Тестовое задание для Terrasoft

Задание

Реализовать программу анализа текста. Входной текст произвольный и может быть большим по объему. Количество и содержание метрик определяется самостоятельно. Требования к алгоритму: Программа должна быть расширяема к изменению списку метрик. Масштабируемость.
Метрики можете предлагать самостоятельно, например самый частый символ или количество восклицательных предложений. Процент существительных слов в тексте и т.д

Решение

Решение состоит из трех проектов:

- **`TextGenerator`**, консольное приложение для генерации случайного текста. Принимает два параметра:
  - Путь для генерируемого файла. Если параметр не указан, то будет сгенерирован файл *text.txt* в текущей директории.
  - Размер файла в байтах. Значение по умолчанию 1000000.

- **`TextAnalyser`**, консольное приложение, которое реализует поставленную задачу анализа текста. Принимает один параметр: путь к текстовому файлу. Если не указан, то будет попытка прочитать файл *text.txt* из текущей директории.

- **`TextAnalyser.Tests`**, проект с юнит-тестами для TestAnalyser, в котором используется Xunit юнит-тест фреймфорк.

Архитектура

Для обработки больших объемов данных, данные будут обрабатываться поточно, чтобы не держать их в памяти.

Задача разбита на две подзадачи:

1. Разбивка входящего текста на семантические элементы (токены).
2. Передача полученных токенов объектам реализующим метрики.

1. Разбивка текста на токены

За данную подзадачу отвечает класс Tokeniser:

```
public class Tokeniser
{
  public Tokeniser(params TokenBuilder[] tokenBuilders)
  {…}

  public IEnumerable<Token> Tokenise(StreamReader stream)
  {
    …
    yield return token;
    …
  }
   …
}
```

Метод `Tokenise` получает на вход текстовый поток и преобразует его в поток токенов, возвращая итератор. Таким образом `Tokeniser` может разбивать сколь угодно большие объемы текста, не заботясь о памяти.

Выделение токенов из текста происходит с помощью объектов классов производных от `TokenBuilder`, которые `Tokeniser` получает через свой конструктор. Это дает возможность конфигурировать `Tokeniser` на производство нужных типов токенов.

В текущем решении реализованы следующие `TokenBuilder`-ы:

- `CharacterTokenBuilder`, выделяет отдельные символы
- `WordTokenBuilder`, выделяет слова
- `SentenceTokenBuilder`, выделяет предложения
- `NumberTokenBuilder`, выделяет целые числа

Рассмотрим чуть подробнее класс TokenBuilder:

```
public abstract class TokenBuilder
{
  …
  public event EventHandler<TokenReadyEventArgs>
TokenReady = delegate { };

  public void OnNextChar(char nextChar, long position)
  {…}

  public void OnEnd()
  {…}
   …
}
```

Он содержит два основный метода:

- OnNextChar, который принимает и обрабатывает следующий символ из текстового потока.
- OnEnd, с помощью этого метода TokenBuilder-у сообщается что текстовый поток закончен.

Как только TokenBuilder формирует очередной токен, вызывается событие TokenReady, в аргументы которого передается токен и его позиция в тексте.

Таким образом алгоритм Tokenise сводится к следующему:

- подписаться на события TokenReady всех TokenBuilder-ов
- получая из текстового потока символы передавать их TokenBuilder-ам
- При срабатывании события TokenReady получить токен и вернуть его через итератор.

2. Передача токенов метрикам

За вторую подзадачу и в целом все решение задания отвечает класс TextAnalyser:

```
public class TextAnalyser
{
  public TextAnalyser(Tokeniser tokeniser)
  {…}

  public void Analyse(StreamReader stream, params
Metric[] metrics)
  {
    foreach (Token t in _tokeniser.Tokenise(stream))
    {
      foreach (Metric m in metrics)
        m.OnNextToken(t);
    }
  }
  …
}
```

Метод `Analyse` получает на вход текстовый поток и массив объектов класса `Metric`. Текстовый поток передается `Tokeniser`-у, который был получен через конструктор и в ответ возвращается итератор токенов. Каждый токен передается каждому объекту класса `Metric`, которые выполняют необходимые подсчеты по анализу текста.

Рассмотрим базовый класс `Metric` и производную от него generic-версию:

```
public abstract class Metric
{
  public abstract void OnNextToken(Token token);
}
```

```
public abstract class Metric<TToken> : Metric where
TToken : Token
{
  public override void OnNextToken(Token token)
  {
    if (typeof(TToken) != token.GetType())
      return;

    OnNextToken((TToken) token);
  }

  protected abstract void OnNextToken(TToken token);
}
```

Базовый класс имеет абстрактный метод OnNextToken, который принимает следующий токен из потока для обработки. Производная generic версия параметризуется типом токена, для которого создана данная метрика, например метрика для символов, метрика для слов, для предложений и тд. Перегруженная версия метода `OnNextToken` отфильтровывает токены неподходящего типа и вызывает в свою очередь другой абстрактный метод `OnNextToken`, принимающий токен конкретного типа. Данный метод в свою очередь надлежит переопределять производным классам, реализующим собственно конкретную метрику.

В данном решении реализованы следующие метрики:
- `MostFrequentCharacterMetric`, наиболее частый символ.
- `WordsCountMetric`, суммарное число слов.
- `ExclamationSentenceCountMetric`, число восклицательных предложений.
- `NumbersSumMetric`, сумма всех целых чисел встреченных в тексте.
- `AverageNumberMetric`, среднее арифметическое всех целых чисел встреченных в тексте.

Заключение

Представленное решение решает задачу обработки больших объемов данных благодаря поточному подходу. Расширяемость обеспечена гибкой архитектурой, которая позволяет расширять функционал в двух измерениях:

- Реализация производных классов от класса `TokenBuilder`, позволяет разбивать текст на произвольные токены.

- Реализация производных классов от класса `Metric`, позволяет произвольным образом производить анализ текста.