CO3095 / CO7095 Coursework 2019-20

José Miguel Rojas

November 3, 2019

Disclaimer on Plagiarism and Collusion

This is an **individual piece of coursework** that is assessed. Plagiarism, including collusion, is penalised. For further information check the section *Referencing and Academic Integrity* in your Student Handbook (`https://campus.cs.le.ac.uk/ForStudents/`).

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 (for MSc Students October 2018 intake: `https://goo.gl/pdBicw`) and Declaration of Academic Honesty (for all other students: `https://campus.cs.le.ac.uk/ForStudents/plagiarism/DoAIF.pdf`) that you will have already signed.

General remarks

- This assignment is due on **Friday 13 December 17:00 UK time** and must be submitted via BlackBoard.
- The solution to the assignment must be submitted as **a single PDF file of at most 15 pages** containing one section for each task and using font size 12 and line spacing 1.5.
- This assignment contributes **30%** towards the overall module grade.
- This assignment will be marked anonymously. Do not include your name in any submission files and **use only your student number to identify your submission**.
- Learning outcomes being assessed: You will be able to apply software project planning and cost estimation techniques. You will be able to choose appropriate sofware testing strategies and apply them to existing source code.

Task 1. Project Planning

Your software development company has been approached by a estate agency that requires the development of a web and mobile application for advertising properties for sale and managing appointments to view them. The agency wants to call their new web and mobile apps *LetsMove*. The web app will enable the agency to advertise properties for sale, and the mobile app will allow property buyers to browse these properties and arrange viewings. The agency has provided you with this informal project brief:

1. An web frontend to allow estate agents (employees of the agency) to manage properties (add, modify, remove) and interact with clients.
2. Agents should be able to upload details and pictures of properties for sale onto a database; they should also be able to schedule and manage viewings for these properties.
3. A mobile app should provide a registration facility for general users looking for properties to buy (buyers). As part of this registration, the buyer needs to provide their preferences and also documentation about their income and employment. This registration should also trigger a credit score check with external credit score companies such as Experian.

4. The database should securely and reliably store all the details about properties and buyers and also keep track of scheduled viewings and properties' lifecycles (available, sold subject to contract, sold, etc.).

5. Buyers using the mobile app should be able to edit their preferences for properties (e.g., locations, number of rooms, prices, etc.).

6. When a estate agent adds a new property on the web application, all the buyers who have declared related interests should be notified about it on the mobile device where they have installed the app.

7. Buyers should be able to search properties using a flexible search mechanism. They should be able to look at the details of any property and they should be able to mark any property as saved, favourite, or ignored. When interested in a property, the app should allow the buyer to book an appointment by presenting the property availability for viewing. A booking fee using major credit and debit cards is applied to confirm an viewing appointment.

8. Estate agents using the web application should be able to print a daily schedule for each property.

9. Estate agents using the web application should also be able to view and edit the status of a property (e.g., when an offer has been received or it has been sold).

10. The system should perform periodic backups of the database, which is expected to grow very quickly given the large amount of business the company deals with. The system should be capable of dealing with a high load of users.

11. The system should be very reliable. Only a very small amount of down-time is tolerable and failures should be rare.

Questions

1. Carefully study the project brief provided by the client and identify all the activities that you will need to carry out to develop the system. Notice that not all the points in the brief correspond exactly with one activity. Also, consider that there might be additional activities that will need to be completed (e.g., business analysis, testing, etc.).

2. For each requirement provide a crude estimation of how many lines of code its implementation might require. Provide this number in thousands of lines of code (KLOC). For instance, 450 lines of code = 0.45 KLOC.

3. Use historical data to estimate how much effort will be required to implement each requirement (using your estimates for KLOC). The NASA93 dataset can be downloaded from `https://terapromise.csc.ncsu.edu/repo/effort/cocomo/cocomo2/nasa93-dem/nasa93-dem.arff`). Within this data you should specifically focus on the relationship between the effort and the KLOC column. Compute a Linear Regression model on this data to calculate the relationship between KLOC and effort. This should give you a *cost per KLOC* (variable a), and *an initial cost* (variable c) (if any).

