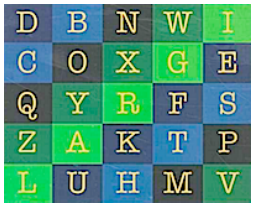
GALWAY-MAYO INSTITUTE OF TECHNOLOGY

Department of Computing & Mathematics

Object-Oriented Programming (2015)
ASSIGNMENT DESCRIPTION & SCHEDULE

A Multi-threaded Cypher Breaker



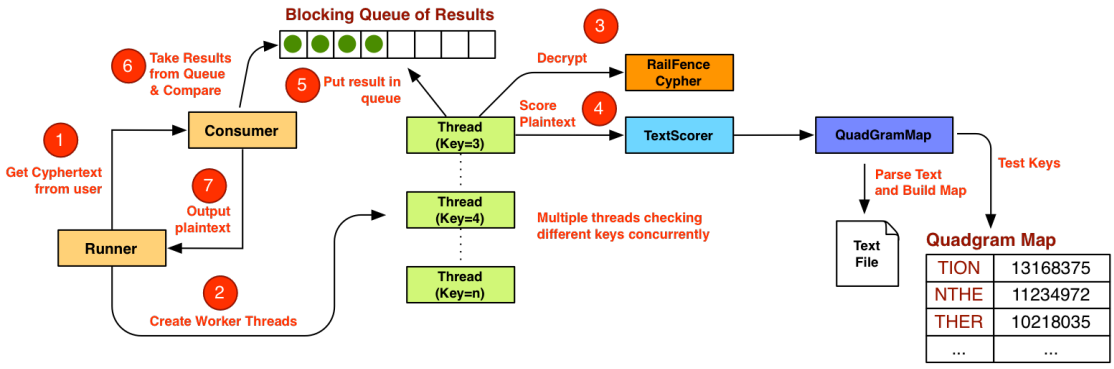
Note: This assignment will constitute 50% of the total marks for this module.

1. Overview

The field of cryptanalysis is concerned with the study of cyphers, having as its objective the identification of weaknesses within a cryptographic system that may be exploited to convert encrypted data (cypher-text) into unencrypted data (plaintext). Whether using symmetric or asymmetric techniques, cryptanalysis assumes no knowledge of the correct cryptographic key or even the cryptographic algorithm being used.

Assuming that the cryptographic algorithm is known, a common approach for breaking a cypher is to generate a large number of keys, decrypt a cypher-text with each key and then examine the resultant plaintext. If the text looks similar to English, then the chances are that the key is a good one. The similarity of a given piece of text to English can be computed by breaking the text into fixed-length substrings, called *n*-grams, and then comparing each substring to an existing map of *n*-grams and their frequencies. This process does not guarantee that the outputted answer will be the correct plaintext, but should give a good approximation that may well be the right answer.

You are required to implement a cryptanalytic API for breaking a Rail Fence cyphers. An overview of the application components is depicted below:



The following resources are already available on Moodle under "Resources for Main Assignment":

- **RailFence.java:** A full implementation of the Rail Fence cypher with methods for encrypting and decrypting text for a given integer key. In cryptanalysis, enemy knowledge of the algorithm (by hand, mechanisation or software) is often assumed by the person using the cypher. For example, the Enigma system assumed that a code breaker already had their own Enigma machine.
- **4grams.txt:** A Zip archive containing a set of 389,373 English quadgrams and their frequencies.
- **TextScorer.java:** a class that computes the score for how "English" a piece of text is.
- **build.xml:** An ANT build script for compiling the application, generating JavaDocs and packaging the resources.

2. Minimum Requirements

1. Use the package name **ie.gmit.sw**.
2. **Input** the plaintext or file name containing plaintext from a console. Use a *java.util.Scanner* for console input.
3. **Parse** the file *4grams.txt*, extract each quadgram and frequency and add each line to a *ConcurrentHashMap*.
4. **Create a BlockingQueue** to store results, i.e. plaintext, key and score, generated by one of the worker threads.
5. **A set of worker threads** should decrypt and score the plaintext with every integer key between 2 and the maximum size of the Rail Fence (half the length of the text). The result returned by each thread should be placed in the *BlockingQueue*.
6. **A single consumer object** should read results from the queue and return the best result when all the worker threads are finished. Note that a BlockingQueue will not know when to stop waiting… A common technique is to use a "poison" object to tell the queue to die.
7. The programme should **output the plaintext**, key and score of the highest scoring decrypted text.

The whole point of this assignment is for you to demonstrate an understanding of the principles of object-oriented design by using abstraction, encapsulation, composition, inheritance and polymorphism WELL throughout the application. Hence, you are also required to provide a UML diagram of your design and to JavaDoc your code. Please pay particular attention to how your application must be packaged and submitted. Marks will be deducted if you deviate from the requirements.

3. Deployment and Submission

- ***The project must be submitted by midnight on Sunday 10th January 2016*** using both Moodle and GitHub.

  GitHub:
  Submit the HTTPS clone URL, e.g. https://github.com/myaccount/my-repo.git of the public repository for your project. All your source code should be available at the GitHub URL. *You should try to use GitHub while developing your software and not just push changes at the end.*

  Moodle
  The project must be submitted as a Zip archive ***(not a rar or WinRar file)*** using the Moodle upload utility. You can find the area to upload the project under the " A

Multi-threaded Cypher Breaker (50%) Assignment Upload" heading in the "Notices and Assignments" section of Moodle.

