Mendelova univerzita v Brně
Provozně ekonomická fakulta

Automatizace kontroly kódu během vývoje

Diplomová práce

Vedoucí práce:

Ing. Jiří Lýsek, Ph.D. **Bc. Lenka Chalupová**

Brno 2016

Poděkování

Ráda bych upřímně poděkovala panu Zbyškovi Němcovi ze společnosti Kentico software s.r.o, za cenné rady při psaní práce a především za možnost vytvořit nástroj pro společnost Kentico s.r.o. Poděkovala bych také mé rodině za morální podporu během celého studia a v neposlední řadě vedoucímu práce Ing. Jiřímu Lýskovi, Ph.D. za užitečné rady při tvorbě práce.

Čestné prohlášení

Prohlašuji, že jsem tuto práci: Automatizace kontroly kódu během vývoje vypracoval/a samostatně a veškeré použité prameny a informace jsou uvedeny v seznamu použité literatury. Souhlasím, aby moje práce byla zveřejněna v souladu s § 47b zákona č. 111/1998 Sb., o vysokých školách ve znění pozdějších předpisů, a v souladu s platnou *Směrnicí o zveřejňování vysokoškolských závěrečných prací.*

Jsem si vědom/a, že se na moji práci vztahuje zákon č. 121/2000 Sb., autorský zákon, a že Mendelova univerzita v Brně má právo na uzavření licenční smlouvy a užití této práce jako školního díla podle § 60 odst. 1 Autorského zákona.

Dále se zavazuji, že před sepsáním licenční smlouvy o využití díla jinou osobou (subjektem) si vyžádám písemné stanovisko univerzity o tom, že předmětná licenční smlouva není v rozporu s oprávněnými zájmy univerzity, a zavazuji se uhradit případný příspěvek na úhradu nákladů spojených se vznikem díla, a to až do jejich skutečné výše.

V Brně dne 17. května 2016 ____________________________

Abstract

Chalupová, L. Automation of control code during development. Diploma thesis. Brno: Mendel University, 2016.
This thesis deals with the analysis of existing tools designed to check the code throughout the development and implementation of tools for the Kentico company that will check the written code at runtime and suggest repairs so that they are easily applied. This tool will be implemented in the C# programming language and applied in the development environment Microsoft Visual Studio 2015 and created on Roslyn platform.

Keywords

Automation of code check, C#, Roslyn, Microsoft Visual Studio

Abstrakt

Chalupová, L. Automatizace kontroly kódu během vývoje. Diplomová práce. Brno: Mendelova Univerzita, 2016.
Tato diplomová práce se zabývá analýzou existujících nástrojů vytvořených pro kontrolu kódu již během vývoje a implementací nástroje pro společnost Kentico software s.r.o., který bude vykonávat kontrolu psaného kódu během jeho tvorby a navrhovat opravy, tak aby byly jednoduše aplikované. Vytvořený nástroj bude implementován programovacím jazykem C# a aplikovaný ve vývojovém prostředí Microsoft Visual Studio 2015 za pomocí platformy Roslyn.

Klíčová slova

Automatická kontrola kódu, C#, Roslyn, Microsoft Visual Studio

Obsah

1 Úvod a cíl práce 12

1.1 Úvod ... 12

1.2 Cíl práce ... 12

2 Vývojový proces ve společnosti Kentico 14

3 Technologie C# 16

3.1 Platforma .NET framework ... 16

3.2 Historie a verze programovacího jazyka C# ... 17

3.3 Srovnání s programovacími jazyky C++ a Java ... 18

3.4 Vývojové prostředí Microsoft Visual Studio ... 21

3.4.1 Kompilátory ... 21

4 Nástroje pro automatickou kontrolu kódu 25

4.1 ReSharper ... 25

4.1.1 ReSharper API ... 26

4.1.2 Základní přehled funkcí nástroje ReSharper ... 26

4.2 Roslyn ... 31

4.2.1 Architektura Roslyn ... 32

4.2.2 Syntaktické stromy ... 32

4.2.3 Kompilace a sémantický model ... 37

5 Metodika práce 39

5.1 Syntaktické pravidla pro psaní kódu ... 39

5.2 Pravidla pro statickou analýzu kódu ... 40

5.3 Architektura ... 41

6 Implementace nástroje pro automatickou kontrolu kódu 43

6.1 Vytvoření nástroje pro automatickou kontrolu kódu ... 43

6.1.1 Vytvoření analyzátoru kódu ... 44

6.1.2 Oprava kódu pomocí Code Fix ... 48

6.1.3 Třída založena na Rewriter ... 51

6.2 Kompilace nástroje 51
6.3 Možnosti použití analyzátoru 52
6.4 Testování nástroje pro analýzu kódu....54

7 Závěr 58

7.1 Zhodnocení vývoje nástroje....58
7.2 Možnosti budoucího rozvoje 59

8 Literatura 60

A Obsah přiloženého CD 64

B Seznam syntaktických pravidel 65

C Seznam pravidel pro statickou analýzu kódu 66

Seznam obrázků

Obr. 1 Komponenty .NET frameworku 17

Obr. 2 Historie programovacího jazyka C# 18

Obr. 3 Překlad zdrojového kódu kompilátorem 22

Obr. 4 Fáze kompilátoru 22

Obr. 5 API vrstvy 23

Obr. 6 Vrstvy ReSharperAPI 26

Obr. 7 Zobrazení chyby v ReSharper 27

Obr. 8 Návrh rychlé opravy v ReSharperu 27

Obr. 9 Zvýraznění kódu v ReSharperu 28

Obr. 10 CamelHumps v ReSharperu 28

Obr. 11 Inteligentní dokončování kódu v ReSharperu 29

Obr. 12 Automatické generování kódu v ReSharperu 29

Obr. 13 Souborové šablony v ReSharperu 30

Obr. 14 Vyčištění kódu v ReSharperu 31

Obr. 15 Zobrazení vnitřní části kompilátoru 32

Obr. 16 Syntaktický strom 33

Obr. 17 Zapnutí nástroje Syntax Visualizer 34

Obr. 18 Část nástroje Syntax Visualizer – Syntax tree 35

Obr. 19 Část nástroje Syntax Visualizer – Properties 36

Obr. 20 Architektura vytvořeného řešení nástroje 42

Obr. 21 Vytvoření nového projektu s využitím šablon 44

Obr. 22 Zobrazení informací o pravidlu pomocí žárovky ve Visual Studiu 46

Obr. 23 Znázornění opravy jména konstanty složené z více slov 50

Obr. 24 Přidání analyzátoru k danému projektu 52

Obr. 25 Spuštění analyzátoru 52

Obr. 26 Instalace nástroje 53

Obr. 27 Nainstalovaný nástroj ve vývojovém prostředí 53

Obr. 28 Výsledek testovací metody – přijatelná chyba 56

Obr. 29 Výsledek proběhlého testování 56

Obr. 30 Nástrojové okno Error List s nalezenými chybami 57

Seznam tabulek

Tab. 1 **Tabulka s rozdílnými klíčovými slovy v jazyka C# a Java** **20**

Tab. 2 **Příklady syntaktických pravidel pro psaní kódu** **40**

Tab. 3 **Příklady pravidel pro statickou analýzu kódu** **41**

Tab. 4 **Metody pro registraci akcí** **45**

Seznam zdrojových kódů

Zdrojový kód 1 - Implementace informací o pravidlu pomocí lokalizovaných řetězců 46

Zdrojový kód 2 - Vytvoření pravidla se stupněm diagnostiky Error 46

Zdrojový kód 3 - Implmentace pravidla pro konstanty 47

Zdrojový kód 4 - Implementace analýzy pro zakázané metody 48

Zdrojový kód 5 -Implementace zděděných metod v CodeFix pro konstanty 49

Zdrojový kód 6 - Implementace metody MakeUpper 49

Zdrojový kód 7 - Implementace metody pro zakázané metody v CodeFix 51

Zdrojový kód 8 - Metoda pro odstranění regionů 51

Zdrojový kód 9 - Překryté metody v testovací třídě 54

Zdrojový kód 10 - Metoda pro testování víceslovného názvu konstanty 55

1 Úvod a cíl práce

1.1 Úvod

Kontrola kódu je důležitá nejen z hlediska přehlednosti a správnosti, ale také z důvodů kontroly dodržování daných pravidel ať už firemních nebo jen syntaktických a sémantických. Existují všeobecné zásady pro psaní kódu, aby zdrojový kód byl dobře čitelný nejen pro jeho tvůrce, ale v podstatě i pro kohokoli jiného.

Je tomu již desítky let, co se vyvinuly objektově orientované jazyky a byly pro ně vytvořená vývojová prostředí, které nabízí programátorovi celou škálu nástrojů pro usnadnění psaní kódu. Nevymyslí celý kód za něj, ale mohou nabízet různé pomocné funkce, jako například generování některých funkcí nebo zvýrazňovat určité části kódu. Generátory kódu mohou napomoci vývojáři vytvořit například konstruktory dle zvolených parametrů nebo gettry a settry pro proměnné. Zvýraznění částí zdrojového kódu, například klíčových slov, slouží pro vývojáře k lepší orientaci v psaném kódu. Pro vývojové prostředí existuje nespočet doplňků kompatibilních s mnoha různými programovacími jazyky.

Příkladem takových doplňků a v dnešní době nejpoužívanější funkcionalitou jsou nástroje, které by měly napomoci ke kvalitnějšímu kódu a také k dodržení určitých pravidel. Nástroje slouží jako doplněk pro vývojové prostředí, jako je například Microsoft Visual Studio, a mají předdefinovaná různá pravidla pro kontrolu kódu. Nástroje pro automatickou kontrolu psaného kódu mají výhodu, že doplňují vývojáři kód a pomáhají mu tak napsat program rychleji a především kvalitněji. Jedním z takových nástrojů pro vývojové prostředí Microsoft Visual Studio je i platforma Roslyn, která nabízí i možnost dopsání vlastních pravidel, které chceme, aby byly hlídány již během psaní samotného kódu.

Ve společnostech se snaží docílit sjednocení struktury kódu, aby se v něm vyznalo více programátorů. Proto musí být každý napsaný projekt zkontrolován, zda jsou v něm dodržena pravidla nejen struktury a formátu kódu, ale i pravidla definována přímo firmou. Kontrola kódu, která je důležitou součástí vývojového procesu, je v mnoha firmách stále manuální a je pro společnost jak časově tak i finančně náročné.

1.2 Cíl práce

Cílem diplomové práce je vytvoření nástroje, který bude kontrolovat definovaná pravidla pro strukturu, formát a použití konstruktů programovacího jazyka v kódu produktu za vývojáře již během samotného psaní kódu.

Součástí práce je také seznámení s procesem vývoje, nástroji a s pravidly pro psaní kódu, které musí vývojáři ve společnosti Kentico software s.r.o. dodržovat. V práci bude také obsažena analýza již dostupných nástrojů umožňujících kontrolu kódu během jeho tvorby dle zadaných pravidel.

Samotný nástroj pro automatickou kontrolu kódu bude implementován v programovacím jazyku C# za pomocí platformy Roslyn a bude aplikován ve vývojovém prostředí Microsoft Visual Studio 2015.

Přínosem této práce je vytvoření nástroje, který umožní částečně nahradit manuální kontrolu kódu, která je často součástí vývojového procesu automatizovaným a preemptivním přístupem. Součástí práce bude i vyhodnocení přínosu implementovaného řešení a možnosti jeho budoucího vývoje a nasazení ve spločnosti Kentico software s.r.o.

2 Vývojový proces ve společnosti Kentico

Na vývoji hlavního produktu společnosti Kentico CMS/EMS se podílí 7 vývojových týmů. Tento program je sestaven pomocí ASP.NET a .NET frameworku verze 4.6. Kompletní produkt se skládá přibližně z dvou stovek knihoven, z nich přibližně třetina obsahuje testy. Celý vývojový proces je tvořen za pomoci vývojového prostředí Microsoft Visual Studio. Aktuálně se ve společnosti využívají dvě poslední verze, 13 a 15. Vývojář si může zvolit verzi, která mu více vyhovuje.

Společnost vlastní také licence nástroje ReSharper, tento nástroj je využíván přibližně dvěma třetinami vývojářů. Pro nástroj ReSharper má společnost vytvořenou vlastní sdílenou šablonu, pomocí které je zajištěna soudružnost formátování psaného kódu pro všechny vývojáře. Tento nástroj však není přizpůsoben pro statickou analýzu kódu, tedy nenabídne třeba vývojáři náhradu standartní knihovny za knihovnu vytvořenou ve společnosti.

Jelikož vytvořený produkt není tvořen jednotlivým zákazníkům na míru, ale je vyvíjen pro širokou veřejnost, musí zohledňovat a řešit určitou sadu požadavků tak, aby mohl být použit kýmkoli bez úprav. Produkt také slouží jako vývojová platforma, proto se ve společnosti snaží dodržovat určité pravidla, aby jeho API a i psaný kód tvořil dojem, jako by byl vytvořen pouze jednou osobou a působil konzistentně i pro další vývojáře.

Všechny vývojové týmy ve společnosti Kentico vyvíjí pomocí agilní metodiky SCRUM. V rámci této metody musí týmy dodržovat Definition of DONE[1], který jim nařizuje, že veškerý napsaný kód musí projít review. V průběhu review se kontrolují pravidla pro psaní kódu a doporučení pro výslednou architekturu. Ve společnosti Kentico se review provádí buď párově, nebo separátně. Probíhá tak, že zkušenější vývojář ve společnosti projde changsety, které byly vytvořeny jeho kolegou, který na konkrétní issue/story pracoval. Typicky se ve společnosti Kentico provádí manuálně. Především u nových vývojářů ve společnosti, kteří přišli z jiné firmy anebo právě dostudovali, bývá nalezeno největší množství chyb, což je způsobeno především neznalostí veškerých dodržovaných pravidel pro psaní kódu a best practices pro psaní kódu ve společnosti Kentico.

Ve společnosti Kentico mají kromě code reviews vytvořený vlastní nástroj pro statickou analýzu kódu, který je nazván BugHunter. Tento nástroj je vytvořen v programovacím jazyce C# a slouží ke kontrole, zda jsou splněna specifika vývojové platformy, například se ve společnosti Kentico používají některé vlastní vytvořené knihovny a namespacy místo systémových, protože obsahují sofistikovanější nadstavbu a umožňují tak vývojáři větší rozšiřitelnost. Nástroj BugHunter se spouští během přípravy instalačního balíčku, což probíhá přibližně jedenkrát za dvě hodiny a jeho výpis je poslán e-mailem vývojáři, který se přihlásil k jeho odběru. Případné problémy, které neobjeví manuální review, se tak odhalí později.

[1] Definiton of DONE – je jednoduchý seznam činností (psaní kódu, komentáře, testování, poznámky k verzi, projektová dokumentace, atd.), které přidávají prokazatelnou hodnotu výrobku. (Panchal, 2008)

Může nastat, že jsou problémy odhaleny až poté, co je konkrétní issue otestována a musí se tak vrátit zpět celým procesem k vývojáři, který ji vytvořil, případně je na ni zadán defekt, který se musí řešit separátně.

Ve společnosti Kentico se snaží docílit celistvosti kódu za dodržování daných pravidel pro psaní kódu a také pro statickou analýzu kódu. Z těchto důvodů chtějí využít nový kompilátor Roslyn, aby byla dostupná kontrola psaného kódu již během vytváření kódu vývojáři. Kontrola již při psaní kódu usnadní práci vývojáři, který tvoří danou část kódu, ale také by mohla částečně odpadnout manuální kontrola kódu.

Špatně napsaný zdrojový kód, který není zjištěn a opraven již v počátku vývoje, může mít později větší dopad a způsobit i značné obtíže při údržbě kódu a samozřejmě se zvyšují i finanční náklady na údržbu a opravu zdrojového kódu. Největší dopad špatně napsaného kódu je u pravidel pro statickou analýzu kódu, kde vývojář použije k implementaci metodu nebo konstrukt ze standardních knihoven místo vlastní nadstavby vytvořené v Kenticu. Tyto chyby se mohou u zákazníka projevit nedeterministicky a jen v určitých případech. Jelikož se chyba nemusí projevit vždy, ale jen v daných případech, z tohoto důvodu je pro vývojáře, ke kterým je chybný zdrojový kód navrácen, hůře odhalitelná. Při zjištění takových chyb samozřejmě narůstají také režie kolem komunikace se zákazníkem, který chybu nahlásí supportu. Poté musí vývojář nalézt a opravit danou chybu nejen na již vydané verzi, u které byla chyba objevena, ale také na aktuálně vyvíjené verzi. Na obou verzích následuje code review a samozřejmě musí být obě verze opět otestovány. Ve společnosti se snaží vydávat takové opravy včas v rámci tzv. 7 days bug fixing policy, což je náročnější při hůře odhalitelných chybách.

Zjištěná chyba v kódu zpomalí vývoj ve společnosti, kde se musí chyba opravit a otestovat, ale potenciálně i u zákazníka, kterého může zjištěná chyba brzdit ve vlastním vývoji, což může mít za následek i finanční ztráty.

