Fat Mike’s Pizza Joint

Chandler DeLoach, Ethan Hendrix, Jayse White
College of Charleston, Department of Computer Science

---Table of Contents---

----

1. **Introduction**

   Page 3

2. **Instructions**

   Pages 4

3. **Examples of Execution**

   Page 5 -7

4. **Chapter 0**

   Pages 8-9

5. **Chapter 1**

   Pages 10-11

6. **Chapter 2**

   Pages 12-13

7. **Chapter 3**

   Pages 14-15

8. **Chapter 4**

   Pages 16-18

9. **Chapter 5**

   Page 19

10. **Chapter 6**

    Page 20

11. **Evaluation**

    Pages 21

----------------------- ---Introduction--- -----------------------

For this project, we were tasked with developing an automated testing framework that we would implement over an H/FOSS project. The process was split into several deliverables, starting with choosing a project, cloning their GitHub directory to our local machines, compiling it, then begin developing our testing framework. This report is organized into several chapters, each chapter marking a deliverable that our team submitted.

In the spirit of agile development, our project changed over time. These changes will become evident as you read throughout the chapters. Whereas our team started with one idea, this changed and evolved over time into a project that looks totally different than we initially conceived.

---Instructions---

This section will cover the instructions that a user would need to follow to user our testing framework on their code base.

Set Up:

1. *git clone* [*https://github.com/CSCI-362-01-2017/Fat-Mike-s-Pizza-Joint.git*](https://github.com/CSCI-362-01-2017/Fat-Mike-s-Pizza-Joint.git)
2. *cd Fat-Mikes-Pizza-Joint/TestAutomation/scripts/*
3. To allow testing of your own project's code, you will want to create a symlink in within the *TestAutomation/project/src/* directory.
    1. A symlink to this directory is provided for your convenience inside of *scripts/*.
    2. To create the symlink:
        1. *cd src/*
        2. *ln -s <path_to_your_projects_root>*
        3. *cd .. (*to return to *scripts/)*
4. Write your test cases in *TestAutomation/testCases* following the syntax. Again, for convenience, a symlink to this directory is provided inside of *scripts/*
5. You may want to create a symbolic link for your project's root directory in *TestAutomation/scripts* for your convenience

To Test Your Code Base:

1. Write test cases
2. navigate to *TestAutomation/scripts* directory
3. *./runAllTests.sh*
    1. This will run all test cases within the *testCases/* directory, and display the results as a report displayed in your web browser.
4. *./runSingularTest.sh testCases/testCaseX.yml*
    1. This will run a singular test case of your choosing and display the results as a report displayed in your web browser

