**SIT323 Cloud Application Development**

**Trimester 2, 2018**

**Assessment Task 1 – Programming Task 1**



Design Folio by Connor Vernon

Partner: Richardo Wijaya

216150769

**Contents**

1. Introduction 3
2. Test Scenario 1 3
3. Test Scenario 2 4
4. Test Scenario 3 6
5. Test Scenario 4 8
6. Test Scenario 5 10
7. Test Scenario 6 13
8. Test Scenario 7 15
9. Test Scenario 8 18
10. Test Scenario 9 20
11. Test Scenario 10 23
12. Test Scenario 11 25
13. Test Scenario 12 27
14. **Introduction**

Unit testing involves conducting tests on individual methods within a program to investigate if the expected value matches that of the actual value. Unit testing is light weight as it isolates itself from the main program code which makes it extremely fast to test a whole range of values. In this case, Unit testing is conducted on a Crozzle application that validates the files it opens. It also creates error logs for the user so that the error can easily be identified. In this crozzle application, Twelve methods are to be tested throughout the program which range from checking that the value of the crozzle properties are the same within the crozzle files, analysing that the correct values can be retrieved from new changes and see that the expected error message is displayed when there is a program breaking change.

1. **Test Scenario 1**
   1. **Justifications**
      1. **Test Case 1.1**

This test involved checking if the string value passed into the method could be interpreted as a Boolean value. In this scenario, “true” was passed through the parameters of the “isBoolean” function with the expected result being true. In the configuration file, there was only one Boolean value that acted as a toggle for displaying crozzle values for upper or lower case, so it was important to run test that Boolean variables can be read from the text file. As both the expected and actual result was true, this indicated that the unit test passed and that the function can correctly works.

* + 1. **Test Case 1.2**

Like test case 1.1, it involved checking that the string passed was interpreted as a string except that the string value passed has been deliberately made invalid. The parameter passed was “falsee” with the extra “e” placed to see if the program can successfully return false value. The reason for doing this test was to see that the crozzle application can detect an incorrect Boolean value. Both the actual and expected result returned false meaning that the unit test passed.

* 1. **Learning**

What I learned from conducting tests on the isBoolean function is that the TryParse method is used to convert the string values to Boolean Values. From doing test case 1.1, It gave me a better understand on how to read certain values from a text file and this can become useful when I want to create my own programs that take true and false values from a text file. By implementing test case 1.2, I learnt that the TryParse method will return false hence indicating that the value could not be converted into a Boolean value.

* 1. **Test Scenario Form**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Test Scenario ID** | **Test Description** | | | | |
| 1 | Check that the strings are correctly interpreted as a Boolean value. If the expected and actual return values are true, the method works correctly to identify Boolean values. Expected and actual false returns mean that it can identify non-Boolean. | | | | |
| **Test Method** | **Method Tested** | | | | |
| UnitTest1.TestMethod1() | Boolean Validator.isBoolean(String field, out Boolean aBoolean) | | | | |
| **Test Case ID** | **Parameters** | **Expected Data** | **Actual Data** | **Test Result** | **Test Comments** |
| 1.1 | field = “false” | expectedReturn1 = true  No errors | actualReturn1=true  No Errors | passed | “false” is a proper Boolean variable |
| 1.2 | field =”falsee” | expectedReturn2 = false  error message = [0] | ActualReturn2=false  error = [0] | passed | Program returns false due to invalid Boolean value |

**Data references:**

[0] Error message = “not a Boolean”

**Test case 1.1 and 1.2 files:**

No files needed

1. **Test Scenario 2**
   1. **Justifications**
      1. **Test Case 1.1**

Conducting this test involved investigating if the string value passed can be correctly interpreted as an integer value. In this this, the string value “25’ was passed through the isInt32 function with the expected result being True. The results from conducting this test was that the expected and actual result were true indicating that the test has passed. It was a crucial test as many of the crozzle files contain integer values for configurations such as max number and minimum number of rows for controlling how display of the crozzle. If the unit test did not pass, it would mean that integer values will not be able to be read from the crozzle files.

* + 1. **Test Case 1.2**

In contrast to the previous test, the string passed was in the form of a double or floating point value. Doing this test was crucial to see if floating or double values can be used in the text file. As the TryParse method was specifically named “isInt32”, the expected value for the test was false. After conducting the unit test, both the expected and actual result came back as false indicating the unit test has passed and that the function can’t take decimal values.

* + 1. **Test Case 1.3**

Finally, this test case involved checking that the string value passed in into the isInt32() function can successfully convert the value if it is in the form of a negative number. The reason for testing this was to see if negative values can be retrieved from the crozzle file. This would be a particular test to consider as it wouldn’t make sense to have max rows being negative values. For the value passed, it was “-25” with the expected result being true as “-25” is classed as a negative integer. The results were that both the expected result and the actual result were true indicating a passed unit test and the function allowing negative values.

* 1. **Learning**

Similar to the isBoolean() function, the TryParse method has opened new doors for me on how to strictly read integer variables from text files. From testing 1.2, I have learnt that the isInt32() function strictly takes whole numbers and this is particular useful as you can’t have rows that are for example 10.5. Test case 1.3 was particularly interesting as I found out that the TryParse returns true for negative values. Examining that abnormal behaviour had given me ideas that the integer tryParse could have a range implemented also before returning true or false so that negative values don’t get allowed.

* 1. **Test Scenario Form**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Test Scenario ID** | **Test Description** | | | | |
| 2 | Check that the strings are correctly interpreted as integer values. If the expected and actual return values are true, the method works correctly to identify Integer values. Expected and actual false returns mean that it can identify non-Integers. | | | | |
| **Test Method** | **Method Tested** | | | | |
| UnitTest1.TestMethod2() | Boolean Validator.isInt32(String field, out int anInteger) | | | | |
| **Test Case ID** | **Parameters** | **Expected Data** | **Actual Data** | **Test Result** | **Test Comments** |
| 2.1 | field = “25” | expectedReturn1 = true  no errors | actualReturn1=true  no errors | Passed | True as the string value can be interpreted as integer |
| 2.2 | field = “25.32” | expectedReturn2 = false  error = [0] | actualReturn2 = false  error = [0] | Passed | Found that floating point numbers cannot be read |
| 2.3 | field = “-25” | expectedReturn3 = true  no errors | actualReturn3=true  no errors | Passed | True as the string value can be interpreted as integer |

**Data references:**

[0] Error message = “not an Integer”

**Test case 2.1, 2.2 and 2.3 files:**

No files needed

1. **Test Scenario 3**
   1. **Justifications**
      1. **Test Case 1.1**

Initiating this test involved investigating if the string passed into the isHexColourCode() function can be interpreted as a hex colour value. For this case scenario, “#2eec71” was passed into the function with the expected value to be true. It’s important to conduct this test as the Crozzle configuration files takes hex values for drawing the crozzle graphics such as the “BGCOLOUR\_EMPTY\_TD” setting for determining what colour the empty tiles should be. For the result, the expected value matched that of the actual value hence the test passed and the application is able to read the hex value successfully.

