

UmCarroJá: Análise e Teste de Software

Henrique Faria A82200

Departamento de Informática, Universidade do Minho

Resumo Neste trabalho propusemo-nos a Analizar e testar o software feito no ambito da disciplina de *Programação Orientada a Objetos*. Este relatório encontra-se estruturado em 4 secções: *Qualidade do Código Fonte*, *Refactoring da Aplicação*, *Teste da Aplicação* e *Análise de Desempenho da Aplicação*. Foram também utilizadas as seguintes ferramentas para realizar o trabalho: Eclipse, SonarQube, JStanley

Palavras-chave: Code Smells · Technical debt

1 Glossário

Code smells não são bugs e também não estão tecnicamente incorretos. No entanto estes indicam fraquezas no design de uma aplicação que podem comprometerla quer diminuindo o progresso do desenvolvimento da mesma quer provocando bugs ou falhas no futuro. Maus code smells podem provocar resultados adversos aos que se pretendem na aplicação conhecidos como technical debt.

Technical debt é um conceito de software que reflete o custo adicional implicito de modificação de código futuro como consequência da utilização de uma solução limitada mas facil de implementar em vez de implementar uma um pouco mais trabalhosa mas que não demande refazer ou reimplementar código no futuro.

2 Qualidade do Código Fonte

2.1 SonarQube

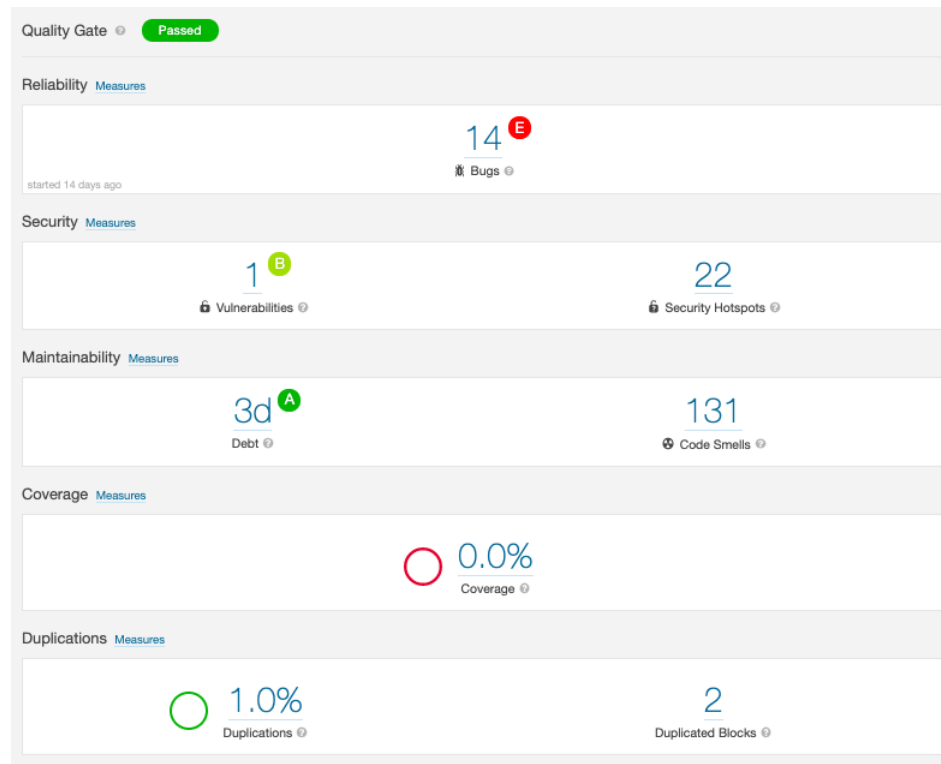


Figura 1. Menu geral de avaliação do SonarQube

Como se pode ver pela figura 1 este projeto possui alguns bugs, pelo menos 1 vulnerabilidade uma quantidade consideravel de code smells e 2 blocos duplicados.

