System Applications Software Development and Testing for the Spaceport Command and Control System

Samantha Niemoeller

Major: Computer Science & Mathematics

NASA Kennedy Space Center

2018 Fall Session

Date: 30 NOV 2018

System Applications Software Development and Testing for the Spaceport Command and Control System

Samantha Niemoeller[1]
John F. Kennedy Space Center, Kennedy Space Center, FL, 32899

Known as "America's Spaceport," one of Kennedy Space Center's (KSC) primary responsibilities is the successful preparation for and launch of rockets into space. KSC's Engineering Software Branch has been tasked with creating a new command and control system that will provide check-out and launch control for future rockets and spacecraft. While work on the software began several years ago, development is ongoing and the operators who use the software on a daily basis have requested several features to improve their user experience. My internship in the fall of 2018 involved developing the source code and unit tests for two of these requested features: "Display Data with Persistence" (DDP) and "Save Events Button" (SEB). DDP's primary goal is to aid with ergonomics. Currently, users must press-and-hold on the mouse button to display information about points on a data plot. Once DDP is integrated, users will have the ability to double-click on a data plot to display that same information with persistence. Independent from DDP, the SEB provides users the ability to take information about different events that occur in the control system and save that data into a simple Comma Separated File (.csv) file format for easier analysis at a future time.

Nomenclature

COTS	=	Commercial Off-The-Shelf Software
.csv	=	"Comma Separated File" file format
DDP	=	Display Data with Persistence
ET	=	Events Table
GUI	=	Graphical User Interface
IDE	=	Integrated Development Environment
KSC	=	Kennedy Space Center
LCC	=	Launch Control Center
NASA	=	National Aeronautics and Space Administration
SEB	=	Save Events Button feature

I. Introduction

Known as "America's Spaceport," Kennedy Space Center (KSC) is home to NASA's Exploration Ground Systems (EGS) which is responsible for the oversight and coordination of the successful preparation and launch of rockets and their payloads into space. Under EGS, KSC's Engineering Directorate's Software Branch has been tasked with creating a new command and control system that will provide check-out and launch control for future rockets and spacecraft. This system will be used in KSC's Launch Control Center (LCC) to monitor and control the vehicle and ground support equipment during preparation and launch activities.

Software to run this system has been under continuous development for several years. While full-time engineers are continually developing the software to meet the system engineering requirements, there are several software features that the console engineers have requested to significantly improve their user experience. In fall of 2018, I developed the source code and unit tests for two of these additional features.

The first of these is the Display Data with Persistence feature (DDP). Currently, to display data associated with points on a data plot, engineers using the software must press-and-hold the mouse button to keep the data visible.

[1] Fall Intern, Software Branch, NASA Kennedy Space Center, Santa Monica College, Los Angeles, CA

DDP will give users the option of simply double-clicking to display that same information with persistence, which will aid with ergonomics.

The second feature is the Save Events Button (SEB). When something noteworthy occurs in the control system (such as a sensor's reading being out of limits), it is considered an event and is displayed in an events table (ET) for users to monitor. Currently, to save the data from these events for future analysis, users must make note of each event by hand, which is tedious and could potentially lead to a loss in precision or other related human errors. The SEB circumvents these issues and allows all of the data in an ET to be saved to a Comma Separated File (.csv) file format for easier analysis at a later date.

II. Methodologies

A. Software Used

A high level, object oriented programming language was used to create the source code and unit tests. These were written in an Integrated Development Environment (IDE). Within the IDE is a testing platform that NASA engineers had previously developed in order to simulate the final system application with mock data; this testing platform allowed me to rapidly run mock functional tests to ensure my source code produced the desired behaviors. After passing these tests and creating the corresponding unit tests, the code was integrated into the main codebase using Commercial Off-The-Shelf (COTS) version-control software. A COTS collaboration application was used to facilitate code review. These applications were accessed through a Unix-based operating system, where basic shell scripting was used to configure environment variables and run the source code.

