Katedra informatiky
Přírodovědecká fakulta
Univerzita Palackého v Olomouci

DIPLOMOVÁ PRÁCE

Podpůrné nástroje pro výuku pokročilého programování

2023

Bc. Jakub Večeřa

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

Studijní program: Aplikovaná informatika,
Specializace: Vývoj software

Bibliografické údaje

Autor:	Bc. Jakub Večeřa
Název práce:	Podpůrné nástroje pro výuku pokročilého programování
Typ práce:	diplomová práce
Pracoviště:	Katedra informatiky, Přírodovědecká fakulta, Univerzita Palackého v Olomouci
Rok obhajoby:	2023
Studijní program:	Aplikovaná informatika, Specializace: Vývoj software
Vedoucí práce:	Mgr. Petr Krajča, Ph.D.
Počet stran:	39
Přílohy:	elektronická data v úložišti katedry informatiky
Jazyk práce:	český

Bibliographic info

Author:	Bc. Jakub Večeřa
Title:	Tools for Advanced Programming Tutorials
Thesis type:	master thesis
Department:	Department of Computer Science, Faculty of Science, Palacký University Olomouc
Year of defense:	2023
Study program:	Applied Computer Science, Specialization: Software Development
Supervisor:	Mgr. Petr Krajča, Ph.D.
Page count:	39
Supplements:	electronic data in the storage of department of computer science
Thesis language:	Czech

Anotace

Cílem této diplomové práce bylo vytvořit sadu vzájemně propojených nástrojů, které umožňují demonstrovat tvorbu větších programů. Jinými slovy se jedná o programy, které se skládají z více souborů se zdrojovými kódy. Sada nástrojů plní funkci verzovacího systému, který podporuje inkrementálně vytvářet verze programu, vracet se k předchozím verzím, včetně možnosti jejich editace a propagace změn do dalších verzí, a zároveň umožňuje tvorbu podpůrných textů.

Synopsis

The aim of this master thesis was to create a set of tools working together to demonstrate a development of larger programs - in other words, programs consisting of multiple source code files. These tools function both as a version control system, which supports the creation of incremental versions, and as a system for producing supporting tutorial texts. The version control feature allows the user to switch to previous versions, edit them, and propagate these changes to subsequent versions.

Klíčová slova: verzovací systém; tvorba podpůrných textů; texty pro výuku pokročilého programování; propagace změn; Pandoc

Keywords: version control system; producing supporting texts; texts for advanced programming tutorials; propagation of changes; Pandoc

Chtěl bych poděkovat svému vedoucímu Mgr. Petru Krajčovi, Ph.D. za odborné vedení, za pomoc a rady při zpracování této práce.

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

1 Úvod 8

2 Verzovací systém 9
2.1 Typy verzovacích systému . . . 9
2.1.1 Lokální verzovací systémy . . . 9
2.1.2 Centralizované verzovací systémy . . . 9
2.1.3 Decentralizované verzovací systémy . . . 10

3 Implementace verzovacího systému 11
3.1 Operace pro manipulaci řádků souboru . . . 11
3.2 Implementace diff . . . 12
3.2.1 Popis algoritmu diff . . . 12
3.3 Implementace patch . . . 14
3.4 Reprezentace metadat verzovacího systému . . . 14
3.5 Propagace změn do dalších verzí . . . 17
3.5.1 Zohlednění editací předchozích verzí . . . 18
3.5.2 Inverze editací . . . 19

4 Implementace příkazů verzovacího systému 21
4.1 Příkaz init . . . 21
4.2 Příkaz add . . . 21
4.3 Příkaz commit . . . 23
4.3.1 Implementace inverzí . . . 23
4.3.2 Zpětné inverze . . . 23
4.4 Příkaz checkout . . . 24
4.5 Příkaz reset . . . 24

5 Příklady nástrojů pro tvorbu textů k programu 25
5.1 Markdown . . . 25
5.2 TEX . . . 25
5.3 Jupyter notebook . . . 26
5.4 GitBook . . . 26
5.5 Pandoc . . . 26

6 Implementace nástroje pro tvorbu textů k programům 27
6.1 Dynamické snippety a odkazy . . . 27
6.2 Příkaz publish . . . 27

7 Uživatelská příručka 28
7.1 Instalace nástroje . . . 28
7.2 Příkazy . . . 28
7.2.1 Příkaz init . . . 29
7.2.2 Příkaz add . . . 29
7.2.3 Příkaz commit . . . 29

7.2.4 Příkaz checkout . . . 30
7.2.5 Příkaz version . . . 30
7.2.6 Příkaz diff . . . 30
7.2.7 Příkaz publish . . . 31
7.3 Případ užití . . . 33

8 Budoucí rozvoj nástrojů **34**
8.1 Implementace příkazu pro přesun souborů . . . 34
8.2 Podpora dalších formátů . . . 34
8.3 Přepisování hodnot snippetu a určení verze . . . 34
8.4 Podpora více uživatelů . . . 34
8.5 Další možnosti . . . 35

Závěr **36**

Conclusions **37**

A Obsah elektronických dat **38**

Literatura **39**

Seznam tabulek

1 Seznam příkazů a jejich argumenty 32

Seznam zdrojových kódů

1 Pseudokód algoritmu diff 13
2 Zdrojový kód pro patch 14
3 Datový objekt pro metadata a verzi 16
4 Algoritmus procesu sestavování souboru 19
5 Zdrojový kód modulu init. 22
6 Pseudokód algoritmu pro zjištění verzí, které budou invertovány, nebo u kterých bude případně provedena zpětná inverze. 24

1 Úvod

Motivací vývoje tohoto nástroje bylo umožnění provádět změny v předchozích verzích s automatickou propagací těchto změn do verzí následujících. Tato potřeba vychází zejména z komplexního vývojového procesu, kde je často nezbytné doplnit komentáře nebo upravit existující kód v minulých verzích.

Mějme na mysli vzorový scénář, jako je komplexní tutoriál s mnoha soubory, který postupně přidává ukázky dalších funkcí. Ve většině případů je úprava jednoho snippetu kódu spojena s nutností aktualizovat a synchronizovat všechny související snippety v následujících verzích. Tento proces je časově náročný a náchylný k chybám.

Naše řešení představuje nástroj, jak se s tímto problémem vypořádat. Nabízí možnost automatické propagace změn, což vede ke snížení chyb a časových nároků na správu verzí. Tato vlastnost je obzvláště užitečná v případě dokumentací nebo podpůrných textů, kde je nutnost ručního přepisu snippetů výrazně snížena.

Výhodou je, že každá verze programu má k dispozici podpůrný text, který je přímo relevantní pro danou verzi. Tím se eliminuje potřeba prohledávat historii změn a manuálně upravovat dokumentaci.

Hlavním cílem nástrojů, které během této práce vznikly, je vytvořit prostředí pro uživatele tvořící podpůrné texty k programům. Teoretická část této práce slouží k popsání některých částí jejich implementace a zároveň také demonstruje výsledek využití nástrojů. Právě pomocí nástrojů je také vygenerována.

Pro práci s verzovacím souborem byla určena minimální sada operací potřebných k implementaci, aby byl nástroj použitelný. Byly to operace pro inicializování, změnu verze a uložení změny souboru. Implementaci operací je věnována sekce 4 a příklady užití jsou popsány v sekci 7.

Verzování i tvorba podpůrných programů jsou zakomponovány do jednoho nástroje, v textu práce jsou ale rozděleny do kapitol zvlášť, jelikož se z principu jedná o nástroje dva.

2 Verzovací systém

Verzovací systém je systém, který zaznamenává změny souborů v průběhu času tak, aby bylo možné libovolnou verzi znovu vyvolat [2]. Klíčovou součástí takového systému je repozitář reprezentující datovou strukturu, která schraňuje metadata pro množinu souborů nebo adresářovou strukturu.[3]

Jedním z možných způsobů verzování souborů je před každou změnou udělat kopii daného souboru a upravovat pouze tuto kopii. Při opakování stejného úkonu dochází k tomu, že máme v adresáři mnoho souborů, které v ideálním případě ve svém názvu nesou informaci rozpoznávající danou verzi. Tento postup lze aplikovat i na celé adresáře. Mnohdy ale dochází k tomu, že se uživatel ztratí. Dále je tento systém velice náchylný k jakýmkoliv chybám při mylném přesunu nebo přepisu souborů.

2.1 Typy verzovacích systému

2.1.1 Lokální verzovací systémy

K řešení tohoto problému programátoři dávno vyvinuli lokální verzovací systémy, které měly jednoduchou databázi pro kontrolu nad změnami souborů [2].

Jako zástupce je možné uvést systémy SCCS (Source Code Control System) a RCS (Revision Control System). Zajímavou vlastností obou systémů je způsob, jakým zaznamenávají změny, a to pomocí rozdílů (delty) mezi jednotlivými verzemi.

