

MASARYK UNIVERSITY
FACULTY OF INFORMATICS



Custom Roslyn Tool for Static Code Analysis

MASTER'S THESIS

Zuzana Dankovčiková

Brno, Spring 2017

MASARYK UNIVERSITY
FACULTY OF INFORMATICS



Custom Roslyn Tool for Static Code Analysis

MASTER'S THESIS

Zuzana Dankovčíková

Brno, Spring 2017

This is where a copy of the official signed thesis assignment and a copy of the Statement of an Author is located in the printed version of the document.

Declaration

Hereby I declare that this paper is my original authorial work, which I have worked out on my own. All sources, references, and literature used or excerpted during elaboration of this work are properly cited and listed in complete reference to the due source.

Zuzana Dankovčíková

Advisor: Bruno Rossi PhD

Acknowledgement

TODO: This is the acknowledgement...

Abstract

TODO: This is the abstract ...

Keywords

roslyn, C#, compilers, code review, .NET compiler platform, Kentico, analyzer, code fix..., ...

Contents

1	Introduction	1
2	Compilers	2
2.1	<i>Lexical analysis</i>	2
2.2	<i>Syntax Analysis</i>	4
2.2.1	Error Handling	6
2.3	<i>Semantic Analysis</i>	7
2.4	<i>Intermediate Code Generation</i>	7
3	Static Code Analysis	9
3.1	<i>Source Code vs. Compiled Code Analysis</i>	9
3.2	<i>How It Works</i>	10
3.2.1	Build a Model	10
3.2.2	Perform the Analysis	10
3.2.3	Rules	11
3.2.4	Report the Results	12
3.3	<i>What Problems Can Be Solved by SCA</i>	12
3.4	<i>Advantages</i>	13
3.5	<i>Disadvantages</i>	14
3.6	<i>Static Code Analysis Tools Available on the .NET Platform</i>	15
3.6.1	FxCop	15
3.6.2	StyleCop	15
3.6.3	CodeRush Classic	16
3.6.4	Resharper	16
3.6.5	Analyzers Written with Roslyn	16
4	The .NET Compiler Platform	18
4.1	<i>The Compiler Pipeline</i>	18
4.2	<i>The .NET Compiler Platform's Architecture</i>	20
4.2.1	The Compiler APIs	20
4.2.2	Workspaces APIs	21
4.2.3	Feature APIs	22
4.3	<i>Syntax Tree</i>	22
4.4	<i>Semantics of the Program</i>	24
4.5	<i>Analyzers and Code Fixes</i>	25
4.5.1	Diagnostic Analyzer	26

4.5.2	Code Fix Provider	28
4.6	<i>Deployment</i>	29
5	Implementation of Custom Analyzers	30
5.1	<i>The Original BugHunter</i>	30
5.1.1	Need for Semantic Analysis	31
5.1.2	Ease of Use	32
5.1.3	Suppressing the Results	32
5.2	<i>Defining Analyzer Categories</i>	33
5.2.1	Abstraction over Implementation	33
5.2.2	CMS API Replacements	35
5.2.3	CMS API Guidelines	35
5.2.4	CMS Base Classes	35
5.2.5	String and Culture	35
5.3	<i>API Replacement Analyzer</i>	36
5.3.1	Configuration	36
5.3.2	Analyzer for Member Replacement	37
5.3.3	Analyzer for Method Replacement	38
5.4	<i>Method Invocation Analyzer</i>	39
5.4.1	Template Method Pattern	39
5.4.2	Usage	42
5.4.3	Analyzers for String and Culture Checks	43
5.5	<i>Code Fixes</i>	43
5.6	<i>Tests</i>	44
6	Measuring and Optimizing the Performance	47
6.1	<i>CMS Solution Size</i>	47
6.2	<i>Measuring the Performance</i>	48
6.2.1	Performance of Separate Analyzers	49
6.3	<i>Optimizations</i>	51
6.3.1	SystemIOAnalyzer	51
6.3.2	Other Optimizations	54
6.4	<i>BugHunter Analyzers vs Original BugHunter Build Times</i>	56
6.5	<i>Performance Consideration for Large Solutions</i>	56
6.6	<i>Tool Evaluation</i>	56
7	Conclusion	57

Index	58
A Source Codes in IS	59
B Questionnaires	60
C Deployment and Versioning	61
C.1 <i>Two NuGet packages with BugHunter Analyzers</i>	61
C.2 <i>Applying old BH configuration</i>	61
D Analyzers' Documentation	62

List of Tables

- 5.1 Analyzers sorted into categories 34
- 6.1 Statistics about projects with installed analyzers 48

List of Figures

- 2.1 TODO: Phases of the compiler [1] 3
- 2.2 Abstract Syntax Tree 5
- 3.1 The process of static analysis, adopted from [secure-programming-sca] 10
- 4.1 Compiler pipeline [roslyn-overview] 19
- 4.2 .NET Compiler Platform Architecture [roslyn-succinctly] 20
- 4.3 Syntax tree of an invocation expression 23
- 4.4 An example of a Roslyn analyzer syntax node analysis 28
- 4.5 A example of a code fix live preview 29
- 5.1 Class diagram for ApiReplacementForMemberAnalyzer 38
- 5.2 Class diagram for MethodInvocationAnalyzer depicting the template method design pattern 40
- 5.3 Object diagram for EventLogArgumentsAnalyzer using inner class that derives from MethodInvocationAnalyzer 42
- 6.1 Data flow diagram for *Report-Analyzer.ps1* script. First two processes are run arbitrary number of times. 50
- 6.2 TODO change labels. Box plots of execution times of different versions of SystemIOAnalyzer. 54
- 6.3 Activity diagram of SystemIOAnalyzer action. 55

1 Introduction

TODO...

2 Compilers

As per [1], a compiler is a program that can read a program in a *source* language and translate it into a semantically equivalent program in a *target* language while reporting any errors detected in the translation process. The compiler may sometimes rely on other programs. For example, *preprocessor* is responsible for collecting the source code to be fed to the compiler by expanding shorthands (macros) into source language statements.

The compilation process can be divided into two parts: *analysis* and *synthesis*; commonly referred to as *front end* and *back end* of the compiler.

The purpose of the analysis part is to break up the source program into chunks and build up a grammatical structure that it corresponds to, based on the source language grammar. This structure is subsequently transformed into an intermediate representation of the source program. Along the way, the compiler collects information about the program and stores it into a data structure called *symbol table*. If any errors in syntax or semantics are encountered, analysis part shall inform the programmer about the problem. Otherwise, both intermediate representation and symbol table are passed to the synthesis part where they are used for the construction of the target program.

The two main steps of compilation process internally consist of different phases as shown in Figure 2.1. Each phase transforms one representation of source language into another, and passes it to the following phase while working with the symbol table during the process. In synthesis phase, an optional machine-independent optimizations can take place and are done on the top of intermediate representation. After target machine code is generated, additional machine-dependent code optimizations are performed.

For the purpose of this thesis, mainly the analysis part is relevant and following section will elaborate on its respective phases.

2.1 Lexical analysis

The compilation process starts with *lexical analysis* or *scanning*. The scanner transforms stream of characters of the source program, as

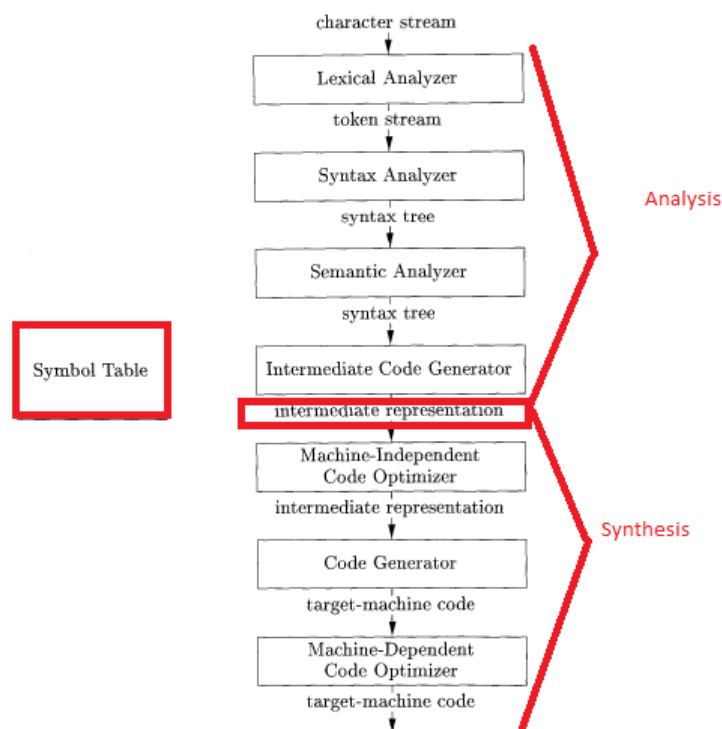


Figure 2.1: TODO: Phases of the compiler [1]

written by the programmer, into the series of meaningful sequences called *lexemes*. Most programming languages allow for an arbitrary number of white spaces to be present in the source text to aid readability. However, white spaces, similarly as comments, are unimportant for the target code generation itself, and thus the lexical analyzer is responsible for discarding them completely.

In order to be able to correctly recognize the lexeme, lexical analyzer may need to read ahead. For example, in C-like languages if the scanner sees < character, it cannot decide whether it is a lexeme for "*less than*" operator or it is a part of "*less than or equal to*" lexeme. In order to do that, it needs to read ahead and see if the following character is = or not. Reading ahead is usually implemented with an input buffer which the lexical analyzer can read from and push back to. The use of buffer also boosts the performance as fetching block of characters is more efficient as fetching one at a time [1].

The lexical analyzer typically uses regular expressions to identify the lexemes and for each lexeme, it outputs a *token* (or *token object*) of the form

$$\langle \text{token-name}, \text{attribute-name} \rangle. \quad (2.1)$$

For an input sequence

$$\text{total} = 42 + \text{base} * \text{interest} \quad (2.2)$$

the scanner output could be

$$\langle \text{id}, 0 \rangle \langle = \rangle \langle \text{num}, 1 \rangle \langle + \rangle \langle \text{id}, 2 \rangle \langle * \rangle \langle \text{id}, 3 \rangle \quad (2.3)$$

Lexemes can be divided into logical groups such as identifiers, relational operators, arithmetical operators, constants or keywords as seen in the example above. The scanner often uses regular expressions to identify tokens.

Each identifier (*id*) has an attribute which points to the entry of the symbol table, where information about identifier name, type or position in the source text is stored. Similar holds for constants like "42" in the example. In the (2.3) example, the assignment and addition symbols do not have attributes but different representation can be used, such as $\langle \text{bin-op}, 2 \rangle$. In this case, *bin-op* would denote it is a binary operator and number two would be a pointer to symbol table with all symbols for binary operations while the second index suggests that it represents an addition.

2.2 Syntax Analysis

The stream of token objects along with partially populated symbol table is an input for the subsequent compiler phase – *syntax analysis* or *parsing*. The parser has to verify that the sequence of token names can be produced by the grammar for the source language and for a well-formed program, it shall output a *syntax tree* or often referred to as an abstract syntax tree (AST)¹.

1. The AST is an intermediate representation of source program in which each interior node represents an operation (programming construct) with the children of the node representing the arguments of that operation. As opposed to *parse syntax tree*, in which interior nodes are nonterminals of the grammar, ASTs are more lightweight and they might omit some nodes which exist purely as a result of grammar's production rules [secure-programming-with-sca].



Figure 2.2: Abstract Syntax Tree

The resulting AST for the token stream generated in (2.2) is depicted in Figure 2.2. The tree shows how multiplication precedence rule of the language's grammar was applied on the expression.

The syntax analyzer uses a context free grammar (CFG) to form the syntax tree. The CFG is defined by a 4-tuple consisting of:

Terminals – token names (first component of the token) as obtained from the previous compilation step.

Nonterminals – syntactic variables that help to impose the hierarchical structure of the language and represent set of strings.

Start symbol – a special nonterminal which set of strings represents the language generated by the grammar.

Productions – rules that specify how nonterminals can be rewritten to sequences of zero or more terminal and nonterminal symbols.

