TODO: Team Name

JYTHONMUSIC TESTING FRAMEWORK

Curtis Motes, Paige Peck, Ryan Lile | CSCI 362 | December 1st, 2016

Table of Contents

Introduction 2
Introduction: Project Candidates 3
Chapter 1 ..... **4**
Deliverable 1: Part 1 4
Deliverable 1: Part 2 5
Chapter 2 7
Deliverable 2 7
Five Test specifications 8
Chapter 3 ..... **10**
Deliverable 3 10
Chapter 4 12
Deliverable 4 12
All tests run 13
Chapter 5 ..... **15**
Deliverable 5 15
Fault template and fault injections 15
Fault injections running 17
Chapter 6 19
Overall Experience 19
Project Feedback 20

Introduction

Our team consist of two computer science majors and a computing in the arts major. We come from varying backgrounds with vast experiences. Every team member could contribute their skills to accomplish the overarching goals of the project. The project required the use of Git which Paige had experience with from his internship, Bash which Curtis was our expert on this from his experience with Linux systems, and project planning which Ryan was the lead on this through his experience with a research lab.

In large software engineering projects, it becomes difficult to ensure that any new component does not interfere 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 JythonMusic 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.

Introduction: Project Candidates

Our original project candidates consisted of three projects: OpenMRS, STEM, and SugarLabs. OpenMRS is a Java based medical records platform that is open source and community developed. STEM is Spatiotemporal Epidemiological Modeler which is also Java based and is used to model the spread of diseases using spatial and temporal models. SugarLabs is a Python based operating system that is used for children. This was our original choice while testing the activity "Memorize 45". As shown in more detail in Deliverable 1 Part 1, we ended up scrapping this due to the size and dependencies required for SugarLabs. Once we concluded that SugarLabs wasn't going to work and the other two candidates were not feasible, we chose JythonMusic as our project to create a testing framework. We chose JythonMusic because it is written in Python and the documentation online is well written due to it being created by the College of Charleston professor Dr. Manaris.

Chapter 1

Deliverable 1: Part 1

For SugarLabs, there was a good amount of documentation on how to install, build, and develop for the project. The community is still active as well which will help immensely throughout this projects lifetime. However, there was many issues during installation and building of SugarLabs.

Installing SugarLabs on to two of our computers was a challenge. On both occasions, VirtualBox crashed and in one instance, Ubuntu needed to be reimaged. After this and several restarts, we could install SugarLabs on one of the computers, while the other seemed to still be hung up on install. From there, we worked on installing SugarLabs through the Git terminal. It seemed to be going fine until we ran into several more hang-ups. During the build, there was a "command failure" for building "browse". About an hour of searching lead us to other people having this issue, and there were some fixes that worked for some. We attempted these fixes, but they did not work for us. After failing the build of SugarLabs and requiring help on this project, we looked at the source code of the project "Memorize" and saw that there was a demo set up. It creates an instance of the game, but because it is in Python, it does not have any JUnit tests included.

We were unable to successfully build SugarLabs and will need further assistance and troubleshooting to continue. If possible, and if we continue to run into issues with SugarLabs, it was discussed to change projects. Once we get SugarLabs up and running, it could be quick on getting the needed testing done for "Memorize".

Deliverable 1: Part 2

After the numerous issues, we experienced with SugarLabs, we decided to switch our project to JythonMusic. JythonMusic is "an open source environment for music making and creative programming activities." Written in Python, JythonMusic is an environment intended for programmers and musicians that allows for the use of MIDI files to be played with simple GUIs and simplistic code to allow for the manipulation of these music files. The website has good documentation such as the API, and a full tutorial on how to use the environment.

Installing JythonMusic was just a simple download of the program from the website and unzipping the folder. Some example files and JEM Jython Environment for Music, were included with this, which is a user interface for running the files. Building and running JythonMusic wasn't too difficult as the website gives a good explanation of how to run individual files. Running the files from the JEM worked just fine with no hang-ups, however running the files from the Terminal following the directions of the website gave some errors on occasion.

