Západočeská univerzita v Plzni

Fakulta aplikovaných věd
Katedra informatiky a výpočetní techniky

Diplomová práce

Proces verzování a modulární Git klient pro vývoj počítačových her

Plzeň 2023 Václav Jirák

ZÁPADOČESKÁ UNIVERZITA V PLZNI
Fakulta aplikovaných věd
Akademický rok: 2022/2023

ZADÁNÍ DIPLOMOVÉ PRÁCE

(projektu, uměleckého díla, uměleckého výkonu)

Jméno a příjmení:	**Bc. Václav JIRÁK**
Osobní číslo:	**A22N0125P**
Studijní program:	**N3902 Inženýrská informatika**
Studijní obor:	**Softwarové inženýrství**
Téma práce:	**Proces verzování a modulární Git klient pro vývoj počítačových her**
Zadávající katedra:	**Katedra informatiky a výpočetní techniky**

Zásady pro vypracování

1. Seznamte se s běžnými procesy verzování při vývoji software se zaměřením na specifika vývoje počítačových her.
2. Navrhněte zjednodušený proces verzování pro vývoj počítačových her.
3. Zvolte či navrhněte jazyk pro popis verzovacích procesů a realizujte v něm proces navržený v bodě 2.
4. Implementujte GUI klienta nástroje Git pro realizaci procesů popsaných v jazyce zvoleném v bodě 3.
5. Vytvořené řešení řádně otestujte a zhodnoťte jej z pohledu použitelnosti (usability).

Rozsah diplomové práce: **doporuč. 50 s. původního textu**
Rozsah grafických prací: **dle potřeby**
Forma zpracování diplomové práce: **tištěná/elektronická**

Seznam doporučené literatury:

dodá vedoucí diplomové práce

Vedoucí diplomové práce: **Ing. Jakub Daněk**
Katedra informatiky a výpočetní techniky

Datum zadání diplomové práce: **9. září 2022**
Termín odevzdání diplomové práce: **18. května 2023**

L.S.

Doc. Ing. Miloš Železný, Ph.D.
děkan

Doc. Ing. Přemysl Brada, MSc., Ph.D.
vedoucí katedry

V Plzni dne 11. října 2022

Prohlášení

Prohlašuji, že jsem diplomovou práci vypracoval samostatně a výhradně s použitím citovaných pramenů.

V Plzni dne 22. června 2023

Václav Jirák

Poděkování

Rád bych tímto poděkoval Ing. Jakubu Daňkovi za ochotu, cenné rady, trpělivost a čas, který mi věnoval při vypracování diplomové práce.

Abstract

In the first part of this thesis, the specifics of game development related to version control are identified. Based on these specifics and the analysis of common versioning processes, a simple versioning process for game development is then designed. Furthermore, a solution in the form of a new language for describing versioning processes and a Git client is designed and implemented, the goal of which is to simplify the use of Git and any versioning processes.

Abstrakt

První část této práce se týká zejména identifikace specifik vývoje her souvisejících s verzováním softwaru. Na základě těchto specifik a analýzy běžných procesů verzování je poté navrhnut jednoduchý proces verzování pro vývoj her. Dále je navrženo a implementováno řešení ve formě nového jazyka pro popis verzovacích procesů a Git klienta, jehož cílem je zjednodušení používání Gitu a používaných procesů verzování.

Obsah

1 Úvod

Videohry dnes patří k jednomu z nejběžnějších typů softwaru a zvolnění jeho růstu je stále v nedohlednu. Podle odhadů se očekává, že velikost trhu s videohrami v roce 2024 vzroste až na 218,7 miliard dolarů [19]. Je tedy více než vhodné zabývat se problémy souvisejících s jejich vývojem.

Herní průmysl má totiž svá specifika, která – pokud se s nimi adekvátně nevypořádá – mohou do procesu vývoje přinášet značné tření. Většina dnešních her je vyvíjena v rámci týmu, kde umělci s malými nebo žádnými zkušenostmi s vývojem softwaru vytváří pro hru umělecká díla a spolupracují se softwarovými vývojáři píšící kód. A se softwarovým vývojem je úzce spjato používání tzv. systémů správy verzí. Některé z těchto nástrojů — jako například Git – však byly vyvinuty primárně pro správu zdrojových kódů softwarovými vývojáři, což v případě vývoje her není optimální, zejména kvůli komplexitě Gitu, která může být pro umělce těžko uchopitelná.

Herní vývojáři se tedy často vydávají cestou používání jiného systému správy verzí, který je z hlediska daných specifik vhodnější, ale může s ním být spjato několik úskalí – jsou například placené nebo neposkytují možnosti, které Git efektivně poskytuje.

Jedním z hlavních cílů této práce je identifikovat všechna problematická specifika týkající se verzování ve vývoji her pomocí nástroje Git a následně navrhnout a implementovat řešení, které bude problémy způsobené danými specifiky limitovat.

Práci lze rozdělit na několik částí. První část se týká analýzy obecné problematiky verzování, která se následně zaměří na specifika vývoje videoher. Následuje část týkající se procesů verzování, v rámci níž budou nejprve představeny běžně používané prvky a následně bude navrhnut proces verzování pro vývoj videoher. V další části bude provedena analýza možností popisu procesů verzování, jejíž výsledkem bude buď zvolení existujícího jazyka, nebo navržení jazyka nového. Dále bude implementován Git klient pro realizaci procesů popsaných v jazyce vzešlého z předchozí části. Na závěr budou výstupy práce zhodnoceny a uvedeny možná vylepšení.

2 Verzování softwaru

2.1 Problematika verzování softwaru

Vývoj softwaru často bývá náročný proces v rámci něhož několikačlenný tým souběžně kolaboruje nad stejnou množinou zdrojových kódů a dalších souborů souvisejících s výsledným softwarem. Pokud nejsou průběžné změny souborů prováděných jednotlivými členy týmu určitým způsobem řízené, vývoj se může stát neudržitelným a vyústit až v chaos. A právě touto problematikou se mimo jiné zabývá verzování softwaru.

Verzování – v literatuře často zmiňováno pod anglickými názvy *version control*, *source control* nebo *revision control* – je proces sledování a spravování změn souborů [3]. Tyto soubory mohou být jakéhokoliv typu, ale nejčastěji je verzování spojováno právě s vývojem softwaru .

Aniž by si to lidé uvědomovali, tak mnoho jich již nejjednodušší formu verzování někdy použilo. Verzováním lze totiž nazvat i proces uchovávání různých kopií souboru s vhodným pojmenováním, jako například `logo-v1.jpg` a `logo-v2.jpg`. Metodu lze případně rozšířit i pro několikačlenný tým použitím například sdílené složky.

I když tato metoda v určitých případech může fungovat – příkladem mohou být krátké školní eseje – tak ve většině případů selhává a stává se neefektivní vzhledem k tomu, že je nutné udržovat mnoho téměř identických kopií, což vyžaduje velkou sebekázeň ze strany přispívajících a často vede k chybám. Navíc ve vývoji softwaru a i dalších prostředích je stále běžnější, že jeden dokument nebo i úryvek kódu současně upravuje několik členů týmu, jehož členové mohou být geograficky rozptýleni a mohou sledovat různé a v horším případě i opačné zájmy. Další úskalí této formy verzování může nastat v případě, kdy chceme zjistit, kdo je autorem dané změny a případně proč byla daná změna provedena. Aby se těmto nedostatkům předešlo, musí se zvolit sofistikovanější forma verzování.

Jednou ze sofistikovanějších metod verzování řešící některé zmíněné problémy je důkladné zaznamenávání změn pro každý soubor v tabulkovém procesoru s dodatečnými informacemi, jako například verze, autor, popis

změny a podobně. Příklad verzování touto metodou je nastíněn v tabulce 2.1 modelující změny souboru `HelloWorld.c` a finální verze souboru je poté znázorněna ve fragmentu kódu 2.1. Použitím této metody odpadá potřeba uchovávání celých kopií souborů a díky dostupným datům z tabulky se lze postupnými kroky vrátit k libovolné předchozí verzi souboru.

Tabulka 2.1: Příklad verzování souboru pomocí tabulkového procesoru

Verze	Autor	Datum	Popis	Řádky	Změna
1	Václav	1.1. 8:00	Vytvoření souboru, přidání std library	1	#include <stdio.h>

2	Václav	1.1. 8:12	Přidání main funkce	2-5	int main() {
					
					return 0;
					}

3	Václav	1.1. 8:15	Přidání výpisu	3	printf("Hello, Václav");
4	Karel	2.1. 8:00	Úprava výpisu	3	printf("Hello, Karel");

```
#include <stdio.h>
int main() {
   printf("Hello, Karel!");
   return 0;
}
```

Fragment kódu 2.1: Finální verze souboru z tabulky 2.1

I tato metoda samozřejmě není příliš použitelná vzhledem k tomu, že je od členů týmu vyžadována ještě větší sebekázeň při vyplňování potřebných dat do tabulky než v případě nejjednodušší metody verzování. Je zde ale zmíněná z toho důvodu, že obdobný princip je základem některých tzv. systémů správy verzí, které tuto aktivitu automatizují a poskytují mnoho dalších užitečných funkcí [3].

2.2 Systémy správy verzí

Systémy pro správu verzí (*Version control system, VCS*) jsou softwarové nástroje, které pomáhají zejména softwarovým týmům spravovat změny zdrojového kódu a dalších souborů [36]. Umožňují vývojářům spolupracovat, provádět změny a spravovat různé verze projektu. V důsledku toho je vývoj efektivnější, zkracuje se doba vývoje a zvyšuje se pravděpodobnost úspěchu projektu.

Systémy pro správu verzí jsou nejčastěji provozovány ve formě samostatně běžících aplikací, ale jsou i případy, kdy jsou přímou součástí různých typů softwaru. Příkladem mohou být textové a tabulkové procesory [13] a systémy pro správu obsahu, jako například historie stránek Wikipedie [31], kde slouží mimo jiné k obraně proti vandalismu. Tato práce se ale bude dále zaměřovat pouze na systémy pro správu verzí ve formě samostatně běžících aplikací. Konkrétně budou postupně představeny tři třídy systémů pro správu verzí – lokální, centralizované a distribuované – včetně některých jejich zástupců.

2.2.1 Lokální systémy správy verzí

Lokální systémy správy verzí jsou nejjednodušším typem systémů správy verzí. Příklad, který by se mohl zařadit to této kategorie, byl již nastíněn v kapitole 2.1 – verzování pomocí manuálního uchovávání několika verzí souborů s vhodným pojmenováním. V tomto případě se však nejedná o softwarový nástroj, ale spíše o manuální proces a jak již bylo řečeno, tento přístup je velmi běžný vzhledem k jeho jednoduchosti, ale je také velmi náchylný k chybám.

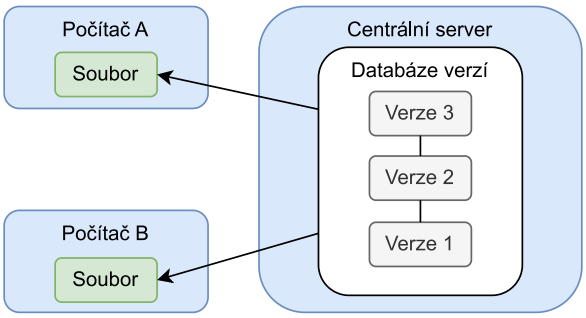
První lokální systém pro správu verzí (a systém pro správu verzí obecně) byl vytvořen roku 1972 v Bell Labs pod názvem *SCCS (Source Code Control System)* [26]. Tento systém sdílel mnoho vlastností s moderními systémy správy verzí, jako je schopnost vytvářet, upravovat a sledovat změny souborů. Nebylo v něm však možné, aby vícero uživatelů zároveň pracovalo nad jedním souborem.

O deset let později, roku 1982, byl vyvinut nový systém zvaný *RCS (Revision Control System)* [25]. RCS stále podporoval provádět úpravy vždy pouze jednomu uživateli a podporoval pouze práci na jednotlivých souborech, nikoli na celém projektu. Byl však průkopníkem nového způsobu sledování změn zvaným *reverse delta*, což je obdobný způsob verzování popsaném na konci kapitoly 2.1. Neukládaly se tedy již všechny verze souboru, ale namísto toho byla použita vždy poslední verze souboru jako základ, ze kterého se případně postupně vracelo k předchozím verzím.

2.2.2 Centralizované systémy správy verzí

Zejména kvůli narůstající potřebě spolupráce několikačlenného týmu na různých systémech vznikly tzv. centralizované systémy správy verzí.

V centralizovaném systému správy verzí jeden jediný server ukládá všechny verze a změny všech souborů [36]. Uživatelé mohou nad daným souborem provést tzv. *check out*, čímž je uživateli umožněno na daném souboru pracovat a provést změny, které poté lze zapsat na centrální server pomocí operace *commit*. Případně lze vytvořit zvláštní větev projektu, pracovat na ní samostatně a poté jí sloučit zpět do hlavní větve. Všechny změny jsou tedy synchronizovány s centrálním serverem. Po mnoho let byl tento způsob jediným standardem pro verzování. Princip centralizovaných systémů správy verzí je znázorněn na obrázku 2.1.



Obrázek 2.1: Diagram centralizovaných systémů správy verzí.

Kromě hlavní výhody, tedy podpory spolupráce několikačlenného týmu na různých systémech, poskytují centralizované systémy i několik dalších výhod, zejména v porovnání s distribuovanými systémy správy verzí, které jsou popsány v kapitole 2.2.3. První z nich je jejich jednoduchost vzhledem k tomu, že distribuovaná povaha systémů může být zejména pro netechnické profese hůře uchopitelná. Dále lze mezi výhody zařadit fakt, že každý člen týmu má vždy přístup k posledním verzím souborů napříč týmem, což výrazně snižuje možnost vzájemného přepisování práce, duplikování úsilí nebo plýtvání úsilím prováděním změn v souborech, u kterých platí, že nelze jednoduše nebo vůbec sloučit několik zároveň prováděných změn. Typickým příkladem takovýchto souborů jsou binární soubory.

