ISEL
INSTITUTO SUPERIOR DE ENGENHARIA DE LISBOA

Trabalho Prático 3
Modelação e Programação

LICENCIATURA EM ENGENHARIA INFORMÁTICA E MULTIMÉDIA
ANO LETIVO 2018/2019

Turma LEIM 21D

Data:12/05/2019

Docentes:
Eng. Teófilo

Alunos:
Gonçalo Almeida (A43746)
Luis Fonseca (A45125)

Índice

1. Introdução

Neste trabalho, o tema posto em estudo foi em relação a classes abstratas, interfaces, herança e o estudo de diagramas UML. A maneira de como foi feita a resolução dos exercícios foi feita em relação ao tema do trabalho anterior que consistia no uso de objetos.

2. Cenário dos Percursos

2.1. Percurso

A classe de nome "Percurso" consistia em descrever um percurso e guardar apenas o atributo "nome".

Recebendo apenas um construtor, no qual permite verificar os argumentos inválidos, caso o comprimento da *String* for maior que 0, e se é formada por letras e dígitos, para verificar que o nome continha caracteres e dígitos foram usadas as funções "*Character.isDigit()*" e "*Character.isLetter()*".

O método *getNome()* que devolve o nome do percurso.

Os métodos *getInicio(), getFim(),getDistancia(),getDeclive(), getLocalidades() e getDescricao()* serão abstractos nesta classe, no qual, estes métodos pertencem a uma classe abstrata que não possuem implementação.

O método *validarNome()* no qual é um booleano que verifica se um nome inserido no percurso contem dígitos, letras e o seu comprimento é maior que um. Para verificar essas tais restrições foram usadas as mesmas funções usadas no construtor.

O método *getString()* que retorna uma string com o nome do percurso.

O método *print(String prefix)* que imprime na consola o prefixo.

2.2. Percurso Simples

A classe PercursoSimples descreve um percurso simples, no qual, irá guardar os dados de: início, fim, declive e distância.

Nesta classe iremos ter dois construtores, um deles irá servir para verficar os argumentos inválidos. Para o atributo nome, recorreu-se à função super, esta função serve para chamar o construtor da superclasse. O *inicio* e o *fim*, que serão verificados com o método *validarNome()* e verificar quando é que a distância não é negativa.

O outro constructor serve para fazer uma cópia do percurso, recorrendo ao super, para chamar a funcao *getNome()* , e recorrer ao atributo *this*, para aceder aos atributos da classe.

O método *clone()* que permite criar uma cópia do percurso recebido.

O método *getInicio()* que retorna o início do percurso.

O método *getFim()* que retorna o fim do percurso.

O método *getDistancia()* que devolve a distância do percurso.

O método *getDeclive()* que devolve o declive do percurso.

O método *toString()* que retorna uma *String* do tipo de um dos testes apresentados em baixo.

O método *equals()* retorna true se o tipo de percurso é do tipo PercursoSimples, caso tenha o mesmo nome, início e fim.

O método *getLocalidades()* que devolve um array com todas as localidades de início e fim.

O método *getDescricao()* que retorna a String "simples".

2.3. Percursos Compostos

A classe PercursoComposto descreve um percurso que pode conter outros percursos dentro dela. Esta classe conterá um array de Percurso, um número de percursos existentes e pode conter Percursos Simples ou Percursos Compostos misturados nesse array, desde que estejam em sequência e não contenham replicações dos nomes das localidades.

Nesta classe iremos ter três construtores. O primeiro consiste em receber apenas um percurso, um nome, e um número máximo de percurso. Para isso usou-se o operador *this* no qual irá fazer referência aos atributos da própria classe.

O segundo construtor serve para verificar os argumentos inválidos, se o array ter pelo menos um percurso, os percursos dentro desse array tem de estar em sequência, e não pode haver repetições de localidades.

O terceiro construtor irá servir para criar uma cópia do percurso recebido.

O método *clone()* que permite criar uma cópia do percurso recebido.

