ČESKÉ VYSOKÉ UČENÍ TECHNICKÉ V PRAZE
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

Název:	Webová aplikace pro řešení třícestné merge kolize
Student:	Mikuláš Dítě
Vedoucí:	Ing. Vojtěch Jirkovský
Studijní program:	Informatika
Studijní obor:	Web a multimédia
Katedra:	Katedra softwarového inženýrství
Platnost zadání:	Do konce letního semestru 2017/18

Pokyny pro vypracování

1) Popište merge algoritmy používané v současných verzovacích systémech.
2) Seznamte se s nejnovějšími publikovanými merge algoritmy.
3) Navrhněte, jak použít jeden z těchto algoritmů ke spojení abstraktních syntaktických stromů vybraného programovacího jazyka.
4) Navrhněte jednoduchou webovou aplikaci, která bude obsahovat
- uživatelské rozhraní pro zadání kolizního stavu
- řešení vypočítaná zvoleným algoritmem
- uživatelské ohodnocení kvality řešení a
- historii zadaných kolizních stavů
5) Implementujte a publikujte tuto aplikaci na internetu.
6) Otestujte funkčnost a porovnejte efektivitu řešení kolizí oproti současným systémům.

Seznam odborné literatury

Dodá vedoucí práce.

L.S.

Ing. Michal Valenta, Ph.D.
vedoucí katedry

prof. Ing. Pavel Tvrdík, CSc.
děkan

V Praze dne 5. ledna 2017

České vysoké učení technické v Praze

Fakulta informačních technologií

Katedra softwarového inženýrství

Bakalářská práce

Webová aplikace pro řešení třícestné merge kolize

Mikuláš Dítě

Vedoucí práce: Ing. Vojtěch Jirkovský

10. května 2016

Poděkování

Děkuji Ing. Vojtěchu Jirkovskému za vedení této práce, Ing. Miroslavu Hrončokovi za tipy k psaní práce a XeLaTeX šablonu, a Prof. Ing. Adam Heroutovi, Ph.D za články o psaní diplomové práce.

Prohlášení

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

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona, ve znění pozdějších předpisů. V souladu s ust. § 46 odst. 6 tohoto zákona tímto uděluji nevýhradní oprávnění (licenci) k užití této mojí práce, a to včetně všech počítačových programů, jež jsou její součástí či přílohou a veškeré jejich dokumentace (dále souhrnně jen „Dílo“), a to všem osobám, které si přejí Dílo užít. Tyto osoby jsou oprávněny Dílo užít jakýmkoli způsobem, který nesnižuje hodnotu Díla a za jakýmkoli účelem (včetně užití k výdělečným účelům). Toto oprávnění je časově, teritoriálně i množstevně neomezené. Každá osoba, která využije výše uvedenou licenci, se však zavazuje udělit ke každému dílu, které vznikne (byť jen zčásti) na základě Díla, úpravou Díla, spojením Díla s jiným dílem, zařazením Díla do díla souborného či zpracováním Díla (včetně překladu), licenci alespoň ve výše uvedeném rozsahu a zároveň zpřístupnit zdrojový kód takového díla alespoň srovnatelným způsobem a ve srovnatelném rozsahu, jako je zpřístupněn zdrojový kód Díla.

V Praze dne 10. května 2016 ……

České vysoké učení technické v Praze

Fakulta informačních technologií

Odkaz na tuto práci

DÍTĚ, Mikuláš. *Webová aplikace pro řešení třícestné merge kolize*. Bakalářská práce. Praha: České vysoké učení technické v Praze, Fakulta informačních technologií, 2017. Dostupný také z WWW: ⟨`https://gitlab.fit.cvut.cz/ditemiku/bp`⟩.

Abstrakt

Tato práce dokumentuje soudobý stav algoritmů pro začleňování změn a předkládá přehled nejnovějších publikovaných diff algoritmů. Verzovací systémy nyní pracují na úrovni řádků. Dále tato práce prezentuje nový způsob, jak verzovat soubory přes abstraktní syntaktický strom. Konkrétně byla implementována webová aplikace, která zvoleným algoritmem řeší kolize při spojování změn a vznikla testovací sada, na které je řešení porovnáno s existujícím verzovacím programem.

Klíčová slova rozšíření pro git, slučování stromů, abstraktní syntaktický strom php, řešení kolize spojování změn, automatizace

Abstract

This work investigates state of merge algorithms in contemporary versioning software and lists multiple state-of-the-art algorithms. Most versioning algorithms in used have line granularity. This work proposes a new merge algorithm based on abstract syntax trees. This algorithm was implemented and published as a web application, available to the public. Alongside this application, I created a minimal benchmark for validating merge results.

Keywords git extension, tree merging, PHP abstract syntax tree, merge resolution, automation

Obsah

Úvod

Při programování v týmu je často potřeba začlenit do zdrojových kódů dvě kolizní změny. Současné algoritmy vycházejí z porovnávání řádků a jsou rychlé. S nárůstem výpočetního výkonu jsou ale nyní k dispozici i algoritmy sofistikovanější, které zvládnou automaticky vyřešit víc konfliktů.

Implementací algoritmů pracujících nad stromy se programátorům usnadní začleňování změn zdrojových souborů. Ruční řešení kolizí je zdlouhavé a náchylné na vznik těžko detekovatelných chyb, čemuž chceme předejít.

Jako vývojář na tento problém narážím denně, stejně tak moji kolegové. Automatizace programátorům nejenom ušetří práci, ale především usnadní a urychlí začleňování změn od jednotlivých autorů do sdíleného repozitáře.

Kapitola 1

Soudobé merge algoritmy a publikované alternativy

V této kapitole je nejprve popsán společný teoretický základ všech algoritmů. Navazuje shrnutí implementací současných diff (porovnávacích) algoritmů a jejich využití ve verzovacích nástrojích. Následně je rozebrán základ pro jemnější a tedy přesnější porovnávání na úrovni abstraktních syntaktických stromů.

1.1 Teoretický základ pro textové algoritmy

V dalším textu bude použito následujících termínů [1]:

abeceda Σ: Konečná množina symbolů.

řetězec nad abecedou: Konečná posloupnost symbolů abecedy.

formální jazyk nad abecedou: Množina všech posloupností symbolů dané abecedy.

Nejdelší společná podsekvence

V originále *LCS, Longest common subsequence.*

Tento problém a jeho algoritmické řešení jsou základem řádkového porovnávání souborů. Je využit například v Unixovém nástroji `diff` [2], od kterého se odvíjí složitější porovnávací a verzovací algoritmy. LCS je speciální případ Problému editační vzdálenosti (často také zvaný konkrétně Levenshteinova vzdálenost) [3].

Formálně je problém poprvé definován Hirschbergem takto [4]: řetězec $c = c_1c_2 \cdots c_p$ je *podsekvence* řetězce $a = a_1a_2 \cdots a_m$ právě tehdy, pokud existuje přiřazení $F : \{1, 2, \ldots, p\} \to \{1, 2, \ldots, m\}$ takové, že $F(i) = k$ pouze pokud $c_i = a_k$ a F je monotóní striktně rostoucí funkce. Odstraněním $m - p$ symbolů (ne nutně po sobě jdoucích) z a vznikne c. Příkladem podsekvence řetězce `fabrika` jsou `frk` nebo `brka`.

Funkce $\text{Ssq}(c)$ je definována jako zobrazení řetězců do množiny řetězců, kde pro parametr c funkce Ssq vrací množinu všech podřetězců c.[†]

Společná podsekvence c řetězců a, b je právě taková podsekvence, která je zároveň podsekvence a a podsekvence b. Formálně $\text{Ssq}(a, b) = \text{Ssq}(a) \cap \text{Ssq}(b)$. Definuji obecnou množinovou variantu $\text{Ssq}(A) = \bigcap_i \text{Ssq}(A_i)$, kde A je množina řetězců.

Nejdelší společná podsekvence je množina všech společných podsekvencí, pro které neexistuje žádná delší podsekvence. Formálně:

$$\text{LCS}(A) = \{c \in \text{Ssq}(A); \nexists\, c_{longer} \in \text{Ssq}(A) : |c_{longer}| > |c|\}$$

kde A je množina řetězců a $|c|$ značí počet znaků řetězce c.

Maier [5] zavádí problém *Yes/No LCS* (*YNLCS*).
Existuje pro dané $k \in \mathbb{N}$ a množinu řetězců R aspoň jedno c z $LCS(R)$ takové, že $|c| >= k$? Formálně:

[†]První výskyt množinové definice jsem nalezl v Maier 1978 [5] a dále v Hsu a Du 1982 [6]. Hirschberg v své práci [4] tuto definici ještě nepoužívá.

$$\text{YNLCS}(R, k) = \begin{cases} 1 & \exists\, c \in \text{LCS}(R) : |c| \geq k \\ 0 & \text{jinak} \end{cases}$$

kde R je množina řetězců nad stejnou abecedou a $k \in \mathbb{N}$ je požadovaná minimální délka podsekvence.

Obecné řešení YNLCS problému – pro R s velikostí větší než 2 – má třídu složitosti NP-úplné [5]. Pro tuto práci to znamená, že nelze efektivně přímo využít LCS pro více velkých vstupů (větev 1, společný základ, větev 2).

Algoritmické řešení LCS

Podle *A survey of LCS algorithms* [7] se typicky tento problém neřeší obecně. Zjednodušuje se na dva vstupní řetězce LCS ($\{a, b\}$), nebo na Nejdelší rostoucí podposloupnost, kterou se ale tato práce dál nezabývá, protože nemá v kontextu verzování praktické využití.

Nejrychlejší známý algoritmus od Masek a Paterson [3] má časovou složitost $O(n^2/\log(n))$, ale vyžaduje řetězce nad konečnou abecedou.

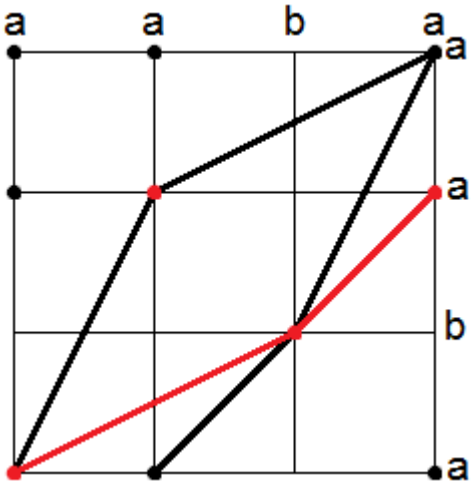
Naivní řešení

$A_i := i$. řádek prvního řetězce $\mathcal{A}$
$B_j := j$. řádek druhého řetězce $\mathcal{B}$
$P(i, j) :=$ délka LCS do i. řádku $\mathcal{A}$ a j. řádku $\mathcal{B}$
function $P(i, j)$
    if $i = 0 \vee j = 0$ **then**
        return 0
    else if $A_i = B_j$ **then**
        return $1 + P(i - 1, j - 1)$
    else ▷ $A_i \neq B_i$
        return $\max(P(i - 1, j), P(i, j - 1))$
    end if
end function

Tento algoritmus běží s časovou i prostorovou náročností $O(mn)$, kde m je počet řádků $\mathcal{A}$ a n počet řádků $\mathcal{B}$.

Hunt–McIlroy algorithm - Essential Matches

Hledá nezbytné shody (*essential matches*) neboli k-kandidáty (k-*candidates*), které definuje jako pár $\{i, j\}$ pro který platí, že $A_i = B_j$ a současně $P(i, j) > \max(P(i-1, j), P(i, j-1))$. Efektivita tohoto řešení vychází z toho, že výsledné LCS může být tvořeno pouze k-kandidáty. [2]



Obrázek 1.1: Znázornění redukce možných cest pro LCS řetězců aaba a abaa. Černé body a čáry označují postup naivního řešení. Červené body označují k-kandidáty. Červenou čarou je zvýrazněno jedno z možných řešení. Ilustrace převzata z Wiki Commons [8].

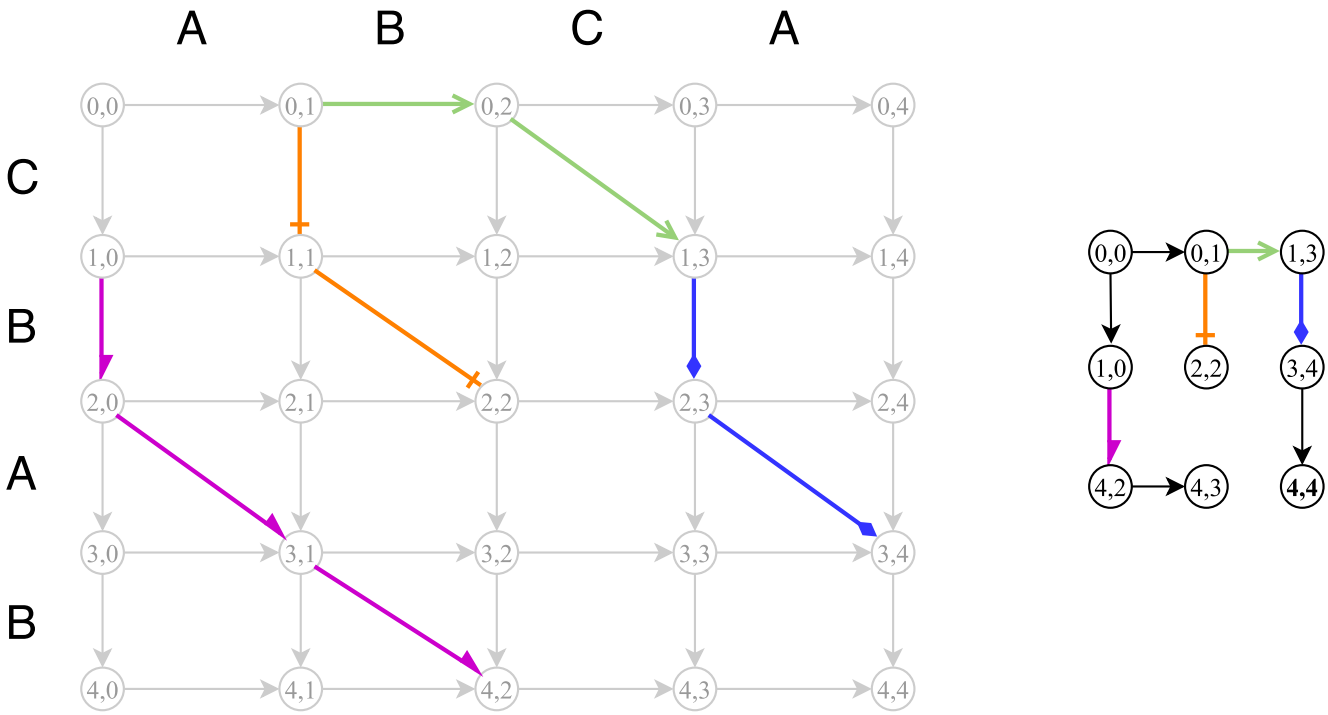
Originální pseudokód je publikován v *An algorithm for differential file comparison* [2, strana 3]. Má časovou složitost $O(n \log m)$, prostorovou $O(mn)$. Ve všech ohledech je překonán Myersovým algoritmem.