V porovnání s distribuovanými systémy mají centralizované systémy samozřejmě i několik nevýhod. Tou nejzřetelnější je možná existence tzv. *single point of failure* ve formě centralizovaného serveru [36]. To znamená, že pokud je server na nějaký čas mimo provoz, pak během tohoto času nemůže nikdo spolupracovat nebo uložit provedené změny. Dalším důsledkem může být nevratná ztráta veškerých dat, kromě posledních verzí souborů, na kterých zrovna uživatelé pracují, pokud se pevný disk centrálního serveru poškodí a neexistuje žádná jeho záloha.

První centralizovaný systém byl vyvinut roku 1986 – *CVS (Concurrent Versions Systems)* [5]. Uživatelé mohli po dokončení změn nad souborem provolat příkaz `UpdateVersion` čímž se soubor na serveru aktualizoval na novou verzi. CVS, stejně jako RCS, neuchovával celé verze souborů, ale pouze změny od předchozí verze. Díky použití klient–server modelu a snadné podpoře tzv. větvení je CVS již modernějším příkladem systému správy verzí. Následovala i celá další řada centralizovaných systémů, jako *Subversion* nebo *Perforce*, které jsou používány dodnes, což mimo jiné vyplývá z provedeného průzkumu, který je podrobně popsán v kapitole 3.2.

2.2.3 Distribuované systémy správy verzí

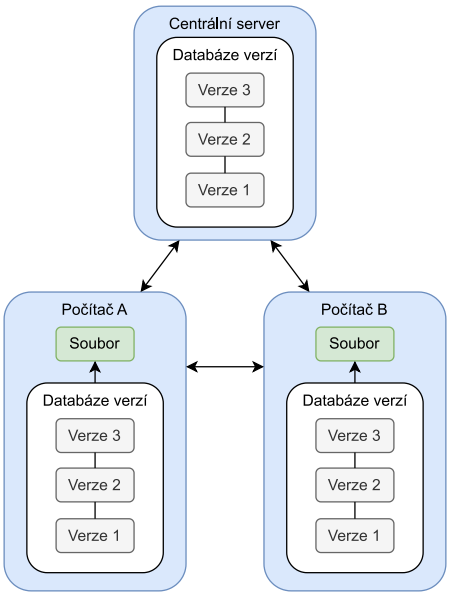
Posledním typem systémů správy verzí jsou tzv. distribuované systémy. Základním principem je, že uživatelé neprovádí pouze check out nad danými soubory, nýbrž mají na svém systému k dispozici kompletní kopii repozitáře, včetně celé historie [36].

Ve své podstatě tedy neexistuje jeden centrální repozitář. V "plně" distribuovaném projektu každý přispěvatel spravuje svou vlastní verzi projektu, přičemž různí přispěvatelé hostují své vlastní verze a podle potřeby stahují změny od ostatních uživatelů [36].

Takovéto uspořádání ale může být často velmi obtížně udržovatelné, což vede k tomu, že je u mnoho projektů použito jiné paradigma, ve kterém je jeden repozitář označen jako centrální, tedy repozitář, se kterým se ostatní repozitáře pravidelně synchronizují [36]. Tento repozitář je tedy považován za oficiální a je kolektivně spravován. V důsledku toho není verzování již "plně" distribuované vzhledem k existenci centrálního prvku ve formě repozitáře. Zatímco tedy distribuované systémy správy verzí obecně povolují vývojářům kopírovat jakýkoliv repozitář, v tomto paradigmatu se kopíruje vždy jen ten oficiální v rámci projektu. Toto paradigma je více probráno v kapitole 4.1.3.

Obecný princip distribuovaných systémů správy verzí je znázorněn na následujícím obrázku 2.2.

Jedna z hlavních výhod distribuovaných systémů správy verzí vyplývá z jejich základní charakteristiky, tedy že jsou distribuované, čímž přirozeně vzniká několik offline záloh. Pokud tedy nastane scénář typu pádu centrálního serveru, který by mohl být při použití centralizovaných systémů až



Obrázek 2.2: Diagram distribuovaných systémů správy verzí.

katastrofální, tak v případě distribuovaných systémů stačí prohlásit nejaktuálnější kopii repozitáře za primární a synchronizovat ji se znovu zprovozněným či novým centrálním serverem. Na rozdíl od centralizovaného systému správy verzí odstraňuje tedy distribuovaná správa verzí spoléhání se na jedinou zálohu, díky čemuž je vývoj spolehlivější. Obvyklá představa je, že nutnost existence kompletní kopie repozitáře na každém systému může být plýtváním místem, což lze ale ve většině případů označit za představu mylnou vzhledem k tomu, že se projekty často skládají zejména ze zdrojových souborů ve formátu prostého textu a mnoho systémů správy verzí soubory komprimuje. Dopad na úložiště pevného disku je tedy ve většině případech minimální [7]. Existují ale výjimky, u kterých je tato představa oprávněná, jako například v případě vývoje her, čemuž se podrobněji věnuje kapitola 3.1.

Další výhoda vyplývající z distribuované povahy je možnost pracovat offline, což má několik důsledků. Jeden z nich je zřejmý, tedy možnost pracovat bez připojení k síti, které je potřeba pouze k synchronizaci s centrálním repozitářem. Většina lokálních operací lokálních je také velmi rychlá vzhledem k absenci nutnosti síťové komunikace. Lze také provádět všelijaké experimenty nad projektem s využitím systému pro správu verzí a experiment lze poté buď zveřejnit pro ostatní členy týmu, nebo jej lze zrušit a ostatní členové týmu se o tomto experimentu ani nemusí dozvědět a nijak je neovlivní [7].

Distribuované systémy správy verzí jsou také ideální volbou k vývoji tzv. *Free and open-source software (FOSS)* vzhledem k flexibilitě procesů verzování [36]. Příkladům takovýchto procesů verzování je věnována mimo jiné kapitola 4.

Existuje ale také několik potenciálních nevýhod distribuovaných systémů správy verzí. Lze mezi ně zařadit fakt, že je jejich distribuovaná povaha složitější k pochopení a používání, a to zejména pro netechnické uživatele, což mimo jiné vychází i z průzkumu v kapitole 3.2. Je tedy vhodné zajistit, aby všichni členové týmu byli s distribuovanými systémy správy verzí a zvoleným verzovacím procesem důkladně obeznámeni, či v některých prostředích dokonce i používání distribuovaných systémů zvážit, jak i vyplývá z citátu z eseje *Undiagnosing Subversion* [48] od Grega Hudsona – *„V mnoha prostředích je nejdůležitější vlastností systému správy verzí křivka učení.“* (přeloženo z angličtiny).

I když je v mnoha případech zamykání souborů zbytečná, až nežádoucí vlastnost systémů správy verzí, tak v některých případech – jako například při vývoji her – může být žádaná [57]. Při vývoji her totiž mohou být verzovány společně se zdrojovými soubory také soubory vytvořené umělci, u nichž je ve většině případů nežádoucí paralelní práce vícero členů týmu vzhledem k tomu, že následné sloučení jednotlivých změn může být velice složité nebo i nemožné, viz kapitola 3.1.3. V případě paralelní práce tedy může nastat nutnost zachovat pouze jednu změnu a ostatní ignorovat a úsilí nejméně jednoho člena týmu tedy může přijít vniveč. A jelikož podpora zamykání je podmíněna neustálým přístupem k serveru, který zamykání spravuje, tak ze své podstaty většina distribuovaných systémů pro správu verzí zamykání nepodporuje vzhledem k tomu, že jsou navrženy i pro podporu offline práce. Této problematice se mimo jiné podrobněji věnuje kapitola 3.1.3.

Příklad vývoje her lze použít i pro další potenciální nevýhodu distribuovaných systémů správy verzí. Pří vývoji her, jak již bylo řečeno, se verzují i soubory vytvořené umělci, což jsou často binární soubory [57]. A vzhledem k povaze, jak většina distribuovaných systémů přistupuje k verzování binárních souborů – viz kapitola 2.3.1 věnována vnitřní reprezentaci Gitu – může velikost repozitáře narůst do mnoha desítek gigabytů až terabytů [57]. To může být problém vzhledem k nutnosti existence kompletních kopií repozitáře na stroji každého vývojáře.

Mezi distribuované systémy správy verzí se řadí například *GNU Arch*[1], *Monotone*[2], *Darcs*[3], *Mercurial*[4] a v současné době nejpopulárnější systém správy verzí vůbec [28] – Git[5] – kterému je věnována samostatná kapitola 2.3.

2.3 Git

Během raných let údržby Linuxového jádra (1991–2002) byly změny předávány jako záplaty a archivované soubory [36]. Od roku 2002 se začal pro verzování projektu Linuxového jádra používat proprietární distribuovaný systém správy verzí s názvem *BitKeeper*. Používání tohoto nástroje přetrvávalo do roku 2005, kdy vznikly neshody mezi komunitou vyvíjející Linuxové jádro a komerční společností vyvíjející BitKeeper a nástroj přestal být pro vývojový tým bezplatný. To podnítilo vývojovou komunitu Linuxu k vývoji vlastního nástroje, který měl být inspirovaný právě nástrojem BitKeeper. Pro nový nástroj byly vytyčený některé cíle, zejména:

- rychlost,
- jednoduchý návrh,
- podpora pro nelineární vývoj,
- plná distribuovanost,
- schopnost být efektivně použitelný pro velké projekty, jako je například Linuxové jádro.

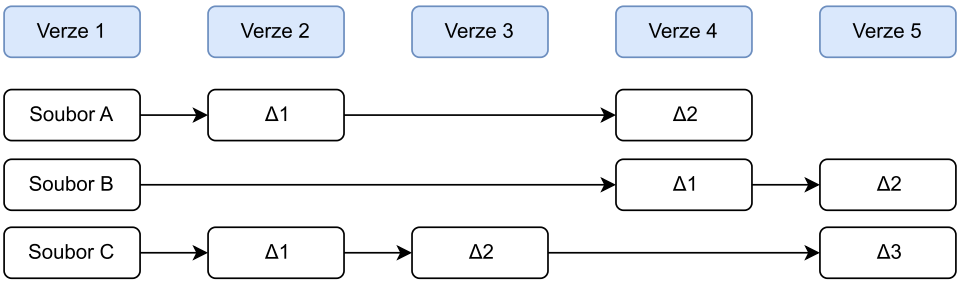
Nástroj byl pojmenován Git a jeho první verze byla vydána ještě v roce začátku vývoje, tedy roku 2005 [36].

2.3.1 Vnitřní reprezentace

Při návrhu byl k verzování zvolen zcela jiný přístup v porovnání s ostatními systémy správy verzí [36]. Většina systémů správy verzí totiž přemýšlí nad ukládanými informacemi jako nad množinou jednotlivých souborů s postup-

[1] www.gnu.org/software/gnu-arch/
[2] www.monotone.ca
[3] darcs.net
[4] www.mercurial-scm.org
[5] git-scm.com

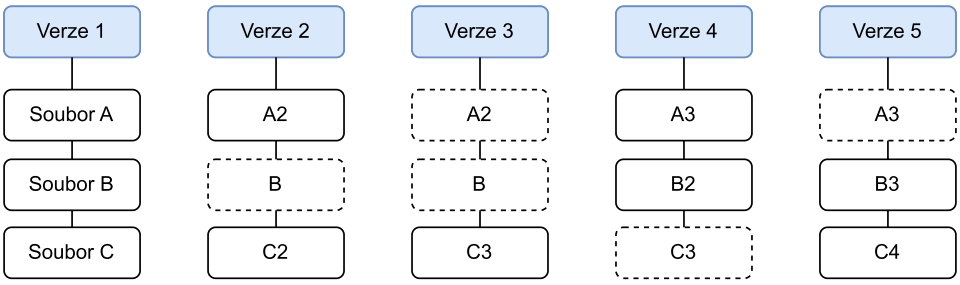
nými změnami nad nimi prováděnými. Tyto systémy se běžně označují jako tzv. *delta-based* systémy správy verzí, jak již bylo dříve zmíněno. Princip delta-based systémů je znázorněn na obrázku 2.3.



Obrázek 2.3: Princip delta-based systémů správy verzí.

Git ale nad daty přemýšlí jinak, a to jako nad posloupností snímků miniaturního souborového systému [36]. Pokaždé, když se provedou změny a provede se tzv. *commit*, se vytvoří snímek současného stavu všech souborů v daném momentě a uloží se na daný snímek reference. Pokud se v rámci commitu změní soubor, tak se v principu vytvoří kompletní kopie daného obsahu souboru a aplikují se na ni provedené změny. Tato kopie je poté referencována v novém snímku. Kopie obsahu souborů, které se v porovnání s předchozím snímkem nezměnily, se z důvodu úspory diskového místa nevytváří a nový snímek tedy referencuje obsah daného souboru z předchozího snímku.

Git tedy nad daty přemýšlí jako nad sérií snímků daného projektu, který je reprezentovaný miniaturním souborovým systémem. Jednou z obrovských výhod tohoto návrhu je umožnění velmi jednoduché implementace větvení, kdy větev není v principu nic jiného, než prostý ukazatel na konkrétní snímek. Popsaný princip je znázorněn na obrázku 2.4.



Obrázek 2.4: Princip ukládání dat jako posloupnosti snímků projektu.

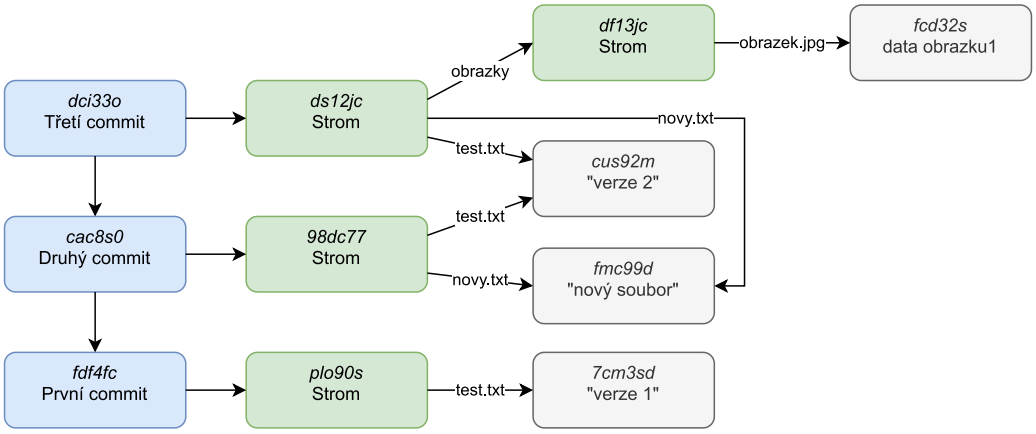
Realizace popsaného principu je provedena pomocí tří objektů [61]:

- *Blob* je obsahem jedné verze souboru [61]. Tento objekt neobsahuje žádná metadata typu jméno či časové razítko. Jeho interním identifikátorem je pouze hash jeho obsahu.

