VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV INFORMAČNÍCH SYSTÉMŮ
DEPARTMENT OF INFORMATION SYSTEMS

VLIV KOPÍROVÁNÍ KÓDU NA KVALITU SOFTWAROVÉHO PROJEKTU

INFLUENCE OF CODE COPYING ON THE QUALITY OF A SOFTWARE PROJECT

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE **ZDENĚK CHOVANEC**
AUTHOR

VEDOUCÍ PRÁCE **Ing. ZBYNĚK KŘIVKA, Ph.D.**
SUPERVISOR

BRNO 2018

Vysoké učení technické v Brně
Fakulta informačních technologií

Ústav informačních systémů (UIFS) Akademický rok 2018/2019

Zadání bakalářské práce

Student: **Chovanec Zdeněk**
Program: Informační technologie
Název: **Vliv kopírování kódu na kvalitu softwarového projektu**
Influence of Code Copying on the Quality of a Software Project
Kategorie: Překladače

Zadání:

1. Seznamte se s API StackOverflow a podobných služeb.
2. Prostudujte metody hledání stejného nebo podobného kódu (např. podobnost AST, Levenshteinova vzdálenost, n-gramy).
3. Dle pokynů konzultanta/vedoucího získejte sadu ukázek kódu, které se potenciálně často kopírují (např. ze služby StackOverflow).
4. Dle konzultací navrhněte nástroj pro zjišťování míry kopírování kódu daného projektu odjinud s vyhodnocením ovlivňování kvality celého softwarového projektu.
5. Nástroj implementujte a ověřte na vytvořené testovací sadě projektů. Výsledky testů zhodnoťte a navrhněte vylepšení do budoucna.

Literatura:

- Stack Exchange API, dostupné na https://api.stackexchange.com/ [cit. 2018-09-24]
- Ondřej Krpec: Rozpoznání plagiátů zdrojového kódu v jazyce PHP, bakalářská práce, Brno, Fakulta informačních technologií VUT v Brně, 2015
- dle pokynů konzultanta a vedoucího

Pro udělení zápočtu za první semestr je požadováno:

- Body 1 až 4.

Podrobné závazné pokyny pro vypracování práce viz http://www.fit.vutbr.cz/info/szz/

Vedoucí práce: **Křivka Zbyněk, Ing., Ph.D.**
Konzultant: Tišnovský Pavel, Ing., Ph.D., RedHatCZ
Vedoucí ústavu: Kolář Dušan, doc. Dr. Ing.
Datum zadání: 1. listopadu 2018
Datum odevzdání: 15. května 2019
Datum schválení: 28. října 2018

Abstrakt

Tato práce se věnuje problematice nalezení zkopírovaných úseků kódu. Pozornost je přitom zaměřena na programovací jazyk Python verze 3 a na úseky kódu pocházející ze služby Stack Overflow. Cílem práce je vyhodnocení vlivu kopírování kódu na kvalitu softwarového projektu. Detekce úryvků kódu uvnitř softwarového projektu je provedena pomocí nástroje NiCad. Výchozí proces detekce byl upraven tak, aby se podařilo nalézt co největší počet shodných úryvků. Kvalita kódu je měřena podle míry obsahu duplicitního kódu a dále podle počtu nahlášených chyb daného projektu. Vliv kopírování na kvalitu je vyhodnocen na vzorku open-source projektů pocházejících ze služby GitHub. Vztah mezi přítomností úryvku ze služby Stack Overflow a kvalitou projektu byl prokázán formou statistického testu.

Abstract

This work delves into the field of code clone detection, focusing on Python programming language. We are interested in finding Stack Overflow snippets in real projects. The main goal of this work is to evaluate the influence of copy-and-paste programming on the overall quality of a software project. The NiCad clone detector is used to detect similar code fragments. Additionally, modification of this tool is presented in order to improve the detection process. Code quality is assessed by the volume of duplicate code and by the number of reported issues. The impact is evaluated on a sample of open-source projects hosted on GitHub. We perform a series of statistical hypothesis tests and conclude that there is a correlation between source code quality and the presence of Stack Overflow snippet in the code base.

Klíčová slova

kvalita kódu, duplicitní kód, detekce duplicitního kódu, úryvek kódu, Stack Overflow, Python 3, NiCad, TXL, transformace zdrojového kódu

Keywords

code quality, duplicate code, code clone detection, code snippet, Stack Overflow, Python 3, NiCad, TXL, source code transformation

Citace

CHOVANEC, Zdeněk. *Vliv kopírování kódu na kvalitu softwarového projektu.* Brno, 2018. Bakalářská práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Ing. Zbyněk Křivka, Ph.D.

Vliv kopírování kódu na kvalitu softwarového projektu

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením pana Ing. Zbyňka Křivky, Ph.D. Další informace mi poskytl pan Ing. Pavel Tišnovský, Ph.D. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

.....
Zdeněk Chovanec
16. května 2019

Poděkování

Rád bych poděkoval vedoucímu práce panu Ing. Zbyňku Křivkovi, Ph.D. za připomínky a poskytnuté konzultace. Poděkování rovněž patří konzultantovi panu Ing. Pavlu Tišňovskému, Ph.D. za jeho připomínky a za udělenou volnost při výběru způsobu řešení této práce.

Obsah

Kapitola 1

Úvod

Při vývoji softwarových produktů není ničím neobvyklým kopírování fragmentů zdrojového kódu. Zkopírování obvykle bývá mnohem rychlejší, než-li vytvoření potřebného úseku kódu od nuly. Přihlédneme-li také k množství stránek, které se přímo zaměřují na sdílení zdrojového kódu, pak se není čemu divit, že k této činnosti dochází.

Kopírování však sebou přináší rizika. Nesprávná interpretace toho, co daný úsek kódu vykonává, může vést k zanesení chyb do softwaru. Tyto chyby mohou být těžko detekovatelné a nemusí se zpočátku vůbec projevit. Chybně pracující úryvek může být zkopírován na další místa. Zpětné dohledání všech instancí daného úryvku nemusí představovat jednoduchou záležitost.

Tato práce má za úkol vyhodnotit, jak se kopírování kódu podepisuje na kvalitě projektu. Nejprve musíme získat kód, jenž může být zkopírován. Z toho důvodu se zaměříme na volně dostupné internetové zdroje. Poté, co získáme potenciálně kopírované úryvky, můžeme zahájit proces detekce shody kódu úryvku a kódu uvnitř projektu.

Kapitola 2 vysvětluje základní pojmy z oblasti detekce shodného či podobného kódu. Mimo obecných pojmů také uvádí některé konkrétní přístupy a nástroje, které lze přímo aplikovat.

Kapitola 3 vytyčuje základy pro vyřešení zadaného problému a podává informace o rozhodnutích, jež bylo nutno učinit. To zahrnuje výběr programovacího jazyka, na který se v této práci zaměříme či nalezení vhodného zdroje úryvků kódu.

Kapitola 4 vychází ze základů vytyčených v předchozí kapitole a popisuje konkrétní detaily implementace. Největší část je věnována úpravám nástroje pro detekci shodných úryvků kódu.

Kapitola 5 se věnuje vyhodnocení vlivu kopírování úryvků kódu na kvalitu projektu. V této kapitole jsou položeny tři otázky týkající se ovlivňování kvality. Tyto otázky jsou postupně zodpovězeny.

Závěrečná kapitola 6 vytváří stručný souhrn celé práce.

Kapitola 2

Techniky a přístupy užívané k detekci stejného/podobného kódu

Hledání stejného či podobného zdrojového kódu je úzce spjato se dvěma oblastmi – odhalováním plagiátů (*plagiarism detection*) a detekcí duplicitního kódu (*code clone detection*).

Pojem plagiátorství je velmi často spojen se vzdělávacími institucemi. Student kvůli nedostatku času či znalostí zkopíruje kód jiného studenta (ať už oprávněně, nebo neoprávněně) a ten vydává za vlastní. Plagiátorům při odhalení obvykle hrozí tvrdé postihy. To je jedna z příčin, proč jsou plagiátoři nuceni zkopírovaný kód dodatečně poupravit. Tyto úpravy zahrnují změnu názvu identifikátorů, přidání/odstranění komentářů, transformaci řídicích struktur a podobně.

Detekce duplicitních úseků kódu se týká primárně velkých softwarových projektů. Programátor zkopíruje část existujícího řešení, dle potřeby provede malé úpravy, a tím vyřeší aktuální problém. Duplikace kódu je v rozporu se zásadami strukturovaného programování, komplikuje proces údržby a navyšuje náklady s tímto procesem spojené. Míra duplicitního kódu může sloužit jako indikátor kvality softwarového produktu.

Za účelem odhalování plagiátorství a detekce duplicitního kódu vznikla celá řada nástrojů, z nichž lze jmenovat následující: MOSS (plagiátorství), JPlag (plagiátorství), NiCad (duplicitní kód), Simian (duplicitní kód) či Deckard (duplicitní kód). Detektory duplicitního kódu lze využít k odhalení plagiátorství a naopak. Rozdíl mezi nimi je v tom, že detektory duplicitního kódu odhalují duplikáty i v rámci jednoho souboru/projektu, zatímco nástroje pro odhalení plagiátorství hledají shodu mezi různými soubory/projekty.

Součástí dokumentace nástrojů je vždy sekce deklarující množinu podporovaných jazyků. Nástroje se dále odlišují v typu duplicitního kódu, který jsou schopny detekovat. Jednotná definice vymezující jednotlivé typy však neexistuje. Následující rozdělení typů duplicitního kódu (převzato z [14]) se snaží sjednotit existující popisy:

- **Typ I:** Identické úseky kódu s výjimkou rozdílů v bílých znacích (prázdné řádky, tabulátory, atd.), komentářích a rozložení kódu. V literatuře se pro tento typ klonů používá označení *exact clones*.
- **Typ II:** Strukturálně/syntakticky identické úseky kódu mimo rozdíly v názvech identifikátorů, literálech, komentářích a rozložení kódu.

- **Typ III:** Zkopírované úseky kódu s dalšími modifikacemi. Mimo změny v názvech identifikátorů, literálech, komentářích a rozložení kódu mohou být příkazy dále pozměněny, přidány či odstraněny. V literatuře se tento typ označuje jako *near-miss clones*.
- **Typ IV:** Týká se dvou či více úseků kódu, které provádějí stejný výpočet, ale jsou implementovány pomocí jiných syntaktických konstrukcí.

Rozložením kódu se rozumí umístění elementů ohraničujících bloky kódu. Například v jazyce C lze dosáhnout rozdílu v rozložení kódu použitím různých konvencí umisťování složených závorek. S rostoucím typem roste obtížnost detekce duplikátů. Typy I – III vycházejí z textové podobnosti kódu. Typ IV naopak vychází ze sémantické podobnosti dvou úseků programů.

Nutnou součástí každého nástroje pro odhalení plagiátorství či duplikátů musí být mechanismus označení podezřelých úseků. Jednou z možností je hlášení všech nalezených párů (tyto páry se označují jako *clone pairs*). Alternativou k předchozího přístupu je agregování nalezených párů do tříd (označení *clone class*).

Klonová relace (*clone relation*) je relací ekvivalence, a proto je reflexivní, symetrická a tranzitivní. Prvky, jež jsou v klonové relaci tvoří duplicitní pár/klon (*clone pair*), který se detektorem snažíme nalézt. Dva úseky kódu jsou v klonové relaci, právě když obsahují identickou posloupnost. Třída klonů (*clone class*), jež slouží k agregaci duplicitních párů, je množina obsahující všechny úryvky kódu, kde libovolné dva úryvky jsou v klonové relaci. [10]

Definice identické posloupnosti dvou úseků kódu závisí na míře abstrakce, kterou na zdrojový kód aplikujeme. Tato posloupnost může být tvořena původními řádky, normalizovanými řádky či tokeny. Mějme kupříkladu dva soubory obsahující fragmenty vyznačené v tabulce 2.1. Pokud jako členy posloupnosti budeme uvažovat pouze typy tokenů, pak nalezneme 3 duplicitní páry – $(A1,\ B1)$, $(A1, B2)$, $(B1, B2)$ a 1 třídu klonů – $\langle A1, B1, B2\rangle$.

	fragmenty souboru *A*	**fragmenty souboru *B***	

1	`for (int i = 0; i < size; i++) {`
	`  array[i] =  '\0';`
	`}`
		`for (int j = 64; j < len; j++) {`
		`  table[j] = 'a';`
		`}`
		`for (int k = 0; k < size; k++) {`
		`  cells[k] = 'x';`
		`}`
			1
			
			2

Tabulka 2.1: Zajímavé fragmenty z pohledu detekce duplicitního kódu.

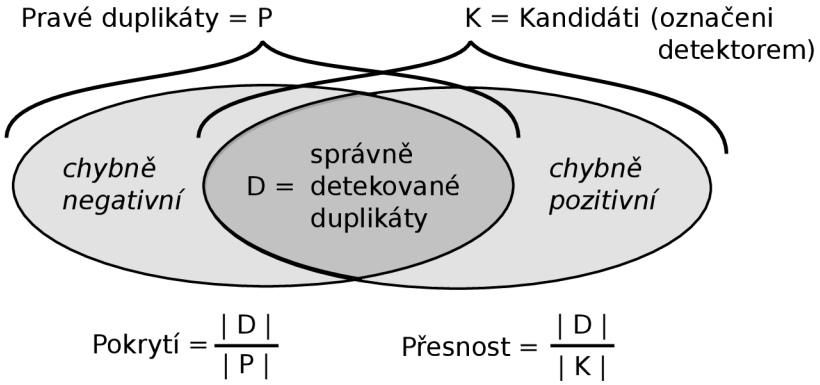
Ke zjištění kvality detektoru duplicitních úseků kódu se běžně používají dvě metriky – pokrytí a přesnost. Tyto metriky lze použít pro hodnocení jakéhokoli nástroje, který řeší problém binární klasifikace (zda instance má danou vlastnost či nikoliv).

- **Pokrytí** (*recall*) je definováno jako podíl počtu správně detekovaných instancí vůči všem relevantním instancím.
- **Přesnost** (*precision*) je podíl počtu správně detekovaných instancí vůči všem detekovaným instancím.

Relevantní instance jsou ty, jež mají námi hledanou vlastnost (například obsahují určitou posloupnost řádků). S těmito dvěma metrikami souvisí pojmy:

- **Chybně pozitivní** (*false positives*) je označení pro instance, které byly detektorem označeny jako relevantní, ale ve skutečnosti jimi nejsou.
- **Chybně negativní** (*false negatives*) je označení pro instance, které detektorem nebyly označeny jako relevantní, ale správně označeny být měly.

Ilustrace pokrytí a přesnosti je na obrázku 2.1. Výsledek klasifikace daného detektoru je označen jako „Kandidáti“. „Pravé duplikáty“ je označení pro množinu obsahující všechny instance, které vznikly kopírováním kódu.



Obrázek 2.1: Pokrytí (*recall*) a přesnost (*precision*) (převzato z [7]).

Obecně lze proces detekce rozdělit do dvou fází. První fáze zahrnuje transformaci zdrojového kódu do interní reprezentace. Druhou fázi představuje aplikace více či méně efektivního algoritmu, jenž je schopen v interní reprezentaci detekovat shodné vzory.

Používané techniky se liší mírou abstrakce, kterou uplatňují nad zdrojovým kódem. Základní principy jednotlivých technik společně s příklady konkrétních přístupů/nástrojů jsou popsány v následujících podkapitolách.