There were several files in the library folder that had test cases written at the bottom of the source code. Below is a brief explanation of the test cases written:

- Gui.py – The tests created several displays that tested creating buttons, sounds associated to the buttons, and simulating mouse clicks on the buttons. There are also simple animations that are created for the testing as well.
- Guicontrols.py – The unit test for this file was testing a vertical and horizontal slider and toggle switches.

- Midi.py – This unit test creates a connection between an input and output MIDI device to test the I/O functionality.
- Osc.py – This unit test is used for an OSC input object, according to Wikipedia an OSC is “is a protocol for networking sound synthesizers, computers, and other multimedia devices for purposes such as musical performance or show control.” The test connects to a localhost on the host machine to send messages such as “helloWorld”.
- Timer.py – This unit test creates a timer object and counts the seconds passed. The other files included in the library did not have a specific section for unit testing and if there were further tests, they were in the middle of the code.

JythonMusic is much easier than our original choice of project, SugarLabs, but it has its own challenges. There were some issues with compiling and building, but for the most part we could get it up and running.

Chapter 2

Deliverable 2

For the testing process, we planned on individually testing different methods by hand from the terminal. Then we planned to build a testing framework and test both individual methods as well as systems of methods. We began by testing methods from the timer.py class. The methods included are setRepeat, isRunning, start, stop, and getDelay which are all from the Timer2 class.

At the planning process, we had the previously mentioned test methods specified, and we created a testing schedule that went as follows:

- 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 built the testing framework
- October 28th: Have 10 tests implemented within the framework
- November 4th: Have 20 tests implemented within the framework
- November 10th: Have all 25 test cases running through the testing framework

For the test recording procedures, we planned to have a file describing what method/requirement is being tested along with the inputs and expected outputs. We also planned to have a test report created from each test with the expected output, input, and test description. The tests required JythonMusic software, a Jython compiler, and the executable jMusic program.

The 5 tests previously mentioned were specified by the time of this deliverable and went as follows, with the “Test X” being the template for the tests we would be writing:

- **Test X:**

    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)

- **Test1:**

    1. testIsRunning
    2. This test will run the isRunning method, which will check if a timer object has been started.
    3. Timer2.start
    4. Timer2.isRunning
    5. test input(s) including command-line argument(s)
    6. false

- **Test 2:**

    1. testSetRepeatsetFlagTrue
    2. This test will check that if the flag is true that repeat is set to true also
    3. Timer2.repeat
    4. Timer2.setRepeat()
    5. test input(s) including command-line argument(s)
    6. true

- **Test 3:**

    1. testStartWithoutRepeat
    2. A timer task starts on command
    3. running
    4. Timer2.start()
    5. n/a
    6. Timer2.running = true

- **Test 4:**

    1. testStop
    2. A timer object stops after being created and started.
    3. Timer2
    4. Timer2.stop()
    5. n/a
    6. Timer2.running = false

- **Test 5:**

    1. testGetDelay
    2. Ability to use delay time interval
    3. Timer2.timeInterval
    4. Timer2.getDelay
    5. setDelay(time)
    6. time

The largest constraint we ran into for this deliverable was time. We are composed of three full time students with families and several jobs. We knew at this point that it would be difficult to find time to meet up, but we managed to find several times where we were all free to work on the project together when necessary.

Chapter 3
Deliverable 3

Deliverable 3 consisted of creating the shell of the testing framework. This deliverable was far more technical than its predecessor. Our testing framework revolves around runAllTests.sh. The runAllTests script will create a table in an html file in the reports directory within the TestAutomation directory containing each test and the result of the comparison of the test output to its corresponding oracle. Each testCase.txt file will contain information about what is being tested and inputs to be used in the testCaseExecutable files. Each test case will have its own executable, but this may be reworked for efficiency down the line. Each test case will also have its own oracle, which can be found in the /oracles directory.

