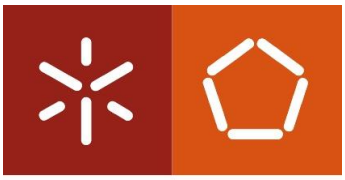

Universidade do Minho

Departamento de Informática

Trabalho Prático
Análise e Teste de Software

Bruno Martins (a80410)
Eduardo Jorge Barbosa (A83344)
Filipe Monteiro (a80229) Márcio Sousa (A82400)
Miguel Brandão (A82349)

24 de Janeiro de 2020

Conteúdo

1 Introdução

2 Qualidade de Código

Nesta secção irá ser abordada a qualidade de código de cada *demo*. Para tal, iremos executar uma análise manual de código, com o objetivo de identificar potenciais atributos negativos, bem como analisar *code smells* identificados pelo *sonarqube*. Com estes dados, torna-se então possível fazer uma asserção sobre a qualidade do código apresentado.

2.1 *Demo 1*

Numa análise preliminar manual foi possível detetar alguns *code smells*, sendo a maioria do tipo *Bloaters*, *Object Orientation Abusers* e *Dispensables*.

Rapidamente se descobriu que maior parte dos *smells* encontram-se concentrados no pacote *Models*. Sendo este pacote responsável por toda a lógia de negócio, e consequentemente o pacote mais extenso, esta observação não é inesperada.

Aumentando a granularidade da nossa análise, algumas classes problemáticas foram detetadas:

- Car
- Cars
- Parser
- UmCarroJa
- Controller
- Views
- Menu

Apesar de esta lista ser algo extensa, muitos dos erros identificados numa classe são transversais às restantes.

Na classe *Car* começamos por observar uma grande quantidade de variáveis de instância. Isto leva a que esta classe fique muio extensa e trate de muita lógica associada, podendo este comportamento ser abstraído para outra classe. Ainda nesta classe foi detetado um *switch statement* mas devido ao contexto de *parsing* de dados este não necessita de ser modificado.

Por sua vez, na classe *Cars* foi possível identificar um método, *getCar*, muito extenso. Este método, que se estende por 29 linhas, apresenta pedaços de código repetidos. Esta lógica poderia ser abstraída e extraída para métodos auxiliares, diminuindo assim o tamanho do *getCar*.

Um *smell* aparentemente berrante encontra-se na classe *Parser*, onde existe um *switch* com 80 linhas. No entanto, devido à natureza da aplicação, sujeita ao paradigma orientado a objectos, bem como a experiencia dos alunos que a programaram, este comportamento é difícil de ser abstraido em vários métodos ou reduzir o mesmo.

Na classe *UMCarroJa* podemos verificar a utilização de *streams* com uma complexidade excessiva, o que leva a que o código possa ficar um pouco ofuscado. Nesta classe deverão ser extraídos muitos dos comportamentos das *streams* para outros métodos de forma a tornar o código de cada uma mais conciso, legível, e adequado ao paradigma em que foi programado. Esta análise também levou à deteção de código duplicado em muitos métodos.

O pior caso encontrado trata-se sem dúvida da classe *Controller*. Esta é a classe que menos se encontra de acordo com as boas práticas de programação. Começamos por detetar um método com mais de 300 linhas de código, para agravar ainda mais a situação, este método, é composto unicamente por um *switch* que ocupa a maioria dessas linhas. Este método deveria ser no mínimo dividido em vários métodos auxiliares ou deveria ser extraído comportamento para outras classes.

Também no pacote *Views* foram detetados alguns *smells*. Começamos por detetar na classe *Table* o método *toString*, que além de ser extremamente longo, aplica funções matemáticas que

podem ser simplificadas de modo a não serem tão complexas. Na classe *Menu* voltamos a ver um abuso de *switchs* com muita lógica dentro de cada *case*.

Podemos então concluir que algumas dificuldades prevalecem ao longo do código, nomeadamente a abstração de padrões repetidos e desacoplamento de responsabilidades.

2.1.1 SonarQube

Após a fase de análise manual foi utilizado um *software* de análise estática para detetar outras más práticas de desenvolvimento.

