Katedra informatiky
Přírodovědecká fakulta
Univerzita Palackého v Olomouci

DIPLOMOVÁ PRÁCE

Správa verzí zdrojových kódů založená na syntaxi a sémantice jazyka

2023

Bc. Tomáš Krejčí

Vedoucí práce:
Mgr. Petr Krajča, Ph.D.

Studijní program: Aplikovaná informatika,
prezenční forma

Bibliografické údaje

Autor:	Bc. Tomáš Krejčí
Název práce:	Správa verzí zdrojových kódů založená na syntaxi a sémantice jazyka
Typ práce:	diplomová práce
Pracoviště:	Katedra informatiky, Přírodovědecká fakulta, Univerzita Palackého v Olomouci
Rok obhajoby:	2023
Studijní program:	Aplikovaná informatika, prezenční forma
Vedoucí práce:	Mgr. Petr Krajča, Ph.D.
Počet stran:	64
Přílohy:	elektronická data v úložišti katedry informatiky
Jazyk práce:	český

Bibliographic info

Author:	Bc. Tomáš Krejčí
Title:	Syntax and Semantics Aware Version Control System
Thesis type:	master thesis
Department:	Department of Computer Science, Faculty of Science, Palacký University Olomouc
Year of defense:	2023
Study program:	Applied Computer Science, full-time form
Supervisor:	Mgr. Petr Krajča, Ph.D.
Page count:	64
Supplements:	electronic data in the storage of department of computer science
Thesis language:	Czech

Anotace

V rámci této práce byl vytvořen systém pro správu zdrojových kódů (VCS), který využívá znalosti o konkrétním programovacím jazyce k řešení kolizí při slučování změn. Systém pracuje na úrovni abstraktního syntaktického stromu a je schopen odhalit význam konkrétních změn a tyto změny promítnout do zdrojových kódů. Implementace VCS je navržena tak, aby ji bylo možné použít s libovolným programovacím jazykem a již obsahuje podporu pro jazyk Java (do verze Java SE 17). Součástí práce je také srovnání s existujícími VCS.

Synopsis

As part of this thesis, was created version control system (VCS), which uses knowledge of a specific programming language to resolve merge conflicts. The system works at the abstract syntax tree level and is able to detect meaning of concrete changes and project these changes into the source codes. VCS implementation is designed to be used with any programming language and already includes support for the Java language (up to Java SE 17 version). The thesis also includes a comparison with existing VCS.

Klíčová slova: Systém pro správu zdrojových kódů; abstraktní syntaktický strom; porovnání stromů; kolize při slučování

Keywords: Version control system; abstract syntax tree; tree comparison; merge conflict

Děkuji, Mgr. Petru Krajčovi, Ph.D. za vedení mé práce a připomínky při vývoji.

Odevzdáním tohoto textu jeho autor/ka místopřísežně prohlašuje, že celou práci včetně příloh vypracoval/a samostatně a za použití pouze zdrojů citovaných v textu práce a uvedených v seznamu literatury.

Obsah

Seznam obrázků

Seznam tabulek

Seznam vět

Seznam zdrojových kódů

1 Úvod

Verzovací systémy jsou v dnešní době nepostradatelným nástrojem pro vývoj softwaru, který využívají celé týmy vývojářů i jednotlivci pracující na svých soukromých projektech. Umožňují provádět změny ve zdrojových kódech bez obavy ze ztráty původních dat, souběžný vývoj více vývojářů na stejném projektu, porovnávání změn mezi jednotlivými verzemi a mnoho dalšího.

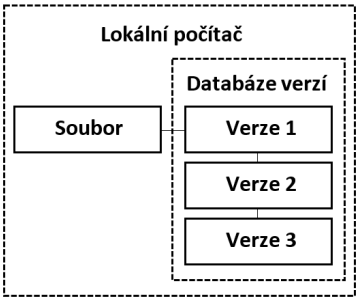
Pokud při vývoji v týmu provedou různí vývojáři úpravy ve stejných souborech, může při slučování jejich změn dojít ke kolizím. K tomu dochází v případě, že verzovací systém není sám schopen určit, které změny by se měly aplikovat. Toto rozhodnutí potom zůstává na vývojáři a v případě rozsáhlejších změn pro něj nemusí být jednoduché správně interpretovat jejich význam a samotné slučování tak může být poměrně náročné.

Z tohoto důvodu byl v rámci této práce navržen a vytvořen verzovací systém **Syntax VCS**, který pracuje na úrovni abstraktního syntaktického stromu a pro detekci změn mezi jednotlivými verzemi používá algoritmus pro porovnání **AST**. Proto dokáže interpretovat syntaktický význam provedených změn a pomoci tak vývojáři zdárně vyřešit kolize při slučování.

1.1 Systém pro správu zdrojových kódů

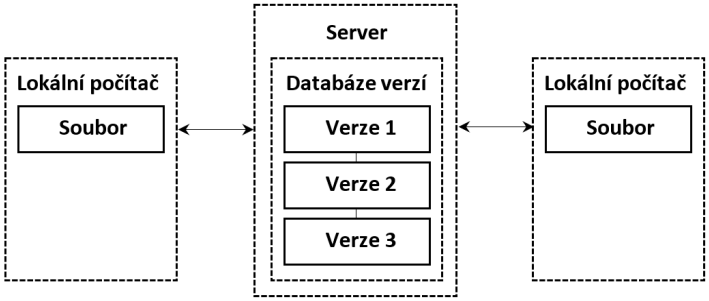
Systém pro správu zdrojových kódu [1], nebo také verzovací systém (**VCS** - **V**ersion **C**ontrol **S**ystem), je nástroj pro sledování a správu změn provedených v konkrétních souborech (tzv. verzování). Mimo jiné umožňuje obnovu dřívějších verzí sledovaných souborů, porovnávání změn mezi verzemi téhož souboru a také uchovává informace o provedení úprav (autor, čas, atd.). Obvykle se verzovací systém používá při vývoji softwaru pro zaznamenávání jednotlivých verzí zdrojových kódů aplikace, může ale pracovat obecně s libovolnými typy souborů. Verzovací systémy můžeme rozdělit na tři typy:

Local Version Control System (LVCS) – jedná se o jednoduchou databázi, která lokálně na disku uživatele uchovává všechny změny u sledovaných souborů (viz Obrázek 1). Z toho plynou jeho dvě hlavní nevýhody – spolupráce více lidí v týmu na stejném projektu je velmi složitá (téměř nemožná) a při smazání dat z databáze budou ztraceny i změny v souborech.



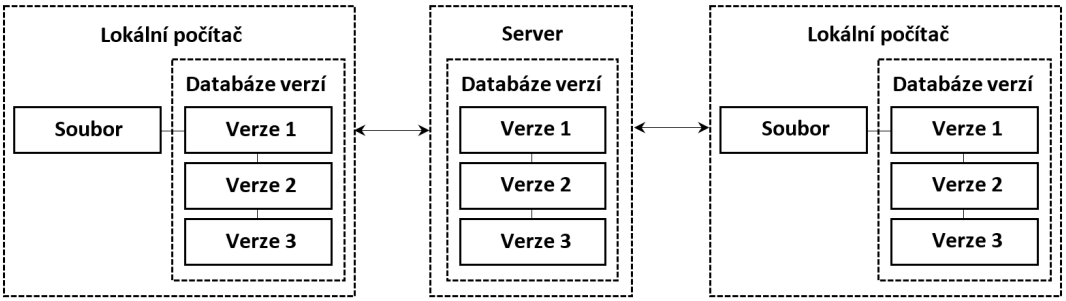
Obrázek 1: Local Version Control System [1]

Centralized Version Control System (CVCS) – v tomto případě se všechny změny ve verzovaných souborech uchovávají na centrálním úložišti (serveru), odkud si je pak uživatelé stahují (viz Obrázek 2). Výhodou tohoto řešení oproti **LVCS** je umožnění kooperace více uživatelů, stále ale zůstává hlavní nevýhoda centralizovaného systému – pokud dojde k porušení dat na centrálním serveru, zůstanou uživatelům jen lokálně stažené verze souborů a zbytek změn bude ztracen (v případě, že data ze serveru nejsou zálohována). Navíc v případě nedostupnosti serveru není možné stahovat a nahrávat nové verze, což může být pro uživatele limitující.



Obrázek 2: Centralized Version Control System [1]

Distributed Version Control System (DVCS) – v současnosti nejpoužívanější řešení, protože odstraňuje všechny nevýhody **LVCS** a **CVCS**. Celá historie verzí se uchovává na serveru (vzdáleném repozitáři), ale zároveň každý uživatel vlastní její kompletní kopii (viz Obrázek 3). V případě ztráty dat ze serveru je pak možné od libovolného uživatele celou historii obnovit. Navíc nejsou uživatelé vázáni na dostupnost serveru – v případě výpadku ukládají nové verze pouze lokálně a následně po zpřístupnění serveru tyto změny synchronizují. Tímto způsobem lze také pracovat s více repozitáři najednou a spolupracovat tak s různými týmy.



Obrázek 3: Distributed Version Control System [1]

V následujících kapitolách budou popsány a porovnány jedny z nejpoužívanějších open-source verzovacích systémů.

1.1.1 Git

Git [1] je **DVCS** vytvořený vývojáři Linuxu, v čele s Linusem Torvaldsem, pro vývoj linuxového jádra inspirovaný **BitKeeperem** (komerční **DVCS** původně použitý pro vývoj jádra). Hlavními požadavky na **VCS** byly rychlost, jednoduchý design, silná podpora nelineárního vývoje, plná distribuovatelnost a schopnost efektivně spravovat velké projekty (jako je linuxové jádro).

Narozdíl od většiny ostatních verzovacích systémů, které pracují s uloženými daty jako se skupinou souborů a seznamy změn těchto souborů v čase, zpracovává **Git** data ve formě sady tzv. snímků, které odpovídají stavu všech souborů v daném časovém okamžiku (nezměněné soubory jsou v rámci optimalizace nahrazeny referencí na původní, již uložený soubor).

Jeho přednostmi jsou vysoká rychlost (hlavně v případě velkých projektů s mnoha soubory), široká a velmi aktivní komunita (v případě problémů je jednoduché najít řešení nebo požádat o pomoc) a vysoká četnost použití (je aktuálně jedním z nepoužívanějších verzovacích systémů a většina vývojářů je s ním obeznámena). Naopak mezi jeho hlavní nevýhody patří značná komplikovanost a složitost příkazů (hlavně pro nové uživatele, kteří dosud neměli s podobným typem **VCS** žádné zkušenosti).

1.1.2 Mercurial

Mercurial [2] je **DVCS** vytvořený Mattem Mackallem jako alternativa k již zmiňovanému **BitKeeperu** (stejně jako **Git** - vývoj obou verzovacích systémů byl zahájen prakticky ve stejnou dobu), když došlo k jeho zpoplatnění.

Oproti **Gitu** je **Mercurial** zaměřen na celkovou jednoduchost, což může být v některých případech výhoda (pohodlnější používání pro uživatele a nižší náročnost na naučení), ale i nevýhoda (není tu rozdíl mezi lokálním a vzdáleným repozitářem, atd.).

1.1.3 Apache Subversion

Apache Subversion [3], dříve jen **Subversion** (**SVN** - **S**ub**V**ersio**N**), je **CVCS** vytvořený firmou CollabNet, Inc. jako nástupce **CVS**. Zachovává základní principy **CVS**, ale je robustnější, flexibilnější a práce s ním je pro uživatele podstatně snadnější.

Hlavní nevýhodou oproti **Gitu** a **Mercurialu** je centralizace (**CVCS**), která neumožňuje pracovat bez připojení ke vzdálenému repozitáři. Naopak jednoduchost použití je srovnatelná s **Mercurialem** (příkazy a jejich chování je výrazně podobné) a navíc obsahuje nativní podporu pro **WebDav** protokol.

1.1.4 Syntax VCS

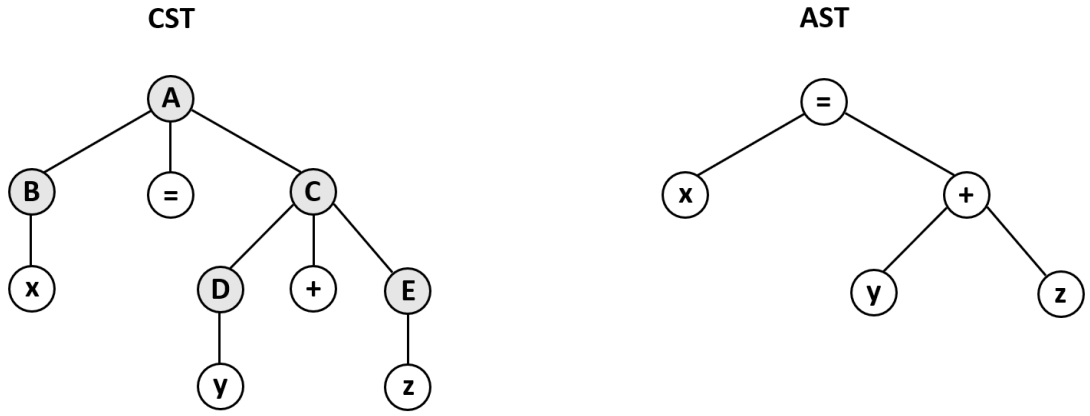
Syntax VCS je **LVCS** vytvořený v rámci této diplomové práce, který stejně jako **Git** (viz Kapitola 1.1.1) pracuje se změnami ve formě tzv. snímků. Tyto snímky jsou ale narozdíl od Gitu (a všech ostatních zmíněných verzovacích systémů) tvořeny **abstraktními syntaktickými stromy** (více v následující Kapitole 1.2), které odpovídají stavu repozitáře v daném okamžiku. Pro detekci změn mezi snímky potom využívá algoritmus pro porovnávání **AST** (více v následující Kapitole 2). To umožňuje lépe interpretovat provedené změny ve zdrojových kódech a poskytnout tak uživateli komplexnější informace při porovnávání změn mezi verzemi nebo řešení kolizí při slučování změn.

Tento přístup k porovnávání změn je hlavní výhodou **Syntax VCS** (ostatní zmíněné **VCS** porovnávaní změny na úrovni řádků souboru). Díky složitosti porovnávání (je nutné porovnávat uzly **AST** místo řádků souborů) a vytváření verzí (vytvoření snímku - **AST**) je ale verzovací systém poměrně pomalý a navíc je možné ho zatím používat pouze lokálně (**LVCS**).