De seguida apresenta-se um relatório detalhado dos tipos de erros e da sua gravidade:

2.2 Bugs

versão com bugs, sem bugs, com smells sem smells (por tipos de smells) -> discriminar o impacto dos smells

Blocker Bugs:

- Não usar blocos try/catch ao escrever em ficheiros.
Ficheiro: UMCarroJa.java

Podemos visualizar nas seguintes imagens o código analisado e a solução respetiva.

```
public void save(String fName) throws IOException {  
    FileOutputStream a = new FileOutputStream(fName);  
    ObjectOutputStream r = new ObjectOutputStream(a);  
    r.writeObject(this);  
    r.flush();  
    r.close();  
}
```

Figura 2. Código com bug

```
public void save(String fName) throws IOException {  
    FileOutputStream a = new FileOutputStream(fName);  
    ObjectOutputStream r = null;  
    try {  
        r = new ObjectOutputStream(a);  
        r.writeObject(this);  
        r.flush();  
    } catch (Exception e) {  
        System.out.println("Can't write to file!!\n");  
    } finally {  
        r.close();  
    }  
}
```

Figura 3. Código corrigido

- Não usar blocos try/catch ao ler de ficheiros.
Ficheiro: UMCarroJa.java

Podemos visualizar nas seguintes imagens o código analisado e a solução respetiva.

```
public static UMCarroJa read(String fName) throws IOException, ClassNotFoundException {  
    FileInputStream r = new FileInputStream(fName);  
    ObjectInputStream a = new ObjectInputStream(r);  
    UMCarroJa u = (UMCarroJa) a.readObject();  
    a.close();  
    return u;  
}
```

Figura 4. Código com bug

```

public static UMCarroJa read(String fName) throws IOException, ClassNotFoundException {
    FileInputStream r = new FileInputStream(fName);
    ObjectInputStream a = null;
    UMCarroJa u = null;
    try{
        a = new ObjectInputStream(r);
        u = (UMCarroJa) a.readObject();
    } catch(Exception e) {
        System.out.println("Can't read the specified file!!\n");
    } finally {
        a.close();
    }
    return u;
}

```

Figura 5. Código corrigido

Note-se que é usado no fim dos blocos *try/catch* o bloco *finally* para fechar o descritor de escrita/leitura. Isto serve para, caso alguma coisa corra mal na escrita em/leitura de um ficheiro, o descritor ser fechado.

Critical Bugs:

- Guardar e reutilizar variáveis random.
Ficheiro: Traffic.java

Para resolver o problema basta verificar que o random estava a ser gerado sempre que a função *getTrafficDelay()* era invocada. Para resolver basta gerar o random uma unica vez quando a classe for criada e usar o mesmo sempre que a função em causa for invocada.

```

class Traffic {
    Random b = new Random();
    public double getTrafficDelay(double delay) {
        int a = LocalDateTime.now().getHour();
        Random b = new Random();
        if(a == 18 || a == 8)
            return (b.nextDouble() % 0.6) + (delay % 0.2);
        if(a > 1 && a < 6)
            return (b.nextDouble() % 0.1) + (delay % 0.2);
        return (b.nextDouble() % 0.3) + (delay % 0.2);
    }
}

```

Figura 6. Recolocação do método de geração de um número Random

Major Bugs:

- Não obrigar a usar o método redefinido usando override.
Ficheiro: Car.java

.....Rever.....

Para resolver este problema fo preciso renomear o método equals para o método isEqual, visto que usando um *@Override* este método obrigaria a implementar um método de super tipo.

```
public boolean isEqual(CarType a) {
    return a == this || a == any;
}
```

Figura 7. Definição da função isEqual

Minor Bugs:

- Obrigar o override do equals e não o do método hashCode().
Ficheiros: Car.java, Cars.java, Cliente.java, Owner.java, Parser.java, Rental.java, Rentals.java, User.java, Users.java.