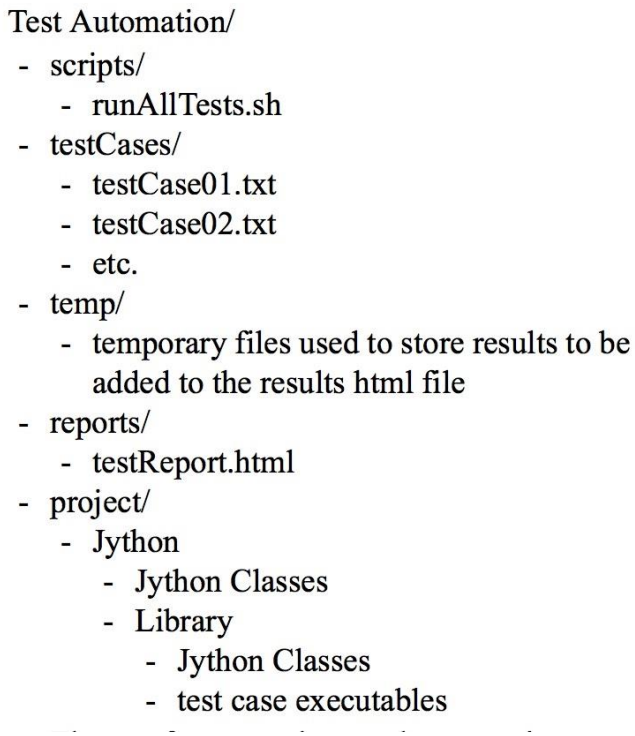

Figure 1) The Testing Framework

This deliverable was certainly the most technical one of the project, and our team performed well for having limited knowledge/experience in creating shell scripts. The biggest challenge was coordinating time to meet up as a full team because of busy school/work/family schedules. However, these times were crucial to the organization of the project. Continuing forward, we will implement the rest of our test cases following the schedule set in Deliverable 2.

Chapter 4
Deliverable 4

For our testing framework, we created a driver for each test which runs the methods that are being tested. We kept the format of our test cases, but had to put the test files in the library of the project for them to run properly due to how JythonMusic runs. We have our test text files that is called by our script and has the test file's name to run.

The template for our test cases is the following:

- Test executable
- Requirement that is being tested
- Component
- Method
- Input
- Expected Output

Figure 2) Test framework running

The test executable begins with the word test and ends with a two-digit number indicating what order the test should run, i.e. test01

- The requirement is an English description of what the function will be doing
- The component is the class that the method will run from
- The method is the function that will be tested
- The input is the parameters that will be passed in to the function
- The expected output is what the actual output will be tested against

