TECHNICKÁ UNIVERZITA V LIBERCI
Fakulta mechatroniky, informatiky
a mezioborových studií

# NÁVRH MÍRY PRO VÝPOČET TOXICITY KÓDU A JEJÍ IMPLEMENTACE

## Diplomová práce

*Studijní program:* *N2612 – Elektronika a informatika*
*Studijní obor:* *1802T007 – Informační technologie*
*Autor práce:* ***Bc. Vladimír Antoš***
*Vedoucí práce:* *Ing. Roman Špánek, Ph.D.*

Liberec 2016

TECHNICKÁ UNIVERZITA V LIBERCI
**Fakulta mechatroniky, informatiky a mezioborových studií**
Akademický rok: **2015/2016**

# ZADÁNÍ DIPLOMOVÉ PRÁCE

(PROJEKTU, UMĚLECKÉHO DÍLA, UMĚLECKÉHO VÝKONU)

| | |
|---|---|
| Jméno a příjmení: | **Bc. Vladimír Antoš** |
| Osobní číslo: | **M14000150** |
| Studijní program: | **N2612 Elektrotechnika a informatika** |
| Studijní obor: | **Informační technologie** |
| Název tématu: | **Čistý kód: návrh míry pro výpočet toxicity kódu a její implementace** |
| Zadávající katedra: | **Ústav mechatroniky a technické informatiky** |

Zásady pro vypracování:

1. Student se seznámí s vlastnostmi a podmínkami, které musí splňovat čistý kód.
2. Student připraví rešeršní práci o existujících mírách toxicity kódu a jejich možnému využití pro automatické měření toxicity kódu.
3. Navrhne novou míru toxicity s přihlédnutím k požadavku na její snadnou a efektivní algoritmizaci a připraví její pilotní implementaci pro vybrané programovací jazyky.
4. Porovná navrženou míru s existujícími.

Rozsah grafických prací: **dle potřeby dokumentace**

Rozsah pracovní zprávy: **cca 40–50 stran**

Forma zpracování diplomové práce: **tištěná/elektronická**

Seznam odborné literatury:

[1] **Robert C. Martin: Clean Code: A Handbook of Agile Software Craftsmanship, Prentice Hall; 1 edition (August 11, 2008), ISBN-10: 0132350882**

Vedoucí diplomové práce: **Ing. Roman Špánek, Ph.D.**
Ústav mechatroniky a technické informatiky

Datum zadání diplomové práce: **10. října 2015**

Termín odevzdání diplomové práce: **16. května 2016**

prof. Ing. Václav Kopecký, CSc.
děkan

doc. Ing. Milan Kolář, CSc.
vedoucí ústavu

V Liberci dne 10. října 2015

# Prohlášení

Byl jsem seznámen s tím, že na mou diplomovou práci se plně vztahuje zákon č. 121/2000 Sb. o právu autorském, zejména §60 – školní dílo.

Beru na vědomí, že Technická univerzita v Liberci (TUL) nezasahuje do mých autorských práv užitím mé diplomové práce pro vnitřní potřebu TUL.

Užiji-li diplomovou práci nebo poskytnu-li licenci k jejímu využití, jsem si vědom povinnosti informovat o této skutečnosti TUL; v tomto případě má TUL právo ode mne požadovat úhradu nákladů, které vynaložila na vytvoření díla, až do jejich skutečné výše.

Diplomovou práci jsem vypracoval samostatně s použitím uvedené literatury a na základě konzultací s vedoucím diplomové práce a konzultantem.

Současně čestně prohlašuji, že tištěná verze práce se shoduje s elektronickou verzí, vloženou do IS STAG.

Datum:

Podpis:

Děkuji vedoucímu diplomové práce panu Ing. Romanu Špánkovi, Ph.D. za neocenitelné rady a pomoc při tvorbě mé diplomové práce. Dále bych chtěl poděkovat panu doc. Ing. Daliboru Frydrychovi, Ph.D. za velmi vstřícný přístup a cenné rady.

## Abstrakt

Diplomová práce se zabývá problematikou čistoty zdrojového kódu. Při vývoji aplikací je důležité psát nejen funkční, ale i čistý a znovupoužitelný kód. První část práce se zaměřuje na pravidla vývoje čistého kódu. Dále se zabývá metrikami pro určení čistoty kódu a nástroji, které jeho psaní usnadní. Druhá část práce představuje aplikaci implementující metriky na zdrojové soubory. Program porovnává naměřené výsledky se vzorovými daty a zjišťuje čistotu jednotlivých tříd. Výhoda zmíněného řešení spočívá v možnosti přizpůsobení se potřebám vývojářů a kontrole pouze těch metrik, jenž jsou pro ně důležité.

## Klíčová slova

Čistý kód, analýza zdrojového kódu, metriky čistoty kódu, cyklomatická složitost, graf toxicity.

## Abstract

The thesis deals with issues of purity of the source code. During the process of developing the applications it is very important to write not only just functional code but also the clean code and the reusable one. The first part concerns of the rules of development of well-arranged code. Furthermore, this part is being concerned of the metrics for determining the purity of code and tools that can make his writing easier. The second part of the thesis presents an application that implements the metrics of a source code. The application compares, as well, the results with the data and tries, simultaneously, to find out the purity of each category. The benefit of this solution consists in the possibility of adapting the needs of developers and, on the other side, to the control of those metrics which ones are most important for them, only.

## Keywords

Clean code, analyze of the source code, metrics of clean code, cyclomatic complexity, chart of toxicity

# OBSAH

**Seznam obrázků a tabulek** .................... **8**
**Seznam použitých zkratek** .................... **9**
**1. Úvod** .................... **10**
**2. Čistý kód** .................... **11**
2.1 Názvy .................... 11
2.2 Metody .................... 11
2.3 Komentáře .................... 12
2.4 Třídy .................... 13
2.5 Výjimky .................... 15
**3. Metriky pro určení čistoty kódu** .................... **17**
3.1 Počet řádků .................... 17
3.2 Metoda funkčních bodů .................... 18
3.3 Halsteadova metrika velikosti programu .................... 18
3.4 Cyklomatická složitost .................... 19
3.5 Zobrazení metrik, graf toxicity kódu .................... 20
**4. Nástroje pomáhající s psaním čistého kódu** .................... **22**
4.1 Vývojová prostředí .................... 22
4.2 Nástroje pro kontrolu čistoty kódu .................... 24
**5. Implementace metrik pro hodnocení toxicity kódu** .................... **26**
5.1 Obecné informace .................... 26
5.2 Stanovení metrik .................... 26
5.3 Kompilátor Roslyn .................... 29
5.4 Popis funkčnosti aplikace .................... 32
5.5 Struktura projektu .................... 33
5.6 Ukázka aplikace .................... 37
5.7 Výsledky analýzy .................... 37
5.8 Využití .................... 40
**6. Závěr** .................... **41**
**Citovaná literatura** .................... **42**
**Obsah přiloženého CD** .................... **43**

# SEZNAM OBRÁZKŮ A TABULEK

**Obrázek 1:** Příklad porušení principu jedné odpovědnosti ........................................ 13

**Obrázek 2:** Správné použití principu jedné odpovědnosti ........................................ 14

**Obrázek 3:** Příklad principu inverze závislostí ........................................................ 15

**Obrázek 4:** Příklad control flow grafu...................................................................... 19

**Obrázek 5:** Hledání chyb pomocí nástroje Inspector ................................................ 22

**Obrázek 6:** Nástroj Code Metrics v Microsoft Visual Studio .................................... 24

**Obrázek 7:** Ukázka podobnosti metod a její řešení.................................................... 28

**Obrázek 8:** Struktura kompilátoru Roslyn ................................................................ 30

**Obrázek 9:** Ukázka použití kompilátoru Roslyn – získání deklarací tříd .................... 32

**Obrázek 10:** Blokové schéma aplikace ..................................................................... 33

**Obrázek 11:** Jmenné prostory v projektu Cleaner...................................................... 34

**Obrázek 12:** Schéma struktury pro uložení informací ze zdrojového kódu ................ 35

**Obrázek 13:** Aplikace CleanCodeAnalyzer ............................................................... 37

**Obrázek 14:** Výsledky analýzy pro knihovnu Cleaner................................................ 38

**Obrázek 15:** Graf toxicity kódu pro knihovnu Cleaner............................................... 39

**Tabulka 1:** Kalibrační data pro jmenný prostor System .NET frameworku ................ 38

# SEZNAM POUŽITÝCH ZKRATEK

| | |
|---|---|
| SRP | Single Responsibility Principle, princip, který určuje, že třída má mít jen jednu odpovědnost |
| DIP | Dependency Inversion Principle, třídy na vyšších úrovních nesmí záviset na nižších |
| LOC | Lines Of Code, metrika počítající řádky kódu |
| LLOC | Logical Lines Of Code, metrika počítající logické řádky kódu |
| FP | Function Point, metrika, která odhaduje pracnost programu |
| WPF | Windows Presentation Foundation, technologie .NET pro tvorbu uživatelského rozhraní |

# 1. ÚVOD

Každý programátor musí umět nejenom nějaký programovací jazyk nebo používání algoritmů, ale také by měl znát pravidla tzv. čistého kódu. Čistý kód je takový, který neobsahuje žádné nadbytečné informace, nepotřebuje dodatečné vysvětlení a je přehledný a snadno čitelný. Kód je vizitkou každého programátora a proto by měl dbát na jeho čistotu. Pokud programátor pracuje sám a na malých projektech, které nikdo jiný nebude číst, není tolik nutné pravidla dodržovat. Při vývoji velkých systémů nebo při týmové práci je ale nutností, aby tým udržoval jednotný styl psaní.

Cílem mé diplomové práce bude blíže se seznámit s pravidly psaní čistého kódu a implementací míry toxicity kódu. V první části práce budu popisovat obecné principy i pravidla konkrétních programovacích jazyků a zaměřím se na již existující nástroje pro detekci toxicity kódu. V druhé části potom představím vlastní návrh systému, který bude určovat čistotu kódu.

# 2. ČISTÝ KÓD

Čistý kód je takový, který neobsahuje žádné nadbytečné informace a je snadno čitelný, přesná definice však neexistuje. Pro jeho psaní jsou pouze pravidla nebo doporučení, která by se měla dodržovat. Psaní čistého kódu přináší spoustu výhod. V čistém kódu se snadno orientuje, jednoduše a rychle se provádí dodatečné úpravy. Tato kapitola se bude zabývat obecnými pravidly psaní čistého kódu. Zaměřím se zde na několik hlavních pravidel, která poté budou použita v druhé části mé práce.

## 2.1 Názvy

V každém kódu se nachází velké množství jmen. Pojmenování mají proměnné, třídy, metody, argumenty i jmenné prostory. Je tedy důležité volit smysluplná jména. Jméno nám musí říci, co daná funkce nebo proměnná dělá, k čemu je určena i jak se použije. Názvy by se měli volit tak, aby k nim nebyl potřebný žádný další komentář. Tvorba smysluplných názvů klade nároky na dobré popisné schopnosti programátora. Jména, která tvoří jen jeden znak, jsou v kódu hůře čitelná a špatně se vyhledávají.