* + 1. **Test Case 1.2**

Although the program can read hex values, it is important to also check that invalid hex values can be identified in the program. The value “##ecc71” is passed through the function with the expected value of false due to the second character being illegal and not meeting hex value conditions. The result of the test was the expected value matched that of the actual value which means the test passed at detecting an invalid hex value.

* + 1. **Test Case 1.3**

The final test case conducted on the function is similar to that of 1.2. The difference is that the value of “#2ecc7g” is checked to see if all the values after the hash are within a range specified by a Regex expression. The expected value from passing the string will be false as the letter “g” is not a valid character as letters within regex range from “a” to “f”. As a result, the expected value was the same as the actual value hence the test passed and was able to detect the character that didn’t mean the expression.

* 1. **Learning**

From conducting these tests, I learnt how hex values are structured and that a Regex expression is an efficient way of checking for a valid hex value. Being able to do unit test around regex expression has expanded my knowledge in that sector of coding such as setting the required number of characters in a value using curly braces({6}) and anything that is between “^” and “$” means that the value has to meet the conditions inside. I further expanded my knowledge on validation by conducting 1.2 and 1.3 which had me explore on the Regex symbols used for structuring required data/conditions. For the future, I can implement regex expressions for data validation which outweighs my previous knowledge. It is a short and efficient way of validating values compared to using countless IF statements.

* 1. **Test Scenario Form**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Test Scenario ID** | **Test Description** | | | | |
| 3 | Check that the strings are correctly interpreted as colour hex values. If the expected and actual return values are true, the method works correctly to identify Hex values. Expected and actual false returns mean that it can identify illegal characters | | | | |
| **Test Method** | **Method Tested** | | | | |
| UnitTest1.TestMethod3() | Boolean Validator.isHexColourCode(String hexColour) | | | | |
| **Test Case ID** | **Parameters** | **Expected Data** | **Actual Data** | **Test Result** | **Test Comments** |
| 3.1 | hexColour = “#2ecc71” | expectedReturn1 = true  No errors | actualReturn1 = true  No errors | Passed | The colour Green. Passed as it met the requirements of REGEX |
| 3.2 | hexColour = “##ecc71” | expectedReturn2 = false  Error = [0] | actualReturn2 = false  Error = [0] | Passed | False due to illegal symbol after the initial hash tag |
| 3.3 | hexColour = “#2ecc7g” | expectedReturn3 = false  Error = [0] | actualReturn3 = false  Error = [0] | Passed | False due to a character being out of range |

**Data references:**

[0] Error message = “not a Hex colour code”

**Test case 3.1, 3.2 and 3.3 files:**

No files needed

1. **Test Scenario 4**
   1. **Justifications**
      1. **Test Case 1.1**

This test case involved observing the string value passed into the KeyValue.TryParse function can be interpreted as a key-value pair and meets the conditions of the selected Regex expression. For this case, the method takes two parameters which are “MAXIMUM = 100” for the example Key value data and “@’ [a-zA-Z0-9]’ ” for the example key pattern. The reason for using “MAXIMUM = 100” for the key value data is because the configuration files contains that key value pair and testing the string passed will help investigate if it is identified to be a key value pair. The expected result for this test case is true as there are two values on each side of the “=” symbol and meets the condition of the function that it contains a “=” symbol. . The results of the test case was that the expected and actual return results both came out true therefore the test passed.

* + 1. **Test Case 1.2**

Similar to test case 1.1, the regex key pattern will remain the same but the key value data passed into the method has been modified to “MAXIMUM =”. The reason for conducting this test case is to see if the method can detect that the string value passed does not meet the requirements of a key value pair. This is crucial for the Crozzle application so that it doesn’t take empty data that may crash the application. The expected result for this test case is false as it doesn’t meet the array length of 2. The results from this unit test is that the expected and actual results were the same hence the test passed for detecting invalid key value pairs.

* + 1. **Test Case 1.3**

The final test case conducted was bizarre as it involved using the same key value from the 1.2 but adding a space after the “=” symbol. The reason for testing this scenario was to examine if the string passed could be perceived even though one of the values were just a whitespace. Like 1.2, the crozzle application shouldn’t take empty values as it can cause problems to the application. The expected value of the test was true as strings can have empty values and after conducting the test, it was found that the expected matched that of the actual value meaning that the test successfully passed. What happens if we don’t specify the maximum number of duplicate words? Can whitespace be interpreted as null or zero? The results from this scenario have raised many questions.

* 1. **Learning**

Examining the test results from test case 1.1 have expanded my knowledge on how you can set a format standard on text files for the particular application. As for unit test 1.2 and 1.3, it made me aware of extra conditions that can be implemented into the KeyValue function. Test case 1.2 was logical in that it was bound to fail from my analysis of the method. I had learned that a constant integer variable called “KeyValueLength” had been set to 2. When the string passed was split and put into an array, there was only the “MAXIMUM” keyword as no value had been specified after the “=” symbol. Comparing this to test case 1.3, the results came back as a valid key value when whitespace was added after the “=” symbol. This had me thinking that a condition could be implemented into the Key Value method to discard a value that is only white space. Besides the significance found between 1.2 and 1.3, it broadened my knowledge on how to split key values into two separate values which comes useful when I make readable and meaningful configuration files.

