

# Eden



## A Sahana Software Foundation Testing Framework

By: Aaron Monahan  
Steven Draugel  
Jeremy Butcheck  
Patrick Owens

Date: December 01, 2015



**School of Science and Mathematics**  
**Department of Computer Science**  
**CSCI 362: Software Engineering**  
**Dr. Jim Bowring**

**Table of Contents**

Introduction .....	
Chapter 1 .....	
Chapter 2 .....	
Chapter 3 .....	
Chapter 4 .....	
Chapter 5 .....	
Building the Repository .....	
Testing Framework .....	
Test Case Examples .....	
Fault Injection .....	
Conclusion .....	

## **Introduction:**

The following is a final report of our project which details the steps our team took to build an automated testing framework in an open source project. The Sahana Software Foundation is an open source humanitarian platform that supports a number of software products that help manage relief efforts as a result from natural disasters. Eden is a project under Sahana that uses databases to organize data. It has been deployed in many instances since its conception including: 2013 Typhoon Yolanda in the Philippines, 2012 wildfires in Chile, and 2012 Hurricane Sandy in US.<sup>1</sup> We chose to work with Eden because it's repository was well documented. Additionally, the source code was written in Python, a language all of our group members are familiar with.

Each chapter of this report is text taken from deliverables submitted in chronological order from the beginning of the project. This will show the methods we came up with as the assignment progressed.

## **Chapter 1:**

So far our group has been able to check out a copy of the Eden project and build/install it into a local vm running Linux Redhat. Initial installation proved to be a little bit of a challenge due to confusing installation instructions. They have, as we have seen with other installation procedures, failed to document every step in the process. They did this either because they simply forgot or, more likely, just assumed that the user had sufficient knowledge in installing their brand of software. We found out later that Sahana Eden had helpful information on the installation process for developers.

After the initial installation hurdles were overcome, we followed the test suite build procedure. Again, this process was not well documented and lead to wasted time looking up where files were located. Once the test suite was built we ran the test suite script runner. This script ran like it was supposed to but none of the tests passed. This is because the test runner script calls a file from a web address that the user needs admin rights (login credentials) to access. As of right now, we have requested access.

Along with python code which we will test, the test suite uses the Robot Framework which is a generic test automation framework for acceptance testing and acceptance test-driven development. Robot works with Selenium to create tests. The suite implements multiple tests like:

- Smoke Testing - clicks through on every link within Sahana Eden and can be used to check for errors on pages and broken links.
- Unit Tests - used to test whether specific "Units" of code are working. They are used extensively to test the Sahana Eden "S3" Framework.
- Role Tests - used to check the permissions of user roles.
- Benchmark Tests - a simple way of measuring the performance of the Sahana Eden "S3" Framework.
- Load Tests - measures the performance of the entire Sahana Eden stack.

## Chapter 2

During this deliverable we laid out our overall testing schedule as well as the systems and code that we would be testing. First off we decided to test two aspects of the Eden project; the Selenium test framework and Python code.

- Web tests
  - Login functionality (valid/invalid)
  - 'Organizations' text located on screen (see fig. 1)
- Database tests
  - Registered user
  - Assets
  - Volunteers
  - Warehouses
- Unit tests
  - Python methods receiving test inputs



**Tested items**

The products of the software process that are to be tested.

- Web Client
- Source Code
- Database

**Testing schedule** - An overall testing schedule and resource allocation.

September 29, 2015 - Deliverable #2

- Produce detailed test plan
- Test cases at least 5 out of 25 mapped out

October 22 - Deliverable #3

- Rework test plan if needed
- 10/25 test cases
- Build automated testing framework

November 12 - Deliverable #4

- 25 of 25 test cases using automatic framework

November 24 - Deliverable #5

- Design and inject 5 specific failures into code

December 1

- Final Report
- Poster
- Presentation

**Hardware and software requirements**

Software:

- Web2py
- Linux Ubuntu/Virtualbox
- Selenium Framework

- Python

### Constraints

- Group size
- Time Conflicting schedules - Our group has difficulty finding times that are convenient for everyone to meet

## **Chapter 3**

For this deliverable we wrote a bash script to integrate our different python framework test scripts. The end goal of this script was some sort of html page that worked as a summary of our test suite results. Our testing was limited to the behavior of the client using a web-driver. Tangentially we tested other components of the web stack by looking at different client pages which relied on other subsystems in the server code. Later on we plan to have a companion driver integrated into our bash script that can execute component testing.