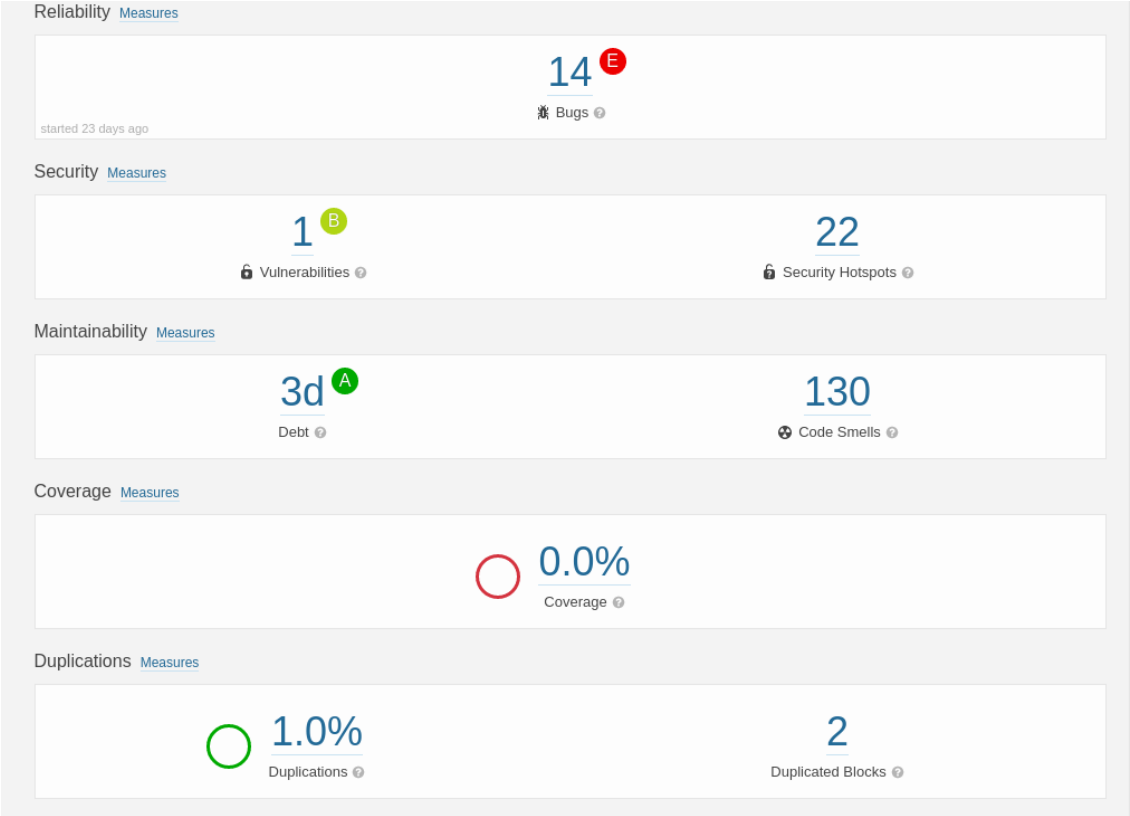

Figura 1: Resultado da análise do *SonarQube*

Depois de analisar o *code smells* apontados pelo programa foi possível observar que este também apontou a classe *Controller* como sendo a que apresenta um maior número de erros, apontando falhas do tipo: código duplicado, complexidade cognitiva demasiado elevada (112, sendo o máximo 15) e incoerências no *switch* tais como a falta de um caso por omissão.

Ao longo da restante análise foi possível identificar outros *smells*:

- Demasiados parâmetros em métodos ou construtores;
- Extração de métodos para que não haja métodos muito longos;
- Código duplicado;
- Código morto;
- Substituir a implementação do *clone*;

Também foi detetado que o *software* aponta muitos *smells* de convenções, como por exemplo: nome de classes, nome de variáveis. Relativamente a estes pontos, mesmo que muitas vezes apontados como *major flaws*, não foi dada tanta importância como aos referidos anteriormente. De notar que a análise executada pelo *Sonarqube* corrobora as falhas encontradas na análise manual.

2.1.2 *Smells* energéticos

- Uso de Exceções;
- Utilização de *gets* e *sets*;
- Utilização de *streams*;
- Não reutilização de Objetos;

2.2 *Demo 2*

Numa primeira análise ao código deste projeto podemos verificar logo a falta de uma aquitetura definida. Esta observação torna-se óbvia devido à não utilização de pacotes para a divisão lógica do código. Isto tornou a análise muito mais complicada porque por vezes não era claro o que deveria fazer cada classe.

Observando de forma geral todas as classes vemos o abuso de comentários ao longo do código, quer para explicar o significado de alguma variável ou esclarecer alguma decisão num algoritmo. Este utilização claramente excessiva entra na classe de *smells Dispnsables* pois torna a leitura de código desnecessariamente mais complexa.

Nas restantes classes vemos um erro recorrente que é a escrita de métodos muito longos. Isto verifica-se por exemplo nas classes: *CoordinateManager, Input, ParDatas, ParseDados, UmCarroJa e Weather*. As correções destas classes podem ir desde do *refactor* destes métodos em métodos *auxiliares*, até a criação de novas classes. Por exemplo, a classe *UmCarroJa* apresenta comportamento que poderia ser atribuído a classes diferentes pois abrange domínios muito distintos.

Na classe *Aluguer* podemos ver uma extensa lista de variáveis de instância. Desta classse deveria ser extraída pelo menos uma outra classe, de modo a abstrair o comportamento associado aos diversificados tipos das mesmas.

A clara falta de uma arquitetura leva à violação de vários princípios de bom *software* (*Dont repeat yourself*, *KISS*, abstração de comportamento repetido, responsabilidade única, etc).

2.2.1 *SonarQube*

Na figura seguinte podemos observar a análise geral feita pelo *software*.

