Análise e Teste de Software
Trabalho Prático

-

Mestrado em Engenharia Informática
Universidade do Minho
Relatório

Grupo nº3

PG41091	Nelson José Dias Teixeira
PG41081	José Alberto Martins Boticas
PG41094	Pedro Rafael Paiva Moura
A80499	Moisés Manuel Borba Roriz Ramires

6 de Dezembro de 2019

Resumo

No ano lectivo 2018/2019, no contexto da disciplina de Programação Orientada a Objectos (POO) leccionada no Departamento de Informática da Universidade do Minho, os alunos tiveram de desenvolver em grupo uma aplicação Java, denominada por *UmCarroJá*, para gerir um serviço de aluguer de veículos particulares pela internet. No contexto da disciplina de Análise e Teste de Software (ATS) pretende-se que neste projeto se apliquem técnicas de análise e teste de software, estudadas nas aulas, de modo a analisar a qualidade de duas das soluções desenvolvidas pelos alunos de POO.

Conteúdo

Capítulo 1

Introdução

Neste projeto foi-nos proposto a realização de várias tarefas de forma a analisar a qualidade das duas soluções desenvolvidas pelos alunos de POO no ano lectivo de 2018/2019. Entre estas tarefas destacam-se as seguintes:

1. Analisar a qualidade do código fonte dos sistemas de *software*. Nesta análise identificam-se *bad smells* no código fonte e o seu *technical debt*;
2. Aplicar *refactorings* de modo a eliminar os *bad smells* encontrados e deste modo reduzir (se possível eliminar) o *technical debt*;
3. Testar o *software* de modo a ter mais garantias que ele cumpre os requisitos do enunciado da aplicação *UmCarroJá*;
4. Gerar *inputs* aleatórios para a aplicação *UmCarroJá* que simulem execuções reais (tal como foi fornecido em POO);
5. Analisar a performance (tempo de execução e consumo de energia) das versões iniciais do *software* (i.e., com *smells*) e as obtidas depois de eliminados os *smells*.

Os cinco pontos mencionados acima foram agrupados em quatro tarefas finais, cada uma das quais com uma percentagem na avaliação final do trabalho prático. As abordagens tomadas pelo grupo sobre cada uma destas tarefas serão expostas nos capítulos seguintes deste relatório. De salientar que também existem tarefas extras que complementam cada uma das tarefas referidas anteriormente.

Capítulo 2

Análise e Especificação

2.1 Tarefa 1 - Qualidade do código fonte

Nesta etapa, tal como o nome indica, será feita a análise da qualidade do código fonte da aplicação *UmCarroJá* desenvolvido pelos alunos de POO. Como tal, através da ferramenta *Sonarqube*, serão indicados o número de erros no código (*bugs*), vulnerabilidades, *code smells* e o respetivo *technical debt*. Para além destas, existe uma **tarefa extra** que consiste em definir regras adicionais na ferramenta *Sonarqube* para encontrar *red smells* (ou qualquer outro *smell* não suportado pelo mesmo) na aplicação desenvolvida.

2.1.1 Versão 1 - *demo1*

Na primeira versão desenvolvida pelos alunos de POO, *demo1*, foi possível observar alguns erros no código fonte e bastantes *code smells*. Apresenta-se de seguida, por categorias, a análise qualitativa desta mesma implementação.

2.1.1.1 Fiabilidade

Nesta secção da análise qualitativa da aplicação desenvolvida observam-se e identificam-se unicamente os erros (*bugs*) presentes no código fonte. Como tal, após verificar a informação existente na ferramenta *Sonarqube*, os elementos deste grupo depararam-se, essencialmente, com quatro tipos de erros. Entre eles destacam-se:

1. a implementação do método *equals()* numa determinada classe sobrepõe a predefinida, pelo que também deve ser codificado o método *hashCode()*;
2. o método *equals()* presente numa determinada classe necessita de ser sobreposto à implementação predefinida ou, simplesmente, renomeado;
3. o objeto *Random* presente numa determinada classe deve ser reutilizado;
4. o objeto *ObjectOutputStream* deve ser fechado através de uma clausula *try-catch-finally*.