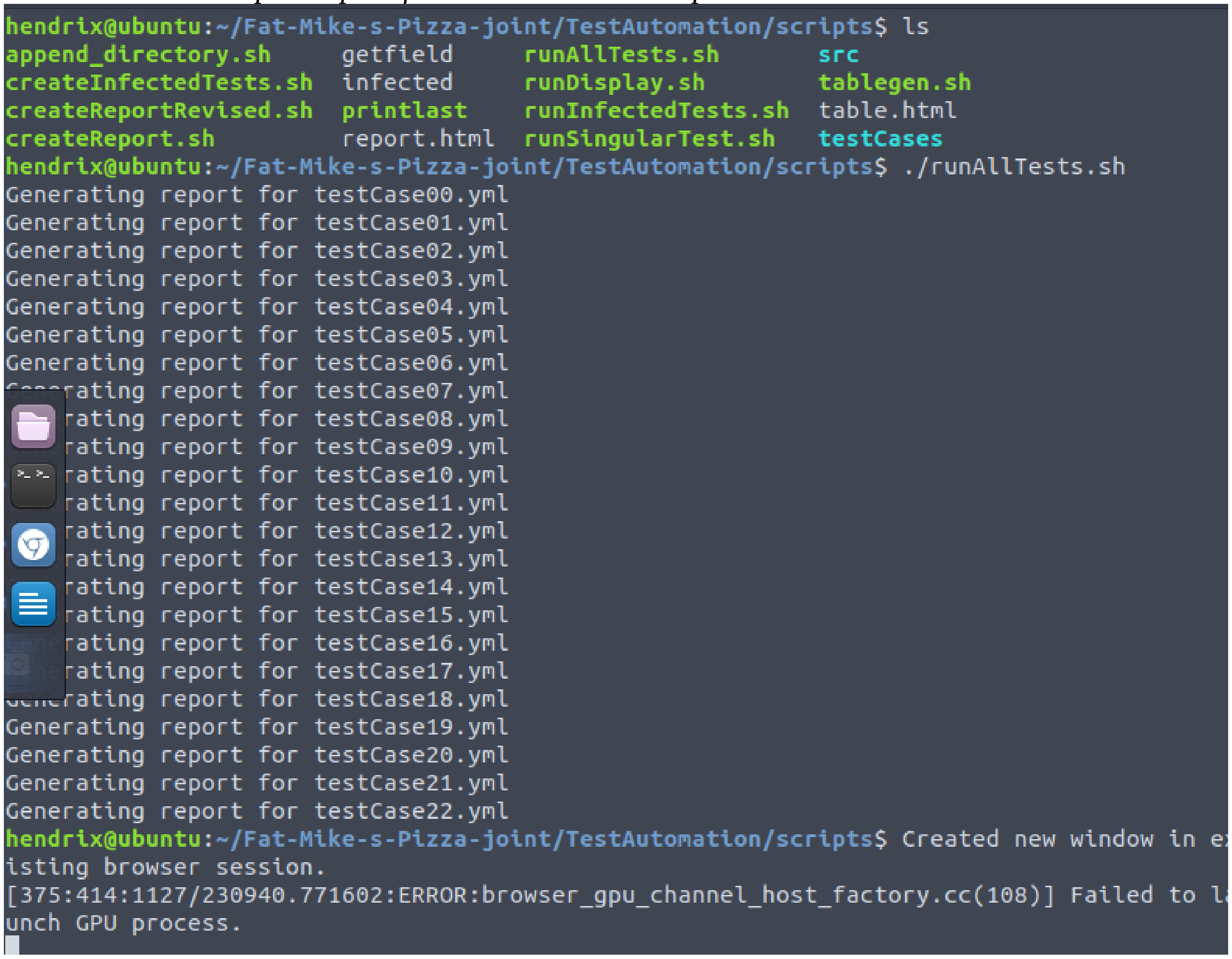
To Set Up Fault Injection:

1. Have written test cases
2. Edit *TestAutomation/scripts/infected*
    1. This will contain a directory path, specifying where to look within your project files for modified code containing faults you have injected
3. *./createInfectedTests.sh*
    1. This will clone test case files into an infected test cases directory
4. *./runInfectedTests.sh*
    1. this will run the tests form your infected test cases directory

----
---Example Execution---

This section will include some screen shots showing the example execution of the scripts.