An example of a production denoting the construction of a while-cycle would be

$$stmt \rightarrow \mathbf{while} \ (\ expr \) \ \{ \ stmt \ \}, \quad (2.4)$$

where nonterminals *stmt* and *expr* stand for a statement and expression respectively (defined further by other productions). Symbols in bold represent terminals of the grammar – open and close parenthesis, and curly braces, while keyword.

2.2.1 Error Handling

There are several types of errors that can be encountered during the compilation process:

- **lexical errors** such as misspelling the identifier name,
- **syntactic errors** like missing semicolon,
- **semantic error** for example incorrect number of function arguments,
- **logical errors** that do not really prevent the program from compiling but can indicate possible mistakes (for instance using the assignment operator `=` instead of the comparison operator `==` in condition of an if-statement).

It's parser responsibility to report the presence of potential syntactic error and recover from the error in order to continue with syntactic analysis and be able to detect any subsequent errors. There are two main strategies for the error recovery [1]

Panic-Mode Recovery

In this method, after parser encounters an error, it searches for a *synchronizing token* (usually delimiters such as semicolon or close brace) and until found, all the symbols are thrown away one by one. Even though panic-mode recovery often discards significant amount of input while searching for the synchronization token, it is guaranteed not to end up in an infinite loop.

Phrase-Level Recovery

Another approach the parser can take to recover from an erroneous input is to try to perform a local correction. This can be achieved by replacing the prefix of the following input by some tokens that would enable syntactic analyzer to continue parsing. A prime example of phrase-level recovery is inserting a missing semicolon or replacing comma with a semicolon. Even though this technique is very powerful, as it can cope with all possible problems in the input, it might lead to infinite loops (e.g. always inserting symbols ahead of the current symbol).

2.3 Semantic Analysis

While syntax analysis is able to check the conformance of the program to the grammar of the source language, it is not an ultimate tool. Some language rules cannot be implied by CFG and an additional step is needed to ensure semantic consistency. To do this, the semantic analyzer uses the AST and the information from symbol tables collected in previous phases. While working, it can also add more details about symbols or even modify the AST.

A vital part of semantic analysis for any statically typed language² is *type checking*. Semantic analyzer has to ensure, that each operator is applied to matching operands. For example, a multiplication operator can be called with either a pair of integers or a pair of floating-point numbers, that also implies the result of the operation. If the semantic analyzer encounters an expression where multiplication is used with numbers of different types, it must perform a type conversion called *coercion*. To coerce the integer into floating-point representation it may be necessary to alter the AST and insert an additional node to explicitly state that integer should be treated as floating-point [1].

Semantic analyzer utilizes the information from symbol table to perform all sorts of other checks, to prevent semantic errors such as:

- **wrong arguments** – number and types of arguments applied to a function call,
- **multiple declaration** – variable with the same name declared more than once in one scope,
- **undeclared variable** – usage of a variable before its declaration.

2.4 Intermediate Code Generation

The semantic analysis is followed by the *intermediate code generation* which completes the front end part of the compilation process. Depending on the specific compiler implementation, the *intermediate representation (IR)* that is the result of this phase can take different forms.

2. In statically typed language, type errors are reported by compiler during translation process, whereas in dynamically typed programming languages conversions between incompatible types are only discovered during runtime and can cause program failure.

The IR should be easy to produce and easy to translate into the target machine code. For some compilers, the IR may be the abstract syntax tree itself.

Together with symbol table, IR is passed to back end part of compiler – synthesis, where machine independent optimizations can be performed. These contain *control flow analysis* where control flow graph is constructed and utilized in subsequent *data flow analysis*. As a result of these optimizations, compiler might remove dead code from the IR or perform other optimizations that will lead to shorter and more efficient target code.

The following chapters on static code analysis and .NET compiler platform will build upon fundamentals presented here and show how these concepts are relevant when considering the implementation of a static code analyzer.

3 Static Code Analysis

Static code analysis refers to a process of assessing the program based on its form, structure, content and documentation and reasoning over its possible behaviours without actually executing the code. The aim of static analysis is to check the compliance to specific rules and identify parts of the program that might lead to possible vulnerabilities. The term static code analysis is mostly used when speaking of an automated tool. In contrast, *code inspections* or *code reviews* are performed by humans and can benefit from using static code analysis tools [oswap-sca, ppt-sca].

3.1 Source Code vs. Compiled Code Analysis

There are two different approaches when analyzing a program by an automated tool: analyzing the source code (as seen by the compiler), and analyzing the compiled code – either some form of byte code¹ or an executable.

Sometimes it might be very complicated, or even infeasible, to obtain the actual source code of the program to be analyzed and the only possibility is to analyze the executable code. When the tool is looking at a compiled version of the program, the ambiguity of how the source will be translated by the compiler is removed and thus the analyzer does not need to guess.

However, analyzing compiled code can be very complex. Even if the tool manages to decode the binary, it lacks the original type information. Moreover, the optimizations performed by the compiler obscure the original meaning of the program and making sense of semantics out of implementation may be unattainable. Likewise, if the error is found, reporting it to the programmer can be challenging since there is not always a clear mapping from binary back to source.

Although the above-mentioned complications speak clearly against analyzing binaries, the situation is different when analyzing byte code formats (such as Java bytecode), where the type and debugging

1. An intermediate representation of a program, also known as "portable code", which is often for just-in-time compilation by interpreters.

information is present. The following sections will discuss the theory behind the static code analysis.

3.2 How It Works

There are many tools for static code analysis and each can analyze different flaws in the program. However, for a majority of them, the basic structure looks the same, as depicted in Figure 3.1.

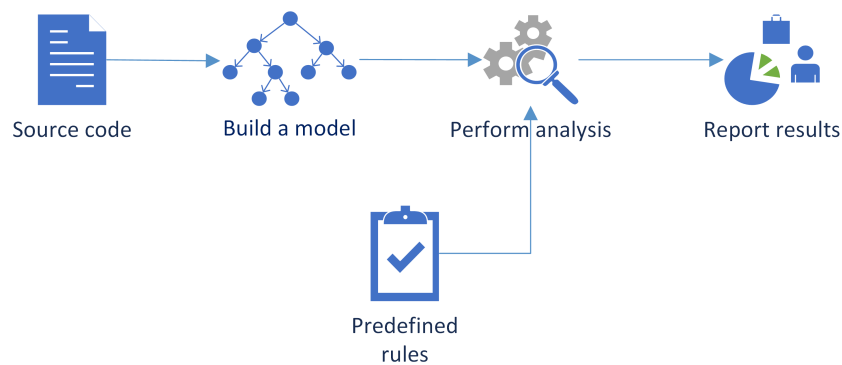


Figure 3.1: The process of static analysis, adopted from [secure-programming-sca]

3.2.1 Build a Model

In order to analyze the program, the analysis tool must first understand it. Therefore, the initial task is to *create a structured model* that represents the source code. This model has a lot in common with the AST and symbol tables that were discussed in Chapter 2. In fact, model building phase of static code analyzers closely mimics the front end part of the compilation process, executing lexical analysis, parsing and semantic analysis.

3.2.2 Perform the Analysis

After obtaining the model, the next step is to perform the actual analysis. Many different algorithms can be applied in this step and it is

common that they are combined into one solution. The approaches are often derived from techniques used by the compilers, specifically:

Control Flow

In order to explore different execution paths that can take place when the program is executed, the static code analysis tool can construct a *control flow graph* on the top of the AST. The nodes of the graph represent basic blocks – sequences of program instructions that will be all executed once block is entered. The edges between basic blocks represent different paths that the program can take depending on matched conditions. Any back edges in the graph signal potential loops in the program execution.

Tracking Data Flow

Data flow analysis is used to examine how data passes through the program. Compilers utilize data flow analysis when doing code optimizations in order to remove unreachable code and allocate registers. An example of how data flow analysis can be used by static analysis tools is to check that memory is always freed only once – function `free(p)` was called at most once with the address stored in pointer `p`.

Taint Analysis

According to [oswap-sca], taint analysis attempts to identify variables containing possibly tainted user input using data flow analysis technique. If these variables are used as arguments to vulnerable functions without being sanitized first, the tool reports their usage as vulnerable. The taint propagation analysis is particularly relevant for security analysis, a prime example being the detection of a potential SQL injection.

3.2.3 Rules

As stated in [sca-for-security]: “...if a rule hasn’t been written yet to find a particular problem, the tool will never find that problem.” This implies that the rules that specify what the static analysis tool should report are just as important (or even more important) as the heuristics and algorithms implemented by the tool. Best tools for static code analysis

externalize the rule set in order to easily add, remove or alter the rules, without modifying the tool itself.

3.2.4 Report the Results

An often overlooked part of the static analysis is the result reporting. The [coverity-sca] asserts, that if a programmer cannot understand the output of the static analysis, the results are effectively useless as misunderstood explanation ends up with error being ignored or, worse, interpreted as a false positive.

As discussed in [security-programming-sca], good static analysis tool should provide means of *grouping and sorting* the results, *suppressing the unwanted results* (either directly in the code with pragmas or code annotations or alternatively in a configuration file) and mainly *explaining the results*. Every issue that is detected by the tool should provide a short title followed by a detailed description of the problem, severity of the issue, recommendations on how the problem can be fixed and possible further references to the topic. The tool can additionally provide a confidence level estimating the likelihood that the finding is really correct.

3.3 What Problems Can Be Solved by SCA

There are different types of problems the static analysis tool can tackle. This section enumerates some of the categories applicable to static code analysis, as listed in [secure-programming-sca].

Type Checking

The integral part of every compiler for statically typed language. Rules are typically implied by the language itself.

Style Checking

The style checker defines rules for spacing, naming, commenting and general program structure that affect mostly the readability and the maintainability of the program.

Program Understanding

These tools aim to provide a high-level program understanding beneficial mainly for larger codebases. They are most effective when integrated into the IDEs where they can support "go to declaration" or "find all references" features or even automatic program refactorings such as renaming or extracting a variable.

Bug Finding

The purpose of these type of static analyzers is to point out common mistakes in the code. They report warnings in parts of program that are compliant with the language specification but might not express the programmer's intent, such as ignoring the return value of a function call.

Special type of bug finding checker is *security review*, where specific vulnerabilities found in the source code are reported. Security review searches for possible exploitations like buffer overflow or tainted inputs.

3.4 Advantages

One of the key factors that advocate the use of tools for static analysis, is how early in the development process they can be applied. As opposed to dynamic testing, static code analysis can be performed on unfinished or even uncompileable code. The longer the defect stays in the system, the more damage it can cause and the higher are the costs of fixing it. As stated in [code-complete], the costs of fixing a defect introduced during construction of a program are 10-times higher if detected during system testing and 10 to 25-times higher in production, than it would be to fix it while still in development. Therefore, it is desirable to detect bugs as early as possible, which is where static code analysis can be leveraged.

Static inspections detect symptoms together with causes whereas testing only points out the symptoms with further effort required to find the source of the problem before it can be fixed [code-complete].

Manual code inspections can be very time-consuming and require high level of expertise from the reviewer. Static code analysis help

to make the code review process more efficient by checking for well-known flaws which do not have to be considered during code review.

Another advantage of automated code analysis is repeatability and scalability. Code analysis tool can be part of Continuous Integration² (CI) process and can be also integrated to programming IDEs³.

As such, they are a great for programmers who get instant feedback and learn more about mistakes they made. The tools enforce higher code quality and guidelines compliance. As a result, the code should be more consistent, maintainable and easier to debug.

3.5 Disadvantages

The Rice's theorem [**direct-proofs-of-rices-theorem**] says, that any non-trivial question about program's semantics is undecidable. As a consequence, there will never be a static analysis tool able to answer all the questions perfectly. The tools can produce *false positives* (a problem which does not actually exist is reported) and *false negatives* (the program contains a problem, but it was not reported by the tool).

Prevailing complaints against static analysis tools concern false positives. A long list of false positives means real bugs can be overlooked and programmers can eventually lose trust in the tool.

Worse, from the security perspective, are false negatives. Not only the bug was not found and might cause future problems, but they also provide a false sense of security to the programmers.

As presented earlier in this chapter, a vast majority of code inspection tools must build a model of the source program in order to be able to analyze it. This requires duplication of compiler's logic, which itself is fairly complicated, and there is no guarantee that the tool interprets the source exactly the same as the compiler does. Moreover, for the authors of the tool, it means the parsing logic has to be always up to date with the language version in use.