Na totalidade existem cerca de **14 erros** no código fonte. Quanto à severidade destes erros, existem 2 de tipo *blocker*, 2 de tipo *critical*, 1 de tipo *major* e 9 de tipo *minor*.

De salientar que, apesar da existência de alguns erros presentes nesta implementação, estes são facilmente corrigíveis.

2.1.1.2 Segurança

Ao nível da segurança, esta implementação apresenta apenas **uma vulnerabilidade** cujo grau de severidade é do tipo *minor*. A ferramenta *Sonarqube*, por forma a eliminar esta mesma, sugere encapsular a amostragem de um determinado erro através de um objeto *LOGGER*. Dado que existe uma e uma só vulnerabilidade deste tipo, o *software Sonarqube* atribui nota *B* a este contexto. De notar também que existem 22 casos no código fonte que precisam de ser revistos por forma a verificar se existem ou não ainda mais vulnerabilidades.

2.1.1.3 Manutenção

Neste segmento do relatório, observam-se e identificam-se os *code smells*. Estes inferem o grau de interpretabilidade do código fonte e, por isso, permitem avaliar se é ou não possível evoluir a versão atual do programa desenvolvido.

Neste caso, foi possível verificar a existência de **131 *code smells***. Apresentam-se de seguida o grau de severidade de cada um destes:

- ***minor***: 60 ocorrências;
- ***major***: 24 ocorrências;
- ***critical***: 41 ocorrências;
- ***blocker***: 6 ocorrências.

Como se pode constatar, cerca de 54% dos *code smells* possuem uma gravidade considerável, pelo que se pode inferir que esta implementação levará, potencialmente, a uma interpretabilidade razoável por parte do programador.

Segundo a ferramenta *SonarQube*, demoraria cerca de 2 dias e 5 horas a corrigir todos estes "defeitos". Este facto traduz aquilo a que chamamos o *technical debt*, isto é, a probabilidade de ocorrências de erros no futuro. Esta ferramenta atribui a nota *A* no que diz respeito ao *technical debt*.

2.1.1.4 Cobertura

Quanto à cobertura não foram testados nenhuns aspetos intrínsecos a esta implementação.

2.1.1.5 Duplicação de código

No que diz respeito à duplicação de código existem cerca de 2 blocos repetidos numa das classes implementadas, representando apenas 1% do código total.

2.1.2 Versão 2 - *demo2*

Na segunda versão desenvolvida pelos alunos de POO, *demo2*, foi possível observar bastantes erros no código fonte e muitos *code smells*. Apresenta-se de seguida, por categorias, a análise qualitativa desta mesma implementação.

2.1.2.1 Fiabilidade

Nesta secção da análise qualitativa da aplicação desenvolvida observam-se e identificam-se unicamente os erros (bugs) presentes no código fonte.

Após consultar a informação presente na ferramenta *Sonarqube*, foi possível verificar que, no total, existem **33 *bugs*** nesta implementação, sendo que 27 são do tipo *minor*, 2 são do tipo *major* e 4 são do tipo *blocker*.

Todos estes erros estão contidos em 6 categorias. Apresenta-se de seguida não só estas últimas como também o respetivo número de erros associados à mesma:

1. a implementação do método *equals()* numa determinada classe sobrepõe a predefinida, pelo que também deve ser codificado o método *hashCode()* (10 ocorrências);
2. *boxing* e *unboxing* de objetos não devem ser imediatamente revertidos (10 ocorrências);
3. necessidade de realizar *cast* a um dos operandos na operação de divisão (7 ocorrências);
4. objetos do tipo *ObjectInputStream*, *ObjectOutputStream*, *FileInputStream* e *BufferedReader* devem ser fechados através de uma clausula *try-catch-finally* (4 ocorrências);
5. objetos do tipo *Calendar* não devem conter a referência *static* e, como tal, devem ser instanciados (1 ocorrência);
6. a exceção *NullPointerException* pode ser lançada e, como tal, deve ser utilizada a cláusula *try-catch* para prevenir a referência a um apontador nulo (1 ocorrência).

