Software Testing 2019-20: Practical

(Updated on 3rd March)

Background

This page describes the practical for the Informatics Software Testing course. It will be marked out of **100 points**, and is worth 25% of the assessment of the course. This practical will be undertaken individually or in groups of two, and will be assessed on the basis of the group submission.

Important dates. This practical is comprised of 4 tasks with the following issue dates and deadlines:

- Issued: 25th February.
- **Deadline: 27th March, 4pm GMT**.

The policy for late submission follows the UG3 course guide (`http://www.inf.ed.ac.uk/teaching/years/ug3/CourseGuide/coursework.html`).

Plagiarism and academic misconduct. The practical will be scrutinized for plagiarism and academic misconduct. Information on academic misconduct and advice for good scholarly conduct is available at `http://web.inf.ed.ac.uk/infweb/admin/policies/academic-misconduct`.

Tools. You will need the following tools to solve this practical:

Java Development Kit This practical is based on the Java programming language. We recommend using JDK 8, which is available by default on the DICE machines.

Integrated development environment We recommend using the Eclipse IDE for this practical. Eclipse is available on the DICE machines, otherwise it can be downloaded from here: `https://www.eclipse.org/getting_started/`.

Unit testing This practical uses the JUnit 4 framework for unit testing (we use JUnit 4 for compatibility with the other required tools). JUnit 4 is already available on the DICE machines. This article is a reasonable introduction to using JUnit with Eclipse: `https://www.vogella.com/tutorials/JUnit/article.html#eclipse-support-for-junit-4`. A more thorough introduction to JUnit 4 can be found here: `https://www.ibm.com/developerworks/java/tutorials/j-junit4/`.

Coverage analysis Tasks 2 and 3 require a coverage analysis tool with support for measuring branch coverage: we recommend using EclEmma within Eclipse, which is already available on the DICE machines. Alternatives to EclEmma are discussed at `http://java-source.net/open-source/code-coverage`.

Automatic test generation Task 3 uses the Randoop tool to generate JUnit 4 test cases automatically. Instructions to download and run Randoop can be found here: `https://randoop.github.io/randoop/manual/index.html` and in the description of Task 3 below. We recommend reading up to and including section *Generating tests* before solving the task.

System under test. This practical uses `st1920.automaton`, a regular expression Java package, as the system under test. `st1920.automaton` is a simplified version of `dk.brics.automaton` , an open-source package created by Anders Møller at Aarhus University, Denmark (see `https://www.brics.dk/automaton/`). The package `st1920.automaton` is available in the Learn course website in two formats:

- as a Java Archive (JAR) file containing a build with injected bugs for Task 1 (`automaton.jar`); and
- as a ZIP file containing the source code without injected bugs for Tasks 2, 3, and 4 (`automaton.zip`).

Tasks

Task 1: Finding Bugs with Unit Testing [30 points]

The goal of this task is to apply unit testing to find bugs in `st1920.automaton` in a black-box fashion, that is, following its functional specification rather than its source code structure.

Functional specification of the system under test. The system under test provides a utility class `RegExpMatcher` with a single static method that serves as an interface to the entire package:

```
public static boolean matches(String m, String re)
```

This method returns `true` if the string `m` matches the pattern specified by the regular expression `re`, and `false` otherwise. In `st1920.automaton`, regular expressions are specified by defining and composing patterns using the operations listed in Table 1.

pattern1 `|` *pattern2*	matches either *pattern1* or *pattern2*
pattern1 `&` *pattern2*	matches both *pattern1* and *pattern2*
pattern1 pattern2	matches *pattern1* followed by *pattern2*
pattern `?`	matches zero or one occurrence of *pattern*
pattern `*`	matches zero or more occurrences of *pattern*
pattern `+`	matches one or more occurrences of *pattern*
pattern `{`n`}`	matches n occurrences of *pattern*
pattern `{`n`,}`	matches n or more occurrences of *pattern*
pattern `{`n`,`m`}`	matches between n and m occurrences of *pattern*
`~` *pattern*	matches the patterns excluded by *pattern*
`[` *charclass1 charclass2 ...* `]`	matches a character belonging to at least one of *char-class1*, *char-class2 ...*
`[^` *charclass1 charclass2 ...* `]`	matches a character not belonging to any of *char-class1*, *char-class2 ...* (a character class *charclass* is either a single character c or a character range *c1*`-`*c2*)
`.`	matches any single character
`#`	does not match anything
`@`	matches any string
`"` *string* `"`	matches the entire string *string*
`<` n `-` m `>`	matches a number between n and m

Table 1: Regular expression operations in `st1920.automaton`.

For example, the following regular expression defines syntactically valid email addresses:

```
([a-zA-Z0-9])+\@([a-zA-Z0-9])+\.([a-z]){2,3}
```

Note that, for the sake of illustration, this regular expression is made overly simple. For example, it matches the string `foo@bar.com` but not `foo@ed.ac.uk` .

Patterns within regular expressions can be enclosed in parenthesis to enforce the desired operator precedence. Reserved characters (`|&?*+,~^.#@"<>()\`) can be used as regular characters by escaping them with backslash (`\`) or double quotes (`""`). Ill-formed regular expressions lead to exceptions of type `IllegalArgumentException`.