Pro úsporu místa RCS ukládá delty namísto celé revize. Jednotkou změny je řádek, tj. pokud je změněn jediný znak, RCS považuje celý řádek za změněný. Tento přístup byl zvolen, protože bylo možné použít implementaci diff [1] z UNIX, který vypočítával delty na základě řádků. [4]

2.1.2 Centralizované verzovací systémy

Poměrně zásadním nedostatkem lokálního verzovacího systému je přístup pouze na jednom zařízení. Aby bylo možné spolupráce více uživatelů, musí existovat způsob, jak soubory jednotlivým uživatelům zpřístupnit. Jedním řešením jsou centralizované verzovací systémy, u kterých poskytnutí souborů zajišťuje centralizovaný server. Příklady systémů tohoto typu jsou CVS (Concurrent Versions System) a Subversion.

Tohle řešení má oproti lokálním mnoho výhod. Jako například to, že každý do jisté míry ví, co ostatní dělají. Administrátoři mají vysokou kontrolu nad tím, co každý může dělat, a je mnohem snadnější řídit centralizované systémy, než řešit problematiku lokálních databází na každém zařízení.[2].

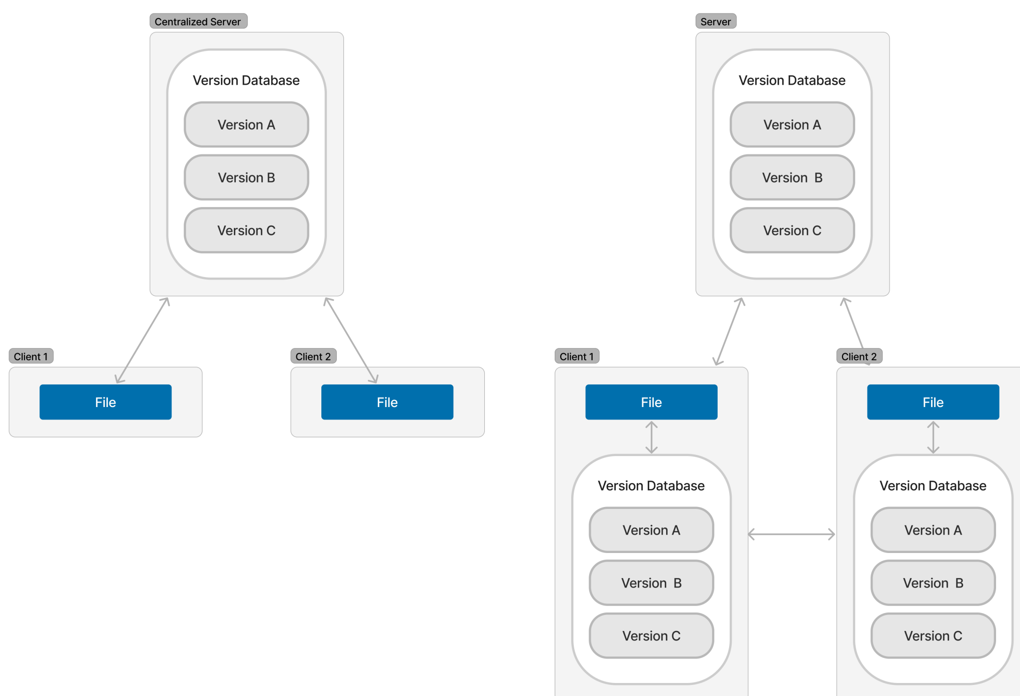
Kritickými problémy centralizovaného systému jsou případy, kdy u centralizovaného serveru dojde k výpadku. Po celou dobu výpadku to všem uživatelům

[1] *Diff* je nástroj pro porovnání dat, který vypočítává a zobrazuje rozdíly mezi obsahy souborů.[1]

spolupráci znemožní. V horších případech může také dojít k napadení systému a eventuální ztrátě všech dat.

2.1.3 Decentralizované verzovací systémy

Druhým typem řešení, jak soubory uživatelům zpřístupnit, je použití decentralizovaných systémů. Namísto čtení jen poslední verze si uživatelé na svoje zařízení propojí celý obsah verzovacího systému. Obrázek 1 porovnává charakteristiky centralizovaného a decentralizovaného systému. Git, který je jedním zástupcem tohoto typu, má například příkaz *clone*, u kterého dochází k naklonování repozitáře do nově specifikované složky. Ten také automaticky zajistí provázanost se zdrojem nazvaným git-remote. Lokálně je pak možné si těchto zdrojů provázat více. Při selhání git-remote nedochází k ztrátě dat. V případě, že výměna dat probíhá přes internet, je také možné pracovat bez připojení a až po obnově spojení lze remote o změnách informovat. Jako další zástupce bych také zmínil verzovací systémy Darcs a Pijul. Darcs představuje zajímavou teorii o aktualizacích (z anglického *patch*), která je více rozvedena v literatuře [6]. Pijul pak na tuto teorii navazuje.



Obrázek 1: Porovnání centralizovaného (vlevo) a decentralizovaného (vpravo) systému. Inspirací pro tento diagram byla literatura [2]

3 Implementace verzovacího systému

Pro implementaci verzovacího systému byl zvolen jazyk Python a to i přes jeho potenciální nevýhody týkající se výkonu. Bylo nutné využít prototypování pro zjištění, jakým způsobem s nástrojem pracovat, a jazyk Python byl ideální pro tento úkol.

Pro Python je napsáno mnoho přehledně zdokumentovaných knihoven, které usnadnily práci se soubory, interakci uživatele s nástrojem pomocí příkazového řádku a také proces serializace a deserializace[2] souborů ve formátu JSON[3].

Kód je v jazyce Python organizován do modulů. V případě tohoto nástroje náleží každému souboru jeho modul a je kladen důraz na přehlednost jejich organizace. V textu této práce je často odkazováno na seznam (anglicky list), který je jedním ze základních datových typů jazyka. Pro zajištění určitého standardu byl kód psán za pomocí nástroje *Pylint*[4] a nástroje pro formátování *Black*.

Nástroj pro verzovací systém nevyužívá žádné knihovny třetích stran. Jedním z největších ulehčení mu však nabízí modul *argparse*. Modul *argparse* může být využit pro parsování argumentů příkazového řádku a je jedním z největších modulů standardní knihovny jazyka Python.[8]

3.1 Operace pro manipulaci řádků souboru

První úkol spočíval ve stanovení vhodné reprezentace pro provádění změn v souboru. Snahou bylo nad změnami přemýšlet co nejvíce atomicky, a proto vznikly dvě operace. Jedna z operací je operace přidání, označená jako *add*. Druhá operace je operace odebrání, označená jako *remove*. Díky jejich atomickému charakteru lze vytvářet posloupnosti těchto operací pomocí seznamů, což reprezentuje změny souboru.

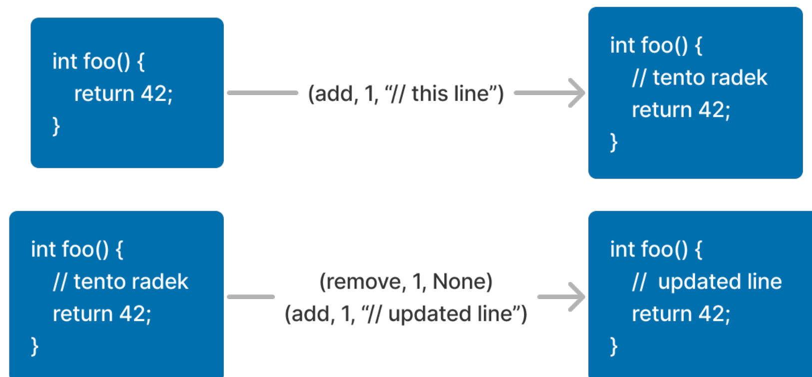
Tyto operace jsou reprezentovány jako trojice: název operace, číslo řádku a data. V případě operace *remove* jsou v současné implementaci nástroje data nahrazeny hodnotou *None*, která nevyjadřuje žádnou hodnotu.

Soubory jsou také reprezentovány jako seznamy. Při čtení souboru je každý jeho řádek uložen do paměti jako položka seznamu. Operace *add* a *remove* manipulují s prvky seznamu, jenž jsou číslovány od nuly, a zrcadlí tak úpravy řádků. Pro ilustraci aplikace těchto operací na soubor lze odkázat na diagram 2.

[2] Pojem *serializace* se vztahuje na proces ukládání interních dat programu do výstupního souboru, kdežto *deserializace* označuje opačný proces.

[3] JSON je akronymem pro JavaScript Object Notation a je jedním z formátů pro reprezentaci strukturovaných dat.

[4] Pylint je statický analyzátor pro Python, který zároveň kontroluje běžné konvence pro psaní kódu.



