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 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 Zakona c. 121/2000Sb. (copyright law), and with the rights connected with the copyright act including the changes in the act.

Prague, January 3rd, 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.

Abstract

The so-called cascading undo command has been introduced by Aaron Cass and Chris Fernandes . This new approach to the undo command overcomes the weakness of the linear undo command. It allows undoing an arbitrary action from the history while watching the dependencies among the actions. However, there is not a visualization of cascading undo yet. Thus, in this thesis we discuss, introduce, develop, and evaluate several visualizations for cascading undo. Unlike the linear undo visualizations, cascading undo visualizations have to deal with dependencies among user actions. We believe that an overview of the dependencies should be presented to a user before committing and undo command. The visualizations we proposed are flexible enough to reflect the possible complexity of the user actions and their dependencies.

Abstrakt

Aaron Cass a Chris Fernandes představili takzvaný kaskádový příkaz zpět. Tento nový způsob příkazu zpět překonává slabiny všudypřítomného lineárního příkazu zpět. Umožňuje totiž zrušit libovolnou akci z historie akcí dokumentu. Při tom bere v potaz závislosti mezi těmito akcemi. Nicméně nikdo ještě nevyvinul vizualizaci pro kaskádový příkaz zpět. V této práci diskutujeme, představujeme, vyvíjíme a hodnotíme několik vizualizací kaskádového příkazu zpět. Na rozdíl od vizualizace lineárního příkazu zpět musí kaskádový příkaz zpět počítat se vzájemnými závislostmi provedených akcí. Věříme, že uživatelé by měli mít přehled o těchto závislostech ještě před jejich vlastním odebráním. Námi navržené vizualizace jsou flexibilní natolik, aby zvládly zobrazit komplexitu uživatelských akcí a závislostí mezi nimi.

Contents

List of Figures

Chapter 1

Introduction

According the Pareto's law (also known as 80/20 rules) 80 percent of the results comes from 20 percent of the effort [Koc99]. This law is applicalbe 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 debited to 20 percent of bugs [Roo02].

The same principle applies to software performace. 80 percent of time is spent in 20 percent of code. Some argue that it is even more, 90/10. So, investing programmers' effor to the 20 percent of code may have great effect on the overall speed - if that 20 percent of code be discovered.

Programmers are not particularly successful at guessing which part of the code is crutial for the perfomance [McC04]. Therefore profilers help to automate the search of bottleneck 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. .NET programs' performance profiling ist the main area of this thesis.

The performace profiling dynamically analysises execution behaviour of a program. It tracks various runtime related data as frequency and duration of function calls. Analysis and visualition of the gathered data provides useful hint on the program's code runtime characterics and helps during optimalization and exploration of the program.

Profiling can be achived by various means as e.g. by inserting tracing code into either the source code or the binary executalbe 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 kind, precision and granuality of the gathered data.

In the world of .NET performace profiling exist already 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 into development tools.

In this thesis, we will to 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. We will overview some contemporarily used profilers targeting .NET platform.

2.1 Profiling modes

An application can be profiled in several ways that differ in the results' precision, profiling overhead.

2.1.1 Sampling profiling

In this mode, the profiler stops periodically every profilee's thread 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 in full speed with minimum side effects, such as additional memory allocations, cache faults and context switches.

The profiling overhead does not depend on the number of method called by the profilee as in other modes. The introduced error of the results is equal to half of the sampling period.

The profiler cannot count the number of method calls and the method duration are computed purely based on gathered statistics.

No source code or binary alternations in required, the runtime ability to stop execution to do a call stack snapshot or alternatively an interrupt instruction can by used.

2.1.2 Tracing profiling

This way of profiling relies on a runtime environment or some kind of hardware notification when a method is entered and left and leads to accurate results, providing exact count of method calls and their durations, however, with some added overhead. In addition, the overhead increases with the method calls count and slows down the profilee.

As with 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

The profilee code or binaries 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 alternation the original profilee behaviour or even introducing bugs.

Instrumentation can be perform 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 analysed. It brings precise result, however, the additional burden can alter the program's behaviour.

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

2.2.2 Method granularity

Only information regarding an entire method run are collected. The lower overhead can ease the profilee and still provide very informative results.

This level of granularity is the only option for the sampling profilers and most cases the tracing profilers. The instrumentation profilers can be easily converted from the line-by-line to the method granularity.

2.2.3 Selective granularity

The profiling process can be filtered with combined levels of granularity, where some parts of code are monitored on the line-by-line and others only on the method manners or not at all in order to capture desired statistic 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 is a detailed call stack analysis 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. As usually, the measurement accuracy is counterweighted with the computational complexity.

Wall 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 is in either a wait, a sleep or a join mode.

User and kernel time

In comparison with the wall time, the user and kernel time measurement counts solely time spent executing a method exclusive 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 measurement, 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

When relations among methods calls allows 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 in cost of higher CPU and memory overhead. A example of a call tree is shown on the figure 2.2.