2.1 Porovnávání řádků/řetězců

Nástroje, jež pracují přímo s textovou reprezentací zdrojového kódu, nahlíží na daný program jako na sekvenci řetězců. Řetězec reprezentuje řádek zdrojového textu. Fáze transformace obvykle zahrnuje odstranění komentářů a bílých znaků. Proces nalezení stejného či podobného kódu spočívá v porovnání (transformovaných) řádků, přičemž se hledá úplná shoda mezi řádky. Podobným způsobem pracuje například nástroj `diff`.

Tato technika, jak byla doposud prezentována, si neporadí s přejmenováním identifikátorů, přeuspořádáním operandů v rámci výrazu a obecně se změnami části řádku. Potíže dále mohou způsobovat volitelné syntaktické konstrukce jazyka. Typický případ takové konstrukce v jazyce C demonstruje ukázka na obrázku 2.2.

Mezi nesporné výhody použití tohoto přístupu patří nezávislost na zdrojovém jazyce. Není totiž potřeba lexikální ani syntaktický analyzátor, a proto lze existující nástroj snadno přizpůsobit pro nový jazyk.

```
if (u)
  v;
```

```
if (u) {
  v;
}
```

Obrázek 2.2: Výraz následující podmíněný příkaz lze v jazyce C volitelně uzavřít do složených závorek.

2.1.1 Duploc

Cílem nástroje Duploc je detekce duplicitního kódu, přičemž hlavním požadavkem při jeho vývoji byla jazyková nezávislost. Nástroj již není dostupný (přesto lze nalézt implementace, které vychází z článku původních autorů – [8]).

V první fázi jsou ze zdrojového kódu odstraněny komentáře a bílé znaky. Takto upravený řádek tvoří řetězec. Řetězce (řádky) jsou postupně porovnány mezi sebou, přičemž musí nastat úplná shoda. „Naivní“ algoritmus porovnání n vstupních řádků má časovou složitost $O(n^2)$. Nástroj užívá optimalizaci spočívající v hešování vstupních řetězců do B skupin. Porovnávány jsou pouze všechny dvojice řetězců v rámci jedné skupiny. Použitím této techniky se časová složitost zredukuje Bkrát. Shodné řetězce jsou zaznamenány do matice. Sekvence shodných řádků z ní musí být následně vyextrahovány. [8]

Jestliže na vstupní text aplikujeme pokročilejší transformace a výsledek normalizujeme, lze eliminovat nedostatky plynoucí z textového porovnání řádků. Tím lze zvýšit pokrytí, avšak za cenu snížení přesnosti nástroje. Vybrané transformace dle [7] zahrnují následující:

- převod velkých písmen na malá v případě, že je daný jazyk nerozlišuje
- odstranění prvků jazyka, které ohraničují příkazy či bloky příkazů
- odstranění běžných nebo nezajímavých jazykových konstrukcí (např. příkaz `break`) či typových modifikátorů (např. `const`)

Příklady normalizace elementů jazyka jsou demonstrovány v tabulce 2.2. Při použití normalizace se ztratí část jazykové nezávislosti, neboť je k ní potřeba, byť velmi jednoduchý, lexikální analyzátor.

Element jazyka	**Příklad**	**Náhrada**
Identifikátor	`counter`	`p`
Řetězcový literál	`"Abort"`	`"..."`
Znakový literál	`'y'`	`'.'`
Celočíselný literál	`42`	`1`
Desetinný literál	`0.314159`	`1.0`
Základní číselný datový typ	`int, short, long, double`	`num`
Jméno funkce	`main()`	`foo()`

Tabulka 2.2: Normalizace jednotlivých elementů zdrojového jazyka (převzato z [7]).

2.1.2 NiCad

Nástroj NiCad (*automated detection of near-miss intentional clones*) je na rozdíl od Duploc dostupný[1] a stále udržovaný nástroj (poslední revize 15. dubna 2019). Nástroj v závislosti na konfiguračních parametrech detekuje duplicitní kód typu I, II nebo III. Při detekci duplikátů typu III je použit algoritmus pro nalezení nejdelší společné podposloupnosti (viz kapitola 2.1.3). Samotnému textovému porovnání řádků předchází dvě fáze.

Nejdříve je zdrojový text zpracován syntaktickým analyzátorem a převeden do podoby derivačního stromu. Z něj jsou vyextrahovány uzly/fragmenty představující potenciální klony. To mohou být například definice funkcí, či bloky kódu (v jazycích vycházejících z jazyka C se jedná o bloky ohraničené složenými závorkami). Za účelem zajištění konzistentní struktury fragmentu při zpětném převodu na text je použit formátovaný tisk (*pretty-printing*). Formátovaný tisk eliminuje možné rozdíly v použité konvenci odsazení či řádkování původního zdrojového textu.

Druhou fází je normalizace. Možné normalizace zahrnují přejmenování identifikátorů, odstranění nezajímavých částí (např. deklarace proměnných) či abstrakce vybraných elementů jazyka (např. nahrazení literálů zástupným symbolem). Obě tyto fáze jsou implementovány pomocí programů v jazyce TXL. [5]

Nástroj v současné době podporuje detekci klonů na úrovni bloků kódu a definic funkcí v 8 jazycích – PHP, Java, C#, Python, WSDL, ATL, C a Ruby. Vzhledem k modulární architektuře nástroje lze tuto skupinu rozšířit napsáním příslušných TXL programů pro nový jazyk či detekovaný fragment. Výsledek detekce je dostupný jak ve formátu XML, tak i ve formě HTML stránky.

2.1.3 Nejdelší společná podposloupnost

Nejdelší společná podposloupnost (*longest common subsequence*) dvou (případně více) posloupností je sekvence symbolů, které jsou obsaženy v obou (všech) posloupnostech ve stejném pořadí, přičemž tato podposloupnost je nejdelší možná. Příklad je ilustrován v tabulce 2.3. Při hledání duplicitního kódu je symbolem řádek zdrojového textu, případně heš tohoto řádku.

Posloupnost	Nejdelší společná podposloupnost
X: j k l n o p s t Y: j m n o p q r s u v	$\longrightarrow$ j n o p s

Tabulka 2.3: Dvě posloupnosti a jejich nejdelší společná podposloupnost.

Jedním z možných způsobů, jak ji nalézt, je postupně vytvářet podposloupnosti první posloupnosti, od největších po nejmenší, a pokoušet se je nalézt v posloupnosti druhé. Jestliže posloupnost obsahuje n symbolů, pak lze vytvořit 2^n podposloupností. Časová složitost tohoto naivního algoritmu by byla exponenciální, což je v praxi nepřípustné.

Problém lze efektivně vyřešit za pomoci dynamického programování. Princip spočívá v tom, že nejdříve nalezneme délku nejdelší společné podposloupnosti, nikoliv podposloupnost samotnou. Mějme posloupnosti X a Y, nechť dále platí následující:

$$\begin{aligned} X &= (x_1, x_2, \ldots, x_m) \text{ a } X_i \text{ značí prefix } (x_1, \ldots, x_i) \\ Y &= (y_1, y_2, \ldots, y_n) \text{ a } Y_j \text{ značí prefix } (y_1, \ldots, y_j) \end{aligned}$$

[1] http://www.txl.ca/txl-nicaddownload.html

Délka nejdelší společné podposloupnosti $c[i,j]$ posloupností X_i a Y_j je pak dána vztahem 2.1 [16].

$$c[i,j] = \begin{cases} 0 & \text{pokud } i = 0 \text{ nebo } j = 0 \\ c[i-1,j-1]+1 & \text{pro } i,j > 0 \text{ a } x_i = y_i \\ \max(c[i-1,j], c[i,j-1]) & \text{pro } i,j > 0 \text{ a } x_i \neq y_i \end{cases} \tag{2.1}$$

Pomocí matice o rozměrech $m \times n$ a přepsáním vzorce 2.1 do podoby algoritmu lze nalézt délku nejdelší společné podposloupnosti. Z matice je pak možné podposloupnosti vyextrahovat. Algoritmus má stejnou prostorovou i časovou složitost $O(n.m)$.

2.2 Porovnávání metrik

Při kopírování kódu může docházet k jeho úpravám jako je přidání/odstranění komentářů, bílých znaků nebo přemístění funkcí. Struktura kódu však zůstává zachována. Právě toho se snaží využít metody založené na metrikách. Jako metriky mohou posloužit například Halsteadovy metriky (viz kapitola 2.2.1), cyklomatická složitost (viz kapitola 2.2.2), počet řádků a podobně.

Při hledání podobného kódu se porovnávají získané metriky namísto samotného kódu. Vektor metrik reprezentuje syntaktickou jednotku jazyka jako je definice třídy, funkce či `begin` – `end` blok. Získání některých hodnot metrik vyžaduje převedení zdrojového kódu do podoby abstraktního syntaktického stromu [14].

Na rozdíl od metod hledajících totožné řádky, nelze v případě úplné shody všech metrik s jistotou rozhodnout, zda nedošlo pouze k náhodné shodě hodnot a odpovídající úseky kódu jsou ve skutečnosti odlišné. Jestliže jsou porovnávané struktury krátké, pak získané metriky nejsou dostatečně reprezentativní. Při detekci duplikátů metriky často slouží k redukci počtu kandidátů, kteří jsou následně porovnáni pomocí jiné techniky.

V případě, že je pro získání hodnot metrik nutný převod do podoby abstraktního syntaktického stromu, pak tento převod vytváří velkou závislost na zdrojovém jazyce.

2.2.1 Halsteadovy metriky

Tyto metriky byly navrženy ve snaze kvantitativně popsat vlastnosti programu. Jejich výpočet vychází přímo ze zdrojového kódu, a tudíž nevyžaduje převod do jiné podoby. Program se skládá z tokenů, které lze rozdělit na dvě kategorie – operátory a operandy [9]. Do kategorie operátorů patří například běžné operátory daného jazyka (`+`, `--`, `>=`, atd.), rezervovaná slova (`break`, `while`, atd.) nebo volání funkcí. Skupinu operandů mimo jiné tvoří identifikátory, konstanty či rezervovaná slova specifikující typ (např. `bool`, `int`).

Halsteadovy metriky jsou založeny na následujících čtyřech proměnných:

- n_1 – počet unikátních operátorů
- n_2 – počet unikátních operandů
- N_1 – celkový počet operátorů
- N_2 – celkový počet operandů

Poté, co analyzujeme zdrojový kód a získáme hodnoty těchto proměnných, jsme schopni odvodit hodnoty Halsteadových metrik pomocí vztahů 2.2 – 2.8.

$$\text{Velikost slovníku:} \quad n = n_1 + n_2 \tag{2.2}$$

$$\text{Délka programu:} \quad N = N_1 + N_2 \tag{2.3}$$

$$\text{Odhadnutá délka programu:} \quad \hat{N} = n_1 \log_2 n_1 + n_2 \log_2 n_2 \tag{2.4}$$

$$\text{Objem programu:} \quad V = N \times \log_2 n \tag{2.5}$$

$$\text{Programová náročnost:} \quad D = \frac{n_!}{2} * \frac{N_2}{n_2} \tag{2.6}$$

$$\text{Programátorské úsilí:} \quad E = D * V \tag{2.7}$$

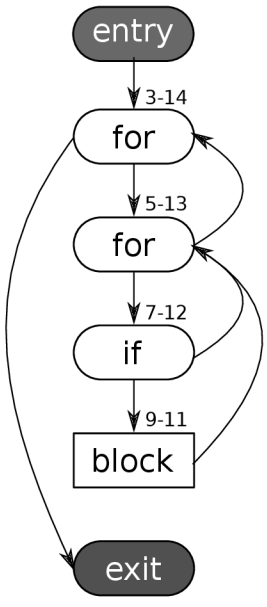
$$\text{Odhad počtu obsažených chyb:} \quad B = \frac{V}{3000} \tag{2.8}$$

2.2.2 Cyklomatická složitost

Jedná se o metriku, pomocí níž je možné odhalit těžko udržovatelné či testovatelné softwarové moduly. Vychází z toho, že složitost programu lze vyjádřit číslem, které udává počet lineárně nezávislých cest v grafu řízení toku programu [13].

Graf řízení toku programu (*control flow graph*) se skládá ze vstupního uzlu, výstupního uzlu a uzlů představujících základní bloky. Základní blok je v tomto kontextu chápán jako posloupnost příkazů, které jsou vždy prováděny jeden za druhým. Hrany mezi uzly odpovídají větvení programu. Lineárně nezávislá cesta je cesta obsahující alespoň jednu hranu, jež není součástí žádně jiné cesty. Funkce implementující algoritmus bubble sort (v jazyce C) včetně odpovídajícího grafu řízení toku programu je znázorněna na obrázku 2.3.

```
1   void bubble_sort(int array[], int size)
2   {
3     for (int i = 0; i < size - 1; i++)
4     {
5       for (int j = 0; j < size - i - 1; j++)
6       {
7         if (array[j] > array[j+1])
8         {
9           int tmp = array[j+1];
10          array[j+1] = array[j];
11          array[j] = tmp;
12        }
13      }
14    }
15  }
```



Obrázek 2.3: Funkce `bubble_sort` a odpovídající graf řízení toku programu.

Cyklomatická složitost C je dána vztahem 2.9, kde N udává počet uzlů grafu a E udává počet hran. Alternativně lze využít vztah 2.10, kde P udává počet vyhodnocení pravdivostní hodnoty. Vyhodnocení probíhá v rámci příkazů `if`, `while`, atd. Například konstrukce

`if (a && b) {...}` přispěje k výsledné cyklomatické složitosti hodnotou 2, neboť dojde ke dvěma vyhodnocením pravdivostní hodnoty (zkrácené vyhodnocení logických podmínek zde neuvažujeme). Výhodou druhého vztahu je, že k němu není nutná konstrukce grafu. Aplikujeme-li tyto vztahy na funkci na obrázku 2.3, dojdeme k závěru, že $C = 8-6+2 = 4$ (dle 2.9), respektive $C = 3+1 = 4$ (dle 2.10).

$$C = E - N + 2 \tag{2.9}$$
$$C = P + 1 \tag{2.10}$$

Výpočet se nejčastěji provádí nad funkcemi nebo třídami. Tato metrika se používá například k hodnocení kvality kódu nebo při vytváření sady testů. Stejně tak ji lze využít během vývoje softwarového projektu stanovením horní hranice, která by neměla být překročena.

2.2.3 Mayrand a kolektiv

Přístup prezentovaný v [12] využívá 21 metrik pro detekci shodných funkcí. Na základě zdrojového kódu je nejdříve vybudován abstraktní syntaktický strom. Ten je převeden do interní reprezentace, ze které jsou získány hodnoty metrik.

Metriky jsou rozděleny do tří kategorií. Každá metrika má pevně stanovenu hodnotu delta, jež slouží jako práh při rozhodování. Příklad metrik a hodnot delta pro kategorii „rozložení zdrojového kódu“ je v tabulce 2.4. Dvě funkce jsou z pohledu dané kategorie

Zkratka	Popis	Delta
ComDecVol	Objem deklaračních komentářů[2]	10
ComStrVol	Objem řídících komentářů[3]	10
ComLogNbr	Počet komentářů	5
LocNbr	Počet neprázdných řádků	5
VarLenAvg	Průměrná délka názvu proměnné	2

Tabulka 2.4: Metriky z kategorie „rozložení zdrojového kódu“ dle [12].

prohlášeny za rozdílné, jestliže existuje metrika, ve níž se liší o hodnotu větší než delta.

Při hledání duplikátů jsou nejdříve porovnány jména funkcí. Poté dochází k vyhodnocení metrik a zjištění podobnosti v jednotlivých kategoriích. Na základě těchto čtyř dílčích výsledků jsou funkce zařazeny do jedné z osmi kategorií podobnosti. Jestliže ani jeden výsledek nenahlásil shodu/podobnost, pak se jedná o dvě různé funkce.

