CITS5501 Software Testing and Quality Assurance
Semester 1, 2021
Project

Details

Version: 0.1
Date: 2021-05-14
Changelog:
- 2021-05-14 – initial version

Please check the CITS5501 website to ensure that you have the latest version.

The goal of this project is to assess your understanding of concepts covered in lectures and practised in lab/workshops.

- The assignment contributes 35% towards your final mark this semester, and is to be completed as individual work. It is marked out of 50.
- The deadline for this assignment is **23:59 pm, Sunday 23th May**. However, there is a grace period of 5 days – any projects submitted will be accepted without penalty until 23:59pm, Friday 28th May.
- The assignment is to be done individually.
- The submission procedure is given below under the heading "Submission".
- You are expected to have read and understood the University Guidelines on Academic Conduct. In accordance with this policy, you may discuss with other students the general principles required to understand this assignment, but the work you submit must be the result of your own effort.

Background

You are working for Tempestuous Software, a startup which is developing their product *Agendum*, a "to-do" list and appointments app.

Amongst other features, it:

- monitors current weather and traffic conditions in the user's city;
- uses AI to analyse items on the user's to-do list; and
- tries to determine whether weather or traffic conditions could interfere with any of the user's tasks to do or appointments, and alerts the user if so.

Tasks

1. Input Space Partitioning

One of the Java methods in the system is `assessWeatherImpact`. It takes as input the ID number of an item on the user's "to-do" list, a String representing the user's location (a city name), and a URL to Tempestuous's weather-reporting server.

Here is the documentation and signature for the method:

```
/** Return a measure of the likely impact of the current
* weather on the specified to-do item.
*
* The return value will be a number between 0.0 (no impact)
* to 1.0 (item is likely impossible to currently carry out),
* inclusive.
*
* @param taskID An ID number for an item on the user's to-do list
* @param city The city the user is located in
* @param serverURL The URL of a web-server providing weather updates
* @return A number representing the assessed impact of the
*     weather on the to-do list item.
*/
public double assessWeatherImpact(String taskID, String city, URL
    ↪ serverURL)
```

You will need to apply the *Input Space Partitioning* technique in order to develop unit tests for the `assessWeatherImpact` method. Note that you *need not write code for the unit tests*; but you should be able to state what they should do.

a. Explain whether any other parameters besides `taskID`, `city`, and `serverURL` need be considered. (2 marks)
b. Work through the steps of the Input Space Partitioning process so as to derive at least two test cases for the `assessWeatherImpact` method. You need not exhaustively list every characteristic and partition you would use in the process, but should mention at least two characteristics. (10 marks)
c. Based on your answer to (b), describe two test cases for the `assessWeatherImpact` method. If any test doubles might be needed, state what sort. (5 marks)

If you need to make any assumptions, or would need additional information to provide an answer, state the assumptions and what information you would need.

Suggested answer length: 1–2 pages.

Question (a)

Other parameters almost certainly do need to be considered. This is not a `static` method, so there may be instance variables that affect the execution of the method; there may also be "global" variables attached to other classes that could affect the execution.

We would need more information to know exactly what those parameters are, but at a minimum, there must be a system representation of a *user* and of their *todo* list. It could be that this is stored in a database; in that case, the parameters might consist of a connection to that database (plus, the state of the database).

We can also reasonably guess that rules about what tasks are feasible in what sort of weather might not be hard-coded, but rather represented by (say) a database or machine-learning system. Those affect the method outcome, so could also be a parameter.

Question (b) - ISP steps

The steps involved are:

1. Identify functions

- In this case, we assume we are just testing the `assessWeatherImpact` function.

2. Identify all parameters

- In this case, we don't have information about all the parameters, but can still apply ISP to the parameters we do know about: `taskID`, `city`, `serverURL`
- We might also assume that we have access to a variable `List<Task> taskList`, and that every item on the list has a task ID.

