FAKULTA
INFORMAČNÍCH
TECHNOLOGIÍ
ČVUT V PRAZE

Zadání bakalářské práce

Název:	Výpočet softwarových metrik
Student:	Filip Gregor
Vedoucí:	Ing. Ladislav Vagner, Ph.D.
Studijní program:	Informatika
Obor / specializace:	Teoretická informatika
Katedra:	Katedra teoretické informatiky
Platnost zadání:	do konce letního semestru 2021/2022

Pokyny pro vypracování

1. Seznamte se s problematikou softwarových metrik. Zaměřte se na metriky, které má smysl určovat pro kratší programy odevzdané jako studentská řešení cvičných školních úloh.

2. Seznamte se s existujícími programy pro výpočet softwarových metrik. Zaměřte se na řešení, která počítají metriky programů napsaných v jazycích C, C++ a Java.

3. Seznamte se s infrastrukturou kompilátoru clang a s možnostmi psaní doplňků do tohoto kompilátoru.

4. Navrhněte a realizujte vlastní program pro výpočet softwarových metrik programů napsaných v jazycích C a C++. Implementujte zejména výpočet metrik podle bodu (1). Tam, kde to má smysl, využijte pro řešení infrastrukturu kompilátoru clang.

5. Otestujte realizované řešení na sadě testovacích dat. Dosažené výsledky porovnejte s výsledky dostupných open-source řešení.

Elektronicky schválil/a doc. Ing. Jan Janoušek, Ph.D. dne 26. ledna 2021 v Praze.

FAKULTA
INFORMAČNÍCH
TECHNOLOGIÍ
ČVUT V PRAZE

Bakalářská práce

Výpočet softwarových metrik

Filip Gregor

Katedra teoretické informatiky
Vedoucí práce: Ing. Ladislav Vagner Ph.D.

13. května 2021

Poděkování

Rád bych poděkoval svým rodičům za neuvěřitelnou podporu během celého studia. Také děkuji své přítelkyni. Bez její podpory, tolerance a neustálého povzbuzování by tento text pravděpodobně nevznikl.

V neposlední řadě bych chtěl poděkovat Ing. Ladislavu Vagnerovi Ph.D. za odborné vedení práce, cenné rady, které mi k práci poskytl a za zajímavé téma.

Prohlášení

Prohlašuji, že jsem předloženou práci vypracoval samostatně a že jsem uvedl veškeré použité informační zdroje v souladu s Metodickým pokynem o dodržování etických principu při přípravě vysokoškolských závěrečných prací.

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona, ve znění pozdějších předpisu. V souladu s ust. § 2373 odst. 2 zákona č. 89/2012 Sb., občanský zákoník, ve znění pozdějších předpisu, tímto uděluji nevýhradní oprávnění (licenci) k užití této mojí práce, a to včetně všech počítačových programu, jež jsou její součástí či přílohou a veškeré jejich dokumentace (dále souhrnně jen „Dílo"), a to všem osobám, které si přejí Dílo užít. Tyto osoby jsou oprávněny Dílo užít jakýmkoli zpusobem, který nesnižuje hodnotu Díla a za jakýmkoli účelem (včetně užití k výdělečným účelum). Toto oprávnění je časově, teritoriálně i množstevně neomezené. Každá osoba, která využije výše uvedenou licenci, se však zavazuje udělit ke každému dílu, které vznikne (byť jen zčásti) na základě Díla, úpravou Díla, spojením Díla s jiným dílem, zařazením Díla do díla souborného či zpracováním Díla (včetně překladu) licenci alespoň ve výše uvedeném rozsahu a zároveň zpřístupnit zdrojový kód takového díla alespoň srovnatelným zpusobem a ve srovnatelném rozsahu, jako je zpřístupněn zdrojový kód Díla.

V Praze dne 13. května 2021 . . . .

České vysoké učení technické v Praze
Fakulta informačních technologií
© 2021 Filip Gregor. Všechna práva vyhrazena.
Tato práce vznikla jako školní dílo na Českém vysokém učení technickém v Praze, Fakultě informačních technologií. Práce je chráněna právními předpisy a mezinárodními úmluvami o právu autorském a právech souvisejících s právem autorským. K jejímu užití, s výjimkou bezúplatných zákonných licencí a nad rámec oprávnění uvedených v Prohlášení na předchozí straně, je nezbytný souhlas autora.

Odkaz na tuto práci

Gregor, Filip. *Výpočet softwarových metrik*. Bakalářská práce. Praha: České vysoké učení technické v Praze, Fakulta informačních technologií, 2021.

Abstrakt

Cílem práce je popsat jednotlivé softwarové metriky a vytvořit software, který je bude měřit pro zdrojové soubory v jazycích C a C++. Pro implementaci měřiče byl využit překladač Clang. Jsou popsány jeho rozhraní a jak je využít pro statickou analýzu kódu. Implementovaný měřič měří více metrik než dostupné open-source alternativy. Proti nim byl porovnán na různých projektech.

Klíčová slova Softwarové metriky, statická analýza kódu, překladač, Clang, LibTooling, cyklomatická složitost, Lines of Code, NPATH, objektově orientované programování

Abstract

This thesis presents various software metrics, how they are measured and what they say about the measured source code. Next, compiler Clang and it's interfaces are described. One of those interfaces is then used for implementing custom metric measurer for source code in C or C++. Said measurer is then compared against other open-source measurers and thoroughly tested.

Keywords Software metrics, static code analysis, compiler, Clang, LibTooling, cyclomatic complexity, Lines of Code, NPATH, object oriented programming

Obsah

1 Úvod **1**
1.1 Motivace . . . 1
1.2 Cíle . . . 1
1.3 Struktura práce . . . 2

2 Softwarové metriky **3**
2.1 Kvalita softwaru . . . 3
2.2 Softwarové metriky zaměřené na velikost kódu . . . 3
2.3 Softwarové metriky zaměřené na složitost kódu . . . 5
2.4 Objektově orientované metriky . . . 11
2.5 Existující analyzátory softwarových metrik . . . 16

3 Clang **19**
3.1 Motivace . . . 19
3.2 Překladače . . . 19
3.3 Clang a jeho rozhraní . . . 21
3.4 LibClang . . . 22
3.5 Clang Plugins . . . 22
3.6 LibTooling . . . 23

4 Implementace **25**
4.1 Kostra programu . . . 25
4.2 Měřiče metrik . . . 25
4.3 Implementace jednotlivých měřičů . . . 26
4.4 Detaily implementace . . . 31
4.5 Porovnání s ostatními měřiči . . . 32

5 Testování **33**
5.1 Testování na velkém projektu . . . 33

5.2 Testování na malém projektu 35

6 Závěr **39**
6.1 Shrnutí práce 39
6.2 Přínosy práce 39
6.3 Navazující práce 40

Literatura **41**

A Seznam použitých zkratek **45**

B Graf porovnávající měřiče **47**

C Uživatelská příručka **49**

D Obsah přiloženého CD **51**

Seznam obrázků

3.1 Schéma jednoduchého překladače. 20
3.2 Zjednodušený příklad abstraktního syntaktického stromu. 21

4.1 AST Clangu pro výraz `std::string = "Hello"`. 27

5.1 Odchylka u počtu příkazů. 34
5.2 Odchylka u cyklomatické složitosti. 35
5.3 Odchylka u fan-in. 36
5.4 CCCC - Odchylka u cyklomatické složitosti. 37
5.5 PMCCabe - Odchylka u počtu příkazů. 38

B.1 PMCCabe - Odchylka u cyklomatické složitosti. 47

Seznam tabulek

4.1 Přehled metrik, které jednotlivé programy měří 32

Kapitola 1

Úvod

1.1 Motivace

Hodnocení kvality návrhu studentských programů je na školách, kde se učí programování, běžná rutina. Na posouzení kvality kódu typicky slouží code review. Vyučující musí ručně projít studentův kód a najít v něm nedostatky. Taková činnost může být časově náročná, obzvlášť pokud je studentů mnoho. Není v silách vyučujících takto projít všechny programy, které studenti odevzdávají.

Tato práce se zabývá softwarovými metrikami. Jedná se o funkce, které měří nějakou vlastnost zdrojového kódu. Ty mohou do jisté míry tento proces nahradit. Studenti by tak mohli mít alespoň rámcovou zpětnou vazbu na kvalitu jejich řešení.

1.2 Cíle

Cíle práce jsou shrnuty v následujících bodech:

- Nastudovat vhodné softwarové metriky pro krátké studentské programy a existující měřiče metrik.
- Popsat, jak fungují překladače a jak využít rozhraní překladače Clang [1] pro vytvoření analyzátoru zdrojových kódu.
- Implementovat měřič softwarových metrik pro jazyky C a C++ za pomocí překladače Clang.
- Otestovat implementovaný měřič a porovnat výsledky implementovaného měřiče s existujícími open-source alternativami.

1.3 Struktura práce

Kapitola 2 se zabývá existujícími metrikami. U každé metriky je popsána její definice a způsob, jak se měří. Jsou také popsány existující měřiče metrik. V kapitole 3 je nastíněno, jak fungují překladače. Dále je popsán překladač Clang a jeho rozhraní, díky kterým je možné jednoduše vytvořit program pro statickou analýzu zdrojového kódu. Poté je v kapitole 4 popsána implementace samotného měřiče a jednotlivých metrik. Jsou zde popsána pravidla pro jejich počítání a případné vychýlení od originálních definic metrik z kapitoly 2. Kapitola 5 popisuje porovnání implementace s open-source alternativami a její otestování. V kapitole 6 jsou shrnuty přínosy práce a témata, kterými je možno na práci navázat.

Kapitola 2

Softwarové metriky

2.1 Kvalita softwaru

Pro potřeby práce rozdělíme kvalitu softwaru na dvě kategorie. První z nich vnímá software jako kvalitní, pokud vrací očekávané výsledky. Tato vlastnost je bezesporu velmi důležitá. U krátkých konzolových aplikací je realizace takového měření poměrně jednoduchá. Program má nějaký vstup a výstup. Pro určitou podmnožinu vstupů otestujeme, jestli vrací očekávaný výstup. Nutno podotknout, že takto zachytíme pouze ty případy, kdy program funguje správně. Obvykle není možné takto program otestovat pro celou možnou množinu vstupů.

Druhá z nich je kvalita návrhu. Pokud je software složitý, je těžší ho udržovat a rozšiřovat. Kontrolu je možné provádět přes code review. Vyučující ručně zkontroluje studentův program a najde nedostatky v návrhu. Taková činnost bývá časově náročná a není realizovatelná pro všechny projekty všech studentů. Tzv. softwarové metriky vystihují různé vlastnosti programu přímo z jeho zdrojového kódu. Jejich samotné měření je možné zautomatizovat. Studenti tak mohou dostat alespoň rámcovou zpětnou vazbu o kvalitě návrhu jejich programů.

Softwarová metrika je funkce, jejíž definiční obor je část zdrojového kódu. Často to bývá funkce nebo třída. Obor hodnot je typicky číslo, které popisuje nějakou vlastnost objektu na vstupu. Obvykle se jedná o celá čísla nebo čísla z rozsahu $\langle 0, 1 \rangle$.

2.2 Softwarové metriky zaměřené na velikost kódu

Tyto metriky jsou relativně jednoduché na pochopení a interpretaci. Dřív se používaly na posouzení produktivity programátora[1]. Toto uvažování se ale ukázalo jako liché [2]. Alternativní interpretace je následující: těla funkcí, která

[1] Čím více toho napíše, tím více je produktivní.

jsou dlouhá, mohou být zbytečně komplikovaná. Takové funkce bývá obvykle vhodné rozdělit do více menších funkcí.

2.2.1 Lines of Code

Lines of Code (LOC) neboli počet řádků je jedna z nejpoužívanějších metrik [3]. Jedná se o prosté spočítání počtu řádků v programu nebo ve funkci. Tato metrika se může dále rozšiřovat.

Někteří programátoři používají prázdné řádky na oddělení logických částí kódu. Pokud bychom tyto řádky započítali, jejich kód by byl dle této metriky horší než kód programátorů, kteří tyto techniky nepoužívají. To stejné platí i o komentářích. Je tedy žádoucí vynechat řádky, které jsou prázdné nebo obsahují jen komentáře. V menších programech může vyšší hodnota této metriky znamenat, že je nějaký kus kódu zbytečně opakován. Může tedy indikovat nedostatečné využití principu DRY (Don't repeat yourself) [4].

