SE465
Software Testing, Quality Assurance, and Maintenance
Assignment 3, version 0.9

Patrick Lam

Release Date: February 8, 2019

Due: 11:59 PM, Friday, March 29, 2019
Submit: via git.uwaterloo.ca

Getting set up

We will create a copy of the starter repo for you in your `git.uwaterloo.ca` account. You need to log in to `git.uwaterloo.ca` for that to work. There is a Vagrant VM description, but you shouldn't need it.

Tools used (all required): PMD; C++; Java plus Maven; Valgrind (not strictly required but highly recommended).

Submission summary

Here's what you need to submit in your fork of the repo. Be sure to commit and **push** your changes back to `git.uwaterloo.ca`.

1. in directory `q1`, your modified `a1q1-automarker.xml` file and `A1Q1Test.java`;
2. in directory `q2`, file `icalendarlib-memory.diff` as described in the question;
3. in directory `q3`, either file `bugreports.txt` or `bugreports.pdf`;
4. in directory `shared/itext`, files `pom.xml` and `src/test/java/com/itextpdf/text/pdf/TaggedPdfText.java`.
5. in directory `q5`, either file `codereview.txt` or `codereview.pdf`;

Once again, you may choose to move the `q?` directories into the `shared` subdirectory if you want to access them in Vagrant. We'll mark submissions either in the original place or under `shared`.

Question	TA in Charge
1	?
2	?
3	?
4	?
5	?

Question 1 (10 points)

A few months ago, I wrote an automarker for your Assignment 1 Question 1 submissions. You will know enough to write this automarker yourself. We'll focus on just the technically challenging part.

I've included PMD in the a3 skeleton and provided a template PMD `a1q1-automarker.xml` file. Use the following command in your VM to run your automarker:

```
~/shared/pmd/bin/run.sh pmd -f text -d ~/shared/q1/A1Q1Test.java -R ~/shared/q1/a1q1-automarker.xml
```

Your task is to write a PMD rule that detects JUnit 4 test methods which have no calls to assert methods with arguments named `mockCommandSender.getLastMessage`. JUnit 4 test methods have a `Test` annotation. Calls to assert methods are Statements with a PrimaryPrefix descendant whose name starts with "assert". Submit your `a1q1-automarker.xml` file in directory `q1/`.

In file `q1/A1Q1Test.java`, write test methods that show that your query works properly; these tests should show that your query flags methods that it should and doesn't flag methods that it shouldn't.

Question 2 (10 points)

Recall `icalendarlib` from Assignment 2. I've made some changes to it to make it more Valgrind-friendly and again placed it in `shared/icalendarlib`. The code contains 4 memory errors. Using `valgrind`, or otherwise, find the errors and submit the diff for your changes in file `q2/icalendarlib-memory.diff`. I believe that two of the errors are not reachable from the current `main.cpp` file. You'll need to add more code to `main.cpp` if you want to trigger them.

Question 3 (10 points)

In this question, you will critique and improve an existing bug report in Mozilla and write a bug report from scratch.

(a) Read Mozilla bug report 112785 (`https://bugzilla.mozilla.org/show_bug.cgi?id=112785`). What are some problems with the initial bug report (as seen in the "Title" and the "Description")? Identify four problems. How would you improve this bug report? (5 points)

(b) The class `q3/HashTable.java` is an implementation of a hash table using linear open addressing and division in Java. This program has a bug, because the `put` function will not update the element associated with the given key if an entry with the same key already exists. The hash table implementation should always update the element, even if the element's key already exists in the hash table. (See the comment in the `put` function).

Write a good bug report for this bug using the Bugzilla bug report format. (5 points)

(Recommended exercise, not for marks: write a JUnit test that illustrates this bug.)

Question 4 (10 points)

In this question, add unit tests to the `com.itextpdf.text.pdf.TaggedPdfTest` class for the `add(final Element o)` method of the `com.itextpdf.text.List` class from iText (in `shared/itext`). Your solution must use mock objects (with a mock object library of your choice; modify your `pom.xml` accordingly). Your tests must kill the following mutants:

```
public boolean add(final Element o) {
    if (o instanceof ListItem) {
        ListItem item = (ListItem) o;
        if (numbered || lettered) { // mutant 1: || -> &&
            Chunk chunk = new Chunk(preSymbol, symbol.getFont());
            chunk.setAttributes(symbol.getAttributes());
            int index = first + list.size(); // mutant 2: index = first (NOTE EDIT), mutant 3: list -> item
            if ( lettered )
                chunk.append(RomanAlphabetFactory.getString(index, lowercase));
            else
                chunk.append(String.valueOf(index));
            chunk.append(postSymbol);
            item.setListSymbol(chunk);
        }
        else {
            item.setListSymbol(symbol);
        }
        item.setIndentationLeft(symbolIndent, autoindent);
        item.setIndentationRight(0);
        return list.add(item); // mutant 4: list -> item
    }
    else if (o instanceof List) {
        List nested = (List) o;
        nested.setIndentationLeft(nested.getIndentationLeft() + symbolIndent);
        first--;
        return list.add(nested); // mutant 5: nested -> null
    }
    return false;
}
```

I'm aware that the unit tests that come with iText also kill the mutants. But they don't use mock objects. As I wrote above, your solutions are required to use mock objects.

Here is an additional hint:

1. Be aware of the EasyMock factory method `notNull()` and class `Capture<T>`.

Question 5 (10 points)

In this question, you will perform code review. You may: 1) review your own code from the past; 2) review some code that a friend provides for you; or 3) review code that I suggest, namely the `com.itextpdf.text.pdf.SimpleBookmark` class from iText (which we saw in Question 4). The advantage of (1) and (2) is that you have the opportunity to get your questions about the code answered. The code may be in any language but should be between 500 and 1000 lines of code. You may also review a pull request of a similar magnitude.

Your task is to apply the code review checklist at `https://blog.fogcreek.com/increase-defect-detection-with-our-code-review-checklist-example/`. In particular, pick 3 questions from that list and answer them for the code that you are reviewing. Explain your answer in a couple of sentences, supporting it with examples from the code that you're reviewing. Put your solution in either file `q5/codereview.md` (Markdown syntax preferred) or `q5/codereview.pdf`.