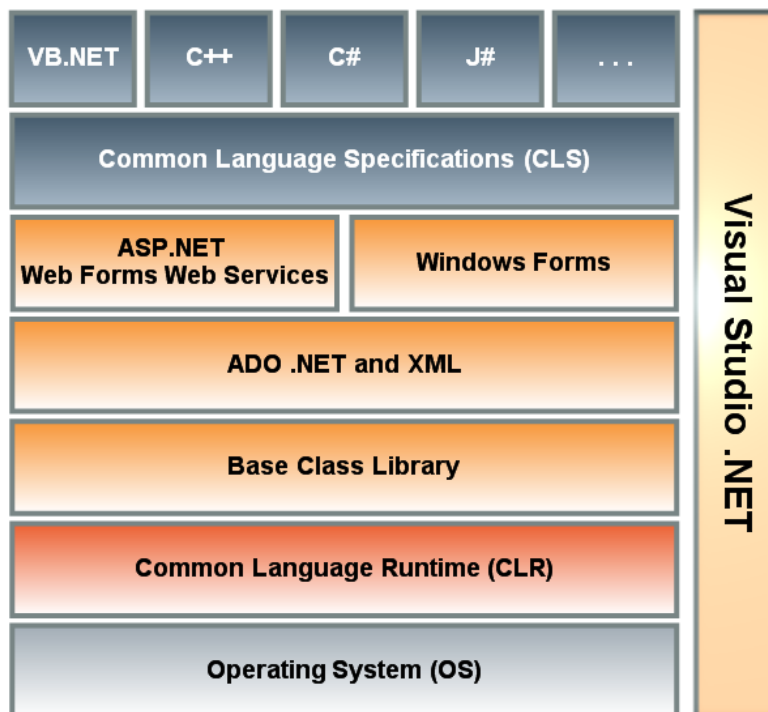
3 Technologie C#

C# je elegantní a typově bezpečný objektově orientovaný programovací jazyk, který umožňuje vytvářet robustní aplikace, které pracují na .NET Frameworku. Programovací jazyk C# lze použít pro tvorbu aplikací klienta systému Windows, webové služby XML, aplikace typu klient-server, databázové aplikace a mnohem více. (Microsoft)

3.1 Platforma .NET framework

Platforma .NET je úzce spjatá s programovacím jazykem C# a byla vyvinuta společností Microsoft ve stejné době. Hlavními důvody pro vývoj platformy .NET byly například nekompatibilita jednotlivých programovacích jazyků a s tím také související obtížnější spolupráce mezi programy a knihovnami napsaných v odlišných jazycích, problémy s verzemi knihoven a nebo zastaralý a nepřehledný způsob vývoje webových aplikací. Tyto a další problémy efektivně řeší platforma .NET za použití řízeného běhového prostředí a systémových knihoven, což jsou základní prvky aplikací. Princip řízených běhových prostředí je také využit u platformy Java firmy Sun Microsystems. Tento princip spočívá v tom, že je přidána ještě jedna vrstva k převodu zdrojového kódu do kódu strojového. Tato vrstva představuje mezikód, do kterého jsou zdrojové kódy zkompilovány a tento mezikód je poté běhovým prostředím na cílové platformě (Windows, Linux) převeden do strojového kódu. Tento způsob však není vhodný pro náročnější aplikace (např. počítačové hry), protože způsob překladu představuje vyšší náročnost na výkon uživatelského počítače. Časté využití má však u obchodních aplikací, které nejsou na výpočetní výkon tolik náročné a jejich rychlost je naprosto vynikající. (Puš, 2004)

.NET Framework se stará o řadu nízko-úrovňových a stereotypních povinností, jakými jsou správa paměti, vytváření a rušení objektů, spouštění a zastavování vláken kódu, bezpečnost a kontrola oprávnění k prováděným operacím, načítání potřebných knihoven a komponent do paměti, apod. O tyto téměř neviditelné, avšak velmi důležité operace se stará část zvaná Common Language Runtime (CLR), kterou lze vidět na Obr. 1 jako nejnižší vrstvu. Další komponentou je Base Class Library (BCL), která obsahuje nejčastější pomocné funkce. Knihovna ADO.NET slouží pro práci s daty s možností jejich XML reprezentace. V další vrstvě jsou knihovny pro vývoj uživatelského rozhraní – Windows Forms pro desktopové aplikace a ASP.NET pro webové uživatelské rozhraní.(Microsoft, 2005) Důležitou částí je Common Language Specification (CLS), což je společná jazyková specifikace, s kterou souvisí Common Type System (CTS). Díky použití CLS a CTS je zajištěna rovnocennost programovacích jazyků. Pro vývoj v .NET aplikací lze využít jeden z programovacích jazyků vyšších úrovně, mezi které patří např.: C#, Visual Basic .NET, J#, C++, JScript apod. (Puš, 2004)



Obr. 1 Komponenty .NET frameworku
Zdroj: (Projectcsharp.com, 2010)

3.2 Historie a verze programovacího jazyka C#

Při vývoji .NET frameworku vznikl tým, který měl za úkol vybudovat nový programovací jazyk, v té době s názvem Cool. V červenci 2000 byl projekt .NET veřejně představen na Professional Developers Conference a programovací jazyk byl přejmenován na C#. Knihovny tříd a ASP.NET běhové prostředí bylo přesunuto na C#.

Na počátku vývoje byl programovací jazyk C# svou syntaxí hodně podobný programovacímu jazyku Java a měl k němu blíže než k programovacímu jazyku C++, který je často považován za předchůdce programovacího jazyka C#. Od vydání verze 2.0 v listopadu 2005 se C# a Java rozvíjí rozdílněji a stávají se tak méně podobné. V následující verzi 3.0 bylo přidáno několik významných funkcí jako např. LINQ, lambda výrazy, rozšíření metod a anonymních tříd. (Hasan, 2012) Verze 4.0 přinesla podporu dynamic/DLR. Ve verzi 5.0 byla přidána vlastnost async a await. Nejnovější verze 6.0 přináší především syntaktická vylepšení jazyka, která by měla usnadnit vývojáři psaní kódu. (Holec, 2015) Nová verze je zaměřena především na optimalizaci každodenních programovacích drobností a vytváření čistšího kódu. Mezi vylepšení nové verze patří:

- Getter-only auto properties – odpadává nutnost definovat private členy pro jednoduché get a set funkce, vlastnosti zde mohou být také pouze read-only
- Auto properties initializers – vlastnosti je možné inicializovat přímo v deklaraci
- Null-conditional operator – ve zdrojových kódech je často kontrolováno, zda objekt, k němuž je přistupováno není null, proto vznikl operátor ?. přezdívaný Elvis
- String interpolation – zjednodušuje formátování řetězců pomocí String.Format, používá se zde $ a odebere se String.Format, nahradí se používané parametry {0} jejich smysluplnými variantami např. {Jmeno}
- Nameof operator – řeší předání názvu členu jako řetězce, které je náchylné na chyby a obtížně se poté opravuje

(Šimeček, 2015)

C#	Release	.NET	Visual Studio	Top features
C# 1.0	2002/1	1.0	VS .NET 2002	MANAGED CODE
C# 1.2	2003/4	1.1	VS. NET 2003	
C# 2.0	2005/9	2.0	VS 2005	Generics; Anonymous methods; Nullable types; Partial class; Covariance Contra-variance
C# 3.0	2007/11	2.0 3.0 3.5	VS 2008 VS 2010	Lambda expressions; Extension methods; Auto-properties; LINQ; Implicit type (var)
C# 4.0	2010/4	4	VS 2010	Optional parameters; Dynamic binding; Named arguments
C# 5.0	2012/9	4.5	VS 2012 VS 2013	Async feature
C# 6.0	2015/?	4.6	VS 2015	Exception filters; String inter-polation; Parameter-less ctors; Auto-property initializers

www.miroslavholec.cz

Obr. 2 Historie programovacího jazyka C#
Zdroj: (Holec, 2015)

3.3 Srovnání s programovacími jazyky C++ a Java

Syntaxe C# zjednodušuje mnoho složitostí z programovacího jazyka C++ a obsahuje spoustu funkcí, které nenabízí vývojáři programovací jazyk Java. Proces kompilace C# je jednodušší než v jazycích C a C++ a flexibilnější než v jazyce Java. V C# odpadává nutnost implementace samostatných hlavičkových souborů a nejsou zde žádné požadavky, aby metody a typy byly deklarovány v určitém pořadí. C#

může ve zdrojovém souboru definovat libovolný počet tříd, struktur, rozhraní a událostí. (Microsoft)

C# na rozdíl od C++ je tzv. čistě objektový jazyk. V Javě a C# na rozdíl od C++ pracujeme pouze s dynamickými objekty vytvářenými pomocí operátoru new. Nemusíme se starat ani o rušení vytvořených objektů, protože .NET Framework obsahuje, stejně jako Java, Garbage colector, který se postará o automatické odstranění nepoužívaných objektů. Další společnou vlastností C# a Javy na rozdíl od C++ je podpora více vláknového programování. Od Javy se však liší například podporou přetěžování operátorů. V C# a C++ lze používat pozdní i časnou vazbu, zatím co v Javě se používá pozdní vazba.

C# obsahuje zcela jiné knihovny než C++ a Java. Tyto knihovny jsou společné pro všechny programovací jazyky pro .NET Framework, některé ze tříd knihoven se mohou podobat třídám knihovny programovacího jazyka Java, avšak práce s nimi se může lišit. (Virus, 2002)

U jazyka C# neexistuje datový typ pointer jako u C++ a používají se místo něj reference, které mohou mít i hodnotu null a jsou na ni automaticky inicializovány. Jazyk C# nepoužívá ani globální proměnné a deklarace třídy není ukončena středníkem. Pořadí tříd v souboru není důležité, programátor nemusí řešit vzájemné vztahy mezi třídami a ani případně vytvářet tzv. dopředné deklarace. Jazyk C# nepodporuje vícenásobnou dědičnost, avšak jedna třída může být implementovat více rozhraní. (Holub, 2002)

C# na rozdíl od jazyka Java umožňuje přetěžování operátorů. Jazyk C# díky automatickým vlastnostem umožňuje jednodušší a přehlednější zápis vlastností, které mají být jen pro čtení, což se v programovacím jazyku Java implementuje pomocí privátní proměnné a k ní vytvořenému getteru.

Za jednu z největších výhod jazyka C# je považován LINQ (Language INtegrated Query), jedná se o způsob tvorby dotazů nad různými datovými strukturami například databázemi. Nezáleží na tom, jestli chceme vybrat prvek z pole, seznamu nebo databáze, vždy je použita stejná syntaxe. (Kulman, 2010)

Další výhodou C# jsou struktury, které v Javě nejsou povoleny. Sice většinou stačí použít běžnou třídu, ale v některých situacích je efektivnější použít struktury. V současné době je jazyk Java podporován na více operačních systémech bez rekompilace kódu na rozdíl od jazyka C#. (Microsoft)

Tab. 1 Tabulka s rozdílnými klíčovými slovy v jazyka C# a Java

C# klíčové slovo	Java klíčové slovo	Popis	C# příklad	Java příklad

Base	super	Prefix operátor, který odkazuje na nejbližší základní třídu.	public MyClass(string s) : base(s)
			{
			}
			public MyClass() : base()
			{
			}
				Public MyClass(String s)
				{
				super(s);
				}
				public MyClass()
				{
				super();
				}

Bool	boolean	Primitivní typ, který vrací hodnotu true nebo false, ale ne obojí.	bool b = true;	boolean b = true;

namespace	package	Vytvoří prostor, aby se zabránilo kolizím názvů.	namespace MySpace
			{
			}
				//package musí být první klíčové slovo v souboru třídy
				package MySpace;
				public class MyClass
				{
				}

using	import	Používá se pro zahrnutí dalších knihoven do projektu.	using System;	import System;

:	extends	Operátor nebo modifikátor v definici třídy, který znamená, že tato třída je podtřídou ze seznamu tříd oddělených čárkami (a	//A je podtřída B
			
			public class A : B
			{
			}
				//A je podtřída B
				
				public class A extends B
				{
				}

		rozhraní v C#).		

:	implements	Operátor nebo modifikátor v definici třídy, který znamená, že implementuje seznam rozhraní oddělených čárkami (a tříd v C#).	//A implementuje I
			public class A : I
			{
			}
				//A implementuje I
				public class A implements I
				{
				}

Zdroj: (Microsoft)

3.4 Vývojové prostředí Microsoft Visual Studio

Microsoft Visual Studio je sada nástrojů pro vytváření softwaru. Nabízí nástroje od fáze plánování návrhu prostřednictvím uživatelského rozhraní, kódování, testování, ladění až po analýzy kódu.

Pomocí sady Visual Studio je možné vytvořit různé druhy aplikací, od jednoduchých obchodních aplikací a her pro mobilní zařízení až po komplexní systémy ve velkých podnicích a datových centrech. Lze zde vytvořit:

- aplikace a hry, které lze spustit v systému Windows, ale také Android a iOS
- webové stránky a webové služby založené na technologii ASP.NET, JQuery, AngularJS, apod.
- aplikace pro platformy a zařízení, jako je Azure, Office, SharePoint, Kinect, apod.
- hry a graficky náročné aplikace pro různé zařízení Windows, zahrnující Xbox, použití DirectX

Ve výchozím nastavení Visual Studio poskytuje podporu pro C#, C a C++, JavaScript, F# a Visual Basic.
(Microsoft)

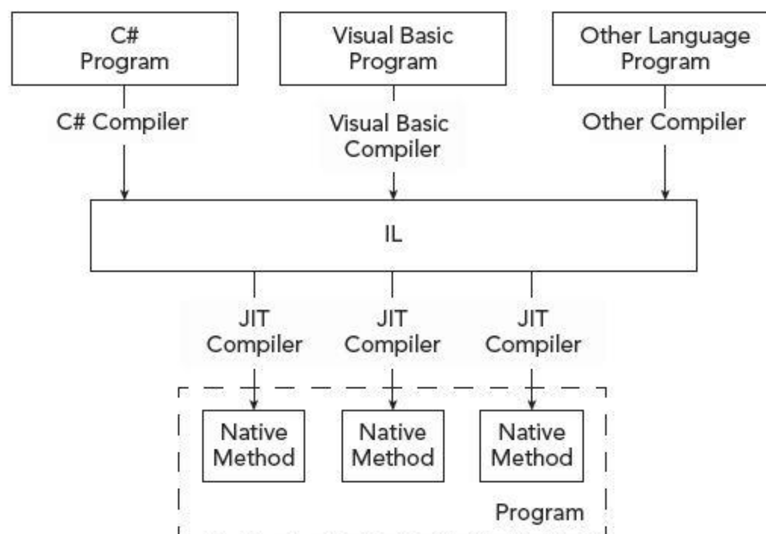
3.4.1 Kompilátory

Při vývoji byl meziproduktový jazyk nazván Microsoft Intermediate Language (MSIL), poté co byl spuštěn .NET se název změnil na IL. Mezinárodní standardizační organizace Ecma vytvořila standard Common Language Infrastructure (CLI), který definuje Common Intermediate Language (CIL).

IL se na první pohled svou strukturou blíží jazyku symbolických instrukcí. Jazyk symbolických instrukcí je pro člověka čitelná verze strojového kódu, která může běžet na konkrétním typu počítače. Pro sdílení programů na více počítačích je jednodušší použít IL, která poskytuje vrstvu mezi C# a strojovým kódem. Je to jako virtuální jazyk symbolických instrukcí, který musí být ještě zkompilován

do spustitelného strojového kódu. V .NET vykonává tuto kompilaci Common Language Runtime (CLR).

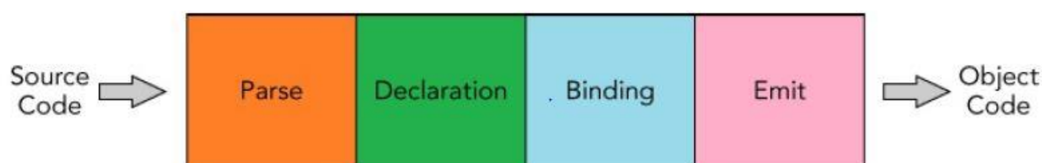
CLR používá just-in-time kompilátor (JIT kompilátor). Když program vyvolá metodu, JIT kompilátor překládá její IL kód do strojového kódu, vytvoří si bod a přejde na něj a poté spustí strojový kód. Je-li metoda volána znova, tak bod již odkazuje do strojového kódu, takže metoda nemusí být znova sestavena. Kompilátor JIT tak šetří nějaký čas při opakované kompilaci.



Obr. 3 Překlad zdrojového kódu kompilátorem
Zdroj: (Stephens, 2014)

CLR kromě JIT kompilace poskytuje také některé nízko-úrovňové služby, jako je správa paměti, nebo zpracování výjimek. (Stephens, 2014)

Na Obr.4 lze vidět fáze kompilátoru, od zdrojového kódu k objektovému kódu.

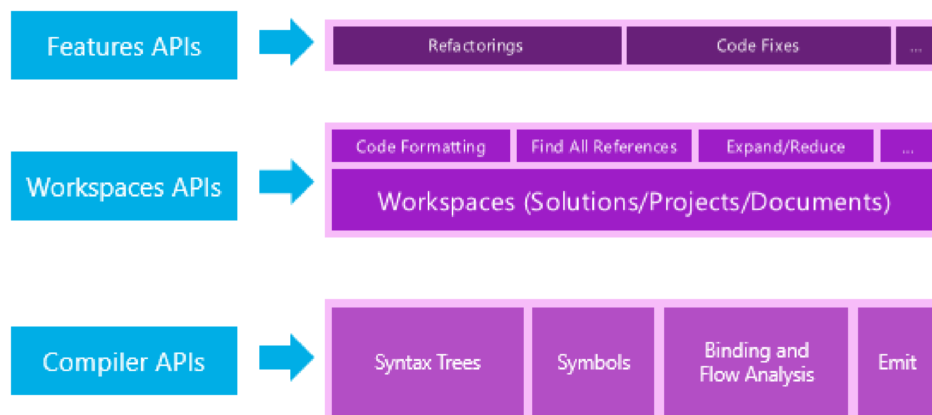


Obr. 4 Fáze kompilátoru
Zdroj: (Johnson, 2015)

Nejdříve se rozdělí zdrojový kód na atomické elementy (klíčové slova, symboly, nebo jiné typy identifikátorů), poté rozdělí tokeny pomocí syntaktické gramatiky jazyka. Na konci první fáze je syntaktický strom, který reprezentuje lineární tokeny ve stromové struktuře. V další fázi (Declaration) se analyzuje syntaktický strom a vytvoří se tabulka symbolů. Tabulku symbolů tvoří seznam identifikátorů a podrobnosti o typu, rozsahu a veškerých souvisejících metadatech. Výstupem této fáze je hierarchická tabulka symbolů. Ve fázi Binding jsou symboly z předchozí fáze

odladěny do symbolů identifikovaných v syntaktickém stromě. To zahrnuje nejen proměnné, které jsou součástí zdrojového kódu, ale také jakékoli reference funkcí. V poslední fázi (Emit) se všechny informace získané z předchozích fází spojí a vytvoří IL kód potřebný k vytvoření finální aplikace.