Uvažme nyní tento kód:

```
    return x < 10 ? (x > 5 ? true : false) : false
```

který obvykle bývá méně přehledný než

```
if (x < 10 && x > 5)
    return true;
else
    return false;
```

Navíc v C a C++ se příkazy oddělují středníkem nebo závorkami. Toto je tedy validní funkce:

```
int strlen(const char *str)
{ int len = 0; while(*str) ++ len; return len; }
```

Zde může metrika narazit. Řešením může být počítání jednotlivých příkazů. To ale nevyřeší situaci s ternárním operátorem výše, proto je nutné zvolit úplně jinou strategii.

2.2.2 Halsteadovy metriky

Halstead [5] zadefinoval program jako posloupnost tokenů. Tyto tokeny se potom dělí na operátory a operandy. Metriky vycházejí z těchto měření:

- ν_1 = počet unikátních operátorů
- ν_2 = počet unikátních operandů
- N_1 = počet operátorů
- N_2 = počet operandů

Halstead poté definoval následující metriky:

- Vocabulary: $n = \nu_1 + \nu_2$
- Length: $N = N_1 + N_2$
- Volume: $V = N \cdot log_2(n)$
- Level: $L = \frac{V*}{V} = \frac{2}{\nu_1} \cdot \frac{\nu_2}{N_2}$
- Difficulty: $D = \frac{1}{L}$

Metriky Vocabulary, Length a Volume nám dávají různé náhledy na velikost programu [6]. Halstead nikde přímo nespecifikoval, co je operátor a operand, a neexistuje ani nějaký obecný konsenzus [5]. Při vytváření vlastního měřiče je tedy nutné toto explicitně definovat. U rozhodování, co je operátor a operand, je nutná opatrnost, protože všechny Halsteadovy metriky na nich přímo či nepřímo závisí. U nějakých výrazů to může být přímočaré. Jde například o aritmetické výrazy. Uvažme ale následující kus kódu:

```
int a = 3;
```

Intuitivně je `=` operátor, `a` a `3` jsou operandy. Rozdělit tokeny `int` a středník už není tak jednoduché. Středník je možné interpretovat jako operátor. Jeho funkce je ukončení jednoho příkazu. Další možnost je ignorovat ho a vůbec nepočítat. Deklaraci `int` můžeme považovat jako součást proměnné, tedy jako operand, nebo jako něco, co určuje podstatu proměnné, tedy operátor [7]. Halstead zadefinoval mnohem více metrik. Je možné je najít například v [5, 8], pro účely práce jsou ale příliš abstraktní.

2.3 Softwarové metriky zaměřené na složitost kódu

Následující metriky popisují složitost programu. Ta existuje v mnoha formách, nejčastěji se bude jednat o různé přístupy k počítání cest skrz zdrojový kód.

2.3.1 Cyklomatická složitost

Vystihnout skutečný počet cest skrz program nemusí být moc užitečné. Pokud program obsahuje cyklus, takových cest je nekonečně mnoho [9]. Cyklomatická složitost (Cyclomatic Complexity), kterou představil McCabe [10], se zaměřuje na počet lineárně nezávislých cest v programu. Cesta k této metrice vede přes teorii grafů. Nejdříve zavedeme pojem, který nám pomůže reprezentovat program jako graf.

Definice 2.3.1 *Graf řízení toku je orientovaný graf, kde každý vrchol reprezentuje blok kódu, který neobsahuje žádné větvení nebo místo, ve kterém se graf větví. Hrany tohoto grafu reprezentují tok řízení v programu.*

McCabe využil následující vlastnosti grafů.

Definice 2.3.2 *Nechť $G = (V, E)$ je neorientovaný graf. Množinu $E' \subseteq E$ nazýváme eulerovskou, pokud graf $G' = (V, E')$ má všechny stupně sudé.*

Lemma 2.3.3 *Množina E' je eulerovská, právě když existují kružnice E_1 až E_n takové, že tyto kružnice jsou navzájem disjunktní množiny a E' je jejich sjednocení.*

Věta 2.3.4 *Nechť $G = (V, E)$ je neorientovaný souvislý graf. Potom množina všech charakteristických vektorů eulerovských množin v G tvoří vektorový prostor nad tělesem $GF(2)$ dimenze $|E| - |V| + 1$. Dimenzi tohoto grafu nazýváme cyklomatické číslo.*

Více včetně definic všech pojmů a důkazů viz [11]. Program budeme reprezentovat grafem toku řízení. Přidáme jednu hranu, která spojuje začátek a konec programu. Tím vznikne silně souvislý graf. V takovém grafu určitě existuje alespoň jeden cyklus. Cyklomatické číslo takového grafu je

$$v(G) = (|E| + 1) - |V| + 1 = |E| - |V| + 2.$$

Dle McCabeho si můžeme vybrat bázi tak, aby bázové vektory odpovídaly sledům v programu. Cyklomatické číslo zůstane stejné nezávisle na volbě báze, navíc z lineární algebry je známo, že báze je soubor lineárně nezávislých vektorů. Dle McCabeho je tedy cyklomatické číslo počet lineárně nezávislých cest v programu.

Konstrukce grafu řízení toku nemusí být triviální. McCabe ve svém článku uvedl ještě jeden způsob, jak cyklomatickou složitost počítat. Konkrétně

$$v(G) = \pi + 1,$$

kde π je počet predikátů, které větví kód. Toto dává smysl i intuitivně, např. `if` rozdělí kód na dvě větve. Stačí tedy spočítat všechny predikáty, které větví kód. McCabe doporučuje počítat i operátory `&&` a `||`.

McCabe uvedl, že cyklomatická složitost kódu by neměla být větší než 10. Cyklomatická složitost nám dává dolní odhad na počet cest, který musíme mít, abychom prošli všechny cesty v programu. To je jeden z důvodů pro držení nízké cyklomatické složitosti. Více cest ve funkci totiž znesnadňuje její testování [9].

2.3.1.1 Nedostatky

Brian A. Nejmeh ve svém článku [9] uvedl několik nedostatků této metriky. Jde například o to, že cyklomatická složitost se k různým řídícím konstrukcím chová stejně, tedy například `if`, `for` a `while` vnímá jako stejné příkazy. Navíc nerozlišuje mezi zanořením jednotlivých příkazů, např. tři `for` cykly sekvenčně za sebou budou mít rozdílnou složitost na pochopení, než kdyby byly zanořené [12].

2.3.2 NPATH

NPATH [9] je metrika, která si klade za cíl vyřešit nedostatky McCabovy cyklomatické složitosti. Cílem metriky je počítat acyklické cesty skrz program. Je definována přímo na jednotlivých řídících konstrukcích v jazyce C.

if příkaz

```
if (<expr>)
    <if-range>
```

$$\text{NP(if)} = \text{NP}(\langle\text{if-range}\rangle) + \text{NP}(\langle\text{expr}\rangle) + 1$$

if-else příkaz

```
if (<expr>)
    <if-range>
else
    <else-range>
```

$$\text{NP(if-else)} = \text{NP}(\langle\text{if-range}\rangle) + \text{NP}(\langle\text{else-range}\rangle) + \text{NP}(\langle\text{expr}\rangle)$$

while příkaz

```
while (<expr>)
    <while-range>
```

$$\text{NP(while)} = \text{NP}(\langle\text{while-range}\rangle) + \text{NP}(\langle\text{expr}\rangle) + 1$$

do-while příkaz

```
do
    <do-range>
while (<expr>);
```

$$\text{NP(do)} = \text{NP}(\langle\text{do-range}\rangle) + \text{NP}(\langle\text{expr}\rangle) + 1$$

for příkaz

```
for (<expr1>; <expr2>; <expr3>)
    <for-range>
```

$$\text{NP(for)} = \text{NP}(\langle\text{for-range}\rangle) + \text{NP}(\langle\text{expr1}\rangle) + \text{NP}(\langle\text{expr2}\rangle) + \text{NP}(\langle\text{expr3}\rangle) + 1$$

switch příkaz

```
switch (<expr>)
    |
    <case-range_1>
    ...
    <case-range_n>
    <default-range>
```

$$\text{NP(switch)} = \text{NP}(\langle\text{expr}\rangle) + \text{NP}(\langle\text{default-range}\rangle) + \sum_{i=1}^{n} \text{NP}(\langle\text{case-range}_\text{i}\rangle)$$

? operátor

```
<expr1> ? <expr2> : <expr3>
```

$$\text{NP}(?) = \text{NP}(\langle\text{expr1}\rangle) + \text{NP}(\langle\text{expr2}\rangle) + \text{NP}(\langle\text{expr3}\rangle) + 2$$

Operátor `?` je v tomto podobný jako `if-else`.

goto příkaz

Když je příkaz `goto label` spuštěn, program bude dále pokračovat od zadaného návěští. Příkaz `goto` může provádět skoky dopředu nebo dozadu[2]. Kvůli této dvojznačnosti se s `goto` v kontextu této metriky obtížně nakládá, a tak se vůbec nepočítá. Ačkoli by mohl v programu vytvořit mnoho větví, v praxi se používá velmi zřídka. Navíc použití `goto` v programu většinou značí špatně napsaný kód[13].

break příkaz

Příkaz `break` zapříčiní vystoupení z těla cyklu nebo z těla `switch` příkazu. V kontextu cest v programu na něj můžeme nahlížet jako na poslední příkaz v jednom bloku kódu. Jeho složitost je tedy 1.

výrazy

Jedná se o výrazy, kde vystupují logické operátory, tedy

```
<expr1> op1 <expr2> op2 ... op(n-1) <exprN>
```

kde `op-i` jsou logické operátory `and` (`&&`) nebo `or` (`||`). Za každý tento operátor ve výrazu se zvyší NPATH o 1. Uvažme následující kus kódu v jazyce C:

[2]Potom se chová podobně jako cyklus.

```
if(a && b) {
    S1;
}
else {
    S2;
}
```

Tento kód je ekvivalentní kódu

```
if(a)
    if(b)
        S1;
    else
        S2;
else
    S2;
```

Je vidět, že `&&` je ekvivalentní příkazu `if`. Operátor `or` je možné přepsat podobně. Pro `if(a || b) S1; else S2;` platí, že

```
if(a)
    S1;
else
    if(b)
        S1;
    else
        S2;
```

je ekvivalentní. Můžeme tedy vyjít z `if-else` příkazu definovaného výše a za každý takový operátor zvýšit složitost o jedna.

continue příkaz

Příkaz `continue` zastaví vykonávání následujícího kódu a vynutí novou iteraci. V kontextu cest reprezentuje hranu zpět do začátku cyklu. NPATH s touto konstrukcí nepracuje.

return příkaz

Složitost příkazu `return <expr>` je rovna NP(`<expr>`), tedy složitosti výrazu, který obsahuje.

Sekvenční příkazy a volání funkce

Sekvenční příkazy se vykonávají sekvenčně. Existuje vždy jedna cesta do dalšího příkazu, složitost je tedy 1. Totéž platí i o volání funkce.

Funkce

Složitost jedné funkce v jazyce C je

$$\text{NPATH} = \prod_{i=1}^{N} \text{NP}(\text{Statement}_\text{i})$$

Zároveň jakýkoli rozsah obsahující sekvenční příkazy má složitost rovnou součinu složitostí těchto příkazů. Za povšimnutí stojí, že je zde násobení. Například pokud bude program obsahovat několik vnořených `if` příkazů, jejich složitost bude součtem dílčích složitostí jednotlivých `if` příkazů. Naopak pokud se v programu nachází za sebou, bude složitost součinem složitostí jednotlivých `if` příkazů. Zde se NPATH velmi liší od cyklomatické složitosti. Ta totiž nebere zřetel na to, jak je program poskládaný.

Uvedeme si jedno úskalí této metriky. Uvažme následující kód:

```
if(a == b)
    S1;
if(a == b)
    S2;
```

Nechť `S1` nijak nemění proměnné `a` a `b`. Jeho NPATH složitost je 4. Ve skutečnosti jsou to ale jen dvě cesty, protože podmínky jsou stejné. NPATH s tímto neumí pracovat a naměří větší počet acyklických cest skrz program, než jich ve skutečnosti je. Tento nedostatek má i cyklomatická složitost.

2.3.2.1 Srovnání s ostatními metrikami

Brian A. Nejmeh [9] ve svém článku uvedl výsledky měření NPATH, LOC, počtu tokenů a cyklomatické složitosti na různých zdrojových kódech v jazyce C a korelace těchto měření. Metriky LOC, počet tokenů a cyklomatická složitost měly mezi sebou vysokou korelaci, NPATH s ostatními metrikami nikoli. Podle [9] se metriky LOC, počet tokenů a cyklomatická složitost řídí hlavně lexikálním obsahem programu, proto nejsou tak citlivé na počet cest ve funkci.

