Katedra informatiky
Přírodovědecká fakulta
Univerzita Palackého v Olomouci

BAKALÁŘSKÁ PRÁCE

Grafické uživatelské rozhraní pro systém správy verzí Git

2021

Jaroslav Večeřa

Vedoucí práce: Mgr. Radek Janoštík

Studijní obor: Aplikovaná informatika, prezenční forma

Bibliografické údaje

Autor:	Jaroslav Večeřa
Název práce:	Grafické uživatelské rozhraní pro systém správy verzí Git
Typ práce:	bakalářská práce
Pracoviště:	Katedra informatiky, Přírodovědecká fakulta, Univerzita Palackého v Olomouci
Rok obhajoby:	2021
Studijní obor:	Aplikovaná informatika, prezenční forma
Vedoucí práce:	Mgr. Radek Janoštík
Počet stran:	36
Přílohy:	1 CD/DVD
Jazyk práce:	český

Bibliograhic info

Author:	Jaroslav Večeřa
Title:	Graphical user interface for version control system Git
Thesis type:	bachelor thesis
Department:	Department of Computer Science, Faculty of Science, Palacký University Olomouc
Year of defense:	2021
Study field:	Applied Computer Science, full-time form
Supervisor:	Mgr. Radek Janoštík
Page count:	36
Supplements:	1 CD/DVD
Thesis language:	Czech

Anotace

Grafické rozhraní pro Windows, které zprostředková přehlednou a intuitivní práci se základními funkcemi systému Git, a to převážně pomocí grafu.

Synopsis

Windows graphical user interface that allows intuitive git workflow using graph representation.

Klíčová slova: git; verzování; graf; grafické rozhraní

Keywords: git; version control; graphical interface

Děkuji Mgr. Radkovi Janoštíkovi za ochotu a pomoc při vývoji programu i psaní textu práce. Také děkuji rodině a blízkým osobám za podporu.

Místopřísežně prohlašuji, že jsem celou práci včetně příloh vypracoval/a samostatně a za použití pouze zdrojů citovaných v textu práce a uvedených v seznamu literatury.

datum odevzdání práce podpis autora

Obsah

Seznam obrázků

Seznam tabulek

Seznam vět

Seznam zdrojových kódů

1 Úvod

Při vývoji programů, zvláště pak těch netriviálních, je často třeba dělat změny nebo nové verze. Protože však není možné vytvořit program bez chyb, objevuje se také potřeba vracet se k libovolným starším verzím a zakládat na nich nové, či dokonce vyvíjet více verzí zároveň v případě skupiny lidí. Tato činnost lze samozřejmě provádět ručně, zabírá to ale čas a zvyšuje riziko chyby. Může dojít ke změně souboru nebo jeho smazání na špatném místě. Přece jen udržovat si v úložišti desítky záloh nebo obměn dat a spravovat je ručně vyžaduje podrobný popis jednotlivých verzí, který časem musí být značně nepřehledný. Z tohoto důvodu byly vytvořeny takzvané verzovací systémy.

Systém správy verzí (SSV) je zpravidla softwarový nástroj, umožňující spravovat verze projektu částečně automaticky (nebo alespoň přehledně a jednoduše). Přitom daný projekt nemusí být zdrojovým kódem v nějakém programovacím jazyce, může se jednat vlastně o libovolná data. Vracet zpět provedené změny, nebo pracovat ve skupině lidí může být užitečné například i grafikům, střihačům videí, spisovatelům a podobně. Systém uchovává jak soubory samotné, tak různé informace související se správou verzí. To se samozřejmě napříč konkrétními systémy liší, obvykle je ale k dispozici:

- Popis změny (ručně zadaný)
- Čas změny
- Autor změny a jeho kontaktní údaje

Tyto údaje jsou zejména užitečné v případě týmu lidí pracujících na společném projektu, historicky však nejprve vznikla skupina takzvaných lokálních SSV.

1.1 Lokální SSV

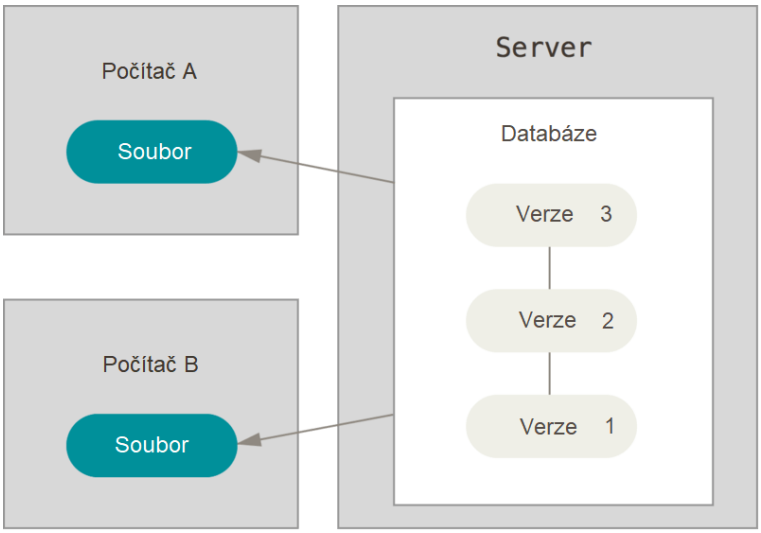
Tyto systémy se soustředily na práci jednotlivce, celý projekt byl ukládán na místním disku a nebyl nikde sdílen. Konkrétním zástupcem je například Revision Control System (RCS). RCS si uchovává poslední podobu daného souboru spolu se zpětnými rozdíly. Aplikací těchto rozdílů (delt) na soubor lze rekonstruovat některou jeho předchozí verzi.

Nevýhoda lokálního SSV je, že projekt je umístěn pouze na jednom zařízení, a při chybě disku tak hrozí ztráta dat. Další nevýhodou je nemožnost projekty pohodlně sdílet po síti.

1.2 Centralizované SSV

Centralizované systémy (CSSV) [1] jsou historicky dalším vývojovým stupněm SSV. Představují opačný extrém k lokálním SSV, na rozdíl od nich totiž většinu souborů a práci přesouvají od koncového uživatele na jeden společný centrální

server, ten je skrze síť dostupný odkudkoliv na světě. Metodu zachycuje obrázek 1 [2].



Obrázek 1: Centralizovaný systém správy verzí

Od chvíle, kdy uživatel A udělá na souboru S nějaké změny a nasdílí je do společné databáze, už žádný další uživatel nemá aktuální verzi souboru S. Pro obdržení aktuální verze musí svoji kopii každý opět aktualizovat. V případě, že další uživatel provedl jiné změny na témže souboru, systém se je pokusí sloučit dohromady. Uživatelé se tedy nemusí starat o případy, ve kterých nedojde k závažnému konfliktu. Pokud však uživatel A upravil stejný řádek souboru S, jako uživatel B, ale jinak, je třeba se o vyřešení konfliktu postarat ručně výběrem správné verze, případně spojením obou změn.

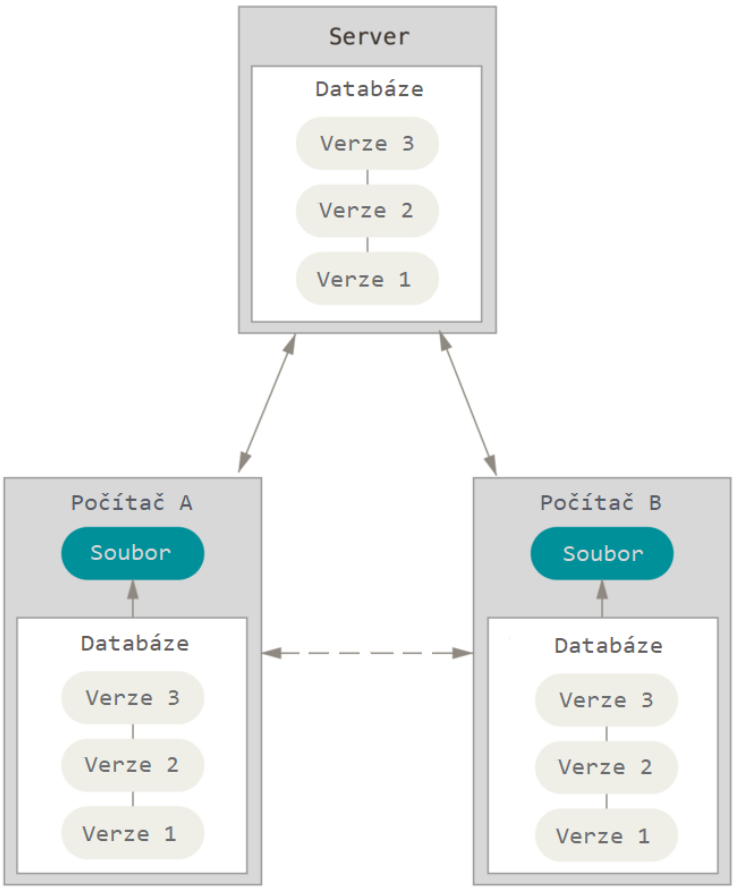
Pro snížení počtu konfliktů je v CSSV dostupná funkce větvení. Umožňuje vytvářet historii změn s jinou než lineární strukturou. Větvení se často používá pro implementaci ucelené funkcionality programu, nebo pro experimentální záležitosti, které nemusejí být po svém ukončení začleněny do projektu. Uživatel může pracovat na verzích větve aniž by ovlivňovaly verze větve jiného uživatele.

