IPS Instituto Politécnico de Setúbal
Escola Superior de Tecnologia de Setúbal

Programação Avançada

Ano Letivo de 2020/21

6 de Janeiro de 2021

Teste

- O teste tem a duração de 2H ;
- O teste tem de ser respondido no enunciado nas zonas afetas às respostas;
- Os alunos que desistam só podem sair decorridos 30min e têm de assinar "desisto" no enunciado;
- Todas as implementações solicitadas terão de ser efetuadas na linguagem Java.

Número do aluno: (preencha também em cada folha no rodapé)
Nome (em maiúsculas):

Grelha de Avaliação **(a preencher pelo docente**):

1.1	1.2	1.3	1.4	1.5	1.6		
2.1	2.2	2.3	2.4				
						TOTAL	

GRUPO 1 – Resposta Curta (8 valores)

Responda às questões nas zonas atribuídas.

1. (1val) Complete o algoritmo COUNT_INTERNAL de forma a este calcular o número de nós internos de uma árvore:

```
COUNT_INTERNAL(tree)
    IF IS_EMPTY(tree) THEN
        RETURN 0
    ENDIF
    count <- 0
    IF IS_INTERNAL(tree.root) THEN
        [instrucao 1]
    END IF
    FOR EACH child FROM tree.children
      [instrucao 2]
    END FOR
    RETURN [instrucao3]
```

R:

0.25 [instrução 1] – count < - 1 (ou count <- count +1)

0.5 [instrução 2] – count <- count + COUNT_INTERNAL(child)

0.25 [instrucao 3] – count

Numero:_____

2. (1val) Considere a *travessia de grafos* e em particular a travessia **depth-first**, cujo algoritmo se apresenta de seguida:

```
DFS(Graph,vértice_raiz)
    Marque vértice_raiz como visitado
    Coloque o vértice_raiz na pilha
    Enquanto a pilha não está vazia faça:
        - seja v o vértice que retira da pilha
        - processe v
        Para cada vértice w adjacente a v faça:
             Se w não está marcado como visitado então:
              - marque w como visitado
              - insira w na pilha
```

Complete a seguinte implementação em Java, fornecendo o código das linhas em falta.

```
public void DFS(Graph<V,E> graph, Vertex<V> origin) {
    Stack< /* A */ > stack = new Stack<>();
    List< /* A */ > visited = new ArrayList<>();
    visited.add(origin);
    stack.push(origin);
    while( /* B */ ) {
        /* C */
        process(v);
        for( /* D */ ) {
            if(!visited.contains(v)) {
                visited.add(v);
                /* E */
            }
        }
    }
}
```

R:

A – Vertex<V>

B – !stack.isEmpty()

C – Vertex<V> v = stack.pop();

D – for(Edge<E,V> e : graph.incidentEdges(v)) { Vertex<V> w = graph.opposite(v, e);

E – stack.push(v);

3. (2val) Considere um problema de representação de uma rede *peer-to-peer* . Cada nó desta rede consiste num computador do qual se sabe o seu *IP* (texto, e.g., "168.0.0.1"). Cada computador pode ligar-se a outros computadores através de uma ligação *TCP/IP* da qual se conhece a velocidade de transmissão em Mbit (e.g., 100) e o valor de *ping* em milisegundos (e.g., 14). Note que numa ligação *TCP/IP* ambos os computadores podem enviar/receber dados em simultâneo.

a) 0.5 b) 0.5

a) Se formos representar este problema utilizando grafos, utilizaria de um **dígrafo** ou um **grafo** (não direcionado) para representar este problema? **Justifique**.

R: Sendo uma representação da rede, o grafo é mais apropriado pois uma ligação entre dois computadores é automaticamente bidirecional, i.e., "ambos os computadores podem enviar/receber

dados em simúltaneo". A utilização de um dígrafo implicaria sempre um par de arestas entre dois vértices, o que poderia dar a entender que seria possível conexões só num sentido, o que não é verdade.

b) Forneça as **assinaturas e atributos** das classes envolvidas para representar;

i. O tipo V a armazenar nos vértices:

```
public class Computer {
        private String ip;

}
```

i. O tipo E a armazenar nas arestas:

```
public class Link {
        private int speed ;
        private int ping;

}
```

4. (1val) Considere a *interface* Dao em anexo. Considere também uma aplicação, onde o padrão respetivo será aplicado, que permita manipular/persistir os alunos *alumni* de uma única instituição. Acerca de cada aluno é apenas sabido o seu número, nome e média final de curso.
Forneça a **assinatura e atributos** da classe `AlunoAlumni` e a *interface* `AlumniDao` que, para além de todas as operações de Dao, deve disponibilizar uma operação para obter todos os alunos cuja média se encontre num intervalo dado fornecido através de dois argumentos `min` e `max`).

