 de Janeiro de 2018

Conteúdo

Introdução

O projeto apresentado de Análise e Teste de Software tem como objetivo aplicar os conhecimentos adquiridos durante as aulas da Unidade Curricular, nomeadamente nas suas duas vertentes: análise de software e teste de software. Foi apresentado ao grupo um projeto de software desenvolvido em Java e onde se pretendia detetar bad smells e fazer a refabricação do código em questão, criar testes para ver se o software cumpre os requisitos propostos, analisar a qualidade dos testes desenvolvidos e por fim, monitorizar o consumo de energia durante o uso da aplicação.

Capítulo 1

Análise do Software

1.1 Teste de Software

Não se pode garantir que todo software funcione corretamente, sem a presença de erros, visto que os mesmos muitas vezes possuem um grande número de estados com fórmulas, atividades e algoritmos complexos. Falhas podem ser originadas por diversos motivos. Por exemplo, a especificação pode estar errada ou incompleta, ou pode conter requisitos impossíveis de serem implementados, devido a limitações de hardware ou software. A implementação também pode estar errada ou incompleta, como um erro de um algoritmo. Portanto, uma falha é o resultado de um ou mais defeitos em algum aspeto do sistema. O teste de software pode ser visto como uma parcela do processo de qualidade de software e quando falamos em testes de software devemos sempre lembrar que estes são divididos em diversos tipos, de acordo com seu objetivo particular, de seguida apresentamos os testes desenvolvidos de acordo com o seu tipo para a aplicação em estudo.

1.1.1 Testes Unitários

O teste unitário ou de unidade é uma modalidade de testes que se concentra na verificação da menor unidade do projeto de software. É realizado o teste de uma unidade lógica, com uso de dados suficientes para se testar apenas a lógica da unidade em questão. Em sistemas construídos com uso de linguagens orientadas a objetos, como Java, essa unidade pode ser identificada como um método, uma classe ou mesmo um objeto.

Assim, o teste unitário testa o menor dos componentes de um sistema de maneira isolada. Cada uma dessas unidades define um conjunto de estímulos (chamada de métodos), e de dados de entrada e saída associados a cada estímulo. As entradas são parâmetros e as saídas são o valor de retorno, exceções ou o estado do objeto.

Esse tipo de teste é de responsabilidade do próprio desenvolvedor durante a implementação do sistema, isto é, após codificar uma classe, por exemplo, ele planeja, codifica e executa o teste de unidade.

Os testes de unidade são considerados o estagio inicial da cadeia de testes a qual um software pode ser submetido. Essa categoria de testes não atende a testar toda a funcionalidade de uma aplicação, que fica a cargo de outros testes, como de integração ou de performance.

JUnit

O JUnit possibilita a criação das classes de testes. Estas classes contêm um ou mais métodos para que sejam realizados os testes, podendo ser organizados de forma hierárquica, de forma que o sistema seja testado em partes separadas, algumas integradas ou até mesmo todas de uma só vez. Além disso, esta framework tem como objetivo facilitar a criação de casos de teste, além de permitir escrever testes que retenham seu valor ao longo do tempo, ou seja, que possam ser reutilizáveis.

Testes realizados para a UMER

O primeiro passo foi testar se os métodos mais relevantes da classe `cliente` estavam corretamente implementadas de modo a verificar se a classe estava vulnerável a determinadas falhas. Assim, foi criada a classe `atorTeste` que realiza os seguintes testes:

- `testarRegistaViagem()` - verifica se quando uma viagem é registada ela fica registada no histórico do ator
- `testarMaiorDesvio()` - verifica se quando se tenta invocar métodos sobre viagens e não existe nenhuma viagem registada se é lançada a excepção `NenhumaViagemException`
- `public void testarEquals()` - verifica se o método equals da classe `ator` está bem implementado
- `testarViagemEntreDatas()` - verifica se o método `viagemEntreDatas` se encontra corretamente implementado

De seguida fizeram-se alguns testes sobre a classe `Coordenada`, nomeadamente sobre os métodos mais relevantes a mesma:

- `testarEquals()` - verifica através de vários testes se o método `equals` está bem implementado
- `testarDistancia()` - verifica se o método que calcula a distância entre duas coordenadas é corretmente calculada

Por último realizaram-se testes à classe `motorista`, nomeadamente aos métodos mais complexos, uma vez que os restantes métodos das outras classes correspondiam apenas a métodos `get` e `set` que por uma inspeção do código se verificou estarem bem implementados.

- `testarPrecoViagem()` - verifica se o preço de uma viagem é corretamente calculado. Ao executar este teste verificamos que esta classe está vulnerável a um erro pois aceita um número negativo para o preço e devolve um resultado errado
- `testarAtualizaDados()` - este teste verifica se os dados são atualizados quando o motorista faz uma viagem
- `testarTotalFaturado()` - verifica se o método `registaViagem` está a funcionar, nomeadamente se o total faturado pelo motorista após o registo da viagem é atualizado quando se regista uma nova viagem

1.1.2 Cobertura

O Cobertura é uma ferramenta para calcular a percentagem de código coberta pelos códigos criados. Pode ser usado, por exemplo, para detetar partes do código que não estão a ser testadas.

