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

20 de Janeiro de 2020

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.

2.1.3 Tarefa extra

Quanto à realização desta tarefa, após dialogar com os docentes desta unidade curricular, optou-se por dar prioridade à segunda e última tarefa extra deste projeto. Esta supremacia surge pelos simples facto de a adição de novas regras à ferramenta *SonarQube* ser impraticável. Isto é, quando as novas regras são geradas, é criado um novo ficheiro *jar* com a informação respetiva. Posteriormente, este é incorporado na ferramenta *SonarQube*, dentro da pasta *extensions/plugins*, tal como é sugerido no tutorial indicado nos *slides* dos docentes. Ao arrancar o software *SonarQube*, este acaba por colapsar ou terminar abruptamente sem indicar qualquer tipo de erro. Desta forma, tal como já tinhamos indicado juntos dos docentes, fica aqui explícito a razão pela qual a execução desta tarefa extra fica incompleta.

2.2 Tarefa 2 - *Refactoring*

Nesta tarefa são 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. É também apresentado um estudo detalhado sobre os *smells* encontrados nas aplicações fornecidas, os *refactorings* aplicados e o *technical debt*.

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.

  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.

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

  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*".

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

  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.

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

  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.

Vulnerability

- ***minor***
  Use a logger to log this exception.

  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.

  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.

  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.

  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.

  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*".

  Rename this constant name to match the regular expression '^[A-Z][A-Z0-9]*(_[A-Z0-9]+)*$'.

  Convenções de código é algo que permite uma colaboração mais eficiente entre equipas e programadores. Esta regra diz que os nomes das constantes devem seguir aquela expressão regular.

  No *IntelliJ*, basta fazer "*Rename*"à variável, e este substitui logo todas as ocorrências desta variável no projeto.

  Make the enclosing method "static"or remove this set.

  Não é recomendável uma variável estática ser modificada por um método não estático.

  Com o *IntelliJ*, é possível transformar um método não estático num método estático com o refactoring "*Make Static*".

- ***Major***
  2 duplicated blocks of code must be removed.

  Este *code smell* é difícil de resolver, porque, por vezes, é complicado identificar os blocos de código duplicados. E se os mesmos forem identificados, pode ser difícil solucionar o problema.

  Uma possível solução poderá ser a criação de um método genérico que possibilite replicar o código.

  Either remove or fill this block of code.

  A maior parte das vezes que um bloco de código se encontra vazio é porque, de facto, existe código em falta.

  Nestes casos, programador deve remover ou preencher o bloco de código manualmente.

  Rename this class to remove "Exception"or correct its inheritance.

  Classes que contêm no nome a palavra *Exception* devem estender a classe `Exception`.

  Para resolver este problema no *IntelliJ*, basta aplicar um "*Rename...*".

  Constructor has 9 parameters, which is greater than 7 authorized.

  Idealmente, um método não deve ter tantos parâmetros. Neste caso, decidimos não fazer *refactoring* pois implica a alteração de muito código.

  Extract this nested try block into a separate method.

  Blocos "*try/catch*"não devem ser colocados dentro de outros blocos "*try/catch*". A legibilidade do código é afetada severamente porque é difícil perceber que bloco é que vai apanhar cada exceção.

  Para resolver isto, o programador deve extrair o bloco "*try/catch*"para um método.

  Remove this assignment of "id".

  Campos estáticos não devem ser atualizados em construtores, mas sim serem inicializados estaticamente.

  Este refactoring é executado manualmente.

  Method has 9 parameters, which is greater than 7 authorized.

  Idealmente, um método não deve ter tantos parâmetros. Neste caso, decidimos não fazer *refactoring* pois implica a alteração de muito código.

  Remove this unused private "getX"method.

  Métodos privados que não são utilizados devem ser removidos. São código desnecessário.

  No *IntelliJ*, é possível usar a opção "*Safe Delete*"para apagarmos o método.

  Rename field "menu".

  Uma variável de instância não deve ter o mesmo nome que a classe que a contém.

Para resolver este problema, basta mudar o nome da variável.

Replace the synchronized class "Stack"by an unsynchronized one such as "Deque".

Classes sincronizadas têm um impacto muito negativo na performance da aplicação. Devem ser usadas as suas classes dessincronizadas correspondentes.

