Instituto Politécnico de Setúbal – Escola Superior de Tecnologia de Setúbal

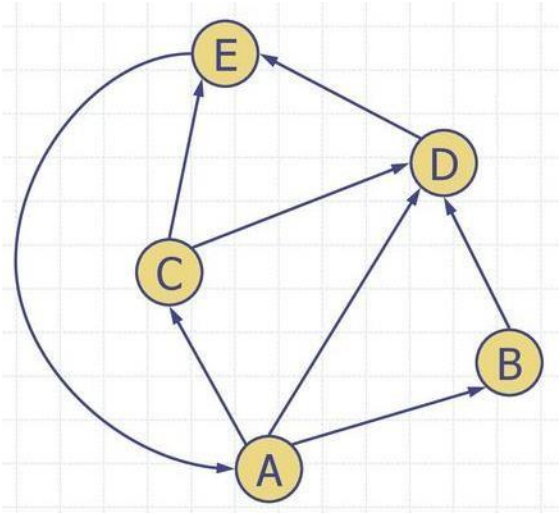
Licenciatura em Engenharia Informática

Unidade Curricular de Programação Avançada

Docente: André Sabino

Ano Letivo 2020/2021

SOCIAL NETWORK



Ana Rita Leal – 180221025

Francisco Moura – 180221015

Miguel Rosa – 180221023

Tiago Farinha – 180221011

Turmas SW-04 e SW-05

ÍNDICE

TADs IMPLEMENTADAS

Neste projeto, foram implementadas as TADs Stack, Queue, List e Map. A TAD Stack foi utilizada de acordo com o padrão de Software Memento. (ex: classe Caretaker)

```
public class Caretaker {
    private final SocialNetwork socialNetwork;
    private final Stack<Memento> undo;
```

A TAD Queue foi utilizada no método que permite percorrer o dígrafo em largura. (ex: classe DirectGraph)

```
private ArrayList<Vertex<V>> BFS(Vertex<V> v) {
    ArrayList<Vertex<V>> path = new ArrayList<>();
    Set<Vertex<V>> visited = new HashSet<>();
    Queue<Vertex<V>> queue = new LinkedList<>();
    visited.add(v);
    queue.add(v);
    while (!queue.isEmpty()) {
        Vertex<V> vLook = queue.remove();
        path.add(vLook);
        for (Edge<E, V> edge : outboundEdges(vLook)) {
            if (!visited.contains(edge.vertices()[1])) {
                visited.add(edge.vertices()[1]);
                queue.add(edge.vertices()[1]);
            }
        }
    }
    return path;
}
```

As TADs List e Map foram utilizadas várias vezes ao longo do projeto. (ex: classes Interest e SocialNetwork)