Para corrigir este problema, basta definir um hashCode() que chame super.hashCode() como se pode ver na figura seguinte.

```
@Override
public int hashCode() {
    return super.hashCode();
}
```

Figura 8. Solução do problema de Override do método hashCode

2.3 Vulnerabilitys

- Utilizar printStackTrace() pode revelar informação sensível.
Ficheiro: Parser.java

2.4 CodeSmells

Blocker CodeSmells:

Critical CodeSmells:

- Possuir um método complexo com cerca de de 290 linhas, é muito difícil manter e até mesmo perceber um código tão extenso.
Ficheiro: Controller.java

Para resolver o problema cada case do switch foi dividido em 1 função de complexidade inferior de média 15 linhas. podemos ver nas duas figuras em

baixo, um dos cases e a função para o qual foi passado o código correspondente.

```

case Login:
    error = caseLogin();
    break;

public String caseLogin() {
    String error = "";
    try {
        NewLogin r = menu.newLogin(error);
        user = model.login(r.getUser(), r.getPassword());
        menu.selectOption((user instanceof Client)? Menu.MenuInd.Client : Menu.MenuInd.Owner);
        error = "";
    }
    catch (InvalidUserException e){ error = "Invalid Username"; }
    catch (WrongPasswordException e){ error = "Invalid Password"; }
    return error;
}

```

Figura 9. Solução do problema de complexidade extrema do método run()

- Repetir várias vezes a atribuição da mesma string pode tornar o código confuso e ineficiente.

Para ultrapassar essa dificuldade essa string passou a ser criada e guardada numa constante quando um objeto da classe é inicializado e essa constante é depois atribuída quando necessário.

Ficheiros: Controller.java, Rental.java, Weather.java e Menu.java

- Usar switch sem caso *default*:. Bastou substituir o ultimo *case "..."*: por *default*: para cumprir o mesmo objetivo do código anterior. No caso do ficheiro Menu.java o default foi adicionado após todos os cases existentes para garantir que o programa corria da forma correta. Ficheiros: Controller.java, Car.java, Parser.java e Menu.java
- Ter constantes numa enumeração escritas em letras minúsculas. Basta escrevê-las em maiúsculas para que passem a seguir a convenção. Para além disso todas as ocorrências destas palavras sofrerão a mesma modificação. Ficheiros: Car.java e Menu.java
- Atualizar uma variável static através de um método não estático.
Ficheiro: Rentals.java

Para contornar este problema, transformou-se a variável *static private int id* em *private int id*.

Major CodeSmells:

- A existência de blocos try/catch vazios não afeta o desempenho da aplicação mas pode ser um erro de falta de código, neste caso, assumimos que foi deixado propositadamente vazio, logo para não alterar o código restante preencheu-se o bloco com a atribuição de uma String vazia á variavel error. Ficheiros: Controller.java e Main.java

```
catch (UnknownCompareTypeException ignored) {error = "";}}
```

Figura 10. Preenchimento do bloco vazio com uma linha de código que não afete a aplicação

- A utilização de uma classe que reporta uma exceção e não estende a class *Exception* viola a convenção, para corrigir o erro mudou-se a expressão *extends Throwable* para *extends Exception*.
Ficheiros: InvalidNewRentalException.java e InvalidNumberOfArgumentsException.java

O código corrigido é apresentado na seguinte figura.

```
public class InvalidNewRentalException extends Exception {
    private static final long serialVersionUID = 4378462538950802892L;
}
```

Figura 11. Troca de Throwable por Exception

- A utilização de uma classe que reporta uma exceção e não estende a class *Exception* viola a convenção, para corrigir o erro mudou-se a expressão *extends Throwable* para *extends Exception*.
Ficheiros: InvalidNewRentalException.java e InvalidNumberOfArgumentsException.java

O código corrigido é apresentado na seguinte imagem.

```
public class InvalidNewRentalException extends Exception {
    private static final long serialVersionUID = 4378462538950802892L;
}
```

Figura 12. Troca de Throwable por Exception

- Ao reportar algo o utilizador deve ser capaz de aceder aos logs facilmente, estes logs têm de ter um formato uniforme, devem ser guardados e dados sensíveis

devem ser guardados de forma segura. A utilização de `System.out.println()` pode comprometer um destes aspetos. Portanto tivemos de importar a biblioteca `-textitjava.util.logging.Logger` e criar um *Logger* para guardar o output disponibilizado ao utilizador.