2. Process in which developers contribute regularly (multiple times a day) into a shared repository where the code is continuously being verified by an automated build and suite of automated tests.

3. Integrated Development Environment

3.6 Static Code Analysis Tools Available on the .NET Platform

Even if imperfect, static analysis tools are still valuable asset in software development process. This section presents tools for static code analysis that are commonly used on the .NET platform.

3.6.1 FxCop

FxCop is a free tool by Microsoft for analyzing managed code assemblies (targeting .NET Framework) for conformance to .NET Framework Design Guidelines⁴ in areas such as design, localization, performance, security, naming or portability.

It includes more than 200 predefined checks and a possibility to add custom rules using FxCop SDK⁵. It is available in two forms: fully featured application with graphical user interface and a command line tool that is easily integrated to automated build process.

3.6.2 StyleCop

Another tool by Microsoft is an open source project StyleCop⁶ for analyzing C# code for conformance to style and consistency with .NET Framework Design Guidelines. Unlike FxCop, StyleCop analysis is performed on the source code, which enables it to look for a different set of style violations. The rules are divided into categories such as documentation, naming, ordering, spacing and readability.

Some of the rules are: placing the opening curly brace on a new line, spaces around binary operators, method names starting with an upper-case. The tool is configurable and development team can specify its own style to be checked, for example, to enforce spaces over tabs. It is available either a Visual Studio extension or a NuGet package that can be installed to the project.

4. [https://msdn.microsoft.com/en-us/library/ms229042\(v=vs.110\).aspx](https://msdn.microsoft.com/en-us/library/ms229042(v=vs.110).aspx)

5. Software Development Kit

6. <https://github.com/StyleCop/StyleCop>

3.6.3 CodeRush Classic

CodeRush Classic⁷ is a solution-wide static code analysis tool for Visual Studio by vendor DevExpress. It enhances the IDE with more advanced features like assembly decompilation, automated code generation, advanced code selection, code formatting and cleanup. The tool focuses on developer's productivity by not only finding bugs but also providing an automated way of fixing them.

It provides an API enabling developers to extend the basic functionality with 3rd party plugins such as spell checker or copy project. The CodeRush Classic provides static analysis not only for .NET languages, but also for JavaScript, HTML and XML.

3.6.4 Resharper

Very similar to CodeRush, ReSharper is a Visual Studio extension for .NET developers by Jet Brains⁸. It analyzes code quality of C#, Visual Basic, ASP.NET, JavaScript, TypeScript, CSS, HTML and XML. For each of these languages it is possible to define code style and formatting to make the tool compatible with the coding standards followed by a development team.

ReSharper provides hundreds of quick-fixes that solve discovered problems and has support for automated solution-wide refactorings. On the top of static analysis there are additional plugins for performance (dotTrace) and memory (dotMemory) profiling, test runner and code coverage tool (dotCover) or .NET decompiler and assembly browser (dotPeek).

3.6.5 Analyzers Written with Roslyn

The tools described above have one aspect in common – they all need to parse the code before they can analyze it. The cost of maintaining a custom C# parser and keep it up to date with every new language version is fairly difficult, inefficient, and of course, costly.

With the release of new .NET Compiler Platform (Roslyn), which is discussed in detail in the following chapter, the need for parsing C#

7. <https://www.devexpress.com/Products/CodeRush>

8. <https://www.jetbrains.com/resharper>

and Visual Basic sources is eliminated and tools that build upon this platform can concentrate solely on analysis itself.

Some vendors, like Jet Brains, who invested years of development into the creation of the tools, claim [2], it does not pay off to rewrite the whole program so that it uses new Microsoft compiler. Not only would it take an enormous effort to rewrite all the functionality to use new framework, but they would risk destabilizing the product and losing years of optimizations and testing. Moreover, ReSharper is multilingual tool whereas .NET Compiler platform "only" provides C# and Visual Basic parsers.

Other companies, such as DevExpress with CodeRush⁹ or Microsoft with StyleCop Analyzers, decided to use this new approach. The effects on Visual Studio performance were immediate, since solution does not have to be parsed by the tool, nor duplicate syntax trees stored. As a result, the load times and memory consumption were significantly lowered.

The Roslyn APIs also gave rise to more open source projects dealing with static code analysis, such as CodeCracker¹⁰. As challenging as it was in the past, static code analysis is now rather easy, thanks to powerful analysis APIs. Following chapter takes a look at the .NET Compiler Platform and how it can be used to perform the static code analysis.

9. CodeRush Classic refers to the version before Roslyn

10. <https://code-cracker.github.io>

4 The .NET Compiler Platform

In the .NET world, the compiler used to be a black box that given the file paths to the source text, produced an executable. This perception was changed in 2015 when Microsoft introduced the .NET Compiler Platform (commonly referred to as "Project Roslyn").

Not only have been compilers for both Visual Basic and C# rewritten into the entirely managed code¹, they also expose the internals of the compiler pipeline via a public .NET API². This makes them a platform (also known as *compiler-as-a-service*) with rich code analysis APIs that can be leveraged by developers to perform analysis, code generation or dynamic compilation in their own programs [3]. Those can be then easily integrated into Visual Studio all without the hard work of duplicating compilers' parsing logic.

This chapter will take a look at how the Roslyn API layers are structured, how the original source code is represented by the compiler and how developers can build tools upon the compiler's API. Note, that although Roslyn provides equivalent APIs for both VisualBasic and C#, this thesis only focuses on the latter since it is relevant for the practical part of the thesis.

4.1 The Compiler Pipeline

Roslyn compilers expose an API layer that mirrors the traditional compiler pipeline (see Figure 4.1). Instead of a single process of generating the target program, each compilation step is treated as a separate component [roslyn-overview]:

- **Parse phase** consists of *lexical analysis (scanner)* and *syntactic analysis (parser)*. First, the lexical analyzer processes the stream of characters from the source program and groups them into meaningful sequences called *lexemes*. Those are subsequently

1. The term managed code refers to program source code written in one of high-level programming languages available for use with Microsoft .NET Framework, and require a Common Language Runtime virtual machine in order to be executed.

2. Application Programming Interface

processed by the *syntax analyzer* that creates a tree-like structure of tokens based on the language grammar [1].

- **Symbols and metadata phase** where named symbols are generated based on the declarations from the source and imported metadata.
- **Bind phase** in which the identifiers from the source code are matched to their respective symbols.
- **Emit phase** where all the gathered information is used to emit an assembly.



Figure 4.1: Compiler pipeline [roslyn-overview]

In each phase, the .NET Compiler Platform creates an object model containing gathered information and exposes it through the API in form of .NET objects. These objects are also used internally by Visual Studio³ to support basic IDE functionality. For instance **syntax tree**, that is the result of the parse phase, is used to support formatting and colorizing the code in the editor. The result of the second phase – **hierarchical symbol table**, is the basis for *Object browser* and *Navigate to* functionality. Binding phase is represented as an **object model that exposes the result of the semantic analysis** and is utilized in *Find all references* or *Go to definition*. Finally, the Emit phase produces the Intermediate Language (IL) byte codes and is also used for *Edit and Continue* feature [roslyn-overview].

3. The new generation of Visual Studio leveraging from the Roslyn compiler are called vNext and first one was VS 2015.

4.2 The .NET Compiler Platform's Architecture

The Roslyn's architecture consists of two main layers - Compiler and Workspaces APIs, and one secondary layer - Features API, as seen on Figure 4.2.



Figure 4.2: .NET Compiler Platform Architecture [roslyn-succinctly]

One of the key concepts of .NET Compiler Platform is immutability. The compiler exposes hundreds of types that represent all information about source code from Project and Document to SyntaxTrees with almost all of those types being immutable. This means, that once created, the object cannot change. In order to alter it in any way, new instance must be created, either manually, or from an existing instance by applying one of many `With()` methods that the API provides.

The immutability enables the compiler to perform parallel work without need to create duplicate objects or apply any locks on them. This concept is useful for the command line compiler but it is considered extremely important for IDEs where it enables for one document to be handled by multiple analyzers in parallel.

4.2.1 The Compiler APIs

As discussed in the previous section, the Compiler APIs offer an object model representing the results of syntactic and semantic analysis produced by the respective phases of the compiler pipeline. Moreover, it also includes an immutable snapshot of a single compiler invocation, along with assembly references, compiler options, and source files.

This layer is agnostic of any Visual Studio components, and as such can be used in stand-alone applications as well. There are two separate, though very similar, APIs for Visual Basic and C#, each providing functionality tailored for specific language nuances.

Diagnostic APIs

Apart from parsing code and producing an assembly, the compiler is also capable of raising diagnostics, covering everything from syntax to semantics, and report them as errors, warnings or information messages [3]. This is achieved through the compilers' Diagnostics APIs that allow developers to effectively plug-in to compiler pipeline, analyze the source code using the exposed object models, and surface custom diagnostics along with those defined by the compiler itself. These APIs are integrated to both MSBuild ⁴ and Visual Studio (2015 and newer), providing seamless developer experience. Since the practical part of this thesis, and both Chapter 5 and 6, rely on Diagnostic APIs to provide custom diagnostics, they are discussed in more detail in Section 4.5.

Scripting APIs

As a part of the compiler layer, Microsoft team has introduced new Scripting APIs that can be used for executing code snippets. These APIs were not shipped with .NET Compiler Platform 1.0 and are part of v2.0.0 RC3⁵.

4.2.2 Workspaces APIs

Workspace represents a collection of solutions, projects, and documents. It provides a single object model containing information about the projects in a solution and their respective documents; exposes all configuration options, assembly and inter-project dependencies, and provides access to syntax trees and semantic models. It is a star-

4. The Microsoft Build Engine <https://github.com/Microsoft/msbuild>

5. Release candidate 3, as per <https://github.com/dotnet/roslyn/wiki/Scripting-API-Samples> [26-02-2017].

ting point for performing code analysis and refactorings over entire solutions.

Although it is possible to use the `Workspace` outside of any host environment, the most common use case is an IDE providing an instance of `Workspace` that corresponds to the open solution. Since the instances of `Solution` are immutable, the host environment must react to every event (such as user key stroke) with an update of the `CurrentSolution` property of the `Workspace`.

4.2.3 Feature APIs

This layer relies on both compiler and workspaces layers and is designed to provide API for offering code fixes and refactorings. Features APIs were also utilized while working on the practical part of this thesis.

4.3 Syntax Tree

As mentioned in the previous sections, the product of the syntactic analysis is a syntax tree. It enables developers to work with the code in a managed way instead of working against plain text. Syntax trees are used for both analysis and refactorings, where the new code is generated either manually or as a modified version of the existing tree. While being immutable, syntax trees are thread-safe and analysis can be done in parallel.

It is important to point out, that in a same way the compiler constructs a syntax tree from the source text, it is also possible to round-trip back to the text representation. Thus, the source information is always preserved in full fidelity. This means that every piece of information from source must be stored somewhere within the tree, including comments, whitespaces or end-of-line characters, which is a major difference to the general concept of compilers discussed in Chapter 2.

Figure 4.3 shows a syntax tree of an invocation expression as obtained from Syntax Visualizer⁶ extension available in Visual Studio. This tool is useful for understanding how Roslyn represents particular

6. <https://roslyn.codeplex.com/wikipage?title=Syntax%20Visualizer>

language constructs and is widely utilized whenever one needs to analyze the code. Following sections explain what are the main building blocks of such syntax tree, referring to Figure 4.3.



Figure 4.3: Syntax tree of an invocation expression

Syntax Nodes

Syntax nodes (blue color) are non-terminal nodes of a syntax tree, meaning they always have at least one other node or token as a child. Nodes represent syntactic constructs of a language such as statements, clauses or declarations. Each type of node is represented by a single class deriving from `SyntaxNode`. Apart from common properties `Parent`, `ChildNodes` and utility methods like `DescendantNodes`, `DescendantTokens`, or `DescendantTrivia`, each subclass exposes specific methods and properties. As shown in Figure 4.3, `InvocationExpression` has two properties, `IdentifierName` and `ArgumentList` both of which are `SyntaxNodes` themselves.

