VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA ELEKTROTECHNIKY A KOMUNIKAČNÍCH TECHNOLOGIÍ

FACULTY OF ELECTRICAL ENGINEERING AND COMMUNICATION

ÚSTAV TELEKOMUNIKACÍ

DEPARTMENT OF TELECOMMUNICATIONS

DETEKCE PLAGIÁTORSTVÍ V SOFTWAROVÝCH PROJEKTECH V PŘEDMĚTU BEZPEČNOST DATABÁZOVÝCH SYSTÉMŮ

DETECTION OF PLAGIATORISM IN SOFTWARE PROJECTS IN THE BDS COURSE

BAKALÁŘSKÁ PRÁCE

BACHELOR'S THESIS

AUTOR PRÁCE — Marek Szymutko

AUTHOR

VEDOUCÍ PRÁCE — Ing. Mgr. Pavel Šeda, Ph.D.

SUPERVISOR

BRNO 2023

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
FAKULTA ELEKTROTECHNIKY A KOMUNIKAČNÍCH TECHNOLOGIÍ

Bakalářská práce

bakalářský studijní program **Informační bezpečnost**

Ústav telekomunikací

Student: Marek Szymutko ***ID:*** 231285

Ročník: 3 ***Akademický rok:*** 2022/23

NÁZEV TÉMATU:

Detekce plagiátorství v softwarových projektech v předmětu Bezpečnost databázových systémů

POKYNY PRO VYPRACOVÁNÍ:

Bakalářská práce se bude zaměřovat na prevenci podvádění studentů při vypracovávání softwarových projektů. V rámci této práce bude student analyzovat možnosti a způsoby podvádění a možnou prevenci tohoto chování. Bude zmapován celý proces odevzdávání projektů na veřejně hostovaný vývojářský cloud založený na verzovacím systému Git (GitHub, GitLab) s ohledem na technické detekční možnosti. V rámci praktické části bude provedena implementace detekčních algoritmů v programovacím jazyce Python. Výsledná aplikace bude bude využita v případové studii kontrolující semestrální projekty v předmětu Bezpečnost databázových systémů (BPC-BDS). Výsledky budou vhodně analyzovány a popsány.

DOPORUČENÁ LITERATURA:

[1] Vykopal, J., Švábenský, V., Seda, P. and Čeleda, P., 2022, February. Preventing Cheating in Hands-on Lab Assignments. In Proceedings of the 53rd ACM Technical Symposium on Computer Science Education V. 1 (pp. 78-84).

[2] Riabov, V.V., 2016. Teaching online computer-science courses in LMS and cloud environment. International Journal of Quality Assurance in Engineering and Technology Education (IJQAETE), 5(4), pp.12-41.

Termín zadání: 6.2.2023 ***Termín odevzdání:*** 26.5.2023

Vedoucí práce: Ing. Mgr. Pavel Šeda, Ph.D.

doc. Ing. Jan Hajný, Ph.D.
předseda rady studijního programu

UPOZORNĚNÍ:

Autor bakalářské práce nesmí při vytváření bakalářské práce porušit autorská práva třetích osob, zejména nesmí zasahovat nedovoleným způsobem do cizích autorských práv osobnostních a musí si být plně vědom následků porušení ustanovení § 11 a následujících autorského zákona č. 121/2000 Sb., včetně možných trestněprávních důsledků vyplývajících z ustanovení části druhé, hlavy VI. díl 4 Trestního zákoníku č.40/2009 Sb.

ABSTRAKT

Plagiátorství je rozšířený problém, kterému lze předcházet preventivními nebo detekčními metodami. V práci jsou shrnuty možnosti detekce plagiátorství pomocí automatizovaných metod. Pro získávání dat z projektů byl využit volně šiřitelný kompilátor abstraktních syntaktických stromů, jehož fungování bylo v práci demonstrováno. Byl navržen způsob odevzdávání softwarových projektů tak, aby studenti nemohli vzájemně vidět své projekty. Využita k tomu byla cloudová služba GitLab. Pomocí skriptů jazyka Bash byl automatizován způsob zakládání samostatných studentských prostředí v této službě. Také byly vytvořeny skripty pro zakládání skupin studentů a archivace studentských repozitářů ve službě GitLab. V jazyce Python byl vyvinut systém k hledání podobností ve studentských projektech z předmětu BPC-BDS, psaných v jazyce Java nebo Python. Využít jej lze však i v jiných předmětech. Tento systém využívá metrik a abstraktních syntaktických stromů. Výsledek porovnání projektů i jejich jednotlivých částí je reprezentován celočíselnou hodnotou a zapsán pro přehlednost do tabulkového souboru formátu `xlsx`. Práce se věnuje i silným a slabým stránkám implementovaného způsobu hledání plagiátů. Také jsou shrnuty problémy, které vyvstaly v průběhu řešení práce. Součástí je i případová studie ohledně plagiátorství v předmětu BDS v akademickém roce 2022/2023.

KLÍČOVÁ SLOVA

Abstraktní syntaktické stromy, API, Bash, detekce, Git, GitLab, Java, metriky, plagiátorství, Python

ABSTRACT

Plagiarism is a widespread problem, which can be fought by prevention or detection methods. This thesis contains a summary of plagiarism detection methods through automated means. To parse source code, an open-source abstract syntax tree compiler was employed. The functionality of this compiler was demonstrated in this thesis. To reduce the mutual visibility of students' projects, a proposal for the submission process was created. GitLab cloud service was employed for this purpose. Initialization of these students' spaces is performed via Bash scripts. Other scripts to archive and create spaces for groups of students in the GitLab service were also created. A similarity-detecting tool was created in Python programming language. This tool was specialized to be employed in the subject BPC-BDS for the detection of plagiarism in students' assignments written in Java or Python. It can also be used in other subjects though. For similarity detection, numerical metrics and abstract syntax trees were used. The comparison output of the projects and their individual parts is represented with integer value and outputted into a tabular file of the format `xlsx`. This thesis also summarizes the strengths and weaknesses of the implemented solution and lists problems that were encountered in the process of implementation. A case study about plagiarism in the subject BDS in the academic year 2022/2023 is also included in this thesis.

KEYWORDS

Abstract syntax trees, API, Bash, detection, Git, GitLab, Java, metrics, plagiarism, Python

Vysázeno pomocí balíčku `thesis` verze 4.07; `http://latex.feec.vutbr.cz`

SZYMUTKO, Marek. *Detekce plagiátorství v softwarových projektech v předmětu Bezpečnost databázových systémů*. Brno: Vysoké učení technické v Brně, Fakulta elektrotechniky a komunikačních technologií, Ústav telekomunikací, 2022, 58 s. Bakalářská práce. Vedoucí práce: Ing. Mgr. Pavel Šeda, Ph.D.

Prohlášení autora o původnosti díla

Jméno a příjmení autora:	Marek Szymutko
VUT ID autora:	231285
Typ práce:	Bakalářská práce
Akademický rok:	2022/23
Téma závěrečné práce:	Detekce plagiátorství v softwarových projektech v předmětu Bezpečnost databázových systémů

Prohlašuji, že svou závěrečnou práci jsem vypracoval samostatně pod vedením vedoucí/ho závěrečné práce a s použitím odborné literatury a dalších informačních zdrojů, které jsou všechny citovány v práci a uvedeny v seznamu literatury na konci práce.

Jako autor uvedené závěrečné práce dále prohlašuji, že v souvislosti s vytvořením této závěrečné práce jsem neporušil autorská práva třetích osob, zejména jsem nezasáhl nedovoleným způsobem do cizích autorských práv osobnostních a/nebo majetkových a jsem si plně vědom následků porušení ustanovení § 11 a následujících autorského zákona č. 121/2000 Sb., o právu autorském, o právech souvisejících s právem autorským a o změně některých zákonů (autorský zákon), ve znění pozdějších předpisů, včetně možných trestněprávních důsledků vyplývajících z ustanovení části druhé, hlavy VI. díl 4 Trestního zákoníku č. 40/2009 Sb.

Brno ....

podpis autora*

*Autor podepisuje pouze v tištěné verzi.

PODĚKOVÁNÍ

Rád bych poděkoval vedoucímu této bakalářské práce, panu Ing. Mgr. Pavlu Šedovi, Ph.D. za odborné vedení, konzultace, rychlé odpovědi na dotazy, trpělivost a podnětné návrhy k práci.

Obsah

Úvod

Tato práce se věnuje detekci plagiátorství v softwarových projektech studentů kurzu BPC-BDS (bezpečnost databázových systémů). Součástí tohoto předmětu jsou tři semestrální projekty, přičemž výstupem posledního z nich je aplikace psaná v jazyce Java nebo Python. Tento projekt je poměrně obsáhlý, proto by byla důkladná kontrola člověkem časově velmi náročná. Pro automatizaci kontroly plagiátorství v BPC-BDS byl v rámci této práce vyvinut detekční systém. Řešení však lze nasadit i v jiných předmětech, kde jsou odevzdávány projekty v jazycích Java nebo Python.

Také je nutné navrhnout efektivní způsob odevzdávání studentských prací. Pro tento účel byly vytvořeny skripty v jazyce Bash, které připravují prostředí pro odevzdávání projektů. Do tohoto prostředí studenti odevzdávají své projekty prostřednictvím verzovacího systému Git. Ten je totiž využíván v rámci celého světa a napříč programovacími jazyky pro vývoj otevřeného i komerčního software [1]. Seznámení studentů s tímto systémem je tedy pro studenty vhodné.

K odevzdávání studentských projektů byla zvolena služba GitLab [2], jejíž prostředí lze konfigurovat přes webovou API (rozhraní pro programování aplikací, *Application Programming Interface*). Díky tomu je možné vytvořit pro každého studenta samostatně oddělené prostředí k odevzdávání.

Dále byl v programovacím jazyce Python implementován vlastní způsob detekce plagiátorství založený na vědomostech získaných v rámci teoretické části. Program využívá převodu zdrojového kódu na abstraktní syntaktický strom a s jeho pomocí je porovnávána struktura projektů za použití číselného vyjádření podobnosti částí. Tento způsob kontroly plagiátorství je plně deterministický a výsledky porovnání se nemění při opětovném spuštění programu.

První kapitola práce je věnována plagiátorství, častým způsobům kopírování a také metodám detekce zkopírovaných či velmi podobných zdrojových kódů. Druhá kapitola je věnována abstraktním syntaktickým stromům včetně ukázky struktury takového stromu.

Praktické části se pak věnují zbývající kapitoly. Způsob odevzdávání studentských projektů, včetně popisu skriptů k tomu potřebných, je popsán ve třetí kapitole. Ve čtvrté kapitole je pak rozebrán způsob implementace detekčního algoritmu včetně dosažených úspěchů a limitací tohoto řešení. Tato kapitola končí ponaučením z implementace detekčního algoritmu.

Pátá kapitola se pak věnuje případové studii studentských projektů v předmětu BPC-BDS v akademickém roce 2022/2023. Také je zde popsán průběh běhu programu a jeho porovnání s aplikací JPlag [3, 4].

1 Plagiátorství

Plagiátorství je způsob podvádění, při kterém je cizí práce neprávem přivlastňována jinou osobou. Plagiátorství se týká jak psaného textu, tak i softwarových projektů. Proti podvádění bylo již v minulosti vyzkoušeno a nasazeno mnoho postupů, s různou efektivitou. Jako příklad je možné uvést rozsazení studentů nebo dočasné zabavení chytrých telefonů a hodinek při písemných zkouškách. Dalším příkladem je také detekce kopírovaného textu v závěrečných pracích. S těmito a dalšími postupy je možno se setkat denně na jakékoli škole.

Redukce plagiátorství je realizována dvěma základními způsoby [5]:

1. Předcházení plagiátorství
2. Detekce plagiátorství

1.1 Předcházení plagiátorství

V rámci preventivních metod je vhodné zavést zásady, které budou přínosné pro studenty, kteří se kopírování vyvarují a naopak potrestat osoby z plagiátorství usvědčené. Tato prevence se netýká pouze jednoho předmětu nebo instituce, ale celé společnosti. Je vyžadováno, ať není podvádění vnímáno jako přirozené, ale jako jev, kterému by se měl každý člověk chtít vyhnout [5].

Jako metody prevence proti podvádění lze také považovat ztěžování či zamezení kopírování. Může se jednat například o vytváření personalizované sady úloh pro každého studenta, při kterých je nepravděpodobné, že budou dva stejní studenti řešit stejnou sadu úloh [6]. Také je možné vytvářet unikátní zadání, jejichž zadání na internetu nelze dohledat, ačkoli je tato metoda velice náročně aplikovatelná pro velký počet studentů [7].

1.2 Detekce plagiátorství

Právě těmito metodami se semestrální práce bude zabývat v praktické části. Detekční metody nejsou samy schopny eliminovat podvádění, pouze ho minimalizovat. Bez dlouhodobého nasazení preventivních metod nelze kopírování úplně vymýtit [5]. Detekce plagiátorství je většinou automatizovaný proces, který je schopný zpracovat poměrně velké množství informací a na základě podobností stanovit, které práce jsou potenciálními plagiáty. Je však nutné, aby byl výsledek automatizovaného porovnávání ověřen člověkem. Takto s největší pravděpodobností nebude autor potrestán na základě falešně pozitivního výsledku, který dokáže lidský faktor vyvrátit. Na tento fakt upozorňují i autoři již zavedených a spolehlivých detekčních algoritmů, jako je například systém MOSS vytvořený na Stanfordově univerzitě [8].

U některých typů zadání je možné sledovat znaky plagiátorství i jinými indikátory, než jen odevzdanou prací. Pokud k takovým informacím je přístup, je možno do detekce zahrnout i čas a místo odeslání práce. Pokud je například práce odevzdána příliš rychle na to, aby ji dokázal člověk v tomto čase vyřešit, je velmi pravděpodobné, že byly výsledky práce zkopírovány [6].

1.3 Způsoby podvádění při softwarových projektech

Studenti se mohou pokoušet kopírovat celé projekty, nebo jen některé funkcionality. Své plagiátorství pak mohou určitými způsoby maskovat tak, aby nebylo toto kopírování na první pohled zřejmé. Již dávno byly vytipovány časté strategie maskování podvádění při softwarových projektech, zde jsou seřazeny od nejméně sofistikovaných po nejvíce sofistikované [9]:

1. Změna komentářů nebo formátování
2. Změna identifikátorů (názvy proměnných, metod, tříd...)
3. Změna pořadí operandů ve výrazech
4. Změna datových typů (například záměna `int` za `double`)
5. Záměna výrazů za jejich ekvivalenty (záměna `while found == false do...` za `while not found do...`)
6. Přidávání přebytečných proměnných nebo výrazů
7. Záměna pořadí nezávislých částí kódu
8. Změna struktury iteračních výrazů (záměna `while` a `for`)
9. Změna struktury výrazů `if`, `else` a `switch`
10. Záměna volání metody za tělo této metody
11. Použití nestrukturních výrazů typu `goto` (ale ty nejsou v `Javě` využívány)
12. Kombinování vlastních a zkopírovaných částí kódu

Čím více z těchto metod podvádění se podaří detekčním algoritmem zachytit, tím více pokusů o plagiátorství bude odhaleno. Na druhou stranu může vést k falešně pozitivním výsledkům v případě, že je pro větší počet projektů stejné zadání (což je i případ předmětu BPC-BDS).

1.4 Proces detekce plagiátorství

Existuje více způsobů, jak automatizovanou detekci podvádění při tvorbě softwarových projektů provádět. Většina algoritmů k detekci ale používá následující kroky [10]:

1. Rozbor a načtení znaků k porovnání
2. Výpočet podobnosti párů znaků a uložení mapy podobností
3. Výběr souborů s velkým výskytem podobností

Rozbor a načtení znaků je otázka výběru sledovaných znaků, které mají být mezi projekty porovnávány.

Výpočet podobnosti úzce souvisí s povahou vybraného znaku. Pro každý typ znaku existuje mnoho metod, každá různě náročná na implementaci i hardwarové prostředky a různě spolehlivá (lze například hledat bijekce abstraktních syntaktických stromů nebo pouze zkoumat číselné metriky). Tomuto tématu se věnuje část 1.5.

Výběr souborů s velkým výskytem podobností pouze označí práce k lidské kontrole. Existuje jistá pravděpodobnost falešně pozitivních výsledků, která závisí na použitém algoritmu, kterou by měl lidský faktor být schopen eliminovat.