B. Technical Points of Contact

Throughout the development of these features, I worked with the subteam that handles the system applications development for the command and control system. My subteam technical points of contact, Samuel Goff[2] and William Denis[3], were instrumental in pointing me towards code to use as a possible reference and sharing their deep knowledge of the main codebase and programming language. They walked me through each new stage of the software development cycle, answered questions, and provided feedback during code review. I attended the daily subteam tag-ups, where I learned what the other developers were working on, shared my progress, and posed any questions to the group.

In addition to the subteam, Jason Kapusta[4], Kevin Teufer[5], and Tony Ciavarella[6] met with me on a weekly basis to provide in-progress feedback on the functionality of the code and on coding style. This allowed for any course-corrections to be made at an early stage of development, gave tremendous insight into how the console engineers in the operations community would use the features, and allowed for a smooth code review process. Additionally, Jonathan Serrano Otero[7] and Jamie Szafran[8] provided a tremendous amount of assistance with getting my computer set up, helping me navigate the new software, and providing language-specific guidance.

III. Features and the Software Development Cycle

A. DDP

1. Objective

In the LCC at KSC, when sensors and other hardware on the rocket transmits data to the control room, the console engineers have the ability to view these measurements in real-time on data plots. For visual reference, Fig. 1 illustrates a mock-up of the style of data plots that the engineers would see.

[2] Team Co-Lead, NE-XS, NASA Kennedy Space Center
[3] Team Lead, NE-XS, NASA Kennedy Space Center
[4] Software Architect, TOSC Contractor, Kennedy Space Center
[5] LCC Flow Manager, TOSC Contractor, Kennedy Space Center
[6] Software Architect, TOSC Contractor, Kennedy Space Center
[7] Computer Engineer, AST, Ground Data Systems, NE-XS, NASA Kennedy Space Center
[8] Computer Engineer, AST, Software Systems, NE-XS, NASA Kennedy Space Center

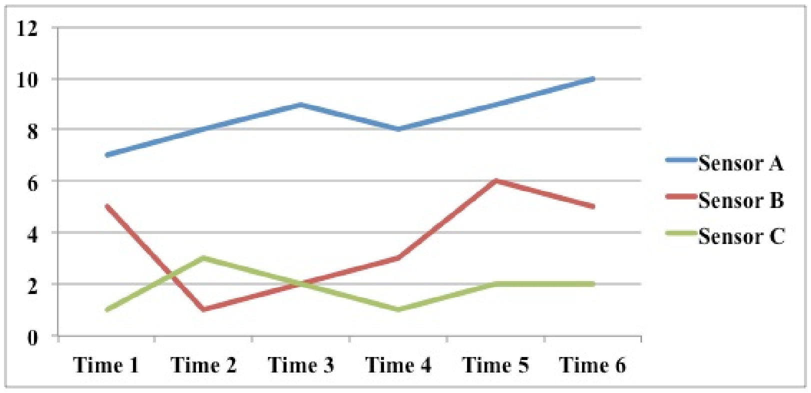

Figure 1. Mock-Up of a Plot.

Previously, when console engineers wanted to view more detailed data readouts from the hardware, they would hover the mouse over the plot and press-and-hold on the mouse button. This would cause the data to appear on the screen, similar to the illustration in Fig. 2. While still holding down the mouse button, the user could drag the pointer across the plot, and the data would continually update based on the current location of the pointer. While this worked well, over time the users found that when they needed to see the data for several seconds or wanted to freeze the data on a particular position, it would create ergonomic issues for them.

In response to this issue, the console engineers requested a new feature (later called DDP). This feature would add the ability for a user to simply double-click on the plot to have the data be displayed with persistence (i.e. it would not automatically become hidden when the mouse button was released). Then, once the user had finished looking at the data, they could single-click or double-click on the plot again to re-hide the data. However, because the users had grown accustomed to the press-and-hold method of displaying the data, that functionality needed to be maintained as well.

2. Source Code

To implement DDP, my first step was to study the existing source code to see how the existing "press-and-hold" functionality was written, so that I could maintain a style consistent with the existing system. I traced the flow of control between different methods in the code, studied the inherited classes to learn their setup for handling mouse events, and looked at how double-clicks were handled in unrelated parts of the codebase. I discovered that the mouse events that triggered a change in the visibility of the data were simply a "Mouse Pressed" event and a "Mouse Released" event. I also learned that a "double-click" could be identified by the computer as a single "Mouse Click" event but with click count of 2.