2.3 Porovnávání sekvence tokenů

Lexikální analyzátor je část překladače, která postupně načítá znaky zdrojového textu a identifikuje lexémy. Lexémy jsou následně transformovány do podoby tokenů. Typ tokenu a jeho dodatečný obsah závisí na typu přečteného lexému. Tokeny jsou dále předány na vstup syntaktickému analyzátoru k dalšímu zpracování. Ukázka konkrétních lexémů, jejich typů a odpovídajících typů tokenů je v tabulce 2.5.

Během tokenizace zdrojového kódu bývají ignorovány komentáře a bílé znaky. Ve výsledné posloupnosti tokenů odpovídá všem identifikátorům stejný typ tokenu. Z toho vy-

[2] Počet znaků obsažených v komentářích v deklarační části.
[3] Počet znaků obsažených v komentářích v rámci vlastního těla funkce.

Typ lexému	Lexém	Token
identifikátor	`array_length`	TOKEN_ID
celočíselný literál	`42`	TOKEN_INT_LIT
řetězcový literál	`"Hello world!"`	TOKEN_STR_LIT
operátor porovnáni	`==`	TOKEN_REL_OP

Tabulka 2.5: Lexémy, jejich typy a odpovídající typy tokenů.

plývá přirozená odolnost vůči změnám na úrovni přejmenování identifikátorů nebo přidání/odstranění bílých znaků a komentářů.

Dvě sekvence tokenů jsou porovnány mezi sebou za účelem nalezení shody. K porovnání dvou sekvencí lze užít algoritmus pro nalezení nejdelší společné podposloupnosti (viz kapitola 2.1.3). Alternativně lze míru podobnosti mezi dvěma sekvencemi popsat pomocí Levenshteinovy vzdálenosti (viz kapitola 2.3.1).

Jelikož pracujeme se sekvencemi tokenů, jsou tyto metody citlivé na přidání nebo odebrání úseků kódu. Tato slabá místa lze částečně eliminovat použitím tzv. „otisku dokumentu“ a algoritmu Winnowing (viz kapitola 2.3.2). [14]

2.3.1 Levenshteinova vzdálenost

Hodnotu Levenshteinovy vzdálenosti (též zvaná jako editační) lze interpretovat jako míru podobnosti mezi dvěma řetězci. Vzdálenost je dána minimálním počtem operací přidání, odebrání nebo substituce takových, abychom z jednoho řetězce dostali druhý [11]. Například pro slova *domluva* a *promluva* je Levenshteinova vzdálenost rovna hodnotě 2.

Levenshteinovu vzdálenost mezi prvními i znaky řetězce a a prvními j znaky řetězce b značenou $\mathrm{lev}_{a,b}(i,j)$ lze vyjádřit vztahem 2.11. Levenshteinova vzdálenost řetězců a, b o délkách $|a|$, $|b|$ je pak dána jako $\mathrm{lev}_{a,b}(|a|,|b|)$.

$$\mathrm{lev}_{a,b}(i,j) = \begin{cases} \max(i,j) & \text{pokud } \min(i,j) = 0 \\ \min \begin{cases} \mathrm{lev}_{a,b}(i-1,j)+1 \\ \mathrm{lev}_{a,b}(i,j-1)+1 \\ \mathrm{lev}_{a,b}(i-1,j-1)+1_{(a_i \neq b_j)} \end{cases} & \text{jinak} \end{cases} \tag{2.11}$$

kde $1_{(a_i \neq b_j)}$ je charakteristická funkce definována následovně:

$$1_{(a_i \neq b_j)} = \begin{cases} 0 & \text{jestliže } a_i = b_j \\ 1 & \text{jinak} \end{cases}$$

Na základě matematického popisu lze jednoduše implementovat rekurzivní algoritmus výpočtu Levenshteinovy vzdálenosti. Pro porovnání dlouhých řetězců je tato implementace nepoužitelná vzhledem k exponenciální časové složitosti. V praxi se proto využívá technik dynamického programování, díky kterým lze časovou složitost značně zredukovat.

2.3.2 Otisk dokumentu a algoritmus Winnowing

Otisk dokumentu (*document fingerprint*) slouží k jeho snadné identifikaci. Velikost otisku je mnohem menší než velikost samotného dokumentu. Potřebujeme-li z dokumentu získat určité informace, přičemž tyto informace jsou zahrnuty v otisku, pak je mnohem efektivnější využít otisk, nežli zpracovávat celý dokument.

Většina technik, které se snaží nalézt shodu částí dokumentů, rozdělí daný dokument na tzv. k-gramy. K-gram je označení pro sousedící podřetězec délky k. Parametr k je zvolen uživatelem. Například řetězci `abcdef` odpovídají po řadě tyto 3-gramy: `abc`, `bcd`, `cde`, `def`.

Abychom získali požadovaný otisk, je nutné na k-gramy aplikovat hešovací funkci. Otisk dokumentu tvoří podmnožina vzniklých hešů. Jeden z přístupů, jak vybrat heše pro otisk je tzv. *0 mod p*, kde p je pevně daná konstanta [15]. V takovém případě budou vybrány heše dělitelné hodnotou p. Pozice vybraných hešů nehraje roli, a tudíž neexistuje limit pro maximální vzdálenost dvou hešů. V praxi to znamená, že mezera mezi heši může být neomezeně velká. Nachází-li se shodné části dokumentů v této mezeře, pak je nebude možné odhalit. Tento problém se snaží napravit algoritmus Winnowing.

Winnowing definuje okno o velikosti w, kde w udává počet po sobě jdoucích k-gramů. Použitím tohoto okna se omezí maximální velikost mezery mezi dvěma vybranými heši. Zároveň je garantováno, že bude detekován každý sdílený podřetězec o délce alespoň $w + k - 1$. Algoritmus staví na dvou předpokladech týkajících se shodných podřetězců:

1. podřetězce, jejichž délka je větší nebo rovna hodnotě prahu garance t, budou vždy detekovány

2. podřetězce musí být delší než práh šumu k, aby mohly být detekovány

Konstanty t a k ($k \leq t$) jsou zvoleny uživatelem. Řetězce kratší než k neuvažujeme, neboť porovnáváme heše k-gramů. Větší hodnota k znamená větší důvěryhodnost shody, avšak zároveň negativně ovlivňuje citlivost vůči přeuspořádání obsahu dokumentu. Délka okna je dána jako $w = t - k + 1$. Za předpokladu rovnoměrně rozložených hodnot hešů technika *0 mod p* vybere celkem $1/p$ celkového počtu hešů. Naproti tomu algoritmus Winnowing vytvoří otisk dokumentu užitím $\frac{2}{w+1}$ celkového počtu hešů. [15]

Winnowing. *V každém okně vyber minimální hodnotu heše. Jestliže se v okně nachází více hešů s minimální hodnotou, vyber ten nejpravější z nich. Všechny vybrané heše nyní tvoří otisk dokumentu.* [15]

Činnost algoritmu Winnowing při výběru hešů pro získání otisku je znázorněna v tabulce 2.6.

Původní řetězec:	`functionaction`
Sekvence 4-gramů:	`func unct ncti ctio tion iona onac nact acti ctio tion`
Sekvence hešů 4-gramů:	`73 56 13 90 91 74 12 15 96 90 91`
Okna o délce 3 s heši:	`[73 56 13] [56 13 90] [13 90 91] [90 91 74] [91 74 12] [74 12 15] [12 15 96] [15 96 90] [96 90 91]`
Otisk dle algoritmu Winnowing:	`13 74 12 15 90`

Tabulka 2.6: Ukázka činnosti algoritmu Winnowing.

2.3.3 MOSS

MOSS (*Measure Of Software Similarity*) je systém provozovaný od roku 1997 sloužící k odhalování plagiátorství. Funguje na principu webové služby. Komunikace probíhá pomocí skriptu, který odešle vybrané soubory na vzdálený server k vyhodnocení. V současné době

podporuje jazyky C/C++, Java, Python, JavaScript a mnoho dalších. Výsledky jsou vizualizovány ve vygenerovaných HTML stránkách, kde si lze prohlédnout podezřele podobné úseky kódu bok po boku. [1]

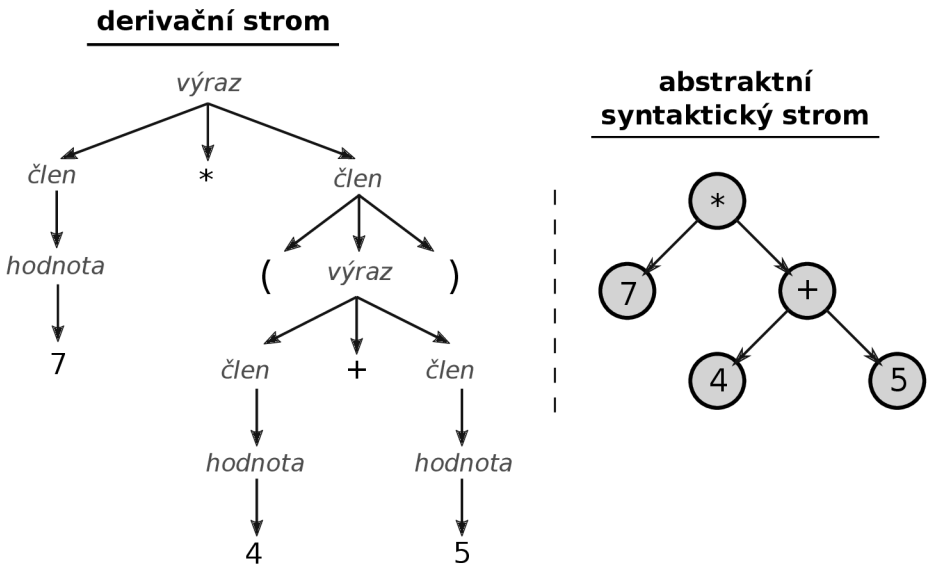
Konkrétní detaily implementace nejsou známy. Je však známo, že tento nástroj využívá otisky dokumentů a algoritmus Winnowing [15].

2.4 Porovnávání stromových struktur

Tokeny dodávané lexikálním analyzátorem zpracovává syntakticky analyzátor (*parser*), přičemž kontroluje syntaxi vstupního kódu. Během této činnosti vytváří interní datovou strukturu. Strukturou může být například derivační strom (*parse tree*) nebo jeho modifikace. Vnitřní uzly derivačního stromu odpovídají neterminálním symbolům gramatiky daného jazyka (výraz, příkaz, atd.) a listové uzly reprezentují terminální symboly (identifikátor, klíčové slovo, literál, aj.).

Z derivačního stromu lze vytvořit abstraktní syntaktický strom (AST). AST zachycuje strukturu zdrojového kódu a abstrahuje nevýznamné syntaktické elementy jazyka (např. středník jako oddělovač příkazů či čárka pro oddělení parametrů funkce). Derivační strom i AST pro výraz 2.12 lze vidět na obrázku 2.4.

$$7 * (4 + 5) \tag{2.12}$$



Obrázek 2.4: Derivační strom (*parse tree*) a abstraktní syntaktický strom (*abstract syntax tree*) pro výraz 2.12.

Detekce stejného/podobného kódu může spočívat v porovnání abstraktních syntaktických stromů a nalezení shodných podstromů. „Naivní“ porovnání každého podstromu s každým podstromem má pro AST o N uzlech časovou složitost $O(N^3)$ [2].

Techniky založené na porovnávání AST jsou robustnější vůči změnám ve zdrojovém kódu. To plyne z toho, že AST zachycuje strukturu programu a opomíjí drobné modifikace. Pro sestrojení AST je však nutná přítomnost jak lexikálního, tak i syntaktického analyzátoru. Tím vzniká úzké provázání mezi nástrojem a konkrétním jazykem.

Kapitola 3

Návrh řešení

Předchozí kapitola poskytla přehledové informace z oblasti detekce duplicitního kódu a také zmínila některé metriky, jež lze použít pro hodnocení kvality kódu. Nyní je doba předvést nabyté povědomí v praxi.

Nebudeme-li mít k dispozici úryvky, které mohou být zkopírovány, pak nejsme schopni pokročit dále. Proto se nutně musíme zaměřit na nalezení vhodného zdroje těchto úryvků. Programovacích jazyků stejně jako projektů, jež jsou pomocí nich implementovány, je nepřeberné množství. Obecné vyhodnocení vlivu kopírování kódu na kvalitu projektu je nad rámec této práce. Z toho důvodu se pokusíme nalézt odpověď pouze pro jeden konkrétní jazyk. Zřejmě nejdůležitější a zároveň nejsložitější část práce představuje nalezení shodných úryvků. Následné vyhodnocení ovlivňování kvality pak z velké míry záleží na úhlu pohledu, jakým na kvalitu kódu nahlížíme.

Následující podkapitoly se věnují jednotlivým podproblémům z předchozího odstavce a odůvodňují vybraný způsob řešení.

3.1 Volba programovacího jazyka

Ze zástupu existujících programovacích jazyků byl vybrán Python. Jedná se o velmi populární dynamicky typovaný skriptovací jazyk. Popularitu ostatně dokládá 3. místo v počtu originálních repozitářů na GitHubu (viz obrázek 3.1) nebo 1. místo v počtu položených otázek za měsíc na Stack Overflow (duben 2019).

Od většiny ostatních jazyků se odlišuje tím, že odsazení jednotlivých příkazů hraje roli a používá se k vymezení bloku kódu (namísto kupříkladu složených závorek). To jenom podtrhuje vlastnost, na které si jazyk zakládá – čitelnost kódu[1]. Jazyk je dostupný ve dvou verzích – starší Python 2 a novější Python 3, jenž není zcela zpětně kompatibilní se starší verzí. Jelikož oficiální podpora jazyka Python 2 končí v roce 2020, bude veškeré úsilí směřovat směrem k Pythonu 3.

3.2 Možné zdroje úryvků kódu

Středem pozornosti jsou webové služby/domény, jež shromažďují úryvky zdrojového kódu. Zejména pak ty, které obsahující velké množství těchto úryvků (řádově stovky tisíc), aby pravděpodobnost nalezení shody mezi úryvkem a kódem uvnitř projektu byla co největší.

[1] Více o filosofii jazyka shrnuje *The Zen of Python*: https://www.python.org/dev/peps/pep-0020/.

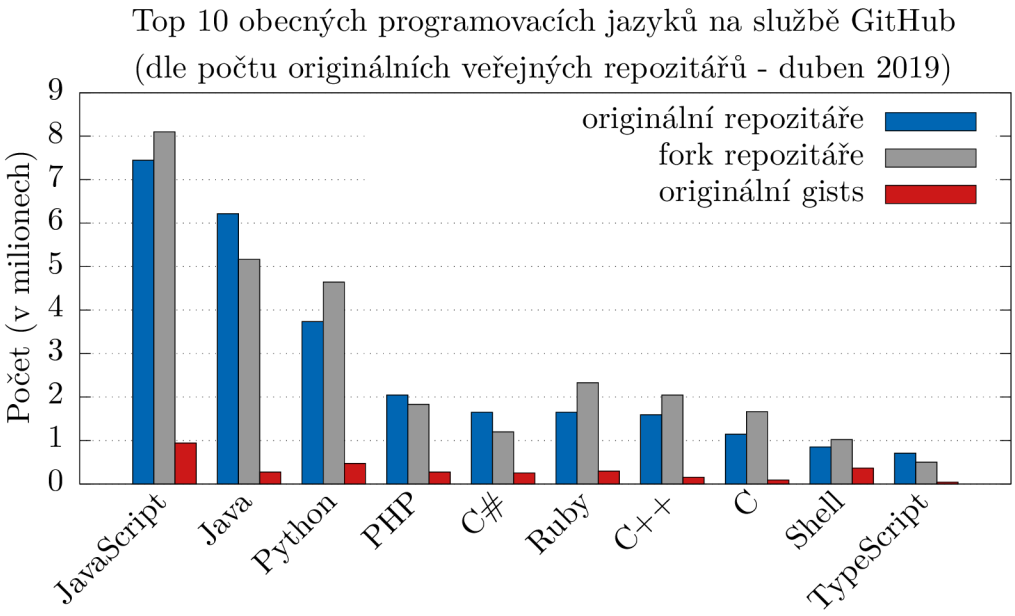
Toto kritérium splňují služby GitHub, Stack Overflow a Pastebin. Reálně použitelné příspěvky pro účely této práce lze získat pouze z domény Stack Overflow. Odůvodnění spolu s dalšími informacemi lze najít v následujících podsekcích.