Za chybný název lze považovat i takový, který v sobě nese informaci o datovém typu (např. `userNameString`). Takové kódování zanáší do programu nadbytečné informace, protože vývojová prostředí sama napovídají datový typ. Navíc pokud by se změnil datový typ proměnné, musel by se měnit i její název. Existují ale případy, kdy je dobré k názvu přidávat nějakou informaci navíc. Například proměnné, které patří do adresy (`name`, `lastName`, `street`, `city`, `state`). V případě, že by se tyto proměnné používaly samostatně, nebude jasné, že se jedná o adresu. Vhodné je proto tuto informaci do názvu doplnit (`addrName`, `addrStreet`, `addState`). Lepším řešením je zabalit proměnné do objektu `Address`.

Názvy tříd obvykle obsahují podstatná jména. Metody se skládají ze slovesa nebo slovesného spojení (`LoadData`, `Save`). Název by měl vysvětlovat, co funkce provádí. Přístupové metody vlastností začínají předponou `get` nebo `set`. [1]

## 2.2 Metody

Metody a funkce jsou zapouzdřené části kódu, provádějící nějakou činnost. Metoda by měla měnit stav objektu, nebo o něm vracet nějakou informaci. Při jejich psaní by se mělo dbát na to, aby byly co nejkratší a hlavně aby prováděly pouze jedinou akci. Příliš dlouhé

metody vedou k nepřehlednosti kódu a není na první pohled patrné, co je úkolem metody. Pořadí metod ve třídě by mělo být takové, aby každá další metoda byla na vyšší úrovni abstrakce. Takový kód se snadněji čte od shora dolů.

Do každé metody může vstupovat několik argumentů. I v tomto případě platí, že argumentů by mělo být použito co nejméně. V ideálním případě by metoda měla obsahovat nejvýše jeden argument. Příkladem může být funkce zjišťující existenci souboru `boolean fileExists(string path)`. Čtenář takového kódu jasně ví, co daná funkce dělá a co od ní může očekávat jako návratovou hodnotu.

Použití více argumentů se většinou nevyhneme a ne vždy to musí být chybou. Záleží na konkrétním návrhu metody. Nepříliš dobře je například zvolený druhý argument metody `FileStream.Open(string path, FileMode mode) [2]`. Na základě tohoto argumentu metoda soubor otevře pro čtení, pokud neexistuje, vytvoří nový soubor, nebo stávající přepíše novým souborem. Není tedy na první pohled patrné, jaké operace tato metoda provádí. Přehlednější řešení je rozdělit všechny akce do samostatných metod.

V případě, že metoda potřebuje více argumentů, je někdy lepší, všechny zabalit do nějakého objektu a metodě jej předávat jako jeden parametr.

## 2.3 Komentáře

Zdrojový kód bychom měli psát tak, aby nepotřeboval žádný komentář. Pokud musíme k nějakému kódu přidat komentář, znamená to, že kód je špatně pochopitelný. Přidáním komentáře se situace nezlepší, naopak přibude do kódu nadbytečná informace, která ho ještě více znepřehlední. Mnohem lepší je se tedy zamyslet, jestli by kód nešel napsat lépe. Často musíme kód různě přesouvat a upravovat, ale málokdy upravíme i komentáře a ty jsou tedy neaktuální a nepřesné. Někdy stačí pouze vymyslet vhodnější název, který řekne více, než dlouhý popis.

Vedle komentářů popisujících část kódu jsou ale komentáře, které jsou nutné a užitečné. Například informace o autorských právech na začátku skriptu. Nepíše se zde ale celé znění licence, pouze odkazy na externí dokumenty. Vhodné komentáře mohou být například u matematických výrazů, z jejichž kódu nemusí být na první pohled patrné, jak výraz vypadá ve skutečnosti. Velmi užitečným nástrojem jsou komentáře, generující dokumentaci (JavaDoc, PHPDoc). Tyto komentáře vysvětlují, co metoda provádí, jaké

má parametry a návratovou hodnotu. V jazycích, které nepoužívají datové typy je to jediný způsob, jak může vývojové prostředí zjistit, jakých hodnot může parametr metody nabývat. Proto by zde měly být uvedeny vždy alespoň datové typy argumentů a návratové hodnoty. [1]

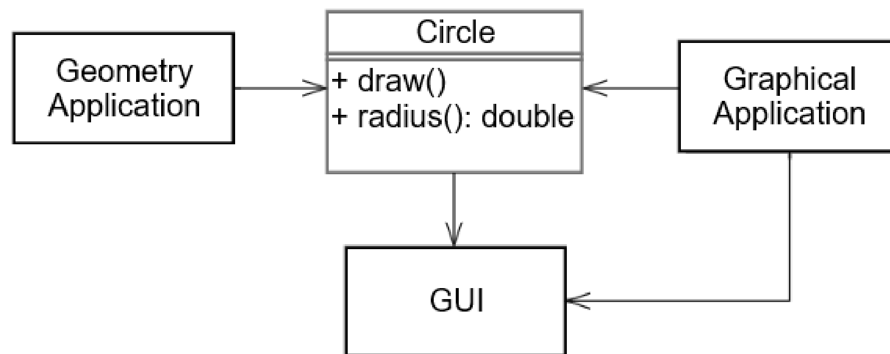
## 2.4 Třídy

Každá třída by měla začínat seznamem proměnných. Na začátku třídy mají být umístěné konstanty, poté statické proměnné a nakonec privátní členy. Za nimi by měl následovat konstruktor, poté veřejné a nakonec soukromé metody.

Stejně jako u metod i u tříd platí, že by měly být krátké a měly by zodpovídat pouze za jednu činnost. V případě, že má třída příliš mnoho metod, určitě půjde rozdělit do více tříd, popřípadě využít dědičnost.

### 2.4.1 Princip jedné odpovědnosti

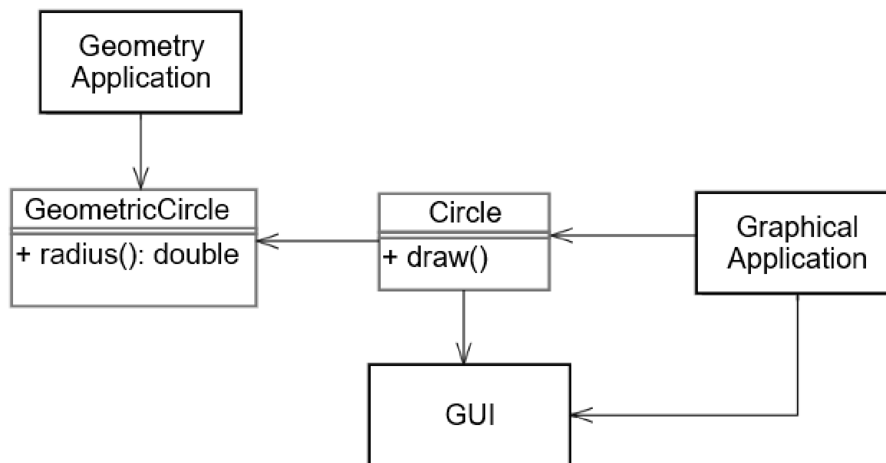
Princip jedné odpovědnosti SRP (Single-Responsibility principle) říká, každý objekt by měl být zodpovědný pouze za jednu činnost. Pokud má třída více odpovědností, může se stát, že při změně jedné dojde k ovlivnění těch ostatních. Tento princip vysvětluje obrázek 1.




**Obrázek 1:** Příklad porušení principu jedné odpovědnosti (převzato z [3])

Na schématu je třída `Circle`, kterou využívají dvě rozdílné aplikace. První programji využívá pouze pro geometrické výpočty, ale nikdy kruh nechce vykreslovat. Druhá aplikace bude využívat výpočty, ale zároveň chce kruh vykreslit. Třída `Circle` má na starosti dvě závislosti, stará se o vykreslení kruhu i o výpočty. V případě, že by první aplikace třídu `Circle` změnila, mohlo by dojít k ovlivnění vykreslení kruhu.

Řešení je nastíněno na obrázku 2. Původní třída se rozdělila na dvě, `Circle` a `GeometricCircle`. Ke třídě `GeometricCircle` má přístup pouze geometrická aplikace. Třídu `Circle` využívá aplikace pro vykreslování. Aby měla přístup i k výpočtům, má `Circle` referenci na třídu `GeometricCircle`. [3]




**Obrázek 2:** Správné použití principu jedné odpovědnosti (převzato z [3])

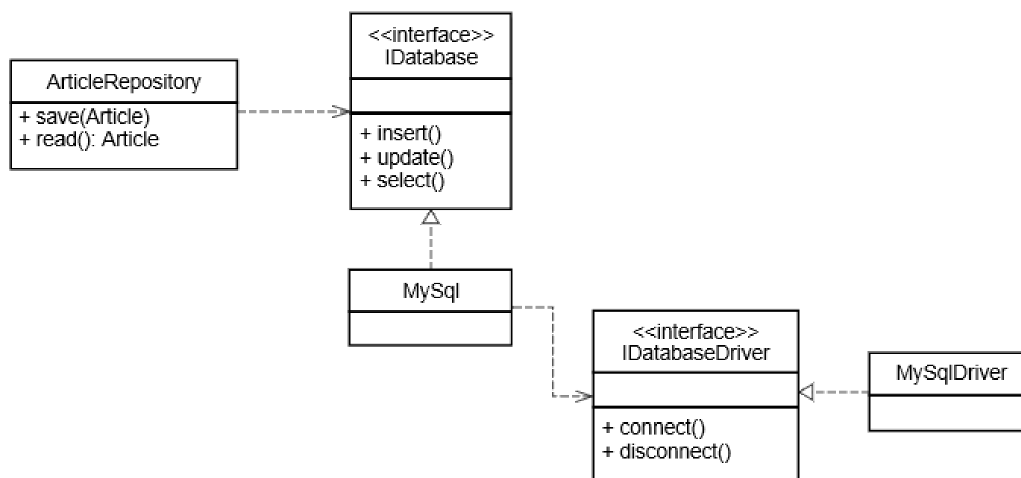
### 2.4.2 Princip otevřenosti a uzavřenosti

Pravidlo otevřenosti a uzavřenosti říká, že třídy (ale i metody a funkce) mají být otevřené pro rozšiřování, ale uzavřené pro úpravy. To znamená, že třídu můžeme rozšiřovat o další funkčnost, ale nemáme právo měnit její zdrojový kód. Rozšíření funkčnosti nelze dosáhnout změnou samotné třídy, ale použitím abstrakce. [3]

### 2.4.3 Princip inverze závislosti

Princip inverze závislosti DIP (Dependency inversion principle) je nástroj, který nám pomáhá k tomu, aby byl systém flexibilnější a znovupoužitelnější. DIP říká, že třída na vyšší úrovni by neměla být závislá na třídě nižší úrovně, ale obě mají být závislé na abstrakci, ta nesmí obsahovat žádnou implementaci. V praxi se tento princip realizuje pomocí rozhraní. [3]

Na obrázku 3 je příklad implementace DIP. Obsahuje knihovnu přistupující k databázi a uživatelskou třídu `ArticleRepository`, která k databázi přistupuje přes rozhraní `IDatabase`. Nezná tedy strukturu nižších tříd, a pokud dojde k jejich změně, třídy `ArticleRepository` se nebude týkat.