Nový kompilátor Roslyn odhalil řadu API, které podporují rozšíření a manipulaci s výsledky z každé fáze kompilátoru. A pro každou z API není závislost na vývojovém prostředí Visual Studio.



Obr. 5 API vrstvy
Zdroj: (Microsoft, 2014)

- Compiler API – odhaluje objektový model, který odpovídá objektům a výstupům z každé fáze kompilátoru. Obsahuje také reprezentaci kompilátoru vztahující se k aktuálnímu procesu, obsahující možnosti, soubory zdrojového kódu a všechny odkazované knihovny.

  Ve skutečnosti existují dva kompilátory API, které jsou v Roslyn k dispozici - jeden pro jazyk C# a jeden pro jazyk Visual Basic. Jsou velmi podobně strukturované, ale jsou postaveny tak, aby co nejlépe odpovídaly silným stránkám jednotlivých jazyků.
- Diagnostic API- kompilátor vytváří spoustu diagnostických informací, které zahrnují vše od syntaxe a sémantiky po chyby a varování. Prostřednictvím tohoto API je možné uživatelem definovat vlastní analyzátory, které budou zapojeny do procesu kompilace. Pomocí interakce s Diagnostic API, svůj vlastní set pravidel je možné bez problému integrovat s nástroji, které vývojář již používá.
- Workspaces API – poskytuje výchozí bod pro provádění analýzy kódu a refaktoring v rámci celého projektu. Současně se Workspace API používá k implementaci funkcí Microsoft Visual Studia, jako je nalezení všech referencí a formátování. Ale i zde, tak jako i u ostatních API v Roslyn, není žádná závislost na Visual Studiu. Vaše nástroje pracují bez ohledu na to, zda jsou k dispozici prostřednictvím Visual Studia. (Johnson, 2015)

Microsoft vytvořil nový kompilátor Roslyn, který je napsán v jazyce C# a Visual Basic. Vznikl z několika důvodů, především proto, že starý kompilátor napsaný v jazyce C++ byl na hranici svých možností a přidání jakékoli nové funkce představovalo problém a bylo pro vývojářský tým již dost namáhavé jej upravovat. Kompilátor Roslyn si může každý vývojář jednoduše stáhnout a také jej podle libosti rozšiřovat. (Holec, 2015)

4 Nástroje pro automatickou kontrolu kódu

Nástroje pro kontrolu kódu vznikly jako doplněk pro vývojové prostředí. Tyto nástroje by měly vývojáři usnadnit psaní zdrojového kódu, vyhledávat chyby v kódu podle určitých pravidel.

Základním doplňkem pro kontrolu kódu implementovaným ve vývojovém prostředí Microsoft Visual Studio je IntelliSense. Tento doplněk se snaží především předcházet psaní chyb prostřednictvím automatického doplňování kódu při jeho psaní. Pro mnohé vývojáře byl však nedostačující a začali se tak vyvíjet komplexnější nástroje. Jedním z nich je ReSharper, který se vyvíjí již přes 10 let. ReSharper je rozdělen na více verzí a také více licencí. Před pár lety se začal vyvíjet také nástroj nazvaný Roslyn, který by měl především sloužit pro Microsoft Visual Studio od verze 2015 již jako implicitní kompilátor místo starého těžkopádného a již nedostačujícího kompilátoru, který byl napsán v programovacím jazyce C++. Výhodou kompilátoru Roslyn je nejen to, že je zdarma poskytován již v základním balíčku Microsoft Visual Studia, ale především to, že nabízí vývojářům možnost vytvářet vlastní pravidla pro kontrolu kódu. Microsoft tak nabízí možnost nahlédnout vývojářům do struktury kompilátoru a upravit si ho na míru dle požadovaných pravidel, které se ve společnosti mají dodržovat.

4.1 ReSharper

ReSharper byl vyvinut společností JetBrains (http://www.jetbrains.com), první verze tohoto nástroje byla vypuštěna do světa již v roce 2004, jako druhý produkt této společnosti. ReSharper poskytuje více verzí, podle toho, na který programovací jazyk je zaměřeno programování. Nabízí komerční nebo osobní licenci, lze však získat i zdarma, pokud jste učitel, instruktor nebo pokud se pracuje na nekomerčním open source projektu. Zdarma lze využít plnou edici Resharperu, jako 30-ti denní zkušební verzi.

ReSharper je komplexní nástroj pro zlepšení produktivity vytvořený pro vývojové prostředí Microsoft Visual Studio. ReSharper dělá refaktoring kódu jednodušší, šetří čas tím, že podporuje navigaci v rámci projektu a pomáhá s generováním kódu. Napomáhá psát chytřejší kód díky funkcím, jako je generování kódu a šablony kódu. Navigace a vyhledávání pomáhá najít potřebné věci rychleji. ReSharper také nabízí analýzu kódu a jeho vyčištění, vyhledá a zvýrazní chyby a potenciální problémy a špatné postupy v kódu.

Nezáleží na tom, jakou aplikací vývojář implementuje, protože ReShaprer poskytuje rozsáhlou podporu pro mnoho jazyků a technologií jako např.: C#, VB.NET, ASP.NET, ASP.NET MVC, HTML, JavaScript, CSS, XML,XAML.

Konfigurace ReSharper ve vývojovém prostředí Visual Studio je rozdělena do čtyř oblastí:

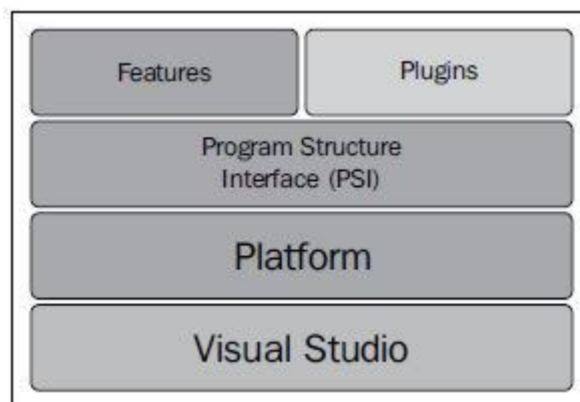
- *Environment* – umožňuje změnit obecná nastavení ReSharper, jako je například uživatelské rozhraní a integrace Visual Studia

- *Kontrola kódu* – umožňuje změnit nastavení týkající se analýzy kódu
- *Editace kódu* – umožňuje změnit formátování, pojmenování a pravidla pro čištění kódu
- *Nástroje* – umožňuje změnit nastavení ReSharper nástroje, například frameworky pro testování, vzory použité pro to-do položky a navigaci externích zdrojů.

4.1.1 ReSharper API

ReSharper poskytuje otevřené API, které je to stejné API, co bylo využito k vytvoření všech funkcí ReSharperu. Z architektonického hlediska je ReSharper API rozdělen do třech vrstev:

- *Platforma* – je první základní vrstva, která umožňuje pracovat přímo s Visual Studio API.
- *Rozhraní struktury programu (PSI)* – je nejpoužívanější vrstva, protože slouží jako analyzátor pro jazyky podporované ReSharperem. Sestaví abstraktní syntaktický strom (AST), kterým můžete přistupovat a procházet se skrz svůj plugin.
- *Funkce a Pluginy* – tvoří nejvyšší úroveň, která představuje v ReSharperu navigaci nebo živé šablony. Jak lze vidět, vestavěné prvky jsou na stejné úrovni jako pluginy, což znamená, že my jako vývojáři pluginu máme přístup ke stejným metodám jako vývojáři v JetBrains, kteří tvoří nové funkce.(Gąsior, 2014)



Obr. 6 Vrstvy ReSharperAPI
Zdroj: (Gąsior, 2014)

4.1.2 Základní přehled funkcí nástroje ReSharper

ReSharper je robustní nástroj pro kontrolu kódu, který nabízí mnoho funkcí. Pro spoustu z těchto funkcí jsou k dispozici klávesové zkratky pro snadnější použití.

- *Analyzátor kódu* –ReSharper zvýrazňuje zjištěné chyby a problémy přímo v editoru Visual Studia. ReSharper také zobrazuje kromě chyb varování, například o redundantních částích kódu nebo nesprávném formátu řetězce.

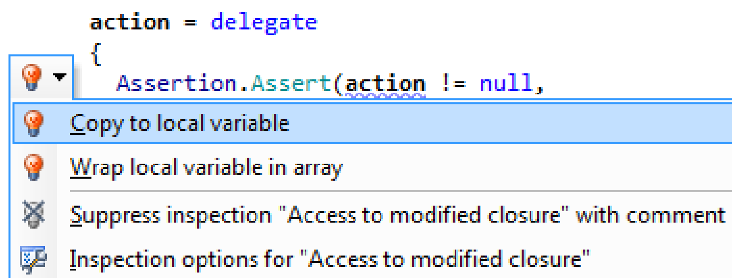
```
if(FRead == null)
    throw new ArgumentNullException("FRead");
new TextReader(stream).WithDispose(FRead);
```

Cannot create an instance of the abstract class 'System.IO.TextReader'

Obr. 7 Zobrazení chyby v ReSharper
Zdroj: (JetBrains s.r.o.)

Součástí analýzy kódu jsou i návrhy kódu, které mohou poskytnout pohled do struktury kódu. Tyto návrhy jsou zvýrazněny zeleným vlnitým podtržením.

ReSharper poskytuje rychlé opravy pro většinu problémů v kódu, pomáhá tím řešit problémy okamžitě při psaní kódu. Rychlé opravy jsou reprezentovány červenou žárovkou v případě závady, nebo žlutou žárovkou pro varování, návrhy a rady. Žárovka se objeví na levé straně příslušného řádku.



Obr. 8 Návrh rychlé opravy v ReSharperu
Zdroj: (JetBrains s.r.o.)

- *Asistence při kódování* – ReSharper poskytuje řadu funkcí pro získání dalších informací o kódu. Dokáže vybrat a zvýraznit bloky kódu a aplikovat lokální transformace kódu.

  ReSharper rozšiřuje výchozí podporu zvýraznění ve Visual Studiu vlastním zvýrazněním kódu položek v různých jazycích. Lze také konfigurovat vlastní barvu pro každou položku.

```
XMLRPCRequest input = new XMLRPCRequest(context);
XMLRPCResponse output = new XMLRPCResponse(input.MethodName);
switch (input.MethodName)
    {
        case "metaWeblog.newPost":
            output.PostID = NewPost(input.BlogID, input.UserName,
                                    input.Publish);
            break;
        case "metaWeblog.editPost":
            output.Completed = EditPost(input.PostID, input.UserNa
                                        input.Publish);
```

Obr. 9 Zvýraznění kódu v ReSharperu
Zdroj: (JetBrains s.r.o.)

K dispozici je i funkce pro doplňování kódu, tato funkce vkládá potřebné syntaktické prvky, jako jsou závorky nebo středník.

ReSharper poskytuje bohatou sadu nástrojů pro práci s regulárními výrazy. Můžeme tak rychle analyzovat stávající výrazy, najít a opravit chyby. Při tvorbě nových regulárních výrazů pomáhá ReSharper s automatickým dokončením a potvrzením.

Při psaní párových znaků, jako je (, {, [, v kódu je druhý párový znak vložen automaticky. Tato funkce lze také jednoduše vypnout.

- *Kompletace kódu* – ReSharper rozšiřuje nativní dokončování kódu ve Visual Studiu (IntelliSense) pokročilejšími funkcemi. Například dokáže zúžit seznam návrhu při psaní kódu. Automaticky importuje vybrané druhy a rozšíření metod.

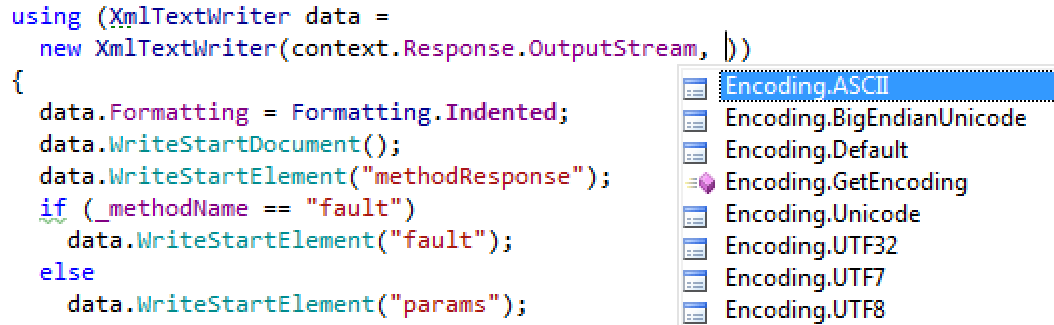
ReSharper má k dispozici podpůrné CamelHumps, které napomáhá dokončit jakoukoli položku zadáním jen jeho velkých písmen.

```
data.Formatting = Formatting.Indented;
data.ws
    WriteStartAttribute
    WriteStartDocument
    WriteStartElement
    WriteState
    WriteString
    WriteSurrogateCharEntity
```

Obr. 10 CamelHumps v ReSharperu
Zdroj: (JetBrains s.r.o.)

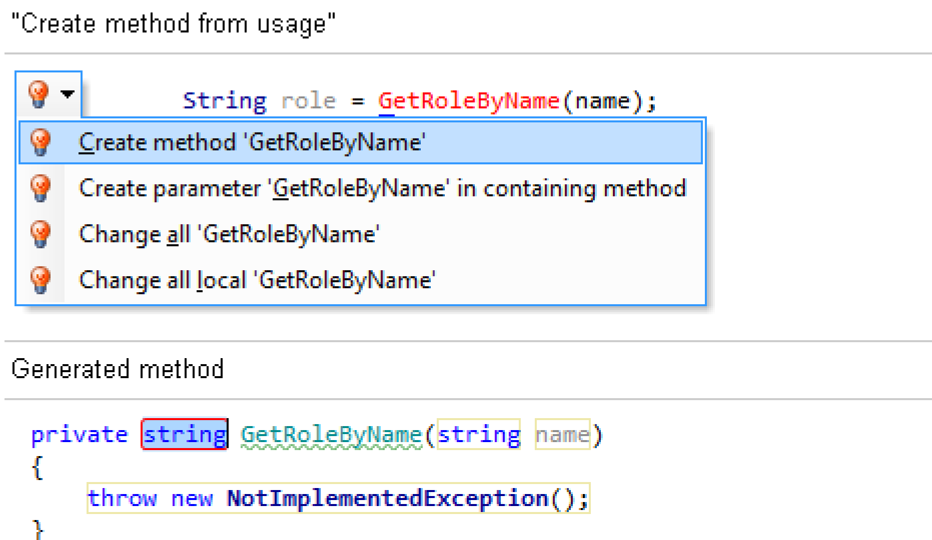
Doplňování kódu rozpoznává proměnné, metody, klíčová slova apod. ReSharper automaticky navrhne předpony při deklarování identifikátorů, podle vlastního pojmenovaného stylu. Je-li seznam pro dokončování kódu prázdný nebo neobsahuje očekávané položky, lze stisknout klávesovou zkratku pro dokončení podruhé a ReSharper rozšíří seznam o protected, private a interní členy. Inteligentní dokončování kódu vyvolané pomocí klávesové zkrat-

ky Ctrl+Alt+Mezerník filtruje seznam metod nebo proměnných tak, aby odpovídali očekávanému typu výrazu. Inteligentní dokončování kódu může rovněž navrhnout vytvoření anonymní metody, lambda výrazu a běžných metod



Obr. 11 Inteligentní dokončování kódu v ReSharperu
Zdroj: (JetBrains s.r.o.)

- *Generování kódu* – S ReSharperem je možné použít metody, proměnné, vlastnosti, události a třídy předtím, než jsou deklarovány. ReSharper navrhne rychlou opravu pro C# nebo několik kontextových akcí pro VB.NET. ReSharper nejen vytvoří metodu, ale také její návratovou hodnotu a typy jejích parametrů.



Obr. 12 Automatické generování kódu v ReSharperu
Zdroj: (JetBrains s.r.o.)

ReSharper nabízí generování konstruktorů a jejich proměnné a základní vlastnosti, které budou inicializovány v konstruktoru. V jakékoli třídě lze rychle vygenerovat metodu ToString() k přepsání. Rychle lze vygenerovat také metody Equals() a GetHashCode() pro aktuální typ. ReSharper nevytváří pouze

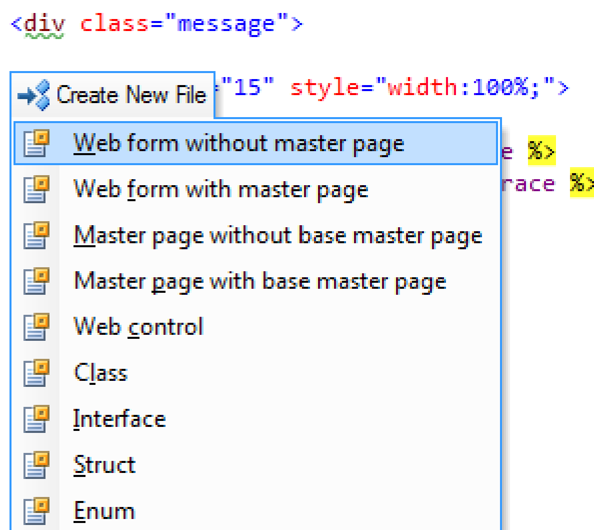
metody, ale také generuje potřebný kód pro kontrolu rovnosti nebo pro výpočet hash kódu.

- *Šablony kódu* – ReSharper nabízí několik druhů šablon kódu, které pomáhají napsat konstrukci společného kódu rychleji. Lze si vytvořit také vlastní šablony, které jsou specifické pro vaše návyky kódu a splňují dané požadavky.

  ReSharper nabízí více než 60 předdefinovaných Live Templates pro většinu podporovaných jazyků a technologií. Ve většině případů nemusí vývojář psát cokoli sám, stačí si jen vybrat ze seznamu doporučených hodnot.