The syntax and rules to combine the regular expression operators described above are formalized as a BNF grammar here: `https://www.brics.dk/automaton/doc/index.html?dk/brics/automaton/RegExp.html`. If you want to learn more about regular expressions and the rich body of theory underlying them, see for example the classic text of Hopcroft *et al.* "Introduction to Automata Theory, Languages, and Computation".

Target bugs. We have injected 15 different bugs in the `st1920.automaton` package. The estimated difficulty of finding these bugs ranges from easy (5 bugs), to medium (5 bugs), to hard (5 bugs). All bugs can be triggered using appropriate calls to the `matches` method in the utility class `RegExpMatcher`. For clarity, the bugs are numbered and manifest themselves as exceptions triggered with the following error message:

```
java.lang.IllegalStateException: Bug <N> found (<DIFFICULTY>).
```

where `<N>` indicates the bug number (from 1 to 15) and `<DIFFICULTY>` indicates the estimated difficulty of finding the bug.

Tip: if you run out of ideas to create new test cases, you can try to apply automatic unit test generation using the Randoop tool as described in Task 3.

Input.

- A Java Archive (JAR) file containing a build of the `st1920.automaton` package with injected bugs (`automaton.jar`).

Deliverables.

- A file `Task1.java` with JUnit 4 test cases that trigger the injected bugs. Each test case should trigger one of the `IllegalStateException` exceptions described above.

Assessment. This task will be assessed by counting the number of different bugs that are triggered by the test cases in `Task1.java`, weighted by their level of difficulty: easy bugs are worth 1 point, medium bugs are worth 2 points, and hard bugs are worth 3 points.

Submission. To submit your work you should designate one member of the group as a submitter for the group. The submitter will gather the deliverable for the task and execute this command on a DICE machine:

```
submit st cw1 Task1.java
```

Task 2: Analyzing Code Coverage [5 points]

The goal of this task is to measure and analyze the **branch coverage** of the `st1920.automaton` code achieved by executing the test cases developed in Task 1. The measurements should exclude coverage of the test cases themselves. The easiest way to achieve this with EclEmma is by placing the test classes under a separate directory, for example `test/st1920/automaton`.

Input.

- A ZIP file containing the source code of the `st1920.automaton` package without injected bugs (`automaton.zip`).

Deliverables.

- A file `task2.jpg` containing a screenshot of the branch coverage report as shown by the coverage measurement tool. Please make sure that the total branch coverage (excluding added test code) is clearly visible in the screenshot.

Assessment. This task will be assessed by scrutinizing the results reported in `task2.jpg`.

Submission. To submit your work you should designate one member of the group as a submitter for the group. The submitter will gather the deliverable for the task and execute this command on a DICE machine:

```
submit st cw1 task2.jpg
```

Task 3: Improving Coverage Manually and Automatically [30 points]

The goal of this task is to improve the branch coverage of the tests developed in Task 1 by introducing additional test cases targeting the `matches` method. The additional tests (on top of the tests developed in Task 1) should be designed using the Randoop automatic unit test generation tool first (`https://randoop.github.io/randoop/`), and then completed with manually-generated tests that exercise branches uncovered by Randoop. Remember that the measurements should exclude coverage of the test cases themselves.

Task 3.1: Improving Coverage Automatically With Randoop

Randoop takes a Java class and generates test cases with sequences of method and constructor calls. To generate up to `N` JUnit test cases for the Java class `RegExpMatcher` in an Eclipse project located under directory `$PROJECT_DIR`, download the file `https://github.com/randoop/randoop/releases/download/v4.2.1/randoop-all-4.2.1.jar` into a directory `$RANDOOP_DIR`, and run the following command on a terminal:

```
java -classpath \
$RANDOOP_DIR/randoop-all-4.2.1.jar:$PROJECT_DIR/build \
randoop.main.Main gentests --testclass=st1920.automaton.RegExpMatcher \
--output-limit=N
```

If successful, Randoop will output one or more files named `RegressionTest*.java` with the generated test cases.

Tip: keep in mind that Randoop does not directly create random primitive values, only sequences of method and constructor calls. To create random strings that can be used as arguments to the `matches` method, you will have to add additional methods to `RegExpMatcher` that build and combine strings. For example, to generate random regular expression strings, you can add methods such as:

```
public static String makeAlpha() {
    return "a";
}

public static String makeNum() {
    return "1";
}

public static String makeConcatenation(String l, String r) {
    return l + r;
}
```

Randoop will then generate tests that create random strings by combining calls to the above methods, for example (simplified and structured for clarity):

```
@Test
public void testRandoop() {
    String s1  = makeAlpha();                   // "a"
    String s2  = makeNum();                     // "1"
    String s3  = makeConcatenation(s1, s2);     // "a1"
    String s4  = makeAlpha();                   // "a"
    String s5  = makeAlpha();                   // "a"
    String s6  = makeConcatenation(s3, s4);     // "a1a"
    String s7  = makeConcatenation(s6, s5);     // "a1aa"

    String s8  = makeAlpha()                    // "a"
    String s9  = makeNum()                      // "1"
    String s10 = makeConcatenation(s8, s9);     // "a1"

    assertFalse(RegExpMatcher.matches(s7, s10));
}
```

Note that, whenever possible, Randoop might evaluate sequences of method calls to generate more concise test cases, such as:

```
@Test
public void testRandoop() {
    assertFalse(RegExpMatcher.matches("a1aa", "aa"));
}
```

Tip: if you want to generate strings differently for each of the arguments of `matches`, you can create a wrapper method

```
public static boolean matches(MatchString m, REString re)
```

Then, you can create wrapper classes `MatchString` and `REString`, and define different building and combination operations for each of these classes.

