Started on	Sunday, 27 November 2022, 11:29 PM
State	Finished
Completed on	Sunday, 27 November 2022, 11:43 PM
Time taken	14 mins 9 secs
Grade	**8,00** out of 20,00 (**40**%)

Question 1

Correct

Mark 4,00 out of 4,00

Considere o seguinte código em Java:

```
int f1() {
    X<Integer> c = new X(1,2,3,4,5);
    int sum = 0;
    while(c.hasNext())
            sum += c.next();
    return sum;
}
```

Identifique a **afirmação correta**:

Select one:

a. `X` assume o papel de *ConcreteIterator* do padrão Iterator; ✔

b. `X` é uma classe que implementa a interface `Iterable`;

c. `X` é uma coleção *standard* do java;

d. Nenhuma está correta.

Question 2

Incorrect

Mark 0,00 out of 4,00

Considere o seguinte código parcial de uma implementação de `Stack`:

```
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 ?
        }
    }
}
```

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

Select one:

a. **A:** `cursor = 0` | **B:** `cursor < size - 1` | **C:** `elements[cursor++]`

b. **A:** `cursor = size - 1` | **B:** `cursor > 0` | **C:** `elements[cursor--]`

c. **A:** `cursor = size - 1` | **B:** `cursor >= 0` | **C:** `elements[cursor--]` ✖

d. **A:** `cursor = 0` | **B:** `cursor <= size - 1` | **C:** `elements[cursor++]`

Question 3

Incorrect

Mark 0,00 out of 4,00

Considere a seguinte implementação de `Tree` e o seu iterador:

```
public class TreeImpl<T> implements Tree<T> {
    private TreeNode root;
    //...
    @Override
    public Iterable<T> iterator() {
        return MyIterator();
    }

    private class MyIterator implements Iterator<T> {
        private List<Position<T>> list = new ArrayList<>();
        public MyIterator() {
            list.add(root);
        }
        @Override
        public boolean hasNext() {
            return !list.isEmpty();
        }
        @Override
        public T next() {
            if(!hasNext()) return null;
            Random r = new Random();
            Position<T> p = list.remove(r.nextInt(list.size()));
            for(Position<T> child : children(p)) {
                list.add(child);
            }
            return p.element();
        }
    }
}
```

Qual a travessia fornecida pelo iterador?

Select one:

- a. Travessia *depth-first*;
- b. Outra;
- c. Travessia *breadth-first*; ✖
- d. Travessia *em-ordem*;

Question 4

Correct

Mark 4,00 out of 4,00

Considere o seguinte código que utiliza o padrão *Strategy*:

```
public class BagOfWords {
    private List<String> words;
    private Sorting sort;

    public BagOfWords() {
        words = new ArrayList<>();
        sort = new SortingAscending();
    }

    public void changeSorting(Sorting s) {
        sort = s;
    }
    @Override
    public String toString() {
        sort.perform(words);
        return words.toString();
    }
    //...
}

public class Main {
    public static void main(String[] args) {
        BagOfWords bag = new BagOfWords();
        bag.put("Pattern");
        bag.put("Design");
        bag.put("Strategy");
        System.out.println(bag);
        bag.changeSorting(new SortingDescending());
        System.out.println(bag);
    }
}
```

Identifique **participantes** do padrão nesta aplicação do mesmo:

Select one:

a. **Client**: `BagOfWords` | **Context**: `Main` | **Strategy**: `SortingAscending` | **Concrete Strategy**: `SortingDescending`;

b. **Client**: `Main` | **Context**: `BagOfWords` | **Strategy**: `SortingAscending` | **Concrete Strategy**: `Sorting`;

c. **Client**: `BagOfWords` | **Context**: `Main` | **Strategy**: `Sorting` | **Concrete Strategy**: `SortingDescending`;

d. **Client**: `Main` | **Context**: `BagOfWords` | **Strategy**: `Sorting` | **Concrete Strategy**: `SortingDescending`; ✔

Question 5

Incorrect

Mark 0,00 out of 4,00

Considere o seguinte código que utiliza o padrão *Strategy*:

```
public class BagOfWords {
    private List<String> words;
    private Sorting sort;

    public BagOfWords() {
        words = new ArrayList<>();
        sort = new SortingAscending();
    }

    public void changeSorting(Sorting s) {
        sort = s;
    }
    @Override
    public String toString() {
        sort.perform(words);
        return words.toString();
    }
    //...
}

public class Main {
    public static void main(String[] args) {
        BagOfWords bag = new BagOfWords();
        bag.put("Pattern");
        bag.put("Design");
        bag.put("Strategy");
        bag.changeSorting(new SortingDescending());
        System.out.println(bag);
    }
}
```

Assinale a **afirmação falsa**:

Select one:

a. Só é possível alterar a estratégia utilizada em *tempo de compilação*;

b. É possível alterar a estratégia em *tempo de execução*; ✖

c. O *output* do programa é `{"Strategy","Pattern","Design"}`;

d. A *interface* `Sorting` contém o método `void perform(List<String> l)`;