- *Strom* – pokud se budeme držet pohledu na Git, jako na miniaturní souborový systém – si lze představit jako jeden adresář [61]. Obsahuje seznam názvů, z nichž je ke každému přiřazen mimo jiné identifikátor objektu typu blob nebo strom. Interním identifikátorem je opět hash, který vzniká použitím hash funkce na všechny potomky. Pokud tento objekt nemá žádného předka, tak se jedná o kořen snímku celého projektu.

- *Commit* spojuje objekty typu strom do hierarchie představující jednotlivé revize projektu [61]. Obsahuje zejména identifikátor kořenového objektu strom, časové razítko, textový popis revize a identifikátor commit objektů, které jsou bezprostředními předchůdci tohoto commitu. Předchůdce může být buď jeden nebo i více v případě, že současný commit vznikl sloučením. Pro právě jeden commit v repozitáři platí, že nemá žádného předchůdce, a to pro commit nejstarší. I objekty typu commit jsou reprezentovány hodnotou hashovací funkce.

Každý ze zmíněných objektů je tedy identifikován hodnotou hashovací funkce *SHA-1* svého obsahu [36]. Princip, jak jsou objekty vytvářeny a jak se sebou objekty navzájem souvisí, je znázorněn na obrázku 2.5.



Obrázek 2.5: Příklad Git objektů

2.3.2 Pracovní oblasti

Lze říci, že z pohledu změn rozděluje Git projekt do čtyř základních oblastí [56]:

- *Pracovní adresář* (anglicky *Working directory*) představuje jednu revizi projektu, na které se zrovna pracuje [36]. Všechny soubory dané revize jsou vytaženy z komprimované Git databáze a umístěny na disk, aby se mohli použít a případně upravit.

- *Oblast připravených změn* (anglicky *Staging area*) obsahuje informace o tom, co bude součástí příštího *commitu* [36]. Změny do oblasti připravených změn obvykle přidává uživatel z pracovního adresáře, pokud je s danými změnami spokojen a jsou připraveny ke commitu. Tato oblast je také označována jako *index*.

- *Repozitář* je místo, kde Git uchovává veškerá metadata a databázi objektů daného projektu [56]. Dá se označit za nejdůležitější část Gitu. V případě tzv. klonování se kopíruje právě vzdálený repozitář, se kterým se lze případně pravidelně synchronizovat.

- *Oblast odložené práce* (anglicky *Stash area*) lze použít v případě, kdy je potřeba mít čistý pracovní adresář a oblast připravených změn, ale v jedné z daných oblastí se zrovna nachází změny, které ale budou v budoucnosti žádané [56]. Dané změny tedy lze přesunout do oblasti odložené práce a v případě potřeby je lze opět obnovit.

2.3.3 Git Large File Storage

Jak již bylo řečeno, Git funguje velmi dobře pro projekty skládající se zejména z textových souborů a případné velké binární soubory jsou z pohledu verzování statické. Pokud je nutno v rámci projektu velké binární soubory často měnit, jako například v případě vývoje her, tak Git přestává být efektivní, což vyplývá z vnitřní reprezentace popsané v kapitole 2.3.1. Velikost repozitáře totiž může narůst do rozměrů, kdy už není žádané, aby jej měl každý uživatel zkopírovaný na svém stroji. A právě tento nedostatek byl jeden z důvodů vzniku rozšíření *Git LFS* [12].

Git LFS je rozšíření Gitu vyvinuté společnostmi *Atlasian*, *Github* a několika dalšími open-source přispěvateli [2]. Zmíněný problém řeší rozšířením Gitu o možnost ukládání velkých binárních souborů do zvláštního úložiště a v

Git repozitáři si uchovává pouze ukazatele na konkrétní verzi souboru v daném úložišti. Uživatel tedy nemá při zkopírování repozitáře na svůj stroj neustále k dispozici všechny verze daného souboru. Stahování jednotlivých verzí souborů nastává až v případě zvolení konkrétní revize projektu, kdy se z dodatečného úložiště stáhne pouze relevantní verze daného souboru.

Velká výhoda Git LFS je, že jeho používání pro správu velkých binárních souborů je naprosto transparentní [2]. Běžnému uživatel stačí rozšíření pouze nakonfigurovat a poté může Git používat standardně bez jakékoliv změny v již zavedených postupech. Git LFS se totiž automaticky stará o proces výměny ukazatelů za skutečné soubory.

Git LFS je podporováno mnoha populárními Git hostingy, jako například *Github* [10], *Gitlab* [12] nebo *Bitbucket* [4]. Git LFS lze ale samozřejmě zprovoznit také na vlastním serveru využitím například referenční implementace[6].

Kromě ukládání souborů mimo repozitář poskytuje Git LFS také možnost zamykání souborů [2]. Repozitář lze nakonfigurovat tak, aby určité soubory byly uzamykatelné. Tyto soubory jsou poté ve výchozím stavu dostupné pouze pro čtení, dokud uživatel nad daným souborem nezíská exkluzivní zámek pro zápis. Po získání zámku je tedy umožněna úprava souboru pouze uživateli držícím zámek, dokud zámek neuvolní pro případné další zájemce o úpravu daného souboru.

Použití Git LFS rozšiřuje Git o – v některých případech – užitečné funkce, s kterými se ale také pojí několik potenciálních úskalí, jako například:

- *Komplexita*: Nutnost konfigurace rozšíření a potřeba dodatečné instalace softwaru přináší zvýšenou složitost přípravy vývojového prostředí. Pro mnoho prostředí je ale již vhodná konfigurace vytvořená, ze kterých lze vyjít a provést pouze vhodné úpravy. Příkladem mohou být konfigurace pro populární herní enginy *Unity*[7] a *Unreal Engine*[8].

- *Vyšší nároky na serverové úložiště*: Dalším z potenciálních úskalí mohou být vyšší nároky na serverové úložiště vzhledem k tomu, že Git LFS neprovádí nad soubory žádnou kompresi [59]. Toto úskalí samo-

[6]https://github.com/git-lfs/lfs-test-server
[7]https://gist.github.com/nemotoo/b8a1c3a0f1225bb9231979f389fd4f3f
[8]https://github.com/MOZGIII/ue5-gitignore/blob/master/.gitattributes

zřejmě odpadá v případě, kdy se na serverové úložiště dostávají soubory již komprimované.

- *Limitovaná podpora hostingů*: I přesto, že bylo dříve v této kapitole zmíněno, že Git LFS podporuje mnoho populárních hostingů, tak samozřejmě může nastat situace, kdy daný hosting rozšíření nepodporuje. Z toho vyplývá určitá limitace ve volbě výběru hostingu.

- *Limitovaná podpora klientů*: Limitace výběru platí také pro Git klienty. Ne všechny totiž podporují například možnost zamykání souborů. Příkladem populárního klienta nepodporující zamykání je *SourceTree* [51].

- *Limitace možnosti offline práce* - Vzhledem k nutnosti ukládání některých souborů na centrálním serveru se zvyšuje závislost na připojení k síti a tím tedy odpadá – v případě nutnosti měnit tyto soubory – jedna z výhod distribuovaných systémů správy verzí, tedy možnost práce offline. Pokud se využívá i možnosti uzamykání souborů, tak tato limitace ještě narůstá.

Před zavedením Git LFS je tedy vhodné důkladně zvážit, zda přínos rozšíření převažuje nad zmíněnými úskalími, což v některých prostředích platit může, jako například při vývoji her, kde obě zmíněné vlastnosti – možnost ukládání velkých binárních souborů mimo repozitář a zamykání souborů – mohou velkým přínosem, jak je zmíněno v kapitole 3.1.

3 Verzování ve vývoji videoher

Videohry patří mezi jeden z nejběžnějších typů softwaru. Například v červnu 2021 představovaly hry na *Apple App Store* 21,8% ze všech dostupných aplikací [19]. Herní trh je na vzestupu a očekává se, že velikost trhu s videohrami vzroste v roce 2024 až na 218,7 miliard dolarů.

Většina dnešních her je vyvíjena pomocí herních enginů [58] – jako například *Unity* nebo *Unreal Engine* – kde umělci s malými nebo žádnými zkušenostmi s vývojem softwaru a nástroji s vývojem softwaru spjatými vytváří pro hru umělecká díla a spolupracují se softwarovými vývojáři píšící kód [33].

Nicméně v mnoha aspektech je vývoj her obdobný, jako vývoj jakéhokoliv jiného druhu softwaru. Tedy i u vývoje her je tedy vhodné používání osvědčených postupů a nástrojů pro softwarový vývoj, jako například používání systémů pro správu verzí. Některé z těchto nástrojů – jako například Git – však byly vyvinuty primárně pro správu zdrojových kódů softwarovými vývojáři [33], což není v případě vývoje her optimální stav vzhledem ke specifikům, které již byly v této práci dříve zmíněny. Těmto specifikům se v souvislosti s Gitem dále také podrobněji věnuje kapitola 3.1.

3.1 Git a specifika verzování při vývoji her

Mnoho specifik vývoje her vyplývá z něčeho tak základního, jako je složení týmu. Při vývoji her úzce spolupracují umělci se softwarovými vývojáři na vytváření a dodávání herního softwaru, což při používání Gitu – nástroje určeného primárně pro softwarové vývojáře – může přispívat ke značnému tření při práci. Níže se konkrétním specifikům, způsobující toto tření, detailněji věnují jednotlivé kapitoly. U každého specifika je popsáno, jak ke tření přispívají a jak jej lze limitovat či odstranit.

3.1.1 Netechnické profese součástí vývoje

Designéři a umělci – tvořící významnou část týmů pro vývoj her – jsou zpravidla méně zvyklí na relativně složité postupy pojící se s používáním

Gitu, jako například větvení a commitování, a jeho distribuovaná povaha pro ně může být špatně uchopitelná. Vyplývá to mimo jiné z průzkumu, kterému je věnována kapitola 3.2. Přesto je potřeba, aby jejich výstupní artefakty byly spolu s kódem součástí dodávky výsledného softwaru [33].

Jedním z možných řešení je umělce od Gitu naprosto odstínit. Jejich výstupy se místo toho mohou nahrávat na separátní úložiště typu *Google Drive*[1] nebo *Dropbox*[2]. S tímto přístupem se ale samozřejmě pojí několik problémů. Tím nejzásadnějším je, že vzniká potřeba dodatečného kroku pro sestavení aplikace a případné testy, čímž vzniká tření hlavně v rámci automatizace.

Přestože je Git nástroj určený primárně pro příkazovou řádku [49], tak existuje i řada grafických uživatelských rozhraní, viz oficiální list nástrojů třetích stran [8]. Použitím grafického rozhraní se limituje nutnost používání příkazové řádky a pamatování si přesného znění jednotlivých příkazů, což zvyšuje uživatelskou přívětivost, která je pro netechnické profese zásadní. Avšak i při používání grafických rozhraní pro provádění základních příkazů se problém strmé křivky učení zcela nevyřeší vzhledem k tomu, že distribuovaný model Gitu jako takový a sémantika jednotlivých příkazů jsou pro netechnické profese těžko uchopitelné. Používá-li se navíc v týmu komplexnější proces verzování – viz kapitola 4 – do kterého by měli být zapojeni právě i umělci, tak je těžká uchopitelnost ještě výraznější.

Řešením může být grafické uživatelské rozhraní, které si je vědomé definovaného procesu, a uživatelům nabízí pouze relevantní akce. Návrh a implementace takového nástroje je součástí této práce a je mu věnována kapitola 7.

3.1.2 Velikost souborů

Jak bylo popsáno v kapitole 2.3, každý vývojář používající Git má na svém stroji kopii celého repozitáře včetně celé historie. Postupem času se tedy repozitář může rozrůst do stavu, kdy je příliš objemný pro pevné disky a příliš pomalý pro stahování. Git byl totiž vytvořen primárně pro práci s textovými soubory, ale ve vývoji her tvoří velké binární soubory významnou část projektu a díky tomu se repozitář může rychle zvětšit za rozumné limity [49].

[1] https://www.google.com/drive/
[2] https://www.dropbox.com/

Tomuto problému se lze v rámci Gitu vyhnout zavedením rozšíření Git LFS, viz kapitola 2.3.3, díky kterému není obsah velkých binárních souborů přímou součástí repozitáře.

3.1.3 Složitost slučování změn

Předchozí kapitola se týkala problematiky velikosti souborů. V herním průmyslu ale velikost není jediné specifikum týkající se souborů. Další specifikum vyplývá z povahy souborů spravovaných umělci, konkrétně složitost slučování paralelně provedených změn jednoho souboru několika umělci. Na tento problém se dá pohlížet ze dvou úhlů pohledu – z technického a uměleckého.

Technický pohled na problém

Technický pohledem se myslí fakt, že většina souborů produkovaných umělci jsou binární soubory, jako například textury, obrázky, zvukové soubory a další [34]. A pro binární soubory platí, že nad nimi nelze jednoduše řešit tzv. konflikty.

Konflikt může vzniknout v případě, kdy je snaha o sloučení dvou nebo více změn stejného souboru a tyto změny si z pohledu systému pro správu verzí navzájem odporují [36]. U zdrojových a ostatních textových souborů je šance, že současně prováděné změny nad daným souborem půjde jednoduše sloučit i v případě konfliktu, kdy lze jednotlivé změny porovnat a ručně aplikovat. To samozřejmě neplatí pro všechny případy a i řešení konfliktů nad textovými soubory může být velmi náročné a může nastat situace, kdy je třeba rozhodnout se pro aplikování pouze jedné změny a ostatní ignorovat. Ale v případě, kdy se jednotlivé změny týkají jiných částí souboru a nepřekrývají se, nemusí konflikt ani vzniknout [34].