Currently we are running the test script in the web2py environment so that we have access to the entire web stack. Later on this will be useful for testing things like the database, business logic, service APIs, etc.

Initially the implementation of this project was less very difficult. We wanted to test the behavior of the client, which was somewhat of a complex issue. We leveraged pre existing testing code that defined simpler behavior from which, in the future, we could create more complex behavior to test for. For instance, from their testing library they have a start browser procedure that we can use in our own code before checking other client behavior like links, data entry, account creation etc. Unfortunately we were not able to write a script to work with this code, so there are some errors to our implementation. The first issue was that their test suite requires new tests to be in their test suite directory, to solve this we symlinked our test executable folder to their directory. The second issue was that because we wanted our tests to run sequentially, we were moving and removing our test scripts into this folder as the test driver ran.

As we get more familiar with the testing framework we will be able to move more of the functionality into our own code and use less of the pre existing code and/or use their code more effectively allowing us to simplify the testing process.

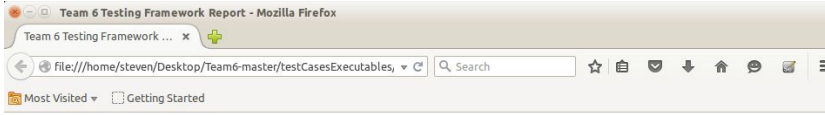
As a closing thought, we are happy with the prototype script and are excited to write test cases for it. In the future we plan to add a bit of design to the html report. As it stands, the page is quite plain. Finally, as we become more familiar with the pre existing code we will add different types of test cases to our test suite.

## **Chapter 4**

Currently our testing framework does two things: tests the source Python code, and tests the web interface using Selenium. Reporting is limited now since it is separated into three reporting documents, namely: Deliverable #1,2,3.

Python source code testing for the project is being done using the traditional method. We are using 15 text files containing the class name, method name, input parameters, oracle name, and driver name to provide input to a main test case runner. The runner looks into the testCases folder and systematically runs a test for each .txt file contained within it. During each test the runner checks the input parameter against the output parameter contained in the oracle. If they are correct, a report file is updated with a pass or fail along with the test case number and description of the test.

Selenium Framework testing is done a little different than the Python source code testing. There are five test cases that the main runner will see. However, the runner will pickup on "selenium" contained in the driver line. When this happens, another script handles the testing of the Selenium framework test cases. Each test file contains commands for running each test as well as checking for the correct inputs and outputs. Currently there are no oracles for these tests since selenium has a built in way of handling this. After each test is done, an HTML document is updated with the result.



Class Name:	Module Name:	Input:	Description:	Oracle:	Result:
modules feedparser	parse_date_iso8601	2014331337	parses the date to send the information to another method to build a date	oracle08.txt	PASS
modules EXIF	make_string	aaaaa	Turns the input into a string	oracle12.txt	PASS
modules EXIF	make_string	12345	Turns the input into a string	oracle11.txt	PASS
modules feedparser	parse_date_iso8601	99:53:00+05:00	parses the date to send the information to another method to build a date	oracle10.txt	PASS
modules EXIF	s2n_motorola	L	converts the input into an int	oracle20.txt	PASS
modules EXIF	make_string	123abc	Turns the input into a string	oracle14.txt	PASS
modules EXIF	s2n_motorola	&	converts the input into an int	oracle18.txt	PASS
modules feedparser	parse_date_iso8601	201540105	parses the date to send the information to another method to build a date	oracle06.txt	PASS
modules feedparser	parse_date_iso8601	15:53:00+05:00	parses the date to send the information to another method to build a date	oracle09.txt	PASS
modules EXIF	s2n_motorola	a	converts the input into an int	oracle16.txt	PASS
modules feedparser	parse_date_iso8601	2014331207	parses the date to send the information to another method to build a date	oracle07.txt	PASS
modules EXIF	make_string	abc123	Turns the input into a string	oracle15.txt	PASS
modules EXIF	make_string	!\$%	Turns the input into a string	oracle13.txt	PASS
modules EXIF	s2n_motorola	3	converts the input into an int	oracle17.txt	PASS