No *IntelliJ*, o *refactoring* "*Type Migration*"faz isto automaticamente, apenas é necessário selecionar a nova classe.

- *Minor*
  Move this file to a named package.

  De acordo com a especificação da linguagem Java, *packages* sem nome são fornecidos apenas como conveniência para projetos pequenos. Idealmente, estes não existem.

  No *IntelliJ* basta criar um novo *package* e mover o ficheiro para lá.

  Use isEmpty() to check whether the collection is empty or not.

  Para testar se uma coleção está vazia é recomendado utilizar o método `isEmpty()`. Não só é mais intuitivo como há casos em que a sua perfomance é melhor.

  O programador tem de fazer esta modificação manualmente.

  Use the opposite operator ("$\ddot{<}$=") instead.

  É desnecessário inverter o resultado de uma comperação booleana. É preferíve inverter a comparação em si.

  O *IntelliJ* disponibiliza este *refactoring*.

  The return type of this method should be an interface such as "List"rather than the implementation "ArrayList".

  Declarações devem usar interfaces de coleções em vez de uma implementação específica.

  Reorder the modifiers to comply with the Java Language Specification.

  Modificadores devem ser declarados na ordem correta, seguindo a especificação da linguagem Java.

  Remove the parentheses around the "e"parameter

  Inputs lambda com um só elemento não devem ter parentesis.

Antes de serem aplicados os refactorings, o projeto apresentava uma *technical debt* de 2 dias e 5 horas. Após a remoção dos diversos tipos de bugs e smells, pode-se constatar a diferença nesta *technical debt*:

- **Blocker** - 2 dias e 2 horas, ou seja, 3 horas de diferença;
- **Critical** - 2 dias, ou seja, 5 horas de diferença;
- **Major** - 2 dias e 2 horas, ou seja, 3 horas de diferença;

- **Minor** - 1 dia e 5 horas, ou seja, 1 dia de diferença;
- **Todos** - 4 horas e 20 minutos, ou seja, 2 dias e 40 minutos de diferença;

2.2.1.1 *Smells* de energia

Para além dos *code smells*, desenvolveu-se mais uma versão para este demo1. Para isso utilizou-se o *plugin* do `jStanley`, no `Eclipse`, de forma a tentar alcançar uma versão mais eficiente, do ponto de vista energético. A análise permitiu identificar algumas possibilidades de poupanças de energia, seis no total. Veremos agora, em maior detalhe, as mudanças efetuadas.

- **Change ArrayList<Rental> by AttributeList**
  Esta poupança de energia foi identificada no ficheiro `Controller.java`, mas foi decidido que a mesma não seria tomada em conta, uma vez que isto implicaria diversas mudanças, incluindo *streams*, métodos de outras classes e afins. Como não foi possível identificar a verdadeira extensão do impacto que teria esta mudança, tomou-se a decisão de manter a forma original.

- **Change ArrayList<String> by AttributeList**
  Esta poupança de energia apareceu cinco vezes no ficheiro `Model.java`. Por ser uma implementação mais fácil e com menos impacto em outras partes do projeto, procedeu-se às cinco alterações. Embora simples e aparentemente poucas (num projeto extenso), estas alterações pareceram ter um impacto positivo no consumo de energia da aplicação, como veremos a seguir, no capítulo 2.4.

2.2.2 Versão 2 - *demo2*

Para o demo2, decidiu-se explorarem-se apenas os *code smells*. De realçar que a remoção de alguns destes implicou também a remoção de alguns *bugs*.

Code Smells

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

  As regras de sobreposição de *clone* são demasiado complicadas de se compreenderem. Devem usar-se, em substituição, construtores por cópia.

  Uma solução para este problema é apagar os métodos *clone* e substituir a sua utilização por construtores de cópia.

- ***Critical***
  Refactor this method to reduce its Cognitive Complexity from 17 to the 15 allowed, ou equivalente.

  Neste caso, a maior parte da complexidade cognitiva resulta da excessiva existência de cláusulas *"catch"* e/ou do aninhamento ou uso excessivo de ciclos ou "if".