Coverage Report - All Packages

Package	# Classes	Line Coverage		Branch Coverage		Complexity
(default)	25	0%	0/308	0%	0/206	2.042
All Packages	25	0%	0/308	0%	0/206	2.042

Classes in this Package	Line Coverage		Branch Coverage		Complexity
App	N/A	N/A	N/A	N/A	0
Ator	N/A	N/A	N/A	N/A	0
Carrinha	N/A	N/A	N/A	N/A	1.833
Carro	N/A	N/A	N/A	N/A	1.833
Cliente	N/A	N/A	N/A	N/A	1.818
ComparadorDesvio	N/A	N/A	N/A	N/A	3
ComparadorGastoDescrescente	N/A	N/A	N/A	N/A	5
ComparadorMotoristaDesvioMaximo	N/A	N/A	N/A	N/A	11
Coordenada	N/A	N/A	N/A	N/A	2.273
EmailAlreadyInUseException	N/A	N/A	N/A	N/A	1
EmailDoesNotExistException	N/A	N/A	N/A	N/A	1
InvalidIntervalException	N/A	N/A	N/A	N/A	1
Login	0%	0/308	0%	0/206	0
Menu	N/A	N/A	N/A	N/A	2
Mota	N/A	N/A	N/A	N/A	1.833
Motorista	N/A	N/A	N/A	N/A	0
NenhumaViagemException	N/A	N/A	N/A	N/A	1
Registo	N/A	N/A	N/A	N/A	7.75
Utilizadores	N/A	N/A	N/A	N/A	0
ValueOutOfBoundsException	N/A	N/A	N/A	N/A	1
Veiculo	N/A	N/A	N/A	N/A	1.571
Viagem	N/A	N/A	N/A	N/A	2.211
atorTeste	N/A	N/A	N/A	N/A	1.286
coordeanadaTest	N/A	N/A	N/A	N/A	1
motoristaTest	N/A	N/A	N/A	N/A	1

Figura 1.1: Relatório do Cobertura

Foi importante correr o Cobertura para ser possível o grupo ter uma melhor perceção da percentagem de código coberta pelos testes anteriormente explicitados.

1.2 Análise do Código

Foram usadas algumas técnicas exploradas durante as aulas para detetar e corrigir os bad smells, analisando assim a performance e correção do código.

1.2.1 Bad Smells

Em *Software Engineering* , um bad smell é uma designação para um excerto de código que apesar de não estar errado tecnicamente nem impedir o programa de funcionar, indicam fraquezas ou pontos negativos no código que podem por em causa a performance e/ou correção do software.

1.2.2 Refactoring

O processo de *Refactoring* diz respeito ao ato de modificar um sistema de software, melhorando a estrutura interna do código, sem alterar o comportamento externo. Foi por esta razão que desenvolvemos os testes do software em primeiro lugar, garantindo assim que os resultados seriam os mesmos antes e depois das alterações. O uso da técnica de refabricação aprimora a conceção do software, corrigindo eventuais bad smells presentes durante a sua conceção.

Listing 1.1: Original

```
if(ms.size() == 0) System.out.
    println("Nao existem
    motoristas no sistema UMeR.")
    ;
else for(Motorista m : ms)
try{
System.out.println("Nome: "+m.
    getNome()+" e-mail: "+m.
    getMail()+" Desvio: "+String.
    format("\%.2f",m.maiorDesvio
    ().getDesvio()));}
catch(NenhumaViagemException e){
    System.out.println("Objeto
    corrompido?");}}
```

Listing 1.2: Refactored

```
if(ms.isEmpty()) {System.out.
    println("Nao existem
    motoristas no sistema UMeR.")
    ;
} else {
for(Motorista m : ms) {
try{
System.out.println("Nome: "+m.
    getNome()+" e-mail: "+m.
    getMail()+" Desvio: "+String.
    format("\%.2f",m.maiorDesvio
    ().getDesvio()));
}catch(NenhumaViagemException e)
    {System.out.println("Objeto
    corrompido?");}}}}
```

Capítulo 2

Análise de energia

2.1 Green Software

No campo das tecnologias de informação, a poupança de energia costumava ter o seu foco na eficiência energética do hardware, no entanto quando o hardware opera sob software ineficiente, a sua eficiência também pode diminuir substancialmente. Assim, têm sido feitos esforços na medida de otimizar o consumo energético por parte do software, surgindo a definição de *Green Software*, que se descreve como o "software que pode ser desenvolvido e usado eficientemente com o mínimo impacto no ambiente".

Contudo, esta otimização exige numa primeira instância que se seja capaz de medir e monitorizar o consumo energético, neste contexto surgem ferramentas que permitem recolher métricas sobre a performance energética. A recolha destas métricas deve ser feita durante o programa em execução e por isso é chamada Análise Dinâmica.

2.1.1 RAPL

Running Average Power Limit foi introduzido pela *Intel* e está presente em todos os processadores Intel x86. O RAPL é uma ferramenta que permite estimar qual a quantidade de energia que está a ser consumida por todos os cores do CPU.

Assim, no âmbito deste projeto usou-se o RAPL para medir os consumos de energia antes e depois da refabricação do código, de modo a perceber a influência que a eliminação de maus cheiros de código tem sobre a eficiência energética.

2.2 Resultados durante a execução

Para medir as várias métricas durante a execução do software fornecido foram definidos vários testes independentes, e feitas as medições no programa antes e depois da refabricação. Foi necessário medir várias vezes para garantir o menor erro possível, sendo que os resultados apresentados são a média ponderada das medições. De seguida listam-se os resultados.

2.2.1 Tarefa 1

Tarefa: Operações repetidas, mostrar os dados do cliente 100 vezes.