Syntax Tokens

As opposed to nodes, syntax token (green color) represent terminals of the language grammar, such as keywords, punctuation, literals and identifiers. For the sake of efficiency, `SyntaxToken` is implemented as a value type (C# structure) and there is only one for all kinds of tokens. To be able to tell them apart, tokens have `Kind` property. For example, `SomeFunction` is of kind `IdentifierName`, whereas `"("` character is `OpenParenToken`.

Syntax Trivia

In order to enable refactoring features, syntax trees must also store information about whitespaces, comments and preprocessor directives that are insignificant for compilation process itself. This information is represented by another value type – `SyntaxTrivia` (white color). Trivia are not really parts of the tree itself, rather they are properties of tokens accessible by their `LeadingTrivia` and `TrailingTrivia` collections.

4.4 Semantics of the Program

As explained in Chapter 2, even though syntax trees are enough to describe the proper form of the program (compliance to the language grammar), they cannot enforce all language rules, for example, type checking. In order to tell whether a method is called with the right number of arguments, or operator is applied to operands of the right type, it's inevitable to introduce semantics.

Its one of the core compiler's responsibilities to populate symbol tables with information about all elements and their properties from the source program. Attributes such as identifier name, type, allocated storage, scope; or for method names the number and types of arguments and their return values; are all stored in order to be utilized later when producing intermediate language.

Symbols

In .NET Compiler Platform, a single entry of a symbol table is represented by a class deriving from `ISymbol`. The symbol represents every distinct element (namespace, type, field, property, event, method or parameter) either declared in the source code or imported as metadata from a referenced assembly. Each specific symbol has its own methods and properties often directly referring to other symbols. For example `IMethodSymbol` has a `ReturnType` property specifying what is the type symbol the method returns.

Compilation

An important immutable type, that represents everything needed to compile a C# (or Visual Basic) program is a *Compilation*. It contains all source files, compiler options and assembly references. *Compilation* provides convenient ways to access any discovered symbol. For instance, it is possible to access the entire hierarchical symbol table rooted by global namespace, or look up type symbols by their common metadata names.

Semantic Model

When analyzing a single source file of a compilation, all its semantic information is available through a *semantic model*. The *SemanticModel* object can answer many questions such as:

- What symbol is declared at the specific location in the source?
- What is the result type of an expression?
- What symbols are visible from this location?
- What diagnostics are reported in the document?

This makes semantic model very useful when performing static code analysis concerned with more than just syntax.

4.5 Analyzers and Code Fixes

Thanks to the Compiler APIs it is possible for Visual Studio to provide live static code analysis detecting any code issues as programmer types. Apart from general analyzers that are shipped with Visual Studio, it is possible to define custom, domain specific rules. The tricky task of running the analysis on background thread, showing squiggles in the IDE, populating the *Error List*, and providing the *light bulb* with code fix suggestions, is left to Visual Studio.

In order to understand the chapters describing the implementation part of the thesis, it is vital to know how the Diagnostic APIs work and how they are leveraged when writing a custom analyzer.

4.5.1 Diagnostic Analyzer

An analyzer is an instance of a type deriving from `Microsoft.CodeAnalysis.Diagnostics.IDiagnosticAnalyzer` which must be annotated with `DiagnosticAnalyzer` attribute specifying the targeted programming language (C# or Visual Basic)[3]. In the text of this thesis, these instances are referred to as "*end-analyzers*", in order to distinguish them from abstract analyzers or helper classes containing analysis logic. The end-analyzer can contain one or more custom *rules* for detecting domain specific errors or code issues. Each such rule is defined by `DiagnosticDescriptor`. Once the analyzer finds an issue, the diagnostic descriptor is used to create a *diagnostic* which also includes data collected by the compiler, such as location.

Diagnostic Descriptor

For any Roslyn end-analyzer it is mandatory to override an immutable property `SupportedDiagnostics`. It returns an immutable array of diagnostic descriptors which consist of:

- DiagnosticId** – unique value identifying a single rule (e.g. "*BH103*").
- Title, MessageFormat, Description** – localizable strings that will be displayed in Error List or diagnostic's tooltip on hover. The message format can be also passed arguments upon diagnostic creation, to give more details about the concrete issue detected.
- Category** – the category that the analyzer belongs to, such as code style, naming, design, etc. The list of categories defined for this thesis can be found in Section 5.2.
- DefaultSeverity** – one of Error, Warning, Info, Hidden, None. Note, that only Error severity prevents the project from compiling successfully.
- IsEnabledByDefault** – when this flag is set to false, the rule must be turned on manually in rule set editor.
- HelpLinkUri** – optional URI with online documentation.

Performing the analysis

Besides providing the code that performs the actual analysis (commonly referred to as *action*), it is necessary to register the action to tell the .NET Compiler Platform when to invoke it. To do so, the

end-analyzer must override the `Initialize` method. This method represents the entry point of the analyzer and is invoked exactly once per *session*⁷. The method accepts one argument, of type `AnalysisContext`, exposing number of methods for registering actions. Depending on when the action should be invoked, a callback with analysis logic is passed to one of these methods [**analyzer-action-semantics**]:

RegisterSyntaxNodeAction – the action will be invoked on every syntax node encountered by the compiler if the kind of the node matches one of the kinds provided upon registration.

RegisterSymbolAction – the action will be invoked on complete semantic processing of a symbol that matches one of symbol kinds that the action was registered with.

RegisterSyntaxTreeAction – the action will be invoked as soon as whole document is parsed.

RegisterSemanticModelAction – the action will be invoked after the semantic analysis of the document is finished.

RegisterCompilationStartAction – the action will be invoked when the compilation starts (before any other actions). The context can be then used to register other actions within the compilation as well as the corresponding `RegisterCompilationEndAction`.

RegisterCodeBlockStartAction – action will be invoked before any of actions applicable to syntax nodes within the code block have been invoked. Optionally the analyzer might register the code block end action (`RegisterCodeBlockEndAction`).

RegisterCompilationAction, **RegisterCodeBlockAction** – the action will be invoked once per compilations, or code block, respectively.

The compilation or code block start actions are usually registered for so called *stateful analyzers*. These report diagnostics about specific code unit, as syntax node or a symbol (same as *stateless analyzers*), but within the enclosing unit of code block or the compilation. They have to be designed carefully to perform effectively and not cause any memory leaks.

7. For batch compilations the session is a single run of the compiler. For hosted environments, where the analysis runs on the background thread, the session can last as long as the IDE is open. For more information see [**analyzer-action-semantics**]

For both categories of analyzers it is vital to be written defensively and not to throw any exceptions. Otherwise, Visual Studio might disable it. An example of an analyzer detecting an access to the `Cookies` property of `HttpResponse` object can be seen in Figure 4.4. It is actually a very simplified version of `HttpResponseCookiesAnalyzer` implemented for this thesis.

```
public override void Initialize(AnalysisContext context)
{
    context.RegisterSyntaxNodeAction(Analyze, SyntaxKind.SimpleMemberAccessExpression);
}

private void Analyze(SyntaxNodeAnalysisContext syntaxNodeAnalysisContext)
{
    var memberAccess = syntaxNodeAnalysisContext.Node as MemberAccessExpressionSyntax;
    if (memberAccess == null || memberAccess.Name.Identifier.ValueText != "Cookies")
    {
        return;
    }

    var semanticModel = syntaxNodeAnalysisContext.SemanticModel;
    var accessedType = semanticModel.GetTypeInfo(memberAccess.Expression).Type as INamedTypeSymbol;
    if (accessedType == null || accessedType.ToString() != "System.Web.HttpResponse")
    {
        return;
    }

    var diagnostic = Diagnostic.Create(Rule, memberAccess.GetLocation(), memberAccess);
    syntaxNodeAnalysisContext.ReportDiagnostic(diagnostic);
}
```

Figure 4.4: An example of a Roslyn analyzer syntax node analysis

4.5.2 Code Fix Provider

A code fix is a quick action suggesting a possible solution to a diagnosed code issue found by the analyzer. It can be applied by invoking the quick actions (Ctrl+.) integrated into Visual Studio light bulb. Before applying, developer can view the live preview of the code fix action, as depicted in Figure 4.5.

A custom code fix is defined by a class deriving from `Microsoft.CodeAnalysis.CodeFixer`. By overriding the `FixableDiagnosticIds` it claims which diagnostics it solves. The logic of fixing the diagnosed issue is placed in `RegisterCodeFixAsync` method, which ultimately generates whole new fixed document.

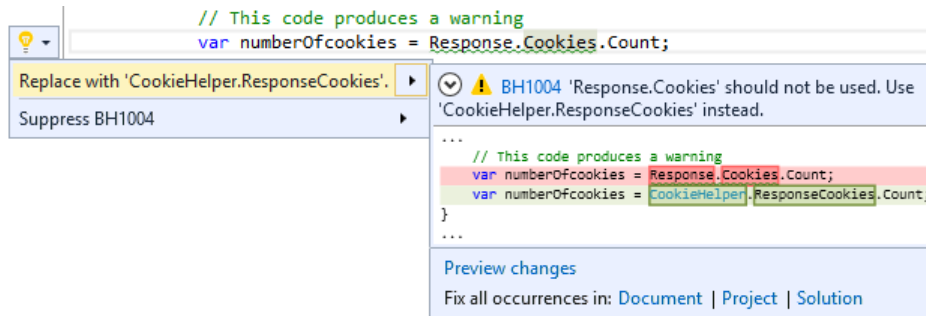


Figure 4.5: A example of a code fix live preview

4.6 Deployment

There are two different approaches when it comes to deploying the analyzers and code fixes: NuGet package, or Visual Studio extension.

With the IDE extension, every developer has to install it manually from Visual Studio Gallery⁸. Once installed, the analysis is performed on any open solution or project. Therefore, it is only appropriate for analysis concerned with common language or framework guidelines.

On the other hand, when the analyzers are installed as a NuGet package. Only the project that has it installed will be analyzed. The reference to the NuGet is stored in shared repository, and everyone working with the project has the analyzers installed automatically. Moreover, only the NuGet installation influences the compilation process and, as such, can be run on build boxes as part of continuous integration process.

8. Visual Studio Gallery provides an access to various tools that extend the default Visual Studio functionality.

5 Implementation of Custom Analyzers

Kentico Software is an IT company based in Brno, developing an all-in-one solution for web content management, e-commerce, and online marketing, using the ASP.NET¹ architecture. The leading product is Kentico CMS (Content Management System). It has been on the market since 2004 with the 11th version currently being developed.

Over the course of almost 13 years, the solution grew to an enormous extent. To give an overview, the current version of the Kentico CMS solution contains 249 projects and a total of 14,881 documents. A lot of knowledge was accumulated with many internal utilities developed and guidelines created. Many features, such as globalization, were developed in-house first, only to be added as an integral part of the .NET Framework a few years later.

The size of the company grew to more than 80 developers working on one product. It became significantly harder to share all the knowledge about the internal best practices and the process of onboarding new employees was challenging. With the increasing complexity of the project, the main focus of the code reviews was shifted to the general architecture and inspection of how the new code influences the other parts of the system, in order not to break anything. The manual and often repetitive tasks of checking the compliance with the conventions and correct use of the API during the code review became very tiresome and dragged the focus of the reviewer away from the higher perspective on the code. This led to a need for an automated tool that would take care of the repetitive checks that needed to be done by the reviewer.

5.1 The Original BugHunter

BugHunter is a simple console application developed at Kentico that searches for violations of internal rules and best practices in the CMS solution. Given the file path to the solution folder, the BugHunter performs the checks and outputs the results to a file. Each issue found is reported with a short message plus additional information on its

1. Open source framework for developing web applications by Microsoft.

location in the code (source file and line number). It contains various checks not, only for C#, but also a few for JavaScript (like no skipped tests allowed), ASPX² (preferred usage of Kentico controls to default ones), or XML files.

Following sections explain the main downsides of the original BugHunter and how this thesis addresses them.

5.1.1 Need for Semantic Analysis

The biggest disadvantage of the previous BugHunter was that the tool performed the checks purely by string comparison. It did not build any model of the analyzed code and therefore possessed no knowledge of the semantics.