1.2 Abstraktní syntaktický strom

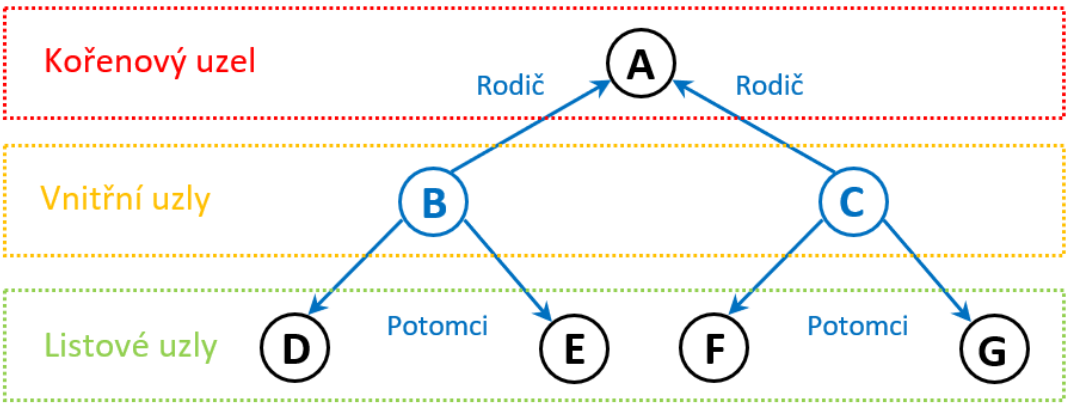
Abstraktní syntaktický strom (**AST** - **A**bstract **S**yntax **T**ree) [4], nebo také pouze syntaktický strom je stromová reprezentace syntaktické struktury zdrojového kódu v konkrétním programovacím jazyku. Typicky je generován syntaktickým analyzátorem v době překladu programu na základě **konkrétního syntaktického stromu**.

Konkrétní syntaktický strom (**CST** - **C**oncrete **S**yntax **T**ree), nebo také derivační strom, se generuje při syntaktické analýze a jeho uzly odpovídají symbolům gramatiky pro daný jazyk (terminální symboly a neterminální symboly). Narozdíl od **CST**, jsou uzly **AST** symboly zdrojového kódu bez dodatečných informací, které jsou pro syntaktickou analýzu nepodstatné (neterminální symboly a některé terminální symboly – např. závorky). Rozdíl mezi **CST** a **AST** je zobrazen na následujícím Obrázku 4 – oba stromy odpovídají výrazu $x = y + z$.



Obrázek 4: Rozdíl mezi CST a AST

Základní typy uzlů **AST** a vztahy mezi nimi jsou potom vyznačeny na Obrázku 5. Každý uzel může mít maximálně jeden nadřazený uzel (uzel ve vyšší vrstvě stromu, se kterým je spojen hranou), který označujeme jako **rodičovský uzel** nebo pouze **rodiče**. Zároveň může mít libovolný počet podřazených uzlů (uzlů v nižší vrstvě stromu, se kterými je spojen hranou), které označujeme jako **potomky**. Uzel v nejvyšší vrstvě stromu, který nemá žádný rodičovský uzel, označujeme jako **kořenový uzel**. Uzly v nejnižší vrstvě stromu, které nemají žádné potomky, potom označujeme jako **listové uzly**. Všechny ostatní uzly, které nejsou **kořenové** ani **listové**, označujeme jako **vnitřní uzly**.



Obrázek 5: Typy a vazby uzlů AST

1.3 Change Distilling algoritmus

Jedná se o algoritmus [5] pro porovnávání dvou **AST** založený na Chawatheho algoritmu pro detekci změn v hierarchicky strukturovaných datech (popsaný v **Change Detection in Hierarchically Structured Information** [6]). Jeho

autory jsou Beat Fluri, Michael Würsch, Martin Pinzger a Harald C. Gall a má sloužit pro detekci změn mezi různými verzemi téhož programu.

Princip algoritmu spočívá v tom, že se pro dva porovnávané stromy $AST1$ a $AST2$ nejdříve detekují dvojice podobných uzlů $\langle x, y \rangle$ (kde $x \in AST1 \wedge y \in AST2$) a následně se vytvoří minimální editační skript, který transformuje $AST1$ na $AST2$. Na základě Chawatheho algoritmu jsou pro nalezení odpovídající dvojic uzlů definovány dvě kritéria:

Podobnost listových uzlů – listové uzly x a y jsou podobné, pokud mají stejné typy (funkce l) a míra podobnosti (funkce sim) jejich hodnot (funkce v) je větší než daná hodnota f (viz Definice 1). Pro dosažení optimálních výsledků je hraniční míra podobnosti nastavena na $f = 0.6$.

Definice 1 (Podobnost listových uzlů [5])

$$match_1(x, y) = \begin{cases} true & \text{pokud } l(x) = l(y) \text{ a } sim(v(x), v(y)) \geq f \\ false & \text{jinak} \end{cases}$$

Podobnost vnitřních uzlů – vnitřní uzly x a y jsou podobné, pokud mají stejné typy (funkce l), míra podobnosti (funkce sim) jejich hodnot (funkce v) je větší než daná hodnota f a pro podstromy $AST(x)$ (kde je x kořenovým uzlem) a $AST(y)$ (kde je y kořenovým uzlem) platí, že podíl velikosti množiny všech dvojic podobných listových uzlů (funkce $common$) pro podstromy $AST(x)$ a $AST(y)$ a maxima (funkce max) z počtu listových uzlů podstromů $AST(x)$ a $AST(y)$ je větší než daná hodnota t (viz Definice 2). Pro dosažení optimálních výsledků jsou hraniční míry podobnosti nastaveny na $f = 0.6$ a $t = 0.6$.

Definice 2 (Podobnost vnitřních uzlů [5])

$$match_2(x, y) = \begin{cases} true & \text{pokud } l(x) = l(y) \text{ a } \frac{|common(x,y)|}{max(|x|,|y|)} \geq t \\ & \text{a } sim(v(x), v(y)) \geq f \\ false & \text{jinak} \end{cases}$$

Editační skript je potom posloupnost operací na **AST**, které mohou být pěti různých typů (viz Obrázek 6):

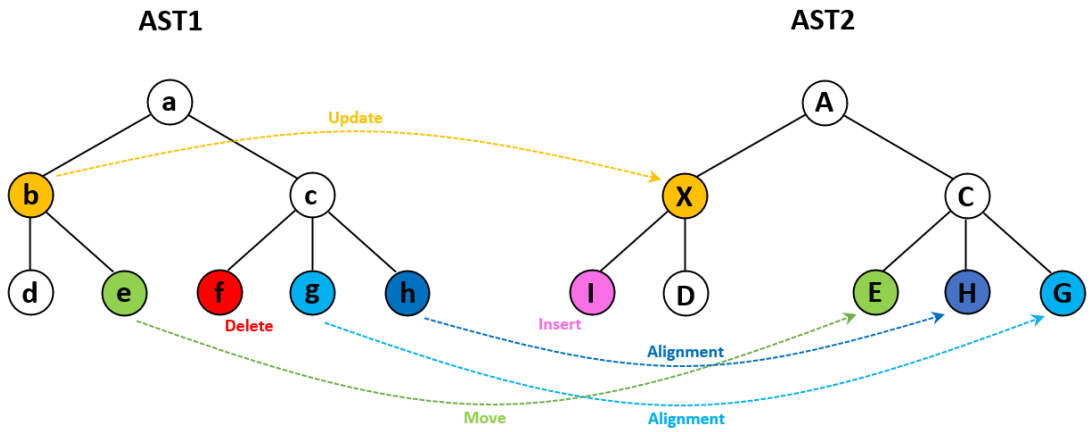
$Insert(x, y)$ – přidání nového listového uzlu x jako potomka daného uzlu y.

$Delete(x, y)$ – smazání daného uzlu x z jeho rodičovského uzlu y.

$Update(x, val)$ – změna hodnoty daného uzlu x na hodnotu val.

$Move(x, y)$ – přesun daného uzlu x z jednoho rodičovského uzlu do druhého y (změna rodičovského uzlu).

$Alignment(x, pos)$ – přesun daného uzlu x v rámci jednoho rodičovského uzlu na pozici pos (jedná se o operaci Move, u které se nezmění rodičovský uzel, ale pouze pořadí jeho potomků).



Obrázek 6: Typy operací nad AST

Při jeho generování se prochází uzly $AST2$ do šířky a postupně se přidávají editační operace, jejichž postupnou aplikací na uzly $AST1$ vznikne strom ekvivalentní k $AST2$. Algoritmus vychází z dvojice uzlů $\langle x, y \rangle$, kde x je kořenovým uzlem $AST1$ a y je kořenovým uzlem $AST2$ a rekurzivně prochází všechny potomky $y_1, y_2, ..., y_n$ uzlu y následujícím způsobem:

- Pokud pro daného potomka y_x z $AST2$ neexistuje dvojice podobných uzlů $\langle x_x, y_x \rangle$, pak dojde k přidání operace $Insert(x, y_x)$.
- Jinak mohou nastat 2 situace:
  - Pokud rodičovským uzlem uzlu x_x není uzel uzlu x, pak dojde k přidání operace $Move(x_x, x)$, následně mohou nastat 2 situace:
    - Pokud není hodnota uzlu x_x totožná s hodnotou uzlu y_x, pak dojde k přidání operace $Update(x_x, val)$, kde val je hodnota uzlu y_x.
    - Jinak se nepřidává žádná další operace.
  - Jinak mohou nastat 3 situace:
    - Pokud pozice uzlu x_x je jiná, než pozice uzlu x_x, pak dojde k přidání operace $Alignment(x_x, pos)$, kde pos je pozice uzlu x_x a následně mohou nastat 2 situace:

- Pokud není hodnota uzlu x_x totožná s hodnotou uzlu y_x, pak dojde k přidání operace $Update(x_x, val)$, kde val je hodnota uzlu y_x.
- Jinak se nepřidává žádná další operace.

- Pokud není hodnota uzlu x_x totožná s hodnotou uzlu y_x, pak dojde k přidání operace $Update(x_x, val)$, kde val je hodnota uzlu y_x.
- Jinak se nepřidává žádná další operace.

Všechny operace jsou při přidávání do editačního skriptu rovnou aplikovány. Nakonec, po průchodu všemi uzly $AST2$, se rekurzivně prochází všichni potomci $x_1, x_2, ..., x_n$ kořenového uzlu x následujícím způsobem:

- Pokud pro daného potomka x_x z $AST1$ neexistuje dvojice podobných uzlů $\langle x_x, y_x \rangle$, pak dojde k přidání operace $Delete(x, x_x)$.

- Jinak se nepřidává žádná další operace.

2 Algoritmy

2.1 Algoritmus pro porovnávání AST

Syntax VCS používá pro porovnání dvou **AST** algoritmus (viz Algoritmus 1), založený na **Change Distilling** algoritmu pro porovnávání **AST** popsaném v Kapitole 1.3. Pro účely verzovacího systému bylo nutné provést několik úprav, které budou popsány na následujících řádcích.

2.1.1 Algoritmus pro detekci podobných uzlů

Předně bylo nutné původní algoritmus podstatně zrychlit, jelikož verzovací systém běžně pracuje s velkým množstvím zdrojových kódů a porovnání výsledných **AST** by trvalo příliš dlouho. Toho bylo docíleno úpravou detekce podobných uzlů v **AST**. Výpočet podobnosti (viz Definice 3) je pro všechny uzly **AST** stejný – nerozlišuje se mezi vnitřními (viz Definice 2) a listovými uzly (viz Definice 1). Navíc se výrazně zjednodušil – oproti původnímu algoritmu se nepočítá míra podobnosti hodnot uzlů (funkce `sim`), ale pouze se zjišťuje, zda jsou hodnoty totožné (viz Definice 3). Také se nebere v potaz počet podobných listových uzlů – místo toho se zohledňuje míra podobnosti pro všechny potomky daného uzlu `x` (označeno x_c) a `y` (označeno y_c).

Definice 3 (Míra podobnosti uzlů)

$$matchRatio(x,y) = \begin{cases} 0 & \text{pokud } l(x) \neq l(y) \\ 0.5 + \sum_{i \in x_c} \max_{j \in y_c} matchRatio(i,j) & \text{pokud } v(x) \neq v(y) \\ 1 + \sum_{i \in x_c} \max_{j \in y_c} matchRatio(i,j) & jinak \end{cases}$$

Stejně jako v původním algoritmu (viz Kapitola 1.3) se hledá dvojice uzlů, která si je nejvíce podobná (pro daný první uzel dvojice z $AST1$ neexistuje žádný jiný uzel
z $AST2$, který by měl vyšší míru podobnosti (dle Definice 3), než druhý uzel dvojice z $AST2$. Toho je v algoritmu docíleno seřazením výsledných dvojic podle míry podobnosti a jejich postupným přidáváním do množiny výsledných dvojic, dokud nejsou pokryti všichni potomci daného uzlu (viz řádky 24-36 v pseudokódu Algoritmu 1).

Algoritmus 1 Funkce compare

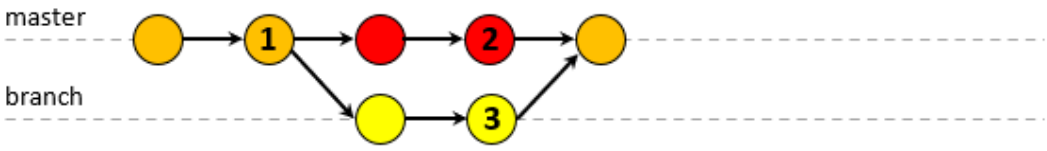
```
1: procedure COMPARE(node1, node2)
2:     result ← NEW MATCHRESULT()
3:     result.ratio ← 0
4:     result.matchSet ← []
5:     if node1.type ≠ node2.type then return result
6:     end if
7:     if node1.value = node2.value then
8:         result.ratio ← result.ratio + 1.5
9:     else
10:        result.ratio ← result.ratio + 0.5
11:    end if
12:    matchedResults ← []
13:    for all child1 ∈ node1.children do
14:        for all child2 ∈ node2.children do
15:            matchResult ← NEW MATCHRESULT()
16:            matchResult.node1 ← child1
17:            matchResult.node2 ← child2
18:            matchNodes ← COMPARE(child1,child2)
19:            matchResult.ratio ← matchNodes.ratio
20:            matchResult.matchSet ← matchNodes.matchSet
21:            add matchResult to matchedResults
22:        end for
23:    end for
24:    nodes1 ← node1.children
25:    nodes2 ← node2.children
26:    sort matchedResults by ratio in descending order
27:    for all matched ∈ matchedResults do
28:        if nodes1 and nodes2 are empty then return result
29:        end if
30:        if matched.node1 ∈ nodes1 and matched.node2 ∈ nodes2 then
31:            remove matched.node1 from nodes1
32:            remove matched.node2 from nodes2
33:            result.ratio ← result.ratio + matched.ratio
34:            add matched.matchSet to result.matchSet
35:        end if
36:    end for
37:    return result
38: end procedure
```

2.1.2 Generování editačního skriptu

Dalším problémem původního algoritmu (viz Kapitola 1.3) bylo upřednostňování *Move* operace i v případech, kdy tato operace z hlediska sémantiky kódu nemá žádný význam (např. v případě smazání modifikátoru u jedné metody a přidání stejného modifikátoru u jiné metody). Použití operace *Alignment* zůstalo zachováno, jelikož změna pořadí potomků uzlu obvykle význam má (např. změna pořadí parametrů metody, reorganizace kódu v rámci třídy, apod.), ale operace *Move* byla výrazně omezena a ve většině případů nahrazena operacemi *Insert* a *Delete*. Díky této úpravě je výsledný algoritmus rychlejší a nedochází k nesprávným přesunům uzlů. Nevýhodou této úpravy je částečné potlačení schopnosti algoritmu správně detekovat některé přesuny uzlů (např. pro změny na úrovni souborů a složek v repozitáři).