Fig 1: A picture of the web results produced by our testing framework.

## Chapter 5

For the fifth deliverable in the project, we were assigned to design and inject 5 faults into our code that causes at least 5 tests to fail, but not all tests to fail. These 5 faults should be simple, such as flipping some login in 'if' statements or a small typo in a function call. (see figure 2.)

As a progress report from our last deliverable, we have a functioning framework that automatically runs tests on the Eden application. The framework is automated to be run through the command line and display 25 test results in an html page. Since deliverable 4, we have caught up on our framework and functions as normal.

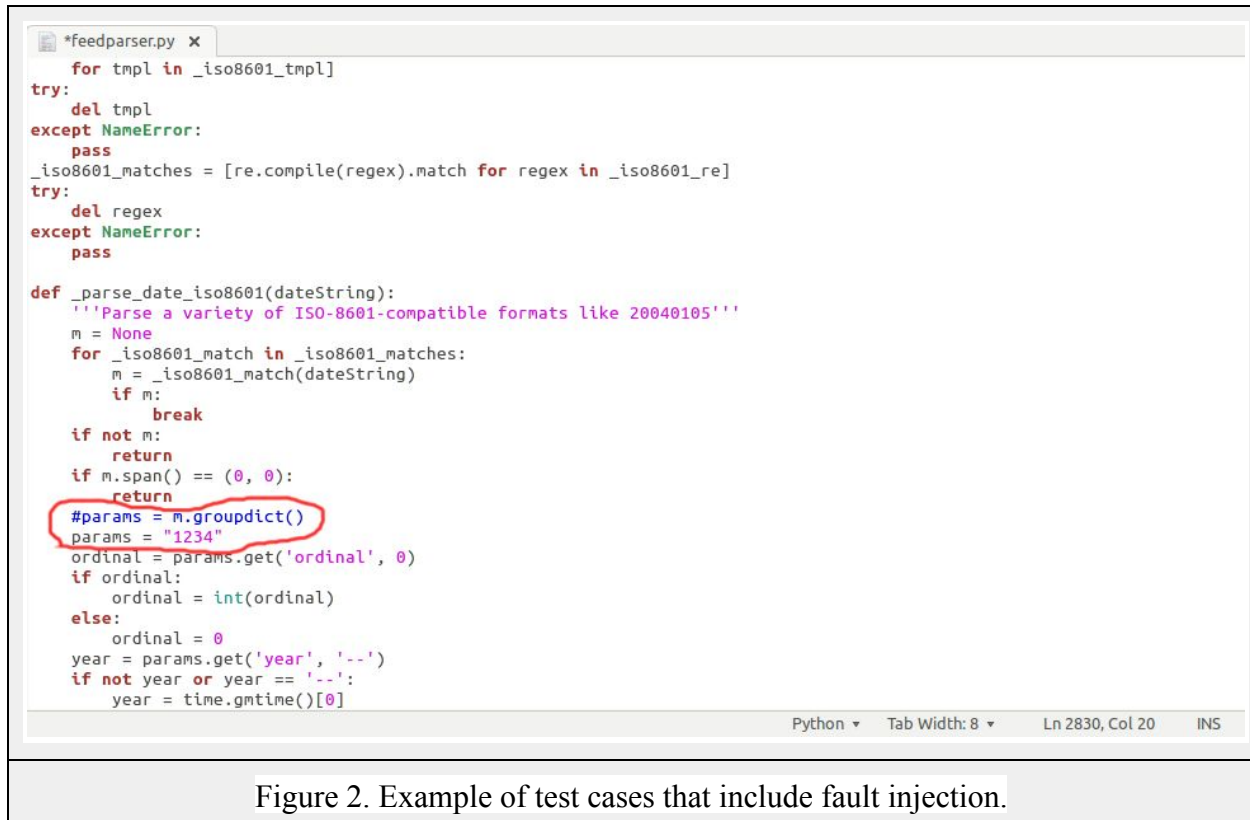


Figure 2. Example of test cases that include fault injection.

## **Building the Repository**

The previous chapters detailed in chronological order our methods of building this project. Now, we will talk about the final product in its design and functionality.

Our Github repository contains 6 directories:

- docs - contains:
  - Team6\_poster.pdf - the pdf version of a poster which is ready to be printed and presented
  - Team6\_powerpoint.pptx - the powerpoint presentation which will be presented in class Thursday, December 03.
- oracles - 25 .txt files containing:
  - The test name
  - The Description of the test
  - The expected output
- scripts - contains:
  - runAllTests.sh - The runner script that kicks off the testing framework and runs:



- The Python driver
  - The Web Report Builder
- temp - contains:
  - the Eden directory to combat any dependency issues
  - temporary .txt files used in reporting
- testCases - 25 .txt files containing:
  - The class name (including directory and subdirectory)
  - The method name
  - The input parameter
  - The oracle name
  - The driver for running the test
- testCasesExecutables - contains:
  - The python driver file
    - Used to parse the test cases, call the class and methods, pass in parameters, and collect results
  - The web reporting python build file
    - Parses the reporting document and builds a web page to display the results
    - Automatically opens a browser

## **Testing Framework**

The following is the basic functionality of our framework: Located in Team6/scripts/runAllTests.sh - When executed the script runs our Python Driver which compares both our test cases and oracles. The PC's default browser then opens and displays the test results in an HTML file.

All runAllTests.sh does is run: python ../testCasesExecutables/pythonDriver.py, then:  
python ../testCasesExecutables/webbuilder.py

The python driver does all the heavy lifting in the framework. It runs through each test case file, parses the information and writes it out to /temp/report.txt.

From here, the web builder pulls the information from report.txt and builds the web page.