Myers

Myersův algoritmus dále rozšiřuje Huntův algoritmus.

Myers definuje N jako součet délek vstupních řetězců a D jako délku nejkratšího editačního skriptu. L je délka LCS a platí tedy $D = 2(N - L)$. Jeho algoritmus je vystaven na předpokladu, že $O(ND)$ je pro typické použití časově efektivnější, než $O(NL + NlogN)$ [9].

Tento iterativní algoritmus je postaven na úvaze, že nejsložitěji se k výsledku můžeme dobrat v $m + n$ operacích, kde m a n jsou délky vstupů, a to pokud n znaků smažeme a m přidáme (BÚNO). Toto je ilustrováno na obrázku 1.1. Aby vznikl co nejmenší výstup, je potřeba použít shodné znaky v obou vstupech: to znázorňují diagonální přechody v diagramu. Algoritmus pracuje s tabulkou přechodů, kde si každá buňka pamatuje odkud se na ni došlo a na jaké pozici v LCS se lze dostat. Postupně se od nejkratší vzdálenosti prochází možnosti „pohyb vpravo“ a „pohyb dolů“ a vyplňuje se tabulka přechodů. Existuje-li diagonální přechod, jako cíl přechodu se vyplní až poslední diagonála (příkladem je přechod `[1,0] - [4,2]`). Pokud se při průchodu zjistí, že buňka je již vyplněna a obsahuje nižší cílovou pozici, je přepsána. Ukázka je na diagramu 1.2.



Obrázek 1.2: Myersův algoritmus aplikovaný na vstupy abca a cbab. Na pravé straně je tabulka přechodů. Odpovídající hrany jsou barevně označeny.

Pseudokód je v publikovaném článku [9].

Porovnávání hashů řádků Hunt doporučuje v kapitole *3. Hashing* neporovnávat přímo řádky, ale jejich hashe [2]. To může vést na nalezení falešně pozitivní shody. Pro hashe s adekvátně velkým výstupy je to ale nepravděpodobné, a po nalezení výsledku lze tyto shody odfiltrovat.

1.1.1 Využití LCS v algoritmu `diff`

Hunt–McIlroy

Poprvé publikován jako *An algorithm for differential file comparison* [2], tento algoritmus je rozšířením řešení LCS.

Staví nad dynamickým řešením porovnávající dva soubory s m, resp. n řádky, které běží v čase $O(mn)$ a má $O(mn)$ prostorovou náročnost.

Tento algoritmus je veskrze překonán algoritmem Myersovým.

Myers

Jako diff lze rovnou použít Myersův LCS algoritmus popsaný v kapitole 1.1.1. Při vyplnění tabulky přechodů je získána i cesta, kterou se k LCS došlo. Pohyb vpravo reprezentuje smazání symbolu (typicky řádku) z prvního vstupu, pohyb dolů odpovídá vložení symbolu z druhého vstupu. V literatuře se pojem Myers LCS a Myers diff volně zaměňuje.

1.1.2 Algoritmus a program diff3

Aplikace, kterou má tato práce za cíl vytvořit, přijímá i data ve formátu `diff3` [10]. Tento formát byl vybrán především proto, že jej může generovat Git v případě kolize. Další výhodou toho formátu je, že v jednom souboru obsahuje všechny tři vstupy a uživatel si tedy nemůže splést terminologii v naší aplikaci. Navíc je vstup kompaktní: nezměněné části obsahuje pouze jednou, což by ale vyřešila i komprese.

Formát `diff3` je výstupem stejnojmenné GNU utility `diff3`, kterou naprogramoval Randy Smith v roce 1988 a byla zveřejněna v rámci balíčku `diffutils` [11]. Tato utilita byla akademicky popsána v roce 2007 v článku *A Formal Investigation of Diff3* [12].

1.2 Současné implementace diff algoritmů

GNU diff

Nástroj `diffutils` [11] je vznikl jako první. Implementací vytvořil McIlroy společně s Huntem (1.1.1).

Později správci `diffutils` přešli na Myersovu alternativu.

V současné době není tato knihovna – s výjimkou své rozšířenosti – dobrá volba. Neobsahuje žádné pokročilé algoritmy a slouží pouze jako základ či záložní algoritmus (*fallback*) pro ostatní programy.

libxdiff

Knihovnu vytvořil Davide Libenzi [13] a pro tuto práci je obzvlášť zajímavá, protože obsahuje celou řadu nových přístupů a řešení. V této knihovně jsou pokročilé diff algoritmy jako Patience diff (1.4.2) a Histogram diff (1.4.2). Tyto algoritmy jsou detailně rozebrány v kapitole věnující se verzovacímu systému Git.

Google Diff-Match-Patch

Knihovna implementuje Myersův algoritmus. [14]

Zajímavá na této knihovně je optimalizace od Mike Slemmer v `cpp` kódu [15], která pro složitější porovnávané soubory (alespoň 100 řádků) převede řádky na jednodušší interní strukturu a potom je mapuje zpět.

Další inspirativní optimalizací je oříznutí identického prefixu a postfixu v obou porovnávaných textech [16].

1.3 Definice merge algoritmu

Definuji jednoznačně merge algoritmus jako funkci, která dostává tři řetězce ze stejné abecedy (významem jsou to zdrojový kód a, zdrojový kód b a společná báze). Výstupem je jeden řetězec, který zahrnuje změny z obou zdrojových kódů, případně chyba, pokud změny spojit nelze.

V dokumentaci verzovacích nástrojů je často jako merge označován složitější algoritmus, který hledá společné předky, identifikuje tzv. cherry-picks, … a nakonec používají merge podle mé definice. V těchto případech na to je v práci upozorněno a je zdokumentován pouze relevantní kód.

1.4 Současné verzovací nástroje

Pro studii běžně používaných merge algoritmů se zaměřím na tyto verzovací systémy: Apache Subversion (SVN), Mercurial a Git. Zdrojové soubory proprietárního software Microsoft Team Foundation Server, Perforce Helix a další zkoumat nemůžu. Dále se nebudu věnovat méně známým open-source nástrojům jako například Bazaar, který obsahoval první implementaci Patience Diff. Tento algoritmus je ale popsán v kapitole o Gitu 1.4.2.

1.4.1 Algoritmy v Apache Subversion

SVN je zástupcem Client-Server verzovacího modelu. Repozitář s kódem existuje pouze jeden – centrální – a bez připojení k němu nelze vytvářet nové revize/commity. Situace vyžadující merge nastane při explicitním požadavku na spojení dvou větví.

Do verze 1.5 SVN používalo pouze 2-way merge, protože chyběly meta-informace pro nalezení společného předka. Po přidání *Merge tracking* [17] byl v repozitáři každý merge zaznamenán.

Ve vyšších verzích jsou v `mergeinfo` k dispozici informace o poslední společné revizi, které jsou pro automatické řešeni kolizí a 3-way merge nezbytné.

SymmetricMerge

SVN v dokumentaci tvrdí, že používá algoritmus SymmetricMerge [18]. Ten interně volá metodu `three_way_merge(m.base, m.tip, m.target)`, ke které se mi nepodařilo najít dokumentaci ani zdrojový kód. Podle signatury jde ale o funkci merge, která odpovídá definici v této práci. Podle vývojáře Subversion je tato funkce přímo převedena na volání aplikace `diff3` [19].

SVN tedy používá `diff3` algoritmus.

1.4.2 Algoritmy v Git

Podobně jako SVN i Git má zmatek v názvosloví a jako merge označuje celou řadu algoritmů. Takzvané *merge strategies* [20] označují způsoby, jak nalézt samotné řetězce, nad kterými se 3-way-merge bude volat. Analogie v SVN je SymmetricMerge.

Na rozdíl od SVN lze v Gitu změnit (nastavením parametru `--diff-algorithm=`), jaký 3-way-merge se interně má použít. Podle současné dokumentace (2.11) jsou k dispozici tyto algoritmy: myers, minimal, patience a histogram.

První implementace Gitu pouze vytvářela nový systémový proces `diff` [21]. Toto typicky volalo GNU implementaci.

Git nyní používá knihovnu `libxdiff`, která je jiná než GNU diff.

Myers diff

Výchozí nastavení Gitu. Snaží se optimalizovat editační vzdálenost změn. Algoritmus je implementován přesně podle Myersova článku [9], popsaného v kapitole 1.1.1.

Patience diff (Bram's diff)

K tomuto algoritmu neexistuje žádný publikovaný článek a Bram Cohen (mj. autor BitTorrent protokolu) ho pouze představil na svém blogu [22] a později lehce zdokumentoval [23]. Vznikl implementací ve verzovacím nástroji Bazaar a o dva roky později byl přidán do Gitu [24] skrze knihovnu `libxdiff`.

1. Dokud jsou odshora řádky identické na levé a pravé straně, posun o řádek dolů.
2. Dokud jsou odspodu řádky identické na levé a pravé straně, posun o řádek nahoru.
3. Pro všechny unikátní řádky (takové co se na každé straně vyskytují právě jednou) se vytvoří LCS.
4. Kroky 1-2 se opakují pro každou sekci mezi namapovanými řádky.

Předností tohoto algoritmu je seskupování společných změn, což Myersův algoritmus nedělá.

Na testovací sadě o velikosti 3000 běžel Myers diff 1.93 (1.75+0.17) sekund a Patience diff 2.25 (2.07+0.16) sekund. Patience tedy není výrazně pomalejší a je vhodný pro praktické použití. Výsledky jsou převzaty z mailing listu Gitu [25].

```
diff --git a/a.php b/b.php
index f002945..bd6a48b 100644
--- a/a.php
+++ b/b.php
@@ -1,11 +1,11 @@
 <?php

-function beta()
-{
-       return cos(time());
-}
-
 function alpha()
 {
        return sin(time());
 }
+
+function beta()
+{
+       return cos(time());
+}
```

(1) Ukázkový výstup Patience diff

```
diff --git a/a.php b/b.php
index f002945..bd6a48b 100644
--- a/a.php
+++ b/b.php
@@ -1,11 +1,11 @@
 <?php

-function beta()
+function alpha()
 {
-       return cos(time());
+       return sin(time());
 }

-function alpha()
+function beta()
 {
-       return sin(time());
+       return cos(time());
 }
```

(2) Myers diff

Kód 1.1: Rozdíl mezi výstupem Patience diff (vlevo) a Myers diff. Levý diff lépe reprezentuje záměr změny: přesunutí funkce `beta` za `alpha`.

Histogram diff

Jde o rozšiřující implementaci Patience diff, původně v Jgit [26] a později v Gitu [27].

1. Pro levou stranu se vytvoří histogram výskytů jednotlivých řádků.

2. Pro každý řádek z pravé strany se najde počet výskytů v předpočítaném levém histogramu.

3. Vybere se taková LCS, která má celkem v levé straně nejmenší počet výskytů.

4. Kolem LCS se vstup rozdělí a rekurzivně se algoritmus volá pro sekce před a po LCS.

Výstup je často stejný jako u Patience diff. Rozdíl může nastat pouze pokud se v kódu vyskytuje více neunikátních řádků. Hlavním rozdílem oproti Patience je lepší výkon. Podle stejného měření jako výše běžel Histogram diff 1.90 (1.74+0.15) sekund, tedy dokonce o něco kratší dobu, než Myers diff.

Heuristiky

Kromě samotných diff algoritmů ovlivňují přehlednost výstupu tzv. *sliders* (posuvníky). Jde o situace, kde kraje editačního skriptu lze cyklicky posouvat, například u složených závorek mezi těly metod. Situace je přehledně viditelná na ukázce 1.2.
Git od verze 2.11 implementuje postprocessing heuristiku, která správně opraví 97,8 % těchto situací [28].

```
@@ -2188,12 +1996,6 @@                @@ -2188,12 +1996,6 @@
                 return dir->nr;                          return dir->nr;

         /*                                      -       /*
-         * Stay on the safe side.               -        * Stay on the safe side.
-         * if read_directory has                -        *  if read_directory has
-         */                                     -        */
-        clear_sticky(dir);                      -       clear_sticky(dir);
-                                                -
-        /*                                              /*
          * exclude patterns are                         * exclude patterns are
          * create_simplify                              * create_simplify
          * subset of positive                           * subset of positive
```

Kód 1.2: Ukázka špatně zarovnaného editačního skriptu (vlevo) a posunutého skriptu o jeden řádek. Výstup vpravo je uznávaný jako čitelnější. Oba skripty nicméně vedou na identický výsledek.

1.4.3 Algoritmy v Mercurial

Mercurial využívá c implementaci Pythonové knihovny `difflib` [29]. Toto je především z historického důvodu, protože samotný Mercurial je naprogramovaný v Pythonu.

Cílem této knihovny je spočítat uživatelsky přívětivý a pohledný diff. Liší se od UNIXového `diff` důrazem na nejdelší *spojitou* a „nezašpiněnou" (whitespace, ...) podsekvenci. Windowsový nástroj `windiff` má jinou zajímavou vlastnost a to seskupování jedinečných elementů z každé věty. Tyto dva přístupy dohromady dávají intuitivnější výsledek. (Volně přeloženo z komentářů originálního zdrojového kódu [30].)

Algoritmus použitý v `difflib` má $O(N^2)$ časovou složitost, kde N je počet vstupních řádků [31].

1.5 Teorie pro nové diff/merge algoritmy

Současné algoritmy používané ve verzovacích systémech nedostačují. Manuální řešení kolizí je náchylné na chybu a nedá se běžně reprodukovat (výjimkou je git, který utilitou RERERE umí řešení kolizí přehrávat opakovaně). Programátoři se typicky snaží kolizím vyhnout a rozdělují si podle toho v týmu práci, omezují svou práci s větvemi a podobně. Chytřejší algoritmy, popsané v následujících dvou kapitolách, nabízí alternativní přístup k 3-way merge problému.