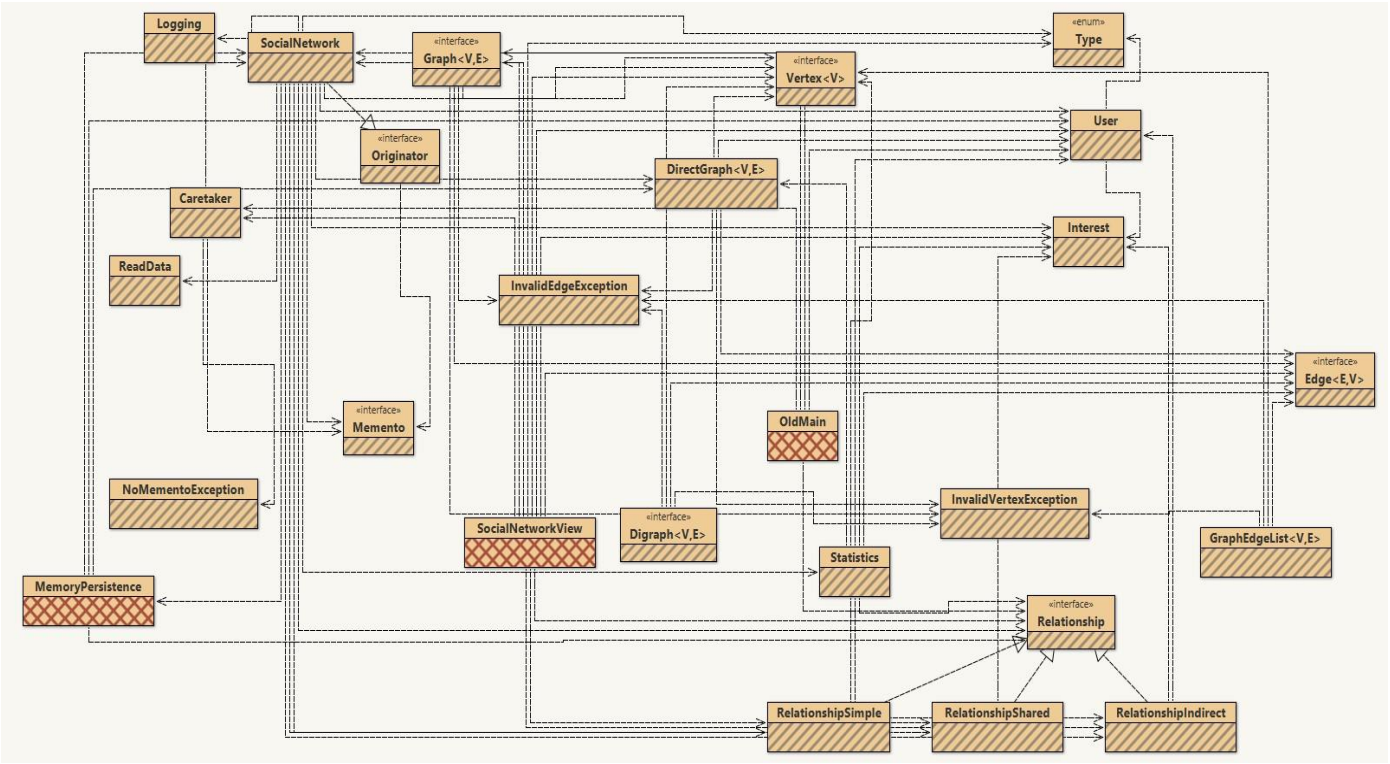
```
public class Interest implements Serializable {
    private final int id;
    private final String name;
    private final ArrayList<String> idsOfUsers;
```

```
public class SocialNetwork implements Originator, Serializable {

    private final HashMap<Integer, User> users;
    private final HashMap<Integer, ArrayList<String>> relationships;
    private final HashMap<Integer, Interest> interests;
    private final Logging log = Logging.getInstance();
    private final Statistics statistics;
    private DirectGraph<User, Relationship> graph;
    private String userNamesFile, relationshipsFile, interestNamesFile, interestsFile;
    MemoryPersistence memoryPersistence;
```

Implementamos ainda um Dígrafo que permite uma ligação entre vértices (Utilizadores) e arestas (Relações).

DIAGRAMA DE CLASSES



PADRÕES DE SOFTWARE

Foram utilizados os padrões Strategy, Singleton, Memento, DAO e uma variante do MVC abordada nas aulas teóricas denominada de ModelView na realização deste projeto. Contudo, consideramos que poderíamos ter implementado os padrões Observer e a totalidade do MVC.

O padrão Memento permitiu que guardássemos o estado do dígrafo em utilização no SocialNetworkView à medida que o fossemos percorrendo e realizando as ligações entre os vários Users.

O padrão Data Access Object (DAO) permitiu que guardássemos objetos, instanciando-os com o modo de persistência pretendidos e utilizando métodos para aceder ao objeto pretendido. Neste caso, os objetos encontram-se guardados em ficheiros no formato Json e foram criados métodos para salvar e atualizar um dado ficheiro.

O padrão Strategy foi utilizado para criar uma interface com um template genérico e facilmente adaptável, como forma de divisão dos vários relacionamentos em diretos simples, diretos com partilha de interesses e indiretos, facilitando as suas interligações e o funcionamento com a interface Serializable. Além disso, esta padrão também foi utilizado para criar uma interface com um método modelConstructor() que dependendo da concretização da interface, poderia ser Total ou Iterativo.

Relativamente ao padrão Observer, consideramos que o mesmo poderia ser relevante no que toca a notificar o utilizador quando é feita uma atualização a um modelo. Desta forma, sempre que o utilizador atualizasse um modelo anteriormente guardado, seria notificado sempre que fossem feitas alterações ao estado atual do mesmo.