**Obrázek 3:** Příklad principu inverze závislostí

## 2.5 Výjimky

Do oblasti čistého kódu patří i umění správně zpracovávat chyby. V moderních programovacích jazycích se místo chybových hlášení používají výjimky. Kód s výjimkami je mnohem čistí, protože se nemusí testovat každý chybový stav zvlášť. V nějakém místě vznikne výjimka, která je poté zachycena v `catch` bloku a tam ošetřena. Aby bylo možné zpětně určit příčinu výjimky a místo jejího vzniku, musí výjimka nést dostatek informací. Chybové hlášení výjimky má obsahovat popis operace, která skončila chybou, případně důvod proč k tomu došlo. Pro určení místa vzniku se může využít trasování zásobníku výjimky.

Jazyk Java má k dispozici tzv. kontrolované výjimky (checked exception). Všechny možné výjimky jsou uvedeny v signatuře metody a musí být ve volající metodě buď zpracovány v bloku `catch`, nebo dále propagovány pomocí `throws`. Kontrola ošetření výjimek se provádí při kompilaci kódu. Používání kontrolovaných výjimek vede k porušení principu otevřenosti a uzavřenosti. Pokud k výjimce dojde na nižší úrovni aplikace a blok catch je umístěn o několik úrovní výše, musí mít všechny metody v signatuře tuto výjimku uvedenou. Dochází tak k tomu, že metoda na nižší úrovni dokáže ovlivňovat metody vyšších úrovní. Tyto výjimky by se měli využívat minimálně a pouze v případech, kdy není nutné výjimku zachytit.

Speciálním případem práce s chybami je, že metoda v případě neúspěšné akce vrací hodnotu `null`. Tento přístup je ale chybný, vede k zvyšování složitosti kódu. Musí

se u každé takové metody testovat, jestli vrátila null. Pokud by se na test zapomnělo a v metodě došlo k chybě, nikdy bychom se o chybě nedozvěděli, na rozdíl od výjimky. [3]

# 3. METRIKY PRO URČENÍ ČISTOTY KÓDU

Předchozí kapitola pojednávala o některých pravidlech pro psaní čistého kódu. Znalost těchto pravidel je předpokladem k psaní přehledného kódu. V této kapitole budou představeny metriky, díky kterým můžeme určit, jak je daný kód kvalitní.

Metrika je formulována jako hodnotící funkce a slouží k porovnání naměřených hodnot. Měření se provádí proto, abychom mohli určit kvalitu nových nástrojů a metod, produktivitu vývojářů nebo vytvořili odhad požadavků na nové nástroje. Metriky se rozdělují na: [4]

- Přímé – velikost kódu (LOC), počet chyb, rychlost,
- Nepřímé – složitost, funkčnost, spolehlivost, udržovatelnost.

## 3.1 Počet řádků

Mezi nejjednodušší metriky patří počítání řádků (LOC). Tato metrika slouží k měření objemu kódu a může být použita pro získání počtu řádků ve třídě, v metodě nebo v souboru či v celém systému. LOC je metrika, která nijak objektivně nehodnotí kvalitu kódu. Nedozvíme se z nic o tom, jak dobře je kód napsaný. Může nám pomoci detekovat příliš složité části kódu. Existuje několik alternativ jak počítat řádky kódu: [5]

- Počítání fyzických řádků – nejjednodušší způsob výpočtu, započítává i prázdné řádky a komentáře,
- Logické řádky (LLINES) – spojení několika fyzických řádků do jednoho logického celku. Sčítá logické celky bez ohledu na to, kolik obsahují fyzických řádků,
- Logické řádky kódu (LLOC) – sčítají se pouze řádky kódu, nezapočítává prázdné řádky, komentáře, definice proměnných import balíčků.

Další varianty této metriky už vyžadují složitější analýzu. Můžeme počítat například pouze proveditelné řádky. K nim mohou být přičítány i definice datových typů nebo ošetření výjimek. [4]

Jak dlouhé by tedy měli metody nebo třídy být? Tímto problémem se zabývá Martin Lippert a Stefan Roock ve své knize [6]. Definovali tzv. pravidlo třiceti, které říká, že metoda by měla mít průměrně 30 řádků kódu (bez komentářů a prázdných řádků).

Třída by pak měla obsahovat maximálně 30 metod, což je 900 řádků kódu. Jmenný prostor nebo package nemá obsahovat více než 30 tříd, dohromady tedy 27000 řádků kódu. Jednotlivé části systému pak mají obsahovat 30 jmenných prostorů nebo balíčků a konečně celý systém obsahuje maximálně 30 částí. Dohromady by tedy systém měl obsahovat maximálně 27 000 tříd a 24,3 milionu řádků. Toto pravidlo, je pouze orientační a rozhodně by se nemělo brát doslovně. Někdy by rozdělení dlouhé, ale správně napsané metody kód znepřehlednilo mnohem více.

## 3.2 Metoda funkčních bodů

Metoda funkčních bod (FP) se využívá k odhadu pracnosti vyvíjeného programu. Pracnost se stanovuje z určení složitosti, protože se vychází z úvahy, že čím je systém složitější, tím je také pracnější. Při určení pracnosti se postupuje v několika krocích. Nejprve se systém rozdělí do kategorizovaných prvků, funkčních bodů (např. externí vstupy, výstupy atd.). Prvkům je přiřazena složitost vynásobená pevně stanoveným váhovým faktorem. Určení složitosti se provádí pomocí tabulek s vlastnostmi prvků. Sečtením hodnot složitosti se získá neupravený počet funkčních bodů. Po zahrnutí dalších faktorů ovlivňujících složitost, které nejsou ještě obsaženy v prvcích, získáme upravený počet funkčních bodů. Poté se stanoví celková doba trvání realizace programu v člověkodnech pomocí tabulek. [4]

## 3.3 Halsteadova metrika velikosti programu

Tato metrika určuje velikost programu ze součtů operátorů a operandů. Je založena na předpokladu, že program se skládá z konečného počtu programových jednotek. Těm se říká tokeny. Operátory jsou například aritmetické operace, indexy, závorky nebo středníky. Operandy jsou všechny proměnné. Pro určení metriky se používají následující prvky:

- $n_1$ – počet unikátních operátorů v implementaci,
- $n_2$ – počet unikátních operátorů v implementaci,
- $N_1$ – celkový počet operátorů,
- $N_2$ – celkový počet operandů.

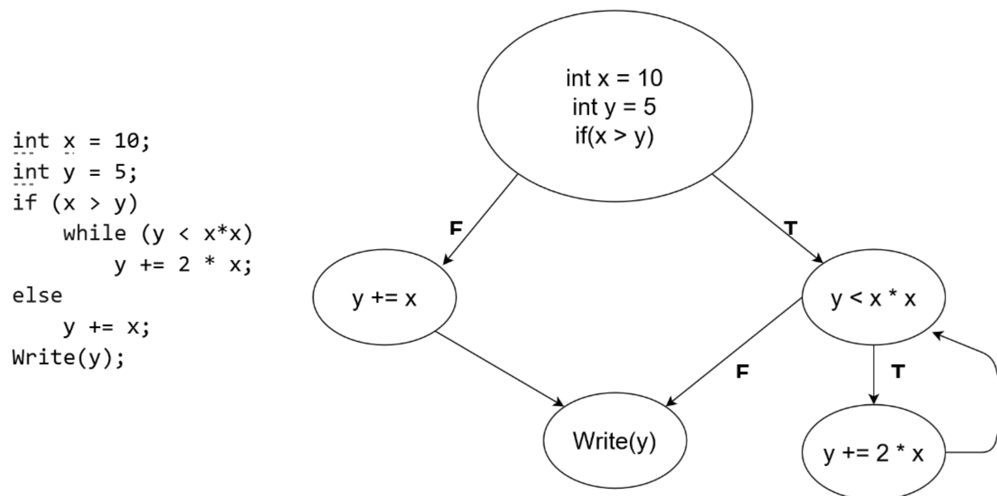
Z těchto informací můžeme vypočítat například délku programu $N = N_1 + N_2$, odhad délky programu $EN = n_1 \log_2 n_1 + n_2 \log_2 n_2$, koeficient čistoty programování

(pure ratio) $PR = \frac{EN}{N}$. Dále můžeme určit třeba počet bitů, které jsou potřeba pro rozhodnutí při volbě každé z $n$ položek programového slovníku $V = \mathrm{N} * \log_2 n$.

Výhodou této metriky je, že nevyžaduje složitější analýzu struktury programu a je nezávislá na programovacím jazyce. Dokáže také měřit celkovou kvalitu programu a předpovídat námahu na údržbu. Nevýhodou je, že abychom mohli provést odhad délky EN, musí být program dokončen. Výhodnější je tedy používat jako metriku cyklomatickou složitost, která pracuje již s návrhem aplikace.

## 3.4 Cyklomatická složitost

Cyklomatická složitost je jednou z nejstarších metrik. Byla vymyšlena již v roce 1976 Thomasem J. McCabem. Je to číselné vyjádření složitosti programu. Určuje množství nezávislých cest v metodě nebo funkci. Výpočet se provádí pomocí control flow grafu. Je to orientovaný graf, který se skládá z uzlů a hran. Uzly reprezentují bloky kódu a hrany přechody mezi bloky. Uzel, ze kterého hrany pouze vystupují, reprezentuje vstupní blok, naopak pokud hrany do uzlu pouze vstupují, jedná se o výstupní blok.




Obrázek 4: Příklad control flow grafu

Na obrázku je příklad control flow grafu. Z něj je možné určit cyklomatickou složitost, která je popsána vztahem: [7]

$$V(G) = E - N + 2$$

E … počet hran grafu

N … počet uzlů grafu

Výpočet cyklomatické složitosti pro Obrázek 6 bude vypadat takto: $V(G) = 6 - 5 + 2$.

Cyklomatická složitost představuje minimální počet cest, ze všech lineárních kombinací cest v daném modulu. Graf je možné popsat i pomocí vektorů. Každá z cest v grafu má přiřazený vektor s prvky, které odpovídají hranám cesty. Hodnota každého prvku ve vektoru odpovídá součtu průchodů přes všechny hrany v cestě. Vektory je možno přepsat do matice, kde řádky odpovídají cestám a sloupce hranám. Lineární algebra říká, že hodnost matice je menší nebo rovna počtu sloupců. V tomto případě to znamená, že počet cest bude vždy menší než počet hran. Hodnost matice je rovna cyklomatické složitosti. [8]

Cyklomatická složitost je číslo, které čím je větší, tím složitější je program. Proto je při vývoji softwaru dobré stanovit si, jaké největší hodnoty může nabývat. Thomas McCabe původně stanovil hodnotu 10. Určil ji pro zastaralé programovací jazyky a pro moderní jazyky je nízká. Například navrhoval, aby se do výpočtu nezahrnoval velmi oblíbený příkaz `switch`. Dnes se naopak doporučuje tento příkaz nepoužívat, nebo pouze minimálně. Pro rozsáhlé projekty, na kterých pracují zkušení vývojáři a dodržují se pravidla OOP, není problém vyšší cyklomatická složitost. [8]

## 3.5 Zobrazení metrik, graf toxicity kódu

Po analýze zdrojového kódu je vhodné nějak rozumně zobrazit výsledky. Může to být buď do tabulky, kde v řádcích jsou zkoumané třídy a ve sloupcích hodnoty metrik. Přehlednější zobrazení je pomocí grafu toxicity. Jedná se o sloupcový graf, ve kterém každý sloupec představuje jednu třídu. Výška sloupce pak znázorňuje míru toxicity této třídy. Míra toxicity je složena z několika barevně odlišených metrik. Sloupce v grafu jsou seřazeny od nejvyššího po nejnižší. Čím vyšší má sloupec hodnotu, tím toxičtější je třída. Díky tomuto grafu vidíme na první pohled problematické třídy, a protože jednotlivé

metriky mají různé barvy, snadno zjistíme podle převládající barvy v grafu, na jaké problémy bychom se měli zaměřit. Třídy, které mají nulovou míru toxicity, se do grafu nezobrazují.

