rsync Configuration Utility

Milestone 3

Rose-Hulman Institute of Technology - CSSE 376

Eric Henderson
Tom Most
Kevin Risden

1 Metrics

Test Coverage

Definition: This is defined to make sure that every line of code written is covered by at least one test case. This does not include each execution path, but makes sure that every executable line of code is run at least once.

Measurement: This is measured by the Python coverage package which automates the checking of each executable line of code and makes sure that it is run at least once by our test cases. It makes sure that every possible executed line is executed at least once.

Analysis: This will be used in analysis to show that our test cases at bare minimum run each executable line of code at least once. By tracking this metric over time, we will be able to show how we add new test cases to cover more of our code base.

Why: By showing this, it gives us the credibility that we have test cases that cover at least the bare minimum of our code base. We will also gain the insight to know where to focus more intensive testing if some of the code base is not being tested at all.

Test Progress

Definition: Test progress is defined as the number of test cases expected to be finished, the number of test cases finished tracked over the course of the project, and the number of test cases that have been coded for and passed.

Measurement: The measurement of this metric will be mostly manual as we will set a goal for the expected number of test cases and then once the week is over determine how many test cases were written by subtracting the current total from last week's total test cases. The addition of test cases passed will be determined in the same way but use the test cases passed instead of total test cases.

Analysis: The analysis of this data will include showing a graphic similar to an s curve showing when and how many test cases were written. This will help us learn about our own test case writing and see when in the process that we actually wrote the test cases.

Why: This metric will help us determine our own test case writing characteristics and see if our development follows the s-curve as shown in class. We will also be able to determine how well we are able to guesstimate how many test cases will be written for each week.

SLOC

Definition: The source lines of code metric will be the actual number of source code lines that do not include blank lines or comment lines. Line lengths are a maximum of 79 characters and since Python is white space delimited there will be no multiple statements on a single line.

Measurement: The measurement of this metric will be done by a package called CLOC that counts the code, blank, and comment lines of code and outputs it in a nice format. We will be tracking not only for the entire project but for individual files and methods as well.

Analysis: The collection of this will enable us to break down the lines of code by file and then look at each class individually. We can view how the classes and files expand lines of code over time and see if certain files are larger and whether they should be split into multiple classes.

Why: Having a metric that looks directly at the code being written will help us focus on the code we are writing. If a not complicated class takes up a large number of lines there may be a more efficient way of doing this. Furthermore, looking at lines of code will let us learn about our own tendencies in regards to how much code is written a week.

Comments Lines/Method

Definition: This is defined as the total comment lines divided by the total number of methods. The length of a comment line should be no more than 79 characters in length so there will not be one long line of a single comment.

Measurement: The measurement of this metric will require taking the total comment lines from the SLOC evaluation and then divide it by the total number of methods.

Analysis: The number of comment lines tracked over time should show us how well we document our code so that others can read it later. We think this should increase over time as the final details are fleshed out and more documentation is added to the code.

Why: Since it has been suggested to us that we add comments to our code to help the readability, it would be helpful for us to track how many comment lines occur throughout each method in the code base. We will learn about our tendencies to write comments in the methods and what we need to improve on.

Cyclomatic Complexity

Definition: Cyclomatic Complexity is the measure of linearly independent paths through a program.

Measurement: The measurement of this will be automated using tools from http://www.traceback.org/2008/03/31/measuring-cyclomatic-complexity-of-python-code/ where the script will determine the complexity of each method in each file of the code base.

Analysis Usage: By looking at the Cyclomatic Complexity of each method then we will be able to show that some methods are more complex than others are and may need to be split. By examining this over time, we should see that the overall Cyclomatic Complexity levels out over the different methods and that large Cyclomatic Complexity methods are split into smaller ones.

Why: Looking at this metric will help us write code that is well balanced between methods and that one method does not do all the work. We should be able to learn how to refactor code that is too complex and show our habits over time.