* 1. **Test Scenario Form**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Test Scenario ID** | **Test Description** | | | | |
| 4 | Checking that the string value passed matches a key-value pair. . If the expected and actual return values are true, the method works correctly to identify keyvalue pairs and matches the key Pattern specified. Expected and Actual values that equal false mean that the function can correctly identify an invalid key value pair. | | | | |
| **Test Method** | **Method Tested** | | | | |
| UnitTest1.TestMethod4() | Boolean KeyValue.TryParse(String originalKeyValueData, String keyPattern, out KeyValue aKeyValue) | | | | |
| **Test Case ID** | **Parameters** | **Expected Data** | **Actual Data** | **Test Result** | **Test Comments** |
| 4.1 | originalKeyValueData = "MAXIMUM = 100"  KeyPattern = [0] | expectedReturn1 = true | actualReturn1 = true | Passed | True because String passed has a equals sign |
| 4.2 | originalKeyValueData = "MAXIMUM="  KeyPattern = [0] | expectedReturn2 = false | actualReturn2 = false | Passed | False due to no value given for one of the pairs |
| 4.3 | originalKeyValueData = "MAXIMUM= "  KeyPattern = [0] | expectedReturn3 = true | actualReturn3 = true | Passed | Whitespace added after “=”. |

**Data references:**

[0] = @"[a-zA-Z0-9]"

**Test case 4.1, 4.2 and 4.3 files:**

No files needed

1. **Test Scenario 5**
   1. **Justifications**
      1. **Test Case 1.1**

This test case involved analysing the return result of the Score() function based off the original crozzle files and making sure that the expected score matched that of the actual score that was calculated from the function. When running the crozzle application, the score after validation is returned as “303” in a string format. Unlike the other unit tests, this function took no parameters so the return result was entirely based of what was in the configuration files. After running the test, the expected value matched that of the actual value meaning that the test passed and was able to produce the correct score as the files were error free. The reason for conducting this unit test was to ensure that the correct score was outputted and could be displayed if no errors were present.

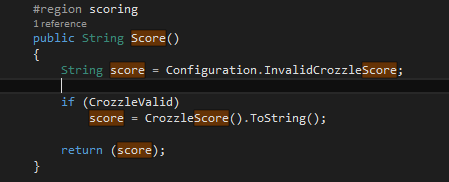
* + 1. **Test Case 1.2**

This next test case involved making illegal changes to the Crozzle files so that errors are prone in the application. Testing can then be done to see if a score is displayed or not. The changes made to the Crozzle file was having “ROW=10,RONALD,6” removed. The Score() function was then calculated using the new crozzle file. After debugging the application with the new Crozzle file, the return value was “INVALID CROZZLE” due to two errors which were “error 1: code 11002: the number of horizontal words intersecting AL is 0, but this is not within [1, 100]” and “error 2: code 11005: the number of groups of connected words is 2, but this is not within [1, 1]”. Since we know errors are present in the Crozzle application, the expected value is “INVALID CROZZLE”. After running the test, the expected value matched that of the actual value from the Crozzle application therefore the test passed and the Crozzle application was able to properly output an error message of “INVALID CROZZLE” when error were present. The importance of the test was to ensure that a score would not be shown if there were error messages.

* + 1. **Test Case 1.3**

Finally, this test case involved extending two names in the Crozzle file so that a test can be conducted to see if the new score is different to that of test case 1.1. The rows and columns edited in the original Crozzle file were “ROW=9, JACK,1” and “COLUMN=7,TOM,1”. These were changed by adding extra letters to the end of their names. The new names were “ROW=9, JACKIE,1” and “COLUMN=7,TOMORY,1”. Since there were extra letters to display for the crozzle application, it was expected that the new score was going to be larger than the original score. The expected score for the Crozzle would be “336” which was calculated by observing the configuration file where it had the settings for points of each intersecting letter. After running the test, the expected result matched the actual result meaning the Crozzle application can successfully return the correct score based off the rules in the configuration file.

* 1. **Learning**

Conducting tests on 1.1 had me understand the crozzle application better as it involved a lot of debugging since I could not find out why the Score function would return an empty string. After analysing the code, I had learned that the score was only generated if the “crozzleValid” Boolean value was set to true otherwise the Score() function wouldn’t get the actual score from the CrozzleScore() function. 

The solution was to execute a function in the unit test called Validate() which is responsible for determining if the “crozzleValid” variable will be true or false. I found out that the Validate function was responsible for telling the Score program if the crozzle files were valid by checking that the words meet the conditions of the config properties.

For test case 1.2, I learned how “INVALID CROZZLE” was displayed when there was an illegal error made to the Crozzle files. By removing the row that contained the name “RONALD” from the Crozzle file, it created problems as crozzle rules were broken within the application. I also had experience with looking at the log file output which stated what rules were broken in the application which makes it easy to find the error within text files. Overall, this test has made me realise the importance of good error messages as it makes debugging and fixing errors easy.  
  
When doing test case 1.3, there were many errors that were logged at the start and I found out that it was due to not having “JACKIE” and “TOMORY” in the wordlist file. It gave me a better understand on how the errors are generated in the Crozzle application. From 1.3, I also learned how the score was generated. Starting from “303”, changing “Jack “to “Jackie” incremented the score by 1 since the intersection at “e” meant that you get 1 point according to the configuration file. As Jackie connected to a new word, it incremented the score by 10 and I learned that words that connect to another word adds 10 points. This then made the total score “314”. Changing “Tom” and “Tomory” had the biggest change to the score as the letter “y” was an intersection. The amount of points rewarded for words intersecting at “y” was 32 therefore making the score total to “336”.

* 1. **Test Scenario Form**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Test Scenario ID** | **Test Description** | | | | |
| 5 | Making sure that the expected score matches the actual score generated when the crozzle application is validated. If the expected matches the actual value, the method can successfully generate the correct score. | | | | |
| **Test Method** | **Method Tested** | | | | |
| UnitTest1.TestMethod5() | String Crozzle.Score() | | | | |
| **Test Case ID** | **Parameters** | **Expected Data** | **Actual Data** | **Test Result** | **Test Comments** |
| 5.1 | n/a | expectedTrueScore = “303”  No errors | Crozzle.Score = “303”  No errors | Passed | True because the expected score matches actual score from the validation |
| 5.2 | n/a | expectedInvalidScore = “INVALID CROZZLE”  Errors = [0][1] | Crozzle2.Score = “INVALID CROZZLE”  Errors = [0][1] | Passed | A row has been removed from the first crozzle text file making the score invalid |
| 5.3 | n/a | expectedNewWordsScore = "336";  No errors | Crozzle3.Score() = “336”  No errors | Passed | Two new words added hence there is a higher score of 336 |

**Data references:**

[0] Error messages = error 1: code 11002: the number of horizontal words intersecting AL is 0, but this is not within [1, 100]

[1] Error messages = error 2: code 11005: the number of groups of connected words is 2, but this is not within [1, 1]

**Test case 5.1, 5.2 and 5.3 files:**

5.1 – “Test 1 Configurations.txt”, “Test 1 Wordlist.txt”, “Test 1 Crozzle.txt”

5.2 – “Test 1 Configurations.txt”, “Test 1 Wordlist.txt”, “Test 1 Crozzle\_invalidScore.txt”

5.3 – “Test 1 Configurations.txt”, “Test 1 Wordlist\_scoreNewWords.txt”, “Test 1 Crozzle\_scoreNewWords.txt”

1. **Test Scenario 6**
   1. **Justifications**
      1. **Test Case 1.1**

Investigating this test case involved checking the return value of the Boolean “ErrorsDetected” variable. If there were duplicate words that exceeded that of max duplicate count, it is expected that the CheckDuplicateWords() function would return true for errors. As the test is conducted on the original file, the expected value is false as the properties “MINIMUM\_NUMBER\_OF\_THE\_SAME\_WORD” and

“MAXIMUM\_NUMBER\_OF\_THE\_SAME\_WORD” in the configuration file both equal to 1. There are also no duplicates in the crozzle file meaning that the “ErrorsDetected” variable is still expected to be false. The result of conducting the test was the expected and actual value were both false meaning that the test passed. The reasoning of conducting this test is to ensure that errors are not logged when the crozzle files meet the conditions of true validity.