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

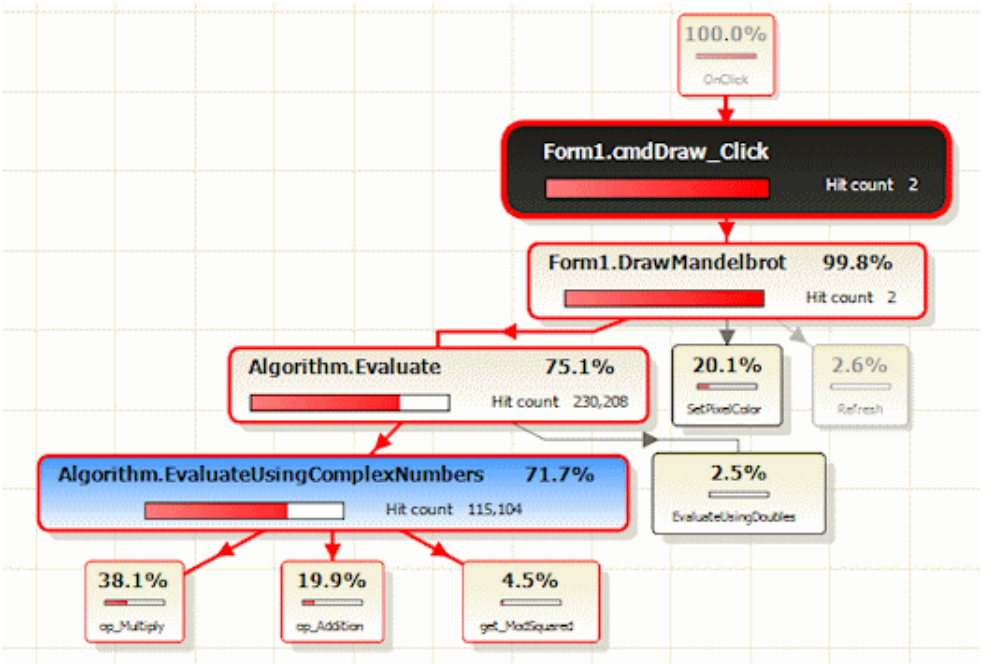

Figure 2.2: Red-Gate ANTS Profiler call graph

2.4 Available .NET performance profilers

On the market, there is many various options in the field of .NET performance profiling. The commercial solutions offer many features and integration with others 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 do not have required financial and time resources for this challenging and but 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 others JetBrains tools. All the profiling modes as presented and the profiling can run remotely.

www.jetbrains.org

2.4.2 Redgate ANTS Performance Profiler

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

It offers some integration in the Visual Studio and allows to see code directly from the profiling tool.

www.red-gate.com

2.4.3 EQATEC Profiler

There is no doubt that this profiler offers the widest targeting platforms options. It can be used to profiler virtually anything in the ".NET world". It uses some 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

2.4.4 GlowCode

GlowCode is a 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. For native, managed, and mixed code. [ES11]

We were unable to find anything about the Visual Studio integration. So we assume that it is not supported.

www.glowcode.com

2.4.5 Visual Studio Profiler

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

2.4.6 Others

There is even more very comparable profiling suites offering very similar functionality as aforementioned solutions as the Telerik JustTrace, the SpeedTrace Pro and maybe even other that we have not discovered.

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 choice is not as vast as the commercial, so far offering only two solutions.

2.4.7 SlimTune

SlimTune is a free profiler that offers advanced features as remote profiling 32-bit and 64-bit profiling, 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

2.4.8 Prof-It for C#

A unique way of the profiling introduces the Prof-It for C# profiler. 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

Summary

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

Chapter 3

Performance Profiling on .NET

In this chapter, we will first briefly review some .NET framework core facilities, mainly CLR, and then explore options to the performance profiling on this platform.

3.1 .NET framework core facilities

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

.NET based applications do not contain machine-level dependent instructions and cannot be, therefore, 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 (Just-in-time compiler) that uses processor specific information during the compilation. This mechanism is comparable to Java and its bytecode.

The .NET platform is an implementation of Common Language Infrastructure standards (CLI) and forms a solid base for development and execution of programs that can be written in any programming language on any platform. The center elements of .NET are the runtime environment 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 already mentioned ones the JIT compilation also provides assemblies 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.

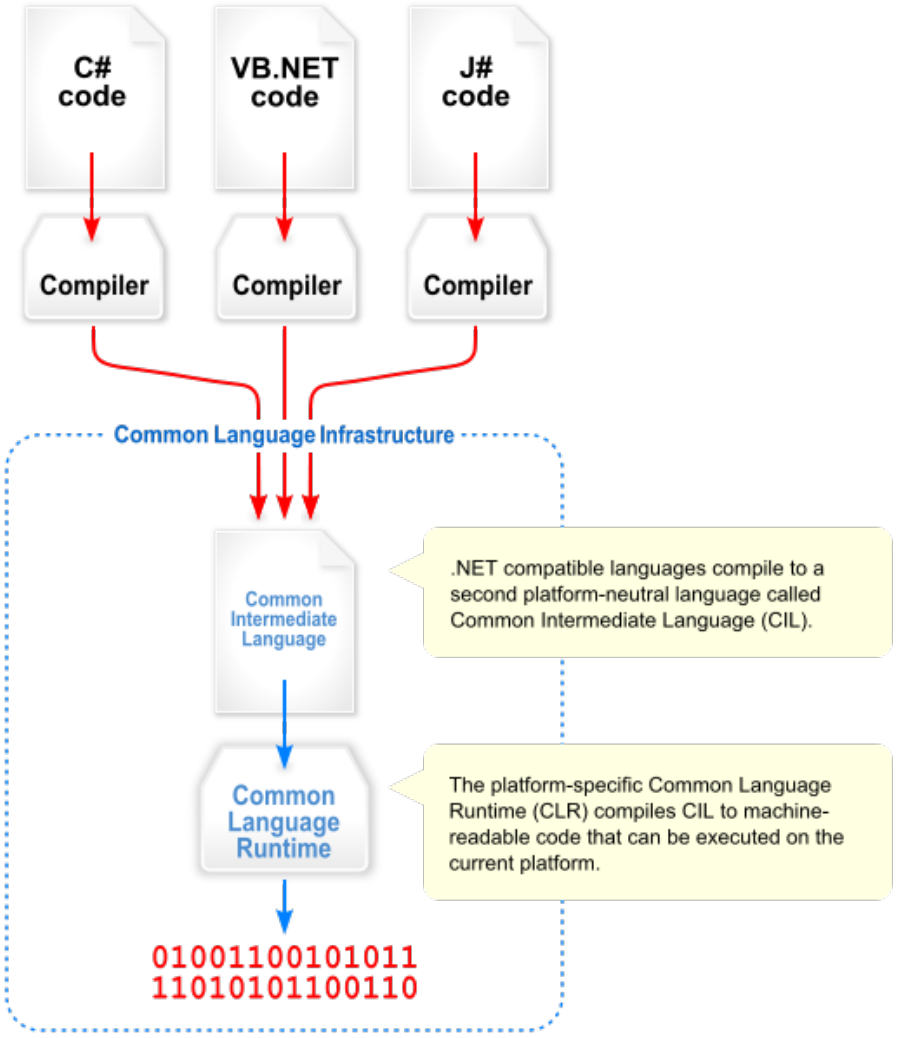

Figure 3.1: Overview of the Common Language Infrastructure [Wik11]

3.2 Profiling modes in .NET

All the profiling modes (sampling, tracing, instrumentation), chapter 2, are feasible in the .NET environment. There is even more ways how to achieve some of them, however, with very different 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 on some program's 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 this problems and let a programmer sample all managed call stacks and read metadate 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 difficult, 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 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#. Obvious limitation is only one 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 all 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 on before the JIT compilation.