Obrázek 2: Aplikování operací na soubor

3.2 Implementace diff

Následující část se zabývá implementací algoritmu *diff*, jehož úkolem je vracet změny mezi řádky ve formě seznamu atomických operací *add* a *remove*. Nebylo možné využít existující algoritmy poskytované standardním nástrojem *diff*, protože z vlastností operací pro manipulaci řádků souboru je kladen větší důraz na jejich jednotlivé pořadí. Kvůli charakteristikám seznamů a povaze operací *add* a *remove* musí naše implementace algoritmu diff zohlednit posunutí řádků.

Výsledkem porovnání dvou souborů je seznam operací ve formě trojic představených v předchozí sekci. Při postupné aplikaci operací na první soubor je výsledkem druhý soubor.

3.2.1 Popis algoritmu diff

Algoritmus diff, jak je prezentován v pseudokódu 1, je popisován následujícími kroky. Pro lepší ilustraci jsou jednotlivé položky seznamů v tomto popisu označovány jako *řádky*.

1. Algoritmus postupně porovnává jednotlivé řádky dvou vstupních seznamů *First* a *Second* a postupně do výstupního seznamu *Changes* zapisuje jejich úpravy.

2. V případě, kdy se řádky rovnají, se postupuje na další.

3. Pokud se řádky liší, dojde k vyhledání prvního výskytu aktuálního řádku ze seznamu *First* v seznamu *Second*.

4. Když takový řádek v *Second* neexistuje, je na konec seznamu *Changes* zapsán záznam o odebrání tohoto řádku. Ze seznamu *First* je tento řádek odebrán (posunutím indexu i na další řádek).

5. Jestliže takový řádek v seznamu *Second* existuje, algoritmus pro každý řádek v seznamu *Second*, od aktuálního řádku až po nalezený řádek, doplňuje do seznamu *Changes* záznam o přidání řádku. Jakmile je nalezený řádek dosažen, jsou tyto řádky z druhého seznamu odebrány (posunutím indexu j na index nalezeného řádku výskytu).

6. Pokud jsme prošli celý seznam *First* a v seznamu *Second* stále zůstávají nějaké řádky, jsou do seznamu *Changes* doplněny záznamy o přidání těchto řádků.

7. Naopak, pokud jsme prošli celý seznam *Second* a v seznamu *First* stále zůstávají nějaké řádky, algoritmus do seznamu *Changes* přidá záznamy o odebrání těchto řádků.

```
1  Diff(seznam First, seznam Second)
2     Changes <- prázdný seznam
3     i, j, numAdded, numRemoved <- 0, 0, 0, 0
4
5     while i < length(First)
6        if j >= length(Second)
7           Pro každý zbývající prvek v seznamu First
8              Changes.Append(("remove", j, None))
9        if First[i] = Second[j]
10          i <- i + 1, j <- j + 1
11          continue
12
13       foundIndex <- první výskyt řádku First[i] v Second od indexu j
14
15       if foundIndex = None (Pokud se nevyskytuje)
16          Changes.Append(("remove", i - numRemoved + numAdded, None))
17          i <- i + 1
18          numRemoved <- numRemoved + 1
19
20       else (Pokud se vyskytuje)
21          localNumAdded <- 0
22          for k in range(j, foundIndex):
23             Changes.Append(("add", j + localNumAdded, Second[k]))
24             localNumAdded <- localNumAdded + 1
25          numAdded <- numAdded + localNumAdded
26          j <- foundIndex
27
28       Pro každý zbývající prvek v seznamu Second
29          Changes.Append(("add", i + numAdded, Second[j]))
30
31       return Changes
```

Zdrojový kód 1: Pseudokód algoritmu diff

3.3 Implementace patch

Účelem *patch* je aktualizovat soubor na základě seznamu operací. Podle dané operace dochází buď k přidání položky do seznamu na daný index, nebo odebrání ze seznamu. Čísla reprezentující řádky v jednotlivých operacích tedy odpovídají indexům seznamů.

Zdrojový kód 2 pro *patch* zahrnuje funkce *add* a *remove* ilustrují činnosti pro manipulaci řádků souboru. Níže uvedené kroky popisují jeho funkčnost.

1. Iteruj seznamem trojic.
2. Na základě operace rozhodni, zda dojde k přidání nebo odebrání řádku.
3. Prováděj operace nad kopií vstupních dat.
4. Takto upravená kopie je výstupem.

```
1  def add(data: list, line: int, payload):
2      return data.insert(line, payload)
3
4
5  def remove(data: list, line: int):
6      return data.pop(line)
7
8
9  def patch(original: list, log: list) -> list:
10     data = original.copy()
11
12     for operation, line, payload in log:
13         if operation == "remove":
14             remove(data, line)
15         elif operation == "add":
16             add(data, line, payload)
17
18     return data
```

Zdrojový kód 2: Zdrojový kód pro patch

3.4 Reprezentace metadat verzovacího systému

Tato sekce se detailně věnuje rozboru jednotlivých prvků metadat a následně je uvádí do celého kontextu, aby byl zřejmý jejich vzájemný vztah a účel.

Základní jednotkou změny **souboru** je *log* a datová struktura symbolizující tuto změnu se skládá z následujících položek:

- *uuid*: univerzálně unikátní identifikátor[7]
- *operation*: název operace
- *source*: cesta ke zdrojovému souboru
- *path*: cesta k souboru obsahující změny
- *created_at*: datum vytvoření

Během úpravy jedné verze může docházet ke změnám ve více souborech. Základní jednotkou změny **verze** je *commit*, který jednotlivé logy seskupuje pomocí seznamu. Jeho datová struktura obsahuje tyto položky:

- *uuid*: stejné jako u logu
- *created_at*: stejné jako u logu
- *message*: popis úpravy
- *logs*: seznam logů

Systém dovoluje tvorbu libovolného počtu verzí, přičemž každá z nich zahrnuje seznam provedených úprav. Data jsou organizována takto:

- *uuid*: stejné jako u předešlých
- *created_at*: stejné jako u předešlých
- *name*: název verze
- *commits*: seznam všech úprav verze

V současné verzi nástroje univerzálně unikátní identifikátor (UUID) nemá žádné konkrétní využití. Je to způsobeno skutečností, že některé příkazy a specifické vlastnosti ostatních verzovacích systémů dosud nebyly implementovány. Kapitola 8 uvádí několik příkladů, kde budou užitečné.

Bylo rozhodnuto, že všechny změny budou zaznamenávány do jednoho JSON souboru umístěném ve skryté složce *.vc*, která se nachází v kořenovém adresáři verzovaného programu.

Po definování všech datových struktur je nyní možné specifikovat strukturu metadat jako celku. Následuje seznam klíčů celého JSON objektu spolu s jejich hodnotami.

1. *current_version*: název aktuální verze
2. *stage*: seznam logů
3. *versions*: seznam verzí

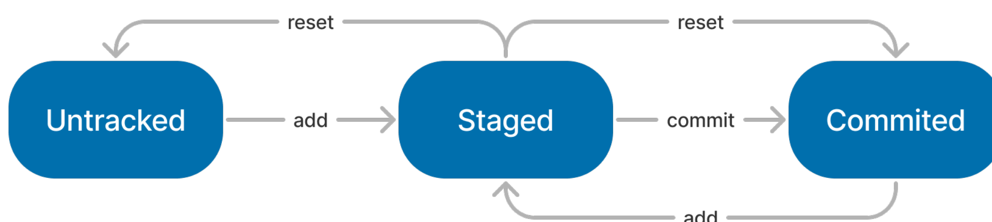
Stage lze názorně přirovnat k prostoru, kde jsou jednotlivé soubory připravovány pro další úpravu. Soubor se nachází v této fázi, pokud nad ním byla provedena operace verzovacího systému *add* (více v sekci 4.2).

