CO3095 / CO7095 / CO7508 Assignment

Dr José Miguel Rojas

v1.1
5 November 2020

Version history

- v1.1 (06/11/2020): Fixed typos and minor formatting issues; *no meaningful changes in tasks.*
- v1.0 (05/11/2020): Assignment released.

Disclaimer on Plagiarism and Collusion

This is an **individual piece of coursework** that is assessed. Plagiarism and/or collusion are penalised. For further information check the section *Referencing and Academic Integrity* in your Student Handbook (`https://blackboard.le.ac.uk/webapps/blackboard/content/listContentEditable.jsp?content_id=_2221452_1&course_id=_28630_1&mode=reset`).

By submitting your solution, you are stating that this solution is the result of your **sole individual work** and that you are aware of the consequences of incurring in plagiarism and/or collusion, as summarised in the Declaration of Academic Integrity (`https://blackboard.le.ac.uk/webapps/blackboard/content/listContentEditable.jsp?content_id=_1860780_1&course_id=_15369_1`) you will have signed already.

General remarks

- This assignment is due on **Friday 11 December 17:00:00 UK time** and must be submitted via BlackBoard.
- This assignment contributes 30% towards your overall module grade.
- This assignment will be marked anonymously.
- Learning outcomes being assessed: Students will be able to choose appropriate strategies for software testing and validation, and discuss how to implement them.
- The solution to the assignment must be submitted as **a single ZIP file** containing **exactly** two files:
  - A Word file named **tasks1-3.docx** file containing the solutions to tasks 1-3.
  - A Java file named **SortedIntegerListTest.java**.

Tasks

Task 1: Function-Point Analysis [20%]

Your software development company has been tasked to develop a system to manage the inventory in a shoe shop. Whenever a pair of shoes is received in the shop from providers, the barcode printed on the box is scanned and a record for the pair of shoes is entered into the inventory database. When a pair of shoes is sold, the barcode is scanned again and the item is removed from the inventory database.

The data kept for each pair of shoes is their brand, model, size and colour. Shop assistants can query the inventory using bluetooth handheld devices. If an item is not in the inventory, the system will communicate with external systems to check availability in other shops. The system should be easy to install in multiple sites and accessible.

With this information, your task is to use a Function-Point Analysis to estimate the size and complexity of the system. You can make any reasonable assumption for information that is not provided in the brief, making sure you justify all your decisions. You must:

- Compute Unadjusted Function-Point Counts; you *must justify briefly* the values you assign for each function point.
- Incorporate Technical Complexity Factors; you *must justify briefly* the values assigned for each factor.
- Calculate a final Function Point Count.

Task 2: White-box Metrics [20%]

For the method `merge` in the Java class `SortedIntegerList` (Appendix A), calculate:

- The cyclomatic complexity (include Control Flow Graph and calculation process).
- The five Halstead metrics: Program length, Program vocabulary, Volume, Difficulty, and Effort.
  - Consider the following as your list of operators (the list includes the comma itself, *every token outside this list must be considered as an operand*):

```
=,>,<,!,~,?,->,==,>=,<=,!=,&&,||,++,--,
+,-,*,/,&,|,^,%,<<,>>,>>>,+=,-=,*=,/=,&=,
|=,^=,%=,<<=,>>=,>>>=,instanceof,(,),[,],;
```

Task 3: Black-box Testing [30%]

Let us suppose you have a Java method with the following signature:

```
public static boolean tweet(String message, ArrayList<Image> photos,
                            String username, String password)
```

The `tweet` method takes four arguments as input: a message, a list of photos, a username and a password. The method returns `true` if the tweet is posted successfully to Twitter, and `false` otherwise.

The `tweet` method will return `true` *iff*:

- The message contains at most 140 characters.
- The list of photos is empty or contains at most 5 photos.
- The username exists and the password is correct.

The method will throw an exception if the message is longer than 140 characters, if any of the photos is corrupted, or if the Twitter service is not responding.

Your task is to use the Category Partition Method to construct a test set for the `tweet` Java method, identifying *at least* three categories for each of the input arguments. You must:

1. Present the categories, providing a justification for each of them.
2. Identify reasonable constraints to link categories if/where appropriate.
3. Calculate the number of possible test frames and discuss the impact of any constraints.
4. Identify one test frame and provide a test case for it, that is, provide an example of concrete *inputs and output* conforming to the chosen test frame.

Note: You do not need the code of the method to apply the category partition method.

Task 4: White-box Testing [30%]

Given the Java class named `SortedIntegerList` (Appendix A; also available to download as a Java file).