It is apparent how big the role the profiling API in the profiling of .NET applications plays. 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.

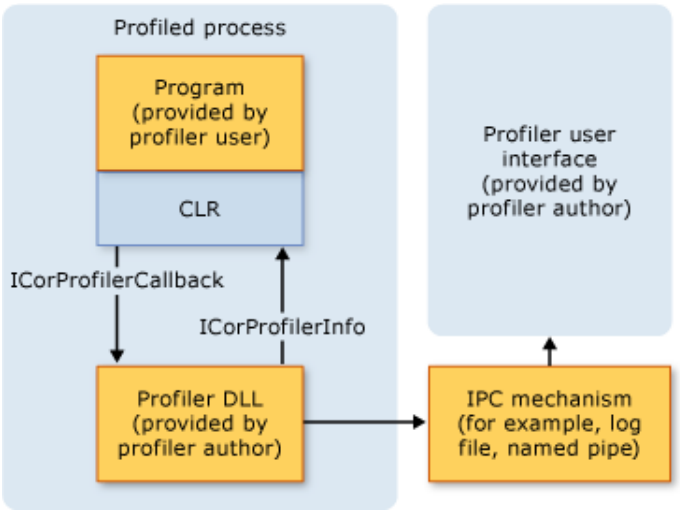

Figure 3.2: Profiling architecture [Mic11b]

Profiling a .NET application is not as straightforward as profiling a conventional application compiled into the machine code. The main reason are aforementioned CLR features that the conventional profiling methods cannot properly identify and thus allow the profiling. The profiling API provides a way how to acquired 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 it is a versatile diagnostic tool for the .NET platform that can be used in many program analysis scenarios, e.g. a code coverage utility for unit testing.

To make use of the profiling API, as depicted on 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 a in process COM server . The CLR calls methods in that interface to notify profiler of one of many runtime events of the profiled process. The profiler can query the state of the profiled application and 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 profiler process. The rest of the profiler, analysis and UI, should happen in an independent process.

Unfortunately, the profiler DLL cannot be written in managed .NET code. It actually makes sense. It the interfaces were implemented using managed code, it would be self triggering the CLR notification and would eventually end up in never ending recursion or a deadlock. For example, imagine that you register a method entered 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. Be

careful with that! If you changed the machine wide environmental variables then every .NET process would be profiled. The profiler can even be 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 is 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 methods of the *ICorProfilerCallback* interface receives additional data in form of input parameters. And some methods come in pairs started-finished (*GarbageCollectionStarted-GarbageCollectionFinished*) or enter-leave (*ExceptionSearchFunctionEnter-ExceptionSearchFunction*

The parameters are either unsigned int (UINT) Ids or pointers to some CLR structures. E.g. *ThreadCreated* get its threadId parameter of type ThreadID (only *typedef*ed UINT). The Ids are opaque values, but we think they are bare pointers to the memory where the metadate reside since they are always defined as UINT_PTR.

To get more information such as functions name, defining class, assembly, actually pretty much everything, what is in the metadate, from the Ids, the profiling API offers the

ICorProfilerInfo interface and the *IMetadataImport* interface and system of metadata Tokens corresponding with the metadata.

The profiling API offer a lot and there is a lot more to write about. However there also some limitation to it. Among others 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 the part of the profiler responsible for collecting the data from the profiled application and for sending them to the analytic part of the Visual Profiler frontend. There are many requirement for a profiling backend to fulfil such as speed, multithreading, correctness, low memory consumption and others. In this chapter, we will 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 the 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 before the 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 beginning.

In the abstract of the thesis we have committed ourself 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 that the other. In the end, we opted for the tracing profiling mode since it does not require any changes to target assemblies.

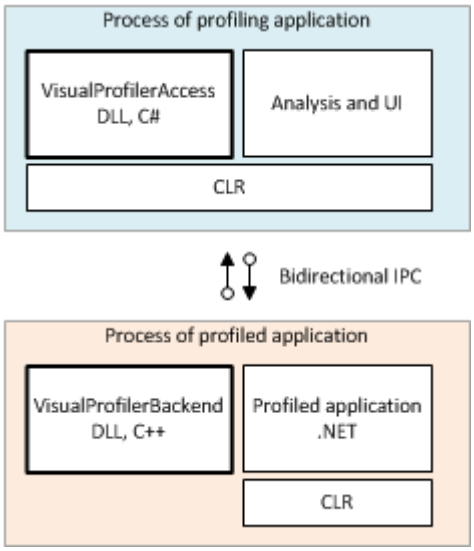

Figure 4.1: The conceptual architecture of the backend

4.2 Software architecture of the backend

The backend runs in two process. The first process, written in C++ and called the Visual Profiler Backend, is the actual profiled application with the hosted profiler DLL 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 profiled 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 is responsible for their processing so they can be later used by an analytic and UI frontend.

The whole relation is shown on the figure 4.1.

The both chosen profiling mode 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 (interprocess communication) to the Visual Profiler Access. The Active Template Library (ATL) framework from Microsoft is used to simplify programming of the Component Object Model (COM) in C++

Let us now introduce you the inner structure and rough outline.