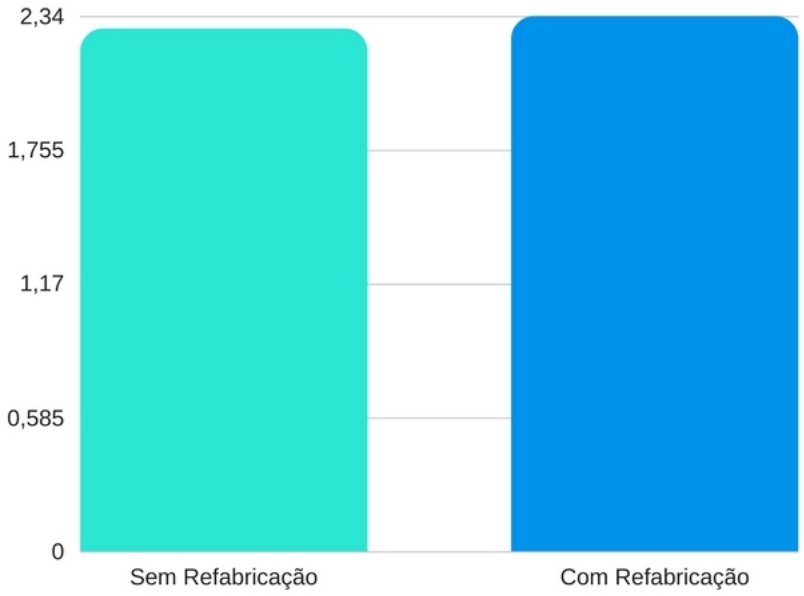

Figura 2.1: Power Consumption of Dram

Figura 2.2: Power Consumption of CPU

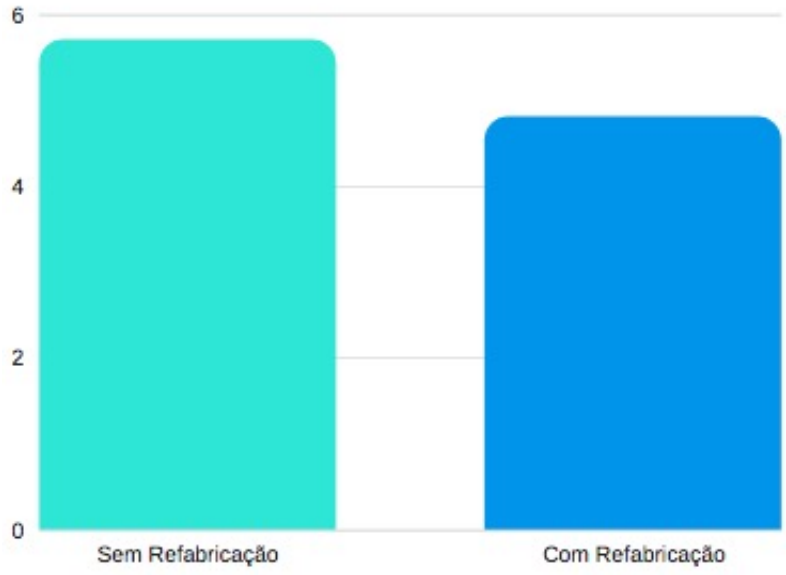

Figura 2.3: Power Consumption of Package

2.2.2 Tarefa 2

Tarefa: Sequência de operações.

1. Inserir 2 veículos;
2. Inserir 1 cliente;
3. Inserir 2 motoristas;
4. Fazer login motorista1;
5. Associar motorista1 a veículo 1;
6. Fazer logout motorista1;
7. Fazer login motorista2;
8. Associar motorista2 a veiculo2;
9. Fazer logout motorista2;
10. Fazer login cliente;
11. Chamar veiculo P/ cliente;
12. Fazer logout cliente;
13. Logout.