3. Identify characteristics of parameters

For task ID:

- A major relevant characteristic is, Is the task ID one of the tasks on the task list? If it is not, presumably some sort of exception should be thrown.
- Tasks IDs probably have some standard format – e.g. "`T012345678`", perhaps – so we could ask, "Is the task ID validly formed?"

Some task ID characteristics which *aren't* relevant to the present scenario:

- *Is it a `String`?* This is not applicable, as Java is a compiled, statically-typed language with strong type guarantees - it's impossible (assuming the compiler and runtime have been properly implemented) to pass a non-String for this parameter.
- *Is it zero-length?* This is unlikely to be relevant, except insofar as zero-length strings probably aren't valid task IDs.
- *Does it contain non-ASCII characters?* Again, unlikely to be relevant. Presumably task IDs are system-chosen, not user chosen, so can be restricted to whatever subset of characters we like.

For `city`:

- We could ask, what happens if the city is not one that we service? Presumably an exception should be thrown.

(Note also that here, in a properly internationalized system, we potentially *might* want to check that our system can handle city names containing non-ASCII characters; presumably, natives of a city want to be able to refer to it in their own language. But then again, by the time the method we're considering has been called, city names might all have been converted to a canonical, internal form. It would depend on the system.)

part b continued

For `serverURL`:

- Checking the Java documentation reveals that the `URL` type already checks validity of URLs – so there's not much point using "Is the URL syntactically correct?" as a characteristic.
- But we *could* ask, What happens if the URL specifies a server that isn't reachable? (As an aside: if we're using a standard network library, then at the unit test level, we might simply assume it works as described – there's no point writing tests for testing someone else's library. But at the integration or system test level, we do at some stage want to check that when servers aren't reachable, the system degrades gracefully.)

Many other characteristics are possible, but a summary of the ones above is:

1. taskID
   a. Does it refer to one of the tasks on the task list, yes or no?
      (Gives rise to 2 partitions)
   b. Is it in a valid format, yes or no?
      (Gives rise to 2 partitions)
2. city
   a. Is this a city we service, yes or no?
      (Gives rise to 2 partitions)
3. url
   a. Is the URL reachable?

4. Choose partitions

So far, we just have four characteristics, with 2 partitions each.

So we *could* choose all combinations (except invalid ones), giving rise to about 16 tests. (One invalid combination, though, is 1(a) = Yes, and 1(b) = No; so probably 15 tests).

Realistically, however, we know this is a small part of a much larger system, and it makes sense to choose our tests more parsimoniously - *base choice* selection is usually a sensible approach.

Good base choices might be:

- 1(a) – refers to a task on the list
- 1(b) – is in a valid format
- 1(c) – is a city we service
- 1(d) – is reachable

Then we vary just *one* of these at a time, getting 8 test cases.

Question (c) - test cases

For each test case, we need to identify:

- Input values
- Expected result
- Potentially, any setup that needs to be done ("prefix values") or special commends needed to observe the test result and return the system to a stable state ("postfix values").

It's also helpful to give your test case a name and/or description. Based on part (b), two possible test cases follow. Some assumptions:

- We will assume some class exists – call it "`Predictor`" – which can take in an English language description of an activity, plus a weather report, and output a `double`, representing likely impact of the weather on the activity. We will assume we are writing *unit* tests so we will want to mock that an instance of that class.
- We will assume a `TaskList` class exists – we will likely need to mock an instance of that as well.
- We will assume that "Arizona, TX" is a city we service, and that task IDs are in the format "T012345678" (letter T plus 9 digits), and that the server URL is like the one shown.
- If this is a unit test – we'll want a way of mocking network connections. (Mock-creation libraries will usually give us a way of doing this.)
- `URL` is its own class, and good unit testing practices would therefore seem to suggest it should be mocked. But, in reality, I doubt most people would bother to create a mock for `URL`. A `URL` object is cheap and easy to create, and mocking it would probably take more effort than is warranted. (Consider also: we don't mock `String` or `Integer` or `Double` objects; why not?) No marks penalty for suggesting that `URL` could be mocked, though, since that is true and in accordance with the principles we've seen.