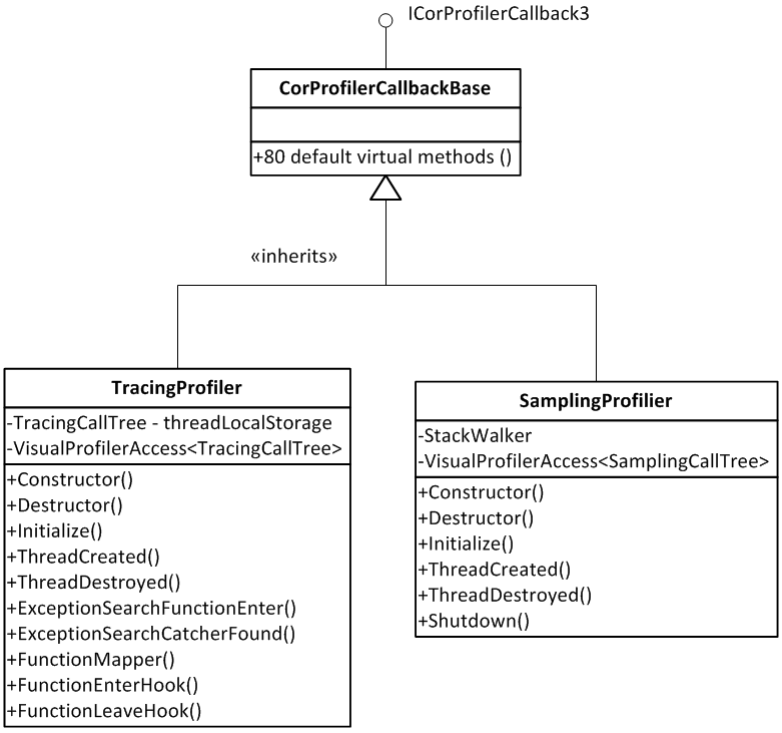

Figure 4.2: Implementations of the *ICorProfilerCallback3* interface

4.3 Implementations of the *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 it, mainly because only a handful of the method is usually made use of. For that reason we created a default class *CorProfilerCallbackBase* as depicted on the figure 4.2. This provides empty implementation of every methods of the interface and derived classes can only override desired methods. This idea makes it easier to have clearly arranged classes, the clean code rules.

4.3.1 *TracingProfiler* class

This class is home for the tracing profiler mode implementation, the 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 passed a pointer to the *ICallProfilerInfo3* interface. Here registers the profiler the function ID mapper (more detail later in this subsection), the function enter/leave, the thread created/destroyed and the exception events notifications. At this point is everything set up and the profiled application starts running.

During the profiling session there is a need to trace the method enter and leave notifications for every managed thread simultaneously. This is a demanding task if you take

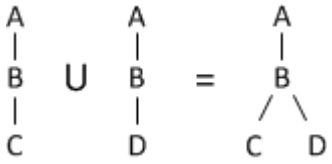

Figure 4.3: Union of thread's call stack snapshots forming a call tree. A very simple example: A calls B, B calls C, C returns to B, B calls D, D returns to B, B return to A

in account the every single call to and return from a managed method is augmented by overhead of this notifications.

Our first approach to store the collected profiling information was to push the data, a function ids, into an enter stack and a leave stack. Every managed had its own private enter and leave stacks. All stacks were saved by the corresponding thread ids 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). So the critical section had to be added to access the hash table.

That was fairly easy to implement and did indeed worked. However, performed not 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's call stacks forms a call tree as depicted on the figure 4.3.

The *TracingCallTree* class was created based on the idea. A simplified representation of the *TracingCallTree* class is depicted on the figure 4.4. Every tree element (class *TracingCallTreeElem*) contains cumulative profiling results such as the number of the enter and leave, time 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, its tree child, 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 transition are very quick.

The other improvement was in using a thread local storage of managed threads for storing a reference to the call trees. Thanks to this the lookup in the call tree hash table can be avoided in thread safe manners 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, than the without profiling. This slowdown is valid only for the unoptimized build of the profiler (debug settings), the optimized version performs even better.

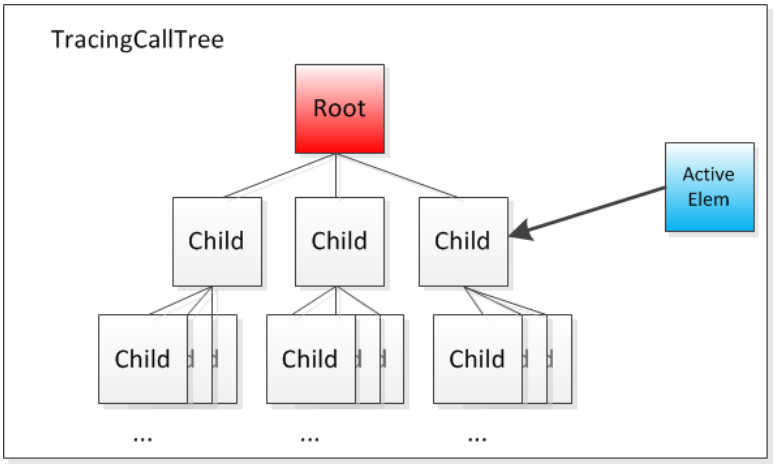

Figure 4.4: A simplified representation of the TracingCallTree class. The active element pointer points to the currently executing function element with the call tree hierarchy.

Let us move on with the profiler's activity. 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 with it an instance of the *TracingCallTree* class. Such an 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.

And then a managed method in the profiling 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 methods. 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, 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 call 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 a upward transition on the thread's tracing call tree.

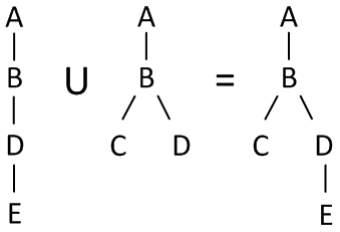

Figure 4.5: Union of a stack snapshot to a thread's sampling call tree.

If a thread of the profiled application ends its execution without any exception the CLR call the ThreadDestroyed method and lets the tracing profiler to do some book-keeping and close the corresponding tracing 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 the *ExceptionSearchFunctionEnter* invokes for every searched function on the thread's call stack till it find 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 were not found the profiled application would crash due to an unhandled exception.

After the profiling application exited, the IPC send out, in the *Destructor* function, the profiling data and is ended and 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 on 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 to it a pointer to the *ICallProfilerInfo3* interface. The sampling profiling set the mask to value *COR_PRF_MONITOR_THREADS* a *COR_PRF_ENABLE_STACK_SNAPSHOT* 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 threads' stacks. The sampling is being started.

