

Czech Technical University, Prague  
Faculty of Electrical Engineering



Master's Thesis  
**Performance Profiling for .NET Platform**

*Bc. Jan Vratislav*

Supervisor: Ing. Miroslav Uller

Study Program: Electrical Engineering and Information Technology

Study Branch: Computer Science and Engineering

January 2012



# **Declaration**

I hereby declare that I have completed this master's thesis independently and that I have listed all the literature and publications used.

I have no objection to usage of this work in compliance with the act "§60, zákon číslo 121/2000Sb." (copyright law), and with the rights connected with the copyright act including the changes in the act.

Prague, January 2nd, 2012



# Acknowledgement

I thank to Ing. Miroslav Uller for accepting and supporting my thesis.

I thank to my closest and my friends for their support.

I thank to Bc. Petr Diviš for proofreading the text of this thesis and for valuable hints.



# **Abstract**

The performance profiling is a powerful way how to get a valuable insight into software applications. In order to be precise and effective, a performance profiler needs to run with low performance and memory overhead and present the result in a transparent way to a developer. In this thesis we analyze performance profiling in the context of Microsoft .NET platform, with focus on the C# programming language, and introduce our own implementation of a profiler, the Visual Profiler. The Visual Profiler features tracing and sampling profiling engines together with an innovative way of presenting the profiling results within the integrated development environment Microsoft Visual Studio 2010.

# **Abstrakt**

Výkonové profilování je mocný nástroj při zkoumání chování softwarových aplikací. K zajištění maximální přesnosti a efektivity musí výkonový profiler běžet s minimálními paměťovými a výkonnostními nároky a prezentovat vývojáři naměřené výsledky v přehledné formě. V této práci se zabýváme analýzou výkonového profilování na platformě Microsoft .NET, v kontextu jazyka C#, a představujeme vlastní implementaci profileru, Visual Profiler. Visual Profiler nabízí stopovací a vzorkovací profilování společně s inovativním způsobem zobrazování naměřených výsledků v integrovaném vývojovém prostředí Microsoft Visual Studio 2010.



# Contents

<b>List of Figures</b>	<b>xiv</b>
<b>1 Introduction</b>	<b>1</b>
<b>2 Overview of Performance Profilers</b>	<b>3</b>
2.1 Profiling modes . . . . .	3
2.1.1 Sampling profiling . . . . .	3
2.1.2 Tracing profiling . . . . .	3
2.1.3 Instrumentation profiling . . . . .	4
2.2 Granularity of profiling . . . . .	4
2.2.1 Line-by-line granularity . . . . .	4
2.2.2 Method granularity . . . . .	4
2.2.3 Selective granularity . . . . .	4
2.3 Profiling results . . . . .	5
2.3.1 Time measurement . . . . .	5
2.3.2 Call trees . . . . .	5
2.4 Available .NET performance profilers . . . . .	6
2.4.1 JetBrains dotTrace . . . . .	6
2.4.2 Redgate ANTS Performance Profiler . . . . .	7
2.4.3 EQATEC Profiler . . . . .	7
2.4.4 GlowCode . . . . .	7
2.4.5 Visual Studio Profiler . . . . .	7
2.4.6 Others . . . . .	7
2.4.7 SlimTune . . . . .	8
2.4.8 Prof-It for C# . . . . .	8
<b>3 Performance Profiling on .NET</b>	<b>9</b>
3.1 .NET framework core facilities . . . . .	9

3.2	Profiling modes in .NET . . . . .	11
3.2.1	Sampling mode . . . . .	11
3.2.2	Tracing profiler . . . . .	11
3.2.3	Instrumentation profiling . . . . .	11
3.3	Profiling API . . . . .	11
3.3.1	Supported notifications . . . . .	13
3.4	Summary . . . . .	14
<b>4</b>	<b>Visual Profiler Backend</b>	<b>15</b>
4.1	Choice of the implementation strategy and profiling modes . . . . .	15
4.2	Software architecture of backend . . . . .	16
4.3	Implementations of <code>ICorProfilerCallback3</code> interface . . . . .	16
4.3.1	<code>TracingProfiler</code> class . . . . .	17
4.3.2	<code>SamplingProfiler</code> class . . . . .	20
4.4	Call tree and call tree element classes . . . . .	21
4.5	Metadata . . . . .	23
4.6	Profiling enabled assembly . . . . .	25
4.7	Measuring profiling data . . . . .	25
4.7.1	Tracing profiler . . . . .	25
4.7.2	Sampling profiler . . . . .	26
4.8	Sending the profiling results . . . . .	26
4.8.1	Serializing the profiling data . . . . .	27
4.8.2	Live updates . . . . .	27
4.9	Summary . . . . .	29
<b>5</b>	<b>Visual Profiler Access</b>	<b>31</b>
5.1	Metadata, call trees and call tree elements . . . . .	31
5.2	Deserialization . . . . .	32
5.3	Visual profiler access API . . . . .	33
5.4	Source code locations . . . . .	35
5.5	Summary . . . . .	35
<b>6</b>	<b>User Interface and Visual Studio Integration</b>	<b>37</b>
6.1	Visually mapping profiling results to source code . . . . .	37
6.2	Source code coloring and bird's-eye view . . . . .	37
6.3	Implementation of the user interface . . . . .	39

6.3.1	WPF . . . . .	39
6.3.2	Model-View-ViewModel . . . . .	41
6.3.3	User interface - model . . . . .	42
6.3.4	User interface - view-model . . . . .	43
6.3.5	User interface - view . . . . .	43
6.4	Visual Studio 2010 IDE integration . . . . .	45
6.4.1	Macros . . . . .	45
6.4.2	Snippets . . . . .	45
6.4.3	Templates . . . . .	47
6.4.4	Editor Extensions - MEF . . . . .	47
6.4.5	Add-In . . . . .	47
6.4.6	Package . . . . .	47
6.5	Visual Profiler integration in Visual Studio . . . . .	48
6.5.1	Commands . . . . .	48
6.5.2	Tool window . . . . .	48
6.5.3	Editor adornment extension . . . . .	49
6.6	Summary . . . . .	49
<b>7</b>	<b>Testing and Evaluations</b>	<b>51</b>
7.1	Testing . . . . .	51
7.2	Testing of Visual Profiler Backend . . . . .	51
7.3	Testing of Visual Profiler Access and UI . . . . .	52
7.4	Evaluation of overhead . . . . .	52
7.5	Evaluation of exactness . . . . .	53
7.6	Impact of assembly filtering on results . . . . .	53
7.7	Summary . . . . .	54
<b>8</b>	<b>Conclusion and Future Work</b>	<b>55</b>
<b>Bibliography</b>		<b>57</b>
<b>A</b>	<b>Visual Profiler Installation</b>	<b>59</b>
A.1	System requirements . . . . .	59
A.2	Hardware requirements . . . . .	59
A.3	Installation of VSIX package . . . . .	59
<b>B</b>	<b>Running Visual Profiler</b>	<b>62</b>

B.1	Enabling profiling . . . . .	62
B.2	Running Visual Profiler . . . . .	62
B.3	Screenshots . . . . .	63
<b>C</b>	<b>Source code</b>	<b>65</b>
C.1	Solutions and projects . . . . .	65
C.2	Dependencies . . . . .	65
<b>D</b>	<b>Content of enclosed DVD</b>	<b>66</b>

# List of Figures

2.1	Profiling modes comparison . . . . .	5
2.2	Red-Gate ANTS Profiler call graph . . . . .	6
3.1	Overview of the Common Language Infrastructure [Wik11] . . . . .	10
3.2	Profiling architecture [Mic11b] . . . . .	12
4.1	The conceptual architecture of the backend . . . . .	16
4.2	Implementations of the <code>ICorProfilerCallback3</code> interface . . . . .	17
4.3	Union of thread call stack snapshots forming a call tree. A very simple example: A calls B, B calls C, C returns to B, B calls D, D returns to B, B return to A . . . . .	18
4.4	A simplified representation of the <code>TracingCallTree</code> class. The active element pointer points to the currently executing function element with the call tree hierarchy. . . . .	18
4.5	Merge of a stack snapshot with a thread's sampling call tree. . . . .	20
4.6	The <code>StackWalker</code> class . . . . .	21
4.7	The hierarchy and relation between the call tree and the call tree element classes used to trace the collected profiling data. . . . .	22
4.8	The hierarchy and relation between the metadata classes used to store .NET metadata. . . . .	24
4.9	The <code>VisualProfilerAccess</code> class encapsulates the communication with the frontend . . . . .	26
4.10	The <code>SerializationBuffer</code> class members . . . . .	27
4.11	The structure of the serialization buffer . . . . .	28
4.12	The live update activity diagram: cooperation between the tracing profiler and the live update - simplified example . . . . .	28
5.1	The metadata inheritance hierarchy closely resembles its counterpart from the chapter 4. . . . .	31
5.2	The call tree and call tree element inheritance hierarchy closely resembles their counterparts from the chapter 4. . . . .	32

5.3	The <code>DeserializationExtensions</code> class, which adds extension methods to the <code>System.IO.Stream</code> class in order to provide deserialization of the basic types. . . . .	32
5.4	The communication logic classes . . . . .	34
6.1	Seesoft - mapping each line of code into a thin row, colored according to a statistics of interest and showing the context in a preview window. . . . .	38
6.2	Laying a colored rectangle over the source code. The color hue of the rectangle corresponds to a measured value of the method. . . . .	38
6.3	bird's-eye view representation of source files represented as columns and methods represented as rectangles with a highlighted method and detailed information. . . . .	40
6.4	Overview of the main new features of WPF [Mos11] . . . . .	41
6.5	Schema of the MVVM pattern [Mos11] . . . . .	41
6.6	Call trees and metadata are merged to form source file groups with aggregated methods. . . . .	42
6.7	The class hierarchy of the model part. . . . .	42
6.8	The <code>ViewModelBase</code> class implements <code>INotifyPropertyChanged</code> interface. . . . .	43
6.9	Placement of the views in the final UI look. . . . .	44
6.10	Main extension points of Visual Studio IDE. . . . .	46
6.11	Commands in the <code>Tools</code> menu in Visual Studio 2010. . . . .	49
A.1	Visual Studio Extension Installer - before installation . . . . .	60
A.2	Visual Studio Extension Installer - after successful installation . . . . .	60
A.3	The Extension Manager in Visual Studio 2010 . . . . .	61
B.1	Visual Profiler - before running . . . . .	62
B.2	Visual Profiler - stopping a profiler session . . . . .	63
B.3	Visual Profiler - profiling the Mandelbrot testing application . . . . .	63
B.4	The layout error - the code coloring does not align with method bodies . . . . .	64
D.1	DVD content . . . . .	66

# Chapter 1

## Introduction

According the Pareto's law (also known as 80/20 rule) 80 percent of the results come from 20 percent of the effort [Koc99]. This law is applicable to software development. 80 percent of all end users generally use only 20 percent of a software application's features. Microsoft reported that 80 percent of errors and crashes in Windows and Office are caused by only 20 percent of bugs [Roo02].

We believe that the same principle applies to software performance as well. 80 percent of time is spent in 20 percent of code. So, investing effort of programmers to the 20 percent of the code may have great effect on the overall performance of the code - if that 20 percent of the code could be discovered.

Programmers are not particularly successful at guessing which part of the code is crucial for the performance [McC04]. Therefore profilers help to automate the search for bottlenecks and hotspots in applications and their source code and provide valuable metrics, such as execution times of specific part of code or memory usage of a given object. This thesis focuses mainly on performance profiling on the Microsoft .NET platform.

The performance profiling dynamically analyzes execution behavior of a program. It tracks various runtime related data such as frequency and duration of function calls. Analysis and visualization of the gathered data provides useful hint on the program's code runtime characteristics and helps during optimization and exploration of the program.

Profiling can be achieved by various means as for instance by inserting tracing code into either the source code or the binary executable of the program or by runtime sampling of thread call stacks of the program or by listening to events invoked by a program's runtime engine.

Each of aforementioned approaches differs in overhead imposed to the program and in type, precision and granularity of the gathered data.

In the world of .NET performance profiling, there already are few full-fledged solutions targeting almost all .NET platforms from desktop to Windows Phone applications. However, they are mostly commercial and do not provide deep integration with development tools.

In this thesis, we will introduce a development-time .NET profiler offering various profiling methods and allowing direct interaction directly from the Microsoft Visual Studio 2010.



# Chapter 2

## Overview of Performance Profilers

In this chapter we present various characteristics and mechanisms of performance profilers and overview some contemporarily used profilers targeting the .NET platform.

### 2.1 Profiling modes

An application can be profiled in several ways that differ in the precision of their results and profiling overhead.

#### 2.1.1 Sampling profiling

In this mode, the profiler stops periodically every thread of a profilee<sup>1</sup> and inspects method frames on its call stacks. The output snapshot is not an exact representation of the runtime conditions rather a statistical approximation, but the profiling overhead is very low and the profilee runs almost at full speed with minimum side effects, such as additional memory allocations, cache faults and context switches. The profiling overhead in the sampling mode does not depend on the number of methods called by the profilee opposed to other profiling modes.

The profiler cannot count the number of method calls. It can only count their occurrence on top of a stack. Method durations are computed purely based on gathered statistics. No source code or a binary alternation is required. The runtime ability to stop execution to do a call stack snapshot or alternatively an interrupt instruction can be used.

#### 2.1.2 Tracing profiling

This way of profiling relies on a runtime environment or some kind of hardware notification. The notification is performed when a method is entered or left. This method leads to accurate results that provide exact count of method calls and their durations. However, there is performance overhead. In addition, the overhead increases with the number of method calls and slows down the profilee.

---

<sup>1</sup>profiled process

Similar to the sampling profiling, there are no changes to code or binaries required. The profiler has to only register callback methods for the entry/leave notifications with the runtime or the hardware profiling infrastructure.

### 2.1.3 Instrumentation profiling

Code or binaries of the profilee are modified by injection of arbitrary code or instructions in order to collect profiling data. This approach can have similar effects and results as the tracing profiling, but offers a lot more choice of what and how to profile. There are chances of alteration of the original profilee behavior or even introducing bugs.

Instrumentation can be performed on every level of the software live-time (source, compilation, binary, runtime) and can be both manual, performed by a developer, as well as automatic, done by a compiler or a runtime environment.

## 2.2 Granularity of profiling

A profiler can measure profiling result on different levels of a program. Not every profiling approach can achieve any granularity due to its limitations. Speed of the profilee is always trade-off between granularity and accuracy.

### 2.2.1 Line-by-line granularity

Line-by-line profiling is the most accurate method and most demanding. Every program statement is measured and analyzed. It brings precise result; however, the additional burden can alter the program's behavior.

Such a fine granularity can be achieved only by the instrumentation profiling.

### 2.2.2 Method granularity

Only information regarding an entire method run is collected. The lower overhead can improve performance of the profilee and still provide very informative results.

This level of granularity is the only option for the sampling profilers and most types of the tracing profilers. The instrumentation profilers can operate in the method granularity mode.

### 2.2.3 Selective granularity

The profiled process can be filtered with combined levels of granularity, where some parts of code are monitored on the line-by-line level and others only on the method level or not at all in order to capture desired statistics with lowest possible overhead and highest possible accuracy.

		granularity		
profiling mode	method	line-by-line	overhead	accuracy
sampling	yes	no	very low	moderate
tracing	yes	no	high	high
instrumentation	yes	yes	very high	very high

Figure 2.1: Profiling modes comparison

## 2.3 Profiling results

During the process of profiling various kinds of data can be collected. For some application, it is sufficient to count the method hit count; for others a detailed call stack analysis is required. Again, the amount of result information places overhead on the profilee, in this case, primary on the memory (caches, pages...) and secondary on the CPU.

### 2.3.1 Time measurement

There are several different kinds of the program time measurement a profiler can provide. Not every profiling mode can provide every single measurement. Usually, measurement accuracy is counterweighted with computational complexity.

#### Wall-clock time

The wall time measurement starts when a thread enters a method and ends when the thread leaves the method. The resulting time does not reflect if the method does useful computation or if it is in wait, sleep or join state.

#### User and kernel time

In comparison with the wall-clock time, the user and kernel time measurement count solely the time spent executing a method except the time spent by waiting, sleeping or joining.

Every profiling mode is capable of both the wall time as well as the user and kernel time measurements, either by statistically distributing the program execution time over the method hit counts or by reading CPU registers or system performance counters.

### 2.3.2 Call trees

Relations among method calls allow the profiler to reconstruct a call tree. The call tree reveals hit count of functions and, more importantly, what function calls which function. This insight helps to find hot spots and understand the runtime conditions. Additional function parameter analysis is also possible, but at the cost of higher CPU and memory overhead. An example of a call tree is shown in the figure 2.2.

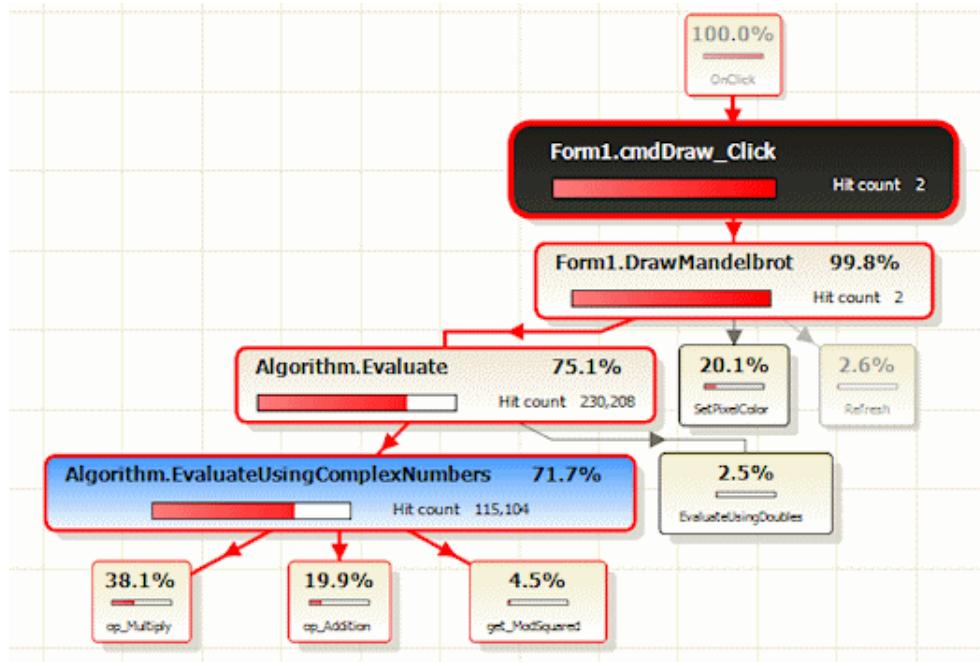


Figure 2.2: Red-Gate ANTS Profiler call graph

Some profilers do not track calls hierarchy and they only record function hit count and duration. In this case, the call tree is referred by the term flat call tree.

