

Análise de Teste e Software

Fase 3 - Projeto:UMer

José Dias
(A78494)

Pedro Silva
(PG38935)

Tiago Fraga
(A74092)

2 de Janeiro de 2019

Conteúdo

1	Introdução	3
2	Análise das Métricas do Código	4
2.1	Métricas do código fonte	4
2.2	Métricas dos testes	6
3	Maus Cheiros	7
3.1	Bad Smells	7
3.2	Red Smells	8
3.3	CPD - Cópia de código	9
4	Refactoring	10
5	Análise do tempo de execução	11
5.1	Sem Refactoring	11
5.2	Com Refactoring	12
5.3	Sem Refactoring vs Com Refactoring	12
6	Conclusão	13

1 Introdução

Na terceira e última fase do projeto, iremos efetuar uma análise à qualidade do software desenvolvido. Esta análise será efetuada em duas etapas.

Numa primeira etapa, será efetuada uma análise da qualidade do código da aplicação, começando por trabalhar com um conjunto de ferramentas que nos forneceram a capacidade de analisar quantitativamente as métricas do código fonte da aplicação, bem como dos testes desenvolvidos na segunda fase do projeto.

Numa segunda etapa, será feita uma análise de performance do software. Começaremos pela deteção de *Bad Smells* e *Red Smells*.

Após efetuarmos a deteção de *Bad Smells*, iniciamos o processo de *Refactoring* do código consoante a interpretação dos resultados da etapa anterior.

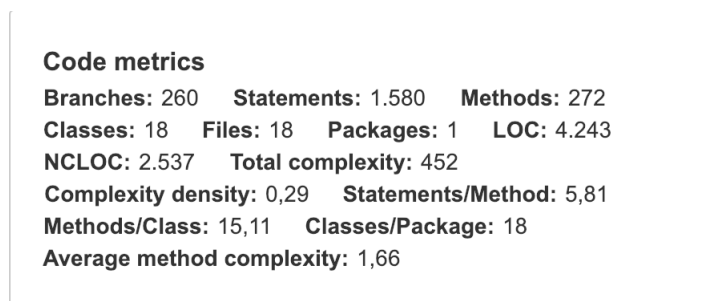
Por fim, temos como objetivo calcular o tempo de execução da aplicação, bem como a quantidade de energia e memória despendida pela mesma.

2 Análise das Métricas do Código

2.1 Métricas do código fonte

Neste capítulo, apresentamos o valor quantitativo das várias métricas do projeto em estudo.

De forma a fazer este cálculo, utilizamos a ferramenta *open-source* denominada de **clover**. Após executar os comandos que se encontram presentes na documentação da ferramenta, foi gerado um relatório em *HTML*, com o acesso a toda a informação gerada.



Code metrics		
Branches: 260	Statements: 1.580	Methods: 272
Classes: 18	Files: 18	Packages: 1
LOC: 4.243	NCLOC: 2.537	Total complexity: 452
Complexity density: 0,29	Statements/Method: 5,81	
Methods/Class: 15,11	Classes/Package: 18	
Average method complexity: 1,66		

Figura 1: Métricas do código

Através da imagem retirada do relatório é possível visualizar alguns parâmetros importantes:

- **Métodos:** 272;
- **Classes de Java:** 18;
- **Ficheiros:** 18;
- **Pacotes:** 1;
- **LOC - N° de linhas de código:** 4243;
- **NLOC - N° de linhas de código não comentadas:** 2537;
- **Atribuições:** 1580;
- **Atribuições por Método:** 5.81;
- **Métodos por Classe:** 15.11;
- **Complexidade:** 452;
- **Complexidade média por Método:** 1.66;

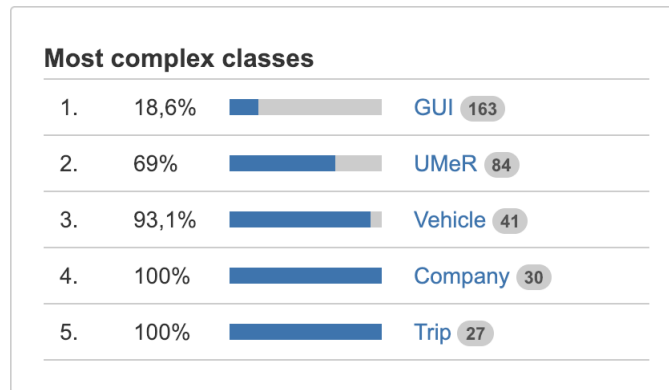


Figura 2: Métricas do código

- **Densidade da Complexidade:** 0.29;

Através desta imagem podemos visualizar as classes mais complexas do projeto.