V případě binárních souborů není ale obvyklé, že by se změny prováděly pouze v malé části souboru. Jednotlivé změny většinou zasahují do velké části souboru a často se překrývají. Jako příklad lze uvést zvukové soubory, kde malá změna – jako například mírná úprava úrovní v ekvalizéru – může znamenat změnu téměř celého obsahu souboru [34].

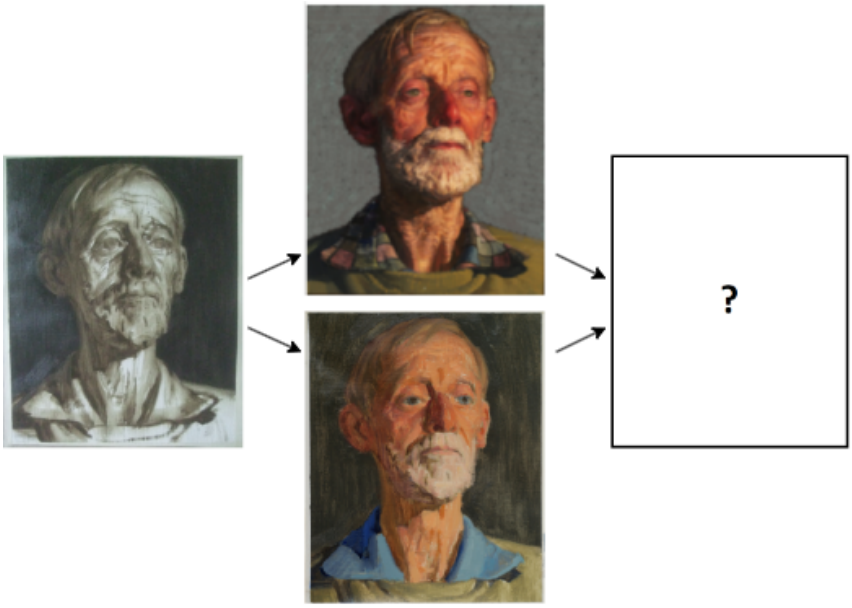
Je také vhodné zmínit, že se tento problém nelimituje pouze soubory spravovaných umělci. Například v případě Unreal Engine může být totiž v určitém smyslu binárním souborem i kód. V Unreal Engine totiž existuje koncept tzv. *blueprintů* umožňující vizuální skriptování [30]. A právě blueprinty jsou

ukládány v binárním formátu.

Umělecký pohled na problém

Pokud by se vzalo v potaz, že je technická stránka tohoto problému vyřešena a existoval by nástroj umožňující jednoduché slučování změn nad daným binárním formátem, tak problém z druhého zmíněného pohledu na věc ale zůstává stále relevantní – tedy z pohledu uměleckého.

Ten se na tento problém dívá ze spíše filozofického hlediska. Práce na uměleckém díle totiž probíhá vždy v určitém kontextu, který je v umění velice důležitý [34]. Dá se tedy říci, že jednotlivé změny jsou relevantní pouze v daném kontextu. Příklad je znázorněn na obrázku 3.1, kde jsou znázorněny dvě varianty obrazu, které vycházely ze stejné verze. Pokud by bylo následně potřeba obě verze sloučit, tak je zřejmé, že by to bylo z uměleckého hlediska téměř nemožné a jedna varianta by se musela ignorovat.



Obrázek 3.1: Ukázka důležitosti uměleckého kontextu [37].

Z poznatků v této kapitole popsaných lze tedy udělat závěr, že konflikty nejsou u uměleckých souborů žádané a ideálně by se jim mělo předcházet. Toho se dá do určité míry docílit tím, že se zavede mechanismus zabraňující paralelní práci na souboru.

Zabránění vzniku konfliktů komunikací v týmu

Jednou z metod zabránění paralelní práce je komunikace v týmu, čímž je myšleno, že je nějakým způsobem komunikováno – například záznamy ve sdílené tabulce – na kterém souboru právě jednotliví členové týmu pracují, viz příklad v tabulce 3.1. Předtím, než člen týmu začne pracovat na souboru, se tedy musí podívat do dané tabulky, zda není právě upravován někým jiným. Pokud se v tabulce záznam s daným souborem již nachází, tak jej nesmí editovat. Pokud v tabulce takový záznam není, tak jej tam může přidat a soubor upravit. Po skončení práce vytvořený záznam musí z tabulky smazat.

Tato metoda je samozřejmě náchylná na chyby, neškáluje se dobře s velikostí týmu a odporuje základní myšlence vzniku systémů pro správu verzí – tedy automatizaci. Přesto je ale tato metoda používána, jak vyplývá z průzkumu popsaném v kapitole 3.2.

Tabulka 3.1: Příklad komunikace právě upravovaných souborů pomocí tabulky.

Soubor	**Autor**	**Datum**	**Popis**
logo.png	Václav	1/2/2023	Aktualizace loga
faceTexture.png	Karel	2/2/2023	Úprava textur
hairTexture.png	Karel	2/2/2023	Úprava textur

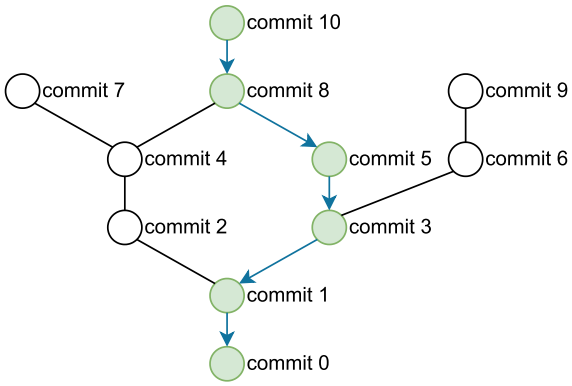
Zabránění vzniku konfliktů omezením na základě commit grafu

Další metoda je postavena na základní grafové teorie a relací mezi commity. Commity v Gitu totiž tvoří tzv. *commit graf*. Pokud se commit graf udržuje v takovém stavu, kde pro daný soubor platí, že všechny commity ovlivňující daný soubor tvoří tzv. cestu, tak lze s jistotou říci, že nad daným souborem nemůže nikdy vzniknout konflikt [34].

Příklad commit grafu, kde je tato podmínka splněna pro soubor `foo.bar`, je znázorněn na obrázku 3.2, kde jsou zeleně označeny commity, ve kterých se změnil právě soubor `foo.bar`. Lze vidět, že právě tyto commity tvoří v grafu cestu, která je znázorněna modře. Pokud by se soubor `foo.bar` změnil například v `commitu 4`, tak by daná podmínka byla porušena a již by nebylo zaručeno zabránění vzniku konfliktů nad daným souborem.

Zajištění této podmínky lze provést zavedením validace pro každý nový commit. Konkrétně se jedná o validaci zajišťující, že daný commit je potomkem

všech commitů v rámci něhož se daný soubor změnil. Tato validace se provádí před každým commitem a pokud validace selže, tak se commit nevytvoří. Lze si ověřit, že tato podmínka je splněna pro všechny označené commity na obrázku 3.2.



Obrázek 3.2: Commit graf splňující podmínku cesty pro soubor `foo.bar`.

Doposud se popis této metody omezoval pouze na lokální repozitář, což samozřejmě v kontextu Gitu není ve většině případů relevantní. Aby byla metoda použitelná, je ji tedy nutno rozšířit o podporu distribuovaného prostředí. Takovým rozšířením může být zavedení "globálního grafu", vůči kterému se všechny commity lokálních repozitářů validují a zároveň se do něj veškeré commity automaticky publikují. Globálním grafem se v tomto případě nemyslí centrální repozitář – kterému se věnuje například kapitola 4.1.3 – ale jedná se o další společný repozitář v rámci projektu sloužící výhradně pro účely zamezení konfliktů. Implementací této metody je open-source projekt *Git Global Graph*[3].

Výhoda této metody, v porovnání například s explicitním zamykáním souborů, je odpadnutí nutnosti držení dodatečné informace o souborech – tedy informace o vlastnících zámků. Další výhodou je, že zamykání je víceméně zakódované v commit grafu a není tedy nutno manuálně zamykat či odemykat soubory, což může být výhoda například i při používání grafických uživatelských rozhraní, jelikož ne každé zamykání podporuje [51].

Tato metoda má ale také několik nevýhod. Patří mezi ně nutnost existence dalšího repozitáře a relativní nevyzrálost implementace [34]. Největší nevýhodou ale je, že zamykání probíhá až na úrovni commitů. Uživatel tedy může na daném souboru kdykoliv bez problému pracovat, ale pokud se po-

[3]https://github.com/Kleptine/gitglobalgraph

kusí s danými změnami vytvořit nový commit, tak může být tento commit odmítnut a provedená práce tedy může vyjít jako zbytečná.

Zabránění vzniku konfliktů explicitním zamykáním

Poslední zde zmíněná metoda byla již popsána v kapitole 2.3.3, tedy explicitní zamykání souborů pomocí rozšíření Git LFS. Zamykání se provádí příkazem `git lfs lock <soubor>` a odemykání příkazem `git lfs unlock <soubor>` [2].

Vyzrálost rozšíření Git LFS a jeho podpora populárních hostingů [4][10][12] lze zařadit mezi výhody této metody. Výhodou také je, že Git LFS začíná operovat již na úrovni souborového systému a nikoliv až na úrovni systému správy verzí, jak tomu je u metody předchozí [2]. Rozšíření totiž zajišťuje, že všechny uzamykatelné soubory jsou ve výchozím stavu nastavené pouze pro čtení a zápis je povolen pouze v případě, kdy uživatel drží zámek pro daný soubor. Situace, kdy uživatel zjistí, že tento tento soubor nemůže upravovat, až při vytvoření commitu, tedy nenastává a tím se snižuje šance na vznik zbytečné práce, která může v případě předchozí metody nastat.

Mezi nevýhody lze zařadit přidání dodatečného manuálního prvku do procesu verzování, tedy nutnost zamykání a odemykání souborů. S explicitním zamykáním se také pojí zvýšené nároky na pečlivost při používání systému správy verzí, zejména při odemykání souborů. Pokud se totiž soubor odemkne dříve, než se v rámci rozumného verzovacího procesu očekává – například před sloučením *feature* větve do hlavní vývojové větve, viz kapitola 4 – tak může nastat konflikt navzdory používání zamykání.

3.2 Průzkum

Vzhledem k tomu, že nebylo nalezeno dostatek materiálů obsahujících informace pro provedení určitých závěrů, byl proveden v rámci této práce průzkum. Mezi hlavní cíle patřilo potvrzení hypotézy, že je používání systémů správy verzí – zejména Gitu – pro netechnické profese těžko uchopitelné a zjištění dostatku informací pro určení požadavků na proces verzování navržený v kapitole 5. Další cíle se týkaly hlubšího proniknutí do současného stavu verzování ve vývoji her.

3.2.1 Sběr dat

Pro sběr dat byl vytvořen formulář zaměřující se zejména na následující body:

- Jaké se používají systémy správy verzí a jak je uživatelé hodnotí.
- Zda je umožněna souběžná práce nad soubory a případně jak se jí zamezuje.
- Jak často nastávají konflikty a jak složité je jejich řešení.
- Jaké jsou nároky na model větvení.
- Jaké praktiky s verzováním úzce spojené se používají a případně jaký je důvod jejich nepoužívání.

Bylo vybráno několik online komunit, které byly vyhodnocené jako vhodné pro tento průzkum. V rámci těchto komunit byl zmíněný formulář zveřejněn v termínech 26.10.2022 a 25.11.2022. Dále byl formulář zaslán konkrétním lidem zabývající se herním vývojem. Počet respondentů z jednotlivých skupin je uveden v tabulce 3.2.

Tabulka 3.2: Komunity pro sběr dat.

Název skupiny	**Platforma**	**Počet respondentů**
Games Development Community CZ/SK	Facebook	58
Game Developers	Facebook	7
Game Development	Facebook	5
r/GameArt	Reddit	11
r/gamedev	Reddit	5
r/indiegames	Reddit	4
Konkrétní lidé	-	3
	Celkem:	**93**

3.2.2 Otázky a analýza odpovědí

Odpovědi ze všech skupin byly sjednoceny do jednoho souboru `Vysledky/pruzkum_odpovedi.csv`, který je přiložen k této práci. Otázky lze tématicky rozdělit do pěti kategorií:

- Kategorie 1: Kategorizace respondenta.
- Kategorie 2: Systémy správy verzí.
- Kategorie 3: Verzovací proces.
- Kategorie 4: Praktiky úzce související s verzováním.
- Kategorie 5: Otevřené otázky.

Níže jsou jednotlivé kategorie představeny včetně jednotlivých otázek. U otázek, pro které platí, že jsou relevantní i bez kontextu ostatních odpovědi, jsou odpovědi prezentovány v nepozměněné formě. Pro některé otázky je ale kontext ostatních odpovědí zajímavý nebo dokonce i zásadní a pro nalezení vhodných informací byl třeba pečlivější vhled do dat. Pro takové otázky jsou poté odpovědi prezentovány v určitém kontextu, jako například omezením se na podmnožinu respondentů podle daného kritéria za účelem hledání souvislostí.

Součástí některých otázek byly i vysvětlivky, které jsou v rámci této kapitoly vynechány. Příkladem je otázka 3.3, kde bylo účastníkovi v poznámce vysvětleno, co se myslí konfliktem. Některé otázky jsou také podmíněné odpovědí z předchozích otázek. Příkladem může být otázka *4.1.1: Z jakého důvodu Váš tým nepraktikuje code review?*, která se uživateli zobrazila pouze v případě, kdy uživatel odpověděl negativně na předchozí otázku *4.1: Praktikuje Váš tým tzv. code review?*.