2.3.3 Henry and Kafura's Information Flow

Henry and Kafura's Information Flow [14] se zaměřuje na tok informací mezi jednotlivými moduly. Popisuje závislost jednoho konkrétního modulu na ostatních modulech a ostatních modulů na jednom konkrétním modulu. Využijeme následující definici:

Definice 2.3.5 *Lokální tok informací z modulu A do B existuje právě tehdy, pokud nastává jedno z následujících:*

1. *A volá B,*
2. *B volá A a A mu navrátí hodnotu, kterou B využije,*

3. *C volá A i B a návratovou hodnotu z A využije při volání B.*

Díky této definici popíšeme míru závislosti jednoho konkrétního modulu na ostatních modulech či ostatních modulů na jednom konkrétním.

Definice 2.3.6 *Nechť A je modul, potom*

- *hodnota* fan-in(A) *je součet lokálních toků informací do modulu A a počet datových struktur, ze kterých modul A čte informace.*
- *hodnota* fan-out(A) *je součet lokálních toků informací z modulu A a počet datových struktur, jejichž hodnoty modul A mění.*

Pokud budeme za modul považovat funkci, je neformální interpretace této metriky následující: fan-in(A) je počet funkcí, které funkce A volá a počet datových struktur, ze kterých čte informace a fan-out(A) je počet funkcí, jenž funkci A volají a počet datových struktur, které A modifikuje.

Autoři tvrdili, že se složitost jedné funkce skládá ze složitosti jejího kódu a také ze složitosti vztahů funkce s jejím okolím. Proto složitost funkce definovali následovně

$$\text{length} \cdot (\text{fan-in} \cdot \text{fan-out})^2.$$

Parametr length je možné spočítat výše uvedenými metrikami pro měření délky kódu, například LOC. Je možné použít i metriku měřící složitost kódu, například cyklomatickou složitost. Výraz $\textit{fan-in} \cdot \textit{fan-out}$ vyjadřuje počet kombinací vstupních zdrojů a výstupních cílů. Tento výraz má přiřazenou vyšší váhu díky mocnině. Autoři totiž věří, že je vztah funkce k jejímu okolí důležitější než složitost jejího kódu [14].

Tato metrika nevychází z lexikální složitosti programu, proto by korelace s ostatními metrikami měla být nízká. V [15] je uvedeno, že právě s cyklomatickou složitostí a Halsteadovým počtem tokenů má tato metrika nízkou míru korelace.

2.4 Objektově orientované metriky

Objektově orientované programování je založené na třech hlavních pilířích, totiž zapouzdření, polymorfismus a dědění. Většina objektově orientovaných metrik se alespoň jedním z těchto aspektů zabývá. Některé měří jiné vlastnosti. Mohou se zabývat mírou provázanosti s ostatními objekty, tedy jak je jeden objekt na ostatních objektech závislý. Také je možné analyzovat, jaká je složitost nebo velikost nějakého objektu.

2.4.1 Chidamber-Kemererovy metriky

Chidamber a Kemerer definovali následující metriky [16]:

Weighted Methods Per Class (WMC)

Definice 2.4.1 *Uvažme třídu s metodami* $M_1, \dots, M_n$*. Nechť* $c_1, \dots, c_n$ *jsou popořadě složitosti metod* $M_1, \dots, M_n$*. Potom*

$$\mathrm{WMC} = \sum_{i=1}^{n} c_i.$$

Je to tedy suma složitostí všech metod v dané třídě. Autoři tuto metriku zdůvodňují mimo jiné tím, že čím vyšší je složitost třídy, tím více usilí a času musí být vynaloženo na udržování a rozšiřování dané třídy. Zároveň třída A, která dědí od třídy B, má z definice *WMC* minimálně tak velkou složitost jako B. Je to kvůli tomu, že dědí všechny metody nadtřídy. Toto platí, pokud podtřída žádnou metodu nepřepisuje.

Složitost c_i můžeme měřit libovolnou metrikou, která měří vlastnost funkce. Můžeme tedy použít cyklomatickou složitost, NPATH apod. Je možné použít i LOC. To by v jistém smyslu reprezentovalo velikost třídy. Triviální případ je, že každé $c_i = 1$. Hodnota metriky by se poté rovnala počtu metod.

Depth of Inheritance Tree

Tato metrika uvádí počet tříd v dědícím stromě mezi konkrétní třídou a kořenovou třídou. Kořenová třída je třída, která od žádné další nedědí. V jazycích, které umožňují vícenásobnou dědičnost, se metrika definuje jako maximum délky ze všech těchto stromů. Čím větší toto číslo bude, tím se může zvětšit složitost této třídy, protože dědí metody od tříd předchozích.

Number of Children

Hodnota metriky pro třídu A je rovna počtu tříd, které přímo dědí od třídy A. Čím vyšší je tato metrika, tím větší je znovupoužití třídy A. Zároveň je nutná opatrnost při změně rozhraní. Je pravděpodobné, že bude nutné upravovat i její potomky. Je vhodné se důkladně věnovat otestování takové třídy. Potenciální chyba by totiž mohla postihnout i všechny její potomky.

Coupling between Object Classes

Hodnota metriky určuje počet tříd, se kterými je třída A provázaná. Třída A je provázaná s B, pokud metody třídy A volají metody třídy B nebo přistupují k členským proměnným třídy B. Čím menší tato provázanost je, tím méně je třída závislá na ostatních a tím jednodušší je její znovupoužití. Třídy s vysokou provázaností je také nutné důkladněji testovat.

Pokud bude potřeba část třídy A měnit, je nutné dávat pozor na třídy, které jsou s ní provázané. Změny v A mohou mít dopad na třídy, které třídu A používají. Tato metrika pouze počítá, kolik provázaných tříd existuje. Všechny

provázanosti tedy vnímá jako stejně silné. Toto ale nemusí vždy odpovídat realitě [17].

Response for a Class

Definice 2.4.2 *Nechť třída A má metody $M = \{M_1, \dots, M_n\}$. Potom*

$$S = M \cup \left(\bigcup_{\forall i} R_i \right)$$

kde R_i je množina metod, které volá M_i.

Výsledná metrika je potom definována jako $|S|$. Pokud pro volání jedné metody je nutné zavolat mnoho dalších, zvyšuje se její složitost. Může se tedy ztížit i její testování a ladění.

Lack of Cohesion in Methods

Lack of Cohesion in Methods (LCOM) zkoumá absenci soudržnosti v dané třídě. Analyzuje, jak moc různé dvojice metod využívají stejné atributy.

Definice 2.4.3 *Nechť C je třída s metodami $M_1, \dots, M_n$. Nechť I_i je množina atributů z třídy C, jenž metoda M_i používá.*
Nechť $P = \{\{i,j\} \mid I_i \cap I_j = \emptyset\}$ a $Q = \{\{i,j\} \mid I_i \cap I_j \neq \emptyset\}$. Pokud $\forall I_i = \emptyset$, definujeme $P = \emptyset$.
Samotnou metriku potom definujeme následovně:

$$LCOM = \begin{cases} |P| - |Q| & \text{pokud } |P| \geq |Q| \\ 0 & \text{jinak} \end{cases}$$

Neformálně řečeno, dvě metody si jsou podobné, pokud používají alespoň jeden stejný atribut. Pro všechny možné dvojice metod ve třídě je LCOM rovné počtu dvojic, které si nejsou podobné mínus počet dvojic, které si jsou podobné.

Pokud má třída vysokou soudržnost (tedy nízkou hodnotu této metriky), znamená to, že intenzivně pracuje s atributy a různé metody spolu logicky souvisí. U tříd s nízkou soudržností se zvyšuje složitost, a tedy i pravděpodobnost výskytu chyby. Takové třídy bývá vhodné rozdělit na více podtříd.

Tato metrika byla v některých případech označena jako nedostačující [18]. Metrika totiž nemá žádnou horní mez. Proto není jednoduché poznat, jaká hodnota je a jaká není žádoucí. Další důvod je to, že existují nepodobné třídy, u kterých je tato metrika rovná nule. Soudržnost v těchto třídách je ale jiná. I přesto byla velmi používaná [19].

2.4.2 Míra soudržnosti ve třídě

LCOM měřilo chybějící soudržnost ve třídě. Tato metrika naopak měří míru soudržnosti. Zároveň se snaží vyřešit dříve zmíněné nedostatky. Jedna z takových metrik je *Sensitive Class Cohesion Metric* (*SCOM*) [18].

Definice 2.4.4 *Nechť I_k je množina atributů, ke kterým přistupuje metoda k. Potom*

$$CC_{ij} = \frac{|(I_i \cap I_j)|}{min(|I_j|, |I_i|)}$$

vyjadřuje sílu propojení mezi dvěma metodami i a j.

Zároveň dvojicím metod, které používají více atributů, přiřadíme vyšší váhy:

$$\alpha_{ij} = \frac{|(I_i \cup I_j)|}{a},$$

kde a je počet atributů.

Definice 2.4.5 *Nechť existuje ve třídě n metod, potom*

$$LCOM = \frac{2}{n(n-1)} \sum_{i=1}^{n} \sum_{j=i+1}^{n} \alpha_{ij}\ CC_{ij},$$

kde CC_{ij} je síla propojení metod z definice 2.4.4. Výraz $\frac{2}{n(n-1)}$ je inverze počtu dvojic metod, tedy $\binom{n}{2}^{-1}$.

Metrika je normalizována do intervalu $\langle 0, 1 \rangle$. Hranice intervalu reprezentují dva extrémy.

1. $SCOM = 0 \implies$ Neexistuje pár metod, který by používal stejné atributy.

2. $SCOM = 1 \implies$ Všechny metody používají všechny atributy.

2.4.3 Lorenz and Kidd

Tyto metriky se rozdělují do dílčích kategorií. Metriky jsou poměrně přímočaré. Obvykle se jedná o počet výskytu nějakých vlastností, např. počet metod [20, 21].

Metriky orientované na velikost

Tyto metriky různě popisují velikost třídy.

1. Počet veřejných method
2. Počet metod
3. Počet veřejných atributů
4. Počet atributů

Vyšší počet metod může znamenat, že třída provádí příliš věcí. Mohlo by pomoci rozdělit třídu do více menších tříd. Pokud je vyšší počet veřejných atributů, je možné, že třída porušuje princip zapouzdřenosti.

Metriky orientované na dědičnost

Tyto metriky popisují vlastnosti tříd, které dědí od jiných.

1. Počet poděděných metod
2. Počet přepsaných metod
3. Počet nových metod

Metoda je nová, pokud neexistuje se stejným názvem a parametry v rodičovské třídě.

Ostatní metriky

Popisují obecné vlastnosti tříd.

1. Průměrný počet parametrů na metodu
2. Počet spřátelených tříd

Spřátelené třídy porušují princip zapouzdření. Měly by být používány s opatrností. Pokud jsou někde spřátelené třídy, je dobré jim věnovat pozornost [22].

2.4.4 Metrics for Object Oriented Design (MOOD)

Následující metriky [23] opět popisují vlastnosti objektově orientovaného návrhu. Autoři se mimo jiné snažili, aby metriky byly nezávislé na jazyku a systému a aby se daly jednoduše počítat. Naměřené hodnoty jsou z rozsahu 0 až 1, kde 0 značí, že se daná vlastnost vůbec neobjevuje, 1 naopak značí úplný výskyt této vlastnosti.

Method Hiding Factor (MHF) a Attribute Hiding Factor (AHF)

Jedná se o dvojici metrik měřící míru zapouzdřenosti. První z nich uvádí počet privátních metod ku všem metodám, druhá počet privátních atributů ku všem atributům. Nechť existují třídy $C_1, \dots, C_n$. Počet metod ve třídě C_i je dán rovností

$$M_d(C_i) = M_v(C_i) + M_h(C_i),$$

kde $M_v(C_i)$ je počet viditelných (public) metod a $M_h(C_i)$ je počet schovaných (private a protected) metod ve třídě C_i. Samotnou metriku definujeme následovně:

$$MHF = \frac{\sum_{i=1}^{TC} M_h(C_i)}{\sum_{i=1}^{TC} M_d(C_i)},$$

kde TC je celkový počet tříd. Analogicky se definuje i AHF, jen metody nahradíme atributy.

Method Inheritance Factor (MIF)

Metrika popisuje míru dědičnosti, konkrétně jaký je poměr zděděných metod oproti všem metodám.

$$MIF = \frac{TM_i}{TM_a},$$

kde TM_i je počet zděděných metod a TM_a je počet všech metod. Formálněji zde [23]. Pokud je toto číslo nízké, je možné, že není adekvátně využíván princip znovupoužitelnosti.