* + 1. **Test Case 1.2**

Similar to test case 1.1, this unit test involves making copies of the original files and editing them so that there is are two names of “RON” in the crozzle file. The configuration file has been changed so that the “MAXIMUM\_NUMBER\_OF\_THE\_SAME\_WORD” property has been set from 1 to 2. This should allow having 2 of the same name and it is expected that no errors will be generated since all the files meet the right conditions of validity. After running the tests, both the expected and actual result returned false meaning that the test successfully passed. The importance in conducting this unit test case is so we can see if the CheckDuplicateWords function successfully logs no errors for a valid change in the files and to see the flexibility of changing how the crozzle application behaves.

* + 1. **Test Case 1.3**

This final test case involves using the original configuration file where the Max and Min properties are set back to 1 but using the edited crozzle file that has the duplicate word. It is important to investigate whether the application successfully logs errors when there are duplicate words exceeding the maximum limit allowed so that we know the CheckDuplicateWords performs correctly. Since the number of duplicate words exceed that of the limits specified, the expected value of “ErrorsDetected” is true. After running the test, the expected and actual result matched meaning that the test passed. This is also meaning that the application was able to successfully log errors if duplicates words exceeded that of the limit specified.

* 1. **Learning**

Test case 1.1 had me playing around with the properties and seeing if I could change the original files to allow duplicate words which was a success. I learned the importance of conducting this test as it ensures that changing the properties of the configuration file to allow extra duplicates should not display an error. When conducting test case 1.2, I had found some interesting findings in the application. The parameters of the method had no effect on the method code or return result as it was never used. I have realised that a Boolean variable could be returned to indicate a duplicate error based off the passed parameters. After analysing the code and playing around with the configuration files, it was only the configuration properties that were used with the CheckDuplicateWords method for finding errors. Overall, conducting these tests broadened my skills in unit testing and design by not being restricted to checking properties of a file; but also seeing if any errors have been logged.

From completing the test case of 1.3, I had learnt that it is important to check that the function in the crozzle application can also successfully detect errors. The error message that was output to the log gave me the correct information on what was the cause of the crozzle being invalid. It was a meaningful message which stated that “RON” occurs twice but does not fit in the range of having a minimum of 1 duplicate as well as a maximum of 1. These were the exact settings I changed and if I was to fix the application to be valid, the error message has stated exactly what needs to be fixed. Like test unit 5 on the Score Method, I have learned that having meaningful error messages is helpful for quick debugging and that we should conduct tests to see that errors are only displayed when there is error.

* 1. **Test Scenario Form**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Test Scenario ID** | **Test Description** | | | | |
| 6 | Checking that the Crozzle application can successfully find duplicates within the Crozzle. If errors are detected, the application method can successfully log errors if the duplicate word count exceeds the max limit specified in the config files. | | | | |
| **Test Method** | **Method Tested** | | | | |
| UnitTest1.TestMethod6() | void Crozzle.CheckDuplicateWords(int lowerLimit, int upperLimit) | | | | |
| **Test Case ID** | **Parameters** | **Expected Data** | **Actual Data** | **Test Result** | **Test Comments** |
| 6.1 | lowerLimit = 1  upperLimit = 1 | expectedFalseErrors = false  no errors | crozzleSequences.ErrorsDetected = false  no errors | Passed | No errors can be detected in the original files |
| 6.2 | lowerLimit = 1  upperLimit = 2 | expectedFalseErrorsNew = false  no errors | crozzleSequences2.ErrorsDetected = false  no errors | Passed | No errors as two duplicates allowed in new file |
| 6.3 | lowerLimit = 1  upperLimit = 1 | expectedTrueErrorsNew = true  error = [0] | crozzleSequences3.ErrorsDetected = true  error = [0] | Passed | Errors returned as duplicates exceed max duplicates |

**Data references:**

[0] Error Messages = error 1: code 11003: the number of times RON occurs is 2, but this is not within [1, 1]

**Test case 6.1, 6.2 and 6.3 files:**

6.1 – “Test 1 Configurations.txt”, “Test 1 Wordlist.txt”, “Test 1 Crozzle.txt”

6.2 – “Test 1 Configurations\_checkDuplicates.txt”, “Test 1 Wordlist\_checkDuplicates.txt”, “Test 1 Crozzle\_checkDuplicates.txt”

6.3 – “Test 1 Configurations.txt”, “Test 1 Wordlist\_checkDuplicates.txt”, “Test 1 Crozzle\_checkDuplicates.txt”

1. **Test Scenario 7**
   1. **Justifications**
      1. **Test Case 1.1**

Initiating this test case involved seeing that the original files could be successfully validated by the Validate() function. In test case 5, this function was used to determine if a crozzle score can be returned or not based off the conditions and if any errors were detected. Although the program runs fines and calculates the correct score of “303”, it’s always ideal to debug and test the backend as we only see the graphics of the application. In this case, we are checking the Boolean value of the “CrozzleValid” variable. It is expected that this variable will be true as the application displays no errors. After running the tests, the expected value matched that of the actual value meaning that the test successfully passed on finding no errors within the program and allowing the score function to return a score.

* + 1. **Test Case 1.2**

Like unit test 5, there have been changes to the crozzle file where the word “RONALD” has been removed which leaves the word “AL” disconnected from the rest of the crozzle. The rules in the original configuration specify that the group of connected words has to have a minimum and maximum of 1 connected group and it must also have at least one word intersecting it. From these known settings in the configuration file, we can assume the expected value of the “CrozzleValid” variable to be false. The reason for conducting this test is to see that the Validate function can successfully detect these errors otherwise we wouldn’t know if the crozzle we are using is invalid. After conducting the test, the expected value matches that of the actual value meaning that the test has passed and that the Validate method has successfully been adding errors to a list.