  Uma solução para este problema utilizando o refactoring do *IntelliJ*, é extrair estas partes do código para novos métodos, caso não exista outra solução. Ou então, para casos de usos excessivos de "if", substituí-los por "switch". Podem ainda simplificar-se os métodos.

Define a constant instead of duplicating this literal "Latitude Inválida"6 times, ou equivalente.

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.

Add a default case to this switch.

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

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

Use static access with "java.util.Calendar"for "DAY_OF_MONTH", ou equivalente.

Por razões de clareza, membros *static* de uma class base nunca devem ser acedidos usando um nome do tipo derivado, por ser confuso e criar a ilusão de que existem dois membros *static* diferentes.

Com o *IntelliJ*, é possível selecionar o membro e carregar na opção "*Add static import for "java.util.Calendar.DAY_OF_MONTH"*".

Make "destino"transient or serializable, ou equivalente.

Campos em classes *Serializable* devem ser *serializable* ou *transient*, mesmo que a classe nunca seja explicitamente serializada ou de-serializada. Isto previne falhas e ataques.

A solução foi declarar as variáveis *transient*.

- ***Major***
  3 duplicated blocks of code must be removed, ou equivalente.

  Este *code smell* é difícil de resolver, porque, por vezes, é complicado identificar os blocos de código duplicados. E se os mesmos forem identificados, pode ser difícil solucionar o problema.

  Uma possível solução poderá ser a criação de um método genérico que possibilite replicar o código.

  Either remove or fill this block of code.

  A maior parte das vezes que um bloco de código se encontra vazio é porque, de facto, existe código em falta.

  Nestes casos, programador deve remover ou preencher o bloco de código manualmente.

  Constructor has 9 parameters, which is greater than 7 authorized, ou equivalente.

Idealmente, um método não deve ter tantos parâmetros. Neste caso, decidimos não fazer *refactoring* pois implica a alteração de muito código.

Extract this nested try block into a separate method.

Para resolver isto, o programador deve extrair o bloco "*try/catch*"para um método. O *IntelliJ* permite fazê-lo de forma simples.

Remove this unused private "initApp"method.

Métodos privados que não são utilizados devem ser removidos. São código desnecessário.

No *IntelliJ*, é possível usar a opção "*Safe Delete*"para apagarmos o método. Esta opção verifica que não existem chamadas deste método e que é seguro apága-lo.

Rename "consumo"which hides the field declared at line 33, ou equivalente.

Uma variável declarada num *scope* mais exterior não deve ser sobreposta ou escondida por outra, impactando a legibilidade e, consequentemente, manutenibilidade do código. Pode também potenciar a criação de futuors *bugs*.

Para resolver este problema, basta mudar o nome da variável.

Add the "@Override"annotation above this method signature.

Usar anotações aumenta a legibilidade do código ao tornar óbvio que um método é sobreposto.

A solução é adicionar a anotação por cima do método.

Remove this "Double"constructor.

Construtores para *String, Double* e outros objetos usados para "embrulhar"primitivas nunca devem ser usados. Usá-los é menos claro e utiliza mais memória.

O *IntelliJ* permite remover facilmente estes "embrulhos"com a opção "*Remove boxing*".

Remove this useless assignment to local variable "kilometers", ou equivalente.

Quando um valor é calculado e depois não é utilizado, pode haver um erro no código, ou então um desperdício de recursos.

Nestes casos, deve apagar-se a atribuição, ou então alterar o código subsequente por forma a utilizar essa atribuição.

Replace this use of System.out or System.err by a logger

Se um programa escreve diretamente para o *standard output*, informações sensíveis podem ser expostas.

Apesar disto, foi decidido que não se iriam tratar estes *smells* porque a maioria das escritas para o *standard output* implica a interface criada.

Add a private constructor to hide the implicit public one.

Classes de utilidade não devem ser instanciadas. O Java adiciona um construtor público implícito por cada classe que não defina um explicitamente.

A solução é adicionar um construtor privado à classe, que pode até ser vazio.

This block of commented-out lines of code should be removed.

Código não utilizado deve ser apagado e não comentado.

O código comentado deve ser apagado.

Merge this if statement with the enclosing one.

Fundir "ifs"aumenta a legibilidade do código.