Por fim, teríamos aplicado a totalidade do padrão MVC para dividir responsabilidades da classe SocialNetworkView que trata da visualização da classe SocialNetwork. Dessa forma, estaria implementando uma classe Model, uma classe View e outra Controller.

POSSÍVEIS BAD SMELLS

Ao longo do nosso projeto, deparámo-nos principalmente com vários Bad Smells.

Os mesmos podem ser visualizados na seguinte tabela:

BAD SMELL	DESCRIÇÃO	ONDE APARECE E QUANTAS VEZES	TECNICA DE REFACTORING
Speculative Generality	O código foi criado para oferecer suporte a recursos futuros antecipados que nunca foram implementados.	RelationshipIndirect (4x) Interest (2x) Logging (2x) RelationshipShared (2x) SocialNetwork (3x) SocialNetworkView (1x) Statistics (1x) User (1x)	Nestas situações, foram apagados os pedaços de código
Data Class	Classe formada apenas por getters e setters	User FileObject	Não fazer nada, porque é um BAD SMELL que não representa perigo Move Method
Long Method	Métodos longos	SocialNetworkView (4x) Statistics (x2) DirectGraph (x1)	Divisão do código por vários métodos (Extract Method) Consolidate ConditionalExpression
Duplicated Code	Código Duplicado	SocialNetworkView (x30) Logging (x2) Iterative Model (x1)	Utilização do Extract Method de forma a chamar um método que contenha o código em comum
Dead Code	Quando os requisitos mudaram ou as correções foram feitas, ninguém teve tempo de limpar o código antigo.	SocialNetworkView (6x) Statistics (x2) DirectGraph (x3) FileObject (x9) Logging (x3) RelationshipIndirect (x4) ViewObjectCreator (x1)	Nestas situações, foram apagados os pedaços de código
Primitive Obssession	Uso de constantes de string como nomes de	SocialNetwork (1x)	Replace Data with Object

	campo para uso em matrizes de dados		
Large Classe	Uma classe contém muitos campos / métodos / linhas de código.	SocialNetworkView SocialNetwork	Extract Class
Switch Statement	Quando um método contém um switch ou uma cadeia de ifs demasiado complexa	SocialNetwork (x1)	Consolidate Conditional Expression
Large Class	Quando uma classe contém demasiadas linhas de código (métodos, parâmetros etc).	SocialNetwork	Extract Class (neste caso, criando as classes SocialNetworkLog e SocialNetworkController)
Message Chains	Cadeias de chamadas de métodos para apenas retornar um simples resultado(por exemplo: a.getb().getc().getd())	SocialNetworkView (x3) MemoryPersistance (x2)	Extract Method Hide Delegate

As seguintes figuras são correspondentes a alguns exemplos dos bad smells e técnicas de refactoring que encontramos:

```
public class SocialNetwork implements Originator, Serializable {

    private final HashMap<Integer, User> users;
    private final HashMap<Integer, ArrayList<String>> relationships;
    private final HashMap<Integer, Interest> interests;
    private final Logging log = Logging.getInstance();
    private final Statistics statistics;
    MemoryPersistence memoryPersistence;
    private DirectGraph<User, Relationship> graph;
    private String userNamesFile, relationshipsFile, interestNamesFile, interestsFile;
```

Figura 1 – Code Smell: Primitive Obsession

```java
public class SocialNetwork extends Subject implements Originator, Serializable {
    private final HashMap<Integer, User> users;
    private final HashMap<Integer, ArrayList<String>> relationships;
    private final HashMap<Integer, Interest> interests;
    private final Statistics statistics;
    private final MemoryPersistence memoryPersistence;
    private DirectGraph<User, Relationship> graph;
    private FileObject fileObject;
```

Figura 2 - Refactoring do Primitive Obssession

```java
/**
 * Método responsável por atribuir o utilizar inbound e outbound do relacionamento
 *
 * @param inboundUser  representa o utilizador inbound
 * @param outboundUser representa o utilizador outbound
 */
public void setUsers(User inboundUser, User outboundUser) {
    if (inboundUser == null || outboundUser == null) return;
    users[0] = inboundUser;
    users[1] = outboundUser;

}
```