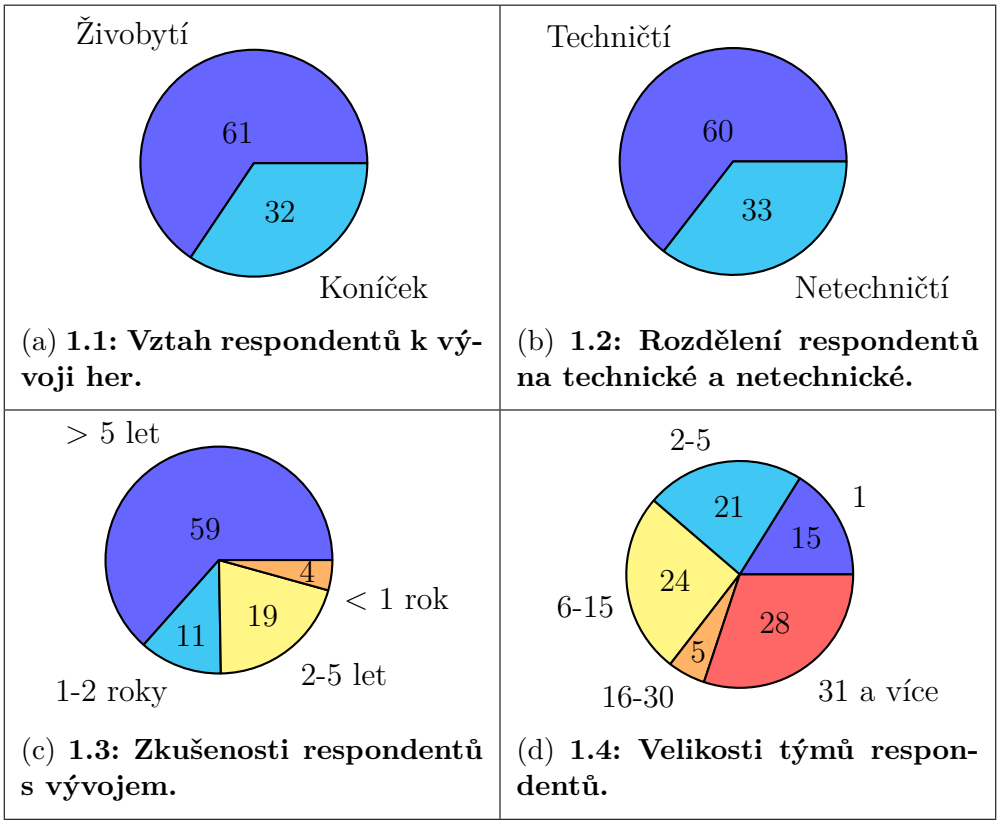
Kategorie 1: Kategorizace respondenta

První důležitou kategorií je jednoduchá kategorizace respondenta, na které poté staví analýza v rámci dalších kategorií. Kategorizace proběhla pomocí čtyř otázek, konkrétně:

- Otázka 1.1: Jaký máte vztah k vývoji her?
- Otázka 1.2: Jaké zastáváte pozice?
- Otázka 1.3: Jak dlouho se vývoji her (či jiného softwaru) věnujete?
- Otázka 1.4: V kolikačlenném týmu pracujete?

U otázky 1.2 bylo možné vybrat několik možností, jako například programátor, umělec, tester a další. Pro potřeby analýzy ale byly odpovědi zobecněny pouze na technické a netechnické pozice. Pokud respondent zvolil technickou i netechnickou pozici zároveň, byl zařazen mezi technické pozice.

Odpovědi na všechny zmíněné otázky jsou vizualizovány v jednotlivých grafech na obrázku 3.3.



(a) **1.1: Vztah respondentů k vývoji her.**

(b) **1.2: Rozdělení respondentů na technické a netechnické.**

(c) **1.3: Zkušenosti respondentů s vývojem.**

(d) **1.4: Velikosti týmů respondentů.**

Obrázek 3.3: Kategorizace respondenta.

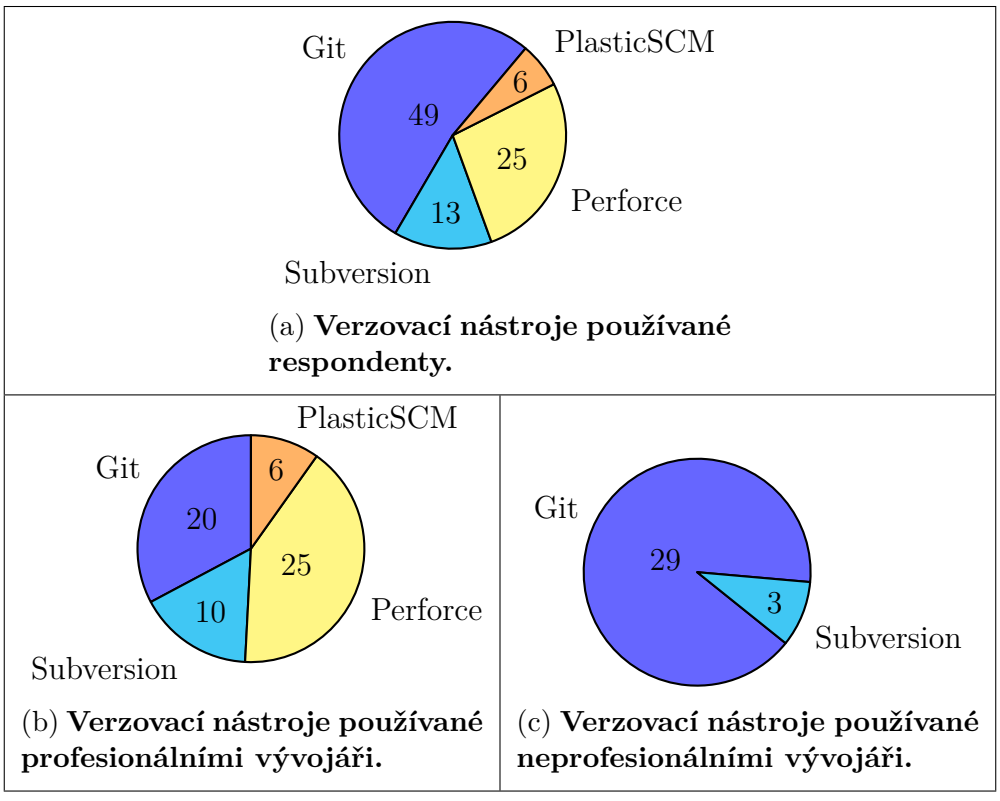
Kategorie 2: Systémy správy verzí

Druhá kategorie otázek se věnuje průzkumu používaných systémů správy verzí, jejich hodnocení a případným problémů. Hlavním cílem bylo potvrzení hypotézy, že je používání Gitu pro netechnické profese problematické, což může být příčinou vzniku tření při vývoji.

Kategorie se skládá z těchto otázek:

- Otázka 2.1: Jaký hlavní nástroj používáte pro verzování?
- Otázka 2.2: Jak byste daný nástroj ohodnotil/a?
- Otázka 2.3 (volitelná): Pokud se Vám nástroj nepoužívá dobře, jaké jsou důvody?
- Otázka 2.4: Jak často se obracíte na zkušenějšího kolegu, abyste vyřešili problém s verzovacím nástrojem?

V rámci této kategorie nejprve proběhla analýza používání jednotlivých nástrojů z několika pohledů. Výsledky jsou vizualizovány v grafech na obrázku 3.4.



(a) **Verzovací nástroje používané respondenty.**

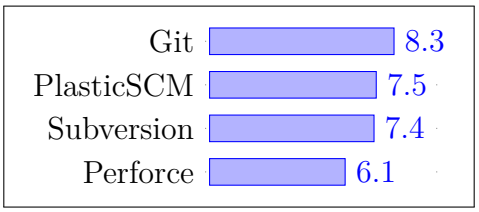
(b) **Verzovací nástroje používané profesionálními vývojáři.**

(c) **Verzovací nástroje používané neprofesionálními vývojáři.**

Obrázek 3.4: Používané systémy správy verzí.

Graf (a) vizualizuje odpovědi všech respondentů. Graf (b) je poté limitován pouze na respondenty, pro které je vývoj her živobytí a graf (c) je naopak limitován na respondenty, pro které je vývoj her koníček. Z grafů lze vyčíst, že v neprofesionálním prostředí jasně dominuje Git, což lze přisoudit například faktu, že je v porovnání s Perforce – který je nejpopulárnější naopak v profesionálním prostředí – zcela bezplatný [36][22]. Je tedy vidět, že Git je v herním vývoji velmi relevantní a má smysl zabývat se problémy z kapitoly 3.1.

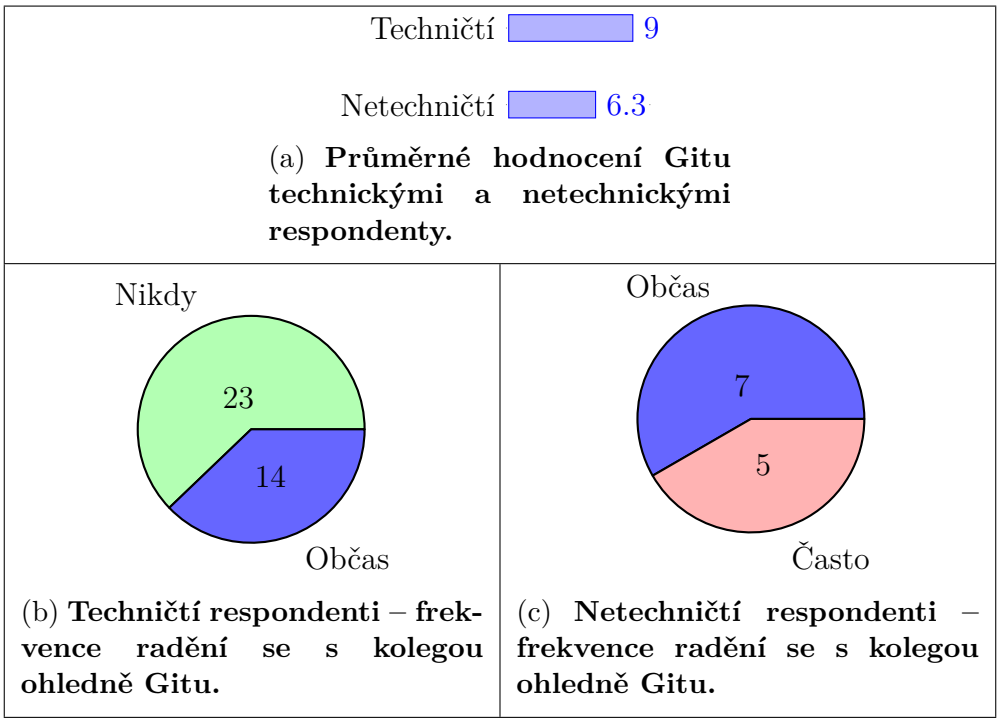
Dále bylo prozkoumáno hodnocení jednotlivých nástrojů, viz obrázek 3.5, kde je hodnocení jednotlivých nástrojů zprůměrováno. Respondent nástroj hodnotil na škále 1–10. Průměrné hodnocení na obrázku 3.5 je třeba brát s rezervou vzhledem k tomu, že se velikost vzorků pro jednotlivé nástroje výrazně liší.



Obrázek 3.5: Průměrné hodnocení systémů správy verzí respondenty.

Respondentovi bylo také umožněno slovně popsat případné nedostatky daného nástroje. U Gitu se například zmínily problémy s verzováním binárních souborů, slučování změn binárních souborů a scén v herním enginu Unity a složitost příkazů pro neprogramátory.

Analýza v rámci druhé kategorie se poté omezila pouze na Git a byla snaha potvrdit hypotézu, že netechnické profese často shledávají Git jako příliš složitý a mají potíže s jeho používáním. Vzorek respondentů klesl – po omezení pouze na ty, kteří používají Git jako hlavní systém správy verzí – na 12 netechnických a 37 technických. Z vizualizací na obrázku 3.6 lze vyčíst, že techničtí respondenti hodnotí Git velice kladně a nepotřebují se často radit ohledně jeho používání. Naopak u netechnických profesí je vidět v hodnocení znatelný propad, což se odráží i v častější nutnosti radit se s kolegou.



(a) **Průměrné hodnocení Gitu technickými a netechnickými respondenty.**

(b) **Techničtí respondenti – frekvence radění se s kolegou ohledně Gitu.**

(c) **Netechničtí respondenti – frekvence radění se s kolegou ohledně Gitu.**

Obrázek 3.6: Porovnání snadnosti používání Gitu mezi technickými a netechnickými profesemi.

Kategorie 3: Verzovací proces.

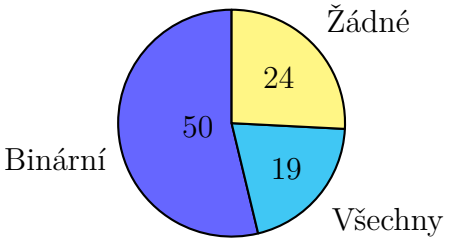
Otázky v rámci třetí kategorie jsou zaměřeny na prvky používaných verzovacích procesů, četnost konfliktů a složitost jejich řešení. Hlavním cílem následné analýzy bylo určit některé požadavky pro návrh verzovacího procesu v kapitole 5. Kategorie obsahovala následující otázky:

- Otázka 3.1: Jsou v rámci Vašeho týmu pevně daná pravidla, která se v rámci verzování dodržují?
- Otázka 3.2: Je umožněno více členům týmu souběžně pracovat nad jedním souborem?
- Otázka 3.2.1: Jak je docíleno toho, že nelze pracovat nad jedním souborem více členy zároveň?
- Otázka 3.3: Přibližně jak často nastávají tzv. konflikty?
- Otázka 3.3.1: Pokud nastane konflikt, máte na starosti jeho řešení?

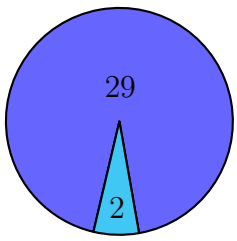
- Otázka 3.3.2: Pokud nastane konflikt, jak obtížné je obvykle jeho řešení?

- Otázka 3.4: Je možné v rámci Vámi používaného verzovacího procesu dlouhodobě udržovat v jednom repozitáři paralelně více verzí hry?

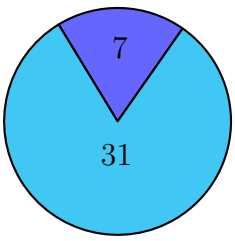
Jelikož systém správy verzí je nástroj, který lze používat mnoha způsoby, byla položena otázka 3.1 za účelem zjištění, zda jsou v rámci týmu pevně daná pravidla pro používání daného systému, která se v rámci týmu dodržují. Celkem 85% ze všech respondentů má tyto pravidla v rámci týmu zavedena. Pokud se navíc respondenti omezí pouze na ty, kteří působí v týmu s více jak 15 členy – což tvoří více jak polovinu respondentů – tak kladně odpovědělo dokonce 100%. Je tedy zřejmé, že mít zavedený pevný proces verzování dodržovaný všemi členy týmu je považováno drtivou většinou týmů za vhodné.