Ficheiros: `Main.java` e `Menu.java`

O código com o `Logger` é apresentado na seguinte imagem.

```
import java.util.logging.Logger;

public class Main {
    private final static Logger LOGGER = Logger.getLogger(Main.class.getName());
    public static void main(String[] args) {
        UMCarroJa model = new UMCarroJa();

        try {
            model = UMCarroJa.read(".tmp");
            LOGGER.info("adasdsada1");
        }
        catch (IOException | ClassNotFoundException e) {
            LOGGER.info("adasdsada2");
            new Parser("db/logsP00_carregamentoInicial.bak", model);
        }
        try { Thread.sleep(10000);} catch (Exception e) {}
        new Controller(model).run();
        try {
            model.save(".tmp");
        }
        catch (IOException ignored) {}
    }
}
```

Figura 13. Uso de `Logger` para seguir requerimentos dos outputs

- A utilização de muito parâmetros pode significar que esta classe está a fazer muitas coisas.

Ficheiros: `Car.java` e `RegisterCar.java`

Achamos por bem para já não alterar este codeSmell, pois a função recebe 9 argumentos quando só devia receber no máximo 7. O excesso de parâmetros não prejudica o desempenho de forma alguma.

- A presença de métodos que não são utilizados (dead code) deve ser removida.

Ficheiros: `Point.java` e `StringBetter.java`

Na figura em baixo podemos ver um exemplo dos métodos removidos. Na classe `StringBetter.java` removeu-se o método `setStr()`.


```
private Double getX() {
    return this.x;
}

private Double getY() {
    return y;
}
```

Figura 14. Métodos removidos da classe Point.java

- A presença de uma variável com o mesmo nome da classe em que se insere pode confundir, as boas práticas indicam que devem ser nomes distintos. Portanto a variável *menu* foi renomeada para *mymenu*.
Ficheiro: Menu.java

Minor CodeSmells:

- Nomes de pacotes com letras maiúsculas.
Ficheiros: todos os packages exceto o main.java.

Para resolver este problema basta substituir as letras maiúsculas nos nomes dos packages por minúsculas, o eclipse trata de renomear o nome do package nas declarações feitas dentro dos ficheiros do próprio package.

Em seguida temos um exemplo da correção de uma dessas ocorrências.

```
package Controller;|
package controller;
```

Figura 15. Solução do problema de renomeação de packages

- Utilização de métodos ineficientemente.
Ficheiros: Controller.java e Menu.java.

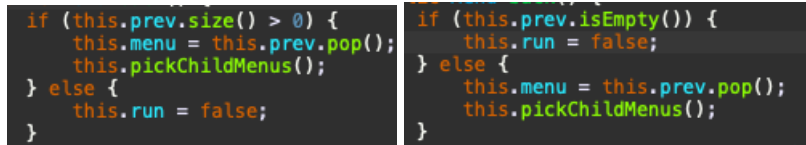
Neste caso num if é utilizado o método `size()` do `ArrayList` e seguidamente verifica-se se o array está vazio. A solução para esta ineficiencia passa por invocar o método `isEmpty()` do `ArrayList`.

Em seguida temos um exemplo da correção de uma dessas ocorrências.

```
if (!lR.size() == 0){
```

Figura 16. Solução do problema de ineficiência no uso de funções de Coleções na classe Controller

Como na classe *Menu.java* foi preciso trocar a execução do `if` com o `else`, para uma melhor percepção do que foi feito, a mudança será mostrada na figura seguinte.



```

if (this.prev.size() > 0) {
    this.menu = this.prev.pop();
    this.pickChildMenus();
} else {
    this.run = false;
}

if (this.prev.isEmpty()) {
    this.run = false;
} else {
    this.menu = this.prev.pop();
    this.pickChildMenus();
}

```

Figura 17. Solução do problema de ineficiência no uso de funções de Coleções na classe Menu

- Classe sem package.
Ficheiro: main.java.