Výhodou CSSV je například šetření místa jednotlivých uživatelů. Ti na svých zařízeních mají fyzickou kopii pouze aktuální verze projektu, zbytek historie je uložen na serveru. To však přináší nemalá rizika v případě výpadku. Pokud uživatel není schopen připojení k síti, nemůže na projektu pracovat, jelikož veškerá práce se systémem vyžaduje síťové připojení. Pokud dokonce dojde k porušení této společné databáze, veškerá historie dat je nenávratně ztracena, stejně jako v lokálním SSV se totiž nachází pouze na jednom místě. Drobnou výhodou zůstává, že alespoň aktuální verze se nachází na více zařízeních.

Typickým zástupcem těchto CSSV je systém Subversion [3].

1.3 Distribuované SSV

Distribuované systémy (DSSV) [1] se vyvinuly po centralizovaných a představují jakýsi kompromis obou předchozích systémů. DSSV se snaží těžit z výhod obou metod. Projekt lze snadno sdílet pomocí sítě. Kromě uživatelů obsahuje servery, na kterých se nachází tzv. vzdálené repozitáře. Tyto repozitáře plní stejnou funkci jako v CSSV, nejedná se však o jedinou kopii projektu. Každý uživatel, který s ním pracuje, vlastní úplnou kopii celé historie. Jednotlivé repozitáře uživatelů a serverů se mezi sebou aktualizují pomocí k tomu určených příkazů. Tyto příkazy se často nazývají *push* (pro pokus včlenit změny do zvoleného vzdáleného repozitáře), fetch (pro stažení dat ze vzdáleného repozitáře) a pull (pro včlenění dat ze vzdáleného repozitáře).



Obrázek 2: Distribuovaný systém správy verzí

Jak je v obrázku 2 [2] naznačeno přerušovaným spojení mezi uživateli A a B, přímé sdílení projektu sice DSSV umožňuje, není však tolik využívané. Mnohem častěji sdílí uživatelé práci se společným serverem, který tak hraje roli jakéhosi pasivního uživatele, se kterým ostatní komunikují.

Příklady takových nástrojů jsou Mercurial[4], Bazaar[**bazaar**], nebo Git[5], kterého se tato práce týká.

1.4 Nástroje pro práci se SSV

Se systémy správy verzí lze pracovat pomocí nejrůznějších softwarových produktů. Velice často lze SSV ovládat příkazovou řádkou. Dalším vylepšením bývá možnost využít speciálního programu, který pracuje jako běžná příkazová řádka, má však zabudovaný systém zvýrazňování syntaxe, či pomocné výpisy informací o projektu. Pro nejrůznější SSV vzniká i celá řada grafických nástrojů.

Cílem této práce je právě takový grafický nástroj vytvořit. Konkrétně pro systém Git.

2 Git

Git je distribuovaný SSV vytvořený Linusem Torvaldsem pro jeho projekt Linux, kterému žádný dostupný SSV nevyhovoval. Git na rozdíl od ostatních SSV uchovává historii takovým způsobem, že většina operací nad soubory je výrazně rychlejší.

Většina verzovacích systémů má uložený soubor a pro každou jeho verzi dopředné, či zpětné rozdíly. Pomocí skládání těchto rozdílů lze znovu zhotovit libovolnou verzi souboru. Hlavní výhoda tohoto přístupu spočívá v ušetřeném místě, oproti ukládání celé kopie se totiž ušetří části, které se mezi jednotlivými verzemi nezměnily. Má to však dopad na rychlost opětovného sestrojení konkrétních verzí.

Git naproti tomu uchovává pro jednotlivé verze celé kopie (nazývají se snapchoty). Rychlost sestrojení souborů v historii tak není ovlivněna množstvím verzí, které od té doby byly zhotoveny. Git samozřejmě ukládání optimalizuje, a to tak, že pokud nejsou v souboru provedené změny, místo nového snapshotu se uloží odkaz na starý. Celý repozitář také podléhá bezeztrátové kompresi.

Verze projektu také budeme nazývat revize. Každá revize, kromě počáteční, má jeden nebo více předků. Jeden v případě prostého vytvoření další verze, více v případě slévání změn z více verzí (merge). Celá historie se tak dá reprezentovat jako orientovaný, souvislý a acyklický graf s uzly vyjadřujícími revize a hranami vyjadřujícími vztah rodič – potomek. Vytvořit aplikaci reprezentující tímto způsobem historii vytvořenou Gitem je také hlavní cíl této práce.

Jednotlivé větve pak Git ukládá vnitřně pouze jako ukazatele na poslední revizi větve. Při vytváření nových verzí se na ně tyto ukazatele posunují. Žádná data o tom, v jaké větvi byla revize původně vytvořena, nejsou k dispozici a často se nedají nijak dohledat. Tato informace bude později důležitá při popisu heuristiky rozmístění uzlů v grafu na straně 26.

Poté ještě v repozitáři existuje speciální ukazatel hlavy (HEAD), který ukazuje na větev, či přímo revizi, které jsou aktuálně prohlíženy.

Následující seznam popisuje základní funkce pro práci s Gitem.

Commit vytvoří novou verzi, přitom se na ni přesune ukazatel větve, na kterou ukazuje HEAD, případně se posune HEAD, pokud ukazuje přímo na revizi.

Branch Vytvoří nový ukazatel větve na aktuální verzi.

Checkout znovu sestrojí verzi předanou argumentem. Buď formou větve, kdy HEAD začne odkazovat na onu větev, nebo formou revize přímo, potom HEAD odkazuje na revizi a repozitář se nachází v experimentálním módu.

Stash v závislosti na argumentu ukládají, aplikují a mažou provedené změny od poslední verze na strukturu zásobníkového charakteru.

Merge spojuje vybrané větve do jedné. V případě, že nelze konflikty automaticky vyřešit, je o to uživatel požádán.

Rebase oproti merge manipuluje s historií. Přeskládá revize aktuální větve (B_1) jako by se od dané větve (B_2) oddělovaly až na konci B_2.

Diff je nástroj pro vytvoření rozdílů v souborech, nebo celých verzích.

Log ukazuje strukturu historie.

Fetch stáhne historii ze vzdáleného repozitáře.

Pull provede fetch a následně merge.

Push naopak začlení změny do vzdáleného repozitáře.

2.1 Základní postupy práce v Gitu

Git poskytuje velkou volnost ve způsobu správy větví a to jak lokálně [6] , tak v práci se vzdálenými repozitáři [7].

2.1.1 Postupy větvení

Dlouhodobé větve Při práci tímto způsobem obvykle repozitář obsahuje větve tří úrovní. První úroveň tvoří hlavní větev (často s názvem *master*, nebo *main*), která obsahuje pouze dobře otestované revize připravené k publikaci.

Vedle toho bývá k dispozici větev s názvem jako je *next*, nebo *develop*, která obsahuje méně stabilní kód, který není ucelený, nebo teprve čeká na otestování. Z této větve jsou revize podle potřeby slučovány do hlavní větve.

Poslední úroveň tvoří větve pro jednotlivé funkcionality. Tyto větve slouží k oddělení funkcionalit, které jsou v současné době ve vývoji. Z nich je práce po dokončení slučována do *develop* větve.

Krátkodobé větve Tento způsob práce (někdy nazývaný *topic branch*) není tak striktní ve způsobu slučování větví. Dovoluje slučování starších větví do novějších i naopak a tím vzniká tendence udržovat větve krátkodobější a s menším počtem revizí.

Dá se říct, že se jedná o odlehčenou verzi metody dlouhodobých větví, která obsahuje pouze nejnižší úroveň stability. Pro tyto vlastnosti je postup vhodný spíše u menších projektů.

2.1.2 Distribuovaná práce s větvemi

Centralizovaný postup Tento postup je hojně využíván hlavně pro svoji jednoduchost. Také je vhodný při přechodu z centralizovaného SSV pro jejich podobnost. Centralizovaný postup je vhodný pro týmy, ve kterých nehraje velkou roli hierarchie vývojářů, popřípadě nejsou příliš početné.

Prostředkem pro sdílení projektu je jediný vzdálený repozitář, ze kterého/který přispěvatelé aktualizují. Uživatelé, kteří chtějí na projektu pracovat, provedou operaci merge, či clone. Pokud naopak chtějí svoji práci sdílet do vzdáleného repozitáře, provedou push. V případě, že jiný uživatel mezitím sdílel svoji práci, nemá daný uživatel aktuální verzi a musí nejprve včlenit historii z centrální databáze do své, poté až provést push. Přitom samozřejmě může nastat, sice nepravděpodobná, ale stále možná situace, kdy, než uživatel stihl včlenit změny a provést push, opět někdo aktualizoval centrální databázi. Proces je potom třeba opakovat.