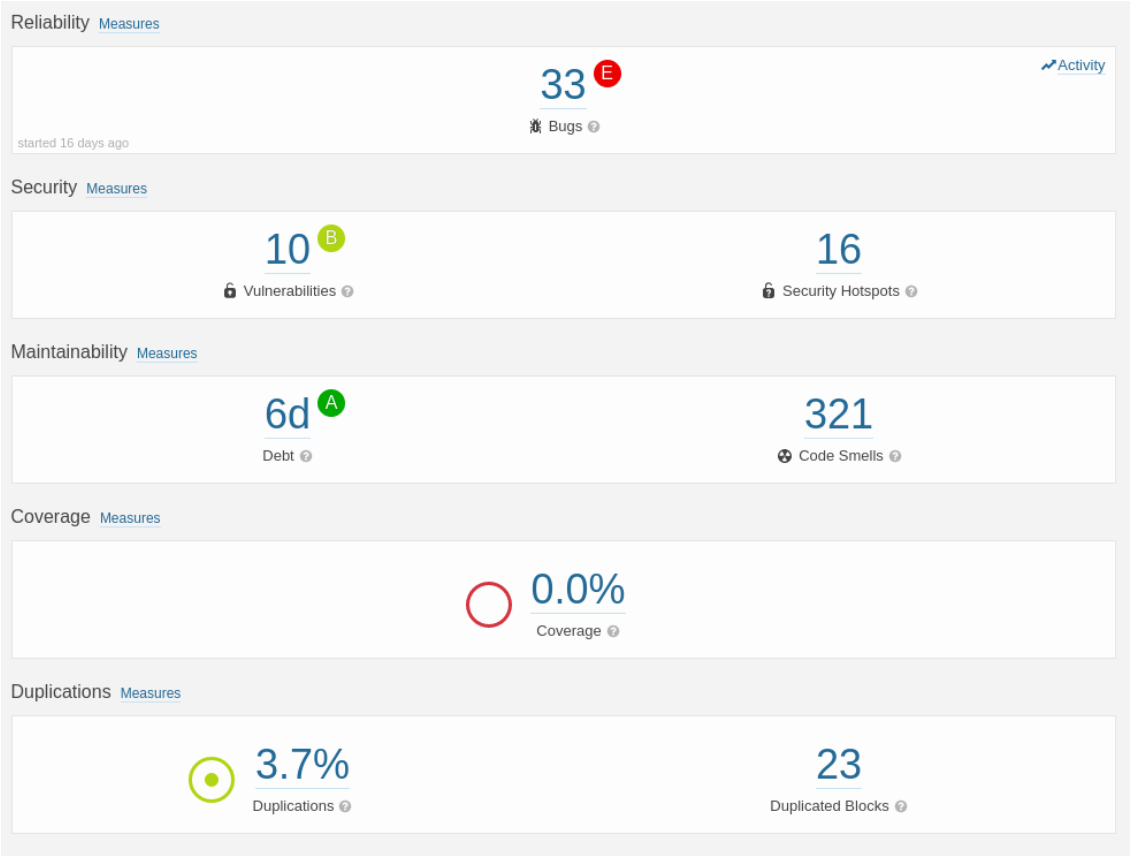

Figura 2: Resultado da análise do *SonarQube*

Na análise do *SonarQube* vemos que, comparativamente ao projeto anterior, existem muitos mais *smells* levando a uma dívida técnica muito maior.

A maioria dos *smells* apontados são repetitivos, sendo estes:

- Demasiados parâmetros em métodos ou construtores;
- Possivel extraçao de métodos para que não haja métodos muito longos;
- Código duplicado;
- Código morto ou desnecessário;
- Substituir a implementação do *clone*;
- Reduzir complexidade cognitiva;
- Tornar variáveis serializáveis;
- Utilização excessiva de comentários;

Tal como no projeto anterior são apontados muitos *smells* de convenções, sendo estes mais uma vez dados pouca importância.

2.2.2 *Smells* energéticos

- Uso de Exceções;

- Utilização de *gets* e *sets*;
- Não reutilização de Objetos;
- utilização de *streams*;

2.3 Análise Comparativa

Fazendo uma análise comparativa entre os dois projetos rapidamente verificamos que quer pela observação humana, quer pela análise do *Sonarqube*, a *Demo 1* apresenta um código de mais fácil leitura e sem tantos *smells* associados. Isto deve-se em certa parte à falta de uma arquitetura definida e um aparente não conhecimento profundo tanto da linguagem como do paradigma dos programadores da *demo2*. No entanto, podemos ver que existem alguns erros recorrentes em ambos os projetos, presumivelmente provenientes da falta de experiência no desenvolvimento de *software*.

No caso dos *smells* energéticos ambos os projetos repetem os mesmos smells, sendo estes os mais críticos a utilização de *streams* e o uso e abuso de exceções em toda a extensão do código.