## 2.4 Available .NET performance profilers

On the market, there are many various options in the field of .NET performance profiling. The commercial solutions offer many features and integration with other tools, however, not with Visual Studio in most cases. There are also a few open source alternatives.

Unfortunately, we could not carry out deeper performance and features assessment of available profilers, since we did not have required financial and time resources for this challenging and very interesting undertaking.

### Commercial solutions

#### 2.4.1 JetBrains dotTrace

dotTrace profiles .NET Framework 1.0 to 4.0, Silverlight 4, or .NET Compact Framework 3.5. It offers partial integration with the Visual Studio and other JetBrains tools. All the profiling modes described above are supported and the profiling can run remotely.

[www.jetbrains.org](http://www.jetbrains.org)

### 2.4.2 Redgate ANTS Performance Profiler

This tool targets similar set of .NET applications as JetBrains dotTrace. In addition it offers SQL and I/O profiling, live results, all the profiling modes and time schemas.

It offers basic integration with the Visual Studio and allows seeing code directly from the profiling tool.

[www.red-gate.com](http://www.red-gate.com)

### 2.4.3 EQATEC Profiler

There is no doubt that this profiler offers the widest targeting platform options. It can be used to profile virtually anything in the ".NET world". It uses unknown kind of instrumentation and thus the choice of the profiling mode is restricted, however it is configurable. A binary has to be modified before a profiling session.

There is no integration with the Visual Studio whatsoever and it seems that the source code result overview cannot be displayed either.

[www.eqatec.com](http://www.eqatec.com)

### 2.4.4 GlowCode

GlowCode is performance and memory profiler for Windows and .NET programmers who develop applications with C++ or any .NET Framework language. GlowCode helps to detect memory leaks and resource flaws, isolate performance bottlenecks, profile and tune code, trace real-time program execution, ensure code coverage, isolate boxing errors, identify excessive memory usage, and find hyperactive and loitering objects. It can be used for native, managed, and mixed code. [ES11]

The site did not mention anything about the Visual Studio integration. So we assume that it is not supported.

[www.glowcode.com](http://www.glowcode.com)

### 2.4.5 Visual Studio Profiler

This tool is integrated to Visual Studio and supports native and managed code profiling. It has the sampling and instrumenting profiling modes. It is shipped only with higher editions of Visual Studio.

### 2.4.6 Others

There are many other professional profiling solutions comparable to those mentioned above (among others, the Telerik JustTrace, the SpeedTrace Pro).

It is a challenging task to pick the right solution for one's needs with the right licensing options.

## Open source solutions

The open source alternatives are not as numerous as the commercial ones.

### 2.4.7 SlimTune

SlimTune is a free profiler that offers advanced features such as remote profiling, 32-bit and 64-bit profiling, and live results. There is very little information available and user has to dive into the source code to find out about its inner mechanisms.

[code.google.com/p/slimtune](http://code.google.com/p/slimtune)

### 2.4.8 Prof-It for C#

Prof-It profiler for C# introduces a unique way of profiling. It is a line-by-line instrumentation profiler with its own source viewer allowing import and profiling of Visual Studio projects. It presents results as an overlay over the source code and as a list of methods, blocks and classes.

Unfortunately, this project does not seem to be actively developed.

[dotnet.jku.at/projects/Prof-It](http://dotnet.jku.at/projects/Prof-It)

## Summary

In this introduction, we have looked at and explained sampling, tracing and instrumenting profiling modes. Then we focused on profiling results and their granularity, mainly call trees and different kinds of inspected durations. In the end of the chapter, a short overview of current commercial and open-source scene was presented.

# Chapter 3

## Performance Profiling on .NET

In this chapter, we will start with a brief review of main .NET framework core facilities, mainly CLR, and then explore options of the performance profiling on this platform.

### 3.1 .NET framework core facilities

.NET framework (or just .NET) is a software platform developed by Microsoft for developing and executing software applications. .NET consists of a runtime environment, which executes programs, class libraries, interfaces and services. It offers ready to use solutions to software developers for standard programming tasks, so they only have to concentrate on program logic itself.

.NET based applications do not contain machine-level dependent instructions and therefore, cannot be directly executed by CPUs. Instead of that, they are composed from a human readable “middle” code instructions, so called Common Intermediate Language (CIL). In order to execute CIL, it has to be first compiled into processor specific machine-level instructions by the JIT compiler<sup>1</sup> that uses processor specific information during compilation. This mechanism is comparable to Java and its bytecode.

The .NET platform is a Microsoft’s implementation of the Common Language Infrastructure standard (CLI) and forms a solid base for development and execution of programs that can be written in many programming languages and target many platforms. The center elements of .NET are the runtime environment, called Common Language Runtime (CLR), the Basic Class Library (BCL) and diverse utility programs for management and settings.

Many programming languages such as C#, F#, Visual Basic and others can be translated to the CIL and then run on the CLR.

CLR provides huge amount of services and features. Besides the already mentioned JIT compilation, it provides assembly loading, type system, garbage collection and memory management, security system, native code interoperability, debugging, performance and profiling interfaces, threading system, managed exception handling, application domains and many others.

---

<sup>1</sup>Just-in-time compiler

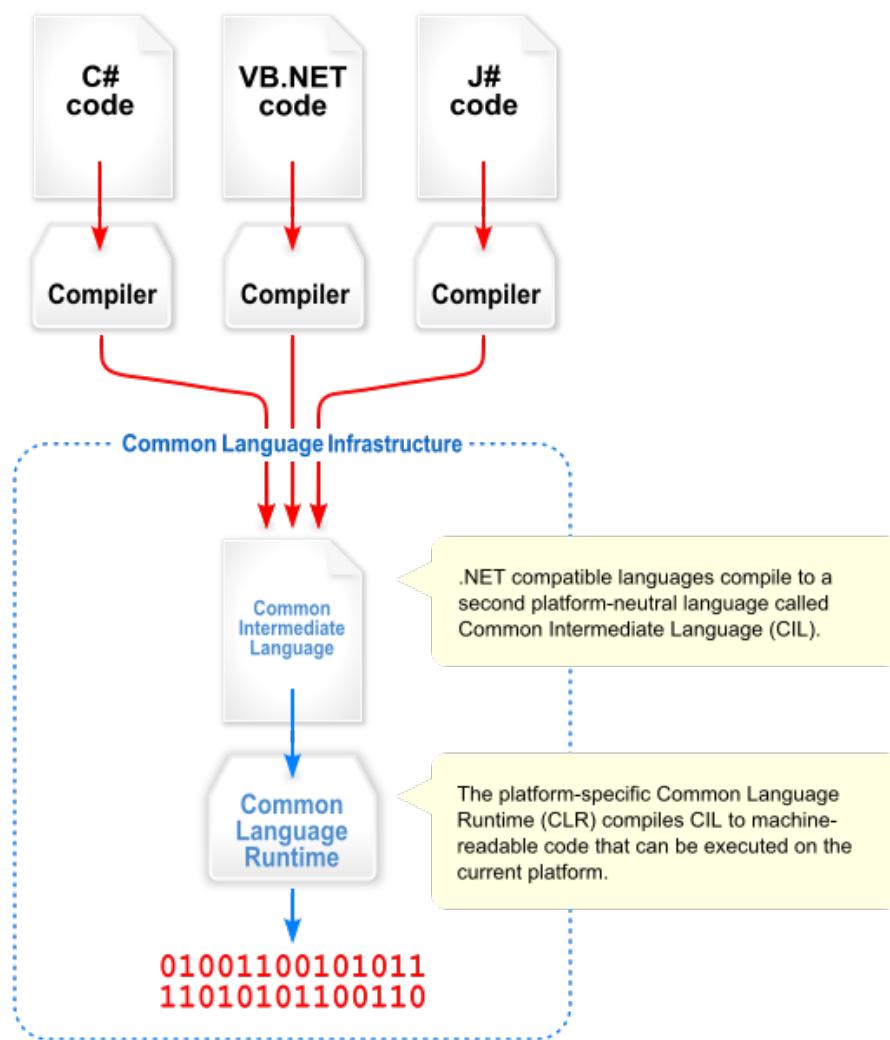


Figure 3.1: Overview of the Common Language Infrastructure [Wik11]

## 3.2 Profiling modes in .NET

All the profiling modes (sampling, tracing, instrumentation), mentioned in the chapter 2, are feasible in the .NET environment. There is even more ways how to achieve some of them, however, with various implementation complexity.

### 3.2.1 Sampling mode

Due to the JIT compilation, real address of a function remains unknown till the point of the JIT compilation of the function, which might not even occur if the function is not part of a program execution path. Therefore, there is not a direct way how to acquire the function and its memory address mapping without digging into the CLR runtime structures. Luckily, the Profiling API answers these problems and lets a programmer sample all managed call stacks and read metadata regarding the methods.

### 3.2.2 Tracing profiler

As stated in the introduction in the chapter 2, the tracing profiling must be supported by the runtime engine or the targeting hardware. The Profiling API offers a very elegant way to implement this profiling strategy and to trace the CLR activity.

### 3.2.3 Instrumentation profiling

This mode is the most complicated, nevertheless, with the most profiling possibilities.

The first possibility how to create a .net instrumenting profiler is by an injection of the profiling source code right into the application source code before a compilation from a high level programming language to the CIL. Various features of the higher level language have to be taken in the account during this process, e.g. lambda methods and closures in C#. The obvious limitation is a single target programming language.

The second possibility is similar to the first and it only goes one level down, to manipulate CIL of already compiled .NET assemblies or modules. This approach targets every .NET program regardless of the programming language. However, certain problems have to be solved, e.g. the target assembly signing.

The third possibility is to use the JIT compilation notifications of the Profiling API. The original CIL can be modified before the JIT compilation.

It is apparent how big role the Profiling API plays in the profiling of .NET applications. It is a great starting point for every profiling mode implementation.

## 3.3 Profiling API

This section is based on articles, documentation and examples provided on the Microsoft Developer Network (MSDN) [Mic11b]. The provided information is valid for the version of the Profiling API that comes with the .NET 4.0.

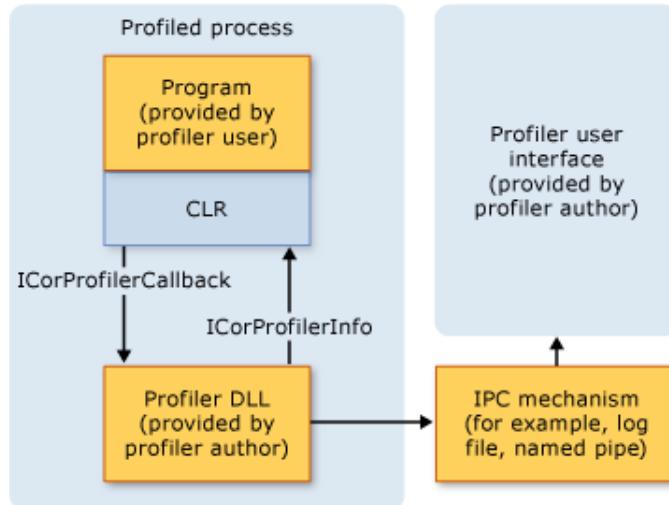


Figure 3.2: Profiling architecture [Mic11b]

Profiling a .NET application is not as straightforward as profiling a conventional application compiled into machine code. The main reasons are aforementioned CLR features that conventional profiling methods cannot properly identify and that complicate the profiling. The Profiling API provides a way how to acquire the profiling data for a managed application with minimum effect on the performance of the CLR and the profiled application.

The purpose of the Profiling API is not to be only a profiling tool, as its name suggests, but to be a versatile diagnostic tool for the .NET platform that can be used in many program analysis scenarios, for instance a code coverage utility for unit testing.

To make use of the Profiling API, as depicted in the figure 3.2, a profiler's native code DLL containing profiling custom logic is loaded to the profiled application's process. The profiler DLL implements the `ICorProfilerCallback` interface and creates an in-process COM server. The CLR calls methods in that interface to notify profiler of one out of many runtime events of the profiled process. The profiler can query the state of the profiled application and of the events by calling back into CLR using methods of `ICorProfilerInfo` interface.

It is important to keep the profiler DLL as simple as possible. Only the data monitoring part of the profiler should be running in the profiled process. The rest of the profiler, analysis and UI, should run in an independent process.

Unfortunately, the profiler DLL cannot be written in managed .NET code. It actually makes sense. If the interfaces were implemented using managed code, it would cause a self triggering notification and would eventually end up in a never ending recursion or a deadlock. For example, imagine that you register a method enter notification handler and you call a managed method within the handler. You would get notified of that.

After the COM implementation of the interfaces is finished, the CLR needs to be aware of the profiling DLL and forced to load it and call its methods. Simply enough, this is accomplished by setting environment variables (`COR_ENABLE_PROFILING`, `COR_PROFILER`, `COR_PROFILER_PATH`). No XML setting, no registry entries have to be changed. One has to careful with that. If you changed the machine wide environmental variables then every

.NET process would be profiled. The profiler can be even attached in the runtime.

### 3.3.1 Supported notifications

The CLR notification can be divided into following groups as listed in [Mic11b].

1. CLR startup and shutdown events.
2. Application domain creation and shutdown events.
3. Assembly loading and unloading events.
4. Module loading and unloading events.
5. COM vtable creation and destruction events.
6. Just-in-time (JIT) compilation and code-pitching events.
7. Class loading and unloading events.
8. Thread creation and destruction events.
9. Function entry and exit events.
10. Exceptions.
11. Transitions between managed and unmanaged code execution.
12. Transitions between different runtime contexts.
13. Information about runtime suspensions.
14. Information about the runtime memory heap and garbage collection activity.

The implementation of the `ICorProfilerCallback` interface is mandatory to use the Profiling API. The developer of the profiler has to implement all methods of the interface and there are 80 of them. Luckily, for the majority of methods, those out of interest, it is simply enough to return `S_OK` `HRESULT`. The others have to provide code to handle the events and desired logic.

Most of the methods of the `ICorProfilerCallback` interface receive additional data in form of input parameters. And some methods come in pairs started-finished (`GarbageCollectionStarted - GarbageCollectionFinished`) or enter-leave (`ExceptionSearchFunctionEnter - ExceptionSearchFunctionLeave`).

The parameters are either unsigned int (`UINT`) ids or pointers to some CLR structures. For instance the `ThreadCreated` function gets its `threadId` parameter of type `ThreadID` (only `typedefed` `UINT`). The Ids are opaque values, but we think they are bare pointers to the memory where the metadata reside since they are always defined as `UINT_PTR`.

To get more information, such as function names, defining classes, assemblies, and other metadata, by an id, the Profiling API offers the `ICorProfilerInfo` and the

`IMetadataImport` interfaces and a system of metadata Tokens corresponding to the metadata.

The Profiling API offers a lot and there is a lot more to write about. However, there are also some limitations to it. Among others, there are no profiling for unmanaged code and no remote profiling.

## **3.4 Summary**

In this chapter, the .NET framework was briefly introduced. We have outlined some .NET profiling strategies for all the profiling modes and introduced the powerful Profiling API for profiling managed applications.

# Chapter 4

## Visual Profiler Backend

The backend is a part of the profiler responsible for collecting data from a profiled application and for sending them to the analytic part of the Visual Profiler frontend. There are many requirements for a profiling backend to fulfill, such as speed, multi-threading, correctness, low memory consumption and others. In this chapter, we introduce the profiling modes we chose to implement and review the inner implementation of the backend and design decision we had to make.

### 4.1 Choice of the implementation strategy and profiling modes

Based on the analysis of the profiling modes on the .NET platform in the chapter 3, we made a decision to use the Profiling API as the base for our own profiler instead of implementing everything from scratch. The Profiling API offers a matured API with intrinsic support of all .NET features. This helps to avoid many unnecessary implementation problems that could be encountered during an implementation of a .NET profiler entirely from the beginning.

One of the primary goals of this work was to create a method granularity profiler supporting two distinct profiling modes for the .NET.

The choice of the sampling profiling mode was straightforward because of its stochastic nature in that it completely differs from the other two modes.

On the other hand, the difference between the tracing and the instrumentation profiling is not that vast. They both measure exact results and put similar overhead on the profiling. However, the selected method granularity makes them both even more comparable from user's point of view. They both have their implementation pitfall and neither of them would be easier to program than the other. In the end, we opted for the tracing profiling mode since it does not require any changes in target assemblies.

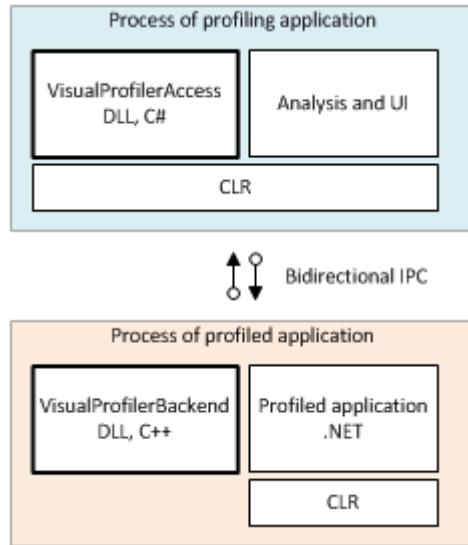


Figure 4.1: The conceptual architecture of the backend

## 4.2 Software architecture of backend

The backend runs in two processes. The first process is the actual profiled application with the hosted profiler DLL, written in C++ and named the Visual Profiler Backend, that is loaded by the CLR and accessed by the COM interface, as described in the chapter 3. The sampling and tracing profilers are implemented as distinct COM CoClasses and only one of them can run at a time. The CoClasses share the common code base for the profiling data collection and the interprocess communication with the managed code.