Nestes casos, deve fundir-se o *if* aninhado com o anterior. Este *refactoring* deve ser feito manualmente.

Make this anonymous inner class a lambda.

Classes anónimas podem ser complicadas e incertas.

O *IntelliJ* dá a opção "*Replace with lambda*"para estes casos.

- ***Minor***
  Move this file to a named package.

  De acordo com a especificação da linguagem Java, *packages* sem nome são fornecidos apenas como conveniência para projetos pequenos. Idealmente, estes não existem.

  No *IntelliJ* basta criar um novo *package* e mover o ficheiro para lá.

  Use isEmpty() to check whether the collection is empty or not.

  Para testar se uma coleção está vazia é recomendado utilizar o método `isEmpty()`. Não só é mais intuitivo como há casos em que a sua perfomance é melhor.

  O programador tem de fazer esta modificação manualmente.

  Use super.clone() to create and seed the cloned instance to be returned.

  Sobrepôr o método *clone()* sem implementar *Clonable* pode indicar um erro.

  O *IntelliJ* disponibiliza este *refactoring*, no entanto, este pode implicar outras alterações no código que o programador deve ter em atenção. De realçar que este *refactoring* pode ser desnecessário, uma vez que outros *smells* podem implicar a eliminação do método em questão.

  Rename the interface name to match the regular expression '[Î]A-Z][a-zA-Z0-9]*$'.

  Convenções de código é algo que permite uma colaboração mais eficiente entre equipas e programadores. Esta regra diz que os nomes das constantes devem seguir aquela expressão regular.

  No *IntelliJ*, basta fazer "*Rename*"à variável.

  Remove this unused import 'java.util.ArrayList'.

Importar bibliotecas que não são usadas reduz a legibilidade do código, uma vez que a sua presença pode causar confusão.

Pode utilizar-se a opção *Optimize imports* para todo o projeto, de modo a resolver todas as ocorrências deste *smell* de uma só vez.

Remove the literal "true"boolean value, ou equivalente.

Literais booleanos redundantes devem ser removidos de expressões de modo a melhorar a legibilidade do código.

O *IntelliJ* identifica estes casos e permite a simplicação dos mesmos, com a opção *Simplify*.

Rename this field "MIN_LATITUDE"to match the regular expression '[a-z][a-zA-Z0-9]*$', ou equivalente.

A opção "*Rename*", do *IntelliJ*, permite alterar todas as ocorrências do campo no projeto. É apenas necessário que se escreva um nome que corresponda à expressão regular.

Use "Double.parseDouble"for this string-to-double conversion.

Em vez de se criar uma primitiva "embrulhada"de uma *String* para extrair um valor primitivo, deve usar-se o método "parse". O código fica mais claro e mais eficiente.

Também neste casos, o *IntelliJ* permite fazer este *refactoring* de forma simples.

Replace this if-then-else statement by a single return statement.

Deve simplificar-se o retorno de literais booleanos em "if-then-else".

Mais uma vez, o *IntelliJ* permite simplicar estas expressões de forma simples.

Remove this use of "Double"; it is deprecated.

Uma vez descontinuadas, classes, interfaces e os seus membros devem ser evitados.

No caso de ocorrências de "Double", não só a classe foi descontinuada, como também é desnecessário o seu uso. Daí, normalmente, a solução será remover o "embrulho". Essa é opção é oferecida pelo *IntelliJ*.

Remove this unused "kilometers"local variable, ou equivalente.

Manutenibilidade é incrementada se os programadores não tiverem de descobrir para o que é que uma variável é usada. Por isso, variáveis declaradas e não usadas, devem ser removidas.

Tal como noutras situações semelhantes, o *IntelliJ* reconhece variáveis não usadas e dá a opção de remoção das mesmas.

Replace the type specification in this constructor call with the diamond operator («>").

O compilador consegue inferir o tipo, no construtor, através da declaração.

Assim, podem apagar-se os tipos nos construtores, deixando apenas o operador «>".

Reduce the total number of break and continue statements in this loop to use at most one.

Este *refactoring* é feito no interesse boa programação estruturada.

O programador tem de reduzir o número destas expressões manualmente. O uso de "if"e booleanos pode ajudar.

Declare "nif"on a separate line.

