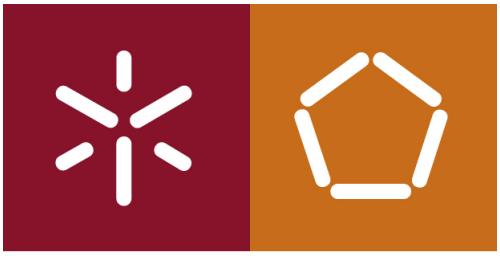


Universidade do Minho

Mestrado Integrado de Engenharia Informática

Análise e Teste de Software

Trabalho Prático 1

Diogo Soares	A74478
Ricardo Leal	A75278
Samuel Martins	PG37163

30 de Dezembro de 2018

Conteúdo

1 Introdução **2**

2 Métricas de Código Fonte **3**
2.1 SLOCCount 3
2.2 OpenClover 3

3 Métricas dos Testes **4**

4 Refactoring e Análise de Execução **5**
4.1 Bad Smells 5
4.2 Refactoring 6
4.2.1 Algoritmo Usado 6
4.2.2 Resultados Obtidos 6

5 Conclusão **7**

1 Introdução

Neste terceiro projeto da unidade curricular de Análise e Teste de Software é pretendido que analisemos a qualidade do software UMeR desenvolvido pelos alunos de segundo ano. Esta análise é baseada em duas componentes, análise da qualidade do código da aplicação e análise da *performance* do software.

2 Métricas de Código Fonte

Nesta secção iremos analisar o código da UMeR e tentar perceber a sua qualidade através do uso de ferramentas apresentadas nas aulas da unidade curricular. No uso destas ferramentas foram excluídas as classes referentes ao parser, visto não fazer parte do projeto inicial.

2.1 SLOCCount

Esta ferramenta conta as linhas de código útil (ignora linhas em branco e comentários) e dá estimativas de tempo necessário para escrever esse mesmo software. Como podemos ver na imagem que se segue, o código da UMeR juntamente com o código do parser contem 2558 linhas de código. Para desenvolver este projeto, um programador precisaria de cerca de cinco meses. Este projeto teria um custo estimado de 72,433 $.

```
SLOC    Directory       SLOC-by-Language (Sorted)
2558    top_dir         java=2558


Totals grouped by language (dominant language first):
java:          2558 (100.00%)




Total Physical Source Lines of Code (SLOC)                = 2,558
Development Effort Estimate, Person-Years (Person-Months) = 0.54 (6.43)
 (Basic COCOMO model, Person-Months = 2.4 * (KSLOC**1.05))
Schedule Estimate, Years (Months)                         = 0.42 (5.07)
 (Basic COCOMO model, Months = 2.5 * (person-months**0.38))
Estimated Average Number of Developers (Effort/Schedule)  = 1.27
Total Estimated Cost to Develop                           = $ 72,433
 (average salary = $56,286/year, overhead = 2.40).
SLOCCount, Copyright (C) 2001-2004 David A. Wheeler
SLOCCount is Open Source Software/Free Software, licensed under the GNU GPL.
SLOCCount comes with ABSOLUTELY NO WARRANTY, and you are welcome to
redistribute it under certain conditions as specified by the GNU GPL license;
see the documentation for details.
Please credit this data as "generated using David A. Wheeler's 'SLOCCount'."
```

Figura 1: Resultado do SLOCCount

Estas estimativas não são muito confiáveis pois baseiam se puramente no número de linhas de código do software.

2.2 OpenClover

Na análise com o OpenClover além de excluirmos as classes acima referidas também excluímos a classe "GUI.java"(classe de frontend) sobre a qual não realizamos testes na tarefa anterior. Conseguimos identificar a diferença de quase 1500

linhas de código entre a simples contagem de linhas e a contagem de linhas de código realmente (ignorando comentários). Se a este número de linhas NC (1381) o número de linhas NC da classe "GUI.java"(1138, resultado obtido com o SLOC-Count) obtemos um número e linhas bastante próximo do resultado da ferramenta anteriormente utilizada.

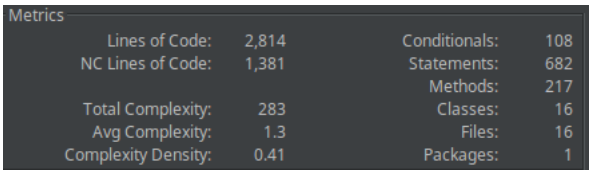


Figura 2: Métricas com o OpenClover

Além do simples número de linhas, o OpenClover permite-nos tirar mais informação global sobre o projeto em si, como o número de classes, métodos e até o número de expressões condicionais usadas.

3 Métricas dos Testes

Nesta secção do relatório iremos mostrar a percentagem de código coberta pelos testes por nós realizados na tarefa anterior. Para isso, foi necessário criarmos novas classes de testes com exatamente os mesmos testes já realizados. Isto deveu-se a uma incompatibilidade do OpenClover com os testes gerados automaticamente pelo EvoSuite e como os nossos testes tinham sido criados nessas classes geradas, esta foi a melhor forma encontrada para analisar com o OpenClover. Esta análise foi realizada num novo projeto de modo a não alterar os testes da 2ª fase do trabalho.

Posto isto, podemos verificar que os nossos testes cobrem 81.8% da aplicação (GUI.java excluída) sendo que apenas 67.6% das expressões condicionais foram testadas.