This caused an enormous number of false negatives. A typical example would be the tool trying to prevent the access to the `Cookies` property of the `HttpRequest` object. The old BugHunter check looked something like this:

```
if(line.Contains("Request.Cookies")){// report an error}
```

If the programmer used the most common approach³ of accessing the `Cookies` collection on the `Request` object, everything was fine and the BugHunter correctly reported the problem. However, it had no means of detecting other possible accesses to the property. If the `Request` object was passed to a function as an argument or stored in a variable, the accesses of the cookies collection could not be detected by the BugHunter.

There were other use cases that could lead to false negatives when using just primitive string comparisons. For example, in C# it is allowed to put an arbitrary number of whitespace characters between the object, the dot token, and the property being accessed. This could have been solved by using a bit more complicated regular expression, but it would still cover only a fraction of use cases. Things like using variable or even alias usings⁴ would stay undetected by the tool.

2. TODO

3. Accessing the static property of the class that is handling the request.

4. [https://msdn.microsoft.com/en-us/library/aa664765\(v=vs.71\).aspx](https://msdn.microsoft.com/en-us/library/aa664765(v=vs.71).aspx)

The only way to solve this was to introduce the semantics into the analysis which was a relatively easy task for new Roslyn analyzers developed as part of this thesis.

5.1.2 Ease of Use

As mentioned in the Chapter 3, the key to success of a static code analysis tool is the ease of use. This means that running the tool should not be complicated and interpreting the results should be straightforward. If this is not the case, programmers will not use the tool or they will ignore the results it produces.

With the original BugHunter, a developer had to run the separate application that took for about 30 seconds. Then, he or she had to analyze the results before going back to the IDE in order to look up the reported issues and fix them.

In a discussion with the developers at Kentico, they all admitted, that more often than not, they submitted the changes to the version control system without running the BugHunter locally.

The console application was run by the build server itself as part of the continuous integration. If the BugHunter detected any issues, the developer who caused them would be notified by an email and would have to fix them as part of the no-warning policy. This process prolonged the time it took to finish the implementation and generated a demand for a more developer-friendly solution.

Since Roslyn analyzers are part of Visual Studio IDE, programmers get an instant feedback and see the warnings before submitting the code without the need to run an external application.

5.1.3 Suppressing the Results

Another downside of the previous solution was the way certain portions of the code were excluded from the analysis. Since the BugHunter is a console application and the CMS solution is completely independent of it, the configuration had to be placed in BugHunter's installation directory. This was deemed as not transparent, as it was not clear, why in one file certain code was okay, whereas, in the other, it was reported as an issue, without taking a look into a huge configuration file.

Moreover, the only way to exclude a certain piece of code from the analysis was to put the whole file, in which it appeared, into the configuration of the rule to be suppressed. This way of configuration lacked the granularity and was opaque.

On the other hand, Roslyn provides numerous ways to turn off a certain rule for a project, a file, or even a line of code. More information on which approach was taken when deploying the analyzers can be found in the Appendix C.

5.2 Defining Analyzer Categories

The `Category` property of the `Microsoft.CodeAnalysis.DiagnosticDescriptor` class provides a way for semantically distinguishing the analyzers and giving additional information on what type of problem the particular category is concerned with. The original BugHunter did not group the checks into the semantic categories but was structured rather by the technology targeted by the check (C#, ASPX, JavaScript, web parts, etc.).

Therefore, one of the first tasks before the implementation was to determine which checks are suitable candidates for refactoring into new Roslyn analyzers and subsequently group them into categories by the type of the task they performed. Basically, all the checks that were not obsolete and were concerned with C# internal guidelines were rewritten. Table 5.1 lists all implemented analyzers sorted into categories and following sections briefly elaborate on how the categories were defined and what they represent.

5.2.1 Abstraction over Implementation

Analyzers in this category check that there is no "hard reference" on third party libraries in the code. Violation of this rule would require clients to use a particular version of the library which might collide with the references on other components they are using, resulting in so-called "dependency hell"⁵. Therefore, in CMS solution the hard references are replaced with a thin layer of interfaces and adapters.

5. <https://devnet.kentico.com/articles/referencing-multiple-versions-of-the-same-assembly>

5. IMPLEMENTATION OF CUSTOM ANALYZERS

Category	Analyzer name
AbstrOverImpl	LuceneSearchDocumentAnalyzer
CmsApiReplacements	HttpSessionSessionIdAnalyzer
	HttpSessionElementAccessAnalyzer
	HttpRequestCookiesAnalyzer
	HttpResponseCookiesAnalyzer
	HttpRequestUserHostAddressAnalyzer
	HttpRequestUrlAnalyzer
	HttpRequestBrowserAnalyzer
	HttpResponseRedirectAnalyzer
	HttpRequestQueryStringAnalyzer
	PageIsCallbackAnalyzer
	PageIsPostBackAnalyzer
	FormsAuthenticationSignOutAnalyzer
	ClientScriptMethodsAnalyzer
	SystemIOAnalyzer
CmsApiGuidelines	WhereLikeMethodAnalyzer
	EventLogArgumentsAnalyzer
	ValidationHelperGetAnalyzer
	ConnectionHelperExecuteQueryAnalyzer
CmsBaseClasses	ModuleRegistrationAnalyzer
	WebPartBaseAnalyzer
	PageBaseAnalyzer
	UserControlBaseAnalyzer
StringAndCulture	StringManipulationMethodsAnalyzer
	StringEqualsMethodAnalyzer
	StringCompareToMethodAnalyzer
	StringStartAndEndsWithMethodsAnalyzer
	StringIndexOfMethodsAnalyzer
	StringCompareStaticMethodAnalyzer

Table 5.1: Analyzers sorted into categories

The analyzers make sure, only the interfaces are used throughout the solution.

5.2.2 CMS API Replacements

In Kentico CMS solution there are many helpers that encapsulate the traditional .NET API and might provide an extended functionality. For example, determining the current browser of the user can be done by accessing the `Browser` property of the `HttpRequest` object. However, if this attempt fails there are still other ways to resolve it and the CMS API typically has a helper for such cases. The analyzers detect accesses to members that have an equivalent in CMS API and suggest code fixes.

5.2.3 CMS API Guidelines

These analyzers impose internal guidelines on the CMS API usage. They forbid the direct access to database from presentation layer (`ConnectionHelperExecutequeryAnalyzer`), point out usages of methods that may have more suitable or a safer alternatives¹ (`WhereLikeMethodAnalyzer` or `ValidationHelperGetAnalyzer`), or guide developers to use predefined constants where possible (`EventLogArgumentsAnalyzer`).

5.2.4 CMS Base Classes

Similar to CMS API Replacements category, the analyzers search for classes that inherit from standard .NET classes with CMS alternatives and suggest the correct CMS replacements.

5.2.5 String and Culture

This category of analyzers makes sure that string comparison⁶ and manipulation⁷ methods are always used with the overload specifying `StringComparison` or `CultureInfo` parameter explicitly. This is vital for an application like CMS that is used with culture specific data. For

6. `Equals()`, `Compare()`, `IndexOf()` and their variants

7. `ToLower()`, `ToUpper()`

more information on what issues can be caused when not following this rule, like the *"Turkish-I problem"*, see [\[best-practices-for-using-strings-in-dot-net\]](#).

5.3 API Replacement Analyzer

The CMS API Replacement category contains 15 analyzers which represents almost one half of total 31 analyzers that were created. Vast majority of these analyzers search for a particular member being accessed or invoked on a particular type. Any such usage found by the analyzer shall be reported as CMS API contains a replacement for it, with a code fix suggested where applicable.

In order to prevent code duplication and ease the process of adding new analyzers, the analysis logic for the analyzers in this category was extracted into two helper classes: `ApiReplacementForMemberAnalyzer` and `ApiReplacementForMethodAnalyzer`. These are instantiated and configured by the concrete end-analyzer to perform the analysis on their behalf.

5.3.1 Configuration

There are only a few things the API replacement analyzer need to know in order to perform the actual analysis. This information is encapsulated in `ApiReplacementConfig` object. The object is passed as a constructor argument upon creation of the API replacement analyzer. It contains following properties:

- `ForbiddenMembers` – member names to look for (e.g. "Cookies"),
- `ForbiddenTypes` – names of fully qualified (super)types the members belong to⁸ (e.g. "System.Web.HttpRequest"),
- `Rule` – an instance of `DiagnosticDescriptor` that defines the diagnostic raised upon detection of a forbidden usage.

The main reason behind `ForbiddenTypes` being an array of strings rather than a single string, is the lack of inheritance hierarchy between

8. The forbidden type is treated as the highest type in the inheritance hierarchy the member could belong to.

`HttpRequest` and `HttpRequestBase` objects in the .NET framework. They basically contain the same members but do not derive from one another⁹ and therefore, the analyzer must treat both types separately.

On the other hand, allowing for multiple forbidden members on one type under one diagnostic ID may also make sense, which is why `ForbiddenMembers` are also defined as an array.

5.3.2 Analyzer for Member Replacement

The task for the `ApiReplacementForMemberAnalyzer` is to subscribe to all possible member accesses and analyze them. If the particular access is regarded as forbidden, given the configuration supplied (`ApiReplacementConfig` object), the analyzer raises a diagnostic.

It is not enough for the analyzer to subscribe only to `SimpleMemberAccessExpression` syntax kind. It also needs to analyze the `ConditionalAccessExpression` (null conditional operator, `'?.'`) which is a new C# 6.0 language feature¹⁰.

Even though it would be possible to subscribe to both kinds of syntax nodes with one callback, the underlying syntax of these two is so much different, it would make the code cluttered and full of if-statements. Moreover, if Microsoft decides to add another possibility to access members in next language version, the existing code would have to be modified.

Therefore, it was decided to encapsulate the strategies for analyzing a particular syntax node into separate classes. The API replacement for member analyzer instantiates the strategy helpers, configures them, and tells them to run the analysis methods as callbacks subscribed syntax node of a particular kind. This makes the code easily extendible.

The UML class diagram for `ApiReplacementForMemberAnalyzer` is depicted in Figure 5.1. Note, that the `DiagnosticAnalyzer` is an abstract class defined in the .NET Compiler Platform Diagnostics API and must be extended by the custom end-analyzers.

9. More information on why this is so can be found in <https://msdn.microsoft.com/en-us/library/system.web.httprequestwrapper.aspx>