# 4. NÁSTROJE POMÁHAJÍCÍ S PSANÍM ČISTÉHO KÓDU

S psaním čistého kódu nám mohou pomoci různé nástroje a vývojová prostředí, která umí kód formátovat a kontrolují, jestli je napsán podle specifikace jazyka. Právě o těchto nástrojích tato kapitola pojednává. Zaměřuje se na možnosti kontroly toxicity kódu ve vývojových prostředích a vybraných nástrojích.

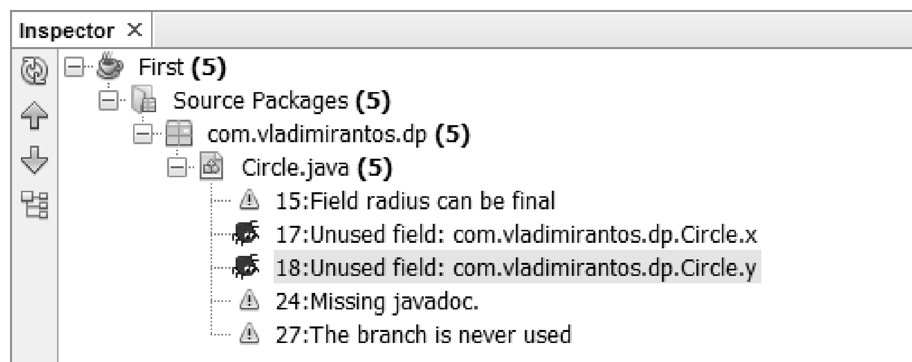
## 4.1 Vývojová prostředí

Vývojová prostředí již sama poskytují dobrý základ psaní čistého kódu. Umí například kód formátovat podle předepsané specifikace daného jazyka. Také dovedou v kódu najít redundantní nebo nepoužívané či nedosažitelné části a například také vypočítávají různé metriky kódu. Výhodou vývojových prostředí je, že analýzu provádí v reálném čase a ihned tak upozorňují na případné problémy.

### 4.1.1 Netbeans IDE

Netbeans IDE je prostředí distribuované zdarma pod licencí GPL2. Vzniklo původně v ČR jako studentský projekt. Později byl Netbeans koupen společností Sun Microsystems a nyní patří společnosti Oracle corporations. Toto prostředí je primárně určeno pro Javu, ale díky pluginům lze použít pro C/C++, PHP a další. [9]

V tomto vývojovém prostředí zajišťuje kontrolu čistoty kódu nástroj Inspector a jeho plugin FindBugs. Dokáže nalézt v kódu širokou škálu různých problémů. Bohužel neumí zobrazit souhrnné výsledky analýzy. Po jeho spuštění se objeví okno s výběrem kategorie, která se má kontrolovat. Ve výchozím stavu se zaměřuje pouze několik málo kategorií (viz Obrázek 4).




**Obrázek 5:** Hledání chyb pomocí nástroje Inspector

Inspector umí i základní kontrolu názvů. Například u metod zjistí, jestli jsou psané stylem `cammelCase`. Nerozpoznává ale, že název proměnné je jednoznakový, nebo že obsahuje například nadbytečnou informaci o datovém typu. Užitečnou vlastní nástroje Inspector je, že dokáže napovídat i správné modifikátory.

### 4.1.2 IntelliJ IDEA

IntelliJ IDEA je komerční vývojové prostředí vyvíjené společností Jetbrains. Určené je primárně pro jazyk Java, ale rozumí i dalším jazykům jako je Clojure, Go, Groovy, JavaScript nebo XML. IntelliJ IDEA obsahuje celou řadu funkcí pomáhajících s psaním čistého kódu.

Součástí jsou nástroje Code Cleanup a Code Analysis. Jejich úkolem je nejenom detekce chyb při kompilaci, ale starají se také i o formální chyby při psaní. Například umí odhalit nedosažitelný kód, neinicializované nebo nepoužívané proměnné, analyzuje závislosti nebo hledá duplicity v kódu. Dokáže také upozornit na podmínky, které se vždy vyhodnotí záporně. Nástroj Code Analysis také dokáže určit, jestli je možné převést abstraktní třídu na rozhraní. Musí pro to být splněno několik podmínek. Abstraktní třída nesmí mít žádné datové členy, implementace metod a nesmí být potomkem jiné metody.

Výhodou těchto nástrojů jsou široké možnosti konfigurace. Je možné vytvářet vlastní profily se sadami kontrol a nastavovat míru závažnosti každé sledované kontrole. Dále je možné nastavit, jestli se budou prohledávat všechny soubory v projektu, všechny otevřené soubory nebo jen části kódu.

### 4.1.3 Microsoft Visual Studio

Vývojové prostředí od společnosti Microsoft je určené převážně pro vývoj aplikací v jazyce C#, VB.NET a C/C++. Jako všechny ostatní vývojová prostředí obsahuje celou řadu nástrojů pro analýzu, kontrolu a formátování kódu. Podobné nástroje jsem popisoval u předchozích vývojových prostředí, proto zde zmíním pouze nástroj pro výpočet metrik kódu. Vypočítává několik různých metrik jako je cyklomatická složitost, index udržovatelnosti kódu, hloubku dědičnosti, počet řádků nebo počet závislostí tříd. Výpočet provádí pro jednotlivé metody a třídy. Umí ale ukázat výsledné hodnoty pro soubory, jmenné prostory i pro celý projekt.

| Hierarchy | Maintainability Index | Cyclomatic Compl... | Depth of Inheritance | Class Coupling |
|---|---|---|---|---|
| Cleaner (Debug) | 88 | 519 | 3 | 143 |
| Cleaner.Utils.Extension | 88 | 11 | 1 | 7 |
| Cleaner.Analyzer | 85 | 16 | 1 | 25 |
| IAnalyzer | 100 | 2 | 0 | 2 |
| CcaAnalyzer | 82 | 6 | 1 | 8 |
| ClassAnalyzer | 73 | 8 | 1 | 21 |
| Analyze() : Clas | 75 | 1 | | 12 |
| ClassAnalyzer(C | 87 | 1 | | 1 |
| MethodStatistic | 70 | 2 | | 10 |
| PropertyStatisti | 71 | 2 | | 10 |
| VariableStatisti | 71 | 2 | | 8 |
| Cleaner.Analyzer.Statis | 94 | 41 | 2 | 10 |

**Obrázek 6:** Nástroj Code Metrics v Microsoft Visual Studio

## 4.2 Nástroje pro kontrolu čistoty kódu

Vedle vývojových prostřední pomáhají s psaním čistého kódu i další nástroje. Mohou být ve formě pluginů nebo jako samostatné aplikace. Tato část se zaměřuje pouze na nástroje pro .NET a jazyk C#.

### 4.2.1 JetBrains ReSharper

Plugin do Visual Studia JetBrains ReSharper je velmi užitečný nástroj pomáhající s psaním přehledného kódu. Mezi jeho přednosti patří výborný refaktoring kódu, rychlá navigace a pokročilé napovídání při psaní. ReSharper pomáhá snižovat složitost kódu. Pokud máme v kódu nějakou složitou rovnici, může jí automaticky zabalit do externí metody. Složitý kód dokáže přepisovat do jednodušší podoby, typicky u dotazů v jazyce LINQ. Užitečné je také inteligentní napovídání při pojmenovávání. U proměnných a vlastností napovídá, jak by se mohly jmenovat v závislosti na jejich datovém typu, popřípadě jejich změnu na implicitní datový typ. U rozhraní zase upozorňuje, že jejich název nemá prefix I. Další užitečnou funkcí ReSharperu je přepisování vlastností na automatické. V C# verzi 6.0 napovídá i přepsání na tzv. lamba metodu nebo vlastnost. Spousta programátorů jistě ocení i napovídání vhodných modifikátorů, mazání redundantního či nedosažitelného kódu.

### 4.2.2 CodeMaid

Open source plugin do Visual Studia. CodeMaid nabízí podobné funkce jako ReSharper. Umožňuje čistit zdrojový kód automaticky, při uložení souboru nebo na vyžádání. Lze zvolit, jestli se bude provádět kontrola pouze souboru nebo celého projektu. CodeMaid umí odstranit nepoužívané části kódu, doplnit přístupové modifikátory, odstraňovat prázdné řádky či regiony. Dále vypočítává k metodám jejich cyklomatickou složitost. Zajímavou funkcí je jistě setřídění členů třídy.

# 5. IMPLEMENTACE METRIK PRO HODNOCENÍ TOXICITY KÓDU

Předchozí kapitola představila nástroje, které usnadňují vývoj přehledného kódu. Nyní přichází čas naimplementovat vlastní řešení. Výše zmíněné nástroje a vývojová prostředí prováděly analýzu kódu v reálném čase a uměly i samy kód opravovat. Moje aplikace bude provádět analýzu kódu až po vytvoření. Užitečnou funkcí bude zobrazování grafu toxicity kódu, ze kterého je ihned patrné, které třídy jsou nejproblematičtější a na jaké problémy je potřeba se zaměřit.

Původním cílem bylo navrhnout metriku, která by dokázala určit, jak je kód toxický. Jako ideální se však jeví kombinace různých metrik, které budou zobrazeny odděleně.

## 5.1 Obecné informace

Aplikaci jsem nazval CleanCodeAnalyzer (zkratka CCA). Pro implementaci aplikace jsem zvolil jazyk C# ve verzi 6.0 a analyzovat budu kód rovněž v tomto jazyce, protože se zpracováním zdrojového kódu může pomoci kompilátor Roslyn, který má API pro analýzu.

Aplikace je rozdělena na dvě části. Jádro, které zpracovává a analyzuje zdrojový kód, je umístěné v samostatné DLL knihovně. Druhou částí je grafická aplikace vytvořená pomocí technologie WPF. Výhodou tohoto rozdělení je, že logika je nezávislá na technologii grafické aplikace a může být použita například pro webovou aplikaci či plugin do Visual Studia.

## 5.2 Stanovení metrik

Prvním krokem implementace je určení klasifikačních pravidel a metrik, které kontrolují zdrojový kód. Zaměřuji se na správné pojmenovávání tříd a jejích členů, na správnou strukturu třídy, podobnost metod. Pro určení složitosti kódu jsem zvolil cyklomatickou složitost.

Problémem u některých metrik je stanovení hranice, kdy je kód ještě čistý a kdy už ne. Jak tedy rozhodnout, jestli naměřená hodnota odpovídá správně napsanému kódu? Například u metriky LOC se uvádí, že metoda by měla být co nejkratší. Tuto metriku

nelze přesně kvantifikovat a stanovení hranice mezi přehledným a toxickým kódem je tak velmi obtížné.