R:

```
public class AlunoAlumni {                  //0.25
        private String numero;
        private String nome;
        private double mediaCurso;
}

//o Dao persiste instancias de AlunoAluni e a chave é do mesmo tipo de 'numero' (identificador)

public interface AlumniDao extends Dao<AlunoAlumni, String> {       //0.75

        //todos os métodos herdados da interface Dao

        Collection<AlunoAlumni> getAllStudentsWithAverageBetween(double min, double max);
}
```


Numero:_____

5. (1val) Considere o padrão **Memento** e a classe `Bingo` cujo estado se pretende guardar ao longo do tempo. Complete o código dos métodos `createMemento` e `setMemento` e da *inner class* `MyMemento`.

```
public class Bingo implements Originator{
    private int numeroSerie;
    private final List<Integer> numeros;

    //...
    @Override
    public Memento createMemento() { //0.25
        return new MyMemento();

    }
    @Override
    public void setMemento(Memento saved) { //0.25
        /* atente ao modificador 'final' */
        MyMemento memento = (MyMemento)saved;

        this.numeroSerie = memento.numeroSerie;
        this.numeros.clear(); //numeros é final, nao posso "substituir"

        this.numeros.addAll(memento.numeros);

    }

    private class MyMemento implements Memento { //0.5
        //vou considerar o número de série tb, pois não é final.
        //mas se só considerassem a lista de números dava cotação total

        private int numeroSerie;

        private int List<Integer> numeros;

        public MyMemento() { //se passarem por construtor tb está ok.

            this.numeroSerie = numeroSerie; //inner class tem acesso

            this.numeros = new ArrayList<>(numeros); //cópia da lista!!

          }
    }

    public interface Memento {
        /* propositadamente vazia */
    }
```

6. Considere o código do Anexo onde as classes A, B e C, implementam o padrão MVC

   a) (1val) Indique o papel assumido por cada uma das classes

   Classe A – Controller

   Classe B – Model

   Classe C – View

1.0– tudo OK 0 cc

b) (1val) Complete o método main de forma a executar a aplicação MVC.

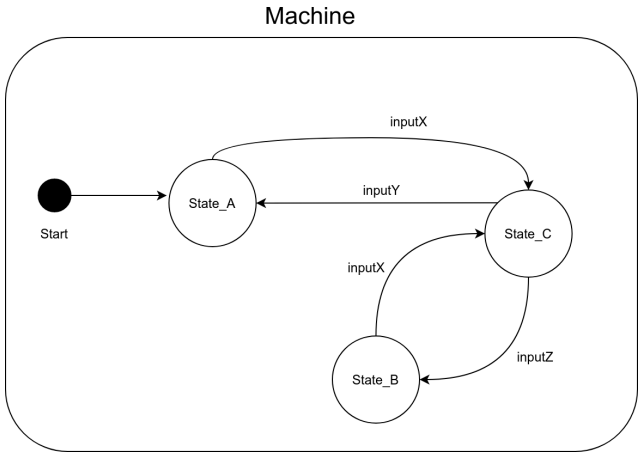
```
public static void main(String[] args) {

        B b= new B(10);
        C c= new C(b);
        A a= new A(b,c);
        a.execute();
    }
```

0.5 instanciação , 0.5 invocação métodos

GRUPO 2 – Desenvolvimento (12 valores)

1. (3val) Utilizando o padrão de desenho **State**, implemente a lógica da classe `Machine` ilustrada na imagem seguinte:



Se no diagrama um determinado estado não "responde" a um determinado input, significa que se mantém nesse estado.

```
public class Machine { //1.5val
        private State state; //interface ou classe abstrata.
        public Machine() {
                state = new State_A(this);
        }
        public void changeState(State s) {
                state = s;
        }
        public void inputX() { state.inputX(); } //delegar métodos para o estado atual
        public void inputY() { state.inputY(); }
        public void inputZ() { state.inputZ(); }
}
```


Numero:_____

```
public abstract class State { //0.5val
                        //a classe abstrata permite já conter o "context", caso contrário e utilizando
                                //uma interface, tem de ser duplicado para os estados concretos
        protected Machine context;
        public State(Machine context) { this.context = context; }
        public abstract void inputX(); //métodos delegados para os estados concretos
        public abstract void inputY();
        public abstract void inputZ();
}

//1val - classes State_A, B e C
public class State_A extends State {
        public State_A(Machine context) { super(context); }

        public void inputX() {
                context.changeState(new State_C(context));
        }
        public void inputY() { /* do nothing*/ }
        public void inputZ()  { /* do nothing*/ }
}

//A mesma lógica para as classes State_B e State_C e de forma a respeitar a maquina de estados
```

2. (3val) Pretende-se realizar uma aplicação, utilizando o padrão de desenho do tipo Factory, para criar instâncias de um **Jogo** em função da opção escolhida pelo utilizador. Considere que a classe Jogo e as suas 3 subclasses: `JogoGalo`, `JogoSolitario` e `JogoQuem` já se encontram implementadas.

   a) Qual dos padrões `Factory` estudados se aplica melhor a esta situação? **Justifique** a sua resposta.

   SimpleFactory, pois apenas temos hierarquia de Produto, e um único criador.