The second process is programmed in C# and is called the Visual Profiler Access. It starts the profiled application, written in C++, with the desired profiling mode in a separate process, initialize the interprocess communication for transferring the profiling data and it is responsible for the data processing so the data can be later used by an analytic and UI frontend. The whole relation is shown in the figure 4.1.

Both profiling modes are implemented in the Visual Profiler Backend part. It also contains bunch of supporting classes for handling metadata information for methods, classes, modules and assemblies, for caching the profiling intermediate results and for serializing the results and transmitting them over IPC<sup>1</sup> to the Visual Profiler Access part. The Active Template Library (ATL) framework from Microsoft is used to simplify programming of the Component Object Model (COM) in C++.

## 4.3 Implementations of ICorProfilerCallback3 interface

The `ICorProfilerCallback3` interface is the center point of the Profiling API. With its 80 methods it is a little bit impractical to implement, mainly because only a handful of the methods is usually made use of. For that reason we created a default class

<sup>1</sup>inter-process communication

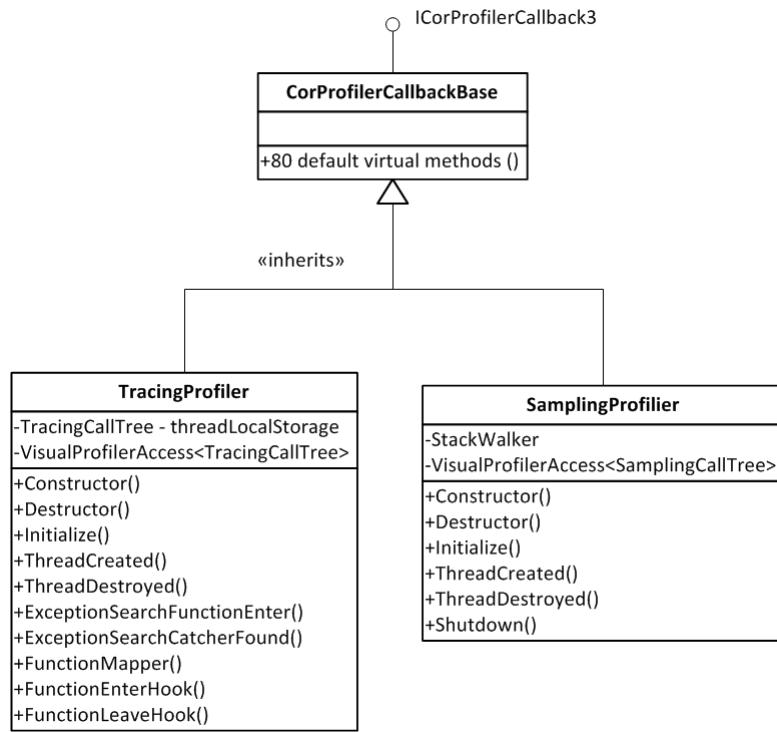


Figure 4.2: Implementations of the *ICorProfilerCallback3* interface

*CorProfilerCallbackBase* as depicted in the figure 4.2. This provides empty implementation of every method of the interface and derived classes can only override methods we need. This idea makes it easier to have clearly arranged classes and meet the clean code rules.

#### 4.3.1 TracingProfiler class

This class is home for the tracing profiler mode implementation, see figure 4.2. It starts its lifetime by running the IPC with the Visual Profiler Access on a separate thread in the *Constructor()* method. After that, the *Initialize()* method is invoked by the CLR and receives a pointer to the *ICorProfilerInfo3* interface. Here the profiler registers the function ID mapper (more details later in this subsection), the function enter/leave, the thread created/destroyed and the exception events notifications. At this point everything is set up and the profiled application starts running.

---

During the profiling session there is need to trace the method enter and leave notifications for every managed thread simultaneously. This is a demanding task. If you realize that every single call to and return from a managed method is augmented by overhead of this notification.

Our first approach to storing the collected profiling information was to push the ids of managed functions, into an enter-stack and a leave-stack. Every managed thread had its own private enter and leave stacks. All stacks were saved by the corresponding thread ids

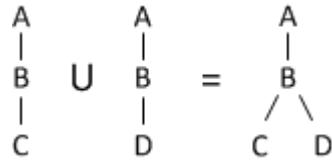


Figure 4.3: Union of thread call stack snapshots forming a call tree. A very simple example: A calls B, B calls C, C returns to B, B calls D, D returns to B, B return to A

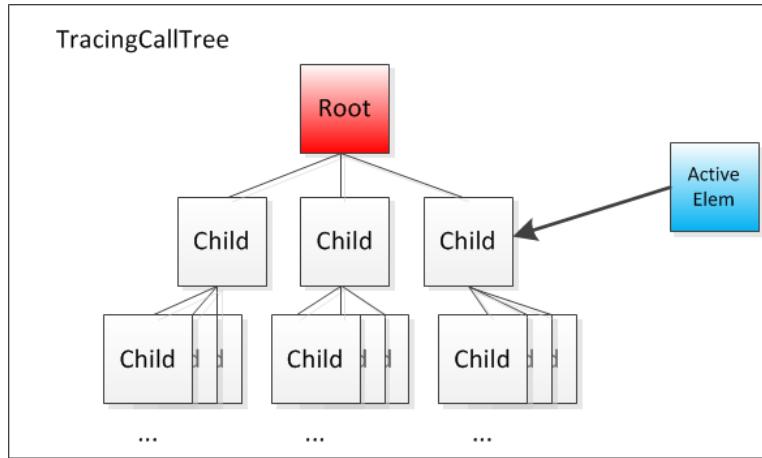


Figure 4.4: A simplified representation of the `TracingCallTree` class. The active element pointer points to the currently executing function element with the call tree hierarchy.

in a common hash table. During the processing of a function enter or leave notification the hash table had to be searched. This created a potential race condition (inserting a new thread to hash table and reading from it at the same time). The access to the hash table required synchronization.

That was fairly easy to implement and did indeed work correctly. However, it did not perform very well. Firstly, it used so much memory that after tens of seconds the application memory working set reached hundreds of mega bytes. Secondly, the synchronized hash table lookups slowed down the application. The profiled application was running approximately 50 times slower! (according to the rough duration measurements in the profiled application). It was clearly unacceptable. A better solution had to be found.

Subsequently, we realized that an union of snapshots of thread call stacks forms a call tree as depicted in the figure 4.3.

The `TracingCallTree` class was created based on this idea. A simplified representation of the `TracingCallTree` class is depicted in the figure 4.4. Every tree element (class `TracingCallTreeElem`) contains cumulative profiling results such as the number of entering, leaving and timing related data and some other auxiliary data. The object of the `TracingCallTree` class contains an active element pointer and a root pointer. The active element pointer points to an element representing a currently executing function and the root element pointer represents the top most method, e.g. the Main method.

When a method calls another method, represented by a child of the tree node, the active element pointer only moves to the child and the profiling data is modified. This solution

performs very well. Even a huge call tree takes negligible amount of memory and the traversals are very quick.

The other improvement was in using a thread local storage of managed threads for storing a reference to call trees. Thanks to this the lookup in the call tree hash table can be avoided in thread-safe way with significant speed gain.

The change from a stack to a call tree and usage of the thread local storage brought together a huge speed and memory performance enhancement. The profiled application runs now 4 times slower (compared to 50 times before) then a non-profiled application. This slowdown is valid only for the non-optimized build of the profiler (debug settings), the optimized version performs even better.

---

Let us move to the activity of the profiler. The profiler receives a notification from the CLR whenever a new managed thread is created - via the method `ThreadCreated`. The notification executes within the context of the newly created thread. The tracing profiler registers the thread id and associates an instance of the `TracingCallTree` class with it. Such association is saved into a hash table, represented by C++ `std::map`, under the thread id key. The reference to the `TracingCallTree` object is stored into the thread local storage in order to avoid repetitive search in the hash table by every single enter/leave notification.

After that a managed method in the profiled application is about to be entered. But before it is actually entered the CLR gives a chance to the tracing profiler to express its interest in this particular method. The CLR invokes the `FunctionIdMapper` method and passes it a function id (type `FunctionID`) of the method. The profiler caches the method's metadata (the name, the declaring...) obtained via the function id and if the method is from an assembly with enabled profiling, as mentioned in the section 4.6, the profiler expresses its interest in receiving the enter/leave notifications for the method, or hooks to it, by setting an output parameter of the `FunctionIdMapper` to `true` value. The function id mapping occurs only once for every managed method.

If a hooked managed method is being entered, the CLR calls the profiler's function-enter handler and passes to it the same function id as to the `FunctionIdMapper` method. This notification is very performance sensitive (very repetitive) and therefore the function-enter handler is declared with the naked attribute (`_declspec(naked)`), the compiler generates code without prolog and epilog code. We had to write own prolog/epilog code sequences using inline assembler code to call yet another function that makes a downward transition on the thread's tracing call tree.

When a function is being left, the profiler's function-leave handler is called and again receives as an input parameter the function id. The function-leave handler has to be declared with the naked attribute as well. It just calls another function to make an upward transition on the thread's tracing call tree.

If a thread of the profiled application ends its execution without any exception, the CLR calls the `ThreadDestroyed` method and lets the tracing profiler to do some book-keeping and to close the corresponding tracing call tree.

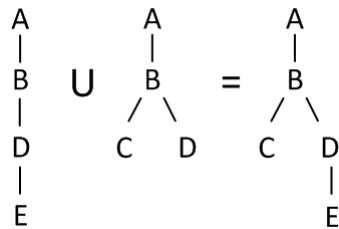


Figure 4.5: Merge of a stack snapshot with a thread’s sampling call tree.

If an exception occurs on a thread then a search for a `catch` clause begins by walking up a thread’s call stack. The CLR invokes the `ExceptionSearchFunctionEnter` for every searched function on the thread call stack until it finds the matching `catch` clause and calls the `ExceptionSearchCatcherFound` function. The tracing profiler uses this exception handling mechanism to transition upwards the corresponding tracing call tree together with the thread’s call stack. If the `catch` clause was not found the profiled application would crash due to an unhandled exception.

After the profiling application’s exit, the IPC sends out the final profiling data (this happens in the `Destructor` function) and it is ended. At this moment the profiling session is considered to be over.

### 4.3.2 SamplingProfiler class

In this class resides the profiler implementing the sampling profiler mode, shown in the figure 4.2. The sampling profiler starts by connection the IPC communication from its `Constructor` method on a separate thread. The CLR calls after that the `Initialize` method and passed a pointer to the `ICorProfilerInfo3` interface to it. The sampling profiling sets the mask to `COR_PRF_MONITOR_THREADS` and `COR_PRF_ENABLE_STACK_SNAPSHOT` values in order to receive threads related notifications and to enable managed call stack’s snapshots. An instance of the `StackWalker` class is also created to carry out the exploration of the managed thread stacks. The sampling is being started.

In this moment it is the right time to start the profiled application. When a managed thread is created, it is registered with the stack walker, in the `ThreadCreated` notification handler fired by the CLR.

To collect the profiling information the sampling profiler uses an object of the `SamplingCallTree` class, an alternative to the `TracingCallTree`, for every managed thread. Every time a stack snapshot is created it is added to the sampling call tree of the corresponding thread, as indicated in the figure 4.5.

The constructor of the `StackWalker` class, as shows the figure 4.6, accepts the sampling period in milliseconds as its parameter. When the method `StartSampling` is called, the stack walker creates a new thread and periodically gets the stack snapshots for every managed thread.

To get the stack snapshot, the stack walker calls the `ICorProfilingInfo3::DoStackSnapshot` method for every registered managed thread and passes it the manage thread id (type

StackWalker
-SamplingThread
+RegisterThread()
+DeregisterThread()
+StartSampling()
+Sample()
+DeregisterAllRegisteredThreads()
+MakeFrameWalk()

Figure 4.6: The StackWalker class

`ThreadID`) of the thread, whose stack should be explored. It also passes a pointer to the function `MakeFrameWalk` that is called for every single stack frame on the thread's call stack to the `DoStackSnapshot`. Within the `MakeFrameWalk` function, the stack walker records function ids of the encountered stack frames by putting them in a linear collection, C++ `std::vector`. As soon as the last stack frame is explored by the `MakeFrameWalk` function the `DoStackSnapshot` returns. The stack walker merges the just acquired stack snapshot with the sampling call tree, caches the metadata of methods and repeats the same process for other registered threads.

The CLR suspends the inspected thread during a call to `DoStackSnapshot` function. Therefore ,the `MakeFrameWalk` function has to be fast to minimize the performance overhead on the profiled application.

When a managed thread is about to end its execution the CLR calls the `ThreadDestroyed` notification handler and the thread's id is unregistered from the stack walker.

The stack walker is not allowed to perform its stack walks after the profiling session is over. Otherwise, an error during reading the metadata occurs. In order to avoid this, the sampling profiler overrides the `Shutdown` method of the `ICorProfilerCallback` interface. This method is called by the CLR right before the end of the profiling session and the sampling profiler blocks the CLR's thread until the last stack walk is completed. After that the application terminates, the collected data is send out and the IPC finishes together with the profiler.

## 4.4 Call tree and call tree element classes

The classes of the tracing and the sampling call trees and call tree elements share many features and properties, mainly regarding the initialization, the tree structure, the storing of the collected profiling data and thread synchronization. This led us to create the common abstract classes `CallTreeBase` and `CallTreeElemBase`. The inheritance hierarchy is shown in the figure 4.7

Both base abstract classes are templated. The template parameters are to be specified by inheriting the abstract base classes, as shown in the code snippet below. It might seem a bit strange that the template parameters are of the types that inherit the base class. However, this allows us to put the common generic logic that only differs in the templated types in one place and to specify the type later by inheriting the base class.

```
class TracingCallTree : public CallTreeBase<TracingCallTree, TracingCallTreeElem>
```

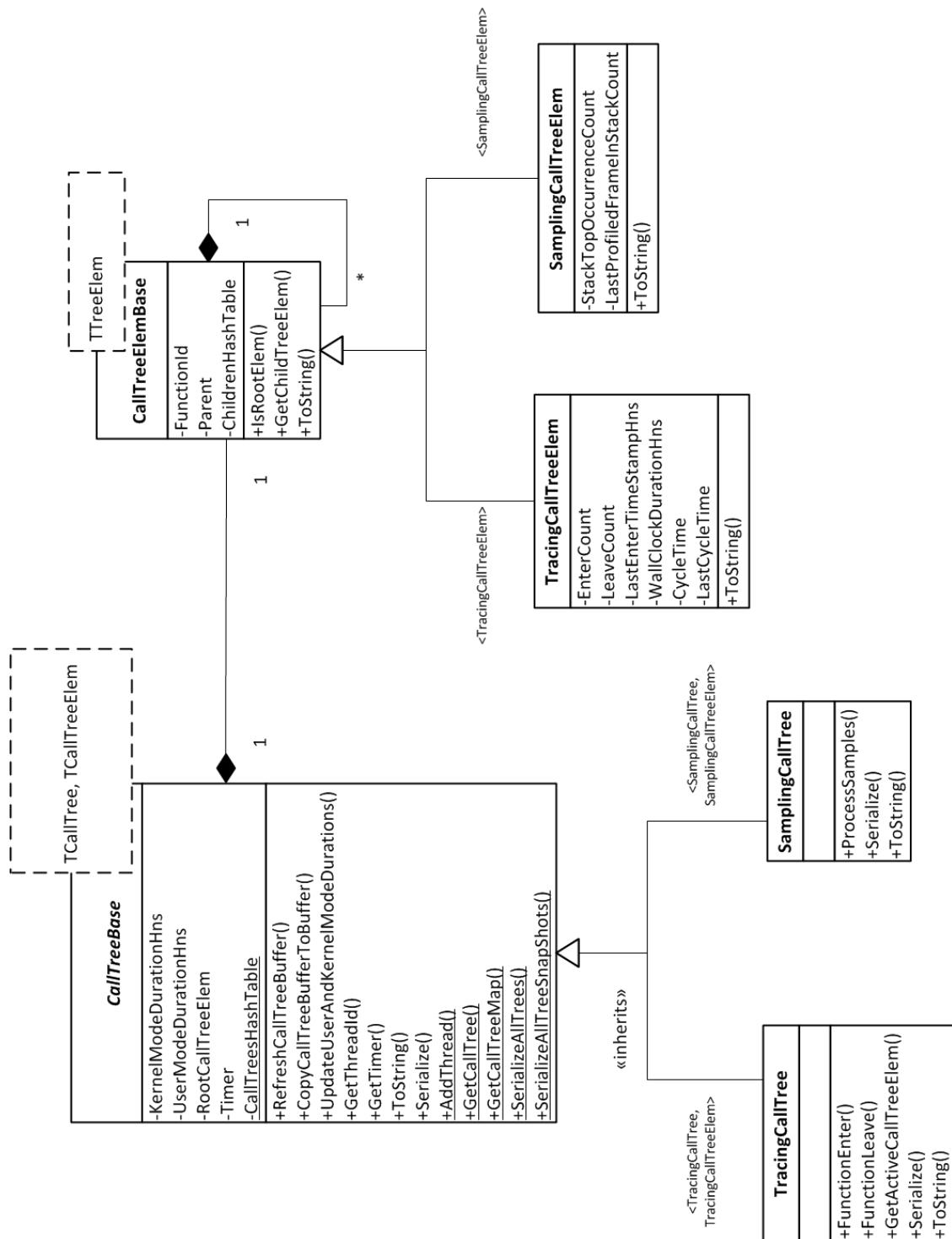


Figure 4.7: The hierarchy and relation between the call tree and the call tree element classes used to trace the collected profiling data.

```
class SamplingCallTree : public CallTreeBase<SamplingCallTree, SamplingCallTreeElem>
class TracingCallTreeElem: public CallTreeElemBase<TracingCallTreeElem>
class SamplingCallTreeElem : public CallTreeElemBase<SamplingCallTreeElem>
```

The `CallTreeBase` class defines a static hash table for storing the call trees by their ids, it provides a handful of static synchronized methods to add, get and serialize the content of the static hash table. In addition, it defines instance methods and fields for distinct purposes such as thread time related measurement, the root call tree element pointer, the abstract serialization method and other auxiliary members.