Figura 3 - Code Smell: Speculative Generality (Refactoring: método removido)

```java
public void insertEdge(User user1, User user2, boolean addIndirect) {
    if (user1 == null || user2 == null) {
        return;
    }

    List<Interest> tempInterests = new ArrayList<>();
    boolean relationshipDirect = false;
    Relationship relationship;

    for (Interest interest : this.interests.values()) {
        if (interest.getIdsOfUsers().contains(String.valueOf(user1.getID())) &&
                interest.getIdsOfUsers().contains(String.valueOf(user2.getID()))) {
            tempInterests.add(interest);
        }
    }

    if (this.relationships.get(user1.getID()).contains(String.valueOf(user2.getID()))) {
        relationshipDirect = true;
    }

    //HashSet<Edge<Relationship, User>> tempEdges = new HashSet<>(this.graph.edges());
    //tempEdges

    if (!tempInterests.isEmpty() && !relationshipDirect && addIndirect) {
        relationship = new RelationshipIndirect(tempInterests);
        this.graph.insertEdge(user1, user2, relationship);
        log.addRelationshipIndirect(user1.getID(), user2.getID(), tempInterests.size());
    } else if (tempInterests.isEmpty() && relationshipDirect && !addIndirect) {
        relationship = new RelationshipSimple();
        this.graph.insertEdge(user1, user2, relationship);
        log.addRelationshipDirect(user1.getID(), user2.getID(),  interests: 0);
    } else if (!tempInterests.isEmpty() && relationshipDirect && !addIndirect) {
        relationship = new RelationshipShared(tempInterests);
        this.graph.insertEdge(user1, user2, relationship);
        log.addRelationshipDirect(user1.getID(), user2.getID(), tempInterests.size());
    }
    updateLog();

}
```

Figura 4 - Code Smell: Long Method

```java
public void insertEdge(User user1, User user2, boolean addIndirect) {
    if (user1 == null || user2 == null) return;

    boolean relationshipDirect = false;

    List<Interest> tempInterests = this.interests.values().stream().filter(interest -> interest.getIdsOfUsers()
            .contains(String.valueOf(user1.getID())) &&
            interest.getIdsOfUsers().contains(String.valueOf(user2.getID()))).collect(Collectors.toList());

    if (this.relationships.get(user1.getID()).contains(String.valueOf(user2.getID()))) relationshipDirect = true;

    checkInterest(user1, user2, addIndirect, tempInterests, relationshipDirect);
    SocialNetworkLog.updateLog();

}

private void checkInterest(User user1, User user2, boolean addIndirect, List<Interest> tempInterests, boolean relationshipDirect) {
    Relationship relationship = new RelationshipSimple();
    if (!tempInterests.isEmpty() && ((!relationshipDirect && addIndirect) || (relationshipDirect && !addIndirect))) {
        relationship = new RelationshipIndirect(tempInterests);
    }
    this.graph.insertEdge(user1, user2, relationship);
    SocialNetworkLog.getLog().addRelationshipDirect(user1.getID(), user2.getID(), tempInterests.size());
}
```

Figura 5 - Refactoring: Long Method

```java
if (!tempInterests.isEmpty() && !relationshipDirect && addIndirect) {
    relationship = new RelationshipIndirect(tempInterests);
    this.graph.insertEdge(user1, user2, relationship);
    log.addRelationshipIndirect(user1.getID(), user2.getID(), tempInterests.size());
} else if (tempInterests.isEmpty() && relationshipDirect && !addIndirect) {
    relationship = new RelationshipSimple();
    this.graph.insertEdge(user1, user2, relationship);
    log.addRelationshipDirect(user1.getID(), user2.getID(), interests: 0);
} else if (!tempInterests.isEmpty() && relationshipDirect && !addIndirect) {
    relationship = new RelationshipShared(tempInterests);
    this.graph.insertEdge(user1, user2, relationship);
    log.addRelationshipDirect(user1.getID(), user2.getID(), tempInterests.size());
}
```