   (0.5 + 0.5)

   b) Para o padrão selecionado identifique cada um dos participantes.

   Product -Jogo (0.25)

   Concrete Product – JogoGalo, JogoSolitario, JogoQuem (0.25)

   Client – 0.25

   Factory – 0.25

   c) **Complete** o código em falta no método `main` e **escreva o código** relativo à(s) classe(s) em falta:

```
public class Main {
    public static void main(String[] args) {
      char op;
      Jogo jogo;
        do {
            System.out.println("menu");
            System.out.println("A - JogoGalo");
            System.out.println("B - JogoSolitario");
            System.out.println("C - JogoQuem");
            System.out.println("Q - Quit");
            System.out.println("Introduz a opção >");
            op = readInput();
            if (op != 'Q') {
              //completar

              jogo=FactoryJogo.create(op);          [0,5]

              jogo.execute();

            }
        } while (op != 'Q');
    }
}
```


Numero:_____

[0.5]

```
public class FactoryJogo()
      public static Jogo create(char op){
        switch (op){
              case ´A´: return new JogoGalo();
              case ´B´: return new JogSolitario();
              case ´C´: return new JogoQuem();
              default: throw new ArgumentException(“ Not Allowed”);
        }
}
```

3. (4val) Considere a classe MapBSTree em anexo, que é uma implementação parcial do ADT Map usando como estrutura de dados uma árvore binária de pesquisa.

a) Com base no código anterior, forneça a implementação do método get :

```
/**
* Returns the value to which the specified key is mapped, or null if this map
contains no mapping for the key.
* If this map permits null values, then a return value of null does not
necessarily indicate that the map contains * no mapping for the key; it's also
possible that the map explicitly maps the key to null.
* @param key the key whose associated value is to be returned
* @return the value to which the specified key is mapped, or null if this map
contains no mapping for the key
* @throws NullPointerException if the specified key is null and this map does
not permit null keys (optional)
*/
```

```
public V get(K key) throws NullPointerException {
    if(key == null) throw new NullPointerException("This implementation does not support null keys.");

    BSTNode nodeForKey = searchNodeWithKey(key, this.root);
    if(nodeForKey == null) return null;
    else return nodeForKey.value;
}

private BSTNode searchNodeWithKey(K key, BSTNode treeRoot) {
    if( treeRoot == null ) return null;

    int comparison = key.compareTo(treeRoot.key);
    if( comparison == 0)
        return treeRoot;
    else if( comparison < 0) //search in left sub-tree
        return searchNodeWithKey(key, treeRoot.left);
    else //search in right sub-tree
        return searchNodeWithKey(key, treeRoot.right);
}
```

length : 964 lines : 27 Ln : 6 Col : 26 Sel : 0 | 0 Windows (CR LF) UTF-8

}

Verificações de excepção -0.5

Chamada da função recursiva - 0.5

Função recursiva -1.5

b) De forma a implementar um teste unitário para testar o método acima, complete a classe abaixo.

```
class MapBSTTest {
    private MapBST<Integer,String> map;

    @BeforeEach
    void setUp() {
        map= new MapBST<Integer, String>();
        map.put(2,"Value2");
        map.put(3,"Value3");
        map.put(4,"Value4");
        map.put(1,"Value1");
    }
```


Numero:_____

```
        @Test
        void get() {
                    assertEquals(“Value3”,map.get(3));
                    assertEquals(“Value2”,map.get(2));
                    assertEquals(“Value1”,map.get(1));
                    assertEquals(null, map.get(0));
                    assertThrows(NullPointerException.class, ()->
      map.get(null));}
```

Criterios – 0.5 – values normais (vários) 0.5 null 0.5 exception

4. (2val) Considere a classe `GraphAdjacencyList` em anexo. Contém uma implementação da *interface* `Graph` utilizando uma estrutura de dados baseada em lista de adjacências.

   Com base no código anterior, forneça a implementação do método `opposite` :

```
/**
* Given vertex v, return the opposite vertex at the other end
* of edge e.
*
* If e is not incident to v, returns null.
*
* @param v          vertex on one end of e
* @param e          edge connected to v
* @return           opposite vertex along e
* @exception InvalidVertexException
            if the vertex is invalid for the graph
* @exception InvalidEdgeException
            if the edge is invalid for the graph
*/
public Vertex<V> opposite(Vertex<V> v, Edge<E, V> e) {

      //validações != null omitidas

      if(!vertices.contains(v)) throw new InvalidVertexException();


      boolean found = false;

      for(MyVertex w : vertices) {

            if(w.incident.contains(w)) found = true;

      }

      if(!found) throw new InvalidEdgeException();

      // daqui para baixo 1,5 v | as validações anteriores 0,5val

      MyVertex mV = (MyVertex)v;

      if(!mV.incident.contains(e)) return null; //nao está conetada a v


      for(MyVertex w : vertices) {

            if(w.incident.contains(w) && w != v) return w;

      }


      //nao é esperado, mas temos de devolver alguma coisa

      return null;

}
```

(fim de enunciado)

Numero:_____