* + 1. **Test Case 1.3**

For this final test, we are using the same files as 1.2 but testing a different aspect within the Validate function. So far, only a true and false value is indicating validity and appropriate error messages have not been checked. In this test case, we are investigating if the Validate function adds the right errors to the error list based off the errors within the crozzle files. We can assume the expected error message of the first error is “code 11002: the number of horizontal words intersecting AL is 0, but this is not within [1, 100]” which is an error message generated from the log file and code 1002 is the error code for Horizontal Intersection Error based off the CrozzleErrors.resx file. After conducting the test, the expected value matched the actual value meaning that the test passed. It was important to conduct this test to see that we are logging the appropriate error messages that match the broken conditions.

* 1. **Learning**

Like the previous test units, I have got a better understanding of the program when conducting test case 1.1. Conducting unit test on the Validate function helped me understand that the primary goal of the function is to get all word sequences and then check that the words match the conditions in the configuration file. There were many conditions I found in the code such as checking max limit on word total and that each word at least has one intersection. If there were any errors in the validate method, there would be an error added to an error list object. I also found out that the value of CrozzleValid is dependent on if there are errors in the list or not. Relating back to test unit 5, I could not get a score without the validate function because the Score is dependent on the CrozzleValid variable from the validate function.

Test cases 1.2 and 1.3 further expanded my knowledge on unit testing by making changes to the crozzle files to not meet the conditions of the configuration file. It made me understand the importance of not only checking that the right value can be returned, the right errors matching the rules broken need to be returned too. It makes sure that the application error checking is working fine. The broken conditions in the crozzle file were the same as that of the invalid crozzle files of the score. From the false return of the CrozzleValid variable and the expected result matching the actual result, I had investigated that the Validate Function has properly analysed crozzle files not meeting certain conditions. To further increase the evidence of test case 1.2, I wanted to challenge myself by checking that the expect error message matches an error message returned from the application. From the results of 1.3, it helped me understand that the program is logging the right errors. Test case 1.3 helped me come up with news way to explore unit testing by diving deeper to find the cause of a value changing from true to false.]

* 1. **Test Scenario Form**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Test Scenario ID** | **Test Description** | | | | |
| 7 | Confirming that the Crozzle Application can determine that the Crozzle displayed is valid and checking that the correct error message is returned if there are any errors present. | | | | |
| **Test Method** | **Method Tested** | | | | |
| UnitTest1.TestMethod7() | void Crozzle.Validate() | | | | |
| **Test Case ID** | **Parameters** | **Expected Data** | **Actual Data** | **Test Result** | **Test Comments** |
| 7.1 | n/a | expectedTrueValidityOriginal = true | crozzle.CrozzleValid = true | Passed | True as Crozzle files meet all conditions |
| 7.2 | n/a | expectedFalseValidityNew = false | crozzle2.CrozzleValid = false | Passed | False value as validate method detected errors |
| 7.3 | n/a | expectedErrorMessageNew = [0] | crozzle2.CrozzleGridErrors[0] = [0] | Passed | Correct error message matches that of the error in the program |

**Data references:**

[0] Error Messages = “code 11002: the number of horizontal words intersecting AL is 0, but this is not within [1, 100]"

**Test case 7.1, 7.2 and 7.3 files:**

7.1 – “Test 1 Configurations.txt”, “Test 1 Wordlist.txt”, “Test 1 Crozzle.txt”

7.2 – “Test 1 Configurations.txt”, “Test 1 Wordlist\_validateFalse.txt”, “Test 1 Crozzle\_validateFalse.txt”

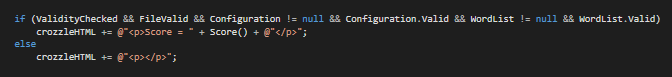
7.3 – “Test 1 Configurations.txt”, “Test 1 Wordlist\_validateFalse.txt”, “Test 1 Crozzle\_validateFalse.txt”

1. **Test Scenario 8**
   1. **Justifications**
      1. **Test Case 1.1**

Initiating this test case involved checking that the toStringHTML() function returned the correct HTML string value. In this case, we are running the validate function before saving and loading any HTML from other methods. Since the Validate function and run by the test before anything else, a score of “303” should be generated within the string HTML value since the files being tested are original and meeting all conditions of the configuration file. The reason for checking the HTML return value of the original files is to understand the structure of the HTML code that is rendering parts of the crozzle application. If there are 10 rows that we see in the crozzle application or crozzle files, we want to check that the function is return the right amount of HTML row tags. This test can help identify bugs and logical errors such as an extra row being printed onto the application. After running the tests, the expected value matches that of the actual value meaning that the test has passed. As seen in the results, we have also checked that the validate function has successfully added the HTML content for displaying the score to the crozzle application

* + 1. **Test Case 1.2**

In contrast to test case 1.1, the validate function in the test code will now by initiated after the saving and the loading of the HTML string values. Since the HTML content is stored in variables before it can be validated, a score has not been calculated therefore the expected and actual HTML string should not have any content about score. There is a conditional statement in the function that checks the Boolean value of variables.



As seen in the snippet above, the ValidityChecked variable should be false when the HTML contents were saved and loaded. As said before, this is because the validate function hasn’t checked the crozzle files. Without the ValidityCheck being true, we can assume the expected result will contain the content in the else statement. As a result, the expected and actual values match meaning that the test has passed on checking the HTML for an empty score. Conducting this test case ensures that the condition for checking the validity of crozzle files is working correctly and we now know that not using the Validate() function before loading and saving HTML data will not print a score to the crozzle application.

* + 1. **Test Case 1.3**

The process of this test case involves using the new crozzle files from unit test 5 where some of the words have additional letters. This means that if we run the function and the crozzle files are validated, we are expecting the HTML content for score to be higher than that of test case 1.1. Test Case 1.3 incorporates the newly edited crozzle files from unit test 5 therefore we are expecting the score in the HTML content to also be equal to “336”. After running the test, the expected value matched that of the actual value meaning that test successfully passed and the function can successfully return the string with the new score.

* 1. **Learning**

Conducting this test unit was the most difficult of the twelve investigated. The reason for this is because the return result of ToStringHTML was incredibly long so it was difficult to compare the expected value and the actual value during the debug. The changes that had to be were “Console.Write” was used instead of “Console.WriteLine” when saving the return file to a text file as this caused the expected string to have a (\r\n) added to the end of the string. The crozzle application also had trouble validating the configuration and wordlist files and the solution to this was to remove the”.\ “in the file path of the crozzle file. What I got out of this test scenario is that when there are long string values, the result can be saved to a text file as viewing the result in debug had poor visibility. For test case 1.1 and 1.2, I learned that the outcome of results change drastically depending on the position of the Validate() function. Placing the Validate() function after the expected and actual results were obtained meant that the Score was never calculated and therefore there was no score in the ToStringHTML() return value. Conducting test case 1.3 furthur pushed my unit testing knowledge by seeing if the new score can be outputted into the ToStringHTML() return result. 1.3 ensured that programmers know that the score contents in the HTML string will change based off the crozzle files. Overall, it helped me figure out that unit tests can be conducted by just changing the position of one of the methods in the unit tests as for this test unit, the output of the HTML string relied on the validation method.