The `CallTreeElemBase` class is a plain data class and holds a function id, a pointer to the parent element and a hash table, C++ `std::map`, to hold pointers to children elements.

The derived classes `TracingCallTree`, `SamplingCallTree`, `TracingCallTreeElem`, `SamplingCallTreeElem` then only define their special behavior and data.

## 4.5 Metadata

We have referred to the term metadata in the preceding chapter without specifying it. The metadata is information stored in .NET assemblies describing their data types, identity, relations with other assemblies, security permissions, attributes and other additional information.

The `ICorProfilingInfo3` and `ICorProfilingCallback3` interfaces provide mostly only opaque identifiers for the methods, classes, module, assemblies, parameters and attributes. The way to get the corresponding metadata is to convert the identifiers of a metadata token to corresponding metadata. The `ICorProfilingInfo` provides a few methods to achieve such conversion. Once the metadata token is available the `IMetaDataImport` comes in handy [Mic11a] to acquire whatever information about the metadata, pretty much everything what is available in the .NET `System.Type` namespace. The `ICorProfilingCallback3::GetTokenAndMetadataFromFunction` and other functions return a reference to the `IMetaDataImport` object through the output parameter.

Our strategy is to cache metadata, classes, modules and assemblies of methods by ids. The profiler in its both mutations always infers the metadata from a function id identifier. The tracing profiler does that in the `FunctionIDMapper` function and the sampling profiler right after the `DoStackSnapshot` function returns.

A similar inheritance hierarchy as by the `CallTreeBase` class and its derived classes is employed by the metadata. There is a templated `MetadataBase` class. It takes two template arguments `TId` and `TMetadata`. The `TId` is the opaque identifier used in the `ICorProfilerCallback3` interface, e.g. the `FunctionID` or the `AssemblyID`. The `TMetadata` is the deriving type. The `MetadataBase` class defines a static cache, C++ `std::map`, for each templated class and static methods to manipulate the cache. The `MetadataBase` class inherits from the `MetadataSerializer` class adding the serialization capabilities. The entire hierarchy is shown in the figure 4.8.

The `MethodMetadata` class has a member that points to the `ClassMetadata` class, that in turn points to the `ModuleMetadata` class and this points to the `AssemblyMetadata`

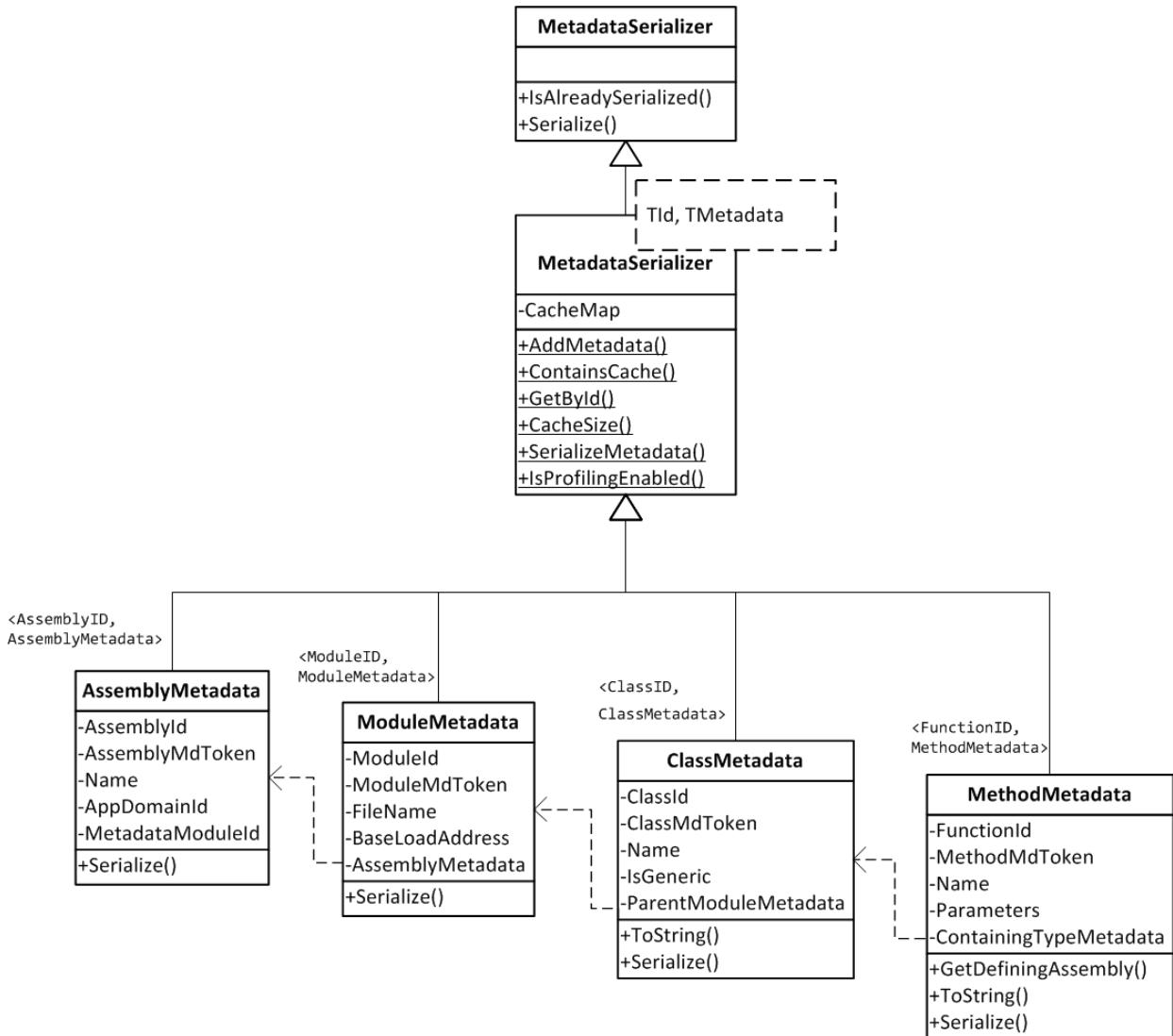


Figure 4.8: The hierarchy and relation between the metadata classes used to store .NET metadata.

class. Thus, it is possible to get metadata for each function id.

## 4.6 Profiling enabled assembly

Even a very small .NET console application loads hundreds of methods. If the profiler profiled every method of .NET assemblies in the profiled application it would impose huge overhead on the entire application without any direct benefits. The profiling of the .NET functions would not provide much useful information about the hot spots - places where to start tuning up the profiled application.

Just to provide an example of the immense effect of the filtering. When we profiled our Mandelbrot test application, see the chapter 7, with the disabled filtering the profiler recorded over 6000 unique method invocations. A log file, where the profiler was dumping text representation of trees, had 17,5 MB. With the enabled filtering the log had only 13 KB with 32 methods invoked. The main contributors to those 6000 methods were the method from the `mscorlib` and the `System.Windows.Forms` assemblies. Many of the 6000 methods were called only once.

This is why the filtering on the assembly level was introduced. An assembly is profiling enabled if it has the `[assembly: VisualProfiler.ProfilingEnabled]` attribute applied. If a function id, actually its respective method, does not belong to a profiling enabled assembly then the profiler does not bother with tracking its performance data.

## 4.7 Measuring profiling data

### 4.7.1 Tracing profiler

The tracing profiling is very accurate. It collects the method's enter/leave count, the wall-clock time and the user and kernel mode time. For this reason, it uses system functions to measure the wall-clock time, the user time and the kernel time. Namely the `GetThreadTimes`, the `QueryThreadCycleTime` and the `GetSystemTimeAsFileTime` system functions.

The `GetThreadTimes` function returns the amount of time that the thread has executed in user and kernel modes via its output parameter. However, the results are within 15 ms tolerance which is too much for measuring the duration of individual functions. Therefore, in order to minimize the error of the measurement the user and kernel modes time is always measured cumulatively for a thread tracing call tree as the whole and then divided among its tracing call tree element.

The tracing call tree element uses the `QueryThreadCycleTime` function, which is very precise in comparison with the `GetThreadTimes` function, to determine the number of CPU clock cycles used by its corresponding function. This value includes cycles spent in both user mode and kernel mode. The number of cycles is later used to distribute the cumulative user and kernel mode times in the tracing call tree measured by the `GetThreadTimes` function and so the measurement error is hindered in this way.

Every tracing call tree element is self responsible for the wall-clock time measurement

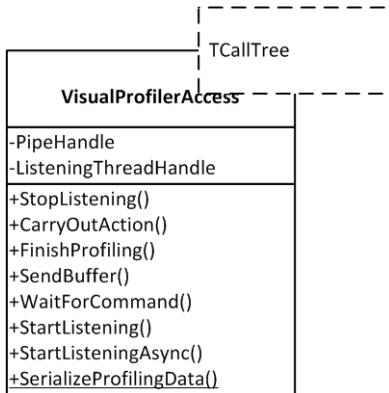


Figure 4.9: The `VisualProfilerAccess` class encapsulates the communication with the frontend

and uses the `GetSystemTimeAsFileTime` function for that.

The method's enter/leave count is measured by incrementing counters by every function enter/leave callback.

#### 4.7.2 Sampling profiler

The sampling profiler is purely stochastic. During a stack walk, it collects how many times a method occurred on top of the thread call stack and how many times it was the last top most profiled enabled method, which means the method called other non-profiled methods.

The sampling call tree keeps track of the wall-clock time and the user and kernel mode time for its corresponding managed thread using the `GetThreadTimes` and the `GetSystemTimeAsFileTime` system functions.

The measured values are later distributed to determine the duration of the profiled methods.

### 4.8 Sending the profiling results

The Visual profiler backend runs in a different process than the analytic frontend with a user interface. Therefore the profiling data is transmitted to the frontend process using the inter-process communication (IPC), the named pipes.

A named pipe is created by the visual profiler access before the start of a profiling session, see figure 4.1. The name of the named pipe is handed over in the process “`VisualProfiler.PipeName`” environmental variable. The communication logic is enclosed in the `VisualProfilerAccess` class, see figure 4.9. Both the tracing and the sampling profiler start in their constructors to listen to the named pipe on a separate thread and end the communication in their destructors.

The instance of the `VisualProfilerAccess` class then waits for a command from the frontend and answers with an appropriate action. The enumeration of the actions and

SerializationBuffer
+Clear()
+SerializeMetadataId()
+SerializeThreadId()
+SerializeFunctionId()
+SerializeMdToken()
+SerializeUINT()
+SerializeBool()
+SerializeWString()
+SerializeDebugString()
+SerializeMetadataTypes()
+SerializeProfilingDataTypes()
+SerializeCommands()
+SerializeActions()
+SerializeULLONG()
+SerializeULONG64()
+CopyToAnotherBuffer()
+Size()
+GetBuffer()

Figure 4.10: The `SerializationBuffer` class members

commands follows. Their names are self-describing.

#### Actions:

- `SendingProfilingData` - the bytes of the profiling data are being sent to the frontend
- `ProfilingFinished` - the profiled application exits

#### Commands:

- `FinishProfiling` - the request to finish the profiled application
- `SendProfilingData` - the request to send profiling data

### 4.8.1 Serializing the profiling data

As presented earlier in this chapter, every call tree and call tree element is capable of serializing its states. We created the `SerializationBuffer` class to aid this task. This class manages a byte array and provides methods for copying of numerous types into the byte array, see figure 4.10.

The actual serialization occurs when the `SendProfilingData` command is received or when the profiled application ends.

Firstly, the metadata are serialized incrementally - only the metadata that has not yet been sent in any previous requests are serialized. This decreases the number of the resulting bytes. On the other hand, the serialized call trees and call tree elements are always fully serialized and sent as they are, see figure 4.11.

### 4.8.2 Live updates

The visual profiler supports a live update feature. It allows viewing the intermediate profiling results while the profiling is running.



Figure 4.11: The structure of the serialization buffer

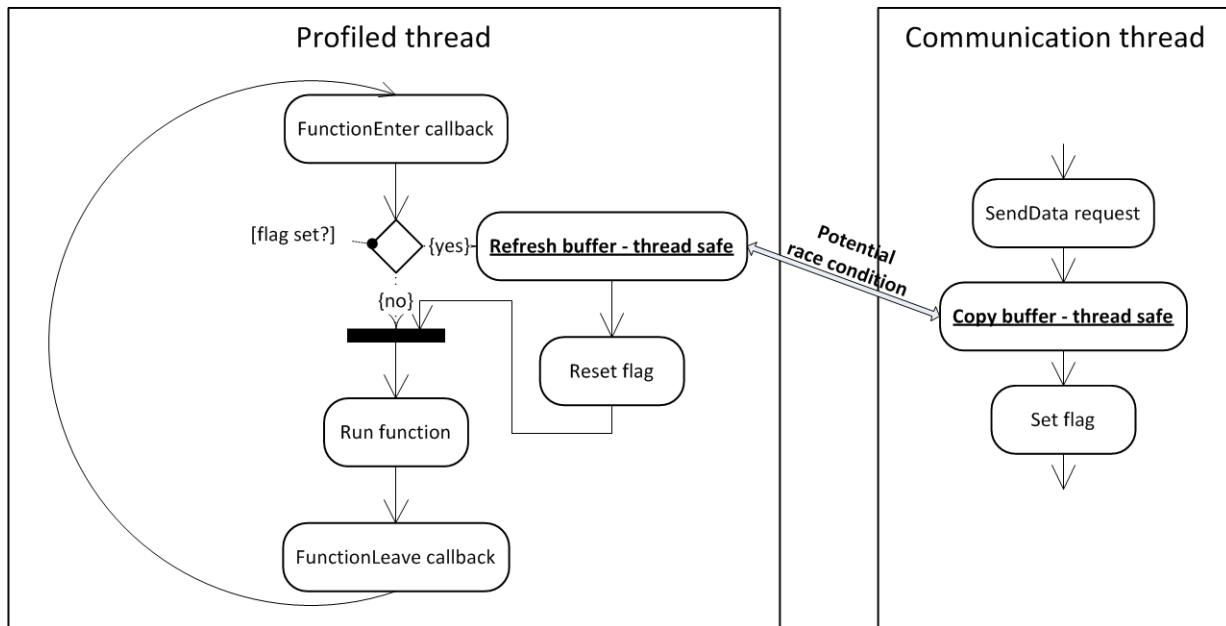


Figure 4.12: The live update activity diagram: cooperation between the tracing profiler and the live update - simplified example

Due to the multi-threaded nature of the profiler and possible race conditions, it is not possible in the course of the profiling session just to access an arbitrary call tree and its call tree elements and perform the serialization. Therefore, an on-demand snapshot mechanism was adopted. It works as follows and shown in the figure 4.12.

Every call tree has its own buffer, instance of the `SerializationBuffer`. After the profiling data have been updated (after the enter/leave notification or the stack walk) the call tree's `RefreshCallTreeBuffer` bool flag is checked and if the flag is set the call tree's metadata is serialized to the call tree's buffer and the flag is reset.

When the frontend sends the `SendProfilingData` command, the buffers of all call trees are collected and sent out to the frontend and the `RefreshCallTreeBuffer` flag is set again. The whole process repeats. However, the buffers were refreshed for the last time before the last `SendProfilingData` command was received - they are therefore not the most recent representation of runtime conditions. This is acceptable since the data is only a preview of the profiling result, not the real results.

Thanks to this we can provide a lightweight and non-intrusive update mechanism with very low overhead.

## 4.9 Summary

In this chapter, we have presented several reasons for the choice of the tracing and sampling profilers and the software architecture of the Visual Profiler Backend was outlined. Afterwards, we explored how both profilers were implemented, how the metadata is acquired and stored, what and how is measured and, finally, how the profiling data is sent to the frontend through a named pipe.



# Chapter 5

## Visual Profiler Access

The Visual Profiler Access is a bridge between the unmanaged and the managed worlds of the profiler. Its main role is to initiate the bidirectional inter-process communication (IPC), to start the profiled process and to prepare the received data to be passed further on.

### 5.1 Metadata, call trees and call tree elements

The data structure in the Visual Profiler Access is similar to the data structure of the Visual Profiler Backend, described in the chapter 4. For the matter of completeness we include the metadata, call trees and call tree elements inheritance hierarchy respectively in the figures 5.1 and 5.2

The metadata derived classes use caches for every implementation of the base `MetadataBase` class. The caches are instances of the `System.Collections.Generic.Dictionary<uint, TMetadata>` class. The key to the dictionary is the metadata id. The metadata are added to the cache as they come from the backend.

The call trees and their call tree elements are not cached, however, they are matched with their corresponding cached metadata during the deserialization.

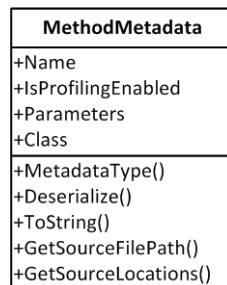


Figure 5.1: The metadata inheritance hierarchy closely resembles its counterpart from the chapter 4.

