Exame- Época Normal de Programação Avançada 2022/23

2 de fevereiro de 2022 (Duração: 2h + 15 minutos tolerância)

IPS Instituto Politécnico de Setúbal
Escola Superior de Tecnologia de Setúbal

Nome:_____ **Número:**_____

GRUPO1	1|	2|	3|	4|	5|	6|	7|	8|
Preencher Aluno								

GRUPO2	1	2	3	4	TOTAL
Preencher Professor					

GRUPO 1 (8 questões)

Questões de Escolha múltipla (seleciona a resposta correta) – Cada resposta correta vale 1 valor, cada resposta incorreta desconta 0,25.

Q1 - O método recursivo pretende determinar se o elemento `search` existe numa *folha* de uma árvore binária que guarda caracteres. Cada nó da árvore é representado pela classe `TreeNode` que contém os atributos {`elem`, `left` e `right`}.

```
boolean exists(TreeNode treeRoot, char search) {
    if( /* (W) */ ) return false;
    if( /* (X) */ ) {
        return /* (Y) */;
    } else {
        return /* (Z) */;
    }
}
```

Qual o conjunto de instruções que complementa corretamente o método?

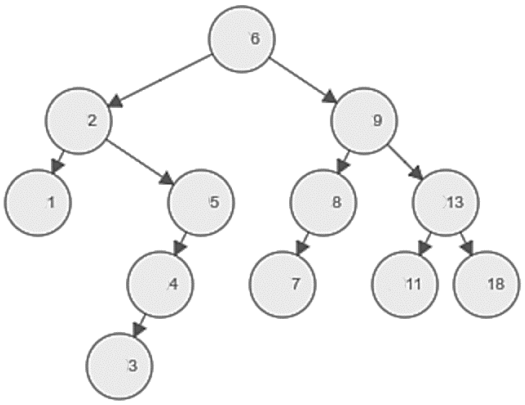
A. (W) `treeRoot == null`
(X) `treeRoot.left == null || treeRoot.right == null`
(Y) `treeRoot.elem == search`
(Z) `exists(treeRoot.left, search) && exists(treeRoot.right, search)`

B. (W) `treeRoot == null`
(X) `treeRoot.left == null && treeRoot.right == null`
(Y) `treeRoot.elem != search`
(Z) `exists(treeRoot.left, search) || exists(treeRoot.right, search)`

C. (W) `treeRoot == null`
(X) `treeRoot.left == null && treeRoot.right == null`
(Y) `treeRoot.elem == search`
(Z) `exists(treeRoot.left, search) || exists(treeRoot.right, search)`

D. Nenhuma das anteriores

Q2 Considere a árvore binária de pesquisa apresentada na figura. Após a remoção do nó 9, qual o conjunto de nós internos da árvore resultante?



A. `{2,4,5,8,13}`

B. `{2,4,5,6,8,13}`

C. `{1,3,7,11,18}`

D. `{2,6,8,13}`

Q3 Considere que se pretende modelar os itinerários de autocarros da TST através de um grafo. Na representação seguinte, assinale a **resposta correta**:

```
Graph<ClassA, ClassB> itinerariosTST;
```

A. `ClassA` é uma classe que representa o bilhete de autocarro e que guarda informação sobre a origem e o destino e o tempo total previsto para a viagem.
B. `ClassB` é uma classe que representa um paragem de autocarro.
C. `ClassB` é uma classe que guarda informação sobre o tempo do trajeto entre duas paragens contíguas
D. `ClassA` é um *vértice* e `ClassB` é uma *aresta*.

Q4 - Considere o seguinte resultado do algoritmo *Dijkstra* sobre um grafo não-orientado e valorado:

vertex	cost	predecessor
A	1	D
B	2	D
C	5	F
D	0	
E	1	D
F	4	A
G	∞	

Selecione a **afirmação falsa**:

A. O custo do caminho entre F e A é 4

B. G é um vértice isolado

C. O caminho de menor custo entre D e F é {D - A - F}

D. O caminho de menor custo entre D e C é {D - A - F - C}

Q5 –Se pretendermos que o iterador fornecido faça uma **travessia da base para o topo da pilha**, que opção satisfaz isto no código em falta?

