



Arquiteturas de Software

Trabalho Prático 2 - Refactoring

Grupo :

Luís Braga (A82088)
Luís Martins (A82298)

Braga, Portugal
15 de Janeiro de 2019

Índice

1	Introdução	3
2	Code smell's e Refactoring aplicado	4
2.1	Bloaters	4
2.1.1	Long Method	4
2.2	Change Preventers	10
2.2.1	Shotgun Surgery	10
2.3	Object-Orientation Abusers	12
2.4	Dispensable	13
2.4.1	Comments	13
2.4.2	Duplicated Code	14
2.4.3	Dead Code	16
2.4.4	Speculative Generality	17
2.5	Couplers	19
2.5.1	Feature Envy	19
3	Resultados Observados do Refactoring	21
3.1	Bugs	22
3.2	Code Smell's	22
3.3	Duplications e Duplicated Blocks	23
4	Conclusão	24

Índice de figuras

2.1	Exemplo de <i>Long Method smell</i>	5
2.2	Proposta de <i>Refactoring</i>	6
2.3	Exemplo de <i>Long Method smell</i>	7
2.4	Métodos auxiliares.	8
2.5	Métodos auxiliares.	8
2.6	Método após o <i>Refactoring</i>	9
2.7	Exemplo de <i>shotgun surgery smell</i>	10
2.8	Exemplo de <i>refactor</i>	11
2.9	Exemplo de <i>comments smell</i>	13
2.10	Exemplo de <i>duplicated code smell</i>	14
2.11	Exemplo de <i>duplicated code smell</i>	15
2.12	Proposta de <i>refactor</i>	16
2.13	Exemplo de <i>dead code smell</i>	17
2.14	Exemplo de <i>speculative generality smell</i>	18
2.15	Exemplo de <i>speculative generality smell</i>	18
2.16	Exemplo de <i>feature envy smell</i>	19
2.17	Exemplo de utilização de uma query na classe <i>AtivoFinanceiroDAOConcrete</i>	19
2.18	Métodos extraídos.	20
2.19	Aspecto de uma operação sobre a base de dados.	20
3.1	Métricas observadas antes do <i>Refactoring</i>	21
3.2	Métricas observadas depois do <i>Refactoring</i>	22

1 Introdução

No âmbito da Unidade Curricular de Arquiteturas de Software do perfil Engenharia de Sistemas de Software, foi fornecido um trabalho aleatório relativo ao primeiro trabalho prático elaborado na cadeira, cujo o objetivo passa por analisar o dito trabalho, com o objetivo de identificar os diferentes tipos de **code smell's** e aplicar as técnicas de **Refactoring**.

O objetivo é aprofundar os conhecimentos sobre as áreas anteriormente referidas, que foram aprendidas nas aulas teórico práticas. O resultado final será uma estrutura interna diferente e melhorada, sendo que as funcionalidades mantêm-se iguais.

Assim, no fim do **Refactoring** ser aplicado ao código fonte fornecido, são mostradas um conjunto de métricas que comprovam o efeito positivo que o **Refactoring** teve no código fonte final.

2 Code smell's e Refactoring aplicado

Neste capítulo, os **code smell's** que foram identificados serão divididas pelas diferentes categorias lecionadas nas aulas, sendo que, para cada categoria, serão apresentados exemplos do código fonte e respectivo **Refactoring** aplicado.

2.1 Bloaters

Os **Bloaters** são uma categoria de code smell's que engloba todo o código, método e classe que tenha crescido muito (que contenha muitas linhas de código). Assim, este crescimento necessita de ser evitado, de maneira a que o código fique mais claro e eficiente.

2.1.1 Long Method

Exemplo 1

```

public final void start(){
    new Thread(() -> {
        try{
            setRunning(true);
            while(isRunning()) {
                numberOfStocks = 0;
                numberOfStocksChanges = 0;
                String json = getJson();
                JSONObject obj = new JSONObject(json);
                jsonToAtivosFinanceiros(obj).forEach(ativo -> {
                    numberOfStocks++;
                    addAtivoFinanceiro(ativo);
                });
                if(changed.size() > 0){
                    trading.putAtivosFinanceiros(changed);
                    changed = new LinkedList<>();
                }
                Thread.sleep( millis: 10000);
            }
        } catch (Exception e){
            setRunning(false);
            e.printStackTrace();
        }
    }).start();
}

```

Figura 2.1: Exemplo de *Long Method smell*.