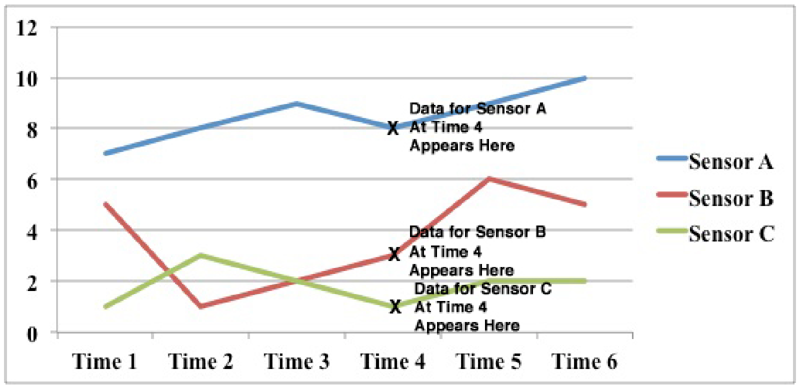

Figure 2. Mock-Up of a Plot with Data Displayed. *This is a similar style to how the detailed data appears when the mouse button is pressed-and-held or double-clicked with the new DDP feature.*

From here, I began modifying the source code. I added in the necessary conditions for when double-clicks and single-clicks occurred, following a similar format as the existing code. Using a test driver, I was able to rapidly perform functional tests of my code using plots that were filled with mock data. However, while I could readily implement the functionality of having the double-click display the detailed data and then a single-click or double-click hide the data, I was unable to do this while retaining the existing press-and-hold functionality. Based on the functional test, it almost seemed like there were multiple mouse events happening with each click. However, my existing code did not reflect these. Further testing revealed that each "double-click" was actually composed of two independent clicks:

1) a single "Mouse Click" event with a click-count of 1
2) a second single "Mouse Click" event, but this time with a click-count of 2

Upon deeper investigation, I discovered that whenever a mouse button was clicked, that single "click" was actually composed of three mouse events that happened in imperceptibly quick succession, and always in this order:

1) a "Mouse Pressed" event
2) a "Mouse Released" event
3) a single "Mouse Click" event, with a click-count of *either* 1 or 2. The click-count would be determined as follows:
    a. if a single-click only occurred, the click-count was 1
    b. if a double-click was performed, then the first click of the double-click had a click-count of 1, and the second click of the double-click had a click-count of 2.

In other words, when a double-click is performed, there are actually 6 separate mouse events occurring. With this new insight into the hidden mechanics of the programming language, I developed a system of three Boolean flags to track the users' actions. This allowed the source code to track its most-recent state and current state, so the appropriate behaviors would occur and the DDP feature would work as desired. For example, if the detailed data was hidden and then a double-click occurred, the computer needed to know that this situation required the detailed data to become visible. But if another double-click occurred after that, the computer would now need to interpret this double-click – the exact same user action as before – as the signal to now hide the detailed data.

3. Unit Tests

Like many undergraduate interns, when I began this project I had never worked with unit tests before. Accordingly, before writing the unit tests for DDP, I spent a significant amount of time researching the theory and best practices for unit testing. I learned that during their development, unit tests are important for thoroughly vetting the source code and checking edge cases. However, my research explained that unit tests' primary benefit is for the future maintainability of a codebase. For example, an engineer two years from now might make a change to the source code for the data plot feature, which in-turn could potentially break the code for DDP. With robust unit tests for the DDP in place, the engineer can simply run the DDP's unit tests and check for these code breaks in a matter of minutes. Without them, the engineer would not realize the DDP code was broken until a time-consuming and costly functional test is performed at a much later date. These time and cost savings become amplified over time as more features are added, hence the importance of thorough unit tests. Furthermore, I learned that in order to develop unit tests quickly, there are several COTS mocking frameworks available. I studied several tutorials to gain a deep understanding of the flow of control in these mocking frameworks and how they should be used to thoroughly test and verify the source code's individual methods.

