4. Tests Generator

Необходимо реализовать многопоточный генератор шаблонного кода тестовых классов для одной из библиотек для тестирования (NUnit, xUnit, MSTest) по тестируемым классам.

**Входные данные:**

* список файлов, для классов из которых необходимо сгенерировать тестовые классы;
* путь к папке для записи созданных файлов;
* ограничения на секции конвейера (см. далее).

**Выходные данные:**

* файлы с тестовыми классами (по одному тестовому классу на файл, вне зависимости от того, как были расположены тестируемые классы в исходных файлах);
* все сгенерированные тестовые классы должны компилироваться при включении в отдельный проект, в котором имеется ссылка на проект с тестируемыми классами;
* все сгенерированные тесты должны завершаться с ошибкой.

Генерация должна выполняться в **конвейерном режиме "производитель-потребитель"** и состоять из трех этапов:

1. параллельная загрузка исходных текстов в память (с ограничением количества файлов, загружаемых за раз);
2. генерация тестовых классов в многопоточном режиме (с ограничением максимального количества одновременно обрабатываемых задач);
3. параллельная запись результатов на диск (с ограничением количества одновременно записываемых файлов).

**При реализации использовать async/await и асинхронный API. Для реализации конвейера можно использовать Dataflow API:**

* ​<https://docs.microsoft.com/en-us/dotnet/standard/parallel-programming/dataflow-task-parallel-library>​
* ​<https://docs.microsoft.com/en-us/dotnet/standard/parallel-programming/walkthrough-creating-a-dataflow-pipeline>​

**Главный метод генератора должен возвращать Task и не выполнять никаких ожиданий внутри (блокирующих вызовов task.Wait(), task.Result, etc). Для ввода-вывода также необходимо использовать асинхронный API (**[**https://docs.microsoft.com/en-us/dotnet/standard/io/asynchronous-file-i-o**](https://docs.microsoft.com/en-us/dotnet/standard/io/asynchronous-file-i-o)**).**

Необходимо сгенерировать по одному пустому тесту на каждый публичный метод тестируемого класса

Пример сгенерированного файла для NUnit:

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using NUnit.Framework;

using MyCode;

​

namespace MyCode.Tests

{

[TestFixture]

public class MyClassTests

{

[Test]

public void FirstMethodTest()

{

Assert.Fail("autogenerated");

}

​

[Test]

public void SecondMethodTest()

{

Assert.Fail("autogenerated");

}

...

}

}

Для синтаксического разбора и генерации исходного кода следует использовать **Roslyn**:

* ​<https://github.com/dotnet/roslyn>​
* ​<http://roslynquoter.azurewebsites.net/>​

Код лабораторной работы должен состоять из трех проектов:

* библиотека для генерации тестовых классов, содержащая логику по разбору исходного кода и многопоточной генерации классов;
* модульные тесты для главной библиотеки;
* консольная программа, содержащая логику по чтению входных данных, загрузке исходных файлов в память и записи результатов работы (сгенерированных тестовых классов) в файлы.

**Задание со звездочкой**

Необходимо сделать генератор более "умным" путем **учёта структуры тестируемого класса**:

* если тестируемый класс принимает через конструктор зависимости по интерфейсам, то в тестовом классе необходимо объявить метод SetUp, в котором создать экземпляр тестируемого класса и Mock-объекты (с помощью Moq или аналогов) всех необходимых ему зависимостей, сохранить их в поля тестового класса для использования в тестах, и передать в конструктор создаваемого тестируемого класса;
* для простоты анализ параметров конструктора тестируемого класса достаточно выполнять по именам и полагаться на соглашение об именовании интерфейсов (комплексный анализ проекта/решения, к которому относится тестируемый класс, выполнять не обязательно);
* необходимо сгенерировать по одному шаблонному тесту на каждый публичный метод тестируемого класса и создать шаблоны для Arrange (подготовка теста), Act (вызов тестируемого кода), Assert (проверка результата) секций метода;
* секция **Arrange** должна содержать объявление переменных со значениями по умолчанию по входным данным тестируемого метода;
* секция **Act** должна содержать вызов тестируемого метода с передачей ему аргументов, объявленных в Arrange, и сохранение результата метода в переменную actual;
* секция **Assert** должна содержать объявление переменной expected с типом, соответствующим возвращаемому значению метода, и одну проверку на равенство actual и expected;
* процедура генерации шаблонов для void методов и для классов, которые принимают в конструктор не только зависимости по интерфейсам, на усмотрение автора,

приветствуется разумная обработка заданных случаев, но это не является обязательным, пропускать и вообще не обрабатывать такие классы/методы нельзя.

Пример результата работы усовершенствованного генератора для NUnit и Moq:

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using NUnit.Framework;

using Moq;

using MyCode;

​

namespace MyCode.Tests

{

[TestFixture]

public class MyClassTests

{

private MyClass \_myClassUnderTest;

private Mock<IDependency> \_dependency;

​

[SetUp]

public void SetUp()

{

\_dependency = new Mock<IDependency>();

\_myClassUnderTest = new MyClass(\_dependency.Object);

}

[Test]

public void MethodTest()

{

// Arrange (комментарии в сгенерированном коде не требуются)

int number = 0;

string s = "";

Foo foo = null;

// Act

int actual = \_myClassUnderTest.MyMethod(number, s, foo);

​

// Assert

int expected = 0;

Assert.That(actual, Is.EqualTo(expected));

Assert.Fail("autogenerated");

}

}

}

​

​

# MaxDegreeOfParallelism

В рамках обсуждения работы был выдвинут тезис, что даже при установке ограничения MaxDegreeOfParallelism в единицу (в ExecutionDataflowBlockOptions) потоки создаются для обработки сразу всех задач. Действительно, если смотреть на значение Process.GetCurrentProcess().Threads.Count во время обработки данных в pipeline, то там число **>1** даже сразу при старте процесса. Помимо потока сборщика мусора, потока вызова финализаторов (деструкторов) и главного потока видны потоки системного пула потоков, а также потоки I/O Completion Ports (<https://docs.microsoft.com/en-us/windows/desktop/fileio/i-o-completion-ports>). Во время работы .NET регулирует количество обоих видов потоков самостоятельно. Что же касается MaxDegreeOfParallelism - это ограничение количества параллельно выполняемых задач. Это число **не равно** количесту потоков в работающем процессе. При правильной работе через Dataflow, CPU-bound задачи (генерация кода, например) будут запускаться в системном пуле потоков в соответствии с MaxDegreeOfPrallelism. IO-bound задачи (например, чтение/запись файлов) не будет требовать больше потоков, чем нужно (за счет использования конструкторов block-ов, которые принимают сразу awaitable Task). Никаких дополнительных ограничений на размер системного пула ставить не нужно, доверьтесь в этом .NET Runtime.