2.2 Algoritmus pro slučování změn

Pro potřeby slučování větví v rámci **Syntax VCS** bylo nutné implementovat algoritmus, který na základě tří vstupních **AST** vytvoří jeden výsledný **AST** obsahující změny z obou větví a navíc určí, zda byly konkrétní změny provedeny v obou větvích nebo jen v jedné z nich. První **AST** odpovídá nejbližšímu společnému commitu (commit je označení pro tzv. snímek, viz Kapitola 1.1.4) pro obě větve, další dva pak posledním commitům ve slučovaných větvích viz Obrázek 7.



Obrázek 7: Commity použité při slučování větví

Algoritmus 2 provede dvakrát porovnání **AST** pomocí algoritmu popsaném v předchozí Kapitole 2.1 – nejdříve porovná **AST** společného commitu (označený na Obrázku 7 jako 1) a posledního commitu v aktuální větvi (označený na Obrázku 7 jako 2), následně pak porovná **AST** společného commitu a posledního commitu v slučované větvi (označený na Obrázku 7 jako 3). Potom se postupně aplikují na **AST** společného commitu výsledné operace z prvního porovnání a následně operace z druhého porovnání. Přitom se u každého uzlu zaznamenává jeho status – číselná hodnota, která se používá pro určení původu změny. Nakonec se provede označení konfliktních změn (pokud nějaké v **AST** existují) a úprava **AST** do finální podoby.

Algoritmus 2 Funkce merge

```
1: procedure MERGE(ast1, ast2, ast3)
2:     changes1 ← COMPARE(ast1, ast2)          ▷ porovnání AST1 a AST2
3:     changes2 ← COMPARE(ast1, ast3)          ▷ porovnání AST1 a AST3
4:     MERGE1(ast1.root, changes1)             ▷ aplikace prvních změn na AST1
5:     MERGE2(ast1.root, changes2)             ▷ aplikace druhých změn na AST1
6:     COMPLETEMERGE(ast1)                     ▷ označení konfliktů a finální úprava AST1
7: end procedure
```

2.2.1 Aplikace operací z prvního porovnání

Proces aplikace operací z prvního porovnání a výpočet statusu uzlů je řešen rekurzivním Algoritmem 3. Po aplikaci operací na potomky aktuálního uzlu **AST** se vypočítá pro všechny potomky jejich status na základě původního statusu uzlu a operace, která na něj byla aplikována (výsledné statusy uzlů zobrazuje Tabulka 1). Na začátku mají všechny uzly **AST** status 0. Odstraněné uzly získají kladný status (pro operace *Delete* a *Move* 1, pro *Update* 2), přidané uzly záporný status (pro operace *Insert* a *Move* -5, pro *Update* -3) a systémové uzly označující začátek a konec změn jsou vytvořeny se statusem 6 (pro operace *Delete* a *Move*) nebo -6 (pro operace *Insert*, *Update* a *Move*).

Následně se metoda rekurzivně aplikuje na všechny potomky aktuálního uzlu, pro které nebyla splněna mezní podmínka (viz řádek 6 Algoritmu 3) – uzly v podstromu daného potomka nemohou obsahovat žádné změny, pokud má potomek status 1 (byl smazán) nebo má status -3 (jedná se o nově přidaný uzel s původní hodnotou uzlu v rámci operace *Update*), a zároveň se nejedná o uzel reprezentující soubor v repozitáři.

Tabulka 1: Statusy uzlů po aplikaci operací z prvního porovnání

	-6	-5	-4	-3	-2	-1	0	1	2	3	4	5	6
Insert							-5						
Delete							1						
Update (původní)							2						
Update (nový)							-3						
Move (přidaný)							-5						
Move (odstraněný)							1						
Systémový (přidaný)							-6						
Systémový (odstraněný)							6						

Algoritmus 3 Funkce merge1

```
1: procedure MERGE1(node, changes)
2:     for all change ∈ changes do
3:         EXECUTE(node, change)                    ▷ aplikace operace na daný uzel
4:     end for
5:     for all child ∈ node.children do
6:         if child.status ≠ 1 and (child.status ≠ 3 or child.type is File) then
7:             MERGE1(child, change)
8:         end if
9:     end for
10: end procedure
```

2.2.2 Aplikace operací z druhého porovnání

Algoritmus 4 pro aplikaci operací z druhého porovnání probíhá obdobným způsobem jako v předchozí Kapitole 2.2.1 s tím rozdílem, že se komplikuje výpočet statusu uzlů (výsledné statusy uzlů zobrazuje Tabulka 2) a mění mezní podmínka (viz řádek 6 Algoritmu 4) pro rekurzivní volání metody. Výpočet statusu uzlu v tomto případě závisí na původním statusu (případně na statusu ostatních uzlů) a v některých případech (operace *Insert*, *Update* a *Move*) může dojít i ke změně stavu ostatních uzlů. Po aplikaci všech operací je možné dle statusu uzlu zjistit, zda byla změna provedena pouze v aktuální větvi (uzel má kladný status), pouze v slučované větvi (uzel má záporný status) nebo v obou větvích zároveň (uzel byl smazán nebo označen statusem 0).

Rekurzivní volání se potom opět neprovádí pro potomky, které už nemohou obsahovat další změny – pokud má potomek status menší než -2 (jedná se o uzel na který byla aplikována operace pouze v Algoritmu 3 a v Algoritmu 4 žádná operace neproběhla) nebo má status 3 (jedná se o nově přidaný uzel s původní hodnotou uzlu v rámci operace *Update*) nebo 4 (jedná se o původní uzel s rozdílnou hodnotou uzlu, na který byla aplikována operace *Update*), a zároveň se nejedná o uzel reprezentující soubor v repozitáři.

Tabulka 2: Statusy uzlů po aplikaci operací z druhého porovnání

	-6	-5	-4	-3	-2	-1	0	1	2	3	4	5	6
Insert		0	5	5	5	5	5	5	5	5	5		
Delete		-1	-1	-1	-1		-1		-1	-1	-1	-1	
Update (původní)		-2		-2		-2		-2	0/4/-4	-2		-2	
Update (nový)		3	3	3	3	3	3	3			3	3	
Move (přidaný)		5	5	5	5	5	5	5	5	5	5		
Move (odstraněný)		-1	-1	-1	-1		-1		-1	-1	-1	-1	
Systémový (přidaný)							6						
Systémový (odstraněný)							-6						

Algoritmus 4 Metoda merge2

```
1: procedure MERGE2(node, changes)
2:     for all change ∈ changes do
3:         EXECUTE(node, change)                    ▷ aplikace operace na daný uzel
4:     end for
5:     for all child ∈ node.children do
6:         if child.status ≥ −2 and (child.status ≠ 3 or child.status ≠ 4 or
   child.type is File) then
7:             MERGE1(child, change)
8:         end if
9:     end for
10: end procedure
```

2.2.3 Dokončení slučování větví

Posledním krokem algoritmu je finální úprava výsledného **AST**, který nyní obsahuje všechny změny z obou větví dohromady. Pokud je zapnuto automatické slučování (více v Kapitole 3.4), budou vymazány všechny uzly se statusy -1 nebo 1 (byly smazány pouze v jedné větvi) a všechny uzly se statusy -5 nebo 5 změní svůj status na 0 (byly přidány pouze v jedné větvi). Tímto krokem budou všechny nekolidující změny označeny jako provedené – v případě, že **AST** nebude obsahovat žádné jiné (kolidující) změny bude merge rovnou proveden a uživatel nebude muset slučování interaktivně potvrzovat.

Následně dojde k označení zbylých změn na základě jejich statusu tak, že bude možné rozlišit, ze které větve bloky změn pocházejí. Pokud v tomto kroku zůstává v **AST** alespoň jedna takováto kolidující změna, nemůže být slučování větví dokončeno a uživatel musí interaktivně určit, které změny se mají provést. Proto se pro podstromy **AST** odpovídající souborům s kolidujícími změnami vygenerují merge soubory, které obsahují příslušný **AST** v textové podobě s označenými bloky změn. Po vyřešení konfliktů uživatelem se výsledné podstromy dosadí na své původní místo v **AST**, čímž je slučování větví dokončeno.

3 Implementace

3.1 Použité technologie

Aplikace byla vyvíjena ve vývojovém prostředí **IntelliJ IDEA** [7] ve verzi Community Edition, které obsahuje podporu pro vývoj aplikací v jazycích Java, Kotlin, Groovy a Scala. Pro vývoj samotné aplikace byl použit jazyk Java ve verzi 11 a čtyři knihovny, které budou popsány v následujících kapitolách.

3.1.1 Knihovna Annotations

Knihovna **Annotations** [8] obsahuje pomocné anotace, pomocí kterých lze specifikovat metadata o třídách, metodách, jejich parametrech, apod. Vývojové prostředí **IntelliJ IDEA** [7] je potom na základě těchto metadat schopné provádět pokročilejší analýzu kódu. V projektu jsou použity hlavně anotace `@Nullable` a `@NotNull` pro kontrolu nullability parametrů.

3.1.2 Knihovna ANTLR v4

ANTLR (**AN**other **T**ool for **L**anguage **R**ecognition) [9] ve verzi 4 je generátor pro vytváření vlastních překladačů (parserů) pro libovolný programovací jazyk na základě **LL(*)** gramatiky. Při parsování zdrojového kódu výsledným parserem je potom buďto vyvolána výjimka (v případě nevalidního zdrojového kódu) nebo je vytvořena odpovídající **AST** struktura (v případě validního zdrojového kódu). Pro účely aplikace byl upraven popis gramatiky pro jazyk Java, který je k dispozici přímo v rámci **ANTLR** repozitáře a vygenerovaný překladač byl potom použit jako základ parseru pro jazyk Java.

3.1.3 Knihovna Picocli

Picocli [10] je knihovna, nebo spíše framework, pro jednoduchou tvorbu konzolových aplikací. Poskytuje nástroje pro definici jednotlivých příkazů včetně přepínačů a parametrů, včetně kontroly jejich validity (omezení na počet parametrů, typ, kombinace přepínačů, apod.). Také zajišťuje výpis nápovědy pro všechny příkazy pomocí příslušného přepínače a obstarává zpracování chybových hlášek v aplikaci.

3.1.4 Knihovna Gson

Knihovna **Gson** [11] umožňuje serializaci Java objektů do **JSON** formátu, resp. deserializaci z **JSON** formátu do Java objektu. Také umožňuje definovat vlastní postup serializace, resp. deserializace pro komplexní objekty a kolekce. Toho se v aplikaci využívá při ukládání, resp. čtení souborů repozitáře, při zvoleném **JSON** formátu.

3.2 Struktura aplikace

Zdrojové kódy aplikace jsou rozděleny do několika balíčků podle zaměření. Jejich obsah a účel bude stručně popsán v následujících kapitolách této sekce.

3.2.1 Balíček ast

Balíček `ast` obsahuje třídy tvořící strukturu **AST**, která se následně používá napříč celou aplikací. Třída `AbstractSyntaxNode` představuje uzel stromu a obsahuje základní operace pro nastavení parametrů uzlu a práci s jeho potomky a rodičovským uzlem. Třída `AbstractSyntaxTree` potom představuje celý strom a obsahuje operace pro nastavení kořenového uzlu a kopírování stromové struktury.

3.2.2 Balíček astparser

Balíček `astparser` obsahuje třídy a rozhraní nutné pro vytváření parserů. Rozhraní `IASTParser` obsahuje metodu `parse` pro převod zdrojového kódu na **AST**. Základní třídou pro tvorbu parserů je potom třída `ASTParserBase`, která toto rozhraní implementuje a poskytuje navíc další pomocné metody. Statická třída `NodeTypes` potom obsahuje konstanty všech základní typů uzlů **AST** v repozitáři.

3.2.3 Balíček common

V balíčku `common` se nachází pomocné třídy, které s aplikací přímo nesouvisí, ale jsou nutné pro její běh. Logování v aplikaci zajišťuje třída `Logger` na základě konfigurace aplikace, která se načítá a spravuje pomocí třídy `Configuration`. V aplikaci se také využívá princip **Dependency Injection** (**DI**), který implementuje třída `DependencyInjection`.

3.2.4 Balíček comparator

Balíček `comparator` obsahuje třídy a rozhraní nutné pro vytváření comparatorů. Rozhraní `IComparator` obsahuje mimo jiné metodu `compare` pro interpretaci změn provedených v **AST**. Jako základní třída pro tvorbu comparatorů potom slouží třída `ComparatorBase`, která toto rozhraní implementuje a poskytuje navíc další pomocné metody. Balíček také obsahuje třídy `RepositoryComparator` a `RepositoryComparatorBase`, které tvoří comparator pro interpretaci změn v adresářích a složkách repozitáře. Dále jsou tu definovány třídy pro všechny operace nad **AST** (`InsertOperation`, `UpdateOperation`, `DeleteOperation` a `MoveOperation`).

3.2.5 Balíček lang

V balíčku `lang` se nachází parsery a comparatory pro všechny podporované jazyky (při přidávání nového jazyka se předpokládá zařazení příslušných tříd do tohoto balíčku). Výchozí parser a comparator pro všechny textové soubory a zdrojové kódy nepodporovaných jazyků se nachází v balíčku `text`. Druhý balíček `java` potom obsahuje parser a comparator pro zdrojové kódy v jazyku Java (viz Příloha B).

3.2.6 Balíček main

Balíček `main` obsahuje pouze třídu `Main` s metodou `main`, která kromě spuštění konzolové aplikace zajišťuje i registraci všech služeb do **Dependency Injection**.

3.2.7 Balíček repository

V balíčku `repository` se nachází třídy, které zajišťují základní operace (vytváření, úprava, serializace, deserializace, atd.) nad interními soubory repozitáře. Jedná se o třídy `Branch`, `Commit`, `Config`, `Head`, `Merge` a `Stage`. Specifická je potom třída `Ignore`, která obstarává přístup k uživatelem vytvořenému `.ignore` souboru, pokud v repozitáři existuje.

3.2.8 Balíček serializers

V balíčku `serializers` se nachází třídy zajišťující serializaci a deserializaci všech interních souborů repozitáře a vytvořených **AST**. Implementována je serializace a deserializace do **JSON** formátu a binárního formátu, který je použit ve výchozí konfiguraci. V případě potřeby je možné doimplementovat podporu pro libovolný další formát.

3.2.9 Balíček services

Všechny služby, které se používají napříč celou aplikací a jsou registrovány do **Dependency Injection** se nachází v balíčku `services`. Každá z nich slouží k jinému účelu:

`ASTCompareService` – implementuje algoritmus pro porovnávání dvou **AST** popsaný v Kapitole 2.1.

`ASTService` – zajišťuje všechny operace nad **AST** – vytvoření **AST** ze zdrojových kódů pomocí parserů, přidávání a odebírání uzlů odpovídajících souborům nebo adresářům v repozitáři, atd.

`CompareService` – umožňuje detekovat a interpretovat změny v **AST** na základě porovnání stromů pomocí `ASTCompareService` a následné interpretace změn pomocí comparatorů.

`MergeService` – implementuje algoritmus pro slučování změn v repozitáři popsaný v Kapitole 2.2.

`RepositoryService` – zajišťuje všechny operace nad repozitářem – vytvoření a mazání repozitáře, práci s jeho interními soubory a adresáři, čtení `.ignore` souboru, atd.