With this new knowledge, I wrote the unit tests for my modified code. I learned how to set up series of mocked classes and methods for the dependencies in DDP's code, run the mocks through the actual code, and verify that the mocks had been altered by the DDP code as desired. Also, since the mouse events in DDP are often dependent on what had happened previously, my unit tests tested multiple *series* of events to ensure the correct behaviors occurred each time. Additionally, I tested for several edge cases to ensure proper functionality in all scenarios.

4. Code Review

Once the source code and unit tests were complete, I uploaded my modified code to the COTS version-tracking software and then set up a code review process via a COTS collaboration application. In the collaboration application, my technical lead and the system architect were able to review my code line-by-line. Based on their experience and familiarity with the entire codebase, they added comments, suggestions, and other modifications that would be needed before the DDP feature could be integrated into the primary codebase that would be used in the LCC. Fortunately, the notes I received primarily involved changing variable names, so they were quickly implemented.

5. Integration and Documentation

After my revised source code was uploaded to version-tracking software and the code review received all necessary approvals, DDP was sent to the system build team. The system build team is responsible for taking everyone's new source code and then building it into a complete software package that is ready for use at the LCC.

Additionally, I revised the functional test procedures for the data plot to include steps to test the new functionality DDP provides. These procedures are written in plain English so correct behaviors can be easily verified.

B. SEB

1. Objective

When a sensor or other hardware device on the rocket transmits a value that is out of the normal range, the control software receives this reading and triggers a notification to be sent to the console engineers in the LCC in the form of an alert. These alerts are referred to as events. As each event occurs, it appears on the engineers' screens in an ET, which is basically a table that lists the events that occur. In addition to the name of the hardware experiencing the anomaly, the ET contains information about the time of occurrence, the severity of the alarm, and additional details about each event.

Previously, if console engineers desired the information contained in an ET to be retained, they would need to take several screenshots, copy-and-paste individual cells of data, or possibly even write the details by hand. Needless to say, this was tedious and error prone.

With the SEB, the console engineers will now have the ability to quickly save an ET's complete dataset contained into a .csv file. Because .csv files are a generic, industry-standard file format, they can be easily read into and processed by a variety of programs and programming languages. This makes them ideal for future analysis, which is a huge benefit for the console engineers.

2. Source Code

Work on the source code for this feature had previously begun before I was tasked to complete it. The basic structure had been established; however, there were several edge cases, user experience issues, file format concerns, threading issues, and unit tests that needed to be created. Similar to my approach on DPP, I spent a bit of time tracing the flow of control and understanding how the existing code worked before making any modifications. During this time, I researched dialog boxes, how computers handle threading, and how our programming language manages the threads. I also met with the engineers who had been involved with the initial work on the SEB; they pointed out different requirements that needed to be met, explained what overall changes they wanted to see, and pointed me towards other classes in the codebase that had similar buttons to save data from the control system into separate files.

During my research phase, I noticed there were multiple classes and code snippets in the codebase similar to the ones I needed for SEB. I weighed the merits of each code snippet and incorporated the best options into the SEB's source code. For example, one class' version of saving the file simply allowed the user to type in a file name and save it to the disk. However, a second class' style was to check for edge cases such as when a user enters the name of an existing file; by entering an existing name, that original file would be overwritten and its data lost. For SEB, I chose to use a modified version of the second class' code to ensure that SEB also had the overwrite protection.

In addition to the overwrite protection, I also added other safeguards. These included making sure a proper .csv file was created, that the user's default file path configuration was accurately updated, that the user was prevented from simultaneously opening two SEB dialog boxes for the same ET, and that different processes were occurring on the correct thread. From a user standpoint, I cleaned up the .csv file output to ensure that it had minimal whitespace.

3. Unit Tests

The basic idea of the unit tests for SEB is the same as for DDP. However, because SEB relied on Graphical User Interface (GUI) dialog boxes instead of mouse events, the COTS software employed to test the code is very different. To test the GUIs, I needed to use a COTS robot-based testing framework. This type of framework creates a "software robot" that mimics a human user; just like a human, the robot can open dialog boxes, type into text fields, click on menus, and test for many other desired behaviors. After studying the robot framework's documentation, to test SEB I created and directed a robot to perform various actions, verified that it interacted with the dialog boxes correctly, and double-checked that the .csv files were correctly created and saved.