1.5 Používané způsoby detekce v softwarových projektech

Tato sekce se bude zabývat způsobem výběru znaků k porovnání podobností. Některé způsoby jsou poměrně jednoduché, jiné zase poměrně sofistikované a náročnější na implementaci. Nejvíce standardní jsou následující postupy [11]:

1.5.1 Použití metrik

Metriky jsou kvantitativním vyjádřením nějaké vlastnosti zkoumaného dokumentu (např. počet atributů třídy). Metoda zkoumání podobností s využitím metrik je starší a zaměřuje se pouze na měřitelné aspekty kódu, které nevyžadují jeho kompletní porozumění. Spočívá ve stanovení sledovaných vlastností a poté převedení souboru na pouhou sérii čísel, jež reprezentovala měřené metriky. Porovnávány jsou pak pouze tyto série čísel. Takový přístup je využitelný jak pro literární díla, tak i pro detekci kopírování v softwarových projektech [12].

Metriky lze dělit podle několika kritérií. Lze je vypočítávat na každém dokumentu zvlášť a poté porovnávat (individuální metriky, v originále *singular metrics*), nebo je lze vypočítávat na více souborech [12]. Pokud se jedná o metriky charakterizující porovnávaný pár souborů, nazývají se párové metriky (v originále *paired metrics*) [12]. Mezi individuální metriky patří například počet symbolů na řádek nebo poměr `while` a `for` cyklů, příkladem párových metrik pak je pak délka nejdelšího společného řetězce po tokenizaci [12]. Rozlišujeme také souborové (v originále *corpal*) metriky (týkající se celého souboru prací) a vícerozměrné (v orig. *multidimensional*) metriky (porovnávají určené množství dokumentů) [12]. Vícerozměrným znakem může být například počet klíčových slov, který je stejný pro určitou

skupinu prací. Pro souborové metriky je vhodným příkladem počet odevzdaných prací obsahující určité klíčové slovo [12].

Metriky lze také rozlišovat podle složitosti detekční funkce. Rozlišují se metriky povrchní (v orig. *superficial*) a strukturní (v orig. *structural*) [12]. Povrchní mohou být měřeny pouhým zkoumáním jednoho nebo více dokumentů bez porozumění použitého textu(uveďme například: počet výskytů klíčového slova `while`). Strukturní pak vyžadují znalost struktury dokumentů, na nichž je detekce prováděna. Může to být například počet možných průchodů programem při různých hodnotách vstupů (jiná cesta zvolená blokem kódu `if` apod.) [12].

Byly definovány i další typy metrik, jako například sémantické nebo statistické metody, ale tyto se používají pro detekci plagiátorství v textech a pro tuto práci nejsou příliš relevantní [5].

1.5.2 Použití tokenizace

Tento způsob využívá transformování zdrojového kódu na logické části, které jsou zbaveny redundantních informací (komentáře, prázdná místa a dokonce i některá klíčová slova). Identifikátory v tokenech částech jsou nahrazeny speciálními řetězci, takže zbývá pouze řetězec obecných příkazů a těchto zástupných identifikátorů [13].

Tyto tokeny jsou poté transformovány podle určitých pravidel tak, aby byly snadněji porovnatelné. Je také vhodné zachovat informaci o tom, odkud nový token pochází, ať je možné detekované podobnosti vhodně reprezentovat [13]. V těchto řetězcích tokenů je pak hledána nejdelší nalezená shoda [4, 8].

V jazyce Java je pak navíc možné zkoumat podobnosti bajtového kódu, který je předkompilován pro interpretaci pomocí JVM [14].

Příklad detekčního algoritmu založeného na použití tokenizace zdrojového kódu, je JPlag [3, 4]. S tímto systémem je implementovaná aplikace porovnávána v části 5.5.

1.5.3 Porovnávání struktury

Zde se opět využívá tokenizace zdrojového kódu, aby detekce nebyla ovlivněna jednoduše změnitelnými identifikátory proměnných, metod, tříd apod. při zachování stejné funkcionality. Poté je porovnávána struktura jednotlivých projektů. Ty tedy nejsou posuzované jen jako řetězce čísel či tokenů, ale jejich reprezentace může mít podobu buď stromu (abstraktní syntaktický strom, viz část 2), nebo grafu (graf průchodů běhu programu, graf závislostí) [10].

Porovnávání struktur spočívá v hledání isomorfismů mezi strukturami posuzovaných projektů. U grafových struktur však neexistuje žádný algoritmus, který by byl schopen určit v polynomiálním čase, zda je dvojice grafů izomorfní [15]. Naopak u stromových struktur takový algoritmus objeven byl [16]. Pro ještě efektivnější

porovnávání struktury zdrojového kódu jsou navíc nadále vyvíjeny nové algoritmy a metody jako například porovnávání grafů závislostí zdrojových kódů [10].

1.5.4 Strojové učení

Pro práci s velkým množstvím dat je strojové učení často skvělou volbou. I v detekci plagiátorství má strojové učení své využití. Lze využívat jak algoritmy s učitelem, tak i algoritmy bez učitele. Strojové učení lze použít pro detekci plagiátorství v softwarových projektech [17], ale i v psaných textech [18]. Pro správné natrénování umělé inteligence je však potřeba poměrně velká sada vstupních zdrojových kódů. Další možností je využití již natrénovaných modelů strojového učení, jako například CodeBERTa [19], nebo je možné využit optimalizační kompilátor s využitím umělé inteligence jako je MILEPOST GCC [20]. Takový kompilátor je schopný kód transformovat do optimalizované formy a podobnost jde spočítat i jednoduchou euklidovskou vzdáleností výsledných nebo metrikami vypočtenými z optimalizovaných kódů [19].

1.6 Metody detekce využité v praktické části

V praktické části bude implementován algoritmus pro detekci plagiátorství užívající porovnávání struktury a metrik projektů. Pro celkovou strukturu bude implementováno hledání nejlepšího protikladu každého sledovaného elementu, a to za použití párových metrik. Tyto metriky budou označeny jako *skóre podobnosti*. Toto bude celočíselná hodnota v rozmezí 0–100 (0 značí nenalezení shody, 100 značí kompletní shodu). Pro každý sledovaný element studentských projektů bude vytvořena reprezentace objektem v programovacím jazyce Python a tyto části mezi sebou pak budou porovnávány po dvojicích. Každé skóre podobnosti bude také obsahovat informaci o *váze shody*. Tato váha slouží k vypočítání váženého průměru shod z jednotlivých částí, aby každá shoda nesla informaci, o tom, jak velká část kódu je porovnávána. Každý výsledek porovnání také ponese informace o porovnávaných částech samotných, aby bylo možné reprezentovat, které části jsou do jaké míry shodné. Implementaci tohoto řešení budou věnovány části 2.3 a 4.1.5.

Individuální číselné metriky budou využity pro porovnávání těl metod a procedurálního kódu. V každém logickém výrazu budou zjištěny využité typy uzlů abstraktního syntaktického stromu. Bude vytvořena mapová struktura, která každému užitému typu instrukce (datovému typu uzlu abstraktního syntaktického stromu, viz dále, kapitola 2) přiřadí počet jeho užití v této části kódu. Skóre podobnosti

jednotlivých částí dvojic funkcí pak bude vypočítáno pomocí vzorce

$$S = \frac{\sum_i^m \left(100 - 100 \cdot \left|\frac{n_{i,a} - n_{i,a}}{n_{i,b} + n_{i,b}}\right|\right)}{m},$$

kde S je výsledné skóre, $n_{i,a}$ je počet výskytů uzlu typu i v projektu a, $n_{i,b}$ je počet výskytů uzlů typu i v projektu b a m je celkový počet typů uzlů při tomto porovnání. Tento vzorec mapuje rozdíl počtu výskytů na hodnotu v intervalu 0–100, přičemž nejvyšší skóre podobnosti (hodnota 100) dosahuje při shodných hodnotách $n_{i,a}$ a $n_{i,b}$. Z jednotlivých hodnot pro každý typ uzlu je poté spočítán aritmetický průměr.

Pokud není daný typ uzlu obsažen v obou projektech, je hledán zástupný typ uzlu definovaný ve slovníku podobných uzlů (tento je v kódu aplikace definován manuálně). Pokud takový zástupný typ nalezen je, proběhne porovnání stejně, jako by byl obsažený v obou projektech uzel původní, avšak skóre je mapováno pouze na interval 0–75 pro zvýraznění neúplné shody (hodnoty 100 v původním vzorci jsou nahrazeny hodnotou 75).

2 Abstraktní syntaktické stromy

Abstraktní syntaktické stromy (také AST, Abstract Syntax Trees) jsou struktury, které reprezentují zdrojový kód pomocí stromových struktur. Pro práci se zdrojovým kódem, při které je potřeba brát v zřetel jeho funkcionalitu a ne na úpravu, jsou abstraktní syntaktické stromy mnohem spolehlivější než samotný zdrojový kód. AST nepracují s formální úpravou kódu, jen s jeho logickou stránkou. Proto jsou vhodné pro detekci plagiátorství, u kterého je pravděpodobné, že došlo k jeho maskováním přeformátováním zdrojového souboru nebo ke změnám identifikátorů. Také je práce s AST jednodušší než práce se zdrojovým kódem [21].

2.1 Vytváření AST

Abstraktní syntaktické stromy jsou vytvářeny pomocí specializovaného kompilátoru zdrojového kódu. Tento kompilátor neprodukuje bajtový kód, ale datovou strukturu popisující kompilovaný kód. Pro účel práce bude využit balíček `javalang`, který je volně přístupný pod licencí MIT z portálu PyPI, a také nativní kompilátor abstraktních syntaktických stromů z balíčku `ast`.

Další způsob vytváření objektů syntaktických stromů bez kompilátoru by byl možný s použitím vlastní syntaktické analýzy (anglicky *parsing*) zdrojového kódu. Nejobvyklejší rychlý způsob této analýzy je užití tzv. regulárních výrazů, ale toto řešení by bylo velice komplikované a jen těžko udržovatelné. Regulární výrazy jsou totiž velice špatně čitelné a jejich implementace se jen stěží mění, pokud tyto výrazy zajišťují analýzu skupin textu. Při správě kódu by bylo nutné dávat pozor na správné dodržení číslování těchto skupin, jinak by došlo k úplnému znefunkčnění kódu. Navíc by toto řešení vyžadovalo i další pomocné procedury ke správnému rozboru, jelikož se regulární výrazy nehodí k analýze úrovní zanoření pomocí závorek [22].

2.2 Příklad AST

Pro představu, jak AST opravdu vypadají, je vhodné uvést příklad. Byl vytvořen jednoduchý `Hello world` program v jazyce `java`. Program je uveden ve výpisu 2.1. Pro převedení tohoto souboru na AST byl napsán jednoduchý skript s použitím balíčku `javalang`. Výstup tohoto skriptu je podobný textu z výpisu 2.2. V něm je patrná struktura AST.

Výsledný text popisující objekt AST byl upraven a převeden do čitelnější formy. Surový výstup tohoto skriptu totiž produkuje výpis do jediného řádku a navíc obsahuje mnoho nevyužitých polí a atributů, která byla z tohoto výstupu pro přehlednost odstraněna.

Výpis 2.1: Výchozí `java` soubor

```
public class Run {
    public static void main (String[] args) {
        System.out.println("Hello world!");
    }
}
```

Z tohoto výstupu je však dobře patrný potenciál, který tento objekt má pro porovnávání podobnosti softwarových projektů. Každý soubor má své importované třídy (atribut `imports`), balíček (atribut `package`) a vlastní abstraktní syntaktický strom (atribut `types`). Všechny tyto atributy je vhodné použít k porovnávání abstraktních syntaktických stromů studentských projektů, aby bylo možné zkoumat kontext volaných metod a referencovaných datových typů z projektu.

2.3 Porovnávání AST

Abstraktní syntaktické stromy lze porovnávat jako pouhé stromové struktury (pouze hledáním izomorfismů, viz část 1.5.3)nebo lze zjednodušit navigaci ve stromové struktuře pomocí zachování informace o typu uzlu. To znamená, že pokud víme, že chceme nalézt uzel k porovnání pro metodu, musí být tento protikladný uzel také metodou a nemůže být např. atribut třídy nebo lokální proměnná.

Pomocí sledování importovaných výrazů lze překládat uživatelsky definované datové typy na množiny importovaných či primitivních datových typů. Jako příklad mějme myšlenou třídu `Student` s atributy `String name` a `int id`. Datový typ `Student` můžeme přemapovat na množinu obsahující pouze datové typy atributů (`String`, `int`), což umožní efektivní porovnávání uživatelsky definovaných datových typů bez ohledu na jejich pojmenování.

Tohoto přemapování lze docílit tak, že bude zkoumán původ importované třídy. Pokud se tato třída nachází ve stejném projektu, jedná se o studentem definovaný datový typ a tento bude vhodné přemapovat.

Bohužel podobné porovnávání pomocí datových typů není úplně možné v Pythonu, protože v něm proměnné nenesou informaci o použitém typu. Programátor může specifikovat, jakého typu by proměnná měla být, ale sledování těchto znaků by mohlo být zneužito při maskování plagiátorství. Pokud by totiž student záměrně uvedl jiný datový typ, než bude do proměnné nahrán, algoritmus pro hledání plagiátorství by byl zmaten a studentský kód by byl stále bez chyb spustitelný. Možným řešením by mohlo být sledovat původ hodnoty vložené do proměnné a podle tohoto původu hledat výsledný datový typ, ale takové vyhledávání by bylo velice náročné

jak programátorsky, tak výpočetně. Tato možnost v praktické části implementována není a převážná část detekce plagiátorství v Pythonu stojí na struktuře projektů a typech uzlů AST použitých v kódu.

Kompilátory abstraktních syntaktických stromů generují AST pro každý soubor zdrojového kódu zvlášť, je tedy ještě potřeba definovat obálkový datový typ pro studentský projekt, který bude sdružovat jednotlivé soubory, a pomocí jehož bude možno vyhledávat importované části kódu mezi soubory. Porovnávat strukturu jednotlivých souborů pak lze poměrně jednoduše, sledováním specifických znaků, které budou definovány pro každou sledovanou entitu abstraktního syntaktického stromu, viz část 4.1.5.

Výpis 2.2: Výsledný AST

```
CompilationUnit(
    imports=[],
    package=None,
    types=[ClassDeclaration(
        body=[MethodDeclaration(
            body=[StatementExpression(
                expression=MethodInvocation(
                    arguments[
                        Literal(value="Hello world!")
                    ],
                    member=println,
                    qualifier=System.out
                )
            )],
            modifiers={'public', 'static'},
            name=main,
            parameters=[FormalParameter(
                modifiers=set(),
                name=args,
                type=ReferenceType(name=String),
                varargs=False
                )]
            )],
        modifiers={'public'},
        name=Run
    )]
)
```

3 Praktická část

V této kapitole bude popsán způsob odevzdávání studentských prací v předmětu BPC-BDS. Jedná se o přípravu prostředí k odevzdání a zkontrolování studentských řešení v rámci prevence plagiátorství. Popis implementace detekčního algoritmu, který s tímto vytvořeným prostředím spolupracuje, je obsažen až v následující kapitole 4.

3.1 Verzovací systémy

Verzovací systémy umožňují efektivní způsob spolupráce více programátorů na stejném projektu [1]. Existují centralizované a distribuované systémy. Centralizované označí jednu verzi kódu jako centrální, změny se provádějí lokálně a je potřeba je synchronizovat s centrální verzí. Mezi tyto systémy patří např. Subversion [23].

Distribuované verze je pak potřeba synchronizovat pouze vzájemně při interakcích jednotlivých repozitářů, není proto potřeba pracovat za neustálého připojení k síti. Repozitáře drží lokálně veškeré změny, a proto je možné měnit verze bez přístupu k centrálnímu serveru. Mezi tyto systémy patří např. Mercurial a Git [1, 23].