* 1. **Test Scenario Form**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Test Scenario ID** | **Test Description** | | | | |
| 8 | Confirming that the method returns the correct string value when the validate function has been initiated or not. The string value is dependent on different settings in each file. If the expected value matches the actual value, the application returns the correct results based off changes such as different scores. | | | | |
| **Test Method** | **Method Tested** | | | | |
| UnitTest1.TestMethod8() | String Crozzle.ToStringHTML() | | | | |
| **Test Case ID** | **Parameters** | **Expected Data** | **Actual Data** | **Test Result** | **Test Comments** |
| 8.1 | n/a | expectedAfterValidate = [0] | crozzle.ToStringHTML() = [0] | Passed | Crozzle file is valid when Validate method called before saving and receiving HTML |
| 8.2 | n/a | expectedBeforeValidate = [1] | crozzle.ToStringHTML() = [1] | Passed | Crozzle file invalid when Validate method called before saving and receiving HTML |
| 8.3 | n/a | expectedNewHTML = [2] | crozzle.ToStringHTML() = [2] | Passed | Correct output from new files which has a different score of 336 |

**Data references:**

[0] = “html.txt”  
[1] = “html2.txt"  
[2] = “html3.txt"

**Test case 8.1, 8.2 and 8.3 files:**

8.1 – “Test 1 Configurations.txt”, “Test 1 Wordlist.txt”, “Test 1 Crozzle.txt”

8.2 – “Test 1 Configurations.txt”, “Test 1 Wordlist\_validateFalse.txt”, “Test 1 Crozzle\_validateFalse.txt”

8.3 – “Test 1 Configurations.txt”, “Test 1 Wordlist\_newWordsScore.txt”, “Test 1 Crozzle\_newWordsScore.txt”

1. **Test Scenario 9**
   1. **Justifications**
      1. **Test Case 1.1**

For the first test case, it involved analysing the return value of crozzleMap.GroupCount() on the original files and seeing that the expected number of groups from the GUI matches the actual value when the GroupCount() function is called. The function takes no parameters so the count value of the function is dependent on the crozzle file and configuration file passed into the application. We can assume the expected value of the GroupCount function will be true as the crozzle application displays all words as one connected group. As a result from conducting the test case, the expected value matched that of the actual value meaning that the unit test passed and the function can return the right value of GroupCount(). The importance of this test case is to ensure that the number of groups we see on the GUI matches the value of the returned value of the GroupCount() function.

* + 1. **Test Case 1.2**

This next test case is similar to that of test case 1.1 but we are checking the newly edited crozzle files that have had changes on “MAXIMUM\_NUMBER\_OF\_THE\_SAME\_WORD” to allow two groups since the original files only allowed one group. The crozzle files had the names of “Jack” and “Mark” removed which should split the once big group of connected words into two separate groups on the GUI. From the observation of the crozzle GUI, we are expecting the GroupCount() function to return the integer value of 2. After running the test, the expected value matched the actual value which means that the test successfully passed in count the new number of groups of connected words. It was crucial to conduct this test to observe if the function could interpret the correct number of group counts when changes have been made to the crozzle files.

* + 1. **Test Case 1.3**

Like 1.2, we are checking newly edited crozzle files have made further changes so that there is a total of four groups of connected words. The configuration files have also been changed to allow the new amount of total groups and we are checking that the GroupCount() function can successfully identify the four groups. As the GUI displays four separate groups, it is expected that the return value will be the integer value of 4. After conducting the test, the expected value matched the actual value meaning that the test has passed again to successfully detect changes to the number of groups of connected words. The reasoning for conducting a similar test to that of test case 1.2 is to fully confirm that the function is working correctly no matter what the group count is.

* 1. **Learning**

Designing these unit tests got me in the habit to ensure that the GroupCount function can always get the correct number of groups specified in the crozzle files. All the tests cases were very similar in that it was getting the total amount of groups but the repetition was important in case there was error when the crozzle hit a certain amount of total groups. I also found out findings that the score got lower as the more groups there were in the crozzle. Test case 1.2 had two groups with a score of 274 and test case 1.3 had four groups with a score of 247. This was due to less intersecting points as most of the points came from them. For future unit testing, I could have conducted tests to see if the correct error messages could be returned when the group count was out of range from the values specified.

* 1. **Test Scenario Form**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Test Scenario ID** | **Test Description** | | | | |
| 9 | Checking that the crozzle application can successfully count the number of groups of connected words in the application. The test is conducted on different changes to the crozzle files. If the expected matches the actual value, the application can successfully identify the number of groups of connected words. | | | | |
| **Test Method** | **Method Tested** | | | | |
| UnitTest1.TestMethod9() | int CrozzleMap.GroupCount() | | | | |
| **Test Case ID** | **Parameters** | **Expected Data** | **Actual Data** | **Test Result** | **Test Comments** |
| 9.1 | n/a | expectedGroupCount = 1  no errors | crozzleMap.GroupCount() = 1  no errors | Passed | All words are connected on crozzle hence 1 group |
| 9.2 | n/a | expectedNewGroupCount1 = 2  no errors | crozzleMap2.GroupCount() = 2  no errors | Passed | Crozzle has two separate groups hence group count 2 |
| 9.3 | n/a | expectedNewGroupCount2 = 4  no errors | crozzleMap3.GroupCount() = 4  no errors | Passed | Crozzle split into 4 different groups hence group count 4 |

**Data references:**

N/A

**Test case 9.1, 9.2 and 9.3 files:**

9.1 – “Test 1 Configurations.txt”, “Test 1 Wordlist.txt”, “Test 1 Crozzle.txt”

9.2 – “Test 1 Configurations\_groupCount2.txt”, “Test 1 Wordlist\_groupCount2.txt”, “Test 1 Crozzle\_groupCount2.txt”

9.3 – “Test 1 Configurations\_groupCount4.txt”, “Test 1 Wordlist\_groupCount4.txt”, “Test 1 Crozzle\_groupCount4.txt”

1. **Test Scenario 10**
   1. **Justifications**
      1. **Test Case 1.1**

Conducting this test case involved making sure that the expected validity of the Configuration file matched that of the Actual validity when the file path is passed into the Configuration.TryParse method. The reason behind this test is to ensure that the tryParse method can successfully check the configuration file for any illegal values such as the minimum number of rows being larger than the maximum number of rows. For this test case, the configuration file passed into the TryParse method is the original one and it is assumed that the expected result will be true as the configuration file breaks no rules and can load in the Crozzle files without generating errors to the log file. After conducting the test, the expected value matched that of the actual value meaning that the test case passed and the TryParse function successfully returned true for a valid configuration file.