10. <https://msdn.microsoft.com/en-us/library/dn986595.aspx>

5. IMPLEMENTATION OF CUSTOM ANALYZERS

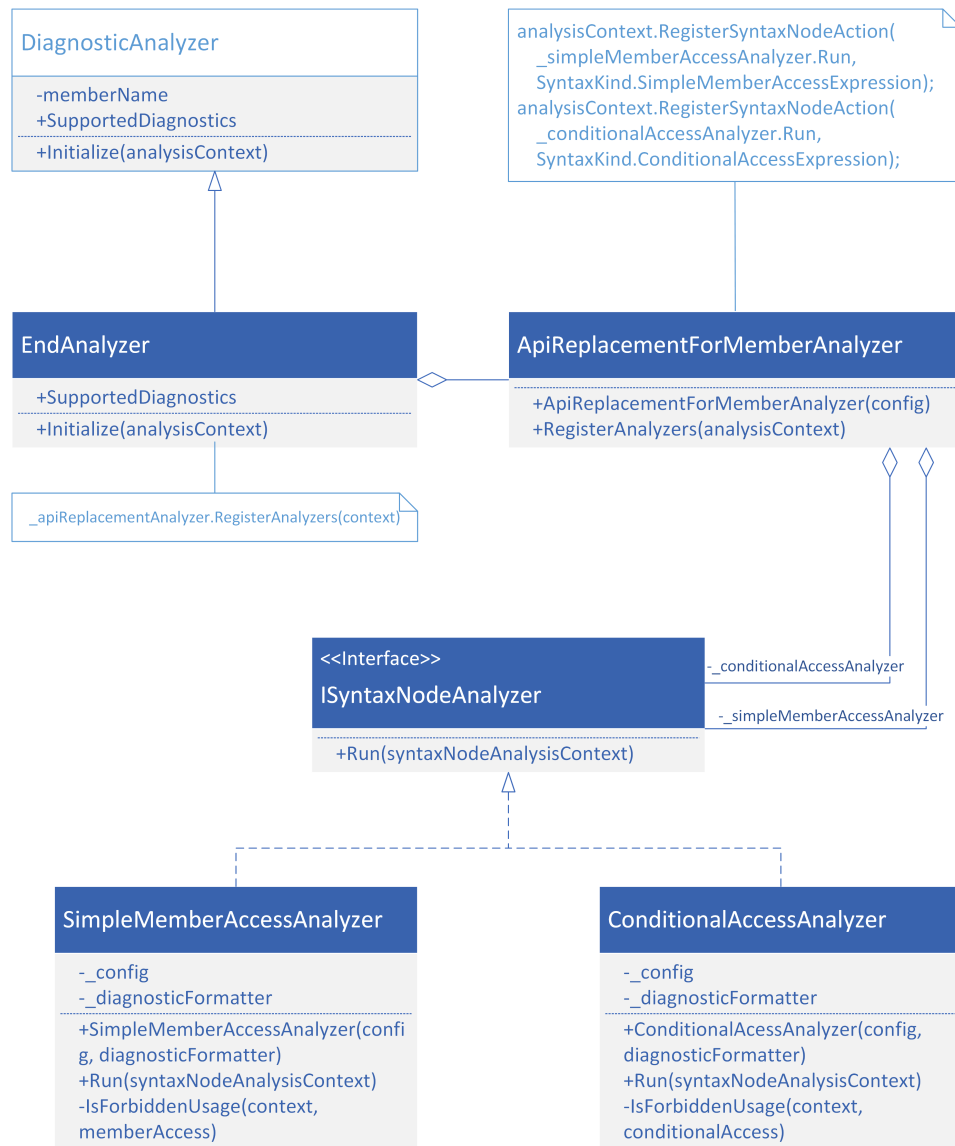


Figure 5.1: Class diagram for ApiReplacementForMemberAnalyzer

5.3.3 Analyzer for Method Replacement

The class `ApiReplacementForMethodAnalyzer` is very similar to analyzer for members that was introduced in the previous section. It also accepts a configuration object defining the invocations of which mem-

bers on which types are forbidden. Instead of subscribing to member accesses it is interested in syntax nodes of kind `InvocationExpression`. It delegates the analysis itself to a strategy defined in `MethodInvocationAnalyzer`, described in the following section.

5.4 Method Invocation Analyzer

This analyzer helper defines an algorithm for analyzing invocation expression syntax nodes. The need for analyzing the method invocations is not limited to the API replacements. The same analysis needs to be performed when enforcing internal guidelines that are concerned with methods. It is important to locate the specific methods invoked on specific types, before it is possible to analyze further whether or not they are used according to the internal API conventions.

5.4.1 Template Method Pattern

This analyzer is utilized by all end-analyzers that involve method invocation analysis. In order to be easily extendible for advanced checks, it is implemented as a template method pattern¹¹. Every step of the template method defines a criteria that must be met in order for the analysis to proceed. Steps that can be overridden have default implementation that continues with the analysis.

It is important to note that the steps are ordered in a way so that the inexpensive checks are performed first, in order to cut off the analysis of the irrelevant nodes without significant performance costs. Figure 5.2 shows the class diagram of the `MemberInvocationAnalyzer` depicting the use of template method design pattern. Following section elaborate on respective steps of the algorithm. Concrete use case depicting an instance of the analyzer can be seen in Figure 5.3.

Check the File Path

Sometimes, an analyzer needs to pre-filter files that are relevant for the diagnostic. For example, when looking for an access to the database layer from the presentation layer, the analyzer is only concerned with

11. <http://www.oodeesign.com/template-method-pattern.html>

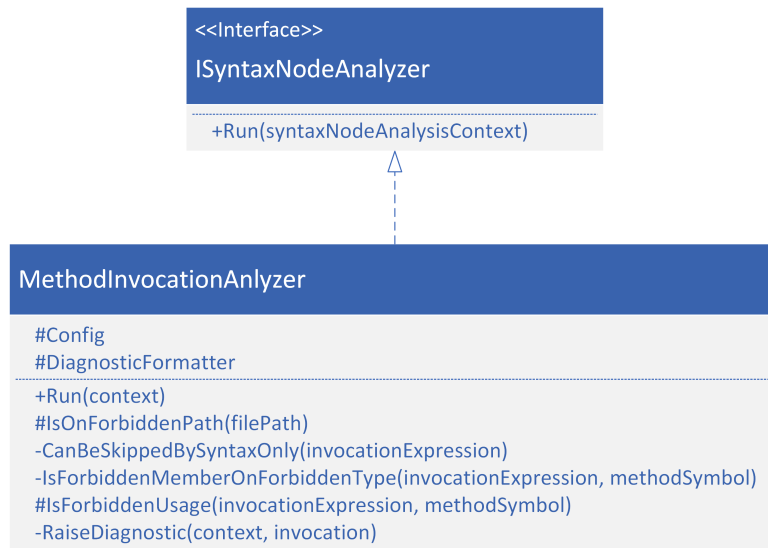


Figure 5.2: Class diagram for MethodInvocationAnalyzer depicting the template method design pattern

the invocation expressions in the files of presentation layer. These can be identified by inspecting the `FilePath` property of the `SyntaxTree` in which the current invocation expression is located. If the file path does not meet the requirements, the current invocation expression does not need to be analyzed further.

Method Name Syntax Analysis

This step exists purely for performance optimization reasons and cannot be overridden by subclasses. It tries to extract the name of the method invoked and compares it to the forbidden members. If the method name is not among the forbidden members, the algorithm stops and no expensive operations had to be done. Otherwise, the algorithm advances to the next step.

Since there are no constraints for what kind of syntax node the `Expression` property of the `InvocationExpressionSyntax` node could be, the logic had to be implemented by hand. The heuristic inspects these three cases:

- **Identifier name** – the simplest case where the method being invoked is a member of the enclosing class.

TODO example + graph

- **Simple member access expression** – invocation on member that is being accessed on object, or class. Accesses can be possibly chained.

TODO example + graph

- **Conditional access expression** – similar to the previous case but the access is conditional. The whole syntax tree is right-folded as opposed to the simple member access left-folded syntax tree.

TODO example + graph

Method Name and Receiver Type Semantic Analysis

Here, semantic model is queried for `IMethodSymbol` that matches the analyzed `InvocationExpressionSyntax`. If the method symbol is found, its property `Name` is checked against forbidden member names. If the check passes, the property `ReceiverType` is inspected and only if it is one of forbidden types or its subtype, the algorithm proceeds.

This method cannot be overridden as it is the integral part of the whole algorithm. It is also the most expensive one, since the checks in this method are performed on symbols obtained by querying the semantic model. However, thanks to the heuristics from the previous step, the majority of irrelevant cases was already filter out.

Context of the Invocation Usage

If it is important to analyze the specificities of the invocation usage (mainly the arguments supplied), the subclasses can override `IsForbiddenUsage()` virtual method. The template method supplies it with two parameters: the invocation expression and the method symbol obtained in the previous step.

Diagnostic Creation

If all the steps above determined the analyzed invocation is forbidden, the last step of the template method makes sure the diagnostic is raised, using the diagnostic descriptor provided upon creation. It is possible to configure the diagnostic creation by providing optional instance of `IDiagnosticFormatter`. It is an interface defined in BugHunter project that determines how the diagnostic of certain syntax nodes shall be formatted (location and message text of the diagnostic).

5.4.2 Usage

Prime example how to use this template method is `EventLogArgumentsAnalyzer`, depicted in Figure 5.3. It defines an inner class that subclasses the `MethodInvocationAnalyzer` overriding its `IsForbiddenUsage()` method.

Then, this inner class is utilized by the end-analyzer (`EventLogArgumentsAnalyzer`) to perform the analysis. Upon instantiation the end-analyzer provides it with a configuration object and a custom diagnostic formatter.

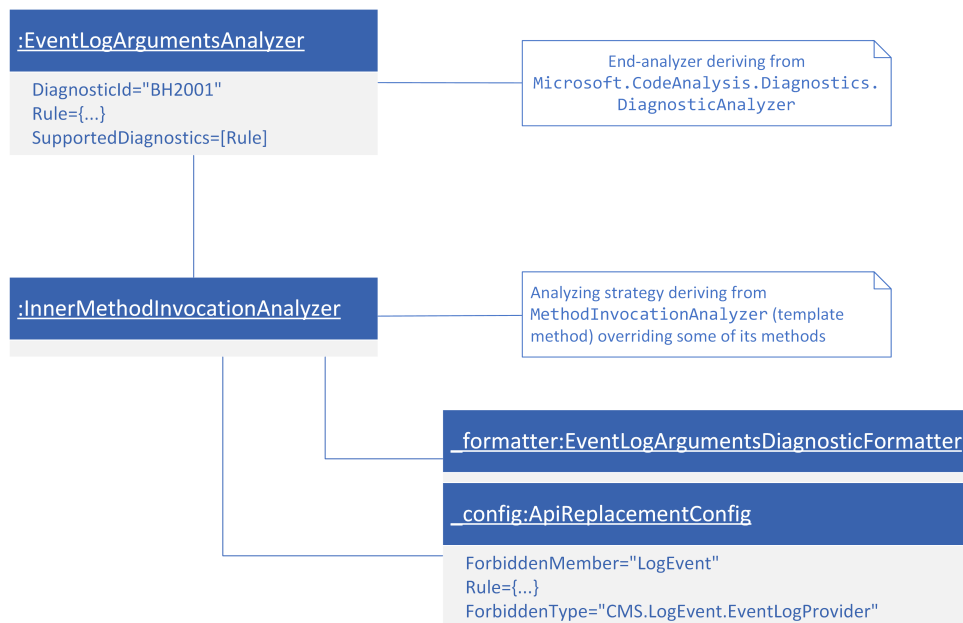


Figure 5.3: Object diagram for `EventLogArgumentsAnalyzer` using inner class that derives from `MethodInvocationAnalyzer`

5.4.3 Analyzers for String and Culture Checks

The analyzers in this category search for methods like `Equals()`, `CompareTo()`, or `IndexOf()`. Considering that methods with same names can be found on many other types as well (`Object`, `Collection`), the first syntax check for method name will often fail to filter out the unrelated usages. Although all these analyzers could use the `MethodInvocationAnalyzer` as a strategy, it was decided to take a different approach here.

The fact that they only need to analyze the invocations on the `String` type, makes the complex analysis of the forbidden types in `MethodInvocationAnalyzer` ineffective. `String` is a sealed class¹², and so no complicated traversal of inheritance hierarchy is needed. In fact, in C# strings are a *special type* and therefore, only a simple check like this is sufficient:

```
receiverType.SpecialType == SpecialType.System_String
```

5.5 Code Fixes

All analyzers that were implemented provide one or more code fixes. There are only two exceptions:

- `SystemIoAnalyzer`, since there is no one to one mapping between `CMS.IO` and `System.IO` APIs so it is not feasible to provide a code fix in every situation,
- `ConnectionHelperExecuteQueryAnalyzer`, which detects database access from presentation layer that points out a possible design flaw without an automated solution.

The code fixes have one significant limitation – providing a fix to an API replacement when the forbidden access is conditional. This is due to the nature of the C# grammar and the way how null conditional operator is represented in the syntax tree. If the access is chained, it would be very difficult to define the exact part of the syntax tree that should be replaced. In fact, there is even an issue¹³ on GitHub of the Roslyn team explaining in detail why this is problematic.

12. Sealed classes cannot be subclassed.

13. <https://github.com/dotnet/roslyn/issues/3110>

Therefore, it was decided not to provide the code fix in such complicated cases. There were two ways for preventing code fix from being suggested. First one was to give the diagnostic on conditional access different diagnostic id from the easily-fixable member access. This was deemed as not particularly user friendly and other options were sought.

The approach taken was to provide an additional information to the diagnostic via the `Properties` property¹⁴. It is a data structure where arbitrary key-value pairs can be stored when diagnostic is raised and can be later read in the code fix class. The analyzer stores a flag, telling the diagnostic is raised for conditional member access. When the code fix method is invoked, it can decide whether or not it is capable of providing a fix for the specific conditional member access.

5.6 Tests

Since it is infeasible to test the functionality of all the analyzers manually by running Visual Studio, it is crucial that all the requirements are covered by an automated set of tests.