## **Test Case Examples**

Test01:05 - Finds the GCD between two inputs

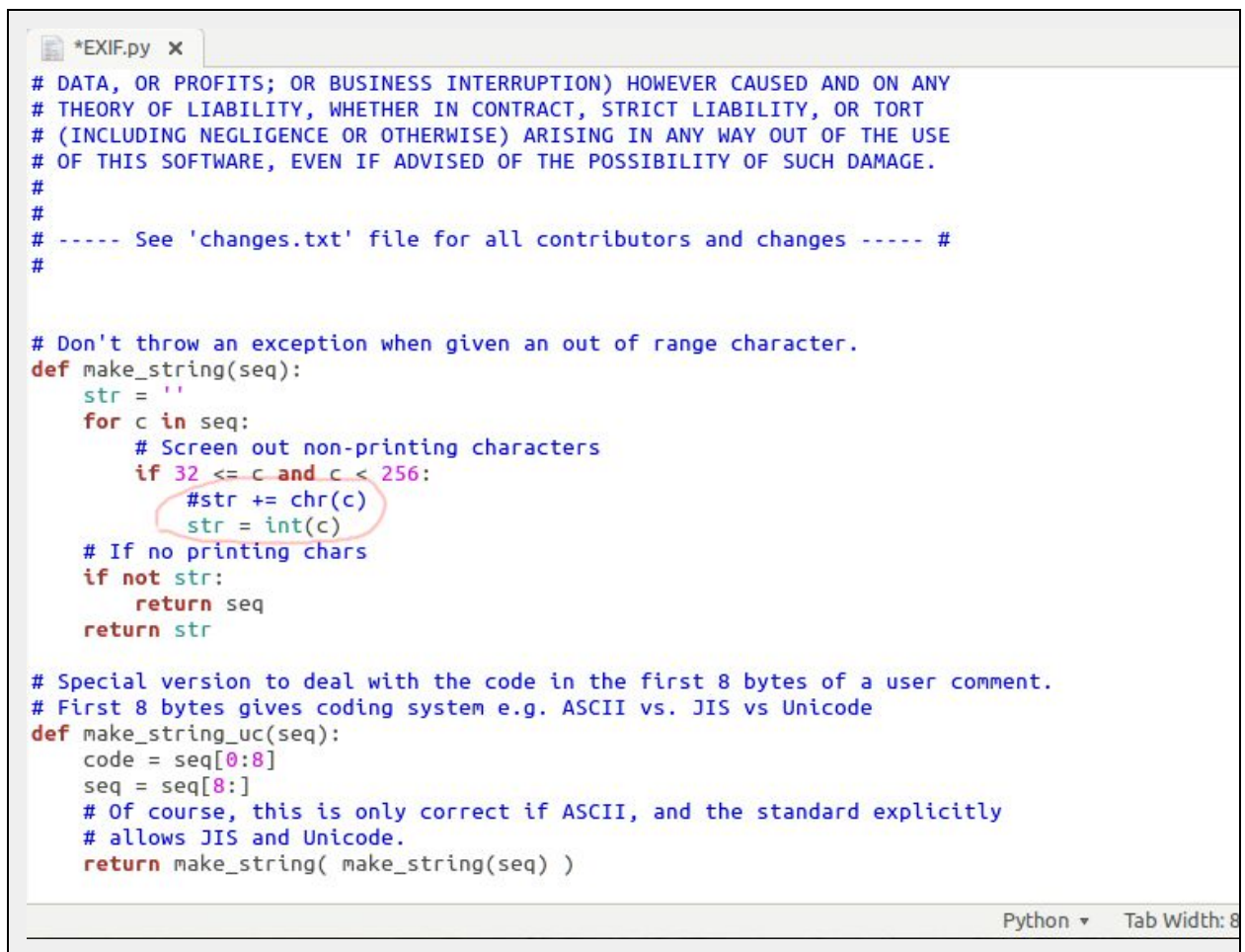
Test06:10 - Parses the date to send the information to another method to build a date