Tal como é possível visualizar na figura 2.1, o método `start` possui um elevado número de linhas e pode ser considerado um método de difícil compreensão, logo trata-se de um *smell* desta categoria e necessita-se de aplicar *Refactoring* sobre ele.

Refactoring

O método de *Refactoring* que foi aplicado para resolver o *smell* foi o **Extract Method**.

Assim, desenvolveram-se 3 novos métodos, de maneira a que o código fique mais legível e perceptível (figura 2.2) e o método original fique mais pequeno.

```

public final void start(){
    new Thread(() -> {
        try{
            setRunning(true);
            atualizaAtivos();
        } catch (Exception e){
            setRunning(false);
            e.printStackTrace();
        }
    }).start();
}

private void atualizaAtivos() throws InterruptedException {
    while(isRunning()) {
        numberOfStocks = numberOfStocksChanges = 0;
        String json = getJson();
        JSONObject obj = new JSONObject(json);
        addAtivos(obj);
        putAtualizacao();
        Thread.sleep( millis: 10000);
    }
}

private void putAtualizacao() {
    if(changed.size() > 0){
        trading.putAtivosFinanceiros(changed);
        changed = new LinkedList<>();
    }
}

private void addAtivos(JSONObject obj) {
    jsonToAtivosFinanceiros(obj).forEach(ativo -> {
        numberOfStocks++;
        addAtivoFinanceiro(ativo);
    });
}

```

Figura 2.2: Proposta de *Refactoring*.

Exemplo 2

```
public void render() {
    layout("MEUS CFDs");
    System.out.println("0. Retroceder");
    printPage(0, cfd);

    if(isUpdated()){
        boolean yes = yesOrNoQuestion("Alguns dos seus contratos foram atualizados\nQuer dar refresh?");
        if(yes){
            mediator.changeView(MEUS_CFDs);
            return ;
        }
    }

    int option = 0;
    boolean optionSelected = false;
    while (!optionSelected){
        String input = scanner.nextLine();
        if(input.matches( regex: "[0-9]+")){
            option = Integer.parseInt(input);
            optionSelected = true;
        }
        else if(input.matches( regex: "[ ]*:[ ]*page[ ]*[0-9]+[ ]*")){
            Pattern pattern = Pattern.compile("[ ]*:[ ]*page[ ]*+([0-9]+)[ ]*");
            Matcher matcher = pattern.matcher(input);
            if(matcher.find()) {
                int pageNumber = Integer.parseInt(matcher.group(1));
                System.out.println("0. Retroceder");
                printPage(pageNumber, cfd);
            }
        }
    }

    if(option > 0 && option <= cfd.size()){
        mediator.changeView(CFD_POSSUIDO, cfd.get(option-1));
    }else if(option == 0){
        mediator.changeView(UTILIZADOR);
    }
    else {
        System.out.println("Não existe esse CFD");
        mediator.changeView(MEUS_CFDs);
    }
}
```

Figura 2.3: Exemplo de *Long Method smell*.

Através da visualização da figura anterior (2.3), é visível que o método `render` é demasiado grande e, portanto, necessita de sofrer *Refactoring*.

Refactoring

A técnica utilizada foi o **Extract Method** e obteve-se os seguintes resultados e métodos auxiliares.


```

private boolean apresentaContratosAtualizados() {
    if(isUpdated()){
        boolean yes = yesOrNoQuestion("Alguns dos seus contratos foram atualizados\nQuer dar refresh?");
        if(yes){
            mediator.changeView(MEUS_CFDS);
            return true;
        }
    }
    return false;
}

private boolean isaCFDOption(int option) { return option > 0 && option <= cfd.size(); }

private int getOption() {
    int option = 0;
    boolean optionSelected = false;
    while (!optionSelected){
        String input = scanner.nextLine();
        if(input.matches( regex: "[0-9]+")){
            option = Integer.parseInt(input);
            optionSelected = true;
        }
        else verifyPage(input);
    }
    return option;
}

private void verifyPage(String input) {
    if(input.matches( regex: "[ ]*:[ ]*page[ ]*[0-9]+[ ]*")){
        Pattern pattern = Pattern.compile("[ ]*:[ ]*page[ ]*+([0-9]+)[ ]*");
        Matcher matcher = pattern.matcher(input);
        if(matcher.find()) {
            int pageNumber = Integer.parseInt(matcher.group(1));
            System.out.println("0. Retroceder");
            printPage(pageNumber, cfd);
        }
    }
}