1. Write a JUnit test suite named `SortedIntegerListTest` that achieves 100% branch coverage for the `SortedIntegerList` class.
2. For each of your JUnit tests, indicate *one unique mutation* in the `SortedIntegerList` class that would be killed by the test (i.e., the test fails if the mutation is applied). Indicate this in your test suite source code by adding a comment **inside each JUnit test** with this format:

```
/**
* Mutant Killed:
* Line: <line number>
* Old line: <line of code in original class>
* New line: <mutated line of code>
*/
```

*You may **not** use the `toString` method to write assertions.*

1 Marking Criteria

Distinction

- The Function Point analysis is professionally applied, including reasonable justifications and clear calculations throughout.
- The calculation of metrics is perfectly executed.
- The categories are carefully thought out and well justified. The constraints between the categories are selected in a sensible way, that does not raise the possibility of excluding valid test cases. The number of resulting test cases has been correctly worked out for the categories, with and without constraints. The underlying reasoning is presented. The test frame is clearly presented, along with an anticipated output.
- The set of JUnit tests achieves 100% branch coverage and valid mutations have been identified for all tests.

Merit

- The Function Point analysis is applied well, calculations are clear and most justifications are reasonable.
- The calculation of metrics is executed very well, with only minor mistakes.
- The categories are carefully thought out, and well justified. There are some reasonable constraints amongst categories, that are justified. The number of resulting test cases has been correctly worked out for the categories. The number of test cases incorporating constraints has been attempted and justified, but may not be correct. The test frame is clearly presented, along with an anticipated output.
- The set of JUnit tests achieves 100% branch coverage but mutations have not been identified for all tests.

Pass

- The Function Point analysis is applied to some extent, but there are flaws in calculations or unclear justifications.
- The calculation of metrics was executed competently, but with at least one major mistake.
- There are some categories there, that make some sense. There are some constraints amongst categories, with justification. The number of resulting test cases has been correctly worked out for the categories. No attempt (or an unreasonable attempt) has been made to work out the number of tests with constraints. The test frame is clearly presented, along with an anticipated output.
- The set of JUnit tests achieves high branch coverage (not 100%) and mutations have not been identified for all tests.

Fail

- There has been a serious misunderstanding of the Function Point Analysis technique.
- The calculations of metrics are missing or severely flawed.
- There has been a serious misunderstanding of the category partition method. The categories do not make sense, and most justifications are fundamentally flawed.
- The set of JUnit tests achieve very low branch coverage (less than 50%) and mutations were not identified.

A `SortedIntegerList` Java Class

```
package lists;

public class SortedIntegerList {

    private Node first = null;

    public SortedIntegerList() {
        first = null;
    }

    public boolean isEmpty() {
        return (first == null);
    }

    public int size() {
        Node curr = first;
        int n = 0;
        while (curr != null) {
            n++;
            curr = curr.next;
        }
        return n;
    }

    public Integer getFirst() {
        if (first == null)
            return null;
        else
            return first.data;
```

```
}

public Integer getLast() {
    Node curr = first;
    Node foll = first;
    while (foll != null) {
        curr = foll;
        foll = foll.next;
    }
    if (curr != null)
        return curr.data;
    else
        return null;
}

public Integer get(int index) {
    Node curr = first;
    int i = 0;
    while (curr != null && i < index) {
        curr = curr.next;
        i++;
    }
    if (curr != null && i == index)
        return curr.data;
    else
        throw new IllegalArgumentException("incorrect index " + index);
}

public int firstIndexOf(Integer data) {
    Node curr = first;
    int i = 0;
    while (curr != null) {
        if (curr.data.equals(data))
            return i;
        curr = curr.next;
        i++;
    }
    return -1;
}

public boolean contains(Integer data) {
    return firstIndexOf(data) != -1;
}

public void insert(Integer data) {
    Node curr, foll;

    if ((first == null) || (data <= first.data)) {
        first = new Node(data, first);
    } else {
        curr = first;
        foll = first.next;
        while ((foll != null) && (foll.data < data)) {
            curr = foll;
            foll = foll.next;
        }
```

```
            curr.next = new Node(data, foll);
        }
    }

    public void insertAll(Integer[] dataArr) {
        for (int i = 0; i < dataArr.length; i++)
            insert(dataArr[i]);
    }

    public void merge(SortedIntegerList l) {
        Node p1, p2, curr;

        p1 = first;
        p2 = l.first;
        if (p1.data <= p2.data)
            p1 = p1.next;
        else {
            first = p2;
            p2 = p2.next;
        }

        curr = first;
        while ((p1 != null) && (p2 != null)) {
            if (p1.data <= p2.data) {
                curr.next = p1;
                p1 = p1.next;
            } else {
                curr.next = p2;
                p2 = p2.next;
            }
            curr = curr.next;
        }

        if (p1 == null)
            curr.next = p2;
        else
            curr.next = p1;
    }

    class Node {
        Integer data;
        Node next;

        public Node(Integer data, Node next) {
            this.data = data;
            this.next = next;
        }
    }
}
```