Řešení jsem nalezl v použití vzorových dat. Aplikaci je tedy nejprve předán projekt, o kterém víme, jak čistě je napsaný. Tento projekt je analyzován a jsou vypočítány metriky pro jednotlivé třídy. Naměřené hodnoty se poté zprůměrují a uloží do konfiguračního souboru. Poté již může být nahrán zkoumaný projekt a jeho naměřené hodnoty se porovnají se vzorovými daty. Toto řešení nabízí několik výhod. První z nich je, že je možné rozhodnout, které metriky vykazují znaky čistého a které toxického kódu. Další výhodou je, že aplikace se bude chovat rozdílně v závislosti na vložených vzorových datech. Každý programátor chápe pod pojmem čistý kód něco jiného a obecné metriky nebo specifikace jazyka jsou tak jen doporučením, jak by se mohl kód psát. Programátoři si tak mohou aplikaci individualizovat podle vzorových dat a přizpůsobit její chování svým potřebám a návykům při psaní. Nevýhodou tohoto řešení je, že nemusí vždy vést ke klasifikaci skutečně čistého kódu.

Bližší popis jednotlivých metrik je v následujících podkapitolách. Vycházím z obecných pravidel pro psaní čistého kódu popsaných ve druhé kapitole i ze specifikace jazyka C# [10].

### 5.2.1 Kontrola názvů

Jak už jsem popisoval ve druhé kapitole, názvy mají být co nejkratší a nejvýstižnější. Aplikace kontroluje, jestli názvy metod a tříd začínají velkým písmenem. U proměnných kontroluje malé písmeno na začátku a také hledá, jestli neobsahují informaci o datovém typu. U všech názvů je zaznamenávána i jejich délka.

Třídě je za každý chybný název přičten bod. Při kalibraci se výsledky zprůměrují a vznikne několik hodnot, které udávají, kolik může být maximálně chyb ve třídě.

### 5.2.2 Délka kódu

Do této kategorie jsem zahrnul více metrik. Počet řádků kódu je rozdělen do tří částí – vypočítávám počet řádků kódu, počet řádků komentářů a počet prázdných řádků. Hodnoty každé z dílčích metrik se zprůměrují a výsledná hodnota udává, maximální počet řádků metody, maximální počet vstupních argumentů atd.

Následující seznam obsahuje všechny metriky z této kategorie:

- Řádky kódu
    - Počet komentářů – nezapočítávají se komentáře určené pro dokumentaci,
    - Počet prázdných řádků,
- Počet členů třídy – datové členy, vlastnosti, metody,
- Počet vstupních argumentů metody,
- Počet tříd v souboru, jmenném prostoru,
- Délka názvů identifikátorů.

### 5.2.3 Podobnost metod

Podobnost metod zkoumám na základě jejich argumentů. Dvě metody můžeme prohlásit za podobné, pokud mají stejný počet argumentů a na stejných pozicích mají argumenty stejného datového typu. Následující obrázek ukazuje, jak je možné tuto situaci vyřešit.

```
class Calculator
{

    public int Addition(int x, int y)
    {
        return x + y;
    }

    public int Subtraction(int x, int y)
    {
        return x - y;
    }
}
```

```
class Calculator
{
    private readonly int _x;
    private readonly int _y;

    public Calculator(int x, int y)
    {
        _x = x;
        _y = y;
    }

    public int Addition()
    {
        return _x + _y;
    }

    public int Subtraction()
    {
        return _x - _y;
    }
}
```

**Obrázek 7:** Ukázka podobnosti metod a její řešení

Obrázek 7 ukazuje modelový příklad podobnosti metod. Třída vlevo obsahuje dvě na první pohled stejné metody. Mají stejný počet argumentů, které jsou stejného datového typu. Tuto třídu lze nahradit třídou vpravo, kde jsou argumenty přesunuty na jedno místo. Tato metrika za každé dvě podobné metody udělí třídě bod. Výsledky všech tříd se zprůměrují a výsledná hodnota udává, kolik může být maximálně podobných metod v projektu. Nejsou sem započítávány přetěžované metody. Později bude tato metrika rozšířena i o zjišťování podobnosti na základě kódu.

### 5.2.4 Cyklomatická složitost

Cyklomatická složitost udává množství nezávislých cest v programu. Číselně vyjadřuje složitost algoritmu. Aplikace ji vypočítává pro každou metodu. Výpočet probíhá tak, že kód metody je rozdělen na jednotlivé řádky. Sečtou se počty příkazů větvení (`if`, `for`, `while`…) a příkazy přiřazení. Z toho lze zjistit počet uzlů a hran v control flow grafu. Dosazením do vzorce je získána cyklomatickou složitost. Výsledky všech jsou poté zprůměrovány a získám číslo reprezentující maximální možnou složitost metody.

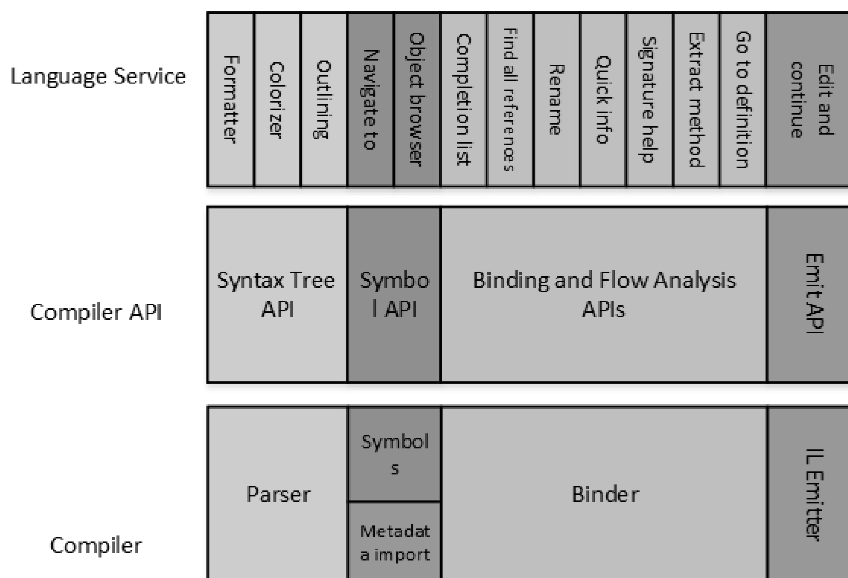
## 5.3 Kompilátor Roslyn

Se zpracováním zdrojového kódu mi významně pomáhá open-source kompilátor Roslyn (jmenný prostor `Microsoft.CodeAnalysis`). V této kapitole se tedy budu věnovat popisu toho, jak tento kompilátor jazyka C# funguje.

Klasické kompilátory fungují jako černé skříňky. Přijímají zdrojový kód, provádí s ním nějaké operace a výstupem je nějaký strojově zpracovatelný kód. Nevýhodou těchto kompilátorů je, že ihned po vygenerování výstupních dat zapomenou všechny mezikroky a neuchovávají nikdy informace, jako jsou syntaktické stromy. Moderní vývojová prostředí ale mají integrované různé nástroje jako je IntelliSense, refaktoring, vyhledávání a další. Tyto nástroje se neobejdou bez analýzy zdrojového kódu a je nutností, aby přistupovaly až ke kompilátoru. Kompilátor Roslyn napsaný v C# .NET je platforma, která poskytuje několik vrstev API pro přístup k nástrojům pro analýzu zdrojového kódu.

### 5.3.1 Struktura kompilátoru

Na obrázku 8 je schéma celé platformy Roslyn. Na nejnižší vrstvě je samotný kompilátor. Jeho činnost vychází z principu tzv. pipeliningu. Zdrojový kód je nejprve zpracován parserem. Kód je tokenizován a uložen do syntaktického stromu. Ve druhém kroku se zpracovaná data rozdělí na jednotlivé symboly, v další fázi se všechny identifikátory v kódu nahradí odpovídajícími symboly. V posledním kroku se vygeneruje CIL byte kód a dojde ke kompletaci výsledného sestavení.




**Obrázek 8:** Struktura kompilátoru Roslyn (převzato z [11])

Nad každou z částí kompilátoru je vytvořená objektová struktura, která umožňuje přístup k informacím konkrétní části. Nad parserem je API, které poskytuje přístup k syntaktickému stromu, z druhé části kompilátoru je možné získat hierarchickou tabulku symbolů. Třetí část vrací výsledky sémantické analýzy a poslední část má API pro práci s IL byte kódem. [11]

### 5.3.2 Struktura API

Aplikační rozhraní je složené ze dvou částí – API kompilátoru a Workspaces API. Objektový model první části API odpovídá jednotlivým částem kompilátoru. Obsahuje nástroje jak pro syntaktickou, tak pro sémantickou analýzu. Protože kompilátor Roslyn je určený nejen pro jazyk C# ale i VB.NET existují dvě API kompilátoru s podobnou strukturou. Tato část API je nezávislá na Visual Studiu. Aplikační rozhraní kompilátoru obsahuje dvě komponenty: [11]

- Skriptovací API – určené pro vytváření skriptů pro hromadné spouštění částí kódu,
- Diagnostické API – zahrnuje syntaktickou a sémantickou analýzu. Toto API je možné rozšiřovat o uživatelem definované analyzátory, které se mohou využít při sestavení. Právě toto API využívají nástroje jako ReSharper, StyleCop nebo FxCop. Tato část je zodpovědná za podtrhování chyb v editor, navrhuje opravy a stará se o používání breakpointů.

Druhou částí celého aplikačního rozhraní je Workspaces API. Je využíváno Visual Studiem, protože nabízí funkce jako je refaktorování, doplňování, formátování nebo vyhledávání v kódu. Toto API má přímý přístup k rozhraní kompilátoru, který mu vykonává analýzu kódu. [11]

### 5.3.3 Syntaktická analýza

Ve své aplikaci CleanCodeAnalyzer využívám z kompilátoru Roslyn právě část syntaktické analýzy, proto se v této podkapitole budu této části kompilátor zabývat a popíši, jak vypadají syntaktické stromy. Využití syntaktické analýzy se v některých případech ukázalo nevýhodné, neboť některé informace ze zdrojového kódu se snadněji získávají z části API pro sémantickou analýzu.

Při syntaktické analýze vzniká tzv. syntaktický strom. Lze k němu přistupovat přes API kompilátoru a nachází využití při analýze, refaktorování, přesunech a dalších operacích s kódem. Provádět úpravy pomocí stromu je jednoduší než upravovat text.

Syntaktický strom musí přesně odpovídat zdrojovému kódu. Obsahuje všechny gramatické a lexikální prvky, mezery, komentáře i direktivy preprocesoru. Ve stromu jsou také obsaženy chyby ze zdrojového kódu. Strom umožňuje také přesně rekonstruovat zpět zdrojový kód. Z každého libovolného uzlu můžeme získat textovou podobu jeho podstromu. Vlastností syntaktických stromů je, že jsou thread-safe a není možné je měnit. Díky tomu tak můžeme přistupovat k jednomu stromu z více vláken a máme jistotu, že se nikdy nezmění. Na stromu lze ale provádět úpravy. K tomuto účelu obsahuje API metody, které vždy vytvoří kopii stromu. Tato kopie je přístupná pouze vláknu, které chce úpravy provádět. Syntaktický strom se skládá ze tří prvků. Obsahuje syntaktické uzly (syntax node), tokeny (syntax token) a prvky, které reprezentují komentáře, bílé znaky a direktivy preprocesoru, jsou označovány jako trivia.

