Command and Control System Software Development

Nashir Janmohamed
Kennedy Space Center

Major: Computer Science and Engineering

Spring 2019 Session

Date: 19 April 2019

Command and Control System Software Development

Nashir A. Janmohamed[1]
John F. Kennedy Space Center, Kennedy Space Center, FL, 32899

With the first launch of the National Aeronautics and Space Administration's Space Launch System heavy-lift expendable launch vehicle and Lockheed Martin's Orion Multi-Purpose Crew Vehicle scheduled for the year 2020, there exists a need to complete development of a new command and control system that will provide systems monitoring and launch control for NASA's Exploration Missions. One remaining task necessary for completion of this command and control system is to create and maintain comprehensive unit tests of the control system software packages. These tests should verify that the implementation of all required and desired functionality works as intended. This testing infrastructure is mostly in place, but the control system's open source automation server still reports software "bugs" (possible flaws or failures which may lead to unintended behavior) and intermittently failing unit tests. Since code correctness is of critical importance for human rated software systems, I was assigned to diagnose the root cause of failing unit tests, eliminate non-determinism in these tests, and fix bugs as reported by the automation server.

Nomenclature

API	=	Application Programming Interface
COTS	=	Commercial Off-The-Shelf
GUI	=	Graphical User Interface
IDE	=	Integrated Development Environment
KSC	=	John F. Kennedy Space Center
LCC	=	Launch Control Center
NASA	=	National Aeronautics and Space Administration
NTRS	=	NASA Technical Reports Server
SLS	=	Space Launch System
UX	=	User Experience

I. Introduction

The system on which I worked is a Class A (Human Rated Space Software System[1]) command and control system that will be used to launch the Space Launch System (SLS) expendable launch vehicle[2] for National Aeronautics and Space Administration's (NASA) Exploration Missions[3]. The John F. Kennedy Space Center (KSC) has been the primary launch center for human spaceflight since 1968, with the Apollo, Skylab, and Space Shuttle programs among those that launched from KSC[4]. SLS is a direct successor to the Space Shuttle program, undertaken with the intent of sending humans farther into space than ever before. Upon completion of successful manned journeys to the moon, NASA plans to send humans to Mars using the SLS and Orion capsule[3].

My duties as an intern included various tasks, mostly involving testing of different sections of the system. In order to accomplish these tasks, upfront training and account access was necessary to obtain access to the software to be tested. Once this was complete, I was able to begin work on my assigned tasks which included: completing a previous intern's project, working through a list of intermittently failing tests and analyzing what caused the tests to fail, and migrating a portion of the codebase to an actively maintained testing framework to replace the current deprecated Application Programming Interface (API) used for automated unit testing of the GUIs.

[1] Control System Software Development Intern, NE-XS, NASA KSC, Santa Monica College.

II. Work Done

A. Getting started

After familiarizing myself with the technology stack used for development, including the Integrated Development Environment (IDE), programming languages, version control and continuous integration tools, code review software, and automation server, I completed a previous intern's project. This provided a high-level understanding of the control system. This system contains, among other features, various displays for use in displaying information like vehicle health, sensor information, and life support. Prior to completion of the previous intern's project, users in the Launch Control Center (LCC) used a list-based selection GUI to open their desired displays. This list contained display names but did not show the image of the display or the data presented in the display. The goal of the project was to add functionality that shows thumbnail images of each display to the display selection GUI. This modification to the GUI allows the users to quickly find their desired display and eliminates the need to memorize the layout and content of every display based only on the filename in the selection list[5]. Changes required were minimal, but completing the project yielded an overview of the debugging and code review processes. The previous intern's modification seemed to work as intended, but a failing unit test indicated otherwise. In this case, the test was correct, and there was an issue with the display shutdown process. The modified code opened display thumbnails, but did not close the thumbnails when they were supposed to be destroyed in the class shutdown method. This was fixed by modifying the shutdown method to properly dispose of all class member variables.

B. Python script

My next task involved working through a list of intermittently failing tests. Many tests would not fail locally; they would only fail when being executed by the automation server in the process of building and testing the entire project (which took around 3-5 hours). A Python script was made in collaboration with a fellow intern, William Su, to parse build reports generated by the automation server in the interest of expediting the process of analyzing failing tests. The generated data also helped in determining the root cause of their failure.

