DIT636/DAT560 -
Assignment 1: Quality and System Testing

Due Date: Sunday, February 12th, 23:59 (Via Canvas)

There are three questions worth a total of 100 points. You may discuss these problems in your teams and turn in a single submission for the team (zipped archive) on Canvas. Answers must be original and not copied from online sources.

Cover Page: On the cover page of your assignment, include the name of the course, the date, your group name, and a list of your group members.

Peer Evaluation: All students must also submit a peer evaluation form. This is a separate, individual submission on Canvas.

Problem 1 - Quality Scenarios (30 Points)

Consider the software for air-traffic control (ATC) at an airport (say, Gothenburg Landvetter Airport). ATC is a service provided by ground-based air traffic controllers (the users of this system) who direct aircraft on the ground and through controlled airspace with the help of the software.

The purpose of the ATC software is to prevent collisions, organize and expedite the flow of air traffic, and provide information and other support for pilots.

The software offers the following features:

- Monitors the location of all aircraft in a user's assigned airspace.
- Communication with the pilots by radio signal (via voice over IP/VOIP).
- Generation of routes for individual aircraft, intended to prevent collisions.
- Scheduling of takeoff for planes, intended to prevent potential collisions.
- Alerts of potential collisions based on the current bearing of all aircraft.
  - To prevent collisions, ATC applies a set of traffic separation rules, which ensure each aircraft maintains a minimum amount of empty space around it at all times.
  - The route advice can be either of "mandatory" priority (to prevent an imminent collision, pilots should follow this command unless there is a good reason not to) or "advisory" priority (this advice is likely to result in a safe route, but a pilot can choose to ignore it).

You may add additional features or make decisions on how these features are implemented, as long as they fit the overall purpose of the system. In any case, state any assumptions that you make and describe any features you create.

For this air traffic control system:

- **Identify one reliability, one availability, one performance, one scalability, and one security requirement (non-functional) that you think would be necessary for this software.**
- **Develop a quality scenario for each requirement to assess whether the final air traffic control system will fulfill the desired quality attribute.**

Use the quality scenario format from Lecture 3: Overview, System State, Environment State, External Stimulus, Required System Response, Response Measure.

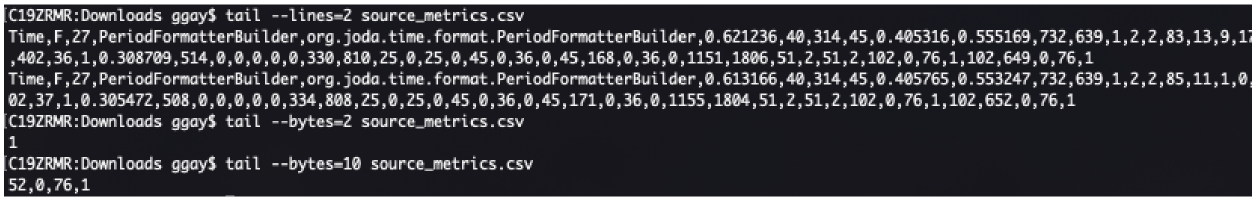
State any assumptions you make about the design or functionality of this software. Requirements should be specific and testable. Scenarios should have single stimuli and specific and measurable system responses. **Please come up with your own original scenarios. Do not take the examples from class and lightly adapt them.**

Problem 2 - System Test Design (40 Points)

The `tail` utility is one of the most common command-line tools used in Mac OS, Ubuntu, and other Linux or Unix-based operating systems.

```
tail [ONE OR MORE OPTIONS] [FILENAMES OF ONE OR MORE FILES]
```

tail takes in one or more options and one or more filenames and then prints the last N (by default, 10) lines of the listed files to the screen. For example:



```
[C19ZRMR:Downloads ggay$ tail --lines=2 source_metrics.csv
Time,F,27,PeriodFormatterBuilder,org.joda.time.format.PeriodFormatterBuilder,0.621236,40,314,45,0.405316,0.555169,732,639,1,2,2,83,13,9,1
,402,36,1,0.308709,514,0,0,0,0,0,330,810,25,0,25,0,45,0,36,0,45,168,0,36,0,1151,1806,51,2,51,2,102,0,76,1,102,649,0,76,1
Time,F,27,PeriodFormatterBuilder,org.joda.time.format.PeriodFormatterBuilder,0.613166,40,314,45,0.405765,0.553247,732,639,1,2,2,85,11,1,0,
02,37,1,0.305472,508,0,0,0,0,0,334,808,25,0,25,0,45,0,36,0,45,171,0,36,0,1155,1804,51,2,51,2,102,0,76,1,102,652,0,76,1
[C19ZRMR:Downloads ggay$ tail --bytes=2 source_metrics.csv
1
[C19ZRMR:Downloads ggay$ tail --bytes=10 source_metrics.csv
52,0,76,1
```

Note: If you have a computer running Mac OS, any Linux distribution, or the Windows Subsystem for Linux, you can open a command line and try this tool yourself. There are also browser-based Linux shells you can use to try the tool[1]. Be aware that this exercise is based on the Linux version of the tool, so some of the options do not work under Mac OS.

We have included an excerpt of the documentation for the tool below. After reviewing the documentation and trying this tool, use the process described in Lecture 5-6 to identify the choices, representative values, and constraints that you would use to create test specifications for this tool:

1. Given the explicit parameters (the list of options and the filename) and other environmental considerations (like the contents and existence of the file), identify the **choices** you control when testing. These are things that you can control that can affect the outcome of executing the function.
2. For each **choice**, identify **representative input values**.
    a. These should not be concrete inputs (e.g., “4”) but abstract types of input that could change the outcome (e.g., “a number > 0 and < size of the file”)
3. Identify **constraints** (IF, ERROR, SINGLE) that limit the combinations of representative values that will be explored when testing.