```

Figura 2.4: Métodos auxiliares.

```

private void nextMenu(int option) {
    if(isaCFDOption(option)){
        mediator.changeView(CFD_POSSUIDO, cfd.get(option - 1));
    }else if(option == 0){
        mediator.changeView(UTILIZADOR);
    }
    else {
        System.out.println("Não existe esse CFD");
        mediator.changeView(MEUS_CFDS);
    }
}

```

Figura 2.5: Métodos auxiliares.

```
public void render() {  
    layout( title: "Meus CFDs");  
    System.out.println("0. Retroceder");  
    printPage( i: 0, cfd);  
  
    if (apresentaContratosAtualizados()) return;  
    int option = getOption();  
    nextMenu(option);  
}
```

Figura 2.6: Método após o *Refactoring*.

Tal como se pode verificar na figura anterior (2.6), o número de linhas do método `render` reduziu consideravelmente.

2.2 Change Preventers

Este tipo de smell's consiste em identificar regiões de código, em que, se fosse mudado algo nessa dada região, seria necessário modificar em muitas outras regiões, o que leva a um custo elevado de manutenção no futuro.

2.2.1 Shotgun Surgery

Exemplo 1

Nos *DAOs* (classes que contém as queries de acesso à base de dados) possuem poucas possibilidades de serem reutilizados sem sofrer várias alterações antes. O que se agrava ainda mais se se mudar o tipo de base de dados a utilizar.

O projecto apresenta uma *interface* que possibilita a utilização de algumas operações sobre uma conexão de uma base de dados, *interface* essa que a *DBConnection* implementa. Contudo, apesar de implementar esta *interface*, os *DAOs* utilizam *SQLExceptions*, o que está dependente de bases de dados *SQL*, e a própria variável que contém a conexão é pouco reutilizável chamando-se *SQLConn*.

```
public void delete(String id) {  
    this.SQLConn.connect();  
  
    try {  
        this.SQLConn.executeUpdate("delete from AtivoFinanceiro where Nome='" + id + "'");  
    } catch (SQLException e) {  
        e.printStackTrace();  
    }  
  
    SQLConn.disconnect();  
}
```

Figura 2.7: Exemplo de *shotgun surgery smell*.

Refactoring

Para melhorar o código nos aspectos referidos mudou-se o nome da variável e alterou-se o tipo de *Exception*, passando de *SQLException* para *Exception*.

Agora, caso o utilizador queira mudar o tipo de base de dados digamos, por exemplo, para *MongoDB* poderia reutilizar o código já desenvolvido e alteraria apenas as queries a utilizar.

```
public void delete(String id) {  
  
    this.Conn.connect();  
  
    try {  
        this.Conn.executeUpdate("delete from AtivoFinanceiro where Nome='" + id + "'");  
    } catch (Exception e) {  
        e.printStackTrace();  
    } finally {  
        Conn.disconnect();  
    }  
  
}
```

Figura 2.8: Exemplo de *refactor*.

2.3 Object-Orientation Abusers

De referir que, infelizmente, o grupo não conseguiu encontrar um *smell* desta categoria e portanto, apenas será explicado o seu conceito.

Os *smell's* pertencentes a esta classe são todos aqueles que não realizem uma aplicação correta dos princípios das linguagens orientadas ao objeto.

2.4 Dispensable

Os smell's desta categoria consistem na eliminação de código que, basicamente, não está a ser usado. Assim, é promovida uma maior eficiência e compreensão do código em questão.

2.4.1 Comments

Exemplo 1

Na seguinte imagem é apresentado um bloco de código que aparenta ser comentário explicativo feito com o intuito de futura geração de *javadoc*. No entanto, este tipo de comentário é inconsistente ao longo do projecto e apenas se encontra nesta classe. Para além disso estes comentários também se encontram incompletos. Assim apesar do seu carácter explicativo acabam por não o ser.

```
public interface DAO<K,T> {  
  
    /**  
     *  
     * @param obj  
     */  
    K put(T obj);  
  
    /**  
     *  
     * @param id  
     */  
    T get(K id);  
  
    /**  
     *  
     * @param id  
     */  
    void delete(K id);  
  
    /**  
     *  
     * @param id  
     * @param obj  
     */  
    void replace(K id, T obj);  
}
```

Figura 2.9: Exemplo de *comments smell*.

Refactoring

O refactoring aplicado resumiu-se à eliminação dos comentários, eliminando assim a inconsistência e tornando o código mais limpo.

