Josh Gilley, Alex Wizes, Clae Wyckoff

CSCI 362 - Software Engineering

Professor Bowring

11/2/20

Chapter 3

Designing and Building our Testing Framework

To say that deliverable 3 encompasses the lion's share of the work for our project would be an understatement, and this is something the Another-One-Bytes-the-Dust team recognized immediately. We quickly allocated responsibilities and laid out a schedule to hammer out designing and building the script as well as our initial driver. Despite some hurdles in the implementation process, sticking to our schedule greatly aided us in working diligently and efficiently to achieve not just results but outright success. Over the course of this chapter, we will discuss designing and building our initial driver and script before moving on to some of the hurdles we faced and some of the innovative solutions we found, then finally we will give a brief reflection on our experiences thus far as a whole.

Building the Driver

Considering the script partially relies on output from the drivers to function properly, we decided to tackle the initial driver before the script. Josh and Clae set to work designing and implementing the driver while Alex chipped away at the script. Designing our driver was fairly simple. We simply imported the proper files, decoded our json test case files into variables we could work with (i.e. variables that contain all of the attributes of the test cases such as ID number, method tested, testing inputs, etc.), and set to work creating the array the driver will

return as a long string. Next, we set the attributes for each test case in the return array according to the attributes in the corresponding spot in the array containing our data on the test cases. At this point, we actually call our test method parse_charset (seen in **Figure 3.1**), which converts text from uppercase to lowercase and formats according to various rules. We then compare the actual output from the test method with the expected output from our array of test cases and pass the result of each comparison to our return array for each test case. At this point, we essentially return the array of test case data (now with the actual output and a pass/fail variable) from our driver to our script

Figure 3.1

```
130     public static function parse_charset($charset) {
131         $charset = strtolower($charset);
132 
133         // shortcuts so that we do not have to load typo3 on every page
134 
135         if ($charset === 'utf8' or $charset === 'utf-8') {
136             return 'utf-8';
137         }
138 
139         if (preg_match('/^(cp|win|windows)-?(12[0-9]{2})$/', $charset, $matches)) {
140             return 'windows-'.$matches[2];
141         }
142 
143         if (preg_match('/^iso-8859-[0-9]+$/', $charset, $matches)) {
144 
145             return $charset;
146         }
147 
148         if ($charset === 'euc-jp') {
149             return 'euc-jp';
150         }
151         if ($charset === 'iso-2022-jp') {
152             return 'iso-2022-jp';
153         }
154         if ($charset === 'shift-jis' or $charset === 'shift_jis') {
155             return 'shift_jis';
156         }
157         if ($charset === 'gb2312') {
158             return 'gb2312';
159         }
160         if ($charset === 'gb18030') {
161             return 'gb18030';
162         }
163 
164         // fallback to typo3
165         return self::typo3()->parse_charset($charset);
166     }
```

One important step we took to build our driver properly was constructing our own class, which can be seen in **Figure 3.2**. This class is just a simple object called driverObject that essentially contains all of the information of a test case (including test case number, driver name, testing inputs, etc.). We also included getter and setter methods for each of the variables included in the object. In our code for the driver we use a number of these objects in an array, one object per test case, and we set each variable for each object according to the info from our json files, the output from our test method, and whether that output matches the expected output. Then we build a long string from all of the data in our array of objects and return that for the script to work with.

Figure 3.2

```
class driverObject {
        var $tcID;
        var $requirement;
        var $driver;
        var $class;
        var $method;
        var $testingInput;
        var $expectedOutput;
        var $success;

        function settcID($id){
                $this->tcID = $id;
        }
        function gettcID(){
                echo $this->tcID."$$$";
        }
        function setRequirement($id){
```

While this seemed simple, initially our driver was much more linear and inefficient. In this previous design the construction of the arrays and process of comparing outputs and returning the long string were very similar. The primary difference is that we didn't have a loop implemented in the older design, so essentially we had a block of code devoted to each test case. Maintaining this code felt like it would be needlessly arduous and inefficient, so we felt the need to redo it to a small extent. The result of this redesign is the current driver, now with a loop that will loop through each test case and do the work, rather than 5 individual blocks of code that each run a very similar test. At the end of the day, we're happy with our code which can be seen in **Figure 3.3**.

Figure 3.3

```php
<?php
//error_reporting(0);
//TEST STRING TO LOWER & REGULAR EXPRESSIONS

define('CLI_SCRIPT', true);
require("driverObject.php");
require_once("../project/moodle1/config.php");
require("../project/moodle1/user/lib.php");
require("../project/moodle1/lib/classes/text.php");

// Create Object of our Selected "Core Text" Class
$text = new core_text;

// Decode Json Objects from Test Case Folder
$tc01 = json_decode(file_get_contents("../testCases/TC01.json"));
$tc02 = json_decode(file_get_contents("../testCases/TC02.json"));
$tc03 = json_decode(file_get_contents("../testCases/TC03.json"));
$tc04 = json_decode(file_get_contents("../testCases/TC04.json"));
$tc05 = json_decode(file_get_contents("../testCases/TC05.json"));
$obj = new driverObject;

// Fill Array with Empty Objects for Test Case Info
$finalArr = array($obj, $obj, $obj, $obj, $obj);

// Fill Array of Test Case Decode Json
$testCaseArr = array($tc01, $tc02, $tc03, $tc04, $tc05);