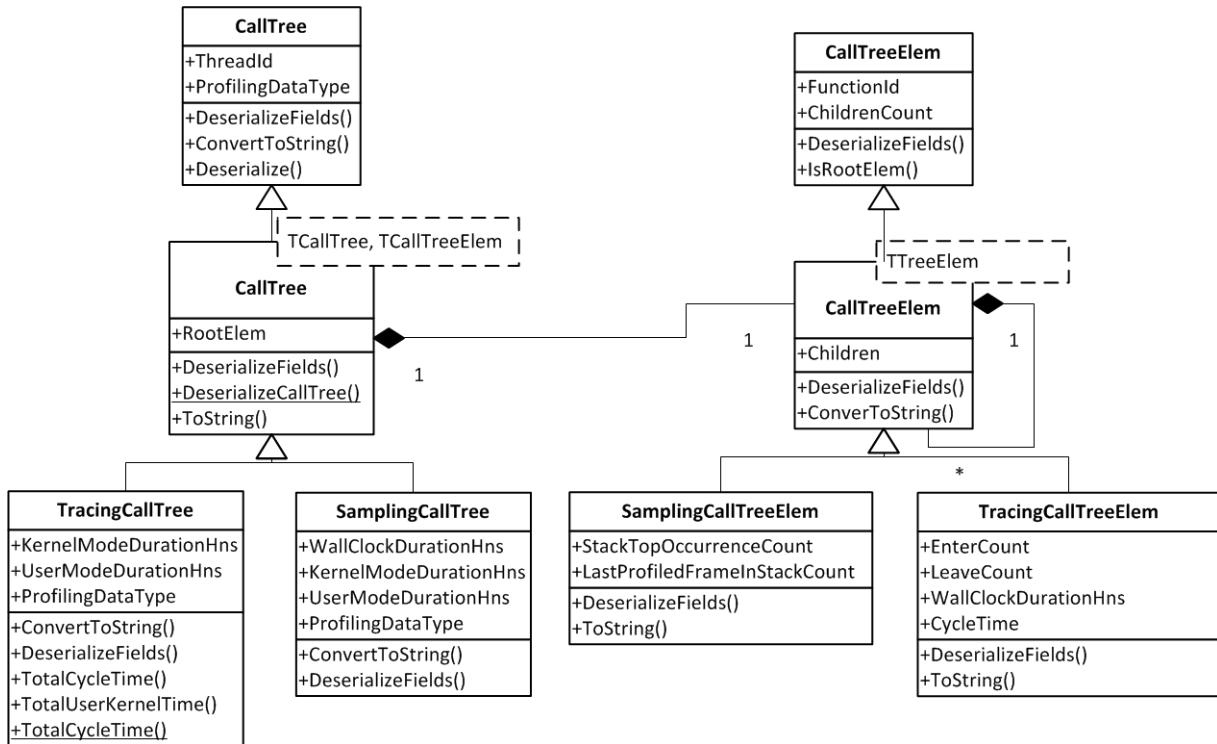


Figure 5.2: The call tree and call tree element inheritance hierarchy closely resembles their counterparts from the chapter 4.

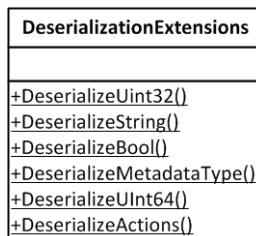


Figure 5.3: The `DeserializationExtensions` class, which adds extension methods to the `System.IO.Stream` class in order to provide serialization of the basic types.

## 5.2 Deserialization

The profiling data comes from the named pipe in form of a byte stream. We added a handful of extension methods to the stream, the `System.IO.Stream` class, to deserialize the basic types, as depicted in the figure 5.3. The base classes of the call tree and the call tree elements classes define an abstract method for deserialization. This method had to be overridden in inherited classes to specify the deserialization logic that corresponds to the logic of the serialization in the backend.

## 5.3 Visual profiler access API

The entry point to the profiler is the `ProfilerAccess<TCallTree>` class, see figure 5.4. To create an instance of this class you have to:

1. specify the `TCallTree` generic parameter, the type of a call tree that will be used to store profiling data,
2. provide an instance of the `System.Diagnostics.ProcessStartInfo` class that specifies the executable of the profiled application,
3. pass one value of the `ProfilerTypes` enum that represents which profiler type to use in the backend,
4. supply the update period for the result fetching and
5. pass a reference to the `EventHandler<ProfilingDataEventArgs<TCallTree>>` delegate, which is called back when a profiling data update is successfully carried out.

When the `StartProfiler` method is called a bidirectional named pipe with asynchronous writing/reading is created. After that, two separate threads are spawned, the first - the command thread - sends commands to the backend and the second - the action thread - listens for actions from the backend. The actions and commands are as follows:

### **Actions:**

- `SendingProfilingData` - the bytes of the profiling data are being sent to the frontend
- `ProfilingFinished` - the profiled application exits
- `Error` - an unexpected situation

### **Commands:**

- `SendProfilingData` - the request to send profiling data

Both the action and the command are only the counterparts to the actions and commands in the backend.

The command thread sends in a specified period the `SendProfilingData` command to the backend. The action thread listens for the actions. When the `SendingProfilingData` action is received, the action thread deserializes the containing profiling data and passes them as the event argument to the specified callback delegate and executes them on a separate threadpool thread. This process is repeated until the profiling application exits.

The `ProfilerAccess` class offers the `Wait` method that blocks a calling thread until the profiling is completed. This can be used to prevent a main application thread to exit prematurely.

The `ProfilerAccess` class is responsible for the management of the communication, whereas `ProfilerCommunicator` class handles the logic of the communication. This decoupling of responsibilities aids the unit testing.

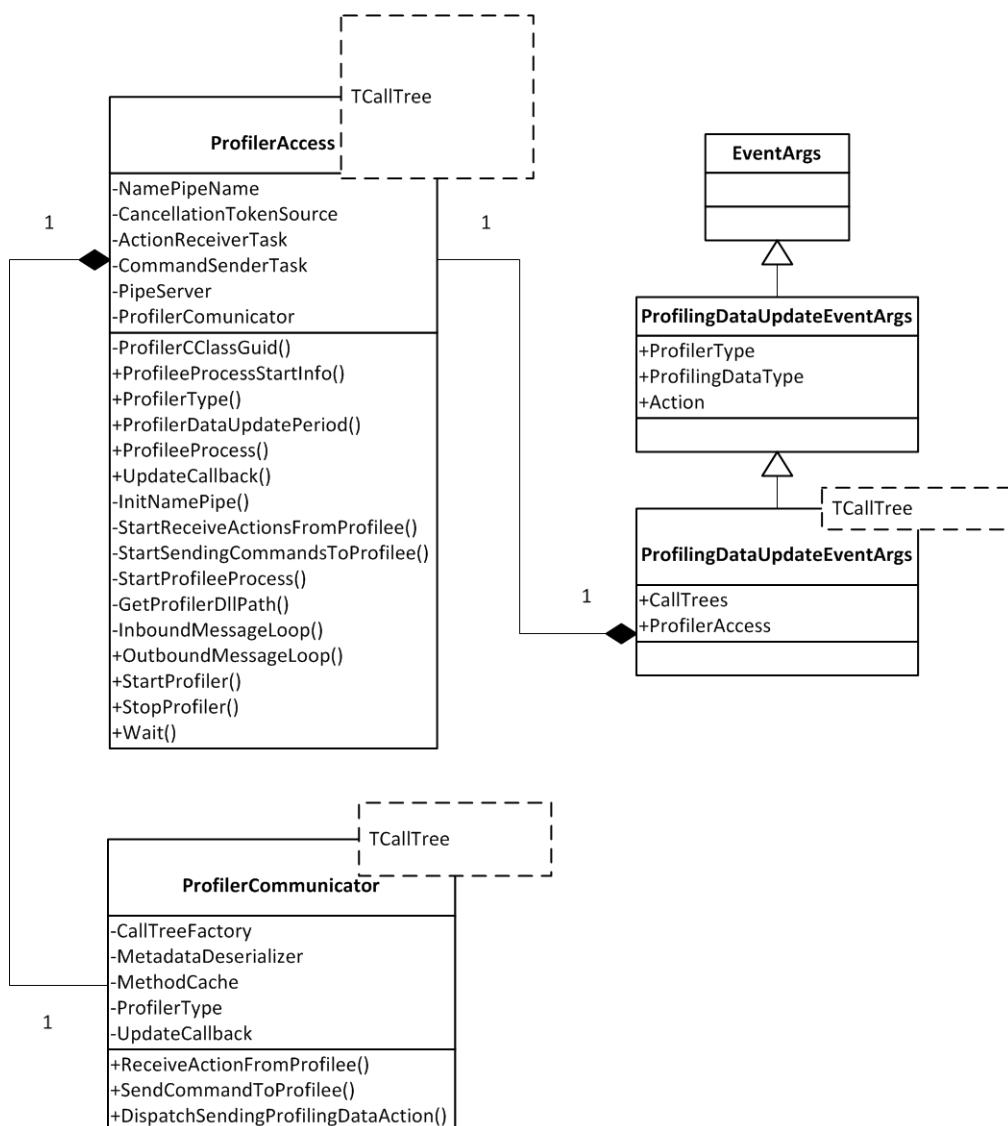


Figure 5.4: The communication logic classes

## 5.4 Source code locations

Finding an exact position of profiled methods in source code is required for the projection of the profiling result to the Visual Studio code editor.

The data regarding source code, symbols, such as names of local variables and line numbers, are not stored in the compiled modules, because it would take too much space. During compilation a PBD<sup>1</sup> file is generated. Its format is proprietary and it is mainly used for debugging purposes. The PDB files also store exact line numbers for methods.

There is an unmanaged API from Microsoft called Debug Interface Access for reading PDB files. This API is somewhat verbose and according the samples it seems not easy to use. Luckily, Microsoft also runs an open source project called Common Compiler Infrastructure (CCI). The project is a set of libraries and APIs that allow analyzing or modifying .NET assemblies, modules and debugging PDB files. The CCI is actually a set of more projects, each targeting different areas. One of these projects is the CCI Metadata. It provides an API, among others, for reading and writing PDB files. The API needs for its correct functionality a valid PDB file for each managed assembly.

We wrapped this API into two functions that accept metadata tokens of a method and return a path to a source code file and positions in it. The metadata from the profiler backend can easily be traced in the source code.

## 5.5 Summary

In this chapter, we have looked at the `ProfilerAccess<TCallTree>` class and highlighted the most important parts of it. The class is more or less a wrapper for the unmanaged functionality of the Visual profiler backend. The source code to method mapping was introduced together with the notion of the PBD files.

---

<sup>1</sup>Program database, extension .pdb



# Chapter 6

## User Interface and Visual Studio Integration

This chapter explains visual presentation of profiling results in the source code editor and integration of the profiler into Visual Studio.

### 6.1 Visually mapping profiling results to source code

The profiling results are usually presented in other profilers in great detail by tables, graphs and tree-structures. The difficulty in reading the data is coping large volumes of data that they present. We believe, developers would benefit from briefer overview mapped to context of source code.

S. G. Eick and J. L. Steffen presented in [ES92] a method for visualizing line oriented profile data. It displays files as rectangles and code as colored lines within each rectangle. Color of lines is determined base on profiling metrics. They claim that their technique allows programmers to discover usage patterns in the code that would be impossible to find using traditional methods. They implemented their ideas in a software tool Seesoft, see figure 6.1.

We followed and extended their idea and proposed and implemented a method level coloring and an overview of results based on profiling results from the Visual Profiler.

### 6.2 Source code coloring and bird's-eye view

To present the profiling results in Visual Profiler, colored rectangles are laid over methods directly in the source code editor as depicted on an example in the figure 6.2. Such approach provides direct feedback to developers and reveals the “hot-spots” in direct relation with the code in very clear way. The color of the overlay represents significance of methods in context of profiling metrics.

The coloring of the source code provides great local awareness of the profiling result but says nothing about the overall results of a profiling session.

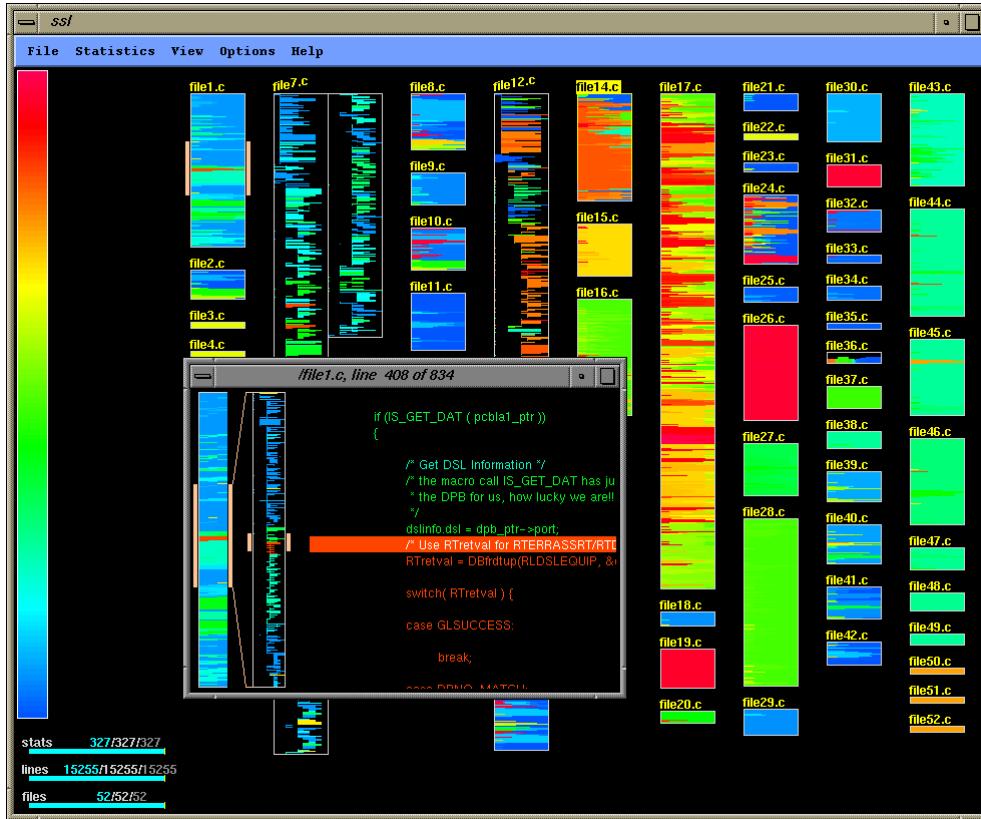


Figure 6.1: Seesoft - mapping each line of code into a thin row, colored according to a statistics of interest and showing the context in a preview window.

```
private static void RunApplication()
{
    Form f = new Form();
    Application.Run(f);
}
```

Figure 6.2: Laying a colored rectangle over the source code. The color hue of the rectangle corresponds to a measured value of the method.

We addressed this problem by introducing a "bird's-eye" view of source code with applied coloring. The bird's-eye view consists of columns of rectangles. The columns represent individual source files and the rectangles individual methods. The height of the rectangles corresponds to amount of source code lines, their order in the column reflects their position in the source file and their color is set according to profiling results. When the mouse cursor hovers over a rectangle the detail information for the corresponding method is displayed. The whole situation is illustrated in the figure 6.3.

The direct source code overlay was missing a possibility to get more exact information about the profiling result and the rectangles from the bird's-eye view were lacking a linkage to source code. So we connected them. When the mouse hovers over a source code overlay, a corresponding rectangle is highlighted and details are displayed in the bird's-eye and when a rectangle in bird's-eye view is clicked, a corresponding source file is opened with the right method highlighted.

The bird's-eye view, in addition, joins all methods over all source file and sorts them according to their measured values and presents them above the source file columns as a row of squares. Color and behavior of the squares follows the same rules as by the rectangles.

Switching between different profiling modes may be accomplished by radio buttons in the top right corner of the bird's-eye view.

The bird's-eye view with source code overlays form together a very solid and conveying way to display profiling information.

We also incorporated a call-graph exploration feature. It allowed seeing relations among methods calls. But in the end we took it away, because the UI started to become clogged and it did not bring much value to understanding the profiling results.

The final UI is a result of many design iterations and mock-ups.

## 6.3 Implementation of the user interface

The user interface is built in WPF<sup>1</sup> framework [Nat10] and follows the MVVM<sup>2</sup> [BB11] UI-design pattern. Let us now briefly introduce WPF and MVVM.

### 6.3.1 WPF

The Microsoft Windows Presentation Foundation is the latest framework (as in the end of 2011) for creating user interfaces with rich user experience. It is part of .NET framework and brought new ideology in composing visual components and their functionality. Its main features are depicted in the figure 6.4.

WPF is vector based, resolution independent and hardware accelerated. It combines ordinary UI components, 2D graphics, 3D graphics and multimedia.

