IPS Instituto Politécnico de Setúbal
Escola Superior de Tecnologia de Setúbal

Relatório de Projeto de Programação Avançada

ÉPOCA NORMAL

Licenciatura em Engenharia Informática

ORIENTADOR
Professor Bruno Silva

GRUPO DE TRABALHO
Alexandre Coelho, 190221093
Constantin Cioaca, 180221053
Diogo Merêncio, 180221055
Sérgio Veríssimo, 190221128

Setúbal, janeiro de 2021

Índice

Lista de Figuras

Lista de Tabelas

Lista de Siglas e Acrónimos

ADTs	Abstract Data Types (Tipos abstratos de dados)

1 Introdução

Este projeto consiste numa implementação de teoria de grafos a uma questão que se levantou neste ainda neste século. As redes sociais, constituem-se de uma complexidade exponencial, com o crescimento de utilizadores, com a diversidade nos interesses e com outros fatores com a mesma importância.

Nesta nossa abordagem, fora construída uma rede social com recurso à linguagem de programação Java [1], e com recurso à aprendizagem sobre grafos, abordada na componente teórica desta unidade curricular, que se denomina por Programação Avançada. Esta implementação compõe-se de uma componente visual/gráfica, onde se é possível visualizar e interagir com uma simulação de uma rede social de pequena dimensão.

Esta rede social, constitui-se de utilizadores, declarados como vértices, e relações entre utilizadores, declarados como arestas.

Estes utilizadores são diferenciados por dois tipos diferentes, os utilizadores adicionados, que são adicionados dentro da aplicação diretamente, ou os utilizadores incluídos, que são inseridos na rede social através das relações diretas do respetivo utilizador adicionado. Essas diferenças podem ser visualizadas através das cores distintas que cada utilizador/vértice possui [F1].

As relações também são diferenciadas por três tipos diferentes, as relações diretas normais, que indicam as relações que o utilizador inserido/adicionado contém sem partilha de interesses, as relações diretas com interesses, que indicam as relações que o utilizador inserido/adicionado contém com partilha de interesses, e por fim, as relações indiretas, que indicam as relações que os utilizadores contém, através de interesses, com utilizadores “desconhecidos”.

Todas as tecnicalidades, estarão explicitas nos outros pontos descritos ao longo deste relatório.

2 Descrição dos ADTs implementados

Neste projeto, apenas optamos pela utilização dos tipos abstratos de dados [2] (ADTs), o ADT Digraph e dos seus componentes, aresta (Edge) e vértice (Vertex) e o ADT Graph. Estes ADT's, foram disponibilizados inicialmente para o desenvolvimento do projeto com o intuito de serem aplicados numa implementação com base numa lista de adjacências. Essa implementação com lista de adjacências [F2], requereu uma modificação extensa no comportamento que nos fora dado como exemplo inicial, que se baseava numa lista de arestas. O comportamento dos componentes do ADT Digraph fora modificado, pois neste caso, numa lista de adjacências, os vértices guardam informação sobre as arestas que lhes pertencem, contrariando assim o exemplo inicial, em que as arestas guardariam informação sobre os vértices que lhes pertencem.

ADT Digraph - Este ADT existe, para representar o funcionamento de um grafo orientado/dígrafo. É constituído por vértices (Vertex), que indicam todos os utilizadores implementados na rede social, e também é constituído por arestas, que indicam todas as relações existentes entre os utilizadores da rede social. Compõe-se de métodos que prevalecem a existência de uma orientação, como o incidentEdges, que nos indica todas as arestas que entram (inbound) no vértice que inserimos como parâmetro, ou o método outboundEdges, que nos indica todas as arestas que saem (outbound) do vértice que inserimos como parâmetro. Todos os outros métodos são complementares aos métodos do ADT Graph [3], sendo que existirá sempre a necessidade de fornecer uma direção em certos métodos, como por exemplo no método areAdjacent, precisamos de inserir os vértices de acordo com a direção pretendida, sendo o primeiro vértice parametrizado inserido aquele que tem a aresta a sair (outbound) e o segundo vértice parametrizado inserido aquele que tem a aresta a entrar (inbound).

ADT Graph – Este tipo abstrato de dados, representa um modo de implementação com base em teoria de grafos. Este tipo abstrato de dados constitui-se da mesma forma que o dígrafo, com vértices (Vertex) e arestas (Edge), mas não contém orientação. Este ADT apenas serve para ser reutilizado na implementação do dígrafo, sendo que contém operações básicas sobre o grafo (ou neste caso, a rede social), como por exemplo a inserção de vértices ou de arestas, a remoção de vértices ou de arestas, a substituição de vértices ou arestas, a contagem de número de arestas e de vértices, e consulta de informação sobre os vértices e arestas em forma de coleção. Constitui-se também de métodos que foram utilizados na construção do dígrafo, explicado anteriormente.

3 Diagrama de classes

O nosso diagrama de classes encontra-se dividido em 2 partes: a parte correspondente ao dígrafo e a parte correspondente as outras classes com refatoração. A parte do dígrafo tem o aspeto que se encontra em baixo.

