White box

December 6th, 2023
Ana Paiva, José Campos

1. Structural Testing (Line and Decision coverage)

For each of the following exercises, perform ‘Structural testing’. In a nutshell,
- Derive tests that exercise all lines of code.
- Compute Cyclomatic Complexity (CC) and derive more tests, if necessary.
- (Optional) Implement the derived tests in Java/JUnit.

1.1

Consider the method `mediaMParesExceptoN` that computes the average of `m` even integers (of a given array of integers) ignoring the first `n` even integers. For example,

```java
mediaMParesExceptoN([1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12], 3, 2);
```

ignores the first 2 even integers (i.e., 2 and 4) and computes the average of the following 3 even integers (i.e., 6, 8, and 10), 8.0.

```java
public static double mediaMParesExceptoN(int[] v, int m, int n) {
 int soma = 0;
 int conta = 0;

 for (int i = 0; i < v.length && conta < m; i++) {
  if (v[i] % 2 == 0) {
   if (n == 0) {
    soma += v[i];
    conta++;
   } else {
    n--;
   };
  }
```

```
 }

 return (double) soma / conta;

}
```

1.2

Consider the method `amplitude` that computes the amplitude of a given array of integers. That is, the difference between the maximum and minimum value in the array.

Java

```java
public static int amplitude(int[] v) {
 if (v == null) {
  throw new NullPointerException();
 }

 if (v.length == 0) {
  return 0;
 }

 int min = v[0];
 int max = v[0];

 for (int i = 1; i < v.length; i++) {
  if (v[i] > max) {
   max = v[i];
  } else if (v[i] < min) {
   min = v[i];
  }
 }
 return max - min;

}
```

2. Modified Condition/Decision Coverage (MC/DC)

For the following exercise, apply ‘Modified Condition/Decision Coverage (MC/DC)’. In a nutshell,
- Perform ‘Modified Condition/Decision Coverage (MC/DC)’ to all decisions in each of the following functions and derive tests that fulfills MC/DC.
- (Optional) Implement the derived tests in Java/JUnit.

The following method `hexaStrToInt` converts a string of hexadecimal digits, i.e., caracteres from `0` to `9` ou `A` to `F`, into an integer value. For example, `hexaStrToInt("1AF")` returns the integer value `431`. Note that the method throws an `IllegalArgumentException` if `str` is `null`, or `""` (empty string), or it has more than 8 characters, or if there is a character that is not a hexadecimal digit.

Java

```java
public static int hexaStrToInt(String str) {
 if (str == null || str.length() == 0 || str.length() > 8) {
  throw new IllegalArgumentException("Invalid argument");
 }

 int value = 0;
 for (int i = 0; i < str.length(); i++) {
  char c = Character.toUpperCase(str.chartAt(i));
  int d;
  if (c >= '0' && c <= '9') {
   d = c - '0';
  } else if (c >= 'A' && c <= 'F') {
   d = 10 + (c - 'A');
  } else {
   throw new IllegalArgumentException("Not a hex digit: " + c);
  }
  value = value * 16 + d;
 }

 return value;

}
```

3. Mutation testing

3.1

Consider the following function `pat` that determines if a given `pattern` occurs in a given array of characters (`subject`). The function returns the index of the first occurrence of the pattern or -1, otherwise.

Java

```java
public int pat(char[] subject, char[] pattern) {
  final int NOTFOUND = -1;
  int iSub = 0, rtnIndex = NOTFOUND;
  boolean isPat = false;
  int subjectLen = subject.length;
  int patternLen = pattern.length;

  while (isPat == false && iSub + patternLen - 1 < subjectLen) {
   if (subject[iSub] == pattern [0]) {
    rtnIndex = iSub; // Starting at zero
    isPat = true;

    for (int iPat = 1; iPat < patternLen; iPat++) {
     if (subject[iSub + iPat] != pattern[iPat]) {
      rtnIndex = NOTFOUND;
      isPat = false;
     break; // out of for loop
     }
    }
   }

   iSub++;
  }

 return rtnIndex;

}
```

- Suggest a mutant for lines of code 9, 14, and 22. You should use the following mutation operators: Relational Operator Replacement (ROR), Arithmetic Operator Replacement (AOR), and Scalar Variable Replacement (SVR).
- Suggest one test input (per mutant) that does not reach (i.e., exercises) the mutated line. If it is not possible to derive such a test, explain why.
- Suggest one test input (per mutant) that does reach (i.e., exercises) the mutated line but does not kill the mutant. If it is not possible to derive such a test, explain why.
- Suggest one test input (per mutant) that does reach (i.e., exercises) the mutated line and does kill the mutant. If it is not possible to derive such a test, explain why.

(Optional) Implement the derived tests in Java/JUnit.

3.2

Consider the function `mediaMParesExceptoN` that computes the average of `m` even integers (of a given array of integers) ignoring the first `n` even integers. For example,

Java
```
mediaMParesExceptoN([1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12], 3, 2);
```

ignores the first 2 even integers (i.e., 2 and 4) and computes the average of the following 3 even integers (i.e., 6, 8, and 10), 8.0.

