Departamento de Informática **Verificação Validação Software**
Faculdade de Ciências **Exame Dois**
Universidade de Lisboa **8 de Julho de 2010**

Inicie cada grupo numa nova página. Duração: duas horas e meia.

Grupo 1. [3 valores] Uma especificação descreve aspectos do comportamento que o *software* deve exibir. Este exercício versa o teste de especificações baseadas em estado, utilizando Máquinas de Estado Finitas. Considere então uma máquina de estados descrevendo um termóstato (de um edifício, por exemplo). As variáveis que definem o estado são as seguintes.

```
parteDoDia : {Manhã, Noite}
temperatura : {Alta, Baixa}
```

O estado inicial é {parteDoDia = Manhã, temperatura = Baixa}. As transições do sistema desencadeiam-se por efeito de 3 operações.

```
avançar // Avança para a parte seguinte do dia, circularmente
subir   // Faz a temperatura subir, se possível
descer  // Faz a temperatura descer, se possível
```

a) Apresente os estado do sistema.

b) Desenhe a máquina de estados através de um grafo onde os nós representam os conjuntos dos valores para as variáveis de estado, e os arcos definem as possíveis mudança de estado. Etiquete cada arco com o nome da operação correspondente.

c) Um caso de teste é uma sequência de operações. Apresente um caso de teste que satisfaça o Critério de Cobertura de Nós. Justifique que o caso de teste satisfaz o critério.

d) Apresente um caso de teste que satisfaça o Critério de Cobertura de Arcos. Justifique que o caso de teste satisfaz o critério.

Grupo 2. [3 valores] Considere o seguinte extracto de uma classe que implementa uma árvore de pesquisa auto-equilibrante, uma árvore AVL. O método que nos interessa, add, junta um elemento à árvore e devolve a nova árvore. Trata-se de um método privado, chamado de um método público com o mesmo nome que passa como argumento a raiz da árvore, para além do elemento a inserir.

```
1  /** @author Koffman and Wolfgang */
2  public class AVLTree <E extends Comparable <E>> {
3    private static class AVLNode <E> extends Node<E> {
4      /** Constant to indicate left-heavy */
5      public static final int LEFT_HEAVY = -1;
6      /** Constant to indicate balanced */
7      public static final int RIGHT_HEAVY = 1;
```

```
8       /** balance is right subtree height - left subtree height */
9       private int balance;
10      ...
11    }
12    /** Recursive add method. Inserts the given object into the tree.
13        post: addReturn is set true if the item is inserted,
14              false if the item is already in the tree.
15        @param localRoot The local root of the subtree
16        @param item The object to be inserted
17        @return The new local root of the subtree with the item
18                inserted
19    */
20    private AVLNode<E> add(AVLNode<E> localRoot, E item) {
21      if (localRoot == null) {
22        addReturn = true;
23        increase = true;
24        return new AVLNode<E> (item);
25      }
26      if (item.compareTo(localRoot.data) == 0) {
27        // Item is already in the tree.
28        increase = false;
29        addReturn = false;
30        return localRoot;
31      }
32      else if (item.compareTo(localRoot.data) < 0) {
33        // item < data
34        localRoot.left = add(localRoot.left, item);
35        if (increase) {
36          decrementBalance(localRoot);
37          if (localRoot.balance < AVLNode.LEFT_HEAVY) {
38            increase = false;
39            return rebalanceLeft(localRoot);
40          }
41        }
42        return localRoot; // Rebalance not needed.
43      }
44      else {
45        // item > data
46        localRoot.right = add(localRoot.right, item);
47        if (increase) {
48          incrementBalance(localRoot);
49          if (localRoot.balance > AVLNode.RIGHT_HEAVY) {
50            return rebalanceRight(localRoot);
51          }
52          else {
53            return localRoot;
54          }
55        }
56        else {
```

```
57            return localRoot;
58          }
59        }
60    }
61    ...
62  }
```

a) Identifique os predicados constantes no programa, através de o seus números de linha.

b) Analise o problema da acessibilidade (*reachability*), indicando para cada um dos predicados da alínea anterior o seu predicado de acessibilidade.

c) Indique valores para os parâmetros localRoot e item que satisfaçam a cobertura de predicados para cada predicado do programa.

d) Indique valores para os parâmetros que satisfaçam desta vez a cobertura de cláusula activa correlacionada.

e) Explique porque é que a cobertura de predicados (num esquema de cobertura baseada em lógica) é equivalente à cobertura de ramos (num esquema de cobertura baseada em grafos).

Grupo 3. [3 valores] Suponha que estamos a testar um módulo que permite aos utilizadores a inserção de novos identificadores numa base de dados. Este exercício foca a selecção de classes de equivalência para as entradas. A especificação estabelece que um identificador deve ter entre 3 e 15 caracteres alfanuméricos, dos quais os dois primeiros devem ser letras.

a) Identifique três características que sugiram partições.

b) Apresente os vários blocos em que se divide cada característica identificada acima.

c) Seleccione *uma* das partições identificada acima. Justifique que se trata de uma boa partição, isto é que satisfaz as propriedades da não sobreposição de blocos e da cobertura da característica pelos vários blocos.

d) Escolha um representante para cada bloco, sugerindo valores para o parâmetro.

e) Para cada característica, designe um dos seus blocos como *base*. Defina uma bateria de testes que satisfaça a cobertura de *escolha básica* (*base choice coverage*).

Grupo 4. [3 valores] Volte a considerar o código apresentado no grupo **2**.

a) De entre as seguintes categorias de mutantes: troca de operador relacional, troca de operador aritmético, troca de constante lógica, e troca de variável/atributo, sugira *três* mutantes para as linhas de código números 23, 34, 37.

b) Para cada um destes mutantes apresente um teste que *não* alcance o mutante. Quando tal for impossível explique porquê.

c) Para cada um dos mutantes descreva um teste que alcance o mutante mas que não provoque infecção. Quando tal for impossível explique porquê.

d) Para cada um dos mutantes descreva um teste que provoque infecção mas que não a propague. Quando tal for impossível explique porquê.

e) Para cada um dos mutantes descreva um teste que mate o mutante. Quando tal for impossível explique porquê.

Grupo 5. [3 valores] Será que o programa exibe um comportamento correcto para uma determinada entrada? Para *software* relativamente simples podemos ter uma ideia clara se a saída está ou não correcta. O caso geral não é assim tão simples. Estudámos quatro métodos para este problema: verificação directa (utilização de um programa para verificar a resposta), computação redundante (cálculo da resposta por um método alternativo), verificações de consistência (verificar se uma parte da resposta faz sentido, se satisfaz alguma propriedade de interesse) e dados redundantes (comparar resultados obtidos com entradas diferentes).

Considere o problema da verificação da correcção de uma rotina de ordenação de um vector de números inteiros, `void sort (int [] v)`.

a) Apresente um programa Java que efectue a verificação directa da propriedade.

b) Como procederia para se certificar que o programa acima verifica a propriedade requerida?

c) De entre os três restantes métodos, escolha um e aplique-o ao problema em mãos.

d) De entre todos os possíveis métodos qual escolheria para o nosso problema? Justique.