Visual parts of UI components are defined in an XML based language called

---

<sup>1</sup>Microsoft Windows Presentation Foundation

<sup>2</sup>Model-View-ViewModel

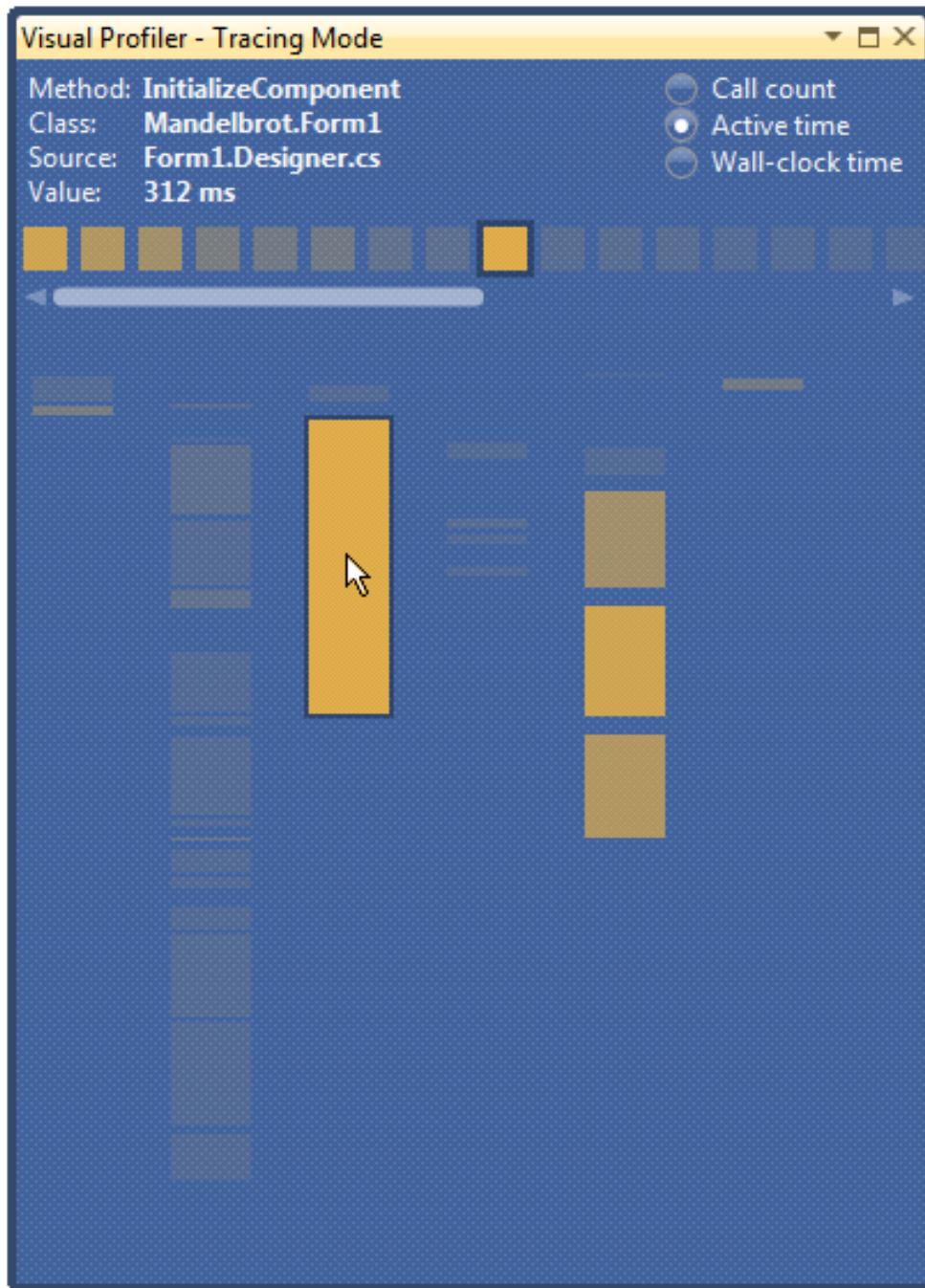


Figure 6.3: bird's-eye view representation of source files represented as columns and methods represented as rectangles with a highlighted method and detailed information.

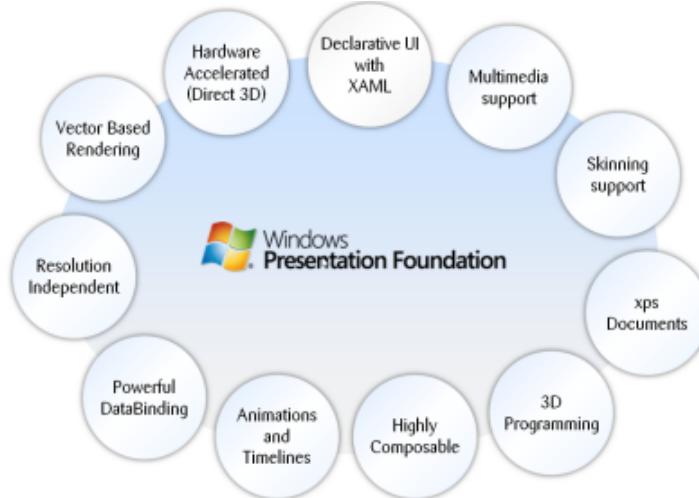


Figure 6.4: Overview of the main new features of WPF [Mos11]

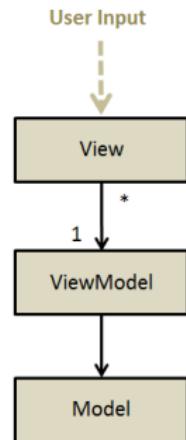


Figure 6.5: Schema of the MVVM pattern [Mos11]

XAML<sup>3</sup> and the behavior is implemented in a managed programming language. The code and XAML use databinding, commands and events to communicate. This architecture results in a high separation of appearance and behavior.

### 6.3.2 Model-View-ViewModel

This UI pattern, see figure 6.5, was introduced with the advent of WPF and is very often used to build low-coupled applications. It consists of a model that is not aware of the other parts, of a view that manages user input and sends it by commands to a view-model and of the view-model that implements program logic. The view gets data from the view-model through databinding.

---

<sup>3</sup>eXtensible Application Markup Language

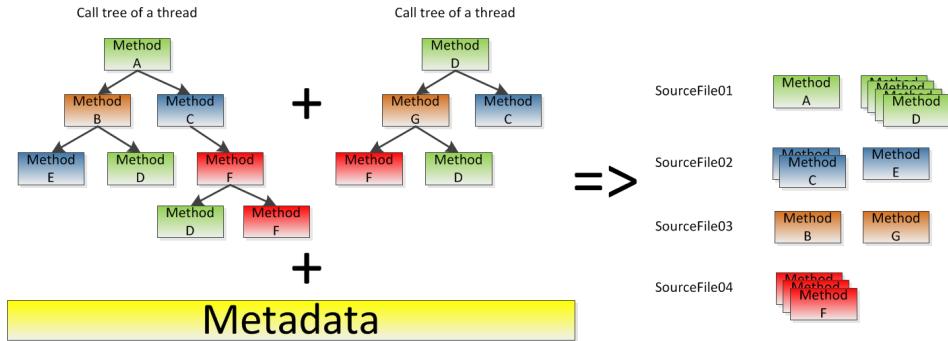


Figure 6.6: Call trees and metadata are merged to form source file groups with aggregated methods.

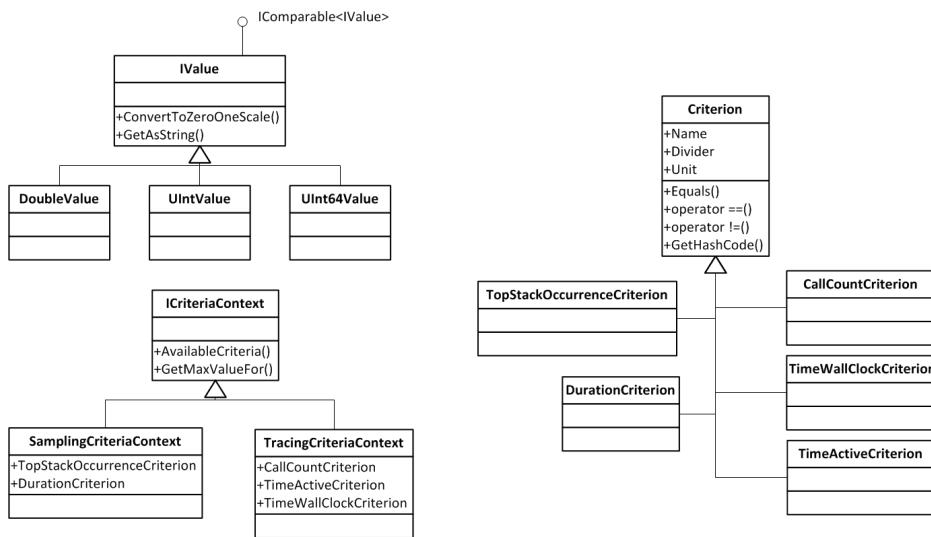


Figure 6.7: The class hierarchy of the model part.

### 6.3.3 User interface - model

The input data for the model come from the Visual Profiler Access as call trees and method, class, module and assembly metadata. The data are converted to groups representing source files and their methods, which then bear all profiling data. This process is depicted in the figure 6.6. This is a non-trivial task because the profiling data are spread across multiple call trees and their elements. Various data analysis have to be performed, for instance data aggregation over call tree elements belonging to the same method, maximum values of different metrics, redistribution of values and so on.

For every profiling mode (tracing or sampling) there are various criteria with different units to measure, what adds to the complexity, (e.g. call count [hits], duration [s...]). In order to make the implementation as reusable as possible (mainly because of UI reusability and independence) we introduced abstractions of values, criteria and criteria contexts, as show in the figure 6.7. The abstract value is capable of converting itself to a string and to a value on scale from 0 to 1 provided a maximum value. The criteria know their names and units. The criteria context tracks all available criteria and their maximum values.

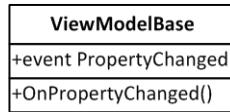


Figure 6.8: The `ViewModelBase` class implements `INotifyPropertyChanged` interface.

### 6.3.4 User interface - view-model

The view-model is responsible for the control of the UI. It holds references to model data and exposes them to the view. The view-model handles user input by commands that are bound to the view. It controls the view by changes of view-model properties, which are databound to dependency properties<sup>4</sup> of the view.

Every view-model inherits from the `ViewModelBase` class, see figure 6.8. This class implements the `INotifyPropertyChanged` interface that allows the WPF databinding to re-databind whenever a view-model property changes. When a view-model property changes, it calls in its setter, the `ViewModelBase`'s `OnPropertyChanged()` method, and passes the property name as an input parameter. The `ViewModelBase`'s `OnPropertyChanged()` method in turn invokes the `ViewModelBase`'s `PropertyChanged` event, which notified the registered handlers (presumably some WPF delegates) about the change to the property. This mechanism is an elegant way to achieve loose coupling between the view and the view-model.

The middle point of the UI logic is the class `UILogic`. This class holds references to all view-models, handles actions from commands and updates views-models that consequently update views.

The other parts of the view-model are:

- `ContainingUnitViewModel` - represents a source file and holds `MethodViewModels`, this view-model is shared by three views
- `CriterionSwitchViewModel` - provides data and commands for switching profiling criteria
- `DetailViewModel` - holds data for a method in focus
- `MethodViewModel` - exposes data and commands for highlighting and changing information in the UI, this view-model is shared by three views

### 6.3.5 User interface - view

The UI is based on a composition of user controls. Dynamic display is accomplished by WPF panels. The placement of the views in the UI is illustrated in the figure 6.9.

The views are:

- `ContainingUnitView` - contains methods, this view has no visual parts, it provides only layout

---

<sup>4</sup>Dependency property - <http://msdn.microsoft.com/en-us/library/ms752914.aspx>

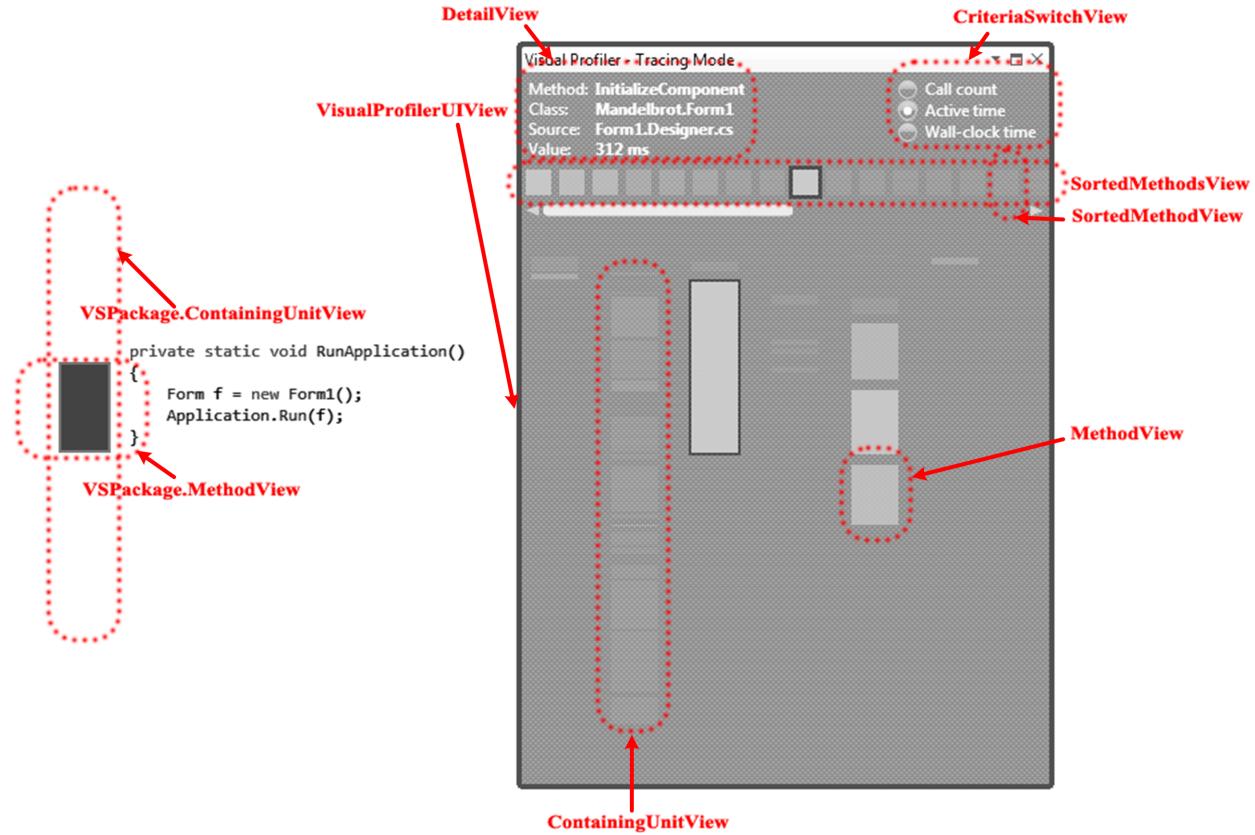


Figure 6.9: Placement of the views in the final UI look.

- **CriteriaSwitchView** - holds the radio buttons for switching between various profiling criteria
- **DetailView** - displays data for a method in focus
- **MethodView** - represents a method in source file, invokes MouseUp, MouseEnter, MouseLeave events
- **SortedMethodsView** - is a container for all methods sorted by an actual criterion
- **SortedMethodView** - same functionality as MethodView
- **VisualProfilerUIView** - the top level container, encompasses all other controls
- **VSPackage.ContainingUnitView** - identical functionality to ContainingUnitView, placed in the source editor
- **VSPackage.MethodView** - like MethodView, resides in the source editor

Every view corresponds to one view-model. However, a view-model might be shared by multiple views. This can save a lot of development effort and coding. The table 6.1 expresses the relation among views and view-models in the UI.

Views	View-models
ContainingUnit View SortedMethodsView VSPackage.ContainingUnitView	ContainingUnitViewModel
CriteriaSwitchView	CriterionSwitchViewModel
DetailView	DetailViewModel
MethodView SortedMethodView VSPackage.MethodView	MethodViewModel

Table 6.1: The relation among views and view-models in the UI.

## 6.4 Visual Studio 2010 IDE integration

The Visual Studio 2010 IDE is a composition of a shell and packages - functional units. The shell provides graphical user interface and services for the packages. The core functionality, project types, testing, compilation services and many more, of the IDE are implemented in the packages. Many third-party extensions of Visual Studio are also packages.

Visual Studio offers numerous ways how to extend it. The philosophy behind extensibility approaches is fragmented, mainly due to historical design decisions, different technologies used and backwards compatibility. Complexity of it is briefly depicted in the figure 6.10. Finding the right interface to use is sometimes real challenge. There is no services discoverability and everything has to be searched for in the scattered documentation and samples.

There are many ways how to add new functionality to the IDE. We will briefly summarize them and then delve into details of add-ins and MEF components. The following approaches may be freely combined.

### 6.4.1 Macros

Macros present ability to add new functionality by composing existing functionality. Functionality in Visual Studio is exposed by commands and there is a macro language similar to Visual Basic that allows executing and registering commands. Macros save repetition - replacing clicks and dialogs with a single key stroke can save a lot of time.

### 6.4.2 Snippets

Snippets do not add new capabilities to Visual Studio, they just save typing and thus error rate. For instance, the snippet for a dependency property is invaluable.

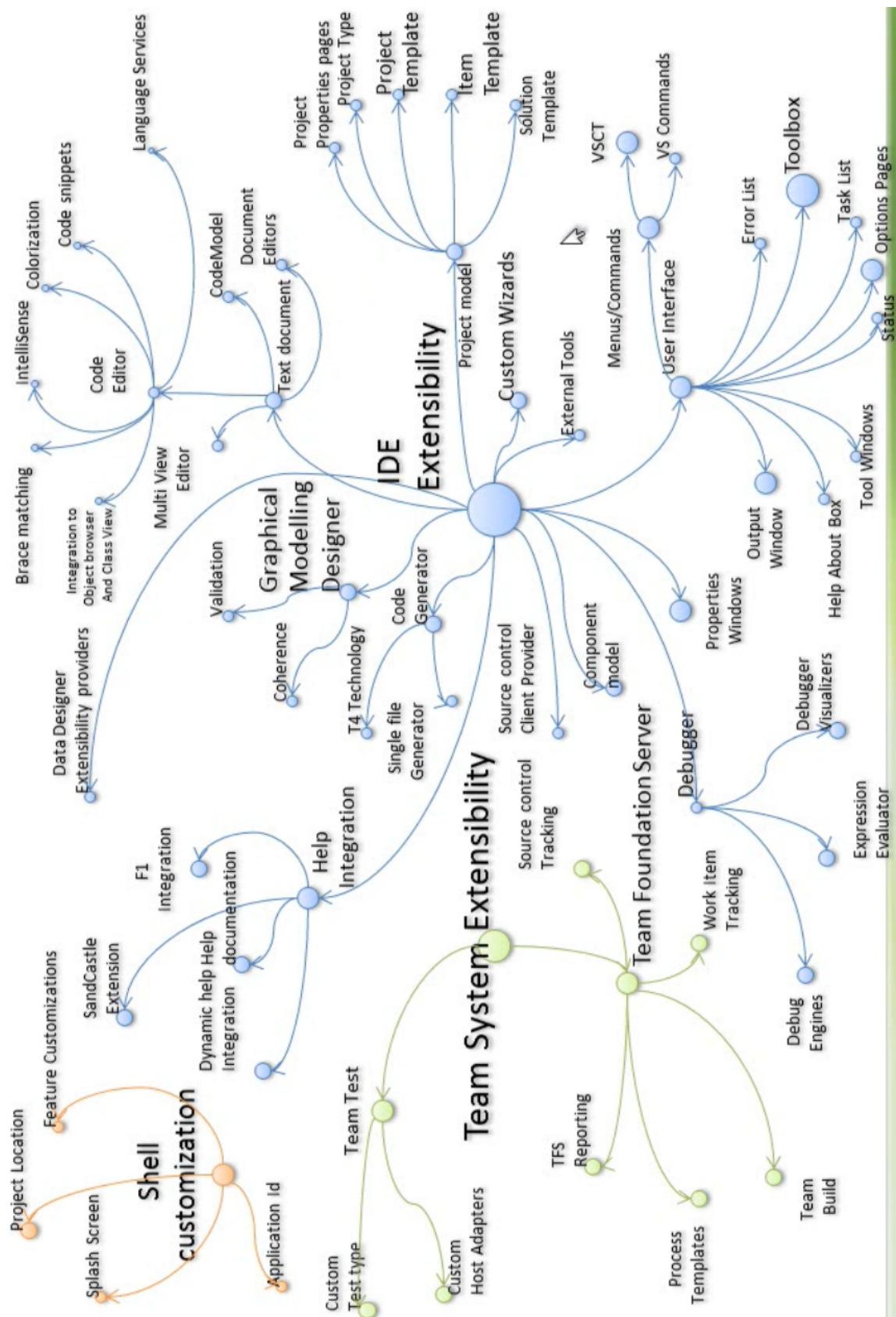


Figure 6.10: Main extension points of Visual Studio IDE.