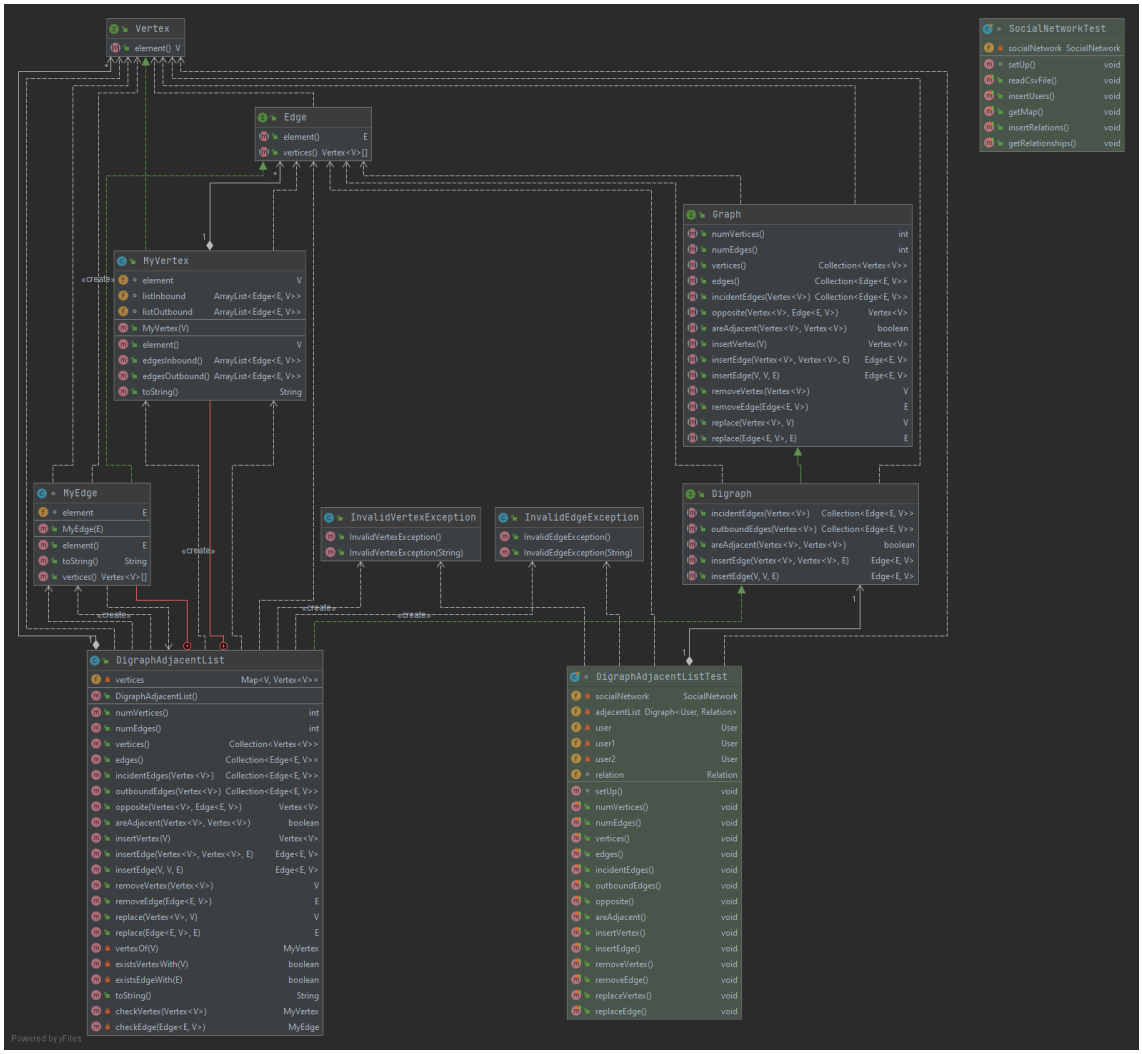

Figura 1 – Diagrama de classes do digrafo

O nosso diagrama de classes, depois de aplicadas todas as devidas técnicas de refatoração [4], encontra-se com o seguinte aspeto.