Z důvodu zajištění přehlednějšího kódu a předejití potenciálním chybám je tento soubor při serializaci převeden do interní formy pomocí speciálních datových tříd. V jazyce Python se tyto třídy definují pomocí klíčového slova *@dataclass*. Definice těchto tříd je uvedena v kódu 3.

```
1  @dataclass
2  class Metadata:
3      current_version: str
4      stage: list[Log]
5      versions: list[Version]
6
7  @dataclass
8  class Version:
9      name: str
10     uuid: str
11     created_at: str
12     commits: list[Commit]
```

Zdrojový kód 3: Datový objekt pro metadata a verzi

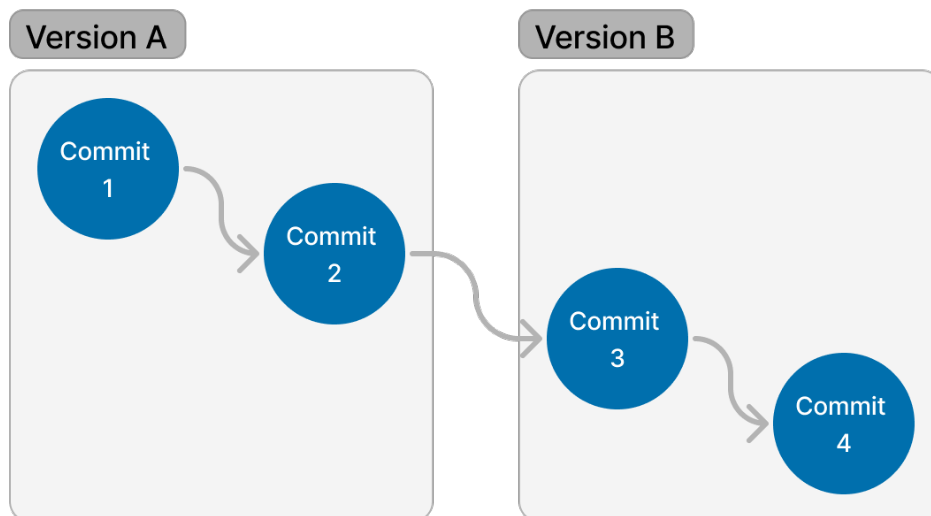
Diagram 3 ilustruje fáze, ve kterých se soubor může nacházet, a operace, které jsou s těmito fázemi spojeny. Jednotlivým operacím se věnuje následující kapitola. Soubor je ve fázi *untracked*, pokud nad ním ještě nebyly provedeny žádné operace. Jakmile soubor dosáhne fáze *committed*, do stavu *untracked* se již nevrátí.



Obrázek 3: Diagram ilustrující fáze souboru a operace umožňující přechody mezi těmito fázemi.

3.5 Propagace změn do dalších verzí

Z předešlého určení struktury metadat pro verzovací systém je možné si povšimnout, že nepočítá s možností větvení, a jednotlivé inkrementy verzí jsou dány posloupností v seznamu. Možnosti větvení budou dále rozebrány v kapitole 8.

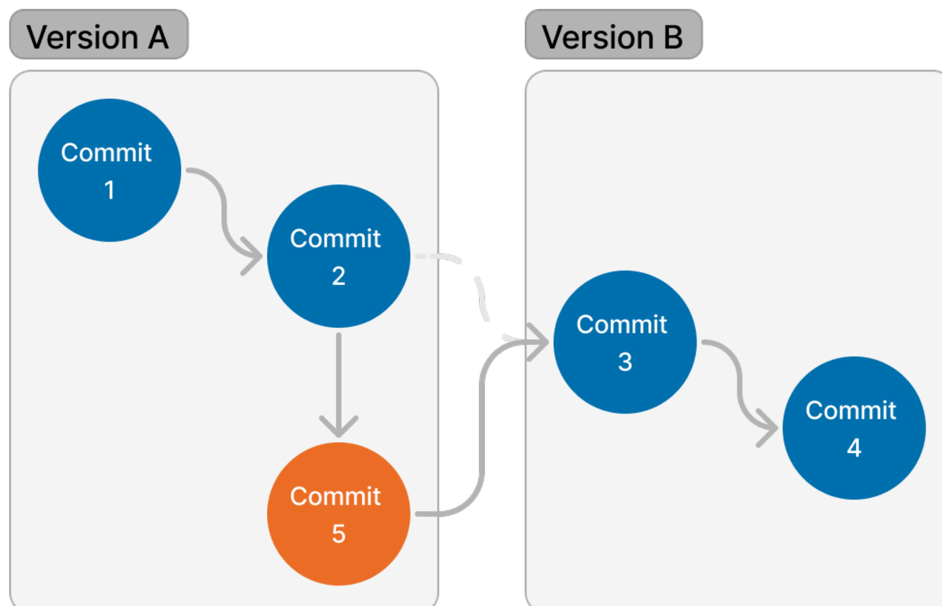


Obrázek 4: Diagram znázorňující postupné úpravy

V diagramu 4 jsou znázorněny dvě verze (A a B) a celkem čtyři úpravy. K jejich postupnému vzniku došlo následovně:

1. Inicializace repozitáře s verzí *A*.
2. Úprava *1* ve verzi *A*
3. Úprava *2* ve verzi *A*
4. Vznik verze *B*
5. Úprava *3* ve verzi *B*
6. Úprava *4* ve verzi *B*

Další diagram 5 ukazuje příklad, kdy by se uživatel chtěl vrátit k předchozí verzi *A*, provést v ní editaci a propagovat ji pak do dalších verzí. Pátá úprava by byla přidána na konec seznamu verze *A* a všechny úpravy v následujících verzích by musely tuto úpravu zohlednit.



Obrázek 5: Diagram znázorňující editaci předchozí verze

3.5.1 Zohlednění editací předchozích verzí

Bylo nutné zajistit způsob, aby úpravy předchozích verzí nenarušily sestavování celého souboru. Sestavování celého souboru závisí na postupném přidávání a odebírání řádků. V době vzniku verze *B* a úpravy *3* se vycházelo z kontextu úpravy *2*. Tím je myšleno provedení operace diff a následná identifikace řádků, které je nutné přidat, nebo případně odebrat.

Pro zohlednění editací předchozí verzí bylo rozhodnuto o nutnosti úpravy sestavování jednotlivých souborů. Namísto pouze postupného aplikování změn jednotlivých řádků bylo potřeba u konkrétních řádků přepočítat, ke kterému z nich bude změna aplikována.

Sestavování souborů je možné pozorovat v pseudokódu 4. Postupně se prochází všechny změny. Pořadí změn je dáno pořadím verzí. Před jejich aplikováním se přepočítávají řádky následovně:

1. Pokud v seznamu všech změn je změna, která byla vytvořena až po aktuálně iterované, ale v pořadí je uvedena dříve, dochází k posunu řádku. Tento posun je postupně spočítán na základě operací *add* (inkrementace) a *remove* (dekrementace).

2. Jinak se nepřepočítávají.

```
List-From-Logs(seznam Logs[dataobjects.Log])
   Final, Timestamp_Difflogs <- prázdný seznam

   for each log in Logs
      difflog <- seznam atomických operací z log.path
      Timestamp_Difflogs.Append((log.created_at, difflog))

   for each created_at, difflog in Timestamp_Difflogs

      After <- Operace vytvořené později, ale z předchozích verzí.

      Updated <- prázdný seznam

      for each operation, line, payload in difflog
         for each compared_op, compared_line in difflogs_after
            if compared_line <= line:
               if compared_operation = "add":
                  line += 1
               elseif compared_operation = "remove":
                  line -= 1

         Updated.Append((operation, line, payload))

      Final <- patch(Final, Updated)

   return Final
```

Zdrojový kód 4: Algoritmus procesu sestavování souboru

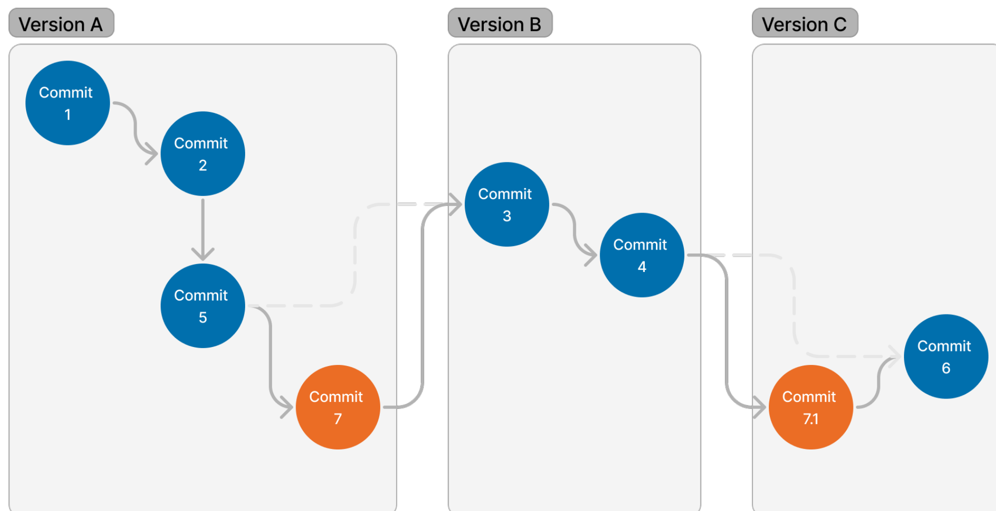
3.5.2 Inverze editací

V případě, kdy chceme docílit úpravy některé verze nebo verzí, ale zároveň zamezit propagace změny do některé z následujících, je možné využít inverzi. V diagramu 6 je možné vidět probíhající úpravu verze *A* s propagací pouze do verze B a následnou inverzí změn ve verzi *C*. Změna číslo *6.1* obsahuje inverzi vůči změně *6* a to následovně:

1. K operaci add je inverzní operace remove.
2. K operaci remove je inverzní operace add.
3. Pořadí všech operací je obrácené

Obrázek 6: Diagram znázorňující inverzi

V následujícím diagramu 7 je možné vidět spojení obou případů do jednoho. Nejdříve došlo k úpravě číslo *5* verze A propagované do všech verzí. Následně došlo k vzniku verze C s úpravou *6*. Nakonec dochází k úpravě verze A s propagací pouze do verze B.