A. [A] cursor = size - 1
[B] cursor >= 0
[C] elements[cursor--]

B. [A] cursor = size – 1
[B] cursor > 0
[C] elements[cursor--]

C. [A] cursor = 0
[B] cursor < size – 1
[C] elements[cursor++]

D. [A] cursor = 0
[B] cursor <= size – 1
[C] elements[cursor++]

```
public class StackImpl<T> implements Stack<T> {
    private T[] elements;
    private int size;
    //...
    @Override
    public T pop() throws EmptyQueueException {
        //...
        return elements[--size];
    }
    @Override
    public Iterable<T> iterator() {
        return new MyIterator();
    }

    private class MyIterator implements Iterator<T> {
        private int cursor;
        public MyIterator() {
            // A ?
        }
        @Override
        public boolean hasNext() {
            return // B ?
        }
        @Override
        public T next() {
            if(!hasNext()) return null;
            return // C ?
        }
```

Q6 - Considere o seguinte código. Qual a técnica de *refactoring* aplicada?

A. Move Method
B. Extract Method
C. Hide Delegate
D. Nenhuma das anteriores

```
//Antes
public SocialNet {
    public Graph<Person, Relation> graph;
    ....
}
public static void main(String[] args) {
    SocialNet s= new SocialNet();
    Vertex<Person> p1 = s.graph.insertVertex(new Person("u1234", "Ana"));
    Vertex<Person> p2 = s.graph.insertVertex(new Person("u1334", "Pedro"));

}
// Após
public SocialNet {
    public Graph<Person, Relation> graph;
    public void addUser(Person p){
      graph.insertVertex(p);
    }
}
public static void main(String[] args) {
    SocialNet s= new SocialNet();
    Vertex<Person> p1 = s.addUser(new Person("u1234", "Ana"));
    Vertex<Person> p2 = s.addUser(new Person("u1334", "Pedro"));
}
```

Q7 - Qual o conjunto de instruções que completa corretamente o código em falta, para os trechos ***TODO1*** e ***TODO4***.

A. `[TODO1]implements Subject`
`[TODO4]senha.addObserver(view);`

B. `[TODO1]extends Observer`
`[TODO4]view.addObserver(senha);`

C. `[TODO1]implements Observer`
`[TODO4]senha.addObserver(view);`

D. `[TODO1]extends Subject`
`[TODO4]view.addObserver(senha);`

```
public class SenhaView //TODO1{
    @Override
    public void update(Object obj) {
        if(obj instanceof Integer)
            System.out.printf("****%d****\n",obj);
    }
}
public class Senha // TODO2{
    private int value=0;
    private static final int max=100;
    public void inc(){
        value=(value)%max+1;
        //TODO3
    }
}
public class Main {
    public static void main(String[] args) {
        Senha senha = new Senha();
        SenhaView view = new SenhaView();
        //TODO4
        for (int i = 0; i < 200; i++)
            senha.inc();
    }
}
}}
```

Q8 – Considere o seguinte código: Qual o *bad smell* presente?:

A. Inappropriate intimacy
B. Temporary field
C. Data clumps
D. Nenhuma das anteriores

```
public class VarianceCalculator {
    private List<Integer> numbers = new ArrayList<>();
    private double mean, aux;

    public void add(int number) {
        numbers.add(number);
    }

    public double compute() {
        aux = 0;
        for(Integer i : numbers) {
            aux += numbers;
        }
        mean = aux / (double)numbers.size();
        aux = 0;
        for(Integer i : numbers) {
            aux += (i-mean)*(i-mean);
        }
        return aux / mean;
    }
}
```

GRUPO II (4 Questões) Cada questão vale 3 valores

Q1– GRAFOS

Considere uma implementação do *ADT Graph* baseada→o na estrutura de dados lista de arestas, de acordo com as seguintes restrições:

– Cada vértice conhece apenas o *elemento* que guarda;
– Cada aresta conhece o *elemento* que guarda **e os dois *vértice*** que liga;
– A estrutura de dados mantém uma *coleção* de todos os vértices;
– A estrutura de dados mantém uma *coleção* de todas as arestas.
	`public class GraphEdgeList<V,E> implements Graph<V,E> {`
	`  /* TODO: atributos + construtor */`
	