Polymorphism Factor (PF)

Metrika uvádí, jak moc je využit polymorfismus. Metrika je definována následovně:

$$PF = \frac{\sum_{i=1}^{TC} \sum_{j=1}^{DC(C_i)} M_o(C_j)}{\sum_{i=1}^{TC} M_d(C_i) \cdot DC(C_i)},$$

kde $M_d(C_i)$ je počet metod, $M_o(C_i)$ je počet přepsaných metod a $DC(C_i)$ je počet potomků třídy C_i a TC je celkový počet tříd.

2.5 Existující analyzátory softwarových metrik

2.5.1 Open-source

CCCC

CCCC [24] je měřič metrik vytvořený Timem Littlefairem pro jeho dizertační práci. Umí zpracovat zdrojové kódy v jazyce C, C++ a Java. Program měří následující metriky:

- Počet řádků
- Cyklomatická složitost
- Hustota komentářů
- Fan-in, Fan-out
- Henry and Kafura's/Shepperd Measure
- Weighted Methods Per Class
- Children Count
- Depth of Inheritance Tree
- Coupling between Objects
- Rejected Lines

Poslední metrika je počet řádků, které parser nebyl schopný zpracovat. Vyšší hodnota této metriky může znamenat, že naměřené výsledky nebudou příliš přesné. Jedná se o konzolovou aplikaci. Je ji tedy možné použít v automatizovaných systémech. Výsledná měření exportuje ve formátu HTML nebo XML, který se hodí pro další strojové zpracování. Je možné využít internetový prohlížeč pro přehledné zobrazení naměřených metrik.

PMCCabe

PMCCabe [25] je jednoduchá konzolová aplikace na měření metrik v jazyce C a C++. Je implementovaný v jazyce C. Byl vytvořen v době, kdy byla priorita mít software rychlý. Následek toho je, že není příliš rozšiřitelný. Měří počet řádků, počet příkazů a cyklomatickou složitost funkce.

Obsahuje dvě verze měření cyklomatické složitosti. Rozdíly mezi nimi spočívají ve `switch` příkazu. První z nich zvyšuje za každý `switch` složitost o 1 nezávisle na počtu `case` příkazů. Druhá zvyšuje složitost za každý `case` příkaz.

SourceMonitor

SourceMonitor [26] je program, který dokáže měřit metriky analyzující funkce pro programy napsané v jazyku C, C++, C#, VB.NET, Java nebo Delphi. Jedná se o GUI aplikaci. Umožňuje export ve formátu XML nebo CSV. Je dostupná pouze pro operační systém Windows.

U každé funkce měří cyklomatickou složitost, počet příkazů, maximální hloubku a počet funkcí, jenž tato funkce volá. Poté měří i další metriky, které shrnují celý projekt. Jedná se o počet řádků a příkazů celého projektu, procento řádků, na nichž se nachází komentáře atd. Funguje na bázi checkpointů, které ukazují vývoj metrik projektu v čase.

2.5.2 Proprietární

Klocwork

Klocwork [27] je komerční statický analyzátor kódu pro zdrojové kódy v jazyku C, C++, C# nebo Java. Kromě metrik obsahuje i detekci tzv. „code smells[3]". Je možné ho integrovat do IDE nebo k sestavovacímu procesu.

Resource Standard Metrics

RSM [28] je multiplatformní měřič metrik pro zdrojové kódy v jazyku C, C++, Java nebo C#. Dle autorů je velmi rychlý a dokáže zpracovat spoustu zdrojových kódů v rozumném čase. Jedná se o konzolovou aplikaci. Je také přizpůsobená na integraci do některých IDE. Dokáže měřit LOC a různé její druhy, cyklomatickou složitost a další. Nabízí i metriky, které sledují vývoj počtu řádků v čase z různých měření. Software toho nabízí kromě metrik mnohem více. Je možné si ho vyzkoušet na 20 souborů zadarmo.

Testwell CMT++

TestWell CMT++ [29] je software od firmy Verifysoft. Zvládá jazyky C, C++ a C#. Od stejné firmy je i měřič pro jazyk Java. Obsahuje měření následujících metrik: Lines of Code, cyklomatická složitost, Halsteadovy metriky, Maintainability Index a Max Nesting Depth. Také se pyšní rychlostí zpracování. Nabízí GUI, které metriky přehledně zobrazí.

[3]Jedná se o kód, který pravděpodobně značí chybu v návrhu. Příklad může být použití příkazu `goto`, dynamického přetypování nebo spřátelených tříd.

Kapitola 3

Clang

3.1 Motivace

Abychom mohli naměřit metriky programů, bylo by vhodné mít zdrojové kódy ve formě, ve které se s nimi bude lépe pracovat. Jedna z možností by bylo vytvořit vlastní analyzátor textu (parser), a to ručně nebo využít nějaké existující nástroje[4]. Software, který potřebuje mít zdrojové kódy zpracované, již existuje. Pokusíme se jeho část k tomuto účelu využít.

3.2 Překladače

Počítač nedokáže přímo spustit zdrojový soubor, proto je nutné ho nejdříve přeložit[5] do strojového kódu. Software, který provádí tento překlad, se nazývá *překladač*. Samotné překladače se skládají z více částí. Na obrázku 3.1 se nachází jednoduché schéma překladače[6].

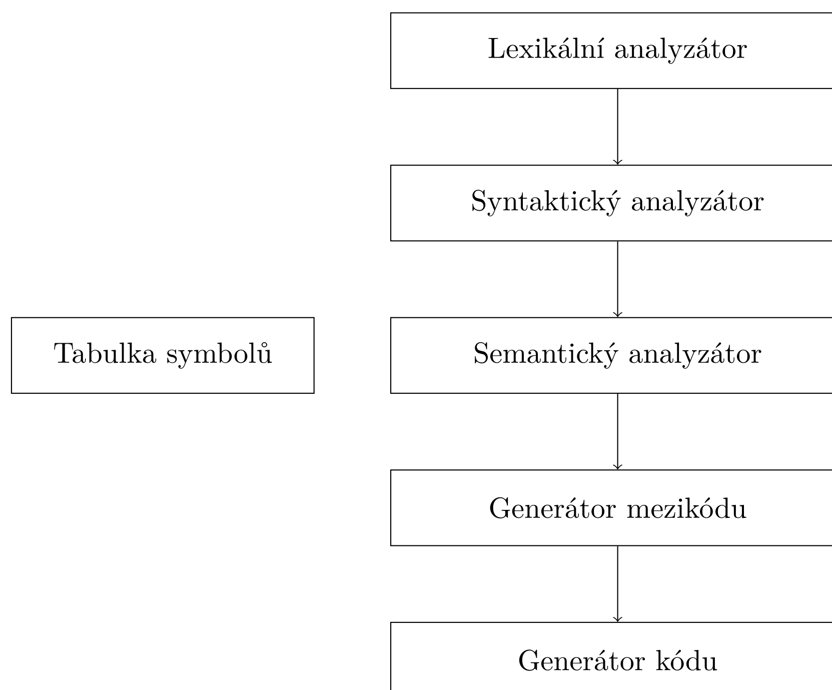
První čtyři bloky patří do tzv. *frontendu* překladače, protože jsou závislé na vstupním programovacím jazyku. Generátor kódu potom patří do tzv. *backendu* překladače, ten je závislý na konkrétní platformě. Frontend ze zdrojového kódu vytvoří tzv. *mezikód*. Backend přijímá tento mezikód a vygeneruje strojový kód pro danou platformu.

Výhoda tohoto přístupu je ta, že pro jednu platformu stačí vytvořit jeden backend. Poté stačí pro různé programovací jazyky napsat různé frontendy. Toto samozřejmě funguje i obráceně. Pokud chceme vytvořit překladač pro jeden jazyk na více platforem, stačí vytvořit jen jeden frontend. Nemusíme pro každou dvojici jazyk-platforma psát nový překladač.

[4] Např. YACC a Bison

[5] Nebo interpretovat jiným programem.

[6] Schéma úplně neodpovídá reálným překladačům. Jsou například opomenuty části, které se věnují optimalizaci.



Obrázek 3.1: Schéma jednoduchého překladače.

3.2.1 Frontend překladače

Lexikální analyzátor

Seskupuje jednotlivé znaky do smysluplných sekvencí neboli tokenů. Tokeny s sebou mohou nést i nějakou jinou informaci, například u čísla to může být jeho hodnota. Uvažme následující příklad v jazyce C:

```
foo = bar(1 + 2);
```

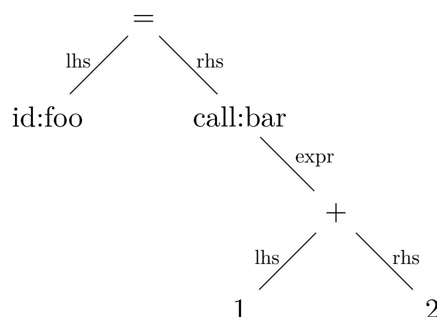
Lexikální analyzátor by tuto sekvenci na tokeny mohl převést následovně:

```
<id:"foo"> <assignment-operator> <id:"bar">
<left-bracket> <int-number:1> <plus-operator>
<int-number:2> <right-bracket> <semicolon>
```

Následující jednotka překladače pracuje s těmito tokeny.

Syntaktický analyzátor

Ze sekvence tokenů vytváří reprezentaci, která odráží gramatickou strukturu jazyka. Typicky to bývá nějaká stromová reprezentace. Obsahuje tedy například i závorky, dvojtečky, středníky a podobně.



Obrázek 3.2: Zjednodušený příklad abstraktního syntaktického stromu.

Sémantický analyzátor

Sémantický analyzátor pracuje se syntaktickým stromem a vytváří tzv. abstraktní syntaktický strom (AST). Ten lépe popisuje logickou strukturu jazyka (narozdíl od gramatické). Příklad zjednodušeného stromu pro předchozí sekvenci tokenů lze najít na obrázku 3.2. Interpretace tohoto stromu je následující: do identifikátoru *foo* máme přiřadit výsledek volání funkce s identifikátorem *bar*, jejíž argumentem je operátor +, jehož levá strana je 1 a pravá je 2.

Ostatní

Ostatní části překladače pro nás nejsou relevantní. Vášnivý čtenář nechť se podívá na [30], kde najde podrobný popis, jak překladače fungují. S AST by se pro měření většiny metrik pracovalo lépe než jen se zdrojovým kódem jakožto sekvencí znaků.

3.3 Clang a jeho rozhraní

Jazyk C++ má velmi složitou gramatiku. Vytvoření vlastního frontendu by zabralo velké množství času. Využijeme LLVM [31], což je kolekce různých částí překladačů a obecně kolekce nástrojů. Z této kolekce využijeme nástroj Clang [1]. Je to frontend k jazyku C, C++ a Objective C. Generuje mezikód pro LLVM-CORE, který slouží jako backend pro LLVM kolekci.

Clang je možné použít jako knihovnu. Obsahuje rozhraní, díky kterým je možné se syntaktickým stromem jednoduše manipulovat. Jedná se o *LibClang*, *Clang Plugins* a *LibTooling*. Existují samozřejmě i jiné překladače jako například GCC (GNU Compiler Collection) nebo MSVC (Microsoft Visual C++). Ty ale žádná taková rozhraní neposkytují.

Oficiální dokumentace Clangu je vygenerovaná z dokumentačních komentářů zdrojového kódu. Samotní vývojáři doporučují pro pochopení studovat zdrojový kód a podívat se na existující programy, které toto rozhraní využívají. Je k dispozici také několik krátkých oficiálních návodů.

Tyto rozhraní využívá i Clang pro některé nástroje. Jeden z nich je například clang-tidy[7]. Ten za pomocí statické analýzy programu nahlásí chyby nebo nějaké obecné doporučení, které by mohlo program zlepšit (pokud například metoda nepoužívá žádné členské proměnné, navrhne z ní udělat statickou metodu).

3.3.1 AST Clangu

Před popisem samotných rozhraní je nejprve vysvětleno, jak vypadá AST Clangu. Pokud je dále v textu zmíněno AST, je tím myšleno AST Clangu. Interně jsou jednotlivé uzly AST reprezentovány jako třídy. Tyto třídy nemají jednoho společného předka. Většina těchto tříd má jednoho z následujících: `Decl`, `Stmt` a `Type`. `Decl` reprezentuje deklarace nebo definice. `Stmt` reprezentuje různé příkazy, například `if`. Reprezentuje také výrazy (expressions). `Type` jsou datové typy. Existují i další základní třídy jako například `QualType`, který reprezentuje typ kvalifikátoru (např. u `const int a;` by `QualType` reprezentoval `const`).