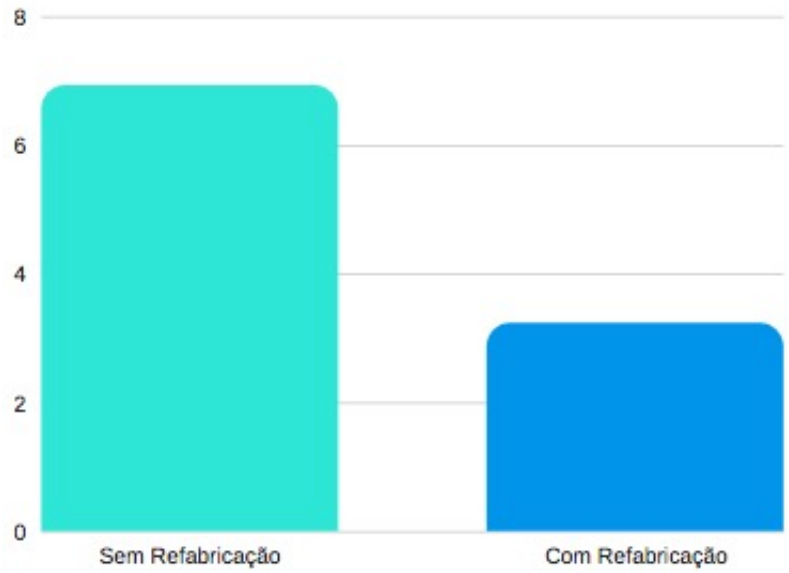

Figura 2.4: Power Consumption of Dram

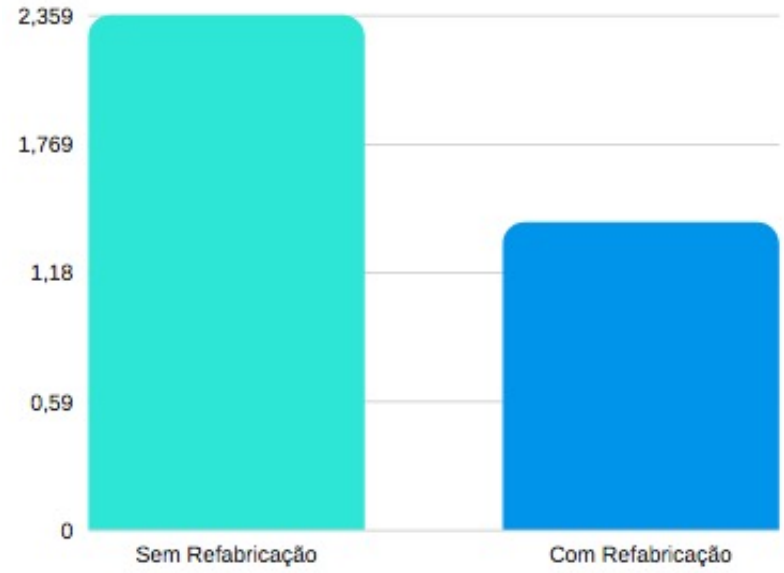

Figura 2.5: Power Consumption of CPU

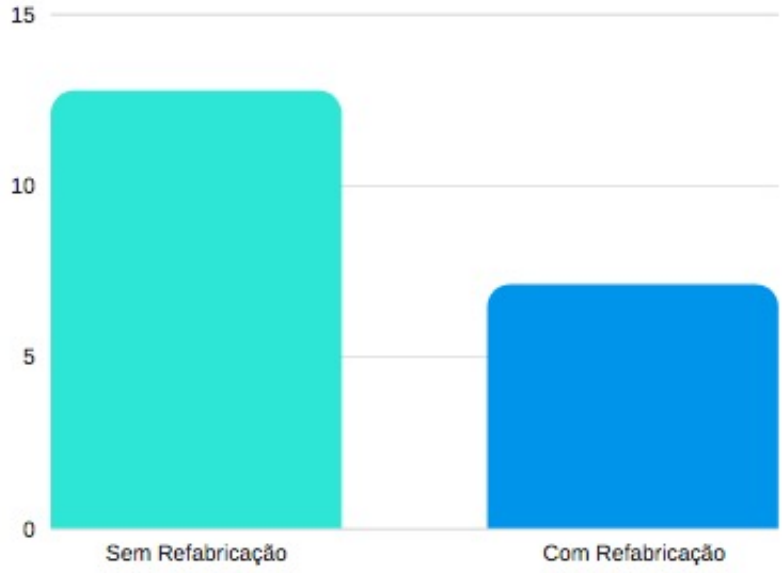

Figura 2.6: Power Consumption of Package

2.2.3 Resultados durante os testes unitários

Para além das medições durante o tempo de execução medimos também a energia usada nos vários testes unitários.

testarAtualizaDados

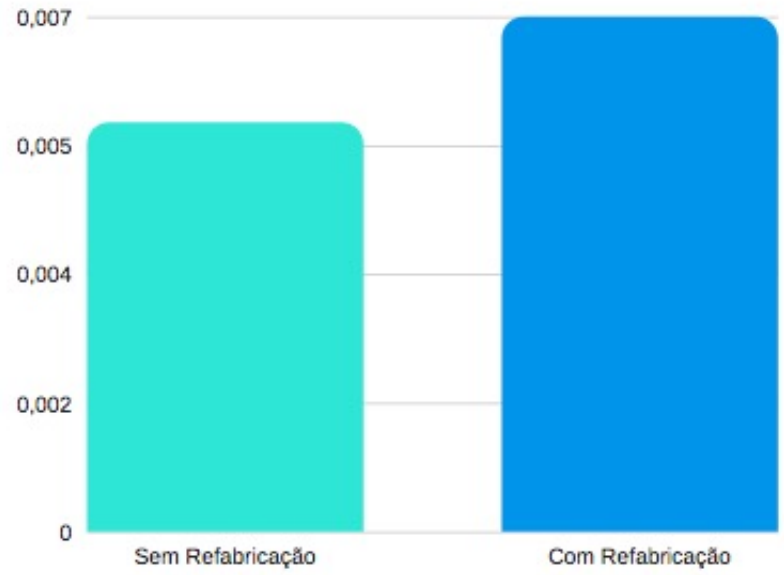

Figura 2.7: Power Consumption of Dram

Figura 2.8: Power Consumption of CPU

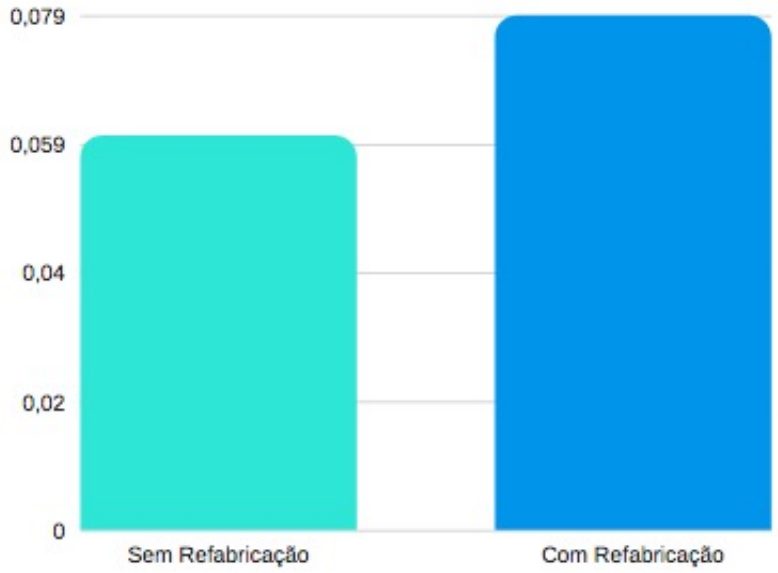

Figura 2.9: Power Consumption of Package

testarEquals(atorTeste)