2.4.2 Duplicated Code

Exemplo 1

O primeiro exemplo apontado para este tipo de *smell* foi encontrado na classe *AtivoFinanceiroDAOConcrete* com a ajuda da ferramenta de análise estática de código *SonarQube*.

O *smell* consistia na repetição de código relativo à criação do *statement*, que só nesta classe se encontrava cinco vezes repetido.

Este *smell* não está só presente na classe *AtivoFinanceiroDAOConcrete*, mas em todos os DAOs "concretos". Onde em alguns aparece, mais do que as cinco vezes referidas acima. Visto que é um caso idêntico em todos decidiu-se representá-lo apenas por este exemplo.

```
public AtivoFinanceiro get(String id) {  
  
    AtivoFinanceiro a = null;  
    try{  
        SQLConn.connect();  
        Connection conn = SQLConn.getConn();  
        Statement stmt = conn.createStatement();  
        ResultSet rs=stmt.executeQuery( sql: "select * from AtivoFinanceiro where Nome='" + id + "'");  
        if(rs.next()){  
            a = new AtivoFinanceiro(rs.getString( columnLabel: "Nome"),rs.getDouble( columnLabel: "ValorUnit"), type: "") {};  
        }  
        SQLConn.disconnect();  
    }  
    catch (SQLException e ){e.printStackTrace();}  
    return a;  
}  
  
@Override  
public void delete(String id) {  
  
    try {  
        SQLConn.connect();  
        Connection conn = SQLConn.getConn();  
        Statement stmt = conn.createStatement();  
        stmt.executeUpdate( sql: "delete from AtivoFinanceiro where Nome='" + id + "'");  
    }  
    catch (SQLException e){e.printStackTrace();}  
    finally {  
        SQLConn.disconnect();  
    }  
}
```

Figura 2.10: Exemplo de *duplicated code smell*.

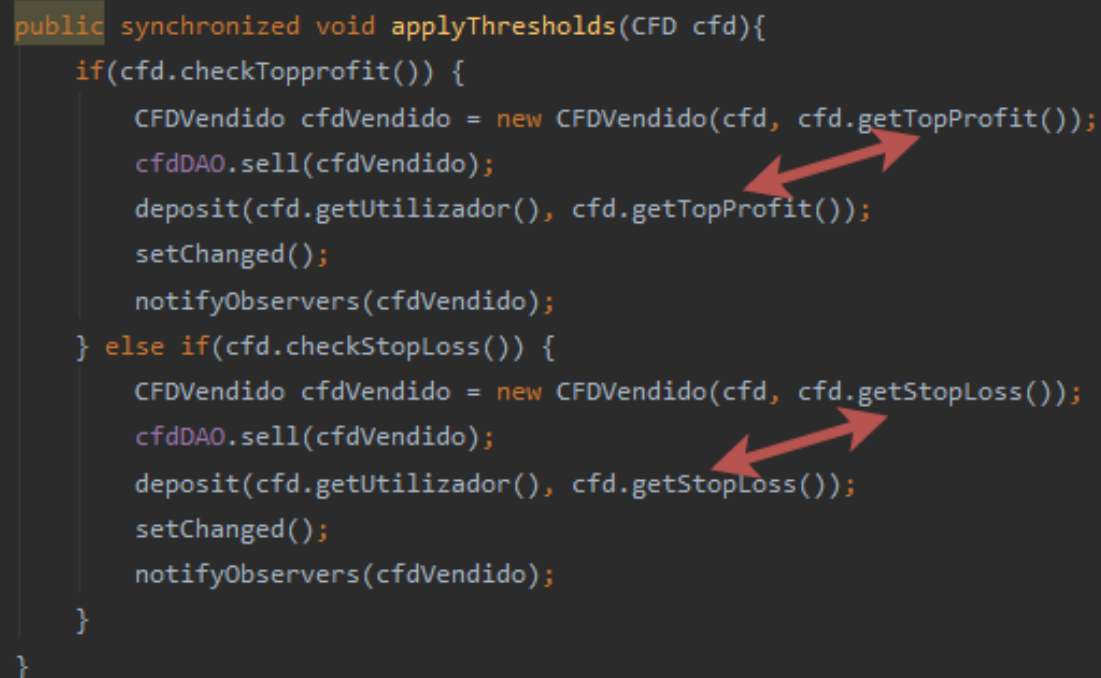
Refactoring

O *refactoring* relativo a este *smell* consistiu na aplicação do **Extract Method**, em que o código rodeado a vermelho na imagem foi transferido para um novo método e substituído em todos os locais onde se encontrava. O `getStatement()` foi criado em todas as classes em que o *smell* estava presente.