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

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

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

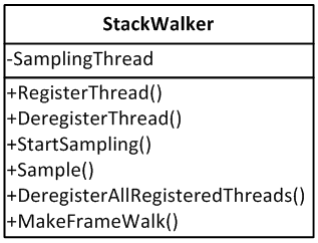

Figure 4.6: The *StackWalker* class

To get the stack snapshot the stack walker calls for every registered managed thread the *DoStackSnapshot* method of the *ICorProfilingInfo3* interface passing it the manage thread id (type ThreadID) of the thread, whose stack should be explored. It also passes to the *DoStackSnapshot* a pointer to the function *MakeFrameWalk* that is called for every single stack frame on the thread's call stack. 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 adds the just acquired stack snapshot with the sampling call tree, caches the methods' metadata and repeats the same process for another registered thread.

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 it execution the CLR calls *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 reading the metadata occurs. In order to avoid this overrides the sampling profiler 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 block the CLR's thread till the last stack walk is completed. Then 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 on the figure 4.7

The both base abstract classes are templated. The template parameter are to be specified by inheriting from the abstract base classes, as shown on 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 to put the common generic logic that only differs in the templated types in one place and specify the type later by inheriting the base class.

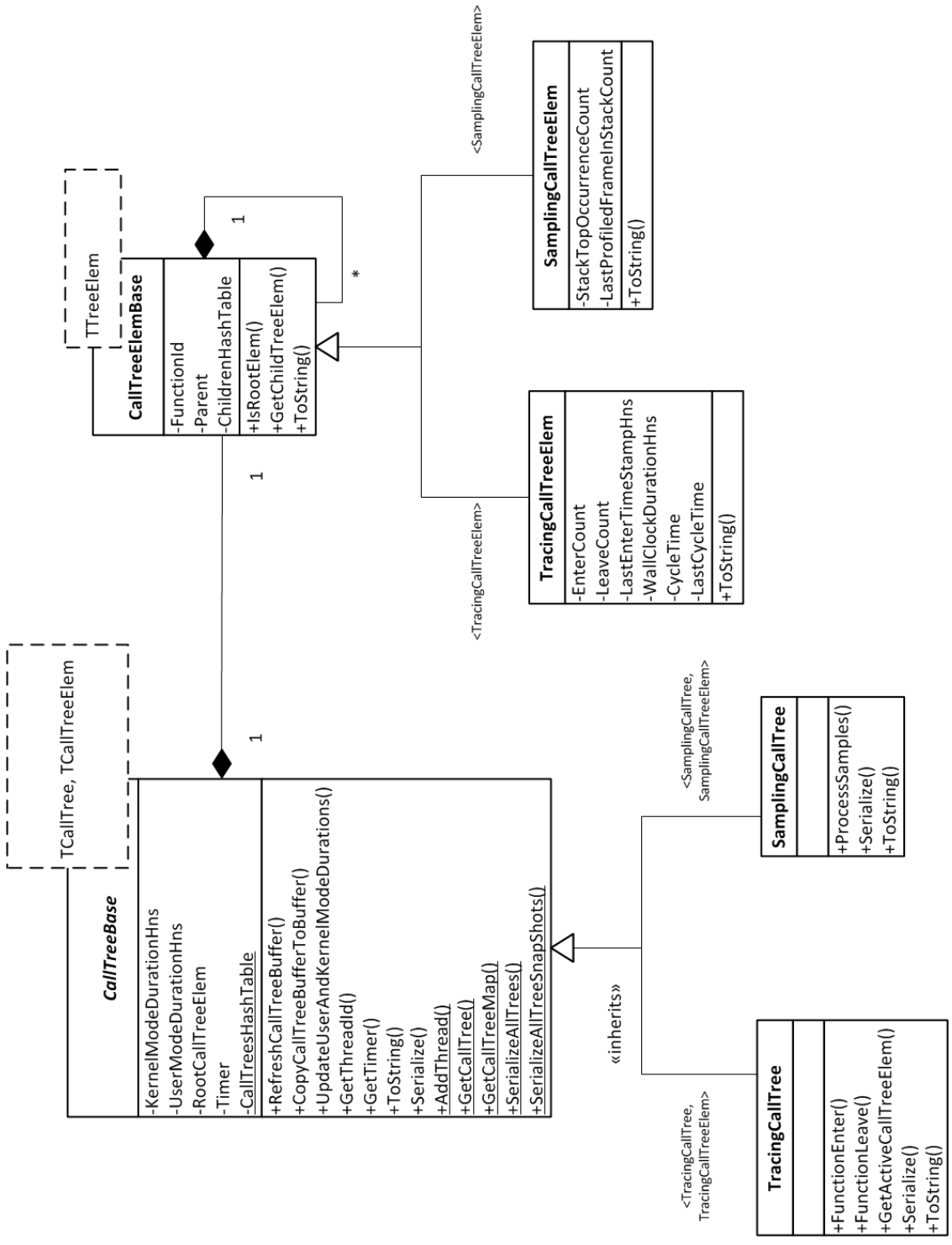

Figure 4.7: The hierarchy and relation between the call tree and the call tree element classes used to trace the collected profiling data. Underlined members are defined as static.

```
class TracingCallTree : public CallTreeBase<TracingCallTree, TracingCallTreeElem>
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 some other auxiliary members.

The *CallTreeElemBase* class is plain data class and holds the 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 behaviour 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 yet other 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 to a metadata token to corresponding metadata. The ICorProfilingInfo provides few method 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 few others functions returns via an output parameter a reference to the *IMetaDataImport* object.