3.2.10 Balíček vcs

Balíček `vcs` obsahuje implementace všech příkazů verzovacího systému používané knihovnou picocli (viz Kapitola 3.1.3). Každá třída odpovídá jednomu příkazu, definuje jeho chování, parametry a případně omezení pro jeho použití. Třída `CommandBase` potom slouží jako základní třída, ze které všechny tyto třídy dědí. Zpracování výjimek a vytvoření samotné konzolové aplikace zajišťuje třída `Program`.

3.3 Podpora programovacích jazyků

Syntax VCS obsahuje v základu podporu pro zdrojové kódy v jazyku Java, je ale zároveň navržen tak, aby ho bylo možné jednoduše doplnit o podporu pro libovolný další programovací jazyk. Pro přidání dalšího jazyka je nutné doimplementovat pouze parser a comparator pro daný jazyk a upravit konfigurační soubor (viz Zdrojový kód 3.4) dle instrukcí v následujících sekcích.

3.3.1 Parser

Parser je třída implementující rozhraní `IASTParser` (detaily v Kapitole 3.2.2), která zajišťuje převod zdrojového kódu v daném programovacím jazyce na jeho reprezentaci v podobě abstraktního syntaktického stromu (více v Kapitole 1.2). Výsledný strom musí splňovat následující kritéria:

- Každý uzel stromu má 3 textové parametry:

  Type – povinný parametr, odpovídá typu uzlu (typy uzlů pro jazyk Java jsou uvedeny v Tabulce 3).

  Value – nepovinný parametr, odpovídá hodnotě uzlu (typicky název proměnné, hodnota literálu, apod.).

  Code – povinný parametr (nepovinný pouze pro kořenový uzel), obsahuje část zdrojového kódu odpovídající danému uzlu.

  Pozn. V tomto textu je pro přehlednost uzel zapisován ve tvaru [<Type>, <Value>, <Code>], parametry bez hodnoty jsou vynechány.

- Code každého uzlu obsahuje placeholdery (jako placeholder slouží konstanta `IASTService.CODE_PLACEHOLDER`) pro kódy potomků daného uzlu tak, že při průchodu stromem do hloubky, kdy dojde k nahrazení placeholderů příslušnými kódy bude výsledkem původní zdrojový kód.

- Kořenovým uzlem je uzel [Content], který odpovídá obsahu souboru se zdrojovým kódem.

Rozhraní `IASTParser` obsahuje pouze jednu metodu, kterou je třeba implementovat:

parse – převádí zdrojový kód (parametr `fileContent` typu `String`) na **AST** a vrací kořenový uzel stromu (objekt `AbstractSyntaxNode` reprezentující uzel [Content]).

Je důrazně doporučeno rozšířit třídu `ASTParserBase`, která zajišťuje korektní formátování kódu uzlů **AST** a nahrazení odpovídajících částí kódu placeholderem. Pro jazyky podobné Javě lze použít třídu `JavaASTParserBase` nebo přímo třídu `JavaASTParser` a doimplementovat pouze specifické konstrukty daného jazyka.

3.3.2 Comparator

Comparator je třída implementující rozhraní `IComparator` (detaily v Kapitole 3.2.4), která zajišťuje interpretaci změn provedených v **AST** (více v Kapitole 2.1).

Rozhraní `IComparator` obsahuje pouze čtyři metody, které je třeba implementovat:

compare – interpretuje operaci provedenou nad **AST** (parametr `operation` typu `EditOperation`). Pokud toho comparator není schopen (interpretace této změny není zatím implementována, operace je provedena s nevalidními argumenty, apod.), označí ji jako **UnrecognizedChange**.

getTextChanges – vrací výsledky interpretace všech změn, které byly rozpoznány metodou `compare` v textové podobě.

getUnrecognizedChanges – vrací všechny operace nad **AST**, které nebyly rozpoznány metodou `compare` (označeny jako **UnrecognizedChange**).

completeAnalysis – metoda, která se spouští pouze na konci porovnání operací nad **AST**. Její implementace může být prázdná, pokud není třeba provádět žádné operace s kompletním seznamem interpretovaných změn.

Je důrazně doporučeno rozšířit třídu `ComparatorBase`, která obsahuje základní metody pro ignorování změn v uzlech podstromů (metody `track` pro označení uzlu jako ignorovaného a `checkTracked` pro zjištění, zda je uzel ignorovaný), přidávání změn (metody `addChange` pro rozpoznané změny a `addUnrecognizedChange` pro nerozpoznané změny) a další pomocné metody. Pro jazyky podobné jazyku Java lze použít třídu `JavaComparatorBase` nebo přímo třídu `JavaComparator` a doimplementovat pouze interpretace specifických změn pro daný jazyk.

3.3.3 Úprava konfiguračního souboru

Kromě implementace parseru a comparatoru je také nutné upravit konfigurační soubor (popsaný v následující sekci 3.4). Sekce languages 3.4 se rozšíří o další sekci pro daný jazyk, která bude obsahovat informace o implementovaném parseru, comparatoru a příponách souborů, které označují zdrojové kódy tohoto jazyka. Struktura této nové sekce je popsána zde 3.4.

3.4 Konfigurace

Konfigurační soubor `appconfig.json` (viz Zdrojový kód 1) obsahuje základní nastavení aplikace ve formátu **JSON**. Soubor lze upravit před vytvořením **JAR** souboru aplikace (v adresáři `/res`) nebo pak přímo v samotném **JAR** souboru, změny se projeví okamžitě. Možnosti nastavení:

version – verze aplikace (aktuálně *1.0.3*).

logging – sekce nastavení logování aplikace.

- **enabled** – zapnutí/vypnutí logování.
- **logsDir** – název adresáře s logovacími soubory.
- **timePattern** – formát času pro jednotlivé logovací záznamy.
- **logFilePattern** – formát názvu logovacích souborů.

commit – sekce nastavení vytvářených commitů.

- **namePattern** – formát názvu commitů.
- **timePattern** – formát času vytvoření commitů.

branch – sekce nastavení vytvářených větví.

- **headBranch** – název **HEAD** větve.
- **defaultBranch** – název výchozí větve v novém repozitáři.

ignore – sekce nastavení .ignore souboru.

- **ignoreFile** – název .ignore souboru.

merge – sekce nastavení slučování větví.

- **autoMerge** – zapnutí/vypnutí automatického sloučení souborů bez kolizí.
- **mergeFilePrefix** – prefix vytvářených merge souborů.

repository – sekce nastavení repozitáře.

- **repositoryDir** – název adresáře repozitáře.
- **commitsDir** – název adresáře s commity v repozitáři.

branchesDir – název adresáře s větvemi v repozitáři.

headFile – název souboru s **HEAD** větví a commitem.

stageFile – název souboru se soubory ve **Stage**.

configFile – název souboru s konfigurací uživatele.

mergeFile – název souboru pro nedokončené slučování větví.

fileFormat – formát souborů v repozitáři (*binary* nebo *JSON*).

comparator – sekce nastavení pro porovnávání obsahu repozitáře.

repositoryComparator – název třídy comparatoru pro adresáře a soubory v repozitáři (implementuje rozhraní `IComparator`).

languages – sekce nastavení podporovaných jazyků zdrojových kódů.

default – sekce nastavení výchozího parseru a comparatoru pro textové soubory a zdrojových kódů nepodporovaných jazyků.

comparator – název třídy comparatoru pro textové soubory (implementuje rozhraní `IComparator`).

parser – název třídy parseru pro textové soubory (implementuje rozhraní `IASTParser`).

***** – sekce nastavení parseru a comparatoru pro každý další podporovaný jazyk (aktuálně pouze sekce *java* pro jazyk Java).

comparator – název třídy comparatoru pro daný jazyk (implementuje rozhraní `IComparator`).

parser – název třídy parseru pro daný jazyk (implementuje rozhraní `IASTParser`).

fileExtensions – přípony souborů pro zdrojové kódy daného jazyka.

```
{
  "version": "1.0.3",
  "logging": {
    "enabled": true,
    "logsDir": "logs",
    "timePattern": "MM_d_yyyy",
    "logFilePattern": "vcs_%s.log"
  },
  "commit": {
    "namePattern": "yyyy-MM-dd'T'HH-mm-ss-A",
    "timePattern": "MMMM d yyyy HH:mm:ss"
  },
  "branch": {
    "headBranch": "HEAD",
    "defaultBranch": "master"
  },
  "ignore": {
    "ignoreFile": ".ignore"
  },
  "merge": {
    "autoMerge": true,
    "mergeFilePrefix": ".merge"
  },
  "repository": {
    "repositoryDir": ".vcs",
    "commitsDir": "commits",
    "branchesDir": "branches",
    "headFile": "head",
    "stageFile": "stage",
    "configFile": "config",
    "mergeFile": "merge",
    "fileFormat": "binary"
  },
  "comparator": {
    "repositoryComparator": "comparator.RepositoryComparator"
  },
  "languages": {
    "default": {
      "comparator": "lang.text.comparator.TextComparator",
      "parser": "lang.text.parser.TextASTParser"
    },
    "java": {
      "comparator": "lang.java.comparator.JavaComparator",
      "parser": "lang.java.parser.JavaASTParser",
      "fileExtensions": [
        "java"
      ]
    }
  }
}
```

Zdrojový kód 1: Konfigurační soubor appconfig.json

4 Uživatelská dokumentace

4.1 Popis příkazů

Syntaxe příkazů je do značné míry inspirovaná verzovacím systémem **GIT** (viz Kapitola 1.1.1), proto by pro uživatele měla být práce se **Syntax VCS** poměrně intuitivní. V následujících kapitolách budou popsány všechny podporované příkazy.

4.1.1 Základní práce s repozitářem

Zobrazení nápovědy pro všechny příkazy

```
C:\test> java -jar lib/syntaxvcs.jar -h
```

nebo

```
C:\test> java -jar lib/syntaxvcs.jar --help
```

Vytvoření repozitáře

```
C:\test> java -jar lib/syntaxvcs.jar init
```

Klonování repozitáře

```
C:\test> java -jar lib/syntaxvcs.jar clone C:/absolute/path
```

Klonování a přepsání existujícího repositáře

```
C:\test> java -jar lib/syntaxvcs.jar clone -f C:/absolute/path
```

nebo

```
C:\test> java -jar lib/syntaxvcs.jar clone --force C:/absolute/path
```

4.1.2 Práce se Stage

Zobrazení aktuálního stavu Stage

```
C:\test> java -jar lib/syntaxvcs.jar status
```

Zobrazení aktuálního stavu Stage včetně ignorovaných souborů

```
C:\test> java -jar lib/syntaxvcs.jar status -i
```

nebo

```
C:\test> java -jar lib/syntaxvcs.jar status --ignored
```

Zobrazení aktuálního stavu Stage včetně nesledovaných souborů

```
C:\test> java -jar lib/syntaxvcs.jar status -u
```

nebo

```
C:\test> java -jar lib/syntaxvcs.jar status --untracked
```

Přidání souboru (více souborů) do Stage

```
C:\test> java -jar lib/syntaxvcs.jar add ./file.txt
C:\test> java -jar lib/syntaxvcs.jar add ./file1.txt ./file2.txt ...
```

Přidání všech souborů do Stage

```
C:\test> java -jar lib/syntaxvcs.jar add -a
```

nebo

```
C:\test> java -jar lib/syntaxvcs.jar add --all
```

Přidání ignorovaných souborů do Stage

```
C:\test> java -jar lib/syntaxvcs.jar add -f ./ignored_file.txt ...
```

nebo

```
C:\test> java -jar lib/syntaxvcs.jar add --force ./ignored_file.txt ...
```

Odebrání souboru (více souborů) ze Stage

```
C:\test> java -jar lib/syntaxvcs.jar remove ./file.txt
C:\test> java -jar lib/syntaxvcs.jar remove ./file1.txt ./file2.txt ...
```

Odebrání všech souborů ze Stage

```
C:\test> java -jar lib/syntaxvcs.jar remove -a
```

nebo

```
C:\test> java -jar lib/syntaxvcs.jar remove --all
```

4.1.3 Práce s commity

Vytvoření commitu

```
C:\test> java -jar lib/syntaxvcs.jar commit -m "commitMessage"
```

nebo

```
C:\test> java -jar lib/syntaxvcs.jar commit --message "commitMessage"
```

Zobrazení commitů v HEAD větvi

```
C:\test> java -jar lib/syntaxvcs.jar log
```

Zobrazení commitů v zadané větvi

```
C:\test> java -jar lib/syntaxvcs.jar log branchName
```

4.1.4 Práce s větvemi

Vytvoření nové větve

```
C:\test> java -jar lib/syntaxvcs.jar branch branchName
```

Výpis existujících větví

```
C:\test> java -jar lib/syntaxvcs.jar branch
```

Přepnutí na HEAD commit zadané větve

```
C:\test> java -jar lib/syntaxvcs.jar checkout branchName
```

Přepnutí na zadaný commit zadané větve

```
C:\test> java -jar lib/syntaxvcs.jar checkout -c commitName branchName
```

nebo

```
C:\test> java -jar lib/syntaxvcs.jar checkout --commit commitName branchName
```

4.1.5 Slučování větví

Merge zadané větve do HEAD větve

```
C:\test> java -jar lib/syntaxvcs.jar merge branchName
```

Dokončení merge po vyřešení merge konfliktů

```
C:\test> java -jar lib/syntaxvcs.jar merge
```

Obnovení obsahu zadaných merge souborů

```
C:\test> java -jar lib/syntaxvcs.jar merge restore ./.merge-file.txt
```

Obnovení obsahu všech merge souborů

```
C:\test> java -jar lib/syntaxvcs.jar merge restore -a
```

nebo

```
C:\test> java -jar lib/syntaxvcs.jar merge restore --all
```

Přerušení merge operace

```
C:\test> java -jar lib/syntaxvcs.jar merge clear
```

4.1.6 Porovnávání změn

Zobrazení změn mezi zadaným commitem a HEAD commitem

```
C:\test> java -jar lib/syntaxvcs.jar diff commitName
```

Zobrazení změn mezi dvěma zadanými commity

```
C:\test> java -jar lib/syntaxvcs.jar diff -c commitName1 commitName2
```

nebo

```
C:\test> java -jar lib/syntaxvcs.jar diff --commit commitName1 commitName2
```

Zobrazení změn mezi HEAD commitem a soubory ve Stage

```
C:\test> java -jar lib/syntaxvcs.jar diff -s
```

nebo

```
C:\test> java -jar lib/syntaxvcs.jar diff --staged
```

Zobrazení změn mezi zadaným commitem a soubory ve Stage

```
C:\test> java -jar lib/syntaxvcs.jar diff -s commitName
```

nebo

```
C:\test> java -jar lib/syntaxvcs.jar diff --staged commitName
```

Zobrazení změn mezi HEAD commitem a soubory v repozitáři

```
C:\test> java -jar lib/syntaxvcs.jar diff -w
```

nebo

```
C:\test> java -jar lib/syntaxvcs.jar diff --working-tree
```

Zobrazení změn mezi zadaným commitem a soubory v repozitáři

```
C:\test> java -jar lib/syntaxvcs.jar diff -w commitName
```

nebo

```
C:\test> java -jar lib/syntaxvcs.jar diff --working-tree commitName
```

4.1.7 Konfigurace

Zobrazení jména uživatele

```
C:\test> java -jar lib/syntaxvcs.jar config name
```

Nastavení jména uživatele

```
C:\test> java -jar lib/syntaxvcs.jar config name "User Name"
```

Zobrazení jména e-mailu uživatele

```
C:\test> java -jar lib/syntaxvcs.jar config email
```

Nastavení jména e-mailu uživatele

```
C:\test> java -jar lib/syntaxvcs.jar config email "user@email.com"
```

4.2 Tutoriál základní práce s VCS

V této sekci budou popsány základní principy práce s **VCS**, včetně konkrétních příkladů. Předpokladem je zprovoznění **VCS** dle instrukcí popsaných v Příloze A.

