Automated testing framework in Jython Music

Paige Peck, Curtis Motes, Ryan Lile
Peckpd@g.cofc.edu, Motescm@g.cofc.edu, Lilerw@g.cofc.edu

Introduction

In large software engineering projects, it becomes difficult to ensure that any new component or edit does not interfere with the pre-existing code. Leaving the system unchecked can result in bugs and situational errors that would otherwise go unnoticed. In this project, we created an automated testing framework for Jython Music an extension of jMusic. As code is added or classes are changed a quick run of the testing framework will show a results page ensuring you of your intended results or explaining in detail where the code broke. The framework is easily expandable to add new tests as the project grows.

Framework and File System

Test Automation/
- scripts/
  - runAllTests.sh
- testCases/
  - testCase01.txt
  - testCase02.txt
  - etc.
- temp/
  - temporary files used to store results to be added to the results html file
- reports/
  - testReport.html
- project/
  - Jython
    - Jython Classes
    - Library
      - Jython Classes
      - test case executables

The test framework runs by executing runalltest.sh from the scripts directory. It goes into
the testCases and reads each test description, finds the applicable test case executable, runs the test, compares the output to the described expected output, and stores the result in a table in a html file.

Deliverables

Deliverable 1 - Project Selection and Installation

The team decided on Jython Music as the H/FOSS project to be used. We downloaded the software from the Jython Music and installed it on our Linux (ubuntu) virtual machine.

Deliverable 2 - Test Plan

The team came up with a test plan for the rest of the semester.

- September 27th -> Create Test Plan and specify 5 tests
- October 7th -> specify 15 test cases
- October 14th -> specify all 25 test cases
- October 18th -> Design and build Testing framework
- October 28th -> Have 10 tests implemented within framework
- November 4th -> Have 20 tests implemented within framework
- November 10th -> Have all 25 test cases running through the Testing framework

The team also decided on the test Case description format to be used by the framework. This format is required for all new test cases to be added to the framework.

1. test number or ID
2. requirement being tested
3. component being tested
4. method being tested
5. test input(s) including command-line argument(s)
6. expected outcome(s)

The team planned the framework. The test automation will run by running a single script. That script will look at all of the test case descriptions. It will look for the file name of the test (line 1 of the description) in the Jython Library. Then the script will run the test and compare the output to the expected output (line 6 of the test description) and store the result in a section of html in a temporary file including information about the test. Once all of the tests are finished running the script will compile all of the temporary files together into a new html file containing a table of the tests that have been run, their details, and the result.

Deliverable 3 - The framework

The team finished planning the framework and implemented it. First, small versions of the total framework were created then put together to form the finished product. The runallscripts.sh was written and tested on fake tests. Then the team wrote the first 5 test cases and executable and tested the framework on them.

Deliverable 4 - The Completed Test Cases

The remaining 20 test cases were planned and written. 5 classes were tested across the 25 test cases. The framework was tested and debugged and the resulting html report was debugged as well.

Test	Requirement	Component	Method	Input	Expected Output	Actual Output	Result
test1.py	returns true when instance is running	Timer2	Timer2.start		True	True	Pass
test2.py	checks that if the flag is true, repeat is also true	Timer2	Timer2.setRepeat(flag)		True	True	Pass
test3.py	timer task starts on command	Timer2	Time2.start()		True	True	Pass

Deliverable 5 - Fault Injection

5 faults were added to the code of Jython music. Errors such as “>” being changed to “<” or “==” being changed to “!=” . The framework was run to see that the tests that were supposed to failed due to the change in code did indeed fail. This was a forced example of what would happen if code was actually modified and accidentally broke the system. The test automation framework would show all of the induced errors and trace them to show exactly where they were broken.

Test	Requirement	Component	Method	Input	Expected Output	Actual Output	Result
test1.py	returns true when instance is running	Timer2	Timer2.start		True	False	Fail
test2.py	checks that if the flag is true, repeat is also true	Timer2	Timer2.setRepeat(flag)		True	Audio: AudioDevice: PulseAudio Mixer, max in = 6, max out = 6 Audio: AudioDevice: default [default], max in = 32, max out = 32 Audio: AudioDevice: I82801AAICH [plughw:0,0], max in = 2, max out = 0 Audio: AudioDevice: I82801AAICH [plughw:0,1], max in = 2, max out = 0 Audio: AudioDevice: Port I82801AAICH [hw:0], max in = 0, max out = 0 ---- Pure Java JSyn www.softsynth.com - rate = 44100, RT, V16.5.14 (build 448, 2012-12-10) Traceback (most recent call last): File "/library/test2.py", line 9, in t.setRepeat(flag) File "/home/paige/TODO-Team-Name/TestAutomation/project /Jython/library/timer.py", line 283, in setRepeat self._repeat = !flag # Fault Injection 3 NameError: global name 'lag' is not defined Output buffer size = 7056 bytes. Output Latency = 40.0 msec	Fail
test3.py	timer task starts on command	Timer2	Time2.start()		True	True	Pass

Experiences

At first it took the team a while to narrow down the H/FOSS project we wanted to build a test automation framework for. The first project was far too complicated so we switched to Jython Music which was more straightforward.

Many of the issues that arose during the project were from inexperience with bash and python. Jython is specifically complicated in the fact that all python code is compiled into Java bytecode and then executed. Once we broke through on the first few issues the rest became much easier as we grew in our familiarity with the project and python.

The planning for the framework was interesting as the three of us each had different approaches. It was interesting to take the different ideas and combine them into a final product. Each member of the team was crucial for completing different aspects of the project and had their own “area of expertise”.

This project was the first time the members had worked in a group on a project start to finish and had to learn good practices in group work. The individual schedules were not conducive to meeting often so many of the assignments were completed in one night apiece. The group learned a lot about the importance of planning and getting down to work when only a small period of time to work was available. Overall the experience of working in a team was good.

Further information

Checkout the full project at:
https://github.com/CSCI-362-01-2016/TODO-Team-Name

Checkout Jython Music at:
https://jythonmusic.me