The Python script created to parse build reports uses built-in Python modules that aid in the process of accessing, extracting, parsing, and formatting the desired data. One module defines functions and classes which help in opening URLs, a second provides an API for extracting and parsing the data format provided by the automation server, and a third implements classes to read and write tabular data in Comma Separated File (.csv) format. The script accesses and parses the object notation representation of the server generated data and creates a .csv formatted list of all failing tests in user-specified builds.

The server contains build data from hundreds of different workspaces and development streams, and the script allows the user to specify their stream of interest. After specifying a stream, the user provides a destination file name that the script will use to create a .csv file of parsed build data. The user is prompted to indicate whether they would like to provide a comma separated list of a set of builds to parse, or whether they would like data from all builds from a particular stream. The corresponding URL for the development stream is then constructed, and the specified build reports are accessed, extracted, parsed, and written to a .csv file. Initially, the tests were grouped by build number, but the current implementation sorts the list of values first by package name, and then by test name within each package. This formatting more intuitively shows the frequency of failing tests, the builds in which it failed, and the error message and stack trace each time it fails.

The .csv generated by the script contains relevant data from the automation server generated build reports, with six columns as follows:

1. Package Name
2. Test Name
3. Build Number
4. Error Details
5. Stack Trace
6. Exclude String

The (1) package name is the file path within the software tree to the failing test. The (2) test name is the name of the failing test. The (3) build number is the number of the automation server build in which the test failed. The (4) error details indicate why the test failed (e.g. expected true but was false). The (5) stack trace is a list of the method calls that the application was in the middle of when an exception was thrown. The (6) exclude string is a string constructed and formatted by the script. The exclude string is used in an existing script run by the automation server to exclude flaky tests from being executed during system builds.

Future modifications to improve the User Experience (UX) and robustness of the tool exist, but consultation with the Software Architect will dictate future direction and modifications. One possible modification would allow the user to specify package name(s), class names(s), and test name(s) if they do not want to examine the builds for all failing tests. Another idea to improve the UX would be to make a simple GUI implementation of all the functionality in the Python script. Lastly, the user may want to search more than one development stream, so the script (or GUI) could prompt the user after parsing the requested input if they would like to parse another set of builds before terminating.

C. GUI Testing API migration

The system performs testing of the GUI components that allow customizability by the user. Testing these graphical components manually is repetitive and incurs a high time cost. To automate these processes, the unit tests use a commercial off the shelf (COTS) API that allows for automated unit testing of GUIs. The API in use for the majority of this software GUI testing is deprecated, requiring migrating a portion of the codebase to an actively maintained testing framework. Migrating the unit tests from one API to the other seemed relatively straightforward, particularly so since the currently maintained API (API #2) claims to be a port of the now deprecated one (API #1). For the most part, the migration was straightforward, and the only discrepancies between implementations were clearly described on the website containing the API #2 documentation. In most cases, the changes were cosmetic rather than functional. In one instance, however, a class member method functioned differently in API #1 and API #2. The method provided a way for the user to simulate keyboard input, useful in the case when it is desired to simulate entering text into a form, deleting a character inside an already filled form, or using the arrow keys to scroll through a list of options. The method worked as expected in API #1, but the API #2 implementation failed to produce the desired behavior, which in turn caused tests to fail.

Upon further inspection, this bug only manifested when using a certain GUI component, working as expected on others. The issue can be avoided a majority of the time, since most "key presses" can be simulated by using an alternate method that enters text into the field i.e.:

```
someForm.enterText("\n"); //to press the enter key
someForm.enterText("\b"); //to press the backspace key
```

but there are many cases where the capability of simulating key presses is essential. There are no ASCII characters for the arrow keys or control key, among others, and so those keys being pressed cannot be simulated using enterText().