	`  // TODO: método(s) solicitado(s)`
	
	`  private class MyVertex implements Vertex<V> {`
	`  /* TODO: atributos + construtor */`
	`  }`
	`  private class MyEdge implements Edge<E, V> {`
	`  /* TODO: atributos + construtor */`
	`  }`
	`}`

a) (1 val) Forneça os **atributos** e os **construtores** das classes *GraphEdgeList, MyVertex* e *MyEdge*.

b) (1 val) Forneça a implementação dos dois seguintes métodos:

```
Edge<E, V> insertEdge(Vertex<V> v, Vertex<V> w, E edgeElement) {
    /* Ignore validação de parâmetros */









}

boolean areAdjacent(Vertex<V> v, Vertex<V> w) {
    /* Ignore validação de parâmetros */









}
```

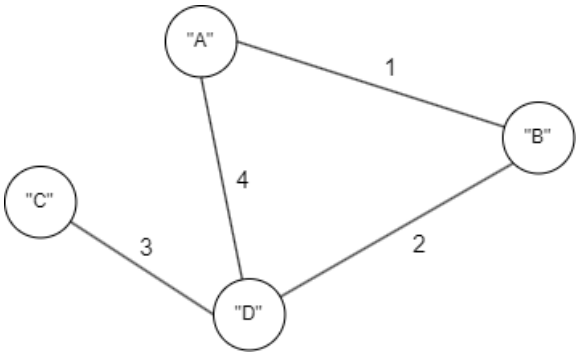
c) (1 val) Pretende-se criar **um (único) teste unitário** para esta implementação.
Assuma que possui adicionalmente o método `Vertex<V> insertVertex(V vertexElement)` que insere um vértice no grafo.

O teste unitário deverá pela seguinte ordem:

i. Instanciar o grafo ilustrado na figura (escolha a parametrização apropriada para os tipos genéricos);

ii. Verificar que os vértices com os elementos `"A"` e `"D"` **são adjacentes**, e;

iii. Verificar que os vértices com os elementos `"A"` e `"C"` **não são adjacentes**.



```java
public class GraphEdgeListTest {
    @Test
    public void create_and_check_graph() {
        /* All code must go here */

    }
}
```

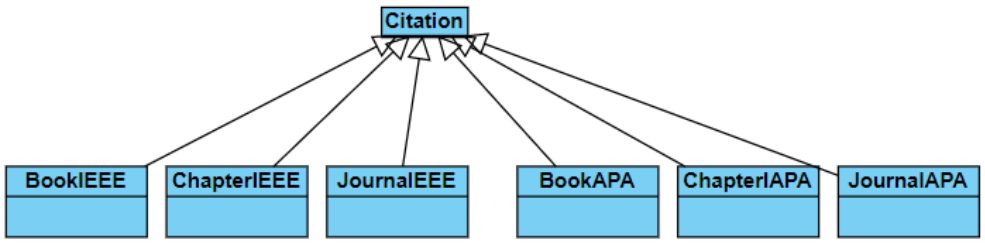
Q2– Pattern Factory

a) (1 valor) Considere as sub-classes de `Citation`: `BookCitation`, `JournalCitation` e `ChapterCitation`; todas possuem construtores sem argumentos.
Complete e implemente classe `CitationFactory` que funciona como uma *Simple Factory* para as citações disponíveis:

```
public class CitationFactory {

    public static ____________ create( ______________________________ ) {
```

b) (1 valor) No problema apresentado, surgiu a possibilidade de suportar duas convenções distintas de citações: IEEE e APA sendo que a hierarquia de classes passou a ser, como se ilustra na figura abaixo.



Aplique o padrão **Factory Method**, indicando o código:

- da *interface* `CitationFactory`
- das fábricas (classes) `IEECitationFactory` e `APACitationFactory`.

c) (1 valor) Exemplifique a **utilização destas fábricas**, instanciando uma citação do estilo APA e outra do estilo IEEE.

```
public static void main(String[] args) {

}
```

Q3 – Pattern Strategy

Considere a classe `X` da *Figura Q3*. Pretende-se aplicar o padrão *Strategy* de forma a retirar a estrutura `switch-case` do método `format`.

a) (0,5 valores) Complete a tabela abaixo que liga as classes/interfaces da solução final com os participantes do padrão Strategy.

