Verificação e Validação de Software (VVS)
Departamento de Informática, Faculdade de Ciências da Universidade de Lisboa
Exame de época de recurso — 25 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 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 { ... }
}
```

A. Partição do espaço de entrada [4 valores]

Considere as seguintes três características e respectivos blocos para testes sobre a classe Stack.
— **CAPACITY** : Capacidade da stack com blocos: **ONE**: Igual a 1; **TWO**: Igual a 2; **THREE_OR_MORE**: Maior ou igual a 3;
— **STATUS**: Uso da stack com blocos **EMPTY**: Vazia; **NENF**: nem cheia nem vazia; **FULL**: Cheia;
— **REPEATED**: existência de elementos repetidos na stack com blocos **YES**: sim; **NO**: não (inclui o caso em que stack está vazia).

(1) Identifique os requisitos de teste para o critério de múltipla escolha base (MBCC: "Multiple Base Choice Coverage") usando as seguintes duas escolhas base: (THREE_OR_MORE, NENF, YES) e (TWO, FULL, NO).

(2) Supondo que os requisitos MBCC em **(1)** são satisfeitos num determinado cenário de teste, garantimos também uma cobertura de pares-de-escolhas (PWC: "Pair-Wise Coverage")? E uma cobertura de cada escolha (ECC: "Each Choice Coverage") ? Justifique.

(3) Caracterize testes para validar pop() para os requisitos MBCC em **(1)**, na forma de sequência de chamadas a métodos de Stack.

(4) Escreva código JUnit/Java para (apenas) dois dos testes da questão anterior.

Considere o método csum (listagem à esquerda) que calcula um vetor com as somas cumulativas dos elementos de um vector de inteiros dado, por ex. csum devolverá {1,3,3,6} para v={1,2,0,3}. Considere também o método m (listagem à direita).

```
1  static int[] csum(int[] v) {
2    if ( v == null || v.length == 0)
3      throw new IllegalArgumentException();
4    int[] r = new int[v.length];
5    for (int i = 0; i < v.length; i++) {
6      for (int j=0; j <= i; j++) {
7        r[i] = r[i] + v[j];
8      }
9    }
10   return r;
11 }
```

```
static int m(boolean flag, int[] a, int[] b) {
  int x = 0;
  int[] v;
  if (flag)
    v = a;
  else
    v = b;
  try {
    int[] c = csum(v);
    if (flag && (a[0]> 0 || a[0] == -1))
      x = c.length;
    x = x + c[0];
  } catch (IllegalArgumentException e) {
    e.printStackTrace();
  }
  return x;
}
```

B. Cobertura por grafos [4 valores]

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

(2) Identifique os caminhos primos no grafo de controlo de fluxo.

(3) Caracterize testes para uma cobertura dos caminhos primos. Para cada requisito inviável que encontrar, indique se este pode ser satisfeito por um teste com visita por "side-trips".

(4) Identifique os requisitos de teste para emparelhamentos "last-def" / "first-use" relativos à chamada a csum no corpo do método m (linha 9).

(5) Caracterize testes para os requisitos da questão anterior.

C. Cobertura Lógica [2.5 valores]

(1) Defina os requisitos de teste para uma cobertura de cláusulas activas geral (GACC: "General Active Clause Coverage") sobre os predicados das linhas 4 e 10 do método m.

(2) Caracterize testes para os requisitos da questão anterior.

(3) Os seus testes satisfazem (também) uma cobertura por cláulas activas restritas (RACC: "Restricted Active Clause Coverage")? Justifique.

D. Testes de mutação [2.5 valores]

Considere as seguintes mutações sobre csum:

$M1$ (l. 2)	v.length == 0	$\longrightarrow$	v.length <= 0	$M2$ (l. 5)	i = 0	$\longrightarrow$	i = 1
$M3$ (l. 5)	i < v.length	$\longrightarrow$	i <= v.length	$M4$ (l. 6)	j <= i	$\longrightarrow$	j < i
$M5$ (l. 6)	j <= i	$\longrightarrow$	j > i	$M6$ (l. 7)	r[i] + v[j]	$\longrightarrow$	r[i] − v[j]

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

(2) Para um dos mutantes mortos em **(1)**, caracterize um outro 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 mortos em **(1)**, caracterize um teste que causa um erro, mas não uma falha (morte fraca). Justifique a sua escolha.

E. Aspectos complementares [1 valor]

(1) Para testes de integração envolvendo vários componentes, qual é o propósito de uma análise CITO ("Class Integration Testing Order") e qual a relevância das dependências entre componentes neste contexto?

(2) O que são testes de sistema? A realização de testes de sistema dispensa a realização por sua vez de testes de integração ou de aceitação num processo de desenvolvimento de software? Justifique.