Exemplo 2

O seguinte bloco de código idêntico encontrava-se na classe *EESTrading*. O método é responsável pela venda do *CFD* quer o *Take Profit* quer o *StopLoss* seja atingido. A implementação consiste num bloco *if - else*.

De imediato, a semelhança é visível e através de uma breve análise verifica-se que apenas o valor de venda (identificado na imagem) muda de uma condicional para a outra.



```
public synchronized void applyThresholds(CFD cfd){
    if(cfd.checkTopprofit()) {
        CFDVendido cfdVendido = new CFDVendido(cfd, cfd.getTopProfit());
        cfdDAO.sell(cfdVendido);
        deposit(cfd.getUtilizador(), cfd.getTopProfit());
        setChanged();
        notifyObservers(cfdVendido);
    } else if(cfd.checkStopLoss()) {
        CFDVendido cfdVendido = new CFDVendido(cfd, cfd.getStopLoss());
        cfdDAO.sell(cfdVendido);
        deposit(cfd.getUtilizador(), cfd.getStopLoss());
        setChanged();
        notifyObservers(cfdVendido);
    }
}
```

Figura 2.11: Exemplo de *duplicated code smell*.

Refactoring


```

public synchronized void applyThresholds(CFD cfd) {

    if (cfd.checkTopprofit() || cfd.checkStopLoss()) {

        double soldValue = cfd.checkTopprofit() ? cfd.getTopProfit() : cfd.getStopLoss();

        CFDVendido cfdVendido = new CFDVendido(cfd, soldValue);

        cfdDAO.sell(cfdVendido);
        deposit(cfd.getUtilizador(), soldValue);
        setChanged();
        notifyObservers(cfdVendido);
    }
}

```

Figura 2.12: Proposta de *refactor*.

As melhorias aplicadas neste código consistiram em aplicar o método *Consolidate Conditional Expression*. Portanto, juntaram-se as condicionais e extraiu-se o valor que deveria ser usado, com um nome que fosse auto-explicativo (*soldValue*). Assim, o número de linhas de código foi reduzido de 12 para 7 e o mecanismo ficou muito mais perceptível.

2.4.3 Dead Code

O código cedido para análise é bastante rico neste tipo de smell, o que indica que talvez tivessem ocorrido várias mudanças de decisão no que toca à implementação final.

Exemplo 1

O exemplo de *dead code* a ser tratado foi extraído da classe *JSONActionsScraper* na qual se encontravam vários *imports* que não eram realmente utilizados, como se pode ver na figura abaixo.

```

import business.Acao;
import business.AtivoFinanceiro;
import business.CFD;
import business.EESTrading;
import org.json.JSONArray;
import org.json.JSONObject;
import org.jsoup.Jsoup;
import org.jsoup.nodes.Document;
import org.jsoup.nodes.Element;

import java.util.*;

```

Figura 2.13: Exemplo de *dead code smell*.

Refactoring

O *refactoring* aplicado consistiu na simples eliminação destes *imports*. O que facilitou a compreensão desta classe e reduziu o número de linhas de código da mesma.

No entanto, ao longo de todo o projecto são vários os exemplos deste *smell*, entre os quais código morto e código comentado. Visto que são obsoletos foram todos eliminados.

2.4.4 Speculative Generality

Este tipo de *smell* indica que durante o decorrer do projecto pensaram antecipadamente em funcionalidades que nunca realmente chegaram a ser implementadas.

Exemplo 1

O presente exemplo consiste na classe *Indice*, que é uma classe obsoleta. Esta classe estende o *AtivoFinanceiro* e fazia todo o sentido que o fosse, contudo nunca foi utilizada. A utilização desta classe faria sentido se o desenvolvimento da aplicação continuasse.

```

package business;

import business.AtivoFinanceiro;

public class Indice extends AtivoFinanceiro {

    public Indice(String name,double value) { super(name,value, type: "Indice"); }

}

```

Figura 2.14: Exemplo de *speculative generality smell*.

Refactoring

O *refactoring* aplicado reduziu-se à eliminação desta classe.