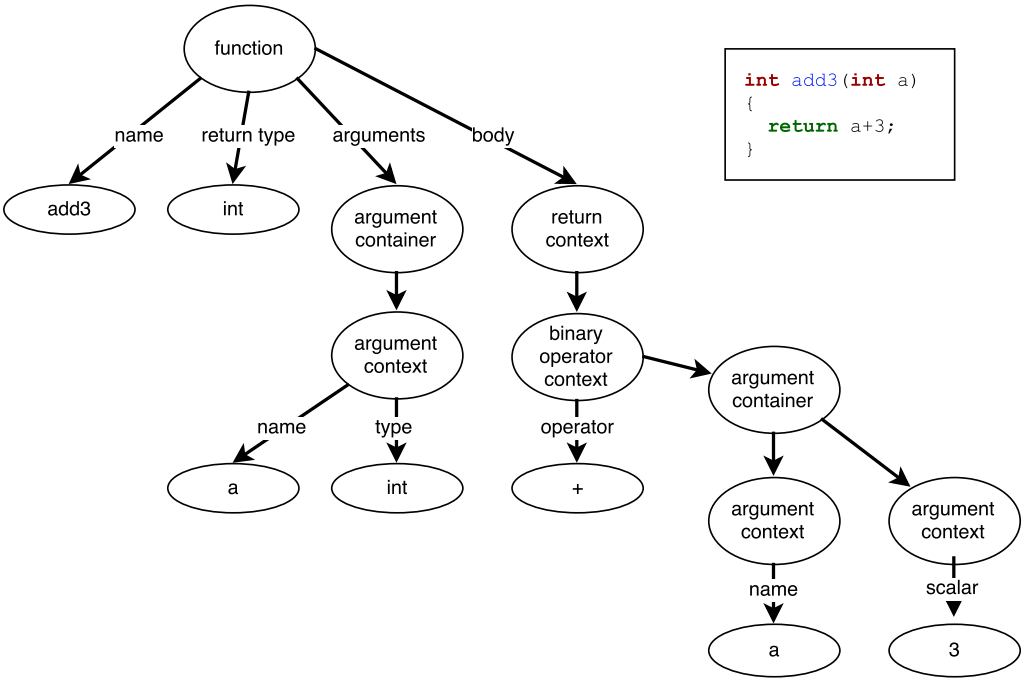
1.5.1 Abstraktní syntaktický strom

AST je uspořádaný zakořeněný strom (*ordered rooted tree*). Má tedy jeden význačný kořen (uzel reprezentující zdrojový soubor) a záleží na pořadí potomků: je rozdíl mezi potomky `[1,÷,2]` a `[2,÷,1]`.

Jde o hierarchickou reprezentaci zdrojového kódu.

Podoba AST typicky záleží na parseru, který ho generuje. Například gramatika pro jazyk PHP [32] generuje v AST uzel pro každý možný kontext. Jednoduchý příkaz `1+1` tak může být reprezentován jedním stromem s desítkami uzlů od `StatementContext`, přes `NotLeftRecursionExpression` a `ChainExpression` až po finální context operátoru. Takové AST je velmi nepřehledné, ale výhodné

pro automatické zpracování. Této podoby PHP AST využívá merge algoritmus z kapitoly 2.3.



Obrázek 1.3: Zdrojový kód a jeden z mnoha možných AST, který tento kód popisuje.

1.5.2 Operační transformace

Operační transformace (OT) jsou od základu odlišný způsob začleňování více změn, než tradiční řádkové algoritmy popsané výše. Jejich snahou je reprezentovat uživatelův záměr, nikoliv výsledný stav [33]. Systém například pracuje s informací, že odstavec byl přesunut mezi jiné dva odstavce [34]. Tradiční diff algoritmy znají pouze operace *smazat* a *vložit*.

Při začleňování změn se přehrají všechny změny seřazené podle kauzality.

Příklad: Při počátečním textu `ahoj světe`, uživatel A upravil první slovo na `dobrý den` (celkem 9 znaků místo původních 4). Druhý uživatel nezávisle na prvním upravil druhé slovo na `nebe` (změnil tedy znaky 6 až 10). Při začleňování změn se nejprve změny seřadí; zde vyberu změny od A jako první, ale protože na sobě tyto změny nezávisí, je to bez újmy na obecnosti. Změnu od uživatele B je nezbytné před aplikací upravit. Místo původního rozmezí

6-10 upravuji nyní rozmezí 11-15 v textu `dobrý den světe`. Po aplikace této upravené změny vznikne výsledný společný text `dobrý den nebe`. Pokud by byly změny uživatele B aplikovány bez transformace, vznikl by (špatně) text `dobrý nebe`, což rozhodně nebyl záměr uživatelů.

Problém konkurenčních změn (*Concurrency Control Problem*)

Některé kolize nelze vyřešit vůbec (například pokud jeden uživatel přidá do věty slovo a druhý uživatel větu nezávisle na první úpravě větu smaže, nebo oba upraví stejné slovo).

Původní článek z roku 1989 [35] uvažuje jeden sdílený systém, který uživatelé mají replikovaný na terminálech. Nabízí řešení jako je zamykání dat před jejich modifikací a transakce. Pro tuto práci jsou vhodnější další zmíněná řešení:

- Detekování závislosti (*dependency detection*), při čemž se kolize rozpoznají podle časového razítka a ručně se řeší uživatelem.
- Vracení změn (*reversible execution*), kde se při identifikaci změn mimo pořadí úpravy vrátí a přehrají v pořadí správném.

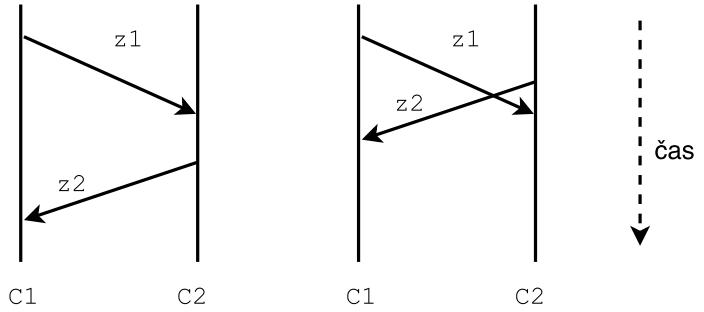
Původní návrh systému využívající OT [35] definoval OT-systém jako korektní, právě pokud splňuje tyto dvě vlastnosti:

- Zachování kauzality (*causality preservation*): pokud si dvě změny a, b předchází, tak u všech uživatelů se nejprve provede a a až pak b. Pro řazení se používá definice běžná v distribuovaných systémech zavedená Lamportem [36].
- Konvergence: po aplikaci všech změn u všech klientů je výsledek identický.

Sun [33] tento model dále rozšiřuje o zachování záměru (*intent-preservation*).

1.5.3 FastMatch

Záměrem tohoto algoritmu je rozšířit Myersův algoritmus 1.1.1 o podporu strukturovaných dat, které v teorii grafů definujeme jako uspořádaný strom (což je například AST). K tradičním operacím vložení (*insert*) a smazání (*delete*) přidává dvě nové operace: úprava (*update*) a přesun podstromu (*subtree move*).



Obrázek 1.4: Nepřekrývající se (vlevo) a překrývající se operace. Svislé čáry $C1$ a $C2$ reprezentují klienty/uživatele. Přeloženo z článku *Concurrency Control in Groupware Systems* [35].

Nepředpokládají se unikátní identifikátory uzlů napříč verzemi souborů: v přípravném kroku je potřeba mezi vstupními stromy uzly namapovat (*matching*). To vede na tzv. *Good Matching Problem*.

Pro diff algoritmus se dále řeší *Minimum Conforming Edit Script Problem*, kde – stejně jako u tradičních řádkových algoritmů – se snažíme minimalizovat velikost editačního skriptu, který jeden vstupní strom převede na druhý.

Kompletní algoritmus je v článku *Change Detection in Hierarchically Structured Information* [34, figure 8, 9].

Logika podle které lze uzly a, b namapovat je následovná:

function MATCH(a, b)
  if type $a \neq$ type b **then**
    return false
  end if
  if a, b jsou listy **then**
    return PODOBNOST(hodnota a, hodnota b) $\geq f$
  else
    $spolecne \leftarrow \{a \cap b \ : \text{MATCH}(a, b)\}$
    return $\frac{|spolecne|}{\max(|a|,|b|)} \geq t$
  end if
end function

Zápisem $|x|$ se rozumí počet potomků. Konstanty f a t lze volit libovolně.

1.6 Nejnovější diff/merge algoritmy

ChangeDistilling

Jde o první článek [37], ve kterém se přístup FastMatch aplikuje na zdrojový kód. Přínosy tohoto článku jsou mj. vytvoření testovací sady, téměř 45 % zrychlení oproti FastMatch a publikace programu pro porovnávání na úrovni uzlů AST.

Pro adaptaci na AST se odstraňuje předpoklad, že v pravém stromu je maximálně jeden list, který se může namapovat na odpovídající list v levém stromu [37, kapitola 2.2.2]. *ChangeDistilling* funguje správně pouze na AST LaTeX dokumentu, který má typicky hodně veliké listy a málo uzlů. Pro obecné AST není *ChangeDistilling* dobře použitelný.

3DM

3DM [38] je zkratkou pro *3-way merging, Differencing and Matching*. Jde o nejstarší mně známou práci, která se třícestnému merge věnuje. Algoritmy ChangeDistilling a GumTree (viz dále) popisují pouze diff a generování editačního skriptu.

Přestože práce primárně popisuje spojování XML dokumentů, lze většinu konceptů a algoritmů použít i na AST.

V *The Tree Matching Algorithm* [38, 6. kapitola] Lindholm představuje nový algoritmus pro mapování uzlů mezi podobnými stromy.

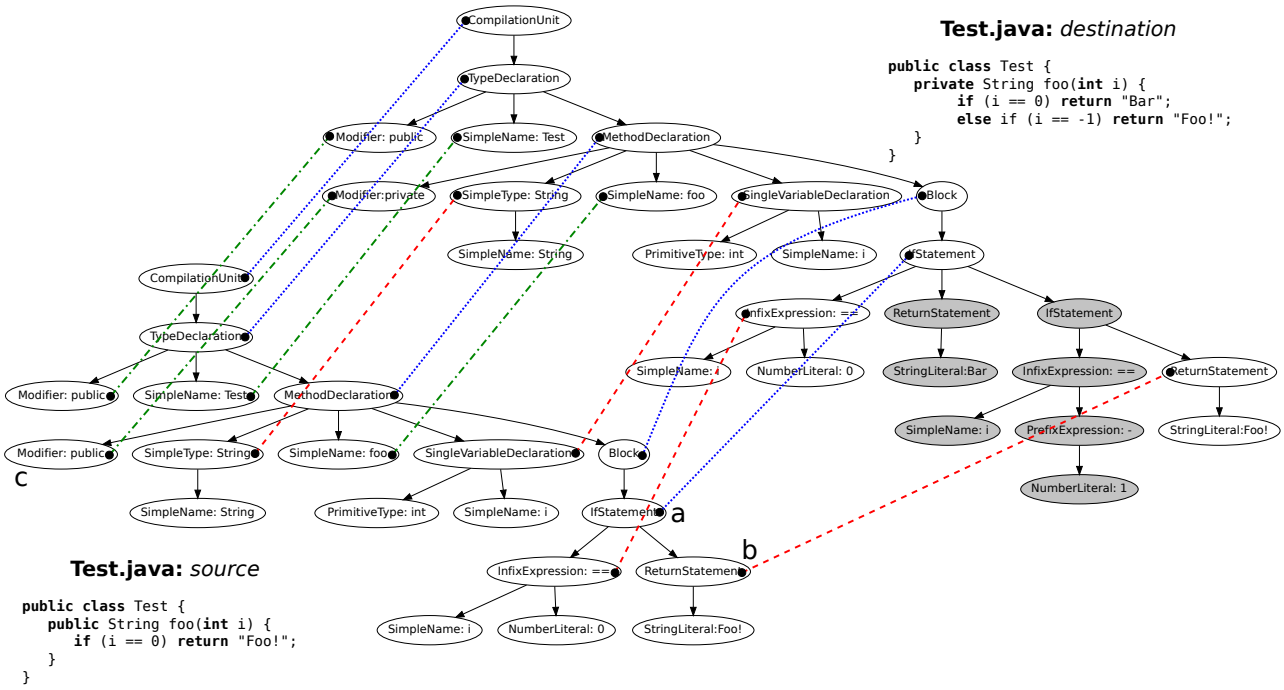
GumTree

Tento algoritmus dále rozšiřuje ChangeDistilling, popsaný v kapitole 1.6.
GumTree přidává podporu pro obecné AST (ne pouze AST LaTeX dokumentu).

Mapování se provádí dvoukrokově:

1. Top-down fáze: Od kořene se namapují isomorfní uzly (uzly se stejným typem, hodnotou a pouze isomorfními potomky). Mapování z toho kroku je tzv. *anchor mapping*.

2. Bottom-up fáze: Dva dosud nenamapované uzly se namapují (*container mapping*), pokud obsahují dostatečný počet namapovaných potomků. Pokud se v tomto kroku dva uzly namapují, provede se ještě doplňkový *recovery matching*, při kterém se nenamapovaní potomci zkusí namapovat agresivněji než v prvním top-down průchodu.



Obrázek 1.5: Ukázka mapování. V top-down fázi se namapují ekvivalentní uzly (znázorněné dlouhou přerušovanou červenou čarou). V bottom-up fázi se namapují podobné uzly (čerchovaná zelená čára) a provede se *recovery mapping* (tečkovaná modrá čára). Nenamapované uzly jsou šedé. Převzato z *Fine-grained and Accurate Source Code Differencing* [39].

Kapitola 2

Třísměrný merge zdrojových souborů přes AST

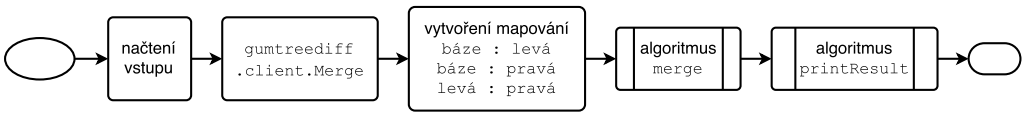
Tato kapitola popisuje logiku přijímající a řešící kolizní stav zdrojových souborů. Takový stav typicky vzniká při nezávislé souběžné editaci, kdy dva vývojáři vychází ze stejné báze, ale nesdílí navzájem své rozpracované úpravy.

Nejdřív popíšu algoritmus obecně a v následujících kapitolách rozeberu každou část samostatně. V první části se načítají tři vstupní soubory (báze, levá a pravá strana) a vytvoří se jejich AST. Algoritmem GumTree [39] se nalezne mapování uzlů mezi bází a levou, resp. pravou stranou. Novým algoritmem na základu 3dm [38] se tyto namapované stromy spojí do jednoho. Na závěr se ze vstupních souborů vyjmou ty části, které odpovídají spojenému stromu a vypíšou se jako výsledek.

Hlavním přínosem této práce jsou tyto algoritmy:

- na spojování abstraktních syntaktických stromů libovolného jazyka popsaného v kapitole 2.3,
- na transformaci AST zpět do kódu pomocí mapování na vstupní zdrojové kódy v kapitole 2.4.