Como era de esperar as duas primeiras são a classe **GUI** e a classe **UMeR**. Isto deve-se ao facto de o principal código para funcionamento da aplicação se situar nestas duas classes.

As outras três classes que terminam o top-5 de classes mais complexas, são as que fornecem todo o suporte necessário a nível de métodos para as duas primeiras funcionarem, daí o elevado grau de complexidade.

2.2 Métricas dos testes

Através do uso da ferramenta descrita no tópico em cima, foi possível fazer o cálculo das métricas para os testes unitários desenvolvidos na fase dois do projeto.

Lines of Code:	13,609	Conditionals:	0
NC Lines of Code:	11,779	Statements:	9,452
		Methods:	498
Total Complexity:	617	Classes:	32
Avg Complexity:	1.24	Files:	32
Complexity Density:	0.07	Packages:	1

Figura 3: Métricas dos Testes

Segundo a imagem retirada do relatório, é possível analisar os seguintes aspetos:

- **Métodos:** 272;
- **Classes de Testes:** 32;
- **Ficheiros:** 32;
- **Pacotes:** 1;
- **LOC - N° de linhas de código:** 13609;
- **NLOC - N° de linhas de código não comentadas:** 11779;
- **Atribuições:** 9452;
- **Complexidade:** 617;
- **Complexidade média por Método:** 1.24;
- **Densidade da Complexidade:** 0.07;

Em relação a estes valores, podemos afirmar que como seria de esperar foram criadas duas classes de testes (*Scaffolding* e onde estão escritos os testes) para cada classe *Java* do projeto, uma vez que não foram gerados testes para a classe *GUI* e *ATS* e que o elevado número de linhas de código também era de esperar, isto porque temos cerca de 400 testes desenvolvidos.

3 Maus Cheiros

3.1 Bad Smells

Dada por terminada a fase da avaliação das métricas do projeto, avançamos para a deteção de **Bad Smells** presentes no mesmo.

Para tal, utilizamos a ferramenta **PMD** que através da linha de comandos e da execução de um comando presente na documentação do software é possível detetar os maus cheiros presentes no código.

O comando utilizado foi o seguinte:

```
$ alias pmd="$HOME/pmd-bin-6.10.0/bin/run.sh pmd"
$ pmd -d /usr/src -R rule-sets/java/quickstart.xml -f text
```

Numa primeira etapa, utilizamos o ficheiro **quickstart.xml** que contém um conjunto de regras pré-definidas para maus cheiros, tendo obtido como resultado um ficheiro cujo excerto do seu conteúdo encontra-se em baixo:

```
ATS.java:10:
All classes and interfaces must belong to a named package
ATS.java:38:
Avoid unused local variables such as 'tree'.
Bike.java:4:
Avoid duplicate imports such as 'java.util.LinkedList'
Bike.java:4:
Avoid unused imports such as 'java.util.LinkedList'
Bike.java:5:
Avoid unused imports such as 'java.util'
```

Numa segunda etapa, e com o objetivo de detetar os maus cheiros mais relevantes, decidimos criar um ficheiro *XML* próprio, com um conjunto de regras que definimos. Após um pequeno estudo de quais as regras que detetam os maus cheiros que podíamos incluir, decidimos que as mais importantes são:

- Classe demasiado grande;
- Método demasiado grande;
- Lista de parâmetros de um método demasiado grande;
- Pelo menos um construtor por classe;
- Evitar apanhar exceções genéricas;
- Verificar o conteúdo dos comentários;
- Tamanho dos comentários;

- Classes com nomes pequenos;
- Métodos com nomes pequenos;
- Variáveis com nomes pequenos;

Após a execução deste ficheiro em formato *XML* com o comando em cima detalhado, obtivemos um conjunto de maus cheiros que na fase de **Refactoring** irão ser tidos em conta, uma vez que temos o objetivo de melhorar a execução do programa e diminuir o consumo de energia e o seu tempo de execução.