Java
```
1. public static double mediaMParesExceptoN(int[] v, int m, int n) {
2.  int soma = 0;
3.  int conta = 0;
4.
5.  for (int i = 0; i < v.length && conta < m; i++) {
6.   if (v[i] % 2 == 0) {
7.    if (n == 0) {
8.     soma += v[i];
9.     conta++;
10.    } else {
11.     n--;
12.    };
13.   }
14.  }
```

```
15.
16.  return (double) soma / conta;
17. }
```

- Suggest a mutant for lines of code 5, 8, 11, and 16. You should use the following mutation operators: Relational Operator Replacement (ROR), Arithmetic Operator Replacement (AOR), and Scalar Variable Replacement (SVR).
- Suggest one test input (per mutant) that does not reach (i.e., exercises) the mutated line. If it is not possible to derive such a test, explain why.
- Suggest one test input (per mutant) that does reach (i.e., exercises) the mutated line but does not kill the mutant. If it is not possible to derive such a test, explain why.
- Suggest one test input (per mutant) that does reach (i.e., exercises) the mutated line and does kill the mutant. If it is not possible to derive such a test, explain why.

(Optional) Implement the derived tests in Java/JUnit.

3.3

Consider the following Java function (noRep) that checks if an array of integers contains no repeated elements.

Java

```java
1. public static boolean noRep(int[] v) {
2.  if (v == null) {
3.   throw new NullPointerException();
4.  }
5.
6.  for (int i = 0; i < v.length - 1; i++) {
7.   for (int j = i + 1; j < v.length; j++) {
8.    if (v[i] == v[j]) {
9.     return false;
10.    }
11.   }
12.  }
13.
14.  return true;
```

```
15. }
```

and the following mutants:

- m1, line 6

```
Unset
- for (int i = 0; i < v.length - 1; i++) {

+ for (int i = 0; i < v.length + 1; i++) {
```

- m2, line 7

```
Unset
- for (int j = i + 1; j < v.length; j++) {

+ for (int j = i + 1; j >= v.length; j++) {
```

- m3, line 7

```
Unset
- for (int j = i + 1; j < v.length; j++) {

+ for (int j = i + 1; j <= v.length; j++) {
```

- m4, line 8

```
Unset
- if (v[i] == v[j]) {

+ if (v[i] != v[j]) {
```

- m5, line 8

```
Unset
- if (v[i] == v[j]) {

+ if (v[i] <= v[j]) {
```

Suggest as many test inputs as necessary to kill all mutants. Ideally, one test should only kill one mutant. If it is not possible to kill any mutant, explain why.

(Optional) Implement the derived tests in Java/JUnit.

3.4

Consider the following Java function (`indexOfMin`) that computes the index of the first occurrence of the minimum value in an array of integers.

```java
Java
1. public static int indexOfMin(int[] v) {
2.  if (v == null || v.length == 0) {
3.   throw new IllegalArgumentException();
4.  }
5.
6.  int iMin = 0;
7.  for (int i = 1; i < v.length; i++) {
8.   if (v[i] < v[iMin]) {
9.    iMin = i;
10.   }
11.  }
12.
13.  return iMin;

14. }
```

and the following mutants:

- m1, line 2

```
Unset
- if (v == null || v.length == 0) {

+ if (v != null || v.length == 0) {
```

- m2, line 7

```
Unset
- for (int i = 1; i < v.length; i++) {

+ for (int i = 0; i < v.length; i++) {
```

- m3, line 7

```
Unset
- for (int i = 1; i < v.length; i++) {

+ for (int i = 1; i >= v.length; i++) {
```

- m4, line 8

```
Unset
- if (v[i] < v[iMin]) {

+ if (v[i] <= v[iMin]) {
```

- m5, line 8

```
Unset
- if (v[i] < v[iMin]) {

+ if (v[i] >= v[iMin]) {
```

Suggest as many test inputs as necessary to kill all mutants. Ideally, one test should only kill one mutant. If it is not possible to kill any mutant, explain why.

(Optional) Implement the derived tests in Java/JUnit.

3.5

Consider the function `amplitude` that computes the amplitude of a given array of integers. That is, the difference between the maximum and minimum value in the array.

Java

```
1. public static int amplitude(int[] v) {
2.  if (v == null) {
3.   throw new NullPointerException();
4.  }
5.
6.  if (v.length == 0) {
7.   return 0;
8.  }
9.
10.  int min = v[0];
11.  int max = v[0];
12.
13.  for (int i = 1; i < v.length; i++) {
14.   if (v[i] > max) {
15.    max = v[i];
16.   } else if (v[i] < min) {
17.    min = v[i];
18.   }
19.  }
20.  return max - min;

21. }
```

- m1, line 6

Unset

```
- if (v.length == 0) {

+ if (v.length != 0) {
```

- m2, line 13

```
Unset
- for (int i = 1; i < v.length; i++) {

+ for (int i = 1; i <= v.length; i++) {
```

- m3, line 14

```
Unset
- if (v[i] > max) {

+ if (v[i] >= max) {
```

- m4, line 16

```
Unset
- } else if (v[i] < min) {

+ } else if (v[i] >= min) {
```

- m5, line 20

```
Unset
- return max - min;

+ return max + min;
```

Suggest as many test inputs as necessary to kill all mutants. Ideally, one test should only kill one mutant. If it is not possible to kill any mutant, explain why.

(Optional) Implement the derived tests in Java/JUnit.