Verificação e Validação de Software (VVS)
Departamento de Informática, Faculdade de Ciências da Universidade de Lisboa
Exame de 1ª época — 6 de Junho de 2014
Duração: **2:30** (tolerância de 30 minutos) — Cotação: **14 valores**

Comece cada parte (A a E) numa página separada.

Considere o seguinte método Java cdiff que conta o número de elementos diferentes num um array de inteiros. Por ex. cdiff retornará 4 para o vector {1,2,1,2,3,0,3}.

```
1  public static int cdiff(int[] v) {
2     if (v == null) throw new IllegalArgumentException();
3     int count = 0;
4     for (int i = 0; i < v.length; i++) {
5        boolean diff = true;
6        for (int j = 0; j < i && diff == true; j++)  {
7          if (v[j] == v[i]) diff = false;
8        }
9        if (diff == true) count++;
10    }
11    return count;
12 }
```

A. Cobertura por grafos [4 valores]

(1) Desenhe o grafo de controlo de fluxo de cdiff.

(2) Identifique os pares definição-uso e os caminhos-du para todas as variáveis.

(3) Caracterize testes para uma cobertura de todos-os-usos (AUC: "All-Uses Coverage"). Assinale os requisitos inviáveis que existirem e se estes podem ser satisfeitos usando testes com visita por "side-trips".

(4) Identifique e explique as relações de subsunção entre cobertura de todos-os-usos, cobertura de todas-as-definições (ADC: "All-Defs Coverage"), e cobertura de caminhos primos (PPC: "Prime Path Coverage").

B. Testes baseados em lógica [1.25 valores]

(1) Identifique os predicados e cláusulas em cDiff.

(2) Caracterize testes para uma cobertura por cláusulas (CC: "Clause Coverage").

(3) Explique se os testes da alínea anterior também satisfazem uma cobertura de predicados ("Predicate Coverage") e uma cobertura de cláusulas activas geral (GACC: "General Active Clause Coverage"). Se necessário defina testes adicionais para satisfazer estes dois critérios.

C. Testes de mutação [3.75 valores]

Considere as seguintes mutações sobre cdiff:

$M1$ (l. 4)	i = 0	$\longrightarrow$	i = 1	$M2$ (l. 4)	i < v.length	$\longrightarrow$	i <= v.length
$M3$ (l. 6)	j < i	$\longrightarrow$	j <= i	$M4$ (l. 6)	diff == **true**	$\longrightarrow$	**true**
$M5$ (l. 9)	count++	$\longrightarrow$	count−−	$M6$ (l. 11)	**return** count	$\longrightarrow$	**return** 0

(1) Caracterize testes que matam (na forma forte) cada mutante, quando tal for possível. Justifique a escolhas de testes e se há mutantes que não podem ser ser mortos.

(2) Para um dos mutantes, caracterize um teste que executa a mutação associada mas não leva a um erro no estado de execução. Justifique a sua escolha.

(3) Para um dos mutantes, caracterize um teste que causa um erro, mas não uma falha (morte fraca). Justifique a sua escolha.

(4) Indique três propriedades desejáveis para um operador de mutação de programas.

Considere o seguinte esqueleto para a API de uma "stack" (pilha) com a usual disciplina "last-in, first-out". Um objecto Stack tem uma capacidade fixa maior ou igual a 1, definida pelo argumento do constructor, e disponibilisa os métodos usuais push(), pop(), e peek(), além de size() e capacity(). Assuma que as excepções IllegalArgumentException, EmptyStackException, e FullStackException podem ser lançadas da forma descrita nos comentários Javadoc.

```
public class Stack {
  // Construtor.
  // @param capacity Capacidade da stack
  // @throws IllegalArgumentException se capacity <= 0.
  public Stack(int capacity) { ... }
  // @return Capacidade da stack.
  public int capacity() { ... }
  // @return Tamanho (usado) da stack.
  public int size() { ... }
  // Adiciona elemento.
  // @param elem Elemento a adicionar.
  // @throws FullStackException se stack se encontra cheia.
  public void push(Object elem)  throws FullStackException { ... }
  // Retira elemento da stack.
  // @return Elemento removido do topo da stack.
  // @throws EmptyStackException se stack se encontra vazia.
  public Object pop() throws EmptyStackException { ... }
  // Devolve elemento no topo da stack, sem a modificar.
  // @return Elemento no topo da stack.
  // @throws EmptyStackException se stack se encontra vazia.
  public Object peek() throws EmptyStackException { ... }
}
```

D. Partição do espaço de entrada [3.75 valores]

Considere as seguintes duas características e respectivos blocos para testes sobre a classe Stack.
— **CAPACITY** : Capacidade da stack com blocos: **ONE**: Igual a 1; **TWO**: Igual a 2; **TWO_OR_MORE**: Maior ou igual a 2;
— **STATUS**: Uso da stack com blocos **EMPTY**: Vazia; **FULL**: Cheia;

(1) Explique o problema em cada uma das características acima. Sugira uma revisão das mesmas.

(2) Considerando a sua revisão, identifique requisitos de teste aplicando o critério de uma escolha base (BCC: "Base Choice Coverage").

(3) Caracterize testes para validar push() e pop() que satisfaçam os requisitos BCC na forma de sequência de chamadas a métodos de Stack.

(4) Escreva os testes anteriores relativos a push() na forma de métodos JUnit.

E. Aspectos complementares [1.25 valores]

Dê resposta claras e sucintas às seguintes questões.

(1) O que são "mock objects"? Qual o seu propósito?

(2) O que são testes de integração, regressão e aceitação? Que situações motivam cada um destes tipos de testes?

(3) O que entende por controlabilidade e observabiidade em testes de software? Refira problemas inerentes a estas propriedades em testes de programas "multi-threaded".