3.2 Red Smells

Outro tipo de maus cheiros que estudamos, foram os **Red Smells**, ou **Energy Smells**, que têm por base melhorar o consumo de energia do programa.

Micro-benchmark	Version	Consumption (W _s)	Energy reduction (%)
Array copying	Manual array copy	102.8	
	System array copy	63.8	37.9
Matrix iteration	By-column iteration	53,776.8	
	By-row iteration	102.6	99.8
String handling	String concatenation (+)	4,456.1	
	String builder	271.7	93.9
Use of arithmetic operations	Add constant to double	5,152.5	
	Add constant to float	5,089.3	1.2
	Add constant to long	3,643.5	29.2
	Add constant to int	838.8	83.7
Exception handling	Use Exception	14,108.6	
	No Exception	28.1	99.8
Object field access	Accessor-based access	9,190.0	
	Direct access	1,700.8	81.4
Object creation	On-demand creation	813.1	
	Object reuse	461.6	43.2
Use of primitive data types	Use of object data types	3,082.3	
	Use of primitive data types	2,356.2	23.5

Figura 4: Tipos de Red Smells

Neste subcapítulo não utilizamos nenhuma ferramenta para fazer a deteção destes maus cheiros. No entanto, iremos verificar manualmente em que local no código conseguimos detetar estes maus cheiros e alterar para uma versão em que obtemos ganhos de energia.

Nesta tarefa iremos verificar se:

- A cópia de *arrays* está a ser efetuada manualmente ou com o auxílio;

- A iteração das matrizes são efetuadas pelas linhas e não pelas colunas;
- A concatenação de *strings* é efetuada com a classe *StringBuilder* e não pelo `+`;
- Se são usadas exceções;
- Se são usados dados do tipo primitivo, tais como `int`, `boolean`, etc, e não dados do tipo objeto.

3.3 CPD - Cópia de código

Outro mau cheiro que procuramos detetar foi o da reutilização de cópias de blocos de código ao longo do projeto.

Para isso utilizamos outra ferramenta denominada por **CPD**, que é executada na linha de comando.

```
~ $ cd ~/bin/pmd-bin-6.9.0/bin
~/.../bin $ ./run.sh cpd --minimum-tokens 100 --files /home/me/src
```

Figura 5: Comando para executar o CPD.

Após a execução desta ferramenta no trabalho em questão foi possível gerar um relatório com o seguinte conteúdo:

Found a 11 line (177 tokens) duplication in the following files:

Starting at line 763 of

/Users/tiagofraga/Desktop/Complementares

/ATS/Trabalhos/Local/TP2/copia/src/java/GUI.java

Starting at line 809 of /Users/tiagofraga/Desktop/Complementares

/ATS/Trabalhos/Local/TP2/copia/src/java/GUI.java

```
Label success = successLabel("Cliente registado com sucesso!");
```

```
Button signupDone_button = doneRegisterButton();
```

```
    signupDone_button.setOnAction(e -> {
```

```
        String email      = ((TextField) email_hbox.getChildren().get(1)).getText();
```

```
        String name       = ((TextField) name_hbox.getChildren().get(1)).getText();
```

```
        String address    = ((TextField) address_hbox.getChildren().get(1)).getText();
```

```
        int    bdayD      = (int) ((ComboBox) bday_hbox.getChildren().get(0)).getValue();
```

```
        int    bdayM      = (int) ((ComboBox) bday_hbox.getChildren().get(1)).getValue();
```

```
        int    bdayY      = (int) ((ComboBox) bday_hbox.getChildren().get(2)).getValue();
```

```
        String password = ((TextField) password_hbox.getChildren().get(1)).getText();
```

Com este pequeno relatório, é possível descobrir, segundo o **CPD**, blocos de código semelhantes.

4 Refactoring

Neste capítulo iremos fazer *refactoring* ao código do programa, ou seja, iremos melhorar o design do programa para que seja de melhor compreensão e mais fácil para futuros desenvolvimentos.

Sendo assim, a primeira etapa de *refactoring* foi através do relatório gerado, pelas regras que nós definimos, dos *Bad Smells*. Como tal, verificamos os maus cheiros contidos no relatório e de seguida procedemos à correção desses maus cheiros.