4.2.1 Vytvoření repozitáře

Repozitář lze vytvořit v dané složce pomocí příkazu **init**, který vytvoří skrytou složku .vcs obsahující následující položky:

složka branches – obsahuje soubory s informacemi o vytvořených větvích (zatím obsahuje pouze soubor master odpovídající výchozí master větvi).

složka commits – obsahuje soubory s informacemi o vytvořených commitech (zatím je prázdná).

soubor config – obsahuje základní informace o uživateli repozitáře (jméno a email).

soubor head – obsahuje informace o aktuální větvi a commitu, pokud nějaký v dané větvi existuje (zatím nastavená **Head** větev na master).

soubor stage – obsahuje **AST** odpovídající repozitáři se soubory a složkami, které byly přidány pomocí příkazu **add** (zatím **AST**obsahuje pouze root element Repository).

Příklad 4 (Příkaz init)
Vytvoření repozitáře v prázdné složce test pomocí příkazu **init**:

```
C:\test> java -jar lib/syntaxvcs.jar init
Initialized empty repository in 'C:\test\'
```

Výsledkem je vytvoření prázdného repozitáře ve složce test.

Příklad 5 (Příkaz clone)
Naklonování repozitáře složky test do složky testClone pomocí příkazu **clone**:

```
C:\testClone> java -jar lib/syntaxvcs.jar clone "C:\test"
Repository in 'C:\test\' was successfully cloned into 'C:\testClone\test\'
```

Výsledkem je vytvoření kopie existujícího repozitáře test ve složce testClone. Pokud by aktuální složka obsahovala složku test, bude uživatel upozorněn, že dojde k přepsání existující složky:

```
C:\testClone> java -jar lib/syntaxvcs.jar clone "C:\test"
Directory 'test' already exists in current directory
Do you want to overwrite existing directory ? (yes/no):
```

V případě použití přepínače **-f** nebo **--force** dojde rovnou k přepsání existující složky:

```
C:\testClone> java -jar lib/syntaxvcs.jar clone -f "C:\test"
Repository in 'C:\test\' was successfully cloned into 'C:\testClone\test\'
```

Zadaná cesta k již existujícímu repozitáři musí být absolutní

4.2.2 Přidání souborů do Stage

Pokud jsou v repozitáři provedeny nějaké změny, které mají být zaznamenány, je potřeba je přidat do **Stage** pomocí příkazu **add**.

Na základě přidaných souborů a složek se upraví **AST** v souboru **Stage**.

Příklad 6 (Příkaz add)

Přidání nově vytvořených souborů test1.txt, test2.txt a test3,txt ve složce test pomocí příkazu **add**:

```
C:\test> java -jar lib/syntaxvcs.jar add test1.txt test2.txt test3.txt
Added files:
      .\test1.txt (Created)
      .\test2.txt (Created)
      .\test3.txt (Created)
```

Nyní bude **Stage AST** navíc obsahovat dva uzly File odpovídající přidaným souborům (a další uzly odpovídající parametrům souborů a jejich obsahu). Pokud soubor již byl přidán do **Stage** a nebyly v něm od té doby provedeny žádné změny, nebude mít příkaz **add** žádný efekt:

```
C:\test> java -jar lib/syntaxvcs.jar add test1.txt
Nothing added
```

Pro přidání všech souborů v repozitáři je možné použít přepínač **-a** nebo **--all**:

```
C:\test> java -jar lib/syntaxvcs.jar add -a
Added files:
      .\test4.txt (Created)
      .\test5.txt (Created)
      .\test6.txt (Created)
```

4.2.3 Ignorované soubory

V repozitáři je možné specifikovat soubory, které běžně nebudou brány v potaz (například při použití příkazů **add**, **remove**, **status**, ...). Pro definici ignorovaných souborů slouží soubor .ignore, který je možné vytvořit v kořenové složce repozitáře (složka, která obsahuje skrytou složku .vcs).

Příklad 7 (Vytvoření .ignore souboru)

Soubor .ignore ve složce test, který definuje, že soubor testIgnored.txt je ignorován:

```
testIgnored.txt
```

Pokud bude uživatel chtít přidat soubor *testIgnored.txt* do **Stage** pomocí příkazu **add**, bude upozorněn, že je soubor ignorován a tedy nebyl přidán do **Stage**:

```
C:\test> java -jar lib/syntaxvcs.jar add testIgnored.txt
Nothing added
Following paths are ignored:
      .\testIgnored.txt
Use 'vcs add -f <files>' command to add them to stage
```

Ignorované soubory lze přidat do **Stage** pomocí přepínače **-f** nebo **--force**:

```
C:\test> java -jar lib/syntaxvcs.jar add -f testIgnored.txt
Added files:
      .\testIgnored.txt (Created)
```

4.2.4 Odebrání souborů ze Stage

Soubory přidané do **Stage** pomocí příkazu **add** lze zase odebrat pomocí příkazu **remove**. Uzly odpovídající odebraným souborům jsou odebrány z **AST** ve **Stage** (samotné soubory jsou ponechány ve složce).

Příklad 8 (Příkaz remove)
Odebrání souborů test1.txt, test2.txt a test3.txt ve složce test pomocí příkazu **remove**:

```
C:\test> java -jar lib/syntaxvcs.jar remove test1.txt test2.txt test3.txt
Removed files:
      .\test1.txt (Created)
      .\test2.txt (Created)
      .\test3.txt (Created)
```

Nyní bude **Stage AST** obsahovat pouze čtyři uzly File odpovídající souborům test4.txt, test5.txt, test6.txt a testIgnored.txt (které byly přidány v předchozích krocích). Pokud soubor ve **Stage** neexistuje, nebude mít příkaz **remove** žádný efekt:

```
C:\test> java -jar lib/syntaxvcs.jar remove testNonexistent.txt
Nothing removed
```

Pro odebrání všech souborů ze **Stage** je možné použít přepínač **-a** nebo **--all**:

```
C:\test> java -jar lib/syntaxvcs.jar remove -a
Removed files:
      .\test4.txt (Created)
      .\test5.txt (Created)
      .\test6.txt (Created)
      .\testIgnored.txt (Created)
```

4.2.5 Zobrazení stavu repozitáře

Pomocí příkazu **status** lze zobrazit informace o souborech přidaných do **Stage** pomocí příkazu **add**. Soubory se dělí do tří skupin:

Staged files – soubory přidané do **Stage** pomocí příkazu **add**.

Unstaged files – soubory přidané do **Stage** a následně pozměněné (ve **Stage** je jejich neaktuální verze).

Untracked files – soubory ve složce repozitáře, které nebyly přidány do **Stage**.

U všech vypsaných souborů je také uveden jejich stav (Created, Modified, Removed).

Příklad 9 (Příkaz status)

Pokud budou soubory test1.txt, test2.txt, test3.txt a test6.txt přidány do **Stage** pomocí příkazu **add**, příkaz **status** je zobrazí jako Staged files a v závorce uvede jejich stav (Created):

```
C:\test> java -jar lib/syntaxvcs.jar add test1.txt test2.txt test3.txt test6.txt
Added files:
      .\test1.txt (Created)
      .\test2.txt (Created)
      .\test3.txt (Created)
      .\test6.txt (Created)
C:\test> java -jar lib/syntaxvcs.jar status
Staged files:
      .\test1.txt (Created)
      .\test2.txt (Created)
      .\test3.txt (Created)
      .\test6.txt (Created)
```

Při změně obsahu souboru test3.txt a smazání souboru test6.txt se zobrazí soubor test3.txt jako Unstaged file se stavem Modified a soubor test6.txt jako Unstaged file se stavem (Deleted):

```
C:\test> java -jar lib/syntaxvcs.jar status
Staged files:
      .\test1.txt (Created)
      .\test2.txt (Created)
      .\test3.txt (Created)
      .\test6.txt (Created)
Unstaged files:
      .\test3.txt (Modified)
      .\test6.txt (Deleted)
```

Ostatní soubory ve složce repozitáře, které nebyly přidány do **Stage** (Untracked files) je možné je zobrazit pomocí přepínače **-u** nebo **--untracked**:

```
C:\test> java -jar lib/syntaxvcs.jar status -u
Staged files:
      .\test1.txt (Created)
      .\test2.txt (Created)
      .\test3.txt (Created)
      .\test6.txt (Created)
Unstaged files:
      .\test3.txt (Modified)
      .\test6.txt (Deleted)
Untracked files:
      .\.ignore
      .\test4.txt
      .\test5.txt
```

Ignorované soubory lze zobrazit pomocí přepínače **-i** nebo **--ignored**:

```
C:\test> java -jar lib/syntaxvcs.jar status -u -i
Staged files:
      .\test1.txt (Created)
      .\test2.txt (Created)
      .\test3.txt (Created)
      .\test6.txt (Created)
Unstaged files:
      .\test3.txt (Modified)
      .\test6.txt (Deleted)
```

```
Untracked files:
      .\.ignore
      .\test4.txt
      .\test5.txt
      .\testIgnored.txt
```

4.2.6 Nastavení uživatelského jména a emailu

Před vytvářením commitů je nutné nastavit informace o uživateli pomocí příkazu **config**, které se pak zobrazí u daného commitu, který uživatel vytvořil.

Příklad 10 (Příkaz commit)
Výpis aktuálního jména a emailu uživatele se provádí pomocí příkazů **config name** a **config email**:

```
C:\test> java -jar lib/syntaxvcs.jar config name
name: undefined
C:\test> java -jar lib/syntaxvcs.jar config email
email: undefined
```

Nastavení aktuálního jména a emailu se provádí opět pomocí příkazů **config name** a **config email**:

```
C:\test> java -jar lib/syntaxvcs.jar config name "John Doe"
name: "John Doe"
C:\test> java -jar lib/syntaxvcs.jar config email "john.doe@email.com"
email: "john.doe@email.com"
```

4.2.7 Vytvoření commitu

Změny v repozitáři, které byly přidány do **Stage**, je možné zaznamenat a uchovat pomocí příkazu **commit**. Při vytváření commitu je smazán obsah **Stage** a vytvořen nový soubor ve složce .vcs/commits, který obsahuje **AST** ze **Stage**, informace o uživateli, který commit vytvořil, informace o větvi, ve které byl commit vytvořen a commit zprávu.

Příklad 11 (Příkaz commit)
Pokud je **Stage** prázdná, nebude mít příkaz **commit** žádný efekt:

```
C:\test> java -jar lib/syntaxvcs.jar remove -a
Removed files:
      .\test1.txt (Created)
      .\test2.txt (Created)
      .\test3.txt (Created)
      .\test6.txt (Created)
C:\test> java -jar lib/syntaxvcs.jar commit -m "Test commit message"
Stage is empty - nothing to commit
Use 'vcs add' command to add files to stage before commit
```

Pokud budou ve **Stage** soubory test1.txt, test2.txt a test3.txt, je možné vytvořit commit pomocí příkazu **commit** s povinným přepínačem **-m** nebo **--message**, který jako parametr očekává commit zprávu (krátký popis změn):

```
C:\test> java -jar lib/syntaxvcs.jar add test1.txt test2.txt test3.txt
Added files:
      .\test1.txt (Created)
      .\test2.txt (Created)
      .\test3.txt (Created)
C:\test> java -jar lib/syntaxvcs.jar commit -m "First test commit message"
```

Po této operaci bude **Stage** prázdná, složka .vcs/commits bude obsahovat nový soubor a do souboru **HEAD** větve ve složce .vcs/branches bude zapsán název vytvořeného commitu.

4.2.8 Vytvoření větve

Při vytvoření repozitáře pomocí příkazu **init** je vytvořena výchozí větev master, nové větve lze vytvářet pomocí příkazu **branch**. Při vytvoření nové větve je vytvořen nový soubor ve složce .vcs/branches, který obsahuje název rodičovské větve (větve, ze které byla tato nový vytvořena) a seznam commitů v této větvi.

Příklad 12 (Příkaz branch)
Seznam větví v repozitáři lze vypsat pomocí příkazu **branch** (aktuálně obsahuje pouze výchozí větev master):

```
C:\test> java -jar lib/syntaxvcs.jar branch
*master
```

HEAD větev je ve výpisu vždy označena hvězdičkou

Pro vytvoření nové větve slouží opět příkaz **branch** se jménem nové větve jako parametrem:

```
C:\test> java -jar lib/syntaxvcs.jar branch newBranch
*newBranch
master
```

Ve složce .vcs/branches se vytvoří nový soubor newBranch, který obsahuje **HEAD** commit jako počáteční commit a master větev jako rodičovskou větev.

Nově vytvořená větev je vždy nastavena jako HEAD větev

4.2.9 Výpis commitů v HEAD větvi

Pro výpis commitů vytvořených v aktuální **HEAD** větvi slouží příkaz **log**, commity jsou řazeny od nejnovějšího po nejstarší. Každý commit obsahuje název commitu, informaci o uživateli, který ho vytvořil, datum a čas vytvoření a commit zprávu.

Příklad 13 (Příkaz log)

Pokud bude ve větvi newBranch vytvořen další commit, u kterého dojde ke změně obsahu souboru test3.txt a smazání souboru test1.txt, bude výpis commitů pomocí příkazu **log** vypadat následovně:

```
C:\test> java -jar lib/syntaxvcs.jar add test1.txt test3.txt test4.txt
Added files:
      .\test1.txt (Deleted)
      .\test3.txt (Modified)
      .\test4.txt (Created)
C:\test> java -jar lib/syntaxvcs.jar commit -m "Second test commit message"
C:\test> java -jar lib/syntaxvcs.jar log
Commit: 2022-06-18T18-21-55-66115064
Author: John Doe (john.doe@email.com)
Date: June 18 2022 18:21:55
Message: "Second test commit message"

Commit: 2022-06-18T18-15-33-65733822
Author: John Doe (john.doe@email.com)
Date: June 18 2022 18:15:33
Message: "First test commit message"
```

4.2.10 Přepínání mezi větvemi

Při přepnutí na jinou větev pomocí příkazu **checkout** se složka repozitáře vrátí do stavu před vytvořením cílového commitu v dané větvi (při přepnutí do původní větve se složka repozitáře vrátí opět do původního stavu). Přepnutí je možné provést na danou větev (potom bude **HEAD** commitem poslední commit v dané větvi) nebo na konkrétní commit (**HEAD** commitem bude daný commit). Před provedením je uživatel upozorněn, že dojde k úpravě složky repozitáře.

