CSCI 362 Fall 2016

Software Engineering

Joey Baldwin

Eric Hofesmann

Marge Marshall

Cygnus Inter Anates

JythonMusic Testing Framework

Table of Contents

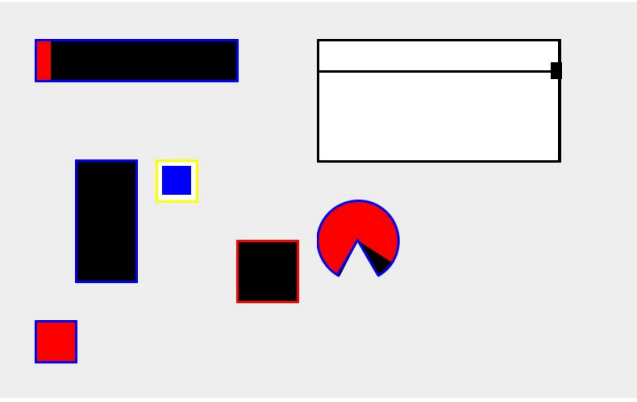
Chapter 1: Introduction to Project

JythonMusic is an open source music software project written in Python, created with Jython, and using several embedded Java software packages, such as jMusic and jSyn. jMusic is a package for music composition, and jSyn is for audio synthesis. These two packages provide a way to utilize the JythonMusic code. Our compile process was not complicated; a jython shell is included, which is used to compile files into Java bytecode, and the code is then run the same as Java code would be.

JythonMusic seems like it would be a useful tool for musicians, music composers, and programmers interested in music. The JythonMusic project gives the developer the means to compose, connect to external devices such as midi and osc, and also includes some simple gui functionality. There is also a user interface, the JEM editor, included in the source. The API is fairly simple and easy to follow.

There are nine files that make up the JythonMusic library, these are audio.py, gui.py, guicontrols.py, image.py, midi.py, music.py, osc,py, timer.py, and zipf.py. Six of these JythonMusic programs contain unit tests. These tests are positioned in main methods at the bottom of the files that they concern. The tests expose a variety of features within JythonMusic including UI elements and business functions. Tests are not ubiquitous throughout the project. Some files remain untested. These tests also are not complete, they appear to only test the essentials. There remain functionalities that have not been tested.

Gui.py is based on Java's Swing library, to give Jython GUI functionality. The test required image and sound downloads to work, as the test referenced files that did not exist and weren't included. However, once replacements were found for these, it generated 6 displays demonstrating control functions, animation and interaction of objects, and handling of input within the gui.



Guicontrols.py includes some extra gui widgets created with audio design in mind. The test demonstrated a range of UI features, but the control demonstrated by the test case was not as intuitive as one would expect from a quality UI -- vertical sliders did not respond to the mouse as it was over the slider, but when it was at the very edge of the display. Similarly, the circle did not respond when the mouse was over it, but as it moved around the full screen in a circular motion.

Midi.py takes input from a midi controller. Thus this has not been able to be thoroughly tested. It was able to be compiled properly. We have midi controller devices available to test input in JythonMusic.

Timer.py is a program that will take a function and repeat it on a set basis. The unit test involved creating a timer that counts seconds. It only tested one of the timer classes present in the file. The test performed what was expected of it, counting seconds, but it did not incorporate a broad testing range.

Zipf.py is a program that is used to calculate the slope and R^2 value the trendline of a zipf distribution. It works by entering an array of numbers and it calculates the zipfian distribution. There are multiple phenomena that were tested including white noise, brown noise, pink noise, monotonous, and a general case. These had to be tested separately by uncommenting the intended array. The slope and R^2 values were printed without errors.

Chapter 2: The Testing Process

The testing process will begin with manual testing, during which individual methods will be tested for expected outputs via the terminal. Manual testing will be done on a select number of methods, before moving on to automated testing. For automated testing, the selection of methods to be tested will be expanded, and these methods will be tested through a unit testing framework.

Requirements Traceability

- freqToNote - This method converts a frequency value given in hertz to the corresponding midi note value and midi pitch bend value
- noteToFreq - This method converts a midi note value to a frequency value in hertz
- frange - This method creates a range of floats, using a start value (inclusive), end value (exclusive) and a step value.
- colorGradient - This method creates a range of color values designated by lists of three integers between 0 and 256, exclusive. The values in the list represent Red, Green, and Blue levels. It uses two such color value lists (a start and a stop color, effectively), and an integer representing the number of gradient shades to generate in between. It then creates a range of color value lists that represent a gradient transition from the starting color value to the stop color value.
- getDelay - This method returns the delay time set for a particular timer
- getRepeat - This method returns whether or not a timer is set to repeat

Tested Items

The tested items will consist of the python programs within JythonMusic including timer.py, music.py, and gui.py. Methods to be tested within these programs include freqToNote noteToFreq, and frange within music.py. In the timer.py program, the methods getDelay and getRepeat will be tested. In gui.py, the method colorGradient will be tested. More tests will be added in the future.