Every analyzer and its code fixes have an extensive suite of tests covering the functionality. The extension methods and helper classes are covered by separate unit tests as well. The pre-generated boilerplate from Visual Studio for testing analyzers was reused and rewritten using NUnit framework¹⁵. Since the default template did not allow to set anything apart from the sources, it had to be customized and some additional features were added.

A typical scenario for testing an analyzer is to:

1. Prepare the source documents to be tested. Typically one C# class stored in a string.
2. Create a compilation out of the source documents.
3. Run the analyzer under test on the compilation.
4. Compare the output of the analyzer with the expected diagnostics.

14. <http://www.coderesx.com/roslyn/html/93197CDD.htm>

15. Testing framework for .NET languages <https://www.nunit.org>

To test the code fix, it is run on the source containing a diagnostic and the document with applied code fix is compared to the expected result.

Referencing Kentico Libraries in Tests

As can be seen, when testing analyzers and code fixes, the compilation is always created from the source documents. For this to work, all references to the libraries used by the source code, or relied upon by the tested analyzer, need to be linked. This contains not only the Microsoft core assemblies that are always added, but in case of the BugHunter analyzers, also Kentico libraries. Therefore, the default testing template had to be enhanced with the ability to explicitly reference desired DLLs in the created compilation.

Faking File Information of Documents in Compilation

The outcomes of some analyzers are dependant on the file path of the analyzed document. Therefore, to test them, it is necessary to be able to fake the name of the source document that will be part of the analyzed compilation. To accommodate this requirement, the the testing template was extended with `FakeFileInfo` structure. It can be passed as an optional argument to the function testing the analyzers, in order to replace the default file name, path, or extension.

Failing Tests on Uncompilable Sources

Another important feature, that was not present in Microsoft template, is that when the source documents are uncompileable (e.g contain a typo), the tests will fail, no matter if the analyzer and/or code fix worked as expected. Since the compilers generally, and, therefore, also the analyzers themselves, are heavily reliant on string constants, it is very important to have this kind of check present.

Failing the build upon uncompileable source was added later in the development and it uncovered quite a few errors. Some of them were rooted back in the original BugHunter. For example it checked for `HttpRequest.Redirect()` which actually does not exist, be-

cause `Redirect()` method is member of `HttpResponse` object, not `HttpRequest`.

Tracking the CMS API Changes

In order to supply the Kentico DLLs to be referenced in the tested on-demand compilation, the test projects have a dependency on `Kentico.Libraries` NuGet package. This, together with tests failing on uncompileable source, has a very pleasant side effect: It tracks any breaking changes in CMS API that affect the analyzers' or code fixes' logic.

For instance, if some method from Kentico API changes the number of arguments, code fix, that introduces the method to the source code, is invalid and its tests will fail. They will also provide a detailed compiler error message, so the developer knows instantly what issues need to be fixed after the upgrade.

Only major versions can contain a breaking change and these are released once a year in Kentico. To keep the BugHunter analyzers up to date, the referenced NuGet should to be upgraded regularly.

This chapter presented selected implementation details and patterns utilized during the development. The next chapter focuses on the performance aspect of the tool. It describes how the performance of the analyzers was measured and what was done in order to enhance it. It also assesses the tool's usability based on feedback obtained from the Kentico development team.

6 Measuring and Optimizing the Performance

When implementing custom Roslyn analyzers, it is vital to consider their performance. The analyzers run on a background thread of Visual Studio (or parallel threads of compilation process, when outside of the IDE) and if implemented carelessly, they might significantly influence the processing power and the memory consumption.

This holds for any custom analyzers written, but even more so if they are to be used with large solutions like Kentico's. Developers at Kentico complain about the responsiveness of Visual Studio with an opened CMS solution. Some of them even admitted they had uninstalled the ReSharper extension, since it slowed down the IDE to an unbearable extent, and claimed that the plugin often caused IDE crashes.

Therefore, in order for the BugHunter analyzers to be usable, they have to be efficient. This chapter explains how the performance of the implemented analyzers was measured and provides a quick look at how some of them were optimized. The end of the chapter summarizes views of Kentico development team on the usability of new BugHunter.

6.1 CMS Solution Size

Before discussing the performance of the BugHunter analyzers, it is important to understand the structure of the CMS solution. This is helpful when considering which parts of the analyzers' code are crucial for the overall performance and which can be neglected.

In order to collect relevant information, *SolutionStatistics* project was created (the source code can be found in the IS attachment). The project loads the CMS solution and collects information on number of projects, documents, syntax nodes. The summary of the results is presented in Table 6.1. Note, that just the data relevant for implemented analyzers are displayed. Therefore, only projects where NuGet with BugHunter analyzers is installed¹ and relevant kinds of syntax nodes are considered.

1. Test and 3rd party projects were excluded from statistics, see Appendix C.

Number of projects	125
Number of documents	11,136
Number of syntax nodes	6,279,993
- IdentifierNameSyntax	1,422,571
- MemberAccessExpressionSyntax	350,942
- InvocationExpressionSyntax	216,716
- ObjectCreationExpressionSyntax	22,635
- ElementAccessExpressionSyntax	17,930
- ClassDeclarationSyntax	10,288
- ConditionalAccessExpressionSyntax	482

Table 6.1: Statistics about projects with installed analyzers

6.2 Measuring the Performance

——— TODO - rewrite??? ———

Measuring the performance is always tricky regardless of the domain. When one wants to measure performance of something as complicated as code analysis plugged into the compilation process, there is a big probability something will go wrong and result will not be accurate.

In the early stages of development, the initiative was to compute benchmarks for different analyzer versions using BenchmarkDotNet framework². However, the overhead of creating the compilation together with the cost of the analyzers' execution were too big for the results to be useful.

There is too much code related to the overhead of running the actual analysis which cannot be influenced from the outside. There are also differences between running the analyzers as part of the compilation versus when they are run by the IDE, where many additional optimizations³ are performed.

2. <https://github.com/dotnet/BenchmarkDotNet>

3. Only part of the code that has been changed since last compilation is recompiled

6.2.1 Performance of Separate Analyzers

Before considering the overall impact of BugHunter analyzers on the compilation of the CMS solution, it is important to tune the performance of separate analyzers. The only way that Microsoft provides for measuring the analyzers' execution times is `/ReportAnalyzer` switch in CSC (CSharp Compiler).

As stated in **[report-analyzer]**, if the compiler is invoked with `/ReportAnalyzer` command line switch, its output will contain total wall clock time spent in executing the analyzers along with relative times of separate analyzers. Since the analyzers can run concurrently, the total time should only be treated as the upper bound. The relative times per analyzer are useful for comparing their performance and identifying any outliers.

In order to compile the whole solution, one must use the MSBuild process, that internally uses the CSC. The command, that also reports the execution times, looks like this⁴:

```
msbuild /t:Clean,Build /p:ReportAnalyzer=True  
/verbosity:diagnostic CMS.sln
```

Parsing and Aggregating Per-project Results

The resulting MSBuild output is over 500MB large and, besides normal compiler output, it contains information on analyzers' execution times for every project that the analyzers' NuGet is installed to. For CMS solution this means 125 projects. For the purpose of this thesis, however, only performance over the whole solution is relevant.

In order to parse the results from the large output, a console application *ReportAnalyzerTimes.Parser* was developed. It extracts all data related to analyzer execution times from the log (based on the format described in **[report-analyzer]**). After that, it sums up the absolute exe-

4. Note, that the official documentation (<https://msdn.microsoft.com/en-us/library/ms164311.aspx>) does not provide any information on running the MSBuild with reporting the analyzers. The only approach that worked, after numerous efforts, was the one shown above. The verbosity level `diagnostic` is vital here, since `verbosity:debug` completely omitted any information on analyzer reporting.

cution times⁵ for same analyzers across the solution (sums per-project results) and outputs them into a file.

Aggregating Results from Multiple Runs

Since a single compiler run should not be relied upon to provide accurate results, a PowerShell script (*Report-Analyzer.ps1*, see Figure 6.1) was written that executes the CMS solution MSBuild multiple times. After every run, it runs the above mentioned *ReportAnalyzerTimes.Parser* application on the compiler output, and results from all runs are saved in multiple text files.

Once the predefined number of builds were completed, another console application, *ReportAnalyzerTimes.Aggregator*, is utilized to aggregate the results into a single CSV⁶ file. For every analyzer, there is a name and the accumulated per-project execution times from every run of the solution's build.

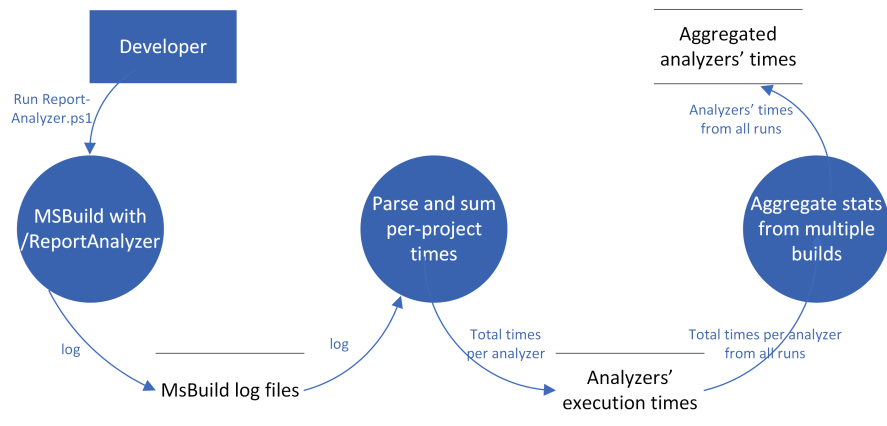


Figure 6.1: Data flow diagram for *Report-Analyzer.ps1* script. First two processes are run arbitrary number of times.

5. The smallest value, that can be reported, for the execution time is <0.001s and it is treated as 0.001s by the parsing application.

6. Comma Separated Values

6.3 Optimizations

The procedure described in the previous section was used to measure and compare the performance of implemented analyzers. The measurements were performed on the CMS solution, as well as on an artificially generated project with uniformly distributed diagnostics for each analyzer (source codes attached in IS).

The slowest analyzers were iteratively optimized and the speed of different versions was also compared using the *Report-Analyzer.ps1* script. The versions that performed the best were then integrated into the final NuGet package.

This section presents the most interesting optimization that was done on `SystemIOAnalyzer` and briefly mentions other optimization techniques as well.

6.3.1 `SystemIOAnalyzer`

After the initial versions of all analyzers were implemented, the first measurements showed that `SystemIOAnalyzer` is by far the slowest one. It took more than 18 seconds to run on the CMS solution, whereas almost all other analyzers executed in less than 0.3 seconds.

This analyzer belongs to CMS API Replacement category and the motivation behind its existence is described in [[system-io-motivation](#)]. It forbids the access to all members from `System.IO` namespace, apart from `System.IO.IOException` and `System.IO.Stream` classes and their subclasses, plus `System.IO.SeekOrigin` enumeration. All other members of the namespace have CMS equivalents, that must be used instead to make sure all parts of the product can work with virtual file systems (like Azure or Amazon storage).

Differences from Other CMS API Replacement Analyzers

What makes this analyzer fundamentally different from all other analyzers in its category is, that instead of blacklisting particular members or methods from Microsoft API, it forbids the whole namespace (all classes, enums, methods, members, etc.) apart from few whitelisted, well defined, exceptions. Other analyzers in CMS API Replacement category can reliably cut off the analysis by looking at syntax and

inspecting the name of a member or a method. Since only certain member names are considered forbidden, vast majority of nodes will only be analyzed on syntax level.

However, it would be infeasible for `SystemIOAnalyzer` to list all possibilities of forbidden API usage. Therefore the analyzer needs to advance to perform more complicated and expensive analysis on symbols. It analyzes whether the symbol is defined in `System.IO` namespace and whether it is not whitelisted. Only then it raises the diagnostic.

Unlike other analyzers in the category, that are bound to member access expressions or invocation expressions, the `SystemIOAnalyzer` must inspect all syntax nodes of kind `IdentifierName`. As seen in Table 6.1, there are more than 1.4 million such nodes. With the naive implementation that directly accesses the semantic model, the performance of the analyzer degrades very quickly. The next two sections explain how the optimization of the analysis algorithm was achieved.

