Рекомендуемые источники:

* Официальная документация Microsoft: [docs.microsoft.com](https://docs.microsoft.com/);
* CLR via C#, Джеффри Рихтер;
* C# программирование для профессионалов, Джон Скит;
* Разработка через тестирование, Кент Бек;
* Эффективная работа с унаследованным кодом, Майкл Физерс;
* Рефакторинг. Улучшение существующего кода, Мартин Фаулер.

## Выполнение работ

* Каждая работа должна быть размещена в отдельном публичном репозитории на любом удобном для вас git-хостинге (github, gitlab, bitbucket, etc).
* Для реализации можно использовать любой CIL-язык. Предпочтительный вариант — C#, так как на его примере все объясняется на лекциях, и он же должен быть использован на экзамене для решения задачи. Рекомендуется использовать .NET 6.
* Код должен следовать стилю программирования, принятому для используемого вами языка. Для С# это [C# Coding Conventions](https://docs.microsoft.com/en-us/dotnet/csharp/fundamentals/coding-style/coding-conventions).

# 1. Tracer

**Затрагиваемые темы**

* Reflection.
* Многопоточное программирование.
* Сериализация.
* Объектно-ориентированный дизайн.
* Плагины.

Необходимо реализовать измеритель времени выполнения методов, используя системный класс [StackTrace](https://docs.microsoft.com/en-us/dotnet/api/system.diagnostics.stacktrace?view=net-6.0).

Класс должен реализовывать следующий интерфейс:

public interface ITracer

{

// вызывается в начале замеряемого метода

void StartTrace();

// вызывается в конце замеряемого метода

void StopTrace();

// получить результаты измерений

TraceResult GetTraceResult();

}

Конкретная структура TraceResult на усмотрение автора, однако публичный интерфейс должен предоставлять **доступ только для чтения:** свойства должны быть **неизменяемыми и использовать неизменяемые типы данных** (IReadOnlyList<T>, IReadOnlyDictionary<TKey, TValue> и подобные), также не должно быть публичных методов, изменяющих внутреннее состояние TraceResult.

Tracer должен собирать следующую информацию об измеряемом методе:

* имя метода;
* имя класса с измеряемым методом;
* время выполнения метода.

Пример использования

public class Foo

{

private Bar \_bar;

private ITracer \_tracer;

internal Foo(ITracer tracer)

{

\_tracer = tracer;

\_bar = new Bar(\_tracer);

}

public void MyMethod()

{

\_tracer.StartTrace();

...

\_bar.InnerMethod();

...

\_tracer.StopTrace();

}

}

public class Bar

{

private ITracer \_tracer;

internal Bar(ITracer tracer)

{

\_tracer = tracer;

}

public void InnerMethod()

{

\_tracer.StartTrace();

...

\_tracer.StopTrace();

}

}

Также должно подсчитываться общее время выполнения анализируемых методов в одном потоке. Для этого достаточно подсчитать сумму времен "корневых" методов, вызванных из потока.

Результаты трассировки вложенных методов должны быть представлены в соответствующем месте в дереве результатов.

Для замеров времени следует использовать [класс Stopwatch](https://docs.microsoft.com/en-us/dotnet/api/system.diagnostics.stopwatch?view=net-6.0).

### Представление результата

Результат измерений должен быть представлен в одном формате на выбор: **JSON,** **XML** и **YAML.** При реализации плагинов следует использовать **готовые библиотеки для работы с данными форматами.**

При этом класс TraceResult не должен содержать никакого дополнительного кода для сериализации: атрибутов, ненужных конструкторов/полей/свойств, реализаций интерфейсов или наследований. Подобный код, если он нужен, должен содержаться только в проекте для конкретного сериализатора.

Классы для сериализации результата должны иметь общий интерфейс (интерфейс должен располагаться в отдельном проекте, см. Организация кода ) и загружаться динамически во время выполнения как "плагины" с помощью метода Assembly.Load (см. [How to: Load Assemblies into an Application Domain, Example](https://docs.microsoft.com/en-us/dotnet/framework/app-domains/how-to-load-assemblies-into-an-application-domain#example)).

Copy

public interface ITraceResultSerializer

{

// Опционально: возвращает формат, используемый сериализатором (xml/json/yaml).