See page 187 in the Software Testing and Analysis textbook for an example solution to a similar problem discussed in the book. See Exercise Session 2 for another example of this process.

---

[1]For example:
- https://cocalc.com/doc/terminal.html
- https://bellard.org/jslinux/
- https://copy.sh/v86/?profile=linux26
- https://www.tutorialspoint.com/unix_terminal_online.php

Note that you do not need to create the final test specifications for this problem, just identify the choices, representative values, and constraints.

Hint: A user can provide multiple options at once. As a starting point, you might want to look at the options as potential choices and the values that can be provided for an option as representative values. Read the documentation carefully, and note that some options interact with other options or with the contents of the files provided. Constraints and representative values should reflect these interactions.

Tail Documentation

tail [ONE OR MORE OPTIONS] [FILENAMES OF ONE OR MORE FILES]

Print the last 10 lines of each FILE to standard output. With more than one FILE, precede each with a header giving the file name. With no FILE, or when FILE is -, read standard input.

Note – when a flag uses capital letters for input (for example, --pid=PID), that means that the user supplies a value. When lower-case is used (for example, --follow=name), that means the literal word is used (name, in this case).

OPTIONS

- --bytes=K
  - output the last K bytes; or use --bytes +K to output bytes starting with the Kth of each file
- --follow[={name or descriptor}]
  - **Note: "name" and "descriptor" are the literal values. For example, you could execute the command `tail –follow=name filename.txt`.**
  - output appended data as the file grows; an absent option argument means 'descriptor'
- -F
  - same as --follow=name --retry
- --lines=K
  - output the last K lines, instead of the last 10; or use --lines +K to output starting with the Kth
- --max-unchanged-stats=N
  - with --follow=name, reopen a FILE which has not changed size after N (default 5) iterations to see if it has been unlinked or renamed (this is the usual case of rotated log files)
- --pid=PID
  - with --follow, terminate after process ID, PID dies
- --quiet
  - never output headers giving file names

- --retry
    - keep trying to open a file if it is inaccessible
- --sleep-interval=N
    - with --follow, sleep for approximately N seconds (default 1.0) between iterations; with inotify and --pid=P, check process P at least once every N seconds
- --verbose
    - always output headers giving file names
- --help
    - display this description text and exits
- --version
    - output version information and exit

If the first character of K (the number of bytes or lines) is a '+', print beginning with the Kth item from the start of each file, otherwise, print the last K items in the file. K may have a multiplier suffix: b 512, kB 1000, K 1024, MB 1000*1000, M 1024*1024, GB 1000*1000*1000, G 1024*1024*1024, and so on for T, P, E, Z, Y.

With --follow, tail defaults to following the file descriptor, which means that even if a tail'ed file is renamed, tail will continue to track its end. This default behavior is not desirable when you really want to track the actual name of the file, not the file descriptor (e.g., log rotation). Use --follow=name in that case. That causes tail to track the named file in a way that accommodates renaming, removal, and creation.

Problem 3 - System Test Design (with Postman) (30 Points)

Consider the following REST API: http://dummy.restapiexample.com/ (there is more documentation at the URL). This API surfaces the following functions:

Route	Method	Example URL	Description
/employee	GET	http://dummy.restapiexample.com/api/v1/employees	Get all employee data
/employee/{id}	GET	http://dummy.restapiexample.com/api/v1/employee/1	Get data for a single employee[2]
/create	POST	http://dummy.restapiexample.com/api/v1/create	Create a new record in the database[3]
/update/{id}	PUT	http://dummy.restapiexample.com/api/v1/update/21/	Update an employee record.
/delete/{id}	DELETE	http://dummy.restapiexample.com/api/v1/delete/2/	Delete an employee record.

1. **If you do not have one already, create a free account at https://www.postman.com/**
2. **Use Postman to create tests for this API. Create 10 test cases in total, with <u>at least one</u> test case for each of these five API functions.**
    a. **A test case has its own input (REST method + request body) and one or more assertions on the output (called "tests" in Postman). Do not simply submit 10 assertions!**
    b. **Test cases should test both normal functionality as well as handling of erroneous input. Test cases may, optionally, also address performance aspects such as response time.**
3. **In your report, include the request type, body, assertions ("tests"), and any other information that you used as part of your test cases. Please explain each test case - describe the goal/purpose, as well as the assertions you used to verify the behavior.**

You may use the process described in Lectures 5-6 (with choices, representative values, and constraints), but do not have to. Use your creativity and the test snippets provided by Postman to ensure that this API works as expected. Try error cases, like an ID of 0, and see if the API works as expected. You may want to consider adding tests to check for quality goals such as performance as well.

[2] If you want to verify the normal results of /employee/{id}, use an ID between 1-24. IDs of 0 or higher than 24 are great for checking "error" cases.
[3] Note - the POST/PUT/DELETE methods do not actually make permanent changes in this example. You will get an appropriate response, but the record will not actually be created, updated, or deleted. You can use the result body to verify the "results" of running these three functions.

We assume you have learned some basics about testing of APIs in DIT341, but please talk to us if you have questions. For a starting place on testing in Postman, see https://learning.postman.com/docs/writing-scripts/test-scripts/

You are welcome to submit the .postman_collection file as well, but it is not required as long as the tests are shown and explained in your report.

Note: The test cases do not have to pass! The tests should reflect how you think the system *should* work, so if they fail, that indicates the system is faulty (in your view).