  Další šablony jsou prostorové, které slouží k rychlému psaní výrazů. Může se jednat o jeden příkaz nebo libovolný blok kódu s if … else, try … catch nebo jiné struktury kódu. ReSharper inteligentně přeformátuje kód a upraví jej a po použití šablony je okamžitě možné pokračovat v psaní kódu.

  ReSharper umožňuje přidat nové soubory s předdefinovanými částmi kódu. Vybrat si šablonu lze za pomocí klávesové zkratky i bez opuštění textového editoru.



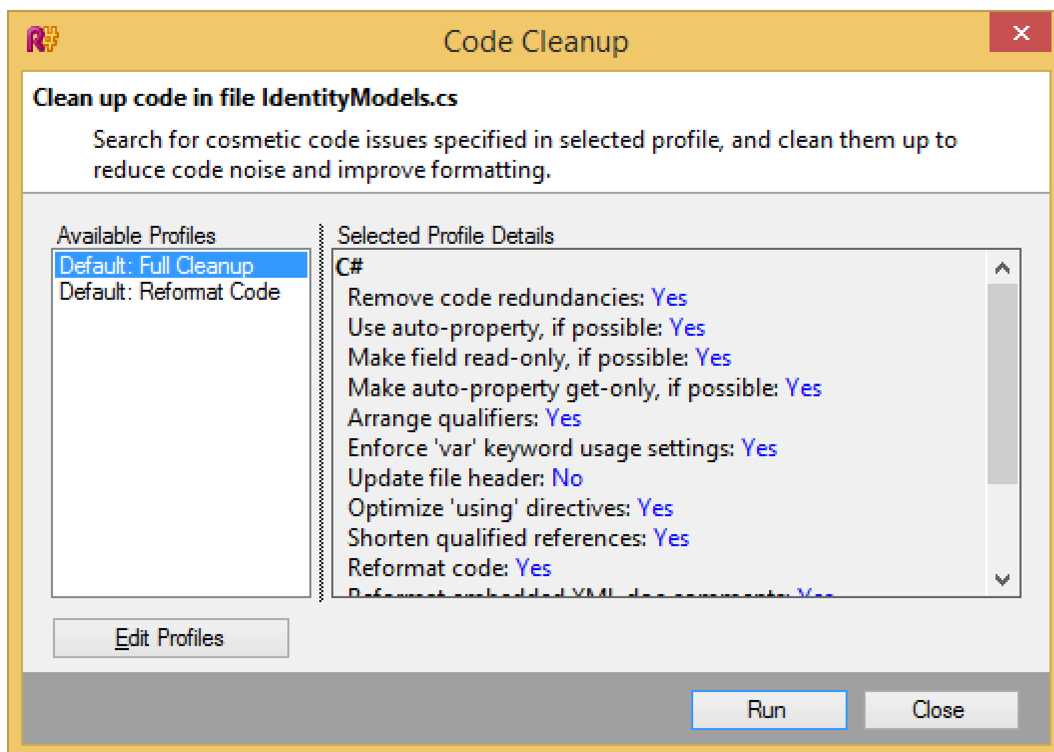
Obr. 13 Souborové šablony v ReSharperu
Zdroj: (JetBrains s.r.o.)

- *Styl kódu* – ReSharper pomáhá udržet konzistentní styl kódu v celé struktuře i v drobných detailech. Výchozí nastavení stylu jsou založena na obecně přijatých konceptech a osvědčených postupech. Pokud se však osobní preference nebo podnikové pravidla liší od výchozích hodnot ReSharper, lze si je nastavit na velmi podobné úrovni a nastavené hodnoty lze snadno sdílet v celém týmu. Porušení stylu kódu ve stávajícím bloku je detekován kontrolou kódu a může být opraven pomocí rychlých oprav nebo vyčištění kódu.

  Vyčištění kódu pomáhá rychle odstranit porušování stylu kódu v jednom nebo i ve více souborech, nebo v celých projektech pomocí klávesové zkratky. Lze si vybrat úplné vyčištění, což platí pro všechna nastavení s výjimkou po-

jmenovaného stylu nebo přeformátování kódu, které se vztahuje pouze na vaše formátovací pravidla.

ReSharper umožňuje definovat vlastní nastavení stylu pro různé jazyky a symboly včetně druhů, jmenných prostor, rozhraní, parametrů metod apod. ReSharper poskytuje různé možnosti formátování pro všechny podporované jazyky. Přeformátování ovlivňuje rozložení závorek, prázdných řádků, zalamování řádků, odsazení a mnoho dalších možností, které lze nastavit.



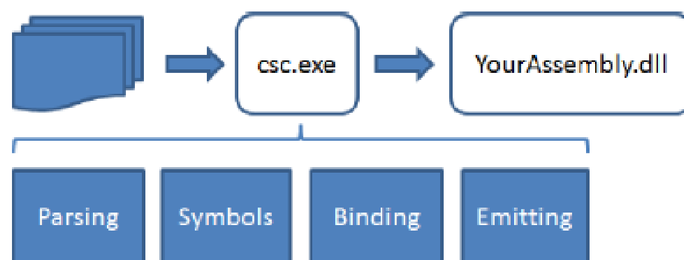
Obr. 14 Vyčištění kódu v ReSharperu
Zdroj: (JetBrains s.r.o.)

ReSharper nabízí mnoho funkcí, které mají napomoci analyzovat kvalitu kódu a eliminovat chyby v psaném kódu. Na chyby v kódu nejen upozorní uživatele, ale nabídne i opravy. Pomáhá uživateli doplňovat a generovat kód pro rychlejší implementaci. Usnadnit práci mohou také šablony dostupné v ReSharperu, nebo šablony vytvořené uživatelem podle jeho zadaných pravidel. (JetBrains s.r.o.)

4.2 Roslyn

Pro většinu vývojářů je vnitřní práce kompilátoru černou skříňkou. Řádky napsaného kódu jsou vstupem a výstupem je vytvořená aplikace. Tato úroveň porozumění je více než dostačující pro běžného vývojáře. Nicméně kompilátor, jako součást překladu, buduje také velké množství informací o struktuře kódu.

Základním posláním kompilátoru Roslyn je zpřístupnění znalostí černých skříněk vývojářům pro jejich aplikace. Takže místo tradičního neprůhledného modelu kód dovnitř, aplikace ven, Roslyn má sadu rozhraní API, které lze využít pro úkoly týkající se kódu potřebných pro vytvořené nástroje a aplikace. S kompilátorem Roslyn se naskytla možnost využít znalosti kompilátoru i třetím stranám. Poskytuje možnost analýzy kódu nebo také nástroje pro refaktoring. (Johnson, 2015)



Obr. 15 Zobrazení vnitřní části kompilátoru
Zdroj: (HAZZARD & BOCK, 2013)

4.2.1 Architektura Roslyn

Architektura Roslyn rozděluje kompilaci do třech fází:

1. Rozdělení kódu do syntaktických stromů (syntaktická vrstva)
2. Spojení identifikátorů se symboly (sémantická vrstva)
3. Vygenerování IL

V první fázi je přečten C# kód a výstupem je syntaktický strom. Syntaktický strom je DOM (Document Object Model), který popisuje zdrojový kód ve stromové struktuře.

Druhá fáze je místo, kde probíhají statické vazby. Jsou přečteny odkazy knihoven a kompilátor zjistí, že například „Console“ se vztahuje k „System.Console“ v mscorlib.dll.

Třetí fáze vytváří výstupní knihovnu. Tato funkcionalita se nepoužívá, pokud se Roslyn aplikuje pro analýzu kódu nebo refaktoring.

4.2.2 Syntaktické stromy

Syntaktický strom je DOM pro zdrojový kód. Roslyn syntaktický strom má následující unikátní funkce:

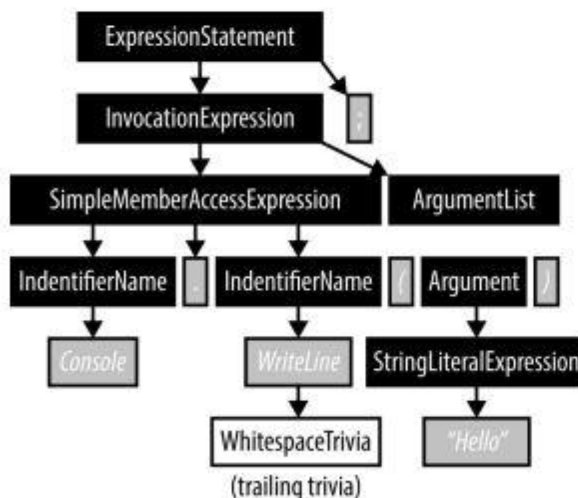
- Může reprezentovat kompletní C# jazyk, ne jen výrazy.
- Může obsahovat komentáře, mezery a další triviální prvky a může zpět s přesností vrátit originální zdrojový kód.
- Přichází s metodou ParseText, která rozdělí zdrojový kód do syntaktického stromu

Syntaktické stromy jsou zde neměnitelné, takže žádný z jeho prvků již jednou vytvořený není možné měnit. To znamená, že vývojové prostředí jako je Visual Studio musí vytvořit nový syntaktický strom při každém zmáčknutí tlačítka v editoru, za účelem aktualizace zvýrazňování syntaxe a automatického dokončování kódu. Nový syntaktický strom je schopen použít většinu prvků z minula. Více vláknový kód může bezpečně přistupovat ke všem částem syntaktického stromu bez zámků.

Syntaktický strom zahrnuje tři hlavní elementy:

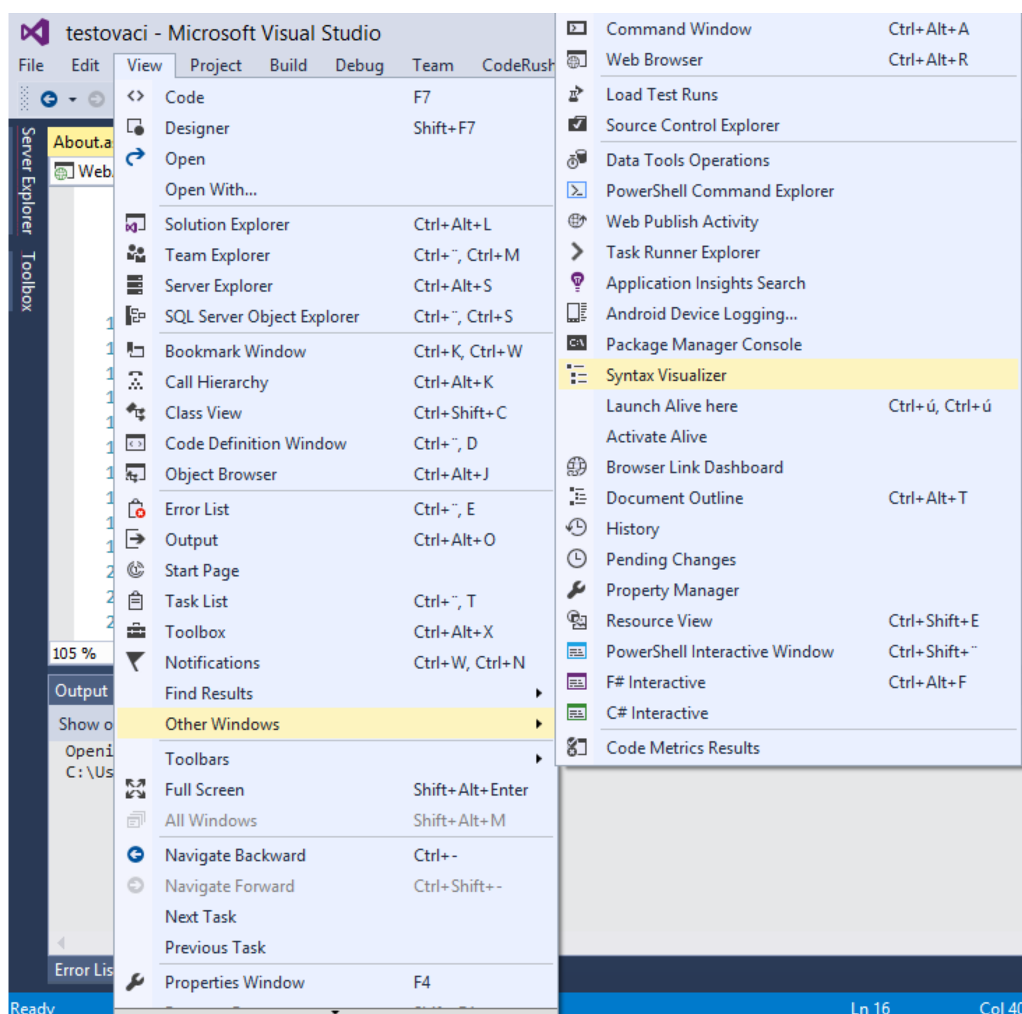
- Nodes (Abstract SyntaxNode class) – Představuje konstrukce C#, jako jsou výrazy, příkazy a deklarace metod. Uzly mají vždy alespoň jednoho potomka, nikdy nemohou být listem stromu. Uzly mohou mít jako potomky uzly nebo tokeny.
- Tokens (SyntaxToken struct) – Představuje identifikátory, klíčová slova, operátory a interpunkci tvořící zdrojový kód. Jediný druh potomka, který mohou tokeny mít, jsou nepovinné počáteční a koncové trivia. Rodičem tokenu je vždy uzel.
- Trivia (SyntaxTrivia struct) – Trivia jsou mezery, komentáře, a kód, který je neaktivní kvůli podmíněné kompilaci. Trivia je vždy spojena s tokenem, který je bezprostředně vlevo nebo vpravo k němu a je respektive přístupný přes tokenovo TrailingTrivia a LeadingTrivia vlastnosti.

Na Obr. 16 je znázorněn syntaktický strom, kde jsou Nodes zobrazeny v černém poli, Tokens v šedém poli a Trivia v bílém poli. (Albahari & Albahari, 2015)



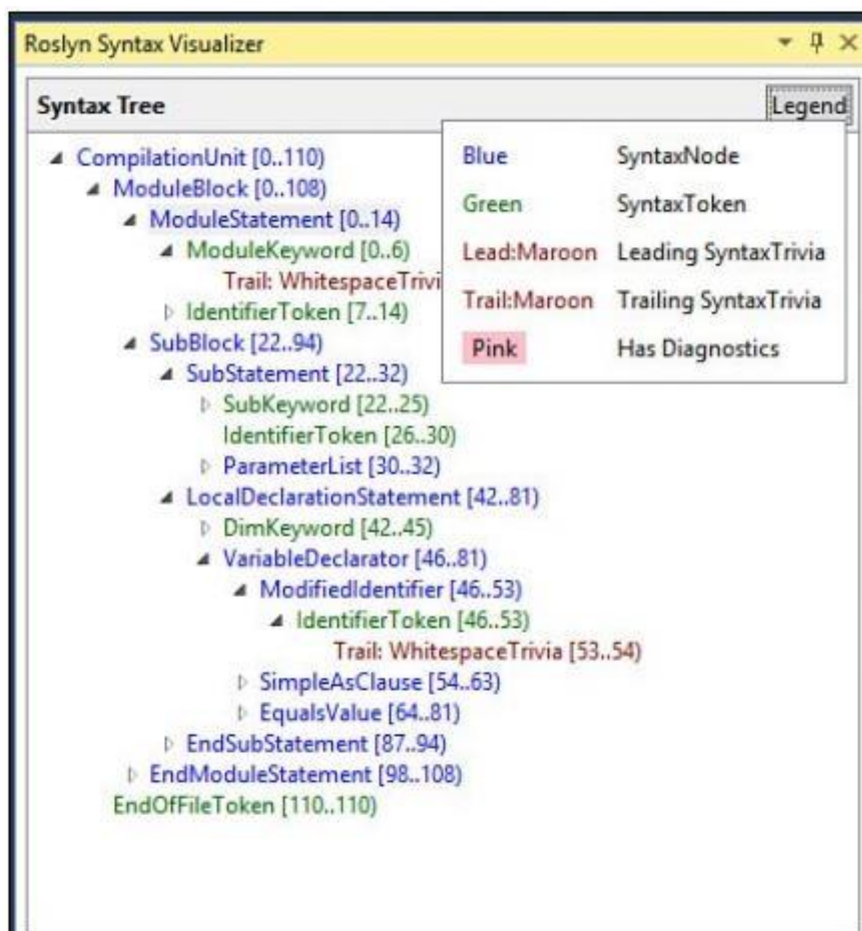
Obr. 16 Syntaktický strom
Zdroj: (Albahari & Albahari, 2015)

Roslyn nabízí také nástroj Syntax Visualizer, který je dostupný přes nabídku View, Other Windows.



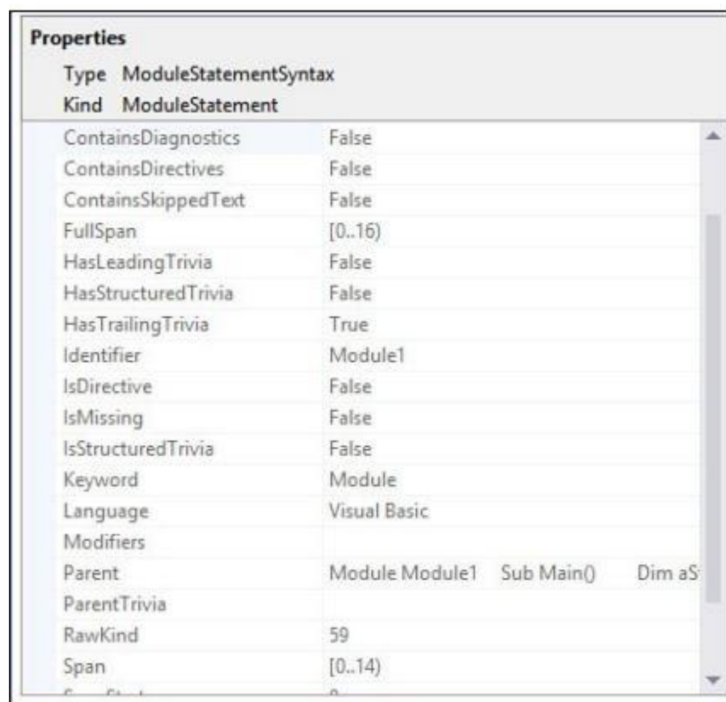
Obr. 17 Zapnutí nástroje Syntax Visualizer

Nástrojové okno Syntax Visualizer je rozděleno na dvě části. V horní části, nazvané Syntax Tree, je zobrazen hierarchický pohled na syntaktický strom pro aktuální soubor, tento pohled je aktualizován při kliknutí kdekoli v kódu. Každý prvek stromu má jinou barvu podle svého významu. Modrá barva představuje Syntax Node, zelená je Syntax Token, kaštanově hnědá barva znázorňuje Syntax Trivia a růžové zvýraznění představuje kód, který má diagnostiku, například chyby nebo varování. Význam barev v Syntax Tree lze kdykoli zobrazit tlačítkem Legend.