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
test4.py	time object stops after being created and started	Timer2	timer2.stop()		False	False	Pass
test5.py	ability to use delay time ionterval	Timer2	Tiemr2.timeInterval()	1000	1000	1000	Pass
test6.py	returns note that was added to phrase	Phrase	Phrase.addNote(pitch, rhythmValue)	1	[1]	[1]	Pass
test7.py	returns 2 notes that were added to phrase	Phrase	Phrase.addNote(pitch, rhythmValue)	1 60	[1, 60]	[1, 60]	Pass
test8.py	returns note that was added to phrase	Phrase	Phrase.addNote(pitch, rhythmValue)	127	[127]	[127]	Pass
test9.py	returns size of phrase after note fails to be added to phrase	Phrase	Phrase.addNote(pitch, rhythmValue)	150	0	1	Fail
test10.py	returns size of phrase after note is added	Phrase	Phrase.addNote(note)	10 20 30 40	4	4	Pass
test11.py	returns size of phrase after emptying note list	Phrase	Phrase.empty()		0	0	Pass
test12.py	returns true if the copied phrase matches original	Phrase	Phrase.copy()		True	True	Pass
test13.py	returns true if the endTime of the note list matches expected duration	Phrase	Phrase.addNote()		True	True	Pass
test14.py	returns true if the duration of the phrase is 10 after adding 4 notes totaling 10 duration	Phrase	Phrase.getEndTime()		True	True	Pass
test15.py	returns true if two phrases match, using a list for one phrase, and individually adding notes to another	Phrase	Phrase.addNoteList(pitchList, rhythmList)		True	True	Pass
test16.py	returns note list of chord added	Phrase	Phrase.addChord(pitchList, rhythm)	1 2 3	[1, 2, 3]	[1, 2, 3]	Pass
test17.py	returns pitch of note (entered as pitch constant) as MIDI integer 0-127	Note	Note.getPitch()	0	0	0	Pass
test18.py	returns pitch of note (entered as pitch constant) as MIDI integer 0-127	Note	Note.getPitch()	1	1	1	Pass
test19.py	returns pitch of note (entered as pitch constant) as MIDI integer 0-127	Note	Note.getPitch()	127	127	127	Pass
test20.py	returns pitch of note (entered as pitch constant) as MIDI integer 0-127	Note	Note.getPitch()	64	64	64	Pass
test21.py	returns the inputted note phrase repeated 2 times	Mod	Mod.repeat()	10 20	[10, 20, 10, 20]	[10, 20, 10, 20]	Pass
test22.py	returns a reverse-order note phrase appended to original phrase	Mod	Mod.palindrome(phrase)	127	[127, 127]	[127, 127]	Pass
test23.py	returns a reverse-order note phrase appended to original phrase	Mod	Mod.palindrome(phrase)		[]	[]	Pass
test24.py	returns a reverse-order not phrase appended tooriginal phrase	Mod	Mod.palindrome(phrase)	40 60	[40, 60, 60, 40]	[40, 60, 60, 40]	Pass
test25.py	returns a reverse-order note phrase appended to original phrase	Mod	Mod.palindrome(phrase)	40 60 80	[40, 60, 80, 80, 60, 40]	[40, 60, 80, 80, 60, 40]	Pass

Figure 3) All tests running

For this deliverable, our main goal was to complete the testing framework and get all 25 test cases written, as well as fixing some of our formatting from deliverable 3. We managed to conquer the most difficult task of completing the script to run all tests with a few hacks to allow for output to be read properly since JythonMusic outputs extra text that is not needed for our testing purposes. We ran into a few hang ups with the coding, specifically the output of notes using ‘<’ and ‘>’ symbols, but the html report would read

those as html tags. We managed to find a work around for this issue and completed all the tasks necessary for this deliverable.

Chapter 5
Deliverable 5

For the final deliverable, we were to create five fault injections into the code to cause some of our tests to fail. We have the lines commented out that causes the errors with a comment saying which fault injection it is.

```
Location of file
The line number that was changed, the class, and the method
Original line
Fault injected line
Tests that will fail
```

Figure 4) Fault template

The faults we injected were as follows:

Fault 1)

- TODO-Team_Name/TestAutomation/project/Jython/library/timer.py
- Line 299, Timer2 class, setDelay method
- if self._timeInterval == timeInterval:
- if self._timeInterval < timeInterval:
- Test 5 fails

Fault 2)

- TODO-Team_Name/TestAutomation/project/Jython/library/timer.py
- Line 320, Timer2 class, start method
- if self._running == True
- if self._running == False
- Test 1 fails

Fault 3)

- TODO-Team_Name/TestAutomation/project/Jython/library/timer.py
- Line 284, Timer2 class, setRepeat method
- self._repeat = flag
- self._repeat = !flag
- Test 2 fails

Fault 4)

- TODO-Team_Name/TestAutomation/project/Jython/library/music.py
- Line 1026, Note class, constructor
- if type(value) == int and value != REST and (value < 0 or value > 127):
- if type(value) == int and value != REST and (value > 0 or value > 127):
- Test 9-16,18-22, 24, and 25 fails

Fault 5)

- TODO-Team_Name/TestAutomation/project/Jython/library/music.py
- Line 1083, Phrase class, addChord method
- n = Note(pitches[i], 0.0, dynamic, panoramic, length)
- n = Note(pitches[i]+1, 0.0, dynamic, panoramic, length)
- Test 16 fails