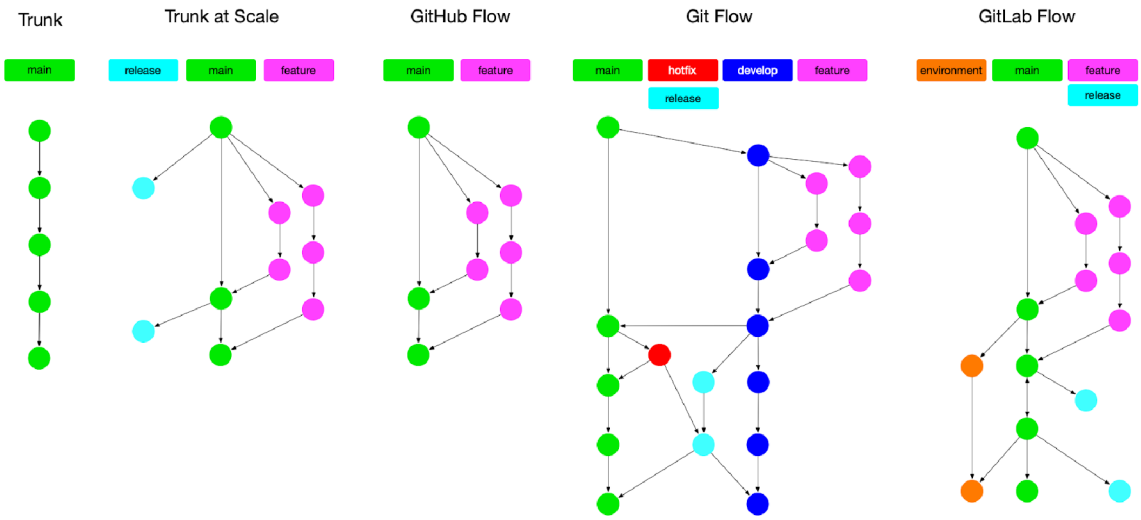
Git je malý a rychlý otevřený verzovací systém kompatibilní napříč platformami. Vyvíjen byl Linusem Torvaldsem (tvůrce Linuxu) a Juniem Hamano, který je také současným správcem projektu. Vývoj začal roku 2005, kdy přestal být BitKeeper volně dostupný. Ten byl do té doby verzovací systém využívaný při vývoji Linuxu [1, 24, 25].

3.1.1 Základní princip Gitu

Git využívá grafovou strukturu pro ukládání verzí kódu. Jednotlivé uzly tohoto grafu jsou nazývány *commity*. Commity jsou jednotlivé změny, které se vývojář rozhodl přidat do kódu a jsou spojeny s časem vytvoření commitu, jeho autorem a zprávou, kterou napsal autor změn [1].

Pokud bychom v Gitu používali pouze commity, dokázali bychom vytvořit pouze lineárně vázanou datovou strukturu. Tento verzovací systém však také podporuje větvení verzí kódu. Větve vždy vycházejí z určitého commitu a mohou se vzájemně spojovat (anglicky *merging*). Výchozí větev je v Gitu označena `main` (ve starších verzích `master`). Mezi větvemi lze jednoduše přepínat. Pokud chce vývojář pracovat na jiné větvi, než na které se nachází, použije příkaz `git checkout <název větve>` a Git vyřeší všechny změny mezi soubory v commitech těchto větví. Příklad struktury větví a commitů je na obrázku 3.1. Na tomto obrázku jsou znázorněny různé

přístupy k užití tohoto systému pro různé účely. Commity jsou v tomto obrázku znázorněny barevnými body, větve jsou barevně odlišeny.



Obr. 3.1: Příklady využití větvení v Gitu, převzato z [26]

Git neukládá v každém commitu celý stav repozitáře, ale jen změny oproti předešlému commitu. Proto je tento verzovací systém velmi rychlý. Změnou větví lze kompletně přepsat soubory v repozitáři, a to beze ztráty informace o původním stavu. Pro navrácení tohoto stavu lze jednoduše přepnout na původní větev. Git také podporuje možnost konfigurace několika zdrojů stejného repozitáře a díky tomu je vhodný pro spolupráci na softwarových projektech.

3.1.2 Vývojářský cloud založený na systému Git

Existují cloudové služby, které využívají Git pro jednodušší správu repozitářů. Příklady takovýchto služeb jsou GitHub, GitLab nebo Azure Repos. Tyto služby poskytují základní sadu nástrojů pro práci s Git repozitáři zdarma. Mezi těmito službami bude v následující části práce vybírán nejvhodnější kandidát pro odevzdávání studentských softwarových projektů.

3.2 Návrh způsobu odevzdávání projektů

Součástí praktické části práce je návrh způsobu odevzdání studentských projektů tak, aby bylo zjednodušena kontrola projektů vyučujícím, a zároveň aby bylo studentům ztíženy podmínky pro podvádění. Je také vhodné, ať se studenti seznamují s možnostmi, které nabízí systém Git, který je hojně využíván pro vývoj softwaru nejen v akademickém prostředí.

K odevzdání studentských projektů bude proto použit vývojářský cloud založený na verzovacím systému Git. Vedoucí práce navrhl pro řešení této práce možné kandidáty cloudového úložiště, u kterých bylo potřeba zjistit, zdali by je bylo možné bezplatně využít pro potřeby této práce.

3.2.1 GitHub

Tato služba má bezplatnou možnost tvorby tzv. organizací. Organizace jsou libovolně velká uskupení uživatelů GitHubu, kteří sdílí prostředí, ve kterém lze vytvářet veřejné nebo soukromé repozitáře. V organizacích lze také vytvářet týmy s různými pravomocemi, které lze nastavit. Také lze s touto platformou komunikovat skrze API za použití příkazu `curl`. Díky těmto vlastnostem by GitHub mohl být zvolen platformou pro odevzdání studentských projektů [27, 28, 29].

3.2.2 GitLab

GitLab dává uživatelům možnost vytvářet skupiny. Ty můžou být buď veřejné, nebo soukromé. Pro bezplatné soukromé skupiny platí maximální limit pěti uživatelů. V každé skupině ale lze vytvářet i podskupiny s vlastním nastavením viditelnosti a s různými pravomocemi pro různé uživatele, takže tento limit lze obejít využitím veřejné skupiny a soukromých podskupin. I tato služba má možnost automatizace dotazů přes API pomocí `curl`. To z ní dělá druhého kandidáta pro řešení této práce [2, 30, 31].

3.2.3 Azure Repos, BitBucket

Tyto služby povolují bezplatnou spolupráci pouze do počtu pěti členů organizace, a proto nejsou vhodné pro odevzdávání studentských projektů [32, 33].

3.3 Vytvoření prostředí pro odevzdání studentských projektů

Po konzultaci s vedoucím práce byla vybrána služba GitLab. Jeho hlavní předností je možnost několikanásobného zanoření skupin, díky čemuž lze jednu hlavní skupinu dále dělit. Tvorbu skupin, podskupin a repozitářů, jakož i modifikace práv pro jednotlivé uživatele, lze obsluhovat autentizovanou aplikací přes poskytnutou API. K autentizaci slouží takzvaný *token*, který si každý uživatel může vygenerovat ve webovém rozhraní GitLabu. V tomto rozhraní lze specifikovat, na co má mít práva aplikace (nebo uživatel) při použití tohoto tokenu.

Pro automatizovanou práci se skupinami v této službě byly vytvořeny skripty v jazyce Bash. Tyto umožňují rychlou inicializaci a správu studentských podskupin s omezenou viditelností, aby studenti vzájemně neviděli odevzdané soubory. Zdrojové kódy skriptů jsou zveřejněny online [34].

3.3.1 Vytvoření samostatných skupin

První skript slouží k inicializaci podskupin určené pro každého studenta samostatně. V těchto skupinách tedy jsou dva uživatelé. Žák a majitel skupiny, tedy učitel. Skript vyžaduje vyplnění GitLab autentizačního tokenu správce skupiny a ID skupiny či podskupiny, ve které je potřeba vytvořit studentské podskupiny. Tyto údaje se uvádějí do souboru pojmenovaného `.env`, jehož nevyplněná verze je uvedena ve výpisu 3.1. V tomto souboru je také proměnná uvádějící název souboru obsahující tzv. *odpadlíky* (studenty, jejichž GitLab jméno nebylo nalezeno).

Výpis 3.1: Soubor `.env`

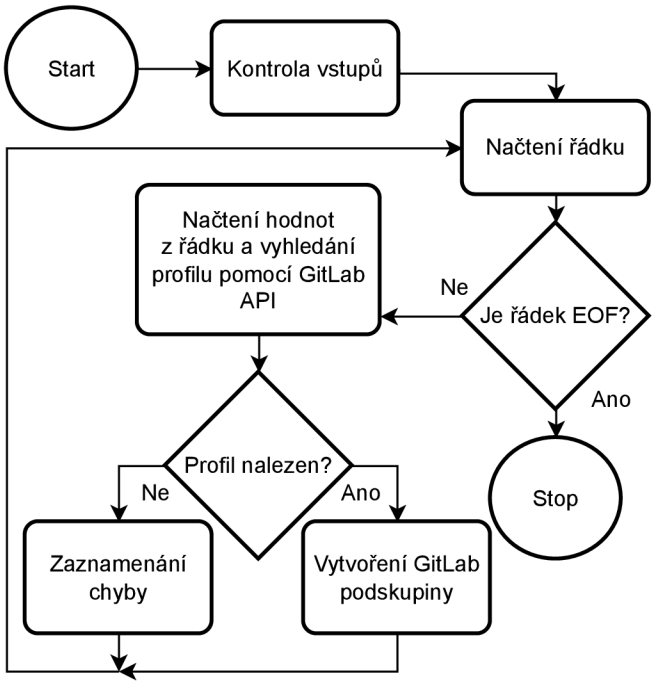
```
TOKEN=
BDS_PROJECTS_SUBGROUP_YEAR_ID=
FILE_NAME=renegades$(date +% F).csv
```

Dále tento skript k fungování vyžaduje soubor ve formátu `csv`, který je mu předán jako argument. Struktura tohoto souboru je jednoduchá, pro každého studenta obsahuje jeden řádek, kde je předvyplněné jeho VUT ID, jméno a příjmení. Po studentovi se vyžaduje pouze vyplnit jméno jeho GitLab účtu, zbytek informací by měl stáhnout učitel z informačního systému školy. Tento soubor tedy slouží k vyhledání studentských účtů a pojmenování podskupin podle studenta, pro přehlednost při hodnocení. Soubor musí každý rok vytvořit učitel předmětu, ale v rámci praktické části byla vytvořena šablona souboru, aby bylo předvyplnění jednodušší. Vyplněný dokument vypadá obdobně jako tabulka 3.1 při otevření v tabulkovém editoru.

Tab. 3.1: Soubor studentů formátu `csv`

ID	Name	GitLab Account
231285	Szymutko Marek	marek-szymutko
000000	Testovič Testan	@borek_testovatel
⋮	⋮	⋮

Fungování skriptu je pak nastíněno v diagramu 3.2. Skript vytváří podskupiny ve skupině, jejíž ID je uvedeno v souboru `.env` (viz výpis 3.1). Skupiny jsou se jménem `<prijmeni>-<jmeno>-id-<vut_id>`, popis skupiny je vždy „GitLab space for



Obr. 3.2: Vývojový diagram skriptu pro vytvoření GitLab skupin

student `<prijmeni> <jmeno>`, ID: `<vut_id>`“. Díky tomu je pro učitele jednoduché se ve skupinách orientovat. Student dostane přidělena práva v této skupině vytvářet repozitáře.

Pokud se nepovede nalézt GitLab účet studenta, je řádek se jménem tohoto studenta přepsán do souboru odpadlíků, jehož jméno je dáno proměnnou `FILE_NAME` ze souboru `.env` (viz výpis 3.1). Z tohoto řádku je odstraněno jméno uživatelského účtu studenta, aby byla naznačena jeho nesprávnost. ID a jméno však v souboru zůstane. Díky tomu učitel jednoduše najde žáky, u kterých se nepodařilo nalezení účtu a vytvoření skupiny na GitLabu. Studentům může přímo zveřejnit soubor odpadlíků, ať se zapíší znovu a správně, a poté spustit skript nad tímto souborem.

Pokud je uživatelský účet nalezen, ale jeho skupina nemůže být vytvořena, je do chybového souboru zkopírován celý řádek ze souboru studentů. To je nejpravděpodobněji způsobeno tím, že cesta ke skupině je již zabraná, tedy tento student již svou skupinu má. Tento jev je nejspíše chybou na straně učitele, který se pokusil spustit skript se stejnými studenty ve stejné skupině.

Pro použití skriptu každý rok je potřeba jen každý rok aktualizovat ID skupiny v souboru `.env` a výpis studentů v souboru formátu `csv`. Skript by samozřejmě mohl být použit i v jiných předmětech. Pro tento účel by bylo vhodné změnit název proměnné `BDS_PROJECTS_SUBGROUP_YEAR_ID`, jiné změny však tento skript nevyžaduje.

Pokud je spouštěn tento skript nad souborem pojmenovaným `students.csv`,

pak je potřeba název souboru poskytnout jako argument při spouštění a správný příkaz je tedy `bash init_groups_and_permissions.sh students.csv`.

3.3.2 Vytvoření skupin pro týmy žáků

Pro vytvoření univerzálnější sady nástrojů byl vytvořen i skript s podporou vytváření studentských skupin o větším počtu žáků. Tento skript k fungování vyžaduje soubor `.env` stejného formátu jako pro předchozí skript. Prázdná verze tohoto souboru je tedy vyobrazená ve výpisu 3.1.

Tento skript vyžaduje 2 vstupní parametry, oba soubory formátu `csv`. První z nich je naprosto totožný se vstupním souborem z předchozího skriptu, jeho příklad je tedy uveden v tabulce 3.1. Tento soubor obsahuje tabulku možných studentů, kteří budou členy generovaných skupin.

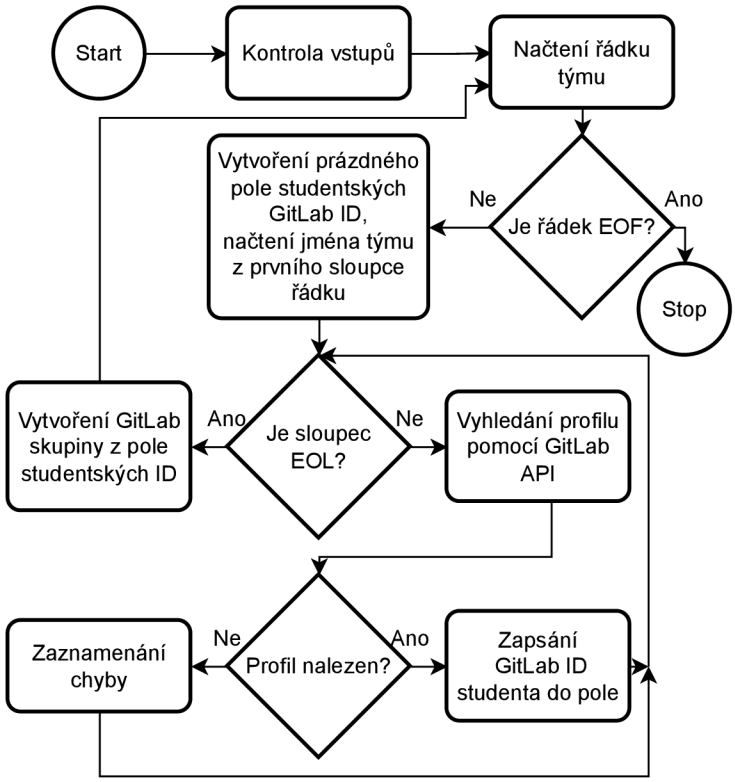
Druhý vstupní soubor obsahuje informace o rozdělení studentů do jednotlivých skupin. V prvním sloupci je vždy název týmu, všechny ostatní sloupce tohoto řádku jsou jednotlivá VUT ID členů tohoto týmu. Členů může být libovolné množství a toto množství může být různé napříč týmy. Ukázkový vyplněný soubor je zobrazen tabulkou 3.2. Tuto tabulku vyplňují studenti sami, proto je vhodné do ní přidat ukázku příkladným vyplněním jednoho řádku učitelem. Tento řádek je poté potřeba smazat před použitím skriptu.

Tab. 3.2: Soubor týmů formátu `csv`

`Team name`	`Student IDs`			
`Habsburkové`	`000000`	`111111`	`333333`	`...`
`Přemyslovci`	`231285`	`000000`		
⋮	⋮	⋮	⋮	⋱

Fungování skriptu je naznačeno v diagramu 3.3. Název skupiny si tedy určí studenti sami. V jejím popisu na platformě GitLab je však napsáno, o které studenty se jedná. Tyto údaje jsou využity z prvního souboru obsahujícího jméno a VUT ID studentů.