Obr. 18 Část nástroje Syntax Visualizer – Syntax tree
Zdroj: (Sole, 2015)

Spodní část okna Syntax Visualizer má název Properties a zobrazuje vlastnosti pro vybranou položku syntaktického stromu. Při vytváření analyzátorů a refaktorování kódu lze získat pomocí této části okna podrobnější informace a pochopit tak typ a druh každého prvku ve stromové struktuře, jako je Syntax Node, Syntax Token a Syntax Trivia. (Sole, 2015)



Obr. 19 Část nástroje Syntax Visualizer – Properties
Zdroj: (Sole, 2015)

SyntaxNode je abstraktní a má C# specifickou podtřídu pro každý druh syntaktického elementu, jako je VariableDeclarationSyntax nebo TryStatementSyntax.

SyntaxToken a SyntaxTrivia jsou struktury a tak jeden typ představuje každý druh tokenu nebo trivia. Pro rozlišení různých druhů tokenů nebo trivia je nutné použít vlastnost RawKind nebo Kind metodu.

Uzly, tokeny a trivia mají mnoho vlastností a metod:

- SyntaxTree vlastnost – vrací syntaktický strom, který zahrnuje objekty.
- Span vlastnost – vrací pozici objektu ve zdrojovém kódu.
- Kind rozšiřující metoda – vrací výčet SyntaxKind, který klasifikuje uzly, tokeny nebo trivia do jednoho z několika stovek hodnot (například IntKeyword, CommaToken nebo WhitespaceTrivia).
- ToString metoda – vrací text (zdrojový kód) pro uzly, tokeny nebo trivia. Pro tokeny je ekvivalentem vlastnost Text.
- GetDiagnostics metoda – vrací chyby nebo varování generované během analýzy.
- IsEquivalentTo metoda – vrací true pokud je objekt identický s jiným uzlem, tokenem nebo trivia instancí.

Další cestou k získání syntaktického stromu je zavolání CSharpSyntaxTree.Create, který prochází v grafu uzly a tokeny. Po rozdělení syntaktického stromu je možné

získat chyby a varování pomocí GetDiagnostics (tuhle metodu lze také zavolat na konkrétní uzel nebo token).

Uzly a tokeny mají vlastnost Parent typu SyntaxNode. Pro SyntaxTrivia je předkem jeho token, přístupný přes vlastnost Token. Všechny uzly, tokeny a trivia mají vlastnost Span typu TextSpan k indikaci počátečního a posledního posunu ve zdrojovém kódu. Uzly a tokeny mají také vlastnost FullSpan, která zahrnuje úvodní a koncové trivia, zatímco Span je neobsahuje.

Trivia patří k tokenu, který je k ní přilehlý. Podle konvence analyzátor klade mezery a komentáře, které následují token až ke konci řádku do tokenových koncových trivia. Cokoli po tom následující je považováno za triviu pro další token.

Direktiva #region je sémanticky prázdná, její role pro analyzátor je pouze zkontrolovat, zda direktiva #region je shodná s direktivou #endregion. Direktivy #error a #warning jsou také zpracovány analyzátorem, který generuje chyby a varování, které lze zavolat na strom nebo uzel pomocí GetDiagnostic. Jsou dva druhy trivia:

- Unstructured trivia – komentáře a mezery a kód který je neaktivní kvůli podmíněné kompilaci.
- Structured trivia – direktivy preprocesoru a XML dokumentace.

Unstructed trivia je považována čistě jako text, zatímco Structured trivia má také jeho obsah rozdělen do miniaturního syntaktického stromu. Vlastnost HasStructure u SyntaxTrivia indikuje, zda je strukturovaná trivia předkem a metoda GetStructure vrací kořenový uzel miniaturního stromu.

Je možné upravovat uzly, tokeny a trivia přes sadu metod s prefixem Add, Insert, Remove, Replace, With, Without, většina z nich jsou rozšiřující metody. Vzhledem k tomu, že syntaktické stromy jsou neměnné, všechny tyto metody vrací nový objekt s požadovanými úpravami, takže původní objekty jsou nedotčené.

Statické metody v SyntaxFactory vytváří uzly, tokeny a trivia, které je možné použít k transformaci existujících syntaktických stromů nebo k vytvoření nových stromů zcela od začátku. Nejtěžší částí je zjištění, co přesně druh uzlů a tokenů vytváří. Řešením je nejdříve analyzovat vzorek kódu zkoumající výsledek v syntaktickém zobrazení.

4.2.3 Kompilace a sémantický model

Kompilace zahrnuje syntaktické stromy, reference a možnosti kompilace. To slouží ke dvěma účelům:

- Umožňuje kompilace do knihovny nebo spustitelného souboru (emit fáze)
- Zpřístupňuje sémantický model, který poskytuje informace o symbolu (získané z binding)

Sémantický model je nezbytný při zavádění funkcí, jako je přejmenování symbolu nebo nabízí výpisy pro doplňování kódu v editoru. Prvním krokem pro dotazování v sémantickém modelu nebo provedením úplné kompilace je vytvoření CSharp-

Compilation. Ve výchozím nastavení se předpokládá vytvoření knihovny, lze však i určit jiný druh výstupu.

Kompilace může generovat chyby a varování a to i v případě, že syntaktické stromy jsou bez chyb. Například to mohou být zapomenuté importy jmenných prostor, překlep, který se odkazuje na typ nebo název členu. Získat varování a chyby lze zavoláním GetDiagnostics na objekt kompilace, zahrnuje také všechny syntaktické chyby.

Neexistují žádné zastřešující DOM spojené se sémantickým modelem. Místo toho máme sadu metod pro volání k získání sémantické informace o konkrétní pozici nebo uzlu v syntaktickém stromu. Sémantický model transformuje identifikátory do symbolů, které mají informace o typu. Všechny symboly implementuje rozhraní ISymbol, jsou zde obsaženy i konkrétnější rozhraní pro každý druh symbolu.

Třídy pro syntaxi uzlů se liší pro C# a VB, ačkoli sdílejí základní abstraktní typ SyntaxNode, mají jazyky jinou lexikální strukturu. ISymbol a jeho odvozené rozhraní jsou sdíleny mezi C# a VB, avšak jejich vnitřní konkrétní implementace jsou specifické pro každý jazyk a výstup z jejich metod a vlastností odráží rozdílné jazykové specifikace.

Pro získání chybové nebo výstražné informace pro část kódu, je možné volat na sémantický model metodu GetDiagnostics, specifikující TextSpan. Volání GetDiagnostics bez argumentu je ekvivalentní k volání stejné metody na objekt CSharpCompilation.

ISymbol má deklarovanou vlastnost DeclaredAccessibility, která indikuje, když je symbol public, protected, internal, apod. Tohle je však nepostačující k určení, zda daný symbol je k dispozici v určité poloze ve zdrojovém kódu. Například lokální proměnné mají lexikálně omezený rozsah. Pokud je volána metoda GetSymbolInfo na typ nebo člena deklarace, nedostaneme zpět žádný symbol. Pro získání symbolu se místo toho musí zavolat metoda GetDeclaredSymbol, pokud selže, tak protože nemůže najít platný deklarovaný uzel. (Albahari & Albahari, 2015)

5 Metodika práce

Ke splnění veškerých cílů této práce bylo potřeba si nejdříve navrhnout celý nástroj a jeho architekturu. K tomuto napomohlo rozčlenění poskytnutých pravidel na jednotlivé kategorie. Společnost Kentico disponuje především dvěma druhy pravidel, u kterých usiluje o jejich dodržování. Podle nichž bylo vytvořeno i hlavní rozčlenění nástroje na syntaktická pravidla a pravidla pro statickou analýzu kódu. Komplexní návrh i rozčlenění pravidel bylo vytvořeno na základě konzultací a dostupných informací ze společnosti. Z kategorií byla vyloučena pravidla pro programovací jazyky, které nejsou prozatím podporovány samotným kompilátorem Roslyn. Po schválení vytvořených kategorií, bylo přistoupeno k návrhu implementace nástroje.

Pro kontrolu syntaktických pravidel je ve společnosti využíván nástroj ReSharper, na který má společnost licence. Tento nástroj je hojně využíván ve společnostech, které chtěj provádět kontrolu syntaxe pravidel. ReSharper je sice nyní více rozšířen, ale platforma Roslyn je OpenSource a poskytuje tak vývojáři nahlédnout do kódu a upravovat si jej. Vývojář si může pomocí platformy Roslyn vytvořit pravidla na míru. Společnost měla k dispozici ke kontrole kódu již vytvořený nástroj BugHunter, který sloužil pro statickou kontrolu kódu při code review a byl založen na využívání regulárních výrazů. Tento nástroj chtěl zadavatel nahradit nástrojem, který by byl schopný stejnou kontrolu provádět již při psaní kódu vývojářem a nabídnout mu možnost korekce kódu podle definované sady pravidel ve společnosti. Pro tyto požadavky byla k implementaci nástroje zvolena platforma Roslyn, která je dostupná jako doplněk v aktuální verzi vývojového prostředí Microsoft Visual Studio 2015. Nástroj byl vyvíjen na základě požadavků zadavatele. Dle požadavků by nástroj měl včas upozornit vývojáře na chybu ve zdrojovém kódu, která je definovaná v pravidlech společnosti a nabídnout mu korektní alternativu ke špatně napsanému kódu. Vytvořený nástroj by měl být spustitelný i nad stávajícími projekty a měl by být schopný vyhledat v nich chyby dle pravidel.

5.1 Syntaktické pravidla pro psaní kódu

Společnost má určitá syntaktická pravidla pro psaní kódu. Jedná se především o pravidla pro psaní kódu a pojmenování různých identifikátorů v programovacím jazyku C#. Napsaný kód je za pomocí kompilátoru Roslyn zkontrolován, zda používá definovanou syntaxi pro daný identifikátor, pokud však nejsou napsány podle pravidel ve společnosti, tak je nabídnuta oprava identifikátoru dle definované syntaxe.

Tab. 2 Příklady syntaktických pravidel pro psaní kódu

Identifikátor	Popis pravidla
Class, Structure	Psáno pomocí Pascalovy notace, nepoužívat stejný název u třídy a namespace.
Namespace	Psáno pomocí Pascalovy notace.
Method	Psáno pomocí Pascalovy notace.
Public property	Psáno pomocí Pascalovy notace, pokud je datového typu boolean použít prefix „Is“.
Non public field	Psáno pomocí Pascalovy notace, použit prefix „m“.
Interface	Psáno pomocí Pascalovy notace, použit prefix „I“.
Constant	Psáno velkými písmeny s využitím symbolu '_' jako rozdělovače slov.

5.2 Pravidla pro statickou analýzu kódu

Pravidla pro statickou analýzu kódu jsou více specifická pro každou společnost. Tyto pravidla určují vývojáři, jaké třídy, metody a konstrukty by neměl využívat při psaní kódu. Například když společnost využívá při psaní kódu především vlastní vytvořené knihovny místo standartních knihoven. Za pomocí kompilátoru Roslyn byly vytvořeny analyzátory pro kontrolu, zda vývojář nepoužívá jiné konstrukty než by při psaní kódu měl využívat, pokud je použije, tak je mu nabídnuta alternativa, která je doporučena dle definovaných pravidel.

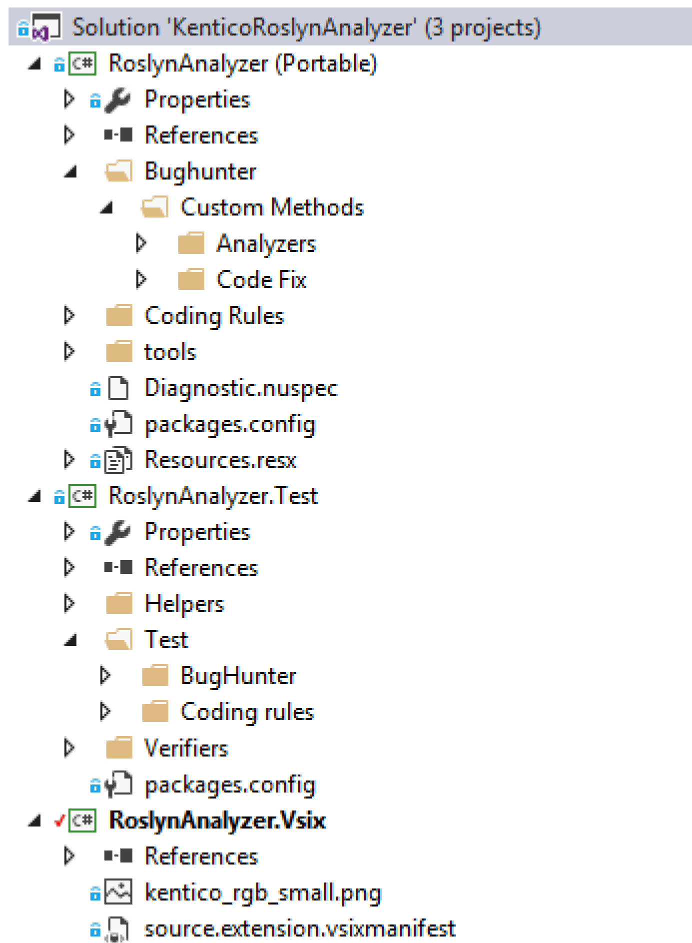
Tab. 3 Příklady pravidel pro statickou analýzu kódu

Název pravidla	Popis pravidla
UserHostAddress	Kontroluje, zda je v kódu použito Request.UserHostAddress, použít místo toho RequestContext.UserHostAddress.
Redirect	Kontroluje, zda je použito v kódu Request.Redirect, upozorní vývojáře, že se nemá používat.
PostBack	Kontroluje, zda je v kódu použito IsPostBack, místo toho použít RequestHelper.IsPostBack().
Callback	Kontroluje, zda je v kódu použito IsCallback, místo toho použít RequestHelper.IsCallback().
Browser	Kontroluje, zda je v kódu použito Request.Browser.Browser, použít místo toho BrowserHelper.GetBrowser().

5.3 Architektura

Celá architektura vytvořeného řešení se skládá ze tří projektů. Důležitým projektem z celého řešení KenticoRoslynAnalyzer je projekt samotného nástroje, který je nazván RoslynAnalyzer. Dalším projektem je RoslynAnalyzer.Test, což je projekt určený pro testování, který obsahuje i pomocné třídy, aby samotné testování bylo jednodušší. Dále obsahuje testovací třídy implementované pro daný analyzátor dohromady s code fixem. Samotné testovací třídy jsou taktéž rozděleny do složek dle hlavních kategorií. Posledním projektem je RoslynAnalyzer.Vsix, který slouží jako spouštěcí projekt pro nasazení do Visual Studia.

Již na začátku byl hlavní projekt RoslynAnalyzer rozdělen na více komponent podle pravidel, na jejichž základě je sestavena i architektura celého projektu. Hlavní rozdělení je dle druhu pravidel na Coding Rules a nebo BugHunter. Kategorie Coding rules je rozdělena na dvě podkategorie Naming conventions a Style guide, každá tato podkategorie, tak jako podkategorie Custom methods v kategorii BugHunter, je rozdělena také na Analyzátory a Code fixy. Pro každý analyzátor je možné vytvořit také více souborů s Code fixy.



Obr. 20 Architektura vytvořeného řešení nástroje

6 Implementace nástroje pro automatickou kontrolu kódu

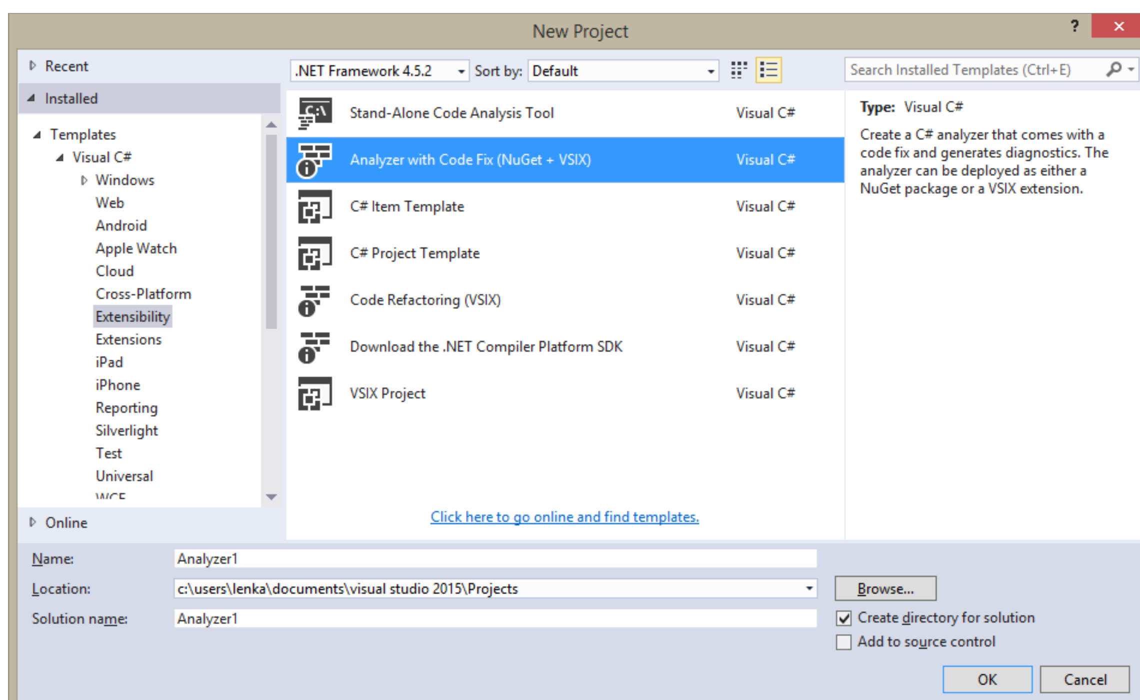
Pro automatickou kontrolu kódu ve společnosti Kentico software s.r.o byl vytvořen nástroj, který je implementován v programovacím jazyku C#, pro vývojové prostředí Microsoft Visual Studio 2015, za pomocí nového kompilátoru Roslyn.