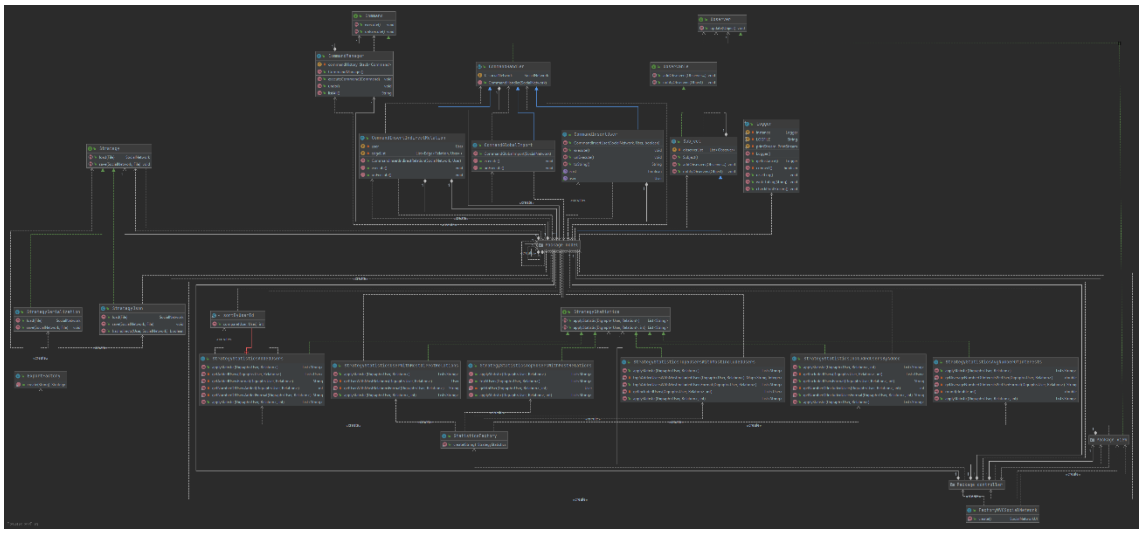

Figura 2 – Diagrama de classes da refatoração

Para consultar as imagens com mais pormenor e detalhe, caso não seja possível, o ficheiro encontra-se embutido nos ficheiros do projeto, mais especificamente na pasta ***relatório***, com os nomes ***Diagrama de Classes - Digrafo.png*** e ***Diagrama de Classes – Refatoração.png***.

4 Documentação de classes

Como possuímos uma documentação de classes e métodos bem explicita através do *JAVADOC*, encontra-se disponível para consultar dentro da respetiva pasta dos ficheiros JAVADOC. Esta que se deve encontrar na pasta ***docs*** incluída no projeto.

5 Padrões implementados

5.1.1 Singleton

O padrão *Singleton* [5] é usado quanto tem de existir exatamente uma instância de uma classe e tem de estar acessível a clientes de um ponto de acesso reconhecido, ou seja, utilizamo-lo para garantir que apenas um objeto exista, independentemente do número de requisições que receber para criá-lo. No nosso caso o *Singleton* [5] é utilizado para guardar os logs que foram feitos, ou seja, quando adicionamos um utilizador de forma iterativa ou até mesmo quando damos a importação global são guardados a data, o id do utilizador adicionado, o id do utilizador incluído e o número de interesses caso existam. Este padrão foi utilizado para permitir ter uma instância única da classe Logger.

5.1.2 Observer

O padrão *Observer* [6] permite que objetos em que o utilizador possua interesse, sejam avisados da mudança de estado ou de outros eventos ocorridos num outro objeto. O padrão é utilizado quando alguns objetos devem observar outros, mas apenas por um tempo limitado ou em casos específicos. No nosso caso, o *Observer* [6] é utilizado para manter o conteúdo da view sempre atualizado, ou seja, sempre que houver alguma alteração no model (*SocialNetwork*) o *Observer* [6] irá notificar a view (*SocialNetworkUI*) e mantê-la atualizada. Este padrão foi utilizado para ao implementar o padrão MVC a view ser notificada qual ocorre alguma modificação.

5.1.3 MVC

O padrão *Model-View-Controller* (MVC) [7] serve para separar as funcionalidades principais do negócio. Esta separação permite partilhar os mesmos dados pelas diferentes vistas, tornando assim, mais fácil as tarefas de desenvolvimento, teste e manutenção para os diferentes clientes.

Participantes do MVC:

Model: O *Model* preocupa-se com os dados persistentes que devem ser apresentados, e com as operações que serão aplicadas para transformar os objetos. Desse jeito, não se preocupa com as interfaces do utilizador, com os dados que serão mostrados e com as ações das interfaces usadas para manipular os dados.

View: A View dispara as operações de consulta do *Model* (via *Controller*) para manipular/obter os dados e visualizá-los, define como os dados serão visualizados pelo utilizador e mantém consistência na apresentação dos dados quando o *Model* muda.

Controller: O *Controller* sincroniza as ações do View com as ações realizadas pelo *Model*, trabalha somente com sinais e não com os dados da aplicação e sabe os meios físicos pelos quais os utilizadores manipulam os dados no *Model*.

No nosso caso o *Controller* (*SocialNetworkController*) vai fazer de "ponte" entre a View e o *Model*. O *Model* (*SocialNetwork*) vai se preocupar com os dados que devem ser apresentados acerca das operações que serão aplicadas para transformar os objetos, e a View (*SocialNetworkUI*) vai definir como os dados serão visualizados pelo utilizador. Este padrão foi utilizado para fazer uma divisão entre a parte gráfica e o backend da aplicação.

5.1.4 Simple Factory

O padrão *Simple Factory* [8] centraliza-se na criação de variantes de objetos em vez de termos inicializado com "*new*", diversas vezes. No nosso caso a *Simple Factory* [8] (*ExportFactory*) serve para criarmos apenas um objeto dependendo da extensão do ficheiro a exportar (.bin e .json).

5.1.5 Strategy

O padrão *Strategy* [9] permite que um objeto troque de estratégia de comportamento em tempo de execução. No nosso caso o padrão *Strategy* [9] é utilizado para guardarmos e carregarmos de duas maneiras diferentes (.bin e .json) e também para conseguirmos reutilizar a estatística com diversos comportamentos (separado por classes), mantendo a singularidade nas classes e também ilegibilidade.