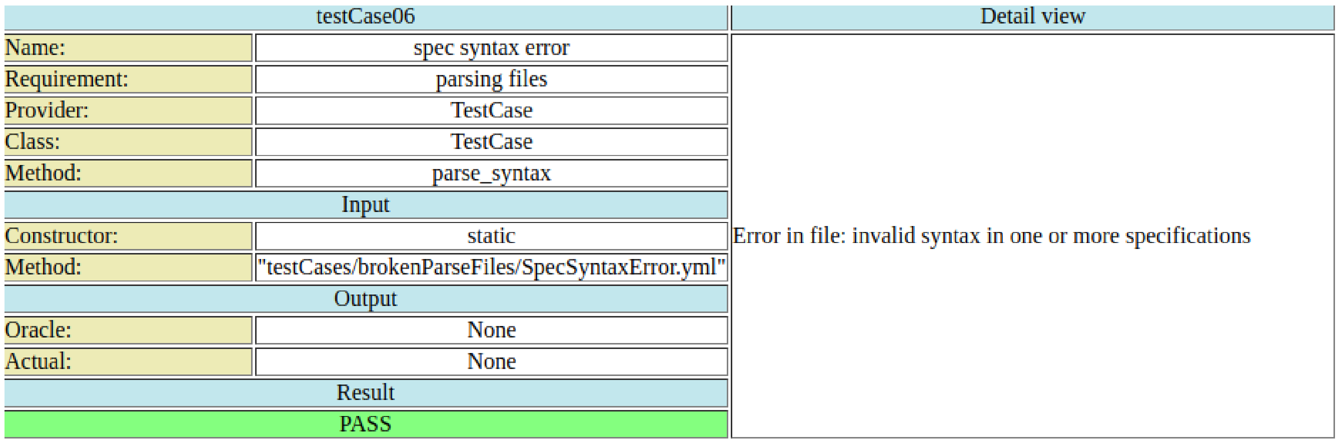
Screen shot 00: *a sample output of the runAllTests.sh script*



```
hendrix@ubuntu:~/Fat-Mike-s-Pizza-joint/TestAutomation/scripts$ ls
append_directory.sh     getfield     runAllTests.sh       src
createInfectedTests.sh  infected     runDisplay.sh        tablegen.sh
createReportRevised.sh  printlast    runInfectedTests.sh  table.html
createReport.sh         report.html  runSingularTest.sh   testCases
hendrix@ubuntu:~/Fat-Mike-s-Pizza-joint/TestAutomation/scripts$ ./runAllTests.sh
Generating report for testCase00.yml
Generating report for testCase01.yml
Generating report for testCase02.yml
Generating report for testCase03.yml
Generating report for testCase04.yml
Generating report for testCase05.yml
Generating report for testCase06.yml
    rating report for testCase07.yml
    rating report for testCase08.yml
    rating report for testCase09.yml
    rating report for testCase10.yml
    rating report for testCase11.yml
    rating report for testCase12.yml
    rating report for testCase13.yml
    rating report for testCase14.yml
    rating report for testCase15.yml
    rating report for testCase16.yml
    rating report for testCase17.yml
    rating report for testCase18.yml
Generating report for testCase19.yml
Generating report for testCase20.yml
Generating report for testCase21.yml
Generating report for testCase22.yml
hendrix@ubuntu:~/Fat-Mike-s-Pizza-joint/TestAutomation/scripts$ Created new window in e
isting browser session.
[375:414:1127/230940.771602:ERROR:browser_gpu_channel_host_factory.cc(108)] Failed to la
unch GPU process.
```

In the above screen shot, you can see what the user would see when they execute the script to run all tests. This script will iterate through the test cases' directory and generate a report for them. After it has done so, it will launch the browser to display the final HTML report. This is what you see in the bottom line. The *Failed to launch GPU process* error can be ignored – this is a local error on the machine executing the code and not an error associated with the execution of this script.

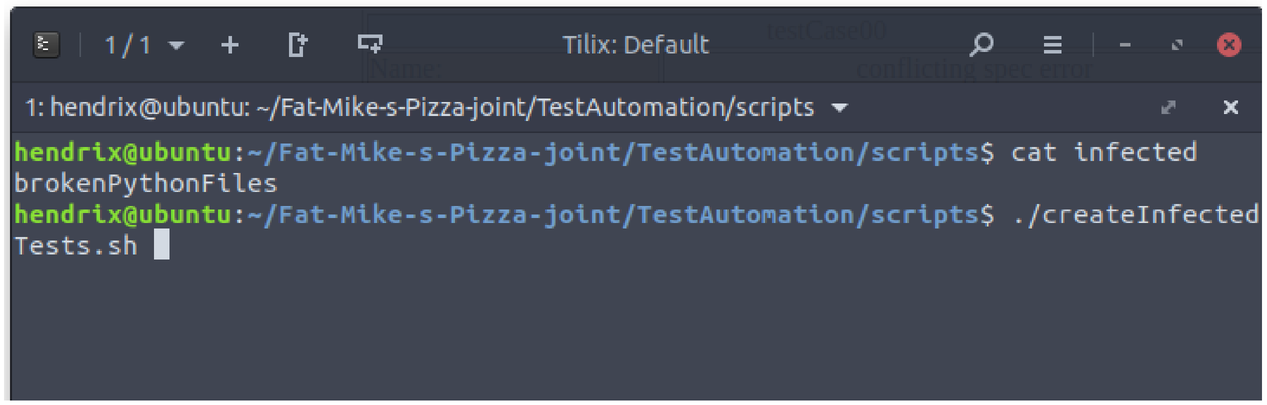
Screen shot 01: *an example of an output table in the report*



testCase06		Detail view
Name:	spec syntax error	
Requirement:	parsing files	
Provider:	TestCase	
Class:	TestCase	
Method:	parse_syntax	
Input		
Constructor:	static	Error in file: invalid syntax in one or more specifications
Method:	"testCases/brokenParseFiles/SpecSyntaxError.yml"	
Output		
Oracle:	None	
Actual:	None	
Result		
PASS		

This screen shot displays one of the many tables that are displayed in the final report. The Result is highlighted green for a PASS and red for a FAIL. The detail view displays the exact error that is returned from execution. A blank detail view indicates the test case was written as a valid test case and no error was returned.