Obrázek 7: Diagram znázorňující sloučení případů 5 a 6.

4 Implementace příkazů verzovacího systému

4.1 Příkaz init

Tento příkaz slouží k inicializování repozitáře v aktuálním pracovním adresáři. Inicializace spočívá ve vytvoření souboru pro metadata ve skrytém adresáři *.vc*. Nejdříve je provedena kontrola, zda tato složka v adresáři již existuje. Na základě argumentu pro název verze se v souboru s metadaty vytvoří prázdná verze a nastaví se jako aktuální. Vzhledem ke krátké implementaci modulu je celý jeho obsah zobrazen v kódu 5, včetně použití knihovny *argpars*.

Významná je *Abstract Base Class* IRunnable, která funguje jako rozhraní a definuje abstraktní metodu *run*. Každou třídu, implementující IRunnable, lze charakterizovat jako *command*, tedy příkaz. Každý takový příkaz je koncipován s cílem aby jej bylo možné použít jako operaci tohoto nástroje prostřednictvím příkazového řádku.

Z počátečního modulu main.py se předává modulům příkazů *subparsers*. Ty následně definují své argumenty, které je možné využít v příkazovém řádku.

4.2 Příkaz add

Tento příkaz slouží k přidání souborů do fáze stage. Jako argument přijímá seznam cest k souborům či adresářům, u kterých kontroluje, zda došlo ke změně.

K tomuto zjištění potřebuje pro každý ze zadaných souborů sestavit jeho celý obsah. Soubory jsou sestavovány postupnou aplikací všech logů pro daný soubor z předešlých verzí. Dále dojde k provedení operace diff mezi sestaveným souborem z logů a aktuálním souborem. Jak již bylo zmíněno, ve skutečnosti

```python
"""Init module"""
import config
import command
from version import create_version


class InitCommand(command.IRunnable):
    """Init command"""

    def __init__(self, subparsers):
        self.parser = subparsers.add_parser("init", help="Initialize")
        self.parser.add_argument(
            "-v", "--version-name", required=True, help="Version name"
        )
        self.parser.set_defaults(func=self.run)

    def run(self, args):
        if config.path_meta.is_file():
            print("Version Control already initialized.")
            return

        metadata = config.dataobjects.Metadata(
            current_version=args.version_name,
            stage=[],
            versions=[create_version(args.version_name)],
        )

        config.path_meta.parent.mkdir(exist_ok=True, parents=True)
        config.path_meta.write_text(config.serialize_meta(metadata))
```

Zdrojový kód 5: Zdrojový kód modulu init.

jde o rozdíl dvou seznamů. Ze zadaného souboru jsou všechny řádky načteny do seznamu a porovnány pomocí diff.

Pokud by tedy v celém adresáři došlo ke změně jen v případě jednoho souboru, je efektivnější tento soubor přímo specifikovat namísto specifikování celé složky. Docházelo by totiž k postupné aplikaci všech atomických operací *add* a *remove*, uložených v Log strukturách odpovídajících souborům dané složky, tak k postupnému čtení všech souborů a jejich porovnání se soubory sestavenými.

Při zjištění o změně v souboru se vytvoří nový soubor ve skryté složce *.vc/logs*, do kterého se zapíšou zjištěné změny, pro připomenutí se jedná o seznam atomických operací. Následuje vytvoření nové instance datové struktury pro Log s daty reprezentujícími změny. Jako hodnota operace je zvolen řetězec “add”. Nakonec se tento log přidá na konec seznamu *staged* v metadatech. Tímto způsobem se proces opakuje pro každý soubor, u kterého bylo zjištěno, že došlo ke změně.

Jednou z věcí, jíž je třeba zajistit, je přidávání upravených souborů, které se ve fázi *stage* již nacházejí. V takovémto případě dochází před předchozím krokem

ke smazání neaktuálního souboru ve skryté složce *.vc/logs* a odebrání položky ze seznamu *staged*.

Tento nástroj v současné době podporuje pouze jednu operaci pro sestavování souborů z logů a to právě operaci *add*. Záměrem je také doplnit operace *move* a *remove*, o tom dále v kapitole 8.

4.3 Příkaz commit

Tento příkaz slouží k přesunu souborů z fáze *staged* do fáze *committed* a zaznamenání tak změn vzhledem k dané verzi. Povinnými argumenty příkazu jsou *message*, neboli zpráva a také seznam verzí, do kterých budou změny propagovány.

V současné době je způsob uspořádání verzí pouze lineární, každá verze má pouze jednoho předchůdce, až na verzi první s předchůdcem žádným. Dále je momentálně implementován pouze případ, ve kterém jsou změny propagovány do následujících verzí.

Příkaz commit na začátku kontroluje, zda je v seznamu verzí aktuální verze, zda seznam neobsahuje předchozí verzi a zda obsahuje pouze existující verze. V opačném případě příkaz končí.

Dalším krokem je inicializace datového objektu *Commit* s tím, že parametru *logs* je předána hodnota celého seznamu *stage*. Tento objekt je přidán na konec seznamu aktuální verze a v posledním kroku se vyprázdní pole *stage*.

Speciální případ nastává, pokud je spolu s příkazem *commit* použit volitelný argument *–dry-run*, přeložitelný jako *nanečisto*. Dojde k vytištění změn, ke kterým by došlo, a to včetně verzí, do kterých se změna propaguje. Soubory zůstávají ve stavu *staged* a nedochází ke změnám.

4.3.1 Implementace inverzí

Problém inverzí byl vysvětlen v sekci 3.5.2 a je implementován tak, že se nejdřív určí, ke kterým verzím je potřeba vytvořit inverzi. K těmto se vytvoří nový datový objekt *Commit* s inverzními změnami. Ten se nakonec přidá na začátek seznamu úprav pro danou verzi.

4.3.2 Zpětné inverze

Může nastat situace, kdy změna není propagována do následující verze, ale je propagována do verze po ní. Tento případ je řešen pomocí zpětné inverze. Během kroku určení verzí pro inverzi se rozhoduje, do kterých verzí budou změny znovu propagovány. Do těchto verzí je na začátku přidána kopie původních změn. Pseudokód v 6 ilustruje algoritmus pro zjištění verzí, které budou invertovány, nebo u kterých bude případně provedena zpětná inverze.

```
1  Get_Versions_To_Inverse(seznam[string] Selected_Versions)
2     Inverse, Re_Inverse <- prázdná množina
3
4     for each version in Selected_Versions
5        Following <- všechny následující verze pro version
6        inverse_found <- False
7
8        for each f_version in Following
9           if f_version not in Selected_Versions and not inverse_found
10             Inverse.add(f_version)
11             inverse_found <- True
12          elseif f_version in Selected_Versions and inverse_found
13             Re_Inverse.add(f_version)
14
15    return list(Inverse), list(Re_Inverse)
```

Zdrojový kód 6: Pseudokód algoritmu pro zjištění verzí, které budou invertovány, nebo u kterých bude případně provedena zpětná inverze.

4.4 Příkaz checkout

Účelem příkazu *checkout* je přepínání mezi verzemi. Pokud se uživatel pokouší přepnout na verzi, která neexistuje, tento příkaz ji automaticky vytvoří a přidá na konec seznamu všech verzí. Při změně verze dojde k přepsání obsahu souborů na základě jejich sestavení z logů, postupným aplikováním operací *add* a *remove*. Tento krok sestavení z logů realizuje algoritmus uvedený v 4.

Při vytváření nové verze je možné využít argument *-f, –from-version* a specifikovat nově vytvořenou verzi jako přímého následovníka té aktuální. Vzhledem k lineární povaze uspořádání verzí však nedochází k větvení, ale k posunutí celého seznamu verzí. Tedy, nově vytvořená verze se zařadí mezi existující verze, přičemž všechny následující verze se posunou o jedno místo vpřed.

4.5 Příkaz reset

Příkaz *reset* slouží k navrácení souborů nacházející se ve fázi *staged* do jejich původní stavu. Tento původní stav může odpovídat fázím *untracked* nebo *committed*. Na pozadí funguje tak, že provede příkaz *checkout* na aktuální verzi, následně smaže všechny soubory ve skryté složce *.vc/logs*, které odpovídají souborům ve fázi *staged*, a seznam *staged* nastaví na prázdný. Pro přeskočení kroku aktualizace souborů se může využít argument *-p, –preserve*.