### 6.4.3 Templates

Templates organize and encompass project definitions. When you bring up a new project dialog in Visual Studio, what it is showing you is all templates it knows about. When you make a choice of a project, a template gets used and all the information like what files to start with, what references will be needed in your project, setting of builds is in template. Templates are ZIP files.

### 6.4.4 Editor Extensions - MEF

Here we have to cite the MEF's website because it describes MEF in very elaborate way:  
“The Managed Extensibility Framework (MEF) is a composition layer for .NET that improves the flexibility, maintainability and testability of large applications. MEF can be used for third-party plugin extensibility, or it can bring the benefits of a loosely-coupled plugin-like architecture to regular applications.” [Com11]

Actually MEF<sup>5</sup> is basically an enhanced dependency injection system.

The new Visual Studio editor is written in WPF and chooses MEF as an extensibility platform. MEF is used exclusively for extending the editor and the modeling and diagramming tools (Ultimate edition).

The editor offer many extension points like content types (code completion, syntax coloring), classification types and formats (key words, syntax errors, comments and so on), tags, margins and scrollbars, **adornments** (any visual effects in the editor) - used in the UI to color methods - mouse processors and drop handlers and IntelliSense. The samples provided by Visual Studio SDK are very good starting point for experimenting with the extensibility<sup>6</sup>.

Editor extensions are deployed as Visual Studio packages.

### 6.4.5 Add-In

An add-in is a compiled DLL that runs in Visual Studio and contains a class that implements **IDTExtensibility2** - a COM interface. It is registered with Visual Studio by an XML file. The add-in receives the Application object at load time, the **DTE2** object, which represents Visual Studio, and it can then call methods on that object to interact with the IDE. An add-in has a very broad scope in comparison with for instance an editor extension. You can interact with almost everything in Visual Studio.

### 6.4.6 Package

A package is a compiled DLL (native or managed) and has to implement the **IVsPackage** interface which provides access to Visual Studio API. Managed Package Framework simplifies package development by providing standard implementations of functions of the

---

<sup>5</sup>Managed Extensibility Framework - <http://mef.codeplex.com/>

<sup>6</sup>Visual Studio Software Development Kit - MSDN website

`IVsPackage` interface. It uses attributes to control registration.

The main differences between add-ins and packages are:

- add-ins were available in the Visual Studio from the very beginning as an extension mechanism and are built on COM components. Packages should overcome some limitations of add-ins and are a newer technology.
- packages are completely integrated in the IDE whereas add-ins are external items for the IDE .
- packages have lower level API access opposed to add-ins.
- the package development kit supports fewer languages than the add-ins deployment model. Add-ins may be developed in any COM enabled environment.
- VSIX files (ZIP files) are used to deploy packages. They are self-containing and well integrated with the Visual Studio extension manager. VSIX files may be shared on the Visual Studio Gallery web site<sup>7</sup> . Add-ins do not have a standardized deployment model and a development team has to take care for installing and registering an add-in.

## 6.5 Visual Profiler integration in Visual Studio

Integration of Visual Profiler in Visual Studio has three parts, `Tools` menu commands for interacting with the profiler, a tool window for hosting the UI and an editor adornment extension for the code coloring. These parts are packed in a single VSIX package.

### 6.5.1 Commands

The Visual Profiler adds three commands to the `Tools` menu as shown in the figure 6.11. The commands are:

- **Start Visual Profiler in Tracing Mode** - builds the current solution, runs the start-up project with the attached profiler in the tracing mode and opens the tool windows with the profiler UI,
- **Start Visual Profiler in Sampling Mode** - does the same as the previous command, except that it starts the profiler in the sampling mode,
- **Stop Visual Profiler** - closes the profiled process and removes the code coloring.

### 6.5.2 Tool window

The tool window is shown right after the profiler starts. The results are updated during profiling in one-second-long intervals. The tool window and its functionality were already described earlier in the chapter and were depicted in the figures 6.3 and 6.9.

---

<sup>7</sup><http://visualstudiogallery.msdn.microsoft.com/>

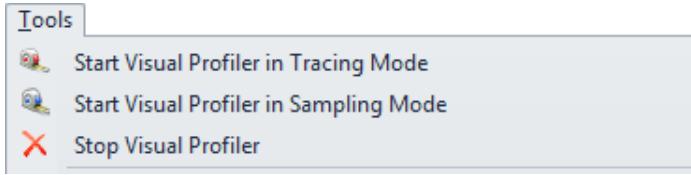


Figure 6.11: Commands in the Tools menu in Visual Studio 2010.

### 6.5.3 Editor adornment extension

The editor extension uses MEF to add a canvas on an editor's adornment layer for every source file encountered during the profiling. Rectangles representing methods and their profiling results are drawn on the canvas. The data are updated live in sync with the main UI. The visual elements are shown in the figures 6.2 and 6.9.

Unfortunately, we encountered a problem or maybe a bug in the editor and the adornment layers. The adornment layers are sometimes seemingly randomly moving during scrolling or navigating in the editor. We initially suspected a wrong usage of the API during adding a canvas to the layer as cause of the problem and tried to resolve it, without success. Then we discovered by means of the .NET reflection that an adornment layer is actually itself a canvas that we managed to color. This colored canvas moved as well together with our canvas. So we assumed that the problem was somewhere else and started googling and stackoverflow-ing, that did not help to find a right answer. There was only one post with a very similar problem - no answer either. We leave this issue to the future work.

## 6.6 Summary

This chapter introduced visual principles used during the mapping of the profiling results to the source code. The technique Visual Profiler uses is based on the proposal of S. G. Eick and J. L. Steffen and their SeeSoft software. We follow their ideas, simplified them and placed them to the real context.

After that, the final UI implementation of the bird's-eye view and the code coloring were presented together with the basic principles of WPF and the Model-View-ViewModel pattern. The usage of these principles was highlighted in the context of the final UI implementation of the Visual Profiler.

Then we moved to the basic concepts of the Visual Studio extensibility, which is a very broad and complex topic.

Our own extension effort was described in the end of the chapter.



# Chapter 7

## Testing and Evaluations

Testing is a crucial part of every software project. In this chapter, we will look at tests used during development of Visual Profiler. Next, we will evaluate performance overhead and correctness of the profiler.

### 7.1 Testing

During the development we employed various testing strategies based on the nature of a particular project.

To test the profiler on a more realistic application we used a multithreaded application rendering Mandelbrot-set (a fractal). It was developed as a class project in 2008. The application and its source code are available without any major changes on the attached DVD.

### 7.2 Testing of Visual Profiler Backend

The backend part is written in C++. Unfortunately, our skills and experience in the C++ platform did not allow us to write unit tests. Therefore the testing of functionality was accomplished manually.

A set of .NET simple testing applications were created to monitor the profiler's behavior. The backend always transformed output call trees into a textual representation that allowed us to verify the expected values, such as call orders, methods hit counts, metadata collections and so on. We also focused on stability of the profiler in cases of exceptions.

Invaluable help was the possibility to debug the profiler backend. Thanks to this we could analyze many problematic areas in the Profiling API.

## 7.3 Testing of Visual Profiler Access and UI

Over seventy unit tests verify the functionality and the correctness of the Access and the UI. We chose the NUnit<sup>1</sup> as a test framework and used it together with the Resharper test runner<sup>2</sup>. It was a great choice since it was a very productive environment. Resharper even allows debugging the unit tests.

The software architecture of the Access and the UI had to be adjusted to allow testing. We did not develop according test driven development paradigm, but we developed “with tests in mind”. The decoupling of classes and introduction of interfaces helped us to ease the testing. To mock dependencies we used a mocking library Moq<sup>3</sup>. Moq is developed to support newest features in .NET and takes full advantage of features like expression trees to easily and type-safely mock behavior.

The UI was covered by unit test only partially, mainly the model part. The biggest reason for that was lack of time in the end of the project. Nevertheless, the Model-View-ViewModel pattern employed in the UI can make testing of the UI logic almost a trivial task.

The Visual Studio Package project template for creating extensibility offers an option of creating an integration test in the initial wizard. Since our extension is a small, proof-of-concept like project, we did use it.

The final user interface was tested by hand. We focused on layout issues, correct functionality and clear logics of the UI.

The installation of the Visual Studio SDK adds an experimental instance of Visual Studio. What this means is that during an extension development, there is no need to risk corruption of the development environment by testing the developed extension in the very same development environment. You can just test it in an independent instance of Visual Studio - an experimental instance. The experimental instance has its own registry and data storage isolated from the main instance. This can even be reset back to what it was after the Visual Studio installation and thus roll back all changes made to the experimental instance. Extensions deployed to the experimental instance can be debugged from the main instance just by hitting F5 - what a great way to develop.

## 7.4 Evaluation of overhead

The overhead imposed on a profiled application is one of the main characteristics of profilers. The overhead greatly depends on the profiling mode and granularity as discussed in the chapter 2.

We conducted measurements of both, the tracing and the sampling mode, profilers on the Mandelbrot fractal rendering application and on a Fibonacci sequence application, created only for the purpose of the measurement. We chose the Fibonacci sequence because of its high recursion, which can very well demonstrate the difference between

---

<sup>1</sup><http://www.nunit.org/>

<sup>2</sup>[http://www.jetbrains.com/resharper/features/unit\\_testing.html](http://www.jetbrains.com/resharper/features/unit_testing.html)

<sup>3</sup> <http://code.google.com/p/moq/>

	without profiling	tracing mode	sampling mode
Fibonacci sequence	1	110,11	1,13
Mandelbrot fractal	1	1,77	1,09

Table 7.1: The multiples of application duration for the tracing and the sampling modes.

the overhead of both profiling modes. The program runs 100 iteration of Fibonacci of 30. The Mandelbrot application in this case represents a “normally” behaving application with loops and function calls. The results are presented in the table 7.1.

As expected, the tracing mode copes poorly with extensive method calls, whereas it makes the application run about 80 % slower in the normal-scenario. On the other hand the sampling mode imposes constant overhead around 10 % regardless of method call intensity.

## 7.5 Evaluation of exactness

The exactness of profiling results depends on the profiling mode and granularity as discussed in the chapter 2. The tracing mode results are very accurate and fully match runtime behavior of an application. Compared to that, the sampling mode results only show trends in the applications. Some method invocations are not even detected. To prove that we ran the Mandelbrot fractal application for both modes and counted the number of methods in the results. The table 7.2 revels that the sampling mode lags behind the tracing mode with only around 70 % of detected methods.

Tracing mode	32 methods
Sampling mode	23 methods

Table 7.2: The number of detected methods in a profiling session.

## 7.6 Impact of assembly filtering on results

The section 4.6 about profiling enabled assemblies discusses reasons for profiling only some assemblies. Leaving out some assemblies from profiling may, nonetheless, hide some important details about method calls and affect the profiling results.

A problem may occur if a method A from a profiled assembly (profiled method) calls a method B from non-profiled assembly (non-profiled method) and the non-profiled method B then calls a profiled method C. The real call tree is A-B-C but the profiler records A-C, because the non-profiled method did not cause a transition in a call tree structure of the profiler.

An interesting effect of this event can be observed in the Mandelbrot test application by measuring the active time (user plus kernel CPU time). The `ApplicationMessageLoop` method is called once and starts only a form (window). It has two lines of code. Interestingly, it ranks as the 3rd most active method. A full, non-filtered profiler run proved our hypothesis about the application message loop acting as the non-profiled method B

from previous paragraph. The application actually handles mouse-move events to display coordinates. The profiled `ApplicationMessageLoop` method starts the non-profiled message loop and the non-profiled message loop invokes the profiled mouse-move handler each time the mouse moves. When the handler returns, it updates time counter of the `ApplicationMessageLoop` method, instead of the counter of the message loop, because the profiler does not have a clue about the message loop methods and its counter.

## 7.7 Summary

The different approaches to testing and ensuring the application correctness, such as unit testing and pure hand testing, were introduced in this chapter. We also pointed out our testing imperfections and attributed them to the lack of time.

After that, we revealed the non-surprising result of the comparison of the profiling modes. The sampling mode is clearly faster and not influenced by structure of an application, whereas the tracing profiler provides more precise data.

Lastly, we looked at an interesting variation of results caused by the assembly filtering.

# Chapter 8

## Conclusion and Future Work

After the thorough initial analysis of profiling modes, granularity and the currently available profilers for .NET platform, we chose to implement tracing and sampling profilers with method-level granularity. The nature of the tracing and sampling profilers is very different. The tracing profiler provides exact information about runtime conditions but slows down profiled applications, whereas the sampling profiler shows only statistical representation of the runtime conditions, but with very low overhead.

We used the Profiling API as a starting point to implement both profilers. The Profiling API exposes a rich set of interfaces to interact with the CLR<sup>1</sup> and to monitor its activity, allowing observing method calls, performing snapshots of thread call stack and querying runtime metadata.

The Profiling API interfaces require their implementers to expose functionality via COM interfaces and to be written as an unmanaged component<sup>2</sup>. Therefore, a profiler backend, responsible for collecting profiling data, was implemented in C++ programming language. Both, the tracing and the sampling profilers, use tree structures to store profiling data, such as number and order of method calls, duration of methods, occurrence of a method on top of a call stack and others.

The profiling backend, the Visual Profiler Backend, is loaded by the CLR at program's start time (it may also be attached later – not implemented) and runs as a part of the profiled process. For the performance reasons, it is strongly recommended to process the profiling data in other process than the profiled one. Therefore, the profiler backend sends the measured data via IPC<sup>3</sup> to a frontend component periodically. The frontend component, the Visual Profiler Access, processes the data and displays it in a UI component, the Visual Profiler UI.

Besides receiving the profiling data, the Visual Profiler Access is a managed gateway to the profiler. It is responsible for initiating a profiling session, and the IPC and for notifying of newly acquired data from the profiler. It also maps data from the profiler backend to source code files using information in PDB files<sup>4</sup> created during compilation of the profiled process.

---

<sup>1</sup>Common Language Runtime - .NET runtime environment

<sup>2</sup>not using CLR as a runtime platform

<sup>3</sup>Inter-process communication - named pipes in our case

<sup>4</sup>Program database file

The visual presentation of results takes place in the Visual Profiler UI. The UI hooks to the Visual Profiler Access and waits for data to come. Upon the data acceptance, the UI combines and processes the received call trees and metadata to form a source code aware view of runtime conditions in the profiled process. The UI is built in WPF<sup>5</sup> combined with the Visual Studio 2010 extensibility model.

The code was tested for correctness by a set of unit tests and manual testing. A performance evaluation showed results of a comparison of the profiling modes. The sampling mode is clearly faster and its overhead is stable, regardless of call tree structure of a profiled application. Whereas, the tracing profiler provides more precise data with volatile overhead in terms of the call tree structure of the profiler application.

The final implementation of the Visual Profiler does not outrun the currently available profilers. The provided set of features is limited and serves more as a proof of concept. The performance and accuracy may be well comparable, though. Its main contributions are its integration into Visual Studio 2010 IDE and the innovative way of displaying profiling results during development-time profiling.

To become a fully-fledged profiler, the Visual Profiler has to add few features and overcome some limitations such as to enable 64-bit application profiling, to target more application platforms (web-apps, windows services, phones, Silverlight-apps...), to add remote profiling, to improve UX<sup>6</sup> by implementing saving and comparing results of different profiling sessions, to focus on navigation in results, to enhance accuracy and speed of profiling and others.

There are still some unresolved issues in the final version of the profiler regarding partial UI instability of the code coloring and the impact of the assembly filtering on results. We are aware of these issues. However, we did not have sufficient capacity to resolve them and thus we delegate them to future work.