K nalezení odpovídajících uzlů mezi jednou ze spojovaných větví a společnou větví se využívá algoritmu GumTree [39], detailně popsaného v sekci 1.6. Algoritmus 3dm, popsaný v 1.6, jsem v originální podobně nepoužil. Původní



Obrázek 2.1: Přehled kroků od vstupních souborů až po vypsání výsledku.

záměr spojit GumTree mapování a Lindholmův algoritmus z 3DM na merge byl zmařen tím, jak moc je v 3DM mapování a merge propojené. Definice *far move*, mapování složitější než 1:1, a další z GumTree bez zbytečně složitých úprav nelze získat. Proto jsem se 3DM pouze inspiroval, ale algoritmus navrhl nový, obecnější a tedy kompatibilní s GumTree.

Přestože cílem této práce je implementovat merge konkrétně pro jazyk PHP, jsou tyto algoritmy obecné a fungují pro AST každého jazyka. Liší se pouze výběr parseru, který je od ostatních částí striktně oddělen.

2.1 Načtení souborů a parsování do AST

Tento krok je triviální a záleží především na jazyku, jehož zdrojové kódy se načítají. Stačí implementovat pouze jeden XML parser a AST ostatních jazyků předpočítat samostatně a předávat serializované do XML.

Výsledkem tohoto kroku jsou tři AST. V dalším kroku se budou uzly těchto AST navzájem porovnávat a mapovat.

Časová komplexita záleží na konkrétním použitém parseru. Pro *Adaptive LL(*)* je $O(n^4)$ (kde n je počet AST uzlů), ale pro typické vstupy je parsování lineární [40].

2.2 Mapování uzlů mezi AST

Detailní fungování tohoto algoritmu je popsáno v kapitole 1.6. V integraci mého řešení nebyl s tímto GumTree modulem žádný problém.

Tento algoritmus běží nejlépe v čase s $O(n^3)$, kde n je počet AST uzlů [41].

2.3 Spojení tří mapovaných AST do jednoho

Následující algoritmus je hlavním přínosem této práce. Popisuje, jak spojit tři AST se vzájemně namapovanými uzly a detekovat kolize. Nejprve definuji pomocné funkce:

function FINDPARENT($left, right$)
  if $\exists$ any mapping m from $left$ to $base$ **then**
    return m
  else if $\exists$ any mapping m from $right$ to $base$ **then**
    return m
  else
    return empty node
  end if
end function

Tento pomocný algoritmus nalezne vhodný uzel z $base$. Pokud neexistuje, což se stane když je $left$ nebo $right$ nově vložený uzel, vrací prázdný uzel.

function FINDDELETED($base, side$)
  $c \leftarrow$ children of $side$
  for all $child \leftarrow$ children of $base$ **do**
    if $\nexists$ mapping of $child$ to c **then**
      yield $child$
    end if
  end for
end function

Vrací přímé potomky $base$, pro které neexistuje vhodné mapování na přímé potomky $side$. Jsou to uzly, které existovaly v $base$, ale ne v $side$, a ve výsledném merge by se neměly objevit.

function MOVEDNODES($base, side, deleted$)
  $base \leftarrow base \ /$ deleted
  $side \leftarrow side \ /$ deleted
  $side \leftarrow side \ / \{child \in side : \nexists$ mapping of $child$ to $base\}$
  $indexes \leftarrow$ find index pairs: index in $base$ and index in $side$
  return nodes which index decreased
end function

Nejprve se z $base$ odstraní smazané uzly. Z $side$ se odstraní smazané uzly a uzly nové (vkládané). Dále se pracuje pouze s uzly, které se vyskytují ve všech třech stromech (přestože funkce přijímá pouze $base$ a jednu $side$, proměnná $deleted$ obsahuje i uzly smazané v třetím stromu). Pro každý ze zbylých uzlů se najde původní a nová pozice. Pokud se pozice snížila, je uzel označen jako posunutý a vrácen. Příklad: pro uzly $base = \{a, b\}$ a $side = \{b, a\}$ se uzel a přesunul z pozice 1 na pozici 2 a tedy není označen jako přesunutý. Uzel b se z 2 přesunul na 1 a funkce ho tedy vrátí.

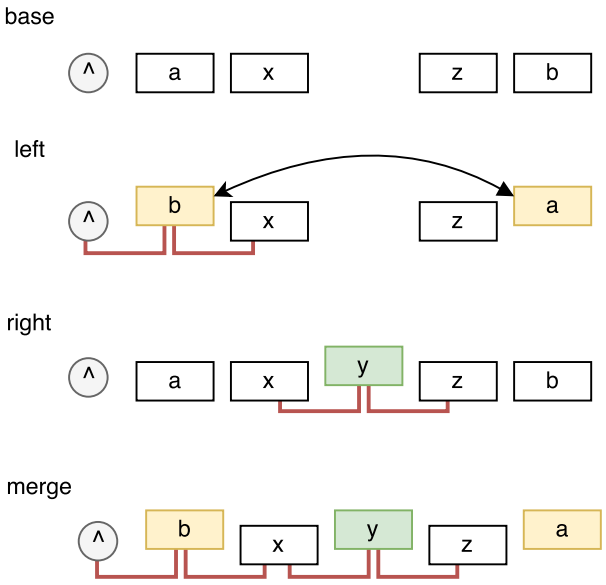
Označování přesunutých uzlů pouze v jednom směru umožňuje spojovat změny uvnitř přesunu bez detekování falešně pozitivní kolize. Například pro $base = \{a, x, z, b\}$, $left = \{b, x, z, a\}$, $right = \{a, x, Y, z, b\}$ se v části $left$ vyhodnotí jako přesunuté pouze $\{b\}$, a ne celá sekvence $\{a, x, z, a\}$ a lze bez kolize zapojit změny z $right$. Detailněji tento případ rozeberu níže u systému zamykání.

```
function makeMergeList(base, side, deleted)
    list ← {}
    insert start marker (empty node) to list
    for all child ← children of side do
        if ∄ mapping of child to base then      ▷ child is new (inserted) node
            mark top of list as right locked
            right lock child
        else if child ∈ movedNodes(base, side, deleted) then
            mark top of list as right locked
            right lock child
        end if
        insert child to list
    end for
    return list
end function
```

Funkce vrací efektivně stejný seznam jako je $side$, ale rozšířený o zámky. Aby se zachoval uživatelův záměr, při vkládání nového uzlu se zamknou uzly na obou stranách. Tato trojice je pak nerozdělitelná a pokud by ji změny z druhého stromu rozbily, dojde ke kolizi. Seskupování uzlů je vyřešeno pravostranným zámkem; levý zámek by fungoval identicky, ale místo označení počátku by bylo nutné pracovat s označením konce a implementace by

byla méně elegantní. Tato část algoritmu je podobná Lindholmovu [38], ale je zjednodušena (odstranění levostranného zámku, *hang-ons*) a logika pro zamykání je jiná.

function CREATENODE($base, left, right$)
  if type of $left$ = type of $right$ **then**
    $type \leftarrow$ common type of $left$ and $right$
    if content of $left$ = content of $right$ **then**
      $content \leftarrow$ content of $left$
    else if content of $base$ = content of $right$ **then**
      $content \leftarrow$ content of $left$
    else if content of $base$ = content of $left$ **then**
      $content \leftarrow$ content of $right$
    else
      conflict
    end if
  else if type of $base$ = type of $right$ **then**
    $type \leftarrow$ type of $left$
    $content \leftarrow$ content of $left$
  else if type of $base$ = type of $left$ **then**
    $type \leftarrow$ type of $right$
    $content \leftarrow$ content of $right$
  else
    conflict
  end if
  return node of type $type$ with content $content$
end function



Obrázek 2.2: Ukázka pravostranného zamykání uzlů. Znak $\wedge$ reprezentuje označení počátku (*start marker*). V $left$ ze prohodily uzly a a b, v $right$ byl vložen nový uzel y mezi x a z. Červeně jsou znázorněny semknuté uzly, které musí být u sebe a nelze je rozdělit bez ztráty záměru. Protože se zámky nepřekrývají, tyto dvě změny lze začlenit bez kolize.

Tento algoritmus vytváří nový uzel ze tří vstupních uzlů. Protože spojujeme na úrovni uzlů, jakákoliv neshoda na této úrovni vyústí v nevyřešitelný konflikt. Prakticky toto bude nastávat pouze u dlouhých řetězců (ukázka je v příloze B.2). Pro ty by bylo vhodné implementovat například samostatný merge na úrovni slov oddělených mezerou. Typem uzlu se rozumí *parser context*, tedy například `PhpBlock`, `AdditionContext` a podobně. Obsah (`content`) uzlu je například `ahoj` pro řetězec.

Na tomto místě by také bylo vhodné varovat před ztrátou změn. Pokud $right$ má stejný $type$ jako $base$ a má pouze nějaké změny v $content$, $left$ změnou $type$ tyto úpravy přepíše. Velmi striktní merge strategie by na tomto místě měla vyhodit kolizi.

```
function MERGE(base, left, right)
    deleted ← FINDDELETED(base, left) ∪ FINDDELETED(base, right)
    list_l ← MAKEMERGELIST(base, left, deleted)
    list_r ← MAKEMERGELIST(base, right, deleted)
    cur_l ← iterator on list_l
    cur_r ← iterator on list_r
    mergedTree ← CREATENODE(base, left, right)
    while cur_l is valid ∨ cur_r is valid do
        if ∃ mapping between left and right then      ▷ locking is irrelevant
            parent ← findParent(cur_l, cur_r)
            append MERGE(parent, cur_l, cur_r) to mergedTree
            advance cur_l
            advance cur_r
        else if previous cur_l is locked ∧ previous cur_r is locked then
            if ∄ mapping between left and right then
                conflict
            end if
            parent ← findParent(cur_l, cur_r)
            append MERGE(parent, cur_l, cur_r) to mergedTree
            advance cur_l
            advance cur_r
        else if previous left was locked then
            append left (including child nodes) to mergedTree
            advance cur_l
        else if previous right was locked then
            append right (including child nodes) to mergedTree
            advance cur_r
        else                          ▷ asserting left and right map to each other
            parent ← findParent(cur_l, cur_r)
            append MERGE(parent, cur_l, cur_r) to mergedTree
            advance cur_l
            advance cur_r
        end if
    end while
    return mergedTree
end function
```

Tato funkce je samotné jádro mého merge algoritmu. Nejprve se sestaví *merge list* pro obě strany. Poté se paralelně prochází potomci obou stran. Pokud mezi aktuálním levým a pravým potomkem existuje mapování, rekurzivně se

spočte jejich spojení, jinak se využije systému zámků. Jsou-li oba předchozí uzly zamknuté, současné uzly se musí shodovat, jinak nastává kolize (jedna ze semknutých ntic by se musela rozdělit). Pokud je zamknutý pouze jeden uzel, jedná se o přemístění nebo vložení, které není v druhé větvi. Poslední možnost nastává, pokud se uzly implicitně shodují, ale v okolí nedošlo k žádné změně, která by vytvořila zámky. Pokud by se uzly neshodovaly, jedna ze stran by byla označena jako odstranění a druhá jako vkládání a došlo by k vytvoření zámků.

Pokud je pouze jeden z kurzorů cur_l nebo cur_r je nevalidní, došlo k vkládání nového uzlu na konec. Všechny funkce s tím musí počítat a mít stejný efekt, jako kdyby místo nevalidního kurzoru dostaly prázdný strom.

Algoritmus `merge` má časovou komplexitu $O(n)$, kde n je počet uzlů v AST.

2.4 Transformace spojeného AST zpět do kódu

Převedení syntaktického stromu zpět do kódu lze dvěma způsoby.

První z nich je pomocí nějakých pravidel každý uzel transformovat do kódu. Tím se ztratí původní formátování a často i čitelnost. Například číslo původně zapsané jako `1e6` by se bez složitých pravidel vypsalo `100000.0`, protože PHP parser informace o původním formátu zahazuje. Podobně by se jinak vypsal string interpolation, místo `"ahoj $name"` by se vypsalo `"ahoj {$name}"`. Ve všech případech jde o validní a funkčně identický kód, ale jiný, než napsal uživatel. Výhodou tohoto přístupu je jednotné formátování všeho kódu. Nemyslím si ale, že toto je úkolem verzovacího systému.

Druhou možností je namapování výstupních AST uzlů na původní zdrojové kódy. Dílčí tokeny budou naformátované vždy tak, jak je napsal uživatel. U příkladu výše by ve výsledném souboru bylo opravdu `1e6`, pokud to tak uživatel zadal na vstupu. Menší problém tvoří bílé znaky, které jsou pro čitelnost kódu velmi důležité (díky odřádkování se typicky seskupují související příkazy a podobně). Parser typicky bílé znaky zahazuje a při spojování vstupních AST s nimi tedy nemůžeme pracovat.

Následující algoritmus mapuje AST na vstupní zdrojové kódy. Očekává AST, které má všechny tisknutelné uzly v listech.

```
function printResult(tree, src_l, src_r)
    if tree is from left side then
        src ← src_l
    else
        src ← src_r
    end if
    if tree is leaf then
        start ← end of token previous to tree
        end ← end of tree token
        gist ← substring(from start to end of tree content)
        print(gist)
    else
        for all child ← children of tree do
            printResult(child, src_l, src_r)
        end for
    end if
end function
```

Využíváme toho, že ANTLR parser vrací pro každý token mapování na pozice v zdrojovém souboru. Při merge si tuto informaci uložíme a bychom s ní zde mohli pracovat. U každého uzlu si také uchováme informaci, jestli byl vygenerován z levého nebo pravého stromu (a tedy z kterého zdrojového souboru). Nemůže být z obou zároveň, protože by došlo při merge ke kolizi.

Důležitý je zde trik, jak obejít chybějící whitespace v parseru. Místo vypsání uzlu od prvního namapovaného znaku začneme zdrojový kód vypisovat už od konce tokenu předešlého. Tím vypíšeme i bílé znaky, které parser zahodil.

Tato strategie ztrácí některé whitespace změny, ale výstupní formát je vždy stejný, jako v části vstupních souborů. Protože výsledný kód je vždy validní, tyto kolize nejsou detekovány a nevyhazují kolize.

Tento algoritmus má časovou komplexitu $O(n)$, kde n je počet uzlů v AST.

Kapitola 3

Implementace

3.1 Merge algoritmus

Vlastní merge algoritmus popsaný v kapitole 2 jsem implementoval jako nový modul v aplikaci GumTree. Převedení algoritmu do zdrojového kódu v jazyku Java se obešlo bez pozoruhodných komplikací. Funkční aplikace je na přiloženém médiu, online[†] a může být začleněna do oficiálního repozitáře GumTree.

