Software Testing Document

March 17, 2024

Team

Kowalski

Sponser

Igor F Steinmacher

Team Mentor

Saisri Muttineni

Team Members

Erick Salazar, Bailey McCauslin, Jake Borneman, Nick Wiltshire

Revision

Version 1

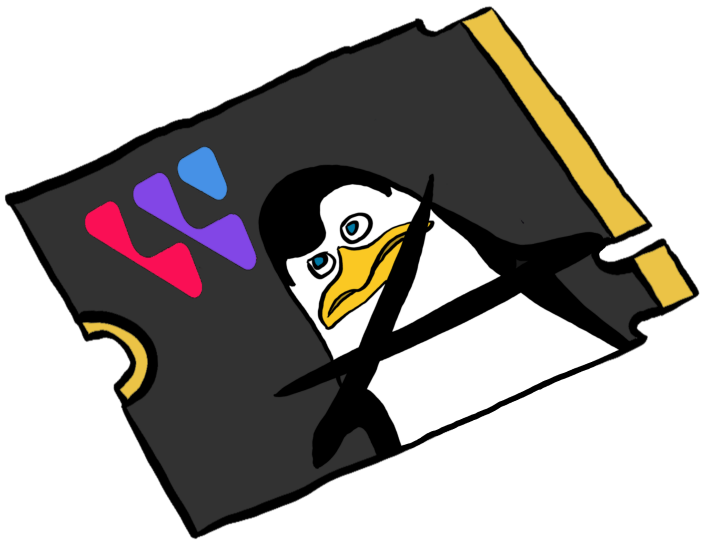

TABLE OF CONTENTS

Introduction

Currently many companies test SSDs for different uses, like desktops, appliances, and cloud computing to make sure the product quality meets the customers' product specification requirements. However, the industry currently lacks the following capabilities, significantly slowing down the research and development process:

- No data collection system, gathering and storing performance data from kernel-level.
- No centralized system for gathering collected data, analyzing it for thresholds, and visualizing results.
- Due to the lack of a centralized system, there is no way to automate the entire process, from data collection to displaying results.

Our framework would enable the collection, storage, analysis, and provision of visibility and monitoring through visual dashboards. The absence of such a framework hinders the automation of proactive failure detection in real-time and near-real-time scenarios. This delay in issue detection not only incurs time and financial costs for the companies but also adversely affects customer satisfaction metrics.

Now, with the complexion of the program, there is room for bugs and issues to occur through the many steps within it. To prevent such things, software testing is a necessity for the program to be as refined as possible. In essence, software testing is simply doing activities and cases where a software is tested to expose any openings or missing coverages. This could be through things called unit testing where the software is directly tested in certain cases to look for issues. Integration testing which is the process throughout the program where each part is registered and the data in between is properly transmitted to one another. Lastly, Usability testing in which the user would use the program and the interfaces that exist to ensure each step works. These types of testing are what the program would be relying on to enhance it's performance and make the user happy.

Leading into the plan, the program would integrate all three priorly mentioned testing styles to ensure the best results from the program. The unit tests would mainly target helper files and functions. This would consist of data existing, checks and expected output results as most of the functions involve data manipulation in some form. Notably the Sanitizer.py file

would have the most unit tests as this file contains many helper functionalities. For Integration, the data is kept checked while traversing through the program from start to finish as that consists of the pipeline. This is more difficult to track as the pipeline automatically culls from the kernel level all the way towards the database but in between, there can be messages and checks to show that the data is correctly done. Usability testing would be the most difficult as the user has to be slightly technical to understand what the data is representing. This might be the most important though as the user could input certain information and actions that would break the code. With all of these planned out, the planned testing is set to be balanced out amongst each other with a heavier priority of the unit testing and the Usability testing. Especially due to the nature of this application, it is heavily influenced by the user that is wanting to collect the data and tracking that all functionalities are working properly.

Unit Testing Introduction

As mentioned with tracking all functionalities and ensuring they work as intended, unit testing would be the best case to test an individual functionality. These can be secluded to a single function of a program and possibly in conjunction with other functions based on the idea of what is wanted to be tested. For the purpose of the system, the intent of these would normally target an individual function and see if certain situations the program would be tested in would either be sufficient or break upon testing. The results may vary depending on the data collected so unit testing such things would be difficult. Although a way to get around that would be to have a test file to be manipulated or worked on by the function. These files would have intentionally incorrect data that all functions would derive from to help with the unit testing while the originally collected data would be also used for other situations.