(a) **Typy souborů pro které je zamezena paralelní práce v rámci týmu respondenta.**



(b) **Způsob zamezení práce při použití Gitu.**



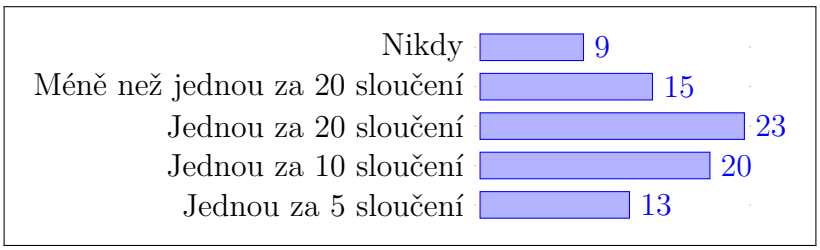
(c) **Způsob zamezení práce při použití ostatních systémů.**

Obrázek 3.7: Zamezení paralelní práce nad souborem v rámci týmů respondentů.

Dále byla analýza odpovědí zaměřena na zamezení paralelní práce nad sou-

bory. Odpovědi jsou vizualizovány v grafech na obrázku 3.7, ze kterých je zřejmé, že si je většina týmů vědoma problému popsaném v kapitole 3.1.3 – tedy složitosti slučování změn binárních souborů – a snaží se takové situaci předejít. Pokud se z dat vynechá Git, lze vidět, že je zamezení paralelní práce většinou dosaženo systémem správy verzí. Situace je naprosto opačná, pokud se data omezí pouze na Git. Zamezení paralelní práce se v tomto případě dosahuje spíše komunikací v týmu a rozšíření Gitu umožňující zamykání se tedy nepoužívá. To lze přičíst například tomu, že zamykací příkazy rozšíření nejsou součástí Gitu jako takového a některá grafická uživatelská rozhraní je nepodporují.

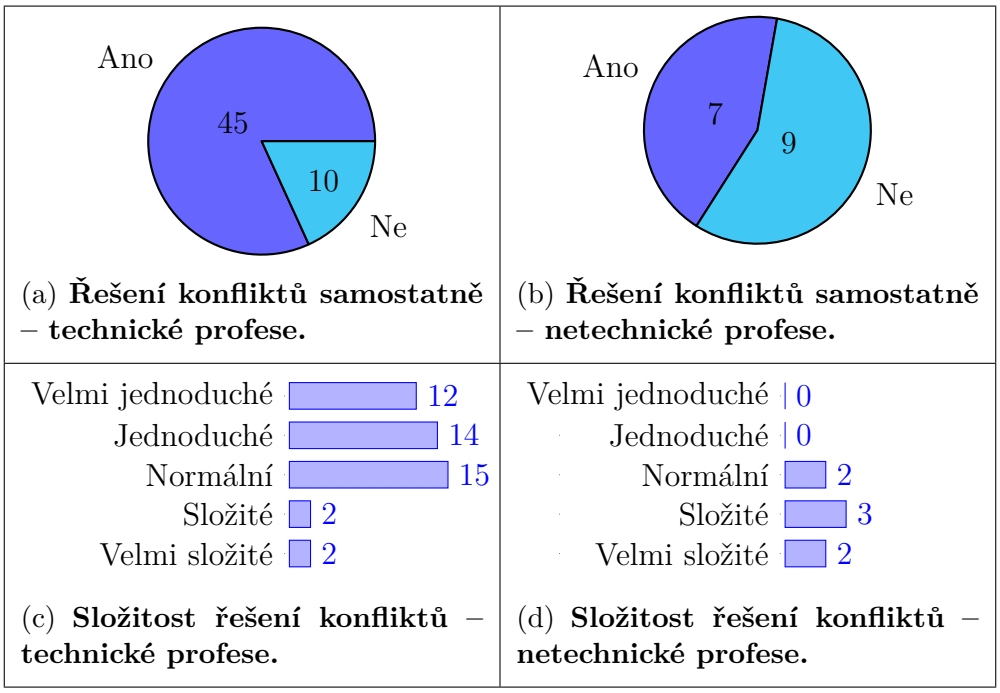
Předposledním tématem v rámci této kategorie byly konflikty. Nejprve byla položena otázka za účelem rámcového průzkumu frekvence konfliktů. Výsledky jsou znázorněny v grafu na obrázku 3.8, kde je vynecháno 13 respondentů, kteří zvolili odpověď "Nevím".



Obrázek 3.8: Přibližná frekvence konkfliktů při slučování větví.

Na obrázku 3.9 je poté vizualizován výstup analýzy způsobu a složitosti řešení konfliktů z pohledu technických a netechnických profesí. Grafy (a) a (b) vizualizují, zda konflikty řeší přímo respondent nebo je má na starosti jiný člen týmu. Na grafech (c) a (d) je poté znázorněna složitost řešení konfliktů pro respondenty, kteří konflikty řeší.

Z prvních dvou grafů lze vyčíst, že v případě, kdy jsou součástí týmu netechnické profese, mohou konflikty přinášet další tření do vývoje tím, že se netechničtí členové často obracejí s řešením na kolegu. Z dalších dvou grafů navíc vyplývá, že netechničtí členové týmu se spíše potýkají se složitějšími konflikty. Poslední tvrzení je ale nutno brát s rezervou vzhledem k malému vzorku dat. Z těchto informací tedy lze usoudit, že je opravdu vhodné vyhýbat se vzniku konfliktů u souborů spravovaných umělci.



(a) **Řešení konfliktů samostatně – technické profese.**

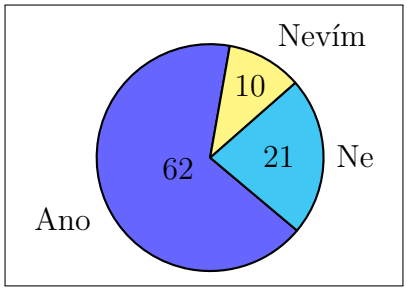
(b) **Řešení konfliktů samostatně – netechnické profese.**

(c) **Složitost řešení konfliktů – technické profese.**

(d) **Složitost řešení konfliktů – netechnické profese.**

Obrázek 3.9: Způsob a složitost řešení konfliktů podle profesí.

Poslední otázka v této kategorii – zda je možné dlouhodobě udržovat v jednom repozitáři paralelně více verzí hry – byla položena čistě za účelem získání požadavků na návrh verzovacího procesu v kapitole 5. Z vizualizace odpovědí na obrázku 3.10 je zřejmé, že je paralelní udržování vícero verzí hry ve verzovacím procesu nutno zohlednit.

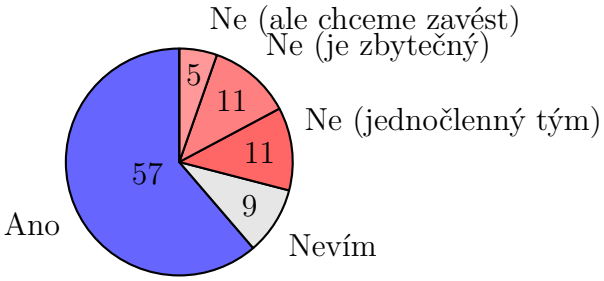


Obrázek 3.10: Paralelní udržování vícero verzí hry v týmech respondentů.

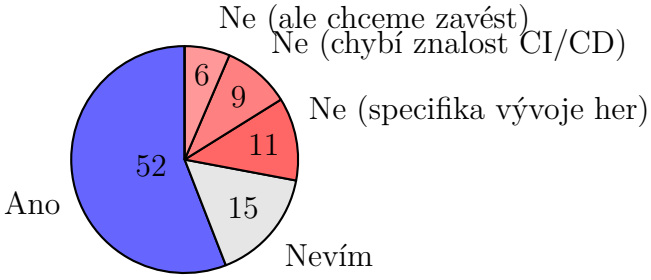
Kategorie 4: Praktiky úzce související s verzováním

Předposlední kategorie se týká seznámení se s užíváním praktik úzce související s verzováním při herním vývoji, jako code review, continuous integration a continous deployment či delivery. Hlavním cílem následné analýzy

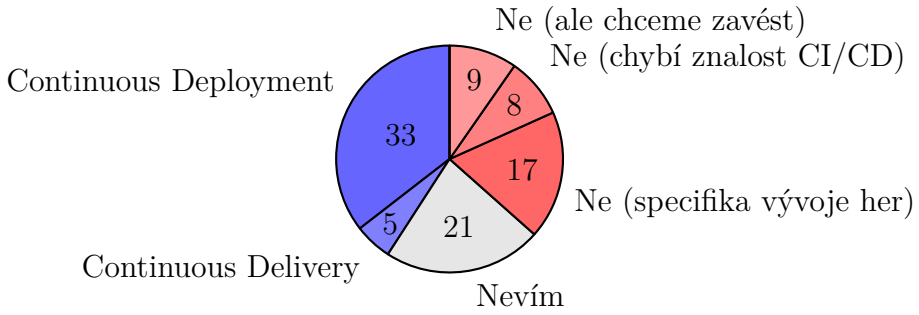
bylo opět určit další požadavky pro návrh verzovacího procesu v kapitole 5.



(a) **Praktikování code review v týmech respondentů.**



(b) **Praktikování Continuous Integration v týmech respondentů.**



(c) **Praktikování Continuous Deployment / Delivery v týmech respondentů.**

Obrázek 3.11: Užívání praktik úzce související s verzováním v týmech respondentů.

Kategorie se sestává z následujících otázek:

- Otázka 4.1: Praktikuje Váš tým tzv. code review
- Otázka 4.1.1: Z jakého důvodu Váš tým nepraktikuje code review?
- Otázka 4.2: Praktikuje Váš tým tzv. Continuous Integration?

- Otázka 4.2.1: Z jakého důvodu Váš tým nepraktikuje Continuous Integration?

- Otázka 4.3: Praktikuje Váš tým tzv. Continuous Deployment / Delivery?

- Otázka 4.3.1: Z jakého důvodu Váš tým nepraktikuje Continuous Deployment / Delivery?

Z grafů na obrázku 3.11 – kde jsou vizualizovány odpovědi na jednotlivé otázky – je zřejmé, že i v herním průmyslu se hojně používají osvědčené praktiky z ostatních prostředí vývoje softwaru a tuto skutečnost je tedy také nutné zohlednit při návrhu procesu v kapitole 5.

Kategorie 5: Otevřené otázky

Poslední kategorií otázek jsou otevřené otázky dávající respondentovi prostor pro vyjádření se nad rámec položených otázek v předchozích kategoriích. Jedná se o následující dvě otázky:

- Otázka 5.1: Potýkáte se / potýkali jste se v rámci Vašeho týmu s některými nedostatky týkající se verzovacího procesu či praktik s ním úzce souvisejícím?

- Otázka 5.2: Chtěl/a byste zmínit některé neobvyklé prvky týkající se verzovacího procesu či praktik s verzováním úzce souvisejícím, které v rámci Vašeho týmu používáte a osvědčily se vám?

Všechny odpovědi lze nalézt v již zmíněném souboru `Vysledky/pruzkum_odpovedi.csv`. Odpovědi lze shrnout do následujících bodů:

- Git je složitý pro umělce a designéry, což se projevuje například tím, že nechtějí používat větvení.

- Git není vhodný pro verzování binárních souborů.

- Byly zmíněny i některé problémy specifické pro jednotlivé herní enginy. Ohledně Unity bylo například zmíněno, že ve verzovacích softwarech nelze číst herní scény. Pro Unreal Engine bylo zmíněna situace, kdy se binárně změní soubor, aniž by se fakticky změnil obsah daného souboru, což vyžaduje buď pečlivé commitování nebo se musí smířit s

tím, že se bude zbytečně zvětšovat repozitář nebo dodatečné úložiště v případě použití Git LFS.

- Byly zmíněny i některé specifické osvědčené praktiky ohledně používání Gitu týkající se commitů. Konkrétně, že by commity měly být co nejmenší a s rozumnými popisy.

Kromě dalšího potvrzení hypotézy, že je Git pro netechnické profese těžko uchopitelný, se tedy zmínily i další problémy a tipy týkající se verzování ve vývoji jak obecném, tak herním.

4 Procesy verzování

Systémy správy verzí jsou nástroje usnadňující vývoj softwaru v mnoha aspektech. Samotné zavedení jejich používání ale nemusí být dostatečné vzhledem k spoustě možností jejich používání. Ve většině případů je tedy v rámci týmu nutné zavést proces definující pravidla používání daného nástroje, které jasně vedou uživatele, jak daný nástroj používat. U Gitu to platí dvojnásob vzhledem k jeho distribuované povaze.

Tato kapitola je nejdříve zaměřena na procesy synchronizace repozitářů týkající se distribuované povahy Gitu. Následně se zaměřuje na konkrétní prvky verzovacích procesů a praktik s verzováním úzce souvisejících.

Kapitola je pojata spíše z pohledu obecného softwarového vývoje, ale značně z ní čerpá následující kapitola 5 týkající se návrhu procesu verzování pro vývoj videoher, kde jsou již specifika vývoje her vzata v potaz.

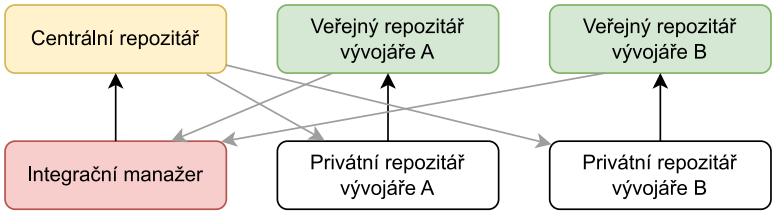
4.1 Synchronizace repozitářů

V centralizovaných systémech správy verzí představují všichni vývojáři rovnocenné uzly pracující nad centrálním úložištěm. Vzhledem k distribuované povaze je ale Git mnohem flexibilnější v možnostech, jak mohou vývojáři kolaborovat nad projektem. V Gitu totiž může jeden uživatel představovat jak uzel, tak úložiště pro jiné uzly. To znamená, že každý vývojář může přispívat kódem do ostatních repozitářů a současně může spravovat veřejný repozitář, z kterého mohou ostatní vycházet při své práci a přispívat do něj. Této vlastnosti lze využít pro různorodou správu projektu. Dále jsou představeny některá populární paradigmata.