Declarar múltiplas variáveis na mesma linha é difícil de ler.

O *IntelliJ* permite, através da opção "*Split into separate declarations*", separar as declarações de todas as variáveis na mesma linha, para linhas diferentes.

Immediately return this expression instead of assigning it to the temporary variable "car", ou equivalente.

É uma má prática declarar uma variável para, logo a seguir, a retornar.

A opção "*Inline variable*", oferecida quando a variável é clicada, permite substituir a declaração da variável pelo seu retorno imediato.

Antes de serem aplicados os refactorings, o projeto apresentava uma *technical debt* de 6 dias. Após a remoção dos diversos tipos de smells, pode-se constatar a diferença nesta *technical debt*:

- **Blocker** - 5 dias, ou seja, 1 dia de diferença;
- **Critical** - 4 dias, ou seja, 2 dias de diferença;
- **Major** - 4 dias, ou seja, 2 dias de diferença;
- **Minor** - 2 dias, ou seja, 4 dias de diferença;
- **Todos** - 0 dias, ou seja, 6 dias de diferença;

Pode chamar à atenção o facto de, a soma da redução do *technical debt* para cada categoria, ser superior ao inicial (6 dias). No entanto, isto é facilmente explicável pelo facto de alguns *smells* se sobreporem, ou seja, a remoção de um de uma categoria, pode querer dizer que outro, de outra categoria, também seja resolvido.

As diferenças entre categorias podem ser explicadas pelas quantidades distintas de *smells* de cada uma.

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 do *code coverage runner* disponibilizado pelo *IntelliJ*.

2.3.1 Versão 1 - *demo1*

2.3.1.1 Testes unitários - *JUnit*

Como iremos ver de seguida, a maior parte dos testes unitários são gerados automaticamente pelo o *EvoSuite*, mas além disso, decidimos fazer uma pequena análise pessoal ao código do projeto. Concentramos a nossa análise no *package* `Model`, mais especificamente nas classes `Cars`, `Rentals` e um `UmCarroJa`, que achamos que são as mais relevantes a nível de evolução do sistema.

Estes testes foram colocados no *package* `test` e estão devidamente anotados seguindo a anotação dos testes manuais do *JUnit*.

Quanto à cobertura, a análise efetuada pelo *IntelliJ* (com destaque apenas no *package* `Model`) mostra que obtivemos um total de cobertura de: 96% em classes, 46% em métodos e 36% em linhas de código.

Como seria de esperar, os testes não apresentam grande cobertura na totalidade da aplicação, pois é complicado cobrir todos os casos devido à elevada extensão da implementação deste projeto.

Uma verdadeira análise de cobertura encontra-se no ponto seguinte, onde é descrita a geração automática de testes unitários com o *EvoSuite*.

2.3.1.2 Geração automática de testes unitários - *EvoSuite*

Usou-se o *plugin* do *IntelliJ* para utilizar o *EvoSuite* no projeto, gerando testes unitários de forma automática para todo o projeto.

Foi possível, com recurso à opção `Run with coverage` do *IntelliJ*, fazer a **análise de cobertura de código**. De seguida, apresentam-se os resultados obtidos.

Tabela 2.1: Análise do consumo energético - *demo1*

***Package*/Classe**	**Classes**	**Métodos**	**Linhas**
Controller	100% (2/2)	100% (5/5)	92% (207/225)
Exceptions	69% (9/13)	100% (0/0)	100% (9/9)
Model	100% (13/13)	100% (134/160)	77% (515/661)
Utils	100% (2/2)	39% (11/28)	50% (20/40)
View	100% (13/13)	100% (75/75)	100% (421/441)
Main.java	100% (1/1)	100% (1/1)	83% (10/12)

A partir desta tabela, podemos perceber que quase todo o código é executado. No entanto, percebe-se que certas partes do mesmo não o são, pelo que a sua remoção deve ser pensada. É fácil perceber que algumas das exceções poderiam ser descartadas. Importa realçar que esta análise foi apenas feita ao código original, ou seja, sem qualquer tipo de *refactoring*. A análise de outras versões, pós-*refactoring*, poderá apresentar ligeiras melhorias.

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. De notar também que os elementos deste grupo tiveram em consideração a geração de *NIF's* e de matrículas sem repetições, permitindo assim uma simulação fidedigna e correta da aplicação.