Příklad 14 (Příkaz checkout)

Pokud je **HEAD** větví větev newBranch, bude složka repozitáře po provedení příkazu **checkout master** obsahovat pouze soubory test1.txt, test2.txt a test3.txt (soubor test1.txt v daném commitu existoval a soubor text4.txt byl přidán až později):

```
C:\test> java -jar lib/syntaxvcs.jar checkout master
Do you really want to checkout and restore files ? (yes/no): yes
Branch: master
Commit: 2022-06-18T18-15-33-65733822
```

Pokud by následně uživatel provedl příkaz **checkout newBranch**, obsahovala by složka navíc soubor test4.txt a soubor test1.txt by v ní neexistoval (tyto změny byly provedeny ve druhém commitu):

```
C:\test> java -jar lib/syntaxvcs.jar checkout newBranch
Do you really want to checkout and restore files ? (yes/no): yes
Branch: newBranch
Commit: 2022-06-18T18-21-55-66115064
```

Přepnutí na konkrétní commit lze provést pomocí přepínače **-c** nebo **--commit**, který jako parametr akceptuje název commitu v dané větvi:

```
C:\test> java -jar lib/syntaxvcs.jar checkout -c 2022-06-18T18-15-33-65733822
    newBranch
Do you really want to checkout and restore files ? (yes/no): yes
Branch: newBranch
Commit: 2022-06-18T18-15-33-65733822
```

Po provedení tohoto příkazu zůstane **HEAD** větví větev newBranch, ale **HEAD** commitem bude předchozí commit (soubory test1.txt bude opět existovat a soubor test4.txt bude smazán).

4.2.11 Výpis rozdílů mezi commity nebo soubory v repozitáři

Porovnat stav repozitáře ve dvou různých commitech lze provést pomocí příkazu **diff** a přepínače **-c** nebo **--commit**, který akceptuje jako parametry názvy dvou porovnávaných commitů, případně pouze název jednoho commitu (v tom případě je jako druhý commit brán **HEAD** commit). Příkaz **diff** nejdříve porovná změny týkající se repozitáře (vytvoření, smazání a přejmenování složek, vytvoření, smazání, přejmenování a změna souborů), následně může uživatel zobrazit porovnání změn v konkrétních souborech.

Příklad 15 (Příkaz diff)

Pokud bude ve větvi master vytvořen další commit (2022-06-18T18-33-42-66822506), kde dojde ke smazání souboru test2.txt, změně obsahu souboru test3.txt a přidání souboru test5.txt, bude uživateli při použití příkazu **diff -c 2022-06-18T18-33-42-66822506 2022-06-18T18-15-33-65733822** vypsán následující seznam změn:

```
C:\test> java -jar lib/syntaxvcs.jar checkout master
Do you really want to checkout and restore files ? (yes/no): yes
Branch: master
Commit: 2022-06-18T18-15-33-65733822
C:\test> java -jar lib/syntaxvcs.jar add test2.txt test3.txt test5.txt
Added files:
      .\test2.txt (Deleted)
      .\test3.txt (Modified)
      .\test5.txt (Created)
C:\test> java -jar lib/syntaxvcs.jar commit -m "Third test commit message"
C:\test> java -jar lib/syntaxvcs.jar diff -c 2022-06-18T18-15-33-65733822
    2022-06-18T18-33-42-66822506
Repository differences:
      Deleted file 'test2.txt' in root directory
      Changed file 'test3.txt' in root directory
      Created new file 'test5.txt' in root directory
File '\test3.txt' differences:
      Changed line 1
File '\test5.txt' differences:
      Added new line 1
```

Porovnat stav repozitáře v daném commitu s aktuálním stavem repozitáře lze provést pomocí přepínače **-w** nebo **--working-tree**. Pokud commit nebude zadán, budou vypsány změny mezi stavem repozitáře **HEAD** commitu a aktuálním stavem repozitáře. Pokud budou v repozitáři vytvořeny nové soubory test6.txt a test7.txt, bude uživateli při použití příkazu **diff -w 2022-06-18T18-15-33-65733822** vypsán následující seznam změn:

```
C:\test> java -jar lib/syntaxvcs.jar diff -w 2022-06-18T18-15-33-65733822
Repository differences:
      Created new file 'test6.txt' in root directory
      Created new file 'test7.txt' in root directory
File '\test7.txt' differences:
      Added new line 1
File '\test6.txt' differences:
      Added new line 1
```

Porovnat stav repozitáře v daném commitu se stavem repozitáře ve **Stage** lze provést pomocí pomocí přepínače **-s** nebo **--staged**. Pokud commit nebude zadán, budou vypsány změny mezi stavem repozitáře **HEAD** commitu a stavem repozitáře ve **Stage**. Pokud bude soubor test6.txt přidán do **Stage**, bude uživateli při použití příkazu **diff -s** vypsán následující seznam změn:

```
C:\test> java -jar lib/syntaxvcs.jar add test6.txt
Added files:
      .\test6.txt (Created)
C:\test> java -jar lib/syntaxvcs.jar diff -s
Repository differences:
      Created new file 'test6.txt' in root directory
File '\test6.txt' differences:
      Added new line 1
```

Obdobným způsobem lze porovnávat i změny ve zdrojových kódech v programovacím jazyku Java.

Příklad 16 (Příkaz diff)

Pokud bude v nové větvi java vytvořen commit (2022-06-18T18-36-15-66158746), kde dojde k přidání souboru test8.java s následujícím obsahem:

```
public class Test {

    private int b = 2;

    public static int test(int a) {
        var c = a + b;
        return c + a + b;
    }
}
```

A následně bude vytvořen další commit (2022-06-18T18-38-36-75698423), kde bude tento soubor upraven následujícím způsobem:

```
public class Test2 {

    private int B = 2;

    public static int test2(int A) {
        var c = A + B;
        return c + A + B;
    }
}
```

Při použití příkazu **diff -c 2022-06-18T18-36-15-66158746 2022-06-18T18-38-36-75698423** bude uživateli vypsán následující seznam změn:

```
C:\test> java -jar lib/syntaxvcs.jar branch java
*java
master
newBranch
C:\test> java -jar lib/syntaxvcs.jar add test8.java
Added files:
      .\test8.java (Created)
C:\test> java -jar lib/syntaxvcs.jar commit -m "Fourth test commit message"
C:\test> java -jar lib/syntaxvcs.jar add test8.java
Added files:
      .\test8.java (Modified)
C:\test> java -jar lib/syntaxvcs.jar commit -m "Fifth test commit message"
C:\test> java -jar lib/syntaxvcs.jar diff -c 2022-06-18T18-36-15-66158746
    2022-06-18T18-38-36-75698423
Repository differences:
      Changed file 'test8.java' in root directory
File '.\test8.java' differences:
      Renamed class 'Test' to 'Test2'
      Renamed method 'test' (parameters int) to 'test2' in class 'Test2'
      Renamed field 'b' (type int) to 'B' in class 'Test'
      Renamed parameter 'a' (type int) to 'A' of method 'test' (parameters int) in
          class 'Test'
```

4.2.12 Slučování větví

Sloučit změny provedené v jedné větvi se změnami provedenými v jiné větvi lze pomocí příkazu **merge**, který akceptuje jako povinný parametr název větve, která má být sloučena s **HEAD** větví.

Příklad 17 (Příkaz merge)

Pokud je **HEAD** větví větev master, může uživatel sloučit změny provedné ve větvi newBranch se změnami ve větvi master (merge newBranch do master) pomocí příkazu **merge**:

```
C:\test> java -jar lib/syntaxvcs.jar checkout master
Do you really want to checkout and restore files ? (yes/no): yes
Branch: master
Commit: 2022-06-18T18-33-42-66822506
C:\test> java -jar lib/syntaxvcs.jar merge newBranch
Merge could not be completed
Resolve merge conflicts in files:
      .\.merge-test3.txt
and use 'vcs merge' command to complete merge
```

Jelikož došlo v obou větvích ke kolidující změně (změna obsahu souboru test3.txt), není možné provést **merge** automaticky – uživatel musí rozhodnout, jaký obsah má mít výsledný soubor test3.txt. Ke každému kolidujícímu souboru je ve stejné složce vytvořen merge soubor ve tvaru .merge-<file-name>.txt. Pro soubor test3.txt bude vytvořen merge soubor .merge-test3.txt s následujícím obsahem:

```
Changed file 'test3.txt' in root directory (HEAD)
Changed file 'test3.txt' in root directory (newBranch)
[0: File: test3.txt]
  <<<<<<< HEAD
  Changed line 1 (HEAD)
  [1: Line: test3B]
  ======= HEAD/newBranch
  Changed line 1 (newBranch)
  [2: Line: test3A]
  >>>>>>> newBranch
```

Soubor obsahuje **AST** v textové se změnami provedenými v obou větvích. Uzly **AST** jsou ve tvaru [<index>: <type>[: <value>]] a jejich odsazení určuje jejich hloubku (kořen nemá žádné odsazení, jeho potomci jsou odsazeni
o jednu mezeru, jejich potomci o dvě mezery, ...). Začátek kolidujícího bloku uzlů je označen pomocí <<<<<<< HEAD (pokud byla změna provedena v **HEAD** větvi) nebo >>>>>>> <branchName> (pokud byla změna proveden v merge větvi) a textového popisu změn. Konec kolidujícího bloku je potom označen pomocí ======= HEAD nebo ======= <branchName>. V případě, že byly kolidující změny provedeny na totožném bloku uzlů, bude začátek bloku **HEAD** větve označen pomocí <<<<<<< HEAD, konec bloku **HEAD** větve a zároveň začátek bloku merge větve pomocí ======= HEAD/newBranch a konec bloku merge větve pomocí >>>>>>> newBranch.

Pro úspěšné provedení merge operace musí uživatel projít všechny merge soubory a upravit jejich obsah tak, aby zde zůstaly pouze uzly **AST**, které odpovídají výslednému obsahu souboru po provedení *merge*. Výsledný soubor musí tedy obsahovat pouze řádky odpovídající uzlům **AST** s korektním odsazením (v případě, že po provedení operace má být soubor smazán bude obsah merge souboru prázdný).

Příklad 18 (Příkaz merge)

Po úpravě všech merge souborů uživatel může dokončit merge operaci pomocí příkazu **merge**. Pokud některý z merge souboru nebyl korektně upraven, bude uživateli vypsán seznam nevalidních merge souborů:

```
C:\test> java -jar lib/syntaxvcs.jar merge
Merge could not be completed
Invalid merge files:
      .\.merge-test3.txt
use 'vcs merge restore <paths>' command to restore merge files and resolve merge
    conflicts
```

Příklad 19 (Příkaz merge restore)

Původní obsah merge souboru lze obnovit pomocí příkazu **merge restore**:

```
C:\test> java -jar lib/syntaxvcs.jar merge restore .merge-test3.txt
Restored merge conflict files:
            .\.merge-test3.txt
```

Pro obnovení obsahu všech merge souborů lze použít přepínač **-a** nebo **--all**:

```
C:\test> java -jar lib/syntaxvcs.jar merge restore -a
Restored merge conflict files:
            .\.merge-test3.txt
```

Příklad 20 (Příkaz merge clear)

V případě, že uživatel nebude chtít dokončit merge operaci, může použít příkaz **merge clear**, který vymaže všechny merge soubory a přeruší slučování:

```
C:\test> java -jar lib/syntaxvcs.jar merge clear
Incomplete merge was successfully discarded
```

Příklad 21 (Příkaz merge)

Pokud bude chtít uživatel ponechat změny provedené v obou větvích (text test3B i test3A), bude výsledný merge file vypadat takto:

```
[0: File: test3.txt]
  [1: Line: test3B]
  [2: Line: test3A]
```

Potom po provedení příkazu **merge** bude vytvořen nový commit ve větvi master, který bude obsahovat sloučené změny z obou větví doplněné o změny provedené v merge souborech:

```
C:\test> java -jar lib/syntaxvcs.jar merge
Branch 'newBranch' was successfully merged to HEAD branch
C:\test> java -jar lib/syntaxvcs.jar log
Commit: 2022-06-18T21-31-36-77496887
Author: John Doe (john.doe@email.com)
Date: June 18 2022 21:31:36
Message: "Merged 'newBranch' into 'master'"

Commit: 2022-06-18T18-33-42-66822506
Author: John Doe (john.doe@email.com)
Date: June 18 2022 18:33:42
Message: "Third test commit message"

Commit: 2022-06-18T18-15-33-65733822
Author: John Doe (john.doe@email.com)
Date: June 18 2022 18:15:33
Message: "First test commit message"
```

Obsah repozitáře bude potom odpovídat tomuto commitu - v repozitáři budou pouze soubory test3.txt, test4.txt a test5.txt a obsah souboru test3.txt bude následující:

```
test3Btest3A
```

Obdobným způsobem lze slučovat větve se změnami provedenými ve zdrojových kódech v programovacím jazyku Java.

Příklad 22 (Příkaz merge)
Pokud bude z java větve vytvořena nová větev newJava s commitem (2022-06-18T18-38-49-12577965) obsahujícím následující úpravu souboru test8.java:

```
public class Test2 {

    private int B = 2;
    private double X = 3;

    public static double test2(int A) {
        var c = A + B;
        return C + X;
  }
}
```

A následně bude ve větvi java vytvořen další commit (2022-06-18T18-39-15-64823597) obsahující následující úpravu téhož souboru:

```
public class Test2 {

    private int B = 2;
    private double Y = 4;

    public static double test2(int A) {
        var c = A + B;
        return C + Y;
    }
}
```

Potom budou při slučování větve newJava pomocí příkazu **merge newJava** detekovány kolidující změny v souboru test8.java:

```
C:\test> java -jar lib/syntaxvcs.jar branch newJava
*newJava
java
master
newBranch
C:\test> java -jar lib/syntaxvcs.jar add test8.java
Added files:
      .\test8.java (Modified)
C:\test> java -jar lib/syntaxvcs.jar commit -m "Sixth test commit message"
C:\test> java -jar lib/syntaxvcs.jar checkout java
Do you really want to checkout and restore files ? (yes/no): yes
Branch: java
Commit: 2022-06-18T18-38-36-75698423
C:\test> java -jar lib/syntaxvcs.jar add test8.java
Added files:
      .\test8.java (Modified)
C:\test> java -jar lib/syntaxvcs.jar commit -m "Seventh test commit message"
C:\test> java -jar lib/syntaxvcs.jar merge newBranch
Merge could not be completed
Resolve merge conflicts in files:
```

```
      .\.merge-test8.java
and use 'vcs merge' command to complete merge
```

Vygenerovaný merge soubor .merge-test8.java bude mít následující obsah:

```
Changed file 'test8.java' in root directory (HEAD)
Changed file 'test8.java' in root directory (newJava)
[0: File: test8.java]
 [1: ClassDeclaration: Test2]
  [2: Modifier: public]
  [3: Body]
   [4: FieldDeclaration]
    [5: Modifier: private]
    [6: PrimitiveType: int]
    [7: Variable: B]
     [8: Initialization]
      [9: IntegerLiteral: 2]
   [10: FieldDeclaration]
    [11: Modifier: private]
    <<<<<<< HEAD
    Created new field 'X' (type double) in class 'Test2' (HEAD)
    [12: PrimitiveType: double]
    Created new field 'X' (type double) in class 'Test2' (HEAD)
    [13: Variable: X]
     Created new field 'X' (type double) in class 'Test2' (HEAD)
     [14: Initialization]
      Created new field 'X' (type double) in class 'Test2' (HEAD)
      [15: IntegerLiteral: 3]
    ======= HEAD/newJava
    Created new field 'Y' (type double) in class 'Test2' (newJava)
    [16: Variable: Y]
     Created new field 'Y' (type double) in class 'Test2' (newJava)
     [17: Initialization]
      Created new field 'Y' (type double) in class 'Test2' (newJava)
      [18: IntegerLiteral: 4]
    >>>>>>> newJava
   [19: MethodDeclaration: test2]
    [20: Modifier: public]
    [21: Modifier: static]
    [22: PrimitiveType: double]
    [23: Parameters]
     [24: Parameter: A]
      [25: PrimitiveType: int]
    [26: Body]
     [27: VariableDeclaration]
      [28: VarType]
      [29: Variable: c]
       [30: Initialization]
        [31: BinaryExpression: +]
         [32: NameLiteral: A]
         [33: NameLiteral: B]
     [34: ReturnStatement]
      [35: BinaryExpression: +]
       [36: NameLiteral: C]
       <<<<<<< HEAD
       Changed return statement value of method 'test2' (parameters int) in class'
           Test2' (HEAD)
       [37: NameLiteral: X]
       ======= HEAD/newJava
       Changed return statement value of method 'test2' (parameters int) in class'
           Test2' (newJava)
       [39: NameLiteral: Y]
       >>>>>>> newJava
```