AST se prochází pomocí metod, které mají dané třídy k dispozici. Uvažme například příkaz `if`. Ten je reprezentovaný třídou `IfStmt`. Ta má mimo jiné k dispozici metody `getThen()`, `getElse()` a `getCond()`, díky kterým je možné přistoupit k podmínce a tělu samotného příkazu.

AST není strom, je to obecný graf. Například z uzlu `CXXRecordDecl`, který reprezentuje (nejen) C++ třídu, můžeme přistupovat ke všem ostatním třídám, které jsou s ní spřátelené nebo ke třídám, ze kterých tato třída dědí. Clang API obsahuje různá rozhraní, která jsou schopná projít tímto grafem jako by to byl strom.

3.4 LibClang

LibClang [32] je rozhraní v jazyce C, existují ale tzv. bindings[8] pro Python. Průchod stromem je realizován přes Visitor pattern [33]. Vývojáři se snaží toto rozhraní dělat stabilní a zpětně kompatibilní napříč verzemi. Abstrahuje AST přes tzv. *Cursors*. Ty reprezentují jednotlivé uzly AST. Toto může být i nevýhoda, tato abstrakce totiž neposkytuje kompletní přístup k AST. Pokud je takový přístup potřeba, není toto rozhraní vhodné.

3.5 Clang Plugins

Clang Plugins [34] jsou určeny pro provedení dalších akcí nad AST při samotné kompilaci. Jedná se o dynamické knihovny. Načítá je při běhu samotný

[7]Tento nástroj používá například IDE Clion.

[8]Jedná se o jakýsi „most“ mezi dvěma programovacími jazyky. Umožňuje aplikaci napsanou v jazyce A používat i v jazyce B.

překladač. Mohou být vhodné, pokud chceme mít při kompilaci nějaká speciální varování navíc (dokážou kompilaci i zastavit). Nehodí se, pokud chceme aplikaci spouštět i mimo sestavovací proces.

3.6 LibTooling

LibTooling [35] je C++ rozhraní. Jednotlivé uzly stromu jsou reprezentovány jako třídy. Obsahuje i abstrakce na manipulaci s AST. Je možné použít jakoukoli část překladače, programátor má k celému Clangu plný přístup. Vývojaři upozorňují, že se toto rozhraní může často měnit. Pro stabilní rozhraní je tedy lepší použít LibClang.

3.6.1 Průchod AST

Zde jsou uvedeny způsoby, jak je možné AST procházet. Tyto způsoby je možné různě kombinovat. Například lze najít definici funkce pomocí prvního způsobu a tělo funkce prohledat pomocí druhého způsobu.

RecursiveASTVisitor

Tento způsob využívá Visitor pattern [33] a Curiously Recurring Template pattern [36]. V Clangu existuje třída `RecursiveASTVisitor`. Ta obsahuje následující tři typy funkcí, které při průchodu používá: `Traverse*`, `WalkUpFrom*` a `Visit*`, kde `*` reprezentuje uzel, pro který je daná metoda volaná, například tedy `TraverseIfStmt` pro `if` příkaz.

Činnost této třídy je sumarizována v následujících třech bodech:

1. Projde všechny uzly v AST preorder nebo postorder průchodem. K tomu slouží metody `Traverse`.
2. Pro každý uzel zavolá metodu `WalkUpFrom`, která projde třídní hierarchií daného uzlu. Začíná na jeho dynamickém typu a končí na základních typech (`Stmt`, `Decl`, `Type`, ...) a pro každdou takovou dvojici (uzel, typ) zavolají metodu `Visit`.
3. Pro kombinaci (uzel, typ) zavolá příslušnou metodu `Visit`. Tato metoda je přímo určená pro přepsání uživatelem a může spustit libovolnou akci nad daným uzlem.

Pro tyto metody existuje uspořádání `Traverse` > `WalkUpFrom` > `Visit`. Jednotlivé druhy metod mohou volat jen ty, které jsou o jedno menší nebo stejné. Tedy například `WalkUpFrom` může volat jiné `WalkUpFrom` nebo `Visit`, nemůže volat `Traverse`.

`WalkUpFrom` nejdříve volá `WalkUpFrom` nadtřídy, až potom volá `Visit`. Nechť `A` je nadtřída `B` a tyto třídy reprezentují uzly v AST. Potom `WalkUpFromB`

nejdříve zavolá `WalkUpFromA` a až poté `VisitB`. `WalkUpFromA` by opět zavolalo `WalkUpFrom`, pokud by mělo nadtřídu, a poté `VisitA` na uzel. `Visit` tedy bude zavolán na všechny nadtřídy. V třídní hierarchii bude nejprve zavolán na nejvyšší třídu v hierarchii a postupně směrem dolu na ostatní podtřídy, až se dostane ke skutečnému typu uzlu.

Je možné vytvořit třídu, která podědí od `RecursiveASTVisitor` a uvede sama sebe do jejího šablonového argumentu. Poté, pokud nám stačí pouze navštívit dané uzly, přepíšeme příslušné `Visit` metody. Tyto metody nemusíme přepisovat pro všechny uzly, ale pouze pro ty, které nás zajímají. Pokud chceme upravovat i průchod, je možné přepsat `Traverse` a `WalkUpFrom` metody.

AST Matchers

Další způsob jsou tzv. AST Matchers. Je podobný funkcionálnímu programování. Příklad matcheru, který najde všechny operátory +, které mají na levé straně celočíselný literál s hodnotou nula:

```
binaryOperator(
    hasOperatorName("+"),
        hasLHS(integerLiteral(equals(0)))
)
```

Matcherů je opravdu velké množství. Jsou rozděleny na tři kategorie. První z nich hledají samotné uzly, v příkladu to je `binaryOperator`. Další potom specifikují vlastnosti, který nalezený uzel musí splňovat, v příkladu je to `hasOperatorName` a `equals`. Poslední druh kontroluje, zdali má nalezený uzel určitý vztah k jinému uzlu, v příkladu to je `hasLHS`.

Ke každému matcheru je možné přidat nějaký callback. To je objekt, který se zavolá, když matcher najde požadovaný uzel. Zároveň je možné nalezený uzel zachytit příkazem `bind` a s nalezeným uzlem potom v callbacku dál pracovat.

Manuální průchod

Další možnost je projít AST po hranách s využitím metod daných uzlů. Například uzel, který reprezentuje deklaraci třídy, obsahuje iterátory na všechny deklarace jeho metod a atributů. Pomocí nich je možné tyto členy projít a provést nad nimi libovolnou další akci. Tato metoda je vhodná, pokud je potřeba procházet AST a znát kontext.

Kapitola 4

Implementace

Tato kapitola popisuje implementaci měřiče metrik a pravidla, kterými se řídí při počítání jednotlivých metrik. Program využívá rozhraní LibTooling popsané v sekci 3.6. Používá i všechny typy průchodů AST, které byly zmíněny v sekci 3.6.1.

4.1 Kostra programu

Kostra programu tvoří třída, která dědí z třídy `RecursiveASTVisitor`. Ta je popsána v podkapitole 3.6.1. Tato třída navštěvuje všechny funkce a třídy. Clang ke každému uzlu AST dokáže přiřadit unikátní číselný identifikátor. Pro ukládání informací o metrikách ale nemůžeme tento identifikátor využít. Uzly totiž nemají stejný identifikátor napříč více různými zdrojovými kódy. Třídy můžeme ukládat podle jejich názvu. U metod tato informace nestačí, mohou být totiž přetěžovány. Jsou identifikovány podle názvu a datového typu jejich argumentů.

Třída vykonává postupně následující činnosti:

1. Navštíví všechny funkce, pro každou z nich vytvoří příslušné měřiče metrik a spustí je nad danou funkcí.
2. Spustí výpočty metrik, které potřebují přístup k celému AST.
3. Vyexportuje vypočtené metriky.

4.2 Měřiče metrik

Jednotlivé měřiče se dělí na dvě kategorie. První potřebují pouze tělo funkce, kterou právě zkoumají. Nepotřebují znát žádný další kontext nebo si pamatovat informace z předešlých měření. Jde například o metriky jako je cykloma-

tická složitost nebo NPATH. Pro tyto typy měřičů existuje abstraktní třída `FunctionVisitor`, která obsahuje abstraktní metodu `CalcMetrics` a různé `Export` abstraktní metody. První zmíněná metoda dostává jako argument definici funkce a provádí vlastní výpočet metrik. Druhé zmíněné metody slouží na vyexportování dat. To, jak třída provede vlastní výpočet a export metrik, je v její režii. Je ale důležité, aby tyto metody obsahovala.

Druhý typ měřičů potřebuje mít k dispozici celý AST. Jedná se například o metriku fan-out. Tyto metriky mají velmi podobné rozhraní jako první typ měřičů. Metoda `CalcMetrics` ale na vstupu očekává celý AST. Metody na export mají jako argument identifikátor funkce nebo třídy, která by se měla vyexportovat.

4.3 Implementace jednotlivých měřičů

Zde jsou popsány implementace jednotlivých metrik. Není to tak, že by pro každou metriku existovala právě jedna třída, která ji měří. Některé třídy měří více metrik najednou.

4.3.1 Lines of Code

Clang obsahuje třídu `SourceManager`. Tato třída obstarává jednotlivé zdrojové soubory, se kterými kompilátor momentálně pracuje. Clang ji například používá při chybových hláškách. Dokáže totiž pro každý uzel AST určit, v jakém souboru a na jakém řádku a sloupci se nachází. Můžeme ji tedy využít na nalezení začátku a konce těla funkce. Tento způsob nám vrátí přesný počet řádků. Jsou tedy započítány i prázdné řádky a komentáře.

4.3.2 Počet příkazů

Spočítat počet příkazů už není tak jednoduché. Přímočarý postup je spočítat všechny uzly, jejichž nadtřída je třída `Stmt`. Ta byla popsána v sekci 3.3.1. Důležité je, že i třída, která reprezentuje `Expr`, je podtřída `Stmt`. Uvažme následující příkaz:

```
std::string s = "Hello";
```

AST pro tento příkaz je možno najít na obrázku 4.1.

Tento příkaz by se počítal za osm místo jednoho. Druhá možnost je nepřičítat uzly, které jsou podtřídou `Expr`. V tom případě by byl problém s příkazy jako je volání funkce nebo přiřazovací operátor. Ty jsou totiž podtřídou `Expr`.

Následující způsob se ukázal jako efektivní. Příkazy, které typicky obsahují nějaké další příkazy, jako je například `if`, `while` nebo i samostatné složené závorky, nazýváme *složené příkazy*. U těchto složených příkazů spočítáme, kolik mají v AST dětí. Pokud některé z těchto dětí je také složený příkaz, opět spočítáme počet jeho dětí. Je nutné vynechat ty děti, které jako příkazy

```
DeclStmt
  `-VarDecls
    `-ExprWithCleanups
      `-ImplicitCastExpr
        `-CXXConstructExpr
          |-ImplicitCastExpr
          | `-StringLiteral
          `-CXXDefaultArgExpr
```

Obrázek 4.1: AST Clangu pro výraz `std::string = "Hello"`.

počítat nechceme. Například příkaz `for` má čtyři děti, první tři jsou popořadě inicializace iterační proměnné, podmínka a inkrementace. Čtvrtý z nich je tělo tohoto příkazu. První tři je tedy nutné přeskočit.

4.3.3 Cyklomatická složitost

Počítání cyklomatické složitosti je poměrně jednoduché. Metrika nepracuje s kontextem, je tedy možné použít AST Matchers. Jednoduše se v těle funkce spočítá počet uzlů, které se shodují s danými matchery. Následující příkazy jsou počítány:

- for
- while
- ternární (conditional) operátor
- for range
- case
- do
- if
- and (`&&`)
- or (`||`)
- catch
- goto

Jiné měřiče, např. CCCC [24], započítavají i příkazy `return`, `continue`, `break`. Tyto příkazy sice zvyšují složitost programu, ale nevětví ho. Proto je implementace nepočítá.

4.3.4 Halsteadovy tokeny

Halsteadovy tokeny byly implementovány přes AST Matchers. Tento přístup bohužel nebyl příliš úspěšný. AST obsahuje spoustu uzlů navíc a je poměrně obtížné vybrat relevantní části. Pokud například metoda přistupuje ke své členské proměnné, v AST se implicitně objeví *this*. Toto se samozřejmě dá zachytit. C++ v kombinaci s AST obsahuje poměrně dost takových situací.

Byla tedy implementovaná metrika, která se snaží zachytit počet operátorů a počet operandů. Jako operátory se počítají příkazy, které kontrolují tok programu, všechny klasické operátory (`+`, `<<`, `*`, `new` apod.) a volání funkcí. Operandy jsou literály, proměnné a `goto` návěští. Navíc je počítán jen celkový počet výskytů, nikoli unikátní.