4. As a mature software development organisation, you will rigorously adopt software engineering best practices (e.g., design patterns) and therefore you expect that the development costs will lower as the size of the system increases. Using the historical data, select different values for the scale factor (variable b) and gauge visually (by plotting the resulting curve) which value of b (if any) produces a better approximation than the linear model alone. Use this model to calculate the "normal" predicted effort for each of the activities.

5. Produce a table, add the normal estimates you have calculated, and then also estimate your own pessimistic and optimal durations. Use these to calculate the expected duration for each activity. Make sure that every row contains an ID for the activity in question, e.g., A, B, C, ...

6. Identify the dependencies between your activities and **draw out a dependency network**. For this part of the exercise, there are a multitude of different possible dependency networks, depending on the assumptions you make about the underlying implementation and the amount of details you

build into your planning. Justify these assumptions and briefly justify the dependencies for each activity.

7. Create a PERT chart capturing the dependencies between the identified activities.
8. Identify the critical path in the PERT diagram and briefly discuss what it represents and the different alternatives arising from the slack times that you have calculated.

Task 2. Black-box Testing

Given the following code that implements the Luhn algorithm for checking credit card validity:

```java
public class Luhn {
    public static final int MASTERCARD = 0, VISA = 1;
    public static final int AMEX = 2, DISCOVER = 3, DINERS = 4;

    // Check that cards start with proper digits for
    // selected card type and are also the right length.
    public int validate(String number, int type) {
        switch (type) {

            case MASTERCARD:
                if (number.length() != 16 ||
                        Integer.parseInt(number.substring(0, 2)) < 51 ||
                        Integer.parseInt(number.substring(0, 2)) > 55) {
                    return 0;
                }
                break;

            case VISA:
                if ((number.length() != 13 && number.length() != 16) ||
                        Integer.parseInt(number.substring(0, 1)) != 4) {
                    return 0;
                }
                break;

            case AMEX:
                if (number.length() != 15 ||
                        (Integer.parseInt(number.substring(0, 2)) != 34 &&
                                Integer.parseInt(number.substring(0, 2)) != 37)) {
                    return 0;
                }
                break;

            case DISCOVER:
                if (number.length() != 16 ||
                        Integer.parseInt(number.substring(0, 5)) != 6011) {
                    return 0;
                }
                break;

            case DINERS:
                if (number.length() != 14 ||
                        ((Integer.parseInt(number.substring(0, 2)) != 36 &&
                                Integer.parseInt(number.substring(0, 2)) != 38) &&
                                Integer.parseInt(number.substring(0, 3)) < 300 ||
                                Integer.parseInt(number.substring(0, 3)) > 305)) {
                    return 0;
                }
                break;
        }
        return luhnValidate(number) ? 1 : 0;
    }

    // The Luhn algorithm is basically a CRC type
    // system for checking the validity of an entry.
    // All major credit cards use numbers that will
    // pass the Luhn check. Also, all of them are based
57      // on MOD 10.
58      private boolean luhnValidate(String numberString) {
59          char[] charArray = numberString.toCharArray();
60          int[] number = new int[charArray.length];
61          int total = 0;
62
63          for (int i = 0; i < charArray.length; i++) {
64              number[i] = Character.getNumericValue(charArray[i]);
65          }
66
67          for (int i = number.length - 2; i > -1; i -= 2) {
68              number[i] *= 2;
69
70              if (number[i] > 9)
71                  number[i] -= 9;
72          }
73
74          for (int i = 0; i < number.length; i++)
75              total += number[i];
76
77          if (total % 10 != 0)
78              return false;
79
80          return true;
81      }
82  }
```

1. Identify a test set to achieve MC/DC coverage for the `validate` method. You are *not* required to consider coverage of the `luhnValidate` method.
   - For each set of inputs, you should list the branches that you would expect to cover. This should include the line number for the decision in question, the specific condition, and whether the outcome of the decision would be true or false.
2. Use the Category Partition Method to construct test sets for the `validate` method.
   - Present the categories, providing a justification for each category.
   - Identify reasonable constraints to link categories where appropriate.
   - Identify the number of possible test frames, and discuss the impact of any constraints.
   - Finally, provide a test case, that is, an example set of inputs and corresponding output conforming to a particular test frame.