I tento skript vytváří výstupní soubory pro zaznamenání chyb. Protože ale má 2 vstupní soubory, i chybové výstupy jsou 2. První vstup má chybové chování totožné s prvním skriptem pro samostatné podskupiny (viz část 3.3.1). Pokud seznam týmů (druhý soubor) obsahuje ID, které neexistuje v prvním souboru, je vygenerován soubor `team_$FILE_NAME`, tedy soubor začínající slovem `team` a pokračující řetězcem definovaným v souboru `.env` (viz 3.1). V tomto souboru jsou zkopírovány celé řádky obsahující chybné VUT ID. Všechna nenalezená VUT ID jsou označena v buňce dvěma vlnovkami z každé strany (např. VUT ID `000000`, které by nebylo



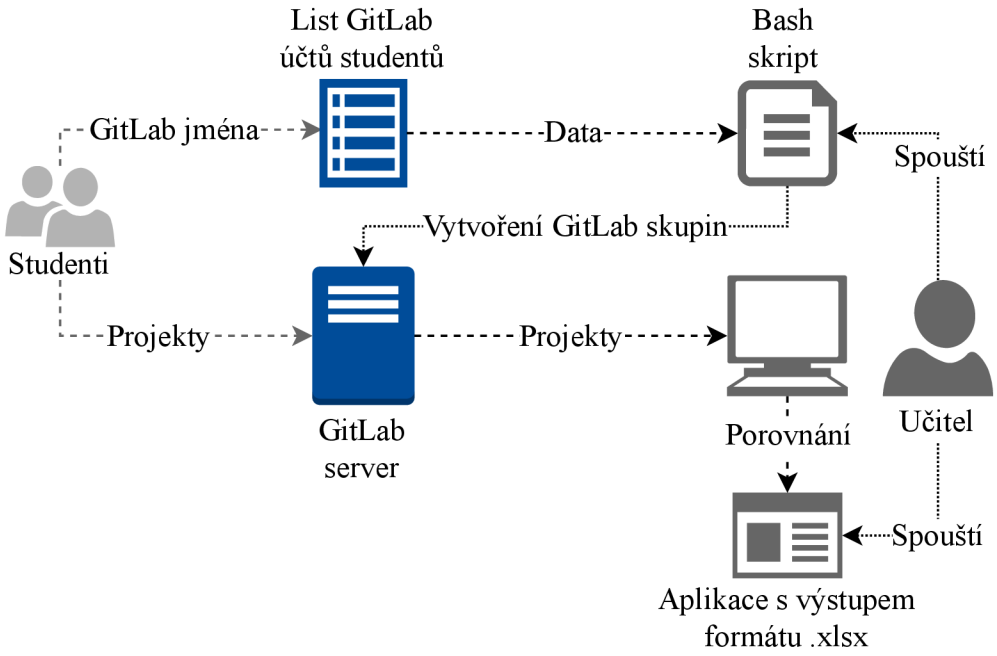
Obr. 3.3: Vývojový diagram skriptu pro vytvoření týmu

v seznamu studentů, bude označeno tímto způsobem: `~~000000~~`). Tyto neúplné skupiny nebudou vytvořeny. Je na učiteli, aby kontaktoval představitele skupiny a opravil chybně dopsané VUT ID. Poté je možno z výpisu vymazat tyto vlnovky a použít jej pro nové spuštění programu pro doplnění chybných skupin.

Pro spuštění skriptu se souborem studentů `students.csv` a týmů `teams.csv` je potřeba zadat příkaz `bash init_team_project_groups_and_permissions.sh students.csv teams.csv`.

3.3.3 Archivace repozitářů

Aby bylo zaručeno, že studenti své projekty po skončení semestru nesmažou, byl vytvořen třetí skript, který má za úkol všem studentům snížit práva ve skupinách. Tento skript také vyžaduje soubor `.env` jako byl použit pro předchozí dva skripty (viz výpis 3.1). Případně může být tento soubor kratší o poslední řádek, protože nezaznamenává chyby do souboru. Tento skript nalezne všechny podskupiny té skupiny, jejíž ID je v souboru `.env`. V každé podskupině poté nalezne všechny členy a všem kromě zakladatele skupiny (záznam o tomto členovi neobsahuje atribut `"created_by"`) sníží práva ve skupině.



Obr. 3.4: Přehled způsobu odevzdávání studentských projektů

3.3.4 Celkový postup odevzdání

Studentům bude na prvním cvičení předložen tabulkový soubor přístupný přes jejich školní Microsoft účty. Tento soubor bude mít formu podobnou jako 3.1. Studenti budou požádáni o vyplnění jejich existujícího GitLab účtu, nebo si založí účet nový. Název tohoto účtu zaznamenají do tabulky.

Učitel vytvoří GitLab skupinu pro aktuální akademický rok. Poté převede dříve zmíněnou tabulku do formátu `csv` a uloží si ji lokálně. S její pomocí a se skriptem popisovaným v části 3.3.1 vytvoří učitel rychle všechny potřebné skupiny pro studenty. Skript je psán v programovacím jazyce Bash na vyžádání vedoucího práce, jelikož je tento skriptovací jazyk přístupný z jakékoli platformy. (Na platformě Windows je možné využít program `git cli` využívající `bash`.)

Studenti pak své projekty nahrávají do repozitářů, které si sami vytvoří ve svých skupinách. Musí pouze dbát na správné pojmenování repozitářů pro jejich spolehlivé nalezení při kontrole plagiátorství. V průběhu semestru totiž studenti vytvoří repozitářů několik, detekce se však týká jen jednoho z nich. Jména repozitářů k vytvoření jsou vždy uvedena v zadání práce, kterou učitel studentům poskytne.

Na závěr semestru jsou pak skupiny archivovány, aby studenti nemohli své repozitáře smazat. K tomu je využit skript popisovaný v části 3.3.3.

Celkový přehled odevzdávání je pak uveden na obrázku 3.4.

4 Kontrola plagiátorství

Kontrola plagiátorství studentských projektů je realizována pomocí programu napsaného v jazyce Python o rozsahu přibližně 2200 řádků kódu. K realizaci tohoto projektu bylo využito vývojové prostředí PyCharm a formátovací modul Black. Jeho zdrojový kód je kromě elektronické přílohy také zveřejněn na platformě GitLab [35].

První funkcionalita, kterou bylo nutné vyvinout, je rozbor a načítání zdrojového kódu v jazycích Java a Python, aby bylo možné s částmi kódu pracovat jako s objekty v Pythonu. Nejprve byl projekt započat s užitím regulárních výrazů, ale toto řešení se rychle ukázalo jako nevhodné kvůli špatné orientaci v kódu. Proto bylo pro práci potřeba najít, vyzkoušet a využít kompilátor abstraktních syntaktických stromů Javy v Pythonu. Využit je kompilátor `javalang`, jak již bylo uvedeno dříve, v části 2.1. Pro kompilování AST Pythonu není potřeba žádné další moduly, lze využít vestavěnou knihovnu `ast`.

Aplikace je spustitelná z příkazové řádky a pomocí modulu `argparse` jí bylo vytvořeno CLI. Pro zobrazení možností lze využít možnost `--help`. Bez specifikování možností zkusí aplikace vyhledat soubor `.env`, stáhnout projekty z GitLab skupiny specifikované v tomto souboru a porovnat je.

4.1 Kroky detekce plagiátorství

Postup byl zvolen tak, aby byl proces detekce co nejjednodušším pro uživatele (tedy učitele). Automatizovány jsou všechny kroky od stažení projektů ze služby GitLab po vytvoření výstupního souboru formátu `xlsx` (formát pro tabulkové procesory).

Před spuštěním programu je potřeba do systému nainstalovat systém Git a Python verze 3. Dále je potřeba doinstalovat potřebné balíčky a případně uvést potřebné hodnoty do souboru `.env`. Tento soubor je podobný jako byl soubor `.env` pro skripty na správu GitLab skupin (viz výpis 3.1), pouze nepotřebuje ke svému chodu třetí proměnnou, tedy název souboru pro chybové výpisy.

Implementovaný princip detekce plagiátorství lze rozdělit do několika kroků:

1. Stažení projektů z GitLabu
2. Rozhodnutí o typu projektu
3. Načtení abstraktních syntaktických stromů z nalezených souborů
4. Uložení AST do vlastních datových typů
5. Porovnání jednotlivých projektů na základě definovaných vlastností
6. Prezentování výsledku porovnání

Všechny tyto kroky budou dále popsány.

4.1.1 Stažení projektů z GitLabu

Tato část vyžaduje, aby v systému, na kterém je program spuštěn, měl nainstalovaný Git pro použití v příkazové řádce. Tuto skutečnost si program ověří, a pokud Git nainstalovaný je, pokračuje dále. Program poté prochází skupiny vytvořené pomocí skriptu popsaného v části 3.3.1 a podle jména vybírá projekty k naklonování. Ke komunikaci s API využívá knihovnu `requests` poskytovanou pod licencí Apache 2.0 [36]. Stahování projektů probíhá pomocí příkazu `git clone` a využívá ke stažení přístupový token, který patří majiteli skupiny. Token je vyžadován, jelikož je přístup studentským skupinám omezen.

Pokud tento proces zjistí, že je daný repozitář již naklonován, aktualizuje jej příkazem `git pull`. Při klonování nebo aktualizaci je pro každý nalezený projekt spuštěno samostatné vlákno, aby stahování projektů probíhalo rychleji.

Také je možné stahovat soubory z jakýchkoli jiných služeb hostující Git repozitáře. Je potřeba jen vytvořit soubor se seznamem takových projektů ve formátu `<url> [název]`, kde `<url>` je adresa, odkud má být projekt klonován a `[název]` je reprezentace projektu po klonování (takto bude pojmenována složka, kde bude projekt uložen). Validní název je jakákoli posloupnost, kterou podporuje daný operační systém pro pojmenovávání složek, včetně mezer. Pro naklonování všech projektů uvedených v seznamu `projects.txt` je potřeba přidat při spuštění možnost `-p projects.txt`. Pokud by tedy byl použit soubor z výpisu 4.1, aplikace by se pokusila klonovat projekty do složek `Project By Marek Szymutko` a `Testan Testovic`, pro klonování druhého repozitáře by byl použit token `token` specifikovaný v adrese repozitáře.

Výpis 4.1: Seznam projektů

```
https://gitlab.com/proj.git Project By Marek Szymutko
https://git:token@gitlab.com/test.git Testan Testovic
```

Stejným způsobem je také možné klonovat vzorová řešení projektu (ta nebudou porovnávána vzájemně, jejich porovnávání bude uskutečněno jen se studentskými projekty). Seznam vzorů v souboru `templates.txt` lze přidat možností `-t templates.txt`. Nebo je možnost pouze pracovat s již staženými projekty a nevyužívat nakonfigurované parametry pro klonování. Pro takový běh je potřeba použít možnost `--offline`. Při této možnosti ani nebude vyžadován nainstalovaný Git.

4.1.2 Rozhodnutí o typu projektu

Vzhledem k implementaci porovnávání projektů v Pythonu a v Javě je potřeba rozlišit, který student odevzdal jaký typ projektu. K tomuto účelu je spočítán počet

souborů s koncovkou `.java` a `.py` a podle výsledku je učiněno rozhodnutí.

4.1.3 Načtení abstraktních syntaktických stromů z nalezených souborů

Zvolené kompilátory AST tvoří objekt abstraktního syntaktického stromu z textového vstupu. Proto jsou obsahy jednotlivých souborů načteny do řetězce, který slouží jako vstup kompilátoru. Z výstupu kompilátoru je poté tvořen strom vlastních datových typů.

4.1.4 Uložení AST do vlastních datových typů

Pro reprezentaci informace zdrojových kódů byly definovány vlastní datové typy. Jsou do nich ukládány jen informace relevantní pro porovnávání, a proto je práce s nimi efektivnější, než s celým objektem reprezentujícím AST. Načítání projektů do paměti je paralelizováno pro rychlejší zpracování.

Tyto typy dědí z abstraktní třídy `ComparableEntity`, která obsahuje základní rozhraní pro porovnávání datových typů. Její dědičné třídy musí mít přetíženou abstraktní metodu `compare`, díky čemuž je porovnávání jednoduché a snadno pozměnitelné. Tato metoda je pro každou třídu datového typu přetížena jiným způsobem, ale vždy je návratový typ této metody instance typu `Report`. Tento typ bude popsán v sekci 4.1.5. V následujících částech budou popsány jednotlivé datové typy pro reprezentaci a porovnání studentských prací. Nejprve budou popsány objekty popisující kód psaný v jazyce Java, poté objekty pro reprezentaci AST jazyka Python.

Datový typ `JavaModifier`

Instance této třídy drží informace o modifikátorech proměnných, tříd a metod. Modifikátory jsou např. klíčová slova `static`, `public`, `final`, `private` apod.

Při porovnání 2 identifikátorů je navráceno buď skóre 100 při shodě nebo 0 při zjištění rozdílných identifikátorů.

Datový typ `JavaType`

Ve zdrojovém kódu jsou datové typy uvedeny před názvem identifikátoru, který bude tohoto datového typu (např. `int cislo = 1;`). Objekty `JavaType` jsou vytvářeny při načítání tříd vytvořených studentem a jsou poté zapsány do atributu `user_types` objektu `JavaProject`. Díky tomu kdykoli je v kódu použit nějaký identifikátor, lze pomocí jeho deklarace zjistit název datového typu. Tento název bude vyhledán

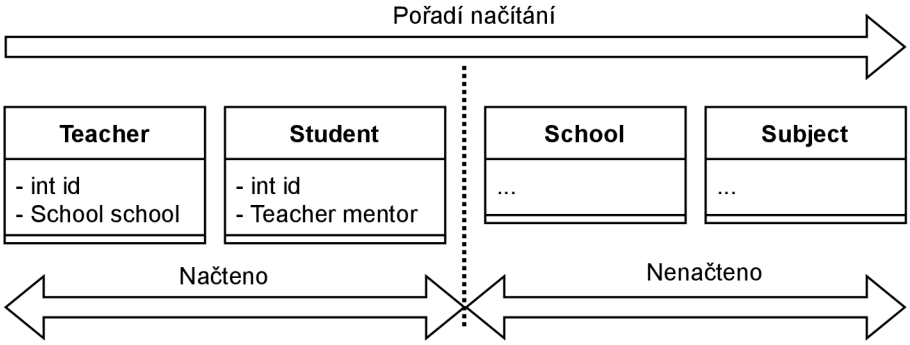
v paměti projektu a pokud bude nalezen, jedná se o uživatelsky definovaný datový typ.

Porovnávání datových typů je závislé na tom, jestli je tento typ uživatelsky definovaný nebo ne. Pokud není, stačí porovnávat názvy datových typů. Skóre bude 100 při shodě, 0 při rozdílných názvech. Pro zamezení maskování zkopírovaných projektů je navíc používán slovník „podobných“ datových typů. Při jeho použití se sice skóre sníží, ale nebude nulové a lze tímto odhalit záměnu datových typů jako `HashSet` a `TreeSet`, které mají stejnou funkcionalitu.

Pokud se jedná o uživatelsky definovaný datový typ, je převeden na sadu vnějších importovaných nebo základních datových typů pomocí atributů třídy, která definuje tento typ. Převod je rekurzivní, tedy nemůže se stát, že by po provedení sada stále obsahovala typy, které jsou výtvorem studenta. Tato sada je pak porovnávána obdobně jako uživatelsky nedefinované datové typy, tedy podle názvu typu. Na pořadí není brán zřetel. Jako skóre je vypočítán průměr všech porovnání a jeho váha je závislá na délce porovnávané sady.

Datový typ `JavaVariable`

Tento objekt reprezentuje identifikátory proměnných a ke každému identifikátoru je přiřazen datový typ, který je známý z deklarace. Proces inicializace typu proměnných je náchylný na chybu způsobenou sériovým načítání souborů, a proto je potřeba, ať jsou detaily datových typů načítány až po inicializaci všech tříd. Pokud by totiž datový typ proměnné nebyl znám v době inicializace objektu této proměnné, byl by mylně vyhodnocen jako uživatelsky nedefinovaný. Tento problém je ilustrován příkladem na obrázku 4.1, který znázorňuje smyšlený studentský projekt.



Obr. 4.1: Problém načítání

Při inicializaci třídy `Teacher` by došlo k problému, jelikož v paměti ještě neexistuje záznam o uživatelsky definované třídě `School`. Proto pokud by byly detaily datového typu `School` načítány ihned, došlo by k nesprávnému vyhodnocení jeho

původu a byl by označen jako uživatelsky nedefinovaný. Při inicializaci třídy `Student` by pak už nedošlo k problému, jelikož je již předem načtena třída `Teacher`, takže by proměnná `mentor` byla správně identifikována jako uživatelsky definovaný datový typ.

Tento problém je v práci řešen *líným načítáním* (anglicky *lazy loading*). Objekty `JavaVariable` se inicializují pouze s textovým identifikátorem datového typu a detaily jsou načítány až po přečtení všech souborů v projektu. Více podrobností o tomto řešení je popsáno v části 4.3.3. Všechny datové typy v projektu jsou po načtení kontextu zapsány do atributu instance třídy `JavaProject` reprezentující daný projekt. Odtud jsou potom předávány reference pro datové typy jednotlivých proměnných[1]. Pro Python podobný systém napsán nebyl, jelikož to není striktně typový jazyk.