Neste caso basta declarar o package a que a classe pertence, como se pode ver na figura seguinte.



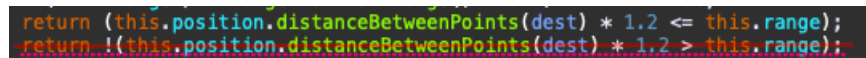
```
package main.java;
```

Figura 18. Declaração do package a que a classe main.java pertence

- Utilização de métodos ineficientemente.
Ficheiro: Car.java.

Para colmatar a ineficiencia do código basta mudar a negação de uma operação de `>`, extremamente custosa, para uma operação de `>=` que tem o mesmo efeito.

A resolução encontra-se na figura seguinte.



```

return (this.position.distanceBetweenPoints(dest) * 1.2 <= this.range);
return !(this.position.distanceBetweenPoints(dest) * 1.2 > this.range);

```

Figura 19. Solução do problema de ineficiência do return

- Declaração de um `clone()` sem `@Override`.
Ficheiros: Car.java, Cars.java, Cliente.java, Owner.java e Point.java.

Para resolver o problema basta adicionar o *@Override* no método, como se pode ver na seguinte figura.

```
@Override
public Car clone() {
    return new Car(this);
}
```

Figura 20. Implementação do método clone() da classe Car

- Método devolve ArrayList em vez de List, dando informação sobre a implementação do método.
Ficheiros: Cars.java e Owner.java.

Para resolver o problema basta por o método a devolver uma interface genérica.

Note-se que as funções que invocam estes métodos devem ser corrigidas declarando o tipo recebido com (*ArrayList<Rental>*).

```
public ArrayList<Car> listOfCarType(Car.CarType b) {
    public List<Car> listOfCarType(Car.CarType b) {
```

Figura 21. Exemplo da mudança do tipo de interface retornada por um método

- Utilização desnecessária de parentesis num filter.
Ficheiros: Cars.java e UmCarroJa.java.

```
.filter((e)-> e.getType().isEqual(b)) .filter(e-> e.getType().isEqual(b))
```

Figura 22. filter sem parentesis

- Declaração de variáveis pela ordem errada, dificultando a leitura do código, estão declaradas da seguinte ordem: static final, final, private. Na figura podemos ver a ordem reescrita da forma correta.
Ficheiro: Rentals.java.

```
class Rentals implements Serializable {
    static private int id;
    private static final long serialVersionUID = 1526373866446179937L;
    private final List<Rental> rentalBase;
```

Figura 23. Ordenação correta das declarações de variáveis

- Declaração de um método em com a primeira letra maiúscula, o nome Original deste era *RESET()*.
Ficheiro: StringBetter.java.

```
private StringBetter reset(){
    return new StringBetter(this.str + "\033{0m");
}
```

Figura 24. Declaração correta do método, usando letra minúscula no início

- Declaração de um método fazendo uso de um underscore no seu nome, seguindo a expressão regular da nomeação de métodos estes não devem ter underscore. De seguida apresenta-se um exemplo de um dos métodos corrigidos.
Ficheiro: StringBetter.java.

```
public StringBetter hideCursor(){
    return new StringBetter(this.str + "\033[?25l");
}
```

Figura 25. Declaração correta do método retirando o underscore e substituindo letra seguinte por maiúscula

- Declaração de variáveis começadas por letra maiúscula.
Ficheiro: NewLogin.java.

Apesar de não ser mostrado, no método NewLogin(), o nome e a password também foram modificado para fazer match com a renomeação demonstrada na figura.

```
private final String user;
private final String password;
```

Figura 26. Declaração correta das variáveis user e password

3 Refactoring da Aplicação

4 Teste da Aplicação

5 Análise de Desempenho da Aplicação

Referências