Task 3.2: Improving Coverage Manually

After generating test cases with Randoop, the branch coverage should have improved noticeably. Still, some branches in the `st1920.automaton` package might remain unexplored. By manually identifying execution paths that cover the unexplored branches and corresponding test cases, it is often possible to improve branch coverage.

Keep in mind that 100% coverage is not achievable, as some code in the package is simplify unreachable from the `matches` method.

Input.

- A ZIP file containing the source code of the `st1920.automaton` package without injected bugs (`automaton.zip`).

Deliverables.

- A file `Task3_1.java` with the test cases generated by Randoop.
- A file `task3_1.patch` containing the changes to the `RegExpMatcher` class, and any wrapper class created to support Randoop's random generation of strings. Patch files can be created with different version control systems, for example using `git diff`.
- A file `Task3_2.java` with the test cases generated manually to improve the coverage of Randoop's test cases.
- A file `Task3_all.java` with all test cases in `Task1.java`, `Task3_1.java`, and `Task3_2.java`.
- A file `task3.jpg` containing a screenshot of the branch coverage report after executing the test cases in `Task3_all.java`. Please make sure that the total branch coverage (excluding added test code) is clearly visible in the screenshot.
- A file `task3.txt` with a brief reflection, organized as a list of bullet points, of advantages and disadvantages of using Randoop in the context of this task.

Assessment. This task will be assessed based on the branch coverage achieved by the submitted test cases (excluding coverage of the test code itself); and the consistency and clarity of the reflection in `task3.txt`. Keep in mind that 100% coverage is not achievable due to code unreachability.

Submission. To submit your work you should designate one member of the group as a submitter for the group. The submitter will gather all the deliverables for the task and execute this command on a DICE machine:

```
submit st cw1 Task3_1.java task3_1.patch Task3_2.java Task3_all.java task3.jpg task3.txt
```

Task 4: Adding Functionality with Test-Driven Development [35 points]

The goal of this task is to extend the functionality of `st1920.automaton` using a test-driven development approach. You will extend the regular expression language provided in the package with a new end-of-line operator `$` matching any of the following character sequences:

- `\n` (Unix)
- `\r\n` (Windows)
- `\r` (Classic Mac OS)

The new operator should not match any other character sequence than these three ones. In terms of the original specification, this task extends the operators listed in Table 1 as shown in Table 2.

pattern1 `|` *pattern2*	matches either *pattern1* or *pattern2*
pattern1 `&` *pattern2*	matches both *pattern1* and *pattern2*
...	...
`<` *n* `-` *m* `>`	matches a number between *n* and *m*
`$`	matches an end-of-line character sequence

Table 2: Regular expression operations extended with end-of-line operator.

In the formal BNF grammar (`https://www.brics.dk/automaton/doc/index.html?dk/brics/automaton/RegExp.html`), the end-of-line operator `$` is a new type of simple expression (*simpleexp*):

```
simpleexp ::= charexp
            |   .        (any single character)
            |   #        (the empty language)  [OPTIONAL]
           ...
            |   <n-m>    (numerical interval)  [OPTIONAL]
            |   $        (end-of-line character sequence)
```

As an example, after adding support for the end-of-line operator, the string `a\naaa\r\n` should match the regular expression `(a+$)+` .

This additional functionality can be developed entirely within the `RegExp` class (and possibly also the `Automaton` class, depending on the implementation strategy).

Input.

- A ZIP file containing the source code of the `st1920.automaton` package without injected bugs (`automaton.zip`).

Deliverables.

- A file `Task4.java` with the set of test cases produced to drive the development process.
- A file `task4.patch` containing the changes to the original source code made to implement the functionality, excluding the test cases. Patch files can be created with different version control systems, for example using `git diff`.
- A file `task4.log` containing the change log of the new functionality and test cases. The change log should at least include, for each change (or *commit* in `git` speak), a single-sentence description and a list of files added, modified, or deleted. In `git`, change logs with this information can be produced by running `git log --name-status` .

Assessment. This task will be assessed by testing `task4.patch` with an independent set of test cases, applying `Task4.java` to an independent implementation, and scrutinizing `task4.log` to ensure that test-driven development has been applied.

Submission. To submit your work you should designate one member of the group as a submitter for the group. The submitter will gather all the deliverables for the task and execute this command on a DICE machine:

```
submit st cw1 Task4.java task4.patch task4.log
```