Testing Schedule

Sept. 27 - Specify and develop 5 test cases, implement manually
Oct. 4 - Have 10 test cases specified
Oct. 11 - Have 15 test cases specified
Oct. 18 - Develop automatic testing framework, implement 5 test cases within the framework
Oct. 25 - Have 10 test cases implemented within framework, have 20 test cases specified
Nov. 1 - Have 20 test cases implemented within framework, have 25 test cases specified
Nov. 10 - 25 test cases implemented within an automatic framework
Nov. 17 - Have 5 faults designed and injected into code and retest using automatic framework
Nov. 22 - Analyze results of testing and fault injection

Test Recording Procedures

The test recording will be accomplished through the use of test case specification, oracles, and test reports. Each test will include a file describing the requirement, components, and methods being tested along with the test inputs and expected outputs. These expected outputs are specified through the use of oracles, where the outputs are manually calculated. A report is then generated with the test descriptions, the expected output, and the actual output of the test.

Hardware and Software Requirements

The main software these test cases will be used on is JythonMusic. The additional software required for this testing process will include a Jython compiler, and the java music programs jMusic and jsyn. Hardware requirements would include a midi output device like speakers.

Constraints

The main constraint affecting this testing process will be due to time limitations. Each deliverable is due only a few weeks apart from the next, thus there is a limit as to how many tests can be performed within that timespan. Other constraints include the size of the development group and the funding. There are only three people in this team and it is an unfunded project.

Chapter 3: Automated Testing Framework

Experiences

Our team communicated remotely through Slack and ended up meeting on Sunday afternoons and evenings. Our work style was that of working in parallel on different aspects of the project and communicating on any problems we ran into to see if other group members had solutions. We ran into some snags with using the python unit testing framework, but ultimately decided to continue using it, as it provided us with experience using a standardized unit testing framework and would accomplish our goals for this assignment.

How-To Documentation

Requirements: Linux OS, Python 2.7

1) Clone GitHub Repository:
git clone https://github.com/CSCI-362-01-2016/Cygnus-Inter-Anates

2) Navigate to the TestAutomation file,
cd TestAutomation/

3) Execute scripts/runAllTests.sh to run the framework and open the html file in the default browser.
./scripts/runAllTests.sh

Architectural Design

The testing framework is built around python's standard testing suite, unittest. Two python scripts create our automated framework: runTests.py and jythonMusicTestCase.py. These scripts read the testCase.txt files.

Test cases are stored in text files ranging from testCase00.txt to testCase25.txt. More can be added at any time. The structure of these can be seen in the section below entitled “Test Cases”.

JythonMusicTestCase.py contains two classes, JythonMusicTestCase and CustomResults which inherit from unittest’s TestCase class and Results class respectively.

JythonMusicTestCase class inputs the contents of the testCase.txt files and imports the required file dependencies to run the test’s intended method. These methods will be executed depending on the assertion provided. This class contains multiple assertions where the intended assertion is dictated in the testCase.txt files. These assertions are either testequal, or testalmostequal.

CustomResults is a class that will compile the results of the assertions along with the inputs. These are then used to generate an html file using a method within this class. One CustomResults object is created for all test cases and will store all assertion outputs.

runTests.py contains one class, called SetupTests, which creates a testSuite object that will contain all jythonMusicTestCase objects. It also contains a method that parses the test case text files and extracts the information. The suite is run in the main of this file, thus if it this program is executed, it will generate an html file of the test results.

File Structure

/Cygnus-Inter-Anates
- /TestAutomation
- /docs
  - README.txt
- /project
  - /src
    - jython, jMusic, jsyn, and imgscalr source code
    - gui.py
    - music.py
    - image.py
    - timer.py
  - /bin
    - compiled files
- /scripts
  - Jython.sh
  - runTests.py
  - runAllTests.sh
- /temp
- /testCases
  - testCase00.txt
  - testCase01.txt
  - …
- /testCasesExecutable
  - jythonMusicTestCase.py
- /reports
  - report.html

Test Cases

The template for our test cases is as follows, with a description below:

<test case id>
<description of function's purpose>
<module name> <class name if applicable>
<function name>
<input>
<test type of expected output>
<expected output>

The test case id is designated with any number of digits, and starts with the first test case at 00.
The description is a plain English explanation of what the function does.
The module name is the module or file in which the function is found.
The class name is the name of the class the function is in, if it is in a class.
The function name is the name of the function being tested.
The input is the current value or collection of parameters being passed to the function.
The test type dictates what type of assertion will be used to determine if the actual output passes or not. This is given in the form of the name of the test functions, such as testequals.
The expected output is what the function should return, given the input parameters.

Chapter 4: Completed Automated Testing Framework

Experiences

For this deliverable, our primary goal was being able to test methods inside of a class. We've managed to stay fairly on schedule, meaning that a functional testing framework was already in place. Much of the work was in adding functionality to the existing framework and specifying new test cases. While what we have currently meets the criteria, we had hoped to be able to expand it in a few other ways that we weren't able to finish, such as specifying test cases for setter methods.