4.1.1 Integrační manažer

Základním principem paradigmatu integrační manažer je existence zvláštního veřejného repozitáře pro každého vývojáře a jednoho centrálního veřejného repozitáře, který je spravován správcem projektu [36]. Každý vývojář si tedy – před započetím práce na projektu – musí vytvořit vlastní veřejnou kopii centrálního repozitáře, ve které provádí veškerou práci. Po dokončení práce uživatel posílá správci požadavek, aby změny v uživatelově repozitáři

zakomponoval do centrálního repozitáře. Princip je znázorněn na obrázku 4.1.

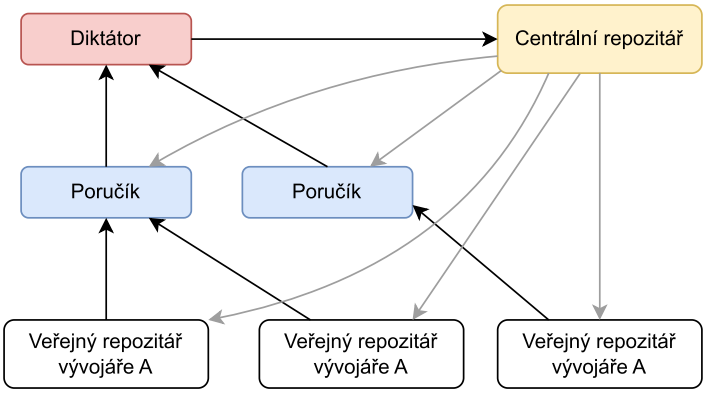


Obrázek 4.1: Princip paradigmatu integrační manažer.

Jednou z – v některých případech žádanou – vlastností je, že ne všichni mají právo na manipulaci s centrálním repozitářem, což je důvod, proč je toto paradigma primárně používané v open-source projektech [1]. Lze jej ale aplikovat i privátní projekty, pokud je potřeba autoritativnější kontrola nad repozitářem.

4.1.2 Diktátor a poručíci

Paradigma diktátor a poručíci lze prakticky považovat za rozšíření integračního manažera. Je vhodnější pro větší týmy, kde mohou být jednotlivé moduly nebo části kódu přiděleny tzv. poručíkovi, který je za danou sekci zodpovědný [36]. Když poručíci uznají za vhodné, dají změny k dispozici další roli tohoto paradigmatu – tzv. diktátorovi. Diktátor má poté podobnou roli, jako integrační manažer z předchozího paradigmatu – má jako jediný přístup k zápisu do centrálního repozitáře a zakomponovává do něj změny zprostředkované poručíky. Všichni členové se pravidelně s centrálním repozitářem synchronizují. Princip je znázorněn na obrázku 4.2.



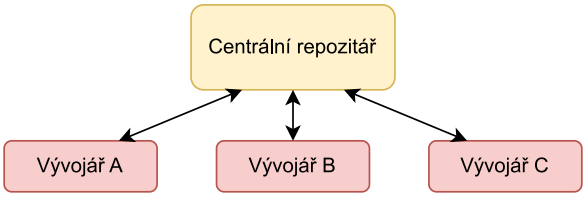
Obrázek 4.2: Princip paradigmatu diktátor a poručíci.

Toto paradigma není běžně používáno, ale může být užitečné ve velkých

projektech s hierarchickým prostředím [36]. Umožňuje totiž vedoucímu projektu – tedy diktátorovi – delegovat práci a shromažďovat velké podmnožiny kódu, než je integruje. Jedním z projektů, které toto paradigma používají, je Linuxové jádro [55].

4.1.3 Centralizované paradigma

V centralizovaném paradigmatu existuje pouze jeden veřejný repozitář, který je prohlášen za centrální [36]. Všechny ostatní repozitáře v rámci projektu jsou poté pouze privátní a nachází se na stroji vývojáře. Každý vývojář tedy nejdříve musí vytvořit lokální kopii daného centrálního repozitáře. Typický postup práce na projektu, se zavedeným centralizovaným paradigmatem, se poté skládá ze tří kroků – synchronizace lokálního repozitáře s centrálním repozitářem, provedení změn v rámci lokálního repozitáře a následná další synchronizace repozitářů. Princip je znázorněn na obrázku 4.3.



Obrázek 4.3: Princip centralizovaného paradigmatu.

V rámci tohoto paradigmatu jsou tedy – z pohledu systému správy verzí – všichni vývojáři stejně důležití. Může se tedy zdát, že je toto paradigma vhodné pouze pro malé týmy, ale se správně zvoleným modelem větvení – viz další sekce v rámci této kapitoly – se i toto paradigma může dobře škálovat s velikostí týmu.

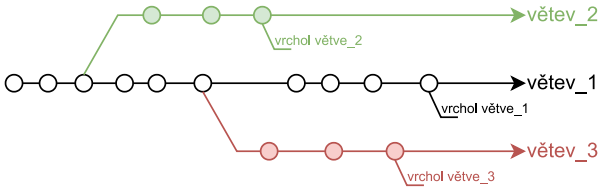
4.2 Základní prvky verzovacích procesů

4.2.1 Větvení

V některých případech je vhodné nebo dokonce i nutné mít v rámci jednoho projektu několik paralelních vývojových linií [32]. A právě tento problém lze řešit tzv. větvením, které je podporováno téměř všemi systémy správy verzí [36].

Větev lze obecně definovat jako konkrétní sekvenci commitů, kde poslední commit je označován jako vrchol větve nebo – v anglické literatuře – jako

head či tip [40]. Větvením je poté označován proces vytváření těchto větví, což si lze představit jako rozdělení jedné větve na dvě. Pokud je tedy například v rámci vývoje zavedena hlavní linie vývoje – viz následující sekce 4.2.2 – tak si každý vývojář může vytvořit větev vycházející právě z hlavní vývojové linie a pracovat nad touto novou větví, čímž se zajistí, že daná práce ovlivňuje pouze novou větev a nikoliv hlavní vývojovou větev. Principu větvení je naznačen na obrázku 4.4.



Obrázek 4.4: Princip větvení.

Jak již bylo řečeno, větvení je podporováno většinou systémů správy verzí, ale jednotlivé implementace větvení se liší. V mnoha implementacích je větvení těžkopádný proces vyžadující vytvoření nové kopie adresáře, což může být u velkých projektů problém [36]. Při návrhu Gitu se ale k větvení přistoupilo naprosto jinak a operace s větvemi jsou velmi rychlé vzhledem k tomu, že větev je pouhý ukazatel na commit, viz popis vnitřní reprezentace Gitu v kapitole 2.3.1.

Vzhledem distribuované povaze Gitu si je také nutno uvědomit, že uživatelé pracují nad lokálními repozitáři, což je ve většině případů klon nějakého vzdáleného repozitáře. Naklonováním automaticky vzniknou kopie větví vzdáleného repozitáře, na které lze nahlížet i tak, že nemají s původními větvemi vzdáleného repozitáře nic společného. V Gitu ale existuje koncept zvaný *remote-tracking* větev.

Remote-tracking větve jsou reference na stav větví vzdáleného repozitáře [36]. Jsou to lokální reference, které nelze manuálně měnit. Mění je totiž automaticky Git v případě síťové komunikace, aby bylo zajištěno, že přesně reprezentují stav vzdáleného repozitáře. Remote-tracking větve jsou pojmenovány podle vzoru `<vzdálený_repozitář>/<větev>`. Tedy například pro zjištění stavu při poslední síťové komunikaci větve `master` na vzdáleném repozitáři pojmenovaném `origin` se stačí podívat na větev `origin/master`.

I přesto, že je větvení v mnoha případech při správném použití opodstatněné, tak je vždy vhodné před jeho zavedením do procesu verzování řádně zvážit, zda je v daném případě opravdu nutné. S větvením se totiž pojí i zásadní

úskalí, kterým se do hloubky věnují další sekce v rámci této kapitoly, zejména sekce 4.3.

4.2.2 Hlavní linie vývoje

Hlavní linie vývoje je větev představující – v určitém smyslu – současný stav projektu [32]. Vývojáři před zahájením nové práce vycházejí z této větve a pokud je daná práce hotová a má se zpřístupnit všem členům, tak se naopak zakomponuje zpět do hlavní linie vývoje.

Pro tuto větev existují různá označení – například `master`, `main` nebo `trunk` – které se většinou odvíjí od konvence v rámci daného systému správy verzí [40].

Je nutné si uvědomit, že hlavní linie vývoje je pouze jedna sdílená větev. V případě Gitu tedy neplatí, že větev `master` – pokud se takto pojmenuje hlavní line vývoje – je vždy hlavní linií vývoje. Pokud se v rámci týmu používá například centralizované paradigma, tak hlavní vývojovou větví je pouze větev `master` v centrálním repozitáři. Ostatní členové týmu mají na svém stroji pouze klon repozitáře a před započetím nové práce tedy musí svou `master` větev synchronizovat s `master` větví v centrálním repozitáři.

Vzhledem k tomu, že je tato větev středobodem vývoje, je vhodné udržovat ji ve stabilním stavu. Definici „stabilního stavu“ si může každý tým zvolit sám, ale obecně lze říci, že by daná větev měla splňovat několik obecných podmínek [40]. Tou nejzákladnější je, že by z dané větve mělo být vždy možno bez problému software sestavit. Dále by měly artefakty projektu – například kód – obsahovat co nejméně chyb a měly by splňovat určité požadavky, co se kvality týče. Drtivá většina kontrol těchto podmínek by měla být ideálně automatizována a prováděna ve vhodné frekvenci.

Kontrolu sestavení finálního softwaru lze snadno automatizovat zavedením tzv. build serveru. Tento server pravidelně provádí sestavení softwaru a informuje o výsledcích [17]. Sestavení by se mělo ideálně provádět pro každý nový commit v hlavní linii vývoje. Pokud samotné sestavení trvá příliš mnoho času – a sestavením pro každý commit by hrozilo například zahlcení serveru – lze zavést i jiná pravidla pro frekvenci sestavování. Pravidla by ale měly obecně být nastavena tak, aby se nový commit stal součástí sestavení pokud možno co nejdříve, v nejhorším případě den po zakomponování daného commitu do hlavní linie vývoje.

Další obecnou podmínku – tedy obsažení co nejméně chyb v artefaktech projektu – lze automatizovat důkladným psaní automatických testů [42]. To v praxi znamená, že se společně s kódem píše i sada automatických testů, které do určité míry zajišťují bezchybnost produkčního kódu. Tyto testy lze poté spouštět opět na build serveru po každém sestavení.

Pokud sestavení nebo testy selžou, tak by vrácení se do stabilního stavu mělo mít tu nejvyšší prioritu. Lze dočasně i „zamrazit“ hlavní vývojovou linii – tedy zakázat přidávání dalších commitů kromě těch, které řeší problémy se stabilitou větve.

Poslední obecnou podmínkou je splnění požadavků na kvalitu artefaktů. Zajištění této podmínky lze také do jisté míry automatizovat například zakomponováním nástroje *SonarQube* do procesu, což je platforma pro kontinuální kontrolu kvality kódu provádějící statickou analýzu [29]. Další populární metodou je tzv. předintegrační revize, což je manuální kontrola výsledné podoby artefaktu dalším členem týmu [40]. O předintegrační revizi a alternativách detailněji pojednává sekce 4.3.4.

4.3 Integrace změn

V sekci 4.2.1 byl představen koncept větvení. Při správném použití může větvení přispět k udržitelnějšímu a přehlednějšímu procesu verzování, ale nese s sebou i potenciální úskalí. Větvení totiž může přispívat k zvětšování „vzdálenosti“ mezi vývojáři, což přináší několik rizik vzhledem k tomu, že se větvení ve většině případů úzce pojí i následným slučováním větví. A frekvence těchto slučování – neboli frekvence integrací – hraje v procesu verzování významnou roli.

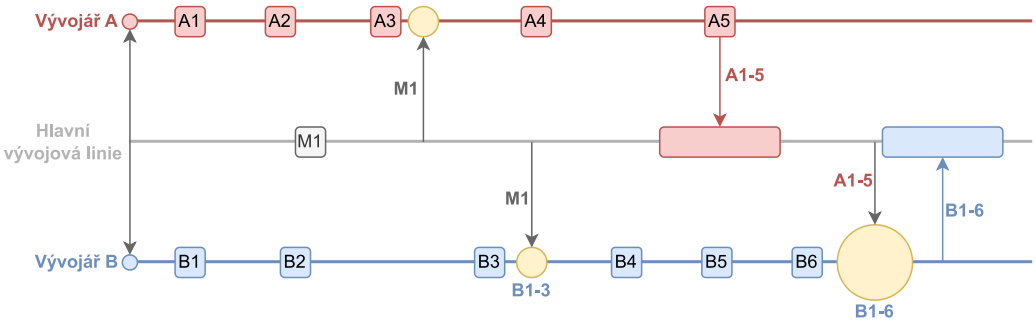
4.3.1 Frekvence integrací

Než budou představeny jednotlivé techniky pro integrace změn, je vhodné zamyslet se nad jednou důležitou vlastností verzovacího procesu týkající se právě integrace změn – a to nad frekvencí těchto integrací. Z průzkumů totiž vyplývá existence korelace mezi frekvencí integrací a úspěšností týmu [39]. Příčina je nastíněna v dalších odstavcích.