3.2.1 GitHub

Služba GitHub je největším hráčem v oblasti hostování repozitářů používajících verzovací systém Git. Mimo samotnou správu verzí nabízí celou řadu dalších funkcí, díky kterým si získala velkou popularitu mezi vývojáři *open-source* projektů.

Jako úryvky kódu je možné charakterizovat tzv. *gists*. *Gists* obecně slouží ke sdílení textu, podobně jako je tomu například u služby Pastebin. Na rozdíl od Pastebin je každý *gist* zároveň git repozitář, a proto je lze verzovat nebo vytvářet forky.

V současné době jsou na GitHubu hostovány desítky miliónů veřejných repozitářů a řádově milióny *gists*. Počty repozitářů a *gists* pro deset nejpopulárnějších jazyků co do počtu originálních repozitářů lze vidět na obrázku 3.1.



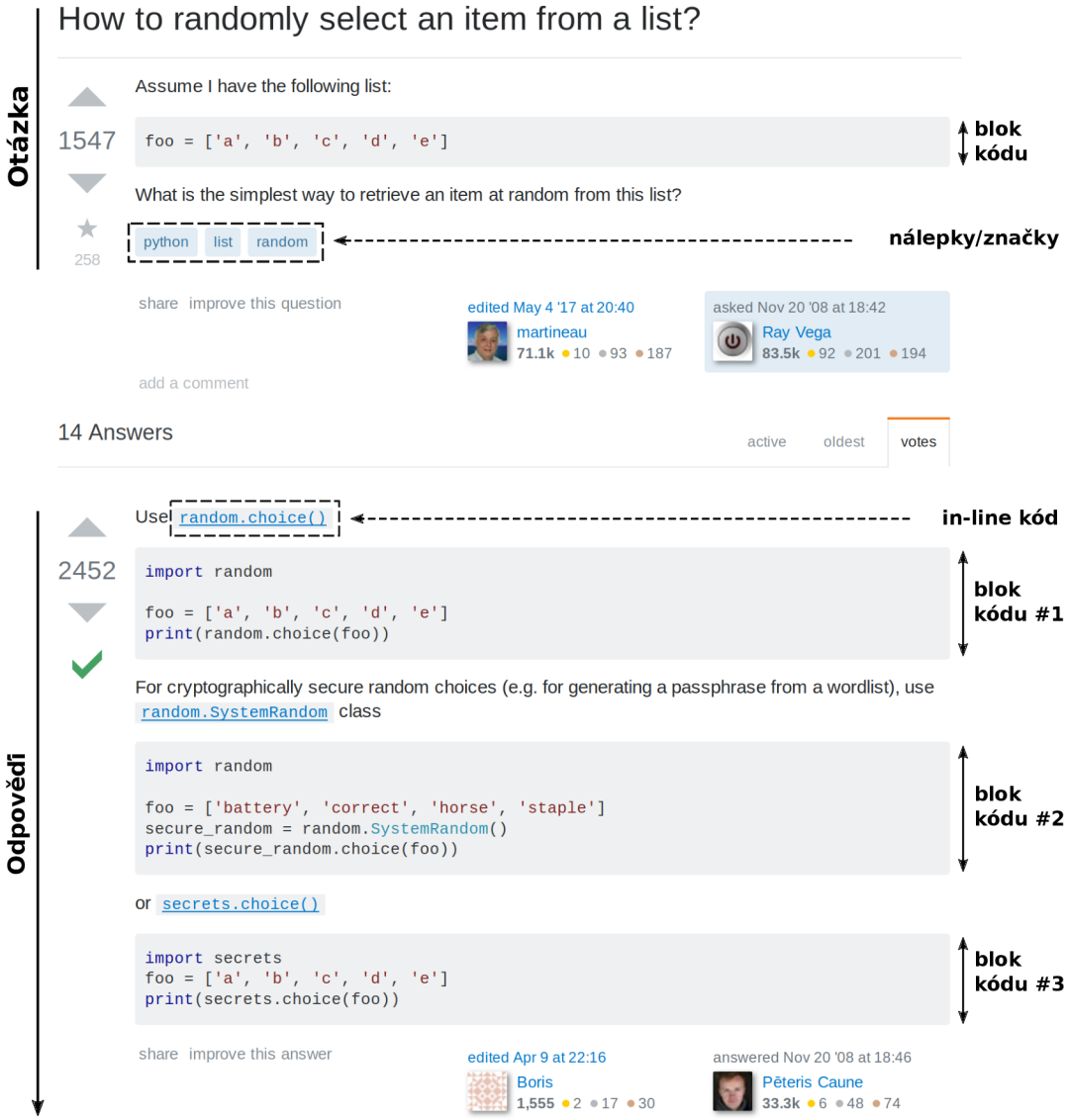
Obrázek 3.1: Počet originálních/fork repozitářů a *gists* na GitHubu.

Služba GitHub v současné době nenabízí možnost, jak programově sesbírat všechny *gists* napsané v určitém jazyce. Přesto, přihlédneme-li k hostování *open-source* repozitářů, lze GitHub s výhodou využít za účelem získání vzorku projektů. Nad těmito projekty bude následně provedeno vyhodnocení ovlivňování kvality.

3.2.2 Stack Overflow

Stack Overflow je mezi softwarovými vývojáři zřejmě nejnavštěvovanější doménou z řetězce stránek Stack Exchange. Jedná se o tzv. *Q&A* (*question-and-answer*) stránku, kde uživatelé mohou pokládat otázky nebo odpovídat na otázky ostatních. Otázky jsou spjaty s programovacími jazyky či programováním obecně a velmi často obsahují úryvky kódu.

Ukázky příspěvků ze služby Stack Overflow lze spatřit na obrázku 3.2. Každý příspěvek může obsahovat odstavce holého textu, *in-line* kód a bloky kódu. Bloky kódu i *in-line* kód se



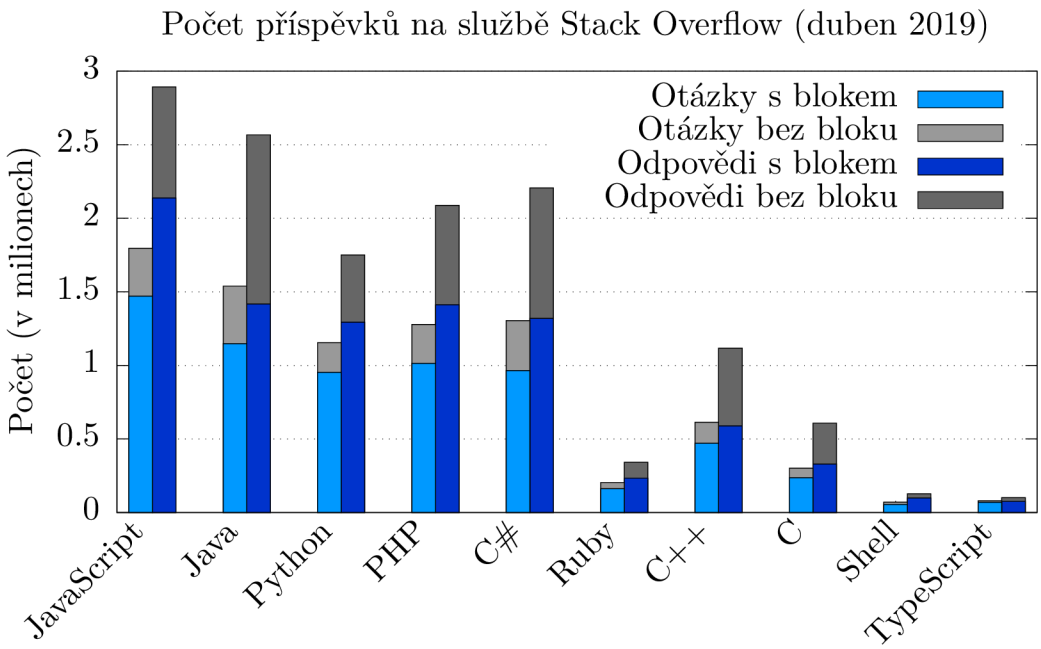
Obrázek 3.2: Ukázka příspěvků (otázka[2], odpověď[3]) ze služby Stack Overflow.

vkládají pomocí speciálního formátovaní při vytváření příspěvku. Bloky kódu představují potenciálně kopírované úryvky. Bloky však nutně nemusí obsahovat kód. Často obsahují výstup programu, ladící informace a podobně. Otázka může obsahovat až 5 nálepek/značek (*tag*). Ty říkají, jakých oblastí se daná otázka týká. Běžně otázka obsahuje nálepku/značku s názvem programovacího jazyka.

Počty otázek a odpovědí deseti vybraných jazyků lze vyčíst z grafu na obrázku 3.3. Data odpovídají počtu příspěvků obsahujících nálepku/značku shodující se s názvem jazyka. Jak je patrné, většina příspěvků obsahuje blok kódu.

Všechen obsah, na němž se podílejí uživatelé, stránek zaštiťovaných doménou Stack Exchange je k dispozici hned v několika podobách. „Živá“ data jsou přístupná skrze REST

[3] https://stackoverflow.com/q/306400, autor: https://stackoverflow.com/users/4872
[3] https://stackoverflow.com/a/306417, autor: https://stackoverflow.com/users/5821



Obrázek 3.3: Počet otázek a odpovědí na službě Stack Overflow rozdělených na základě přítomnosti bloku kódu.

API. Pro obsáhlejší analýzu dat je vhodné využít *Stack Exchange Data Explorer*[4], kde lze s daty interagovat pomocí příkazů jazyka SQL. Data v této databázi jsou synchronizována s těmi reálnými jednou týdně. Pro nejnáročnější aplikace slouží archivy (tzv. *stack exchange data dump*[5]) s daty ve formátu XML. Archivy jsou aktualizovány čtvrtletně.

Archiv obsahující všechny příspěvky ze služby Stack Overflow je ideálním zdrojem úryvků pro účely této práce.

3.2.3 Pastebin

Pastebin je doména zaměřená na sdílení libovolného textového obsahu. Ve většině případů je však předmětem sdílení zdrojový kód. Při vytváření nového příspěvku/úryvku lze navolit zvýraznění syntaxe. Na základě této vlastnosti lze filtrovat úryvky dle jazyka.

Z této domény však nelze získat více než několik desítek úryvků. Přístup k API, jež by umožňovalo získat větší množství relevantních příspěvků je k dispozici jen prémiovým uživatelům. Z toho důvodu nejsou úryvky ze služby Pastebin v této práci uvažovány.

3.3 Nalezení stejných/podobných úryvků

Implementace vlastního nástroje pro vyhledávání podobných/stejných úryvků by byla značně náročná. Přihlédneme-li navíc k množství již existujících volně dostupných nástrojů, bylo by nerozumné začínat od nuly. Zbývá tedy odpovědět na otázku, který z nich vybrat.

Nástroj by měl splňovat následující požadavky. S přiměřeným úsilím by jej mělo být možné rozšířit o nový jazyk. Z grafu na obrázku 3.3 lze vyčíst přibližný počet úryvků, které

[4] https://data.stackexchange.com/
[5] https://archive.org/details/stackexchange

bude nutné zpracovat – více než milion. Nástroj musí být schopen pracovat v takovémto měřítku. Tyto dva požadavky hrají v neprospěch nástrojů založených na porovnávání stromových struktur. Podstatnou roli hraje také původ úryvků. Jelikož služba Stack Overflow nijak neomezuje, co může nebo naopak nesmí být obsahem úryvku, je možné, že získaný úryvek vůbec nebude obsahovat zdrojový kód. Takových úryvků může být velké množství. Detekce těchto „falešných“ úryvků ve zdrojových souborech představuje plýtvání výpočetního času a prostředků.

Po zvážení těchto požadavků byl vybrán nástroj NiCad (viz kapitola 2.1.2). Nástroj byl použit pro nalezení duplicitního kódu ve 48 verzích jádra systému FreeBSD (dohromady 78 485 souborů a 60 578 220 řádků) za použití jednoho jádra procesoru (2.66 GHz) a 2 GB dostupné paměti, přičemž experiment trval přibližně 5.7 hodiny [6]. To dokládá dobrou škálovatelnost. Úvodní fáze zahrnující syntaktickou analýzu propustí dále jen kód, jenž je v souladu se syntaxí daného jazyka, a tím vytváří ochranu před většinou „falešných“ úryvků. Ve prospěch tohoto nástroje hraje také to, že podporuje hledání duplicitního kódu mezi dvěma systémy. Většina ostatních detektorů duplikátů se totiž soustředí na nalezení shodných úryvků jen v rámci daného projektu.

Nástroj již podporuje jazyk Python, nicméně poslední revize gramatiky vychází ze syntaxe Pythonu 2.5. Prvním úkolem proto bude přizpůsobení gramatiky tak, aby byla v souladu se syntaxí verze 3.6 a v současné době nejnovější verzí 3.7. Následně bude zapotřebí vytvořit transformační a normalizační pravidla, která umožní odhalení co největšího počtu shodných úryvků. Dále bude nutné nalézt vhodné hodnoty konfiguračních parametrů za účelem dosažení co největšího pokrytí a přesnosti. Licence, pod kterou je nástroj distribuován, umožňuje použití v této práci včetně případných úprav.

3.4 Vyhodnocení kvality kódu

Hodnocení kvality kódu je do značné míry subjektivní záležitostí. Neexistuje totiž univerzální postup aplikovatelný na různé projekty, pomocí kterého by bylo možné projekty mezi sebou porovnat. Pro někoho může být primárním ukazatelem kvality například čitelnost či udržovatelnost zdrojového kódu, zatímco pro druhé může být na prvním místě efektivita, bezpečnost nebo spolehlivost. Všechny vyjmenované vlastnosti jsou však obtížně měřitelné. Nicméně hovoří-li se o kvalitě kódu, velmi často jsou skloňovány softwarové metriky jako je cyklomatická složitost (viz kapitola 2.2.2) či některé z Halsteadových metrik (viz kapitola 2.2.1). Jak je zmíněno v úvodní kapitole, tak i míra duplicitního kódu může sloužit jako indikátor kvality.

V rámci této práce bude kvalita hodnocena právě podle obsahu duplicitního kódu. Služba GitHub, ze které bude pocházet vzorek projektů, nabízí možnost nahlášení chyby softwaru formou tzv. *issues*. U projektů, jež tuto funkcionalitu využívají, bude kvalita hodnocena navíc ještě podle tohoto měřítka. Cílem bude dokázat, že přítomnost úryvku kódu ze služby Stack Overflow má vliv na dříve zmíněné ukazatele.