Postup s integračním manažerem Tento postup lépe využívá možnosti DSSV a vyžaduje jeden centrální vzdálený repozitář a dále jeden vzdálený repozitář pro každého běžného přispěvatele.

K centrálnímu repozitáři mají opět přístup všichni, ale právo zápisu má jen správce (integrační manažer). Ostatní smí zapisovat pouze do soukromých vzdálených repozitářů.

Pokud chce uživatel aktualizovat svůj repozitář, jednoduše provede pull, či clone, jako u předchozího postupu. V případě sdílení je ale situace odlišná. Uživatel odešle data do svého vzdáleného repozitáře a uvědomí o změnách správce. Ten včlení změny uživatelova vzdáleného repozitáře do svého lokálního repozitáře a poté je sdílí do centrálního vzdáleného repozitáře. Tam k datům mají opět přístup všichni ostatní.

Postup s diktátorem a poručíky Předchozí postup lze ještě vylepšit přidáním více správců a rozdělením jejich rolí do heirarchie: jeden diktátor a ostatní poručíci. Tento postup najde uplatnění spíše u projektů extrémních rozměrů, jako například vývoj Linuxového jádra [8].

Běžní vývojáři svojí práci včleňují na vrchol diktátorovi větve *master* pomocí příkazu rebase. Poručíci potom včlení větve vývojářů do svých větví *master*. Diktátor včlení větve poručíků do své větve *master* a zpřístupní ji do centrálního repozitáře ostatním.

3 Uživatelská část

Většina návodů pro práci s Gitem obsahuje ve velké míře grafovou reprezentaci pro lepší pochopení. Vytvořený program **GitGUI** vnáší tento prvek přímo do práce s ním. Má za cíl zobrazovat stav repozitáře Gitu vizuálně pomocí grafu a

poskytovat přehlednou práci s ním. Cílí tak na začátečníky, kteří se ještě neorientují s příkazovou řádkou, ale i na uživatele, kteří se potřebují rychle zorientovat ve struktuře revizí a větví.

Projekt je však vhodný spíše pro menší repozitáře. Jednak se přehlednost zvoleného zobrazení se zvětšujícím počtem revizí snižuje a jednak se prodlužuje čas potřebný k vykreslení a překreslení grafu.

Nutno podotknout, že uživatelské rozhraní GitGUI je v anglickém jazyce. Angličtina byla zvolena ze dvou důvodů. Většina uživatelů je již zvyklá na použité anglické výrazy a jiný jazyk by je mohl spíše mást. Dalším důvodem jsou špatně přeložitelné výrazy, například *stash* (skrýt), nebo i commit. Protějšky v českém jazyce dostatečně nevystihují danou operaci, nebo by mohly být zavádějící.

3.1 Reprezentace repozitáře grafem

Charakter vztahů *rodič – potomek* u jednotlivých revizí je ideální pro znázornění pomocí acyklického orientovaného grafu. Jednotlivé uzly mohou reprezentovat revize a jednotlivé hrany zase zmiňované vztahy *rodič – potomek* mezi revizemi.

Přitom acykličnost takového grafu je zřejmá, protože v Gitu nově vytvořená revize nemůže mít přidělené potomky a žádnou už vytvořenou revizi r nelze připojit jako rodiče jiného (obecně ne přímého) rodiče revize r.

3.2 Výběr způsobu rozložení grafu

Původní volba byla vzestupné rovinné nakreslení grafu. Pro popis toho, co je to rovinné nakreslení grafu je třeba nejprve vysvětlit rovinné nakreslení grafu.

Před definicí nakreslení grafu je však ještě třeba objasnit pojem *oblouk*.

Definice 1

Mějme libovolné prosté spojité zobrazení $\alpha\colon \langle 0,1\rangle \to \mathbb{R}^2$ intervalu $\langle 0,1\rangle$ do roviny. Potom podmnožinu roviny $a = \{\alpha(x) \mid x \in \langle 0,1\rangle\}$ nazveme obloukem.

Definice 2

Nakreslením grafu $G = (V, E)$ se rozumí prosté zobrazení d, které každému vrcholu $v \in V$ grafu přiřazuje bod $b(v)$ roviny, a každé hraně $e = \{u, v\}$ přiřazuje oblouk $a(e)$ v rovině s koncovými body b_u a b_v. Přitom žádný z bodů b_v $(v \in V)$ není nekoncovým bodem žádného z oblouků $a(e)$ $(e \in E)$.

Teď, když jsme formálně představili pojem nakreslení grafu, je možné zadefinovat rovinné nakreslení grafu Nakreslení grafu $G = (V, E)$, v němž oblouky odpovídající různým hranám mají společné nanejvýš koncové body, se nazývá rovinné nakreslení.

Pro acyklické orientované grafy má navíc smysl uvažovat pojem vzestupné rovinné nakreslení grafu. Rovinné nakreslení grafu $G = (V, E)$ nazveme vzestupné,

platí-li $\forall e = (a, b) \in E,\ a[a_x, a_y],\ b[b_x, b_y] : b_y > a_y$. Podobně lze samozřejmě graf kreslit vertikálně, či v libovolném dalším směru.

Takováto reprezentace by byla ideální kvůli přehlednosti, žádné hrany by se nekřížily a podmínka o postupném rozložení v jednom směru je vhodná pro znázornění, jak byly vrcholy reprezentující revize v čase postupně vytvořeny.

Ne každý acyklický orientovaný graf však má nějaké vzestupné rovinné nakreslení. Grafová reprezentace gitového repozitáře má kromě acykličnosti sice některé další omezující podmínky, existence vzestupného rovinného nakreslení však nelze zaručit.

Jednoduchý příklad takové struktury je vidět na obrázku 3.

Strukturu lze zreplikovat pomocí posloupnosti příkazů zdrojového kódu 1 (vynechány jsou příkazy *add* pro přidání změn do indexu). Z obrázku je zřejmé, že uzly 5, 6 a 7 nelze rozmístit tak, aby se hrany nekřížily.

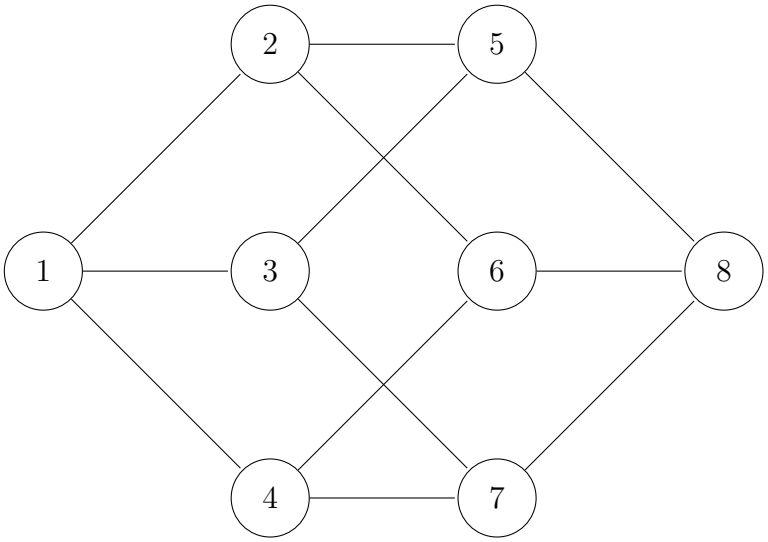
```
1  git branch a
2  git branch b
3  git branch c
4  git checkout a
5  git commit
6  git checkout b
7  git commit
8  git checkout c
9  git commit
10 git checkout a
11 git branch d
12 git merge b
13 git checkout c
14 git branch e
15 git merge b
16 git checkout d
17 git merge e
18 git merge a c
```

Zdrojový kód 1: Vytvoření struktury, která nejde vzestupně rozložit v rovině.

Z předcházející úvahy je vidět, že při stávajících požadavcích (uzly kreslené chronologicky jedním směrem) se nelze vyhnout překřížení hran.

Dalším způsobem, který by sice výše uvedenou podmínku nesplňoval, ale zlepšoval by přehlednost jiným způsobem, je rozdělení revizí do řádků podle větví, kterým patří (tedy ve kterých byly vytvořeny). I zde však narážíme na překážku, a tou je způsob ukládání revizí v gitu. Jak už bylo dříve popsáno, větve jsou v Gitu pouze ukazatele na poslední revize oné větve, není tedy možné dopátrat větev vytvoření.

Poslední možností je tedy způsob, který rozdělí uzly revizí chronologicky jedním směrem (konkrétně doprava). Navíc jsou rozděleny do řádků, které mohou odpovídat větvím vytvoření, není to však zaručeno. Popis algoritmu je popsán v podkapitole 4.2.

