IPS Instituto Politécnico de Setúbal Escola Superior de Tecnologia de Setúbal	**Exame de Programação Avançada 2021/22** Exame de Época Normal 12 de fevereiro de 2022 Duração: 2h

Nome:_____ **Número:**_____

Q1(1.5)	Q2(1.5)	Q3(1.5)	Q4(1.5)	Q5(1.0)	Q6(2.0)
Q7(1.5)	Q8(2.0)	Q9(2.0)	Q10(1.5)	Q11(4.0)	**TOTAL**

Q1 - Padrão Iterator

(1 val) Considere o seguinte código parcial de uma implementação de `Stack` (que utiliza como estrutura de dados uma lista simplesmente ligada sem sentinelas):

```
public class StackLinked<E> implements Stack<E> {
    private Node top;

    private class Node{
        private E elem;
        private Node next;

        public Node(E elem, Node next) {
            this.elem = elem;
            this.next = next;
        }

    }

    public StackLinked() {
        top = null;
    }

    @Override
    public E pop() throws EmptyStackException {
        E elem = top.elem;
        top = top.next;
        return elem;
    }
    //outros métodos
```

```
    private class MyIterator implements Iterator<T> {
        private Node cursor;
        public MyIterator() {
           cursor=top;

        }
        @Override
        public boolean hasNext() {
             return cursor!=null;

        }
        @Override
        public T next() {
           T elem= cursor.element;
           cursor=cursor.next;
           return cursor;
        }
    }
}
```

Complete o código em falta na classe **MyIterator** por forma a que este iterador faça uma **travessia do topo para a base da pilha**.

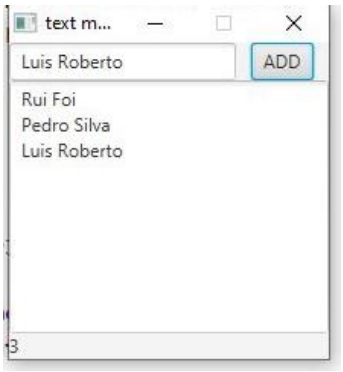
Q2 - Padrão Memento

(1,5 val) Considere o padrão de desenho Memento. Complete o código em falta na classe que assume o papel de *caretaker* que permite fazer "undo" sucessivos dos elementos guardados.

```
public class Caretaker {
    private  Stack<Memento> mementosCollection;
     private Originator originator;

    public Caretaker(Originator originator) {

      mementosCollection= new Stack<>();/*TODO 2*/
      this.originator=originator;

    }

    public void saveState() {

      mementosCollection.push(originator.createMemnto());/*TODO 3*/


    }

    public void restoreState() throws NoMementoException {
        if (mementosCollection.isEmpty()) {
            throw new NoMementoException();
        }

      this.originator.setMemento( mementosCollection.pop()));
                                /*TODO 4*/

    }

}
```

Q3 - Padrão MVC

(1,5 val) Considere o código relativo à implementação do padrão MVC numa aplicação que tem como funcionalidade ir construindo uma sequência de palavras cada vez que se aciona o botão ADD.



Complete o código em falta - abaixo das zonas *//TODO*.

```
public class Main extends Application {

        public static void main(String[] args) {
            launch(args);
        }

        @Override
        public void start(Stage stage) throws Exception {
            //TODO

            Document doc= new Document("doc1");
            DocumentPanel panel= new DocumentPanel(doc);
            DocumentControler ctr= new DocumentController(doc,panel);
            Scene scene = new Scene(panel,200,200);
            stage.setTitle("text make");
            stage.setScene(scene);
            stage.setResizable(false);
            stage.show();
        }
}
public class DocumentController {
    public Document document;
    public DocumentPanel documentPanel;

    public DocumentController(Document document, DocumentPanel documentPanel) {
        this.document = document;
        this.documentPanel = documentPanel;
  //TODO
       document.addObserver( documentPanel);
       documentPanel.setTriigers(this);

}
public void doAdd() {  //TODO
        String str= documentPanel.getInput();
        document.addWords(str);
}
```