Kapitola 4

Implementace

Tato kapitola navazuje na kapitolu předcházející a snaží se vyplnit rámce, které byly v předchozí sekci pouze načrtnuty. Hlavní pozornost je přitom upřena směrem k implementaci robustního mechanismu detekce stejných či podobných úseků kódu s využitím nástroje NiCad. Práce se konkrétně zaměřuje na detekci úryvků v jazyce Python verze 3.6 a 3.7.

Nástroj NiCad je implementován pomocí jazyků TXL, Turing+ a o propojení jednotlivých části se starají shell skripty (konkrétně pro Bash). Převážná část implementace byla napsána v jazyce TXL. TXL programy provádí transformace původního zdrojového kódu jednotlivých úryvků do podoby, která umožní nalezení co největšího počtu potenciálně zkopírovaných úseků. Jazyk TXL byl speciálně navržen pro takovéto transformace a leží někde na pomezí mezi logickými a funkcionálními jazyky. Přesto jej nelze srovnávat s jazyky jako je Haskell, Prolog nebo XSLT. Příloha A obsahuje velmi stručný úvod k jazyku TXL. Bez základních znalostí jazyka nemusí některé konstrukce, jež budou prezentovány, dávat smysl. Pro vyhodnocení ovlivňování kvality kódu, získání sady úryvků kódu a další pomocné účely byly vytvořeny krátké skripty v jazyce Python (verze 3).

4.1 Získání sady úryvků kódu

Za účelem získání sady úryvků kódu ze služby Stack Overflow bylo nutné stáhnout archiv `stackoverflow.com-Posts.7z` (z 2. prosince 2018). Tento archiv obsahuje soubor `Posts.xml` se všemi do té doby publikovanými příspěvky. Soubor má v dekomprimovaném stavu velikost 64 GB. Proces extrakce úryvků (bloků) kódu z jednotlivých příspěvků má na starost skript `extract_snippets.py`. Skript při spuštění vyžaduje dva parametry.

1. parametr – cesta k souboru s příspěvky
2. parametr – jméno adresáře, do něhož mají být úryvky extrahovány

Skript využívá knihovnu `xmltodict`, pomocí níž je možné zpracovávat rozsáhlé soubory ve formátu XML iterativním způsobem. Dále bylo využito knihovny `Beautiful Soup`, která značně usnadňuje práci s daty ve formátu XML či HTML.

S ohledem na povahu služby Stack Overflow se předpokládá, že potenciálně kopírovanými úryvky jsou úseky kódu obsažené v sekci s odpověďmi. Z toho důvodu nejsou bloky kódu uvnitř otázek předmětem extrakce. Doposud je na Stack Overflow dostupných 287 nálepek/značek obsahujících slovo `python` ve svém názvu. Drtivá většina z nich však co do počtu položených otázek nepřesáhne číslovku tisíc. Jelikož je cíleno pouze na Python verze 3.6 a 3.7, byly za relevantní prohlášeny nálepky/značky reprezentované v tabulce 4.1.

python	python-3.x	python-3.6	python-3.7

Tabulka 4.1: Nálepky/značky, kterých se týká extrakce.

Bloky kódu jsou v souboru s příspěvky ohraničeny XML značkami `<code>` a `</code>`. Tímto způsobem je však ohraničen i *in-line* kód. Extrakce se vztahuje pouze na bloky dlouhé alespoň 3 řádky, přičemž se do délky nezapočítávají prázdné řádky a řádky obsahující komentář. Identifikaci komentářů zajišťuje regulární výraz (`^\s*#` – symbol `#`, kterému od začátku řádku může předcházet libovolný počet mezer). Situace, kdy regulárnímu výrazu odpovídá text uvnitř víceřádkového řetězcového literálu, není považována za významnou. Omezení týkající se délky úryvku eliminuje zanedbatelně krátké úryvky včetně *in-line* kódu.

Skript pro každý blok kódu splňující předchozí podmínku vytvoří ve výstupním adresáři soubor s názvem `<id>_<index>.py`, kde `<id>` odpovídá jedinečnému identifikátoru příspěvků ze služby Stack Overflow a `<index>` udává pozici bloku kódu v rámci příspěvku počítáno od 0. Z odpovědi na obrázku 3.2, jejíž identifikátor je 306417, by byly vytvořeny 3 soubory – `306417_0.py` (blok kódu #1), `306417_1.py` (blok kódu #2) a `306417_2.py` (blok kódu #3).

Pomocí skriptu `extract_snippets.py` bylo získáno 1 313 440 úryvků kódu o celkové délce přesahující 15 milionů řádků.

4.2 Sada softwarových projektů

Softwarové projekty pochází ze služby GitHub. Pro jejich získání bylo využito GitHub API. Jako klíč byl zvolen počet hvězdiček daného projektu. Vybrány byly projekty s 5 000 a více hvězdičkami. Tuto podmínku splňovalo 286 projektů. Vyřazeny byly projekty, které obsahovaly méně než 30 zdrojových souborů. Zbytek byl manuálně zkontrolován a v případě, že se nejednalo o softwarový projekt byl rovněž vyřazen. Výsledkem tohoto procesu je vzorek 166 softwarových projektů.

Pro každý projekt byl následně vyhodnocen počet *issues*. Započítány jsou pouze ty *issues*, jejichž označení obsahuje slovo *bug* a zároveň neobsahuje slovo *not*. Jestliže je tímto způsobem označen *pull request*, pak je také započítán. Při sčítání nezáleží, v jakém je daný *pull request*/*issue* stavu, započítány jsou všechny (otevřené i uzavřené).

4.3 Detekce shodných/podobných úryvků

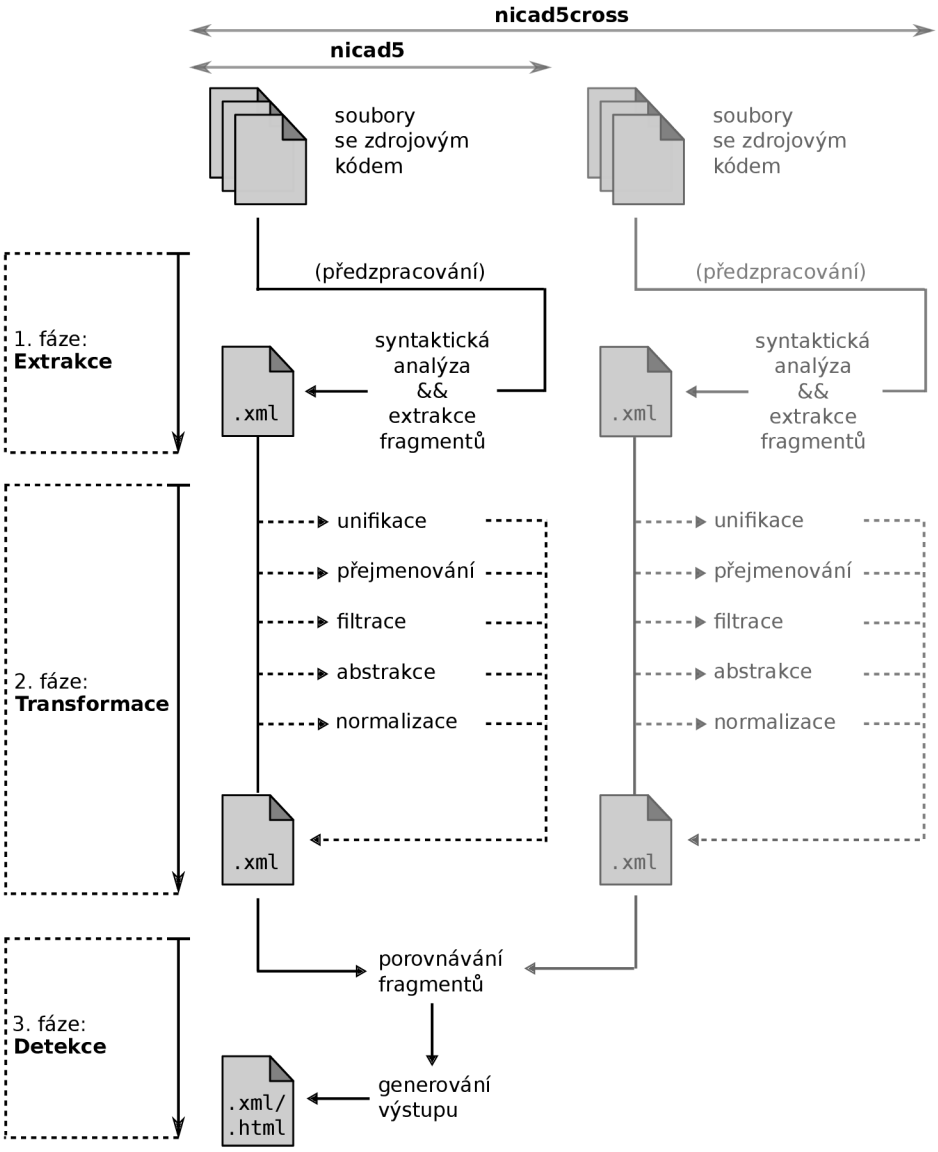
Každý softwarový projekt prošel detekčním algoritmem nástroje NiCad hned dvakrát. Nejprve bylo zjištěno, v jaké míře obsahuje daný projekt úryvky ze Stack Overflow (pomocí `nicad5cross`), následně bylo nutné nalézt míru obsahu duplicitního kódu pro účely hodnocení kvality (pomocí `nicad5`).

Činnost nástroje NiCad lze rozdělit do tří fází, přičemž výstupem každé z nich je vždy soubor ve formátu XML. V případě poslední fáze si lze vyžádat i HTML výstup. Fáze jsou graficky znázorněny na obrázku 4.1 a jejich popis je obsažen v následujících bodech.

1. **Extrakce** zahrnuje syntaktickou analýzu a vyjmutí zajímavých fragmentů (úseků kódu) ze zdrojových souborů. Soubory v jazyce Python ještě před samotnou syntaktickou analýzou musí projít předzpracováním. Zmíněné kroky jsou implementovány v jazyce TXL. Průběh jednotlivých kroků byl v rámci této práce značně poupraven.

2. **Transformace** je souhrnné označení pro řetězec TXL programů, kde každý článek provádí transformaci určitého typu. Transformace jsou aplikovány vždy v tomto pořadí – unifikace, přejmenování, filtrace, abstrakce a normalizace, nicméně libovolný článek (případně všechny) lze vynechat. Výstupem každého článku je XML soubor. Vstupem je buďto soubor s původními fragmenty (výsledek fáze extrakce), a nebo výstup předchozího článku. Nástroj sám o sobě podporuje pouze přejmenování, filtraci a abstrakci. Unifikace a normalizace jsou přínosem této práce.

3. **Detekce** zahrnuje textové porovnání fragmentů řádek po řádku a vygenerování výstupu s informacemi o detekovaných fragmentech.



Obrázek 4.1: Průběh jednotlivých fází nástroje NiCad.

S ohledem na provedené změny ve fázi extrakce a transformace, bylo nutné zasáhnout do způsobu generování výstupu. Jednalo se však pouze o zásah marginálního charakteru, a

proto nebude prezentován. Pro popis úprav úvodních dvou fázi byly vyčleněny následující podsekce.

4.3.1 Extrakce

Každý TXL program vyžaduje bezkontextovou gramatiku, na základě které provádí syntaktickou analýzu vstupního textu a buduje derivační strom. Jazyk TXL poskytuje komfortní prostředky, jak s tímto stromem pracovat. Díky tomu lze snadno vyextrahovat všechny podstromy odpovídající určitému neterminálnímu symbolu gramatiky (např. definici funkce). Tyto podstromy, označované jako fragmenty, jsou převedeny zpět do textové podoby užitím formátovaného tisku (tzv. *pretty-printing*). Výstupem fáze extrakce je soubor ve formátu XML. Text jednotlivých fragmentů uvnitř tohoto souboru je ohraničen značkami `<source>` a `</source>`.

Pro účely této práce byl extrahován fragment blok. Definice tohoto fragmentu byla v této práci rozšířena. Jedná se jednak o klasický blok kódu (neterminál `block`), ale také o celý vstupní soubor (neterminál `file_input`). Tyto dva neterminály představují jeden typ fragmentu a ve fázi detekce mezi nimi není rozlišováno. Blokem kódu je dle gramatiky jazyka Python posloupnost příkazů odsazená alespoň o jednu úroveň. To splňují těla složených příkazů, mezi něž patří cykly, definice funkcí/metod, tříd, apod. Fragment s celým souborem umožní jednak detekovat shodné soubory, ale také postihuje případ, kdy byl celý soubor (např. krátký úryvek ze Stack Overflow) vložen do těla složeného příkazu.

Jak už bylo zmíněno, zdrojový kód je zpracováván na základě bezkontextové gramatiky. Zpracováváme-li zdrojový kód v jazyce Python, pak zde nastává komplikace. Ta vychází z toho, že bílé znaky jsou významné a že jazyk rozlišuje mezi fyzickými a logickými řádky. Odsazení jednotlivých příkazů hraje roli a slouží k vymezení bloků kódu. Zpracování vnořených bloků je kontextově závislé. Vyžaduje totiž použití stavových proměnných (aktuální úroveň odsazení, počet mezer pro odsazení jedné úrovně) nebo stavového zásobníku. Bezkontextová gramatika nedokáže postihnout tyto vazby, a proto musí zdrojové soubory projít předzpracováním.

Předzpracování

Lexikální analyzátor jazyka Python generuje některé tokeny v závislosti na aktuálním kontextu. Je-li například přečten znak nového řádků, pak je vygenerován token `NEWLINE`. To však neplatí nacházíme-li se uvnitř závorek. Tyto závislosti nelze v jazyce TXL nijak promítnout. Proto je nutné každý soubor upravit tak, aby reprezentoval sekvenci tokenů, jež by byla vygenerována lexikálním analyzátorem jazyka Python. Teprve poté budeme schopni soubor zpracovat bezkontextovou gramatikou. Při předzpracování dochází k následujícím úkonům:

1. explicitní vyznačení začátku a konce bloku kódu (převzato)
2. sloučení explicitně spojených řádků (převzato)
3. sloučení implicitně spojených řádků (implementováno)
4. anotace původní pozice jednotlivých bloků formou komentáře (implementováno)

Vyznačením začátku a konce bloku se zbavíme závislosti na stavových proměnných a budeme schopni korektně zpracovat vnořené bloky. Konkrétní příklad této úpravy je na obrázku 4.2.

vstup

```
1  for i in range(10):
2      if i % 2:
3          print("even")
4      else:
5          print("odd")
```

výstup

```
1  for i in range(10): __INDENT__
2      if i % 2: __INDENT__
3          print("even")
4  __DEDENT__     else: __INDENT__
5          print("odd")
6  __DEDENT__ __DEDENT__
```

Obrázek 4.2: Explicitní vyznačení začátku a konce bloku kódu.

Jestliže je posledním znakem na řádku zpětné lomítko (a není součástí víceřádkového literálu nebo komentáře), pak se jedná o tzv. „explicitně spojené řádky“ a lexikální analyzátor jazyka Python negeneruje token pro nový řádek. Při předzpracování je nutné tyto řádky sloučit. Příklad úpravy je na obrázku 4.3.

vstup

```
1  string = "Hello " + \
2    "World"
```

výstup

```
1  string = "Hello " +   "World"
```

Obrázek 4.3: Sloučení explicitně spojených řádků.