Jako základ celého projektu jsem využil implementaci GumTree [42]. Je dostupná pod open-source licencí LGPL-3.0 [43]. Každá část projektu je implementována jako samostatný modul, což zvyšuje čitelnost, rozšiřitelnost a naopak snižuje nechtěnou propojenost (coupling) napříč nesouvisejícími třídami.

Falleri prozíravě připravil abstrakci `Client`, kterou prozatím implementovali pouze rozdílové příkazy `AnnotatedDiffClient`, `WebDiff` a další. Přidal jsem novou implementaci `MergeClient`. V té se ověřuje, že uživatel předal argumentem tři soubory a vzniká instance třídy `MergeMapping`, která mapuje uzly mezi bází a oběma stranami. Projekt GumTree implementuje tři mapování: gumtree (výchozí), change-distiller a xy matcher.

Původní implementace podporuje parsování řady jazyků včetně c, ruby a json. Obsahuje i parser pro PHP, ale pro dávno nepodporovanou verzi 5.2, jejíž vývoj byl oficiálně ukončen v lednu 2011. Navíc je v GumTree využita

[†] `https://github.com/Mikulas/gumtree/releases/`

knihovna pro generování parserů ANTLR ve staré verzi. Nové gramatiky jsou dostupné pouze pro ANTLR verze 4 [44]. Část úsilí jsem vložil do hledání vhodné gramatiky pro aktuální php 7.1 a našel jsem open source implementaci od Ivana Kochurkina [32]. Nepodporuje bohužel všechny varianty, co oficiální PHP parser; nefungují například typové nápovědy pro třídy v deklaraci funkce. Jediný vhodnější parser o kterém jsem se dozvěděl je použit v IDE od firmy JetBrains (jako jsou například IDEA a PhpStorm). Jejich implementace php parseru je ale proprietární a nelze ji pro tuto práci použít.

Po zvolení knihovny a naprogramování této práce zveřejnila firma Microsoft nový PHP parser [45]. Přestože zatím nepodporuje celou řadu syntaktických obratů, jde do budoucnosti o nejslibnější knihovnu. Jde o takzvaný *fuzzy parser*, který dokáže do jisté míry pracovat i s nekompletními a syntakticky nevalidními soubory. To by pro tuto práci jako rozšíření verzovacího nástroje bylo velmi vhodné.

Analogicky k existujícímu ANTLR modulu jsem připravil nový samostatný modul ANTLR4. Díky tomu lze nyní gumtree velmi snadno rozšířit podporu jakéhokoliv jazyka, pro který existuje ANTLR4 předpis. Tyto změny byly zapracovány do pull requestu a odeslány původnímu autorovi ke komentáři a začlenění[†].

Zpětně jsem bohužel zjistil, že přestože je tato gramatika udržovaná a aktualizovaná, podporuje pouze funkce do PHP 5.6. Jediné mě známé parsery, která jsou vždy aktuální, jsou tyto dvě: AST zabudované přímo v jádru PHP, které lze zpřístupnit pomocí rozšíření `php-ast` [46]. Druhý parser je napsaný přímo v PHP `php-parser` [47]. Oba projekty spravuje Nikita Popov, jeden z členů PHP internals, kteří se starají o vývoj jazyka. Lepší postup než rozšiřovat GumTree – jak se zpětně ukázalo – by bylo nechat parsování na PHP, přetransformovat tyto struktury do XML, a ty pomocí GumTree mapovat.

3.1.1 Ukázka výstupu

V tradičním řádkovém algoritmu by tyto změny kolidovaly a vyústily v kolizi. Implementovaný algoritmus pracující nad AST změny dokáže korektně začlenit.

[†] `https://github.com/GumTreeDiff/gumtree/pull/47`

```php
<?php
return 1+1;
```

(1) Společná báze.

```php
<?php
return '2' +1;
```

(2) První změna.

```php
<?php
return 1 + four();
```

(3) Druhá změna.

Kód 3.1: První a druhá změna lze spojit, protože každá ovlivňuje jiný nekolidující token.

```php
<?php
return '2' + four();
```

Kód 3.2: Výsledný kód ze spojených změn. Za povšimnutí stojí i správně spojené změny ve whitespace.

3.1.2 Problémy implementace

```php
<?php
function add($a, $b) {
  return $a + $b;
}
```

Kód 3.3: Společná báze.

```php
<?php
function add($alpha, $beta) {
  return $alpha + $beta;
}
```

```php
<?php
function add($a, $b) {
  return $a->add($b);
}
```

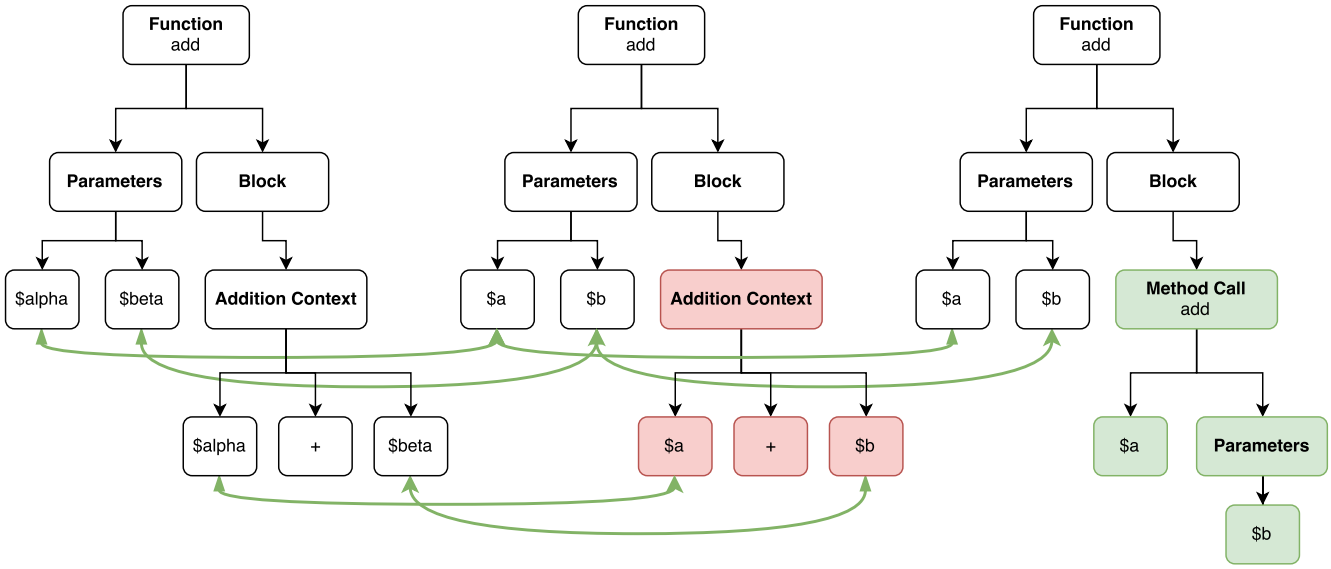
Kód 3.4: První a druhá změna, každá ovlivňující jiný nekolidující token.

Možná řešení tohoto problému jsou tři: upravit algoritmus mapování GumTree, upravit AST aby současný GumTree fungoval podle očekávání, a nebo zavést složitější systém zámků, který by nalezl související uzly (v ukázce kódu například `$a` v definici parametrů a ve volání funkce). Částečným řešením je i vyhazování kolize při detekování ztráty změn (zde ztráta změn v uzlu `Addition Context`), kterou jsem popisoval u algoritmu `createNode` v sekci 2.3.

Rozhodl jsem se upravit AST a uchovávat všechny kontextové uzly. Tím se tento problém vyřešil a merge algoritmus funguje správně. Příklad rozšířeného AST je v příloze C.

```php
<?php
function add($alpha, $beta) {
  return $a->add($b);
}
```

Kód 3.5: Výsledný nefunkční kód se špatným mapováním.



Obrázek 3.1: Zjednodušené AST z ukázky rozbitého mapování. Zelenou tlustou čarou jsou znázorněna mapování mezi listy. Uzly na vyšší úrovni jsou také namapované, ale pro přehlednost byly spoje vynechány. Zelené uzly v pravém stromu nejsou na nic namapované a budou se chovat jako nově vložené uzly. Protože červené uzly nemají mapování v pravém stromu, budou označeny jako smazané.

```php
<?php
function foo($alpha, $beta) {
  return $alpha->add( $beta);
}
```

Kód 3.6: Výsledný správný kód po rozšíření AST.

3.1.3 Výkon a porovnání s Myers

Následující tabulka obsahuje porovnání, jak dlouho trvá implementaci AST merge resp. Myers merge spojit zadané soubory. Baseline je test proti prázdným souborům a obsahuje například parsování vstupních argumentů, příprava pomocných struktur a především v případě Javy inicializace virtuálního stroje.

	baseline [s]	4K soubory [s]	50K soubory [s]
AST merge	1.009	1.264	11.122
Myers merge	0.052	0.054	0.068

Podle očekávání je AST merge aplikace v Javě řádově pomalejší, než roky optimalizovaná implementace Myersova řádkového porovnávání. Zajímavé na těchto výsledcích je především to, že i se stávající implementací lze AST merge běžně používat. Dále toto porovnání bere v potaz čistě procesorový čas; hlavní motivací pro AST merge bylo vyřešit více konfliktů a ušetřit tím uživateli čas strávený manuální úpravou, čehož bylo dosaženo.

Dílčí části algoritmu mají časovou komplexitu $O(n)$, pouze parser běží v patologických situacích v $O(n^4)$. Vysoký čas běhu aplikace je tedy pravděpodobně způsoben nedokonalou implementací, nikoliv špatným návrhem merge algoritmu.

3.2 Architektura webové aplikace a použitá infrastruktura

Webová aplikace, kterou má tato práce za cíl vytvořit, je z uživatelského hlediska jednoduchá a všechna těžká práce probíhá na pozadí, bez interakce s uživatelem.

Specifikace webové aplikace z pohledu uživatele:

- Na úvodní stránce je popis aplikace a přímo formulář pro nahrání zdrojových souborů.
- Výchozí formát je ‘diff3‘, lze přepnout tabem (stejná HTML stránka) na nahrávání tří samostatných souborů.
- Po odeslání formuláře se přímo zobrazuje stránka s výsledkem merge. Kromě samotných výsledků je zde možnost ohodnotit kvalitu výsledku.
- Všechny výsledky a jejich hodnocení se ukládají pro pozdější zpracování. Jejich seznam (historie řešení) je na samostatné stránce.

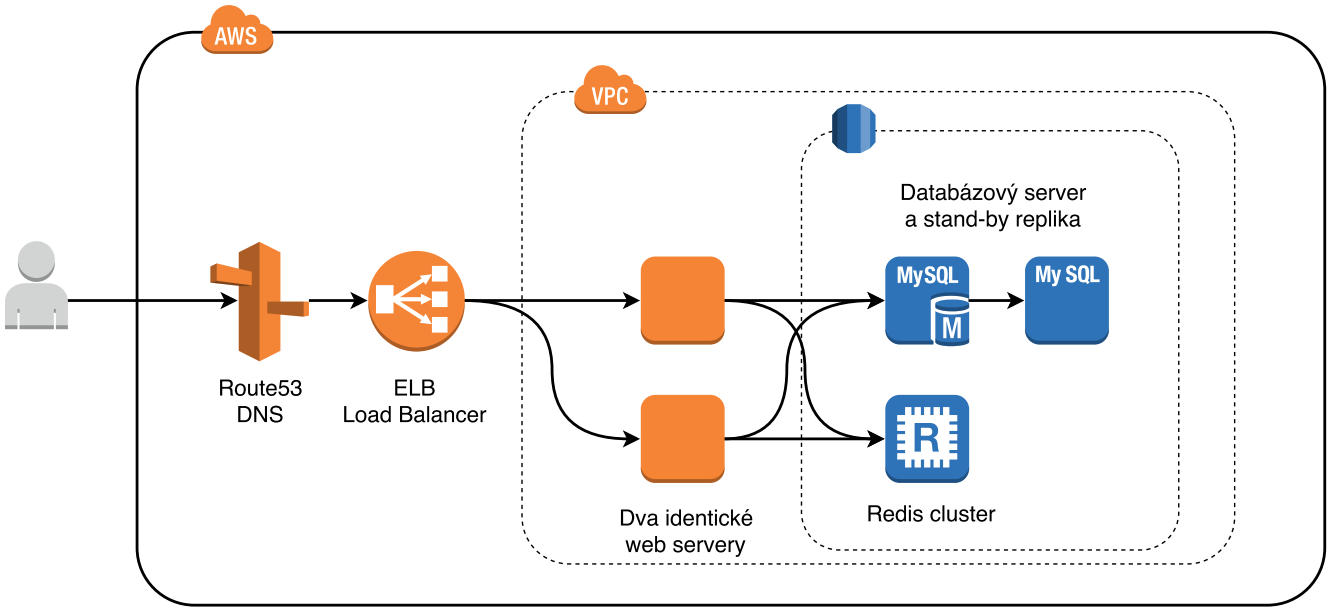
Pro implementaci webového rozhraní jsem zvolil jazyk `PHP 7.1` [48] a aplikační framework `Nette 2.4` [49]. Jako databázovou vrstvu jsem zvolil Amazon Aurora: proprietární databázový systém v cloudové infrastruktuře *Amazon Web Services* (AWS) kompatibilní s MySQL. Aplikace poběží ve virtuálním serveru taktéž v AWS.

Framework `Nette` jsem vybral, protože s ním mám největší praktické zkušenosti a oproti Symfony nabízí větší uživatelský komfort. Další webové PHP frameworky jako například Laravel jsem neuvažoval, protože nejsou dostatečně rozšířené, zdokumentované, nebo nepoužívají základní návrhové vzory jako je například *Dependency injection* a špatně se rozšiřují a testují.