Instance třídy `JavaVariable` jsou pak porovnávány na základě modifikátorů a datového typu. Z těchto porovnání je vypočítán vážený průměr.

Datový typ `JavaStatementBlock`

Objekty této třídy reprezentují jednotlivé instrukce v těle metody. Jedná se o části těla oddělené středníky. Každá část je samostatný podstrom AST, který je v rámci tohoto objektu převeden na mapu přiřazující každému typu uzlu počet jeho výskytů v AST. Také jsou mapovány metody volané z každého bloku a pokud jsou volané metody nalezeny v rámci projektu, budou k porovnání přidány všechny instrukce z těla volané metody.

Objekt nebude vytvořen, pokud se jedná o prázdný výrok. Pokud by tomu tak nebylo, bylo by možné snižovat skóre velkým počtem středníků za sebou.

Tyto části jsou porovnávány pomocí metrik, jak bylo popsáno v části 1.6. Záleží tedy pouze na typu uzlu a počtu jeho výskytů. Definice takového porovnávání je definovaná v abstraktní třídě `AbstractStatementBlock` pro opětovné použití kódu v projektech v jiných jazycích.

Datový typ `JavaParameter`

Jedná se o argument metody. Tento datový typ se od třídy `JavaVariable` liší absencí modifikátorů a odlišnou strukturou podstromu AST, který jej deklaruje.

Argumenty jsou porovnávány pouze pomocí jejich typů, které jsou také načítány líně, podobně jako u třídy `JavaVariable`.

[1] Toto řešení šetří paměť, protože celkem je v paměti jen tolik objektů, kolik bylo definováno datových typů a vytváří se pouze nové reference na ně.

Datový typ `JavaMethod`

Jedná se o reprezentaci metod. Inicializace probíhá načtením třídy, avšak i tato třída využívá líného načítání pro předcházení problémům, jak bylo uvedeno výše.

Pro porovnávání jsou užity argumenty a návratový typ metody. Pokud skóre těchto částí bude větší než 80 bodů ze 100 nebo kód bude spuštěn v módu plnohodnotného skenování (výchozí mód), budou porovnávány také všechny bloky v těle metody. Pro aplikování této hraniční hodnoty je tedy potřeba použít možnost `--fast` při spouštění.

Datový typ `JavaClass`

Tato třída slouží k reprezentaci tříd zdrojového kódu. Třídy jsou porovnávány podle atributů a metod. To znamená, že jejich skóre podobnosti se počítá jako vážený průměr porovnání těchto částí.

Datový typ `JavaFile`

Tento objekt reprezentuje soubor formátu `java`. Nebyla zvolena reprezentace těchto souborů pomocí třídy `JavaClass`, protože jeden soubor formátu `java` může teoreticky obsahovat více tříd.

`JavaFile` drží informace o balíčku a importovaných třídách. Pomocí nich umožňuje hledat datové typy zmíněné v těle tříd souboru, čehož se využívá při hledání původu volaných funkcí a lokálních proměnných.

Tento typ je porovnáván pouze podle tříd, které obsahuje.

Datový typ `JavaProject`

Tento datový typ vůbec nevychází z AST generovaného kompilátorem, pouze sdružuje kompilované soubory do logického celku s určitým rozhraním. Porovnávání projektů probíhá na základě porovnání jednotlivých objektů `JavaFile`, které obsahuje. Tato třída také dědí z třídy `AbstractProject`, která slouží jako rozhraní pro přídavné informace o typu projektu (jazyk, ve kterém byl projekt napsán a binární hodnotu, zdali se jedná o vzorové řešení).

Datový typ `PythonStatementBlock`

Stejně jako `JavaStatementBlock` (popsáno v části 4.1.4) je i tato třída odvozena od `AbstractStatementBlock` a přebírá logiku porovnávání pomocí číselných metrik z této třídy (vzorec pro výpočet podobnosti byl definován v části 1.6).

Datový typ `PythonFunction`

Tento datový typ slouží pro reprezentaci jak funkcí, tak i metod v Pythonu. Velmi se podobá třídě `JavaMethod`.

Při porovnání je prvně použito rozhraní funkcí či metod. Parametry jsou roztříděny podle typu (striktně poziční, argumenty definované klíčovými slovy a ostatní), a poté je pro jednotlivé druhy argumentů aplikován stejný vzorec jako v části 1.6. Pokud je aplikováno plné skenování nebo je překonána úroveň shody rozhraní 80, jsou zahrnuty i uzly AST podle stejného vzorce (stejně jako ve třídě `JavaMethod`).

Datový typ `PythonClass`

Datový typ slouží pro reprezentaci tříd. Porovnávání těchto objektů závisí na výsledcích porovnání metod těchto tříd a na uzlech AST, které nedefinují metody.

Datový typ `PythonFile`

Reprezentuje jednotlivé soubory typu `.py`. Porovnávání probíhá na základě funkcí, tříd a ostatních uzlech AST obsažených v souborech (v Pythonu lze uvádět proceduralní části kódu i mimo třídy a funkce).

Datový typ `PythonProject`

Obdobně jako `JavaProject` slouží tento objekt ke sdružení dílčích částí projektu a k vyhledávání jeho částí. Porovnávány jsou jednotlivé soubory v projektu obsažené.

4.1.5 Porovnání jednotlivých projektů na základě definovaných vlastností

K porovnání jednotlivých stromových struktur dochází voláním metody `compare`, která byla již zmíněna dříve v sekci 4.1.4. Volání této metody je hierarchické. To znamená, že pokud je zavolána nad instancemi typu `JavaProject`, je každá instance `JavaFile` v prvním projektu porovnána s každým souborem z druhého projektu a nejlepší shoda je uložena. V každé instanci `JavaFile` je pak porovnávána každá třída a nejlepší shoda uložena atd. Takto jsou porovnávány všechny části projektu až po datové typy proměnných, či jejich modifikátory. Výstup metody `compare` je objekt typu `Report`. Tato třída má přetížené magické metody `__add__`, `__lt__` a `__eq__`, díky čemuž je možno tyto objekty porovnávat a kombinovat pomocí standardních operátorů, (tj. znaménko + a operátory pro porovnávání $<$, $>$, ...). Tato třída obsahuje následující atributy:

1. Skóre podobnosti (celočíselná hodnota 0–100)

2. Váhu skóre podobnosti[2]
3. Reference na prvky, jejichž porovnáváním `report` vznikl
4. Dědičné `reporty`[3]

Práci s tímto objektem je vhodné demonstrovat na příkladu.

Uvažujme objekt `report_tridy`, který nese informace o porovnání dvojice tříd ze studentských projektů. K tomuto `reportu` má být přidána informace o porovnání metod z těchto tříd. Vybereme tedy množinu všech metod z první třídy (bude mít m prvků) a každý prvek porovnáme pomocí metody `compare` s každou metodou v druhé třídě, což je množina o n prvcích. Výsledek takového porovnání si lze představit jako tabulku o m řádcích a n sloupcích. Ze všech výsledků takového porovnání je poté vybrána nejlepší shoda pomocí metody `max`, která je součástí základních nástrojů v Pythonu a využívá přetížené magické metody `__lt__` a `__eq__` (definují chování operátorů porovnání). Tímto jsme zjistili, která dvojice metod si nejlépe odpovídá a tento výsledek považujeme za vhodnou nalezenou součást bijektivního zobrazení a výsledek tohoto porovnání bude nahrán do objektu `report_metody`. Z tabulky odstraníme řádek a sloupec, který obsahoval vybraný nejlepší výsledek, čímž je zajištěno, že každý prvek je přiřazen jen jednomu jinému prvku.

Vybraný `report_metody` nyní lze přidat do objektu `report_tridy` pomocí řádku kódu podobného tomuto: `report_tridy += report_metody`.

Tato operace využívá magickou metodu `__add__`. Přetížená metoda pro sčítání vypočte vážený průměr skóre těchto dvou objektů a výsledný objekt bude mít váhu rovnou součtu vah těchto objektů. Dále tato metoda obsluhuje připojování dědičných `reportů`. Vzhledem k tomu, že `report_tridy` vznikl porovnáváním třídy a `report_metody` vznikl porovnáváním metod, je `report_metody` připojen mezi dědičné `reporty` objektu `report_tridy`.

Tento postup je opakován, dokud není tabulka o původní velikosti $m \times n$ vyčerpána. K tomu navíc algoritmus drží informace o uzlech, které použity nebudou vůbec (kvůli různé velikosti počátečních množin) a pro ně vloží záznam o nenalezení vhodné bijekce s hodnotou skóre 0. Díky informaci o hierarchii `Reportů` lze poté výsledný strom procházet jako jakoukoli jinou stromovou strukturu.

Příklad tohoto algoritmu je ilustrován tabulkami 4.1. Hodnoty v tabulce jsou skóre porovnání jednotlivých metod. Zde je vybraná nejvyšší hodnota znázorněna modrou barvou, smazané řádky a sloupce barvou červenou. V posledním kroku zbývá metoda `updateCar`, ke které nebyla nalezena žádná vhodná shoda. Záznam o této skutečnosti se přidá k výslednému `Reportu`.

[2]Váha skóre slouží k tomu, aby nebylo skóre podobnosti znehodnoceno. Na váhu má vliv počet prvků, které byly využity k porovnání.

[3]Dědičné `reporty` je pole všech `reportů`, které byly do daného reportu započítány. Reporty tedy tvoří hierarchickou stromovou strukturu, která odpovídá hierarchii volání metody `compare`.

Tab. 4.1: Algoritmus výběru shod

metody	createBook	deleteBook
updateCar	15	42
deleteCar	14	87
createCar	97	35

⟶

metody	deleteBook
updateCar	42
deleteCar	87

4.1.6 Prezentování výsledku porovnání

V praktické části semestrální práce byly implementovány dva způsoby prezentování výsledků porovnávání. Rozsáhlý textový výpis a tabulková reprezentace za použití formátu `xlsx`.

Textový výpis je použitelný pouze při porovnávání dvou projektů, vypisuje přesně které části projektu dosáhly jakého skóre. Pro testování programu byly použity dvě verze vlastního projektu z předmětu BPC-BDS z předchozího roku, každý o délce přibližně 1400 řádků kódu v souborech s koncovkou `java`. Mezi těmito verzemi došlo k různým drobným úpravám pro testování funkčnosti kódu (nejčastěji přejmenování identifikátorů). Výsledný výpis činil přibližně 500 řádků. Pokud by toto řešení bylo použito pro konečnou verzi programu, při porovnávání sta studentských projektů by bylo vygenerováno až 2,5 milionů řádků.[4] Tento výpis byl později zkrácen, aby reprezentoval pouze soubory, třídy a jejich metody. S touto úpravou obsahoval výpis týkajících se stejné dvojice projektů pouze 140 řádků. I takové číslo by ale pro reprezentaci velkého množství projektů nebylo vhodné. Textový výpis je přesto možné zapnout přepínačem `--debug`.

Použití tabulky se jeví jako mnohem přívětivější řešení pro skupiny projektů větší než 2 vzorky. Bylo zvoleno řešení s použitím balíčku `XlsxWriter` (licence *BSD 2-Clause License*) [37]. Pomocí této knihovny je vytvořen tabulkový soubor, ve kterém je vyobrazeno skóre podobnosti při porovnání každého projektu s každým jiným, přičemž řádky a sloupce jsou označeny názvy projektů (při klonování z předem připravené GitLab skupiny obsahují názvy projektů jména, příjmení a VUT ID studentů), jednotlivé buňky tabulky obsahují číselný výstup porovnání podobnosti projektů a každá buňka je pro přehlednost formátována adekvátní barvou (červená pro vysoké skóre, žlutá pro střední a zelená pro nízké). Přesná barva je vypočítávána z hodnoty skóre. Každá buňka tohoto přehledu rovněž slouží jako odkaz na podrobnější rozpis nalezených podobností, které jsou na dalších stránkách tohoto souboru. Navíc do sloupců jsou přidány vzorová řešení, pokud se učitel rozhodne taková řešení zahrnout do porovnání. Příklad celkového přehledu je uveden v tabulce

[4] 100 projektů porovnáváno *každý s každým* vytvoří $\left(\frac{100 \cdot 99}{2}\right)$ porovnání. Po vynásobení tohoto výrazu číslem 500 vyjde 2,475 milionů řádků.

4.2, pro zmenšení rozměrů tabulky byly však názvy projektů zkráceny. Jednotlivé

Tab. 4.2: Celkové výsledky porovnání

	vzor	**proj. 1**	**proj. 2**	**proj. 3**
proj. 1	94		69	97
proj. 2	27	69		35
proj. 3	95	97	35	

výsledky, na které vedou odkazy z první tabulky, jsou pak ilustrovány v tabulce 4.3. Stromová struktura porovnání je znázorněna vysázením prázdných buněk do dokumentu před podřazené části. Levé části projektu vždy náleží projektu, který je na prvním řádku uveden nalevo, pravé části pak náleží projektu napravo. Volitelně je také možné vypsat váhy jednotlivých částí použité při výpočtu váženého průměru.

Tab. 4.3: Jednotlivé výsledky porovnání

Project	proj. 1	proj. 2				69
	JavaFile	WebCtrlr.java	WCtrlr.java			91
		JavaClass	WebCtrlr	WCtrlr		91
			JavaMethod	initData	initPage	100
⋮	⋮	⋮	⋮	⋮	⋮	⋮

4.2 Dosažené výsledky a limitace

Pro efektivní shrnutí vlastností implementovaného řešení bude popsána reakce řešení na všechny uvedené druhy podvádění v softwarových projektech, které byly uvedeny v sekci 1.3.

4.2.1 Změna komentářů nebo formátování

Tento způsob maskování podvádění nemá na funkci systému nejmenší vliv, jelikož se komentáře ani formátování souborů nepromítá do abstraktních syntaktických stromů, které jsou použity k porovnávání.

4.2.2 Změna identifikátorů

Tímto jsou myšleny změny názvů proměnných, metod, tříd, balíčků, argumentů, atributů apod. Identifikátory nepatří k porovnávaným znakům ve studentských projektech. Z tohoto důvodu ani tato metoda obfuskace nebude mít vliv na správnost vyhodnocení plagiátorství.

4.2.3 Změna pořadí operandů ve výrazech

Ani tato metoda maskování kopírování nehraje roli. Výrazy jsou porovnávány jen podle typu výrazu, tzn. podle datového typu uzlu AST. Například v Javě bude řádek `a = b + c;` reprezentován abstraktním syntaktickým stromem podobným jako je ve výpisu 4.1. V Pythonu pak bude fungování obdobné.

Z tohoto výpisu program pro kontrolu plagiátorství tedy použije jen typy uzlů (tedy třídu objektu tohoto uzlu) a jejich množství. Těmi jsou v tomto případě `Assignment`, `MemberReference` a `BinaryOperation`. Na pořadí operandů tedy nezáleží.

Výpis 4.1: AST výrazu `a = b + c;`

```
Assignment(
    expressionl=MemberReference(member=a),
    value=BinaryOperation(
        operandl=MemberReference(member=b),
        operandr=MemberReference(member=c, operator=+)
    )
)
```

4.2.4 Změna datových typů

Tímto je myšlena například záměna typu `int` za `double` a podobně. V Pythonu je toto neaplikovatelné, jelikož Python je netypový jazyk. V Javě se s tímto typem obfuskace práce počítá, a proto obsahuje mechanismy skórování „podobných“ datových typů. Byl definován slovník těchto zaměnitelných datových typů a pokud datové typy nesouhlasí přesně, je použit výraz ze slovníku. Pokud je zjištěna shoda se slovníkem, je výsledek ohodnocen nižším skóre, ale toto skóre není nulové.

Uživatelsky definované datové typy jsou přeloženy na pole uživatelem nedefinovaných datových typů (použity jsou typy atributů tříd, na které tyto datové typy ukazují, toto překládání je rekurzivní) a pro tyto je využito porovnávání s použitím slovníku, jak bylo uvedeno v předchozím odstavci. Takže ani změna názvu datového

typu, který má stejnou strukturu, studentovi nepomůže ke skrytí zkopírovaného kódu.

4.2.5 Záměna výrazů za jejich ekvivalenty

Je myšlena například záměna podmínky `condition == false` za `!condition`. V Javě je proti takové záměně program imunní, jelikož porovnává v tělech metod jen typy uzlů AST, jak již bylo zmíněno v části 1.6. Oba takovéto výrazy jsou na výstupu kompilátoru reprezentovány jako objekt typu `BinaryOperation`, takže program by v tomto případě poznal shodu.