Syntaktické uzly reprezentují všechny konstrukce jazyka. Jsou v nich obsaženy všechny deklarace a výrazy. Žádný uzel v syntaktickém stromu nemůže být koncový a nelze je měnit. V parseru pracuji převážně právě s uzly. S jejich pomocí mohu snadno získat například hlavičky tříd a metod, seznamy proměnných a další. Následující obrázek ukazuje práci se syntaktickým stromem a jeho uzly. V následujícím kódu je nastíněno, jak se dají získat hlavičky tříd ze zdrojového souboru. Obdobným způsobem získávám informace o metodách a členech třídy. Kořenovým uzlem je pro mě potom prvek seznamu

classDeclarationList, z něj mohu opět získat seznam jeho potomků a vybrat například uzly typu MethodDeclarationSyntax nebo MemberDeclarationSyntax.

```
//Vytvoří syntaktický strom a získá kořen stromu.
CompilationUnitSyntax root =
    (CompilationUnitSyntax) CSharpSyntaxTree.ParseText(sourceCode).GetRoot();

//Seznam všech uzlů ve zdrojovém souboru
IEnumerable<SyntaxNode> nodes = root.DescendantNodes();

//Získání seznamu deklarací tříd
List<ClassDeclarationSyntax> classDeclarationList =
    nodes.OfType<ClassDeclarationSyntax>().ToList();
```

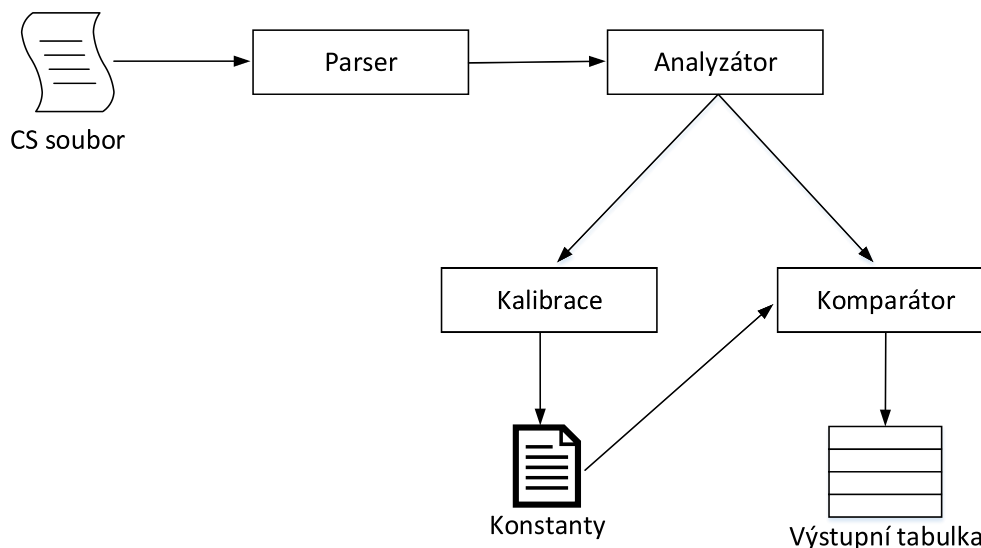
Obrázek 9: Ukázka použití kompilátoru Roslyn – získání deklarací tříd

Dalším prvkem syntaktického stromu jsou tokeny. Představují terminální symboly gramatik. Tokeny jsou nejmenší části zdrojového kódu, nelze je dále dělit. Patří sem klíčová slova, identifikátory, literální symboly a interpunkční znaménka. Posledním prvkem stromů jsou tzv. trivia. Patří sem bílé znaky, komentáře nebo direktivy preprocesoru. Tyto prvky se mohou objevovat kdekoliv mezi tokeny. Tyto prvky nemají rodiče stejného druhu, ale jsou potomky tokenů a všechny jsou typu SyntaxTrivia. [11]

## 5.4 Popis funkčnosti aplikace

Jak jsem uváděl výše, celá práce se skládá ze dvou částí. Nejdůležitější je knihovna tříd, která obsahuje celou logiku a nese název Cleaner. Druhou částí je okenní aplikace vytvořená pomocí technologie WPF. Její název je CleanCodeAnalyzer. Této aplikaci se nebudu nyní věnovat a v následující části mé práce budu popisovat pouze projekt Cleaner.

Na obrázku 10 je znázorněno blokové schéma celého projektu. Skládá se ze čtyř hlavních částí. První z nich je parser, který na vstupu přijímá zdrojový soubor. Ten je zpracován kompilátorem Roslyn a mapován na datovou strukturu, kterou poté přebírá analyzátor. Jeho úkolem je aplikovat metriky. Další postup dat v projektu se liší v závislosti na tom, jestli byl nahrán vzorový projekt. V takovém případě jsou data poslána do kalibrační části. Zde se hodnoty metrik průměrují a výsledky se ukládají jako konstanty do souboru. Pokud aplikace zpracovává již projekt určený ke kontrole, pošlou se data z analyzátoru do komparátoru. Tato část srovnává vzorové hodnoty s hodnotami z analyzátoru. Výsledkem je výstupní tabulka s problematickými třídami a hodnotami metrik.

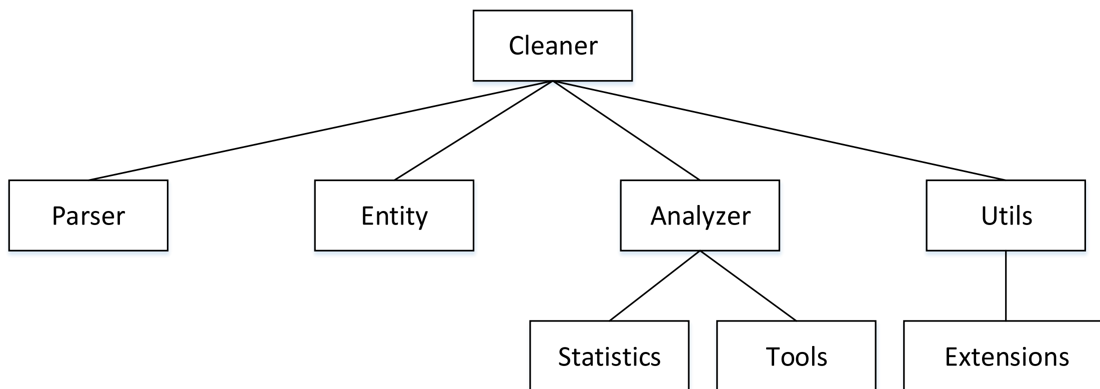



Obrázek 10: Blokové schéma aplikace

## 5.5 Struktura projektu

Knihovna Cleaner je složena z několika jmenných prostorů, jejichž schéma je znázorněné na obrázku 11. Následuje jejich krátký popis:

- `Cleaner` – Základní jmenný prostor, obsahuje konstanty a tovární třídu, přes kterou se přistupuje ke všem částem aplikace.
- `Cleaner.Parser` – Mapuje zdrojový kód do vlastní struktury tříd.
- `Cleaner.Entity` – Datová struktura, podobná stromu, která obsahuje informace o třídách, metodách a datových členech.
- `Cleaner.Analyzer` – Aplikuje metriky na zdrojový kód. Výsledky uloženy do struktury ve jmenném prostoru `Cleaner.Analyzer.Statistics`. Výpočty metrik provádí jmenný prostor `Cleaner.Analyzer.Tools`.
- `Cleaner.Calibration` – Výpočet průměrných hodnot metrik.
- `Cleaner.Comparator` – Porovnává naměřené hodnoty s kalibračními daty.
- `Cleaner.Utils` – Obsahuje pomocné třídy pro práci s výčtovými typy, řetězci a kolekcemi.




**Obrázek 11:** Jmenné prostory v projektu Cleaner

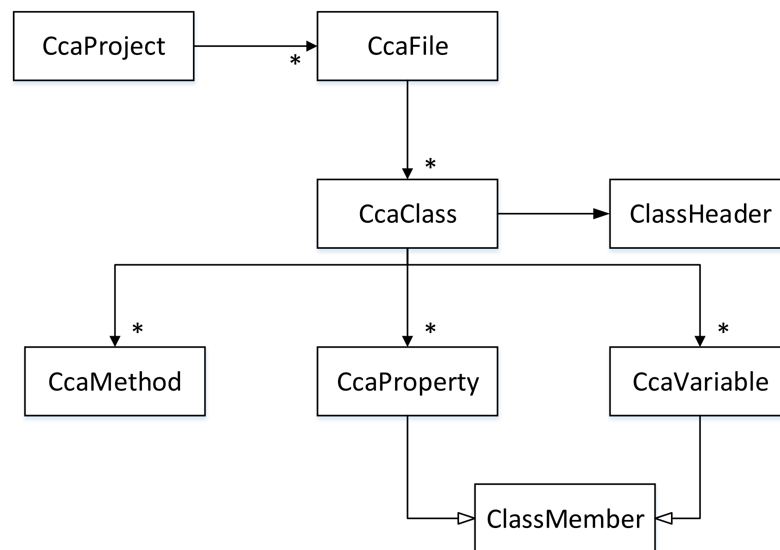
### 5.5.1 Parser

Parser na vstupu přijímá seznam souborů se zdrojovými kódy. Základem je třída `CcaParser`, která je zároveň vstupním bodem do parseru. Tato třída prochází seznam souborů a mapuje ho na seznam tříd typu `CcaFile`. Úkolem parseru je získat z kódu všechny důležité informace a uložit je do struktury, která je popsána v další kapitole.

Při návrhu parseru jsem měl dva cíle. Prvním je, aby parser byl samostatně použitelnou částí. Je ho tedy možné využít i pro jiné účely. Může být použit v aplikaci, která bude provádět analýzu kódu z jiného hlediska než je čistota, nebo třeba v aplikaci, která bude vizualizovat třídy. Může být dobrým základem pro tvorbu pluginů do Visual Studia. Z tohoto důvodu jsou ze zdrojového kódu získávány i informace, které při určování čistoty kódu aktuálně nevyužívám. Mým druhým cílem bylo, aby aplikace v budoucnu uměla zkoumat čistotu kódu i u dalších jazyků, aby se dal jednoduše napsat a vyměnit parser. Tento cíl se mi podařil jen částečně. Snažil jsem se parser navrhnout tak, aby při implementaci dalších jazyků poskytoval již připravenou strukturu metod a jeho návrh byl tak jednodušší. Rozhraní s bohatou podporou pro další vývoj nebylo možné vytvořit. Důvodem bylo úzké spojení s kompilátorem Roslyn, jehož třídy nelze zahrnout do rozhraní. Aby bylo možné pro parser vytvořit nějaké rozsáhlejší rozhraní, bylo by nutné oddělit parser od kompilátoru, například použitím návrhového vzoru Adapter. Vzrostla by tím však složitost celého parseru. Zvolil jsem tedy jednodušší řešení a vytvořil jednoduché rozhraní `ICcaParser`. Toto rozhraní obsahuje jedinou metodu, která vrací namapovanou stromovou strukturu (třídu `CcaProject`). Pro budoucí vývoj to přináší jistou volnost pro návrh dalšího parseru.