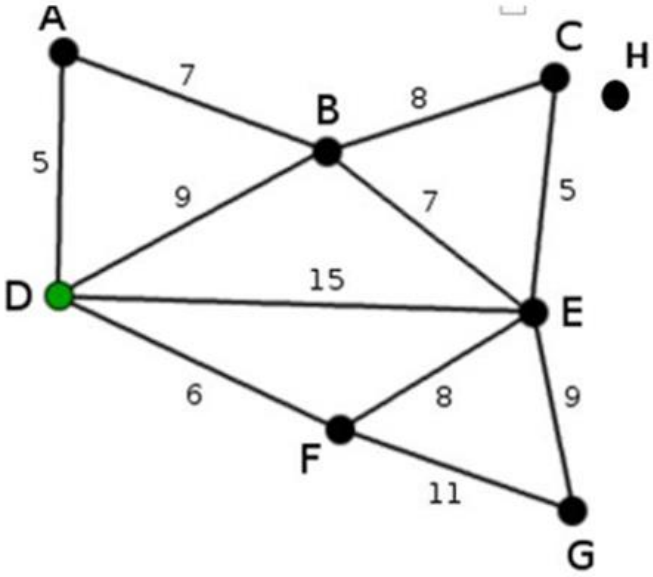
```
  }
}
public class DocumentPanel extends BorderPane implements Observer {
    private Document document;
    private Button btn1;
    private Label lblCounter;
    private TextField wordField;
    private TextArea textArea;

    public DocumentPanel(Document document) {
        this.document = document;
        btn1= new Button("ADD");
        wordField= new TextField("    ");
        HBox btnPane= new HBox(10);
        btnPane.getChildren().addAll(wordField,btn1);
        lblCounter= new Label(document.getCount()+"");
        textArea= new TextArea(document.getFormatWordsList());
        setTop(btnPane); setCenter(textArea);setBottom(lblCounter);
    }

    public void setTriggers(DocumentController ctrl){
        btn1.setOnAction((ActionEvent event) -> { ctrl.doAdd();});
    }

    @Override
    public void update(Object obj) { //TOD

        textArea.setText(document.formatWordList());
        lblCounter.setText(document.getCount() +"");
        wordField.setText(" ");
    }

    public String getInput() {
         return wordField.getText();
    }
}

public class Document extends Subject {
    private String name;
    private List<String> words;


    public Document(String name) {
        this.name = name;
        words= new ArrayList<>();
    }
    public void addWords(String word){  //TODO
         words.add(word);
         notifyObservers(null);
    }
    public int getCount() { return words.size(); }

    public String formatWordsList(){
        String str="";
        for(String word: words)
            str += word + " \n";
        return str;
    }
```

```
}
```

Q4 - Dijkstra

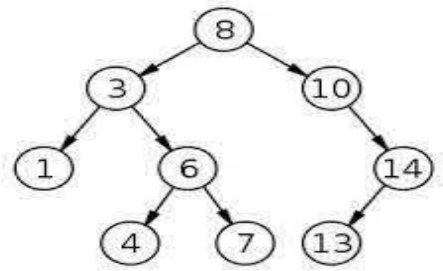
(1,5 val) Aplique o algoritmo de *Dijkstra* sobre o grafo não-orientado e valorado da figura, a partir do vértice de origem D e preencha a tabela resultante.

Vértice	Distancia	Predecessor
A	5	D
B	9	D
C	17	B
D	0	
E	14	F
F	6	D
G	17	F
H	∞	



Q5 - Binary Tree

(1 val) Considere a seguinte árvore binária de pesquisa. Apresente o resultado de a percorrer em **pré-order**



R: **8,3,1,6,4,7,10,14,13**

Q6 - Grafos

Considere a rede social Facebook e a necessidade de modelar utilizadores e relações de amizade utilizando grafos. Sobre um utilizador sabe-se o *username*, *email* e data de adesão à rede; sobre uma relação, a data em que foi estabelecida.

a) (0,5 val) Qual o tipo de grafo mais apropriado para representar esta rede (grafo ou digrafo)? Justifique.

Grafo, porque as relações entre amigos são do tipo bidirecional.

b) (0,5 val) Forneça a assinatura e atributos das classes cujas instâncias armazenaria nos vértices e nas arestas:

Tipo em vértice:

Tipo em aresta:

c) (1 val) Em função das classes propostas nas alíneas a) e b), complete o código abaixo por forma que o método `friendAfterDate` devolva o número de amigos que um *utilizador* fez após uma determinada data. Pode indicar/utilizar (sem implementar) um método qualquer que compare datas.

```
public class NetworkManager {

private                                    network;

//...construtor e outros métodos

public int friendsAfterDate(                                user) {
```

```
public class User{
    private String username;
    private String email;
    private Date dataInitial;
}

public class Friendship{
    private int id;
    private Date date;
}

public class NetworkManager {

private   Graph< User,Friendship> network;
//...construtor e outros métodos
public int friendsAfterDate( Date date, User user) {
    int count=0;
    for( Edge<Friendship,User> edge: network.edges()){
        if( (edge.vertices()[0].element().equals(user) || edge.vertices()[1].element().equals(user)) &&
                edge.element.getDate().compareTo(date)>0))
                    count++;
    }
    return count;
}
```

Q7 - BST

Considere a classe `BST (que representa uma árvore binária de pesquisa)` e que armazena *inteiros*:

```
public class BST {
    private Node root;

    //...
    private class Node {
        int elem;
        Node left;
        Node right;
        //...
    }

    public int countSingularNodes() { /* … */ }

}
```

