Octo-Docs

Software Testing Plan

Dr. James Palmer

Garrison Smith
Peter Huettl
Kristopher Moore
Brian Saganey

4/3/18

version 1

Table of Contents

1. Introduction

Octo-Docs is a team that was formed with a goal to improve how software development groups create, edit, and interact with comments in their projects. The members of team Octo-Docs are Garrison Smith, Peter Huettl, Kristopher Moore, and Brian Saganey and we are working on creating a new documentation management system called CrossDoc. The project is sponsored by Dr. James Palmer, who first proposed that commenting is in need of an improvement.

Software development teams regularly use documentation that is tightly coupled with their project's codebase. This dependence results in documentation that can not be managed independently of the code it describes and can also complicate the integration of documentation groups. The Octo-Docs team aims to address this problem with CrossDoc, a commenting system that connects external comment stores to a codebase. This separation of concerns enables distinct comment categories, external comment management, and advanced comment tooling.

To ensure that CrossDoc is operating efficiently and accurately, we will be conducting extensive software testing. Software testing is the process of systematically analyzing the functionality of a software product. This process is an important step towards creating an effective and fault-proof system. To accommodate this, we will be outlining a rigorous testing plan for three main testing categories: **Unit Testing**, **Integration Testing**, and **Usability Testing**.

Each testing category will contain a testing plan that outlines the modules and submodules of the system to be tested, the range of input parameters, and an analysis of potentially erroneous conditions. The unit and integration testing categories will specifically target the architectural modules that exist in the CrossDoc system and the interactions between them. Alternatively, the usability testing category will outline a testing strategy to be used on potential end-users.

This modular approach to a testing plan will closely mirror the modular architecture currently present in the CrossDoc system. This ensures that the results gained from individual tests will remain relevant as the product continues to develop. The modularity of our MVC architecture will allow us to prioritize user-facing tests. Firstly, we will outline our unit testing plan.

2. Unit Testing

To ensure the functionality of our system, we will be writing extensive unit tests. These unit tests will be responsible for ensuring that individual functions work as intended and return the appropriate output. We will be creating these tests using the `unittest` Python library. This unit testing framework will allow us to dynamically address the validity of our software by providing on-demand test metrics. Alongside the validity metrics, the `unittest` library also provides test execution time metrics to assess the performance of functions.

To improve reproducibility and the robustness of our tests, we will be following a rigorous testing plan that outlines exactly which functions to test, the appropriate parameter ranges to use, and the erroneous inputs that should be addressed. By explicitly detailing our plan, future test expansions or development efforts may easily expand on the work we have done.

2.1. Commands Class

The `Commands` module is responsible for implementing the core functionalities of the CrossDoc program. As such, the functions to test will directly mirror the user-facing functions. Note that the function parameters are listed adjacent to the function name in parenthesis.

`init()`

Equivalence Partitions and Test Parameters:

1. *In CrossDoc repository*
    a. `cwd = "c:/cdoc/"` -> Fatal error, repository not re-created
2. *Not in CrossDoc repository*
    a. `cwd = "c:/"` -> Repository created in "c:/"

`create_store(name, path)`

Equivalence Partitions and Test Parameters:

1. *In CrossDoc repository*
    a. `name = None, path = None` -> Default store created
    b. `name = "Test", path = None` -> Store named "Test" created at default location
    c. `name = None, path = "c:/"` -> Store with default name created in "c:/" directory
    d. `name = "Hello", path = "c:/"` -> Store named "Hello" created in "c:/" directory
    e. `name = "Remote", path = "http://test.com"` -> If valid wiki, store named "Remote" created at "test.com"
    f. `name = "Test", path = None` -> Fatal error, store already exists
2. *Not in CrossDoc repository*
    a. `name = None, path = None` -> Fatal error, store not created
    b. `name = "Example Store", path = "c:/"` -> Fatal error, store not created

`create_comment(text, store, anchor, set)`

Equivalence Partitions and Test Parameters:

1. *In CrossDoc repository*
    a. `text = None, store = None, anchor = None, set = None`
    -> Output usage message
    b. `text = "Test0", store = None, anchor = None, set = None`
    -> Create "Test0" comment with randomly generated anchor
    c. `text = None, store = "TestStore0", anchor = None, set = None`
    -> Output usage message
    d. `text = "My Text", store = "TestStore1", anchor = None, set = None`
    -> Create "My Text" comment in "TestStore"
    e. `text = "Example", store = None, anchor = 00000000, set = None`
    -> Create "Example" comment in first store with anchor 00000000
    f. `text = "Lorem Ips", store = None, anchor = None, set = "testSet1"`
    -> Create "Lorem Ips" comment in first store with set name "testSet1"
    g. `text = "Hello", store = "Store 2", anchor = 00000000, set = None`
    -> Create "Hello" comment in "Store 2" with anchor 0000000

2. *Not in CrossDoc repository*
    a. `text = “Example”, store = None, anchor = None, set = None`
    -> Fatal: not in CrossDoc Repository

`generate_anchor()`

Equivalence Partitions and Test Parameters:

1. *General partition (any calling condition)*
    a. `ga called` -> returns Randomly generated anchor

`fetch_comment(anchor, store, set)`

Equivalence Partitions and Test Parameters:

1. *In CrossDoc repository*
    a. `anchor = None, stor = None, set = None`
    -> Output usage message
    b. `anchor = invalidAnchor, store = None, set = None`
    -> Fatal error, comment anchor not found
    c. `anchor = validAnchor, store = None, set = None`
    -> Returns comment at given anchor
    d. `anchor = validAnchor, store = “TestStore”, set = None`
    -> If comment exists in TestStore, returns comment
    e. `anchor = validAnchor, store = None, set = “TestSet”`
    -> If comment exists in TestSet, returns comment
    f. `anchor = validAnchor, store = “TestStore”, set = “TestSet”`
    -> If comment exists in TestStore, and TestSet, returns comment
2. *Not in CrossDoc repository*
    a. `anchor =  validAnchor, store = none, set = none`
    -> Fatal: not in CrossDoc Repository

`delete_comment(anchor, store, set)`

Equivalence Partitions and Test Parameters:

1. *In CrossDoc repository*
    a. `anchor = None, store = None, set = None`

    -> Output usage message

    b. `anchor = invalidAnchor, store = None, set = None`

    -> Fatal error, comment anchor not found

    c. `anchor = validAnchor, store = None, set = None`

    -> Comment at validAnchor is deleted

    d. `anchor = validAnchor, store = "TestStore", set = None`

    -> If comment exists in "TestStore" it is deleted, else fatal error

    e. `anchor = validAnchor, store = "TestStore", set = "TestSet"`

    -> If comment exists in "TestSet" in "TestStore" it is deleted, else fatal error
2. *Not in CrossDoc repository*
    a. `anchor = validAnchor, store = None, set = None`

    -> Fatal: not in a CrossDoc Repository

`update_comment(anchor, text, store, set)`

Equivalence Partitions and Test Parameters:

1. *In CrossDoc repository*
    a. `anchor = None, text = None, store = None, set = None`

    -> Output usage message

    b. `anchor = invalidAnchor, text = None, store = None, set = None`

    -> Fatal: comment anchor not found

    c. `anchor = validAnchor, text = None, store = None, set = None`

    -> If comment exists in default store, comment updated to " "

    d. `anchor = validAnchor, text = "Foo", store = "sto3", set = None`

    -> If comment exists in "sto3", comment updated to "Foo"

    e. `anchor = validAnchor, text = "Word", store = "Wiki", set = "set1"`

    -> If comment exists in "wiki" and "set1", comment updated to "Word"

2. *Not in CrossDoc repository*
    a. `Anchor = validAnchor, store = None, set = None`
    -> Fatal: not in a CrossDoc Repository

2.2. Parsing Class

The `Parsing` module is responsible for reading in the user's input, processing its meaning, and delegating the input to appropriate command method.