Uvnitř kulatých, hranatých nebo složených závorek dochází k tzv. „implicitnímu spojování řádků“ – lexikální analyzátor opět negeneruje token nového řádku. Slabým místem původního způsobu předzpracování je to, že neslučuje implicitně spojené řádky. To vyžaduje přizpůsobení gramatiky a negativně ovlivňuje výsledky detekce. Je-li například na jednom řádku volání funkce se třemi argumenty, pak se připravíme o možnost detekovat stejné volání pouze s tím rozdílem, že argumenty budou rozmístěny na více řádcích. Opačný extrém může vzniknout například při inicializaci datové struktury větším počtem hodnot, kdy každá hodnota bude na novém řádku. Bude-li se identická inicializace vyskytovat i jinde, dojde k detekci velkého počtu duplicitních řádků, přičemž ve skutečnosti půjde jen o jeden příkaz. Z pohledu této práce jsou takovéto situace nežádoucí. Proto byla implementována pravidla, která sloučí implicitně spojené řádky. Příklad tohoto úkonu je na obrázku 4.4.

vstup

```
1  array = [
2     42,
3     88,
4     97
5  ]
```

výstup

```
1  array = [   42,   88,   97 ]
```

Obrázek 4.4: Sloučení implicitně spojených řádků.

Slučování řádků vede k tomu, že číslo řádku s daným příkazem na vstupu neodpovídá číslu řádku na výstupu. Tyto informace jsou však nutné při generování výstupu. Z toho důvodu byl implementován mechanismus vyznačení délky fragmentů formou komentářů

přidaných za poslední příkaz. Komentáře mají formát `#<start>,<end>`, přičemž `<start>` odpovídá číslu řádku, kde fragment začíná a `<end>` udává, kde fragment končí. První komentář se vztahuje k celému souboru, další komentáře pak odpovídají jednotlivým blokům kódu. Kompletní proces předzpracování souboru je znázorněn na obrázku 4.5.

vstupní zdrojový soubor

```
1    # This comment will be deleted
2    for i in range(10):
3      print(i)
4    # This comment will be deleted
```

výstup procesu předzpracování

```
1  for i in range(10): __INDENT__
2    print(i)
3  __DEDENT__ #2,3
4  #3,3
```

Obrázek 4.5: Proces předzpracování vstupního souboru.

Gramatika

Po předzpracování jsme schopni provést syntaktickou analýzu a vyextrahovat fragmenty. Nutná je však bezkontextová gramatika pro Python verze 3.6 a vyšší. Původní gramatika (pro Python 2.5) posloužila jako odrazový můstek při vytváření gramatiky nové. Zdrojem informací při tvorbě gramatiky byl referenční manuál jazyka[1]. Do gramatiky bylo nutné zanést také změny týkající se lexikální struktury jazyka. Ukázka definice neterminálu pro cyklus `while` a definice nového typu rozpoznávaného tokenu `decinteger` je na obrázku 4.6.

```
define while_statement
    'while [test] ': [suite]
    [opt else_clause]
end define
```

```
tokens
    decinteger  "[123456789](_?\d)*"
              |  "0(_?0)*"
end tokens
```

Obrázek 4.6: Definice neterminálu `while_statement` (vlevo) a nového typu tokenu `decinteger` pomocí regulárního výrazu (vpravo).

Gramatika byla validována na zdrojových souborech webového frameworku Django. Framework obsahuje 689 souborů. Zpracování těchto souborů TXL gramatikou nenalezlo žádnou syntaktickou chybu.

Extrakce fragmentů

TXL program pro extrakci fragmentů v úvodu obsahuje direktivu `include "python3.grm"`. Tato direktiva zpřístupní dříve definovanou gramatiku.

Program byl přizpůsoben tak, aby zpracoval anotační komentáře s délkami jednotlivých bloků. Při extrakci dochází k odstranění dokumentačních řetězců (tzv. *docstring*). V případě, že by odstraněním dokumentačního řetězce vznikl nevalidní kód (např. ve chvíli, kdy se jedná o jediný příkaz těla funkce/metody, třídy), pak je tento řetězec ponechán.

[1] https://docs.python.org/3/reference/

Výsledkem procesu předzpracování je nový soubor s příponou `.pyindent`. Tento soubor je předán na vstup extrakčnímu programu, který vyprodukuje XML výstup. Vstup a výstup fáze extrakce jakožto celku je znázorněn na obrázku 4.7.

Ze sady 1 313 440 úryvků ze služby Stack Overflow bylo získáno 2 129 008 fragmentů. Fragmenty dohromady obsahují více než 19 milionů řádků zdrojového textu. Celkem 605 195 úryvků (přibližně 46 % celkového počtu) neprošlo syntaktickou analýzou, a tudíž z nich nebyl vyextrahován žádný fragment. Ze všech syntaktických chyb byl náhodně vybrán vzorek o velikosti 50. Tento vzorek byl zkontrolován, přičemž se ukázalo, že za většinou z nich stojí nekompatibilita mezi Python 2 a 3. Ve všech případech se skutečně jednalo o nevalidní kód z pohledu syntaxe jazyka Python 3.

vstupní zdrojový soubor (`test.py`)

```
"""Module docstring"""
class Test(object):
    """Class docstring"""

    def __init__(self):
        """Method docstring"""
        self.value = 42

    def value(self):
        """Return value"""
```

XML výstup

```
<source file="test.py.pyindent"
 startline="1" endline="10">
class Test(object) :
__INDENT__
    def __init__(self) :
    __INDENT__
        self . value = 42
    __DEDENT__
    def value(self) :
    __INDENT__
         """Return value"""
     __DEDENT__
 __DEDENT__
 </source>
```

Obrázek 4.7: Fáze extrakce (vyobrazen pouze 1. fragment výstupu z celkových 4).

4.3.2 Transformace

Jelikož transformace stále obstarávají TXL programy, tak i zde dochází k syntaktické analýze a konstrukci derivačního stromu. Stále je nutný import gramatiky, bez které bychom tyto transformace nebyli schopni uskutečnit. Celkem je k dispozici 5 různých typů transformací – unifikace, přejmenování, filtrace, abstrakce a normalizace. Unifikace a normalizace byly implementovány v rámci této práce, zbylé tři byly převzaty.

Přejmenování slouží k nahrazení identifikátorů zástupným symbolem. Existují dvě varianty:

1. slepé – nahrazení všech identifikátoru znakem `x`
2. konzistentní – nahrazení všech výskytů stejného identifikátoru symbolem `x<N>`, kde `<N>` udává pozici identifikátoru v rámci fragmentu (1. identifikátor bude označen jako `x1`, další jako `x2`, ...)

Při filtraci dochází k odstranění všech neterminálních symbolů daného typu z derivačního stromu, a tedy k odstranění odpovídajícího zdrojového textu.

Abstrakce slouží k nahrazení části zdrojového textu zástupným symbolem. Argumentem je název neterminálního symbolu, zástupným symbolem je pak právě tento název. Abstrahujeme-li neterminál `expression`, pak bude příkaz `x = (2+3) * 7` nahrazen za `x = expression`.

Unifikace byla implementována tak, aby plnila dva účely. Prvním z nich je odstranění příliš malých fragmentů. Jako příliš malé byly označeny fragmenty, jež obsahují méně než 5 příkazů a zároveň neobsahují složený příkaz. Podstatnější je však druhý účel – sjednocení zdrojového kódu do stejného tvaru.

Python 3 umožňuje do zdrojového kódu vkládat tzv. typové anotace[2] (*type hints*). Použití těchto anotací je volitelné, a proto jsou při unifikaci odstraněny. Implementace pravidla pro odstranění typové anotace proměnné je znázorněna na obrázku 4.8.

```
rule deleteVarTypeHint
    replace [expression_statement]
        Variable [testlist_star_expr] ': _ [test] '= Value [test]
    by
        Variable '= Value
end rule
```

Obrázek 4.8: Pravidlo pro odstranění typové anotace proměnné.

Python umožňuje zapsat více jednoduchých příkazů na jeden řádek, jestliže jednotlivé příkazy oddělíme středníkem. Při unifikaci jsou příkazy tvořící takovouto sekvenci umístěny na zvláštní řádek a středníky jsou odstraněny.

V případě, že tělo složeného příkazu obsahuje pouze jednoduché příkazy, pak existují dvě možnosti zápisu. Konvencí je zapisovat příkazy tvořící tělo jako blok kódu, tedy na nový řádek odsazený o jednu úroveň. Nicméně syntaxe Pythonu umožňuje také zápis na jeden řádek společně se složeným příkazem. Unifikace transformuje tělo složeného příkazu do konvenční podoby. Konkrétní příklad unifikace fragmentu je demonstrován na obrázku 4.9.

Normalizace je posledním článkem v řetězci transformací. Jejím cílem je upravit fragmenty do podoby vhodné pro fázi detekce. Během této transformace dochází k odstranění řádků se symboly `__INDENT__`/`__DEDENT__`. Původní přístup žádnou normalizaci neprovádí, a proto jsou při následné detekci i tyto řádky předmětem porovnání. To negativně ovlivňuje jak dobu porovnávání fragmentů, tak i výsledky samotné detekce. Budeme-li porovnávat dva fragmenty, přičemž oba obsahují blok kódu odsazený o tři úrovně, pak vždy nalezneme shodu minimálně 6 řádků. Je-li parametr pro minimální délku detekovaného fragmentu menší než 6, pak bude detekován klon, a to i v případě, že se fragmenty ve všech ostatních řádcích liší.

Při testovacích bězích detekce úryvků ze Stack Overflow uvnitř projektu téměř vždy došlo k identifikaci shodného těla metody `__init__`. Tělo této metody obsahuje sekvenci triviálních příkazů tvaru `self.x = x`. Jedním způsobem, jak zamezit detekci těchto fragmentů, je zvýšit parametr pro minimální délku detekovaného fragmentu. Tím se ale připravíme o možnost detekovat zajímavé fragmenty (mimo tělo metody `__init__`), které nedosahují minimální délky. Z toho důvodu byl zvolen jiný přístup. V případě, že se daný fragment skládá pouze z jednoduchých příkazů (neobsahuje cyklus, podmíněný příkaz, apod.), pak jsou jednotlivé příkazy sloučeny do skupin po třech. Každá skupina je na zvláštním řádku.

[2] Více o typových anotacích na: https://www.python.org/dev/peps/pep-0484/.

vstupní fragment

```
<source file="foo.py.pyindent"
 startline="1" endline="3">
def foo(x : int) -> int :
__INDENT__
    if x < 0 : return - 1;
    x += 1; return x;
__DEDENT__
</source>
```

výstupní fragment

```
<source file="foo.py.pyindent"
 startline="1" endline="3">
def foo(x) :
__INDENT__
    if x < 0 :
    __INDENT__
        return - 1
    __DEDENT__
    x += 1
    return x
__DEDENT__
</source>
```

Obrázek 4.9: Fragmentu před a po unifikaci.

Příkazy v rámci jedné skupiny jsou odděleny středníkem tak, aby byla zachována korektní syntaxe. Touto transformací zajistíme, že „triviální“ fragmenty budou detekovány, pouze pokud budou 3krát delší, než je hodnota parametru pro minimální délku fragmentu. Ukázka normalizace „triviálního“ fragmentu je znázorněna na obrázku 4.10.

vstupní fragment

```
<source file="init.py.pyindent"
 startline="4" endline="8">
__INDENT__
    self . name = name
    self . surname = surname
    self . number = number
    self . city = city
    self . state = state
__DEDENT__
</source>
```

výstupní fragment

```
<source file="init.py.pyindent"
 startline="4" endline="8">
    self . name = name; self . s
urname = surname; self . number
= number
    self . city = city; self . s
tate = state
</source>
```

Obrázek 4.10: Normalizace fragmentu složeného pouze z jednoduchých příkazů.

4.3.3 Konfigurace nástroje a detekce

Konfigurační parametry nástroje jsou uloženy v souboru. Soubor obsahuje řádky ve tvaru `<key>=<value>`, kde `<key>` je název parametru a `<value>` je jeho hodnota. Při spuštění nástroje je nutné specifikovat jméno konfiguračního souboru. Fragment je elementární, dále nedělitelnou jednotkou při detekování duplikátu. Detekovat lze pouze shodu celých fragmentů, nikoliv jejich částí. Fragment menší než je hodnota parametru `minsize` či větší než je hodnota parametru `maxsize` je ignorován. Parametr `threshold` udává míru variability mezi fragmenty a ovlivňuje algoritmus porovnání. Hodnota 0.0 značí nulovou toleranci – porovnány jsou pouze fragmenty stejné délky a to řádek po řádku. Hodnota 0.3 slouží k na-

lezení fragmentů, které se liší v maximálně 30 % řádků. Je-li fragment dlouhý 10 řádku a `threshold=0.3`, pak bude porovnáván pouze s fragmenty o délce 7 – 13 řádků. Počet shodných řádků mezi dvěma fragmenty při nenulové hodnotě parametru `threshold` je zjištěn pomocí algoritmu nalezení nejdelší společně podposloupnosti (viz kapitola 2.1.3).

nicad5cross

```
threshold=0.0
minsize=7
maxsize=2500
transform=unify
rename=none
filter=import_statement
abstract=literal
normalize=py3-normalize-blocks
cluster=yes
report=yes
include=""
exclude="[Tt]est"
```

nicad5

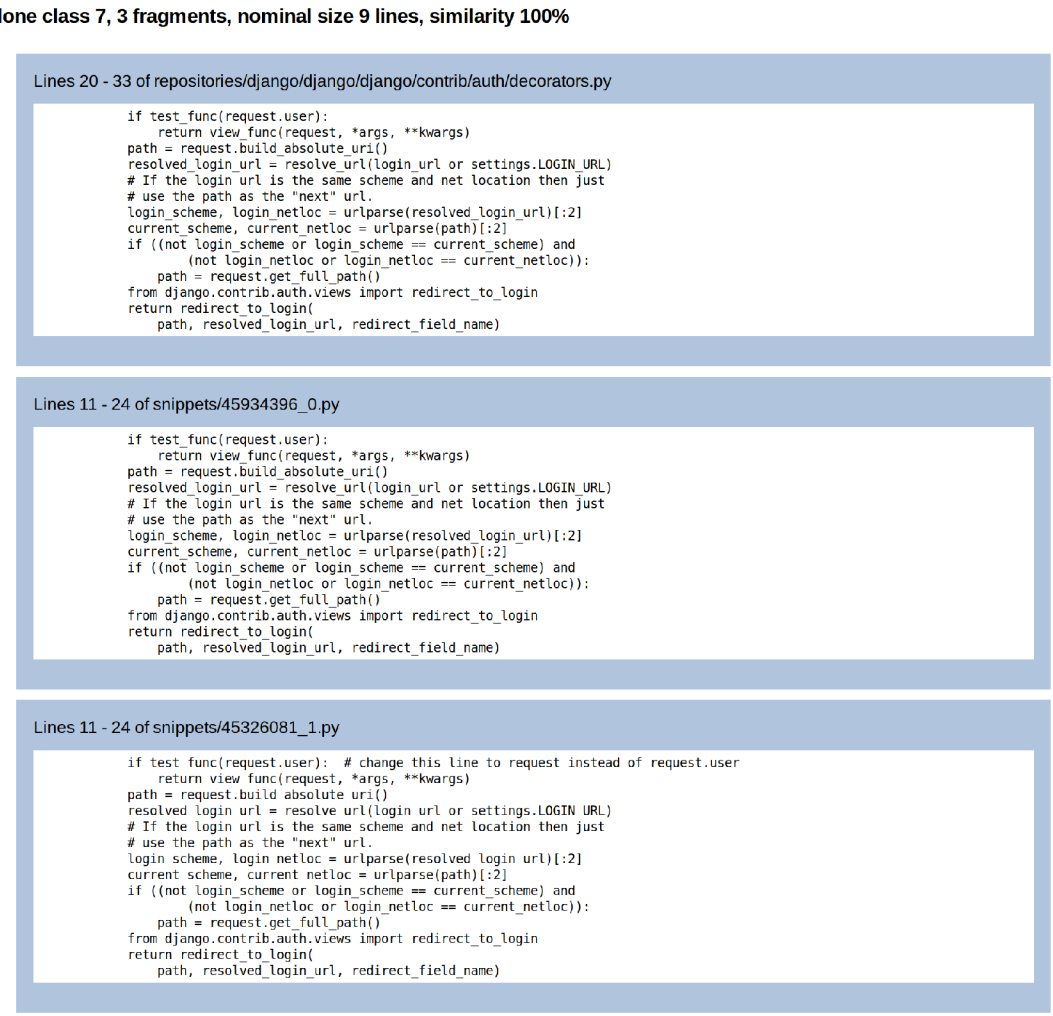
```
threshold=0.3
minsize=10
maxsize=2500
transform=unify
rename=consistent
filter=import_statement
abstract=literal
normalize=py3-normalize-blocks
cluster=yes
report=yes
include=""
exclude="[Tt]est"
```

Obrázek 4.11: Konfigurační parametry nástroje NiCad pro nalezení příspěvků ze Stack Overflow uvnitř projektu (vlevo) a vyhodnocení míry obsahu duplicitního kódu (vpravo).