5.1.6 Command

O *Command* [10] serve para lidar com a gestão da execução de comandos (ordens ou operações), como por exemplo: adiar a execução de operações, armazenamento do histórico de operações, definição de macros (definir uma sequência de operações que poderá ser executada várias vezes), associar eventos do utilizador a execução de operações e desfazer operações realizadas. No nosso caso o comando é utilizado para a inserção global, para a inserção dos utilizadores de forma iterativa e para a inserção das relações indiretas.

6 Refactoring

6.1 Bad smells encontrados

Tabela 1 – Tabela com os bad smells encontrados no projeto

Bad smell	Nº de ocorrências	Técnica refactoring aplicada
Duplicate Code	6	Extract Method
Message Chains	4	Hide Delegate
Dead Code/Speculative Generality	21	Delete the code
Long Method	1	Extract Method
Comments	2	Rename Method
Inappropriate Intimacy	7	Hide Delegate
Data class	3	Move Method/Não fazer nada
Large class	1	Extract Class + Extract Interface
Temporary Field	2	Inline Temp
Primitive Obssesion	0	-
Data Clumps	0	-
Refused Bequest	0	-
Switch Statements	0	-

6.1.1 Duplicate Code

A existência do *bad smell* [11] *Duplicate Code* [12], indica que dois ou mais fragmentos de código aparentam ser semelhantes ou até mesmo iguais. No nosso caso, foi detetada uma repetição na criação de alertas visuais em casos específicos após cliques nos vários botões existentes.

```
this.buttonUndo.setOnAction(e -> {
    if(!model.getRelationships().vertices().isEmpty()){
        controller.undo();
    }
    else{
        Alert a = new Alert(Alert.AlertType.ERROR);
        a.setTitle("Undo error!");
        a.setHeaderText("You can't undo an empty Social Network!");
        a.show();
    }
});
```

Figura 3 – Exemplo de código com *Duplicate Code*

Para correção deste *bad smell* [11], foi aplicado o *Extract Method* [13], em que se criou um método externo a este representado na figura acima. Este método engloba a criação de um alerta e generaliza para todos os alertas a partir dos parâmetros que recebe.

```
private Alert createAlert(String title, String headerText, Alert.AlertType alertType){
    Alert a = new Alert(alertType);
    a.setTitle(title);
    a.setHeaderText(headerText);
    return a;
}
```

Figura 4 – Exemplo de código com o método que surgiu após aplicar o *Extract Method*

6.1.2 Long Method

Como a sua nomenclatura indica, *Long Method* [14], indica a existência de um método que é demasiado extenso. Na nossa classe de *SocialNetworkUI* possuíamos um método que tinha 345 linhas.

```
163
164     public void setTriggers(SocialNetworkController controller) {...}
509
```

Figura 5 – Exemplo de um *Long Method*

Após aplicação de *Extract Method* [13] no código duplicado, conseguimos reduzir o número de linhas do método para 273 linhas

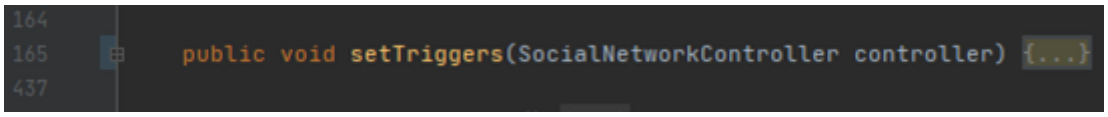

Figura 6 – Exemplo de código após a primeira aplicação do *Extract Method*

Mas como apenas tínhamos aplicado o *Extract Method* [13] para o código duplicado, ainda não tínhamos efetuado para *Long Method*. Após aplicação do *Extract Method* [13] novamente conseguimos reduzir as linhas do método para 47 linhas.

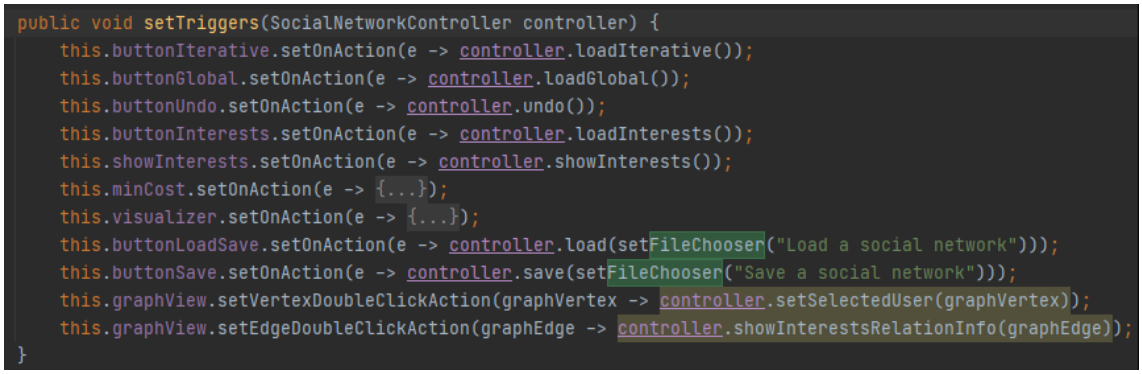

Figura 7 - Exemplo de código após a última aplicação do *Extract Method*

6.1.3 Message Chains

Estas *Message Chains* [15] ocorrem no contexto da necessidade de navegação entre os vários componentes que formam a nossa rede social. Foram construídos para facilitar o acesso às coleções auxiliares que contêm a informação relativamente aos ficheiros, como também para a sua manipulação. Como por exemplo, no código que está presente na figura abaixo, existe a ocorrência de irmos buscar um valor (num mapa, através de uma chave (interesse) diretamente à coleção que está presente noutra classe.

