Smart-Irrigation Backend
CMPS 115
Fall 2015

Introduction

In order to facilitate proper upkeep of coding, a regimented system of regression and integration tests were fabricated. Early stages of the project required manual regression testing and integration testing of various prototype components that were done on a somewhat infrequent basis due to the constraints of a proof-of-concept environment. As time progressed, manual testing yielded to automated testing using curl command line. Using an automated test harness, the integration of new test units became trivial and allowed for continuous testing of checked in code on a frequent basis.

Early Testing Methodology

Early testing was often done using test cases encapsulated in the code being tested. A developer would accept a user story, implement this user story with a predefined case of testing, and then integrate it into the codebase without the test case. In this manner the environment often mirror test driven development, but with the significant flaw that no set schedule for testing was adhered to, and no test units were kept after completion of a user story. Toward the end of the developmental life cycle, these flaws were mitigated through the fabrication of an automated testing harness in python.

Later Testing Methodology

Late into the development process, the flaws of manual regression testing became apparent through time consumption. In order to solve this problem, an automated test framework was created. This test framework would accept an aggregate of modular test units and run them in sequence. Each test would be executed, have it's results collected and compared to standard expected results. Curl was utilized to perform api calls from system shell. Each test cycle would be logged into a master testing log. Failing test units would have their results date-stamped and stored for manual examination. This reduced the overhead of manually testing any api functions to a mere minute.

The Test Harness

A test harness was created to facilitate ease of testing by providing a framework which accepts modular test units sharing similar properties. Each test unit consisted of some number of shell commands that would be piped into the shell using Python's subprocess.check_output command.

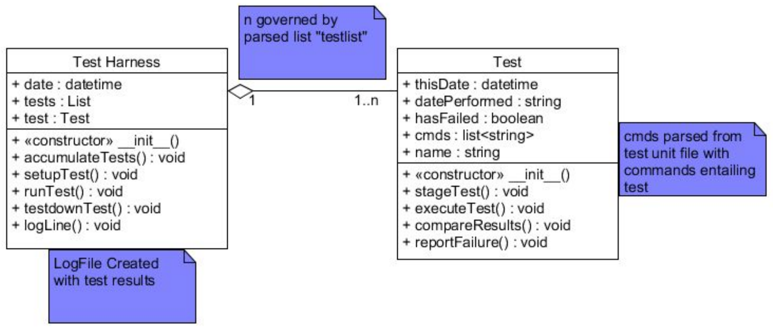

Figure 1, Test Harness UML

The test harness operates by parsing a file named testlist in the QA directory. This testlist contains all the test units the engineer wishes to perform by name, one per line. The harness stores each test in the list 'tests'. These tests are then popped off the structure in the order read (FIFO) and constructs the test object using stored files in the QA directory. The test object will parse a list of commands from its file in the QA/Tests directory. These commands are then executed line by line. If any error/exceptions occur, the test is immediately flagged failure and output is displayed to the engineer. After all commands have been fired the output is collected in stored in a temporary file testResults. This file is then checked for string equality to the stored test result in QA/KnownResults. If any differences are encountered, the test if marked for comparison failure and it's output is stored in QA/FailingTests. All results are logged in a time stamped log file located in the core QA directory. This harness can even accept subprocess calls for supplementary python scripts for more in depth testing. It can also be set up to fire automatically using Cron.

Conclusion and Plan

Manual testing is useful in short term early developmental environments, but the automated testing harness made this process easier for all parties involved. It provided a framework for adding new tests with ease and identifying failures before the end user receives the product. With every new functionality added to the product, a new test (or set of tests) should be delivered alongside that will ensure its validity. This test should then be integrated into the test battery and executed with the harness on a set daily schedule.