Task 3: White-box Testing

Given the following Java implementation of a class named `StringStack`:

```
1   public class StringStack {
2           private int capacity = 10;
3           private int pointer = 0;
4           private String[] objects = new String[capacity];
5
6           public void push(String o) {
7                   if (pointer >= capacity)
8                           throw new IllegalArgumentException("Stack exceeded capacity!");
9                   objects[pointer++] = o;
10          }
11
12          public String pop() {
13                  if (pointer <= 0)
14                          throw new IllegalArgumentException("Stack empty");
15                  return objects[--pointer];
```

```
	}

	public boolean isEmpty() {
		return pointer <= 0;
	}
}
```

Questions.

1. Write a JUnit test suite that achieves 100% branch coverage of the three methods in the class. The test suite must be written in Java and must compile and execute without modifications.
2. For each test, indicate what branches in the code each test is expected to cover.
3. Provide a brief description of mutation testing and its benefits compared to coverage-driven testing, using the produced test suite as example.
4. For each test in the resulting test suite, identify at least one mutant that would be killed by the test (deletion of lines of code is not allowed).

Assessment

Weights

The tasks are weighted as follows:

- **30%:** Project Planning task.
- **30%:** Black-box Testing task.
- **40%:** White-box Testing task.

Marking Criteria

Distinction

- Requirements and activities have been reasonably identified. A PERT chart and its critical path have been constructed and identified correctly.
- The application of the Category Partition method is carefully thought out and well explained (categories are clearly justified, constraints are defined sensibly and do not raise the posibility of excluding valid test cases) and the number of resulting test frames has been correctly worked out. A test case is clearly presented along with an anticipated output.
- The MC/DC criterion is accurately, clearly, and concisely applied and justified. A reasonably small set of inputs that achieves MC/DC coverage is provided.
- The set of unit tests achieves full branch coverage with a reasonably small number of tests which are well described. The test suite compiles and executes without errors. One mutant is presented and discussed for each test in a convincing manner.

Merit

- The planning task has been carefully developed and the artefacts produced are mostly correct (cost estimations, network diagram, PERT chart, critical path). Requirements and activities have been identified but not adequately justified. A PERT chart has been constructed and a critical path has been identified but some problems are evident.

- The Category Partition method has been applied to some degree, but some categories, partitions or constraints are not well justified. The number of test frames incorporating constraints has been attempted and justified, but may not be correct. A test case is presented, but its correspondance to a test frame or the input-output relationship is not clearly justified.
- The application of MC/DC is overall competent but with some weaknessess. A set of inputs at least comes close to achieving MC/DC coverage.
- The set of unit tests developed compiles and executes correctly without modifications and comes close to achieving full branch coverage and the tests are well described. Some mutants are presented and discussed but not correctly justified for all tests.

Pass

- The planning task has been attempted and some of the artefacts produced are correct (cost estimations, network diagram, PERT chart, critical path). Some requirements and activities have been identified, with some evident weaknesses.
- Some categories, partitions and constraints have been identified which make some sense but are not well justified. The number of resulting test frames has been worked out but not in a sufficiently convincing or reasonable manner. An incomplete and insufficiently justified test case is presented.
- The application of MC/DC is understandable but may be erroneous.
- The set of unit tests developed compiles and executes correctly with some modifications and achieves at least 50% branch coverage; some of these tests are well justified. The application of mutation testing to derive mutants is somewhat erroneous Nd only a small portion of mutants are presented and discussed.

Fail

- The planning task has been attempted but all the artefacts produced are flawed (cost estimations, network diagram, PERT chart, critical path).
- There has been a serious misunderstanding of the category partition method. The categories do not make sense, and most justifications are fundamentally flawed.
- The application of MC/DC is severely flawed and the set of inputs is not well justified.
- The set of unit tests developed does not compile or execute or if it does, it achieves less than 50% branch coverage and tests are not well described. Mutants are missing or do not make much sense.