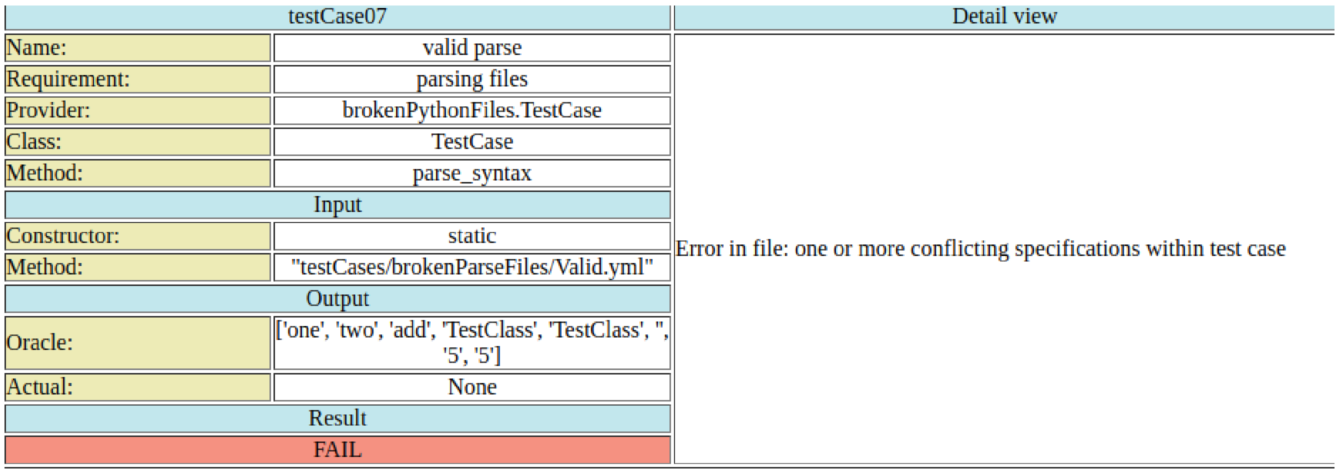
Screen Shot 02: *example of automatically creating your fault-injected files directory*


1 / 1
Tilix: Default
1: hendrix@ubuntu: ~/Fat-Mike-s-Pizza-joint/TestAutomation/scripts

```
hendrix@ubuntu:~/Fat-Mike-s-Pizza-joint/TestAutomation/scripts$ cat infected
brokenPythonFiles
hendrix@ubuntu:~/Fat-Mike-s-Pizza-joint/TestAutomation/scripts$ ./createInfected
Tests.sh
```

In this screen shot, the user has already altered the file *infected* to name the directory for their fault-injected files. After running the second command you see here, *./createInfectedTests.sh*, the testing framework will clone the necessary files and tests from the source directory into the fault-injected directory to ensure that they are isolated. Essentially, this is automating the process of setting up directories to inject faults.

Screen Shot 03: *example fail test due to fault injection*



testCase07		Detail view
Name:	valid parse	Error in file: one or more conflicting specifications within test case
Requirement:	parsing files	
Provider:	brokenPythonFiles.TestCase	
Class:	TestCase	
Method:	parse_syntax	
Input		
Constructor:	static	
Method:	"testCases/brokenParseFiles/Valid.yml"	
Output		
Oracle:	['one', 'two', 'add', 'TestClass', 'TestClass', '', '5', '5']	
Actual:	None	
Result		
FAIL		

In this screen shot, you can see that the test has failed due to the fault being successfully injected. If you notice the provider field, you can see that the *brokenPythonFiles* directory that we specified in the screen shot above.