Kontrola, zda jsou pravidla ve společnosti dodržována, probíhá během vývojového procesu. Některá pravidla lze kontrolovat automaticky, ale neobejdou se zde bez manuální kontroly kódu. Pravidla byla roztřízena do jednotlivých kategorií a poté byla implementována v programovacím jazyce C# s využitím kompilátoru Roslyn.

6.1 Vytvoření nástroje pro automatickou kontrolu kódu

Pro vytvoření nástroje bylo potřeba vývojové prostředí Microsoft Visual Studio 2015, které obsahuje i instalaci samotného kompilátoru Roslyn, který byl využit pro implementaci nástroje. Pro starší verzi Microsoft Visual Studio 2013 lze kompilátor Roslyn doinstalovat. K vytvoření analyzátoru existují specifické NET Compiler Platform SDK šablony, lze je ve Visual Studiu nainstalovat pomocí položky Extensions and Updates v menu Tools. Po nainstalování jsou k dispozici při vytvoření nového projektu tři konkrétní šablony projektu:

- Stand-Alone Code Analysis Tool – umožňuje vytvoření analýzy kódu pro aplikace příkazového řádku
- Analyzer with Code Fix (NuGet + VSIX) – umožňuje vytvoření analyzátoru s jedním nebo více Code fix pro návrh a opravy napsaného kódu. Zahrnuje vše potřebné pro sdílení vytvořeného analyzátoru pomocí NuGet balíčků nebo VSIX balíčků pro Visual Studio.
- Code Refactoring (VSIX) – tato šablona umožňuje vytvoření vlastního refaktoring kódu, který je propojen žárovkou ve Visual Studiu. (Sole, 2015)



Obr. 21 Vytvoření nového projektu s využitím šablon

Při vytváření nástroje pro automatickou kontrolu kódu byl vytvořen nový projekt s využitím šablony Analyzer with Code Fix (NuGet + VSIX). Pro každé pravidlo společnosti byl především vytvořen soubor pro analýzu kódu. Dále pokud byla určena náhrada za chybný kód, byl vytvořen code fix soubor, který za pomocí žárovky ve Visual Studiu nabídne alternativu pro nalezenou chybu nebo varování v kódu. Pro některá pravidla bylo potřeba vytvořit i pomocné soubory, pro úpravu kódu.

6.1.1 Vytvoření analyzátoru kódu

Základem nástroje pro automatickou kontrolu kódu je vytvoření analyzátorů, díky kterým jsou diagnostikovány pravidla pro psaní kódu. Pro vytvoření analyzátoru daného pravidla lze využít šablony Analyzer při přidávání nového souboru. Každá třída analyzátoru pravidla je založena na třídě DiagnosticAnalyzer. Z této třídy je zděděna vlastnost SupportedDiagnostics, která je typu ImmutableArray a slouží pro přidání pravidel, dále je zděděna metoda Initialize, která registruje akci působící na syntaxi nebo symboly.

Tab. 4 Metody pro registraci akcí

Metoda	Popis
RegisterSyntaxNodeAction	Registruje akce, které mají být provedeny při dokončení sémantické analýzy syntaktického uzlu.
RegisterSyntaxTreeAction	Registruje akce, které mají být provedeny poté, co je kód souboru analyzován.
RegisterSymbolAction	Registruje akce, které mají být provedeny při dokončení sémantické analýzy nad objektem vyplývajícího z ISymbol.
RegisterSemanticModelAction	Registruje akce, které mají být provedeny pro analyzovaný dokument kódu v souvislosti se sémantickým modelem dokumentu.
RegisterCompilationStartAction	Registruje akce, které mají být provedeny při spuštění kompilace.
RegisterCopilationEndAction	Registruje akce, které mají být provedeny po dokončení kompilace.
RegisterCodeBlockStartAction	Registruje akce, které mají být provedeny na začátku sémantické analýzy těla metody nebo výrazu, který je vně těla metody.
RegisterCodeBlockEndAction	Registruje akce, které mají být provedeny na konci sémantické analýzy těla metody nebo výrazu, který je vně těla metody.

Zdroj: (Sole, 2015)

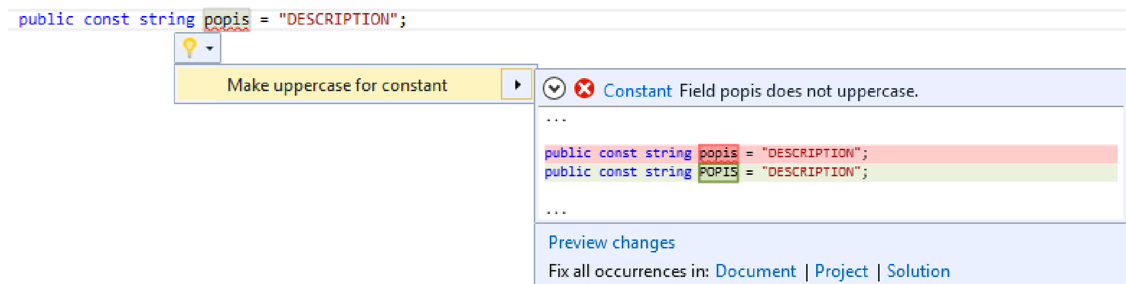
Při implementaci analyzátoru je nejdříve je potřeba definovat pole DiagnosticId, Title, MessageFormat, Description a Category představující informace o pravidlu, které se zobrazují ve Visual Studiu v okně s přehledem chyb a při zobrazení nápovědy u žárovky ve Visual Studiu.

Ve výchozí implementaci, tak jako při implementaci tohoto nástroje, je poskytnuta podpora pro lokalizaci řetězců, která zahrnuje ukládání řetězcových hodnot v souboru Resources.resx pro tento nástroj.

```
public const string DiagnosticId = "Constant";
private static readonly LocalizableString Title = new LocalizableResource-
String(nameof(Resources.NamingC10Title), Resources.ResourceManager, type-
of(Resources));
private static readonly LocalizableString MessageFormat = new LocalizableRe-
sourceString(nameof(Resources.NamingC10MessageFormat), Resources.ResourceManager,
typeof(Resources));
```

```
private static readonly LocalizableString Description = new LocalizableResource-
String(nameof(Resources.NamingC10Description), Resources.ResourceManager, type-
of(Resources));
private const string Category = "Naming";
```

Zdrojový kód 1 - Implementace informací o pravidlu pomocí lokalizovaných řetězců



Obr. 22 Zobrazení informací o pravidlu pomocí žárovky ve Visual Studiu

Samotné pravidlo je tvořeno pomocí typu DiagnosticDescriptor, kterému jsou předávány předcházející definované řetězcové hodnoty s informacemi o pravidlu. V parametru tohoto typu lze také určit, zda má být diagnostika povolena ve výchozím nastavení a je zde definován stupeň diagnostiky, který má hodnotu z výčtového typu DiagnosticServerity. Při implementaci nástroje bylo využito dvou stupňů diagnostiky – Error a Warning. Takto definované pravidlo je vytvořené pomocí vlastnosti SupportedDiagnostics. Lze definovat více pravidel v jednom souboru analyzátoru a poté je vytvořit v této vlastnosti.

```
private static DiagnosticDescriptor Rule = new DiagnosticDescriptor(DiagnosticId,
Title, MessageFormat, Category, DiagnosticSeverity.Error, isEnabledByDefault:
true, description:Description);

public override ImmutableArray<DiagnosticDescriptor> SupportedDiagnostics { get {
return ImmutableArray.Create(Rule); } }
```

Zdrojový kód 2 - Vytvoření pravidla se stupněm diagnostiky Error

Hlavní metoda analyzátoru Initialize, registruje akce (viz. Metody pro registraci akcíTab. 4) na události kompilátoru. Metodu pro registraci dané akce zvolíme dle toho, jaký prvek chceme analyzovat. V metodě lze registrovat i více akcí pro vytvoření více pravidel. Každé vytvořené pravidlo by mělo mít vytvořenou metodu pro analýzu daného prvku. Prvním argumentem metody Initialize je delegát provádějící skutečnou analýzu a druhým parametrem je výčet zvolený dle typu registrované akce. Například u pravidla pro konstanty se registruje akce pomocí RegisterSymbolAction a je použit prvek výčtového typu SymbolKind.Field.

Samotná analýza prvku je vytvořena pomocí podmínky pokud je field konstanta, ale zároveň není výčtového typu, a není napsán velkými písmeny, tak je vytvořena diagnostika, prostřednictvím třídy Diagnostic, s výše definovaným pravidlem, lokací vyhledaného symbolu a jeho jménem.

```
public override void Initialize(AnalysisContext context)
        {
            context.RegisterSymbolAction(AnalyzeField, SymbolKind.Field);
        }

        private void AnalyzeField(SymbolAnalysisContext context)
        {
            var field = (IFieldSymbol)context.Symbol;
            if (field.IsConst && !field.Type.TypeKind.Equals(TypeKind.Enum))
            {
                if (field.Name.ToCharArray().Any(char.IsLower))
                {
                    //vytvoření diagnostiky pravidla
                    var diagnostic = Diagnostic.Create(Rule, field.Locations[0],
field.Name);
                    context.ReportDiagnostic(diagnostic);
                }
            }
        }
```

Zdrojový kód 3 - Implmentace pravidla pro konstanty

Pokud si nejsme jisti druhem symbolu prvku nebo potřebujeme zjistit jakékoli informace o daném prvku, který chceme analyzovat, je nejlepší využít nástroj Syntax Visualizer, který je součástí Visual Studia. Tento nástroj nám pomáhá zjistit jakou použít metodu pro registraci akce.

Tento nástroj byl především využit při implementaci pravidel pro statickou analýzu kódu, kde byla zaregistrována akce RegisterSyntaxNodeAction, která má jako druhý parametr prvek z výčtu SytaxKind, který je vybrán dle toho, co se analyzuje, což nám napomáhá zjistit Syntax Visualizer, který zobrazuje veškeré informace o prvku. Při umístění kurzoru kdekoli v kódu jsou v nástroji zobrazeny veškeré informace daného prvku, což pomáhá při implementaci diagnostiky daného pravidla.

Například u implementace pravidla pro zakázané metody lze pomocí nástroje Syntax Visualizer zjistit, že daná metoda je součástí typu InvocationExpressionSyntax, která se skládá z typu MemberAccessExpressionSyntax, tento typ se může skládat z dalšího MemberAccessExpressionSyntax a nebo jen z typů IdentifierNameSyntax, záleží na tom, jak složitý je zápis v kódu.

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

 private void Forbidden(SyntaxNodeAnalysisContext context)
  {
    var forbiddenrNode = (InvocationExpressionSyntax)context.Node;
    if (!forbiddenrNode.ToString().Contains("CSafe"))
    {
      var expr = forbiddenrNode?.Expression as MemberAccessExpressionSyntax;
      if (expr != null)
      {
```

```
            var ident = expr?.Expression as IdentifierNameSyntax;
            if (ident != null)
            {
             if (expr.Name.ToString().Equals("ToLower") ||
expr.Name.ToString().Equals("StartsWith") || expr.Name.ToString().
Equals("EndsWith") || expr.Name.ToString().Equals("Equals") ||
expr.Name.ToString().Equals("CompareTo") || expr.Name.ToString().
Equals("IndexOf") || expr.Name.ToString().Equals("LastIndexOf"))
              {
                var diagnostic = Diagnostic.Create(Rule, expr.GetLocation(),
expr.Name);
                context.ReportDiagnostic(diagnostic);
               }
             }
        }
    }
  }
```

Zdrojový kód 4 - Implementace analýzy pro zakázané metody

6.1.2 Oprava kódu pomocí Code Fix

Poté co analyzátor detekuje a upozorňuje na problém v kódu, chceme vývojáři navrhnout případnou opravu, která je uvedena po kliknutí na žárovku ve Visual Studiu.

K implementaci souboru pro opravu chyb, lze využít šablonu CodeFix, která je dostupná při přidávání nové třídy. Pro daný analyzátor je vytvořený CodeFix soubor, který navrhuje možnosti oprav chybného kódu, tato třída je potomkem abstraktní třídy Microsoft.CodeAnalysis.CodeFixes.CodeFixProvider. Z této třídy je zděděná vlastnost FixableDiagnosticsIds, která vrací ImmutableArray s prvkem představující diagnosticID z Analyzátoru. Metodou pro zajištění oprav je GetAllFixProvider. Poslední děděnou metodou je RegisterCodeFixesAsync, což je metoda, kde je implementována hlavní logika. Parametrem této metody je struktura typu CodeFixContext reprezentující opravy kódu ze syntaxe uzlů, které mají vytvořenou diagnostiku. V této metodě je získán kořen celého dokumentu, díky kterému je poté nalezen uzel, který chceme opravit. Nakonec je zaregistrovaná akce. Tato akce obsahuje informace zobrazující se v žárovce Visual Studiu a především obsahuje delegáta, který provádí samotnou opravu kódu.

```
private const string title = "Make uppercase for constant";

   public sealed override ImmutableArray<string> FixableDiagnosticIds
   {
    get { return ImmubleArray.Create(Analyzerers.ConstAnalyzer.DiagnosticId); }
        }

   public sealed override FixAllProvider GetFixAllProvider()
    {
       return WellKnownFixAllProviders.BatchFixer;
    }
```

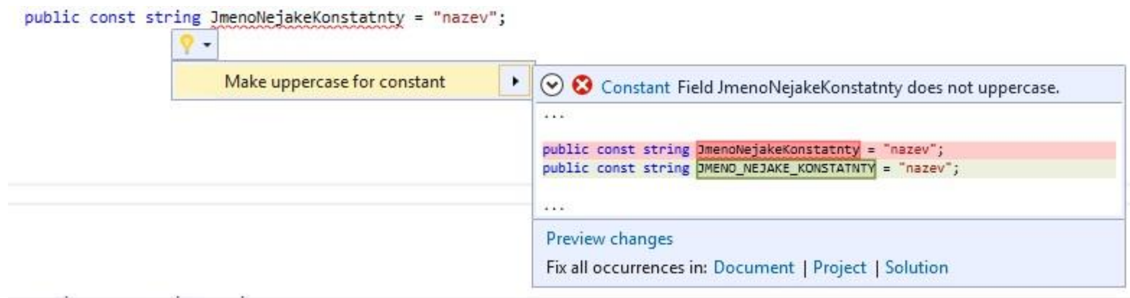
```
   public sealed override async Task RegisterCodeFixesAsync(CodeFixContext con-
text)
   {
  var root = await context.Document.
             GetSyntaxRootAsync(context.CancellationToken).ConfigureAwait(false);
  var diagnostic = context.Diagnostics.First();
  var token = root.FindToken(diagnostic.Location.SourceSpan.Start);
     context.RegisterCodeFix(
                CodeAction.Create(title: title,
               createChangedSolution: c => MakeUpper(context.Document, token, c),
               equivalenceKey: title),
                diagnostic);
        }
```

Zdrojový kód 5 -Implementace zděděných metod v CodeFix pro konstanty

V případě oprav názvů konstant je to vytvořená metoda MakeUpper, která zvětší všechna písmena v názvu konstanty. V této metodě se zjistí jméno předávaného tokenu. Je také definován sémantický model celého dokumentu a pomocí něj je zjištěn symbol předávaného tokenu. Pokud je ve jménu konstanty jakékoli velké písmeno, tak je jméno konstanty tvořeno více slovy, mezi takové slova je vložen znak „_“ a celé jméno je napsáno velkými písmeny. Jinak je pouze přejmenováno jméno konstanty. Pro obě možnosti je společné zavolání přejmenování symbolu pomocí třídy Renamer. Příklad této opravy kódu:

```
private async Task<Solution> MakeUpper(Document document, SyntaxToken declarati-
on, CancellationToken cancellationToken)
   {
     var name = declaration.ValueText;
     var semantiModel = await document.GetSemanticModelAsync(cancellationToken);
     var symbol = semantiModel.GetDeclaredSymbol(declaration.Parent,
cancellationToken);
     var solution = document.Project.Solution;
     var upperChar = name.ToCharArray().Any(char.IsUpper);
            if (upperChar)
            {
                var firstLow = char.ToLower(name[0]) + name.Substring(1);
                var upper = MatchUpper(firstLow);
                var newName = $"{upper}";
                return await Renamer.RenameSymbolAsync(solution, symbol, newName,
solution.Workspace.Options, cancellationToken);
            }
            else
            {
             var newName = $"{name.ToUpper()}";
             return await Renamer.RenameSymbolAsync(solution, symbol, newName,
solution.Workspace.Options, cancellationToken);
            }
       }
```

Zdrojový kód 6 - Implementace metody MakeUpper



Obr. 23 Znázornění opravy jména konstanty složené z více slov

V případě pravidel pro statickou analýzu kódu, se nevyužívá přejmenování daného symbolu, ale je zde aplikováno nahrazení daného uzlu nebo tokenu. Například pravidlo, které má nahradit metody použité mimo konceptuální rámec společnosti, je založeno na náhradě uzlu a jeho přejmenování na metodu definovanou ve společnosti. Při nahrazení daného uzlu je využito pro vytvoření nového uzlu třídy SyntaxFactory, u níž je vytvořen druh IdentifierName, který vrací typ IdentifierSyntax. Parametrem metody IdentifierName je jméno uzlu typu string, v našem případě je zde využito nahrazení názvu uzlu za nový název. Nově vytvořený uzel poté nahradí původní uzel u syntaktického kořene.