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*

Como se pôde constatar na primeira versão da implementação deste projeto de *POO* (isto é, na *demo1*), a especificação de testes unitários em *JUnit* acaba por ser inviável uma vez que existe uma elevada extensão de classes. Como tal, de forma a ser relevante a execução desta tarefa, seria necessário desenvolver muitos casos de teste para estas classes. Só desta forma obter-se-ia uma cobertura de testes muito próximo dos 100%. Dada a existência de ferramentas computacionais que tratam a geração automática de testes unitários, como é o caso do *EvoSuite*, optou-se por dar prioridade a utilização deste utensílio para este tipo de tarefa.

2.3.2.2 Geração automática de testes unitários - *EvoSuite*

Quanto à geração automática de testes unitários, foi possível gerar corretamente todos as classes de teste. No entanto, ao correr não foi possível obter resultados fidedignos, devido ao mau funcionamento do programa em questão. Quanto à cobertura dos testes, foi possível verificar, tal como é expectável, más percentagens a este respeito.

2.3.2.3 Geração automática de testes - *QuickCheck*

Relativamente à geração automática de testes em *QuickCheck*, a abordagem tomada nesta implementação foi exatamente igual à que foi explorada anteriormente na primeira versão (ver informação no capítulo 2.3.1.3).

2.4 Tarefa 4 - Análise de desempenho

2.4.1 Versão 1 - *demo1*

Após as diversas alterações e eliminações de *smells* é necessário verificar se estas tiveram impacto sobre o desempenho do programa. Para tal foi utilizada a ferramenta *RAPL*, disponibilizada pelos docentes, para analisar o consumo energético do programa. Como primeira etapa foram registados os valores energéticos com *logs* de diversos tamanhos (5, 55, 5555, 55555). De seguida encontram-se as tabelas com os resultados obtidos:

Tabela 2.2: Análise do consumo energético - *demo1*

Smells?	**Tamanho do *input***	***dram*** (em *MB*)	***cpu*** (em *ms*)	***package*** (em *J*)
Sim	5	0.024292000000002645	0.1289059999999722	0.258239999999887
Não	5	0.04699700000003304	0.10449199999993652	0.3303230000001349
Sim	55	0.18859800000001314	0.3990479999999934	1.139159999999947
Não	55	0.078185999999959684	0.31451400000003105	0.7086789999998473
Sim	5555	0.3440559999999664	3.8085930000000303	5.577698000000055
Não	5555	0.32525699999996505	3.6011349999999993	5.260498000000098
Sim	55555	3.675475999999435	59.24536100000114	89.38207999999577
Não	55555	3.8429559999999583	60.6569210000016	91.23339899999701

2.4.2 Versão 2 - *demo2*

Tabela 2.3: Análise do consumo energético - *demo2*

Smells?	**Tamanho do *input***	***dram*** (em *MB*)	***cpu*** (em *ms*)	***package*** (em *J*)
Sim	original	2.5678709999999683	30.438354000000004	41.03033400000004
Não	—	—	—	—

2.4.3 Tarefa extra

Nesta tarefa adicional é requerido uma análise detalhada por cada *smell*. Dito por outras palavras, é necessário fazer um estudo sobre como cada *smell* individualmente influência (melhora ou piora) o desempenho do *software*. Como tal, foram executadas ambas as versões do programa em questão com cada tipo de *smells* de forma isolada. Consequentemente, sobre um conjunto de funções do programa, obtiveram-se os seguintes resultados:

Tabela 2.4: Análise do consumo energético *smell* a *smell* - *demo1 -> logs original*

Smells	***dram*** (em *MB*)	***cpu*** (em *ms*)	***package*** (em *J*)	**Tempo** (em *ms*)
noMinor	0.125	1.8643190000002505	2.6281740000013087	56
noMajor	0.10607900000013615	1.8434440000000905	2.383972999999969	67
noCritical	0.10449199999993652	1.999877999999626	2.481445999999778	55
noBlocker	0.10296599999992395	1.799011000000064	2.240355999998428	53
noRedSmells	0.12378000000035172	2.0324699999996483	2.7041619999999966	51
noSmells	0.10351600000012695	1.7691039999999703	2.2634280000002036	62