It was decided, after exhausting all internal options, to ask for outside help in this case. With the help of fellow intern Michael Braun, I made a simple GUI application and a corresponding unit test using both API #1 and API #2 to show the difference in functionality. The application and unit test were generic and specifically sanitized so as not to share any sensitive NASA information or data. I posted the program, test code, and corresponding runtime error as a question on Stack Overflow and as an issue on the API #2 GitHub page. After meeting and discussing the issue in my daily stand-up meeting, it turned out that another team member had a similar issue, and suggested a simple usage change that fixed the problem. Prior to the end of my work term, I plan to update the README in the software tree's API #2 folder to include instructions on avoiding the problem I encountered when trying to simulate keyboard input.

D. General Unit Test and Bug Fixes

I refactored and modified code in various places within the software tree to eliminate bug warnings and unit test failures. Problems encountered during inspection of existing tests and code varied. The root cause was often some form of thread violation. In most cases, this was fixed by instantiating test instances and putting them on their own thread until their execution finishes. GUI testing APIs #1 and #2 (previously mentioned in section C) both provide a class to detect thread violations, and another class to create new instances of objects on a thread intended for GUI components. Another common error involved the teardown method between tests being implemented improperly. This error was fixed by ensuring all the requisite shutdown methods were called, and that all member fields were set to null.

In many cases, tests did not fail locally, but failed when run on the automated build server, which made them difficult to debug. Instead of being able to step through the program and observe the state at any given point, the stack trace and error messages had to be used as the sole means of root cause analysis. Sometimes, this was enough, as the error message provided sufficient information and the stack trace provided a line number at which to begin analysis, but other times, the build server generated an ambiguous and generic error message with no stack trace. In these cases, the only option was to step through the source code line by line in an attempt to identify areas where the unit test or the source code might have a bug that causes it to fail.

Overall, there was no optimal way to debug and fix failing tests, and the approach needed to be changed from one test to another. This created a bottleneck in that each test had its own learning curve, a setback magnified by my initial unfamiliarity with the codebase. Over time, however, I became more familiar with how different files and packages interacted with each other, which greatly reduced the time needed to start making progress on fixing each failing test.

III. Conclusion

The work done over the course of this internship paved the way for future interns and the operations personnel to more easily analyze and understand failing unit tests, reduced the number of bugs and failing unit tests in the project builds, and provided insight into the intricacies of migrating from one GUI testing API to another. The insights and fixes are some that may not otherwise exist without the work done during my internship. If there are still failing unit tests and bugs reported by the automation server at the end of my internship, the most appropriate course of action would be to have future interns continue this work.

Acknowledgments

I am incredibly grateful to both NASA and Universities Space Research Association (USRA) for providing the opportunity for me to be a part of the exciting and forward-reaching work being done at KSC. Thank you to Jamie Szafran and Jill Giles not only for being amazing mentors and teachers, but also for making me feel at home during the workday. Thank you to Oscar Brooks for welcoming me into your department. Thank you to Jason Kapusta, Tony Ciavarella, Jordan Kiser, Kathy Saint, and Sam Goff for your help with all my work-related questions. Thank you to Kathleen Wilcox and Gwendolyn Gamble at the KSC Education Office for facilitating this once-in-a-lifetime experience.

This internship is something I could only dream about just a few short months ago; having the opportunity to work as part of the team responsible for launching humanity farther into the solar system is an experience I will cherish for the rest of my life.

References

[1]Various, "NASA Software Engineering Requirements," NASA NPR-7150.2B, November 2014.

[2]Various, "Space Launch System Fact Sheet," *NASA, George C. Marshall Space Flight Center* [online], June 2018, https://www.nasa.gov/sites/default/files/atoms/files/0080_sls_fact_sheet_10092018.pdf

[3]Gebhardt, C., "NASA finally sets goals, missions for SLS – eyes multi-step plan to Mars," *NASASpaceFlight* [online], April 2017, https://www.nasaspaceflight.com/2017/04/nasa-goals-missions-sls-eyes-multi-step-mars/.

[4]Various, "Kennedy Space Center Implementing NASA's Strategies," *NASA, KSC* [online], Accessed on: April 1, 2019, https://www.hq.nasa.gov/office/codez/plans/KSCImp00.pdf

[5]Chapa, N., "Spaceport Command and Control System Software Development," *NASA Technical Reports Server (NTRS)* [online], Oct. 2018, pp. 2, https://ntrs.nasa.gov/archive/nasa/casi.ntrs.nasa.gov/20190001410.pdf.