```
public sealed override async Task RegisterCodeFixesAsync(CodeFixContext context)
        {
            var root = await context.Document.
GetSyntaxRootAsync(context.CancellationToken).ConfigureAwait(false);
            var diagnostic = context.Diagnostics.First();
            var token = root.FindToken(diagnostic.Location.SourceSpan.Start).
Parent.AncestorsAndSelf().OfType<InvocationExpressionSyntax>().First();
            context.RegisterCodeFix(CodeAction.Create(title: title,
               createChangedSolution: c => Forbidden(context.Document, token, c),
                  equivalenceKey: title),
                   diagnostic);
        }
        private async Task<Solution> Forbidden(Document document, InvocationEx-
pressionSyntax declaration, CancellationToken c)
        {
            var root = await document.GetSyntaxRootAsync(c);
            var memberAccessExpr = declarati-
on.WithArgumentList(declaration.ArgumentList).Expression as MemberAccessExpressi-
onSyntax;
            var name = memberAccessExpr.Name.ToString();
            var newName = name;
            var newRoot = root;
            if (name.Equals($"ToLower") || name.Equals($"IndexOf") || na-
me.Equals("StartsWith") || name.Equals("EndsWith") || name.Equals("Equals") ||
name.Equals("CompareTo") || name.Equals("IndexOf") || name.Equals("LastIndexOf"))
            {
                var newNode = SyntaxFactory.
IdentifierName(declaration.ToString().Replace(name, $"{name}CSafe"));
                newRoot = root.ReplaceNode(declaration, newNode);
            }
            var semantiModel = await document.GetSemanticModelAsync(c);
```

```
            var solution = document.Project.Solution;
            return document.WithSyntaxRoot(newRoot).Project.Solution;
        }
```

Zdrojový kód 7 - Implementace metody pro zakázané metody v CodeFix

Podobným způsobem je implementováno i pravidlo pro odstranění regionů ze zdrojového kódu, které je založeno na nahrazení kořenového uzlu. U tohoto pravidla byla vytvořena pomocná třída, která je založena na abstraktní třídě CSharpSyntaxRewriter.

6.1.3 Třída založena na Rewriter

Třída pro odstranění regionů je založena na abstraktní třídě CSharpSyntaxRewriter. Tato třída je založena na procházení uzlů v syntaktickém stromu, jelikož prvek region a endregion je trivia, tak je zde využita metoda VisitTrivia. V této metodě se v podmínce ověřuje, zda druh SyntaxTrivia je RegionDirectiveTrivia nebo EndRegionDirectiveTrivia a když je tak se vrátí defaultní SyntaxTrivia.

```
public override SyntaxTrivia VisitTrivia(SyntaxTrivia trivia)
        {
            var triviaDefault = default(SyntaxTrivia);
            var isRegionOrEndRegionStructuredTrivia = trivia.HasStructure &&
                (trivia.Kind() == SyntaxKind.RegionDirectiveTrivia ||
                 trivia.Kind() == SyntaxKind.EndRegionDirectiveTrivia);
            if (!isRegionOrEndRegionStructuredTrivia)
            {
                triviaDefault = base.VisitTrivia(trivia);
            }
                // return default(SyntaxTrivia)
            else
            {
                triviaDefault= default(SyntaxTrivia);
            }
            return triviaDefault.WithoutAnnotations();
        }
```

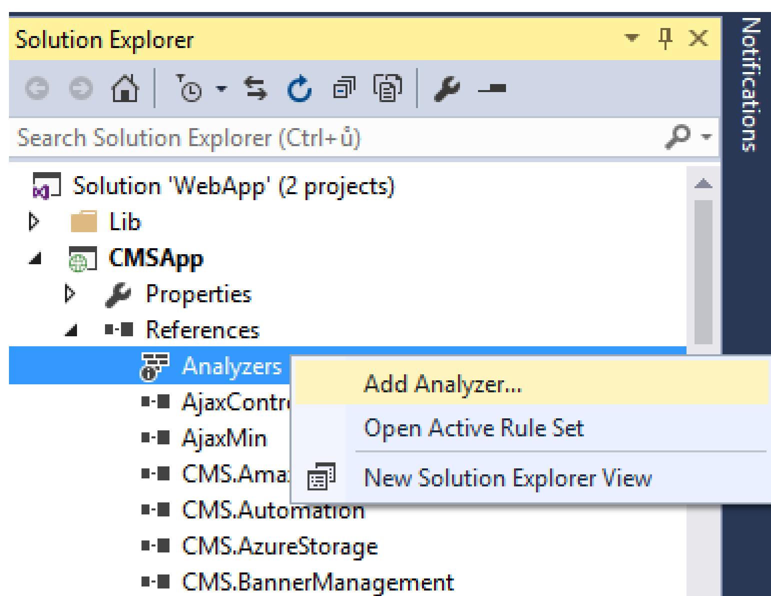
Zdrojový kód 8 - Metoda pro odstranění regionů

6.2 Kompilace nástroje

Při využití šablony projektu analyzátoru s Code Fix se automaticky generuje NuGet balíček, který obsahuje zkompilovaný analyzátor, který lze pak sdílet ostatním vývojářům. Při každém ladění analyzátoru stiskem klávesy F5 nebo při spuštění projektu analyzátoru vytvoří Microsoft Visual Studio soubor .dll (v našem případě RoslynAnalyzer.dll) a opětovně distribuuje NuGet balíček, který obsahuje vytvořený analyzér. Při vytváření se také generuje VSIX balíček, který lze publikovat na Visual Studio Gallery. Tento balíček lze využít i k samotné instalaci nástroje u vývojáře.

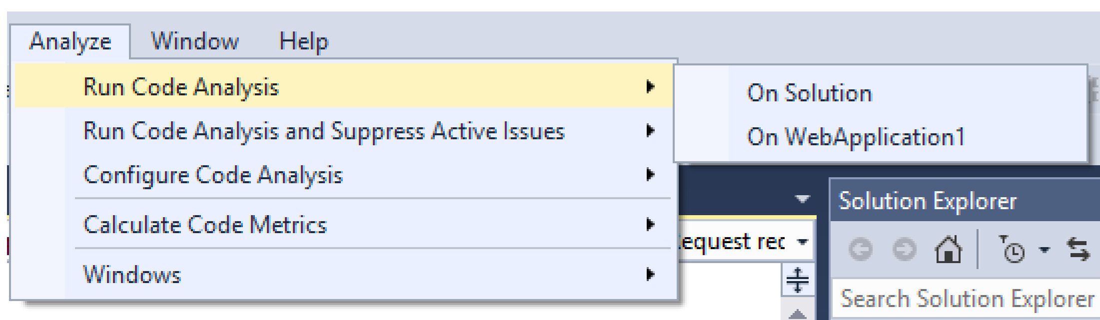
6.3 Možnosti použití analyzátoru

Pro každý projekt lze použít vytvořený nástroj například za pomocí přidáním knihovny analyzátoru do daného projektu. V nástrojovém okně Solution Explorer ve Visual Studiu, kde lze vidět celkovou architekturu celého řešení. Pod položkou References daného projektu je Analyzers, kde se nachází veškeré využívané analyzátory a kde je i možnost přidat požadovaný analyzátor.



Obr. 24 Přidání analyzátoru k danému projektu

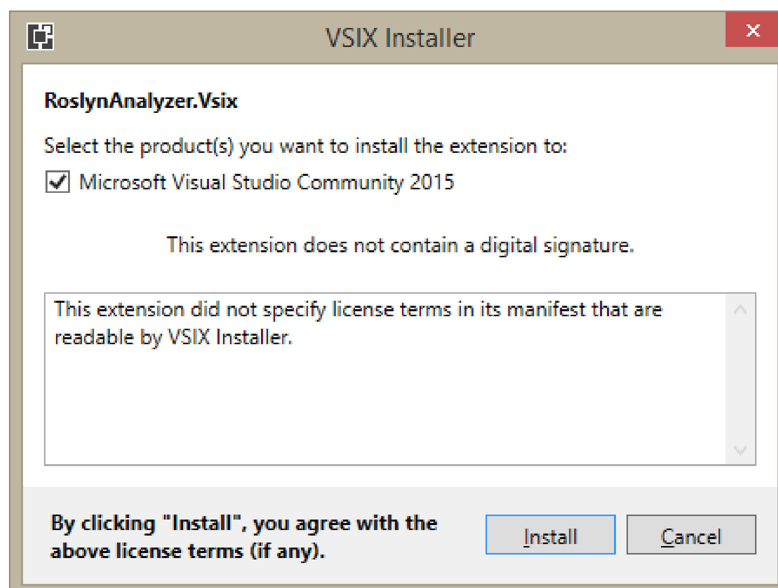
Po přidání analyzátoru do projektu, ho lze využít i k analýze již vytvořeného projektu. Jak lze vidět Obr. 25 analyzátor je možné spustit nad celým řešením nebo nad daným projektem, který máme zrovna v Solution Explorer označen. Veškeré chyby a upozornění jsou poté zobrazeny v nástrojovém okně Error List ve Visual Studiu.



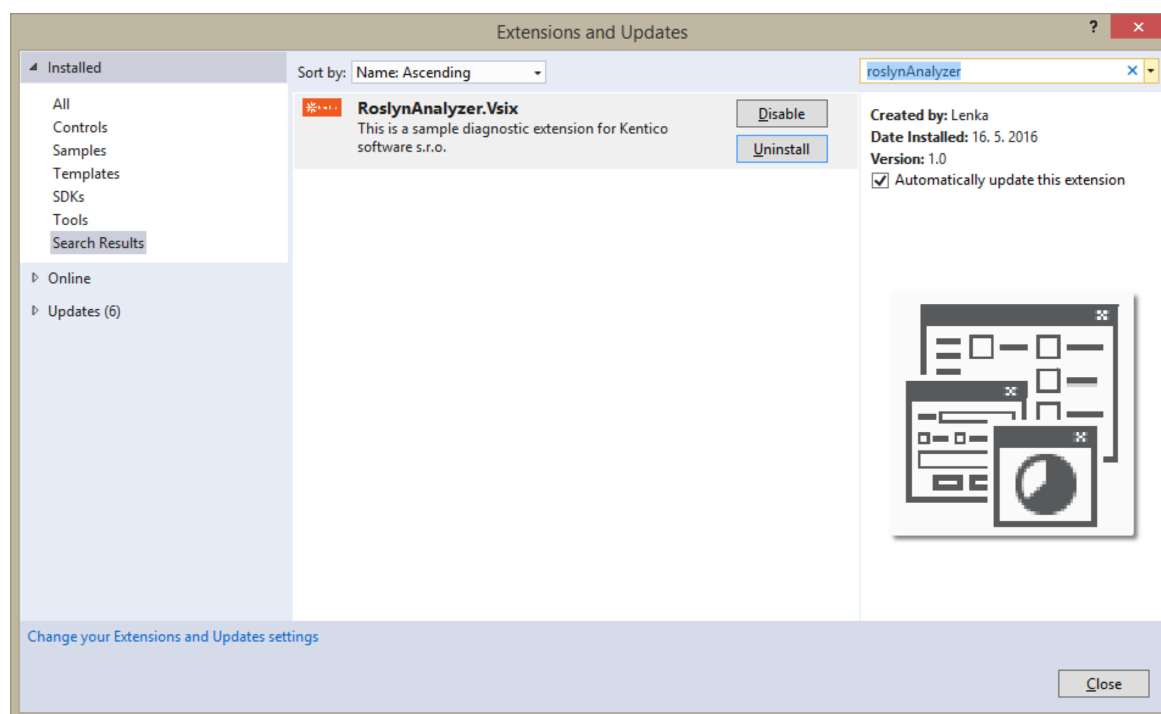
Obr. 25 Spuštění analyzátoru

Analyzátor lze i nainstalovat do vývojového prostředí Microsoft Visual Studio 2015 pomocí instalačního souboru RoslynAnalyzer.vsix. Tímto způsobem je analyzátor

doinstalován přímo do celého vývojového prostředí a je možné ho spustit nad jakýmkoli projektem, který spustíme v Microsoft Visual Studio. Nainstalovaný doplněk lze vypnout nebo zapnout v záložce Extensions and Updates v menu Tools.



Obr. 26 Instalace nástroje



Obr. 27 Nainstalovaný nástroj ve vývojovém prostředí

6.4 Testování nástroje pro analýzu kódu

Testování vytvořeného nástroje probíhalo více způsoby. Vytvořené řešení obsahuje projekt pro testování, který je automaticky vytvořen při využití dané šablony. Tento projekt má pro snadnější testování již naimplementované pomocné třídy, kterých se využívá při samotných testech daných analyzátorů. Pro základní testování analyzátorů byly vytvořeny testovací třídy, společné pro každý analyzátor a code fix pro dané pravidlo. V testovacích třídách byly nasimulovány případy, které by mohly nastat, avšak i přesto se na některé možnosti přišlo až při testování na zdrojových kódech společnosti.

Každá testovací třída je potomkem abstraktní třídy CodeFixVerifier a obsahuje dvě překryté metody GetCSharpCodeFixProvider a GetCSharpDiagnosticAnalyzer, které vrací vytvořený CodeFix a Analyzátor pro dané pravidlo. Dále obsahuje jednotlivé testovací metody s případy, které by mohli nastat při reálném spuštění. Každá tato metoda je založena na porovnávání.

```
protected override CodeFixProvider GetCSharpCodeFixProvider()
 {
     return new ConstCodeFixProvider();
 }

 protected override DiagnosticAnalyzer GetCSharpDiagnosticAnalyzer()
 {
     return new ConstAnalyzer();
  }
```

Zdrojový kód 9 - Překryté metody v testovací třídě

```
[TestMethod]
        public void TestMethod3()
        {
            var test = @"
    using System;
    using System.Collections.Generic;
    using System.Linq;
    using System.Text;
    using System.Threading.Tasks;
    using System.Diagnostics;

    namespace ConsoleApplication1
    {
        class Testovaci
        {
          public const string NazevKonstanty = 'DESCRIPTION';
        }
    }";
            var expected = new DiagnosticResult
            {
                Id = ConstAnalyzer.DiagnosticId,
                Message = String.Format("Const {0} does not uppercase.", "Na-
zevKonstanty"),
```

```
                Severity = DiagnosticSeverity.Error,
                Locations =
                    new[] {
                            new DiagnosticResultLocation("Test0.cs", 13, 31)
                        }
            };

            VerifyCSharpDiagnostic(test, expected);

            var fixtest = @"
    using System;
    using System.Collections.Generic;
    using System.Linq;
    using System.Text;
    using System.Threading.Tasks;
    using System.Diagnostics;

    namespace ConsoleApplication1
    {
        class Testovaci
        {
          public const string NAZEV_KONSTANTY = 'DESCRIPTION';
        }
    }";
            VerifyCSharpFix(test, fixtest);
             }
```

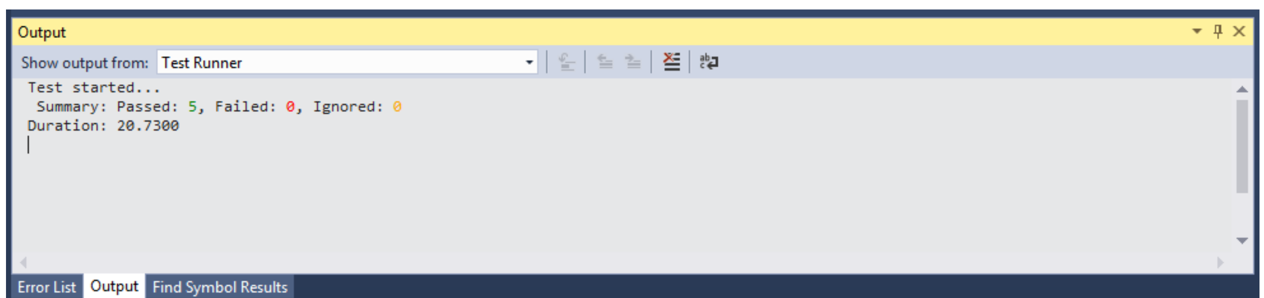
Zdrojový kód 10 - Metoda pro testování víceslovného názvu konstanty

Testovací třída může obsahovat nespočet testovacích metod a měla by pokrýt všechny případy, které mohou nastat při psaní kódu. Testy lze spustit pro celou testovací třídu, nebo pro každou testovací metodu zvlášť. Po spuštění testovací třídy se zobrazí sumarizace všech testovacích metod v nástrojovém okně Output. Při testování vytvořených pravidel pro statickou analýzu kódu, bylo potřeba do pomocné třídy DiagnosticVerifier přidat reference pro využívané namespace. Když dané reference chyběly, nastala chyba při testování diagnostiky daného pravidla. Pokud je v testovací metodě chyba v opravě kódu, tak je zobrazen kód očekávaný a aktuální, jsou zde zobrazeny rozdíly v kódu, je tedy nutné opravit Code Fix souboru pro dané pravidlo.

Při vyhodnocení testovací metody je přípustný jeden ze dvou výsledků testu. Prvním případem je, že daný test proběhl bez chyb. Druhou možností je, že nastala chyba s popisem, že dané jméno neexistuje v aktuálním kontextu. Tato chyba je přijatelná, protože chybně psaný kód je opraven, ale chybí reference k danému jménu, tyto reference si doplní při psaní kódu samotný vývojář. Tato chyba nastává při využívání vlastních vytvořených knihoven místo standardních. Aby test byl vyhodnocen kladně, musely by být firemní knihovny přidány do pomocné třídy pro testování DiagnosticVerifier i do samotného analyzátoru.

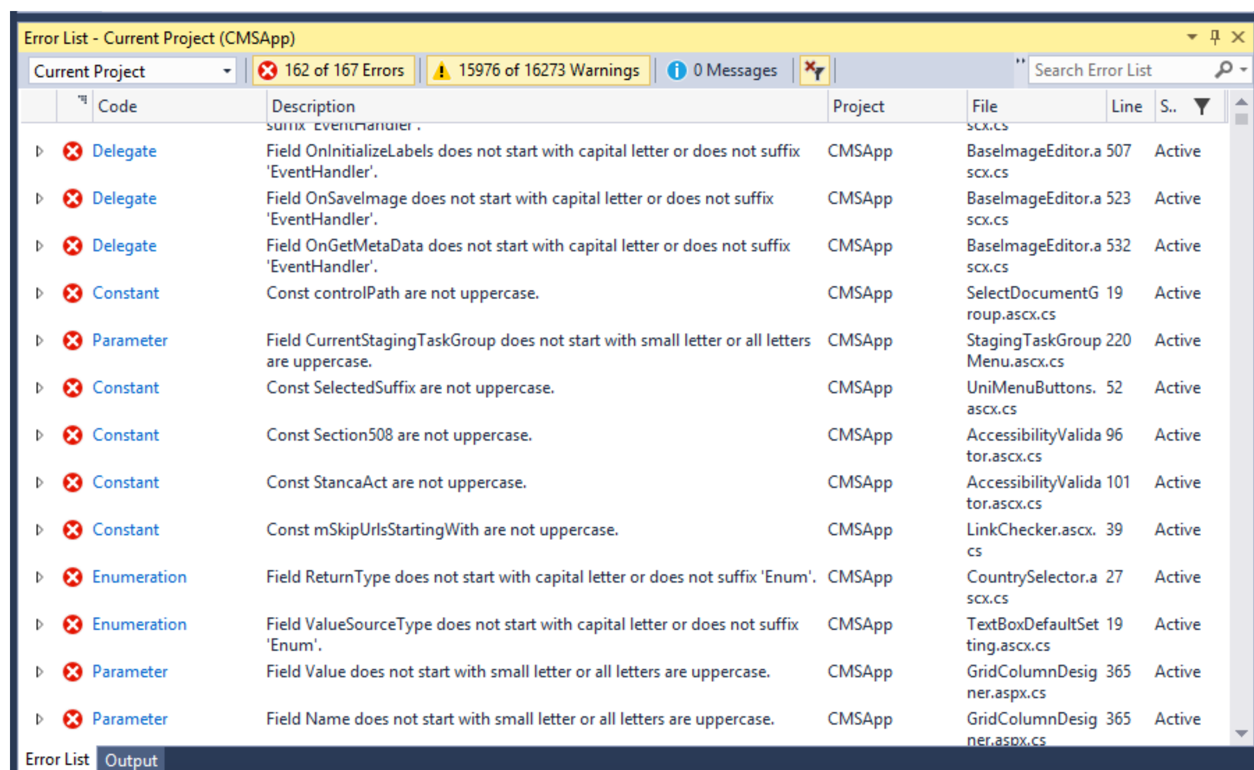
```
Test started...
Failed: Test.Test.BugHunter.BrowserTest.TestMethod3
  Assert.IsTrue failed. Fix introduced new compiler diagnostics:
Test0.cs(10,31): error CS0103: The name 'BrowserHelper' does not exist in the current context