5 Příklady nástrojů pro tvorbu textů k programu

Tato kapitola je věnována pár zástupcům nástrojů, které je možné vyžít pro tvorbu podpůrných textů.

5.1 Markdown

Markdown je jedním ze značkovacích jazyků[5] a umožňuje uživatelům psát textové dokumenty s malou sadou odlehčených anotací. Ze všech značkovacích jazyků je Markdown jedním z nejjednodušších, pokud ne ten nejjednodušší.[5]

Tento značkovací jazyk je velmi rozšířený a lze se setkat s jeho různými implementacemi v mnoha prostředích. V dnešní době je téměř standardem, že každý repozitář na Github či Gitlab má ve svém kořenovém adresáři soubor *readme.md*, který slouží jako úvodní stránka daného repozitáře. Řada vývojových prostředí, jako například Visual Studio Code, automaticky umožňují interaktivní náhled na vykreslený text. Vzhledem k jeho minimalistické syntaxi je tento jazyk velmi přívětivý.

Markdown má jisté nedostatky, včetně nepodpory například odkazování na různé části textu, psaní matematických vzorců a citací. Řešením tohoto nedostatku a mnoho jiných lze dosáhnout využitím nástroje *Pandoc*, kterému se věnuje část 5.5.

Díky své jednoduchosti byl jazyk Markdown vybrán jako vhodný kandidát pro vytvoření prototypu nástroje určeného pro tvorbu textové dokumentace k programům.

5.2 TEX

TEX je typografický systém, který vytvořil Donald E. Knuth na konci 70. let kvůli nespokojenosti se sazbou matematiky, hlavně v monografii The Art Of Computer Programming. Jedná se o značkovací programovací jazyk určený pro sazbu knih a jiných dokumentů.[9].

Nad tímto systémem je postaven LATEX. Program LATEX je speciální verzí TEX, který rozumí LATEX příkazům.[10] V podstatě se jedná o rozšíření s doplněnými makry, které zjednodušují proces sazby a jiné.

Podpora odkazů v dokumentu, poznámek pod čarou a řešení citací je jednou z klíčových vlastností, které zvyšují uživatelskou přívětivost a praktičnost tohoto systému. Práce s citacemi je dále usnadněna nástrojem BibTEX nebo jeho možnými alternativami, jenž mají schopnost automaticky generovat seznam citací, na které se jednotlivé odkazy v textu odkazují.

[5]Značkovací jazyky (z anglického markup) se více zaměřují na sémantiku informací namísto přímému formátování, ke kterému dochází u WYSIWYG (co vidíš, to dostaneš) formátování. U těchto jazyků je možné určovat informaci o tom, kde kapitoly začínají, ale ne jakým způsobem budou nadpisy vypadat.[5]

5.3 Jupyter notebook

Jupyter notebooks představují jeden z nejpopulárnějších nástrojů v oblasti datové vědy. Tento nástroj spojuje text, kód a grafy do jednoho dokumentu, čímž uživatelům umožňuje konzistentně opakovat datovou analýzu. [11]. V roce 2017 získal projekt Jypyter ACM Software System ocenění, stejné ocenění získal i TeX v roce 1986.

Dalším výhodným prvkem tohoto řešení je možnost spouštění jednotlivých částí kódu a zobrazování jejich výsledků. Nicméně, z pohledu některých uživatelů, tento nástroj nemusí být nejvhodnější prostředí pro dokumentaci mnohosouborových projektů, jelikož byl primárně navržen pro interaktivní výpočty, a ne nutně pro komplexní správu dokumentace.

5.4 GitBook

Během hledání nástroje, který by byl propojen s repozitářem, byl objeven GitBook. Na rozdíl od *Jupyter notebooks* se toto řešení jeví jako vhodné pro dokumentaci velkých projektů. Navázání na repozitář je realizováno prostřednictvím integrací, které jsou v současné době omezeny pouze na GitHub a GitLab. Dále se předpokládá, že tato dokumentace bude oddělena od samostatného kódu a neumožňuje použití dynamických úryvků kódu (snippetů), jak by mohlo být od vyvíjeného nástroje očekáváno. Snippety lze definovat pouze staticky. Pro jednotlivce je produkt dostupný zdarma, ale pro týmové využití je zpoplatněný.

5.5 Pandoc

Pro vytvoření takového nástroje, jenž by vyhovoval stanoveným požadavkům, bylo nutné vybrat nástroj, který by umožňoval zpracování určitého značkovacího jazyka do formátu vhodného pro tisk nebo web.

Pandoc, knihovna napsaná v programovacím jazyce Haskell[6], slouží k převádění dokumentů mezi různými značkovacími jazyky. Tato knihovna, která je dostupná i prostřednictvím příkazového řádku, umožňuje také konverzi mezi různými formáty textových procesorů a dokáže produkovat i výstup ve formátu PDF. Pandoc rozšiřuje standardní Markdown o syntax pro tabulky, poznámky pod čarou, citace, matematické vzorce a mnoho dalších funkcí.[12]

Pandoc také podporuje možnost využití šablon, které mohou být při exportu uplatněny, například s využitím TEX souborů. Původní ambice spočívala v generování textu této práce právě s využitím tohoto přístupu. Nicméně se ukázalo, že pro tento typ textu vyžadující detailní pozornost k sazbě, alespoň prozatím nepředstavuje tento nástroj ideální volbu.

[6]Haskell je funkcionální programovací jazyk pro všeobecné použití.[13]

6 Implementace nástroje pro tvorbu textů k programům

Prvním podporovaným značkovacím jazykem, který byl vybrán pro tento nástroj, je Markdown. Tato volba byla učiněna na základě různých důvodů, které jsou vysvětleny v sekci 5.1. Společně s nástrojem Pandoc, popsaným v sekci 5.5, je tento jazyk používán pro tvorbu textů, které je následně možné exportovat do formátu PDF.

Nástroj pro tvorbu těchto dokumentů byl nakonec rozhodnuto integrovat přímo do primárního nástroje pro verzování. Toto rozhodnutí bylo učiněno vzhledem k tomu, jak těsně jsou tyto dvě funkce vzájemně provázány. Nyní je tento integrovaný nástroj pro tvorbu textů přístupný jako příkaz *publish*, což umožňuje tvorbu a export textových dokumentů přímo z verzovacího nástroje.

6.1 Dynamické snippety a odkazy

K dosažení cíle, tj. automatická úprava z propagovaných změn, nám tento nástroj definuje dva nové elementy: *snippet* a *snippet-link*.

Snippet reprezentuje blok kódu, odpovídá mu značka <*snippet*> a má tři argumenty:

1. *path* - cesta k souboru
2. *start* - řádek v souboru od
3. *end* - řádek v souboru do

Značka <*snippet-link*>, která reprezentuje odkaz na řádek kódu ve zdrojovém dokumentu, má dva argumenty:

1. *path* - cesta k souboru
2. *line* - číslo řádku

6.2 Příkaz publish

Vytvořením nových elementů dochází ke vzniku nadstavby jazyka Markdown, která není kompatibilní ani s nástrojem *Pandoc*. Při exportu do formátu PDF je nutné provést mezikrok. Tento proces, včetně jeho exportu, je označen termínem *publikování*. Příkaz pro publikování (*publish*) vyžaduje dva argumenty: cestu k souboru nebo souborům, jež mají být publikovány, a výsledný formát. Aktuálně jsou podporované formáty PDF a Markdown. V kapitole 8 je věnována část možnosti dalších formátů.

V případě publikace ve formátu Markdown jsou všechny elementy nahrazeny bloky kódu a všechny elementy číslem, které odpovídá číslu řádku. Při přepočítávání řádků je brána v úvahu interní funkcionalita nástroje pro verzování.

Verzovací systém v současné době neupravuje výchozí soubor, což může být pro uživatele matoucí, jednotlivá čísla totiž nejsou automaticky přepisována a v budoucnosti bude této záležitosti věnována zvýšená pozornost.

V případě publikace formátu PDF je dokument nejdříve publikován do formátu Markdown. Následně je za použití nástrojů *Pandoc* a *pdfLatex* konvertován do souboru PDF. Interně Pandoc nejdříve vytváří dokument ve formátu TeX, který je poté převeden do PDF.

7 Uživatelská příručka

V kapitolách 3 a 6 byly popsány implementace jednotlivých příkazů nástrojů. V následující kapitole je popsáno, jakým způsobem je možné s nástroji zacházet.

Jak již bylo zmíněno, oba nástroje jsou ve skutečnosti vzájemně propojeny do jednoho celku, a proto budou interakce s nimi popsány společně.

7.1 Instalace nástroje

Nástroje se spouštějí prostřednictvím skriptu main.py, který je implementován v jazyce Python. V budoucnu by bylo možné z tohoto nástroje vytvořit *pip*[7], nebo jiný balíček, ale v současné době je nutné při volání každého příkazu uvádět `python path/to/main.py`. Jako dočasné řešení pro snazší použití byl vytvořen *alias*[8] `vc`.