---Chapter 0---

For our first deliverable, we had to choose a candidate H/FOSS project for which our team wished to develop our automated testing framework. After doing some research of available H/FOSS projects, our team narrowed the pool of candidates down to three good choices. Then, we each debated the choices, putting forth arguments for why we would be more interested in doing one project over the other. Below are our top 3 candidates.

Cadasta: "Technology to help communities document their land rights around the world."
Main website: http://cadasta.org/
GitHub: https://github.com/Cadasta/cadasta-platform
Reasoning: Cadasta caught our attention, because it is an interactive map to document land rights around the world. This is particularly interesting because of the political implications of land rights, especially in regions of the world such as the Middle East. Cadasta is primarily implemented in Python, and its GitHub repository has just under 1,000 commits.

Martus: "Martus, the Greek word for "witness," is a software tool that allows users to document incidents of abuse by creating bulletins, and storing them on redundant servers located around the world. Using Martus helps countries torn apart by civil conflicts come to a consensus and rational understanding of their histories, leading to reconciliation and reform processes."
Main website: https://www.martus.org/
GitHub: https://github.com/benetech/Martus-Project
Reasoning: Martus caught our attention, because it is software aimed at aiding less-developed countries in documenting and understanding reports of abuse. Additionally, it was interesting that Martus emphasizes security and redundant servers. Martus is implemented in Java, and is hosted on a much smaller repository than our other two choices, Martus and ITK.

Insight Segmentation and Registration Toolkit (ITK): "National Library of Medicine: ITK is an open-source, cross-platform system that provides developers with an extensive suite of software tools for image analysis. Developed through extreme programming methodologies, ITK employs leading-edge algorithms for registering and segmenting multidimensional data."
Main website: https://itk.org/
GitHub: https://github.com/InsightSoftwareConsortium/ITK
Reasoning: ITK caught our attention, because it's aimed at aiding developers with image analysis. Additionally, it is not end-user software but a library given to developers that are building tools for end-users in the medical field. It's a huge project with over 45,000 commits, so this would provide invaluable experience for our team with working with large projects. It is additionally appealing, because of the learning opportunity to become more familiar with C++.

After consideration to these projects, we all decided that we were going to pursue the Martus project, given that it aims at less-developed countries while still focusing on security.

------------------- ---Chapter 1--- -------------------

For this deliverable, we were to clone our chosen project’s GitHub repository to our local machines and manually build the code. At first, it seemed like we would be able to do this with our original selection, Martus, but we struggled with compiling the code. Further research, we discovered enough evidence to suggest that the Martus project may be dead. Many of the dependencies that were required for compiling and running this code were out-of-date, and many of them were not hosted for download any longer. Given that we could not compile the code independently and that the project was dead, we decided to consider changing projects.

We revisited the other project candidates listed in the previous chapter. This time, Cadasta, the land-rights documentation program, caught out attention, but we were concerned with ensuring that this project was not dead. Our team noticed there were over one-thousand commits, and they were as recent as the previous week. Additionally, the dependencies were up-to-date and their GitHub Wiki had an easily understood installation instructions for development. Essentially, to install Cadasta for development, the user creates a Vagrant virtual machine to run as their Cadasta server, and then they can SSH into the server to run commands.

What we found inside was that Cadasta had a fairly comprehensive testing framework already implemented. A Python file(runtests.py) runs all of their integrated tests and reports the results. Additionally, they provide the option to export these test results to HTML files. After the tests are finished, it reports a quick overall result: "1998 passed, 501 warnings in 972.86 seconds."

When examining the test result's directory, what you find is a collection of many HTML files that correspond to each unit of the software. At the bottom, is an HTML file labeled "index" that provides a quick overview of all test results. If you wish to view a specific test, however, you can browse to that HTML file specifically. What this does is it allows a developer to get a quick, comprehensive overview of all the unit tests, but still allows them to delve deeper and get a granular view of the unit tests.

Altogether, Cadasta appears to be a much more approachable project that is aimed at OSS development. The easy installation instructions combined with a well-maintained code base give much more confidence than Martus. Additionally, the current testing framework inspires ideas for furthering the testing database.

However, with Cadasta, we would continue to run into unexpected complexity. Cadasta, being a web application, relied very heavily on Django, the Python-written web framework. It would turn out that handling the Django imports and executing the tests would prove too difficult to write an automated testing framework. After this unexpected complexity became too great of a roadblock, we, once again, reevaluated our project and decided to change.