Exemplo 2

Em semelhança ao *Exemplo 1* o exemplo surge no mesmo conteto, mas desta com uma classe denominada *Ouro*.

```

package business;

import business.AtivoFinanceiro;

public class Ouro extends AtivoFinanceiro {

    public Ouro(String name,double value) { super(name,value, type: "Ouro"); }

}

```

Figura 2.15: Exemplo de *speculative generality smell*.

Refactoring

Em semelhança ao exemplo anterior o *refactoring* empregado foi o mesmo, isto é, eliminação da classe.

2.5 Couplers

Neste grupo de smell's, estão inseridos todos aqueles que promovam demasiado o acoplamento entre classes ou mostra o que acontece quando o acoplamento é substituído por delegação.

2.5.1 Feature Envy

Exemplo 1

Em todos os *DAOs* deste projecto é possível observar um *smell* recorrente no acesso à *DBConnection*.

O problema surge porque, apesar de se ter movido e encapsulado o objecto *Connection* na classe *DBConnection*, os *DAOs* continuam a utilizá-lo. Ou seja, moveram-se as variáveis para uma nova classe, contudo a funcionalidade não foi trasladada.

```
public class AtivoFincanceiroDAOConcrete implements AtivoFinanceiroDAO {
    DBConnection SQLConn = new SQLConnection();

    private Statement getStatement() throws SQLException {
        SQLConn.connect();
        Connection conn = SQLConn.getConn();
        return conn.createStatement();
    }
}
```




Figura 2.16: Exemplo de *feature envy smell*.

```
try{
    Statement stmt = getStatement();
    ResultSet rs = stmt.executeQuery( sql: "select Id f
    while(rs.next()){
        cfd = cfddao.get(rs.getInt( columnlabel: "Id"));
```

Figura 2.17: Exemplo de utilização de uma query na classe *AtivoFinanceiroDAOConcrete*.

Refactoring

De forma a corrigir este *smell* foram aplicados **Extract Method** e **Move Method** em que se criaram os seguintes métodos nas classes *DAO*, que foram posteriormente movidos para a classe *SQLConnection* e as assinaturas foram colocadas na interface *DBConnection*.

```

public Integer executeUpdate(String update) {
    Integer r = 0;
    try {
        r = this.conn.createStatement().executeUpdate(update);
    } catch (SQLException e) {
        e.printStackTrace();
    }

    return r;
}

public ResultSet executeQuery(String query) {
    ResultSet rs = null;

    try {
        rs = this.conn.createStatement().executeQuery(query);
    } catch (SQLException e) {
        e.printStackTrace();
    }

    return rs;
}

```

Figura 2.18: Métodos extraídos.

O que levou a que uma operação realizada num *DAO* adquirisse o seguinte aspecto. Em que o *DAO* apenas sabe a query, mas não sabe como está a ser feita a query internamente, pois agora não é ele que a faz, utilizando a variável de uma classe alheia.

```

public boolean login(String username, String password) {
    boolean ret=false;
    try{
        this.SQLConn.connect();
        ResultSet rs = this.SQLConn.executeQuery("select * from Utilizador where" +
            " Nome='"+username+"' and Password='" + password + "'");
        if(rs.next()){
            ret=true;
        }
    }
    catch (SQLException e){e.printStackTrace();}
    finally {
        SQLConn.disconnect();
    }

    return ret;
}

```

Figura 2.19: Aspecto de uma operação sobre a base de dados.

3 Resultados Observados do Refactoring

Através da realização de algumas métricas realizadas pelo grupo, irá ser demonstrado que, após o **Refactoring** ter sido aplicado, o código tornou-se mais eficiente.

Assim, serão apresentadas algumas das métricas aplicadas pelo grupo e que aprovam a veracidade da afirmação acima referida.

Para calcular as métricas, foi utilizado o [SonarQube](#).

Assim, obteve-se o que se pode visualizar na figura 3.1.