O método *addicionarPercursoNoFinal(Percurso percurso)* adiciona o percurso no final, desde que esteja em sequência e haja espaço e não existir repetição de localidades. Para isso comparou-se o início e o fim dos percursos, caso não fosse, era retornado *false*, de seguida foi colocado o comprimento do percurso dentro de um array, do tipo *Percurso* e finalmente criou-se uma cópia do percurso, usando a funcao System.arraycopy().

O método *haveRepetitions(String[] locs1, String[] locs2)* que permite comparar e verificar se existe repetições entre dois arrays.

O método *getLocalidades()* que obtém todas as localidades distintas existentes do percurso. Este método retorna um novo array, com todas as localidades, sem repetições e sem nulls.

O método *getNumLocalidades()* permite obter o número de localidades distintas existentes dentro do percurso compost actual, neste método iram ser colocadas as localidades de ínicio e fim.

O método *addicionarPercursoNoInicio(Percurso percurso)* adiciona o percurso no final, desde que esteja em sequência e haja espaçp e não existir repetição de localidades. Para isso comparou-se o inicio e o fim dos percursos, caso não fosse, era retornado *false*, de seguida foi colocado o comprimento do percurso dentro de um array, do tipo *Percurso* e finalmente criou-se uma cópia do percurso, usando a funcao System.arraycopy().

O método *getInicio()* que retorna o início do percurso.

O método *getFim()* que retorna o fim do percurso.

O método *getDistancia()* que devolve a distância do percurso, no qual consiste no somatório das distancias dos seus percursos.

O método *getDeclive()* que devolve o declive do percurso, no qual consiste no somatório dos declives associados aos respetivos percursos, sendo que neste método foi considerado apenas os declives positivos.

O método *getSubidaAcumulada()* que devolve o declive do percurso, no qual consiste no somatório dos declives associados aos respetivos percursos, sendo que neste método foi considerado apenas os declives positivos.

O método *print(String prefix)* que imprime na consola o prefixo.

O método *getDescricao()* que retorna a String “composto”.

O método *removerPercusosNoFimDesde(String localidade)* no qual remove e devolve todos os percursos desde o fim até à localidade recebida.

O método *removerPercusosNoInicioAte(String localidade)* no qual remove e devolve todos os percursos desde o início até à localidade recebida.

O método *getIdxLocalidadeEmInicio(String localidade)* que devolve o índice do elemento do percurso, dentro do percurso corrente, e tem como partida, uma localidade recebida.

Para terminar a classe do percurso composto, foi também feito um exercício, onde era pedido que fosse feito um teste às funções *getDeclive(), getSubidaAcumulada()* e *getDistancia(),* no qual, em ambas as funções, era retornado o valor da distancia, do declive e da subida acumulada de um percurso composto.

3. Conclusões

Através deste trabalho, o grupo conseguiu perceber como funciona a herança e as classes abstratas e a sua utilidade, a herança é uma funcionalidade que permite que classes herdem de outras classes e que as possam estender com mais funcionalidades, dentro da herança também foi descoberto o conceito de polimorfismo, este conceito é uma propriedade dos objetos, no qual, estes apresentam comportamentos diferenciados ou apresentarem várias formas.
Para além da herança, o grupo também se deparou com métodos/classes abstratas, este termo consiste em uma classe, no qual não se pode criar instâncias, e sendo um método abstrato, não contem corpo. Em termos de classes, uma respetiva classe que tenha pelo menos um método abstrato. Em alguns métodos também foi usado o conceito da Interface, é como se tivesse o conceito de abstrato mas tem mais obrigações.