Pokud bude chtít uživatel ponechat pouze změny provedené ve větvi newJava, bude výsledný merge file vypadat takto:

```
[0: File: test8.java]
[1: ClassDeclaration: Test2]
 [2: Modifier: public]
 [3: Body]
  [4: FieldDeclaration]
   [5: Modifier: private]
   [6: PrimitiveType: int]
   [7: Variable: B]
    [8: Initialization]
     [9: IntegerLiteral: 2]
  [10: FieldDeclaration]
   [11: Modifier: private]
   [12: PrimitiveType: double]
   [16: Variable: Y]
    [17: Initialization]
     [18: IntegerLiteral: 4]
  [19: MethodDeclaration: test2]
   [20: Modifier: public]
   [21: Modifier: static]
   [22: PrimitiveType: double]
   [23: Parameters]
    [24: Parameter: A]
     [25: PrimitiveType: int]
   [26: Body]
    [27: VariableDeclaration]
     [28: VarType]
     [29: Variable: c]
      [30: Initialization]
       [31: BinaryExpression: +]
        [32: NameLiteral: A]
        [33: NameLiteral: B]
    [34: ReturnStatement]
     [35: BinaryExpression: +]
      [36: NameLiteral: C]
      [39: NameLiteral: Y]
```

Po provedení příkazu **merge** bude vytvořen nový commit ve větvi java, který bude obsahovat soubor test8.java odpovídající verzi ve větvi newJava:

```
C:\test> java -jar lib/syntaxvcs.jar merge
Branch 'newJava' was successfully merged to head branch
C:\test> java -jar lib/syntaxvcs.jar log
Commit: 2022-06-18T21-43-19-34675923
Author: John Doe (john.doe@email.com)
Date: June 18 2022 21:43:19
Message: "Merged 'newJava' into 'java'"

Commit: 2022-06-18T18-39-15-64823597
Author: John Doe (john.doe@email.com)
Date: June 18 2022 18:39:15
Message: "Seventh test commit message"

Commit: 2022-06-18T18-36-15-66158746
Author: John Doe (john.doe@email.com)
Date: June 18 2022 18:36:15
Message: "Fifth test commit message"
```

Závěr

V rámci této práce byl vytvořen lokální verzovací systém **Syntax VCS**, který ukládá sady změn (snímky) ve formě abstraktních syntaktických stromů (**AST**). To potom umožňuje interpretovat význam provedených změn pro konkrétní programovací jazyk a poskytnout tak uživateli více informací pro řešení kolizí při slučování změn. **Syntax VCS** obsahuje nativní podporu pro zdrojové kódy v programovacím jazyku Java (ve verzi 17 a nižší), ale je ho možné jednoduše rozšířit o libovolný další programovací jazyk v souladu se zadáním práce. Výpis změn probíhá pouze v textové formě, možným vylepšením je zavedení nástroje pro grafické znázornění změn v daných zdrojových kódech.

V úvodu textové části práce byl potom popsán význam verzovacího systému, včetně porovnání vybraných open-source verzovacích systémů. Také zde byly vysvětleny základní pojmy týkající se **AST** a jejich porovnávání pomocí **Change Distilling** algoritmu. Další kapitola byla věnována krátkému popisu technologií použitých při vývoji aplikace.

Technická dokumentace obsahuje popis algoritmu pro porovnávání **AST**, který je založen na již zmíněném **Change Distilling** algoritmu, a také algoritmu použitém pro slučování změn v rámci **Syntax VCS**. Dále je v kapitole popsána struktura aplikace, podrobný postup pro rozšíření o podporu pro další programovací jazyk. Na závěr jsou potom rozebrány možnosti konfigurace aplikace.

Poslední částí je uživatelská dokumentace, která se skládá z popisu všech příkazů, které jsou aktuálně v **Syntax VCS** podporovány a krátkého tutoriálu, který uživatele provede všemi možnostmi verzovacího systému.

Conclusions

As part of this thesis, a local versioning system **Syntax VCS** was created, which stores sets of changes (snapshots) in the form of abstract syntactic trees (**AST**). This then makes it possible to interpret the meaning of the changes made for a particular programming language, providing the user with more information for resolving conflicts when merging changes. **Syntax VCS** includes native support for source codes in the Java programming language (in version 17 and below), but it can be easily extended to include any other programming language in accordance with the thesis assignment. The listing of changes takes place only in text form, a possible improvement is the introduction of a tool for graphical representation of changes in given source codes.

In the introduction of the text part of the thesis, the meaning of the versioning system was described, including a comparison of selected open-source versioning systems. Also, basic concepts related to **AST** and their comparison using the **Change Distilling** algorithm were explained here. The next chapter was devoted to a short description of the technologies used in the development of the application.

The technical documentation contains a description of the **AST** comparison algorithm, which is based on the previously mentioned **Change Distilling** algorithm, as well as the algorithm used for merging changes within the **Syntax VCS**. The chapter also describes the structure of the application, a detailed procedure for extending support for another programming language. Finally, the configuration options of the application are discussed.

The last part is the user documentation, which consists of a description of all the commands that are currently supported in **Syntax VCS** and a short tutorial that guides the user through all the options of the versioning system.

A Obsah elektronických dat

bin/
: Adresář obsahuje spustitelné soubory a runtime knihovny pro bezproblémový běh programu pro operační systémy Windows (v ZIP archivu) a Linux (v TAR.GZ archivu). V archivech je rovněž soubor README.md obsahující instrukce pro spuštění aplikace, soupis všech jejích funkcí a tutoriál aplikace.

lib/
: Adresář obsahuje knihovny nutné pro běh aplikace popsané v Kapitole 3.1.

res/
: Adresář obsahuje konfigurační soubor aplikace popsaný v Kapitole 3.4.

src/
: Adresář obsahuje všechny zdrojové soubory aplikace popsané v Kapitole 3.2.

text/
: Adresář s textem práce ve formátu PDF, vytvořený s použitím závazného stylu KI PřF UP v Olomouci pro závěrečné práce, včetně všech příloh, a všechny soubory potřebné pro bezproblémové vytvoření PDF dokumentu textu v ZIP archivu, tj. zdrojový text textu a příloh, vložené obrázky, apod.

README.txt
: Textový soubor se základními informacemi o aplikaci a instrukcemi pro její spuštění.

U veškerých cizích obsažených materiálů jejich zahrnutí dovolují podmínky pro jejich veřejné šíření nebo přiložený souhlas držitele práv k užití. Pro všechny použité (a citované) materiály, u kterých toto není splněno a nejsou tak obsaženy, je uveden jejich zdroj, např. webová adresa, v bibliografii nebo textu práce nebo souboru `README.txt`.

B Dokumentace podpory pro jazyk Java

Implementovaný parser a comparator pro jazyk Java podporuje všechny zdrojové kódy do verze Java SE 17. Všechny použité typy uzlů **AST** jsou popsány v Tabulce 3 včetně názorných příkladů **AST** vygenerovaných parserem pro příslušné části zdrojových kódů.

Tabulka 3: Typy uzlů AST pro jazyk Java

Typ uzlu	Příklad kódu	Příklad AST
Annotation	@Annotation() class Class { }	[ClassDeclaration, Class] [Annotation] [Name, Annotation] [Elements] [Body]
AnnotationDeclaration	@interface Annotation { }	[AnnotationDeclaration, Annotation] [Body]

AnnotationElementDeclaration	@interface Annotation {
	int element();
	}
		[AnnotationDeclaration, Annotation]
		[Body]
		[AnnotationElementDeclaration, element]
		[PrimitiveType, int]

Argument	method(argument);	[MethodCallExpression] [Name, method] [Arguments] [Argument] [NameLiteral, argument]
Arguments	method(argument1, argument2);	[MethodCallExpression] [Name, method] [Arguments] [Argument] [NameLiteral, argument1] [Argument] [NameLiteral, argument2]
ArrayAccessExpression	array[0] = 1;	[AssignmentExpression, =] [ArrayAccessExpression] [NameLiteral, array] [IntegerLiteral, 0] [IntegerLiteral, 1]
ArrayCreation	variable = new int[0];	[AssignmentExpression, =] [NameLiteral, variable] [ArrayCreation] [PrimitiveType, int] [ArrayDimension] [IntegerLiteral, 0]
ArrayDimension	int[][] variable;	[VariableDeclaration] [ArrayType] [PrimitiveType, int] [ArrayDimension] [ArrayDimension] [Variable, variable]

Typ uzlu	Příklad kódu	Příklad AST
ArrayInitializer	var variable = new int[] { 1 };	[VariableDeclaration] [VarType] [Variable, variable] [Initialization] [ArrayCreation] [PrimitiveType, int] [ArrayDimension] [ArrayInitializer] [IntegerLiteral, 1]
ArrayType	int[] variable;	[VariableDeclaration] [ArrayType] [PrimitiveType, int] [ArrayDimension] [Variable, variable]
AssertStatement	assert variable == 1;	[AssertStatement] [BinaryExpression, ==] [NameLiteral, variable] [IntegerLiteral, 1]
AssignmentExpression	variable = 1;	[AssignmentExpression, =] [NameLiteral, variable] [IntegerLiteral, 1]
BinaryExpression	variable = 1 + 2;	[AssignmentExpression, =] [NameLiteral, variable] [BinaryExpression, +] [IntegerLiteral, 1] [IntegerLiteral, 2]
Block	if (condition) { }	[IfStatement] [Condition] [NameLiteral, condition] [Block]
Body	class Class { }	[ClassDeclaration, Class] [Body]
BooleanLiteral	variable = false;	[AssignmentExpression, =] [NameLiteral, variable] [BooleanLiteral, false]
BreakExpression	break;	[BreakExpression]

Case	switch(value) {
	case 1: break;
	}
		[SwitchStatement]
		[NameLiteral, value]
		[Case]
		[CaseValue]
		[IntegerLiteral, 1]
		[BreakExpression]

CaseRule	switch(value) {
	case 1 ->1;
	}
		[SwitchExpression]
		[NameLiteral, value]
		[CaseRule]
		[CaseRuleValue]
		[IntegerLiteral, 1]
		[IntegerLiteral, 1]

CaseRuleValue	switch(value) {
	case 1 ->1;
	}
		[SwitchExpression]
		[NameLiteral, value]
		[CaseRule]
		[CaseRuleValue]
		[IntegerLiteral, 1]
		[IntegerLiteral, 1]

Typ uzlu	Příklad kódu	Příklad AST

CaseValue	switch(value) {
	  case 1: break;
	}
		[SwitchStatement]
		  [NameLiteral, value]
		  [Case]
		  [CaseValue]
		  [IntegerLiteral, 1]
		  [BreakExpression]

CastExpression	variable = (int)object;	[AssignmentExpression, =]
		  [NameLiteral, variable]
		  [CastExpression]
		  [PrimitiveType, int]
		  [NameLiteral, object]

CatchClause	try { } catch (Exception e) { }	[TryStatement]
		  [Block]
		  [CatchClause, e]
		  [ReferenceType]
		  [Name, Exception]
		  [Block]

CharLiteral	variable = 'c';	[AssignmentExpression, =]
		  [NameLiteral, variable]
		  [CharLiteral, 'c']

ClassDeclaration	class Class { }	[ClassDeclaration, Class] [Body]

ClassExpression	variable = Object.class;	[AssignmentExpression, =]
		  [NameLiteral, variable]
		  [ClassExpression]
		  [ReferenceType]
		  [Name, Object]

Condition	if (condition) { }	[IfStatement]
		  [Condition]
		  [NameLiteral, condition]
		  [Block]

ConditionalExpression	variable = condition ? 1 : 2;	[AssignmentExpression, =]
		  [NameLiteral, variable]
		  [ConditionalExpression]
		  [Condition]
		  [NameLiteral, condition]
		  [IntegerLiteral, 1]
		  [IntegerLiteral, 2]

ConstructorDeclaration	constructor() { }	[ConstructorDeclaration]
		  [Parameters]
		  [Body]

ContinueExpression	continue;	[ContinueExpression]

Default	switch(value) {
	  default: break;
	}
		[SwitchStatement]
		  [NameLiteral, value]
		  [Default]
		  [BreakExpression]

DefaultRule	switch(value) {
	  default ->1;
	}
		[SwitchExpression]
		  [NameLiteral, value]
		  [DefaultRule]
		  [IntegerLiteral, 1]

Typ uzlu	Příklad kódu	Příklad AST

DefaultValue	@interface Annotation {
	  int element() default 1;
	}
		[AnnotationDeclaration, Annotation]
		[Body]
		[AnnotationElementDeclaration, element]
		[PrimitiveType, int]
		[DefaultValue]
		[IntegerLiteral, 1]

DoWhileStatement	do { } while(condition);	[DoWhileStatement] [Block] [Condition] [NameLiteral, condition]
Element	@Annotation(element = value) class Class { }	[ClassDeclaration, Class] [Annotation] [Name, Annotation] [Elements] [Element] [Name, element] [NameLiteral, value] [Body]
Elements	@Annotation(element1, element2) class Class { }	[ClassDeclaration, Class] [Annotation] [Name, Annotation] [Elements] [Element] [NameLiteral, element1] [Element] [NameLiteral, element2] [Body]
Ellipsis	void method(int... parameters) { }	[MethodDeclaration, method] [VoidType] [Parameters] [Parameter, parameters] [PrimitiveType, int] [Ellipsis] [Body]
ElseIfStatement	if (condition1) { } else if (condition2) { }	[IfStatement] [Condition] [NameLiteral, condition1] [Block] [ElseIfStatement] [Condition] [NameLiteral, condition2] [Block]
ElseStatement	if (condition1) { } else { }	[IfStatement] [Condition] [NameLiteral, condition1] [Block] [ElseStatement] [Block]

EnumConstantDeclaration	enum Enum {
	  CONSTANT
	}
		[EnumDeclaration, Enum]
		[Body]
		[EnumConstantDeclaration, CONSTANT]

EnumDeclaration	enum Enum { }	[EnumDeclaration, Enum] [Body]
ExportsDirective	exports module1;	[ExportsDirective] [Name, module1]