Test11:15 - Converts the input into a string

Test16:20 - Converts the input into an integer

Test21:25 Converts the input into a big endian integer

## Fault Injection

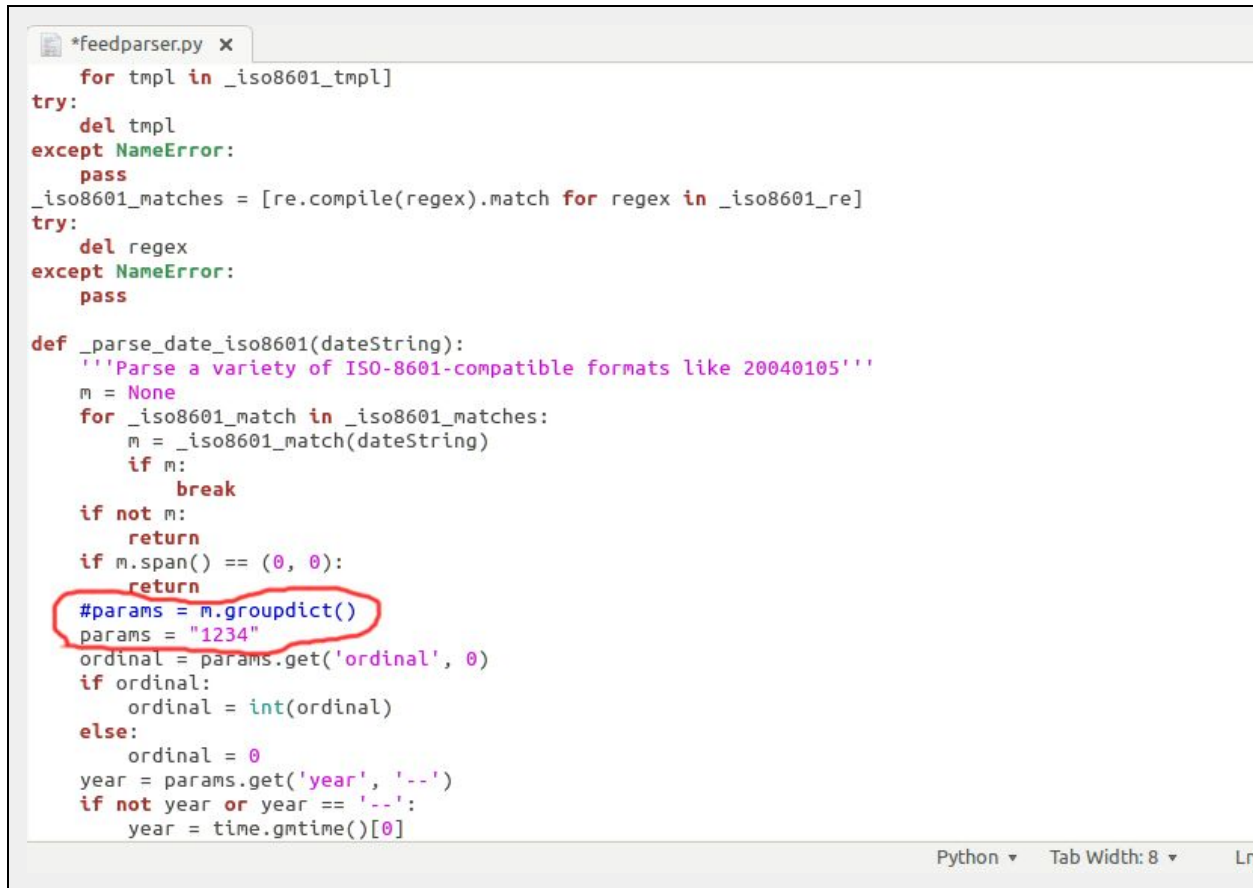


```
*EXIF.py x
# DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY
# THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT
# (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE
# OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
#
#
# ----- See 'changes.txt' file for all contributors and changes ----- #
#
# Don't throw an exception when given an out of range character.
def make_string(seq):
    str = ''
    for c in seq:
        # Screen out non-printing characters
        if 32 <= c and c < 256:
            #str += chr(c)
            str = int(c)
        # If no printing chars
        if not str:
            return seq
    return str

# Special version to deal with the code in the first 8 bytes of a user comment.
# First 8 bytes gives coding system e.g. ASCII vs. JIS vs Unicode
def make_string_uc(seq):
    code = seq[0:8]
    seq = seq[8:]
    # Of course, this is only correct if ASCII, and the standard explicitly
    # allows JIS and Unicode.
    return make_string( make_string(seq) )
```

Python ▾ Tab Width: 8

Figure 3: For test14 we commented out a method that placed a chr in to a variable named str and placed an int there instead. This allowed the method to complete but caused the test to fail due to incorrect code. See Figure 5



```
*feedparser.py x
    for tpl in _iso8601_tpl]
try:
    del tpl
except NameError:
    pass
_iso8601_matches = [re.compile(regex).match for regex in _iso8601_re]
try:
    del regex
except NameError:
    pass

def _parse_date_iso8601(dateString):
    '''Parse a variety of ISO-8601-compatible formats like 20040105'''
    m = None
    for _iso8601_match in _iso8601_matches:
        m = _iso8601_match(dateString)
        if m:
            break
    if not m:
        return
    if m.span() == (0, 0):
        return
    #params = m.groupdict()
    params = "1234"
    ordinal = params.get('ordinal', 0)
    if ordinal:
        ordinal = int(ordinal)
    else:
        ordinal = 0
    year = params.get('year', '--')
    if not year or year == '--':
        year = time.gmtime()[0]
```