`process_command(argv)`

Equivalence Partitions and Test Parameters:

1. *General partition (any calling condition)*
    a. `argv = ["init"]` -> Init command is run with no parameters
    b. `argv = []` -> Usage message is logged
    c. `argv = ["create-comment", "word"]` -> Create comment run with text of "word"
    d. `argv = None` -> Usage message is logged
    e. `argv = 6` -> Throw input value exception

2.3. Logging Class

The `Logging` module is responsible for providing feedback back to the user and has several different logging methods for different situations.

`standard(message)`

Equivalence Partitions and Test Parameters:

1. *General partition (any calling condition)*
    a. `message = None` -> Log blank message
    b. `message = "Example sTrinG"` -> Log "Example sTrinG"
    c. `message = ["5", 0]` -> Log string interpretation of "["5", 0]"

`usage(command)`

Equivalence Partitions and Test Parameters:

1. *Called within a command function*
    a. `command = None` -> Log usage message for current calling command
    b. `command = update_comment` -> Log usage message for `update_comment`
    c. `command = "asdf"` -> Log usage message for current calling command
2. *Called outside a command function*
    a. `command = None` -> Log general CrossDoc usage message
    b. `command = create_comment` -> Log usage message for `create_comment`
    c. `command = -8` -> Log general CrossDoc usage message

`program(message)`

Equivalence Partitions and Test Parameters:

1. *General partition (any calling condition)*
    a. `message = None` -> Log generic message with program name prefix
    b. `message = "Test Case"` -> Log "Test Case" with program name prefix
    c. `message = 0` -> Log "0" with program name prefix

`fatal(message)`

Equivalence Partitions and Test Parameters:

1. *General partition (any calling condition)*
    a. `message = None` -> Log generic fatal message and stop execution
    b. `message = "Example"` -> Log "Example" fatal message and stop execution
    c. `message = -6` -> Log "-6" fatal message and stop execution

3. Integration Testing

As we do the integration testing within our project we are going to be using the `unittest` Python library to do all the testing for integration testing too which was stated above as well. This will allow us to have on-demand testing metrics for the integration testing as well. These metrics will make sure that Crossdoc is working effectively and efficiently.

We will be structuring our integration tests with a *top-down* method. This means that we will validate the functionality of module integration starting from the user-facing elements. As such, we will start with the text editor plugins and follow the execution flow to the `Logging` class.

3.1. Text Editor Plugins to Command Line Program

When it comes to the text editor plugins working with the command line interface. The idea is that the text editor plugins will be able to use all the commands that the `Command` class has such as `create_comment`, `delete_comment`, and `update_comment`. The plugins are going to be used across four text editors: *Sublime*, *Vim*, *Atom*, and *Emacs*. Each plugin will have their own interface for the text editor but it will call the commands in a very similar way from the `Command` class. The idea is to test this the same way the command line program would test itself for the users above. However, in this case, the goal is to test the functions from the `Command` class and make sure that they are producing the correct output from the text editor plugins.

3.1.1. Sublime

The first plugin that we started working on was Sublime because it was Python and CrossDoc was also written in Python. To test, we will ensure that calling any command from the command palette calls the appropriate command from the `Command` class. In Sublime, the command palette is the plugin interface that stores all plugin commands that have been registered with Sublime. Ensuring the connection between this command palette and our `Command` class is vital.

3.1.2. Atom

In Atom, the test will be similar to Sublime due to the similarities in the editor interfaces. In Atom, however, this was in javascript for plugin interfaces so the goal to make sure that the commands from the `Command` class can translate properly from Python to JavaScript and grab the correct command that the command line interface was given. Atom also has a plugin menu drop down as well and just like Sublime we want to make sure that we are testing to make sure that this drop down menu is working properly for the users and giving the appropriate command from the `Command` class.