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
test4.py	time object stops after being created and started	Timer2	timer2.stop()		False	False	Pass
test5.py	ability to use delay time on interval	Timer2	Tiemr2.timeInterval()	1000	1000	0	Fail
test6.py	returns note that was added to phrase	Phrase	Phrase.addNote(pitch, rhythmValue)	1	[1]	[1]	Pass
test7.py	returns 2 notes that were added to phrase	Phrase	Phrase.addNote(pitch, rhythmValue)	1 60	[1, 60]	[1, 60]	Pass
test8.py	returns note that was added to phrase	Phrase	Phrase.addNote(pitch, rhythmValue)	127	[127]	[127]	Pass
test9.py	returns size of phrase after note fails to be added to phrase	Phrase	Phrase.addNote(pitch, rhythmValue)	150	0	1	Fail
test10.py	returns size of phrase after note is added	Phrase	Phrase.addNote(note)	10 20 30 40	4	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/test10.py", line 6, in note1 = Note(int(inary[0]),1) File "/home/paige/TODO-Team-Name/TestAutomation/project/jython/library/music.py", line 1027, in __init__ raise TypeError("Note pitch should be an integer between 0 and 127 (it was " + str(value) + ".") TypeError: Note pitch should be an integer between 0 and 127 (it was 10).Output buffer size = 7056 bytes. Output Latency = 40.0 msec	Fail
test11.py	returns size of phrase after emptying note list	Phrase	Phrase.empty()		0	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) Output buffer size = 7056 bytes. Output Latency = 40.0Traceback (most recent call last): File "library/test11.py", line 7, in note1 = Note(10,1) File "/home/paige/TODO-Team-Name/TestAutomation/project/jython/library/music.py", line 1027, in __init__ raise TypeError("Note pitch should be an integer between 0 and 127 (it was " + str(value) + ".") TypeError: Note pitch should be an integer between 0 and 127 (it was 10). msec	Fail
test12.py	returns true if the copied phrase matches original	Phrase	Phrase.copy()		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) Output buffer size = 7056 bytes. Output Latency = Traceback (most recent call last): File "library/test12.py", line 7, in note1 = Note(10,1) File "/home/paige/TODO-Team-Name/TestAutomation/project/jython/library/music.py", line 1027, in __init__ raise TypeError("Note pitch should be an integer between 0 and 127 (it was " + str(value) + ".") 40.0 msecTypeError: Note pitch should be an integer between 0 and 127 (it was 10).	Fail

Figure 5) Fault injections running

We managed to meet up the day after Deliverable 4 was presented and write all our fault injections in this one meet up. The first thing that we accomplished was cleaning up the formatting and errors that we had from Deliverable 4. Most of these were spelling errors and output formatting. We also made sure that we had a proper README that fully explained how to run our testing framework and how to add tests to it. Once we completed this task, we began work on our five fault injections. This didn't take too long as we already had an idea on where we wanted to inject these faults. We decided we wanted to make a fault injection into the timer file since we had several tests that could be checked with these faults. The other class we injected faults into was the music file as a lot of our tests ran the Note class and injecting a fault here could help show us that most of our tests would fail. After running the testing framework, we saw that all our tests we expected to fail failed. We then commented out the lines that caused faults with a comment at the end of the line to indicate it was a fault injection and which fault injection it was. We learned that multiple tests in a framework shows that a fault occurred and can

more easily show where it occurred when there are multiple calls to the methods. We also learned that one of the methods that was expected to pass before our fault injections fails due to it bypassing a precondition in a separate class.

Chapter 6
Overall Experience

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 JythonMusic 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 to work was available. Overall the experience of working in a team was good.

Project Feedback

As a team, we sat down and discussed what we thought of the project and what our feedback on it is. Our initial feedback is on the beginning days of the project as it felt like there was not enough explanation on what we were going to be doing. Because of this, it also sets students up for failure when they are choosing their project. While we liked that we could change our project after a deliverable or two, this took away time that could have been used perfecting the framework. Besides this, we agreed that the project was interesting and gave us much needed experience on how to use Git, GitHub, and test automation.