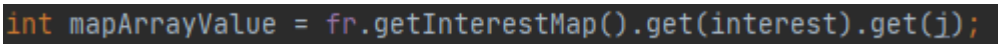

Figura 8 – Exemplo de uma ocorrência de *Message Chains*

Para a resolução deste *bad smell* [11], temos de aplicar a técnica de refatoração *Hide Delegate* [16], de modo a que a coleção não seja acessível. Neste caso específico, seria necessário criar um método que retorne diretamente o valor inteiro, acendo à coleção internamente na classe onde ela é instanciada, como representado na figura abaixo.

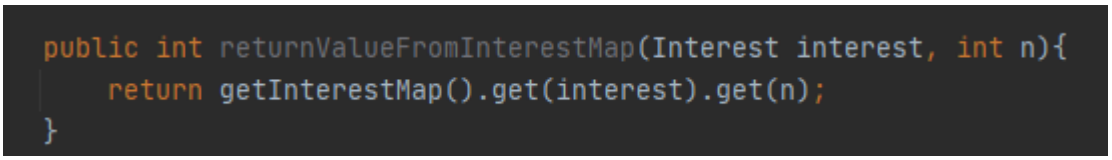

Figura 9 – Exemplo de código da aplicação do *Hide Delegate*

Depois de ser aplicado o *Hide Delegate* [16], o código fica sem as *Message Chains* [15], e fica com este aspeto:

```
int mapArrayValue = fr.returnValueFromInterestMap(interest,j);
```

Figura 10 – Exemplo do resultado após a aplicação do *Hide Delegate*

6.1.4 Dead Code

Quando a este *bad smell* [11], como a sua nomenclatura indica, trata-se de código que é obsoleto para a resolução do problema. Também podemos retratar o *bad smell* [11] de nomenclatura *Speculative Generality* [17], que indica praticamente o mesmo que *Dead Code* [18].

```
/**
 * @param relation
 */
public void removeRelation(Relation relation) {
    if (relation == null) {
        return;
    }
    Edge<Relation, User> removeEdge = null;
    for (Edge<Relation, User> e : relationships.edges()) {
        if (relation.getName() == e.element().getName()) {
            removeEdge = e;
        }
    }
    relationships.removeEdge(removeEdge);
    notifyObservers( obj: this);
}
```

Figura 11 – Exemplo de código de *Dead Code*

Neste caso específico, trata-se de um método que teria sido criado para a possível remoção visual de arestas. Mas essa funcionalidade acabou por não ser opção, por não ser revelante para a nossa abordagem ao projeto. Para tratarmos este *bad smell* [11], é muito simples. Basta eliminar o código.

6.1.5 Comments

Dizemos que existe um *bad smell* [11] de *Comments* [19] quando existem diversos comentários sobre a explicação do método. Encontramos este *bad smell* [11] na nossa classe *StrategyJson*.

```java
@Override
public SocialNetwork load(File file) throws LoadException  {
    List<BackupData> backupData = null;

    try {
        // create a reader
        Reader reader = Files.newBufferedReader(Paths.get(file.getPath()));

        // convert JSON array to list of users
        backupData = new Gson().fromJson(reader, new TypeToken<List<BackupData>>() {
        }.getType());

        // print users
        //backupData.forEach(System.out::println);

        // close reader
        reader.close();

    } catch (Exception ex) {
        ex.printStackTrace();
    }
    SocialNetwork socialNetwork = new SocialNetwork();
    if(backupData == null) throw new LoadException("Can't load the selected file!");
    socialNetwork.loadFromBackup(backupData);

    return socialNetwork;
}
```

Figura 12 – Exemplo de código de com *Comments*

Para resolução deste *bad smell* [11], aplicaríamos o *Rename Method* [20], mas neste caso, o nome do método (*load*) e o nome da classe *StrategyJson*, já seria bastante intuitivo para se saber que aquele método específico, serviria para carregar um ficheiro JSON. Neste caso específico, apenas decidimos remover os comentários.

```java
@Override
public SocialNetwork load(File file) throws LoadException {
    List<BackupData> backupData = null;

    try {
        Reader reader = Files.newBufferedReader(Paths.get(file.getPath()));

        backupData = new Gson().fromJson(reader, new TypeToken<List<BackupData>>() {
        }.getType());

        reader.close();

    } catch (Exception ex) {
        ex.printStackTrace();
    }
    SocialNetwork socialNetwork = new SocialNetwork();
    if(backupData == null) throw new LoadException("Can't load the selected file!");
    socialNetwork.loadFromBackup(backupData);

    return socialNetwork;
}
```