Avšak v Pythonu existuje mnoho způsobů vyjádření takových výrazů (`a!=b`, `not a == b`, `a < b or a > b` apod.). Všechny tyto výrazy používají mírně se lišící sadu uzlů AST a změna by mírně snížila skóre. Pro zmenšení tohoto efektu byly uzly těchto operací přidány do slovníku podobných uzlů.

4.2.6 Přidávání přebytečných proměnných nebo výrazů

Tento způsob maskování kopírovaného kódu by měl opravdu kýžený efekt snižování skóre. Uzly AST jsou totiž porovnávány metodou *každý s každým*. Pokud program nenalezne pro uzel shodu, skóre se sníží. Ani tak ale nebude skóre nulové a toto maskování pouze mírně degraduje výsledek.

Prázdné výrazy jsou pak z objektů filtrovány úplně a přidávání přebytečných středníků v nejmenším nezmění skóre porovnání projektů.

4.2.7 Záměna pořadí nezávislých částí kódu

Záměna pořadí nemá na výsledek programu nejmenší vliv. Kompilátor `javalang` i `ast` sám rozděluje nezávislé bloky do samostatných částí. Těmto blokům je pak vybírán nejpodobnější blok z druhého projektu, způsobem, který byl popsán v části 4.1.5. Jak již bylo zmíněno dříve, hledá se nejlepší shoda, ať už shoda existuje na kterémkoli místě.

4.2.8 Změna struktury iteračních výrazů

Tímto se myslí např. záměna `while` za `for`. S takovouto obfuskací program počítá a obsahuje slovník *podobných* typů uzlů. Podobný slovník byl popsán i v části 4.2.4. Tento slovník ale drží jen překlad podobných uzlů stromu z těl metod. Výsledné skóre je nejvyšší při přesně stejných typech uzlů. Uzly podobné překladem ze slovníku mají skóre snížené na polovinu a při nenalezení shody uzel snižuje celkové skóre.

V Pythonu pak iterační výrazy můžou způsobit větší nepřesnosti. Existují v něm totiž výrazu zvané *list comprehension*, které vypadají například takto: `[i for i in l]`. Při použití tohoto výrazu je použit uzel `ListComp`, avšak výraz lze jednoduše přepsat na `list(i for i in l)`. Tento výraz bude mít naprosto stejnou funkcionalitu, avšak použitý uzel je nyní `Call`, tedy volání funkce s názvem `list`. Samozřejmě, že skóre bude sníženo jen mírně vzhledem k celé metodě či funkci, dokonce i ostatní uzly v tomto výrazu se nezmění. Ale jedná se o způsob snížení výsledného skóre.

4.2.9 Změna struktury výrazů `if`, `else` a `switch`

Jak již bylo zmíněno v části 4.2.5, struktura těl metod neovlivňuje výsledek. Ten je ovlivňován pouze typy uzlů v tělech metod.

Záměna jednotlivých typů těchto výrazů by snížení skóre dosáhla, avšak jen mírného. I pro tyto uzly se využívá slovník podobných typů. V Pythonu verze 3.10 se navíc `switch` vůbec nepoužívá a bloky `if` a `elif` využívají stejné uzly AST.

4.2.10 Záměna volání metody za tělo této metody

I na tento způsob maskování je program připraven a kontroluje, jestli uzel typu `MethodInvocation` (v Javě) nebo `Call` (v Pythonu) ukazuje na metodu v projektu, nebo mimo něj. Pokud je tato metoda definovaná uživatelem, překopíruje její tělo této metody k tělu metody, odkud je volána a až poté proběhne porovnávání. Díky tomu je obfuskace odhalena a skóre sníženo jen mírně (nebude nejspíš nalezena vhodná bijekce pro původně volanou metodu).

4.2.11 Použití nestrukturních výrazů typu `goto`

Tyto výrazy nejsou v Javě ani v Pythonu využívány.

4.2.12 Kombinace vlastních a zkopírovaných částí kódu

Detekční algoritmus zkoumá celé programy do poměrně jemných částí, měl by tedy být schopen odhalit i malé části zkopírovaného kódu. Problém však je s reprezentací informace, jelikož pokud by měl program prezentovat shodu každého porovnávaného objektu, generoval by příliš velké množství výstupu. Proto je výsledek porovnání reprezentován pouze na úroveň projektů, souborů, tříd a metod, další části nejsou zobrazeny. Pokud by tedy studenti zkopírovali jen část metody, ve výsledku by takový podvod nemusel být rozpoznatelný.

4.2.13 Limitace

Kompilátor `javalang` je vyvinut pouze pro Javu verze 8. Pokud je ve studentských projektech použita syntaxe exkluzivní pro některou novější nebo starší verzi, kompilace kódu se nezdaří a soubor bude muset být vyloučen z porovnání. Toto však platí pouze pro jednotlivé soubory a zbytek projektu porovnán bude.

4.3 Ponaučení z práce

Implementované řešení kombinuje více přístupů k detekci plagiátorství. Využívá jak číselných metrik, tak i porovnávání struktury pomocí abstraktních syntaktických stromů. V průběhu implementace řešení bylo zjištěno několik dobrých řešení i slepých cest. Těmto se věnuje tato část.

4.3.1 Regulární výrazy

V průběhu práce na řešení bylo vyzkoušeno načítání kódu pomocí regulárních výrazů, které se ale rychle ukázalo jako nevhodné, a proto bylo v dalším průběhu zavrženo. Regulárních výrazů bylo potřeba velké množství a navíc se nepodařilo pomocí nich vhodně načítat všechny struktury, které se v kódu mohly objevit. Proto byly vytvořeny procedury pro kontrolu hloubky zanoření, které hledaly pravé ukončení metod, tříd a bloků kódu. Ty by jinak byly ukončeny první ukončovací složenou závorkou, ačkoli syntakticky tato závorka mohla patřit pouze vnořené části této metody, třídy či bloku. Tyto procedury však byly tvořeny „na míru“ implementovanému řešení, a proto by bylo velmi náročné v kódu cokoli měnit. Práce na řešení pomocí regulárních výrazů byla ukončena před dokončením implementace načítání metod a tříd, přičemž obě tyto části byly rozpracovány, ale ne zcela funkční.

4.3.2 Strojové učení

Další ponaučení se týká umělé inteligence. Ačkoli je možné detekci plagiátorství realizovat s užitím umělé inteligence, jak bylo uvedeno v části 1.5.4, není toto řešení nejvhodnější pro každou situaci a není nutné ji využít pro vývoj spolehlivého programu. Pro správné natrénování modelu je totiž potřeba velké množství správně upravených a zpracovaných dat. Projekt BPC-BDS se vyučuje teprve druhým rokem, proto není možné odevzdané projekty využít jako vstupní sadu pro strojové učení. Volně dostupné projekty v jazyce Java z internetu by samozřejmě bylo možné použít, ale pro strojové učení s učitelem by bylo potřeba všechny použité části kódu označit jako originál nebo plagiát. Strojové učení bez učitele by pro tento účel mohlo být

vhodnější, ale ani toto by nebylo možné bez dostatečně velkého množství předzpracovaných vstupních dat.

4.3.3 Optimalizace

Praktická část práce také vyžaduje řešení problémů ohledně optimalizace. Při porovnávání všech možných dvojic studentských projektů roste počet porovnání podle vzorce

$$N = \sum_{i=1}^{2} \left(\frac{n_i \cdot (n_i - 1)}{2} + n_i \cdot t_i \right),$$

kde N je celkový počet porovnání a n_i je počet projektů v programovacím jazyce i k porovnání, t_i je pak počet poskytnutých šablonových projektů pro jazyk i. Asymptotická časová složitost je v tomto případě $\mathcal{O}(n^2)$, tedy kvadratická. S množstvím projektů rychle stoupá časová náročnost jejich porovnání, proto je zapotřebí kód optimalizovat. K tomuto zrychlení běhu programu pomohlo využití tzv. `cached property`. V Pythonu jsou `properties` *gettery*, které vypočítávají hodnotu atributu, až když jsou zavolány. Pokud je tento atribut navíc po vypočtení uložen do paměti, jedná se o `cached property`. Pro účel této práce je toto výborná vlastnost, protože se projekt po načtení do paměti již nemění. Stačí tedy tyto atributy vypočítat jednou a pro další porovnání pouze držet hodnotu v paměti. Zároveň je však tato hodnota vypočítána, až když je jisté, že je načten celý projekt, což řeší problém způsobený sériovým načítáním, který byl popsán již dříve, v části 4.1.4.

Další optimalizace spočívá v uvolnění paměti při běhu programu. Pro reprezentaci výsledků porovnání není využit úplně celý strom, protože by byl výsledek příliš velký a nepřehledný. Proto uzly, které nebudou reprezentovány ve výstupu, není potřeba držet v paměti potom, co jsou jejich skóre a váha propagovány do výsledku. I přesto je však využití paměti počítače při běhu poměrně vysoké (při porovnávání projektů studentů BPC-BDS v akademickém roce 2022/2023 bylo využito 4,5 GB paměti touto aplikací samotnou).

5 Případová studie

Poznámka: Tato případová studie vychází z testu podobnosti uskutečněného po archivaci studentských repozitářů a s nejnovější verzí aplikace pro hledání podobností ve studentských projektech. Čísla se tedy mohou mírně lišit od případové studie uvedené v práci pro konferenci EEICT [38], pro jejíž účely byla využita data z 1. 1. 2023.

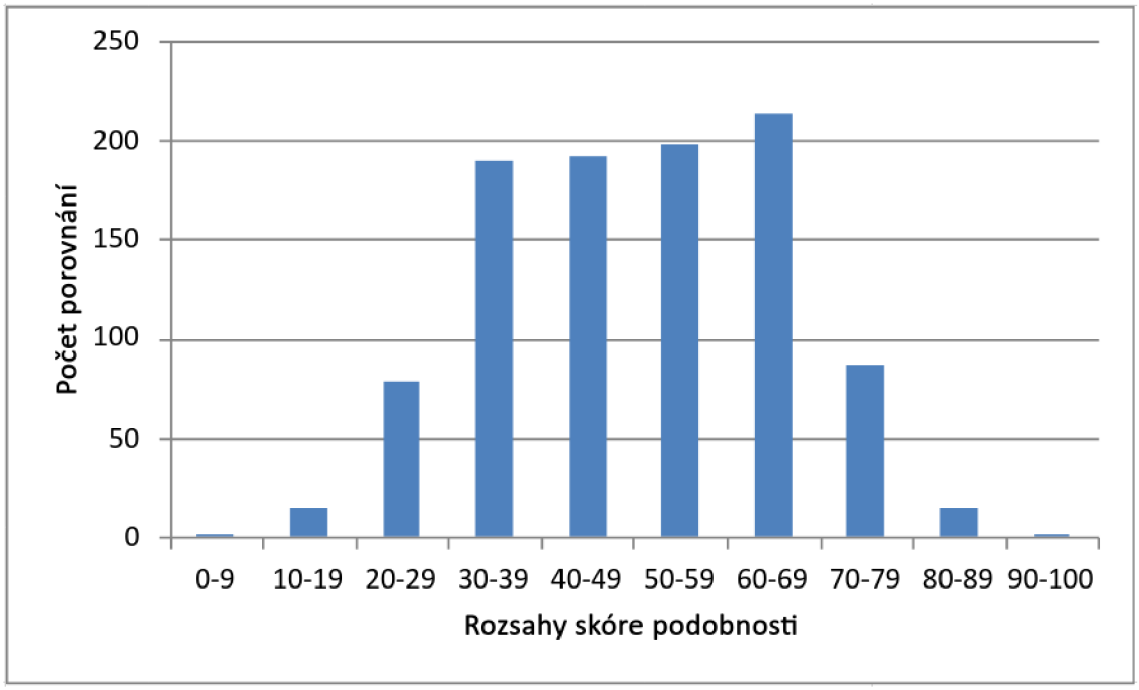
5.1 Celkový přehled

Do předmětu BPC-BDS bylo v akademickém roce zapsáno 60 studentů. Projekt číslo tři odevzdalo do správného adresáře na GitLabu 47 studentů z tohoto počtu. Jazyk Java si pro řešení zadání vybralo 44 studentů, jazyk Python si vybrali dva studenti. Jeden student odevzdal do repozitáře soubory neobsahující zdrojové kódy jazyka Java ani Python (v tomto repozitáři se nacházela pouze dokumentace a koncepční materiály). Odevzdané soubory napsané v Javě byly navíc porovnávány se vzorovým projektem, který dal k dispozici učitel předmětu BPC-BDS.

Celkový počet porovnání provedených pro projekty v Javě je 990 (každý studentský projekt porovnán s každým jiným: 946, a porovnání jednoho vzorového projektu s každým studentským: 44). Pro Python bylo potřeba jen jediné porovnání. Celkový počet bijektivních hodnocení podobnosti je tedy 991. Skóre podobnosti z těchto porovnání je zaneseno do histogramu 5.1.

5.2 Projekty v jazyce Java

Pro studenty píšící projekty v tomto programovacím jazyce bylo učitelem předmětu vytvořeno vzorové řešení. Toto řešení bylo zveřejněno pro inspiraci a také pro pomoc s dílčími úkoly, se kterými by si studenti samostatně nemuseli poradit. Bylo však avizováno, že studentské projekty budou hodnoceny na bázi originality, tudíž vysoké skóre podobnosti se vzorovým projektem není žádoucí, avšak není úplně zakázáno. Anonymizovaná vizualizace veškerých porovnávání v jazyce Java je na obrázku 5.2. Tento obrázek je pouze ilustrativní, vzhledem k počtu studentů je poměrně velký pro tuto stránku a není proto dobře čitelný. Jak již bylo zmíněno dříve, jednotlivé hodnoty v přehledu slouží jako odkazy ve vygenerovaném `.xlsx` souboru. Tyto odkazy směřují na detaily jednotlivých dvojic porovnávání, kde lze přehledně vidět, které části kódu vykazují znaky podobnosti. Ukázka takového detailu je na obrázku 5.3.

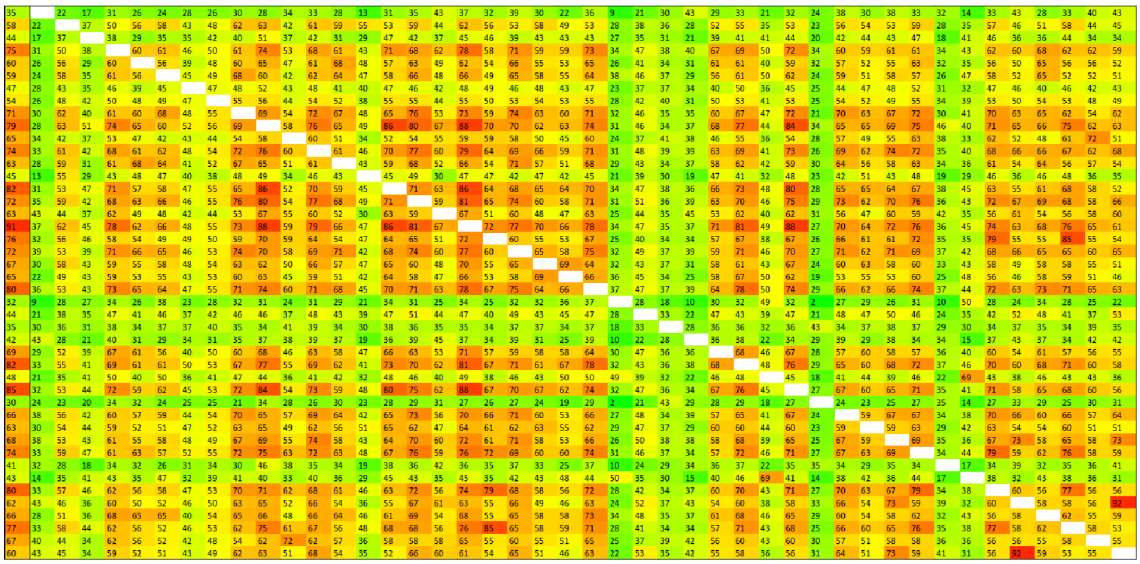


Obr. 5.1: Histogram výsledků porovnání

5.2.1 Vysoká skóre podobnosti

Nejvyšší skóre podobnosti byla hodnota 92 ze 100. Ani jeden ze studentů, kteří tohoto skóre dosáhli, nedosáhl příliš vysoké podobnosti s poskytnutým vzorovým řešením (první z dvojice má shodu se vzorem 62 bodů, druhý pak 60). Při procházení detailů tohoto hodnocení si lze povšimnout vysokého skóre podobnosti u většiny poskytnutých souborů, proto je vhodné tyto soubory otevřít a manuálně zkontrolovat, zdali se nejedná o náhodu. V tomto případě bylo zjištěno, že soubory obsahují nejen stejné konstrukty, ale i formátování. Jedinými pozorovatelnými změnami bylo přepsání hlášek pro logování a přejmenování identifikátorů v kódu. Tito studenti byli při obhajobě konfrontováni učitelem a ke spolupráci se přiznali. Krátký příklad podobného kódu, který se nacházel u těchto studentů, ale ne ve zdrojovém projektu, se nachází na obrázku 5.4.