```

```
for ($i = 0; $i < count($finalArr); $i++){

        $input = $testCaseArr[$i]->testingInputs;
        $expectedOut = $testCaseArr[$i]->expectedOutput;

        //Establish Object
        $finalArr[$i]->settcID($testCaseArr[$i]->testId);
        $finalArr[$i]->setRequirement($testCaseArr[$i]->requirement);
        $finalArr[$i]->setDriver($testCaseArr[$i]->driver);
        $finalArr[$i]->setClass($t
<p>Requirements: %s</p>estCaseArr[$i]->classTested);
        $finalArr[$i]->setMethod($testCaseArr[$i]->methodTested);
        $finalArr[$i]->setTestingInput($testCaseArr[$i]->testingInputs);
        $finalArr[$i]->setExpectedOutput($testCaseArr[$i]->expectedOutput);


        $parsed_charset = $text->parse_charset($input);

        if ($parsed_charset == $expectedOut){
                $finalArr[$i]->setSuccess("PASS");
        } else{
                $finalArr[$i]->setSuccess("FAIL");
        }

        $finalArr[$i]->gettcID();
        $finalArr[$i]->getRequirement();
        $finalArr[$i]->getDriver();
        $finalArr[$i]->getClass();
        $finalArr[$i]->getMethod();
        $finalArr[$i]->getTestingInput();
        echo $parsed_charset."$$$";
        $finalArr[$i]->getExpectedOutput();
        $finalArr[$i]->getSuccess();
}
```

Building the Script

While the driver was being worked on by Josh and Clae, Alex put her efforts into the script. Despite the fact that we'd spent some of our spare time working on the script previously, we knew it would be difficult to fully implement as it's the piece that unites all of the parts of our automated testing framework. And sure enough, it was one of the more challenging components of our project up to this point, but we'll discuss some of those challenges in the next section. Alex made considerable progress nonetheless, and thankfully she received some help as soon as the driver was running perfectly.

After the necessary import statements, our python script starts by running our driver as a php subprocess and getting a long string-like object (seen in **Figure 3.4**) from our driver that

contains all of our data for each of our 5 test cases. Our script then creates and formats the html file that will display a test report in a browser (which can be seen in **Figure 3.5**). It's at this point that our script converts the string-like object to a string, splits it by test case, then splits it by each attribute within each test case. This enables us to write each attribute for each test case to the formatted html file so we can then close and display it in a browser. While seemingly a lot of code, most of the heavy lifting comes from a small portion which can be seen in the loop in **Figure 3.6**.

Figure 3.4

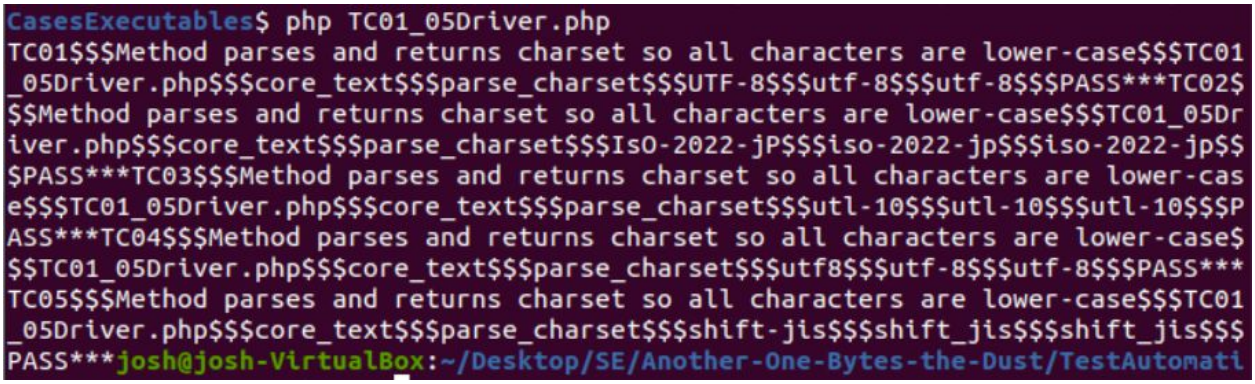
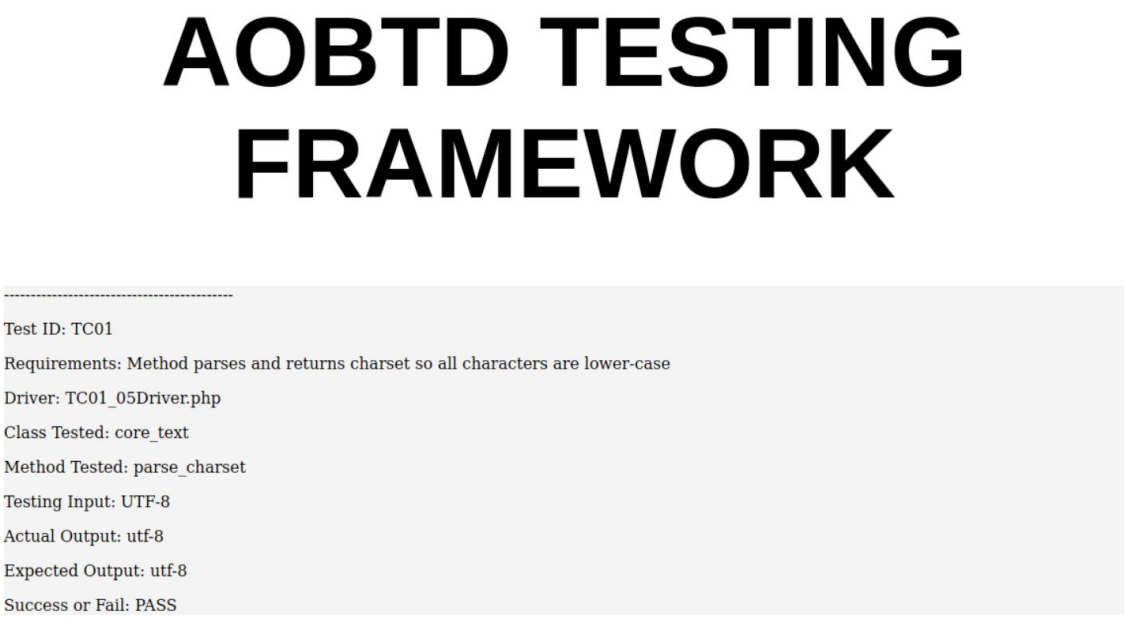