Question (c) - test cases – cont'd

Test case 1 - normal case

Description: This is the "normal" case, in which the base case is selected for all partitions.

Setup:

- Create a mock task list, in which a task with ID 'T012345678' exists
- Create a mock 'Predictor' object, which in response to all queries returns 0.5.
- Create a mock URL-fetching service, which always returns a known weather report.
- Construct an instance of the class `assessWeatherImpact` belongs to.

Inputs:

- taskID: 'T012345678'
- city: 'Arizona, TX'
- url: `http://tempestuous.com/weather/arizona`

Expected output: 0.5

Question (c) - test cases – cont'd

Test case no. 2

Description: This tests the case where the task ID is not in the task list. (We will assume an `IllegalArgumentException` is thrown due to the task ID not being valid.) We probably *still* want to set up the same mocks as before, because we don't want to accidentally make real network requests – even though we *expect* to see an exception thrown.

Setup:

- Create a mock task list, in which a task with ID 'T012345678' does **not** exist
- Create a mock 'Predictor' object, which in response to all queries returns 0.5.
- Create a mock URL-fetching service, which always returns a known weather report.
- Construct an instance of the class `assessWeatherImpact` belongs to.

Inputs:

- taskID: 'T012345678'
- city: 'Arizona, TX'
- url: `http://tempestuous.com/weather/arizona`

Expected output: An `IllegalArgumentException` should be thrown.

2. Logic-based testing

One of the requirements you are given for the product is as follows:

> - If (i) the weather is wet or the conditions are extreme, and (ii) the task is outdoors, flag the task as needing a weather impact assessment.

a. Suggest an appropriate predicate which could be used to represent the logic conditions in this requirement.

   You should include a list of any variables or functions you use, and explain what they represent. (5 marks)

b. Explain how you would make each clause in the predicate *active*. Choose a particular clause, and provide test cases in which that clause is made active, and set to true and false, respectively. (8 marks)

Suggested answer length: $\frac{1}{2}$ page to 2 pages.

Question (a)

An appropriate predicate would be: $(w \vee x) \wedge o$

where:

- w is a Boolean variable representing whether the weather is wet
- x is a Boolean variable representing whether conditions are extreme
- o is a Boolean variable representing whether the activity is outdoors.

There should be *no other variables* in our predicate; if we imagine this being in an `if` statement, it would look like:

```
1  if ((w || x) && o) {
2    // code to flag the task ...
3  }
```

In particular, "flagging the task" is *not* represented by a Boolean variable; rather, it's an action we take.

Question (b)

In general, we make a clause active by making the result of the predicate as a whole *dependent on* that clause.

So in this case, to make each of w, x and o active:

- w - set o to true and x to false.
- x - set o to true and w to false.
- o - set both of w and x to false.

For our test case, we will consider clause w.

For simplicity, we will assume that `weatherIsWet`, `conditionsAreExtreme` and `activityIsOutside` are parameters to a method `flagTask`; in a real test case, we would probably instead have mock objects representing classes that (e.g.) report on weather conditions, check for emergency condition warnings, etc.

Test case 1

Description:	This represents the case where 'weatherIsWet' is active and set to 'true'.
Setup:	*(We assume no setup is needed.)*
Inputs:	• weatherIsWet: `true` • conditionsAreExtreme: `false` • activityIsOutside: `true`
Expected output:	The task is flagged as needed a weather impact assessment.

Question (b) cont'd

Test case 2

Description:	This represents the case where 'weatherIsWet' is active and set to 'false'.
Setup:	*(We assume no setup is needed.)*
Inputs:	• weatherIsWet: `false` • conditionsAreExtreme: `false` • activityIsOutside: `true`
Expected output:	The task is not flagged as needed a weather impact assessment.

3. Syntax-based testing

The Agendum app can be installed on desktop computers and run from the command-line.