Výsledky této práce vycházejí z konfiguračních parametrů znázorněných na obrázku 4.11. Pro nalezení úryvků ze služby Stack Overflow uvnitř softwarového projektu byl nastaven parametr `threshold` na hodnotu 0, díky čemuž jsme schopni projekt prohledat v řádu jednotek minut. Parametr `minsize` o velikosti 7 se ukázal jako optimální pro eliminaci běžných idiomů jazyka. Transformace zahrnuje unifikaci a vyfiltrování příkazů *import*, jenž jsou z pohledu duplicitního kódu nezajímavé. Rovněž dochází k abstrakci literálů a normalizaci. Identifikátory nebyly přejmenovány, neboť to vede k extrémnímu nárůstu falešně pozitivních fragmentů (zhoršení přesnosti). Nalezené páry jsou agregovány do tříd (parametr `cluster`), zároveň je vygenerován HTML výstup (parametr `report`). HTML výstup byl využíván zejména při hledání optimálních hodnot parametrů a sloužil k vizuální kontrole detekovaných fragmentů. Ukázka HTML výstupu je na obrázku 4.12. Soubory, jejichž název (popřípadě cesta k nim) obsahují slovo `test` nebo `Test`, jsou nástrojem zcela ignorovány (parametr `exclude`).

Pro nalezení duplicit v rámci projektu byla použita kombinace předchozích hodnot a výchozího nastavení nástroje NiCad. Výchozí nastavení hledá duplicitní kód typu III, čemuž odpovídá nenulová hodnota parametru `threshold`.

Ze vzorku 166 projektů se nástrojem NiCad nepodařilo zpracovat 5 z nich. U těchto 5 projektů došlo k ukončení běžícího programu ze strany operačního systému. Důvodem je zřejmě chyba v implementaci jazyka TXL a Turing+. Ze zbylých 161 projektů byla detekována shoda úryvku ze služby Stack Overflow a kódu uvnitř projektu v 76 případech. Duplicitní kód byl detekován v 132 projektech.



Obrázek 4.12: Výstup nástroje NiCad ve formě HTML stránky.

4.3.4 Vylepšení do budoucna

V této práci byl nástroj NiCad použit způsobem, který autoři nástroje zřejmě vůbec nezamýšleli. Nalezení shody mezi úryvky ze služby Stack Overflow na straně jedné a zdrojovým kódem softwarového projektu na straně druhé je velmi specifický úkol. Specifický s ohledem na povahu sady úryvků – obrovské množství relativně malých souborů. Soubory o velikosti stovek MB nebo několika GB jsou běžné a souborové systémy jsou pro práci s nimi velmi dobře optimalizovány. Naopak při práci s řádově miliony malých souborů je již zapotřebí mnohem větší režie.

První vylepšení by spočívalo v agregaci jednotlivých úryvků do jednoho souboru (např. ve formátu XML) a v adekvátní úpravě úvodní fáze extrakce fragmentů. Tyto změny by bylo nutné reflektovat při generování výstupu. Například pro jazyk Python, který před samotnou extrakcí fragmentů vyžaduje předzpracování úryvků, by toto vylepšení znamenalo významné zkrácení doby extrakce. Nad každým úryvkem/souborem je totiž nutné spustit dva programy, což při velkém počtu úryvků vyústí ve velké režijní náklady operačního a souborové systému. Pomocí tohoto vylepšení by velká část těchto režijních nákladů odpadla.

Implementace jazyka TXL zahrnuje interpret, překladač, debugger a také profiler tohoto jazyka. Jelikož na optimalizace TXL kódu nezbyl prostor, tak tato oblast skýtá možnosti pro další zefektivnění procesu transformace zdrojového kódu.

Za zmínku rovněž stojí poslední fáze sestávající z porovnávání fragmentů a generování výstupu, a to z důvodu poněkud netradičního implementačního jazyka Turing+. Uživatelská základna tohoto jazyka nebude příliš početná a jen těžko lze odhadnout, jak dlouho bude překladač tohoto jazyka udržován. Z dlouhodobého hlediska nebo při nutnosti rozsáhlejších zásahů do stávající implementace by bylo rozumnější implementovat tuto funkcionalitu pomocí konvenčních programovacích jazyků jako je Java nebo C++.

Při pohledu na nástroj NiCad jako na celek přichází v úvahu rozšíření množiny podporovaných jazyků. Z tohoto pohledu by bylo vhodné zaměřit se na v dnešní době populární jazyky jako je JavaScript, TypeScript nebo Go. Realizace tohoto rozšíření spočívá zejména v sepsání korektní TXL gramatiky pro daný jazyk a přímo nevyžaduje transformační a normalizační pravidla podobná těm v sekci 4.3.2.

Kapitola 5

Vyhodnocení

Ovlivňování kvality kódu se pokusíme posoudit pomocí korelační analýzy. Korelační analýza slouží k nalezení vazby mezi dvěma proměnnými, přičemž každá proměnná je reprezentována posloupností dat. Tato posloupnost obsahuje zaznamenané hodnoty dané proměnné. Proměnnou může být například počet *issues* projektu. V tom případě by posloupnost dat obsahovala počet *issues* všech zaznamenaných/analyzovaných projektů.

Výstupem analýzy bude korelační koeficient a tzv. p-hodnota. Korelační koeficient je číslo v intervalu -1 a 1, které udává, jak je závislost jedné proměnné na druhé silná. Čím více se absolutní hodnota korelačního koeficientu blíží 1, tím je vztah silnější. Hodnota nula značí absenci jakéhokoli vztahu mezi proměnnými. Jestliže je koeficient kladný, pak obě proměnné současně buďto rostou, a nebo klesají. V případě, že jedna proměnná roste a druhá klesá nebo naopak, je korelační koeficient záporný.

Hodnota korelačního koeficientu sama o sobě nestačí. Máme-li malé množství dat, pak získaná hodnota koeficientu není statisticky významná. Běžným způsobem, jak určit statistickou významnost je pomocí p-hodnoty [17]. Jestliže korelační koeficient = 0.5 a p-hodnota = 0.01, pak to znamená, že je 1% pravděpodobnost, že data nabývají stávajících hodnot nebo extrémnějších (více korelovaných) za předpokladu, že mezi proměnnými ve skutečnosti žádný vztah není. Předpoklad, že mezi proměnnými/daty není vztah se označuje jako nulová hypotéza. Nízká hodnota p-hodnoty obvykle vede k zamítnutí nulové hypotézy. Existenci vztahu mezi proměnnými/daty označujeme jako alternativní hypotézu. O tom, na kterou stranu se naklonit, rozhoduje tzv. hladina významnosti testu, značená α. Tato hladina se stanovuje před samotným vyhodnocením testu a běžně se volí hodnota 0.05 [18].

- Jestliže p-hodnota $\leq \alpha$, pak je nulová hypotéza zamítnuta ve prospěch alternativní hypotézy.
- Jestliže p-hodnota $> \alpha$, pak nulovou hypotézu nezamítáme (a alternativní hypotéza se nepotvrdila).

Korelačních koeficientů existuje více. V rámci této práce byl využit Pearsonův a Spearmanův korelační koeficient. Pearsonův korelační koeficient, značený jako r, udává jestli je mezi daty lineární závislost (jak dobře lze daná data proložit přímkou). Naproti tomu Spearmanův korelační koeficient, značený jako ρ (*rho*), říká, jak dobře lze data popsat pomocí monotónní funkce.

Výsledky detekce shodných úryvků a informace o počtech *issues* jsou v různých souborech. Z toho důvodu byl implementován skript `harvest.py`, který posbírá všechny dílčí výsledky a tyto data převede na standardní výstup ve formě JSON objektu. Formát záznamu JSON objektu (pár klíč – hodnota) je znázorněn na obrázku 5.1. Výstupní JSON

objekt obsahuje takovýto záznam pro každý projekt, jenž byl úspěšně zpracován nástrojem NiCad. Hodnota klíče `issues` udává počet *issues* projektu. Klíč `crossclones` obsahuje výsledky detekce detekce úryvků ze Stack Overflow uvnitř projektu a klíč `duplicates` výsledky detekce duplicitního kódu. Význam ostatních klíčů je uveden v tabulce 5.1.

```
<project>: {
    "issues": <n>,
    "crossclones": {
        "npairs": <n>,
        "nclasses": <n>,
        "nlines_total": <n>,
    },
    "duplicates": {
        "npairs": <n>,
        "nclasses": <n>,
        "nlines_total": <n>,
    }
}
```

```
"tensorflow/models": {
    "issues": 112,
    "crossclones": {
        "npairs": 4,
        "nclasses": 1,
        "nlines_total": 64,
    },
    "duplicates": {
        "npairs": 273,
        "nclasses": 116,
        "nlines_total": 5253,
    }
}
```

Obrázek 5.1: Formát záznamu JSON objektu s konkrétním příkladem.

Klíč	**Význam**
`npairs`	počet nalezených párů
`nclasses`	počet nalezených tříd
`nlines_total`	celkem zasažených řádků

Tabulka 5.1: Význam hodnot jednotlivých klíčů JSON objektu s výsledky nástroje NiCad.

Vyhodnocení dat provádí skript `evaluate.py`, jenž na svém vstupu očekává soubor s výstupem skriptu `harvest.py`. Skript používá knihovnu `scipy`, ve které jsou implementovány funkce pro výpočet Pearsonova i Spearmanova korelačního koeficientu včetně p-hodnoty.

5.1 Statistické testy

Hladina významnosti testu byla stanovena na hodnotu $\alpha = 0.05$. Skript `evaluate.py` byl implementován tak, aby dokázal poskytnout odpověď na následující tři otázky.

1. Existuje vztah mezi přítomností úryvku ze Stack Overflow uvnitř projektu a přítomností duplicitního kódu ?

2. Existuje vztah mezi objemem úryvků ze Stack Overflow uvnitř projektu a počtem *issues* daného projektu na službě GitHub ?

3. Je vztah mezi mezi objemem úryvků ze Stack Overflow a počtem řádků tvořících duplicitní fragmenty ?

5.1.1 Otázka č. 1

První otázku lze rovněž formulovat jako: „Je možné jen na základě informace o přítomnosti úryvku ze Stack Overflow očekávat zhoršenou kvalitu projektu (přítomností duplicitního kódu) ?“. Pro zodpovězení první otázky byl vyhodnocen Pearsonův korelační koeficient.

V tomto případě máme dvě sekvence dat. Jedna reprezentuje přítomnost duplicitního kódu a druhá přítomnost příspěvku ze Stack Overflow. Přítomnost, respektive absence je reprezentována hodnotou 1, respektive 0. Výsledek porovnání těchto dvou sekvencí je v tabulce 5.2.

korelační koeficient	***p*-hodnota**
$r = 0.151826$	0.054530

Tabulka 5.2: Korelační koeficient a p-hodnota pro otázku č. 1.

S ohledem na p-hodnotu, jež je větší než stanovená hodnota α, nebyla nulová hypotéza vyvrácena. Vztah mezi přítomností úryvků ze Stack Overflow a přítomností duplicitního kódu se nepodařilo prokázat.

5.1.2 Otázka č. 2

Při hledání odpovědi na druhou otázku byl korelační koeficient vyhodnocen několikrát. První proměnnou reprezentuje sekvence obsahující počty *issues* softwarových projektů. Druhou proměnnou postupně reprezentovaly výsledky detekce úryvků ze Stack Overflow (hodnoty klíčů v tabulce 5.1). Pro tyto dvojice proměnných byl postupně vyhodnocen jak Pearsonův, tak i Spearmanův korelační koeficient. Součástí výsledku je název, u něhož je korelace s počtem *issues* největší a zároveň p-hodnota $\leq \alpha$. Výsledek vyhodnocení je znázorněn v tabulce 5.3.

klíč	**korelační koeficient**	***p*-hodnota**
`nclasses`	$r = 0.318729$	$8.336748e-05$

Tabulka 5.3: Korelační koeficient a p-hodnota pro otázku č. 2.

Jelikož je p-hodnota menší než hladina významnosti testu α, nulovou hypotézu se podařilo vyvrátit ve prospěch alternativní. Přítomnost příspěvku ze Stack Overflow je ve vztahu s počtem *issues*, přičemž největší závislost je na počtu různých úryvků ze Stack Overflow uvnitř projektu. Různé úryvky jsou ve výsledcích detekce reprezentovány různými třídami a jejich počet vyjadřuje klíč `nclasses`.

5.1.3 Otázka č. 3

Nyní provádíme stejný výpočet jako u otázky č. 2 jen s tím rozdílem, že počet *issues* nahradíme celkovým počtem duplicitních řádků. Výsledek vyhodnocení je prezentován v tabulce 5.4.

klíč	**korelační koeficient**	***p*-hodnota**
`nlines_total`	$r = 0.361128$	$2.522370e-06$

Tabulka 5.4: Korelační koeficient a p-hodnota pro otázku č. 3.

Vzhledem k p-hodnotě lze nulovou hypotézu vyvrátit ve prospěch alternativní. Hodnota korelačního koeficientu značí slušnou závislost mezi proměnnými. Největší závislost byla nalezena mezi celkovým počtem duplicitních řádků a celkovým počtem řádků ze služby Stack Overflow. Tato závislost je lineární (Pearsonův korelační koeficient) a při růstu jedné hodnoty má druhá hodnota tendenci také růst.

Kapitola 6

Závěr

Cílem práce bylo vyhodnotit vliv kopírování kódu na kvalitu softwarového projektu. Toto zadání zní vcelku přímočaře. Nicméně cesta k řešení rozhodně přímočará nebyla. Nejdříve bylo nutné seznámit se s technikami, pomocí kterých je možné zkopírovaný kód odhalit. Studiu těchto technik byla věnována poměrně dlouhá doba.

Ta se však zúročila při návrhu řešení a zejména při výběru vhodného nástroje pro detekci shodných úryvků kódu. Pro tento účel vybrán nástroj NiCad, a to se ukázalo být správnou volbou. Při úpravách nástroje bylo nutné pracovat s jazykem TXL. Osvojení tohoto jazyka sice zabralo nějakou dobu, ale rozhodně stálo za to.