Figure 3.5



```
---------------------------------------------

Test ID: TC02

Requirements: Method parses and returns charset so all characters are lower-case

Driver: TC01_05Driver.php

Class Tested: core_text

Method Tested: parse_charset

Testing Input: IsO-2022-jP

Actual Output: iso-2022-jp

Expected Output: iso-2022-jp

Success or Fail: PASS

---------------------------------------------
```

Figure 3.6

```
#! /usr/bin/python3.8

import sys
import subprocess
import webbrowser
import time

result = subprocess.run(
        ['php', '../testCasesExecutables/TC01_05Driver.php'],
        stdout=subprocess.PIPE,
        check=True
)

htmlOpening = '''
<!DOCTYPE html>
<head>
<style>
.header{
        text-align: center;
        font-size: 100px;
        font-family: Arial, Helvetica, sans-serif;

}
.container{
        width: 90%;
        margin-left: auto;
        margin-right: auto;
        height: 380px;
        background-color: #f2f2f2;

</style>
<meta charset="utf-8">
<title>Test Report</title> </head>
<body>
<h1 class="header">AOBTD TESTING FRAMEWORK</h1>
f = open("testReport.html", "w")
f.write(htmlOpening)


x = str(result.stdout)
y = x.split('***')
#print(y)
outstring = ""
for case in y:
        attr = case.split('$$$')
        if (len(attr) == 9):
                whole = innerText % (attr[0].replace("b'", ""), attr[1], attr[2], attr[3], attr[4], attr[5], attr[6], attr[7], attr[8])
                f.write(whole)

f.write(htmlClosing)
f.close()
webbrowser.open("testReport.html")
```

Text Editor

Hurdles Faced

As with any project, we faced a number of hurdles we had to overcome. Aside from the typical struggles with technology/computers and simple errors like syntax errors, we faced two real challenges during this part of our project. The first involved a config file that we didn't realize had to be tailored to each individual user. The other issue was actually related to this issue, though is a different issue altogether involving the same file.

Our initial issue was relatively easy to solve. Having configured the script in a way that it should at least run, two of us were receiving a rather obscure error message about a file named config.php while the third group member's code was running perfectly. After some investigating, we realized that all three of us had our config.php files set with the Moodle database details for the group member whose code was working. After correcting these discrepancies by filling the config.php file with our own Moodle database details, our code ran without any issue, yet this wasn't the end of the issues we faced.

The other significant issue came with the resolution of our previous dilemma. Having personalized our config.php files, we realized that pushing our updates to our github repo and pulling them will write over each other's config.php files, thus necessitating us to perform the same fix from the previous issue each time we pull from our repo. To solve this, it was decided that we include our config.php file into a .gitIgnore folder so that github will ignore the config.php file when we push to and pull from our repo. These were the biggest issues we faced over the course of this deliverable, and thankfully we seem to have a handle on them.

Conclusion/Reflection

All in all this deliverable comprises a huge portion of the work required to complete this project, and it feels good to have delivered what we feel is a strong performance, at least before

any constructive criticism. Having worked through our issues with our initial driver and having created a functional version of our script, we now have a roadmap on how to implement the other drivers and simply need to adjust the script slightly so it can handle multiple drivers. At this point, we are feeling not just comfortable with where we are, but confident that we can quickly complete deliverable 4.