**Abacus Group – Test Application (Poetry Generator) - 9/16/2022**

W.J. Fairchok

Introduction

As part of the hiring process at Abacus Group, the job applicant is expected to design and complete an application in C# or Java. This application is a technical test. The following sections will describe my design and answer the questions associated with the test.

Class Diagram

The class diagram is stored with the PoetryGenerator solution as PoetryGeneratorClassDiagram.cd. the diagram appears in Figure 1 below.

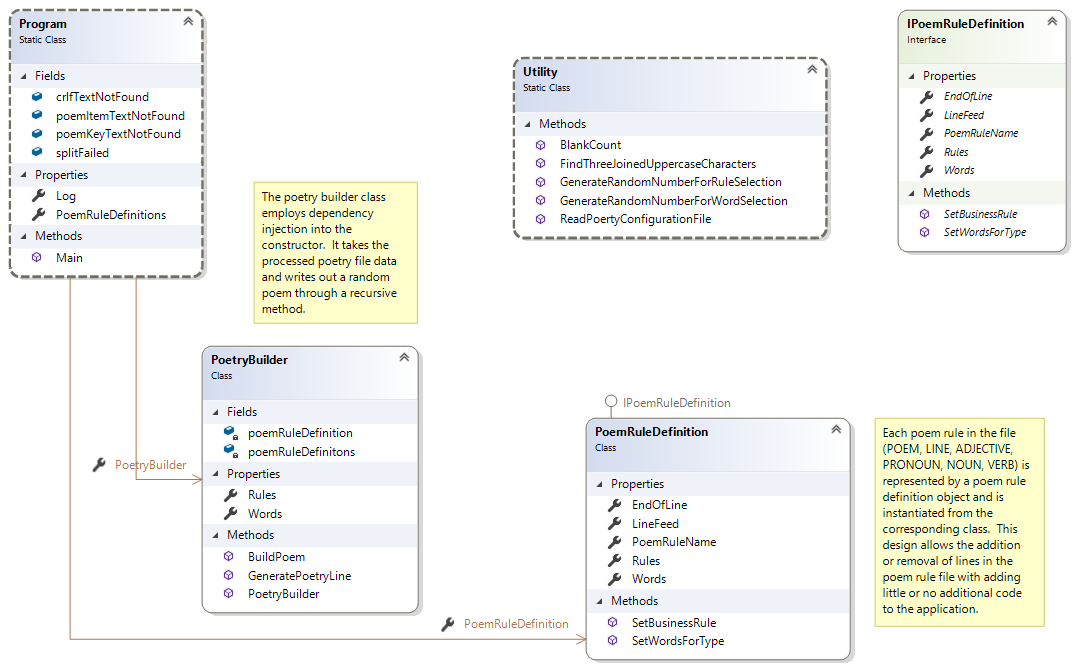


Figure 1 – Class diagram for the PoetryGenerator application

Assumptions and Design for PoetryGenerator

The test document provides the basic requirements and layout of the poetry rules. Here are the assumptions behind my application design:

1. The poetry rule can have additional lines added or removed, which include all new rules not yet defined. This is a common business situation.
2. The application design, therefore, must be flexible and have minimal hardcoding of rule names to minimize application changes in the future.
3. All poetry rules appear to be different, but there are similarities. The application must find commonality to provide a contract or interface to which all poetry rules can subscribe.
4. The provided file has carriage return -line feeds throughout the file. These must be removed and, in some cases, replaced with a blank.

The interface took some time to develop before coding could begin. Some poetry rules have only other rules defined. Others have both poetry words, which are to be randomly selected, and rules, which also are to be randomly selected. Some rules also have $END or $LINEBREAK keywords. All of these items have to be addressed in the interface.

The application was defined as a simple .NET console application. The underlying library chosen was .NET Core 3.1 as this will be supported by Microsoft for some time. The .NET 5 was also a choice, but it’s out-of-support with the introduction of .NET 6. Unfortunately, .NET 6 can’t be used with my licensed 2019 version of Visual Studio Enterprise as it’s only available with Visual Studio 2022. Version 10 of the C# language could also not be used for this test as it is only available in .NET 6. Thus, C# version 9 was used for the test.

Two of the classes for the project, Program and Utility, are static in nature. This means they employ the singleton design patter because one and only one version of the class is available for the application. It’s also possible to instantiate a singleton class, but extra code is required to ensure only one version of the object is available.

The PoemRuleDefinition class defines a common poetry rule object which conforms to the IPoemRuleDefinition interface or contract. This class is applied to all of the poetry objects in the file after some necessary text clean-up is performed. The rules and words for each poetry rule are stored in List<string> objects. Finally, the PoetryBuilder class takes the rules stored in the PoemRuleDefinitions collection and recursively generates a poem from randomly selected words and business rules.

Haiku Changes

If the Japanese Haiku were chosen, only three <LINE> rules would be specified for the poem. However, there is a limit of seventeen syllables, so a syllable count would have to maintained. Words may have to be rejected if the syllable count is not equal to seventeen.

The poem word syllable count could be obtained by using C# to query a webservice such as, <https://www.howmanysyllables.com>. Perhaps all words need to be looked up to get the syllable count and then random poems of seventeen could more quickly be determined.

Faster Design Approaches

Building code that is tightly coupled using hard-coded rules would allow for faster construction and testing. However, changes to the underlying poetry rules file would break the application quickly and may require considerable effort to get the application working. It would allow making text clean-up easier as each class could deal with individual issues rather than writing the more complex algorithm to handle all business rules with their individual peculiarities.

If a “quick and dirty” throw-away application is needed, then a tightly couple design may be preferred. However, expendable applications have a habit of remaining well past their anticipated lifetime and this fact must be taken into consideration prior to beginning work.