Tento způsob počítání ale nepočítá tokeny, nýbrž část uzlů z AST. Pravděpodobně lepší způsob je převést zdrojový kód na sekvenci tokenů (nikoli na AST). Tyto tokeny potom rozdělit na operátory a operandy a spočítat je. Takto bychom počítali čistě syntaktickou stránku programu. Implementovaná metrika tedy vlastně neodpovídá Halsteadovým metrikám a její export není součástí implementovaného programu.

4.3.5 Maximální hloubka

Tato metrika nebyla zmíněna v rešerši. Nicméně byla implementována jako možný ukazatel zbytečně mnoho vnořených příkazů. Na první pohled zvláštní chování je u příkazu `else`. Následující kód

```
if(cond1) { ... }
else if(cond2)
{
    ...
}
```

má maximální hloubku rovnou 3. Je totiž interpretován následovně:

```
if(cond1){ ... }
else
{
    if(cond2) { ... }
}
```

Tento přístup je zvolen kvůli konzistenci. Pokud by v `else` větvi byl jakýkoli jiný složený příkaz než `if`, zvýšení hloubky by přirozeně dávalo smysl.

Metrika byla implementována přes ruční průchod AST, je totiž nutný kontext při počítání hloubky. Byla zvolena podobná strategie jako u počtu příkazů, viz sekce 4.3.2. Hloubka je spočítána rekurzivně pro každý složený příkaz.

4.3.6 NPATH

NPATH je poněkud složitější než cyklomatická složitost, při průchodu AST je totiž nutný kontext. Byla použita třída `StmtVisitor` z Clangu. Tato třída obsahuje metody `Visit` pro každý uzel, který dědí z třídy `Stmt`. Dokáže správně přeposlat danou třídu podle jejího dynamického typu. Druhá možnost by bylo využít dynamické přetypování pro všechny typy příkazů, u kterých je nutné NPATH počítat. Pravidla pro počítání se řídí sekcí 2.3.2. Výjimku z těchto

pravidel tvoří `case` příkazy. Pokud je jich více za sebou, je počítán jen poslední.

Jsou přidány příkazy, které řeší výjimky (`catch`). Ty totiž větví kód, pokud se výjimka v bloku vyhodí. Pravidla pro počítání výjimek jsou následující:

```
try {
    <try-range>
} catch(exc_1) {
    <catch-range-1>
}
...
catch(exc_n) {
    <catch-range-n>
}
```

$$NP(\text{try}) = NP(\langle\text{try-range}\rangle) + \sum_{i=1}^{n} NP(\langle\text{catch-range-i}\rangle)$$

Při použití této definice nebudou naměřené hodnoty přesně odpovídat počtu acyklických cest. Výjimka totiž může být vyhozena kdykoli v `try` bloku. Takový stav je ale obtížné zachytit. Samotné hodnoty měření by navíc byly velmi vysoké, pokud by se takto cesty počítaly. Takto se projeví zvýšená složitost při použití výjimek a naměřená složitost nepřehluší složitost ostatních příkazů.

Samotné počítání je jednoduché. Sekvenční příkazy mají NPATH hodnotu jedna. U ostatních příkazů se vždy navštíví všechny části, které pravidla specifikují (např. u `if` se navštíví podmínka a tělo) a výsledek se připočítá do výsledné hodnoty. Takto se rekurzivně projde celé tělo funkce.

4.3.7 Fan-in a Fan-out

Původní verze této metriky počítá počet modulů. Jelikož budou primárně měřeny krátké programy, je tato metrika interpretována dvojím způsobem. Jako modul lze vnímat funkci nebo třídu. Přístup, kde modul je jedna třída, je zmíněn v sekci 4.3.8. Zde je popsán přístup, kde modul je funkce. Fan-in říká, kolik funkcí momentálně měřená funkce volá, a fan-out říká, kolikrát je volána.

Z empirických důvodů se u fan-in nepočítá volání přetěžovaných operátorů. Často totiž bývají používány na přístupy ke kontejnerům, chytrým ukazatelům nebo pro vstupní a výstupní proudy ze standardní knihovny. Také není počítáno volání konstruktorů a destruktorů.

Na rozdíl od předchozích metrik potřebuje tato přístup k celému AST, nikoli jen k jedné funkci. Třída navštěvuje všechny definice funkcí. V nich poté díky AST Matchers najde jakékoli volání. Pro každé takové volání inkrementuje fan-in současné funkce a fan-out funkce, která je volaná.

Tato metrika nevrací přesné výsledky u hodnoty fan-out, pokud je používán polymorfismus. Z analýzy zdrojového kódu je obtížné zjistit, na jaké třídě

je metoda opravdu volaná. V takových případech nejsou naměřené hodnoty správné.

4.3.8 Objektově orientované metriky

Počítání objektově orientovaných metrik má na starosti jedna třída. Kombinuje všechny tři možnosti procházení AST, které byly zmíněny v sekci 3.6.1. Potřebuje přístup k celému AST, protože u metrik jako je soudržnost (coupling) musí mít povědomí o všech třídách.

Počet dětí a délka dědičného stromu

U každé třídy je možné jednoduše přistoupit ke třídám, ze kterých současná třída dědí. K počtu dětí u těchto tříd připočteme jedničku. Třída, která z žádné třídy nedědí, má délku dědičného stromu 0. Ostatní třídy mají délku stromu rovnou maximu z délky stromu tříd, ze kterých dědí, plus jedna. Takto se délka jednoduše rekurzivně spočítá.

Počet metod a atributů

Jednotlivé deklarace tříd obsahují iterátory na jejich metody a atributy. Spočítat jejich počet je tedy triviální. Zároveň i obsahují indikátor jejich úrovně zapouzdření a zdali přepisují jinou metodu. Není tedy problém spočítat veřejné či privátní atributy a metody.

Lack of Cohesion

V první řadě je potřeba určit, zda jsou dvě metody podobné (musí používat alespoň jeden stejný atribut). Potřebujeme si tedy u každé metody poznamenat názvy atributů, které používají. To lze jednoduše najít pomocí AST Matchers. Nalezené atributy si pro každou metodu uchováme tak, abychom odstranili duplicity. Poté můžeme porovnat všechny dvojice funkcí a spočítat metriku z definice 2.4.3.

Coupling, Fan-in a Fan-out

Zde budeme modul vnímat jako jednu třídu. Fan-in třídy A je počet tříd, jejíchž metody jsou třídou A volány. Fan-out třídy A je počet tříd, jenž volají metody třídy A. Coupling (soudržnost) je součet těchto dvou metrik. Pokud existuje dvojice, která si navzájem volá metody, počítá se jen jednou.

Tyto metriky se měří poměrně jednoduše. Stačí v metodách dané třídy vyhledat volání metod a uložit si název třídy, ke které volaná metoda přísluší. Výsledek je unikátní počet takto uložených názvů. Do tohoto počtu není započítána třída, pro kterou se metrika počítá.

4.4 Detaily implementace

AST Matchers

AST Matchers je možné použít na celý AST nebo jen na jeden daný uzel. Není možné je využít na podstrom AST. Tento problém by se dal vyřešit poměrně jednoduchým zásahem do zdrojového kódu Clangu. Takové řešení není moc přenositelné, protože každý uživatel, který by si chtěl program sám sestavit, by musel také přepsat zdrojový kód Clangu. Byl zvolen jiný způsob, a to vytvoření třídy, která dědí od `RecursiveASTVisitor` a navštěvuje podstrom AST. Na každý uzel v tomto AST poté zavolá předaný AST Matcher. Takové řešení je funkční a přenositelné, jen přidává další vrstvu navíc.

Makra

Ještě před samotným měřením jsou makra expandována. Pokud například kód obsahuje podmíněný blok, který podmínku nesplňuje, bude tento kus kódu při měření ignorován. Také jsou nahrazeny všechny `#define` příkazy jejich těly. Po expanzi maker mohou být některé funkce, které jsou na první pohled poměrně jednoduché, velmi složité.

Hlavní nevýhoda spočívá v tom, že některé prvky standardní knihovny bývají implementovány jako makra. Jedná se například o makro `assert`. To může v některých implementacích obsahovat příkaz `if`, tím pádem zvyšuje NPATH a cyklomatickou složitost.

Výhoda spočívá v tom, že jsou expandovány `#include` direktivy. Pokud je projekt rozdělen na hlavičkové a implementační soubory, budou počítány i hlavičkové soubory, které jsou za pomocí této direktivy přidány. Vynechány jsou hlavičkové soubory systému, tedy například standardní knihovna.

4.4.1 Korektnost zdrojového kódu

Pro spolehlivá měření musí být analyzovaný zdroj korektní z pohledu gramatiky a sémantiky jazyka. AST není možné na nekorektních programech validně sestavit. Na nekorektní programy je možné měřič spustit, výsledek ale není definovaný.

4.5 Porovnání s ostatními měřiči

Tabulka 4.1 ukazuje, které metriky implementovaný software měří ve srovnání s open-source alternativami.

	Práce	CCCC	PMCCabe	SourceMonitor
Lines of Code	✓	✓	✓	✓
Statements Count	✓	✗	✓	✓
Comment Density	✗	✓	✗	✓
Cyclomatic Complexity	✓	✓	✓	✓
NPATH	✓	✗	✗	✗
Maximum Depth	✓	✗	✗	✓
Fan-in	✓	✗	✗	✓
Fan-out	✓	✗	✗	✗
Weighted Methods Per Class	✗	✓	✗	✗
Number of Children	✓	✓	✗	✗
Length of Inheritance Tree	✓	✓	✗	✗
Fan-in, Fan-out (OOP)	✓	✓	✗	✗
Cohesion	✓	✗	✗	✗

Tabulka 4.1: Přehled metrik, které jednotlivé programy měří

Kapitola 5

Testování

Program je primárně určen pro menší školní programy, byl ale otestován i na rozsáhlejším open-source projektu. Implementace byla zároveň porovnána oproti existujícím open-source řešením. Jelikož open-source varianty neměří dostatečně spolehlivě nebo mají jinou definici metrik (například CCCC počítá cyklomatickou složitost jinak, viz sekce 4.3.3), nemohly být použity pro naměření referenčních hodnot. Některé metriky navíc open-source varianty vůbec neměří (např. NPATH).

Program byl zároveň otestován jednotkovými testy. Objektově orientované metriky byly otestované jen jednotkovými testy. Jen počet dětí a délka dědičného stromu byly otestovány na velkém projektu. Objektové fan-in a fan-out byly otestovány na pár třídách ve velkém projektu zmíněném v následující kapitole.

5.1 Testování na velkém projektu

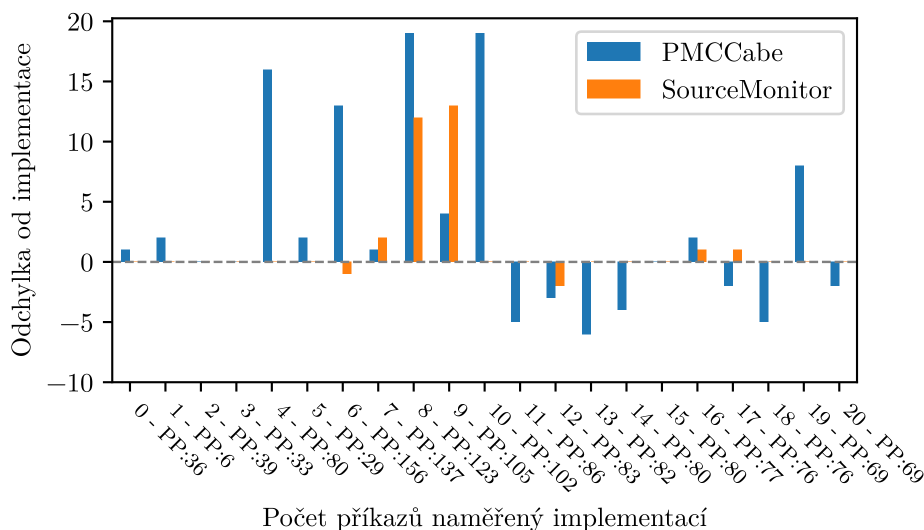
Byl vybrán C++ projekt Box2D [37], který obsahuje více než 700 funkcí a zhruba 120 tříd. Správnost měření byla ověřena pouze na malé podmnožině. Konkrétně bylo vybráno 21 funkcí. 15 z nich jsou funkce, které mají nejvíce příkazů. Zbytek byl vybrán náhodně. Pro tyto funkce byly metriky ručně spočítány. Výjimku tvoří metrika fan-out, u té byl využit nástroj *grep*. Implementace na této podmnožině naměřila správné hodnoty. Výsledky implementace byly rovněž porovnány s open-source programy.