Python ▾ Tab Width: 8 ▾ Ln

Figure 4: For test 10 we commented out a method that placed a specific string into params and instead replaced the string with “1234”. This allowed the method to run but still fail the test due to the oracle expected output and method output being different. See Figure 5

```

Class Name:modules EXIF
Method Name:make_string
Input:aaaaa
Oracle:oracle12.txt
Results: PASS

----Running Test: /home/steven/Desktop/Team6-master/testCases/Test11.txt----
Class Name:modules EXIF
Method Name:make_string
Input:12345
Oracle:oracle11.txt
Results: PASS

----Running Test: /home/steven/Desktop/Team6-master/testCases/Test10.txt----
Class Name:modules feedparser
Method Name:_parse_date_iso8601
Input:99:53:00+05:00
Oracle:oracle10.txt
Results: FAIL

----Running Test: /home/steven/Desktop/Team6-master/testCases/Test20.txt----
Class Name:modules EXIF
Method Name:s2n_motorola
Input:L
Oracle:oracle20.txt
Results: PASS

----Running Test: /home/steven/Desktop/Team6-master/testCases/Test14.txt----
Class Name:modules EXIF
Method Name:make_string
Input:123abc
Oracle:oracle14.txt
Results: FAIL

----Running Test: /home/steven/Desktop/Team6-master/testCases/Test18.txt----
Class Name:modules EXIF
Method Name:s2n_motorola
Input:8
Oracle:oracle18.txt
Results: PASS

----Running Test: /home/steven/Desktop/Team6-master/testCases/Test06.txt----
Class Name:modules feedparser
Method Name:_parse_date_iso8601
Input:201540105
Oracle:oracle06.txt

```

Fig 5: Here you can see the test for the method `_parse_date_iso8601(dateString)` and `make_string` have both failed due to our fault injection in the code.

## **Conclusion**

Our group learned a lot about the challenges of working in a diverse group. Everyone has conflicting schedules and varying priorities. In the end though, our team came together and worked better than a well oiled machine. Other challenges we faced included: repository clashes, different operating installations leading to varying file structures, updating software, learning new languages. In the end though we learned how important the testing process really is.

Repository clashes are one of those things that is bound to happen when more than one person is working on the same code. The key we learned is good communication so that only one person is debugging a section of code instead of multiple people. Different OS installations also lead to some challenges. Linux, being the OS that it is, allows you to install what you want where you want. This lead to initial issues when it came to our framework finding the installation

directory for our HFOSS project. When we figured out a good work around, our testing framework came together very nicely.

Another thing we learned is, keep your software updated. We had one person writing code in a lower revision level environment that phased out built in methods. This cause crazy errors when the higher revision level environments tried to run the code. That in turn lead to needless debugging.

Our final challenge was having to remember Python, learn HTML5, learn Shell scripting, and learning how to run the Sahana Eden software. I think that most of us can agree that, while Python is a powerful language, it's simplicity is its biggest downfall. Shell scripting on the other hand is a really fun and powerful way to automate lots of things within a Linux OS.

In conclusion, we all got an intimate look into what our post graduation life is going to be like. We learned that testing is both an important and difficult task and that there is in no way be all end all multi use testing framework. Each one must be individually tailored to the system they run on. However, the principles we learned have strengthened our understanding of what a good testing framework looks like.

#### Works Cited

1. <http://sahanafoundation.org/about-us/deployments/>
2. <http://eden.sahanafoundation.org/>