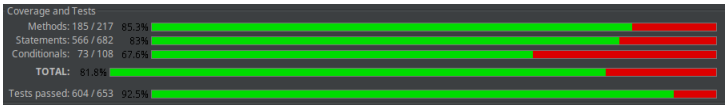


Figura 3: Percentagem de código coberto

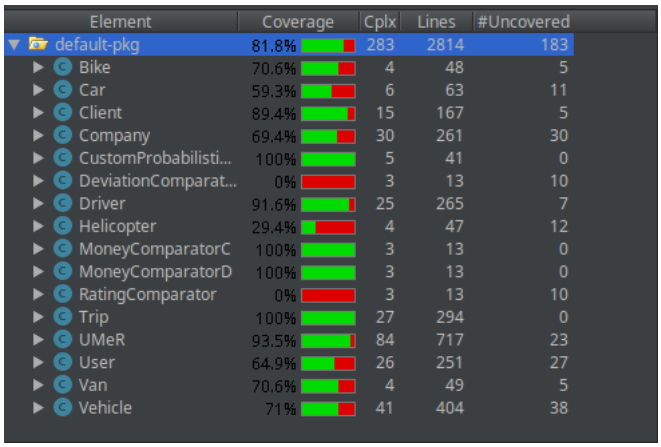
Além disso, conseguimos constatar que para as classes *DeviationComparator* e a classe *RatingComparator* não foram realizados quaisquer testes. Olhando para essas classes novamente facilmente encontramos um erro que não detetamos na fase anterior.

```
if (d2.getDeviation() < d2.getDeviation()) return 1;
```

Outro erro facilmente encontrado com o OpenClover foi na classe *car.java* uma expressão condicional onde o "corpo"não chega a ser executado devido a um mau posicionamento de um ';'.

```
if (v.getPosition().distance(this.getPosition()) <= minDist);
    traffic++;
```

Ambos os erros agora encontrados têm em comum o facto de serem em expressões condicionais que é exatamente a parte de código que menos conseguimos cobrir.



Element	Coverage	Cplx	Lines	#Uncovered
default-pkg	81.8%	283	2814	183
Bike	70.6%	4	48	5
Car	59.3%	6	63	11
Client	89.4%	15	167	5
Company	69.4%	30	261	30
CustomProbabilisti...	100%	5	41	0
DeviationComparat...	0%	3	13	10
Driver	91.6%	25	265	7
Helicopter	29.4%	4	47	12
MoneyComparatorC	100%	3	13	0
MoneyComparatorD	100%	3	13	0
RatingComparator	0%	3	13	10
Trip	100%	27	294	0
UMeR	93.5%	84	717	23
User	64.9%	26	251	27
Van	70.6%	4	49	5
Vehicle	71%	41	404	38

Figura 4: Percentagem de código coberto por classes

4 Refactoring e Análise de Execução

Nesta secção iremos analisar o código e a sua eficiência. Iremos dar alguns exemplos de *bad smells* encontrados assim como comparar os resultados de *performance* com e sem *refactoring*.

4.1 Bad Smells

Iremos agora enumerar alguns dos maus cheiros encontrados no código da UMeR:

- **Parâmetros inutilizados:** em alguns construtores detetamos parâmetros que não são utilizados na criação do objecto em causa;
- **Parênteses desnecessários:** ao longo do software encontramos vários parênteses desnecessários, prejudicando a compreensão do código;

- **Variáveis locais não utilizadas:** alguns métodos contêm variáveis locais desnecessárias;
- **Declarações condicionais que podem ser agrupados:** alguns "if" presentes em métodos podiam e deviam estar agrupados numa única operação.

4.2 Refactoring

Para podermos comparar os resultados todos os testes feitos usando o *Rapl* foram na mesma máquina e nas mesmas condições, sem aplicações a correr em *background* e com o mesmo ficheiro de *input* de maneira a não influenciar o resultado.

4.2.1 Algoritmo Usado

Fizemos seis testes de modo a perceber o que melhora e/ou prejudica o consumo da aplicação. Cada teste foi realizado dez vezes, usando depois a média dos resultados dessas execuções. Optamos por esta abordagem de modo a conseguirmos obter um consumo de energia mais viável.

- **Aplicação original:** corremos o *Rapl* com o código original da UMeR;
- **Sem maus cheiros:** corremos a aplicação depois de eliminados os maus cheiros detetados;
- **Simplificada:** simplificamos partes do código com ajuda do *Intelij*. Ex:

  ```
  (d -> d.isAvailable() == true)
  ```

  passou a

  ```
  (d -> d.isAvailable())
  ```

- **For:** utilizando apenas *for*;
- **Foreach:** utilizando apenas *foreach*;
- **Streams:** utilizando apenas *streams* (*collectors* especialmente).

4.2.2 Resultados Obtidos

Como podemos ver na tabela apresentada, a eliminação dos maus cheiros diminuiu consideravelmente o consumo de energia do CPU e do Package. Além disso, o método mais eficiente será usar *for* em detrimento das *streams*.

	DRAM	CPU	PACKAGE
Original	0.30	2.24	3.06
S/ Maus Cheiros	0.29	2.13	2.87
Simplificada	0.29	2.11	2.87
For	0.28	2.03	2.76
ForEach	0.28	2.03	2.77
Stream	0.28	2.07	2.81

5 Conclusão

Esta tarefa permitiu nos perceber a importância de escrever um bom código e de ganhar boas práticas. Permitiu nos ganhar consciência do impacto que os maus cheiros têm no produto final, por exemplo.

Todo este projeto foi útil e irá influenciar-nos na maneira como programamos, pois ganhamos consciência do que são boas e más práticas de programação. Além disso, permitiu nos aprender como podemos testar o nosso código duma maneira eficiente e lembrar nos do cuidado que devemos ter ao escrevê-lo, pois outras pessoas o poderão ler.

Em suma, acreditamos que este projeto foi bastante enriquecedor, tanto como programadores como *testers* de código.