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 comprometer-la 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

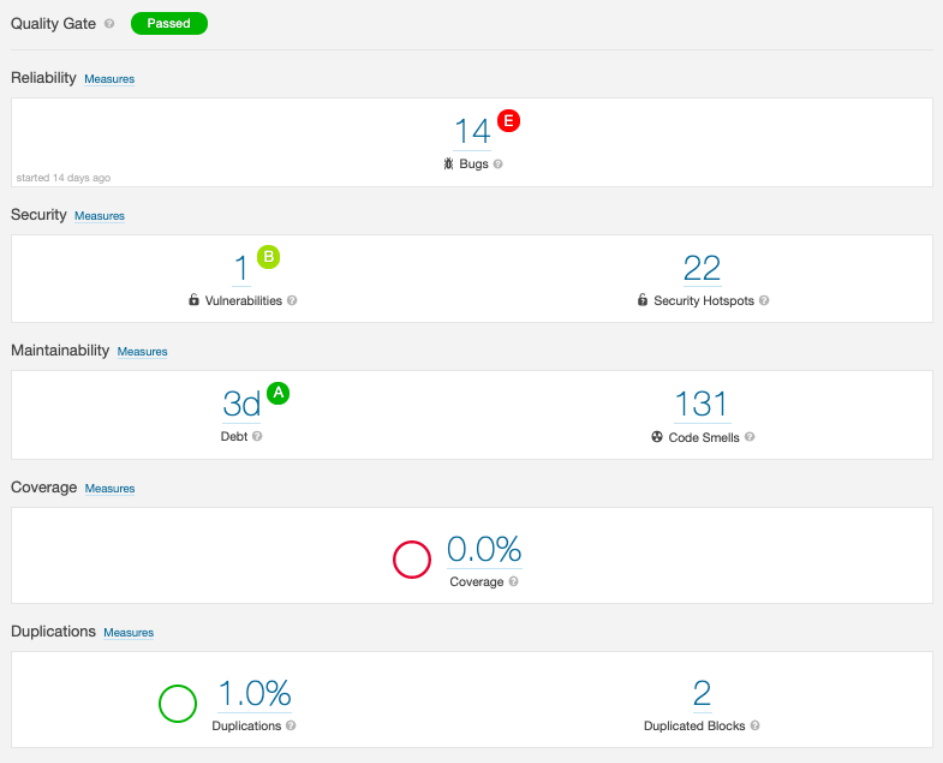

Figura 1. Menu geral de avaliação do SonarQube

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) -> descriminar 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

```java
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 *getTraficDelay()* 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.

```java
class Traffic {
    Random b = new Random();
    public double getTraficDelay(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 redifinido 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 sensivel. Ficheiro:

2.4 CodeSmells

- Blocos de código repetidos.

3 Refactoring da Aplicação

4 Teste da Aplicação

5 Análise de Desempenho da Aplicação

Referências