Figura 13 – Exemplo de código sem *Comments*

6.1.6 Inappropriate Intimacy

Embora este método não seja utilizado noutro sítio, como indica este *bad smell* [11], encontra-se instanciado como método público, o que poderá induzir o programador em erro, numa futura atualização.

```java
public Collection<Interest> getListInterests(int userId) {
    List<Interest> listOfInterests = new ArrayList<>();
    for (Interest interest : fr.getInterestMap().keySet()) {
        if (fr.getInterestMap().get(interest).contains(userId)) {
            listOfInterests.add(interest);
        }
    }
    return listOfInterests;
}
```

Figura 14 – Exemplo de código com *Inappropriate Intimacy*

Podemos prevenir este *Inappropriate Intimacy* [21] ao fazer *Hide Delegate* [16], tornando o método privado e exclusivo apenas à classe onde ele se encontra. Esta técnica de refactoring também se pode intitular de *Encapsulate Field* [22] .

```
private Collection<Interest> getListInterests(int userId) {
    List<Interest> listOfInterests = new ArrayList<>();
    for (Interest interest : fr.getInterestMap().keySet()) {
        if (fr.interestMapValuesContainsUser(interest,userId)){
            listOfInterests.add(interest);
        }
    }
    return listOfInterests;
}
```

Figura 15 – Exemplo de código sem *Inappropriate Intimacy*

6.1.7 Data Class

Data Class [23] refere-se ao facto de uma classe possuir apenas *getters* e *setters* aos seus atributos. Este *bad smell* [11] foi verificado nas classes que nós utilizamos para as peças fulcrais da nossa abordagem, sendo que declaramos o nosso vértice como *User* (*classe User*) e a nossa aresta como *Relation* (*classe Relation*), sendo que a *Relation* se apoia noutra classe que só possui *Interest* (classe *Interest*).

Para resolução deste *bad smell* [11], no caso da classe *Interest*, não fazemos nada, pois não representa nenhum perigo. No caso do *Relation* e *User* também não representa perigo, mas optamos por efetuar Move Method, validando os tipos de *User* e *Relation* na classe em si, e não em métodos/linhas de código do *SocialNetwork*.

```
public boolean isIndirect(){
    if(this.getType() == RelationType.INDIRECT){
        return true;
    }
    return false;
}
```

Figura 16 – Exemplo de código antes de ser aplicado o *Move Method*

6.1.8 Large Class

Este *bad smell* [11] fora detetado na nossa classe de *Statistics*, que possuía demasiados métodos e atributos, apenas para efetuar um leque limitado de métodos estatísticos sobre a rede social.

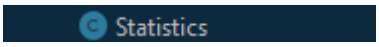

Figura 17 – Existência da nossa classe *Statistics* que implementava as estatísticas

Para resolver este *bad smell* [11] optamos pela implementação de um *Padrão Strategy* [9] para realizarmos as estatísticas. Esta implementação teve como base as técnicas de refactoring *Extract Class* [24] e *Extract Interface* [25].

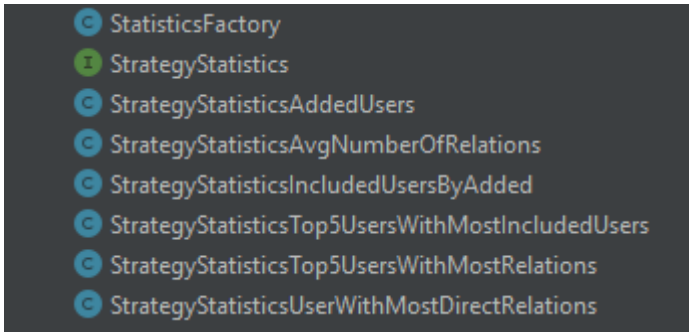

Figura 18 – As estatísticas após a implementação do padrão *Strategy*

6.1.9 Temporary Field

No nosso caso específico, foi encontrado este *bad smell* [11], na classe de *SocialNetworkUI*, onde duas HBox estão instanciadas nos atributos da classe, quando deveríamos estar dentro dos métodos respetivos.

```
private HBox hboxMenuBar;
private HBox hboxIterative;
```

Figura 19 – Exemplo de código de *Temporay Fields*

Para resolvermos este *bad smell* [11] aplicamos a técnica de refactoring de *Inline Temp* [26], passando a variável para dentro do método e a sua inicialização, tornando-a exclusiva ao método onde é utilizada, sendo assim, ficando apenas numa linha de código.

```
/* ITERATIVE */
HBox hboxIterative = new HBox( spacing: 4);
Label labelIterative = new Label( text: "User:");
this.textFieldIterative = new TextField();
this.textFieldIterative.setPrefWidth(40);
this.buttonIterative = new Button( text: "Iterative Import");
hboxIterative.getChildren().addAll(labelIterative, textFieldIterative, buttonIterative);
hboxIterative.setAlignment(Pos.CENTER);
```

Figura 20 – Exemplo de código após aplicar o *Inline Temp*

7 Conclusão

Com este projeto, conseguimos adquirir competências no desenvolvimento de potenciais problemas reais, que possuam uma resolução com base em teoria dos grafos. Embora este projeto esteja relacionado com uma rede social, a sua teoria é aplicável em diversos âmbitos do nosso quotidiano. Podemos aplicar exatamente a mesma implementação a qualquer sistema que se consiga separar por pontos/vértices e ligações/arestas. A única modificação seria nos comportamentos de cada vértice e aresta. Por exemplo, poderíamos ter de arquitetar uma aplicação em Java sobre uma rede de autocarros de qualquer dimensão. Neste problema em concreto, poderíamos definir as paragens de autocarro como vértices e as estradas entre paragens as arestas, e poderíamos definir rotas indiretas que o autocarro poderia ter de optar, caso existissem problemas nas suas rotas diretas. Embora tenhamos aplicado orientação ao nosso grafo, e tenhamos trabalhado com base numa lista de adjacências, os conceitos abordados exteriores a este projeto, mas incluídos na teoria de grafos, foram corretamente absorvidos.