At that point in development of our automated testing framework. We had coded a testing framework that was generic enough to theoretically handle any Python-written code base given that the user written the test cases in the proper syntax. Additionally, our testing framework’s code base was large enough to support the number of test cases needed for the project’s specifications. After consideration and consultation with Dr. Bowring, we changed our project to test itself.

---Chapter 2---

Fat Mike's Pizza Joint - Testing Plan

The Testing Process:

- Write Tests
- Automate running tests
- Collect Data
- Log testing process
- Compare to oracles
- Prepare results for overview

Requirements Traceability: The following requirements will be traced and reported throughout the project as an HTML report**:**

- Reading test cases
- Checking validity of test case's syntax
- Executing test case
- Exporting results of test case to HTML files
- Creating HTML report
- Open HTML in web browser automatically

Tested Items:

- Fat Mike's Pizza Joint Testing Framework
    - Driver.py
        - Execution
            - Handing imports
            - Handling errors
            - Reporting results from test cases
    - TestCase.py
        - Parsing test cases
        - Single line parsing
        - Storing values

Testing Schedule:

- 03 October 2017 – Have initial five test cases
- 31 October 2017 – Initial build of testing framework with first five test cases
- 14 November 2017 – Complete design and building of testing framework with twenty-five test cases
- 21 November 2017 – Inject faults into code base
- 30 November 2018 – Present final report on automated testing framework

Test Recording Procedures:

- Echo results from test cases into HTML files
- Compile complete HTML report
- View complete HTML report in browser

Hardware and Software Requirements:

- Linux 32-bit/64-bit

- Python 3
- Web browser (i.e. Chromium, Firefox)

Constraints:

- *Time*: This project takes place over the course of one semester, therefore we must work to complete all of the work before the semester's end.
- *Scheduling*: Out of three group members, we must coordinate our schedules and create time to perform the necessary work
- *Complexity:* When working with another code base, we may run into unexpected complexity with a foreign code base that we did not expect.

System Tests:

- *Out-of-scope for this project*

Five Initial Test Cases:

- testCase00.yml
  - constructor input: *static*
  - method input: "testCases/brokenParseFiles/ConflictingSpecError.yml"
- testCase01.yml
  - constructor input: *static*
  - method input: “testCases/brokenParseFiles/InvalidSpecTypeError.yml"
- testCase02.yml
  - constructor input: *static*
  - method input: testCases/brokenParseFiles/InvalidSubSpecError.yml"
- testCase03.yml
  - constructor input: *static*
  - method input: "testCases/brokenParseFiles/MissingSpecError.yml"
- testCase04.yml
  - constructor input: *static*
  - method input: "testCases/brokenParseFiles/NullFieldError1.yml"

-------------------- ---Chapter 3--- --------------------

For the third deliverable, we worked on beginning the implementation of our testing framework. This initial implementation and design would include the original five test cases that are listed below:

- testCase00.yml
    - constructor input: *static*
    - method input: "testCases/brokenParseFiles/ConflictingSpecError.yml"
- testCase01.yml
    - constructor input: *static*
    - method input: “testCases/brokenParseFiles/InvalidSpecTypeError.yml"
- testCase02.yml
    - constructor input: *static*
    - method input: testCases/brokenParseFiles/InvalidSubSpecError.yml"
- testCase03.yml
    - constructor input: *static*
    - method input: "testCases/brokenParseFiles/MissingSpecError.yml"
- testCase04.yml
    - constructor input: *static*
    - method input: "testCases/brokenParseFiles/NullFieldError1.yml"

The testing framework begins with the test cases themselves, as they have specific syntax, shown below:

name: *name of the test case*
requirement: *requirement being tested*
component:
    class: *class containing the method being tested*
    provider: *Python file providing the class above*
method: *method being tested*
input:
    constructor: *comma-separated arguments to constructor*
    method: *comma-separated arguments to method*
oracle: *expected method return*

The syntax of the test case is important, because it allows us to create reports in an HTML file to give the user useful information about how their code base fared based on the test case that they specified. The syntax of a test case file is verified to be correct before they are used to create a test case object. The user is notified if a test case is incorrectly written, so that they can return to the test case and re-write it using the proper syntax that we have provided.