3 Refactoring

Nesta secção serão abordados os *refactorings* aplicados a cada *demo* para mitigar os *code smells* e *red smells* detetados.

3.1 Code smells

Foram identificados uma multitude de *code smells* em ambas as aplicações, tanto via análise humana como automática. No entanto, foi considerado que, no contexto deste trabalho, muitos dos *smells* apontados pelo Sonarqube eram supérfluos, pelo que o foco foi mais dirigido para os *smells* descobertos por análise humana e pelos smells apontados pelo IDE, o IntelliJ.

3.1.1 *Demo1*

Aqui, listamos os refactors efetuados no *demo1*, ordenados por classe:

Car

Encontrado por	Linha	Smell	Resolução	Technical Debt
Sonarqube	59	No default case in switch	Move 'throw' exception to default case	1
Sonarqube	76	Long method	Extract streams to methods	15

Parser

Encontrado por	Linha	Smell	Resolução	Technical Debt
Human		Long method	Extract long cases	20
Human		Logic obfuscation	Change if block to execute if the reverse of the original condition is met	10

UMCarroJa

Encontrado por	Método	Smell	Resolução	Technical Debt
Human		Duplicated methods for getBestClientTravel	Remove largest one	10
Human	getBestClientsTravel	Long method	Extract methods	20
Human	getBestClientUses	Method name not descriptive	Change to getBestClientByUses	2
Human		Long streams	Extract part of streams to methods	60
Human	getBestClients	Dead code; method is similar to getBestClientTravel	Remove	5

Controller

Encontrado por	Método	Smell	Resolução	Technical Debt
Sonarqube	154	Collection.isEmpty() should be used to test for emptiness	Use isEmpty() to check whether the collection is empty or not	4

Car

Encontrado por	Método	Smell	Resolução	Technical Debt
Sonarqube	hasRange	Boolean checks should not be inverted	Invert check	2
Human		Class is too large	Extract class Vehicle; refactor dependent code	90
Sonarqube		Unused imports	Remove	1

Cars

Encontrado por	Método	Smell	Resolução	Technical Debt
Sonarqube	listOfCarType	Parentheses should be removed from a single lambda input parameter when its type is inferred	Remove parenthises	2
Sonarqube	getCarCheapest	Unused method parameters	Remove	2

Rentals

Encontrado por	Linha	Smell	Resolução	Technical Debt
Sonarqube	13	Modifiers should be declared in the correct order	Reorder modifiers	2

Menu

Encontrado por	Método	Smell	Resolução	Technical Debt
Sonarqube	back	Collection.isEmpty() should be used to test for emptiness	Use isEmpty() to check whether the collection is empty or not.	2
Human	toString	Large switch cases	Extract cases to variable argument function 'printMenuOptions'	30

Table

Encontrado por	Método	Smell	Resolução	Technical Debt
Human	toString	Expressions can be replaced with Math.max	Replace expressions	2
Human	toString	Long method	Extract methods	30
Human	toString	Non descriptive variable naming	Improved variable naming	30

3.1.2 *Demo2*

Aqui, listamos os refactors efetuados no *demo2*, ordenados por classe:
Comum a todo o código

Encontrado por	Smell	Resolução	Technical Debt
Human	Comments in instance variables	Remove	5
Human	Poor indentation	Auto-reformat code (and auto-optimize imports)	1

PorDatas

Encontrado por	Método	Smell	Resolução	Technical Debt
Human	isAvailable	Complicated code	IntelliJ auto-refactor: simplify	1
Human	isAvailable	Complicated method	Extract 'if' conditions; remove comments	15

UmCarroJa

Encontrado por	Método	Smell	Resolução	Technical Debt
Human	aluguersClassificarCliente	Too many nested blocks	Extract method	10
Human	classificarCliente	Too many variables in method	Extract method, remove unnecessary variables	25

ParseDados

Encontrado por	Método	Smell	Resolução	Technical Debt
Human		Duplicated code; long methods	Remove duplications by extracting methods	60

Input

Encontrado por	Método	Smell	Resolução	Technical Debt
Human		Duplicated code; long methods	Remove duplications by extracting methods	60

UmCarroJaApp Também houve a intenção de fazer um refactor a esta classe, devido ao seu grande número de linhas e confusão. No entanto, por estes mesmos aspetos e devido aos problemas gerais de mau planeamento e organização, determinamos que não seria eficiente, em termos de tempo, de fazer um refactor a esta classe, dado que estimamos que o tempo dispensado a fazê-lo seria melhor empregue na completa reescrita do projeto.

3.2 Red Smells

3.2.1 *Demo1*

Utilização de *gets* e *sets*

Em ambos os *demos*, foi identificada a utilização de getters e setters para acesso às variáveis de instância. Apesar desta ser uma boa prática, o acesso direto às variáveis é mais eficiente em termos energéticos, pelo que deveria ser feito um *refactor* do código, como exemplificado abaixo. No entanto, tal não foi realizado, devido ao custo temporal de fazer um refactor de qualidade ser superior ao custo de refazer o projeto.