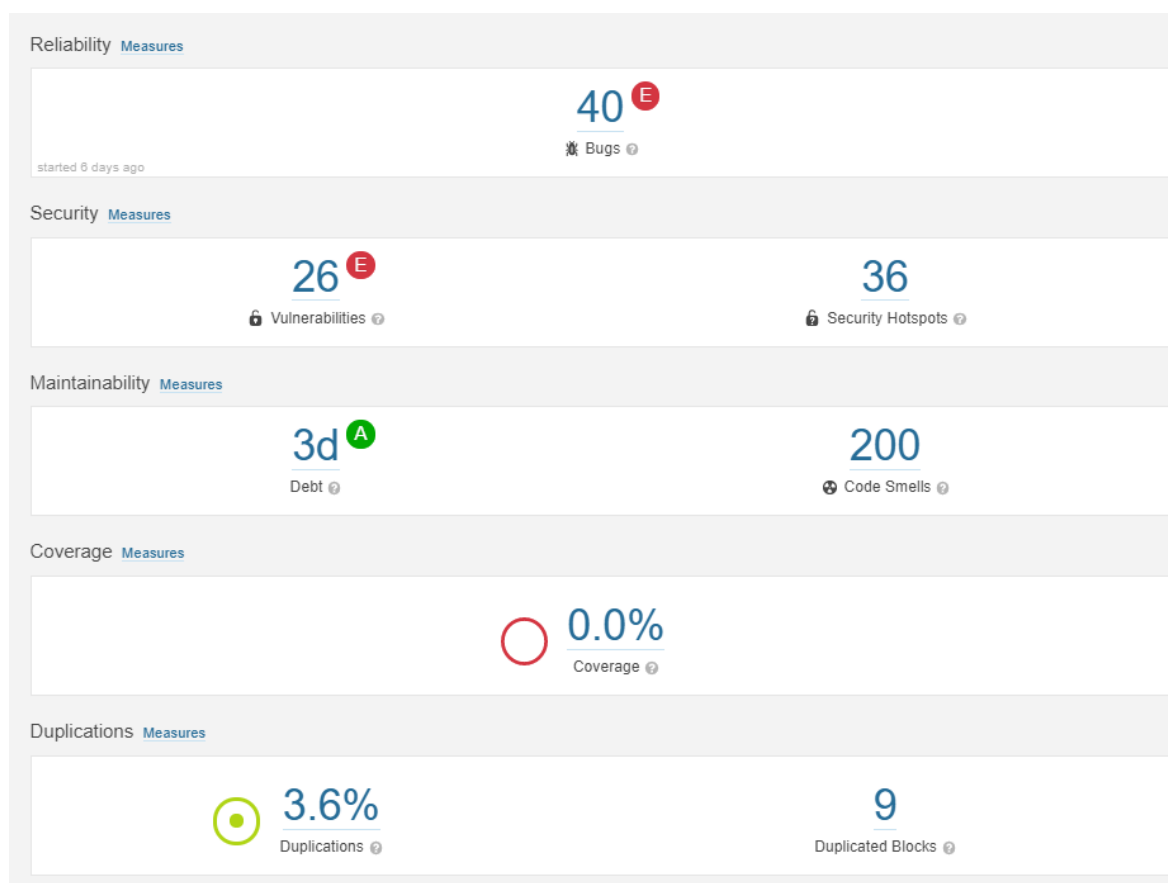


Figura 3.1: Métricas observadas antes do *Refactoring*

Após o Refactoring realizado, obteve-se as seguintes métricas, visivelmente melhoradas (figura 3.2). De referir que não se obteve uma melhoria maior, dado que apenas era para identificar e corrigir alguns exemplos de *smell's* e não, necessariamente, todos.

