**Comp-122 Assignment 2**

**Andrew Lawler**

Introduction

For this assignment I was tasked with creating a cipher text encryption and decryption program. For the first part of the task I was required to create an interface and then implement the interface In part B. For part b, I had to inplement the methods in a class called Caesar. Within this class I had the methods (rotate, frequencies, chiSquared and decipher). I implemented these methods using my knowledge of cipher text and the suitable examples shown to us.

Next Parts

**c.** Testing input for Rotate.java

I implemented a system that will print wrong input length when there is not two inputs. It then tests the inputs to make sure they are in the correct format.

|  |  |  |  |
| --- | --- | --- | --- |
| Input | Expected Output | Actual Output | Comments |
|  | Wrong input length! | Wrong input length! | This shows that the program ends with an error message when there is nothing entered. |
| . | Wrong input length! | Wrong input length! | This result tells us that the program will revoke all types of input when there is not the correct inputs. |
| 3 | Wrong input length! | Wrong input length! | This shows the program handling one integer input and providing the error message. |
| Andy | Wrong input length! | Wrong input length! | This shows that the program handles the one string input and provides the same error message. |
| Andy 3 | No integer first, no string second, wrong input | You did not enter an integer first!  You did not enter a string second!  Wrong inputs! | The output here shows that order matters. It tested the first input and found it was not an integer and also found the second input was not a string. These are the wrong inputs. |
| 3 Andy | dqgb | dqgb | Here you can see if we enter in the correct format it will work. |

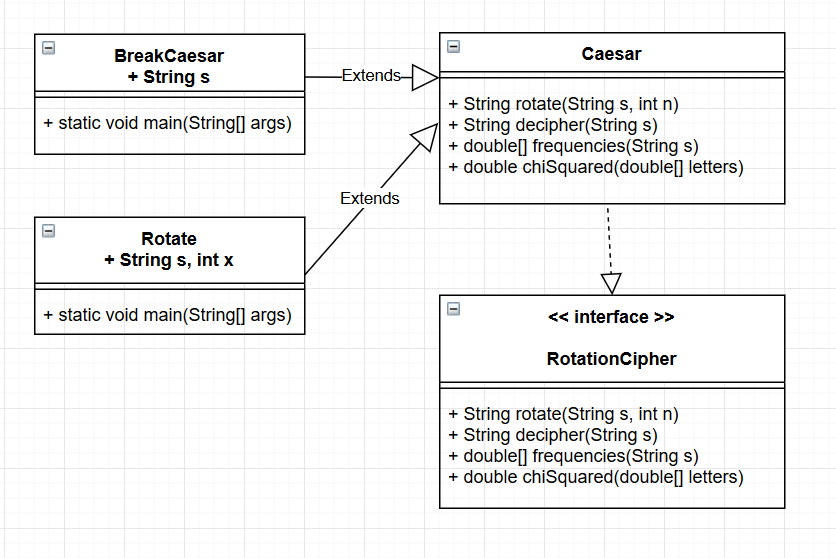
**d.** Testing input for BreakCaesar.java

I implemented the same system as for part c but with one input this time. It will check if the input size is equal to one and will then check if that input is a string. If either case is false we will get a specific error message.

|  |  |  |  |
| --- | --- | --- | --- |
| Input | Expected Output | Actual Output | Comments |
|  | Wrong amount of inputs | Wrong amount of inputs | This shows that the program requires inputs and will not let you enter nothing. |
| 3 | You didn’t enter a string | You did not enter a string! | This shows that you need to enter a string because the input is tested. |
| . | . | . | This was to test that the function did in fact let this in. A dot is allowed as we may form sentences to decrypt. |
| xf bsf ifsf | we are here | we are here | This was to test that the program would work. |

**e.** UML Diagram

This UML class diagram shows the exact layout of the program. First you see the interface called RotationCipher which is being implemented by Caesar. These two classes contain the same methods so I decided to copy them over and clearly indicate which is the interface and that the class Caesar is the one implementing the interface. I also know that the interface implies that the classes are abstract. Because this is known for interfaces I have not shown that the methods are abstract. I have declared them all as public which they are. Next I declared the two other extended classes. First, we have Rotate. Rotate is the class which is responsible for extending Caesar and using the method rotate. This class will take an input for a string and also the rotation amount. The class then uses the method from the base class and completes this rotation and prints the answer. The next class is BreakCaesar. BreakCaesar is responsible for taking an input as a plaintext string which is then decoded into its English counterpart. I showed clearly from this example that the classes both extend the Caesar class.

****

**f.** Java Docs

**g.**

If the language we are examining is not English we would need to modify some things. First, we would need to find the language we are examining and find its key attributes. For example, how many letters are in its alphabet, what order are they in and also what are the frequencies for these letters in this language. Once we have this core information we could then implement it in the exact same way I have done for English. I would just have to make the arrays bigger or smaller and change some things. This is what is so good about the program. As long as we find out the languages key attributes, we can decipher anything.

**h.**

The way I have written my program would allow me to calculate a shift for lower case and upper case. All I would do is I would have to use my rotation method and add a counter to show the positions of the lower-case letters and the positions of the upper case letters. I would then need to test all possible rotations of these amounts using my decipher method to then find the compatible rotation to decipher the text. This process would obviously take longer than just a loop of 26 as we would have to test every possible rotation against every possible rotation. I’d be looking at two embedded for loops to do this. On-top of this edit, I would have to edit my rotation method to take a second input as the one input n will not be able to apply two different rotations.