5.1.1 Porovnání s ostatními měřiči

Následující kapitoly ukazují rozdíly měření u jednotlivých metrik. Hodnota jednotlivých sloupců indikuje, o kolik se hodnota metriky měřená daným nástrojem liší od implementace. Očekává se, že se některé hodnoty budou mírně lišit. CCCC, SourceMonitor a PMCCabe totiž nezapočítávají příkazy uvnitř maker. Navíc i k samotnému měření metrik mohou přistupovat jinak.

Počet příkazů

Na obrázku 5.1 je vidět odchylka u měření počtu příkazů. Tato odchylka je poměrně přijatelná. PMCCabe počítá příkaz `for` jako tři příkazy, implementace tento příkaz počítá jako jeden. To je hlavní důvod pro větší počet příkazů, než naměřila implementace. Zároveň nepočítá příkaz `else` jako samostatný příkaz, proto může být hodnota i menší.



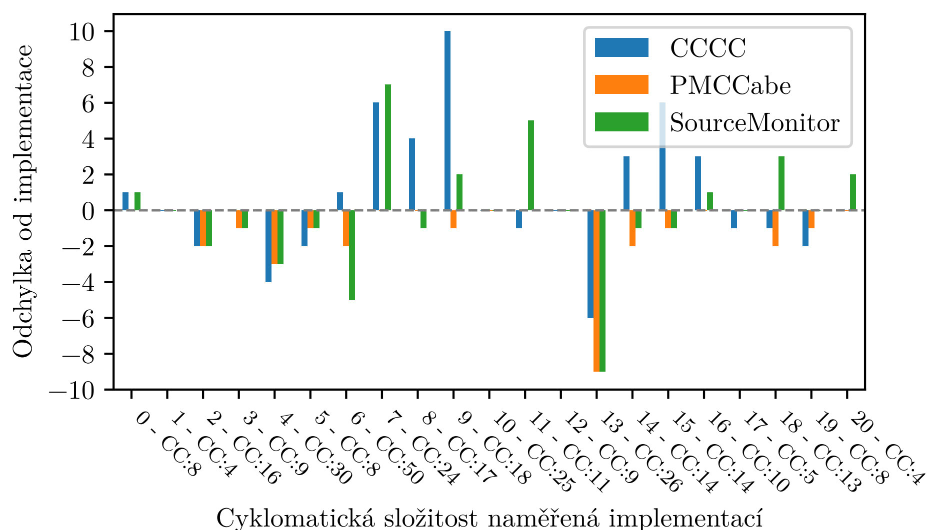
Obrázek 5.1: Odchylka u počtu příkazů.

Cyklomatická složitost

Na obrázku 5.2 je zobrazena odchylka měření cyklomatické složitosti. Je vidět, že hodnoty se u PMCCabe liší jen mírně. U funkce 13 je rozdíl veliký. Důvod je ten, že obsahuje 9 `assert` maker (PMCCabe, CCCC a SourceMonitor je tedy nepočítá), které ve svém těle obsahují příkaz `if`. U CCCC jsou vyšší hodnoty převážně kvůli tomu, že počítá i příkazy jako je `break` a `return`. Zároveň nepracuje s makry a výjimkami, proto může být hodnota i menší.

Maximální hloubka

Tuto metriku měří pouze SourceMonitor. Naměřené hodnoty byly téměř stejné. Chyba nastala pouze u tří funkcí a byla minimální. Rozdíl může zapříčinit způsob počítání příkazu `else`, který je popsaný v sekci 4.3.5.



Obrázek 5.2: Odchylka u cyklomatické složitosti.

Fan-in

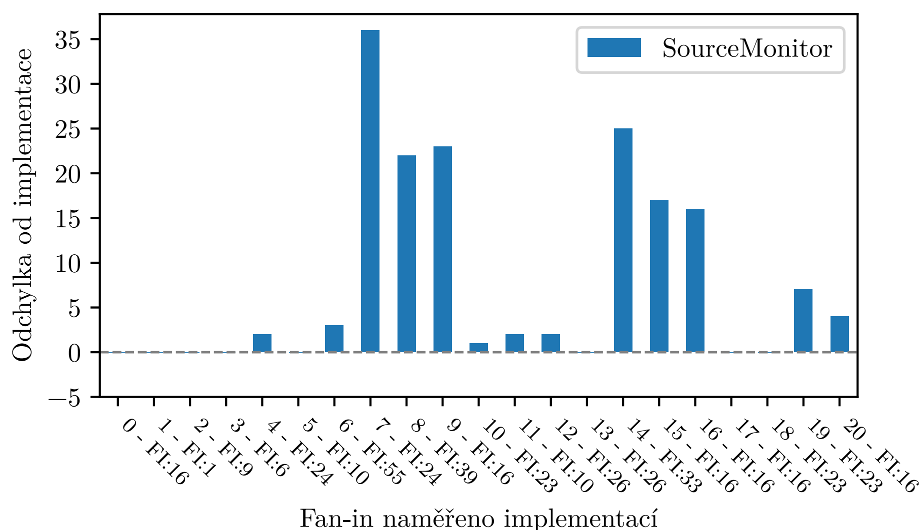
SourceMonitor tuto metriku přímo nazývá *calls*. Měří počet volaných funkcí stejně jako metrika fan-in u implementace. Na obrázku 5.3 se nachází odchylka oproti SourceMonitor. Hodnoty se v některých případech extrémně liší. SourceMonitor do volání funkce počítá i volání konstruktoru, destruktoru, makra, C++ přetypování a podobně. V některých případech naměřil vyšší hodnoty, než kolik volání se ve funkci nachází (včetně zmíněných v minulé větě).

5.2 Testování na malém projektu

Testování proběhlo na dvou menších projektech. Naměřené hodnoty byly pouze porovnány s ostatními měřiči. Ručně byly přepočítány v případech, že se naměřené hodnoty velmi lišily. Jedná se o úkoly do programovacího předmětu na škole FIT ČVUT. Pro tyto úkoly existuje od studentů spousta odevzdaných řešení, nad kterými budou měřiče porovnány. Tyto soubory byly otestovány jen proti měřičům CCCC a PMCCabe.

Pro každou funkci ve zdrojovém souboru byly naměřeny jednotlivé metriky. Z naměřených hodnot byl spočten rozdíl, který vracela implementace a open-source měřiče. Tyto rozdíly byly poté sečteny a vyděleny celkovým počtem řádků funkcí. Hodnoty byly vynásobeny číslem 100. Říkají, jaká je průměrná odchylka na 100 řádků v daném souboru. Pro každý soubor se poté tyto hodnoty zanesly do histogramu.

První úloha byla zrcadlení obrázku. V grafech je pod názvem *Image*. Byla



Obrázek 5.3: Odchylka u fan-in.

to první úloha v semestru, odevzdaných souborů bylo 4328. Druhá úloha byla implementačně náročnější než první. Měla nejmenší úspěšnost v semestru. Odevzdání bylo pouze 1240. Jednalo se o implementaci řetězce, který je slepen z jednotlivých menších řetězců. V grafech je označená jako *String*.

5.2.1 Filtrace

Odevzdaných souborů je velké množství. Studenti mohou odevzdat svůj kód vícekrát[9]. Mezi odevzdanými soubory se tedy nachází několik „skoro“ duplikátů. Často se totiž odevzdání od stejného studenta liší jen nepatrně. Tyto podobné soubory byly kromě nejstaršího odstraněny. Taková filtrace může být výpočetně náročná. Neporovnával se každý soubor s každým. Každý soubor byl porovnán jen s určitou podmnožinou, která byla odevzdána v podobné době.

Byly také odstraněny soubory, které svým řešením vůbec neodpovídaly zadání dané úlohy. Takový šum pravděpodobně vznikl kvůli tomu, že studenti omylem nahráli své řešení pod nesprávné zadání.

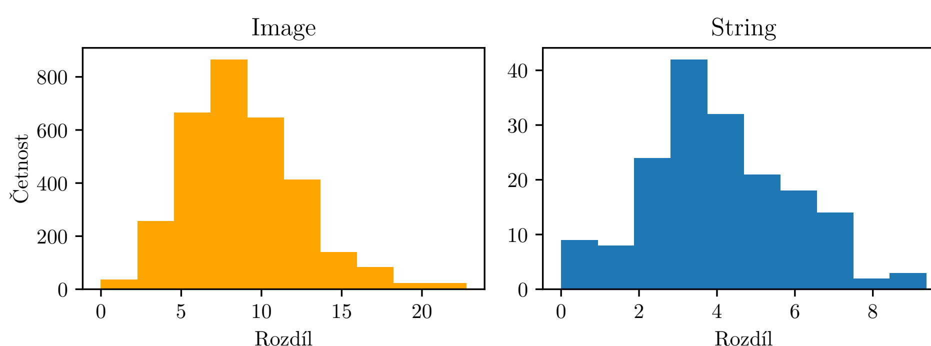
Může se stát, že některý měřič chybně zpracoval zdrojový kód. U implementace toto nastane, pokud zdrojový kód nelze zkompilovat. Takové zdrojové kódy byly vynechány. Dále byly přeskočeny soubory, které open-source měřiče špatně zpracovaly. Jako indikátor tohoto jevu používáme fakt, že měřiče našly méně než polovinu funkcí, které ve zdrojovém kódu našla implementace.

[9]Obvykle mají k dispozici 20 nepenalizovaných pokusů.

Samotné odevzdávací prostředí definuje speciální makro. Pokud bude kus kódu umístěn v podmíněném příkazu, který kontroluje, jestli je toto makro zadefinované, odevzdávací prostředí ho ignoruje. Typicky se toto využívá pro vlastní testovací funkce, které nemusí být součástí odevzdání. Při měření bylo toto makro definováno. Testovací funkce tedy měřeny nebyly. Ostatní měřiče expandovat makra neumí. Porovnávány byly ale jen ty funkce, které naměřila implementace.

Filtrace souborů nebyla perfektní, to ale nebyl účel. Cíl byl především eliminovat většinu šumu, který by se v měřeních mohl objevit.

5.2.2 Cyklomatická složitost



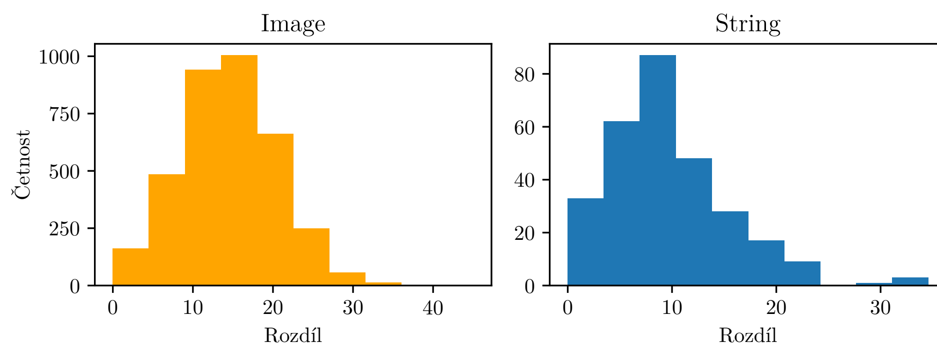
Obrázek 5.4: CCCC - Odchylka u cyklomatické složitosti.

Na obrázku 5.4 je zobrazen histogram. Je zde ukázáno, jak moc se hodnoty naměřené měřičem CCCC lišily. Tato odchylka je pravděpodobně zapříčiněna metodou počítání, kterou CCCC používá. Toto bylo zmíněno v podkapitole 5.1.1.

Na obrázku v příloze B.1 se nachází typově stejný histogram pro měřič PMCCabe. Naměřené hodnoty byly naprosto stejné. V ojedinělých případech byla odchylka minimální.

5.2.3 Počet příkazů

Na obrázku 5.5 je histogram, který ukazuje průměrnou odchylku od počtu příkazů na 100 řádků. Naměřené odchylky nejsou zanedbatelné. V obou projektech bylo namátkově vybráno několik souborů, kde byl rozdíl veliký. V těchto souborech byl rozdíl zapříčiněn existencí `for` příkazů. PMCCabe totiž za každý tento příkaz zvýší hodnotu o tři, implementace jen o jednu.



Obrázek 5.5: PMCCabe - Odchylka u počtu příkazů.

Kapitola 6

Závěr

V této kapitole jsou shrnuty výsledky práce a její přínos. Zároveň jsou uvedeny možnosti, jak na práci navázat.

6.1 Shrnutí práce

V úvodní kapitole 1 byla představena motivace a cíle práce. V kapitole 2 byla provedena rešerše na různé softwarové metriky a na existující měřiče těchto metrik. Bylo uvedeno, jak se metriky měří a co o zdrojovém kódu vypovídají.