- The name of the Zip archive should be <*id*>.zip where <*id*> is your GMIT student number.
- The Zip archive should have the following structure (do NOT submit the assignment as an Eclipse project):

Marks	Category
railfence.jar	A Java Archive containing your API and a runner class with a main() method. The application should be run as follows: **java –cp ./railfence.jar ie.gmit.sw.Runner** You can create the JAR file using Ant or with the following command from inside the "bin" folder of the Eclipse project: **jar –cf railfence.jar ***
src	A directory that contains the packaged source code for your application.
README.txt	Contains a description of the application, extra functionality added and the steps required to run the application. This file should BRIEFLY provide the instructions required to execute the project
design.png	A UML diagram of your API design. Your UML diagram should only show the relationships between the key classes in your design. Do not show methods or variables in your class diagram.
docs	A directory containing the JavaDocs for your application.
build.xml	An Ant build script that compiles the code in the ***src*** directory and builds a JAR archive called **railfence*.jar*.** Use the Ant build script available on Moodle. MAKE SURE THAT THE ANT SCRIPT USES THE CURRENT DIRECTORY (./). **You will lose marks if you use absolute paths. A sample Ant script is available on Moodle.**

4. Marking Scheme

Marks for the project will be applied using the following criteria:

Marks	Category
(50%)	Robustness
(10%)	Cohesion
(10%)	Coupling
(10%)	JavaDocs and UML Diagram
(10%)	Packaging & Distribution (GitHub and Moodle)
(10%)	Documented (and relevant) extras.

You should treat this assignment as a project specification. Any deviation from the requirements will result in a loss of marks. Each of the categories above will be scored using the following criteria:

- 0–30% Not delivering on basic expectation
- 40-50% Meeting basic expectation
- 60–70% Tending to exceed expectation
- 80-90% Exceeding expectations
- 90-100% Exemplary

5. Copying & Plagiarism

Plagiarism is the passing off of the work of another person as one's own and constitutes a serious infraction that may result in disciplinary proceedings. This assignment is an INDIVIDUAL assignment. While collaboration with other class members is acceptable for high-level design and general problem solving, you must individually code, implement and document your own submission.

Supplementary Notes

1) The Rail Fence Cypher

The *Rail Fence* is a symmetric transposition cypher, where the letters of a plaintext are rearranged using a pattern to encrypt a message. The *Rail Fence* cypher has roots in antiquity, where the ancient Greeks used a device called a *scytale* (pronounced skit-ally) to encrypt military communications. It was also used by both the Union and Confederate forces during the US Civil War (1861-65) to send Morse-encoded messages by telegraph.

1. Encrypting a Message with a Rail Fence

The Rail Fence cypher encrypts plaintext by creating an *n* x *k* matrix of characters, where *n* is the length of the plaintext and *k* is the key. The plain-text characters are written into the matrix in a zigzag pattern and resultant cypher-text is created by appending the set of characters in each row to a string. Consider the following matrix for encrypting the text STOPTHEMATTHECASTLEGATES with a key of 5.

S								A								T						
	T						M		T						S		L					S
		O				E				T				A				E				E	
			P		H						H		C						G		T		
				T								E								A			

The encrypted message will be SATTMTSLSOETAEEPHHCGTTEA.

2. Decrypting a Message with a Rail Fence

The decryption of a message requires that the columnar transposition above is undone, i.e. that the original encryption matrix is re-created. This can be performed by reading each character into a *n* x *k* matrix using a zigzag pattern and filling each row in turn. Consider the following states of the matrix to decrypt the cypher-text SATTMTSLSOETAEEPHHCGTTEA with a key of 5:

Zigzag and fill row 1: Adds the first three characters (SAT) to the matrix.

S								A								T						
	?						?		?						?		?					?
		?				?				?				?				?				?	
			?		?						?		?						?		?		
				?								?								?			

Zigzag and fill row 2: Adds the next six characters (TMTSLS) to the matrix.

S								A								T						
	T						M		T						S		L					S
		?				?				?				?				?				?	
			?		?						?		?						?		?		
				?								?								?			

Zigzag and fill row 3: Adds the next six characters (OETAEE) to the matrix.

S								A								T						
	T						M		T						S		L					S
		O				E				T				A				E				E	
			?		?						?		?						?		?		
				?								?								?			

Zigzag and fill row 4: Adds the next six characters (PHHCGT) to the matrix.

S								A								T						
	T						M		T						S		L					S

		O				E				T				A				E				E	
			P		H						H		C						G		T		
				?								?								?				

Zigzag and fill row 4: Adds the last three characters (TEA) to the matrix.

S								A								T						
	T						M		T						S		L						S
		O				E				T				A				E				E	
			P		H						H		C						G		T		
				T								E								A				

At this stage, the original encryption matrix has been created. The plain-text can now be created by zigzagging through the matrix and appending each character to a string.

2) Using n-Gram Statistics

An *n*-gram (gram = word or letter) is a substring of a word(s) of length *n*. For example the quadgrams (4-grams) of the word "Happy Days!" are "Happ", "appy", "ppy ", "py d", "y da", " day" and "days". N-grams can be used to test how similar some decrypted text is to English as follows:

- Create a map that relates *n*-grams to integers, i.e. Map<String, Integer> map = new ConcurrentHashMap<String, Integer>();
- Parse a large document of text, strip out all punctuation and white space, break the remaining text up into strings of length *n* and add them to the map. If the string already exists in the map, increment the value of the related integer.
- Enumerate all the possible keys between a minimum and maximum length. For each key:
    - Decrypt the cypher-text into plain-text with the key.
    - Break the plain-text into *n*-grams.
    - Score the *n*-grams using some statistically sound method.
    - Output the plain-text with the highest score.

A statistically sound score can computed by summing the individual *n*-gram scores computed as follows: *score = log(count(n-gram) / total)*, where *count(n-gram)* is the number of occurrences of an *n*-gram in the map and total is the total number of *n*-grams in the map.