Our strategy is to cache by ids the metadata for methods, their classes, their modules and their assemblies. The profiler in its both mutation 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. FunctionID or 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 on 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* class. Thus, it is possible to get for each function id its respective metadata.

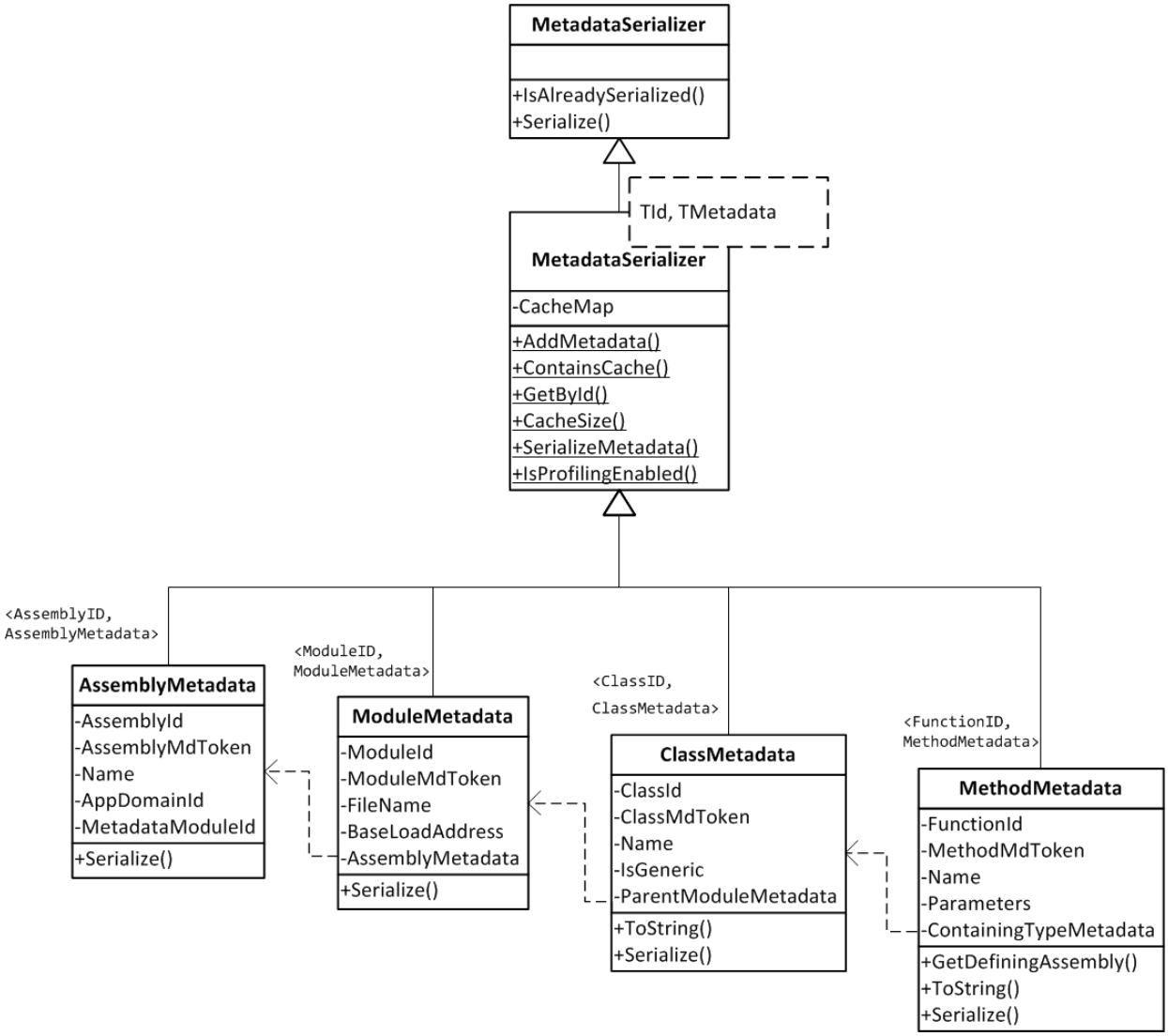

Figure 4.8: The hierarchy and relation between the metadata classes used store .NET metadata. Underlined members are defined as static.

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 and 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 spot, the place where to start tuning up the profiled application.

This is why the filtering on the assembly level was introduced. An assembly is profiling enabled if it has the *[assembly: VisualProfiler.ProfilingEnabled(true)]* 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 use 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 via its output parameter the amount of time that the thread has executed in user and kernel modes. 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 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 it the measurement error hindered.

Every tracing call tree element is self responsible for the wall-clock time measurement 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's call stack and how many times it was the last top most profiled enabled method, that means the method called some other

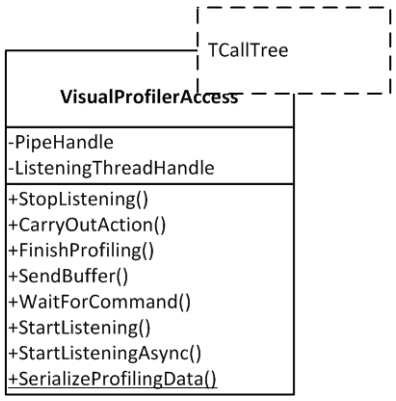

Figure 4.9: The *VisualProfilerAccess* class encapsulates the communication logic with the frontend

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 that the analytic frontend with an user interface. Therefore the profiling data is transmitted to the frontend's 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, the figure 4.1. The pipes name is handed over in the process' "VisualProfiler.PipeName" environmental variable. The communication logic is enclosed in the *VisualProfilerAccess* class, the 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 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

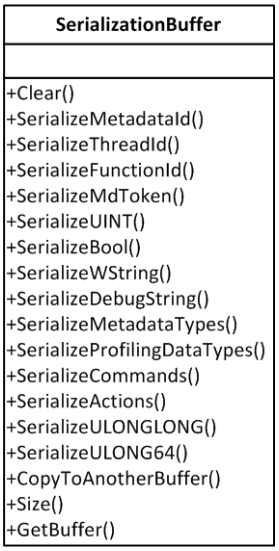

Figure 4.10: The *SerializationBuffer* class members

Figure 4.11: The structure of the serialization buffer

4.8.1 Serializing the profiling data

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