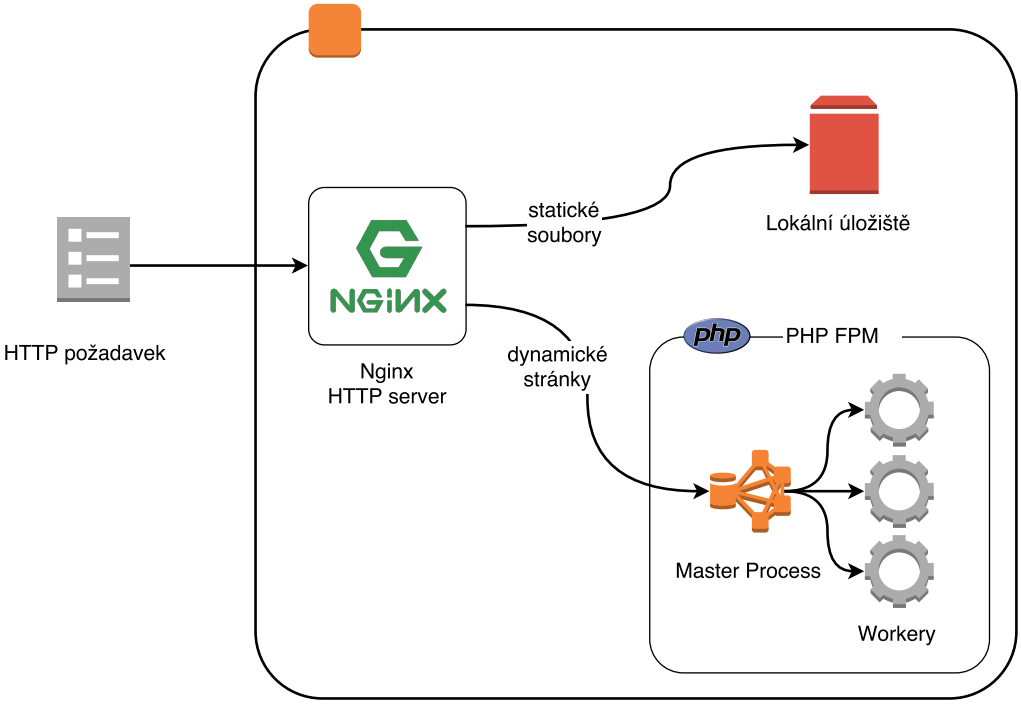
Tradiční web hosting nebyl pro tuto aplikaci použitelný, protože musíme spouštět Java aplikaci GumTree. Druhý extrém, vlastní dedikovaný server, byl rovněž zavržen, protože je zbytečně drahý. Cloudové řešení bylo vybráno jako dobrý kompromis kontroly nad systém a cizí správa a na rozdíl od jedné VPS (virtuální privátní server) lze dosáhnout vyšší dostupnosti (HA, *high availability*). Z cloudových řešení mám praktické zkušenosti pouze s AWS.

Použitá architektura (diagram 3.2) je typická pro aplikace vyžadující vysokou dostupnost, kde load balancer přeposílá rovnoměrně požadavky mezi více webových (aplikačních) serverů, pro které výpadek jednoho z nich neznamená nedostupnost. Podobně je redundantní i databáze, která se aktivně používá pouze jedna, ale souběžně běží replika, která je připravena kdykoliv v případě problému nahradit master. Oba aplikační servery i databázové repliky jsou ve fyzicky jiných místech (v terminologii AWS *availability zones*). Redis slouží jako sdílené úložiště pro sezení (*sessions*).

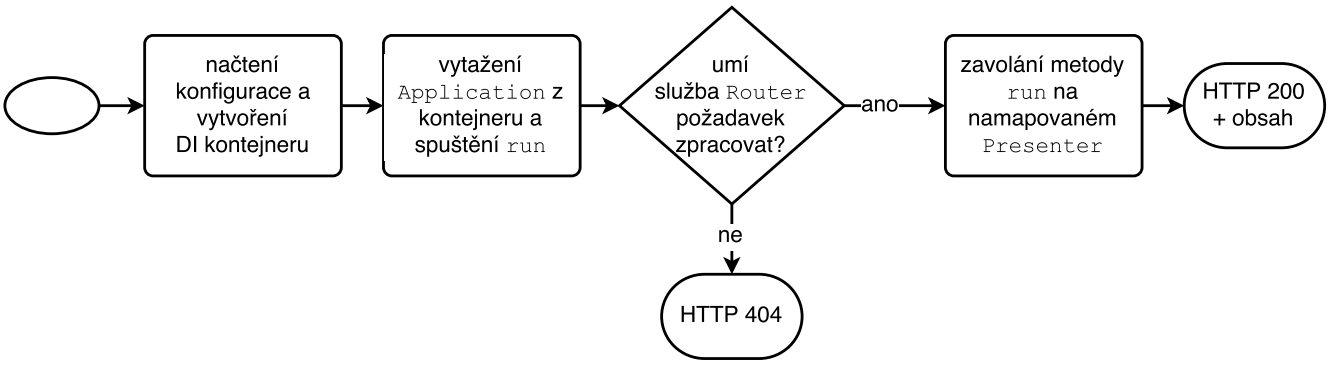
Strukturu webové aplikace a zpracování HTTP požadavku nad frameworkem Nette znázorňují diagramy 3.4 a 3.5, kde je také zdokumentováno propojení Java aplikace a uživatelského rozhraní v PHP.



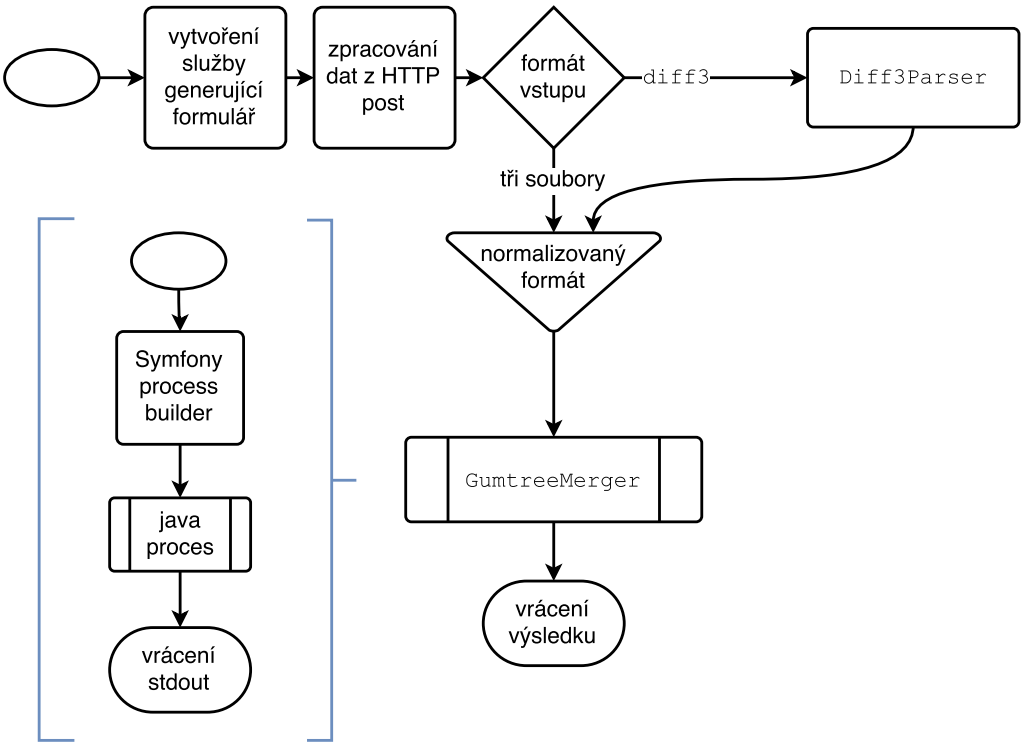
Obrázek 3.2: Vysokoúrovňový pohled na použitou infrastrukturu.



Obrázek 3.3: Zpracování HTTP požadavku na jednom aplikačním serveru. NGINX se také stará o zabezpečení pomocí TLS (probíhá zde tedy tzv. *TLS termination*), což na diagramu není znázorněno.



Obrázek 3.4: Architektura webu z pohledu vývojáře. Průchod požadavku Nette aplikací, stejný pro všechny aplikace. Byznys logika je implementovaná ve službách, které se volají z `Presenteru`.



Obrázek 3.5: Hlavní byznys logika aplikace zajišťující vyřešení kolize zdrojových souborů, které se načtou z formuláře. Na obrázku je rozkreslen i detail merge procesu v `GumtreeMerger` z pohledu webové aplikace, kde se volá spustitelný soubor v jazyku Java.

Uživatelské rozhraní

Rozhraní bylo bez předchozího navrhování a plánování přímo implementováno. Bylo dostatečně jednoduché a z předchozích zkušeností celkem jednoznačně dané. Hlavní use case - nahrání kolizního stavu k řešení v `diff3` – je přímo na homepage. Možnost nahrát samostatně každý soubor je vzdálena jedno kliknutí za záložkou. Podle statistik používání lze tyto dvě možnosti prohodit. Pokud budou obě varianty používány podobně často, bylo by vhodné taby zrušit a oba formuláře zobrazit na jedné stránce, například vedle sebe. Současná implementace ale myslí i na to, aby uživatel nebyl zahlcen vstupními poli a dokázal formulář snadno vyplnit.

Po odeslání formuláře se uživateli buď zobrazí chybová hláška (například pokud zadané soubory nejsou validní PHP), nebo je přesměrován na stránku s výsledkem. Přestože se zobrazují čtyři zdrojové kódy – báze, levá a pravá strana a výsledek – bylo cílem je uživateli prezentovat přehledně.

Uživatelské rozhraní – konkrétně stránka s výpisem výsledku – je v příloze A.

Publikování aplikace

Webová aplikace je veřejně dostupná online na internetu
a to na adrese `https://mergephp.com/`.

3.3 Git plugin

Pro praktické uplatnění v praxi bylo vytvořeno rozšíření pro verzovací systém Git. Po instalaci uživatel může automaticky provádět merge a pro vhodné soubory se transparentně zavolá GumTree merge z této práce.

Jedna z možností byla přidat vlastní *merge tool*, ale ten není dostatečně automatický. Uživatel ho musí ručně vyvolat příkazem `git mergetool` nad kolizním stavem. Další možností bylo vytvořit novou *merge strategy*, která se volá vždy, když se spojují dva commity. To by ale obnášelo duplikaci hodně logiky, kterou má Git zabudovanou.

Místo toho byl implementován vlastní *merge driver* [50]. Jde o skript, který Git zavolá, když narazí na dva kolizní soubory. Zdrojové kódy jsou na přiloženém médiu a také dostupné online[†]. Dokumentace k instalaci je kromě toho také v příloze D.

[†] `https://github.com/Mikulas/gumtree-merge-driver`

Závěr

Byly zdokumentovány diff a merge algoritmy v soudobých open-source verzovacích nástrojích. Z velké části musely být informace čerpány přímo ze zdrojových kódů. Dále byl představen nový směr, kterým se řešení této problematiky podle posledních článků ubírá, mapování stromových struktur. Byly představeny dvě konkrétní implementace tohoto přístupu: 3DM a *GumTree*.

Byl navržen nový algoritmus řešící třícestný merge s využitím existujících programů pro mapování uzlů mezi dvěma stromy. Tento algoritmus byl implementován v jazyku Java. Z porovnání generovaných řešení s verzovacím nástrojem Git vyšel tento algoritmus jako přesnější (generující méně kolizí).

Pro snadné praktické použití vznikl oproti zadání navíc Git plugin, který nový merge transparentně zapojuje.

Dále byla navrhnuta cloudová architektura s vysokou dostupností, webová aplikace a uživatelské rozhraní, a práce byla publikována na internetu na adrese `https://mergephp.com/`.

Práci lze dále rozšiřovat přidáním podpory pro zdrojové soubory jiných programovacích jazyků než je PHP. Implementace je naprogramována obecně a postup pro přidání dalších jazyků je zdokumentovaný. Zajímavá by rovněž byla analýza přístupu, kde se budou aplikovat rozdílná merge pravidla podle konkrétního AST kontextu (například jiná pravidla pro insert v kontextu třídy a pro parametry metody). Pro lepší integraci s verzovacím systémem by bylo vhodné analyzovat diff a vypisovat vysokoúrovňové změny (například „metoda `sum` byla přesunuta do třídy `Numeric`“).

Zdroje

1. LEES, Robert B; CHOMSKY, N. Syntactic Structures. *Language*. 1957, roč. 33, č. 3 Part 1, s. 375–408.
2. HUNT, J. W.; MCILROY, M. D. *An algorithm for differential file comparison. Computer Science*. 1975. Technická zpráva.
3. MASEK, William J.; PATERSON, Michael S. A faster algorithm computing string edit distances. *Journal of Computer and System Sciences*. 1980, roč. 20, č. 1, s. 18–31. ISSN 0022-0000. Dostupné z DOI: 10.1016/0022-0000(80)90002-1.
4. HIRSCHBERG, Daniel S. Algorithms for the Longest Common Subsequence Problem. *J. ACM*. 1977, roč. 24, č. 4, s. 664–675. ISSN 0004-5411. Dostupné z DOI: 10.1145/322033.322044.
5. MAIER, David. The Complexity of Some Problems on Subsequences and Supersequences. *J. ACM*. 1978, roč. 25, č. 2, s. 322–336. ISSN 0004-5411. Dostupné z DOI: 10.1145/322063.322075.
6. HSU, W.J.; DU, M.W. New algorithms for the LCS problem. *Journal of Computer and System Sciences*. 1984, roč. 29, č. 2, s. 133–152. ISSN 0022-0000. Dostupné z DOI: 10.1016/0022-0000(84)90025-4.
7. BERGROTH, L.; HAKONEN, H.; RAITA, T. A survey of longest common subsequence algorithms. In: *Proceedings Seventh International Symposium on String Processing and Information Retrieval. SPIRE 2000*. 2000, s. 39–48. Dostupné z DOI: 10.1109/SPIRE.2000.878178.

8. JONMORRISPOCOCK. *K-Candidate Diagram* [online]. 2013 [cit. 2017-02-18]. Dostupné z: https://en.wikipedia.org/wiki/File:K_Candidate_Diagram.png.

9. MYERS, Eugene W. An O(ND) difference algorithm and its variations. *Algorithmica*. 1986, roč. 1, č. 1, s. 251–266. ISSN 1432-0541. Dostupné z DOI: 10.1007/BF01840446.

10. MACKENZIE, David; EGGERT, Paul; STALLMAN, Richard. *Comparing and Merging Files: diff3 Merging* [online]. 2016 [cit. 2017-04-11]. Dostupné z: https://www.gnu.org/software/diffutils/manual/html_node/diff3-Merging.html.

11. MACKENZIE, David; EGGERT, Paul; STALLMAN, Richard. *Comparing and Merging Files: Overview* [online]. 2016 [cit. 2017-04-11]. Dostupné z: https://www.gnu.org/software/diffutils/manual/html_node/Overview.html#Overview.

12. KHANNA, Sanjeev; KUNAL, Keshav; PIERCE, Benjamin C. A Formal Investigation of Diff3. In: *Proceedings of the 27th International Conference on Foundations of Software Technology and Theoretical Computer Science*. New Delhi, India: Springer-Verlag, 2007, s. 485–496. FSTTCS'07. ISBN 3-540-77049-6, 978-3-540-77049-7. Dostupné také z: http://dl.acm.org/citation.cfm?id=1781794.1781836.

13. LIBENZI, Davide. *LibXDiff: File Differential Library* [online]. 2008 [cit. 2017-02-21]. Dostupné z: http://www.xmailserver.org/xdiff.html.