4. Bibliografia

Slides – Eng. Teófilo - MOP 06 – classes e objetos
Slides – Eng. Teófilo - MOP 06.1 – UML
Slides – Eng. Teófilo - MOP 07 – Herança
Slides – Eng. Teófilo - MOP 08 – Herança, classes abstratas e interfaces

5. Anexos

Percurso

```
public Percurso(String nome) {
if(nome != null){
	if(nome.length() != 0){
		if(nome.substring(0, 1).matches("[A-Za-z]")){
			if(nome.substring(1, 2).matches("[A-Za-z]{1}") || nome.substring(1,
2).matches("[0-9]{1}")){
			this.local = nome;
		}else
	throw new IllegalArgumentException("Tem de ter pelo menos mais 1 letra ou digito");
		}else
	throw new IllegalArgumentException("Não começa por letra");
		}else
	throw new IllegalArgumentException("Nome nao tem comprimento");
		}else
	throw new IllegalArgumentException("Nome é null");
}

public String getNome() {return this.nome;}

public abstract String getInicio();

public abstract String getFim();

public abstract int getDistancia();

public abstract int getDeclive();

public abstract String[] getLocalidades();

protected static boolean validarNome(String local) {
if(local != null){
	if(local.length() != 0){
		if(local.substring(0, 1).matches("[A-Za-z]")){
			if(local.substring(1, 2).matches("[A-Za-z]{1}") || local.substring(1,
2).matches("[0-9]{1}")){
			return true;
					}
				}
			}
		}
		return false;
}

public String toString() {
	return "percurso " + getDescricao() + " " + getNome() + " de "
	+ getInicio() + " para " + getFim() + ", com " + getDistancia()
	+ " metros e com " + getDeclive() + " de declive";
}

public abstract String getDescricao();
```

```
public void print(String prefix) {System.out.println(prefix + this);}
```

Percurso Simples

```
public PercursoSimples(String nome, String inicio, String fim, int distancia, int declive) {
    super(nome);

    if(!validarNome(inicio)){throw new IllegalArgumentException("Inicio inválido");}

    if(!validarNome(fim)){throw new IllegalArgumentException("Fim inválido");}

    if(distancia <= 0) {throw new IllegalArgumentException("A distancia tem de ser positiva!");}

    this.inicio = inicio;

    this.fim = fim;

    this.distancia = distancia;

    this.declive = declive;
}

    public PercursoSimples(PercursoSimples p) {
            super(p.getNome());
            this.inicio = p.inicio;
            this.fim = p.fim;
            this.declive = p.declive;
            this.distancia = p.distancia;
    }
    public PercursoSimples clone() {return new PercursoSimples(this);}

    public String getInicio() {return this.inicio;}

    public String getFim() {return this.fim;}

    public int getDistancia() {return this.distancia;}

    public int getDeclive() {return this.declive;}

    public boolean equals(Object p) {
            if(this == p) return true;

            if(p.getClass() != this.getClass()) return false;

            PercursoSimples percurso = (PercursoSimples) p;

            return percurso.getFim().equals(getFim()) && percurso.getInicio().equals(getInicio())
                                    && percurso.getNome().equals(getNome());
            }

    public String[] getLocalidades() {return new String[] {getInicio(),getFim()};}

    public String getDescricao() {return "simples";}
```