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

First the metadata is serialized. However, incrementally - only the metadata that has not yet been send 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 send as they are, the figure 4.11.

4.8.2 Live updates

The visual profiler supports the live updates feature. It allows to view the intermediate profiling results while the profiling is running.

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 a on-demand snapshot mechanism was adopted. It works as follows and shown on the figure.

Every call tree has it own buffer, instance of the *SerializationBuffer*. After the profiling data have been updated (after the enter/leave notification or the stack walk) the call

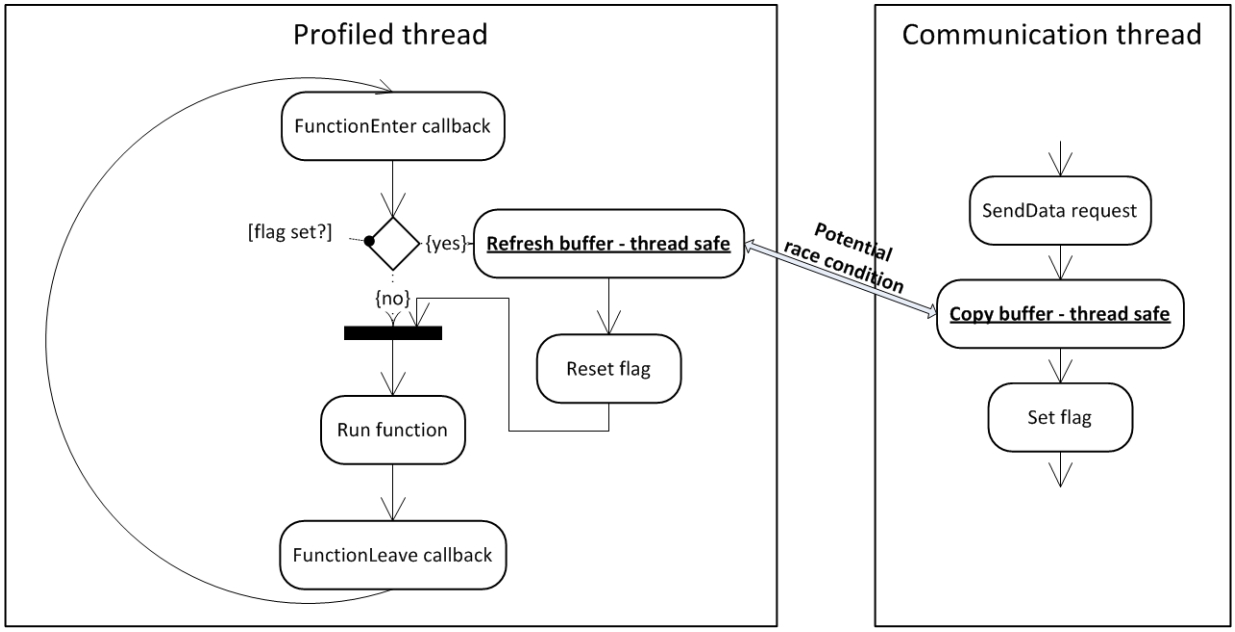

Figure 4.12: The live update activity diagram: cooperation between the tracing profiler and the live update - simplified example

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 send out to the frontend and the *RefreshCallTreeBuffer* flag is set again. And the whole process repeats. However, the buffers were refreshed for the last time when the before last **SendProfilingData** command was received and they are not really update. But this is acceptable since the data is only a preview of the profiling result, not the real results.

Thus, 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 to implement the tracing and sampling profilers and the software architecture of the visual profiler backend was outlined. We moved afterwards to explore how both profilers were implemented, how the metadata is acquired and stored, what and how is measured and, finally, how the profiling data is send to the frontend through a named pipe.

Chapter 5

Visual Profiler Access

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

5.1 Metadata, call trees and call tree elements

The data structure in the Visual Profiler Access closely resembles 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 on the figures 5.1 and 5.2

The metadata derived classes use static caches for every implementation of the generic base *MetadataBase* class. The cache is the *System.Collections.Generic.Dictionary¡uint, TMetadata¿* class. The key to the dictionary is the metadata id. The metadata is added the cache as they come from the backend.

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