Testing Framework Changes

For our testing framework, we've added the capability to test methods within a class. This was done with a driver for each method that is tested, which is only called when a class is specified by the test cases. As a result of this change, we also slightly altered the format of our test cases to include a class parameter, which is passed on the same line as the module name, separated by a space. We have also added the ability to specify the precision of the testalmostequals parameter. This can be modified in the testCase file by adding a comma and an integer next to testalmostequals.

Test Cases

A sample of current test cases:

00
calculate the frequency value of a given midi pitch
music
noteToFreq
69
testalmostequals, 2
440.00

05
Create range of floats
music
frange
1.0, 5.0, 0.5
testequals
[1.0,1.5,2.0,2.5,3.0,3.5,4.0,4.5]

10
calculate the color gradient given two colors and a step
gui
colorGradient
[0,0,0], [255,255,255], 2
testequals
[[0,0,0],[127,127,127]]

17
calculate the midi pitch of a given frequency value
music
freqToNote
451.0
testequals
69, 1751

20
Get if the timer repeats or not
timer Timer2
getRepeat
True
testequals
True

24
Get the delay time interval (in milliseconds)
timer Timer2
getDelay
10000000000
testequals
10000000000

Chapter 5: Fault Injection

Experiences

The fault injections for our framework were fairly straightforward -- it was easy to identify which tests were going to fail. Injection 1 changed every result for method noteToFreq(), injection 2 changed every result for frange() that was not a thrown exception, injection 3 changed the results for tests of frange() whose stop value was one step above the expected final value, injection 4 changed the results of noteToFreq() tests where the midi value was intended to be rounded up rather than down, and injection 5 changed the results for colorGradient tests that had a change in the red value. An important lesson to be learned from these fault injections is that not every fault can be discovered by every test. There were instances in which a test still passed even though there was an error in that method.

Injected Faults

music.py:

(1) Tests 0 - 4: noteToFreq, comment line 1239 uncomment line 1240, we changed
  frequency = concertPitch * 2 ** ((note - 69) / 12.0)
  to
  frequency = concertPitch * 2 ** ((note - 70) / 12.0)
Tests 0 - 4: failed

(2) Tests 5 - 10: frange, comment 611 uncomment 612, we changed
  if step > 0:
  to
  if step < 0:
Tests 5 - 9: failed, results only contained the first value in the list

(3) Tests 5 - 10: frange, comment line 614 uncomment line 615 we changed
  done = start >= stop
  to
  done = start > stop
Tests 5, 6: failed, results included an extra value at the end of the list

(4) Tests 16 - 20: freqToNote, comment line 1225 and uncomment line 1226, we changed
 note = round(x)
 to
 note = x
Tests 16 and 20: passed
Tests 17,18, and 19: failed

gui.py:

(5) Tests 11-15: colorGradient, comment line 337 uncomment line 338
  differenceR = red2 - red1
  to
  differenceR = red1 - red2
Tests 12,13: passed
Tests 11,14, and 15: failed

Chapter 6: Reflections

Github and Linux

For many of us, this was our first experience using github. While we may not have a comprehensive understanding of git or version control just yet, the project has provided us with the chance to exercise many of the basic git commands, as well as experiences with resolving merge conflicts, though merge conflicts were certainly the trickiest issues to resolve. We also gained a more in depth understanding of the linux terminal, and how to work with both bash and python scripts. For example, learning how to change the execution properties of a file.

Teamwork Makes the Dream Work

A vital part of our success was our ability to communicate regularly and work together. Group work is often difficult -- even in groups of talented and hardworking individuals, it's easy to see a project fall apart due to a lack of communication or an inability to resolve conflicts. Our group members were successful in staying in communication with each other about our work via Slack and weekly in-person meetings. These meetings were often productive in terms of the project goals but also offered us the opportunity to talk about other classes, assignments, and things happening outside of school. These asides were valuable for fostering a sense of camaraderie; this may seem like a non-essential or bonus aspect of team dynamics (or in some draconian work environments, a waste of time), but it does aid the flow of communication, ease conflict resolution, and also provides the benefit of positivity within the work environment.

Future Work

Because this project was only a semester long project, and all team members had other commitments, we were not able to make our ideal testing framework. It still has some limitations that future researchers or testers may want to address. One of these limitations is that we only addressed the framework's ability to create drivers for get methods within a class. In the future, the framework would ideally incorporate a way to generalize class initializations to remove the need for drivers entirely. However, if this generalization is not possible, an interface should be created to ensure the use of a test() method in each driver instantiation. Also, when testing for whether or not an exception has been thrown, the results do not indicate a value for the output, as the exception type is not converted to a string. Additionally, because we are building off of Python's unit testing framework which processes test cases altogether as a test suite rather than batching, it would be difficult for our framework to handle a large number of test cases without a large amount of memory.