## 5.5.2 Datová struktura pro uložení informací

Úkolem parseru je projít zdrojový kód a informace z něj mapovat do datové struktury podobné stromu. Tato struktura je uložena ve jmenném prostoru `Cleaner.Entity`. Ze schématu struktury v obrázku 12 můžeme vidět, že hlavní třídou je třída `CcaProject`. Ta v sobě obsahuje seznam všech souborů (`CcaFile`) a ty zase a ty zase udržují seznam všech tříd (`CcaClass`). V této struktuře jsou uloženy informace, jako jsou názvy, modifikátory, návratové typy a argumenty metod. Ke každému prvku je zde zaznamenán i jeho zdrojový kód. Všechny třídy ve struktuře je potomkem rozhraní `IElement`. Toto rozhraní definuje název prvku. Třídní členy implementují zděděné rozhraní `IClassElement`, které je rozšířené o modifikátory. Všechny modifikátory jsou definovány ve výčtových typech.




**Obrázek 12:** Schéma struktury pro uložení informací ze zdrojového kódu

## 5.5.3 Analyzer

Další částí projektu je analyzer. Ten přebírá od parseru třídu `CcaProject` a jeho úkolem je na ní aplikovat výše popsané metriky. Výsledek se poté ukládá do struktury ve jmenném prostoru `Cleaner.Analyzer.Statistics`. Základní třídou analyzeru je `CcaAnalyzer`. Implementuje rozhraní `IAnalyzer` s metodou `Analyze` a vlastností, která vrací seznam statistik.

Výsledky metrik pro danou třídu udržuje třída `ClassStatistics`. Je to základní třída struktury podobné stromu, která má v sobě všechny naměřené hodnoty pro jednotlivé třídy.

### 5.5.4 Kalibrace

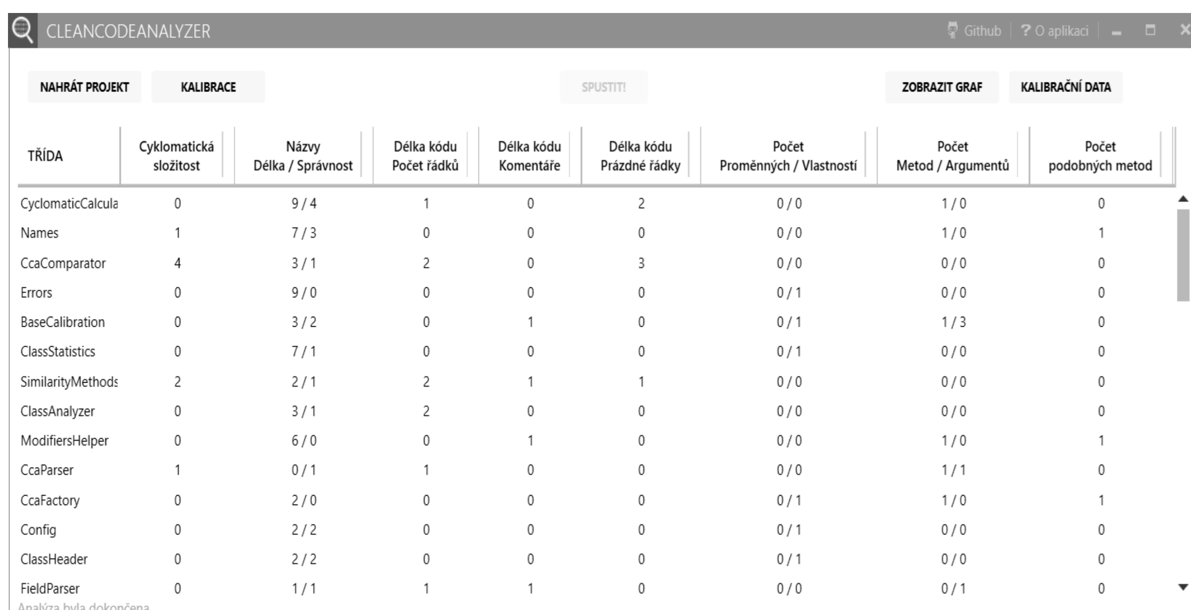
Tato část provádí výpočet průměrných hodnot jednotlivých metrik. Od analyzeru získává seznam tříd typu `ClassStatistics`, obsahující vypočítané metriky pro třídy, metody, vlastnosti i třídní proměnné. Základem je třída `CcaCalibration`, která prochází seznam a provádí výpočty pro třídy. K dispozici má pomocné třídy pro výpočty průměrů metrik nad metodami a vlastnostmi. Výsledky jsou uloženy do textového souboru. Zde jsou uloženy hodnoty ve formátu `metrika=hodnota`.

### 5.5.5 Komparátor

Tato část obsahuje jedinou třídu `CcaComparator`, která přijímá od analyzeru seznam tříd typu `ClassStatistic`, zároveň načítá konfigurační soubor s průměrnými hodnotami metrik. Výsledky se ukládají do objektu `Result`, který obsahuje vždy konkrétní třídu a slovník, kde klíčem je název metriky a hodnotou je počet chyb. Třída `CcaComparator` porovnává všechna naměřená data s hodnotami metrik z kalibrace. Pokud je naměřená vyšší, je v objektu `Result` u dané třídy a metriky zvýšena hodnota o jedničku.

## 5.6 Ukázka aplikace

Na obrázku 13 je ukázka aplikace CleanCodeAnalyzer. Okno se skládá z několika částí. V horní části aplikace jsou tlačítka pro vložení projektu k analýze, kalibraci nebo zobrazení kalibračních dat. Zbytek okna tvoří tabulka s výsledky analýzy, seřazené podle celkového počtu chyb. V prvním sloupečku je název třídy, další sloupečky obsahují výsledky metrik. Výsledky lze řadit podle všech sloupečků kliknutím na jejich nadpisy.




| TŘÍDA | Cyklomatická složitost | Názvy Délka / Správnost | Délka kódu Počet řádků | Délka kódu Komentáře | Délka kódu Prázdné řádky | Počet Proměnných / Vlastností | Počet Metod / Argumentů | Počet podobných metod |
|---|---|---|---|---|---|---|---|---|
| CyclomaticCalcula | 0 | 9 / 4 | 1 | 0 | 2 | 0 / 0 | 1 / 0 | 0 |
| Names | 1 | 7 / 3 | 0 | 0 | 0 | 0 / 0 | 1 / 0 | 1 |
| CcaComparator | 4 | 3 / 1 | 2 | 0 | 3 | 0 / 0 | 0 / 0 | 0 |
| Errors | 0 | 9 / 0 | 0 | 0 | 0 | 0 / 1 | 0 / 0 | 0 |
| BaseCalibration | 0 | 3 / 2 | 0 | 1 | 0 | 0 / 1 | 1 / 3 | 0 |
| ClassStatistics | 0 | 7 / 1 | 0 | 0 | 0 | 0 / 1 | 0 / 0 | 0 |
| SimilarityMethods | 2 | 2 / 1 | 2 | 1 | 1 | 0 / 0 | 0 / 0 | 0 |
| ClassAnalyzer | 0 | 3 / 1 | 2 | 0 | 0 | 0 / 0 | 0 / 0 | 0 |
| ModifiersHelper | 0 | 6 / 0 | 0 | 1 | 0 | 0 / 0 | 1 / 0 | 1 |
| CcaParser | 1 | 0 / 1 | 1 | 0 | 0 | 0 / 0 | 1 / 1 | 0 |
| CcaFactory | 0 | 2 / 0 | 0 | 0 | 0 | 0 / 1 | 1 / 0 | 1 |
| Config | 0 | 2 / 2 | 0 | 0 | 0 | 0 / 1 | 0 / 0 | 0 |
| ClassHeader | 0 | 2 / 2 | 0 | 0 | 0 | 0 / 1 | 0 / 0 | 0 |
| FieldParser | 0 | 1 / 1 | 1 | 1 | 0 | 0 / 0 | 0 / 1 | 0 |

**Obrázek 13:** Aplikace CleanCodeAnalyzer

## 5.7 Výsledky analýzy

Aplikace CleanCodeAnalyzer porovnává naměřené hodnoty metrik se vzorovými daty. Tato kapitola ukazuje, jakých výsledků dosahuje zdrojový kód aplikace CleanCodeAnalyzer, konkrétně pouze její výpočetní jádro umístěné v knihovně Cleaner. Pro porovnání jsem zvolil jmenný prostor `System` z jádra .NET frameworku, který obsahuje přibližně 190 tříd. Analyzovaná knihovna Cleaner obsahuje 50 tříd.

### 5.7.1 Kalibrační data

Po vložení vzorového projektu do aplikace se vypočítají metriky pro všechny třídy a následně se zprůměrují. Pro každou metriku existuje jedna vzorová hodnota. Následující tabulka obsahuje kalibrační data, která se vypočítají ze zdrojového kódu jmenného prostoru `System` v .NET frameworku. V tabulce 1 jsou uvedené průměrné hodnoty metrik.

Tabulka obsahuje informace o tom, kolik mají třídy řádků, kolik obsahují metod a jak jsou složité.

| Název | Hodnota | Název | Hodnota |
|---|---|---|---|
| Délka názvu třídy / správnost | 16 / 85% | Počet řádků metody | 14 |
| Počet řádků třídy | 124 | Počet komentářů / prázdných řádků metody | 2 / 1 |
| Počet komentářů / prázdných řádků třídy | 19 / 13 | Počet argumentů metody | 2 |
| Počet proměnných / vlastností | 4 / 3 | Cyklomatická složitost | 4 |
| Počet metod / podobných metod | 5 / 1 | Délka názvu vlastnosti / správnost | 11 / 89% |
| Délka názvu metod / správnost | 14 / 85% | Délka názvu proměnné / správnost | 15 / 100% |

Tabulka 1: Kalibrační data pro jmenný prostor System .NET frameworku

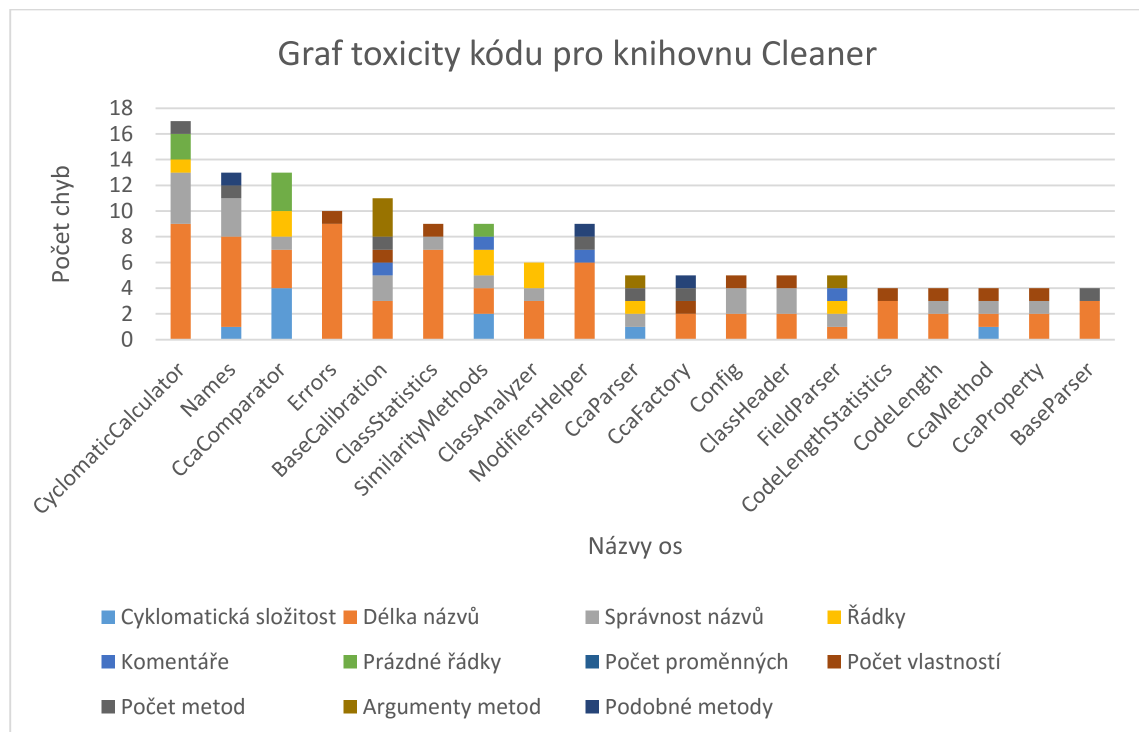
### 5.7.2 Analýza aplikace CleanCodeAnalyzer