* + 1. **Test Case 1.2**

For test case 1.2, we are checking that the configuration object created by the TryParse method has the same values that are contained in the configuration text files. The logic of conducting this test case is to ensure that the configuration values have not been changed within the Crozzle.TryParse() function when it had been validating it. For this test case, we are expecting that the property “MaximumNumberOfUniqueWords” is equal to the integer value of 1000. After running the test case, the expected value matched that of the actual value meaning that the test successfully passed and that Crozzle.TryParse has successfully retrieved the correct value from the configuration file.

* + 1. **Test Case 1.3**

Similar to the test case of 1.2, we are double checking that the configuration properties matched that of the configuration file. The property checked in this test case is “UPPERCASE” and seeing that the value “true” is not changed from false in the tryParse method when the configuration file is validated. We can assume the expected value returned will be “true” as the GUI shows all the letters in the Crozzle as uppercase letters and because the configuration files has the value set to “True” also. After conducting the test, it was found that the expected value matched that of the actual value meaning that the test cased passed. The verdict from the results is that the tryParse method can correctly obtain the value from UPPERCASE and not change that value through validation. It was also important to conduct this test as we don’t want a true value showing lower case letters. It ensures that the function is working correctly.

* 1. **Learning**

From these test cases, I have learned to write proper test cases when comparing the object properties with the properties within the text files. Conducting test case 1.1 had involved checking the validity of the configuration file and writing the test case from the passed test result; programmers can look at this test case and know that the function of the configuration TryParse is working correctly. If this was a false result, the table will help programmer to look in the TryParse method to see what is making test fail. Conducting this test case showed me how the value was affected. If there were any errors added to the error list object, it would cause the TryParse function to return false. Test case 1.2 and 1.3 further expanded my unit testing designs by ensuring that programmer can easily identify that the TryParse function can correctly take values from the configuration files. Overall, conducting these test cases made me understand the importance of writing them as it explores if the TryParse function is working correctly to obtain values from the configuration files. I also got to understand the validation of the TryParse method such as using IsInt32() and IsIBoolean() functions to validate the values inside the configuration file.

* 1. **Test Scenario Form**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Test Scenario ID** | **Test Description** | | | | |
| 10 | Check that values of Configuration properties are the same as the values within the configuration file as well as testing different value types such as integer and Boolean. If these expected values match the actual values, the method can read parts of the config file. Additionally, testing is done if the configuration file is returned as true. This means that the program can correctly open the config file. | | | | |
| **Test Method** | **Method Tested** | | | | |
| UnitTest1.TestMethod10() | Boolean Configuration.TryParse(path, out Configuration aConfiguration); | | | | |
| **Test Case ID** | **Parameters** | **Expected Data** | **Actual Data** | **Test Result** | **Test Comments** |
| 10.1 | path = “Test 1 Configuration.txt" | expectedFileReturn = True | actualFileReturn = true; | Passed | TryParse can successfully find and open the config file |
| 10.2 | path = “Test 1 Configuration.txt" | expectedMaxUniqueWords = 1000 | aConfiguration. MaximumNumberOfUniqueWords= 1000 | Passed | Expected value matches the value in config text file |
| 10.3 | path = “Test 1 Configuration.txt" | expectedTrueUppercase = true | aConfiguration.Uppercase = true | Passed | Tested to see that program can detect uppercase in file. |

**Data references:**

N/A

**Test case 10.1, 10.2 and 10.3 files:**

10.1 – “Test 1 Configurations.txt”

10.2 – “Test 1 Configurations.txt”

10.3 – “Test 1 Configurations.txt”

1. **Test Scenario 11**
   1. **Justifications**
      1. **Test Case 1.1**

This test case involved checking that the words in the wordlist object matched that of the words in the original wordlist file. The reason for checking this is to make sure that each and every word is correctly pulled from the wordlist file and not changed in the process of validating the wordlist file that was passed into the tryParse Function. For this case, the word “BETTY” is checked in the wordlist object using the Contains() function. Since the words are being stored in a String List object, the contains () function will search for the String “BETTY” amongst all the other strings in the list and then return a Boolean value. The expected return value is assumed to be “True” because the wordlist file contains “BETTY” and the GUI correctly displays the word. As a result, the expected value matched that of the actual value meaning that the test successfully passed and the function can correctly obtain all the words from the wordlist file correctly.

* + 1. **Test Case 1.2**

To further justify that the wordlist TryParse() method is getting all the words from the wordlist file, this test case involves seeing that the expected amount of words is equal to the amount in the wordlist file. If the expected value matches that of the actual value, the wordlist object can correctly get all the words from the wordlist file. To check the total number of words in the wordlist object, the Count() method is used to iterate through the list and count the total amount of string elements. We can assume the expected result is equal to 31 as that is the amount of words that are contained in the wordlist file. After conducting the unit tests, the expected result matched that of the actual result meaning that the test successfully passed and the tryParse function correctly retrieved the number of words contained in the wordlist file.

* + 1. **Test Case 1.3**

This final test case involves checking that the wordlist file passed through the TryParse method can be identified as a valid file. The reason for conducting this test case is to check for logical errors that could make valid files be returned as false. For this test case, the “Valid” property is checked and compared to see if it matches the expected value. The “Valid” property is dependent on the correctness of the wordlist file layout and we can assume the expected value is “true” as the crozzle application can open the wordlist file with no issues. As a result, the expected value matched the actual value meaning that the test passed and validation works on the wordlist file.

* 1. **Learning**

In test case 1.1 and 1.2, it has opened new ways on checking that the object properties match that of the properties specified in the crozzle files. I had got experience in using the Contains() function to look for certain sub strings with a String list which is helpful for checking that the correct string values have been passed. This also becomes a useful function for checking certain words when strings become long such as the HTML string returned from the ToStringHTML. The Count() has also been a useful function that I can use for future unit tests. I can use this to make sure that the correct number of elements can be pulled from the properties of a text file. Finally, test case 1.3 taught my many things about validation such as used regex values to make sure that certain fields in text files are alphabetic. Regex values have been an aspect that I can certainly use in the future to help with validation. Overall, conducting all the test cases in this unit test has made me realised the importance of comparing the wordlist properties to the wordlist file as a failed test means that there is a bug somewhere in the method that can change the value of a property.