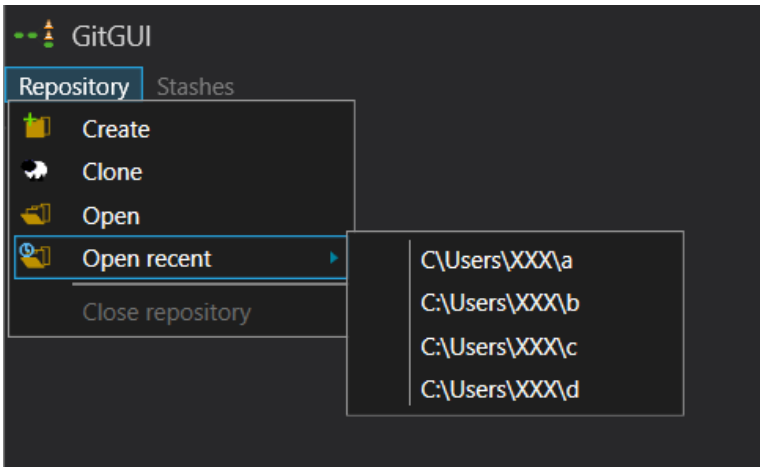


Obrázek 3: Graf, který nelze vzestupně nakreslit.

3.3 Vytvoření a otevření repozitáře

V příkazové řádce se repozitář otevírá jednoduše pomocí otevření adresáře, který je pod správou verzí Gitu. Pokud se na zmiňované cestě ještě repozitář nenachází, je možné jej vytvořit zadáním `git init`.

V programu k tomu slouží záložka *Repository* v menu horní lišty. Jak je vidět na obrázku 4, záložka umožňuje dva způsoby otevření (*Open* a *Open recent*) a položku pro vytvoření nového repozitáře (*Create*).



Obrázek 4: Menu - Repozitář

Po výběru možnosti *Open* se zobrazí klasický dialog prohlížeče souborů, ve kterém je třeba najít požadovaný adresář s repozitářem. Po jejím úspěšném výběru se repozitář otevře a vykreslí graf. V případě, že se jedná o takzvaný bare repozitář (ten neobsahuje pracovní strom souborů a tudíž s ním nejde pracovat

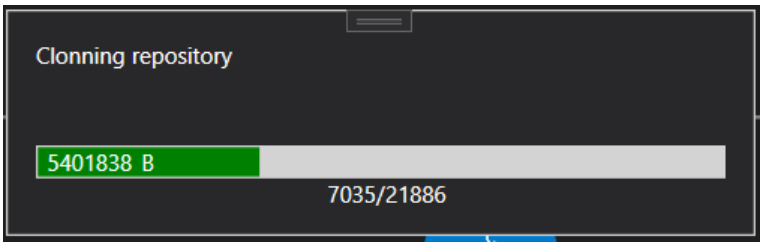
přímo a nemá tak smysl, aby ho program podporoval), je o tom uživatel zpraven chybovou hláškou: *Can't open bare repository*. Pokud se ve zvoleném místě nenachází žádný repozitář, pak je uživatel dotázán, zda-li si přeje vytvořit nový.

Druhou možností je potom *Open Recent*, která obsahuje další rozbalovací menu poskytující výběr nanejvýš pěti nedávno uzavřených repozitářů. Tento výběr je seřazen od nejpozději uzavřeného. Přitom za uzavření repozitáře se bere výběr možnosti *Close* v záložce *Repository*, otevření jiného repozitáře, či zavření aplikace. Pokud se vybraná položka již nenachází ve svém původním umístění, dostane uživatel prostřednictvím dialogového okna na výběr, jestli chce smazat odkaz.

Použití *Create* je analogické k *Open* s tím rozdílem, že pokud už je ve zvoleném umístění existující repozitář, je uživatel informován hláškou *There is already a repository*. Po úspěšném vytvoření se repozitář automaticky otevře.

3.3.1 Klonování repozitáře

Dalším způsobem, jak lokálně vytvořit repozitář, je jeho naklonování. Funkce je přístupná z nabídky *Repository* pod názvem *Clone*. Nejprve se program dotáže na požadované umístění, přitom se chová jako *Create*, potom se dotáže na url vzdáleného repozitáře. Jak je vidět na obrázku 5, během stahování je zobrazeno okno s průběhem klonování.



Obrázek 5: Dialog postupu klonování

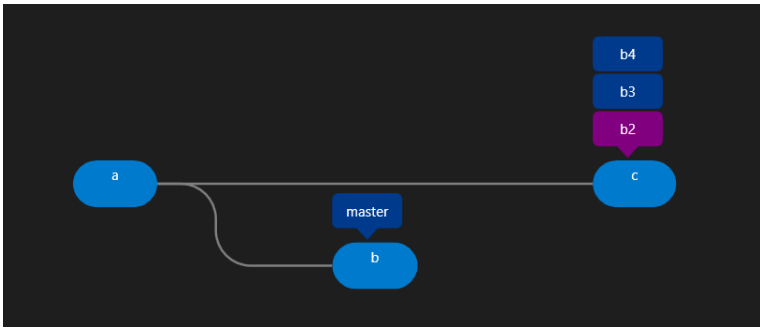
3.4 Práce s grafem a git log

Jednoduchý graf je vyobrazen na obrázku 6.

Skládá se ze světle modrých bublin reprezentujících uzly revizí a tmavších bublin ve tvaru obdélníku reprezentujících větve. Přitom je mezi revizemi naznačen vztah rodičů a potomků hranami, které jsou rovné v případě uzlů na stejném řádku a zaoblené jinak.

Tvar hrany pro uzly na různých řádcích je třeba více přiblížit, jelikož je rovněž potřebný pro výběr algoritmu. Tvar hrany se rozděluje na tyto dva případy:

Rodič je na vyšším řádku, než potomek: Potom se křivka těsně za rodičem lomí obloukem směrem nahoru a před horním řádkem se rovněž obloukem lomí zpět a pokračuje horizontálně.



Obrázek 6: Příklad grafu

Rodič je na nižším řádku, než potomek: Křivka se stáčí stejnýmm způsobem jako v předchozím případě, jen se stáčí až před potomkem (a samozřejmě směrem nahoru).

Jinými slovy větší část křivky se nachází na nižším z daných řádků.

Protože na revizi může v jednu chvíli ukazovat libovolný počet větví, je třeba jejich uzly skládat nad sebe, viz obrázek6, větve b2–b4. S tím také souvisí poloha tlačítka pro vytvoření nové větve. Při najetí myší nad uzel revize, či větve, jeho okraj se zvýrazní a na horní straně se objeví tlačítko se symbolem „+“ znázorňující možnost vytvoření nové větve ukazující na danou verzi (případně na stejnou verzi jako daná větev). Tlačítko se však zobrazí pouze nad horním uzlem větve z posloupnosti větví dané revize, případně na revizi pokud není koncem žádné z větví.

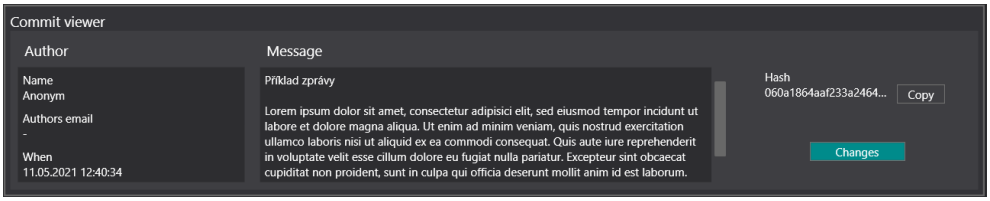
Aktuální hlava (tedy větev, či přímo revize) je potom barevně odlišena fialovou barvou.

Po výběru myší, nebo později popsaným nástrojem pro vyhledávání, se danému uzlu zvýrazní okraj a ve spodní části obrazovky se objeví panel s informacemi. Obě možnosti vzhledu panelu v závislosti na typu objektu jsou na obrázku 7. Jsou k dispozici informace jako zpráva, název, autor, čas vytvoření, hash (a tlačítko pro zkopírování), nebo tlačítko pro zobrazení změn. Panel se vypne po kliknutí do prázdného prostoru grafu, nebo po překreslení grafu.

Každý uzel větve navíc může obsahovat ikony indikující, že je větev vzdálená, nebo že má připojenou některou vzdálenou větev. Panel s informacemi potom ukazuje o kolik verzí je daný uzel před/za sledovanou vzdálenou větví.

V grafu se lze pohybovat pomocí myši po stisknutí a podržení pravého tlačítka v oblasti grafu. Znázorňuje to i kurzor myši pro pohyb ve všech směrech. Celý graf lze pomocí kolečka myši nebo touchepadu zvětšovat a zmenšovat. Hranice pro posun se nachází několik centimetrů za nejkrajnějším objektem grafu.

V krajním případě lze na sebe nahromadit velké množství větví. Tento příklad v praxi nenastává, ale je možný. V takovém případě je výsledek nejen vizuálně ne příliš hezký, ale také je potom možné graf po celé délce posunovat zbytečně vysoko a navíc se kvůli velkému počtu grafických prvků na jednom místě může zhoršovat plynulost posunování v grafu. Do budoucna je proto počítáno s úspornějším řešením. Například zobrazit pouze jeden uzel a tlačítko pro výčet ostatních.