Pro analýzu používám zdrojový kód aplikace CleanCodeAnalyzer a porovnávám s hodnotami uvedenými v předchozí kapitole. Výsledek analýzy aplikace zobrazuje do tabulky, která je znázorněna na obrázku 14.

| Třída | Cyklomatická složitost | Názvy Délka / Správnost | Délka kódu Počet řádků | Délka kódu Komentáře | Délka kódu Prázdné řádky | Počet Proměnných / Vlastností | Počet Metod / Argumentů | Počet podobných met |
|---|---|---|---|---|---|---|---|---|
| CyclomaticCalculator | 0 | 9 / 4 | 1 | 0 | 2 | 0 / 0 | 1 / 0 | 0 |
| Names | 1 | 7 / 3 | 0 | 0 | 0 | 0 / 0 | 1 / 0 | 1 |
| CcaComparator | 4 | 3 / 1 | 2 | 0 | 3 | 0 / 0 | 0 / 0 | 0 |
| Errors | 0 | 9 / 0 | 0 | 0 | 0 | 0 / 1 | 0 / 0 | 0 |
| BaseCalibration | 0 | 3 / 2 | 0 | 1 | 0 | 0 / 1 | 1 / 3 | 0 |
| ClassStatistics | 0 | 7 / 1 | 0 | 0 | 0 | 0 / 1 | 0 / 0 | 0 |
| SimilarityMethods | 2 | 2 / 1 | 2 | 1 | 1 | 0 / 0 | 0 / 0 | 0 |
| ClassAnalyzer | 0 | 3 / 1 | 2 | 0 | 0 | 0 / 0 | 0 / 0 | 0 |
| ModifiersHelper | 0 | 6 / 0 | 0 | 1 | 0 | 0 / 0 | 1 / 0 | 1 |
| CcaParser | 1 | 0 / 1 | 1 | 0 | 0 | 0 / 0 | 1 / 1 | 0 |
| CcaFactory | 0 | 2 / 0 | 0 | 0 | 0 | 0 / 1 | 1 / 0 | 1 |
| Config | 0 | 2 / 2 | 0 | 0 | 0 | 0 / 1 | 0 / 0 | 0 |
| ClassHeader | 0 | 2 / 2 | 0 | 0 | 0 | 0 / 1 | 0 / 0 | 0 |
| FieldParser | 0 | 1 / 1 | 1 | 1 | 0 | 0 / 0 | 0 / 1 | 0 |
| CodeLengthStatistics | 0 | 3 / 0 | 0 | 0 | 0 | 0 / 1 | 0 / 0 | 0 |
| CodeLength | 0 | 2 / 1 | 0 | 0 | 0 | 0 / 1 | 0 / 0 | 0 |
| CcaMethod | 1 | 1 / 1 | 0 | 0 | 0 | 0 / 1 | 0 / 0 | 0 |
| CcaProperty | 0 | 2 / 1 | 0 | 0 | 0 | 0 / 1 | 0 / 0 | 0 |
| BaseParser | 0 | 3 / 0 | 0 | 0 | 0 | 0 / 0 | 1 / 0 | 0 |

Obrázek 14: Výsledky analýzy pro knihovnu Cleaner

Čísla v tabulce znamenají počet chyb. Například cyklomatická složitost u třídy `CcaComparator` má hodnotu 4. To znamená, že v této třídě jsou čtyři metody, které mají vyšší cyklomatickou složitost, než jaká je průměrná. Na dalším obrázku je graf toxicity kódu pro knihovnu Cleaner. Do grafu jsem vybral několik nejhůře hodnocených tříd. Výhodou tohoto grafu je, že na první pohled je vidět, jaké třídy jsou nejvíce toxické a s jakými metrikami mají problémy.




**Obrázek 15:** Graf toxicity kódu pro knihovnu Cleaner

Nejhůře hodnocenou třídou je třída `CyclomaticComplexity`, která má nejvíce problémů s názvy metod. V tomto případě se nelze chybám vyhnout, neboť předkem této třídy je třída z kompilátoru Roslyn `CSharpSyntaxWalker` jejíž metody jsou zde implementovány a názvy jsou pevně dané. Třídou s nejvyšší cyklomatickou složitostí, je třída `CcaComparator`. Je to dáno tím, že tato třída porovnává naměřené metriky se vzorovými daty a obsahuje tak spoustu příkazů `if`. Nejvíce argumentů metod obsahuje třída `BaseCalibration`, v níž jsou umístěné pomocné metody, které jsou přetěžované. Zde se tedy nelze vyhnout tomu, aby metoda neměla více argumentů. Ostatní třídy mají problémy s délkou či správností názvů a počtem vlastností, jedná se však většinou o třídy, které jsou určené pouze k přenosu dat a mají tak více vlastností.

## 5.8 Využití

Aplikace může být pomocníkem při psaní čistého kódu. Její výhodou je, že se dokáže přizpůsobit potřebám programátorů a umí kontrolovat právě ty chyby, které jsou pro ně důležité. Nalezne uplatnění zejména při týmovém vývoji. Dále může být užitečná například při výuce. Učitelé budou moci hodnotit studenty nejenom na základě funkčnosti aplikace, ale i z hlediska čistoty.

# 6. ZÁVĚR

Ve své diplomové práci jsem se snažil uvést čtenáře do problematiky psaní čistého kódu. Popsal jsem pravidla pro jeho psaní i způsoby, jak je možné čistotu kódu změřit. Rovněž jsem zkoumal i nástroje, které pomáhají programátorům udržovat jejich zdrojový kód v čistotě. Zde jsem kladl důraz hlavně na vývojová prostředí, která jsou pro většinu vývojářů nejpoužívanějším prostředkem při psaní čistého kódu. Dále jsem se zabýval jen nástroji pro C# .NET a představil několik pluginů do Visual Studia.

Nabyté poznatky z první části práce jsem využil pro vývoj vlastní aplikace. Ta implementuje několik metrik, například cyklomatickou složitost, délku kódu, kontroluje názvy identifikátorů nebo třeba zkoumá podobnost metod. Úkolem aplikace je porovnávat zdrojový kód se vzorovým kódem, o kterém máme představu jak je napsaný. Pro vzorový i pro analyzovaný kód se vypočtou hodnoty metrik a porovnají se. Jako vzorový může být použit i toxický kód, pak očekáváme, že analyzovaný kód bude mít po testu lepší výsledky.

Aplikace může být dále rozšířena o další metriky, tak aby posuzovala kód z více hledisek. Struktura programu navíc dovoluje přidání jiného parseru, aby aplikace uměla zpracovávat další jazyky. Užitečným rozšířením by mohla být spolupráce s gitem pro porovnání odchozích commitů s kódem v repozitáři. Tím by se zajistilo, aby byl zdrojový kód v celé aplikaci napsán stejným stylem.

# CITOVANÁ LITERATURA

[1] MARTIN, Robert C. *Čistý kód: [návrhové vzory, refaktorování, testování a další techniky agilního programování*. Překlad Jiří Berka. Brno: Computer Press, 2009, 423 s. ISBN 978-80-251-2285-3.

[2] File Open Method. *MSDN: Developer network* [online]. [cit. 2015-11-16]. Dostupné z: https://msdn.microsoft.com/en-us/library/b9skfh7s(v=vs.110).aspx.

[3] MARTIN, Robert C. a Micah MARTIN. *Agile principles, patterns, and practices in C#*. Upper Saddle River, NJ: Prentice Hall, 2007. Robert C. Martin series. ISBN 978-0-13-185725-4.

[4] KLIMEŠOVÁ, Hana. *Metriky a model složitosti*. Praha, 2007. Bakalářská práce. ČVUT FEL.

[5] Lines of code metrics (LOC). *Aivosto: Programming Tools for Software Developers* [online]. [cit. 2015-12-21]. Dostupné z: http://www.aivosto.com/project/help/pm-loc.html.

[6] LIPPERT, Martin a Stephen. ROOCK. *Refactoring in large software projects*. Hoboken, NJ: John Wiley, 2006. ISBN 978-047-0858-929.

[7] MCCABE, T. J. A Complexity Measure. *IEEE Transactions on Software Engineering*. 1976, SE-2(4), 308-320. DOI: 10.1109/TSE.1976.233837. ISSN 0098-5589. Dostupné také z: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=1702388.

[8] MCCABE, Thomas J. a Arthur H. WATSON. *Structured testing: A Testing Methodology Using the Cyclomatic Complexity Metric*. Washington, D. C.: For sale by the Supt. of Docs., U.S.G.P.O., 1996. NBS special publication, 500-99.

[9] Welcome to the NetBeans Community. *NetBeans* [online]. Oracle Corporation [cit. 2015-11-28]. Dostupné z: https://netbeans.org/about/.

[10] ECMA INTERNATIONAL. *C# Language Specification: C# Language Specification*. Ženeva: Ecma International, 2006. 4th. Dostupné také z: http://www.ecma-international.org/publications/files/ECMA-ST/Ecma-334.pd

[11] .NET Compiler Platform (“Roslyn”) Overview. *CodePlex* [online]. Microsoft Corporation, 2015 [cit. 2016-04-24]. Dostupné z: https://roslyn.codeplex.com/wikipage?title=Overview&referringTitle=Documentation.

# OBSAH PŘILOŽENÉHO CD

Přiložené CD obsahuje následující soubory:

- Text diplomové práce
  - Diplomova_prace_2016_Vladimir_Antoš.pdf
  - Diplomova_prace_2016_Vladimir_Antos.doc
  - Kopie_zadani_diplomova_prace_2016_Vladimir_Antos.pdf
- Zdrojové kódy
  - /CleanCodeAnalyzer/ - složka obsahující všechny zdrojové kódy
  - /CleanCodeAnalyzer_aplikace/ - obsahuje spustitelný soubor aplikace
- Obrázky použité v diplomové práci
- Tabulka a graf s naměřenými hodnotami pro zdrojové kódy aplikace CleanCodeAnalyzer