14. FRASER, Neil. *Google: Match, Diff and Patch - project homepage* [online]. 2009 [cit. 2017-02-21]. Dostupné z: https://code.google.com/p/google-diff-match-patch/.

15. FRASER, Neil; SLEMMER, Mike. *Google: Match, Diff and Patch - diff_match_patch.cpp* [online]. 2009 [cit. 2017-02-21]. Dostupné z: https://github.com/lqc/google-diff-match-patch/blob/master/cpp/diff_match_patch.cpp#L272-L282.

16. FRASER, Neil. *Google: Match, Diff and Patch - diff_match_patch.java* [online]. 2009 [cit. 2017-02-21]. Dostupné z: https://github.com/lqc/google-diff-match-patch/blob/master/java/name/fraser/neil/plaintext/diff_match_patch.java#L161-L171.

17. THE APACHE SOFTWARE FOUNDATION. *Subversion 1.5 Release Notes - Merge tracking* [online]. 2008 [cit. 2017-02-18]. Dostupné z: https://subversion.apache.org/docs/release-notes/1.5.html#merge-tracking.

18. FOAD, Julian; THE APACHE SOFTWARE FOUNDATION. *Subversion Wiki: Symmetric Merge Algorithm* [online]. 2012 [cit. 2017-02-18]. Dostupné z: https://wiki.apache.org/subversion/SymmetricMerge#Symmetric_Merge_Algorithm.

19. COLLINS-SUSSMAN, Ben; THE APACHE SOFTWARE FOUNDATION. *Subversion source code* [online]. 2005 [cit. 2017-02-18]. Dostupné z: https://github.com/apache/subversion/blob/9ee5ca717b5be3bcbcd2e133bb3a42a26b5f49a8/notes/propmerging_sanity.txt#L3-L4.

20. HAMANO, Junio C; TORVALDS Linus, et al. *Git - merge-strategies Documentation* [online]. 2014 [cit. 2017-02-18]. Dostupné z: https://git-scm.com/docs/merge-strategies.

21. TORVALDS, Linus. *Git source code - first diff implementation* [online]. 2014 [cit. 2017-02-18]. Dostupné z: https://github.com/git/git/blob/e83c5163316f89bfbde7d9ab23ca2e25604af290/show-diff.c#L39.

22. COHEN, Bram. *The diff problem has been solved* [online]. 2007 [cit. 2017-03-21]. Dostupné z: http://bramcohen.livejournal.com/37690.html.

23. COHEN, Bram. *Patience Diff Advantages* [online]. 2010 [cit. 2017-03-21]. Dostupné z: http://bramcohen.livejournal.com/73318.html.

24. SCHINDELIN, Johannes; HAMANO, Junio C. *git/git - Implement the patience diff algorithm* [online]. 2010 [cit. 2017-03-21]. Dostupné z: https://github.com/git/git/commit/92b7de93fb7801570ddc3195f03f30b9c201a3bd.

25. RAST, Thomas. *Git Mailing List - perf: compare diff algorithms* [online]. 2012 [cit. 2017-03-21]. Dostupné z: http://marc.info/?l=git&m=133103975225142&w=2.

26. PEARCE, Shawn O. *JGit project repository - HistogramDiff* [online]. 2010 [cit. 2017-03-21]. Dostupné z: https://github.com/eclipse/jgit/commit/b533a7293429258f34a6778a45a6c66dac55dc43.

27. CHUAN, Tay Ray; HAMANO, Junio C. *git/git - teach –histogram to diff* [online]. 2011 [cit. 2017-03-21]. Dostupné z: https://github.com/git/git/commit/8c912eea94a2138e8bc608f7c390eb0b313effb0.

28. HAGGERTY, Michael. *Infrastructure for testing the handling of diff sliders* [online]. 2016 [cit. 2017-03-21]. Dostupné z: https://github.com/mhagger/diff-slider-tools.

29. BUEHLMANN, Adrian. *Mercurial - FAQ: Technical Details* [online]. 2008 [cit. 2017-02-23]. Dostupné z: https://www.mercurial-scm.org/wiki/FAQ/TechnicalDetails.

30. HETTINGER, Raymond; VAN ROSSUM, Guido et al. *difflib — Helpers for computing deltas* [online]. 2006 [cit. 2017-02-23]. Dostupné z: https://docs.python.org/2/library/difflib.html.

31. HETTINGER, Raymond; VAN ROSSUM, Guido et al. *difflib — implementation commentary* [online]. 2006 [cit. 2017-02-23]. Dostupné z: https://hg.python.org/cpython/file/3.6/Lib/difflib.py#l113.

32. KOCHURKIN, Ivan. *PHP grammar* [online]. 2015 [cit. 2017-03-14]. Dostupné z: https://github.com/antlr/grammars-v4/tree/master/php.

33. SUN, Chengzheng; ELLIS, Clarence. Operational Transformation in Real-time Group Editors: Issues, Algorithms, and Achievements. In: *Proceedings of the 1998 ACM Conference on Computer Supported Cooperative Work*. Seattle, Washington, USA: ACM, 1998, s. 59–68. CSCW '98. ISBN 1-58113-009-0. Dostupné z DOI: 10.1145/289444.289469.

34. 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. ISSN 0163-5808. Dostupné z DOI: 10.1145/235968.233366.

35. ELLIS, C. A.; GIBBS, S. J. Concurrency Control in Groupware Systems. *SIGMOD Rec.* 1989, roč. 18, č. 2, s. 399–407. ISSN 0163-5808. Dostupné z DOI: 10.1145/66926.66963.

36. LAMPORT, Leslie. Time, Clocks, and the Ordering of Events in a Distributed System. *Commun. ACM*. 1978, roč. 21, č. 7, s. 558–565. ISSN 0001-0782. Dostupné z DOI: 10.1145/359545.359563.

37. 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. ISSN 0098-5589. Dostupné z DOI: 10.1109/TSE.2007.70731.

38. LINDHOLM, Tancred. A 3-way Merging Algorithm for Synchronizing Ordered Trees – the 3DM merging and differencing tool for XML. *Master's thesis, Helsinki University of Technology*. 2001. Dostupné také z: `http://cs.hut.fi/~ctl/3dm/thesis.pdf`.

39. FALLERI, Jean-Rémy; MORANDAT, Floréal; BLANC, Xavier; MARTINEZ, Matias; MONPERRUS, Martin. Fine-grained and Accurate Source Code Differencing. In: *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*. Vasteras, Sweden: ACM, 2014, s. 313–324. ASE '14. ISBN 978-1-4503-3013-8. Dostupné z DOI: `10.1145/2642937.2642982`.

40. PARR, Terence; HARWELL, Sam; FISHER, Kathleen. *Adaptive LL(*) Parsing: The Power of Dynamic Analysis*. 2014. Dostupné také z: `http://www.antlr.org/papers/allstar-techreport.pdf`. Technická zpráva.

41. PAWLIK, Mateusz; AUGSTEN, Nikolaus. RTED: A Robust Algorithm for the Tree Edit Distance. *Proc. VLDB Endow.* 2011, roč. 5, č. 4, s. 334–345. ISSN 2150-8097. Dostupné z DOI: `10.14778/2095686.2095692`.

42. FALLERI, Jean-Rémy; MORANDAT, Floréal. *gumtree - A neat code differencing tool* [online]. 2014 [cit. 2017-03-14]. Dostupné z: `https://github.com/GumTreeDiff/gumtree`.

43. FALLERI, Jean-Rémy; MORANDAT, Floréal. *gumtree - license file* [online]. 2014 [cit. 2017-03-14]. Dostupné z: `https://github.com/GumTreeDiff/gumtree/blob/develop/LICENSE`.

44. PARR, Terence; ANTLR. *ANTLR (ANother Tool for Language Recognition)* [online]. 2014 [cit. 2017-03-14]. Dostupné z: `http://www.antlr.org/`.

45. ITANI, Sara. *Tolerant PHP Parser* [online]. 2017 [cit. 2017-05-02]. Dostupné z: `https://github.com/Microsoft/tolerant-php-parser`.

46. POPOV, Nikita. *PHP AST - Extension exposing PHP 7 abstract syntax tree* [online]. 2014 [cit. 2017-03-16]. Dostupné z: `https://github.com/nikic/php-ast`.

47. POPOV, Nikita. *PHP-Parser - A PHP parser written in PHP* [online]. 2011 [cit. 2017-03-16]. Dostupné z: `https://github.com/nikic/PHP-Parser`.

48. THE PHP GROUP. *PHP: Hypertext Preprocessor* [online]. 2001-2017 [cit. 2016-12-08]. Dostupné z: `https://secure.php.net/`.

49. NETTE FOUNDATION. *Nette Framework: Rychlý a pohodlný vývoj webových aplikací v PHP* [online]. 2017 [cit. 2016-12-08]. Dostupné z: `https://nette.org/cs/`.

50. HAMANO, Junio C. *Git - gitattributes Documentation: Defining a custom merge driver* [online]. 2007 [cit. 2017-04-12]. Dostupné z: `https://git-scm.com/docs/gitattributes#_defining_a_custom_merge_driver`.

Dodatek A

Ukázky webové aplikace

Stránka s výsledkem uživatelova požadavku, včetně automaticky vygenerovaného řešení kolize (příloha je na následující straně).

Request #1

50 % Please vote on quality of this merge.

Merged file

download

```
<?php

class Foo
{
    function a()
    {
    }
}
```

Source A

```
<?php

class Foo {
  function a() {}
}
```

Source base

```
<?php

class Foo {

}
```

Source B

```
<?php

class Foo {
  function b() {}
}
```

Dodatek B

Porovnání kvality merge

(Ukázky začínají na následující straně.)

B.1 Refaktorování funkce

V tomto příkladu bylo v levé straně změněno jméno funkce a v pravé straně parametry a tělo. Algoritmus AST Merge tyto změny správně začlenil. Git detekoval změnu na stejném řádku a řešení vzdal.

```php
<?php
function add($a, $b) {
  return $a + $b;
}
```

Kód B.1: Báze, společný základ.

```php
<?php
function sum($a, $b) {
  return $a + $b;
}
```

(1) Levá změna oproti bázi.

```php
<?php
function add($numbers) {
  return array_reduce($numbers,
    function($carry, $item) {
      return $carry + $item;
  }, 0);
}
```

(2) Pravá změna oproti bázi.

```php
<?php
function sum($numbers) {
  return array_reduce($numbers,
    function($carry, $item) {
      return $carry + $item;
  }, 0);
}
```

(1) Řešení AST merge.

```
<?php
<<<<<<< a.php
function sum($a, $b) {
  return $a + $b;
||||||| base.php
function add($a, $b) {
  return $a + $b;
=======
function add($numbers) {
  return array_reduce($numbers,
    function($carry, $item) {
      return $carry + $item;
  }, 0);
>>>>>>> b.php
}
```

(2) Řešení Gitu.

B.2 Úprava víceřádkového textu

V obou změnách byl upraven jiný řádek ve dlouhém stringu. Git rozpoznal kontext obou změn, nedetekoval kolizi a změny správně začlenil. Algoritmus AST Merge celý string interpretoval jako jeden AST uzel a změny tedy odmítl jako kolizní.

```
<?php
echo "víceřádkový
text je v AST
pouze jeden uzel
a Git je výjimečně
s řádkovým přístupem
jemnější";
```

Kód B.4: Báze, společný základ.

```
<?php
echo "víceřádkový
TEXT je v AST
pouze jeden uzel
a Git je výjimečně
s řádkovým přístupem
jemnější";
```

(1) Levá změna oproti bázi.

```
<?php
echo "víceřádkový
text je v AST
pouze jeden uzel
a Git je výjimečně
s řádkovým PŘÍSTUPEM
jemnější";
```

(2) Pravá změna oproti bázi.

```
- kolize -
```

(1) Řešení AST merge.

```
<?php
echo "víceřádkový
TEXT je v AST
pouze jeden uzel
a Git je výjimečně
s řádkovým PŘÍSTUPEM
jemnější";
```

(2) Řešení Gitu.

B.3 Přidání nové metody

Oba algoritmy skončí kolizí. Git tuto situaci nedokáže vyřešit nikdy a nelze s tím nic dělat. AST Merge by mohl tuto situaci zpracovat, pokud by v kontextu třídy povolil přidání kolizní metody s jinou deklarací.

```php
<?php
class Foo {
}
```

Kód B.7: Báze, společný základ.

```php
<?php
class Foo {
  function __construct() {
    $this->x = 12;
  }
}
```

(1) Levá změna oproti bázi.

```php
<?php
class Foo {
  function cross(Foo $other) {
    return $foo * $other;
  }
}
```

(2) Pravá změna oproti bázi.

```
- kolize -
```

(1) Řešení AST merge.

```
<?php
class Foo {
<<<<<<< a.php
  function __construct() {
    $this->x = 12;
  }
||||||| base.php
=======
  function cross(Foo $other) {
    return $foo * $other;
  }
>>>>>>> b.php
}
```

(2) Řešení Gitu.

B.4 Drobné změny na jiném řádku (I)

Typická úprava v aplikaci. Funkční změna, v tomto případě normalizace vstupního parametru, a formální změna, typicky prováděná nad celým repozitářem. AST Merge správně změny začlení. Git nerozpozná kontext a změny s kolizí odmítne, přestože úpravy jsou vzájemně na jiném řádku.

```php
<?php
function greet($name) {
  echo "Ahoj $name" . "\n";
}
```

Kód B.10: Báze, společný základ.

```php
<?php
function greet($name) {
  $name = ucfirst($name);
  echo "Ahoj $name" . "\n";
}
```

(1) Levá změna oproti bázi.

```php
<?php
function greet($name) {
  echo "Ahoj $name" . PHP_EOL;
}
```

(2) Pravá změna oproti bázi.

```php
<?php
function greet($name) {
  $name = ucfirst($name);
  echo "Ahoj $name" . PHP_EOL;
}
```

(1) Řešení AST merge.

```
<?php
function greet($name) {
<<<<<<< a.php
  $name = ucfirst($name);
  echo "Ahoj $name" . "\n";
||||||| base.php
  echo "Ahoj $name" . "\n";
=======
  echo "Ahoj $name" . PHP_EOL;
>>>>>>> b.php
}
```

(2) Řešení Gitu.