Tabela 2.5: Análise do consumo energético *smell* a *smell* - *demo1 -> logs tamanho 55555*

Smells	***dram*** (em *MB*)	***cpu*** (em *ms*)	***package*** (em *J*)	**Tempo** (em *ms*)
noMinor	12.97338899999977	168.750792999999255	304.18981899999926	17032
noMajor	12.753662000000077	164.5404660000022	295.9130860000005	16745
noCritical	13.174987999999757	168.60394299999825	302.7049559999941	17144
noBlocker	15.805237000000034	183.67248599999948	324.94183299999713	17329
noRedSmells	12.960143999999673	168.22857600000134	298.2542109999995	16595
noSmells	12.254883000000063	160.48974599999933	288.387023999996	16128

Tabela 2.6: Análise do consumo energético *smell* a *smell* - *demo2*

Smells	***dram*** (em *MB*)	***cpu*** (em *ms*)	***package*** (em *J*)	**Tempo** (em *ms*)
noMinor	—	—	—	—
noMajor	2.204895000000306	24.839966000000004	34.237121999998635	1119
noCritical	2.4993890000000647	29.64740000000029	40.53625499999998	1282
noBlocker	—	—	—	—
noRedSmells	—	—	—	—
noSmells	—	—	—	—

Capítulo 3

Conclusão

Após a demonstração da abordagem tomada para cada uma das tarefas propostas neste trabalho prático e, ainda, a exibição dos resultados obtidos em cada uma das mesmas, dá-se por concluído a execução deste projeto. Durante o processo de realização do mesmo foi possível analisar o código fonte de ambas as versões implementadas, identificando os *bad smells* e o seu *technical debt*. Posteriormente, foram aplicados *refactorings* de maneira a eliminar os *bad smells* encontrados e, deste modo, reduzir o respetivo *technical debt*. Numa terceira fase, foi testado com sucesso o *software* em questão de forma a garantir que o mesmo exprimia o comportamento esperado. E, por fim, foi realizada uma análise a nível do desempenho das duas versões implementadas (antes e depois da aplicação de *refactorings*).

Depois de terminadas as tarefas, refletimos sobre os resultados obtidos e das conclusões que podemos tirar, tanto desses resultados como do trabalho realizado.

A primeira tarefa foi importante para assimilarmos alguns dos conteúdos lecionados ao longo da UC, mas também, e principalmente, para termos uma ideia inicial do projeto que analisávamos e do extenso trabalho que teríamos pela frente. Pudemos ainda perceber a importância destas análises para uma manutenção mais fácil do código.

A segunda tarefa, talvez a mais extensa de todas, obrigou à criação de diversas versões do mesmo código. Foi importante para os elementos deste grupo perceberem, e futuramente corrigirem, as (suas) más práticas de programação. Permitiu ainda desenvolver um conhecimento mais profundo do projeto, essencial para a realização da próxima tarefa.

A terceira tarefa resume-se à geração de teste unitários, automáticos ou não. Os testes são importantes no desenvolvimento de *software*, porque permitem identificar potenciais problemas e resolvê-los.

Para a quarta tarefa, realizaram-se análises o desempenho do(s) projeto(s) analisado(s). Para além das versões originais, também se fez uma análise das versões desenvolvidas ao longo deste extenso trabalho. É desta tarefa que podemos retirar mais conclusões. Pode perceber-se, com recurso às tabelas apresentadas, que o tratamento e eliminação de alguns dos diferentes tipos de *smells* não têm um grande impacto no desempenho da aplicação. No entanto, parece haver uma tendência para um melhor desempenho, quando fazemos a análise *smell* a *smell*. Ainda assim, pode argumentar-se que o tratamento de *code smells* realizado, tinha como objetivo melhorar a legibilidade e manutenibilidade do código, pelo que uma melhoria no desempenho é sempre um extra.

Em nota final, podemos concluir que os erros e más práticas são mais comuns do que poderíamos pensar antes de iniciarmos este projeto. É então importante dedicar algum tempo para a análise, *refactoring* e teste do código. Não apenas quando o mesmo está terminado, mas também em outras fases do desenvolvimento.

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.