Je podstatné upozornit, že nástroje byly vyvíjeny v Pythonu verze 3.11., který je v současnosti nejnižší požadovanou verzí pro jejich spuštění.

Dále je pro funkčnost příkazu *publish* ve formátu PDF nezbytné nainstalovat následující nástroje:

1. *pandoc*: nástroj pro konverzi mezi různými značkovacími jazyky
2. *pandoc-crossref* - rozšíření pro pandoc umožňující odkazování
3. *pdflatex* - nástroj pro převod výstupu do formátu PDF

7.2 Příkazy

Pro přehlednost a srozumitelnost je seznam příkazů s jejich argumenty uveden v tabulce 1. Tato tabulka by měla sloužit jako referenční bod pro uživatele při práci s tímto nástrojem.

Na úvod je také vhodné poukázat na možnost zobrazení nápovědy pomocí argumentu *-h*. Tato nápověda je generována přímo knihovnou argparse.

[7]Pip je jedním ze správců balíčku. Ty poskytují nástroje, které pomáhají uživatelům instalovat závislosti na úrovni projektu, programovacího jazyka, frameworku, nebo operačního systému.[14]

[8]V tomto kontextu je pojmem *alias* myšlena zkratka v prostřední příkazového řádku.

7.2.1 Příkaz init

Nástroj umožňuje inicializaci repozitáře prostřednictvím příkazu *init*. Tento příkaz vyžaduje jeden povinný argument: název verze.

```
vc init -v v1
```

Po spuštění tohoto příkazu se v aktuálním adresáři vytvoří skrytá složka *.vc*. Mezi aktuálními omezeními nástroje je skutečnost, že všechny příkazy je nutné volat přímo v inicializovaném adresáři. Navíc, na rozdíl od některých jiných verzovacích systémů, jako je například *Git*, neexistuje automatické zaznamenávání změn souborů.

7.2.2 Příkaz add

Tento příkaz slouží k přidání souboru do fáze *stage*, popsané v sekci 3.4. K samotnému zjištění změn souboru dochází během této operace. Lze použít pro jednotlivé soubory, více souborů najednou, nebo celou složku, jak je demonstrováno níže:

```
# jeden soubor
vc add -p test1.c

# více souborů
vc add -p test1.c dir/test1.c

# celá složka
vc add -p dir
```

7.2.3 Příkaz commit

K zaznamenání změn, které byly přidány do fáze *stage* předchozím příkazem, slouží *commit*. Použití tohoto příkazu je následující:

```
# propagace změn pouze verzím v1 a v2
vc commit -m "update return" -v v1 v2

# propagace změn všem následujícím verzím
vc commit -m "update return" -all

# náhled na potencionální změny
vc commit -m "message" -v v1 --dry-run
```

Následující výstup ilustruje potenciální výsledek volání příkazu s argumentem *–dry-run*. Tento argument umožňuje uživateli zobrazit předpokládané změny pro každou vybranou verzi, aniž by byly tyto změny skutečně provedeny.

```
Version: v1:
--- Source: main.c: ---
+ // add author
  #include <stdio.h>

+ int foo();
+
  int main(int argc, char* argv[])
  {
      printf("Hello, World!");
+     foo();
      return 0;
+ }
+
+ int foo()
+ {
+     printf("Function foo");
  }
--- End of source ---
```

7.2.4 Příkaz checkout

Tento příkaz poskytuje možnost přepínání mezi jednotlivými verzemi. V případě, že cílová verze neexistuje, příkaz ji automaticky vytvoří jako novou.

```
# změna na verzi v2 nebo její případné vytvoření
vc checkout -v v2
```

7.2.5 Příkaz version

Pro určení aktuální verze nebo pro zobrazení všech dostupných verzí slouží příkaz *version*. Bez poskytnutí argumentu vrací informace o aktuální verzi. Použití příkazu s argumentem *-l (–list)* vypíše seznam všech existujících verzí.

```
# vytiskne aktuální verzi
vc version

# vytiskne všechny verze
vc version -l
```

7.2.6 Příkaz diff

Podobně jakým *commit –dry-run* zobrazuje změny souborů, příkaz *diff* slouží k vizualizaci rozdílů v daném souboru mezi dvěma verzemi. Účelem je poskytnout uživateli přehled o změnách, které se mezi danými verzemi udály.

```
# náhled na změny souboru main.c mezi verzemi v1 a v2
vc diff -f v1 -t v2 -p main.c
```

7.2.7 Příkaz publish

Kromě verzování, tento nástroj nabízí možnost publikování podpůrných textů k programům pomocí příkazu *publish*. V současnosti je jediným podporovaným formátem pro vstup formát Markdown. Pokud jde o výstupy, mohou být dokumenty publikovány do formátu Markdown nebo PDF. Tyto dokumenty jsou publikovány do složky *.output* a verzovací systém je automaticky ignoruje. K publikaci je také možné využít šablonu. Tento příkaz funguje i bez inicializovaného repozitáře.

```
# tutorial.md bude publikován ve formátu Markdown
vc publish -p tutorial.md -f md

# publikování všech dokumentů v adresáři
vc publish -p * -f md

# volba PDF jako výstupní formát
vc publish -p input.md -f pdf

# publikování pomocí šablony
vc publish -p input.md -f pdf -t template/kidiplom.tex
```

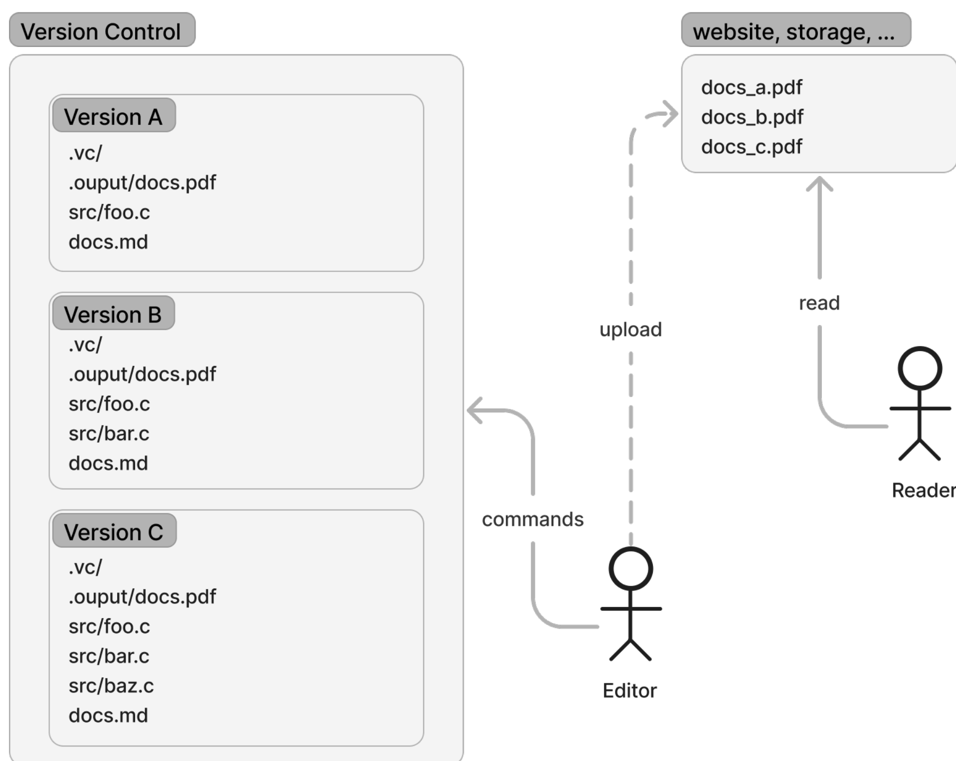
Tabulka 1: Seznam příkazů a jejich argumenty

Příkaz	Argumenty
init	
	-v, –version : inicializovaná verze
add	
	-p, –path : cesta k souboru
checkout	
	-v, –version : název verze
	-f, –from-versions : verze, ze které proběhne případné vytvoření
commit	
	-m, –message : zpráva
	-v, –versions : verze, kterými bude změna propagována
	-all, –all-versions : propagace všem následným verzím
	–dry-run : náhled na změny, nedojde k aplikaci
diff	
	-f, –from-version : první verze
	-t, –to-version : druhá verze
	-p, –path : cesta k souboru
	–files : zda porovnávat dva soubory Příklad: –files -f file1 -t file2
version	
	-l, –list : seznam všech verzí
reset	
	-p, –preserve : zda ponechat soubory beze změny
publish	
	-p, –path : cesta k souboru, souborům nebo adresářům
	-f, –format : výstupní formát
	-t, –template : cesta k šabloně

7.3 Případ užití

Obrázek 8 demonstruje, jakým způsobem může *editor* podpůrných textů pracovat s tímto nástrojem. Využitím příkazů popsaných v této kapitole lze tvořit jednotlivé verze programu a v průběhu tohoto procesu publikovat podpůrné texty. Tyto texty mohou být následně sdíleny čtenářům mimo rámec tohoto nástroje.