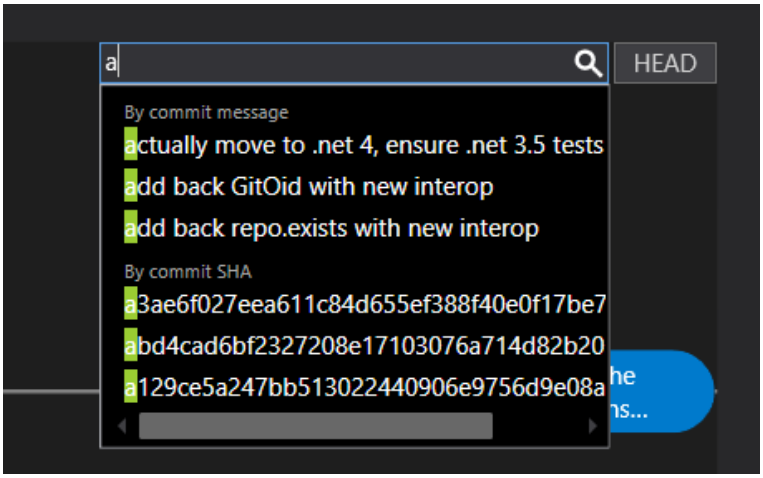


(a) Verze pro revizi

(b) Verze pro větev

Obrázek 7: Panel s informacemi o zvoleném objektu grafu

Koncept příkazu `git log` je v programu GitGUI nahrazen samotným grafem, který zrcadlí historii. Ve větších projektech ale může být obtížné vyhledávat konkrétní revize či větve pouze procházením grafu. Proto je v pravém horním rohu plochy pro zobrazení grafu přístupný nástroj pro vyhledávání (ukázka a obrázku 8 je provedena na projektu [9])



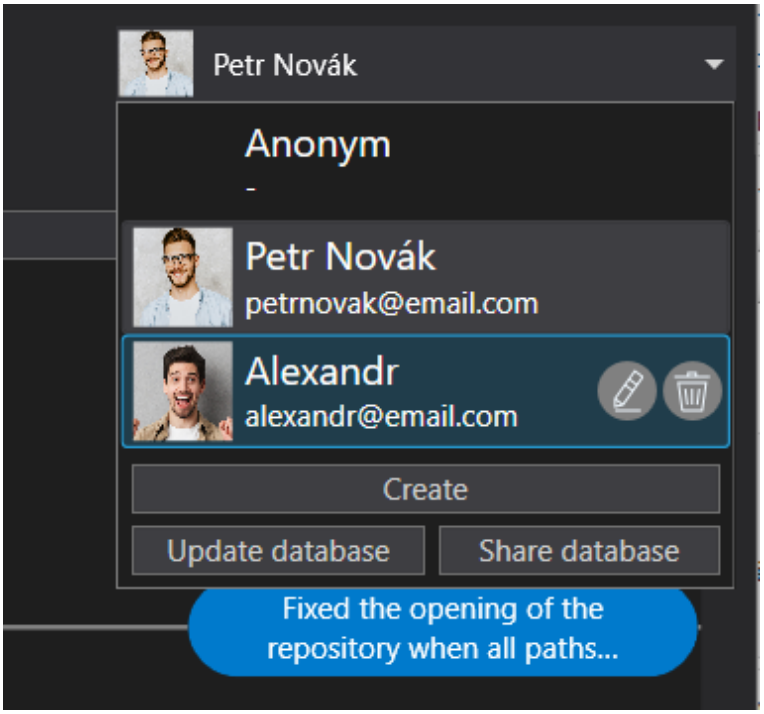
Obrázek 8: Ukázka vyhledávání

Nástroj obsahuje pole pro vložení hledaného výrazu a vyskakovací našeptávač. Kromě toho také obsahuje tlačítko speciálně pro nalezení hlavy repozitáře. Našeptávač se zobrazí v případě, že byl upraven vyhledávací výraz, jeho délka je nenulová a některé položky grafu vyhledávání vyhovují. Přejít na hledanou položku v grafu je možné pouze výběrem (myší, nebo klávesnicí) v našeptávači. Tím se otevře panel s informacemi o položce a graf se posune hledanou položkou do zorného pole. Našeptávač je rozdělen do tří kategorií. Každá z těchto kate-

gorií pak obsahuje nanejvýš tři návrhy, popřípadě není zobrazena neexistuje-li pro hledaný výraz žádný návrh. Kategorie jsou popořadě: Revize podle prvního řádku zprávy, větve podle názvu a revize podle jejich hash kódů. Nutno dodat, že se vyhledávané hodnoty porovnávají pouze od začátku.

3.5 Uživatelé

Současně s verzemi se v Gitu ukládají i informace o autorovi, konkrétně jméno a e-mailová adresa. V programu GitGUI se automaticky použije výchozí uživatel se jménem „Anonym“ a adresou „-“. Tento výchozí uživatel nelze smazat ani upravit. Je však možné dle libosti mazat, upravovat a vytvářet další uživatele. Slouží k tomu rozbalovací nabídka vyobrazená na obrázku 9.



Obrázek 9: Nabídka uživatelů

Jak je na obrázku vidět, nabídka se skládá ze seznamu uživatelů (který po překročení určité velikosti stane rolovatelný) a tlačítek pro vytvoření nového uživatele a sdílení nebo aktualizaci databáze uživatelů.

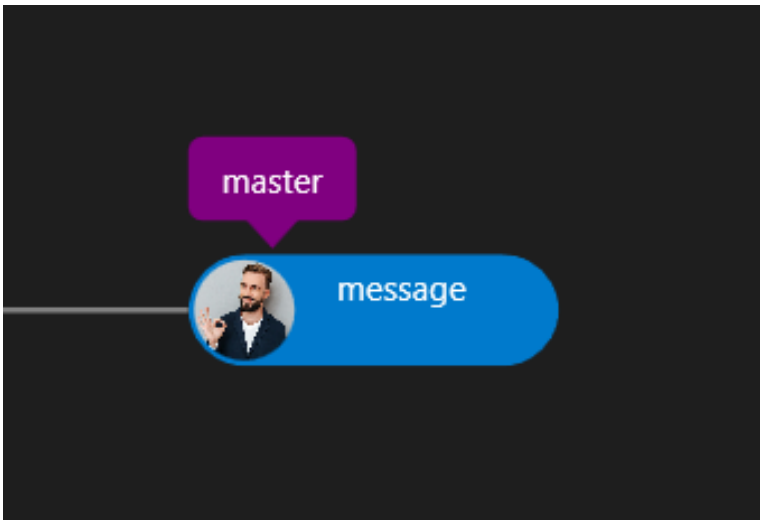
Každý uživatel může kromě, z Gitu známého, jména a adresy obsahovat ještě profilový obrázek. Výběrem tlačítka na pravé straně nevýchozího uživatele lze uživatele upravit, nebo smazat. Uživatel je vytvářen/upravován v samostatném okně. Výběr obrázku je nepovinný, ale jméno a adresa musí být validní. Pokud nejsou zadané hodnoty přípustné, u daného řádku je symbol červeného křížku a tlačítko pro potvrzení není povoleno, v opačném případě se objeví symbol zelené fajfky.

Požadavky pro hodnoty jsou:

Pro jméno: neprázdný řetězec obsahující alespoň jeden znak, který není bílý.

Pro adresu: správný tvar e-mailové adresy.

Všechny revize, které mají autora, jenž je známý, a má přidělený profilový obrázek, mají jeho miniaturu umístěnou v levé části uzlu. Detail je zachycený na obrázku 10.



Obrázek 10: Revize s obrázkem autora

Databázi uživatelů lze také sdílet. Po zvolení této možnosti se program dotáže na umístění, do kterého zkopíruje adresář s daty o uživatelích. Opačně lze z takto sdíleného adresáře aktualizovat databázi programu.

3.6 Vytváření revizí

Stisknutím tlačítka *Commit* se otevře nová karta. Je-li již karta pro vytvoření nové revize otevřena, nevytvoří se nová, ale přepne se na stávající.

Karta se dělí na tři hlavní části: adresář změn, okno pro detail změny a oblast pro specifikaci zprávy revize.

3.6.1 Adresář změn

Nachází se v levé části karty. Adresář tvoří stromovou strukturu. V listových uzlech stromu se nachází soubory, které byly změněny. V nelistových pak rodičovské adresáře až ke kořenové cestě, odpovídající uzel se nazývá *All*.

Každý listový uzel lze vybrat a tím zobrazit v pravé části karty detail změny. Většinou se jedná o jednoduchou textovou informaci, jako například: „Nový soubor“, „Smazaný soubor“, „Přejmenovaný soubor“, „změna je binární“. V případě textové změny je však zobrazen gitový „diff“, neboli rozdíl současné verze od předchozí.