V kapitole 3 byly stručně představeny překladače. Dále byl rozebrán překladač Clang a jeho rozhraní, které je možné využít pro statickou analýzu kódu. Na základě těchto rešerší byl implementován měřič metrik pro jazyky C a C++. Protože je tento měřič nadstavbou nad překladačem Clang, zvládá naprostou většinu C++ gramatiky. Toto je popsané v kapitole 4.

V kapitole 5 je popsáno, jak byl projekt otestovaný a porovnaný s ostatními open-source měřiči. Ve srovnání s ostatními měřiči obstál. Některé naměřené hodnoty se neshodovaly. Bylo odůvodněno, proč se hodnoty liší (např. různé měřiče používají jiné definice některých metrik).

6.2 Přínosy práce

Byl vytvořen měřič metrik, který se dokáže vypořádat s naprostou většinou gramatiky jazyka C++. Tento měřič zároveň měří více metrik než dostupné open-source alternativy. To je zachyceno v tabulce 4.1.

Z výsledku práce plyne, že pokud je nutné vytvořit složitější analyzátor textu pro jazyk C++, může být výhodnější naučit se pracovat s překladačem Clang než vytvářet vlastní analyzátor.

6.3 Navazující práce

Zde jsou uvedeny možnosti, jak na práci navázat nebo ji rozšířit.

- Existuje samozřejmě více metrik, než tato práce uvádí. Je možné implementaci obohatit o další metriky.
- Export výsledku naměřených hodnot je poměrně stručný. Bylo by vhodné implementaci rozšířit o uživatelsky přívětivější přehled napočítaných hodnot, například ve formátu HTML.
- Provést porovnání naměřených metrik s kvalitou posouzenou díky code review. Tento výzkum by mohl ukázat, jak moc jednotlivé metriky opravdu ukazují kvalitu návrhu.
- Metriky se mohou hodit na detekci plagiátů. Většina metrik není ovlivněná tím, že se změní názvy identifikátorů nebo formátování kódu. Toto téma částečně zkoumal *E. L. Jones* [38], je možné na tento výzkum navázat.

Literatura

[1] Clang. `https://clang.llvm.org/`, [cit. 2021-03-19].

[2] Jones, C.: Software metrics: good, bad and missing. *Computer*, ročník 27, č. 9, 1994: s. 98–100, doi:10.1109/2.312055.

[3] LeBlanc, R.: *Software Metrics, A Rigorous and Practical approach*. Třetí vydání.

[4] Haoyu, W.; Haili, Z.: Basic Design Principles in Software Engineering. In *2012 Fourth International Conference on Computational and Information Sciences*, 2012, s. 1251–1254, doi:10.1109/ICCIS.2012.91.

[5] Al-Qutaish, R.; Abran, A.: An Analysis of the Design and Definitions of Halstead's Metrics. 09 2005.

[6] Fenton, N.: Software Measurement: A Necessary Scientific Basis. *Software Engineering, IEEE Transactions on*, ročník 20, 04 1994: s. 199 – 206, doi:10.1109/32.268921.

[7] Virtual Machinery The Halstead metrics. `http://www.virtualmachinery.com/sidebar2.htm`, [cit. 2020-02-26].

[8] Halstead, M. H.: *Elements of Software Science (Operating and Programming Systems Series)*. USA: Elsevier Science Inc., 1977, ISBN 0444002057.

[9] Nejmeh, B. A.: NPATH: A Measure of Execution Path Complexity and Its Applications. *Commun. ACM*, ročník 31, č. 2, Únor 1988: str. 188–200, ISSN 0001-0782, doi:10.1145/42372.42379. Dostupné z: `https://doi.org/10.1145/42372.42379`

[10] McCabe, T. J.: A Complexity Measure. *IEEE Transactions on Software Engineering*, ročník SE-2, č. 4, 1976: s. 308–320, doi:10.1109/TSE.1976.233837.

[11] Matoušek, J.; Nešetřil, J.: *Kapitoly z diskrétní matematiky.* ISBN 978-80-246-1740-4.

[12] Curtis, B.; Sheppard, S.; Milliman, P.; aj.: Measuring The Psychological Complexity of Software Maintenance Tasks with the Halstead and McCabe Metrics. *Software Engineering, IEEE Transactions on*, ročník SE-5, 04 1979: s. 96– 104, doi:10.1109/TSE.1979.234165.

[13] Dijkstra, E.: *Go to Statement Considered Harmful.* USA: Yourdon Press, 1979, ISBN 0917072146, str. 27–33.

[14] Henry, D., S.; Kafura: Software Structure Metrics Based on Information Flow. *IEEE Transactions on Software Engineering*, ročník SE-7, 1981: s. 0–518, ISSN 0098-5589,1939-3520, doi:10.1109/tse.1981.231113. Dostupné z: `http://doi.org/10.1109/tse.1981.231113`

[15] Henry, S.; Kafura, D.; Harris, K.: On the Relationships among Three Software Metrics. ročník 10, č. 1, Leden 1981: str. 81–88, ISSN 0163-5999, doi:10.1145/1010627.807911. Dostupné z: `https://doi.org/10.1145/1010627.807911`

[16] Chidamber, S. R.; Kemerer, C. F.: A metrics suite for object oriented design. *IEEE Transactions on Software Engineering*, ročník 20, č. 6, 1994: s. 476–493, doi:10.1109/32.295895.

[17] Hitz, M.; Montazeri, B.: Measuring coupling and cohesion in object-oriented systems. 10 1995.

[18] Fernández, L.; Peña, R.: A sensitive metric of class cohesion. 2006.

[19] Bansiya, J.; Etzkorn, L. H.; Davis, C.; aj.: A Class Cohesion Metric For Object-Oriented Designs. *J. Object Oriented Program.*, ročník 11, 1999: s. 47–52.

[20] Harrison, R.; Counsell, S.; Nithi, R.: Overview of object-oriented design metrics. 08 1997, ISBN 0-8186-7840-2, s. 230–235, doi:10.1109/STEP.1997.615494.

[21] El-Wakil, M.; El-Bastawissy, A.; Boshra, M.; aj.: Object-oriented design quality models a survey and comparison. *2nd International Conference on Informatics and Systems (INFOS04)*, 01 2004.

[22] Brij; Goel, D. B. M.; Pradeep, P.; aj.: An Overview of Various Object Oriented Metrics. *International Journal of Information Technology and Systems (IJITS) - ISSN 2277–9825*, ročník 2, 01 2013: s. 18–27.

[23] Brito e Abreu, F.: Object-Oriented Software Engineering: Measuring and Controlling the Development Process. 01 1997.

[24] C and C++ Code Counter (CCCC). http://cccc.sourceforge.net/, [cit. 2020-03-23].

[25] PMCCABE. https://people.debian.org/~bame/pmccabe/contribute.html, [cit. 2021-04-24].

[26] SourceMonitor. https://www.campwoodsw.com/sourcemonitor.html, [cit. 2021-04-30].

[27] Klocwork. https://www.perforce.com/products/klocwork, [cit. 2021-04-30].

[28] Resource Standard Metric. https://msquaredtechnologies.com/index.html, [cit. 2021-03-23].

[29] Testwell C++. https://www.verifysoft.com/en_cmtx.html, [cit. 2021-03-23].

[30] Aho, A. V.; Lam, M. S.; Sethi, R.; aj.: *Compilers: Principles, Techniques, and Tools (2nd Edition)*. USA: Addison-Wesley Longman Publishing Co., Inc., 2006, ISBN 0321486811.

[31] LLVM Project. https://www.llvm.org/, [cit. 2021-03-19].

[32] LibClang. https://clang.llvm.org/doxygen/group__CINDEX.html, [cit. 2021-03-19].

[33] Gamma, E.; Helm, R.; Johnson, R.; aj.: *Design Patterns: Elements of Reusable Object-Oriented Software*. USA: Addison-Wesley Longman Publishing Co., Inc., 1995, ISBN 0201633612.

[34] Clang plugins. https://clang.llvm.org/docs/ClangPlugins.html, [cit. 2021-03-19].

[35] LibTooling. https://clang.llvm.org/docs/LibTooling.html, [cit. 2021-03-19].

[36] Abrahams, D.; Gurtovoy, A.: *C++ Template Metaprogramming: Concepts, Tools, and Techniques from Boost and Beyond (C++ in Depth Series)*. Addison-Wesley Professional, 2004, ISBN 0321227255.

[37] Box2D. https://github.com/erincatto/box2d/, [cit. 2021-04-30].

[38] Jones, E.: Metrics based plagarism monitoring. *Journal of Computing Sciences in Colleges*, ročník 16, 01 2001: s. 253–261.

Příloha **A**

Seznam použitých zkratek

OOP Objektově orientované programování

OO Objektově orientované

XML Extensible Markup Language

LOC Lines of Code

CC Cyclomatic Complexity

AST Abstract Syntax Tree / Abstraktní syntaktický strom

API Application Programming Interface

WMC Weighted Methods Per Class

LCOM Lack Of Cohesion in Methods

SCOM Sensitive Class Cohesion Metric

MOOD Metrics for Objected Oriented Design

CCCC C and C++ Code Counter

YACC Yet Another Compiler-Compiler

IDE Integrated Development Environment

GUI Graphical User Interface

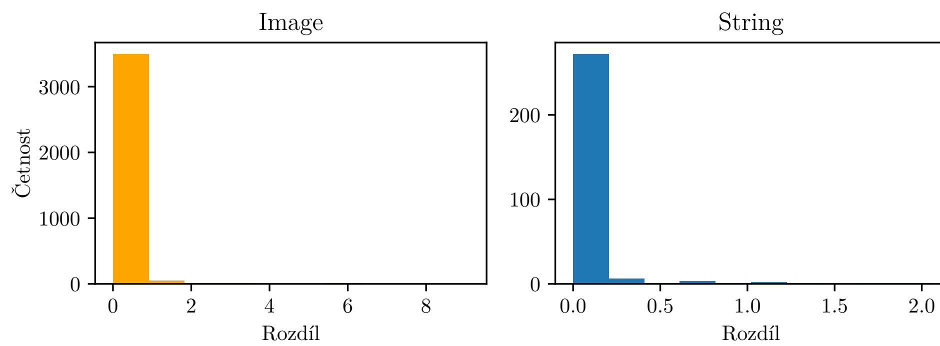
GCC GNU Compiler Collection

HTML HyperText Markup Language

CSV Comma-Seperated Values

Příloha B

Graf porovnávající měřiče



Obrázek B.1: PMCCabe - Odchylka u cyklomatické složitosti.

Příloha C

Uživatelská příručka

Následující kapitola popisuje, jak projekt sestavit a použít. Tento návod byl otestovaný na operačním systému Ubuntu 20.04.2 LTS. Pro sestavení projektu je nutné mít následující balíčky:

- cmake
- clang
- libclang-dev

Pro sestavení projektu se řiďte následujícími kroky:

- Vytvořte složku, kde budete mít sestavený projekt. Spusťte příkaz `cmake` z této složky. Jako argument uveďte složku, kde se nachází implementace projektu.
- Spusťte příkaz `make`.
- Volitelně můžete spustit jednotkové testy příkazem `make test`.

Program je poté možné spustit z příkazové řádky. Jako argument uveďte zdrojové kódy, které chcete analyzovat. Program je nadstavbou kompilátoru, ten dokáže taky přijímat argumenty. Můžete například specifikovat standard, pod kterým chcete kompilovat, nebo definovat makra. Argumenty pro kompilátor uvádějte za znak `--`, příklad:

```
metrics example1.cpp example2.cpp -- -std=c++17 -DDebug
```

Pozor, je možné, že program nenajde Vaše standardní knihovny. To se projeví jako chybová hláška při spuštění programu. Ačkoli měření metrik to nutně nemusí ovlivnit, je doporučeno použít následující řešení. Je nutné explicitně uvést cestu ke knihovnám. Tu je možné zjistit příkazem `clang --print-file-name=include`. Spuštění měřiče by tedy vypadalo následovně:

```
metrics src.cpp -- -I`clang --print-file-name=include`
```

Program dokáže exportovat metriky ve formátu XML. Pro export využijte přepínač `-xml` a případně za něj uveďte název souboru, do kterého chcete metriky exportovat.

Příloha D

Obsah přiloženého CD

```
readme.txt ........................ stručný popis obsahu CD
exe ............................... adresář se spustitelnou formou implementace
src
  impl ............................ zdrojové kódy implementace
  thesis .......................... zdrojová forma práce ve formátu LaTeX
text .............................. text práce
  thesis.pdf ...................... text práce ve formátu PDF
```