(1,5 val) Forneça o código do método `countSingularNodes` que retorna o número de nós que só tem uma subárvore. Recomenda-se uma abordagem recursiva e pode implementar métodos auxiliares.

```
public int countSingularNodes() {
    return countS(root);
}

private int countS(Node node){
    int count=0;
    if (node == null) return count;
    if (( node.left!= null && node.right==null) || (node.left== null && node.right!=null))
        count++;
    return count + countS(node.left) + countS(node.right);
}
```

Q8 - Padrão Strategy

Considere a seguinte classe `NumberSequence`:

```
public class NumberSequence {
    private List<Integer> s = new ArrayList<>();
    public void add(int num) { s.add(num); }

    public int calcStatistic(char op){
        switch(op){
            case 'a':
                if(s.isEmpty()) throw new SequenceException("Empty sequence.");
                Collections.sort(s, Collections.reverseOrder());
                return (s.get(0) + s.get(s.size()-1))/2;
            case 'b':
                if(s.isEmpty()) throw new SequenceException("Empty sequence.");
                Collections.sort(s);
                return s.get(s.size()-1/2);
            default: throw new IllegalArgumentException ("Invalid statistic.”}

    }

}
```

a) (1 val) Aplique o padrão *Strategy* de forma a retirar o switch case do método `calcStatistic`. Apresente o código das classes e interfaces resultantes) incluindo a classe NumberSequence).

```
public interface Strategy{
    int calculate(List<Integer> s);
}

public class StrategyA implemenhts Strategy{
    public int calculate(List<Integer> s){
        if(s.isEmpty()) throw new SequenceException("Empty sequence.");
            Collections.sort(s, Collections.reverseOrder());
            return (s.get(0) + s.get(s.size()-1))/2;
    }
}

public class StrategyB implemenhts Strategy{
    public int calculate(List<Integer> s){
        if(s.isEmpty()) throw new SequenceException("Empty sequence.");
            Collections.sort(s);
            return s.get(s.size()-1/2);
    }

}
```

```
public class StrategyB implemenhts Strategy{
    public int calculate(List<Integer> s){
        if(s.isEmpty()) throw new SequenceException("Empty sequence.");
            Collections.sort(s);
            return s.get(s.size()-1/2);
    }

}

public class NumberSequence {
    private List<Integer> s = new ArrayList<>();
    private Strategy strategy=null;

    public void setStrategy(Strategy st){
        this.strategy=st;
    }

    public void add(int num) { s.add(num); }

    public int calcStatistic(char op){
        if(st==null) throw new IllegalArgumentException ("Invalid statistic.");
        return st.calculate(s);
    }
}
```

b) (0,5 val) Para cada uma das classes/interfaces implementadas, indique a que participante do padrão correspondem:

Classe/interface	Participante
NumberSequence	Context
Strategy	Strategy
StrategyA	ConcreteStrategy
StrategyB	ConcreteStrategy

c) (0,5 val) Forneça um método *main* onde ilustre a utilização das classes resultantes (`NumberSequence` e as outras criadas por si), fazendo o *output* do cálculo das duas opções.

```
public void main(String[] args){
     NumberSequence ns= new NumberSequence();
     ns.setStrategy(new StrategyA());
     ns.add(10);ns.add(5);ns.add(30);ns.add(20);
     System.out.println(ns.calcStatistic());
     ns.setStrategy(new StrategyB());
     System.out.println(ns.calcStatistic());
}
```

Q9 – Factory Method

Considere o código abaixo, que implementa o padrão *Factory Method*:

```
public interface Style {
    public A create(String type, String ...fields);
}

public class Z implements Style {
    @Override
    public A create(String type, String... fields) {
        switch (type) {
            case "xx":
                return new X(fields[0]);
            case "yy":
                return new Y(fields[0], fields[1]);
            default:
                throw new IllegalArgumentException("Does not exist : " + type);
        }
    }

public abstract class A {
    private String name;
    private String content;

    public A(String name) {
        this.name = name;
        this.content = "";
    }
    //getters e setters
}
```

```
public class X extends A{
    private Date date;

    public X(String name) {
        super(name);
        this.date= new Date();
    }
    @Override
    public String toString() {
        return date + "\n" + getName() + "\n" + getContent();
    }
}

public class Y extends A{
    private String dst;

    public Y(String name, String dst) {
        super(name);
        this.dst = dst;
    }

    @Override
    public String toString() {
        return dst + "\n" + getContent() + "\n\t" + getName();
    }
}
```

a) (0,5 val) Para cada uma das classes/interfaces apresentadas, indique a que participante do padrão correspondem:

Classe/interface	Participante
A	Product
X	ConcreteProduct
Y	ConcreteProduct
Z	ConcreteCreator

b) (0,5) Aplicando o padrão *Factory Method* - disponibilizado nas classes acima, complete o *main* de forma a obter o seguinte output:

```
AAA
mm1 mm2
    RRR
```

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

        A a = new Z().create("Y","AAA","RRR");
        a.setContent("mm1 mm2");
        System.out.println(a);
    }
}
```

c) (1 val) Pretende implementar um novo Style, denominado W.
Apresente (apenas) **as assinaturas: (i)** da classe W e **(ii)** do(s) método(s) contido(s) nessa classe.