Největším přínosem této práce je právě úprava nástroje NiCad, díky které lze korektně zpracovat i zdrojový kód jazyka Python 3. Ačkoliv byla implementována pravidla, která transformují zdrojový kód do jednotné podoby, tak se v praxi ukázalo, že většina úryvků by byla detekována i bez použití těchto pravidel.

K vyhodnocení ovlivňování kvality byla použita sada 161 projektů. V každém projektu byly vyhledány úryvky ze služby Stack Overflow a duplicitní úryvky. Bylo prokázáno, že při růstu počtu úryvků ze Stack Overflow uvnitř projektu, má počet *issues* také tendenci růst. O něco silnější lineární vztah byl prokázán mezi počtem řádků ze Stack Overflow a počtem duplicitních řádků.

Jelikož pro vyhodnocení byl použit vzorek 161 projektů, tak otevřenou otázkou zůstává, jaký výsledek by přineslo vyhodnocení této studie ve větším měřítku. Zajímavé by rovněž bylo prozkoumat jiný jazyk a zjistit, jestli se jedná o trend nebo pouze o záležitost jazyka Python.

Literatura

[1] *MOSS: A System for Detecting Software Similarity* [online]. [cit. 20. 1. 2019]. Dostupné na: `https://theory.stanford.edu/%7Eaiken/moss/`.

[2] Baxter, I. D., Yahin, A., Moura, L. et al. Clone detection using abstract syntax trees. In *Proceedings. International Conference on Software Maintenance*. Bethesda, MD, USA: [b.n.], listopad 1998. S. 368–377. ISSN 1063-6773.

[3] Cordy, J. R. The TXL source transformation language. *Science of Computer Programming*. 2006, roč. 61, č. 3. S. 190–210. ISSN 0167-6423.

[4] Cordy, J. R., Carmichael, I. H. a Halliday, R. *The TXL programming language*. Kingston, Ontario K7L 3N6, Canada: Queen's University at Kingston, červenec 2012.

[5] Cordy, J. R. a Roy, C. K. The NiCad Clone Detector. In *2011 IEEE 19th International Conference on Program Comprehension*. červen 2011. S. 219–220. ISSN 1092-8138.

[6] Cordy, J. R. a Roy, C. K. Tuning research tools for scalability and performance: The NiCad experience. *Science of Computer Programming*. 2014, roč. 79, C. S. 158–171. ISSN 0167-6423.

[7] Ducasse, S., Nierstrasz, O. a Rieger, M. On the effectiveness of clone detection by string matching. *Journal of Software Maintenance and Evolution: Research and Practice*. 2006, roč. 18, č. 1. S. 37–58. ISSN 1532-060X.

[8] Ducasse, S., Rieger, M. a Demeyer, S. A language independent approach for detecting duplicated code. In *Proceedings IEEE International Conference on Software Maintenance - 1999 (ICSM'99)*. Oxford, UK: [b.n.], 1999. S. 109–118. ISSN 1063-6773.

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

[10] Kamiya, T., Kusumoto, S. a Inoue, K. CCFinder: a multilinguistic token-based code clone detection system for large scale source code. *IEEE Transactions on Software Engineering*. 2002, roč. 28, č. 7. S. 654–670. ISSN 0098-5589.

[11] Levenshtein, V. I. Binary Codes Capable of Correcting Deletions, Insertions and Reversals. *Soviet Physics Doklady*. February 1966, roč. 10. S. 707–710.

[12] Mayrand, Leblanc a Merlo. Experiment on the automatic detection of function clones in a software system using metrics. In *1996 Proceedings of International*

Conference on Software Maintenance. Monterey, CA, USA: [b.n.], listopad 1996. S. 244–253. ISSN 1063-6773.

[13] McCabe, T. J. A Complexity Measure. *IEEE Transactions on Software Engineering*. 1976, SE-2, č. 4. S. 308–320. ISSN 0098-5589.

[14] Roy, C. K. a Cordy, J. R. A Survey on Software Clone Detection Research. *School of Computing TR 2007-541*. Leden 2007.

[15] Schleimer, S., S. Wilkerson, D. a Aiken, A. Winnowing: Local Algorithms for Document Fingerprinting. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. San Diego, CA, USA: [b.n.], červen 2003. S. 76–85.

[16] Wikipedia. *Longest common subsequence problem* [online]. Poslední modifikace: 8. 4. 2019. [cit. 1. 5. 2019]. Dostupné na: `https://en.wikipedia.org/wiki/Longest_common_subsequence_problem`.

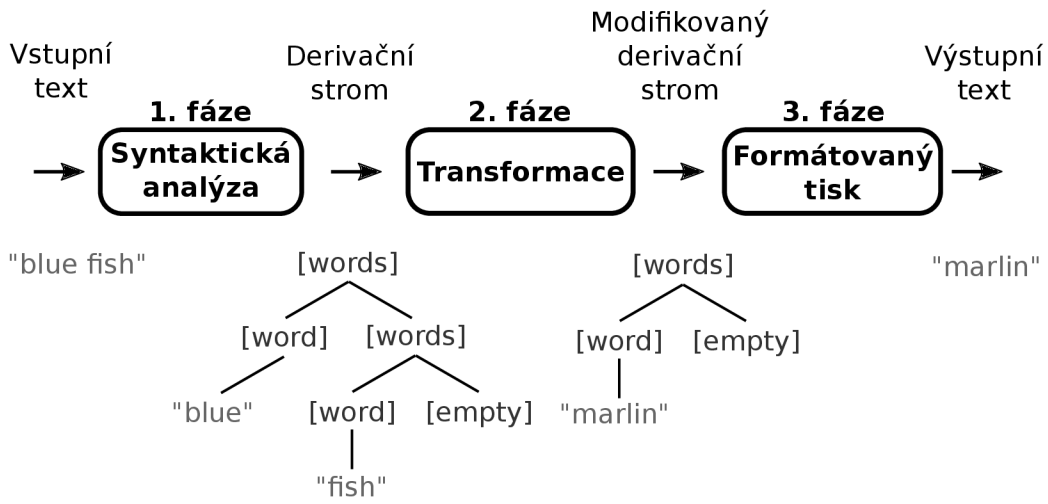
[17] Wikipedia. *Statistical hypothesis testing* [online]. Poslední modifikace: 11. 5. 2019. [cit. 16. 5. 2019]. Dostupné na: `https://en.wikipedia.org/wiki/Statistical_hypothesis_testing`.

[18] Wikipedia. *Statistical significance* [online]. Poslední modifikace: 11. 5. 2019. [cit. 16. 5. 2019]. Dostupné na: `https://en.wikipedia.org/wiki/Statistical_significance`.

Příloha A

Jazyk TXL

Pomocí TXL lze transformovat libovolný vstupní text na formátovaný výstupní text. TXL program se skládá ze dvou části. První částí je bezkontextová gramatika specifikující syntaxi vstupního jazyka. Druhou část tvoří transformační pravidla, která jsou na vstupní text aplikována. Běh každého programu prochází třemi fázemi (viz obrázek A.1).



Obrázek A.1: Tři fáze zpracování vstupního textu pomocí jazyka TXL (převzato z [4]).

Syntaktická analýza – syntaktický analyzátor se snaží zpracovat celý vstup na základě dané bezkontextové gramatiky a vytvořit patřičný derivační strom.

Transformace – zde přicházejí na řadu transformační pravidla a funkce, jež modifikují derivační strom vytvořený v předchozím kroku.

Formátovaný tisk – v tomto kroku se jednotlivé listové uzly modifikovaného derivačního stromu převedou zpět do textové podoby.

A.1 Gramatika

Gramatika obsahuje definice neterminálních symbolů. Notace připomíná rozšířenou Backus-Naurovu formu (EBNF). Gramatika může obsahovat levě či pravě rekurzivní definice a

rovněž může být nejednoznačná. V případě, že je možné daný neterminál interpretovat více způsoby, zkouší se dané derivace v pořadí, v jakém byly uvedeny ve zdrojovém kódu. To má za následek, že pro daný vstup vždy vznikne právě jeden derivační strom.

Základním prvkem gramatiky je příkaz `define`. Ten obsahuje alternativy oddělené svislicí (znak „|“). Každá alternativa se skládá z posloupnosti terminálních a neterminálních symbolů. Ukázka definice neterminálu `if_statement` platná pro jazyk Python je na obrázku A.2. Odkazy na jiné neterminály uvnitř příkazu `define` se uvádějí pomocí jména daného neterminálu uzavřeného v hranatých závorkách (např. `[test]`). Terminální symboly je možné uvést holé. Pouze ty symboly, které jsou zároveň prvky jazyka TXL (např. hranaté závorky) je nutné prefixovat apostrofem (`'`). Dobrou praxí je takto prefixovat všechny terminální symboly. Neterminál uvnitř příkazu `define` může být doplněn modifikátorem. Nejdůležitějšími modifikátory jsou `opt`, `repeat` a `list`.

- `opt` značí, že je daný neterminál volitelný
- `repeat` značí 0 a více opakování daného neterminálu (zápisem `[repeat X+]` lze vynutit 1 a více opakování neterminálu `X`)
- `list` značí 0 a více opakování daného neterminálu oddělených čárkami (zápisem `[list X+]` lze vynutit 1 a více opakování neterminálu `X` oddělených čárkami)

```
define if_statement
  'if [test] ': [suite]
  [repeat elif_clause]
  [opt else_clause]
end define
```

```
define else_clause
  'else ': [suite]
end define
```

```
define elif_clause
  'elif [test] ': [suite]
end define
```

Obrázek A.2: Definice neterminálu `[if_statement]` (pro stručnost neuvedeny definice `[test]` a `[suite]`).

Vstupním bodem gramatiky je neterminál `[program]`, celý vstupní text musí být odvoditelný z tohoto neterminálu. V opačném případě TXL program skončí s chybovým kódem a hláškou indikující, kde nastal prohřešek proti definované syntaxi.

Abychom dokázali korektně zpracovat vstupní text, je mimo bezkontextové gramatiky zapotřebí specifikovat lexikální strukturu jazyka. K tomuto účelu slouží následující konstrukce:

- `comments` – konvence zápisu řádkových a blokových komentářů
- `tokens` – výčet regulárních výrazů definujících třídy tokenů (sekvence vstupních znaků může odpovídat více třídám tokenů, zvolena bude v takovém případě ta, která byla ve zdrojovém kódu uvedena dříve)
- `compounds` – výčet sekvencí znaků, které mají považovány za lexikální jednotku (např. operátory `==`, `>=`, apod.)
- `keys` – výčet klíčových slov jazyka

A.2 Transformace

Po korektním zpracování vstupního textu dle dané gramatiky a vybudování derivačního stromu, přichází na řadu transformační pravidla, která strom modifikují do požadované podoby.

Každé pravidlo/funkce definuje šablonu, jež se snažíme nalézt při průchodu derivačním stromem, a náhradu, která se použije při substituci dané šablony. Šablonu i náhradu tvoří posloupnost terminálních symbolů a proměnných. První výskyt proměnné musí být explicitně typovaný. K tomu slouží zápis `<jméno_proměnné> [<typ>]`, kde typ je jeden z neterminálů. Zápis `Expr [expression]` značí proměnou `Expr` typu `expression`. Při aplikaci se na proměnnou naváže podstrom daného typu. S tímto podstromem lze dále pracovat pomocí odkazu na proměnnou. Speciálním případem je anonymní proměnná `_` (podtržítko), na kterou se nelze odkazovat. Syntaxe zápisu pravidla/funkce včetně ukázky konkrétního pravidla je znázorněna na obrázku A.3.

```
rule <název_pravidla>
  replace [<neterminální_symbol>]
    <předloha>
  by
    <náhrada>
end rule
```

```
rule simplifyAssignments
  replace [statement]
    V [reference] ':= V '+ E [term]
  by
    V '+= E
end rule
```

Obrázek A.3: Syntaxe zápisu pravidla (vlevo) a konkrétní ukázka (vpravo, převzato z [3]).

Pravidla a funkce lze považovat za silně typovaná. Jak šablona, tak i náhrada musí odpovídat typu neterminálu, jenž je uveden za klíčovým slovem `replace`. Je tedy například možné transformovat podmíněný příkaz tak, že odstraníme větev `else` (za předpokladu, že je volitelná), ale není možné nahradit podmíněný příkaz příkazem přiřazení.

Rozdíl mezi funkcí a pravidlem (mimo klíčové slovo) je v rozsahu platnosti. Funkce se pokusí najít podstrom odpovídající šabloně a v případě, že uspěje, provede substituci. Tím její práce končí. Naproti tomu pravidlo prohledává podstrom a provádí změny tak dlouho, dokud se daří nalézt shodu s předlohou. Pokud tedy pravidlo produkuje náhradu, která odpovídá předloze, dojde k nekonečnému cyklu. Tento problém u funkce nehrozí (s výjimkou rekurzivních funkcí).

Aplikace pravidla/funkce se provádí v postfixové notaci. Například aplikaci pravidla G na výsledek aplikace pravidla F nad podstromem X v jazyce TXL zapíšeme jako `X[F][G]`, čemuž v běžných programovacích jazycích odpovídá zápis: `G(F(X))`.

Obecná definice pravidla může být rozšířena o řadu dalších konstrukcí. Mezi ně patří parametry aplikace pravidla, podmínky pro vykonání pravidla (mimo nutné shody šablony a daného podstromu), konstrukce `construct` pro vytvoření nového podstromu/proměnné a `deconstruct` pro rozdělení stromu navázaného na proměnnou na menší části.

Vstupním bodem pro transformaci je funkce/pravidlo s názvem `main`.

A.3 Formátovaný tisk

Ve výchozím nastavení jsou při zpětném převodu do textové podoby jednotlivé listové uzly reprezentující tokeny oddělovány jednou mezerou. Pro vytvoření formátovaného výstupu slouží tzv. formátovací neterminály. Tyto neterminály se umísťují na patřičná místa uvnitř

definic neterminálů vstupního jazyka. Tabulka A.1 obsahuje vybrané formátovací neterminály a jejich význam. Obrázek A.4 demonstruje jejich praktické použití pro korektní výstup XML elementu (bez mezer mezi názvem značky a ohraničujícími symboly).

Formátovací neterminál	Význam
`[NL]`	vynutit zalomení stávajícího řádku
`[IN]`	odsadit všechny následující řádky o 4 mezery
`[EX]`	zmenšit odsazení následujících řádků o 4 mezery
`[SPOFF]`/`[SPON]`	vypnout/zapnout výchozí výstup (s mezerou mezi tokeny)

Tabulka A.1: Vybrané formátovací neterminály a jejich význam.

```
define XML_tag
  '< [SPOFF] [id] '> [SPON] [repeat content] '< [SPOFF] '/ [id] '> [SPON]
end define
```

Obrázek A.4: Praktické použití formátovacích neterminálů (převzato z [3]).

Příloha B

Obsah přiloženého paměťového média

- `README` – soubor se základními informacemi
- `xchova19-kopirovani-kodu.pdf` – text práce ve formátu PDF
- `requirements.txt` – deklarace požadovaných knihoven pro běh Python skriptů
- `extract_snippets.py` – skript pro extrakci bloků kódu z příspěvků služby Stack Overflow
- `harvest.py` – skript pro sběr jednotlivých dílčích výsledků
- `evaluate.py` – skript pro vyhodnocení hodnot korelačních koeficientů
- `data.json` – výstup skriptu `harvest.py` se získanými výsledky všech 161 projektů, které byly analyzovány
- `Nicad-5.1/` – adresář s nástrojem NiCad, přičemž v rámci této práce byly vytvořeny soubory: *txl/py3-**, *txl/python3.grm* a *config/py3-**.