Também conseguimos adquirir conhecimentos sobre a limpeza de código, através de refatoração, que era um conceito desconhecido para nós. A aplicação de padrões e limpeza de code smells, é fundamental na construção/manutenção de um projeto pessoal ou de um projeto empresarial, com o intuito de ser percetível tanto para nós, caso estejamos a rever algo que não nos recordaríamos, ou até mesmo para que outros programadores consigam ter uma visão mais clara da abordagem que escolhíamos para um projeto. Parafraseando uma frase presente no livro *Clean Code: A Handbook of Agile Software Craftmanship de Robert C. Martin* – "*One difference between a smart programmer and a professional programmer is that the professional understands that clarity is king. Professionals use their powers for good and write code that others can understand.*", que indica que um programador profissional, é aquele que utiliza os seus "poderes" para escrever código que seja percetível para outros.

A documentação demonstrou-se também uma prática essencial, que embora já conhecêssemos essa prática, ainda não a implementaríamos automaticamente. Mas com a realização deste projeto, já se incluiu na nossa praticidade, a documentação no código e também documentação externa ao código que explicite o problema e a nossa abordagem.

8 Bibliografia/Net grafia

[1] – Definição de Java na Wikipédia - https://pt.wikipedia.org/wiki/Java_(plataforma_de_software)

[2] – Definição de tipo abstrato de dados de acordo com Wikipédia - https://pt.wikipedia.org/wiki/Tipo_abstrato_de_dado

[5] – Definição de *Singleton* de acordo com o site Refactoring Guru (em inglês) - https://refactoring.guru/design-patterns/singleton

[6] – Definição de *Observer* de acordo com o site Refactoring Guru (em inglês) –

https://refactoring.guru/design-patterns/observer

[7] – Definição de *MVC* segundo a Wikipédia - https://pt.wikipedia.org/wiki/MVC

[8] – Definição de *Simple Factory* segundo a envatotuts+

https://code.tutsplus.com/pt/tutorials/design-patterns-the-simple-factory-pattern--cms-22345

[9] – Definição de *Strategy* segundo o site Refactoring Guru (em inglês) - https://refactoring.guru/design-patterns/strategy

[10] – Definição de *Command* segundo o site Refactoring Guru (em inglês) - https://refactoring.guru/design-patterns/command

[11] – Definição de *Code Smells* de acordo com Martin Fowler, o criador da refatoração de código - https://martinfowler.com/bliki/CodeSmell.html

[12] – Definição de *Duplicate Code* de acordo com o site Refactoring Guru (em inglês) - https://refactoring.guru/smells/duplicate-code

[13] – Definição de *Extract Method* de acordo com o site Refactoring Guru (em inglês) - https://refactoring.guru/extract-method

[14] – Definição de *Long Method* de acordo com o site Refactoring Guru (em inglês) –

https://refactoring.guru/smells/long-method

[15] – Definição de *Message Chains* de acordo com o site Refactoring Guru (em inglês) - https://refactoring.guru/smells/message-chains

[16] – Definição de *Hide Delegate* de acordo com o site Refactoring Guru (em inglês) –

https://refactoring.guru/hide-delegate

[17] – Definição de *Speculative Generality* de acordo com o site Refactoring Guru (em inglês) - https://refactoring.guru/smells/speculative-generality

[18] – Definição de *Dead Code* de acordo com o site Refactoring Guru (em inglês) - https://refactoring.guru/smells/dead-code

[19] – Definição de *Comments* de acordo com o site Refactoring Guru (em inglês) – https://refactoring.guru/smells/comments

[20] – Definição de *Rename Method* de acordo com o site Refactoring Guru (em inglês) - https://refactoring.guru/rename-method

[21] – Definição de *Inappropriate* Intimacy de acordo com o site Refactoring Guru (em inglês) – https://refactoring.guru/smells/inappropriate-intimacy

[22] – Definição de Encapsulate Field de acordo com o site Refactoring Guru (em inglês) - https://refactoring.guru/encapsulate-field

[23] – Definição de *Data Class* de acordo com o site Refactoring Guru (em inglês) - https://refactoring.guru/smells/data-class

[24] – Definição de *Extract Class* de acordo com o site Refactoring Guru (em inglês) –

https://refactoring.guru/extract-class

[25] – Definição de *Extract Interface* de acordo com o site Refactoring Guru (em inglês) - https://refactoring.guru/extract-interface

[26] – Definição de *Inline Temp* de acordo com o site Refactoring Guru (em inglês) - https://refactoring.guru/inline-temp