New document:

    using System;
    using System.Web.UI;

    namespace ConsoleApplication1
    {
        class Test : Page
        {
            public void Test() {
                var browser = BrowserHelper.GetBrowser();
            }
        }
    }
```

Obr. 28 Výsledek testovací metody – přijatelná chyba



Obr. 29 Výsledek proběhlého testování

Další testování proběhlo přímo na zdrojových kódech společnosti Kentico software s.r.o.. Společností byl poskytnut zdrojový kód aplikace, kterou mají dostupnou na webových stránkách. Nad tímto zdrojovým kódem byl spuštěný naimplementovaný nástroj. Zjištěné chyby a varování byly v kódu podtrhnuty příslušnou barvou a také byly zobrazeny v nástrojovém okně Error List. V tomto okně lze vidět popis dané chyby nebo varování, projekt, soubor a řádek, kde se daná chyba nebo varování vyskytuje.



Obr. 30 Nástrojové okno Error List s nalezenými chybami

Díky testování bylo možné zjistit, jaké všechny možnosti mohou nastat při analýze daných uzlů a co vše tedy musí být zahrnuto v diagnostice daného pravidla. Například v analyzátoru pro pravidlo konstant, se musela zahrnout diagnostika, zda není field položkou ve výčtovém typu, protože tyto položky jsou definovány také jako konstanty. Veškeré nedostatky zjištěné při testování byly zahrnuty do diagnostiky a případně k nim implementována oprava kódu v nástroji.

7 Závěr

7.1 Zhodnocení vývoje nástroje

Cílem práce bylo vytvoření nástroje pro automatickou kontrolu kódu během vývoje. Ve společnosti se snaží docílit, aby všichni vývojáři dodržovali stejnou strukturu kódu, která je definovaná množinou pravidel. Pro tyto potřeby byl vyvinut nástroj pro společnost Kentico software s.r.o. Nástroj slouží ke kontrole pravidel pro psaní kódu, ať se jedná o syntaktická pravidla nebo o pravidla pro statickou analýzu kódu. Nástroj provádí kontrolu již při samotné tvorbě kódu, ale lze ho využít i pro kontrolu kódu stávajícího.

Vytvořený nástroj byl rozdělen do více kategorií dle daných pravidel poskytnutých společností Kentico software s.r.o. Společnost disponuje vlastními knihovnami, které jsou často nadstavbou standardních knihoven, měly by se využívat v kódu jako jejich náhrada. Nástroj poskytuje kontrolu samotnému vývojáři již při psaní kódu a lze tak eliminovat chyby obsažené v kódu, které by poté musely být vyhledány ve zdrojovém kódu manuálně. Při oznámení chyby nebo varování nástrojem je vývojáři nabídnuta alternativa použití nebo alespoň popis dané chyby nebo varování. Vývojář tak může ihned kód opravit pomocí nápovědy zobrazené v žárovce ve vývojovém prostředí Microsoft Visual Studio.

Nástroj byl otestován pomocí testovacích tříd při implementaci, tak aby pokryl veškeré případy, které mohou nastat v kódu. Byl také otestován na zdrojových kódech poskytnutých společností a byl spuštěn i přímo ve společnosti Kentico software s.r.o. u některých vývojářů, aby se otestovala funkčnost nástroje přímo ve společnosti. Dostalo se tak zpětné vazby na vytvořený nástroj a nahlášené nedostatky nástroje byly analyzovány a opraveny.

U nových vývojářů ve společnosti Kentico software s.r.o. je dle informací poskytnutých společností přibližně polovina chyb ve zdrojovém kódu způsobena nedodržením syntaktických pravidel a také pravidel pro statickou analýzu kódu, což způsobuje navrácení kódu z code review zpět k vývojáři, který musí chybu opravit. Poté následuje opětovná kontrola pomocí code review. Přínosem vytvořeného nástroje je především schopnost podchytit chyby již během tvorby samotného kódu, což ušetří čas při code review. Tento ušetřený čas se může věnovat psaní nového kódu. Nástroj by měl také eliminovat chyby nahlášené zákazníkem, které jsou zahrnuty v pravidlech. Včasné zjištění chyb, tak zajistí rychlejší a plynulejší vývoj, jak v samotné společnosti, tak u zákazníka, který nemusí čekat na opravu kódu. Manuální kontrola bude zjednodušena a zaměřena na jiné problémy v kódu. Vytvořený nástroj zároveň pomáhá při větší udržitelnosti a škálovatelnosti komplexního řešení, kterým Kentico CMS/EMS je.

7.2 Možnosti budoucího rozvoje

Vytvořený nástroj je možné dál rozvíjet. Jakákoliv nová pravidla je možné doimplementovat do nástroje. Pro každé nové pravidlo bude potřeba napsat analyzátor a případně i CodeFix pro návrh oprav daného pravidla. Stávající pravidla lze upravovat dle nových požadavků, například zahrnout do pravidla další případy, které by mohli u něj nastat.

Možnosti budoucího vývoje nejsou jen záležitostí vytvoření nových nebo upravení stávajících pravidel pro psaní kódu v programovacím jazyce C#. Jelikož samotný programovací jazyk C# se rychle vyvíjí a nabízí v každé nové verzi nové možnosti pro implementaci kódu, což nabízí i další možnosti pro tvorbu pravidel pro psaní kódu. Programovací jazyk C# je jen jeden z jazyků, který je v Kenticu využíván. Jsou zde také využívány k implementaci další programovací jazyky jako například JavaScript, ASP.NET, pro které existují ve společnosti Kentico také pravidla pro psaní kódu. Bohužel kompilátor Roslyn je prozatím poskytován pro programovací jazyk C# a Visual Basic. V budoucnu však může být poskytnut i pro další programovací jazyky což by společnostem umožňovalo rozsáhlejší tvorbu analyzátorů. Samotný kompilátor Roslyn se bude určitě také nadále rozvíjet a nabízet další možnosti pro analýzu kódu a opravy psaného kódu.

Vytvořený nástroj představuje do budoucna pro společnost automatizovanou kontrolu nad psaným kódem a nabízí možnost korekce podle definované sady pravidel. Ve výsledku se zlepší i čitelnost a celková srozumitelnost kódu v rámci celého vývojového týmu. Vývojáři také odpadá nutnost učit se veškerá pravidla pro psaní kódu. Využívání nástroje ušetří nejen čas, ale také finance, které jsou vynaloženy při opravě špatně napsaného kódu.

8 Literatura

ALBAHARI, JOSEPH A BEN ALBAHARI. *C# 6.0 in a Nutshell: The Definitive Reference*. 6. vydání. Sebastopol: O'Reilly Media, Inc., 2015. ISBN 978-1-491-92706-9.

DEL SOLE, ALESSANDRO. *Visual basic 2015 unleashed*. Indiana: Pearson Education, 2015. ISBN 978-067-2334-504.

GASIOR, ŁUKASZ A MICHAL JASEJ. *ReSharper essentials: make your microsoft visual studio work smarter with ReSharper*. Birmingham, England: Packt Publishing Ltd, 2014. Community experience distilled. ISBN 978-1-84969-871-9.

HASAN, NOORUL. *About C# Programming: History of C# Programming* [online]. 2012 [cit. 2016-0-10]. Dostupné z:
http://aboutcsharpprogramming.blogspot.cz/2012/09/history-of-c-programming.html

HAZZARD, KEVIN. A JASON. BOCK. *Metaprogramming in .NET*. Shelter Island, NY: Manning Publications, 2013. ISBN 16-172-9026-2.

HOLEC, MIROSLAV. *Miroslav Holec: Novinky v C# a kejkle s Roslynem* [online]. 2015, 29. března 2016 [cit. 2016-02-10]. Dostupné z: https://www.miroslavholec.cz/blog/novinky-v-c-sharp-a-kekle-s-roslynem

HOLUB, TOMÁŠ. Živě: *Čím se liší C# od C++* [online]. 2002 [cit. 2016-02-05]. Dostupné z: http://www.zive.cz/clanky/cim-se-lisi-c-od-c/sc-3-a-107869/

JOHNSON, BRUCE. *Professional Visual Studio 2015*. Indianapolis: John Wiley Sons, 2015. Programmer to programmer. ISBN 978-1-119-06805-1.

JVP, JAYANTHAN. *Developer In .NET: Components of .Net Framework* [online]. 2011 [cit. 2016-02-10]. Dostupné z:
http://www.developerin.net/a/39-Intro-to-.Net-FrameWork/23-Components-of-.Net-Framework

KULMAN, IGOR. *KULMAN: Java vs C# z hľadiska pohodlnosti programovania* [online]. 2010 [cit. 2016-02-20]. Dostupné z: https://www.kulman.sk/sk/content/java-vs-csharp-z-hladiska-pohodlosti-programovania/

PANCHAL, DHAVAL. Scrum Alliance: What is Definition of Done (DoD)? [online]. 2008 [cit. 2016-04-11]. Dostupné z: https://www.scrumalliance.org/community/articles/2008/september/what-is-definition-of-done-(dod)

PUŠ, PETR. *Živě: Poznáváme C# a Microsoft .NET – 1.díl* [online]. 2004 [cit. 2016-02-16]. Dostupné z: http://www.zive.cz/clanky/poznavame-c-a-microsoft-net--1dil/sc-3-a-120978/default.aspx

STEPHENS, ROD. *C# 5.0: Programmer's reference*. Indianapolis, Indiana: John Wiley & Sons, Inc., 2014. ISBN 978-1-118-84729-9.

ŠIMEČEK, MARTIN. *Czech MSDN Blog: Co je nového v C# 6.0* [online]. 2015 [cit. 2016-02-16]. Dostupné z: https://blogs.msdn.microsoft.com/vyvojari/2015/05/11/co-je-novho-v-c-6-0/

VIRIUS, MIROSLAV. *C# pro zelenáče*. Praha: Neocortex, 2002. Bestseller for all. ISBN 80-863-3011-7.

JETBRAINS. *ReSharper features* [online]. [cit. 2016-02-10]. Dostupné z:
https://www.jetbrains.com/resharper/features

MICROSOFT. *Microsoft Open Technologies: .NET Compiler Platform ("Roslyn")* [online]. 2014 [cit. 2016-02-10]. Dostupné z:
https://roslyn.codeplex.com/wikipage?title=Overview&referringTitle=Home

PROJECTCSHARP.COM. *Project CSharp: Microsoft .NET Framework 3.0 = .NET 2.0 + Windows Communication Foundation + Windows Presentation Foundation + Windows Workflow Foundation + Windows Card Space* [online]. 2010 [cit. 2016-02-10]. Dostupné z: http://www.projectcsharp.com/2010/07/microsoft-net-framework-30-net-20.html

Webový vývoj v ASP.NET 2.0 pomocí bezplatných Express nástrojů pro úplné začátečníky. Microsoft s.r.o. [online]. Microsoft s.r.o., 2005 [cit. 2016-02-12]. Dostupné z: http://download.microsoft.com/download/4/B/1/4B1337D6-B0B0-4535-B72E-417FE4D8251F/ASP_NET2_pro_zacatecniky.pdf

MICROSOFT. *Microsoft Development Network: Visual Studio – sada IDE* [online]. [cit. 2016-02-10]. Dostupné z: https://msdn.microsoft.com/cs-cz/library/dn762121.aspx

MICROSOFT. *Microsoft Developer Network: Introduction to the C# Language and the .NET Framework* [online]. [cit. 2016-02-02]. Dostupné z: https://msdn.microsoft.com/cs-cz/library/z1zx9t92.aspx

MICROSOFT. *Microsoft Developer NetWork: C# and Java: Comparing Programming Languages* [online]. [cit. 2016-02-03]. Dostupné z: https://msdn.microsoft.com/en-us/library/ms836794.aspx

Přílohy

A Obsah přiloženého CD

Složky a jejich obsah:

- Aplikace – knihovna nástroje, instalační soubor VSIX
- Dokumenty – elektronická verze závěrečné práce, návod k instalaci nástroje
- KenticoRoslynAnalyzer – zdrojové kódy nástroje
- Testovací – testovací aplikace pro vyzkoušení pravidel

B Seznam syntaktických pravidel

Název pravidla	Popis	Typ
Class, Structure	Psáno pomocí Pascalovy notace, nepoužívat stejný název u třídy a namespace.	ERROR
Namespace	Psáno pomocí Pascalovy notace.	ERROR
Enum	Psáno pomocí Pascalovy notace, používat sufix „Enum“.	ERROR
Method	Psáno pomocí Pascalovy notace.	ERROR
Public property	Psáno pomocí Pascalovy notace, pokud je datového typu boolean použít prefix „Is“.	WARNING
Non public field	Psáno pomocí Pascalovy notace, použit prefix „m“.	WARNING
Delegate	Psáno pomocí Pascalovy notace, použít sufix „EventHandler“, nepoužívat sufix „Delegate“.	ERROR
Interface	Psáno pomocí Pascalovy notace, použit prefix „I“.	ERROR
Constatnt	Psáno velkými písmeny s využitím symbolu '_' jako rozdělovače slov.	ERROR
Parameter	Psáno pomocí velbloudí notace.	ERROR
GenericTypeParameter	Psáno pomocí Pascalovy notace, používat prefix „T“.	ERROR
CompoundWords	Většina složených výrazů jsou považovány jako samostatná slova. Nezvětšovat každé slovo v tzv. složenině zkrácených slov. Možnost použít Pascalovu i velbloudí notací dle místa použití.	WARNING
Region	Nepožívat regiony. Často to naznačuje, že toho třída dělá až moc a je kandidátem na přepis.	WARNING

C Seznam pravidel pro statickou analýzu kódu

Název pravidla	Popis pravidla	Typ
WhereLike	Kontroluje, zda je v kódu použito WhereLike nebo WhereNotLike, upozorní, že se má místo toho použít StartsWith(), EndsWith(), Contains().	ERROR
UserHostAddress	Kontroluje, zda je v kódu použito Request.UserHostAddress, použít místo toho RequestContext.UserHostAddress.	ERROR
IO	Kontroluje, zda je v kódu použito namespace System.IO, použít místo toho CMS.IO.	WARNING
Cookies	Kontroluje, zda je v kódu použito Request.Cookies a Response.Cookies – oba jsou zakázány. Místo toho použít CookieHelper.RequestCookies nebo CookieHelper.ResponseCookies.	ERROR
Url	Kontroluje, zda je v kódu použito Request.Url, místo toho použít RequestContext.Url.	ERROR
Browser	Kontroluje, zda je v kódu použito Request.Browser.Browser, použít místo toho BrowserHelper.GetBrowser().	ERROR
Redirect	Kontroluje, zda je použito v kódu Request.Redirect, upozorní vývojáře, že se nemá používat.	WARNING
ForbiddenMethod	Kontroluje, zda jsou v kódu použité zakázané metody, místo toho CSafe metody. Zakázané metody- ToLower, ToUpper, StartsWith, EndWith, LastIndexOf.	WARNING
ClientScript	Kontroluje, zda je použito v kódu ClientScript, místo toho použít ScriptHelper a metodu se stejným názvem.	ERROR
PostBack	Kontroluje, zda je v kódu použito IsPostBack, místo toho použít RequestHelper.IsPostBack().	ERROR

Callback	Kontroluje, zda je v kódu použito IsCallback, místo toho použít RequestHelper.IsCallback().	ERROR
LuceneSearchDocumentClass	Kontroluje, zda je v kódu použito LuceneSearchDocument, místo toho použít ISearchDocument.	ERROR