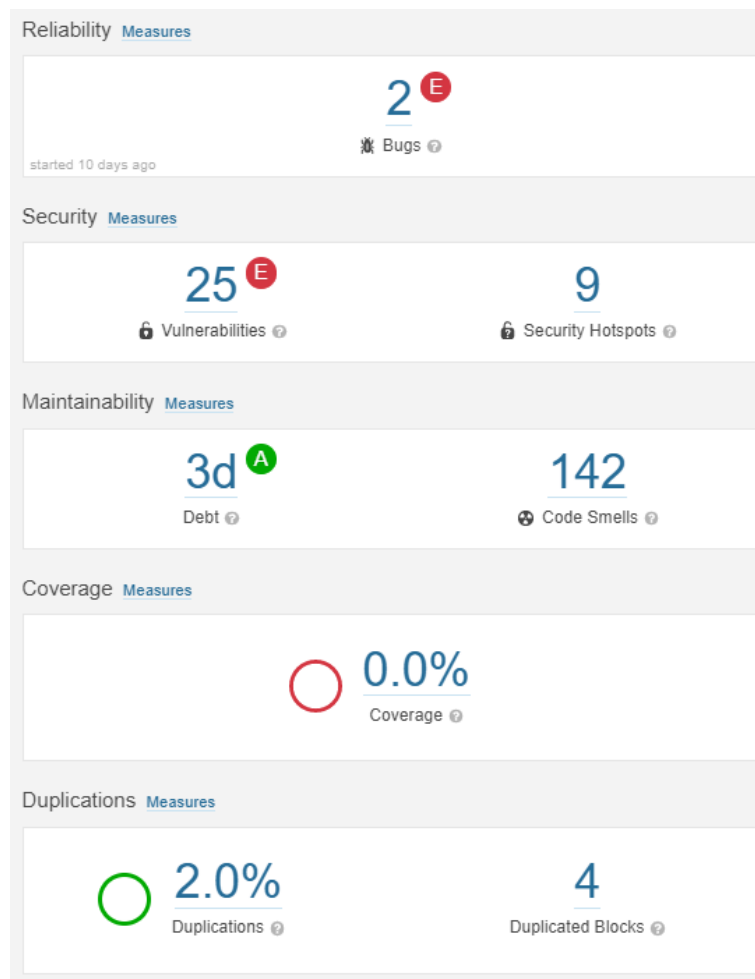


Figura 3.2: Métricas observadas depois do *Refactoring*

Em seguida, são explicadas as métricas que o grupo considerou e a justificação de terem sido consideradas. Além disso, são analisadas as diferenças entre ambas as figuras, segundo as respetivas métricas.

3.1 Bugs

Considerou-se que seria uma boa métrica e importante utilizar, uma vez que a definição de um *Bug* é algo no código que pode alterar o comportamento esperado de um sistema.

No caso do código fonte inicial, podemos dizer que existe um número elevado de bug's, mais precisamente, 40.

Após o *Refactoring* realizado, obteve-se um número de bug's muito mais reduzido, mais concretamente, 2. Assim, foram retirados 38 bug's.

3.2 Code Smell's

Este campo (métrica) é o mais importante, na medida em que estamos o trabalho incide principalmente sobre identificar *code smell's* e corrigi-los.

Inicialmente, no código fonte, existiam 200 *code smell's*.

Efetuada o *Refactoring*, denota-se que o número diminuiu para 142, sendo observável que houve uma correção de 58 *code smell's*.

De referir que se não fosse apenas para mostrar 1 exemplo por cada categoria de *smell*, este número baixaria mais.

3.3 Duplications e Duplicated Blocks

Esta métrica é importante, no entanto, é uma métrica que ao ser reduzida, reduz-se também os *code smell's*, dado que este é um tipo/categoria de *smell*.

O campo *Duplications* é feito a partir do campo *Duplicated Blocks*, uma vez que é a percentagem de duplicações.

Antes do *Refactoring*, 3,6 por cento são duplicações, com 9 blocos duplicados.

Feito o *Refactoring*, podemos verificar que houve uma melhoria significativa, sendo que o número reduziu *Duplicated Blocks* para 4, o que corresponde a 2 por cento a serem *Duplications*.

4 Conclusão

Em primeiro lugar, tal como o professor disse, muitas vezes ter-se-á que deparar com código que não foi feito por nós e, nesse sentido, ter-se-á que perceber o que esse código faz. Por outro lado, poderemos ter que o melhorar, identificar *code smell's* e aplicar técnicas de *Refactoring*, de maneira a que código fique mais compreensível e, na próxima vez, que alguém tiver que olhar para o código, tenha menos dificuldade em perceber o que as classes e os métodos fazem.

Numa fase ulterior à primeira análise, o código fornecido apresentava alguns *bugs* que poderiam comprometer a funcionalidade correta do programa no futuro.

Uma vez dado como terminado o *refactoring*, é possível afirmar que uma grande quantidade ou pelo menos os que mais à vista saltavam eram do tipo *dead code/speculative generality*. O que indica, respectivamente, uma grande quantidade de mudanças num curto espaço de tempo, que resultaria na não limpeza do código, ou enveredar pelas *design desicions* pensadas inicialmente, mas sem nunca chegar realmente a empregá-las. Como reforço desta suposição, temos classes que estão ao mesmo nível da hierarquia, mas que foram, de facto, utilizadas, apesar de pouco desenvolvidas.

Em suma, retira-se que foi muito importante para nós ganhar um pouco dessa experiência de perceber e melhorar código que não foi produzido por nós, na medida em que deverá ser uma situação recorrente, quando estivermos no mundo de trabalho.

Bibliografia

- [1] Source Making,
<https://sourcemaking.com/refactoring>
- [2] SonarQube,
<https://www.sonarqube.org/>