We derive the path to the Python file in the provider field, and the class is directly imported using the class field. The true heart of the test case is in the constructor input field and method input field. These are the actual values that are used to instantiate objects from the python classes, and then to execute the methods.

After writing the test cases in the appropriate directory, *TestAutomation/testCases*, the user can then navigate to *TestAutomation/scripts*, and then run *runSingularTest.sh* or *runAllTests.sh*. The singular test script will take the path to a test case file as an argument, then report the results. The all tests script will take no arguments, and it will iterate through the test cases' directory, executing all test cases, and compiling the results into an HTML file for the user.

-------------------- ---Chapter 4--- --------------------

For this deliverable, we were to complete the design and implementation of our testing framework. We finished writing the test cases for our project, which are listed below:

- **testCase05.yml**
  - constructor input: *static*
  - method input: "testCases/brokenParseFiles/NullFieldError2.yml"
- **testCase06.yml**
  - constructor input: *static*
  - method input: "testCases/brokenParseFiles/SpecSyntaxError.yml"
- **testCase07.yml**
  - constructor input: *static*
  - method input: "testCases/brokenParseFiles/Valid.yml"
- **testCase08.yml**
  - constructor input: *static*
  - method input: "nonmatch","write",[["match","",0]]
- **testCase09.yml**
  - constructor input: *static*
  - method input: "match","write",[["match","",0]]
- **testCase10.yml**
  - constructor input: *static*
  - method input: “ “
- **testCase11.yml**
  - constructor input: *static*
  - method input: “class TestCase”
- **testCase12.yml**
  - constructor input: *static*
  - method input: “requirement:”
- **testCase13.yml**
  - constructor input: *static*
  - method input: “”
- **testCase14.yml**
  - constructor input: *static*
  - method input: “requirement: single line parsing”
- **testCase15.yml**
  - constructor input: TestCase(["name","req","add","NonExistent","TestClass","","10","10"])
  - method input: *void*
- **testCase16.yml**
  - constructor input: TestCase(["name","req","add","TestClass","NonExistent","","10","10"])
  - method input: *void*
- **testCase17.yml**

    - constructor input: TestCase(["name","req","nonexistent","TestClass","TestClass","","10",”10”])
    - method input: *void*
- **testCase18.yml**
    - constructor input: TestCase(["name","req","add","TestClass","TestClass","",10,10])
    - method input: *void*
- **testCase19.yml**
    - constructor input: TestCase(["name","req","add","TestClass","TestClass.py","","something”,10])
    - method input: *void*
- **testCase20.yml**
    - constructor input: TestCase(["name","req","add","TestClass","TestClass","",[10],[10]])
    - method input: *void*
- **testCase21.yml**
    - constructor input: TestCase(["name","req","add","TestClass","Something","","10","10"])
    - method input: *void*
- **testCase22.yml**
    - constructor input: TestCase(["name","req","add","TestClass","TestClass.py","",10,10])
    - method input: *void*

With all of these test cases, we had the following amounts testing various elements of the code base.

- Parsing test cases: eight test cases
- Parsing a single line: five test cases
- Storing a value: two test cases
- Various pieces of driver execution (see test plan): eight test cases

With these test cases, we could adequately test our own testing framework. However, testing our own framework presented some interesting challenges that we may not have expected. One of these challenges, perhaps the greatest, was how to handle executing test cases that we knew would break our code while still allowing our code to continue to execute the rest of the test cases. To overcome this, we decided to create mock instances of the classes that we specified in our test cases. For example, the driver object would create another instance of itself, so that it could execute test cases that would halt execution of the sub-driver while the super-driver could continue to execute. This was also applied to test case objects. A valid test case would create a test case object, but that test case object would contain an invalid test case that would throw the errors we expected.

Another challenge was how to write test cases with bad syntax so we could test parsing, but these, again, should not halt execution of our super-driver. To overcome this, we simply separated the “broken” test cases from the valid test cases that we were actually using. The valid test cases would simply contain a path to the bad test cases. This is not unlike a user would interact with this software, as the path would change for any test case depending on the class and method the user was writing the test case for.