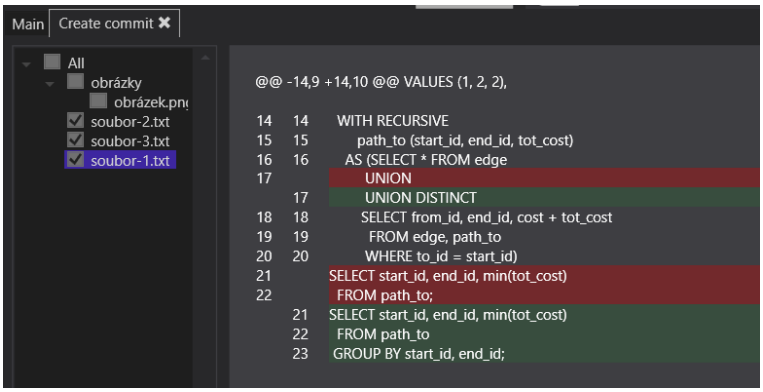
Každý uzel stromu také obsahuje zaškrtávací políčko. Zaškrtnuté změny se projeví v revizi, stejně jako kdyby se v příkazovém řádku přidaly do indexu pomocí příkazu `git add`. Ve výchozím stavu jsou všechny změny aktivní, to platí jak pro nově vytvořenou kartu, tak po jakékoliv změně v repozitáři.

3.6.2 Detail změny

Jak již bylo uvedeno, jediným zajímavým obsahem této oblasti je zobrazení souborového rozdílu. Postupně jsou pod sebou vypsané části (*hunk*) kódu obsahující změnu. Ty mají stejnou hlavičku, jako ve výsledku volání `git diff`. Obsah změny je ale formátován přehledněji a je barevně odlišená stará verze, nová verze a nezměněná část.

3.6.3 Zpráva

Poslední částí je pole pro zprávu revize s tlačítkem potvrzení. Tlačítko je aktivní pouze když je vybrána alespoň jedna změna a text zprávy je neprázdný.



Obrázek 11: Výřez karty vytvoření revize

3.7 Práce s větvemi

Operace `git checkout` lze v GitGUI provést výběrem požadované větve, nebo revize v grafu a stisknutím tlačítka *Checkout*, to je aktivní právě když je některý uzel vybrán.

Kromě vytváření nových větví je potřeba je slučovat. Tato operace se provádí přímo v grafu stylem přetáhnutí uzlu jedné větve na druhou. Tím se vyvolá kontextová nabídka s výběrem požadované operace, a to buď Merge, anebo Rebase.

Při přetahování uzlu může nastat situace, kdy se slučovaná větev nenachází v aktuálním zorném poli. Jedna vlastnost, která toto řeší je možnost oddalovat a přibližovat graf během přesunování. Tou druhou je automatické odsouvání plátna grafu po najetí dostatečně blízko k okraji plátna. Směr odsouvání je od středu ke kurzoru myši.

Ani tyto ovládací mechanismy však výrazně nepomohou v případě velkých repozitářů. Do budoucna by bylo dobré přidat druhotné způsoby provedení Merge/Rebase operace.

Standardní postup při slučování v Gitu je přejít do slučující větve (`git checkout <slucujici vetev>`) a provést merge se slučovanou větví (`git merge <slucovana vetev>`). GitGUI se snaží uživatele odstínit od nutnosti nejprve do větve přejít, a to tak, že akci provede za uživatele a to i v případě, že jsou ve stávající větvi neuložené změny. To je zařízeno pomocí implicitního provádění *stash* (viz dále). Vše samozřejmě nemůže být vždy zcela transparentní, jelikož během slučování může dojít ke konfliktu. V takovém případě se o konflikt musí postarat uživatel a po jeho vyřešení se vrátit do původní větve manuálně.

3.7.1 Konflikty

V konfliktním stavu nelze vytvářet nové revize, provádět stash ani slučovat větve. odpovídající tlačítka jsou proto neaktivní. Také se v tomto stavu otevře prohlížeč konfliktu podobný kartě vytvoření revize. Adresář změn obsahuje rozbalovací nabídku, která slouží pro volbu větve, se kterou se mají provádět rozdíly souborů.

Po manuálním vyřešení konfliktů je možné změny potvrdit, nebo sloučení zrušit. Všechny soubory konfliktů jsou automaticky zašrtnuté, protože bez nich nemá dokončení operace smysl.

Pokud se uživatel rozhodne v konfliktním stavu prohlížeč konfliktu vypnout, je možné jej kdykoliv opět otevřít pomocí tlačítka commit v hlavní kartě. Také se sám otevře po každé změně v repozitáři.

3.8 Prohlížeč revize

Prohlížeč změn v revizi je karta podobná kartě pro vytvoření revize. Obsahuje stejný adresář, ve kterém jde prohlížet změny v jednotlivých souborech a oblast s výpisem zprávy.

3.9 Stash

Stash lze vytvořit pomocí tlačítka *Stash* v hlavní kartě. Pokud jsou nějaké stashe dostupné pro aktuální repozitář, zdostupní se záložka *Stashing* v horní nabídce. Zde je dostupný seznam vytvořených stashů, které lze pomocí barevně odlišených tlačítek aplikovat nebo/a odstraňovat.

3.10 Implicitní stash

Zajímavým prvkem GitGUI je implicitní stash. Vytvoří se automaticky při pokusu o checkout jiné větve (v případě, že jsou provedeny změny, které lze uložit jako stash). Tato vlastnost by měla pomoct zjednodušit práci s Gitem. Je vhodná například pokud se chceme v průběhu práce na revizi podívat na zdrojový kód v jiné revizi, ale nechceme ještě ukládat změny jako samostatnou verzi.

Uložený stash není k dispozici v nabídce stashing, ale je pro uživatele viditelný jako ikona kužele na uzlu revize, od které byly změny provedeny. Vnitřně implicitní stash uložen jako běžný stash a je proto dostupný z příkazové řádky.

Po checkoutu větve ukazující na revizi „obsahující“ implicitní stash se automaticky aplikuje.

3.11 Vzdálené repozitáře

Prostřední panel v horní části okna slouží k práci se vzdálenými repozitáři. Jedná se o rozbalovací nabídku (podobnou prvku pro výběr a správu uživatelů) společně s tlačítky rozdělenými do dvou skupin. Skupina po levé straně nabídky obsahuje pouze tlačítko pro začlenění změn do vzdáleného repozitáře. po pravé straně tlačítka pro stažení a stažení s včleněním.

Operace vyvolané tlačítky *Push* a *Pull* používají jako argumenty vybraný vzdálený repozitář a hlavu lokálího repozitáře. Stažení (*Fetch*) místo hlavy použije větve, které sledují některou vzdálenou větev daného vzdáleného repozitáře.

Nastavení vzdáleného repozitáře umožňuje zadat přihlašovací údaje. Nejsou-li zadány, aplikace se na ně opakovaně zeptá při každém provedení operace, která je vyžaduje.

3.12 Další

Na závěr je dobré zmínit dvě jednoduché funkce, které mohou uživatelský zážitek zpříjemnit. Ve středu titulkového pruhu okna se nachází text s cestou k právě otevřenému repozitáři (je-li nějaký otevřen) a vpravo od tlačítek pro revizi, větvení a stash se nachází ikona, která otevře adresář aktuálního repozitáře v prohlížeči souborů.

4 Programátorská část

Tato kapitola popisuje použité technologie a zajímavé problémy, se kterými se bylo třeba v průběhu vývoje vypořádat.

4.1 Platforma .NET

Platforma .Net je open source a poskytuje programovací jazyky, editory a knihovny pro vývoj aplikací na mobilní zařízení, stolní počítače i web. Platforma používá jazyky C#, Visual Basic a F#. Prostředí je vyvíjeno firmou Microsoft.

Konkrétní implementace .NET se zaměřují na různé oblasti vývoje.

.NET Core Slouží pro tvorbu webových stránek a serverů a pro vývoj konzolových aplikací spustitelných v operačních systémech Windows, Linux a macOS.

.NET Framework Slouží obecně k vytváření aplikací a služeb pro OS Windows. Stejně tak podporuje vývoj webových stránek. Jedná se o původní a současně nejvíce rozšířenou implementací .NET.

Xamarin/Mono Tato implementace je vhodná k vývoji aplikací pro většinu mobilních zařízení (Android, iOS, macOS , ale i další).

.NET Standard Standard je souhrnná specifikace aplikačních rozhraní společná pro všechny implementace. Tyto služby tak lze využít pro vývoj libovolné aplikace v .NET.

4.1.1 .NET Framework

Framework umožňuje psát aplikace pro Windows v libovolném ze tří možných jazyků. Tento kód následně příslušný překladač přeloží do společného jazyka Common Intermediate Language (CLI) nezávislého na původním programovacím jazyce. Soubory s tímto kódem překladač rozdělí do souborů s příponami „.exe“ a „.dll“. Spuštění programu potom vyvolá JIT (Just in time) překladač, který kód přeloží za běhu do strojového kódu a ten spustí. GitGUI je vytvořen konkrétně ve verzi .NET Framework 4.6.1.

4.1.2 Windows Presentation Foundation (WPF)

Windows Presentaion Foundation je součást .NET umožňující snadné vytváření aplikací s grafickým uživatelským rozhraním pro Windows. Proto je také program GitGUI pomocí WPF vytvořen. Základy práce s WPF jsou popsány v knize [10].

WPF odděluje logiku aplikace od grafického aparátu. Logika je psána v C# (či jiném zvoleném jazyce) a uživatelské grafické prvky jsou popisovány pomocí dvou souborů, jeden v jazyce XAML a druhý v C#.