// Может быть удобно для выбора имени файлов (см. ниже).

string Format { get; }

void Serialize(TraceResult traceResult, Stream to);

}

Результирующие файлы могут иметь любые имена: 1.txt, 2.txt, 3.txt. Также можно использовать использовать свойство ITraceResultSerializer.Format для создания файлов с соответствующим расширением: result.json, result.xml, result.yaml.

**Важно**

Код загрузки плагинов **не должен содержать никаких указаний (путей к сборкам, имён классов и т. д.) на сами плагины.** Сборки должны загружаться динамически из папки, следует использовать все найденные реализации интерфейса ITraceResultSerializer.

**Пример результатов:**

{

"threads": [

{

"id": "1",

"time": "42ms",

"methods": [

{

"name": "MyMethod",

"class": "Foo",

"time": "15ms",

"methods": [

{

"name": "InnerMethod",

"class": "Bar",

"time": "10ms",

"methods": ...

}

]

},

...

]

},

{

"id": "2",

"time": "24ms"

...

}

]

}

Обратите внимание, что в результатах работы потока на одном уровне может находиться несколько методов. Это возникает в ситуации, когда StartTrace() и StopTrace() вызываются не везде (два вкладки: с кодом и с результатом):

public class C

{

private ITracer \_tracer;

public C(ITracer tracer)

{

\_tracer = tracer;

}

public void M0()

{

M1();

M2();

}

private void M1()

{

\_tracer.StartTrace();

Thread.Sleep(100);

\_tracer.StopTrace();

}

private void M2()

{

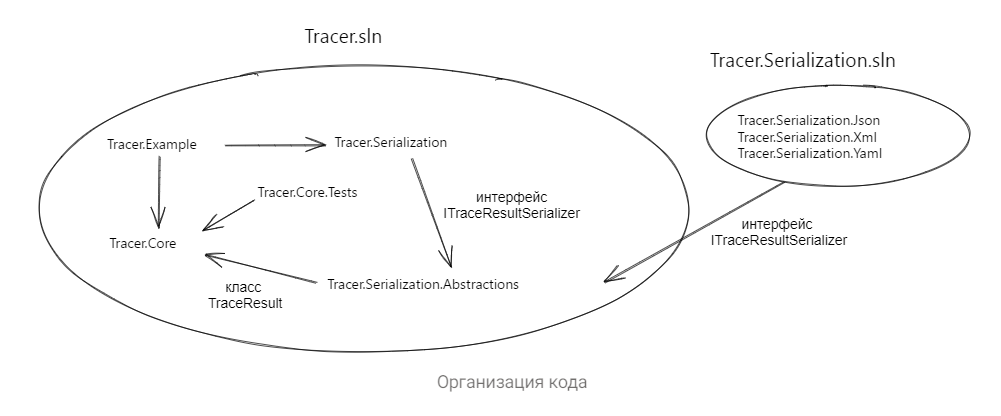
\_tracer.StartTrace();

Thread.Sleep(200);

\_tracer.StopTrace();

}

}

**Организация кода**Код лабораторной работы должен состоять **из двух решений (solutions)**:

* **Tracer.sln:** содержит основной код, тесты и интерфейс для создания плагинов.
  + **Tracer.Core:** основная часть библиотеки, реализующая измерение и формирование результатов.
  + **Tracer.Core.Tests**: модульные тесты для основной части библиотеки.
  + **Tracer.Serialization.Abstractions**: содержит интерфейс ITraceResultSerializer для использования в плагинах.
  + **Tracer.Serialization**: содержит код для загрузки плагинов и сохранения результатов, ссылается на Tracer.Serialization.Abstractions.
  + **Tracer.Example:** консольное приложение, демонстрирующее общий случай работы библиотеки (в многопоточном режиме при трассировке вложенных методов) и записывающее результат в файл в соответствии с загруженными плагинами.
* **Tracer.Serialization.sln:** содержит проекты с реализацией плагинов для требуемых форматов сериализации и ссылку на Tracer.Serialization.Abstractions из основного решения.
  + **Tracer.Serialization.Json**
  + **Tracer.Serialization.Yaml**
  + **Tracer.Serialization.Xml**
  + *Tracer.Serialization.Abstractions*:данный проект из основного решения нужен для использования интерфейса ITraceResultSerializer из проектов .Json, .Yaml и .Xml.