Um exemplo deste refactor pode ser observado na classe *Cliente.java* do *demo1*, sendo os excertos de código original:

```
private Point pos;
private final List<Rental> pendingRates;



private Client(Client u) {
super(u);
this.pos = u.getPos().clone();
this.pendingRates = new ArrayList<>(u.pendingRates);
}
```

e os excertos de código refactored:

```
Point pos;
final List<Rental> pendingRates;



private Client(Client u) {
    super(u);
    this.pos = u.pos.clone();
    this.pendingRates = new ArrayList<>(u.pendingRates);
}
```

3.2.2 Streams

Outro *red smell* identificado em ambos os *demos*, mas particularmente mais predominante no *demo1*, foi o uso de streams. Estas, embora claramente úteis, são menos energéticamente eficientes que os seus equivalentes escritos com ciclos, pelo que foram utilizadas as ferramentas de auto-refactor do IntelliJ para substituir as streams por ciclos em quase todo o código de ambos os projetos, como está representado no seguinte exemplo, da classe *UmCarroJa* do *demo2*.

Excerto original:

```
if (aux != null) {
    return aux.stream().map(Aluguer::clone).collect(Collectors.toList());
}
```

Excerto refactored:

```
if (aux != null) {
    List<Aluguer> list = new ArrayList<>();
    for (Aluguer aluguer : aux) {
        Aluguer clone = aluguer.clone();
        list.add(clone);
    }
    return list;
}
```

3.2.3 Exceptions

Outro *red smell* encontrado foi o uso de *Exceptions*. Apesar de ser possível fazer o refactor do código para eliminar este smell, como exemplificado abaixo, tal seria contraprodutivo pois, num projeto desta dimensão e com os erros de conceção supremencionados, seria mais rápido refazer o projeto do início do que um refactor de qualidade.

Excerto original da classe *Cars* do *demo1*:

```
void addCar(Car a) throws CarExistsException {
    if(this.carBase
            .putIfAbsent(a.getNumberPlate(), a)
            != null) {
                throw new CarExistsException();
        }
}
```

Excerto refactored:

```
void addCar(Car a) {
    this.carBase.putIfAbsent(a.getNumberPlate(), a);
}
```

3.2.4 Não reutilização de objetos

Mais uma vez, devido à abundante presença deste *smell* e aos restantes problemas já destacados, não se pode justificar um refactor ao código.

4 Gerador de logs

Com o objetivo de testar as implementações apresentadas, bem como o seu *refactor*, é necessário gerar um conjunto de *logs* que cada aplicação tem que consumir. O ficheiro de *logs* deve respeitar um formatado previamente acordado.

4.1 Pretty Printer

```
class PrettyPrinter a where
  pp :: a -> String
```

Figura 3: PrettyPrinter Typeclass

Esta *typeclass* é utilizada para representar textualmente os tipos de dados do sistema, respeitando o formato acordado.

4.2 Data

Analisando o ficheiro de *logs* fornecido como exemplo, as seguintes entidades foram identificadas:

- Cliente
- Proprietário
- Aluguer
- Carro
- Classificação

Cada uma destas entidades foi colocada num modulo próprio, com a excepção dos **Clientes** e dos **Proprietários**. Dado que estas entidades partilham vários atributos, foram colocadas no mesmo modulo (**Actor**). Todas as entidades implementam uma instância da *typeclass* **Arbitrary** e **PrettyPrinter**. Visto que os dados gerados necessitam de ser congruentes, a geração de algumas instâncias não pode ser totalmente aleatória, sendo uma exportada uma função do tipo:

```
genX :: [a] -> Gen [x]
```

Figura 4: Função geradora (exemplo)

onde **a** é a informação vinda de um outro gerador e **x** é a entidade gerada.

Graças a grande capacidade expressiva de Haskell, que suporta tipos de dados algébricos, permite facilmente representar estas entidades no nosso código.

4.2.1 Clientes

Um Cliente é representado em Haskell da seguinte forma:

```
data Cliente = Cliente Nome Nif Email Morada Pos Pos deriving (Show)
```

Figura 5: Representação de um Cliente

Visto que um cliente não necessita de nenhuma informação externa para ser gerada uma instância congruente, a função de geração é bastante simples.

```haskell
instance Arbitrary Cliente where
  arbitrary = genCliente

genCliente :: Gen Cliente
genCliente = do
  n <- genNome
  nif <- genNIF
  let email = genEmail nif
  morada <- genMorada
  x <- genPos
  Cliente n nif email morada x `fmap` genPos
```

Figura 6: Geração de um Cliente

De forma a gerar nomes, moradas, emails, e NIFs **com significado** algum cuidado foi tido em conta. Foram criadas algumas listas auxiliares, recolhendo nomes próprios, apelidos, distritos, entre outros dados necessários. O exemplo mais interessante encontra-se na geração do nome:

```haskell
genNome :: Gen Nome
genNome = do
  l <- frequency [(40, pure nomesPropriosM), (60, pure nomesPropriosF)]
  nome <- elements l
  n <- choose (1, 4)
  apelido <- getGen (genApelido n) []
  (return . join . intersperse " ")  (nome : apelido)
```