Jazyk XAML je deklarativní značkovací jazyk popisující stromovou strukturou vzhled daného prvku. Druhá část kódu se běžně označuje jako *code behind*, je psaná v jednom ze zvolených jazyků a slouží k implementaci chování (grafického prvku), které není možné zajistit pomocí značkovacího jazyka.

WPF disponuje velkým množstvím standardních prvků GUI, jako jsou tlačítka, nabídky, tabulky, seznamy, mřížky, obrázky, textové pole a vstupy a další. Ty jdou upravit dle požadavků uživatele pomocí velkého množství proměnných dostupných z XAML i kódu, případně lze kompletně měnit vzhled speciálními technikami jako jsou styly či šablony. Jde tak používat používat předem dané prvky a upravovat je podle svých představ i vytvářet nové.

Ačkoliv WPF zjednodušuje spoustu věcí, není jednoduché provést některé zdánlivě jednoduché úkoly. Příkladem je změna titulkového pruhu okna. I pouhé „přebarvení“ pruhu vyžaduje jeho vytvoření od začátku. Pruh okna je totiž v režii OS. Jedinou možností je tak odstranit rámeček okna a vytvořit jej znovu i s funkcionalitou tlačítek a přetahování okna. Tuto část kódu, jež je společná

pro všechny okna v programu kromě dialogových oken chybových hlášek a výběru souborů a složek, jsem převzal od Davida Rickarda. Reference je uvedená i v kódu, konkrétně v souboru WindowBase.cs.

4.1.3 NuGet

Jako prostředek pro rozšíření funkcionality obsahuje .NET správce balíčků NuGet. Ten umožňuje vytvářet balíčky, sdílet je a přidávat je do projektů.

GitGUI obsahuje některé z balíčků nabízených službou NuGet:

LibGit2Srarp Balíček LibGit2Sharp je nejdůležitějším a nejpoužívanějším balíčkem. Vzniká jako open source projekt na GitHubu [9]. Obstarává v programu velkou část funkcionality Gitu.

Ooki.Dialogs.WPF Tento balíček obsahuje třídu dialogu pro výběr složky. V běžných knihovnách .NET se tato funkcionalita nenachází v takové podobě, aby okno vypadalo stejně jako prohlížeč souborů.

Microsoft.Xaml.Behaviors.Wpf a System.Windows.Interactivity.Wpf Balíčky obsahují funkce, které zjednodušily tvorbu některých komponent v xaml.

4.2 Algoritmus rozmístění uzlů v grafu

Pro lepší přehlednost i zjednodušení algoritmu nejsou žádné dva uzly revizí nad sebou. Uzly revizí jsou rozmisťovány zleva v pořadí od nejdříve vytvořeného. To právě umožňuje neklást požadavky na místo zabrané uzly větví, nemůžou totiž překrývat jiné revize, protože se ve stejném sloupci nachází jen jedna. Dále tedy budou uvažovány jen revize.

Algoritmus jako vstup bere seřazenou posloupnost uzlů a snaží se je rozdělit na řádky tak, aby hrany, jejichž tvar byl již popsán, nebyly překryté žádným uzlem. Neboli každému uzlu přidělí správný řádek. Uzly přitom obsahují seznamy svých předchůdců i následníků. Poslednímu uzlu je přidělen řádek číslo nula a poté je ostatním uzlům od konce přidělováno číslo řádku podle incidujících hran.

Pro účely představení algoritmu bude použit pseudokód podobný skutečné implementaci v jazyce C#. Každý uzel obsahuje tyto vlastnosti:

Predecessors	Seznam předchůdců
Descendants	Seznam následníků
Row	Index řádku
DeployedPredecessors	Počet již umístěných předchůdců
HasPredecessorOnTheSameRow	Udává, jestli je na stejném řádku již umístěný některý předchůdce

Algoritmus se v první řadě snaží uzel umístit na řádek některého jeho následníka. Není-li to možné, hledá odshora volný řádek, přes který už žádná hrana nemůže vést.

```
1  posDesc = PossibleDescendantsOnTheSameRow(n)
2  sortedDesc = n.Descendants.Sort(d1, d2 → d1.Row < d2.Row)
3  if (Satisfies(posDesc, sortedDesc))
4   d = posDesc.Aggregate(n1, n2 → if (n1.Row < n2.Row) n1 else n2)
5   n.Row = d.Row
6   d.HasPredecessorOnTheSameRow = true
7  else
8   for(i = 0 to LastOnRow.Count)
9    if (HasSpace(i))
10    break;
11  n.Row = i
12 n.Descendants.ForEach(d → d.DeployedPredecessors++)
13 LastOnRow[i] = n
```

Zdrojový kód 2: Umístění uzlu n.

```
1  PossibleDescendantsOnTheSameRow(n)
2   posDesc = n.Where(d →
3    !d.HasPredecessorsOnTheSameRow && (
4    d.Predecessors.Count == 1 ||
5    d.Predecessors.Count - 1 == d.DeployedPredecessors))
6   complRows = Map(n.Descendants ∖ posDesc, d → d.Row)
7   return posDesc.Filter(p → p.Row ∉ complRows)
```

Zdrojový kód 3: Výběr možných následníků na stejném řádku.

```
1  HasSpace(i)
2   return LastOnRow[i].DeployedPredecessors ==  LastOnRow[i].
       Predecessors.Count &&  LastOnRow[i].HasPredecessorOnSameRow ==
       false
```

Zdrojový kód 4: Zjištění, zda je na řádku volné místo.

V prvním řádku kódu 2 jsou vybráni následníci, kteří nemají předchůdce na stejném řádku a počet předchůdců je jedna, nebo o jedna menší než počet rozmístěných uzlů. Z toho jsou odebraní následníci se stejným řádkem, jinak by došlo k překřížení.

Dále se algoritmus větví na dva případy. První možností je, že kolekce následníků *posDesc* je neprázdná a současně jsou buď seřazení následníci podle řádků i

```
1 Satisfies(p, s)
2   p ≠ ∅ ∧ (AreSortedByTime(s) ∨ p.Min(d → d.Row) = s.First.Row)
```

Zdrojový kód 5: Predikát kontrolující, jestli je možné použít řádek vrchího z posDesc.

chronologicky, nebo je nejvyšší možný řádek pro právě umisťovaný uzel totožný s řádkem nejvyššího následníka.

Při splnění těchto podmínek je uzlu přidělen horní z navrhovaných řádků a nastaví se patřičné proměnné potřebné k rozmístění ostatních uzlů. U časově seřazených následníků na pozici uzlu nezáleží (hrany zasahují do ostatních sloupců jen ve spodním z incidentních řádků). Pro neseřazené uzly je dostatečná podmínka, aby nebyl žádný následník výš

Nejsou-li podmínky splněny, hledá se první řádek, který zaručuje, že už nemůže obsahovat další horizontálně vedenou hranu křížící tento sloupec.

Nakonec se upraví kolekce posledních revizí na řádku a cyklus pokračuje pro případný další uzel.

4.3 Vykreslování grafu

Původní plán na zobrazování grafu spočíval ve výpočtu pozic uzlů popsaném v předešlé podkapitole a následném umístění odpovídajících grafických prvků. Prvky grafického rozhraní se vložily do položky schopné své vizuální potomky zobrazovat na plátně tak, že obsah, který přeteče, je skryt. Zbývající viditelná oblast je potom rolovatelná.

Problém s touto metodou je dvojí. Všechny prvky jsou od začátku vložené v plátně a všechny se neustále vykreslují. To však pro již vykreslený graf nepředstavuje výkonnostní potíže. Horší je to ale s rychlostí překreslování při otevření projektu, nebo zachycení změny.

Experimentálně se zjistilo, že proces výpočtu algoritmu rozmístění trvá pro větší projekty (kolem 200000 revizí) kolem minuty, kdežto samotné vytvoření odpovídajících UI elementů až desítky minut.

Bylo nutné použít jinou metodu. Algoritmus rozmístění nebylo třeba nijak upravovat, „bottleneck“ se skrývá ve vytváření objektů. Obejít to lze vytvářením pouze nutných (rezumějme viditelných) objektů. Při každé změně polohy nebo velikosti zorného pole plátna se nyní vytvoří nově viditelné prvky a zruší se nepotřebné. Tímto se výrazně zrychlilo počáteční vykreslení za cenu drobného zpomalení při změně zorného pole.

4.4 Sledování změn

Sledování změn v otevřeném repozitáři je z pohledu uživatelského pohodlí nevyhnutelné. V opačném případě by uživatel musel vědět, kdy a jestli změna

nastala a náhled pomocí nějakého ovládacího prvku obnovit. To je samo o sobě nepohodlné, navíc by toto řešení přinášelo problémy spojené s možností omylem pracovat se starými daty.

Dalším důvodem je určité propojení s ostatními způsoby práce s repozitářem. Není vyloučeno (a dokonce se s tím, že by tento pracovní postup bude běžný), že uživatel bude současně pracovat s GitGUI i příkazovou řádkou, či jiným nástrojem pro Git. GitGUI by se mohl využívat pro úkony jednoduššího charakteru nebo pro rychlé získávání přehledu a současně jiný nástroj pro vykonání složitějších příkazů. GitGUI tedy musí počítat s tím, že se může historie kdykoliv změnit.