You are given the following requirements for this version of the app:

- It should be possible to invoke the app with either of the flags `--help` or `--version`. When this is done, the app should print user documentation, or the app version, respectively.
- If neither of the above flags is given, then the app should take two arguments: the name of a tasks file, and the URL for a weather server.

a. Give a *grammar* (using the BNF-based syntax we have used in lectures) that describes all the ways the app can be invoked. You should assume every invocation of the app starts with the word "`agendum`". (10 marks)
b. Explain how many tests would be required in order to have *production coverage* for the grammar. Justify your answer. (5 marks).
c. Provide one test case that could be used to test a particular production. Clearly state any assumptions you need to make. (5 marks)

Suggested answer length: 1–2 pages.

Question (a)

Discussion and assumptions

There are many equivalent ways of writing the same grammar, and many different assumptions that could be made about how to test the portions of the grammar relating to *filenames* and *URLs*.

Here, we will assume that we are running *system tests*: tests where we invoke the program from the command line and check its behaviour. (It's also equally possible to write unit tests or integration tests based on a grammar, but we won't do so.)

Regardless of whether we're running a system or unit tests – the overwhelming likelihood is that we will rely on some other library to parse URLs (e.g. the Java standard library). If so, there's no point spending too much effort on the intricacies of what makes up a valid URL, because then we wouldn't be testing *our* software – we would be testing the URL-parser library. Likewise, there's no point specifying in too much detail of what a valid filename looks like.

(This is much the same reasoning as when we consider whether it's worth writing test cases that do nothing but pass `null` parameters to methods to see if a `NullPointerException` is thrown. The answer is, it is almost certainly *not* worth writing such tests, and we could better spend our time on other things. The only thing that running such tests would prove is that the Java Virtual Machine (JVM) indeed works the way we expect – so we'd effectively be testing the *JVM*, not our software.)

So as far as our grammar goes, we will simply assume that `<filename>` and `<URL>` are special terminal symbols that can't be decomposed any more.

Submissions which did include grammar rules for files and URLs still get full marks (as long as they are roughly correct), but it's always worth bearing in mind that we have only finite time to spend on testing, and will often need to make simplifying assumptions when modelling our system.

The simplest grammar for our program, then, is just:

```
1 <invocation> ::= "agendum" "--version" |
2                  "agendum" "--help" |
3                  "agendum" <filename> <url>
```

Question (b)

Production coverage involves testing each of the possible alternatives, or *productions*, found within a rule.

For our grammar in part (a), there are just three alternatives; therefore to achieve production coverage, we need three tests.

Question (c)

The following test case can be used for a system test. Under "expected output", we give a high-level description of what we expect to see; if we were automating the test (always a good idea), we'd specify properties that the output of the tool should have.

For example, if testing that "`--version`" works: if we know that the output is always of the form:

```
agendum version:  0.2.1
```

then our automated tests could just check that the output is in roughly in that format (as opposed to, say, beginning with the string "`agendum help documentation:`", which would signify we were getting quite the wrong behaviour).

Description:	The `agendum` tool is invoked with the `--version` flag.
Setup:	The command line tool has been compiled and is in the current directory.
Inputs:	Invoke the tool at the command line with `./agendum --version`
Expected output:	The tool version information is shown.

Submission

Submit your assignment as a single PDF report using **cssubmit**. Please do not use other file formats (e.g. MS Word or RTF).

Your report should meet the following requirements:

- It should be in PDF format, and use A4 size pages.
- The font for body text should be between 9 and 12 points.
- It should contain numbered headings, with useful heading titles.
- Any diagrams, charts or tables used must be legible and large enough to read when the PDF is displayed at 100% size.
- All pages (except the cover, if you have one) should be numbered.
- If you give scholarly references, you may use any standard citation style you wish, as long as it is consistent.
- Cover sheets, diagrams, charts, tables, bibliographies and reference lists do not count towards any page-count maximums.