* 1. **Test Scenario Form**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Test Scenario ID** | **Test Description** | | | | |
| 11 | Check that values of Wordlist properties are the same as the values within the wordlist file and that the correct number of words can be obtained from the wordlist file. If these expected values match the actual values, it means the method can correctly identify properties in the wordlist file. Additionally, a test is run to check that the current wordlist file is valid. If the return value is true, it means that the method can identify a valid file if the conditions are met. | | | | |
| **Test Method** | **Method Tested** | | | | |
| UnitTest1.TestMethod11() | Boolean Wordlist.TryParse(String path, Configuration aConfiguration, out WordList aWordList) | | | | |
| **Test Case ID** | **Parameters** | **Expected Data** | **Actual Data** | **Test Result** | **Test Comments** |
| 11.1 | path = “Test 1 Wordlist.txt"  aConfiguration = {CrozzleApplication.Configuration} | expectedContainedWord = true; | aWordList.Contains(“BETTY”) = true | Passed | Wordlist file contains the string “BETTY”. |
| 11.2 | path = “Test 1 Wordlist.txt "  aConfiguration = {CrozzleApplication.Configuration} | expectedFieldNumber = 31 | aWordlist.List.Count = 31 | Passed | Counting the total words in the wordlist file |
| 11.3 | path = “Test 1 Wordlist.txt "  aConfiguration = {CrozzleApplication.Configuration} | expectedValid = true; | aWordlist.Valid = true | Passed | Wordlist file does not break conditions hence valid. |

**Data references:**

N/A

**Test case 11.1, 11.2 and 11.3 files:**

11.1 – “Test 1 Configurations.txt”, “Test 1 Wordlist.txt”

11.2 – “Test 1 Configurations.txt” “Test 1 Wordlist.txt”

11.3 – “Test 1 Configurations.txt” “Test 1 Wordlist.txt”

1. **Test Scenario 12**
   1. **Justifications**
      1. **Test Case 1.1**

This test case involves making sure that the crozzle object properties match the properties specified in the crozzle files. For this case, the “Rows” property which has the value of 10 is checked to see if it matches the Crozzle file property “ROWS=10”. The reason behind this test is similar to that of the previous test cases which is making sure that the crozzle.TryParse() function can correctly obtain the values within the crozzle text files. We do not want the value obtained to change and differ in the TryParse function during validation. We can assume the expected value to be the integer value of “10” as the crozzle files specify ten rows and the GUI displays ten rows. As a result, the expected value matches the actual value meaning that the test case has passed and the crozzle object can correctly get the number of rows as it is passed through the TryParse Function.

* + 1. **Test Case 1.2**

This test case is similar to that of 1.1 where the crozzle object properties are check to see if they match the crozzle properties in the crozzle files. The difference in this test case is to check that the “Columns” value matches that of “COLUMNS=15” in the text file. The reasoning for conducting a similar test case is to reinforce the evidence from 1.1 that the Crozzle.TryParse() function can successfully retrieve the correct number of columns. From observations of the Crozzle GUI and the value speficied in the crozzle file, we can assume that the expected value will be the integer value of “15”. After conducting the unit test, the expected value matched that of the actual value meaning that test successfully passed and the TryParse function has been double checked that it can retrieve the correct values from the crozzle files.

* + 1. **Test Case 1.3**

For this final test case, it involved checking that the Crozzle object can correctly retrieve the file path from the “WORDLIST\_FILE="Test 1 Wordlist.txt" in the crozzle file. The reason for checking this is so we know that the crozzle object can correctly get the word list file name during the TryParse Function. This investigates whether there is an error in the crozzle files that is failing to get the correct file name. In the test case, the actual result is set from the “WordListPath” property which contains the file path of the wordlist file. We can assume the expected result is the path “Test 1 Wordlist.txt” as the crozzle states that the wordlist file is valid. After conducting the tests, the expected value matched that of the actual value meaning that the test successfully passed.

* 1. **Learning**

By investigating the TryParse function, it has developed my unit testing skills to match the expected property value of an object to the actual property specified in the text files. I figured that it’s important to conduct 1.1 and 1.2 as it ensures that the objects can correctly pull the values specified in the text files. By doing a repeat on checking properties for 1.2, getting a pass on the test has educated me to compare all properties with actual values. Getting a failed test would mean that there is an error within the tryParse method and these tables can be used for the programmers to find these potential bugs causing logical errors. Test case 1.3 was similar to the other test cases but it was making sure that the function can successfully retrieve the wordlist file. Conducting this test case got me use to writing file paths that are expected. Properly written file paths in the unit tests will help programmers identify the file paths they need to evaluate if the function fails to go to the correct file path.

* 1. **Test Scenario Form**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Test Scenario ID** | **Test Description** | | | | |
| 12 | Check that values of Crozzle properties are the same as the values within the Crozzle file. If these expected values match the actual values, it means the method can correctly identify properties in the crozzle file. Additionally, testing is done if the expected wordlist path matches that of the actual wordlist path that is passed into the TryParse of crozzle. If it matches, then the method can correctly locate and open files. | | | | |
| **Test Method** | **Method Tested** | | | | |
| UnitTest1.TestMethod12() | Boolean Crozzle.TryParse(String path, Configuration aConfiguration, WordList wordList, out Crozzle aCrozzle | | | | |
| **Test Case ID** | **Parameters** | **Expected Data** | **Actual Data** | **Test Result** | **Test Comments** |
| 12.1 | path = “Test 1 Crozzle.txt"  aConfiguration = {CrozzleApplication.Configuration}  wordlist = {CrozzleApplication.WordList} | actualReturnRows = 10 | crozzle.Rows = 10 | Passed | Application can read rows from crozzle file |
| 12.2 | path = “Test 1 Crozzle.txt"  aConfiguration = {CrozzleApplication.Configuration}  wordlist = {CrozzleApplication.WordList}= null | actualReturnColumns = 15 | crozzle.Columns = 15 | Passed | Application can read columns from crozzle file |
| 12.3 | path = “Test 1 Crozzle.txt"  aConfiguration = {CrozzleApplication.Configuration}  wordlist = {CrozzleApplication.WordList} | expectedWordListPath = “Test 1 Wordlist.txt” | Crozzle.WordListPath = “Test 1 Wordlist.txt” | Passed | Can successfully read the wordlist text file from the crozzle text file. |

**Data references:**

N/A

**Test case 12.1, 12.2 and 12.3 files:**

12.1 – “Test 1 Configurations.txt”, “Test 1 Wordlist.txt”

12.2 – “Test 1 Configurations.txt” “Test 1 Wordlist.txt”

12.3 – “Test 1 Configurations.txt” “Test 1 Wordlist.txt”