The testing libraries and framework that would be used for unit testing are PyTest, PyUnit and mutation testing. With PyTest and PyUnit, these are mainly static testing structures that would help in ensuring values, functions and data are correct through things such as assertions and data manipulations. Mutation testing would be useful in the sense of manipulating the code in any way possible to identify possible hidden bugs that may not be obvious unless done. Mutation can greatly expand the coverage of the code and ensure it is the best it can be.

The Unit testing would be most present within these files:

- Sanitizer.py: Main sanitizer for all csv data files collected from the bt files. Has multiple functions and functionalities that require static test cases and mutations to ensure high coverage and can work through different situations. Will have additional functions and data files for test cases to test the functionality. Best observed with a static file to track modifications of the original file.
- Formatter.py: Main formatter for non-immediate csv file conversions like Bio_Latency and Bio_Error bt files. Has multiple functions based on the file that is inserted into the different classes. Tests work best with modified/manipulated data files to ensure coverage.
- guiConfig.py: Creates the tkinter GUI for inputting information from the user to run the program. Has multiple functions where data transmissions occur and sets the GUI to look a certain way. Best for testing data existence and mutation to see possible failure.

Unit Testing Plan

As mentioned above, the plan is to use PyTest, PyUnit and mutation. Most of the program would have testing cases of these but the focus in this document would be on the previous python files mentioned above.

1. Sanitizer.py

- PyTest + PyUnit:
    - Remove_Request function: This would be targeted through multiple assertions by asserting if the value exists, has been modified from the original or hasn't been modified. This would be done by grabbing test file(s) and inserting with random removals of processes and asserting what to expect. Should always have data sent out either unmodified or modified based on the process_list inserted. No hardcode with exception of process removal.

      Ex: AssertEquals((Remove_Request(file_data, remove_Process = None)), Original_File)

- Format_info function: Responsible for formatting information into non-byte string types. Mainly focus on seeing if original changes occurred through assertnotequals and inserting multiple files that do not have the same format as the purpose of this function. Meaning another file not having byte strings data.

  Ex: AssertNotEquals(Format_Info(Test_File), Original_Test_File))
- DataMerge: Merges all similar data into one giant data file. Has separation through the ops argument and certain activity to keep each row unique. Tests to see if data is pulled when none exists and the ops is not a part of the const python file. Checks to see if the returned data frame is empty and if there are combination issues.

  Ex: AssertEmpty(data_merge(ops))

  AssertExist('ops_Combined.csv)

- Mutation:
  - Regex_Extractor function: Extracts certain information from the byte string wrapped objects and overrides it as a string with missing parts. Due to the regex pattern, there is priority on mutating this to test if the information is properly pulled and formatted through mutations. Soon followed with coverage testing by using a test file to make comparisons between the mutated file affected by regex mutations.
  - Data Merge: Similar to regex extractor with affecting the pattern but have to isolate the code glob. Glob can severely destroy a computer if included in mutation testing and file modifications. Looks for overriding/different collections based on the collected data and stores in the combined.csv file. Comparison function created to test relativity between the mutated data file and the data original file.

2. Formatter.py

- PyTest + PyUnit:
    - Latency_Speed_Separater function: Separates the process speed and number of processes from a raw text file and sends back a culmination together to be stored later into a CSV file. Would test if the data sent has certain values attached to it and looks if the matches are null or not existing.
    - BioPattern_Formatter function: Grabs the raw text file and formats into CSV file. Likely limited testing as the raw text file has separated information and headers existing. Look into testing files that are shoved in and ensure it only works for BioLatency derived bt files and not simply other files. (assertEquals and assertExists, helper functions may be created)
- Mutations:
    - Latency_Formatter Function: Formats all of the Latency based raw text files into a CSV through multiple functions and regex expressions. Mutations would mainly target the regex expressions, operands used for comparing and calculations, and assignments. This will greatly expand coverage through many comparisons and checks of existing information.
    - Latency_Speed_Separater function: Mutate namely the if-else, try-except matches and the locations of the information collected through regex. Doing this would expand coverage takes and avoidances while also modifying the information collected.

3. guiConfig.py

- PyTest + PyUnit:
    - Submit function: Grabs all the user inputs from the GUI and applies it into a dictionary set where the Trace program would then activate and work with the inputs given. Unit tests would ensure that the assignments done would be accurate and sent out properly into the dictionary used for the

Trace file. Additional functionality added to ensure validation checks and intentional insertion for breaking is done

- Mutations:
    - Submit function: Mutates the entire function to take coverage and modifiability. Targets mainly validation checks and information assignments that are used outside of the function. Could also mutate to where actions fail such as destroying the GUI when the user submits the information.