Different versions of `SystemIOAnalyzer`

The very first version of the analyzer (hereafter referred to as V1), had an action registered on syntax nodes of kind `IdentifierName`. It accessed the semantic model straight away in order to perform the analysis on symbols. The V1 took more than 18 seconds to run on the CMS solution.

First optimization attempts were to register the syntax node action within compilation start action. The fastest version with this approach collected all diagnosed nodes in a bag. In the compilation end action, the diagnostics were raised for all collected nodes in a parallel for-loop. On average, this version of the analyzer performed almost 2-times faster than the V1.

However, as pointed out in multiple discussions of Roslyn team's GitHub repository [**compilation-end-action-1**, **compilation-end-action-2**], registering compilation start and its end action, can cause problems for large solutions. This holds particularly for IDE live analysis. Not only can be the analyzer's feedback delayed by several minutes, it might also lead to stale diagnostics in *Error list* window.

The next optimization approach was to minimize the number of actions that need to be invoked. The analyzer must inspect all 1.4

million identifier name nodes. It can either register its action on syntax node of this kind, as the previous versions did, or register a syntax tree action and look for identifier names on its own. This reduces the number of callbacks to at most 11,136 (number of documents).

The logic, however, is more complicated, as the tree needs to be traversed in order to find the nodes to be inspected. On the other hand, if the document does not contain string ".IO" at all, there will be no diagnostic and its analysis can be skipped. The fastest version with syntax tree action took an over 9 seconds to perform the analysis. Even though it was faster than the V1, it was still not ideal.

The Final Version

None of the versions described above provided a significant performance improvement compared to V1. To understand the limitations of the analysis, two artificial analyzers were created, that registered an action on identifier name syntax nodes. The first one only registered an empty callback. The second one obtained the symbol of the inspected node from the semantic model and then did a dummy comparison of its name⁷. Neither analyzer raised any diagnostic in CMS solution.

The results of these measurements were very interesting. While the empty callback took mostly less than 0.5s⁸, the second one run for an average of 7.9 seconds.

These measurement clearly showed, that if the final version was to be faster, it had to implement some heuristic, that would allow to cut off the analysis of inspected node without need to access the semantic model. The analysis logic of the final solution is depicted by the activity diagram in Figure 6.3.

The most important steps of the whole optimization are the second and the third node of the diagram (inspection of containing syntax tree and containing dotted expression⁹). They are performed purely on

7. This was just to make sure that the compiler actually did not skip the semantic model access as part of an optimization, if the obtained symbol had not been used at all.

8. TODO: DO WE WANT THIS OR REAL AVERAGE?? Trimmed average was used for these results, meaning that top and bottom 25% of the results were discarded and the middle 50% was used to compute an average.

9. The *outermost dotted expression of identifier name* is the outermost dotted expression of its parent, if the parent is of kind `QualifiedName` or

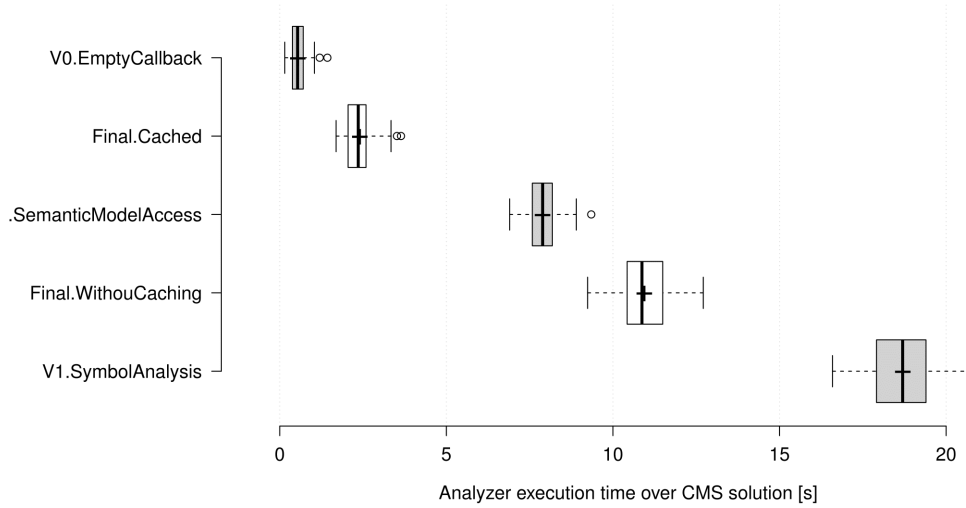


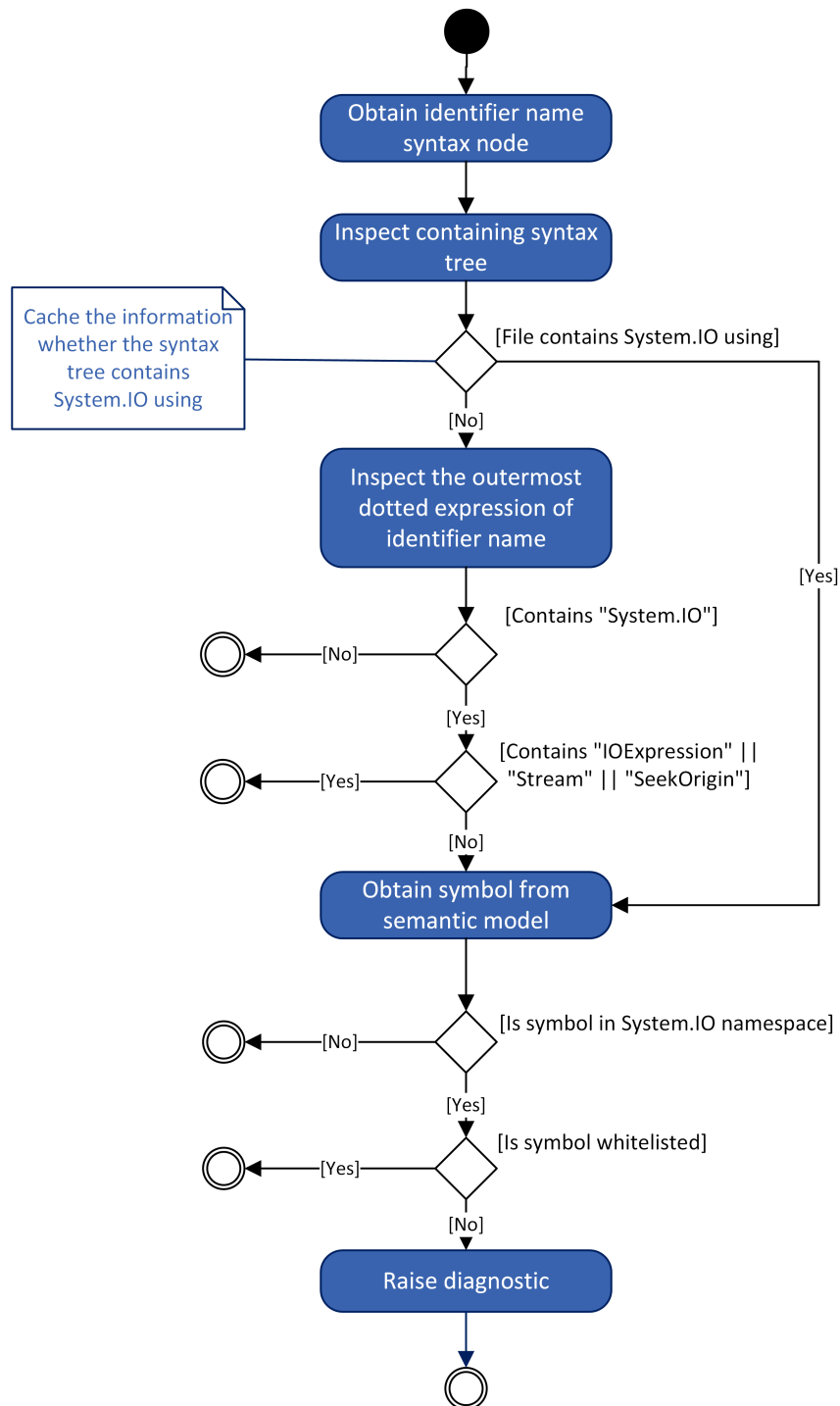
Figure 6.2: TODO change labels. Box plots of execution times of different versions of SystemIOAnalyzer.

syntax and are extremely efficient in eliminating most of the irrelevant nodes.

To prevent multiple inspections of same syntax tree for all its identifier name nodes, the information whether the tree contains `System.IO` using is cached in a concurrent dictionary. This optimization technique managed to shrink the analyzer's execution time on CMS solution to an average 2.36 seconds, whereas the non-cached version took over 10 seconds.

The execution times of final analyzer (with uncached version as well), V1 analyzer and artificial baseline analyzers are depicted in a box plot diagram in Figure 6.2.

`SimpleMemberAccessExpression`, or the node itself otherwise. For example, for identifier name `Stream` which is part of `System.IO.Stream`, the outermost dotted expression is the whole expression.

Figure 6.3: Activity diagram of `SystemIOAnalyzer` action.

6.3.2 Other Optimizations

Just quickly mention other optimizations

CMS Base Classes Analyzers

symbol vs syntax - performance differences

Syntax Heuristics for Method Invocation Analyzer

stats on how much it speeded up relevant analyzers

String Analyzers

how not using the generic MethodInvocationAnalyzer influenced the results

6.4 BugHunter Analyzers vs Original BugHunter Build Times

Compare multiple builds w/o BH, old and new versions... Is it significantly slower??

6.5 Performance Consideration for Large Solutions

<https://github.com/dotnet/roslyn/wiki/Performance-considerations-for-large-solutions> .. can be used later but so far none of team members complained about worse performance

Why stateful analyzers should not have compilation end actions – for large solutions it can take more than one minute for the diagnostic to show up in VS... that is not usable at all and cannot be considered as "live" feedback.

6.6 Tool Evaluation

Questionnaires sent to development team, feedback from senior developers...

7 Conclusion

- what issue it solved (Equals, Response vs Request.Redirect()) - what is the current status of the project - how it helped the development teams... - how to maintain the tool, - how easy it is to add new analyzers,

- talk about kentico selling it's source code, so after stabilization, BH could be also available for them as nuget package and aid with the development practices

Bibliography

- [1] Alfred V. Aho et al. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison Wesley, 2006. ISBN: 978-0321486813.
- [2] URL: <https://blog.jetbrains.com/dotnet/2014/04/10/resharper-and-roslyn-qa/>.
- [3] Alessandro Del Sole. *Roslyn Succinctly*. 2501 Aerial Center Parkway, Suite 200, Morrisville, NC 27560, USA: Syncfusion, 2016.

A Source Codes in IS

The main solution with new Roslyn analyzers (BugHunter.sln) consists of 6 projects within 2 folders: TODO - Do leave the Vsix project there or delete???

TODO - replace with a screenshot???

Analyzers folder with Roslyn analyzers

- BugHunter.Core
- BugHunter.Analyzers
- BugHunter.Web.Analyzers

Tests folder with tests:

- BugHunter.TestUtils
- BugHunter.Core.Test
- BugHunter.Analyzers.Test
- BugHunter.Web.Analyzers.Test

TODO Describe each project.. why 2 for analyzers - nuget distribution

The projects with analyzers use the categories also for folder structure with each folder containing separate subfolders for analyzers and code fixes respectively.

B Questionnaires

TODO...

Sum up the questionnaire results and cite teamleaders' feedback on the tool.

C Deployment and Versioning

TODO...

C.1 Two NuGet packages with BugHunter Analyzers

Why, how – only refer to appendices where this is described

C.2 Applying old BH configuration

what approaches were taken - ruleset file, pragma statements, not installing analyzer nuget at all...

talk about the old configuration of the BH, and how this one maps to Roslyn version (suppression pragmas vs suppression files, some analyzers look at the filepath or file extension directly.. some checks like SystemIO had to be suppressed - level set to none in ruleset file - for files that it did not make sense for... helpers that wrap the "forbidden" functionality were marked with pragma statements around the whole class so that it is clearly visible in code)

D Analyzers' Documentation

TODO... same as will the online documentation?

IDs, categories, which nuget, title, description, what for, severity, enabled by default...