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 on the figure 5.3. The the base classes of the call tree and the call tree elements classes define an abstract method for deserialization. This method has to be overridden in inherited classes and there belong the deserialization logic that copies the logic of the serialization in the backend.

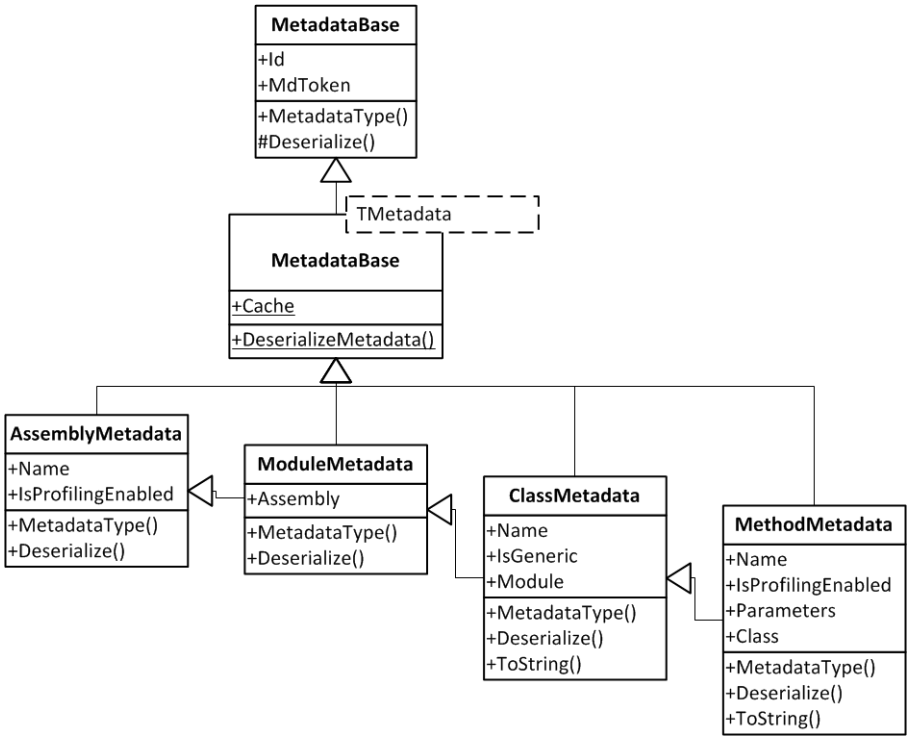

Figure 5.1: The metadata inheritance hierarchy closely resembles its counterpart from the chapter 4.

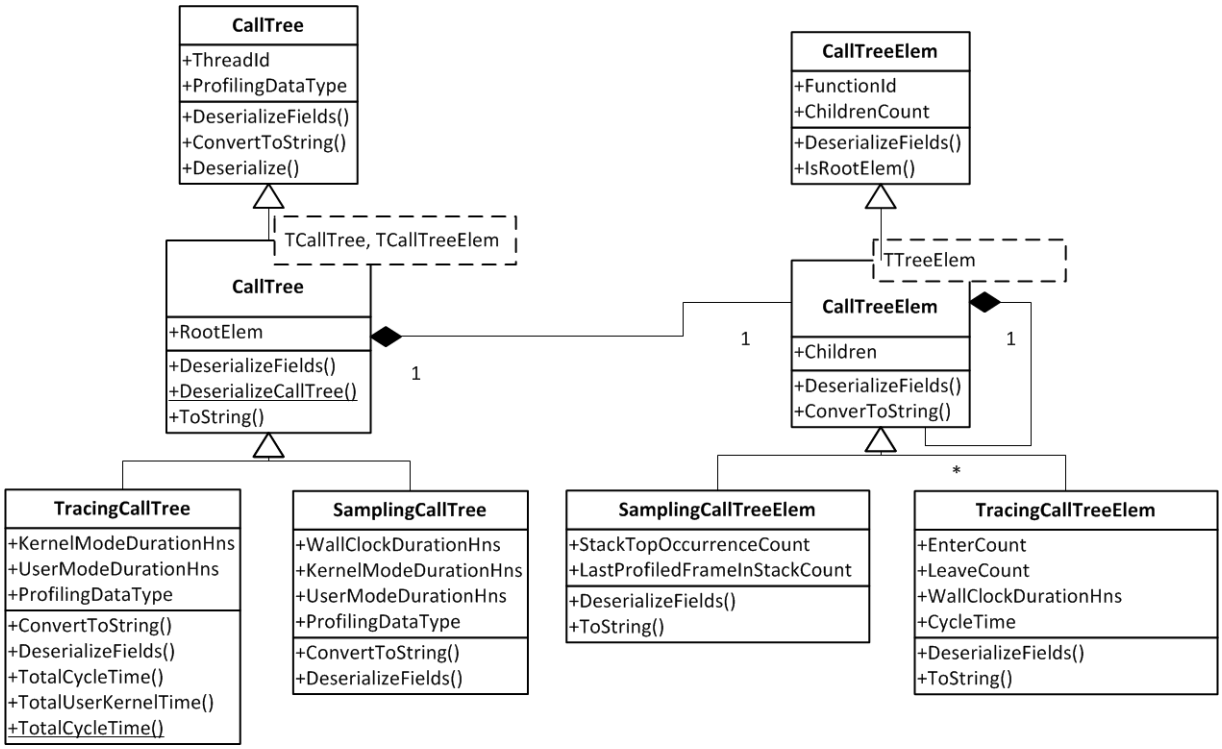

Figure 5.2: The call tree and call tree element inheritance hierarchy closely resembles their counterparts from the chapter 4.

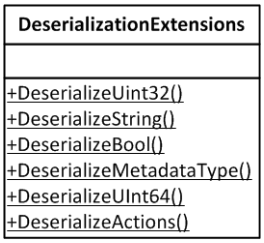

Figure 5.3: The DeserializationExtensions class to add extension methods to the *System.IO.Stream* class for deserialization of the basic types.

5.3 Visual profiler access API

The entry point to the profiler is the *ProfilerAccess*<*TCallTree* > class, the figure 5.4. To use the construct of it 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 specify the executable of the profiled application,
3. pass one value of the *ProfilerTypes* enum that represents what profiler to use in the backend,
4. supply the the update period for the result fetching and
5. pass a reference to the *EventHandler*<*ProfilingDataUpdateEventArgs*<*TCallTree*>> delegate, that is called back when a profiling metadata 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, send commands to the backend and the second, the action thread, listens for action 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 counterpart to the actions and commands in the backend.

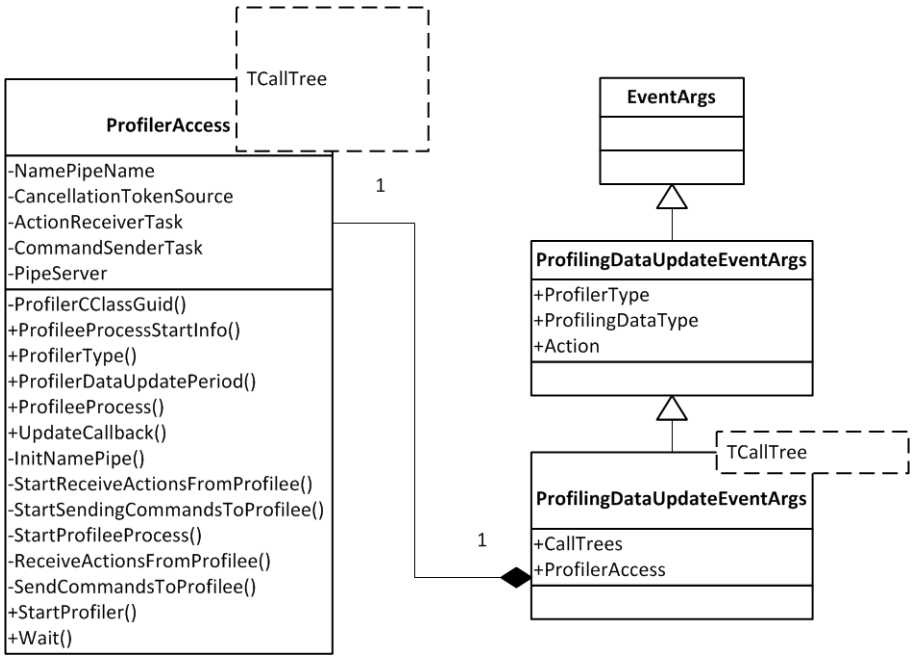

Figure 5.4: The *ProfilerAccess< TCallTree >* class and the from the *System.EventArgs* derived *ProfilingDataUpdateEventArgs* class.

The command thread sends with in the constructor specified period the SendProfilingData command to the backend. The action thread listens for the actions. When the *Sending-ProfilingData* action is received the action thread deserialize the containing profiling data and pass them as the event argument to the specified callback delegate and executes it on a separate thread pool thread. This process repeats itself until the profiling application exits.

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

5.4 Summary

In this chapter, we have looked at the 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.

Bibliography

[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.

[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`, 2011.