Integration Testing Introduction

Integration testing is a type of testing where the program is tested on the interaction between one another. For this program specifically, we have multiple connections that hand-off information to one another. Simply put, the pipeline created within the program is this:

GUI -> Data Collection -> Data Sanitizer/Formatter -> SQL Databases -> Visual Dashboard

This is important to test because throughout the program, there is data that is being handed around to all participants. If the data fails or is corrupted in any manner, the program would fail. Seeing the pipeline handing off the data without issues would ensure that the data can safely travel from the low-level kernel system to the front-end visual dashboard.

To do this type of test is quite simple. The program must simply run and be capable of displaying results onto the dashboard. Throughout the program, there are multiple checks that already exist to catch data exchange failures. The last exchange being displayed onto the dashboard so if at any point of the program the data doesn't get exchanged properly, there would either be exception cases where it would continue to another portion of the program or stop all together.

Integration Testing Plan

1. GUI to data collection:
    - First in the hierarchy of our 5 layered architecture we have a GUI that sends the user inputs to our data collection program(trace). The connection between the two is extremely important because the trace program configures itself based on those inputs, so if something goes wrong then the whole program will break. We are going to test this part through something called “black-box testing”. Meaning we will test through the interface instead of vigorously writing tests in the software. We will still be writing tests to assert truth on an inputted item by the user but that's just to know that the test succeeded. We will vigorously test the input fields that the GUI offers and make sure the connection between the two programs is stable. If there is any unexpected output, crashes, or other bugs that occur during the runtime given certain outputs, then we know we must patch.
2. Data collection to Sanitizer/Formatter:
    - Now, the data we get back from the data collection isn't necessarily in the format we need so we must run the output through a data sanitizer/formatter. To check that these things are integrated properly we will be checking that all files that are expected from a run of the trace program are in the output directory. Once that is verified, we can run these files through the sanitizer and get back the expected output. We however, need to check that this output is in the correct format. To do this we need to check that the headers are correct, the information hasn’t been modified, and that it is in the correct format. If we notice anything off, we know there is a bug somewhere within the sanitizer/formatter or with the output the trace program produced.
3. Sanatizer/Formatter to SQL database:
    - Once the Sanitizer/Formatter has been completed, the files that are converted to a csv file would then be uploaded to the SQL Database. As mentioned, these files are located in the output folder where the SQL database would grab whatever

files are stored and query them into the proper tables they are associated with. There are a few ways to test this situation. First, look into identifying the tables being created and populated with the data. If SQL works as intended, the data from the local system to the database is uploaded and queried into the database, showing that the integration works. Another form of integration testing is when an additional file is added towards the database, it would append new results instead of overriding the data that had already been there. When any of these were to not occur, it is known that there is a bug in the transmission between the two parts, either losing the data from the Sanitizer/Formatter through a CSV file or the database not capable of collecting the information generated from these files.

4. Database to data visualization:

- The database will then be queried by our flask server, and the data will be sent to our visual dashboard through http requests. The data is verified on both the flask server and our typescript front end app, not only ensuring that the right queries are being made, but that the right data is being sent to the dashboards. In the server, there are error checks in place to ensure that the database is running and successfully being queried, as well as logs representing forwarded data. On the dashboards' frontend, there are error checks for successful http requests to the server, ensuring the server is running correctly and that permissions settings are correct for the database. Then, there are error checks for getting actual data from the server, as well as logs that display the data in its raw format which verifies we are getting the right data in the right format. Overall, there are multiple steps in getting from the database to the frontend dashboards that all have error checks and logs, enabling us to identify exactly where issues are coming from and why they are happening.

Usability Testing Introduction

Lastly the program would have usability testing to simulate how a user would interact with the program itself. Usability testing is testing where the program would be used by the user through different actions and variances to see if they can access the main functionalities of the program without knowing the abstractions. The goal for using such testing is to understand how the users would use the program and what ways they would interact with it, leading to possible breaking cases. It also helps the developers understand what users would see when interacting with the program which could expand, modify, or simplify understanding for users. Who the program expects to be used by are people that want to know what is going on in the kernel level system, people experienced in kernel level information, and R&D and Hardware Device Testers. All would like to understand how a product would like to interact with the kernel level data and likely use it to enhance or fix the hardware being observed.

