Verificação e Validação de Software (VVS), 2014/15
Departamento de Informática
Faculdade de Ciências da Universidade de Lisboa
Exame de 1ª época — 12 de Junho de 2015
Duração: **2:30** — Cotação: **14 valores**

Comece cada parte (A a E) numa página separada da folha de exame. Para cada caso de teste que caracterizar, identifique os requisitos cobertos pelo teste. Sempre que detectar um requisito de teste inviável, assinale-o e explique a razão para a inviabilidade.

O método Java hexaStrToInt (abaixo) converte uma string formada por dígitos hexadecimais, caracteres de '0' e '9' ou de 'A' a 'F', num valor de tipo **int**. Por exemplo, hexaStrToInt("1AF") devolve o inteiro $431 = 1 \times 16^2 + 10 \times 16 + 15 = 256 + 160 + 15$. O método lança IllegalArgumentException caso o argumento str seja **null**, ou "" (a string vazia), ou tiver mais de 8 caracteres; neste último caso, o possível número em questão não é representável em 32 bits, a precisão do tipo **int**. A mesma excepção é lançada quando um caractere na string não é um dígito hexadecimal.

```
1  public static int hexaStrToInt(String str) {
2      if (str == null || str.length() == 0 || str.length() > 8) {
3        throw new IllegalArgumentException("Invalid argument");
4      }
5      int value = 0;
6      for (int i = 0; i < str.length(); i++) {
7        char c = Character.toUpperCase(str.charAt(i));
8        int d;
9        if (c >= '0' && c <= '9') {
10         d = c - '0';
11       } else if (c >= 'A' && c <= 'F') {
12         d = 10 + (c - 'A');
13       } else  {
14         throw new IllegalArgumentException("Not a hex digit: "+ c);
15       }
16       value = value * 16 + d;
17     }
18     return value;
19   }
```

A. Cobertura por grafos [3 valores]

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

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

(3) Caracterize testes para uma cobertura de todos-os-caminhos-du (ADUPC: "All-Du-Paths coverage").

(4) Identifique uma variação (eventualmente um subconjunto) dos testes da alínea anterior que satisfaça uma cobertura de todos-os-usos (AUC: "All-uses coverage") mas não ADUPC. Justifique a sua escolha.

(código da primeira página repetido)

```
1  public static int hexaStrToInt(String str) {
2      if (str == null || str.length() == 0 || str.length() > 8) {
3        throw new IllegalArgumentException("Invalid argument");
4      }
5      int value = 0;
6      for (int i = 0; i < str.length(); i++) {
7        char c = Character.toUpperCase(str.charAt(i));
8        int d;
9        if (c >= '0' && c <= '9') {
10         d = c - '0';
11       } else if (c >= 'A' && c <= 'F') {
12         d = 10 + (c - 'A');
13       } else  {
14         throw new IllegalArgumentException("Not a hex digit: "+ c);
15       }
16       value = value * 16 + d;
17     }
18     return value;
19   }
```

B. Testes baseados em lógica [3 valores]

(1) Identifique os predicados e cláusulas em hexaStrToInt e condições de alcance para cada um dos predicados.

(2) Identifique os requisitos de testes para uma cobertura por cláusulas activas geral (GACC: "General Active Clause Coverage").

(3) Caracterize testes para os requisitos GACC da questão anterior.

(4) Os testes que propôs satisfazem ou não também uma cobertura de predicados (PC: "Predicate Coverage") ? E uma cobertura por cláusulas activas restrita (RACC: "Restricted Active Clause Coverage") ? Justifique.

C. Testes de mutação [3 valores]

Considere os mutantes definidos a seguir sobre o código de hexaStrToInt:

$M1$ (l. 2) str.length() == 0 $\longrightarrow$ str.length() != 0
$M2$ (l. 2) str.length() > 8 $\longrightarrow$ str.length() >= 9
$M3$ (l. 9) c <= '9' $\longrightarrow$ c < '9'
$M4$ (l. 11) c <= 'F' $\longrightarrow$ **false**
$M5$ (l. 12) 10 + (c − 'A') $\longrightarrow$ 10 − (c − 'A')
$M6$ (l. 6) **return** value $\longrightarrow$ **return** 0

(1) Para cada um dos mutantes caracterize se possível: **(a)** um teste que mata o mutante (na forma forte); **(b)** um teste que leva um estado de erro mas não a uma falha (morte fraca); **(c)** um teste que atinge a mutação mas não leva a um estado de erro. Justifique as suas respostas, explicando a execução do teste ou porque não é possível encontrar um teste nas condições estipuladas.

(2) Refira três propriedades desejáveis para um operador de mutação de programas.

Considere o seguinte esqueleto para a API de um saco ("bag"). Um saco tem associado um conjunto de elementos e um número de ocorrências por cada elemento nesse conjunto. Por exemplo, para um saco s com 4 elementos "A", 2 elementos "B" e 1 elemento "C", espera-se que s.size() == 7, s. distinct () == 3, s.count("A") == 4, s.count("B") == 2, e s.count("C") == 1.

```
public class MyBag {
  /** Construtor. O saco é inicializado vazio. */
  public MyBag() { ... }
  /** @return Número de elementos no total. */
  public int size() { ... }
  /** @return Número de elementos distintos. */
  public int distinct() { ... }
  /** @return Número de ocorrências de o (0 se não está no saco). */
  public int count(Object o) { ... }
  /** Adiciona uma ocorrência de o.
    * @return Numero de ocorrências de o (valor actualizado). */
  public int add(Object o) { ... }
  /** Remove uma ocorrência de o.
    * Se o não existir no saco, o método não tem efeito.
    * @return Numero de ocorrências de o (valor actualizado). */
  public int remove(Object o) { ... }
}
```

D. Partição do espaço de entrada [3 valores]

Considere as seguintes características e respectivos blocos para testes sobre os métodos add e remove na classe MyBag:
— **SIZE** – Número de elementos no total: **NONE**: 0; **ONE_TO_THREE**: 1 a 3; **MORE_THAN_THREE**: mais de 3;
— **DISTINCT** – Número de elementos distintos: **NONE**: 0; **ONE_OR_TWO**: 1 ou 2; **MORE_THAN_TWO**: mais de 2;
— **OCC** – Ocorrências de o no saco: **NONE**: 0; **ONE_OR_MORE**: 1 ou mais.

(1) Identifique requisitos de teste para o critério de uma escolha base (BCC: "Base Choice Coverage"), definindo a escolha base sem usar blocos NONE.

(2) Caracterize testes sobre add() e remove() para os requisitos BCC, na forma de sequências de chamadas a métodos de MyBag.

(3) Escreva dois dos testes anteriores em Java / JUnit: um teste que exercite add para um dos requisitos BCC e outro que exercite remove para um requisito BCC diferente do primeiro.

E. Aspectos complementares [2 valores]

(1) Para que serve a biblioteca Mockito? Que padrões de programação de testes podemos implementar com ela ? Em que tipos de teste esses padrões são úteis?

(2) O que entende por controlabilidade e observabilidade em testes de software? Que problemas de controlabilidade e observabilidade se colocam em programas "multi-threaded"? E em programas de tempo-real?