Figura 7: Geração do nome de um Cliente

Primeiramente é escolhido o nome próprio, existindo 40% de probabilidade de ser uma mulher e 60% de ser um homem. Estas percentagens são totalmente aleatórias. De seguida é escolhido o número de apelidos (entre 1 e 4). Estando o número de apelidos escolhido, são gerados N apelidos não repetidos através de uma lista.

```haskell
genApelido :: Int -> CondGen [String] [String]
genApelido 0 = pure []
genApelido n = do
  l <- get
  a <- lift (elements apelidos)
  if a `elem` l
    then genApelido n
    else put (a : l) >> genApelido (n -1) >>= \u -> return (a : u)
```

Figura 8: Geração de apelidos

As restantes funções auxiliares utilizadas na geração da informação do cliente seguem todas este principio de criar informação consistente.

Sendo do nosso interesse a geração de múltiplos clientes, uma função foi criada para esse efeito. De forma a criar clientes únicos os clientes são **distinguidos pelo seu NIF**.

```haskell
genClientes :: Int -> Gen [Cliente]
genClientes n = genMultiCond n ( \a@(Cliente _ nif _ _ _ _) -> (a, nif) )
```

Figura 9: Geração de clientes

A função *genMultiCond* é exportada pela biblioteca de geração condicional criada por nós para esta projecto. Esta biblioteca será apresentada mais à frente.

4.2.2 Alugueres

Um Aluguer é representado da seguinte forma:

```haskell
data Aluguer = Aluguer Nif Pos Pos Tipo Preferencia deriving (Show)
```

Figura 10: Representação de um Aluguer

Ao contrário dos clientes, um aluguer necessita de informação externa para gerar instâncias válidas. Isto pode ser notado pela assinatura da função de geração de vários clientes:

```haskell
genAlugueres :: [Nif] -> Gen [Aluguer]
genAlugueres nifs = genMultiFrom nifs ( \nif (Aluguer _ x y t p) -> Aluguer nif x y t p )
```

Figura 11: Representação de um Aluguer

Em vez de receber um número que especifica o numero de alugueres a serem gerados, esta função recebe uma lista de NIFs que deverão ser utilizados para gerar cada aluguer. O número de alugueres gerados é, naturalmente, igual ao comprimento da lista passada. Tal como a função de geração de clientes, a *genAlugueres* utiliza uma função da biblioteca criada.

4.3 Biblioteca de geração condicional genérica

De forma a facilitar a geração das entidades foi criada uma biblioteca de geração condicional genérica, denominada de *CondGen*.

O tipo central desta biblioteca é o *type synonym CondGen st a*, que junta a monad de estado com a monad de geração de valores aleatórios do *QuickCheck*.

```haskell
type CondGen st a = StateT st Gen a
```

Figura 12: Tipo principal

O *type parameter st* é o tipo de dados que *CondGen* utiliza para manter estado, e o *a* é o *type parameter* que representa os dados a gerar.

As funções principais desta biblioteca são as seguintes:

- genMultiCond: gera N entidades que respeitem uma dada condição
- genMultiCondFrom: gera N entidades que respeitem uma dada condição, usando uma lista de valores
- genMultiFrom: gera N entidades a partir de uma lista de valores

```haskell
genMultiCond :: (Arbitrary a, Eq b) => Int -> (a -> (a, b)) -> Gen [a]
genMultiCondFrom :: (Arbitrary a, Eq c) => [b] -> (b -> a -> a) -> (a -> c) -> Gen [a]
genMultiFrom :: (Arbitrary a) => [b] -> (b -> a -> a) -> Gen [a]
```

Figura 13: Funções exportadas pela biblioteca

Estas funções são implementadas à custa de funções internas (não exportadas). A título de exemplo:

```haskell
genMultiCondFrom :: (Arbitrary a, Eq c) => [b] -> (b -> a -> a) -> (a -> c) -> Gen [a]
genMultiCondFrom bs f g = getGen (genMultiCondFrom' f g) (bs, [])

getGen :: CondGen st a -> st -> Gen a
getGen = evalStateT

genMultiCondFrom' :: (Arbitrary a, Eq c) => (b -> a -> a) -> (a -> c) -> CondGen ([b], [c]) [a]
genMultiCondFrom' f g = do
  (l, c) <- get
  if null l
    then return []
    else do
      z <- lift arbitrary
      let p = g z
      if p `elem` c
        then genMultiCondFrom' f g
        else do
          let n = f (head l) z
          put (tail l, p : c)
          u <- genMultiCondFrom' f g
          return (n : u)
```

Figura 14: Exemplo de uma implementação interna

4.3.1 Performance

Para o leitor mas atento, algumas decisões podem parecer pouco eficientes, como por exemplo, gerar uma entidade e só depois verificar que compre o requisito, em vez de gerar verificando o requisito. No entanto, sendo Haskell uma linguagem *lazy*, o código necessita apenas gerar a parte da entidade que é testada pela condição, sendo que se for verificada, gera o resto (quando necessário).