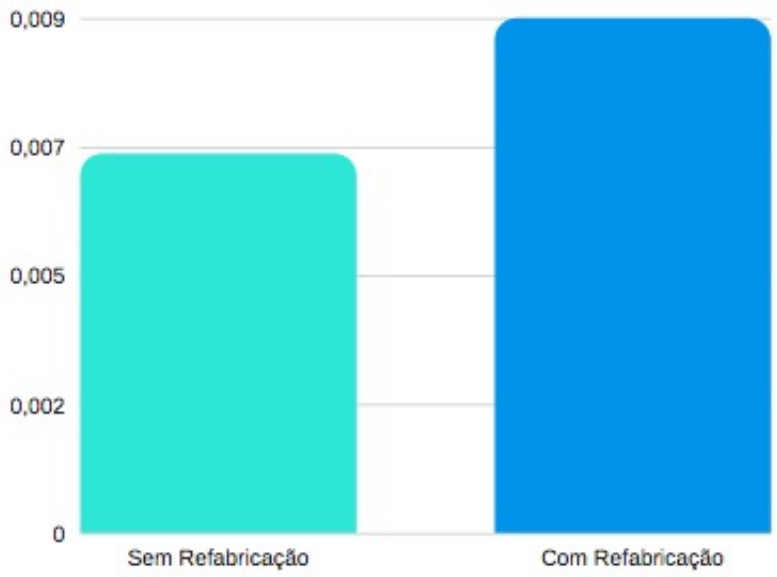

Figura 2.10: Power Consumption of Dram

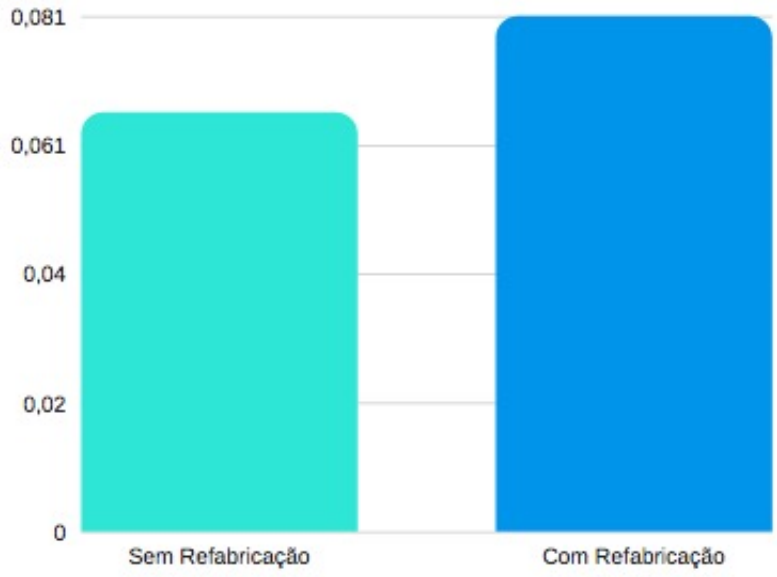

Figura 2.11: Power Consumption of CPU

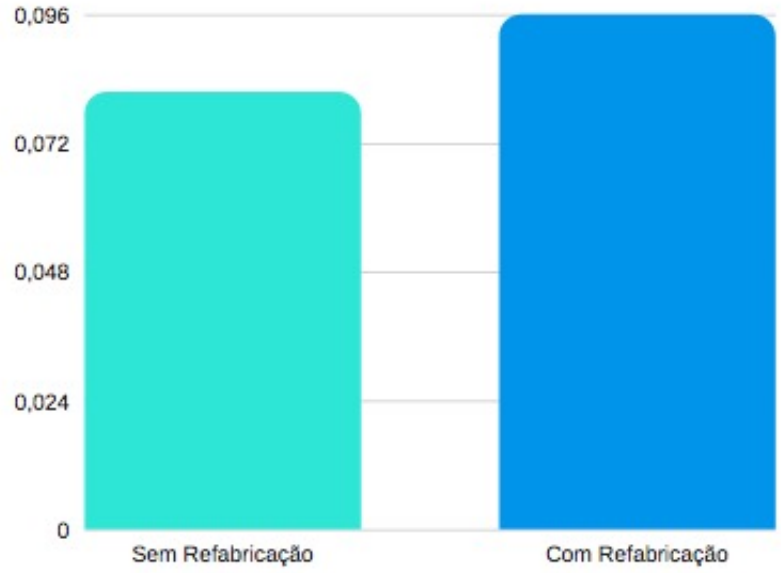

Figura 2.12: Power Consumption of Package

testarViagemEntreDatas

Figura 2.13: Power Consumption of Dram

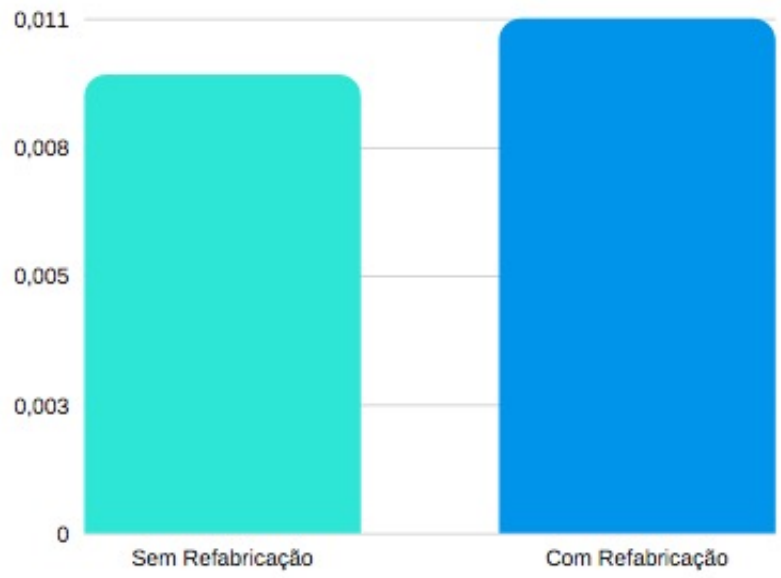

Figura 2.14: Power Consumption of CPU

Figura 2.15: Power Consumption of Package

testarRegistaViagem