De salientar que, apesar da existência de vários erros nesta implementação, estes são corrigíveis.

2.1.2.2 Segurança

Ao nível da segurança, esta implementação apresenta **10 vulnerabilidades** cujo grau de severidade é do tipo *minor*. Consequentemente, a ferramenta *Sonarqube* atribui nota *B* a este contexto, dado que existe pelo menos uma vulnerabilidade do tipo *minor*. Todas estas vulnerabilidades baseiam-se, de forma global, na transformação de variáveis em constantes. De notar também que existem 16 casos no código fonte que precisam de ser revistos por forma a verificar se existem ou não ainda mais vulnerabilidades.

2.1.2.3 Manutenção

Neste parte do presente documento, observam-se e identificam-se os *code smells*. Estes inferem o grau de interpretabilidade do código fonte e, por isso, permitem avaliar se é ou não possível evoluir a versão atual do programa desenvolvido.

Neste caso, foi possível verificar a existência de **330 *code smells***. Apresentam-se de seguida o grau de severidade de cada um destes:

- ***info***: 1 ocorrência (apenas faz-se referência à presença de um comentário *TODO*);
- ***minor***: 182 ocorrências;
- ***major***: 87 ocorrências;
- ***critical***: 50 ocorrências;
- ***blocker***: 10 ocorrências.

Comparativamente à versão número um, esta implementação possui, aproximadamente, 2,5 vezes mais *code smells*, o que leva a concluir que esta, potencialmente, levará a uma interpretabilidade bastante pior.

Segundo a ferramenta *SonarQube*, demoraria cerca de 6 dias a corrigir todos estes "defeitos". Este facto traduz aquilo a que chamamos o *technical debt*, isto é, a probabilidade de ocorrências de erros no futuro. Esta ferramenta atribui a nota *A* no que diz respeito ao *technical debt*.

2.1.2.4 Cobertura

Quanto à cobertura não foram testados nenhuns aspetos intrínsecos a esta implementação.

2.1.2.5 Duplicação de código

No que diz respeito à duplicação de código existem 23 blocos repetidos em duas das classes implementadas, representando apenas 3,7% do código total.

2.2 Tarefa 2 - *Refactoring*

Nesta tarefa serão utilizadas ferramentas como o *autorefactor*, *IDEs* do *Java* que suportam *refactoring*, ou o *jStanley* para identificar e eliminar os *bad smells* e *red smells* existentes no *software* fornecido. Um estudo detalhado sobre os *smells* encontrados na(s) aplicação(ões) fornecidas, os *refactorings* aplicados e o *technical debt* obtidos deverão ser incluídos no relatório.

2.2.1 Versão 1 - *demo1*

Como foi referido anteriormente na secção 2.1.1, nesta versão foram observados vários problemas no código fonte.
Estes problemas estão divididos pelos seguintes tipos: *Bug*, *Vulnerability* e *Code Smells*.

Bugs

- ***blocker***
  Foi observado o seguinte *blocker bug*:
  Use try-with-resources or close this "ObjectOutputStream"in a "finally"clause.
  (Adicionar print)

  A classe `ObjectOutputStream` implementa a interface `AutoCloseable`, o que significa que é necessário fechá-la depois de a usar. Para isto, é recomendado que o objeto seja criado usando o padrão "*try-with-resources*", pois vai ser fechado automaticamente.

  O *IntelliJ* tem este refactoring implementado, logo a correção deste *bug* é automática.