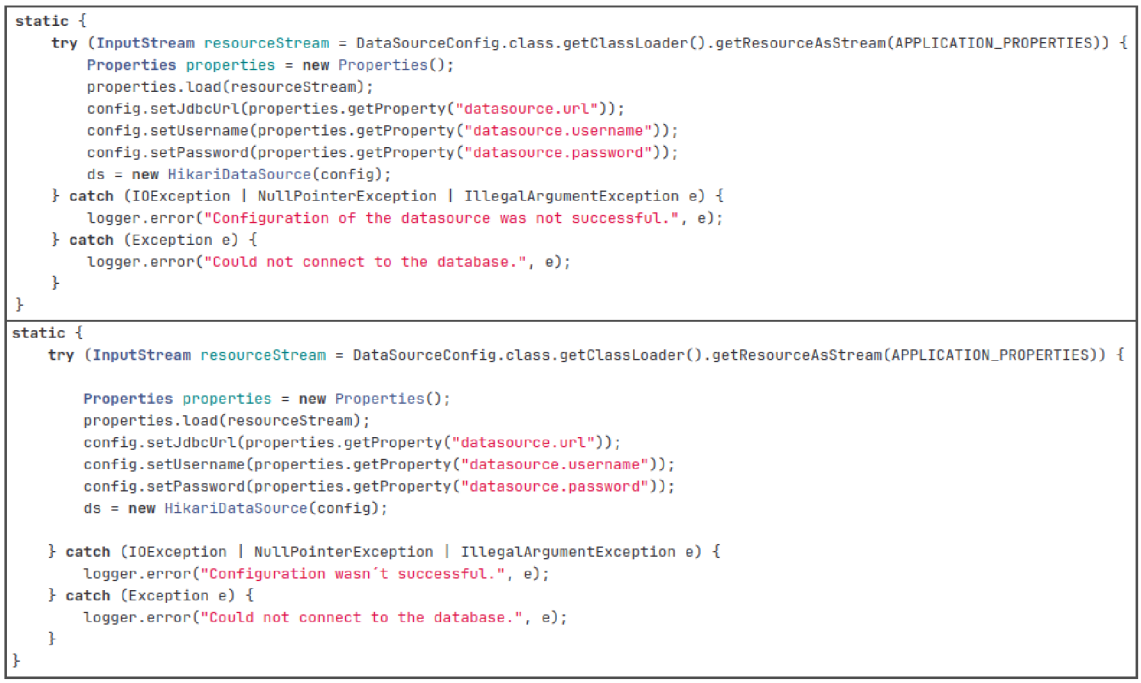
Druhé nejvyšší skóre podobnosti bylo 91 ze 100. Této hodnoty dosáhlo porovnání jednoho studentského projektu se vzorovým řešením. Další skóre nad 90 bodů již ve výsledku porovnání obsažena nejsou, ale existuje ještě poměrně mnoho porovnání, jejichž skóre leží v rozmezí 80–89 (celkem 15). Všechny tyto projekty ale vykazují vysokou míru podobnosti se vzorovým řešením, proto nelze vzájemnou pomoc mezi studenty ani potvrdit, ani vyvrátit. Je však jasné, že projekty nebyly vypracovány úplně samostatně. Určitě totiž byla velká část kódu převzata ze vzorového řešení.



Obr. 5.2: Přehled projektů v Javě

JavaFIle	AuthService.java	ClientAuthentication.java			71
	JavaClass	AuthService	ClientAuthentication		71
		JavaMethod	authenticate	authenticate	78
		JavaMethod	findPersonByEmail	NOT FOUND	0

Obr. 5.3: Detail porovnání dvou souborů



```
static {
    try (InputStream resourceStream = DataSourceConfig.class.getClassLoader().getResourceAsStream(APPLICATION_PROPERTIES)) {
        Properties properties = new Properties();
        properties.load(resourceStream);
        config.setJdbcUrl(properties.getProperty("datasource.url"));
        config.setUsername(properties.getProperty("datasource.username"));
        config.setPassword(properties.getProperty("datasource.password"));
        ds = new HikariDataSource(config);
    } catch (IOException | NullPointerException | IllegalArgumentException e) {
        logger.error("Configuration of the datasource was not successful.", e);
    } catch (Exception e) {
        logger.error("Could not connect to the database.", e);
    }
}
```

```
static {
    try (InputStream resourceStream = DataSourceConfig.class.getClassLoader().getResourceAsStream(APPLICATION_PROPERTIES)) {

        Properties properties = new Properties();
        properties.load(resourceStream);
        config.setJdbcUrl(properties.getProperty("datasource.url"));
        config.setUsername(properties.getProperty("datasource.username"));
        config.setPassword(properties.getProperty("datasource.password"));
        ds = new HikariDataSource(config);

    } catch (IOException | NullPointerException | IllegalArgumentException e) {
        logger.error("Configuration wasn´t successful.", e);
    } catch (Exception e) {
        logger.error("Could not connect to the database.", e);
    }
}
```

Obr. 5.4: Podobné bloky kódu ve studentských projektech

Další vyšší skóre opět kopírují podobnost se vzorovým projektem, nemá tedy cenu se jimi zabývat dopodrobna. Jedinou výjimkou je dvojice studentských projektů ohodnocena 69 body podobnosti ze 100. Tato dvojice totiž má opět nízká skóre při porovnání se vzorovým projektem (43 a 48 bodů), proto je vhodné zkontrolovat, zdali i tato dvojice studentů spolupracovala. A ve zdrojových souborech opravdu je vidět podobnost v těch souborech, na které test plagiátorství ukazuje. Příkladem je obrázek 5.5, kde je vidět velmi podobné a běžně nepoužívané formátování souborů.

5.2.2 Nízká skóre podobnosti

Díky výpočtu skóre podobnosti pomocí vážených hodnot neprodukuje porovnávací algoritmus falešně pozitivní hodnoty způsobené malými bloky kódu. To znamená, že při velmi odlišných projektech bude skóre nízké i přesto, že některé malé části v projektech byly označeny vysokým skóre. Vzhledem k implementaci porovnávání je totiž nenulová šance, že tyto malé bloky vůbec zkopírovány nebyly a nemá smysl jejich vysoké skóre propagovat pro celý projekt.

Příklad takového jevu je dvojice studentů, která byla obdržela skóre 2 ze 100 možných (což je nejmenší dosažená hodnota v celém ročníku). U těchto studentů označil algoritmus vysokou shodou dvojici tříd obsahující podobné metody (*gettery* a *settery*). Vzhledem k tomu, že je do výpočtu zahrnuta váha tohoto porovnání

```java
package org.but.app.services;

import org.but.app.api.DeviceBasicView;
import org.but.app.data.DeviceRepository;

import java.util.List;

public class DeviceService {

    private DeviceRepository deviceRepository;

    public DeviceService(DeviceRepository deviceRepository){this.deviceRepository = deviceRepository;}

    public List<DeviceBasicView> getAllDevices(){return deviceRepository.getAllDevices();}
}
```

```java
package org.but.app.services;

import org.but.app.api.DeviceBasicView;
import org.but.app.data.DeviceRepository;

import java.util.List;

public class DeviceServis {

    private DeviceRepository deviceRepository;

    public DeviceServis(DeviceRepository deviceRepository){this.deviceRepository = deviceRepository;}

    public List<DeviceBasicView> getAllDevices(){return deviceRepository.getAllDevices();}
}
```

Obr. 5.5: Podobné soubory ve studentských projektech

(která je oproti velikosti projektů velmi nízká), konečné skóre se tím téměř vůbec neposune. Výsledek tedy i tak zůstane nízký.

Při porovnání se vyskytlo šest studentů, jejichž nejlepší shoda nepřekročila úroveň 50 ze 100 možných. Nejnižší skóre, které bylo maximem shod daného žáka, byla hodnota 43. Dalších pět studentů pak dosáhlo nejvyšší shody do skóre 60. U nich lze předpokládat, že práci opravdu tvořili samostatně. Algoritmus totiž nezkoumá pouze znaky kopírování, ale logické podobnosti, které jsou vzhledem ke stejnému zadání projektů nevyhnutelné.

5.3 Projekty v jazyce Python

Vzhledem k tomu, že v tomto programovacím jazyce byly odevzdány pouze 2 projekty, bude tato případová studie mnohem kratší. Studenti odevzdávající projekty tohoto jazyka své úlohy nezkopírovali, přesto že v době odevzdávání ještě nebylo porovnávání v tomto jazyce implementováno. Výsledné skóre tohoto jednoho hodnocení je 37 ze 100 možných.

Struktura projektů nebyla podobná. Části kódu označeny jako potenciální shoda

byly podobné kvůli použití stejného grafického uživatelského prostředí oběma studenty (jednalo se o krátké metody, které definovaly vlastnosti oken). Většina kódu pak podobné znaky neměla.

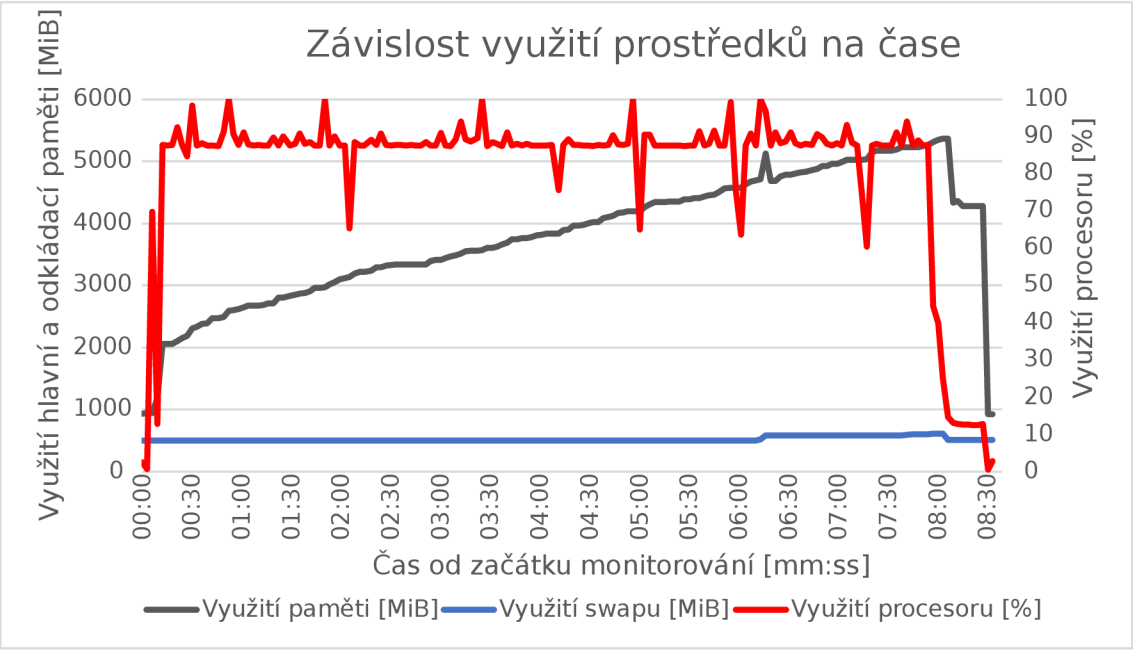
5.4 Popis běhu programu

Program byl spuštěn na laptopu s procesorem Intel i5 8300H Coffee Lake (4 samostatná jádra, 8 vláken, základní frekvence 2,3 GHz, maximální frekvence 4 GHz) s instalovanou pamětí 8 GB a swap pamětí taktéž 8 GB. Projekty byly staženy na pevný disk typu HDD. Operační systém použitý pro tento běh byl Manjaro Linux. V době běhu programu byly otevřená pouze dvě terminálová okna, v jednom byla spuštěna aplikace samotná, v druhém pak měřící skript, který každé 2 sekundy měřil a ukládal data o využití prostředků do souboru formátu `.csv`. Tento měřící skript byl spuštěn několik sekund před spuštěním aplikace a vypnut několik sekund po jejím ukončení.

Během testování byly naklonovány do zařízení všechny potřebné repozitáře, takže běh samotný mohl být spuštěn s použitím možnosti `--offline`. Žádné jiné možnosti využity nebyly. Při načítání projektů do paměti bylo do terminálu vypsáno 18 varovných hlášek. 10 z nich se týkalo nevalidních importů (zmíněné třídy byly v projektu v jiném balíčku nebo přejmenované, většinou šlo o zapomenuté názvy ze vzorového projektu, které ale odevzdaný projekt již neobsahoval) a 8 hlášek způsobily chyby AST kompilátoru. Z toho 7 hlášek bylo u jediného studenta z důvodu použití nové syntaxe nepodporované v Javě verze 8 a poslední hláška byla syntaktická chyba v kódu jiného studenta (v kódu chyběla ukončovací složená závorka ohraničující třídu). I přes tyto komplikace je načítání do paměti poměrně rychlý proces a všechny projekty byly načteny za přibližně 2,5 sekundy.

Plný běh trval asi 8,5 minuty, přičemž paralelní běh na vysoké využití procesoru trval přibližně 8 minut. Při porovnávání projektů je každé 2 sekundy do terminálu vypsán odhadovaný zbývající čas, který tato operace ještě zabere. Při každé aktualizaci je tento text přemazán, aby nezpůsoboval příliš dlouhý výpis. Jak lze vidět na grafu 5.6, využití paměti stoupá napřed skokově, při načtení projektů do paměti, a poté postupně při paralelním běhu generujícím výstup porovnání. Maximální využití paměti bylo přibližně 4,2 GiB, další 1 GiB byl spotřebován operačním systémem. První menší uvolnění paměti nastalo v čase kolem 8. minuty běhu programu, kde došlo k uzavření paralelizovaného prostředí, druhé uvolnění paměti nastalo až po zapsání výsledku do souboru `.xlsx` a ukončení aplikace. Odkládací paměť nebyla nijak výrazně využita.

Aplikace je velmi náročná na prostředky (zejména procesor a paměť), proto je vhodné ji na větších sadách projektů spouštět pouze se zavřenými ostatními okny. Na



Obr. 5.6: Graf využití prostředků

počítačích s menší velikostí paměti je pak vhodné vyzkoušet minimalistický operační systém, na kterém bude aplikace spuštěna.

5.5 Porovnání s aplikací JPlag

JPlag je terminálový detekční systém implementovaný v jazyce Java, založený na tokenech. Podporuje 12 různých programovacích jazyků (mezi nimi i Java a Python) a textové porovnávání. Výstup porovnání (soubor formátu `.zip`) je pak možné procházet ve webovém rozhraní. Tento systém je dostupný pod licencí GPL-3 a je možné ho získat přímo z repozitáře projektu na GitHubu. Vývoj v repozitáři započal 15. 3. 2015 a do projektu se celkem zapojilo 35 přispěvatelů (k 24. 4. 2023) [3].

Podpora jazyků spočívá, podobně jako u implementovaného řešení, na kompilátoru zdrojového kódu. Avšak oproti implementovanému řešení nemapuje výsledek tohoto načítání na různé datové typy, pouze rozhoduje, která část informace se propíše do objektu tokenu. Při porovnání pak hledá nejdelší shody tokenů [4].

5.5.1 Výhody JPlagu oproti implementované aplikaci

Nespornou výhodou JPlagu je rychlost. Při testování byly všechny studentské práce porovnány během 2 sekund, což je mnohem méně než u implementovaného řešení v rámci této práce. Samozřejmě i více podporovaných jazyků je nesporná výhoda,

avšak v předmětu BPC-BDS nemá tento fakt využití. Také zobrazování dílčích výsledků porovnání je přehlednější, protože přímo ve zdrojovém kódu zvýrazňuje, které části vykazují podobnosti, zatímco v implementovaném řešení je pouze vypsán název podobných částí a zdrojový kód musí být otevřen manuálně.