This would be the shortest testing as the program's purpose is for automation and simplicity through abstraction on the back end code so that the user doesn't need to know much about the program to use. The only possible testing suits the user could target would be where the GUI interactions exist which would be the very beginning of the program and the visualization dashboard. The GUI contains certain inputs that take the user's request and runs it while the visualization is an interactable dashboard for whenever the program completes the data collection and automatically stores it into the database. Notably within the Input GUI, there is a process collector in which the BT file must be created prior for collection. There are pre-made ones that users could find or make but for now, there are a few in the github repository of this program. The best opportunity for usability testing would be to have an R&D and Testing department of a hardware device firm to test multiple hardware parts with the program to ensure variance, consistent collections, and display of results. A more accessible opportunity would be for multiple users to use the program to simulate a similar result that would be developed from an R&D and Testing department.

Usability Testing Plan

How the testing would work is we would have multiple interactors and submit a review of the program based on certain criteria: Easy to understand, accessibility, helpfulness, and additional comments. This would be given to users when they download the github repository for their use and if they choose to submit it, it would be sent to an email dedicated to the survey. There would be no need to report back device information unless the user would like to. This survey would also be given to the R&D and Testing department so that they can also submit it for review and understanding on a larger scale.

When it comes to the GUI interactions, we would request first stage (people to simply test the program and not authentic users) users to work with the GUI based on how they want to use it. This would give understanding on what the user first thinks when interacting with the program and further fail proof the software from unintentional breaking of the program to complete failure of starting. After having them test on their own volition, there would be instructions given to them to follow and see how they react towards the program. That way, when a user is confused or having difficulty understanding the program based on the instructions, it shows there needs to be simplicity or further instructions within the program to teach the user. This first stage group would consist of half technical proficient members that understand the intricacies of a computer Operating System and half normal computer users. These simulate both inexperienced and experienced members of the field in which allows the growth for the inexperienced while refinement for the experienced. This process would take about a few weeks for the first stage users but will continue indefinitely so long as the program is used and the survey is sent from other members.

For the visualization dashboard, there will be two focuses and cases. Focus one would be purely on the dashboard while focus two is the entire process from the kernel level system to the dashboard. Case one is purely the user interacting with the dashboard with static results while case two would be running the program to retrieve new data results to be displayed on the dashboard. This testing suite would use the same groups as the GUI interactions. For Focus one and two, it would also follow similar protocols for how the GUI interactions would be, which is letting the users navigate and interact with the dashboard without any hints and then a

set of instructions to follow. Within case one, the data stored would be a preset that has been generated by the team in which the user uses without knowing the database side. Having them not know the preset within the database can show they would interact with the dashboard without first-hand knowledge of the data, showing what implementation might be required such as descriptors, dashboard interaction instructions and so on. After giving instructions, the first stage users could identify things on the instructions that would benefit being implemented onto the dashboard. For case two, this would be done at the end of the other usability testing as this would be interacting with the entire program. The first stage users use their own data collection files that would store into the system and display on the dashboard, this would show how they should interact with the system at an entire level. This result would be displayed on the dashboard as it should append to the current database, modifying the current results.

Conclusion

In all, every software that exists needs to have testing to confirm that the program is working as intended. Specifically for the system we are creating, there are many ways that the program could fail such as the GUI breaking and failing to grab information from the user, the Data collection not collecting anything or going on indefinitely, failure of data transport through other functions, etc. Notably, the more complex the system becomes, the programs would become more prone to errors and malfunctions. Testing in general helps keep the product refined for the users to make and have the developers do less work so long as the errors are snuffed early on instead of when the product has been sent out.

For the program, we had discussed multiple ways to test the program such as; Unit testing to ensure functionality of the program works as intended, Integration testing to ensure the program can properly work between each steps of the program, and Usability testing where the different types of users would be free to find ways to break the program either through their findings or through a set of instructions given by us. Unit Testing is one of the main ways for testing the program due to there being many functions serving individual purposes to have the program properly work. Notably the functions Formatter, Sanitizer, and GUI config would be targeted as their functions serve heavy purposes in collecting and formatting data. Integration testing is the simplest one for the program as the program has a step-by-step process that is

linear so keeping track of information between each section of the program is easy. Lastly for Usability testing, we would have various groups that would assist in it, namely the highly technical people and the less technical people. Their input would be valuable as they would find ways to break the code and identify possible confusions for us to integrate.

Once these tests have been completed, the program would be much more refined for the user to operate. It would also be easier for developers to understand the changes and testing suites that have been created for the program. When they look into it, the testing suites would have records such as input from the usability testing and process intention through the integration testing. As for unit testing, that would have a naming schema for itself based on the way unit tests are integrated such as Formatter_Unit_Test and Formatter_Mutation respectively. Once everything has been done, the users would be capable of operating this program with ease.