



UNIVERSIDADE DO MINHO

MESTRADO INTEGRADO DE ENGENHARIA INFORMÁTICA

Análise e Teste de Software

Trabalho Prático 1

Diogo Soares
Ricardo Leal
Samuel Martins

A74478
A75278
PG37163

19 de Novembro de 2018

Conteúdo

1	Introdução	2
2	Testes Unitários	3
2.1	Gerador de Testes	3
2.2	Mutação de Testes	3
2.3	Cobertura de Testes	3
2.4	Erros Encontrados	4
3	Testes de Sistema	5
3.1	Gerador do Ficheiro	5
3.2	Resultado	5
4	Conclusão	5

1 Introdução

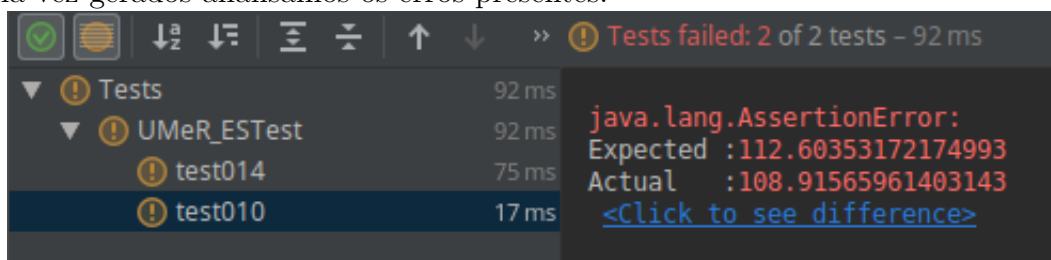
Neste segundo projeto da unidade curricular de Análise e Teste de Software é pretendido que demos continuação aos testes do software da primeira fase. Assim, pretende-se que se teste o mais possível esta aplicação. Para isso é necessário recorrer a testes unitários e a testes de sistema.

2 Testes Unitários

Os testes unitários servem para testar todo o tipo de funções presentes na aplicação. Isto é, com um teste unitário podemos testar se determinada função faz o esperado, ou seja, se o resultado é o que devia ser. O *modus operandi* destes testes é simplesmente comparar o valor obtido com o valor esperado.

2.1 Gerador de Testes

Utilizamos o EvoSuite para gerar testes unitários para todo o código da aplicação. Uma vez gerados analisamos os erros presentes:



Estes dois erros que os testes gerados automaticamente detetaram, não são na verdade erros de relevo, pois estes erros devem se a valores que dependem de valores aleatórios.

2.2 Mutação de Testes

Para analisarmos a qualidade dos testes geramos usamos técnicas de mutação de código, isto é, ir ao código a testar e mudar funções deliberadamente para esta falhar. Um exemplo disso, foi na função *addTrip* trocar as coordenadas da posição final.

```
public void addTrip(Trip t){  
    this.trips.add(t);  
    this.position.setLocation(t.getEnd().getY(), t.getEnd().getX());  
}
```

Se neste caso, os testes falharam, houve outros casos em que tal não aconteceu, construindo nós novos testes para testar essas propriedades.

2.3 Cobertura de Testes

Na imagem seguinte mostramos a cobertura dos testes totais presente no nosso programa.

Element	Class, %	Method, %	Line, %
ATSParser	0% (0/13)	0% (0/99)	0% (0/454)
Bike	100% (1/1)	100% (4/4)	75% (12/1...
Car	100% (1/1)	100% (5/5)	68% (15/2...
Client	100% (1/1)	100% (15/...	90% (38/4...
Company	100% (1/1)	75% (18/2...	71% (63/8...
CustomProbabilisticDis...	100% (1/1)	100% (3/3)	100% (13/...
DeviationComparator	0% (0/1)	0% (0/1)	0% (0/3)
Driver	100% (1/1)	100% (24/...	91% (64/7...
GUI	0% (0/1)	0% (0/72)	0% (0/924)
Helicopter	100% (1/1)	100% (4/4)	75% (12/1...
MoneyComparatorC	100% (1/1)	100% (1/1)	100% (4/4)
MoneyComparatorD	100% (1/1)	100% (1/1)	100% (4/4)
RatingComparator	0% (0/1)	0% (0/1)	0% (0/3)
Trip	100% (1/1)	100% (26/...	100% (75/...
UMeR	100% (1/1)	100% (51/...	95% (268/...
User	100% (1/1)	95% (22/2...	92% (58/6...
Van	100% (1/1)	100% (4/4)	75% (12/1...
Vehicle	100% (1/1)	100% (31/...	96% (109/...

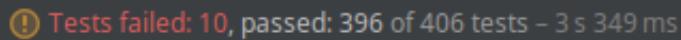
2.4 Erros Encontrados

Iremos agora enumerar os erros por nós encontrados na aplicação.

- **Mudança de disponibilidade de condutor:** Condutor não fica indisponível quando solicitado. Este erro causa erros noutras funções.
- **Number of Reviews:** Quando uma viagem é classificada e consequentemente o condutor, o número de reviews não aumenta.
- **Add Rating de viagens:** Uma viagem é classificada de 0 a 5, depois é convertida para valores de 0 a 100. Este último passo acontece no cliente mas não no rating da viagem.
- **closestAvailableTaxi:** Esta função devolve o táxi mais próximo, mesmo este estando indisponível.
- **Viagens não registadas:** quando se adiciona uma viagem, esta não fica adicionada em todos os intervenientes

- **Clones:** Quando se regista um objecto, o que é adicionado à aplicação deste é o seu clone, fazendo com que a informação presente no próprio objecto não sejam atualizadas.

O total de testes passados foram:



! Tests failed: 10, passed: 396 of 406 tests – 3 s 349 ms

3 Testes de Sistema

O objetivo para esta parte do trabalho, era gerar um ficheiro log como nos foi passado na primeira fase. Para ler um ficheiro diferentes, tivemos de fazer algumas alterações no parser da primeira fase, de modo a este poder ler o novo ficheiro gerado. Neste novo parser foram adicionadas as funções de :

- Registar Veículos;
- Solicitar viagens a condutores/empresas específicos;
- Alterar a disponibilidade do condutor.

3.1 Gerador do Ficheiro

Para gerarmos um ficheiro log, recorremos aos geradores do quickcheck do haskell, tendo conseguido gerar vários tipos de funções acessíveis aos diversos utilizadores. O ficheiro log, conta agora com testes às funcionalidades atrás enumeradas.

3.2 Resultado

Embora com um ficheiro de pequena escala o programa execute, o mesmo não acontece quando este tem várias entradas. Acreditamos que o programa não suporte um grande número de solicitações de viagem por não haver táxis suficientes para tal. Tentamos corrigir isso, por exemplo, aumentando a frequência de registos de condutores (e consequentemente de táxis), mas tal ações não surtiram efeito. Acreditamos por isso que o programa não está preparado para grandes quantidades de dados/ações, sendo também possível uma má otimização no nosso *parser*.

4 Conclusão

Este trabalho permitiu nos estar no papel de *testers* e perceber a dificuldade que é testar o código de outras pessoas. O facto de a aplicação estar bem documentada

ajuda nos a perceber o que é pretendido de cada função e testar assim as suas propriedades. Por outro lado, permitiu nos familiarizar melhor com a ferramenta *quickcheck*, útil para gerar testes em grande escala da aplicação. As nossas maiores dificuldades foram perceber onde devíamos focar nos testes unitários e relembrar os monads. Em suma, achamos que foi um trabalho bastante interessante tanto a nível de ”*testers*” como de aprendizagem em si, apesar de algo trabalhoso.