Percursos Compostos

```
public PercursoComposto(String nome, Percurso percurso, int maxPercursos) {
        this(nome,new Percurso[] {percurso},maxPercursos);
}

public PercursoComposto(String nome, Percurso[] percursos, int maxPercursos) {
        super(nome);

        final boolean CHECKLOCALIDADES = true;

        if (CHECKLOCALIDADES) {
                if(percursos != null ){
                for(int i = 0; i < percursos.length-1; i++){
                if (percursos[i] != null) {

if(!(percursos[i].getFim().equals(percursos[i+1].getInicio())) &&

!(percursos[i].equals(percursos[percursos.length-1]))){
throw new IllegalArgumentException("Os percursos não estão em sequência");
                                                        }
                                                }else
throw new IllegalArgumentException("Os percursos não podem ser nulls");
                                        }
                }else
throw new IllegalArgumentException("Percurso sem percursos simples");
        } else {
                for(int i = 0; i < percursos.length; i++){
                        addicionarPercursoNoFinal(percursos[i]);
                }
        }
        this.percursos = percursos;
        this.nPercursos = maxPercursos;

}

public PercursoComposto(PercursoComposto pc) {
                super(pc.getNome());

                this.percursos = new Percurso[pc.percursos.length];

                System.arraycopy(pc.percursos, 0, this.percursos, 0, pc.percursos.length);

                this.nPercursos = pc.nPercursos;
        }

public Percurso clone() {return new PercursoComposto(this);}

public boolean addicionarPercursoNoFinal(Percurso percurso) {
        if(percurso == null) return false;

        if (nPercursos < percursos.length) return false;

        if(!(percursos[percursos.length-1].getFim().equals(percurso.getInicio()))) return false;
```

```
		Percurso[] novo_array = new Percurso[percursos.length+1];

		System.arraycopy(percursos, 0, novo_array, 0, percursos.length);

		novo_array[percursos.length] = percurso;

		this.percursos = novo_array;

		return true;
	}




	private static boolean haveRepetitions(String[] locs1, String[] locs2) {
		for(int i = 0; i < locs1.length;i++) {
			for(int j = 0; j < locs2.length;j++) {
				if(locs1[i].equals(locs2[j])) {
					return true;
				}
			}
		}
		return false;
	}

	public String[] getLocalidades() {
		String[] localidades = new String[this.getNumLocalidades()];
		int idx = 0;
		localidades[idx] = percursos[0].getInicio();
		idx++;
		for(int i = 0; i < percursos.length;i++) {
			if(percursos[i] instanceof PercursoSimples) {
				localidades[idx] = percursos[i].getFim();
				idx++;
			}else if(percursos[i] instanceof PercursoComposto) {
				PercursoComposto a = (PercursoComposto)percursos[i];
		System.arraycopy(a.getLocalidades(),1, localidades,idx,a.getLocalidades().length-1);
				idx += a.getLocalidades().length-1;
			}
		}
		return localidades;
	}

	private int getNumLocalidades() {
		int contador = 0;
		for(int i = 0; i< percursos.length;i++) {
			if(percursos[i] instanceof PercursoComposto) {
			contador += ((PercursoComposto)percursos[i]).getNumLocalidades()-1;
			}else if(percursos[i] instanceof PercursoSimples) {
				contador ++;
			}
		}
		return contador +1;
	}
```

```
public boolean addicionarPercursoNoInicio(Percurso percurso) {
	if(percurso == null) return false;

	if (percursos.length+1 > nPercursos)  return false;

	if(!(percursos[0].getInicio().equals(percurso.getFim()))) return false;

	Percurso[] novo_array_2 = new Percurso[percursos.length+1];

	System.arraycopy(percursos, 0, novo_array_2, 1, percursos.length);

	novo_array_2[0] = percurso;

	this.percursos = novo_array_2;

	return true;
}
public String getInicio() {
	if(nPercursos > 0) return percursos[0].getInicio();
	return null;
}

public String getFim() {
	if(nPercursos > 0) return percursos[percursos.length-1].getFim();
	return null;
}

public int getDistancia() {
	int distancia = 0;
	for(int i = 0; i < percursos.length;i++) {
		distancia = percursos[i].getDistancia();
	}
	return distancia;
}

public int getDeclive() {
	int declive = 0;
	for(int i = 0; i < percursos.length;i++) {
		declive = percursos[i].getDeclive();
	}
	return declive;
}

public int getSubidaAcumulada() {
	int subAc = 0;
	for(int i = 0; i < percursos.length; i++) {
		int vd = percursos[i].getDeclive();
		if(vd > 0) subAc += percursos[i].getDeclive();
	}
	return subAc;
}

public void print(String prefix) {
	System.out.println(prefix+toString());
	for(int i = 0; i < percursos.length;i++) {
		percursos[i].print(" " + " " + " " + prefix + percursos[i]);
	}
```

```
}

public String getDescricao() {return "composto";}









public boolean removerPercursosNoFimDesde(String localidade) {
if(getIdxLocalidadeEmInicio(localidade) == -1) return false;


if(percursos[getIdxLocalidadeEmInicio(localidade)] instanceof PercursoSimples) {
        Percurso[] newArray = new Percurso[getIdxLocalidadeEmInicio(localidade)];

        System.arraycopy(percursos, 0, newArray, 0, getIdxLocalidadeEmInicio(localidade));

        this.percursos = newArray;
        this.nPercursos = percursos.length;

        return true;
}else {
        if(percursos.length == 1 &&
        percursos[getIdxLocalidadeEmInicio(localidade)].getInicio().equals(localidade)){
                        return false;
                }
        }if(percursos[getIdxLocalidadeEmInicio(localidade)].getInicio().equals(localidade)) {
        Percurso[] newArray = new Percurso[getIdxLocalidadeEmInicio(localidade)];
        System.arraycopy(percursos, 0, newArray, 0, getIdxLocalidadeEmInicio(localidade));
        this.percursos = newArray;
        this.nPercursos = percursos.length;

        return true;
                }else{
        ((PercursoComposto)percursos[getIdxLocalidadeEmInicio(localidade)]).removerPercurs
osNoFimDesde(localidade);
        }
        return true;
}

public boolean removerPercursosNoInicioAte(String localidade) {
if(getIdxLocalidadeEmInicio(localidade) == -1) return false;

if(percursos[getIdxLocalidadeEmInicio(localidade)] instanceof PercursoSimples) {
Percurso[] newArray = new Percurso[percursos.length - getIdxLocalidadeEmInicio(localidade)];

System.arraycopy(percursos, getIdxLocalidadeEmInicio(localidade), newArray, 0,
percursos.length - getIdxLocalidadeEmInicio(localidade));
```

```
this.percursos = newArray;

this.nPercursos = percursos.length;

return true;
}else {
if(percursos.length == 1 &&
percursos[getIdxLocalidadeEmInicio(localidade)].getInicio().equals(localidade)){
        return false;
                }
if(percursos[getIdxLocalidadeEmInicio(localidade)].getInicio().equals(localidade)) {

Percurso[] newArray = new Percurso[percursos.length - getIdxLocalidadeEmInicio(localidade)];

System.arraycopy(percursos, getIdxLocalidadeEmInicio(localidade), newArray, 0,
percursos.length - getIdxLocalidadeEmInicio(localidade));

this.percursos = newArray;

this.nPercursos = percursos.length;

return true;

}else{
((PercursoComposto)percursos[getIdxLocalidadeEmInicio(localidade)]).removerPercursosNoInic
ioAte(localidade);
                }
                return true;
        }
}
private int getIdxLocalidadeEmInicio(String localidade) {
for(int i = 0; i < percursos.length;i++) {
        if(percursos[i] instanceof PercursoComposto) {
        int a = ((PercursoComposto)percursos[i]).getIdxLocalidadeEmInicio(localidade);
        if (a != 1) return i;
        }else if(percursos[i] instanceof PercursoSimples) {
                if(localidade.equals(percursos[i].getInicio())) {
                        return i;
                        }
                }
        }
        return -1;
}
```