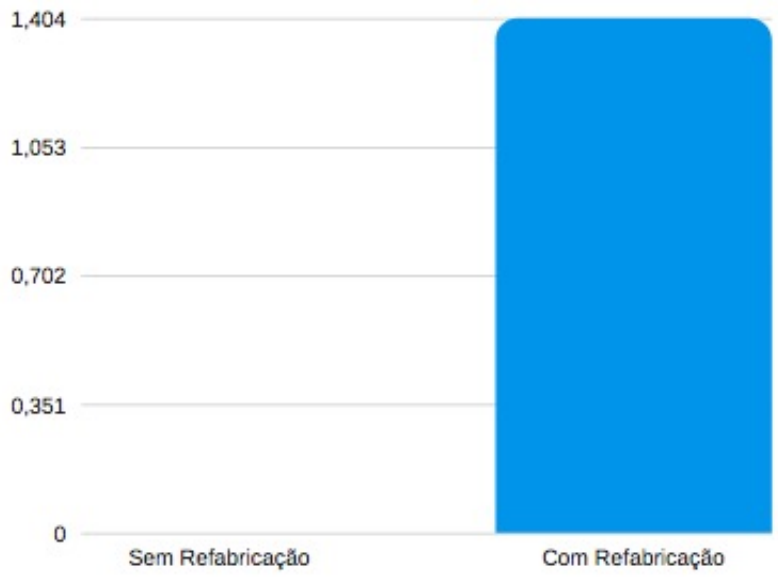

Figura 2.16: Power Consumption of Dram

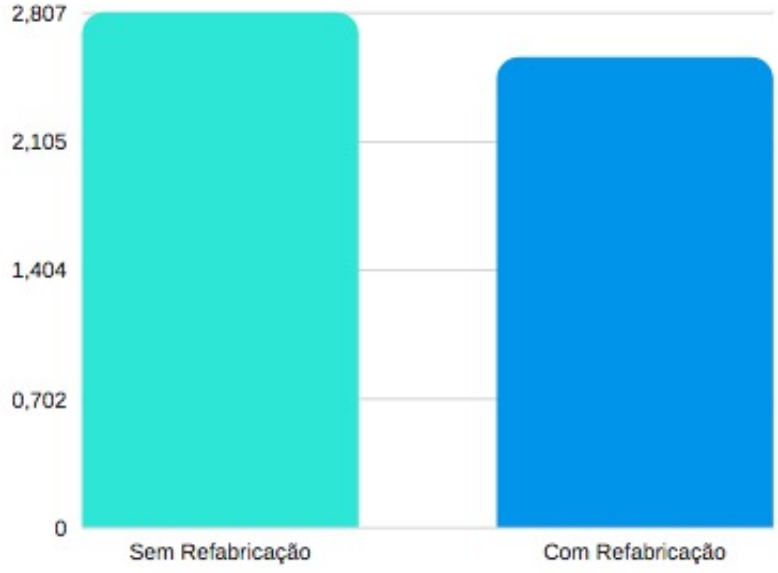

Figura 2.17: Power Consumption of CPU

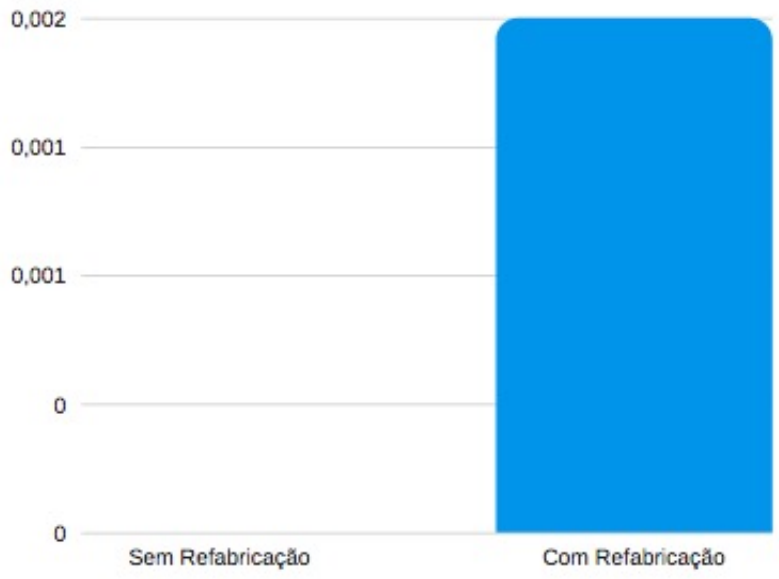

Figura 2.18: Power Consumption of Package

testarMaiorDesvio

Figura 2.19: Power Consumption of Dram

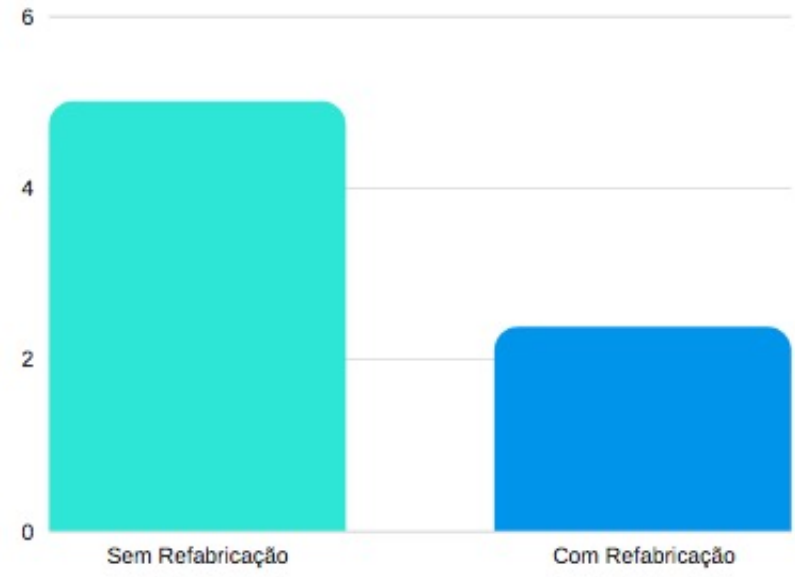

Figura 2.20: Power Consumption of CPU

Figura 2.21: Power Consumption of Package

testarEquals(coordenadaTest)