3.1.3. Vim

In Vim, the editor interface is a command line interface rather than a graphical user interface. Vim also has their own language called vimscript for plugin extensions and because of this, we wanted to make sure that the `Command` class commands could still be called from there. The test is to make sure that when you call a command from Vim that again the appropriate command is given from the output. Vim does not have a drop-down menu for their interface so we just have to test the commands from the `Command` class through Vim.

3.1.4. Emacs

Emacs functions similarly to Vim, and also has its own language called Elisp. Although Emacs plugins are written in a functional language, the end product will still function similarly to all other plugins. As such, we will still test the commands with respect to the `Command` class. Just like Vim, Emacs has a command line interface to call plugin commands so there is no drop-down menu to call extensions so we just have to make sure Emacs can reach the correct command from the `Command` class.

3.2. Parsing to Command Class

The `Parsing` class is responsible for making sure that the user's input is the correct input given when they are calling commands through the command line. The `Parsing` class allows for easy to use interface for the user as well as making sure that the correct commands are given to the parser when a user is using the command line.

The idea for testing the parser is to make sure that when the parser is run, it calls the appropriate commands that were given by the user. The parser's job is to translate the user input and make sure that the command that gets called is the proper command. When testing this we will check to make sure that if the parser calls a specific command from the `Command` class, that the output from the parser is the appropriate command that is supposed to be called from the `Command` class. The parser will also check to see if when an invalid command is given that it runs a command that exits the program and gives the user the correct feedback from the command line.

3.3. Command to Logging Class

The purpose of the `Command` class it to make sure that all the commands that are called by the user return the appropriate output. In addition, if `create_comment` was run from the `Command` class then the `Logging` class has methods to indicate whether or not the run was successful. This will determine whether the logger outputs a fatal error and kills the program.

When a command gets inputted by the user the `Logging` class will give the output to the user in the form of either a standard message, the output of the command itself, or a fatal. When testing the `Command` class to the `Logging` class, we will run commands from the `Command` class and make sure that the `Logging` class can output the commands appropriately. This means that when we test for a standard message, we will run the `create_comment` command and check the printed return value. If the input parameters were valid and the command still failed, we will know there was an error in integrating the classes. For instance, `create_comment` should run the appropriate command and make a comment above the cursor in the current file.

4. Usability Testing

In order to decide how the usability for CrossDoc should be tested, we had to take into consideration who would be using the CrossDoc application. There are two types of users who will be using CrossDoc. The first users for CrossDoc are software developers who are commenting within the code. The second users are technical writers who will be using CrossDoc centralizing commit location to edit and translate commits. The usability testing for CrossDoc will include user feedback, and in-lab user testing from two user groups we have easy access to. The two groups are:

- *Software Developers:* They fit right into our user group for the CrossDoc application, as they are often the ones commenting code in software projects.
- *Technical Writer:* They are the users that will be using Wiki for editing and translating comment outside of the code.

These two user groups will provide necessary feedback that will inform as to whether, or not CrossDoc usability works for the end users.

4.1. Software Developers

The target audience of CrossDoc is software developers, and for this reason, we plan to spend the majority of our usability testing with software developers, to make sure the process of creating a CrossDoc repository for a project and commenting code work efficiently for software developers. The software developers should find it easy to create, push, and pull comments into the repository. They should be able to comment the code normally or edit comments in a centralized location outside of the code. We will test this usability through expert reviews with software developers. By walking them through the CrossDoc application and then giving them a script to follow. While they are working their way through the script, we will be taking notes on where their understanding breaks down so we can fix major issues in the application.

4.2. Technical Writers

CrossDoc usability for technical writers is very simple for editing and translating comments as long as they do not work within the software code. They should be able to edit and translate comments from the repository Wiki. We will test this for now by using students from Northern Arizona University to test the usability of CrossDoc by giving the students a script without a walkthrough to see if the navigation of CrossDoc from the Wiki location is to follow and use for users. We will also be taking notes of the students interacting with CrossDoc, and any feedback they might provide.