Cenário do teste realizado

```
private static void testeDecliveDistanciaSubidaAcumulada() {

Percurso ps0, ps1, ps2, ps3, ps4;

ps0 = new PercursoSimples("A23", "Sagres", "Faro", 67_000, -10);
ps1 = new PercursoSimples("A2", "Faro", "Lisboa", 278_000, 10);
ps2 = new PercursoSimples("A1 1", "Lisboa", "Coimbra", 204_000, 10);
ps3 = new PercursoSimples("A1 2", "Coimbra", "Porto", 113_000, 20);
ps4 = new PercursoSimples("A28", "Porto", "Viana do Castelo", 73_800, 30);
```

```
        PercursoComposto pc2 = new PercursoComposto("PC2",new
        Percurso[]{ps0,ps1,ps2,ps3,ps4},20);

        System.out.println("O declive de PC2 é " + pc2.getDeclive());

        System.out.println("A subida acumulada de PC2 é " + pc2.getSubidaAcumulada());

        System.out.println("A distancia de PC2 é " + pc2.getDistancia());

        System.out.println();
        }
```

Interface dos Percursos

```
public interface IPercurso {

        String getInicio();

        String getFim();

        int getDistancia();

        int getDeclive();

        String getDescricao();

        String[] getLocalidades();
}
```