We were able to get all of these test cases successfully implemented after careful work and great determination. This deliverable was likely the most challenging out of all of the deliverables, because it included the entirety of our design and implementation.

Our testing framework was coming along nicely, and working successfully. All we had left to do was inject faults into our code base.

---Chapter 5---

For this deliverable, we were to inject faults into our code base with creating at least five test cases to fail. Since we were testing our own code base, however, we had to once again work around the problem of injecting faults into our code base while still being able to test it using our code base.

To overcome this, we cloned the necessary Python files into a “brokenPythonFiles” directory. This directory could contain the fault-injected Python files. This eliminates the chance of mixing fault-injected files with non-fault-injected files. After this, we re-pointed the Driver to this directory to execute the test cases.

In practice, the user would not have to do this to this degree, just rewrite the test case to point to a fault-injected file as they specified. Our approach was only to overcome the complexity and challenge of testing our own code base.

We successfully injected faults into our code base to create seven test cases to fail. Thankfully, as we hoped, this did not break the entirety of the test cases. After injecting faults, we had sixteen test cases that were still successfully passing. Additionally, these faults did not break all of the test cases for that specific class. Of the original ten test cases we had for this particular class, we had three test cases still passing, and one of those was for this specific method.

Because of the work we put into the last deliverable, this deliverable proved to be less challenging, and we were able to inject these faults with greater ease than initially anticipated.

-------------------- ---Chapter 6--- --------------------

Over the course of this project, our team learned a great deal about the development process, especially testing. The initial project of learning how to create an automated testing framework proved challenging at times, if not downright frustrating, but it was a worthwhile experiment.

There were some things we did well. For example, we didn’t fear reexamining our project, our progress, or our design. Each deliverable, it seemed we revisited the design of our framework, and tweaked and tuned the design to create a better implementation of our testing framework. Some of these changes were drastic, such as changing the projects, while some were more minor, such as deciding how our scripts should iterate through our test case directory.

Admittedly, there were some things that we did not do well. For example, we didn’t schedule our progress as well as we could have. We went by the original deliverable due dates in the beginning, when we really could have benefited from greater scheduling with better, more-clear goals. The periods between deliverables didn’t have schedules, simply the deadline of having work done before it was turned in. This is undoubtedly why we encountered such difficulty in the fourth chapter. We didn’t pace ourselves in the development, and we were forced to complete our design and implementation under the wire.

Additionally, we may have been able to avoid the debacle of constant project change had we done greater research in the inception stage of our project. Given that, however, we adapted incredibly well over the course of the project, and we didn’t let the difficulties and complexities of the projects we had chosen hinder our performance. In this context, we regrouped, redesigned, and continued building with great efficiency. In a sense, this may have prepared us better beyond college, having the experience of project change at a fast pace.

---Evaluation---

Overall, creating a testing framework is a worthwhile experience. Although we may not necessarily use the testing framework we've created, but it enforces important development ideas and strategies, such as test-driven-development, scripting, and agile development ideas.

It's difficult to assess the accuracy of this project, because of the scope and length of time. One would assume to create an entire testing library, the development cycle would be longer and have a scope greater than ours. Whereas our scope didn't include system tests, an automated testing framework might include some utilities to facilitate that. It's difficult to argue that the length of the project should be lengthened given that it already takes place over the entire course of the semester. Likewise, broadening the scope would likely exceed what's possible in a semester.

It was well-paced. It never felt like a deliverable was too close to manage until the end of the semester when they started to pile up, but this was due because of inclement weather delaying our schedule. If a team paced themselves well throughout the semester and scheduled themselves well between deliverables with clear goals, this could likely still be overcome.

At it's core, it's challenging to develop anything open source or over an open source project. A lot of those challenges are evident in this project. Teams other than our own struggled with their open source projects as well. Build issues were likely the most common, but there were plenty of examples of poorly-commented code and bad design. Many of the code bases we examined seemed ambiguously designed, making it difficult to understand how different aspects of the code interact with one another. Furthermore, within certain files, there seemed to be ambiguous or bad design that made it difficult to understand what was happening within one Python file. Again, it's doubtful that anything could be done about this; this is the nature of open source software. It's worthwhile to have the experience, no doubt. Examining foreign code bases and bad code will happen frequently in the professional world of software development whether working on an open source project or not.