Knihovna System.IO obsahuje třídu FileSystemWatcher. Objekty této třídy jsou schopné sledovat změny v nastaveném adresáři případně i zanořených adresářích. Každé přejmenování, vytvoření souboru, modifikace, nebo smazání souboru spustí na vedlejším vlákně vybrané zpětné volání.

Pro potřeby tohoto projektu není vhodné využívat jako události vyvolávající překreslení grafu přímo všechna zpětná volání. I jednoduchá změna jednoho souboru může vnitřně znamenat více jednoduchých operací (a ty jsou všechny objektem zachyceny). Navíc operace v Gitu můžou najednou vytvořit desítky, nebo stovky souborů (například operace `git add`).

Reagovat na tolik změn je nepřípustné, proto se vytvořila třída ChangesWatcher jako vyšší úroveň abstrakce nad třídou FileSystemWatcher.

ChangesWatcher Vnější rozhraní třídy tvoří selektor *IsActive* pro zjištění, jestli objekt sleduje nějaký adresář a funkce *Watch* a *End* společně s událostí *ChangeNoticed*. Objekt této třídy zachytává v adresáři a podadresářích zvolené cesty všechny typy změn za krátký časový úsek. Jakmile je to vhodné, podá o nich souhrnnou zprávu pomocí události.

V této třídě je důležitá explicitní práce s vlákny. Jakmile FileSystemWatcher zachytí nějakou změnu, spustí se na jeho vlákně funkce *InvokeChangeIfLastNotifyFromSequence* (ukázka v kódu 6). Je potřeba, aby bylo provedení této funkce rychlé, jinak by se mohly ve FileSystemWatcheru začít hromadit události a jeho kontejner přetéct.

Ve zmiňované funkci se nachází kritická sekce realizovaná semaforem. Uvnitř kritické sekce se postupuje následovně: Pokud se již zpracovává událost k poslání, nebo je dokonce prováděn kód, který událost vyvolává (v tomto případě překreslování grafu), pouze se nastaví sdílený příznak opakování *Repeat*. Pokud se nic nezpracovává a funkce je volána poprvé v případné sekvenci rychle za sebou následujících volání, aktivuje se časovač nastavený na jednu sekundu. Jinak (když se nic nezpracovává a časovač je aktivní) se časovač restartuje opět na jednu sekundu.

Když se tedy objeví sekvence změn s časovým rozestupem méně než jednu sekundu, vyčkává se a časovač se neustále resetuje. Jakmile přestanou chodit zprávy o změnách v souborovém systému, dojde časovač na hodnotu nula. Ve speciálním vlákně je tím spuštěna funkce OnTimedEvent. Ve funkci OnTimeE-

vent probíhá nekonečná smyčka, která se zastaví až když v průběhu zpracování ChangeNoticed nedojde k výskytu další změny.

```
1  void InvokeChangeIfLastNotifyFromSequence()
2  {
3     Mutex.WaitOne();
4     if (Processing)
5        Repeat = true;
6     else if (ChangesGroupTimer.Enabled)
7        RestartTimer();
8     else if (First)
9     {
10       First = false;
11       ChangesGroupTimer.Start();
12    }
13    Mutex.Release();
14 }
15
16 void OnTimedEvent(object sender, System.Timers.ElapsedEventArgs e)
17 {
18    while (true)
19    {
20       Mutex.WaitOne();
21       Repeat = false;
22       Processing = true;
23       Mutex.Release();
24       Application.Current.Dispatcher.Invoke((Action)(InvokeChange));
25       Mutex.WaitOne();
26       if (Repeat)
27       {
28          Mutex.Release();
29          continue;
30       }
31       Processing = false;
32       First = true;
33       Mutex.Release();
34       return;
35    }
36 }
```

Zdrojový kód 6: Sledování poslední změny v sekvenci

Bylo by dobré upozornit na některé části kódu 6. Nutnost kritické sekce na řádcích 20–23 je zřejmá, 25–28 / 25–33 však už nemusí být. Kdybychom čistě zkontrolovali podmínku a na jejím základě buď pokračovali další iterací, nebo skončili, mohlo by dojít k následující situaci. Podmínka *Repeat* by se vyhodnotila, protože ale nejsme uvnitř kritické sekce, může nyní pokračovat funkce *InvokeChangeIfLastNotifyFromSequence*. Příznak *Processing* v tuto chvíli musí být pravdivý, proto se vykoná instrukce na řádku číslo pět. Ve funkci *OnTimeEvent* by se nyní provedly další příkazy. Podmínka již byla vyhodnocena negativně,

proto se pouze obnoví ostatní příznaky na původní hodnotu a funkce skončí. Informace, že došlo ke změně, by tím byla ztracena.

Za zmínku stojí i řádek 24. Ten zajistí, že se na hlavním vlákně začne synchronně provádět překreslování grafu a další aktivity. Jakmile práce skončí, funkce pokračuje na řádku 25.

4.5 Ukládání dat

Některé oblasti vyžadují ukládání dat, proto je třeba vyřešit jejich umístění a formát.

Naposledy otevřené repozitáře Je nutné ukládat cesty k repozitářům, se kterými se pracovalo a časy jejich ukončení. Na základě těchto časů jsou záznamy spravovány a seřazovány.

Seznam známých uživatelů Mezi jednotlivými uživateli jde přepínat a nastavovat jejich obrázky. Je tedy třeba jméno uživatele, adresa uživatele a volitelně obrázek.

Vzdálenné repozitáře Vzdálené repozitáře jsou ukládány pro každý projekt pomocí knihovny LibGit2Sharp stejně jako to dělá Git. Je však třeba ukládat po dvojicích údaje o vzdáleném repozitáři a případné přihlašovací údaje, aby je uživatel nemusel opakovaně zadávat.

Implicitní stash Při implicitním vytvoření stash se vytvoří stash tak, aby byla dostupná jako normální stash, je ale třeba někam uložit jejich hash kód, aby se uvnitř aplikace daly rozpoznat.

4.5.1 Umístění a formát

Jak je vidět, všechna data jsou jednoduchá a tudíž nemá smysl pro ně vytvářet databázi. Pro takto jednoduchý úkol lze použít obyčejné uložení v souborovém systému. Data jsou uložena v adresáři AppData.

Závěr

Vytvořený program GitGUI slouží jako jednoduchý nástroj pro přehlednou práci s Gitem. Lze používat samostatně i jako doplněk současně s jiným nástrojem, například příkazovou řádkou.

Splnil jsem hlavní cíle práce, tedy grafický nástroj umožňující provedení základních operací Gitu převážně pomocí grafu. Do budoucna plánuji optimalizovat rychlost aplikace a rozšířit ji o další funkce.

Conclusions

The created GitGUI program serves as a simple tool for clear work with Git. It can be used separately or with another tool, such as the command line, at the same time.

I fulfilled the main goals of the work. Ie I created a graphical tool that allows one to perform basic Git operations mainly using a graph. In the future, I plan to expand functionality and optimize program speed.

A Obsah přiloženého CD/DVD

bin/

Spustitelná verze programu GitGUI jako exe soubor. Ostatní soubory adresáře *bin/* je nutné umístit vždy do stejného adresáře jako spustitelný exe soubor. Pro spuštění programu je třeba instalovat .NET Framework 4.6.1.

doc/

Bakalářská práce v pdf souboru a jeho zdrojové kódy pro typografický systém latex.

src/

Projekt pro Visual Studio obsahující veškeré zdrojové kódy a ostatní soubory potřebné pro kompilaci na výsledný program.

Literatura

[1] OTTE, Stefan. Version Control Systems. 2009.

[2] SCOTT, Chacon. *Getting Started - About Version Control*. 2014. Dostupný z: ⟨https://git-scm.com/book/en/v2/Getting-Started-About-Version-Control⟩.

[3] FOUNDATION, Apache Software. *Apache Subversion*. 2000 . Dostupný z: ⟨https://subversion.apache.org/⟩.

[4] MACKALL, Matt. *Mercurial*. 2021. Dostupný z: ⟨www.mercurial-scm.org/repo/hg-stable⟩.

[5] CHACON, Scott; STRAUB, Ben. *Pro git*. 2014.

[6] SCOTT, Chacon. *Git Branching - Branching Workflows*. 2014. Dostupný z: ⟨https://git-scm.com/book/en/v2/Git-Branching-Branching-Workflows⟩.

[7] SCOTT, Chacon. *Distributed Git - Distributed Workflows*. 2014. Dostupný z: ⟨https://git-scm.com/book/en/v2/Distributed-Git-Distributed-Workflows⟩.

[8] TORWALDS, Linus. *linux*. 2015.

[9] THE LIBGIT2SHARP CONTRIBUTORS. *libgit2sharp*. 2010 . Dostupný z: ⟨https://github.com/libgit2/libgit2sharp⟩.

[10] KHANG, Alex. *Professional WPF and C# Programming: Practical Software Development Using WPF and C#*. 2019.