A outra etapa de *refactoring* foi através dos *Red Smells*.

Através da verificação manual que nos propusemos a efetuar, verificamos que em 8 classes do projeto a concatenação de *strings*, é efetuada através do `+` e não da classe *StringBuilder*.

Verificamos ainda que, em 3 classes do projeto, a cópia de *arrays* estava a ser efetuada manualmente e não com o auxílio do *arraycopy*. Constatamos ainda, que em algumas das classes são usadas exceções.

Depois da identificação deste maus cheiros, estes foram devidamente corrigidos, como se pode ver, um exemplo, nas Figuras 5 e 6.

```
public HashMap<String, Client> getClients() {
    System.arraycopy(this.clients, 0, newClients, 0, this.clients.size());

    return newClients;
}
```

Figura 6: Exemplo de Red Smell corrigido.

```
public String toString(){
    StringBuilder sb = new StringBuilder();

    sb.append("\n");
    sb.append("Posição : "); sb.append(this.position.getX()); sb.append(","); sb.append(this.position.getY()); sb.append("\n");
    sb.append("Pontos : "); sb.append(this.points); sb.append("\n");
    sb.append("Premium : "); sb.append(this.premium); sb.append("\n");
    sb.append("Fila de espera : "); sb.append(this.queue);

    return sb.toString();
}
```

Figura 7: Exemplo de Red Smell corrigido.

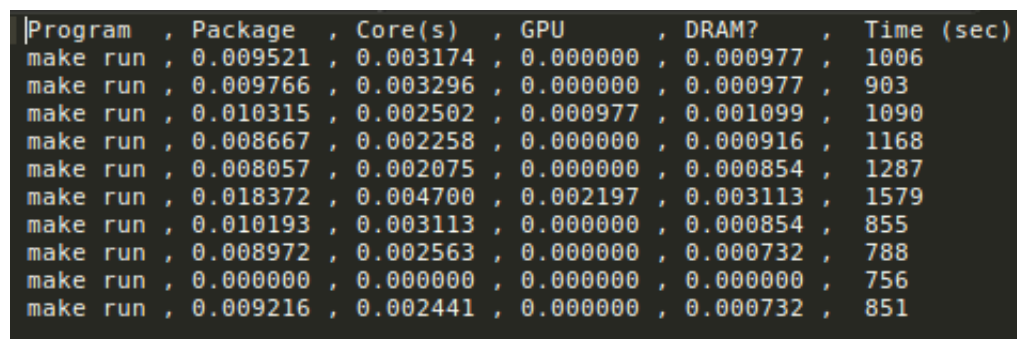
5 Análise do tempo de execução

Neste capítulo iremos apresentar os resultados de usar a ferramenta *RAPL*, para a detecção do tempo de execução dos programas, assim como o consumo de energia e de memória.

Os resultados obtidos foram separados em duas partes. A primeira parte com os resultados do uso da ferramenta sem *Refactoring*. A segunda parte com os resultados do uso da ferramenta com *Refactoring* no código.

5.1 Sem Refactoring

Foram realizadas 10 execuções do programa, com a ferramenta *RAPL*, sem *Refactoring*. O resultado pode ser visto na figura seguinte.



Program	Package	Core(s)	GPU	DRAM?	Time (sec)
make run	0.009521	0.003174	0.000000	0.000977	1006
make run	0.009766	0.003296	0.000000	0.000977	903
make run	0.010315	0.002502	0.000977	0.001099	1090
make run	0.008667	0.002258	0.000000	0.000916	1168
make run	0.008057	0.002075	0.000000	0.000854	1287
make run	0.018372	0.004700	0.002197	0.003113	1579
make run	0.010193	0.003113	0.000000	0.000854	855
make run	0.008972	0.002563	0.000000	0.000732	788
make run	0.000000	0.000000	0.000000	0.000000	756
make run	0.009216	0.002441	0.000000	0.000732	851

Figura 8: RAPL sem Refactoring

Deste resultado podemos retirar os seguintes dados:

- **Tempo de execução(média):** 1.0283 ms
- **Consumo de energia(média):** 0.0026122
- **Consumo de memória(média):** 0.0010254

5.2 Com Refactoring

Foram realizadas 10 execuções do programa, com a ferramenta *RAPL*, com *Refactoring*. O resultado pode ser visto na figura seguinte.