Figura 6 - Code Smell: Switch Statement

```java
private void checkInterest(User user1, User user2, boolean addIndirect, List<Interest> tempInterests, boolean relationshipDirect) {
    Relationship relationship = new RelationshipSimple();
    if (!tempInterests.isEmpty() && ((!relationshipDirect && addIndirect) || (relationshipDirect && !addIndirect))) {
        relationship = new RelationshipIndirect(tempInterests);
    }
    this.graph.insertEdge(user1, user2, relationship);
    SocialNetworkLog.getLog().addRelationshipDirect(user1.getID(), user2.getID(), tempInterests.size());
}
```

Figura 7 - Refactoring: Switch Statement

```java
@Override
public String toString() {
    String s = "******************** Logging ********************";

    for (String logLine : this.log) {
        s = s + "\n" + logLine;
    }

    s = s + "\n******************** End ********************";

    return s;
}
```

Figura 8 - Code Smell: Dead Code

```java
@Override
public String toString() {
    return log.stream().map(logLine -> "\n" + logLine).collect(Collectors
            .joining( delimiter: "",  prefix: "******************** Logging ********************",
                    suffix: "\n******************** End ********************"));
}
```

Figura 9 - Refactoring: Dead Code

```java
protected void exportJSON() {
    try {
        Gson gson = new GsonBuilder().setPrettyPrinting().create();
        FileOutputStream fileOut = new FileOutputStream( name: "outputFiles/exportJSON.json");
        ObjectOutputStream out = new ObjectOutputStream(fileOut);
        out.writeObject(gson.toJson(socialNetwork.getGraph().getVertices()));
        out.close();
        fileOut.close();
    } catch (IOException e) {
        System.err.println(e.getMessage());
    }
}
```

Figura 10 - Message Chains

```java
protected void exportJSON() {
    try {
        Gson gson = new GsonBuilder().setPrettyPrinting().create();
        FileOutputStream fileOut = new FileOutputStream( name: "outputFiles/exportJSON.json");
        ObjectOutputStream out = new ObjectOutputStream(fileOut);
        out.writeObject(gson.toJson(socialNetwork.getVertices()));
        out.close();
        fileOut.close();
    } catch (IOException e) {
        System.err.println(e.getMessage());
    }
}
```

Figura 11 - Refactoring: Message Chains

```java
 */
public void addRelationshipDirect(int idUserAdded, int idUserExistent, int interests) {
    String logCreated = "<" + this.formatLocalDateTime(LocalDateTime.now()) + "> | <" + idUserAdded +
            "> | <" + idUserExistent + "> | <" + interests + ">";
    this.log.add(logCreated);
}

/**
 * Método que permite gerar o Logging para a adição de um interesse
 *
 * @param idUserAdded representa o id do utilizador adicionado
 * @param idInterest  representa o id do interesse
 */
public void addInterest(int idUserAdded, int idInterest) {
    String logCreated = "<" + this.formatLocalDateTime(LocalDateTime.now()) + "> | <" + idUserAdded +
            "> | <" + idInterest + ">";
    this.log.add(logCreated);
}
```

Figura 12 - Code Smell: Duplicate Code

```java
public void addRelationshipDirect(int idUserAdded, int idUserExistent, int interests) {
    addToLog( logCreated: "<" + this.formatLocalDateTime(LocalDateTime.now()) + "> | <" + idUserAdded +
            "> | <" + idUserExistent + "> | <" + interests + ">");
}

/**
 * Método que permite gerar o Logging para a adição de um interesse
 *
 * @param idUserAdded representa o id do utilizador adicionado
 * @param idInterest  representa o id do interesse
 */
public void addInterest(int idUserAdded, int idInterest) {
    addToLog( logCreated: "<" + this.formatLocalDateTime(LocalDateTime.now()) + "> | <" + idUserAdded +
            "> | <" + idInterest + ">");
}
```

Figura 13 - Refactoring: Duplicate Code

Encontrámos também em 2 classes o code smell **Large Class**. Para resolver este problema, utilizámos a técnica de refactoring Extract Class, passando partes de código para classes próprias (Por exemplo, retirámos código da classe SocialNetwork e colocámos em 2 classes mais apropriadas: SocialNetworkLog e SocialNetworkController).