Typ uzlu	Příklad kódu	Příklad AST
Extends	class Class extends Class1 { }	[ClassDeclaration, Class] [Extends] [ReferenceType] [Name, Class1] [Body]
FieldAccessExpression	object.field = 1;	[AssignmentExpression, =] [FieldAccessExpression] [NameLiteral, object] [Name, field] [IntegerLiteral, 1]
FieldDeclaration	int field;	[FieldDeclaration] [PrimitiveType, int] [Variable, field]
FinallyBlock	try { } finally { }	[TryStatement] [Block] [FinallyBlock] [Block]
FloatLiteral	variable = 1.0;	[AssignmentExpression, =] [NameLiteral, variable] [FloatLiteral, 1.0]
ForEachCollectionStatement	for (int variable : collection) { }	[ForEachStatement] [ForEachVariableStatement] [VariableDeclaration] [PrimitiveType, int] [Variable, variable] [ForEachCollectionStatement] [NameLiteral, collection] [Block]
ForEachStatement	for (int variable : collection) { }	[ForEachStatement] [ForEachVariableStatement] [VariableDeclaration] [PrimitiveType, int] [Variable, variable] [ForEachCollectionStatement] [NameLiteral, collection] [Block]
ForEachVariableStatement	for (int variable : collection) { }	[ForEachStatement] [ForEachVariableStatement] [VariableDeclaration] [PrimitiveType, int] [Variable, variable] [ForEachCollectionStatement] [NameLiteral, collection] [Block]
ForIncrementStatement	for (int variable; condition; variable++) { }	[ForStatement] [ForInitializationStatement] [VariableDeclaration] [PrimitiveType, int] [Variable, variable] [ForTerminationStatement] [NameLiteral, condition] [ForIncrementStatement] [UnaryExpression, ++] [NameLiteral, variable] [Block]

Typ uzlu	Příklad kódu	Příklad AST
ForInitializationStatement	for (int variable; condition; variable++) { }	[ForStatement] [ForInitializationStatement] [VariableDeclaration] [PrimitiveType, int] [Variable, variable] [ForTerminationStatement] [NameLiteral, condition] [ForIncrementStatement] [UnaryExpression, ++] [NameLiteral, variable] [Block]
ForStatement	for (int variable; condition; variable++) { }	[ForStatement] [ForInitializationStatement] [VariableDeclaration] [PrimitiveType, int] [Variable, variable] [ForTerminationStatement] [NameLiteral, condition] [ForIncrementStatement] [UnaryExpression, ++] [NameLiteral, variable] [Block]
ForTerminationStatement	for (int variable; condition; variable++) { }	[ForStatement] [ForInitializationStatement] [VariableDeclaration] [PrimitiveType, int] [Variable, variable] [ForTerminationStatement] [NameLiteral, condition] [ForIncrementStatement] [UnaryExpression, ++] [NameLiteral, variable] [Block]

GuardedPattern	switch (value) {
	case int value1 && value1 == 1 ->1;[NameLiteral, value]
	}
		[SwitchExpression]
		[CaseRule]
		[CaseRuleValue]
		[GuardedPattern]
		[GuardedPattern, value1]
		[PrimitiveType, int]
		[BinaryExpression, ==]
		[NameLiteral, value1]
		[IntegerLiteral, 1]
		[IntegerLiteral, 1]

IfStatement	if (condition) { }	[IfStatement] [Condition] [NameLiteral, condition] [Block]
Implements	class Class implements Interface { }	[ClassDeclaration, Class] [Implements] [ReferenceType] [Name, Interface] [Body]
Import	import Class;	[Import] [Name, Class]

Typ uzlu	Příklad kódu	Příklad AST
Initialization	int field = 1;	[FieldDeclaration] [PrimitiveType, int] [Variable, field] [Initialization] [IntegerLiteral, 1]
InstanceOfExpression	variable = object instanceof Class;	[AssignmentExpression, =] [NameLiteral, variable] [InstanceOfExpression] [NameLiteral, object] [ReferenceType] [Name, Class]
IntegerLiteral	variable = 1;	[AssignmentExpression, =] [NameLiteral, variable] [IntegerLiteral, 1]
InterfaceDeclaration	interface Interface { }	[InterfaceDeclaration, Interface] [Body]
IntersectionType	variable = (Class1 & Class2)object;	[AssignmentExpression, =] [NameLiteral, variable] [CastExpression] [IntersectionType] [ReferenceType] [Name, Class1] [ReferenceType] [Name, Class2] [NameLiteral, object]
JavadocComment	/** * Javadoc comment */	[JavadocComment, /** * Javadoc comment */]
LabeledBlock	label: { }	[LabeledBlock, label]
LambdaExpression	variable = parameter ->{ };	[AssignmentExpression, =] [NameLiteral, variable] [LambdaExpression] [Parameters] [Parameter, parameter] [Body]
LineComment	// Line comment	[LineComment, // Line comment]
MethodCallExpression	method();	[MethodCallExpression] [Name, method] [Arguments]
MethodDeclaration	void method() { }	[MethodDeclaration, method] [VoidType] [Parameters] [Body]
MethodReferenceExpression	variable = Class::method;	[AssignmentExpression, =] [NameLiteral, variable] [MethodReferenceExpression] [NameLiteral, Class] [Name, method]
Modifier	public static class Class { }	[ClassDeclaration, Class] [Modifier, public] [Modifier, static] [Body]
ModuleDeclaration	module module { }	[ModuleDeclaration, module] [Body]

Typ uzlu	Příklad kódu	Příklad AST
MultilineComment	/* Multi-line comment */	[MultilineComment, /* Multi-line comment */]
NameLiteral	variable = 1;	[AssignmentExpression, =] [NameLiteral, variable] [IntegerLiteral, 1]
NullLiteral	variable = null;	[AssignmentExpression, =] [NameLiteral, variable] [NullLiteral]
ObjectCreation	variable = new Object();	[AssignmentExpression, =] [NameLiteral, variable] [ObjectCreation] [ReferenceType] [Name, Object] [Arguments]
OpensDirective	opens module1;	[OpensDirective] [Name, module1]
Package	package packageName;	[Package] [Name, packageName]
Parameter	void method(int parameter) { }	[MethodDeclaration, method] [VoidType] [Parameters] [Parameter, parameter] [PrimitiveType, int] [Body]
Parameters	void method(int parameter1, int parameter2) { }	[MethodDeclaration, method] [VoidType] [Parameters] [Parameter, parameter1] [PrimitiveType, int] [Parameter, parameter2] [PrimitiveType, int] [Body]
ParenthesesExpression	variable = (1 + 2);	[AssignmentExpression, =] [NameLiteral, variable] [ParenthesesExpression] [BinaryExpression, +] [IntegerLiteral, 1] [IntegerLiteral, 2]
Pattern	variable = object instanceof Object obj;	[AssignmentExpression, =] [NameLiteral, variable] [InstanceOfExpression] [NameLiteral, object] [Pattern, obj] [ReferenceType] [Name, Object]
Permits	class Class permits Interface { }	[ClassDeclaration, Class] [Permits] [ReferenceType] [Name, Interface] [Body]
PrimitiveType	int variable;	[VariableDeclaration] [PrimitiveType, int] [Variable, variable]

Typ uzlu	Příklad kódu	Příklad AST
ProvidesDirective	provides module1 with module2;	[ProvidesDirective] [Name, module1] [With] [Name, module2]
RecordComponent	record Record(int component) { }	[RecordDeclaration, Record] [RecordComponents] [RecordComponent, component] [PrimitiveType, int] [Body]
RecordComponents	record Record(int component1, int component2) { }	[RecordDeclaration, Record] [RecordComponents] [RecordComponent, component1] [PrimitiveType, int] [RecordComponent, component2] [PrimitiveType, int] [Body]
RecordDeclaration	record Record() { }	[RecordDeclaration, Record] [RecordComponents] [Body]
ReferenceType	Object variable;	[VariableDeclaration] [ReferenceType] [Name, Object] [Variable, variable]
RequiresDirective	requires module1;	[RequiresDirective] [Name, module1]
Resource	try (resource) { }	[TryWithResourcesStatement] [ResourcesStatement] [Resource] [NameLiteral, resource] [Block]
ResourcesStatement	try (resource1; resource2) { }	[TryWithResourcesStatement] [ResourcesStatement] [Resource] [NameLiteral, resource1] [Resource] [NameLiteral, resource2] [Block]
ReturnStatement	return 1;	[ReturnStatement] [IntegerLiteral, 1]
StringLiteral	variable = "str";	[AssignmentExpression, =] [NameLiteral, variable] [StringLiteral, "str"]
SuperConstructorCallExpression	super();	[SuperConstructorCallExpression] [Arguments]
SuperExpression	super.method();	[MethodCallExpression] [SuperExpression] [Name, method] [Arguments]

SwitchExpression	switch(value) {
	case 1 ->1;
	}
		[SwitchExpression]
		[NameLiteral, value]
		[CaseRule]
		[CaseRuleValue]
		[IntegerLiteral, 1]
		[IntegerLiteral, 1]

Typ uzlu	Příklad kódu	Příklad AST

SwitchStatement	switch(value) {
	  case 1: break;
	}
		[SwitchStatement]
		  [NameLiteral, value]
		  [Case]
		  [CaseValue]
		  [IntegerLiteral, 1]
		  [BreakExpression]

SynchronizedStatement	synchronized (expression) { }	[SynchronizedStatement]
		  [NameLiteral, expression]
		  [Block]

TextBlockLiteral	variable = """
	text""";
	
		[AssignmentExpression, =]
		  ]NameLiteral, variable]
		  [TextBlockLiteral, """
		  text"""]

ThisConstructorCallExpression	this();	[ThisConstructorCallExpression] [Arguments]

ThisExpression	this.method();	[MethodCallExpression]
		  [ThisExpression]
		  [Name, method]
		  [Arguments]

ThrowExpression	throw new Exception();	[ThrowExpression]
		  [ObjectCreation]
		  [ReferenceType]
		  [Name, Exception]
		  [Arguments]

ThrowsClause	void method() throws Exception {
	}
		[MethodDeclaration, method]
		  [VoidType]
		  [Parameters]
		  [ThrowsClause]
		  [ReferenceType]
		  [Name, Exception]
		  [Body]

To	exports module1 to module2;	[ExportsDirective]
		  [Name, module1]
		  [To]
		  [Name, module2]

TryStatement	try { }	[TryStatement] [Block]

TryWithResourcesStatement	try (resource) { }	[TryWithResourcesStatement]
		  [ResourcesStatement]
		  [Resource]
		  [NameLiteral, resource]
		  [Block]

TypeArgument	void method(List<Type>parameter)
	{ }
		[MethodDeclaration, method]
		  [VoidType]
		  [Parameters]
		  [Parameter, parameter]
		  [ReferenceType]
		  [Name, List]
		  [TypeArguments]
		  [TypeArgument]
		  [ReferenceType]
		  [Name, Type]
		  [Body]

Typ uzlu	Příklad kódu	Příklad AST
TypeArguments	void method(Map<Type1, Type2>parameter) { }	[MethodDeclaration, method] [VoidType] [Parameters] [Parameter, parameter] [ReferenceType] [Name, Map] [TypeArguments] [TypeArgument] [ReferenceType] [Name, Type1] [TypeArgument] [ReferenceType] [Name, Type2] [Body]
TypeParameter	class Class<Type>{ }	[ClassDeclaration, Class] [TypeParameters] [TypeParameter, Type] [Body]
TypeParameters	class Class<Type1, Type2>{ }	[ClassDeclaration, Class] [TypeParameters] [TypeParameter, Type1] [TypeParameter, Type2] [Body]
UnaryExpression	variable++;	[UnaryExpression, ++] [NameLiteral, variable]
UnionType	try { } catch (Exception1 | Exception2 e) { }	[TryStatement] [Block] [CatchClause, e] [UnionType] [ReferenceType] [Name, Exception1] [ReferenceType] [Name, Exception2] [Block]
UsesDirective	uses module1;	[UsesDirective] [Name, module1]
VarType	var variable = 1;	[VariableDeclaration] [VarType] [Variable, variable] [Initialization] [IntegerLiteral, 1]
Variable	int variable = 1;	[VariableDeclaration] [PrimitiveType, int] [Variable, variable] [Initialization] [IntegerLiteral, 1]
VariableDeclaration	int variable = 1;	[VariableDeclaration] [PrimitiveType, int] [Variable, variable] [Initialization] [IntegerLiteral, 1]
VoidType	void method() { }	[MethodDeclaration, method] [VoidType] [Parameters] [Body]

Typ uzlu	Příklad kódu	Příklad AST
WhileStatement	while(condition) { }	[WhileStatement] [Condition] [NameLiteral, condition] [Block]
Wildcard	void method(List<?>parameter) { }	[MethodDeclaration, method] [VoidType] [Parameters] [Parameter, parameter] [ReferenceType] [Name, List] [TypeArguments] [TypeArgument] [Wildcard] [Body]
WildcardExtends	void method(List<? extends Object>parameter) { }	[MethodDeclaration, method] [VoidType] [Parameters] [Parameter, parameter] [ReferenceType] [Name, List] [TypeArguments] [TypeArgument] [Wildcard] [WildcardExtends] [ReferenceType] [Name, Object] [Body]
WildcardSuper	void method(List<? super Object>parameter) { }	[MethodDeclaration, method] [VoidType] [Parameters] [Parameter, parameter] [ReferenceType] [Name, List] [TypeArguments] [TypeArgument] [Wildcard] [WildcardSuper] [ReferenceType] [Name, Object] [Body]
With	provides module1 with module2;	[ProvidesDirective] [Name, module1] [With] [Name, module2]
YieldStatement	yield 1;	[YieldStatement] [IntegerLiteral, 1]

Literatura

[1] Chacon, Scott; Straub, Ben. *Pro Git*. 2nd. 2014. ISBN 1484200772.

[2] O'Sullivan, Bryan; Bryan, O'Sullivan. *Mercurial: The Definitive Guide*. 2009. ISBN 0596800673.

[3] Pilato, Michael. *Version Control With Subversion*. 2004. ISBN 0596004486.

[4] Campbell, Bill; Iyer, Swami; Akbal-Delibas, Bahar. *Introduction to Compiler Construction in a Java World*. 2012. ISBN 1439860882.

[5] Fluri, Beat; Wuersch, Michael; PInzger, Martin; Gall, Harald. Change Distilling: Tree Differencing for Fine-Grained Source Code Change Extraction. *IEEE Trans. Softw. Eng.* 2007, roč. 33, č. 11, s. 725–743. Dostupný také z: ⟨https://doi.org/10.1109/TSE.2007.70731⟩. ISSN 0098-5589.

[6] Chawathe, Sudarshan S.; Rajaraman, Anand; Garcia-Molina, Hector; Widom, Jennifer. Change Detection in Hierarchically Structured Information. *SIGMOD Rec.* 1996, roč. 25, č. 2, s. 493–504. Dostupný také z: ⟨https://doi.org/10.1145/235968.233366⟩. ISSN 0163-5808.

[7] JetBrains. *IntelliJ IDEA Community Edition*. Dostupný také z: ⟨https://github.com/JetBrains/intellij-community⟩.

[8] JetBrains. *Annotations*. Dostupný také z: ⟨https://github.com/JetBrains/java-annotations⟩.

[9] Parr, Terence; Harwell, Sam; Eric, Vergnaud aj. *ANTLR 4*. Dostupný také z: ⟨https://github.com/antlr/antlr4⟩.

[10] Popma, Remko. *Picocli*. Dostupný také z: ⟨https://github.com/remkop/picocli⟩.

[11] Google. *Gson*. Dostupný také z: ⟨https://github.com/google/gson⟩.