```
public class W implements Style {
    @Override
    public A create(String type, String... fields) {


}
```

Q10 – Abstract Factory

Considere o padrão *Abstract Factory*.

a) (0,5 val) Indique em que categoria se insere? (Criação, Comportamental, Estrutural)

Criação

b) (0,5 val) Qual o **problema** que esse padrão se propõe resolver?

Resolve o problema da criação de objetos de famílias de produtos sem especificar as suas classes concretas. Permitindo resolver problema de acoplamento entre as classes, tornando a classe que precisa de objetos independentes das classes dos objectos concretos que necessita.

c) (0,5 val) Faça uma comparação entre esse padrão e o padrão *Factory Method*.

O Factory method é aplicado quando se tem apenas uma hierarquia de produtos, enquanto que o AbstarctFactory é uma extensão do Factory Method, suportando a criação de várias famílias de produtos.

No Factory Method a entidade criadora é denominada Cretaor, e instanciada através de um interface , com um método criador.

```
public interface Creator{

       public Product create (…..);
```

```
}
```

No AbstractFactory a entidade criadora (AbstractFactory) tem a capacidade de criar vários tipos de Produtos de famílias distintas entre si.

```
public interface AbstractFactory{

public ProductA createA (…..);

public ProductB createB (…..);

public ProductC createC (…..);


}
```

Q11 - Refatoring

Considere o código da figura 1 (ver última página).

a) (1 val) Identifique os seguintes *bad smells*, indicando as linha(s) onde ocorrem.

Bad smell	**Linha(s)**
Temporary Field	6
Data Clump	(4,5) 14, 25
Primitive Obsession	(4,5) (10,11)
Magic Number	10 e 11
Duplicate Code	(15-16) com (26-27)

b) (1 val) Para cada um dos *bad smells* indique qual a técnica que aplicaria (descreva a mesma numa frase).

- Temporary Field:

  Replace field with local variable. -> substituir o atributo cheapesatIndex por uma variável local, ao método.

- Data Clump:

  Extract Class- Criar uma classe producto, com os fields name e price.

- Primitive Obsession:

Replace Array by ArrayList – Substituir o array Product[] por ArrayList<Product>

Nota: Isto seria feito após aplicar o extract class.

- Duplicate Code:

  Extract method – Extrair as linhas duplicadas, para um método search().

c) (0,5 val) Apresente o método construtor **após** a aplicação das técnicas identificadas em b).

```
public Inventory()
    products= new ArrayList();
}
```

d) (1,5 val) Apresente o método **getCheapestProduct após** a aplicação das técnicas identificadas em b).

```
public String getCheapestProduct(){
    if( products.isEmpty() return "None";
    double min= products.get(0).getPrice();
    int cheapestIndex=0;
    for(int i=0; i< products.size(), i++){
        if(products.get(i).getPrice() < min){
            min= products.get(i).getPrice();
            cheapestIndex=i;
    }
    return products.get(i).getName();
    }
}
```

```java
package com.pa;

public class Inventory {
    private String[] itemNames;
    private double[] itemPrices;
    private int cheapestIndex;
    private int size;

    public Inventory() {
        itemNames = new String[100];
        itemPrices = new double[100];
        size = 0;
    }
    public boolean addProduct(String name, double price) {
        for(int i=0; i<size; i++) {
            if(name.compareToIgnoreCase(itemNames[i] )== 0) {
               return false;
            }
        }
        itemNames[size] = name;
        itemPrices[size] = price;
        size++;
        return true;
    }
    public boolean updatePrice(String name, double price) {
        for(int i=0; i<size; i++) {
            if(name.compareToIgnoreCase(itemNames[i] )== 0){
                itemPrices[i] = price;
                return true;
            }
        }
        return false;
    }
    public String getCheapestProduct() {
        if(size == 0) return "None";
        double min = itemPrices[0];
        cheapestIndex = 0;
        for(int i=0; i<size; i++) {
            if(itemPrices[i] < min) {
                min = itemPrices[i];
                cheapestIndex = i;
            }
        }
        return itemNames[cheapestIndex];
    }
}
```

Figura 1 – Código para *refactoring*.

(FIM DO ENUNCIADO)