Participante	**Classe/Interface**
	`X`
	`Main`
Strategy	`StrategyFormat`
ConcreteStrategy	

b) (1,5 valores) Apresente o código resultante da interface `StrategyFormat` e da classe `X`, após ter aplicado o padrão Strategy.

c) (1 valor) Forneça um método `main` onde ilustre a utilização das classes resultantes (`X` e outras criadas por si), mostrando o *output* resultante da aplicação dos diversos algoritmos de formatação (`format`) .
Nota: Deverá criar uma única instancia da classe X e fazer a alteração do algoritmo de formatação em tempo de execução.

```
public class Main {

    public static void main(String[] args) {

}
```

Q4 – Refatoring

Considere o código em anexo na *Figura Q4 - Refactoring*, onde constam originalmente as classes *Person* e *Astrology*.

a) (1,5 val) Preencha a tabela seguinte onde identifica a presença de *code smells* e as respetivas técnicas de refactoring a aplicar, se as considerar necessárias. Caso não se lembre de um termo específico, substitua-o por uma breve descrição.

Code Smell	**Linha(s) Código**	**Técnica de Refactoring**
Primitive Obsession		
Data Clump		
Duplicate Code		
Inappropriate Intimacy		
(Outros) Quais?		

b) (1,5) Produza o código resultante do seu refactoring, de acordo com o identificado na alínea anterior. Pode omitir *getters*, *setters* e código originalmente já omitido.

ANEXOS

Figura Q3

```java
public class X {
    public enum TYPE {A1, A2, A3};
    private String txt;
    private TYPE type;

    public X(String txt, TYPE type) {
        this.txt = txt;
        this.type = type;
    }

    public String format(String inc) {
        switch (type) {
            case A1:
                return txt + inc;
            case A2:
                return txt.substring(0, inc.length());
            case A3:
                return txt.toUpperCase() + "-" + inc.toUpperCase();
            default:
                throw new UnsupportedOperationException(type + " not supported");
        }
    }
}
```

```java
public class Main {

    public static void main(String[] args) {
        X x1 = new X("Bom dia ", X.TYPE.A3);
        System.out.println(x1.format("IPS"));
        X x2 = new X("Bom dia ", X.TYPE.A2);
        System.out.println(x2.format("IPS"));
    }
}
```

Figura Q4

```java
public class Person {
    public final String name;
    public final String dateOfBirth;

    public Person(String name, int day, int month, int year) {
        this.name = name;
        this.dateOfBirth = day+"/+"+month+"/"+year;
    }
}
```

```
11 public class Astrology {
12     private Person[] people;
13     private int peopleSize;
14
15     public Astrology() {
16         people = new Person[100];
17         peopleSize = 0;
18     }
19
20     public addPerson(String name, int day, int month, int year) {
21         if(peopleSize == 100) return;
22
23         int index = -1;
24         for(int i = 0; i < peopleSize; i++) {
25             if(people[i].name.equalsIgnoreCase(name)) {
26                 index = i;
27                 break;
28             }
29         }
30         if(index == -1) return;
31
32         people[peopleSize++] = new Person(name, day, month, year);
33     }
34
35     public String getZodiacSignOf(String name) {
36         int index = -1;
37         for(int i = 0; i < peopleSize; i++) {
38             if(people[i].name.equalsIgnoreCase(name)) {
39                 index = i;
40                 break;
41             }
42         }
43
44         if(index == -1) return "Person not found";
45
46         String sign = "";
47         String[] dmy = people[i].dateOfBirth.split("/");
48         int day = Integer.parseInt(dmy[0]);
49         int month = Integer.parseInt(dmy[1]);
50         if (month == 1) {
51          if (day < 20)
52             sign = "Capricorn";
53          else
54             sign = "Aquarius";
55         }
56         else if (month == 2) {
57             if (day < 19)
58                 sign = "Aquarius";
59             else
60                 sign = "Pisces";
61         }
62         /* ... other else ifs ommited, but present */
63         return "Sign of " + name + " is " + sign;
64     }
65 }
```