  (Adicionar print)

- ***critical***
  Foi observado o seguinte *critical bug*:
  Save and re-use this "Random".
  (Adicionar print)

  Criar um objeto `Random` novo sempre que é necessário um valor aleatório é ineficiente e pode produzir números que não sejam completamente aleatórios. Para uma melhor eficiência e aleatoriedade, é preferível criar um único objeto `Random`, guardá-lo e reutilizá-lo.

No *Intellij*, podemos tranformar uma variável local numa variável de instância através do refactoring "*Introduce Field...*", e de seguida, podemos usar o refactoring "*move assignment to field declaration*".

(Adicionar print)

- ***major***
  Foi observado o seguinte *major bug*:
  Either override Object.equals(Object), or rename the method to prevent any confusion.
  (Adicionar print)

  O nome de método `equals` deveria ser usado exclusivamente para sobrepor `Object.equals(Object)`.

  Usando o refactoring "*Rename*"do *IntelliJ* facilmente resolvemos este problema alterando o nome do método.

  (Adicionar print)

- ***minor***
  Foi observado o seguinte *minor bug*:
  This class overrides "equals()"and should therefore also override "hashCode()".
  (Adicionar Print)

  Se dois objetos são iguais de acordo com o método `equals(Object)`, então chamar o método `hashCode()` em ambos os objetos deve produzir resultados iguais. Para isto acontecer, ambos os métodos devem ser herdados ou sobrepostos.

  Para resolver este problema, o programador apenas necessita de escrever o nome do método `hashCode` e o *IntelliJ* faz *autocomplete* ao método.

  (Adicionar print)

Vulnerability

- ***minor***
  Use a logger to log this exception.
  (Adicionar print)

  É recomendado a utilização de *Loggers* para imprimir objetos da classe `Throwable`, pois estes normalmente contêm informação sensível que pode ser exposta.

  Nesta situação, o utilizador tem de fazer o refactoring manualmente, criando o `Logger` como uma variável estática da classe, e utilizando o método `log` para escrever a exceção num ficheiro *log*.

Code Smells

- ***Blocker***
  Remove this "clone"implementation; use a copy constructor or copy factory instead.
  (Adicionar Print)

  Acho que este não faz sentido mudar, porque eles utilizam estes métodos em streams

- ***Critical***
  Refactor this method to reduce its Cognitive Complexity from 112 to the 15 allowed.
  (Adicionar print)

  Complexidade cognitiva é uma medida de dificuldade sobre o entendimento/compreensão do código. Maior complexidade cognitiva significa que muito provavelmente o programador irá ter uma maior dificuldade na manutenção do seu código.

Neste caso, a maior parte complexidade cognitiva resulta da excessiva existência de cláusulas *"catch"*.

Uma solução para este problema utilizando o refactoring do *IntelliJ*, é extrair estas partes do código para novos métodos.

(Adicionar print)

Define a constant instead of duplicating this literal "Parametros Inválidos"3 times.

Duplicar *strings* literais dificultam o processo de refactoring, pois é necessário ter o cuidado de atualizar todas as ocorrências da *string* em causa. Por outro lado, constantes podem ser utilizadas em vários sítios, mas só precisam de ser atualizadas num único sítio.

Com o *IntelliJ*, é possível utilizar o refactoring "*Introduce Constant...*", que cria uma constante com a *string* selecionada, e automaticamente substitui todos os usos deste literal pela constante criada.

(Adicionar print)

Add a default case to this switch.

Adicionar um caso *default* é programação defensiva e diminui a probabilidade do programa crachar.

O *IntelliJ* permite fazer isto automaticamente, basta selecionar a cláusula *switch* e carregar na opção "*Insert 'default' branch*".