B.5 Drobné změny na jiném řádku (II)

Podobná změna jako v předchozí ukázce. Na rozdíl od předchozího příkladu AST Merge vygeneruje špatný kód. Syntakticky je výsledek dobře, ale funkčně je od očekávaného výsledku odlišný: nesmazal `"\n"` a konstantu přidal do špatného kontextu.

```php
<?php
function greet($name) {
  echo "Ahoj $name\n";
}
```

Kód B.13: Báze, společný základ.

```php
<?php
function greet($name) {
  $fmtName = ucfirst($name);
  echo "Ahoj $fmtName\n";
}
```

(1) Levá změna oproti bázi.

```php
<?php
function greet($name) {
  echo "Ahoj $name" . PHP_EOL;
}
```

(2) Pravá změna oproti bázi.

```php
<?php
function greet($name) {
  $fmtName = ucfirst($name .
  ↪  PHP_EOL);
  echo "Ahoj $fmtName\n";
}
```

(1) Řešení AST merge.

```php
<?php
function greet($name) {
<<<<<<< ../samples/f/a.php
  $fmtName = ucfirst($name);
  echo "Ahoj $fmtName\n";
||||||| ../samples/f/b.php
  echo "Ahoj $name" . PHP_EOL;
=======
  echo "Ahoj $name\n";
>>>>>>> ../samples/f/base.php
}
```

(2) Řešení Gitu.

B.6 Smazání okolních řádků

Podobně jako v předchozím příkladu, Git nenajde kontext a změny nedokáže začlenit. AST Merge změny začlení správně.

```
<?php
class Foo {
        private $x;
        private $y;
}
```

Kód B.16: Báze, společný základ.

```
<?php
class Foo {
        private $x;
}
```

(1) Levá změna oproti bázi.

```
<?php
class Foo {
        private $y;
}
```

(2) Pravá změna oproti bázi.

```
<?php
class Foo {
}
```

(1) Řešení AST merge.

```
<?php
class Foo {
<<<<<<< a.php
        private $x;
||||||| base.php
        private $x;
        private $y;
=======
        private $y;
>>>>>>> b.php
}
```

(2) Řešení Gitu.

B.7 Přidání prvku do hash map

Ani jeden z algoritmů nedokáže tuto situaci vyřešit. AST Merge má potenciál tuto kolizi zvládnout automaticky, pokud bychom přidali za mapování nový krok, ve kterém by se uzly v hash map mapovaly podle klíče. Git s řádkovým přístupem tento případ nikdy nedokáže vyřešit.

```php
<?php
$services = [
  'a' => new Alpha(),
];
```

Kód B.19: Báze, společný základ.

```php
<?php
$services = [
  'alpha' => new Alpha(),
  'gamma' => new Gamma(),
];
```

(1) Levá změna oproti bázi.

```php
<?php
$services = [
  'alpha' => new Alpha(),
  'delta' => new Delta(),
];
```

(2) Pravá změna oproti bázi.

```
- kolize -
```

(1) Řešení AST merge.

```
<?php
$services = [
<<<<<<< a.php
  'alpha' => new Alpha(),
  'gamma' => new Gamma(),
||||||| base.php
  'a' => new Alpha(),
=======
  'alpha' => new Alpha(),
  'delta' => new Delta(),
>>>>>>> b.php
];
```

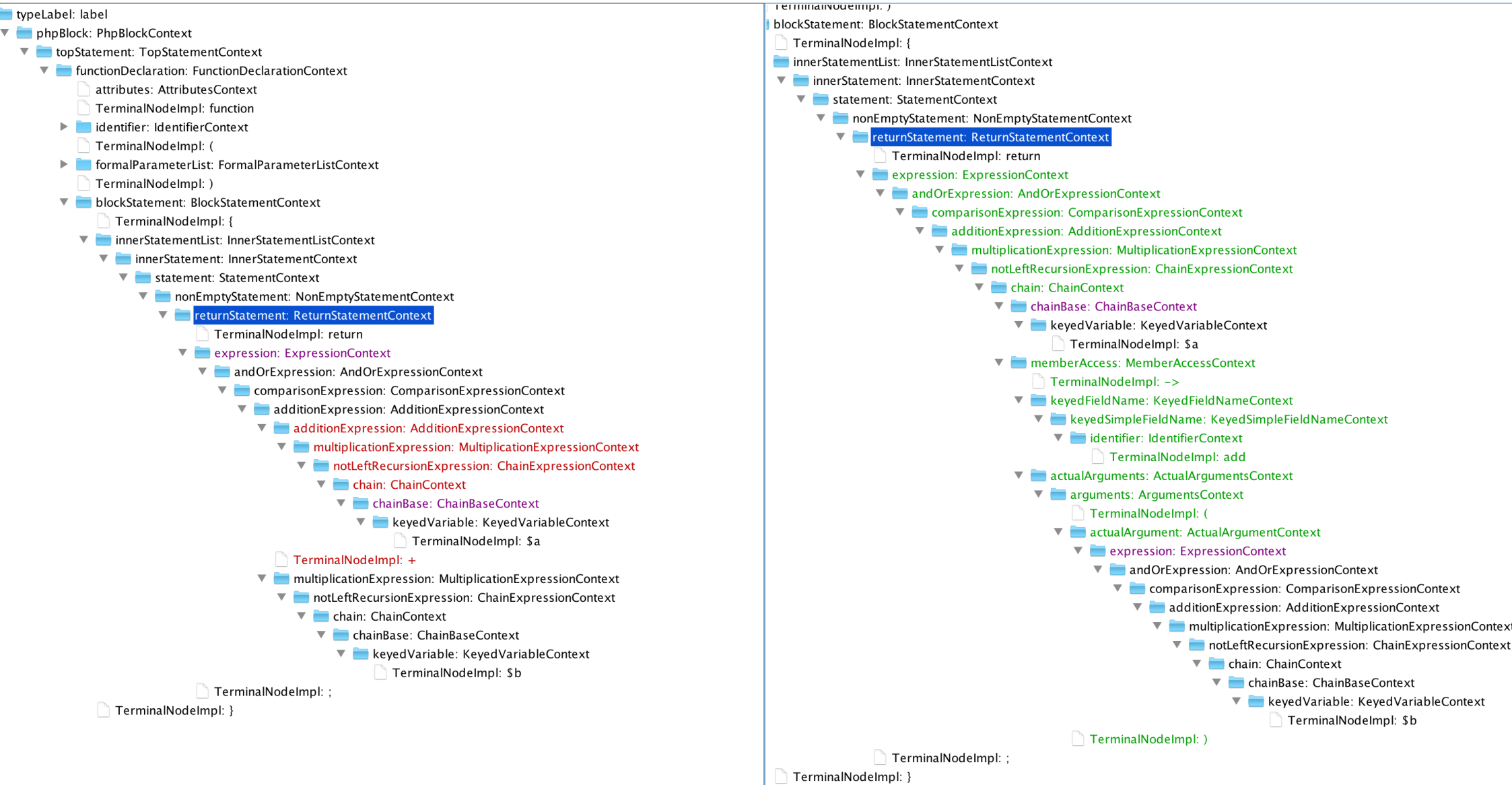
(2) Řešení Gitu.

Dodatek C

Ukázka komplexního AST s plným kontextem

Kompletní AST včetně všech kontextových uzlů. Jde o kód báze a levé strany z ukázky Drobné změny na jiném řádku (II).

(Příloha je na následující straně.)


typeLabel: label
phpBlock: PhpBlockContext
topStatement: TopStatementContext
functionDeclaration: FunctionDeclarationContext
attributes: AttributesContext
TerminalNodeImpl: function
identifier: IdentifierContext
TerminalNodeImpl: (
formalParameterList: FormalParameterListContext
TerminalNodeImpl:)
blockStatement: BlockStatementContext
TerminalNodeImpl: {
innerStatementList: InnerStatementListContext
innerStatement: InnerStatementContext
statement: StatementContext
nonEmptyStatement: NonEmptyStatementContext
returnStatement: ReturnStatementContext
TerminalNodeImpl: return
expression: ExpressionContext
andOrExpression: AndOrExpressionContext
comparisonExpression: ComparisonExpressionContext
additionExpression: AdditionExpressionContext
additionExpression: AdditionExpressionContext
multiplicationExpression: MultiplicationExpressionContext
notLeftRecursionExpression: ChainExpressionContext
chain: ChainContext
chainBase: ChainBaseContext
keyedVariable: KeyedVariableContext
TerminalNodeImpl: $a
TerminalNodeImpl: +
multiplicationExpression: MultiplicationExpressionContext
notLeftRecursionExpression: ChainExpressionContext
chain: ChainContext
chainBase: ChainBaseContext
keyedVariable: KeyedVariableContext
TerminalNodeImpl: $b
TerminalNodeImpl: ;
TerminalNodeImpl: }
TerminalNodeImpl:)
blockStatement: BlockStatementContext
TerminalNodeImpl: {
innerStatementList: InnerStatementListContext
innerStatement: InnerStatementContext
statement: StatementContext
nonEmptyStatement: NonEmptyStatementContext
returnStatement: ReturnStatementContext
TerminalNodeImpl: return
expression: ExpressionContext
andOrExpression: AndOrExpressionContext
comparisonExpression: ComparisonExpressionContext
additionExpression: AdditionExpressionContext
multiplicationExpression: MultiplicationExpressionContext
notLeftRecursionExpression: ChainExpressionContext
chain: ChainContext
chainBase: ChainBaseContext
keyedVariable: KeyedVariableContext
TerminalNodeImpl: $a
memberAccess: MemberAccessContext
TerminalNodeImpl: ->
keyedFieldName: KeyedFieldNameContext
keyedSimpleFieldName: KeyedSimpleFieldNameContext
identifier: IdentifierContext
TerminalNodeImpl: add
actualArguments: ActualArgumentsContext
arguments: ArgumentsContext
TerminalNodeImpl: (
actualArgument: ActualArgumentContext
expression: ExpressionContext
andOrExpression: AndOrExpressionContext
comparisonExpression: ComparisonExpressionContext
additionExpression: AdditionExpressionContext
multiplicationExpression: MultiplicationExpressionContext
notLeftRecursionExpression: ChainExpressionContext
chain: ChainContext
chainBase: ChainBaseContext
keyedVariable: KeyedVariableContext
TerminalNodeImpl: $b
TerminalNodeImpl:)
TerminalNodeImpl: ;
TerminalNodeImpl: }

Dodatek D

Dokumentace Git pluginu

Tato příloha popisuje postup instalace rozšíření pro Git, jehož implementace je popsána v kapitole 3.3. Odkazované zdrojové kódy jsou na přiloženém médiu a online [†].

Požadavky a závislosti

- `Java SE JRE`, alespoň ve verzi `1.7.0`,
- `bash` interpreter,
- `Git`, alespoň ve verzi `v1.5.2`.

[†] `https://github.com/Mikulas/gumtree-merge-driver`

Postup instalace

1. Získejte aplikaci GumTree buď z přiloženého média, nebo z posledního dostupného online vydání [†].
2. Archiv rozbalte na lokální disk a zapamatujte si cestu ke spustitelnému souboru `gumtree`.
3. Následují změny se aplikují na konkrétní Git repozitář, ve kterém chcete GumTree merge používat:
   a) Aktualizujte konfigurační soubor `.git/config` přidáním tohoto ústřižku. Vyplňte reálnou cestu k `gumtree`.

```
[merge "gumtree"]
name = GumTree merge driver
driver = .git/gumtree-driver %O %A %B "cesta-k-gumtree"
recursive = binary
```

   b) Zkopírujte samotný *merge driver* (soubor `.git|merge-driver`) do `.git/merge-driver` a nastavte ho jako spustitelný příkazem `chmod a+x .git/merge-driver`.
   c) Zapněte *merge driver* pro konkrétní přípony v souboru `.gitattributes` přidáním tohoto ústřižku:

```
*.php merge=gumtree
```

Použití

Toto rozšíření je naprosto transparentní a uživatel nemusí kromě počáteční instalace měnit své zaběhlé postupy. Pokud se při volání příkazu `git merge` detekuje kolize nad php soubory, git zavolá nastavený GumTree merge.

[†] `https://github.com/Mikulas/gumtree/releases/`

Zdrojový soubor

```
#!/usr/bin/env bash
set -euo pipefail
IFS=$'\n\t'

# https://git-scm.com/docs/gitattributes#_defining_a_custom_merge_driver

if [[ "$#" -ne 4 ]]; then
        echo "Error: Unexpected number of arguments."
        exit 1
fi

BASE="$1"
OURS="$2"
THEIRS="$3"
GUMTREE="$4"

# GumTree requires valid extension on merged files
cp "$BASE" "$BASE.php"
cp "$OURS" "$OURS.php"
cp "$THEIRS" "$THEIRS.php"

# Clean those temp files regardless of final merge status
function finish {
        rm -f "$BASE.php" "$OURS.php" "$THEIRS.php"
}
trap finish EXIT

# Attemp to merge and save result into "%A" as per Git documentation:
"$GUMTREE" merge "$BASE.php" "$OURS.php" "$THEIRS.php" > "$OURS"
```

Dodatek E

Seznam použitých zkratek

3dm	3-way merging, Differencing and Matching (1.6)
ANTLR	Another Tool For Language Recognition, generátor parserů
AST	Abstrakt syntax tree (1.5.1)
AWS	Amazon Web Services, poskytovatel cloudových služeb
búno	bez újmy na obecnosti
ČVUT	České vysoké učení technické
DMP	Diff-Match-Patch, knihovna od Google (1.2)
GNU	GNU's Not Unix!, operační systém a sbírka programů
HTTP	Hypertext Transfer Protocol, protokol nad kterým běží většina internetu
IDE	Integrated Development Environment, chytrý editor zdrojových kódů
LCS	Longest common subsequence (1.1)
LGPL	GNU Lesser General Public License, open source licence
NGINX	Webový server
NP	Nondeterministic Polynomial time, třída komplexnosti
NSP	Nejdelší společná podsekvence, překlad LCS (1.1)
OT	Operational transformation (1.5.2)
PDF	Portable Document Format, formát dokumentů
PHP	rekurzivní akronym: PHP: Hypertext Preprocessor, programovací jazyk
SQL	Structured Query Language, jazyk na dotazování relačních databází
SVN	Apache Subversion, verzovací nástroj
TLS	Transport Layer Security, skupina kryptografických protokolů
XML	eXtensible Markup Language, strukturovaný datový formát
YNLCS	Yes/No Longest common subsequence (1.1)

Dodatek F

Obsah přiloženého média

Identická data jako na přiloženém médiu jsou také dostupná z url `https://gitlab.fit.cvut.cz/ditemiku/bp/tree/medium/build`.

- `README.md` ..... stručný popis obsahu média
- `tex`
  - `desky` ..... zdrojová forma desek práce
  - `prace` ..... zdrojová forma práce ve formátu Markdown a XeLaTeX
- `git-plugin` ..... skripty a dokumentace rozšíření verzovacího systému Git
- `gumtree` ..... zdrojové soubory Java aplikace s AST Merge algoritmem
- `webova-aplikace` ..... zdrojové soubory webové PHP aplikace
- `BP_Dite_Mikulas_2017.pdf` ..... text práce ve formátu PDF
- `desky.pdf` ..... desky práce ve formátu PDF

Adresářová struktura F.1: Obsah přiloženého média