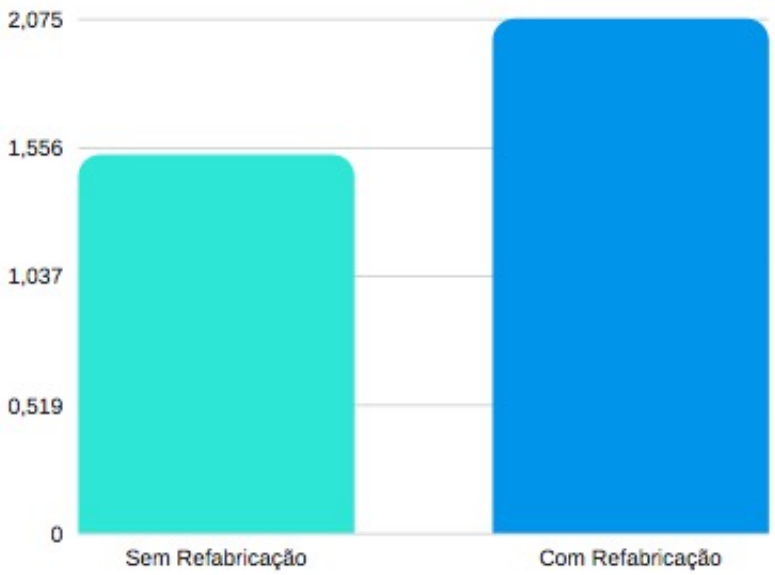

Figura 2.22: Power Consumption of Dram

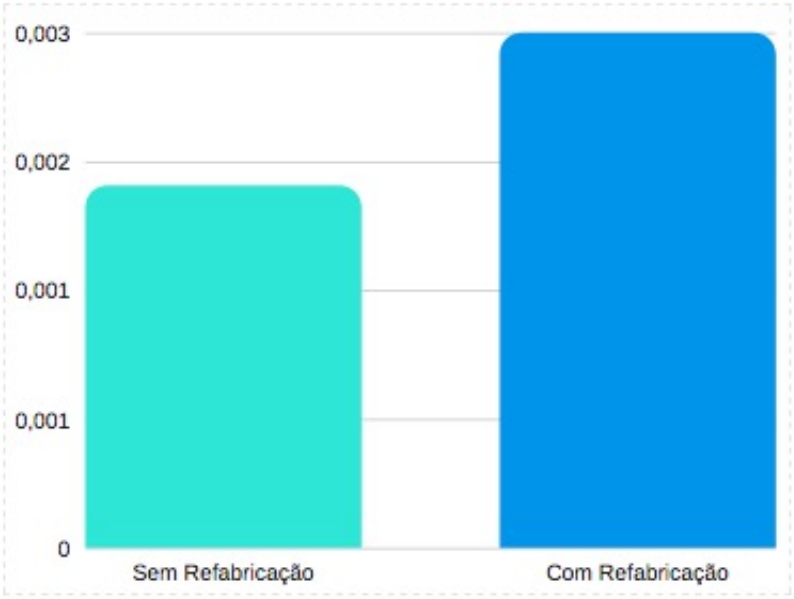

Figura 2.23: Power Consumption of CPU

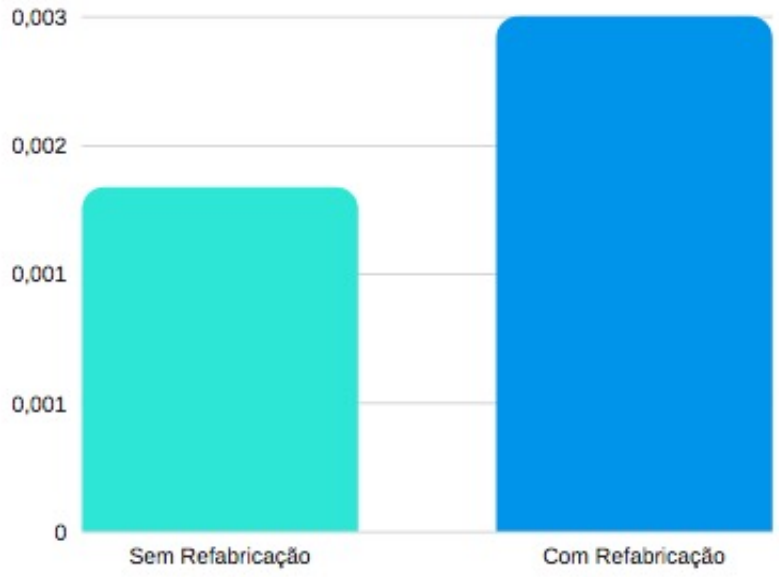

Figura 2.24: Power Consumption of Package

testarDistancia

Figura 2.25: Power Consumption of Dram

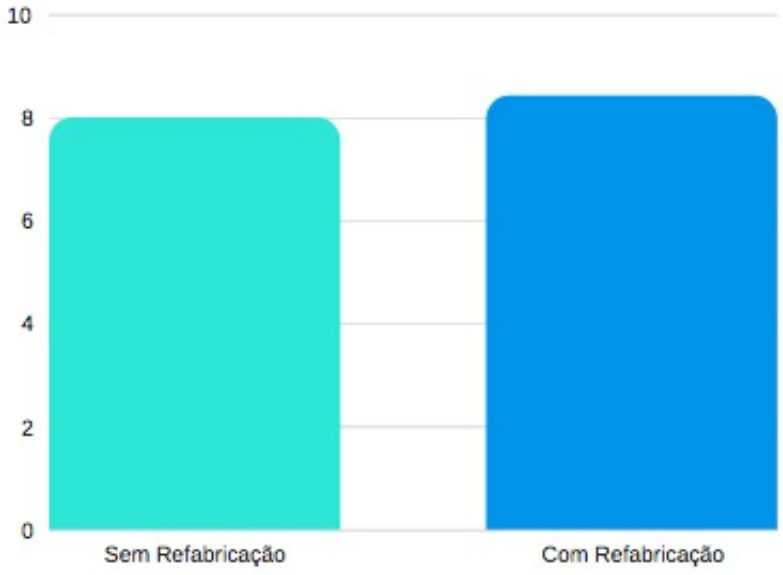

Figura 2.26: Power Consumption of CPU

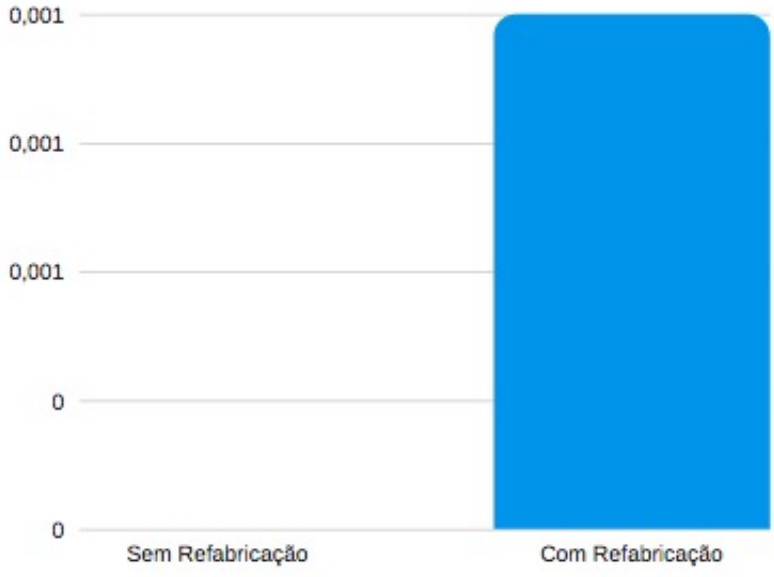

Figura 2.27: Power Consumption of Package

Capítulo 3

Análise dos Resultados e Conclusão

Como podemos ver pelos gráficos acima, a refabricação do código teve impacto no consumo energético. Como dito na secção 1.2.2 a refabricação de código tem como objetivo reestruturar o código de forma a torna-lo mais compreensível por outros programadores e aumentar a sua reutilização e manutenção. Algumas técnicas de refabricação contribuem também para tornar o código mais eficiente, nomeadamente o exemplo fornecido no excerto de código em 1.2.2, em que uma condição mais complexa é substituída por uma mais simples. No programa original era usado o método `size()` da collection e comparado com o valor 0, que pode ser otimizado usando o método de classe mais apropriado para verificar se uma coleção tem 0 elementos - `isEmpty()`.
Outro mau cheiro que tem implicações negativas na performance e consequentemente no consumo energético é o *middle man* - quando uma classe apenas redireciona os métodos para outras classes. Apesar de este caso não ter sido encontrado no projeto analisado, seria outro caso em que se esperaria que a sua refabricação tivesse um impacto positivo no consumo energético.
Assim, podemos concluir que a refabricação é um processo com muitas vantagens e que deve ser aplicada sempre que possível, principalmente quando queremos código mais reutilizável e compreensível, e também quando temos (que deverá ser sempre) preocupação em produzir Green Software.
Por tudo isto, consideramos que os objetivos do enunciado foram todos cumpridos. Na nossa opinião trabalhar com novas ferramentas como o Cobertura e o RAPL foi uma oportunidade de enorme valor, constituindo uma boa forma de aprender ferramentas muito úteis para o futuro.