(Adicionar print)

- ***Major***
- ***Minor***

2.2.2 Versão 2 - *demo2*

2.3 Tarefa 3 - Teste da aplicação

Nesta tarefa, tal como o nome da mesma transparece, serão utilizadas técnicas de teste de software para efectuar testes unitários em *JUnit* e, ainda, o teste de regressão da aplicação *UmCarroJá*. Para além disso, serão utilizadas sistemas para a geração automática de casos de teste para gerar testes unitários e ainda inputs para simular a execução real da aplicação. De notar que nesta etapa também se procede à análise de testes unitários através da ferramenta *JaCoCo*.

2.3.1 Versão 1 - *demo1*

2.3.1.1 Testes unitários - *JUnit*

Quanto à cobertura de testes, foi possível verificar que ...

2.3.1.2 Geração automática de testes unitários - *EvoSuite*

Quanto à cobertura de testes, foi possível verificar que ...

2.3.1.3 Geração automática de testes - *QuickCheck*

Nesta sub-tarefa foi-nos proposto gerar automaticamente ficheiros de *logs* para testar a aplicação *UmCarroJá*. Como seria de esperar, foi adoptada a utilização do sistema *QuickCheck* para o efeito, tal como foi feito nas aulas.

A implementação do gerador pretendido segue, naturalmente, a estrutura apresentada pelo exemplo disponibilizado pelos docentes na página da disciplina. Como tal, foi necessário criar essencialmente 5 tipos de geradores, cada um com o seu tipo de dados:

- *NovoProp*: registo de um novo proprietário;
- *NovoCliente*: registo de um novo cliente;
- *NovoCarro*: registo de um novo carro;
- *Aluguer*: registo de um aluguer efetuado por um cliente;
- *Classificar*: classificação atribuída a um carro ou a um cliente/proprietário.

Também é preciso salientar que existe uma diversidade considerável de atributos associados a estes novos tipos (como por exemplo, *emails*, nomes, marcas, moradas, entre outros). Por forma a garantir uma grande variedade de atributos, a maior parte dos geradores implementados utiliza o combinador *frequency*. Este gerador associa a um determinado tipo de dados uma probabilidade de escolha. Consequentemente, foi possível gerar nomes, apelidos, moradas e marcas de automóveis de forma mais realista e em grande escala.

Por fim, de maneira a executar o gerador proposto, foi necessário parametrizar numa variável o número de registos a serem escritos para o ficheiro *logs.bak* que irá armazenar toda esta informação. Desta forma, durante a execução da função ***genLogsIO*** (responsável por gerar todos os registos dos tipos de dados mencionados anteriormente), é perguntado ao utilizador quantos registos pretende produzir.

2.3.2 Versão 2 - *demo2*

2.3.2.1 Testes unitários - *JUnit*

Quanto à cobertura de testes, foi possível verificar que ...

2.3.2.2 Geração automática de testes unitários - *EvoSuite*

Quanto à cobertura de testes, foi possível verificar que ...

2.3.2.3 Geração automática de testes - *QuickCheck*

2.4 Tarefa 4 - Análise de desempenho

2.4.1 Versão 1 - *demo1*

2.4.2 Versão 2 - *demo2*

Capítulo 3

Conclusão

Apêndice A

Observações

Durante a análise da primeira tarefa associada a este trabalho prático observaram-se, na ferramenta ***Sonarqube***, variados tipos de severidade de erros. Como tal, apresentam-se de seguida, de forma mais detalhada, os mesmos:

- ***minor***:
  Falha que afeta a qualidade do código e que pode ter um impacto minorativo na produtividade do programador: linhas de código longas, entre outros.
- ***major***:
  Falha que afeta a qualidade do código e que pode ter um impacto significativo na produtividade do programador: blocos duplicados, parâmetros não utilizados, entre outros.
- ***critical***:
  Erro com baixa probabilidade de afetar o comportamento da aplicação em produção ou um problema que representa uma falha de segurança. O código deve ser examinado imediatamente.
- ***blocker***:
  Erro com alta probabilidade de afetar o comportamento da aplicação em produção. O código deve ser corrigido imediatamente.