Program	Package	Core(s)	GPU	DRAM?	Time (sec)
make run	0.009277	0.002808	0.000000	0.000916	878
make run	0.009888	0.003113	0.000000	0.001038	941
make run	0.010132	0.002625	0.001770	0.001648	1162
make run	0.010925	0.002502	0.000977	0.001282	1098
make run	0.007080	0.001465	0.000000	0.000732	951
make run	0.007019	0.001526	0.000000	0.000610	913
make run	0.007019	0.001587	0.000000	0.000671	861
make run	0.007812	0.001831	0.000000	0.000671	968
make run	0.007446	0.001526	0.000000	0.000610	933
make run	0.007629	0.001709	0.000000	0.000732	1025

Figura 9: RAPL com Refactoring

Deste resultado podemos retirar os seguintes dados:

- Tempo de execução(média): 0.973 ms
- Consumo de energia(média): 0.0020692
- Consumo de memória(média): 0.000891

5.3 Sem Refactoring vs Com Refactoring

Tendo os resultados do uso do *RAPL* com e sem *Refactoring*, podemos agora comparar os resultados obtidos para ver se houve ou não melhorias, em termos do tempo de execução, consumo de energia e consumo de memória, e verificar se estas melhorias são ou não significativas.

Como tal, verificamos que, com o uso de *Refactoring* no código, houve uma melhoria de cerca de 5% no tempo de execução, assim como, uma melhoria de cerca de 13% no consumo de memória e uma melhoria de cerca de 20% no consumo de energia.

Contudo, podemos concluir que fazer *Refactoring* no código fonte do projeto tem uma ligeira vantagem, em termos do tempo de execução do programa. Podemos também afirmar, que houve um ganho mais significativo no consumo de energia e memória quando é feito *Refactoring*.

6 Conclusão

Ao longo deste trabalho, fomos estudando a complexidade do projeto, assim como diferentes maneiras de encontrar "maus cheiros" e melhorar a compreensão do código, e por fim verificar os tempos de execução e consumos de energia e memória do projeto.

Numa primeira etapa, verificamos as métricas do projeto através da ferramenta **Clover**. Foram verificadas as métricas do código fonte, assim como a complexidade de cada classe, e por fim as métricas dos testes desenvolvidos na segunda fase do projeto.

Na segunda etapa do trabalho, efetuamos a verificação dos maus cheiros no código do programa. Estes maus cheiros foram divididos em três partes, os *Bad Smells*, os *Red Smells* e *CPD*. Na primeira parte foi utilizada a ferramenta **PMD** e a criação de um ficheiro *XML* com as regras mais importantes para este efeito. Numa segunda parte foram detetados os *Red Smells* manualmente. Na terceira parte, foi usada a ferramenta **CPD**, para a deteção de cópia de código no projeto.

Numa terceira fase, foi efetuado o *Refactoring* ao código fonte do projeto. Este *Refactoring* foi realizado através dos **Bad Smells** e **Red Smells** obtidos nas fases anteriores, assim como do material disponibilizado da disciplina sobre *Program Refactoring*.

Numa última fase, foram obtidos os resultados dos tempos de execução e consumos de energia e memória do projeto, com recurso da ferramenta **RAPL**. Podemos verificar que fazer *Refactoring* ao código fonte é mais vantajoso do que não fazer *Refactoring*, em termos do tempo de execução e consumos de energia e memória do programa.

Apesar do trabalho desenvolvido, achamos que futuramente o código fornecido para análise pode ser ainda mais escrutinado, em termos de ser realizado mais *Refactoring*, no sentido de melhorar ainda mais a compreensão do código e também do seu tempo de execução e consumos de energia e memória.

Concluindo, achamos que, ao longo do estudo deste projeto UMeR, conseguimos obter conhecimentos que nos permitem testar por completo um programa, ou uma aplicação, sendo eles a análise e teste da Aplicação UMer, a criação de testes para a correta verificação do software UMeR e a melhoria da qualidade do software UMeR para futuros desenvolvimentos.