4.4 Interface

O gerador utiliza uma interface na linha de comandos, construída utilizando a biblioteca *optparse-applicative*. Esta interface permite especificar o número de proprietários, donos e ficheiro para onde escrever os *logs*.

```
generator - gerador de valores aleatorios para UmCarroJa

Usage: generator-exe [-c|--clientes INT] [-p|--proprietarios INT]
                     (-f|--ficheiro FICHEIRO) --carros INT --alugueres INT
  Gera um ficheiro de logs para UmCarroJa

Available options:
  -c,--clientes INT        Numero de clientes (default: 20)
  -p,--proprietarios INT   Numero de proprietarios (default: 20)
  -f,--ficheiro FICHEIRO   Ficheiro de dump
  --carros INT             Numero de carros
  --alugueres INT          Numero de alugueres
  -h,--help                Show this help text
```

Figura 15: Menu de ajuda do gerador

5 Testes

Nesta secção irão ser abordados os testes unitários que foram feitos a ambos os projetos recorrendo à *framework* de testes *JUnit*.

5.1 *Demo 1*

Numa primeira fase foram escritos alguns testes manualmente de forma a verificar se a aplicação funciona corretamente. Podemos ver no excerto seguinte um exemplo de um teste ao método *distanceBetweenPoints* da classe *Point*:

```
@Test
public void distanceTest1() {
    Point p1 = new Point(0.0, 1.0);
    assertEquals(1.0, p1.distanceBetweenPoints(new Point(0.0,0.0)));
}
```

Figura 16: Teste unitário à classe *Point*

Após a escrita de alguns testes manuais chegou-se à conclusão que esta tarefe além de ser muito demorada é bastante complexa pois as classes começam a depender muito umas das outras.

Sendo assim, recorreu-se à ferramenta de geração de testes automáticos *EvoSuite* que se foca em tentar cobrir todos os caminhos possíveis de cada método.

5.1.1 Cobertura de Testes

Para analisar a cobertura de testes, tanto dos escritos manualmente como dos gerados automaticamente utilizaou-se a ferramenta embutida no *IntelliJ*.

Após a utlização desta ferramenta chegou-se à conclusao que, após uma breve utilização da aplicação, 68% das classes foram testadas e 45% das linhas de código atingidas durante esses testes.

Em média o númeo de classes abrangidas era de 70% em cada uma das execuções com cobertura de testes e cerca 60% o número de linhas cobertas.

5.2 *Demo 2*

Tal como no projeto anterior, foi utilizado o*JUnit* para escrita de testes manuais bem como para a geração automática utilizado o *EvoSuite*. Na figura seguite podemos ver um exemplo de

um teste escrito manualmente:

```
@Test
void testEquals() {
    Utilizador u1 = new Utilizador();
    Utilizador u2 = new Utilizador("Roger Bro", "123", "pi@pi.pt",
                                   "1111","avenida de braga",
                                   new GregorianCalendar());
    assertFalse(u1.equals(u2));
}
```

Figura 17: Teste unitário à classe *Point*

5.2.1 Cobertura energética

Quanto à cobertura de testes, em média 83% das classes são testadas. Dentro destas classes 53% das linhas são cobertas. Tal como no projeto anterior, esta cbertura está dependente das ações do utilizador na aplicação.

5.3 *EvoSuite*

Para ambos os projetos foi utlizado o *EvoSuite* de forma a gerar testes automáticos não dependetes de contexto. Todo o foco é a tentativa de cobertura máxima de caminhos.

6 Análise energética

Para uma análise final de cada projeto, foi exigido medir o consumo energético assim como o tempo de execução de cada um, utilizando *logs* resultantes do gerador criado na Tarefa 3. Isto, tanto para o código original como para o código após *refactor*. Por isso, 4 *logs* foram gerados:

- Log0: 8000 utilizadores e por volta de 2000 alugueres e viaturas;
- Log1: 12000 utilizadores e por volta de 2000 alugueres e viaturas;
- Log2: 8000 utilizadores, 4000 alugueres e 2000 viaturas;
- Log3: 8000 utilizadores, 2000 alugueres e 4000 viaturas;

O *Log0* serve apenas como uma base de análise, sendo que os restantes são os mais importante para ver que partes da estrutura de dados do programa é mais afetada pela quantidade de dados adicionado, sendo que depois, na existência de *smells* nessa estrutura, poderemos fazer uma análise mais detalhada em relação a quão eficaz este foi na diminuição do consumo e tempo de processamento.