---

<sup>5</sup>Windows Presentation Foundation

<sup>6</sup>User eXperience

# Bibliography

- [BB11] David Hill Brian Noyes Michael Puleio Karl Shifflett Bob Brumfield, Geoff Cox. *Developer’s Guide to Microsoft Prism 4: Building Modular MVVM Applications with Windows Presentation Foundation and Microsoft Silverlight (Patterns and Practices)*. Microsoft Press, 2011.
- [Com11] MEF Community. Welcome to the mef community site. <http://mef.codeplex.com/>, 2011.
- [ES92] Stephen G. Eick and Joseph L. Steffen. Visualizing code profiling line oriented statistics. In *Proceedings of the 3rd conference on Visualization ’92*, VIS ’92, pages 210–217, Los Alamitos, CA, USA, 1992. IEEE Computer Society Press.
- [ES11] Inc. Electric Software. The glowcode 8.2 website. <http://www.glowcode.com/>, 2011.
- [Koc99] Richard Koch. *The 80/20 Principle: The Secret to Achieving More with Less*. Crown Business, 1999.
- [McC04] Steve McConnell. *Code Complete: A Practical Handbook of Software Construction, 2nd Edition*. Microsoft Press, 2004.
- [Mic11a] Microsoft. Imetadataimport interface api reference. <http://msdn.microsoft.com/en-us/library/ms230172.aspx>, 2011.
- [Mic11b] Microsoft. Profiling (unmanaged api reference). <http://msdn.microsoft.com/en-us/library/ms404386.aspx>, 2011.
- [Mos11] Christian Moser. Wpf tutorial. <http://www.wpftutorial.net>, 2011.
- [Nat10] Adam Nathan. *WPF 4 Unleashed*. Sams, 2010.
- [Roo02] Paula Rooney. Microsoft’s ceo: 80-20 rule applies to bugs, not just features. <http://www.crn.com>, 2002.
- [Wik11] Wikipedia. .net framework. [http://en.wikipedia.org/wiki/File:Overview\\_of\\_the\\_Common\\_Language\\_Infrastructure.svg](http://en.wikipedia.org/wiki/File:Overview_of_the_Common_Language_Infrastructure.svg), 2011.



# Appendix A

## Visual Profiler Installation

### A.1 System requirements

Visual Profiler is an extension for Visual Studio 2010 Professional and higher. Visual Studio 2010 has to be installed on a target machine. Visual Studio 2010 can be installed on all Windows operating systems starting Windows 2003 provided the latest operating system services pack is present. We recommend installing Visual Studio 2010 SP1.

### A.2 Hardware requirements

Visual Profiler is limited by the hardware requirements of Visual Studio 2010. These are as follow. A computer that has a 1.6GHz or faster processor, 1 GB (32 Bit) or 2 GB (64 Bit) RAM (Add 512 MB if running in a virtual machine), 3GB of available hard disk space, 5400 RPM hard disk drive, DirectX 9 capable video card running at 1024 x 768 or higher-resolution display

### A.3 Installation of VSIX package

To install Visual Profiler to Visual Studio 2010 follow and these steps:

1. open `Install\VisualProfilerVSPackage.vsix`<sup>1</sup> file
2. the Visual Studio Extension Installer dialog appears

---

<sup>1</sup>The path is relative to the root directory of the enclosed DVD

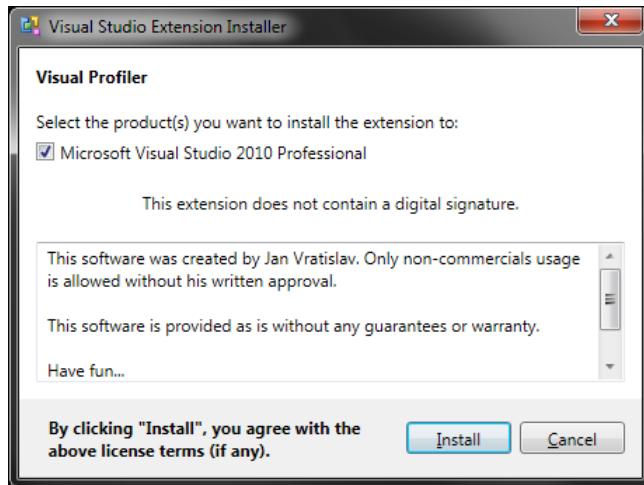


Figure A.1: Visual Studio Extension Installer - before installation

3. make sure you select correct version of Visual Studio 2010 to install the extension to
4. if you agree with the license, click the **Install** button
5. if the installation succeeds, the installation complete dialog appears

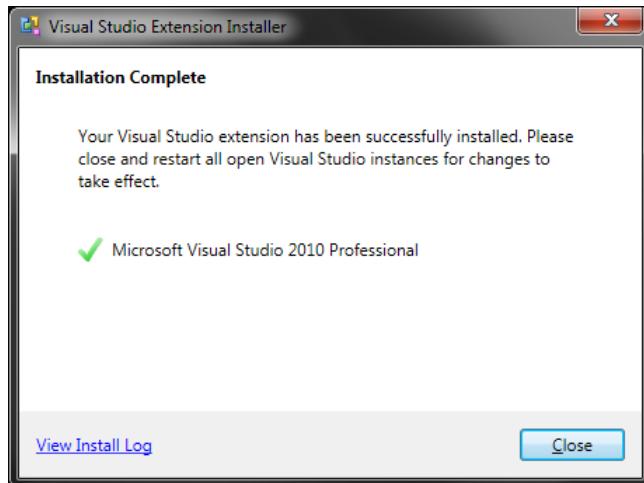


Figure A.2: Visual Studio Extension Installer - after successful installation

6. click the **Close** button, alternatively view the install log file
7. the installation is completed

The Extension Manager manages extensions in Visual Studio 2010. To view status of, disable or uninstall Visual Profiler open Visual Studio 2010 and follow these steps:

1. go to the **Tools** menu
2. click the **Extension Manager...** menu

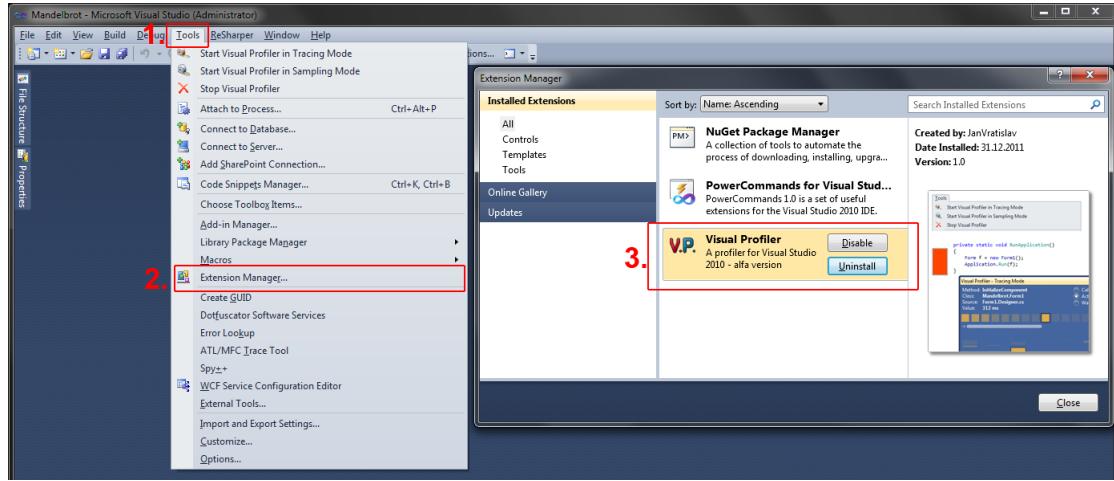
3. view status of, **Disable** or **Uninstall** Visual Profiler

Figure A.3: The Extension Manager in Visual Studio 2010

## 4. any change requires restarting of Visual Studio

# Appendix B

## Running Visual Profiler

### B.1 Enabling profiling

Output assemblies of Visual Studio projects have to be profiling enabled before a profiling session. To make an assembly profiling enabled include the `Install\IncludeInProfiledProjects.cs`<sup>1</sup> file in the project of the assembly or paste its content in its `AssemblyInfo.cs` file. The `IncludeInProfiledProjects.cs` file contains definitions and usage of the custom `VisualProfiler.ProfilingEnabledAttribute` attribute.

### B.2 Running Visual Profiler

Examples in this section use the Mandelbrot testing application, which can be found in `Source\Mandelbrot\`<sup>1</sup>.

Before running Visual Profiler make sure that at least one assembly is profiling enabled in order to see some output.

To run Visual Profile take following steps:

1. set the startup project of your solution to either a console or a windows application project and to x86 platform target
2. select a profiling mode in the **Tools** menu

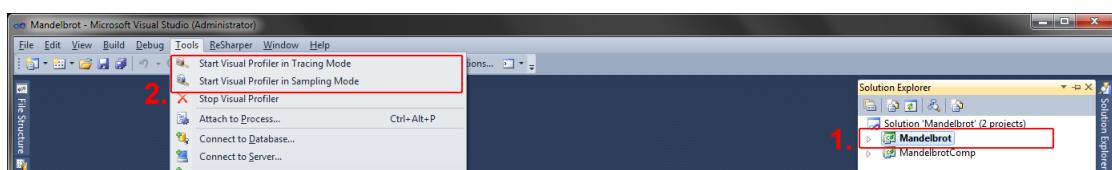


Figure B.1: Visual Profiler - before running

3. the startup project builds and a profiling session starts

<sup>1</sup>The path is relative to the root directory of the enclosed DVD

4. to stop the profiling session go to the **Tools** menu and select the **Stop Visual Profiler** menu



Figure B.2: Visual Profiler - stopping a profiler session

## B.3 Screenshots

We include screenshots of the Mandelbrot testing application running under the profiler. The first screenshot B.3 shows the source code coloring and the tool window with the profiling results

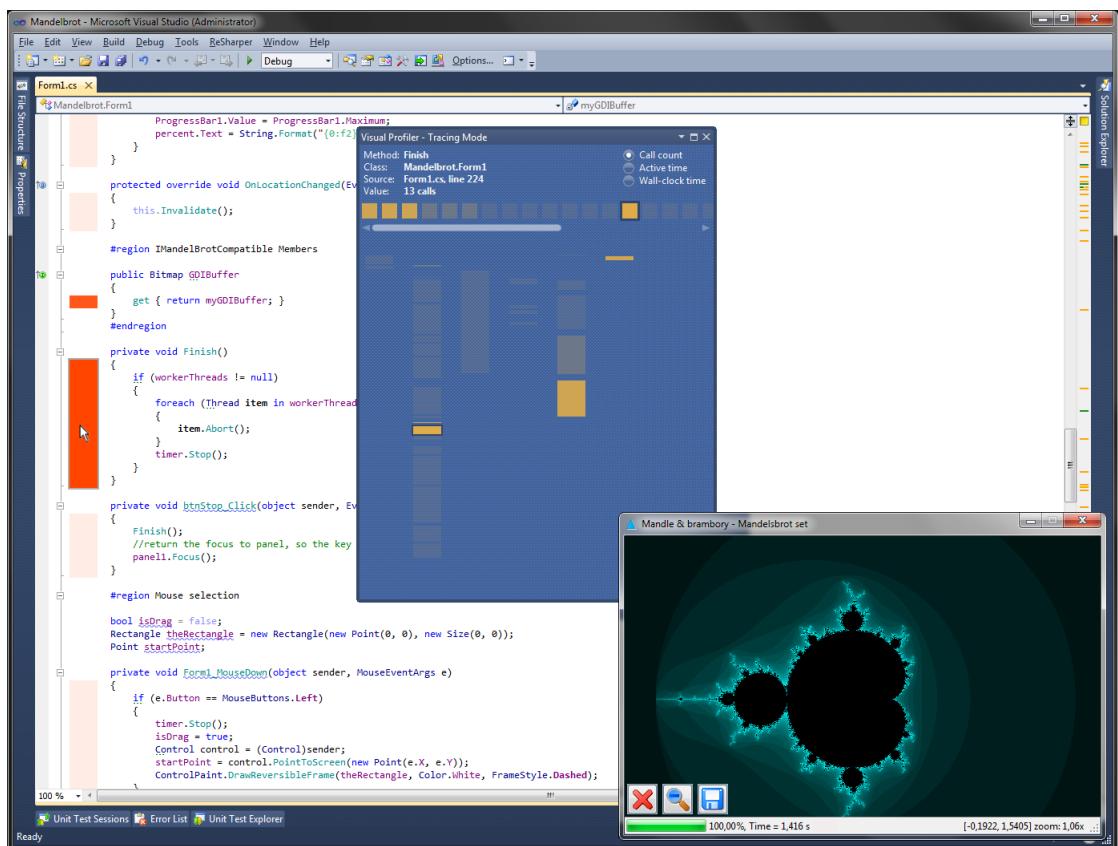


Figure B.3: Visual Profiler - profiling the Mandelbrot testing application

The second screenshot B.4 shows the “mysterious” error in layout of the code coloring. Notice how the coloring moved down and it does not align with method bodies.

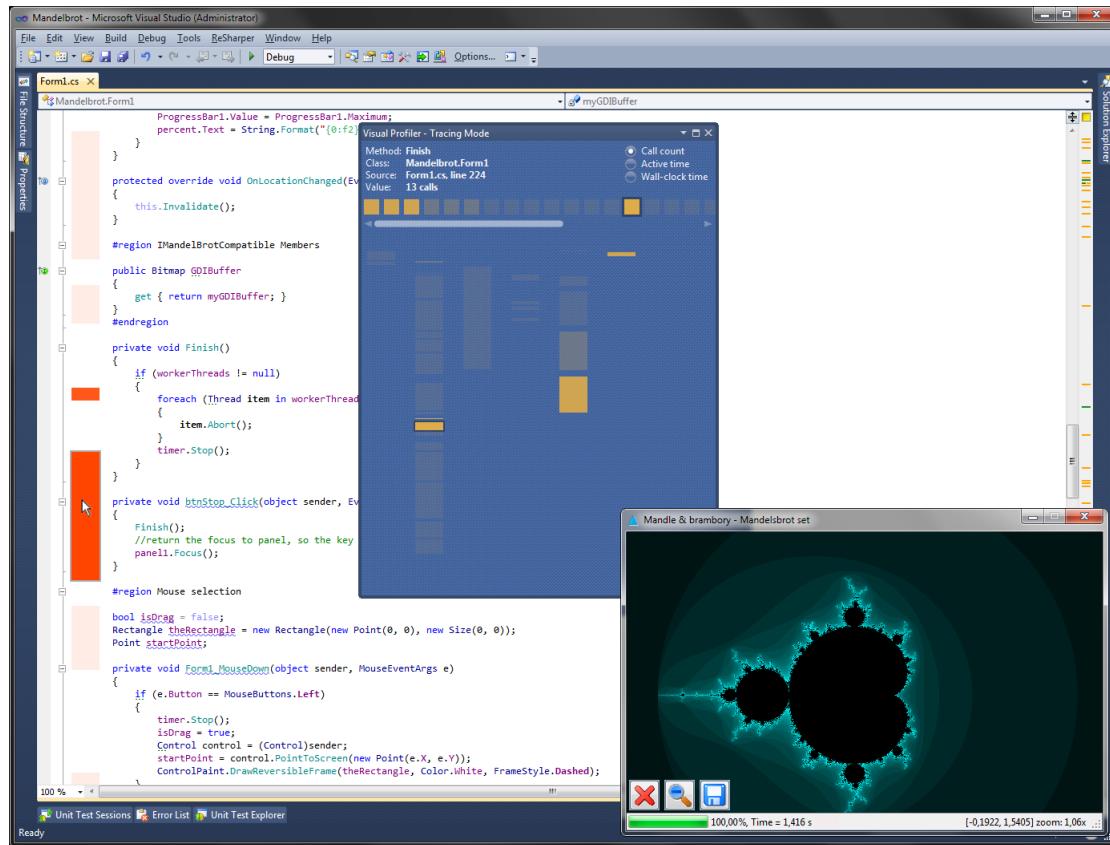


Figure B.4: The layout error - the code coloring does not align with method bodies

# Appendix C

## Source code

### C.1 Solutions and projects

You can find the source code in the `Source\`<sup>1</sup> directory which contains three solutions

- **Visual Profiler** - the main solution, contains VisualProfileBackend, VisualProfilerAccess, VisualProfilerUI, VisualProfilerVSPackage and two test projects  
location: `Source\VisualProfiler.sln`<sup>1</sup>
- **Mandelbrot** - the testing application, contains Mandelbrot and MandelbrotComp projects  
location: `Source\Mandelbrot\Mandelbrot.sln`<sup>1</sup>
- **Metadata** - the Common Compiler Infrastructure Metadata API, contains projects for reading PDB files  
location: `Source\Ccimetadata\Metadata.sln`<sup>1</sup>

### C.2 Dependencies

The `Source\`<sup>1</sup> directory contains all required dependencies for the solutions. Therefore, the solution can be built in standard environment without any further modifications or installations. The only exceptions to this is the VisualProfilerVSPackage project, which requires Visual Studio 2010 Service Pack 1<sup>2</sup> and Visual Studio 2010 SP1 SDK<sup>3</sup> to be installed.

---

<sup>1</sup>The path is relative to the root directory of the enclosed DVD

<sup>2</sup><http://www.microsoft.com/download/en/details.aspx?id=23691>

<sup>3</sup><http://www.microsoft.com/download/en/details.aspx?id=21835>

# Appendix D

## Content of enclosed DVD

The directory structure of the enclosed DVD is depicted in the figure D.1.

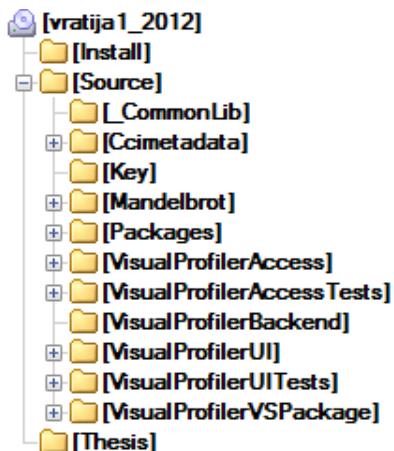


Figure D.1: DVD content

The **Install** directory contains the VSIX package for deploying the Visual Profiler extension into Visual Studio and a C# file for enabling profiling on assemblies.

The **Source** directory contains the source code of this thesis, its dependencies and the Mandelbrot testing application.

The **Thesis** directory contains the PDF document of this thesis.