5.5.2 Výhody implementovaného řešení oproti JPlagu

Pro potřeby obhajob projektů je vhodnější zanést výsledek do tabulky, než do listu porovnávaných dvojic. Ve výstupu programu JPlag je list sice seřazen sestupně, ale pro potřeby obhajob je vhodné vždy při obhajobě vyhledat aktuálního studenta a ne nejvyšší celkové skóre, na kterém se student nepodílel.

Také není pro zobrazení porovnání potřeba žádné zvláštní aplikace, stačí běžný tabulkový editor (vyzkoušeny byly editory LibreOffice, OnlyOffice a Microsoft Office Excel). Dále oproti JPlagu není potřeba specifikovat typy projektů před započtením porovnávání (v implementovaném řešení je o projektech rozhodováno před načtením projektů, u JPlagu jsou ve výchozím stavu porovnávány pouze projekty v Javě).

Propojení prostředí na GitLabu s porovnávacím systémem zlehčuje práci pro učitele. Pokud by však učitel preferoval pouze stažení projektů za použití implementovaného řešení a poté porovnání JPlagem, stačí při spuštění kódu z této práce použít přepínač `-co` nebo `--clone-only`.

Také podpora vzorových řešení je vytvořená na míru pro potřeby předmětu BPC-BDS, avšak podobnou funkcionalitu lze simulovat i v aplikaci JPlag.

5.5.3 Podobné aspekty obou systémů

Obě řešení produkují velmi podobné číselné výsledky porovnání, to znamená, že jsou srovnatelně úspěšné při odhalování podobností v projektech. Oba systémy také vyžadují umístění projektů do lokální složky, ve které lze projekty porovnávat.

Závěr

Pro ztížení kopírování prací mezi studenty byl vytvořen nový automatizovaný způsob vytváření izolovaných prostředí k odevzdání projektů ve službě GitLab. Tento nový způsob odevzdávání byl nasazen již v akademickém roce 2022/2023 pro aktuální studenty kurzu BPC-BDS, ale může být využit i v libovolném předmětu, kde jsou odevzdávány softwarové projekty.

V rámci této práce byl také v jazyce Python implementován algoritmus pro detekci plagiátorství, který byl využit při hodnocení studentských projektů v předmětu BPC-BDS. Byl ale napsán způsobem, který umožňuje využití i v dalších předmětech, kde jsou odevzdávány projekty v jazyce Java nebo Python. Je schopný detekovat velkou škálu pokusů o maskování zkopírovaných částí projektů (je plně imunní vůči změně formátování, komentářů či identifikátorů a následuje volané funkce či metody v kódu pro nalezení přesunutých procedur v rámci projektu). Detekční algoritmus byl optimalizován a k výpočtu podobností je využito paralelní zpracování na procesoru počítače. Reprezentace výsledku je poté zajištěna tabulkovým souborem formátu `xlsx`. Algoritmus je také možné dále rozšiřovat pro další programovací jazyky při správné implementaci výstupu z kompilátorů abstraktních syntaktických stromů.

Při zkoumání výstupu z detektoru plagiátorství byly nalezeny u některých studentů známky plagiátorství. Tito studenti se při obhajobách svých projektů ke spolupráci přiznali.

Bylo také ověřeno, že algoritmus správně rozlišuje projekty, které si podobné nejsou. Objevilo se mnoho projektů, které podle skóre podobnosti byly nejspíše vypracovány bez pomoci spolužáků nebo vzorového řešení.

Tato práce byla také prezentována na studentské konferenci EEICT 2023, kde se v kategorii bakalářských prací na téma *Communication and Information Systems* umístila na 1. místě.

Literatura

1. GITHUB. *About Git*. San Francisco: Github Inc., c2023. Dostupné také z: https://docs.github.com/en/get-started/using-git/about-git.
2. ZAPOROZHETS, Dmytro; SIJBRANDIJ, Sytse. *GitLab: The DevSecOps Platform*. Ukrajina: GitLab Inc., c2023. Dostupné také z: https://about.gitlab.com/.
3. LANDHÄUSSER, Mathias. *JPlag*. San Francisco: Github Inc., c2023. Dostupné také z: https://github.com/jplag/JPlag.
4. SAĞLAM, Timur. *JPlag Wiki: Adding New Languages*. United States: Github Inc., c2023. Dostupné také z: https://github.com/jplag/JPlag/wiki/4.-Adding-New-Languages.
5. LUKASHENKO, Romans; GRAUDINA, Vita; GRUNDSPENKIS, Janis. Computer-Based Plagiarism Detection Methods and Tools: An Overview. In: *Proceedings of the 2007 International Conference on Computer Systems and Technologies*. New York, NY, USA: Association for Computing Machinery, 2007. CompSysTech '07. ISBN 9789549641509. Dostupné z DOI: 10.1145/1330598.1330642.
6. VYKOPAL, Jan; ŠVÁBENSKÝ, Valdemar; ŠEDA, Pavel; ČELEDA, Pavel. Preventing Cheating in Hands-on Lab Assignments. In: *Proceedings of the 53rd ACM Technical Symposium on Computer Science Education (SIGCSE '22)* [elektronická verze "online"]. New York, NY, USA: ACM, 2022, s. 78–84. ISBN 978-1-4503-9070-5. Dostupné z DOI: http://dx.doi.org/10.1145/3478431.3499420.
7. RIABOV, Vladimir. Teaching Online Computer-Science Courses in LMS and Cloud Environment. *International Journal of Quality Assurance in Engineering and Technology Education*. 2017, roč. 5, s. 12–41. Dostupné z DOI: 10.4018/IJQAETE.2016100102.
8. AIKEN, Alex. *Moss: A System for Detecting Software Similarity*. Stanford, California: Stanford University, 1985. Dostupné také z: https://theory.stanford.edu/~aiken/moss/.
9. WHALE, G. Identification of Program Similarity in Large Populations. *The Computer Journal*. 1990, roč. 33, č. 2, s. 140–146. ISSN 0010-4620. Dostupné z DOI: 10.1093/comjnl/33.2.140.

10. SONG, Hyun-Je; PARK, Seong-Bae; PARK, Se Young. Computation of Program Source Code Similarity by Composition of Parse Tree and Call Graph. *Mathematical Problems in Engineering*. 2015, roč. 2015, s. 429807. ISSN 1024-123X. Dostupné z DOI: 10.1155/2015/429807.

11. HAGE, Jurriaan; RADEMAKER, Peter; VUGT, Nikè. Plagiarism detection for Java: a tool comparison. In: *Computer Science Education Research Conference*. 2011, s. 33–46. Dostupné také z: https://www.researchgate.net/publication/262281635_Plagiarism_detection_for_Java_a_tool_comparison.

12. LANCASTER, Thomas; CULWIN, Fintan. Classifications of plagiarism detection engines. *Innovation in Teaching and Learning in Information and Computer Sciences*. 2005, roč. 4, č. 2, s. 1–16. Dostupné z DOI: 10.11120/ital.2005.04020006.

13. HAN, Lifang; CUI, Baojiang; ZHANG, Ru; LI, Zhongxian; WANG, Jianxin; HAO, Yongle. Type Redefinition Plagiarism Detection of Token-Based Comparison. In: *2010 International Conference on Multimedia Information Networking and Security*. 2010, s. 351–355. Dostupné z DOI: 10.1109/MINES.2010.80.

14. JI, Jeong-Hoon; WOO, Gyun; CHO, Hwan-Gue. A Plagiarism Detection Technique for Java Program Using Bytecode Analysis. In: *2008 Third International Conference on Convergence and Hybrid Information Technology*. 2008, sv. 1, s. 1092–1098. Dostupné z DOI: 10.1109/ICCIT.2008.267.

15. SCHÖNING, Uwe. Graph isomorphism is in the low hierarchy. *Journal of Computer and System Sciences*. 1988, roč. 37, č. 3, s. 312–323. ISSN 0022-0000. Dostupné z DOI: https://doi.org/10.1016/0022-0000(88)90010-4.

16. VALIENTE, Gabriel. Tree Isomorphism. In: *Algorithms on Trees and Graphs: With Python Code*. Cham: Springer International Publishing, 2021, s. 113–180. ISBN 978-3-030-81885-2. Dostupné z DOI: 10.1007/978-3-030-81885-2_4.

17. ENGELS, Steve; LAKSHMANAN, Vivek; CRAIG, Michelle. Plagiarism Detection Using Feature-Based Neural Networks. *SIGCSE Bull.* 2007, roč. 39, č. 1, 34–38. ISSN 0097-8418. Dostupné z DOI: 10.1145/1227504.1227324.

18. HUNT, Ethan; JANAMSETTY, Ritvik; KINARES, Chanana; KOH, Chanel; SANCHEZ, Alexis; ZHAN, Felix; OZDEMIR, Murat; WASEEM, Shabnam; YOLCU, Osman; DAHAL, Binay; ZHAN, Justin; GEWALI, Laxmi; OH, Paul. Machine Learning Models for Paraphrase Identification and its Applications on Plagiarism Detection. In: *2019 IEEE International Conference on Big Knowledge (ICBK)*. 2019, s. 97–104. Dostupné z DOI: 10.1109/ICBK.2019.00021.

19. EPPA, Akhil; MURALI, Anirudh. Source Code Plagiarism Detection: A Machine Intelligence Approach. In: *2022 IEEE Fourth International Conference on Advances in Electronics, Computers and Communications (ICAECC)*. 2022, s. 1–7. Dostupné z DOI: 10.1109/ICAECC54045.2022.9716671.

20. FURSIN, Grigori; MIRANDA, Cupertino; TEMAM, Olivier; NAMOLARU, Mircea; YOM-TOV, Elad; ZAKS, Ayal; MENDELSON, Bilha; BONILLA, Edwin; THOMSON, John; LEATHER, Hugh; WILLIAMS, Christopher K. I.; O'BOYLE, Michael; BARNARD, Phil; ASHTON, Elton; COURTOIS, Eric; BODIN, Francois. MILEPOST GCC: machine learning based research compiler. In: *Proceedings of the GCC Developers' Summit*. 2008.

21. KUHN, Thomas; THOMANN, Olivier. Abstract Syntax Tree. *eclipse.org*. 2006. Dostupné také z: https://www.eclipse.org/articles/Article-JavaCodeManipulation_AST/index.html.

22. PADIOLEAU, Yoann. Parsing C/C++ Code without Pre-processing. In: MOOR, Oege de; SCHWARTZBACH, Michael I. (ed.). *Compiler Construction*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, s. 109–125. ISBN 978-3-642-00722-4. Dostupné také z: https://doi.org/10.1007/978-3-642-00722-4_9.

23. PATEL, Suri. *Why you should move from centralized version control to distributed version control*. Ukrajina: GitLab Inc., c2023. Dostupné také z: https://about.gitlab.com/blog/2020/11/19/move-to-distributed-vcs/.

24. BARR, Joe. BitKeeper and Linux: The end of the road? - Linux.com. *Linux.com*. 2005. Dostupné také z: https://www.linux.com/news/bitkeeper-and-linux-end-road/.

25. BROWN, Zack. A Git Origin Story | Linux Journal. *Linuxjournal.com*. 2018. Dostupné také z: https://www.linuxjournal.com/content/git-origin-story.

26. BIRKINSHAW, Chris. *Hello Serverless, Goodbye Git Flow: Branching and Merging Strategies*. Netherlands: Merapar, c2023. Dostupné také z: https://articles.merapar.com/hello-serverless-goodbye-git-flow.

27. GITHUB. *Get the complete developer platform*. San Francisco, c2023. Dostupné také z: https://github.com/pricing.

28. GITHUB. *Getting started with the REST API*. San Francisco, c2023. Dostupné také z: https://docs.github.com/en/rest/guides/getting-started-with-the-rest-api.

29. GITHUB. *About teams*. San Francisco, c2023. Dostupné také z: https://docs.github.com/en/organizations/organizing-members-into-teams/about-teams.

30. SIJBRANDIJ, Sid. *Upcoming changes to user limits on Free tier of GitLab SaaS*. c2023. Dostupné také z: https://about.gitlab.com/blog/2022/03/24/efficient-free-tier/.

31. GITLAB. *API Docs*. c2023. Dostupné také z: https://docs.gitlab.com/ee/api/.

32. AZURE, Microsoft. *Pricing for Azure DevOps*. c2023. Dostupné také z: https://azure.microsoft.com/en-us/pricing/details/devops/azure-devops-services/.

33. ATLASSIAN. *Bitbucket – Plans and pricing*. c2023. Dostupné také z: https://www.atlassian.com/software/bitbucket/pricing?tab=cloud-tab.

34. SZYMUTKO, Marek; ŠEDA, Pavel. *Init Project Groups and Permissions*. Brno: GitLab Inc., c2023. Dostupné také z: https://gitlab.com/but-courses/bpc-bds/instructors-supporting-tools/init-project-groups-and-permissions.

35. SZYMUTKO, Marek. *BDS Similarity Check*. Brno: GitLab Inc., c2023. Dostupné také z: https://gitlab.com/but-courses/bpc-bds/instructors-supporting-tools/bds-similarity-check.

36. REITZ, Kenneth. *Requests: HTTP for Humans™*. United States: Read the Docs, 2010. Dostupné také z: https://requests.readthedocs.io/en/latest/.

37. MCNAMARA, John. *XlsxWriter*. San Francisco: GitHub Inc., c2023. Dostupné také z: https://github.com/jmcnamara/XlsxWriter.

38. SZYMUTKO, Marek; ŠEDA, Pavel. Plagiarism detection in software projects using abstract syntax trees [v tisku]. In: *Proceedings II of the 29th Conference STUDENT EEICT 2023*. Brno: Brno University of Technology, Faculty of Electrical Engineering a Communication, 2023. ISBN 978-80-214-6154-3. ISSN 2788-1334.

Seznam symbolů a zkratek

API	Rozhraní pro programování aplikací – Application Programming Interface
AST	Abstraktní syntaktický strom – Abstract Syntax Tree
BPC-BDS	Předmět na VUT FEKT – bezpečnost databázových systémů
CLI	Rozhraní v příkazové řádce – Command Line Interface
CSV	Formát souboru, ve kterém jsou hodnoty odděleny čárkami – Comma Separated Values
EOF	Speciální znak označující konec souboru – End of File
EOL	Speciální znak označující konec řádku – End of Line
JVM	Virtuální stroj Javy – Java Virtual Machine
Regex	Regulární výrazy, nástroj pro parsování textu – Regular Expressions

Seznam příloh

A Obsah elektronické přílohy

V elektronické příloze jsou přiloženy dva uzavřené celky – skripty v jazyce Bash pro správu studentského prostředí na platformě GitLab a program pro detekci plagiátorství napsaný v jazyce Python. Oba adresáře obsahují návod ke spuštění napsaný ve značkovacím jazyce Markdown.

Bash skripty byly testovány pomocí GNU Bash, verze 5.1.16(1). Python program byl testován s využitím jazyka Python verze 3.10.8.

```
/
├── GitLab ..... adresář se soubory pro správu GitLab prostředí
│   ├── .env ..... Soubor pro vyplnění údajů ke GitLabu
│   ├── .gitattributes
│   ├── .gitignore
│   ├── init_groups_and_permissions.sh ..... Skript pro jednotlivce
│   ├── init_team_project_groups_and_permissions.sh ..... Skript pro týmy
│   ├── lower_permissions.sh ..... Archivační skript
│   ├── README.md ..... Návod k použití skriptů
│   ├── template_team.csv ..... Šablona pro vytvoření souboru týmů
│   ├── template.csv ..... Šablona pro vytvoření souboru studentů
│   └── user.png
└── Detekce ..... Adresář se soubory pro detekci plagiátorství
    ├── detection ..... Implementovaný balíček pro detekci plagiátorství
    │   ├── __init__.py
    │   ├── abstract_scan.py
    │   ├── compare.py
    │   ├── definitions.py
    │   ├── java_scan.py
    │   ├── parallelization.py
    │   ├── project_type_decision.py
    │   ├── py_scan.py
    │   ├── thresholds.py
    │   └── utils.py
    ├── readme_images ..... Adresář pro obrázky použité v README.md
    │   └── hierarchy.png
    ├── .env ..... Soubor pro vyplnění údajů ke GitLabu
    ├── .gitattributes
    ├── .gitignore
    ├── dependencies.txt
    ├── LICENSE
    ├── main.py ..... Spustitelný soubor
    ├── Output.xlsx ..... Anonymizovaný výstup z porovnání studentských projektů
    ├── README.md ..... Návod k použití detekčního systému
    ├── sample_projects.txt ..... Příklad souboru s údaji pro stažení projektů
    └── sample_templates.txt ..... Příklad souboru s údaji pro stažení šablon
```