Tento nástroj neobsahuje žádné aspekty pro víceuživatelskou interakci, existuje pouze jedna role, která je schopna vykonávat všechny příkazy. Navzdory tomu je tento nástroj kompatibilní s dalšími verzovacími systémy, které by mohly umožnit distribuci inicializovaného adresáře všem uživatelům. Předpokladem pro takovou interakci je, že by všem uživatelům byla sdílena i složka *.vc*. Rozšíření tohoto diagramu je prezentováno v následující kapitole.



Obrázek 8: Diagram případu užití

8 Budoucí rozvoj nástrojů

8.1 Implementace příkazu pro přesun souborů

V rámci budoucího rozvoje verzovacího systému je klíčové implementovat příkaz pro přesun souborů, který by změny zaznamenával do metadat.

8.2 Podpora dalších formátů

Rozšíření podpory dalších formátů by představovalo vylepšení pro uživatele. Autorům by to umožnilo psát dokumentaci v jazyce, která je pro ně nejvýhodnější. Co se týče výstupních formátů, rozšíření podpory by znamenalo možnost využití dokumentů pro více účelů.

Jeden z možných formátů na výstup je *HTML*. U tohoto formátu by bylo nutné navíc umožnit specifikovat styly v jazyce *CSS*.

8.3 Přepisování hodnot snippetu a určení verze

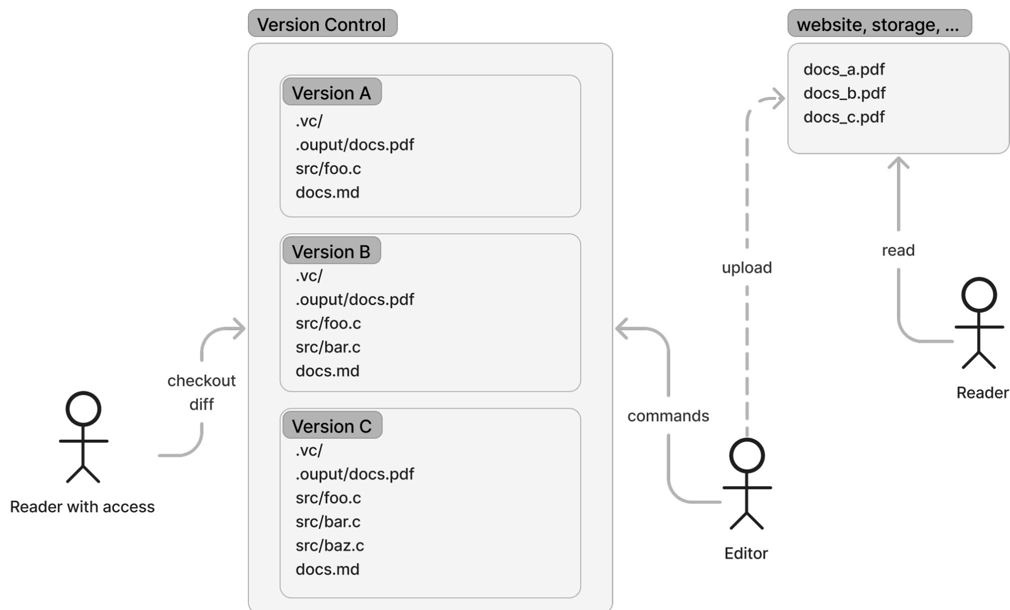
Problém nejasnosti čísel řádků v elementech *snippet* může být vyřešen přepsáním vstupního souboru během operací, jako jsou *commit* a *checkout*, což by vedlo k větší přehlednosti a přesnosti odkazů na konkrétní řádky.

Dalším směrem rozvoje může být rozšíření elementů o nový parametr, který by umožnil specifikovat verze. Snippet by se tak mohl odkazovat i do předchozích a následujících verzí. Navíc by to umožnilo autorům dokumentace lépe ilustrovat a vysvětlit změny v kódu v průběhu času.

8.4 Podpora více uživatelů

Rozšíření verzovacího systému o podporu více uživatelů zahrnuje otázky jako Git vs Subversion přístup (distribuovaný vs centralizovaný).

Nástroj by bylo možné rozšířit o uživatelskou roli, která je oprávněná pouze na nějaké příkazy. Tohle je zobrazeno v rozšířeném diagramu případů užití 9



Obrázek 9: Diagram možného případu užití s dodatečnou uživatelskou rolí

8.5 Další možnosti

- Podpora větvení verzí s využitím UUID.
- Příkaz pro vypsání změn souborů pro sledování historie.
- Optimalizace pomocí mezivýsledků a přepočítání operací při změnách v souborech.
- Zakódování skrytých souborů do úspornějšího formátu nebo do formátu signalizujícího změny.

Závěr

Byla navržena sada nástrojů společně plnící funkci verzovacího systému a nástroje pro tvorbu podpůrných textů. Tyto nástroje umožňují demonstraci procesu vytváření rozsáhlých programů. Verzovací nástroj podporuje inkrementálně vytvářet verze programu, vracet se k předchozím verzím, včetně možnosti jejich editace a propagace změn do dalších verzí. Při tvorbě podpůrných textů lze z vlastností propagací vytvářet bloky kódů, tzv. *snippety*, které se na základě změn automaticky aktualizují. Tvorba těchto dokumentů je momentálně omezena pouze na formát Markdown, jejich výstupem však může být i dokument ve formátu PDF.

Dalším omezením je, že propagace změn je momentálně implementována pouze do novějších verzí. Je tedy nutné se vždy přepnout na nejstarší verzi, z níž jsou změny aplikovány.

Považuji za milou povinnost ještě jednou poděkovat svému vedoucímu, Mgr. Petru Krajčovi, Ph.D., za odborné vedení a za jeho trpělivost při vývoji tohoto nástroje. Konfrontace s odbornými výzvami a soustavné bádání mne inspirovaly a motivovaly k navazujícímu studiu.

Conclusions

A set of tools was developed that functions both as a version control system and a tool for creating supporting text. These tools can be used for producing tutorials on advanced programming. The version control tool supports the creation of incremental versions, as well as switching to and editing previous versions. These edits, also called revisions, can then be propagated to subsequent versions.

While creating the texts, it's possible to use code snippets which are automatically updated based on the propagated changes. The creation of these documents is currently limited only to the Markdown format, but their output can also be made into a PDF format.

Another limitation is that the propagation is currently implemented only for changes made to newer versions. Therefore, it is always necessary to switch to the oldest version, make the updates there, and then apply them to the following versions.

I would like to thank my supervisor Mgr. Petr Krajča, Ph.D. once again for his expert guidance and patience during the development of this tool. Confrontation with these challenges and research tasks inspired and motivated me for subsequent studies.

A Obsah elektronických dat

Elektronická data odevzdaná v systému katedry mají následující strukturu:

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

`README.md`
Instrukce pro spuštění nástrojů, včetně všech požadavků pro její bezproblémový provoz.

`bin/`
Instalátory pro pandoc a pandoc-crossref.

`src/`
Kompletní zdrojové soubory nástrojů v jazyce Python jsou k dispozici v .zip archivu `version-control.zip`.

Literatura

[1] Wikipedia contributors. (2023, July 26). Diff. In Wikipedia, The Free Encyclopedia (cit. 2023-07-28).

[2] Chacon S., Straub B., T. (2014). Pro Git. Apress.

[3] Wikipedia contributors. (2023, March 29). Repository (version control). In Wikipedia, The Free Encyclopedia (cit. 2023-07-28).

[4] Tichy, Walter F. (1982). Design, Implementation, and Evaluation of a Revision Control System.

[5] Mailund, T. (2019). Introducing markdown and pandoc: Using markup language and document converter. Apress.

[6] Darcs: Theory/MergersDocumentation (cit. 2023-07-29).

[7] Leach P. J., Salz R., Mealling M. H. (2005) A Universally Unique IDentifier (UUID) URN Namespace. RFC Editor

[8] Beazley. D. M., Jones. B. K. (2014). Python cookbook. O'Reilly.

[9] Outrata, Jan: Diplomová propedeutika (cit. 2023-08-02).

[10] Lamport, Leslie (1994): Latex: A document preparation system

[11] Hudgeon D., Nichol R. (2019): Machine Learning for Business: Using Amazon SageMaker and Jupyter

[12] Pandoc: Pandoc User's Guide, Description (cit. 2023-08-02)

[13] O'Sullivan B., Stewart D., Goerzen J. (2008). Real World Haskell. O'Reilly.

[14] Hillard D. (2023). Publishing Python Packages: Test, share, and automate your projects. Manning Publications.