4. Code Review, Integration, and Documentation

Similar to DDP, the SEB went through the team's code review, integration, and documentation process.

C. Additional Unit Tests

In addition to creating unit tests for the methods written for DDP and SEB, I was also tasked with creating unit tests for a few methods of legacy code. While these legacy methods had reliably worked for several years, they had never been formally unit tested. Testing another developer's code is particularly challenging, because it requires not only understanding the overall purpose of each method, but also understanding how every line of code and method call interacts.

While several of these additional unit tests covered exception handlers and were primarily for code coverage, one of these methods was extremely complex and was a key method to create the data plots which the DDP feature used.

This particular method incorporated methods that were private, methods that were static, utility classes that could not be mocked nor instantiated, multiple conditionals, and nested dependencies on multiple other methods. After several days of research, testing many different COTS tools, and talking to full-time engineers, I figured out how to set up rigorous tests for this method. While this process greatly increased code coverage and helped ensure future maintainability, it also served as a key learning tool to help familiarize myself with the codebase, learn how object-oriented programming is organized at the professional level, and become a more proficient unit tester.

IV. Conclusion

As a result of this internship, the console engineers have two new features to improve their user experience on a day-to-day basis as well as when they launch rockets into space. First, the DDP feature will help with ergonomics and allow them to better focus their energy on analyzing the data being displayed. Additionally, due to my work on unit tests for the class which contained DDP and the "Additional Unit Tests" methods, I increased the code coverage for that single class as shown in Table 1:

	Code Coverage Before Updates	Code Coverage After Updates
Methods	90%	100%
Lines	84%	98%
Conditionals	53%	72%

Table 1. Code Coverage Improvements.

Second, the SEB will allow the engineers to quickly and efficiently save tables of event data to a .csv file. In the event of any anomalies, this new feature will be key in allowing for thorough analysis to deduce what occurred.

On a personal note, I feel honored to have had the opportunity to intern at Kennedy Space Center and work on a significant aerospace project that will be used in future space exploration. The camaraderie within this department is second-to-none; everyone shares the same goal of helping launch rockets into space, and everyone is happy to help each other to achieve that goal. This internship not only solidified my desire to work in the space industry long-term, but it also provided tremendous exposure to how software is developed in a professional setting and pushed my growth as a software developer in ways that cannot be quantified.

Acknowledgments

First and foremost, I would like to thank NASA for its ongoing efforts to push the boundaries of science and technology and for its amazing ability to inspire our country to explore and reach for the stars. Furthermore, to help grow the next generation of scientists and engineers for these missions, NASA has created an amazing internship program. It has been an honor to be a part of that program, and I am very excited to continue on this career path.

Second, I would like to thank the amazing people at KSC. Jamie Szafran took a chance and welcomed me into NE-XS; she has been an incredible mentor, a wealth of information and resources, and truly helped to keep things real in Florida. Jill Giles has been a fantastic co-mentor, a tremendous source of energy and support, and a wonderful sounding board for life-lessons. Because of Jamie and Jill, I've shifted my major from mathematics back to computer science; for that I will be forever grateful. Oscar Brooks, my supervisor, has gone above and beyond to make the department a success, and Gwen Gamble, Kathleen Wilcox, and Rob Cannon in the KSC Education Office who manage all of logistics of being an intern at KSC with an invisible but deft hand. Thank you all for your hard work and skillful oversight.

In addition to my primary mentors, I would like to thank my technical points of contact and mentors: Jonathan Serrano Otero, Samuel Goff, Jason Kapusta, Jordan Kiser, William Denis, Kevin Teufer, Tony Ciavarella, Richard Ludwig, and the rest of my subteam. You took me under your wings, showed me the ropes, patiently answered my questions and pointed me towards the best coding practices, and occasionally even laughed at my bad puns. I was told that this department is like a family, and you demonstrated that that is in fact true.

Last, I'd like to thank my family, friends, professors, and mentors who have supported me and inspired me in this multi-year journey of transitioning from being a television producer into my new career path in computer science and mathematics. You've been by my side through more ups-and-downs than I ever thought possible. Words will never fully convey my gratitude, so I'll embrace a bit of engineering logic and use a tried-and-true phrase: Thank You.