9 Anexos

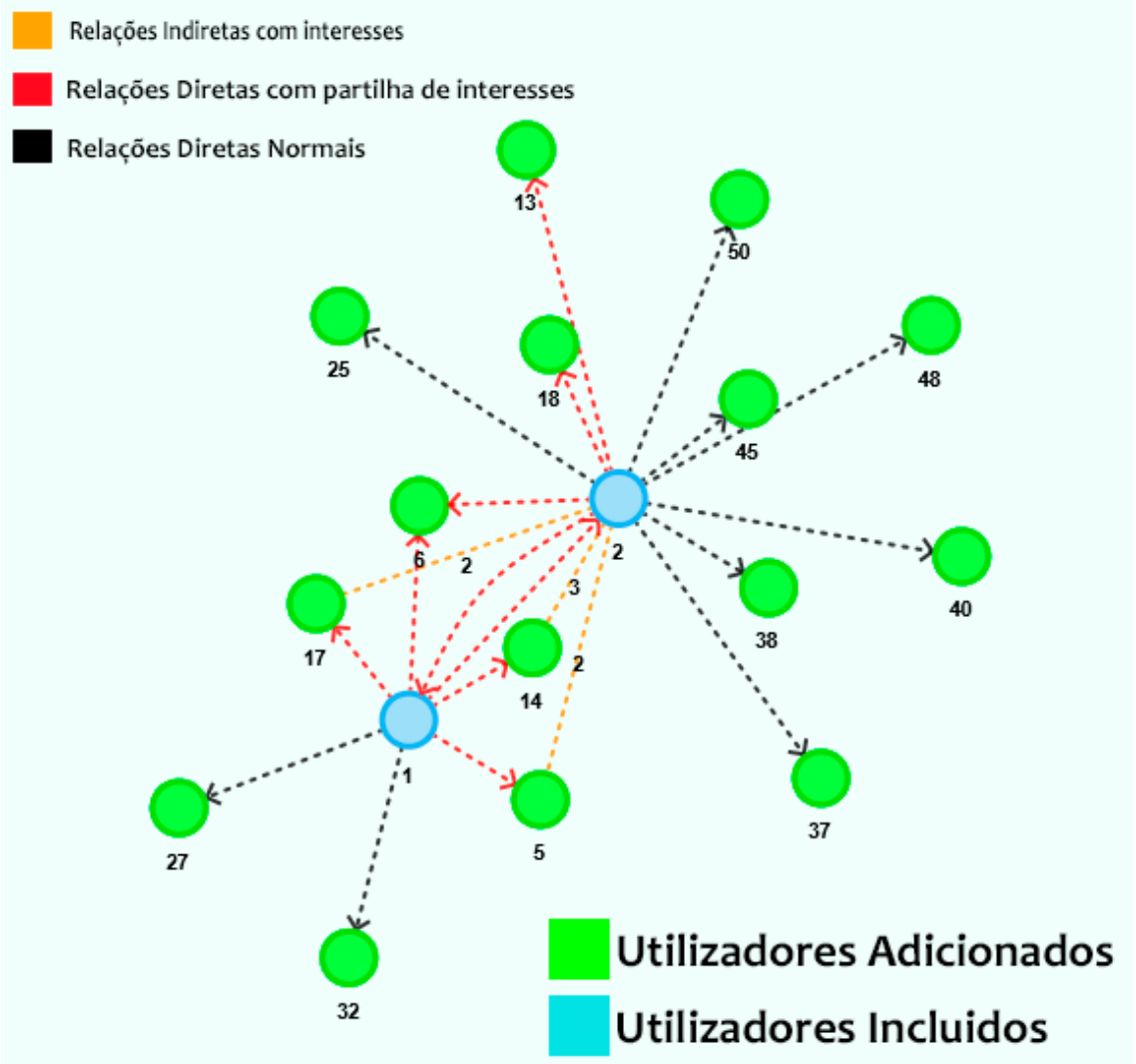

Figura 21 – Legenda do digrafo

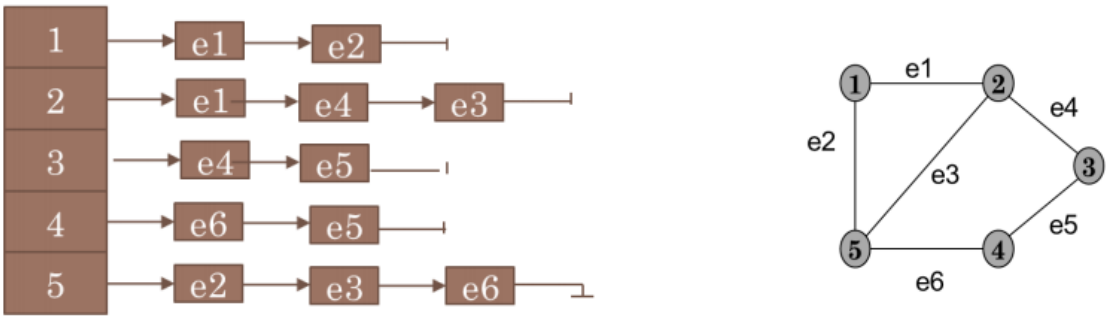

Figura 22 – Lista de adjacências