6.1 Demo 1

	log0	log1	log2	log3
demo1 (com smells)	tempo: 1160 ms energia dram: 1.30 J energia cpu: 23.64 J energia package: 38.01 J	tempo: 1212 ms energia dram: 1.37 J energia cpu: 21.80 J energia package: 27.63 J	tempo: 2123 ms energia dram: 2.39 J energia cpu: 39.20 J energia package: 49.41 J	tempo: 2435 ms energia dram: 2.77 J energia cpu: 37.76 J energia package:67.89 J
demo2 (sem smells)	tempo: 1238 ms energia dram: 1.43 J energia cpu: 22.33 J energia package: 28.30 J	tempo: 1576 ms energia dram: 1.77 J energia cpu: 25.04 J energia package: 32.61 J	tempo: 2205 ms energia dram: 2.29 J energia cpu: 34.46 J energia package: 44.98 J	tempo: 3024 ms energia dram: 3.47 J energia cpu: 43.60 J energia package: 58.13 J

Tabela 1: Projecto demo 1, com e sem *smells*, e respetivos desempenhos perante os diferentes ficheiros de *input*.

Analisando a tabela apresentada é de notar que nos quatro casos apresentados o tempo de execução aumentou na versão *refactored* do código. Considerando que praticamente todos os smells deste projeto (Demo 1) são *red smells* e que a maioria deles foram corrigidos seria de esperar uma leve diminuição do custo energético, o que se verifica em alguns dos casos apresentados. O aumento do tempo de execução pode ser explicado pelo facto de os ciclos-for serem mais lentos do que *streams*.

6.2 Demo 2

	log0	log1	log2	log3
demo2 (com smells)	tempo: 6835 ms energia dram: 8.95 J energia cpu:84.26 J energia package: 117.92 J	tempo: 7941 ms energia dram: 11.43 J energia cpu: 120.58 J energia package: 160.11 J	tempo: 20972 ms energia dram: 27.68 J energia cpu: 267.77 J energia package: 366.13 J	tempo: 9474 ms energia dram: 12.07 J energia cpu: 108.89 J energia package: 226.41 J
demo2 (sem smells)	tempo: 6717 ms energia dram: 11.21 J energia cpu:100.07 J energia package: 117.92 J	tempo: 6484 ms energia dram: 9.57 J energia cpu: 83.26 J energia package: 115.56 J	tempo: 21400 ms energia dram: 30.66 J energia cpu: 263.48 J energia package: 374.42 J	tempo: 6933 ms energia dram: 10.81 J energia cpu: 93.42 J energia package: 128.21 J

Tabela 2: Projecto demo 2, com e sem *smells*, e respetivos desempenhos perante os diferentes ficheiros de *input*.

Analisando a tabela verificamos que apenas nos *logs 1 e 3* se verifica uma diferença de desempenho notável, talvez devido à grande quantidade de dados, situação mais propença a beneficiar de melhorias de código para maior *performance*. O acréscimo de um maior número alugueres demonstra não melhorar o desempenho - pelo contrário, parece diminuir -, mas a variação foi Tao pouca que se pode descartar a mudança no comportamento. Por fim, podemos concluir que será então nos *logs 1 e 3* onde estão localizados *code smells* responsáveis por diminuir a *performance*. Estes *code smells*, com base numa tabela descrita anteriormente, foi apenas 1 identificado e corrigido que podia ter impacto neste tipo de análise do projeto.

7 Conclusão

Neste projecto foi possível analisar duas soluções programadas em Java que pretendiam solucionar o mesmo problema. Analisando o código dos dois projectos é notório que existe uma diferença substancial entre a capacidade técnica dos grupos que programaram estes projectos. Não obstante, ambos os grupos apresentam algumas erros comuns que pensamos surgirem da pouca experiência dos alunos. Ambos os projectos revelam incapacidade de abstracção de padrões repetidos bem como incapacidade de redução de problemas em problemas menores. Estas falhas aparecem no código através de alguns padrões: *switch cases* demasiado extensos, métodos com muita lógica associada, poucos métodos auxiliares, e classes com muitas variáveis de instância.

Alguns *energy smells* também se encontram presentes nos dois projectos com uma exceção notável. A *demo1* utiliza, de forma excessiva, *streams*. Além de serem um *energy smell*, está decisão levou a código muito pouco legível. Outros *red smells* bastante predominantes em ambos os projectos, e com um custo elevado, tratam-se de: utilização de *getters* e *setters*, uso intensivo de exceções, e a não reutilização dos objectos. Apesar de serem *red smells*, estes denotam um certo grau de imaturidade no paradigma orientado a objectos, sendo, por exempo, as exceções utilizadas para *control flow*. Apesar de todos estes *smells* poderem ser *refactored*, alguns encontram-se tão intrínsecos nos projectos que o seu *refactor* implicaria reescrever a aplicação de raiz.

Esta análise permitiu abordar falhas que poderem surgir em projectos de maior escala, com consequências para o cliente. Existe todo um espectro de problemas que os clientes podem levantar: *performance*, custo energético, elegância de código, e organização de código. Quando os projectos já têm alguma dimensão o *refactor* do código torna-se uma tarefa morosa e extremamente penosa, sendo fácil criar uma dívida técnica.