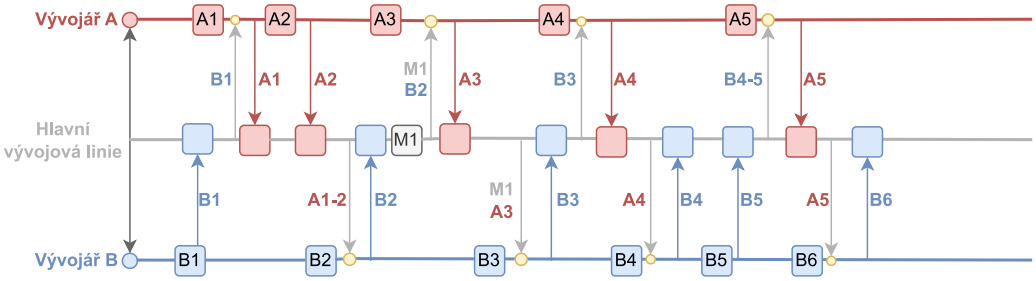
Na obrázcích 4.5 a 4.6 je znázorněn rozdíl nižší a vyšší frekvence integrace mezi vývojáři a hlavní vývojovou linií. V obrázcích je naznačena privátní vývojová linie jednotlivých vývojářů, hlavní vývojová linie a synchronizace

mezi jednotlivými vývojovými liniemi. Dále je žlutě naznačena „velikost“ jednotlivých integrací.

Na první pohled si lze všimnout dvou výrazných rozdílů. První – jak už název napovídá – je samotná frekvence integrací. Důležitější rozdíl ale je v již zmíněné „velikosti“ jednotlivých integrací. Menší integrace zpravidla znamenají menší šanci vzniku velkých konfliktů, s kterými se pojí několik problémů. Prvním problémem je samozřejmě objem práce – čím větší konflikt, tím zpravidla více práce. Jako důležitější problém lze ale označit vyšší riziko související s dobou odhalení konfliktu.



Obrázek 4.5: Integrace s nižší frekvencí.



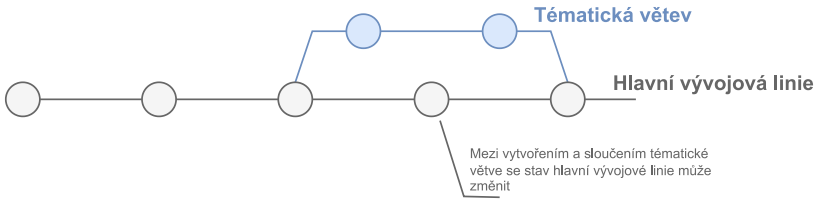
Obrázek 4.6: Integrace s vyšší frekvencí.

Pokud by například potenciální konflikt vznikl v prvních commitech jednotlivých vývojářů, tak v případě integrace s nižší frekvencí by se na něj ve skutečnosti přišlo až při poslední integraci znázorněné na konci diagramu, protože se jedná o první integraci, kde jsou tyto commity sloučeny. V případě vyšší frekvence by se tento konflikt odhalil již při prvním sloučení na začátku diagramu.

Častá integrace tedy zvyšuje frekvenci slučování, ale snižuje jejich složitost a zrychluje jejich odhalení, což má za následek menší frekvenci výskytu velkých, špatně řešitelných konfliktů. V horším případě až konfliktů neřešitelných.

4.3.2 Tématické větve

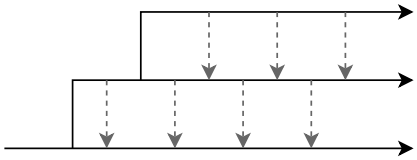
Tématické větve – častěji označováno anglickým pojmem feature branches nebo topic branches – jsou dočasné větve, v rámci nichž se implementuje jedna ohraničená funkcionalita [36]. Před započetím práce na dané funkcionalitě si tedy vývojář vytvoří větev s vhodným pojmenováním, vycházející z hlavní vývojové linie. V nové větvi danou funkcionalitu implementuje a po skončení práce se tato větev sloučí zpět do hlavní vývojové linie. Princip tématických větví je znázorněn na obrázku 4.7.



Obrázek 4.7: Tématická větev.

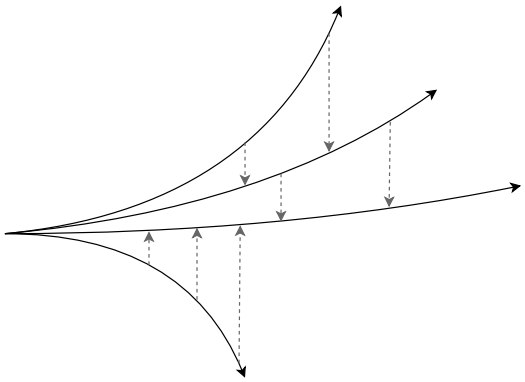
Způsob sloučení do hlavní vývojové linie závisí na zavedeném procesu. Může jej provádět například přímo autor dané funkcionality nebo lze pověřit jiného člena týmu zároveň s předintegrační revizí, viz sekce 4.3.4.

Tématické větve jsou v praxi hojně využívané [40], ale pokud jsou jednotlivé funkcionality příliš rozsáhlé, pojí se s nimi rizika týkající se nízké frekvence integrace, viz předešlá kapitola 4.3.1. Na tento problém lze nahlížet i z takového úhlu pohledu, kde se v průběhu času zvětšuje vzdálenost mezi jednotlivými větvemi. Častý způsob vizualizace větvení je pomocí paralelních čar, viz obrázek 4.8. Na obrázku 4.9 je ale v tomto případě vizualizace vhodnější, jelikož je z ní patrná zvětšující se vzdálenost mezi jednotlivými větvemi v průběhu času.



Obrázek 4.8: Klasický diagram větvení.

I přes zmíněná úskalí ale existují určité kontexty, v rámci nichž je použití tématických větví opodstatněné. Jedním z takových kontextů je vývoj open-source projektů, kde je za projekt zodpovědný správce projektu, což může být jeden člověk nebo menší tým [21]. Správce projektu spolupracuje s větší skupinou vývojářů, kteří jsou většinou do jisté míry považováni za nedůvěryhodné. Správce si tedy nemůže být jistý, jakou kvalitou jejich výstupy



Obrázek 4.9: Diagram větvení znázorňující zvětšující se vzdálenost.

disponují a také existuje nejistota v tom, jak často se budou přispěvatelé dané funkcionalitě věnovat. Pokud tedy přispěvatel plánuje přidat funkcionalitu – ať už s menším nebo větším rozsahem – tak správce nemá jistotu, kdy bude dokončena. V tomto kontextu tedy dává smysl počkat na dokončení kompletní funkcionality. Je také nutno klást větší důraz na ověření kvality výstupů přispěvatelů, viz sekce 4.3.4.

V klasických – například komerčních – softwarových týmech je ale situace naprosto jiná. Tým se většinou skládá z členů, kteří se danému projektu věnují kontinuálně – například na plný úvazek – a lze tedy očekávat, že se budou dané funkcionalitě věnovat pravidelně a výstupy budou odpovídat zavedeným standardům v rámci týmu [21]. Důraz na ověření kvality tedy nemusí být až tak vysoký, jako v případě open-source projektů.

4.3.3 Kontinuální integrace

Tématické větve definují spodní hranici pro množinu změn, která se integruje do hlavní vývojové linie – tedy jasně ohraničené funkcionality. A s tím se pojí určitá úskalí týkající se frekvence integrace, jak bylo řečeno v předchozích sekcích. Alternativou, která se tyto úskalí snaží limitovat, je tzv. kontinuální integrace.

Kontinuální integrace je praktika podporující velmi častou integraci [42]. Při aplikování této praktiky by se mělo integrovat kdykoliv, kdy se udělá určitý pokrok v implementaci funkcionality vhodný k integraci, který ale neporušuje podmínky stability hlavní linie vývoje. Neočekává se tedy, že se budou integrovat pouze kompletní funkcionality, ale naopak by se mělo integrovat co nejčastěji. Frekvence integrace samozřejmě záleží na povaze funkcionality, ale obecným pravidlem je, že by jedna integrace neměla obsahovat

více, jak jednodenní práci. Vhodnější ale je integrovat mnohem častěji. Jak bylo řečeno, frekvence integrace záleží na povaze funkcionality a k obecnému pravidlu samozřejmě existují výjimky, kdy nelze za jeden den vyprodukovat ucelený kus práce, který by byl vhodný k integraci. Příkladem může být práce umělců na složitém grafickém artefaktu.

To, že se integrují i neúplné funkcionality, má několik důsledků, s kterými se tým musí vypořádat. Je totiž nutno zvážit, jak docílit toho, aby se do systému daná část funkcionality integrovala, aniž by se projevovala navenek a nenarušovala stabilitu hlavní vývojové linie. Toho se dá v mnoha případech snadno docílit – pokud se například implementuje algoritmus pro novou funkcionalitu, který se bude volat až při zkompletování celé funkcionality, tak není třeba nic řešit. Existují ale případy, kdy částečnou funkcionalitu nelze jednoduše schovat a je třeba tuto situaci řešit jiným způsobem.

Jednou z pokročilejších technik pro schování funkcionality jsou tzv. *feature toggles*, což je sada praktik umožňující modifikování chování systému bez změny kódu [47]. Mimo schování částečné funkcionality lze tuto techniku použít například i pro selektivní zpřístupnění již hotové funkcionality podskupině uživatelů pro účely testování. Existuje několik technik implementace této techniky, kde jedna z nich je nastíněna ve fragmentu kódu 4.1.

```
function provedVypocet() {
  if (featureIsEnabled("pouzij-optimalizovany-vypocet")) {
    return provedOptimalizovanyVypocet();
  } else {
    return provedZakladniVypocet();
  }
}
```

Fragment kódu 4.1: Princip techniky feature toggles.

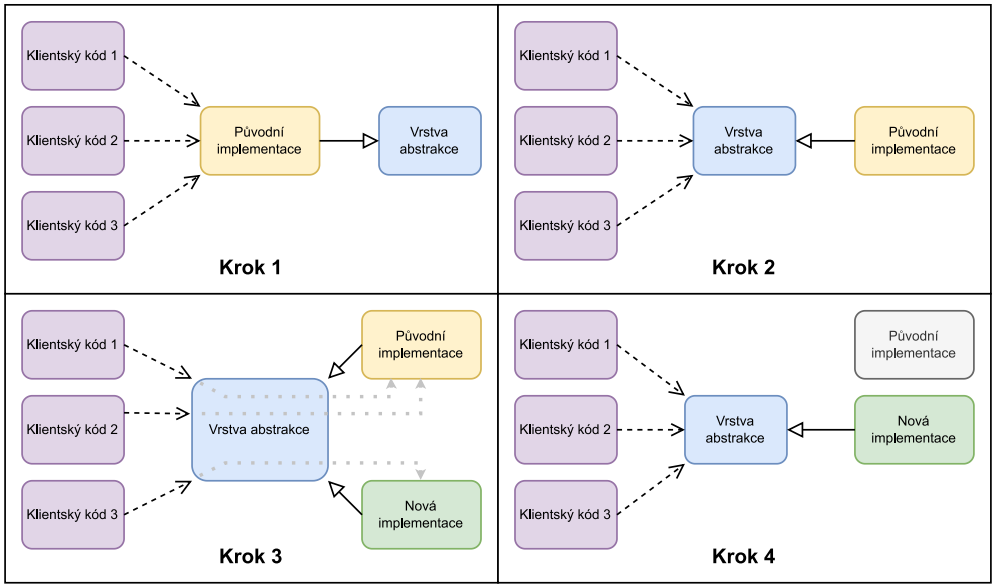
Další technikou je tzv. *branch by abstraction*, což je technika vhodná k postupné integraci větší změny, v rámci níž se nahrazuje jedna funkcionalita za druhou [45]. Techniku lze shrnout do několika kroků, které jsou znázorněny na obrázku 4.10. Jedná se konkrétně o následující kroky:

1. Zavedení abstrakce kolem komponent, které mají být nahrazeny.

2. Postupná integrace této abstrakce do všech částí kódu, kde je tato funkcionalita používána.

3. Započetí práce na nové implementaci odpovídající stejné vrstvě abstrakce z bodu 1 a postupné nahrazování původní implementace novou implementací.

4. Kompletní nahrazení původní implementace novou implementací a odstranění původní implementace.



Obrázek 4.10: Princip techniky branch by abstraction.

Zřejmou potenciální nevýhodou kontinuální integrace jsou tedy vyšší nároky na tým. Je totiž nutno jak praktikovat vyšší frekvence integrací, tak stále udržovat hlavní vývojovou linii ve stabilním stavu. Od technických členů týmu je tedy navíc vyžadována, v porovnání s tématickými větvemi, znalost dodatečných technik – jako feature toggles a branch by abstraction – a pečlivé testování.

Je tudíž vhodné důkladně zvážit, zda je tým schopen kontinuální integraci praktikovat, případně zda investovat do školení členů týmu. Z výzkumů ale vychází, že existuje korelace mezi praktikováním kontinuální integrace a obchodními metrikami, jako je návratnost investic a ziskovost [39].

4.3.4 Předintegrační revize

Předintegrační revize je jednou z metod praktikování tzv. *code review*. Cílem je zajištění toho, že v hlavní linii vývoje budou pokud možno pouze artefakty s kvalitou odpovídající obecným standardům a standardům zavedených v

rámci týmu [35]. Předintegrační revize je manuální mezikrok mezi dokončením práce a jejím integrováním do hlavní vývojové linie. Po dokončení práce se tedy nejdříve požádá o její revizi jiným členem týmu, který může buď dané změny schválit nebo poskytnout zpětnou vazbu s případnými výtkami. V případě výtek se proces opakuje, dokud nejsou obě strany – přispěvatel a posuzovatel – spokojeni. Teprve pak se daný kus práce integruje do hlavní vývojové linie.

Tato technika je hojně používána jak v open-source, tak v komerčním prostředí a v mnoha případech je její použití opodstatněno [35]. Nutnost zmíněného manuálního mezikroku ale do procesu zanáší tzv. integrační tření. K integračnímu tření přispívají aktivity, které vyžadují čas nebo úsilí mezi dokončením práce a integrováním do hlavní vývojové linie, což pro předintegrační revizi platit může [40]. Tření při předintegrační revizi vzniká na dvou úrovních – první je doba začátku zpracovávání revize od vzniku požadavku na revizi, druhá je doba nutná k provedení samotné revize. Čím znatelnější toto integrační tření je, tím méně jsou vývojáři motivování k časté integraci a inklinují spíše k používání tématických větví. I při používání tématických větví je toto tření samozřejmě vhodné minimalizovat, ale není tak kritické, jako v případě kontinuální integrace.

Při praktikování předintegrační revize je tedy vhodné toto tření pokud možno co nejvíce minimalizovat na obou úrovních. Na první úrovni – tedy doby začátku zpracovávání revize od vzniku požadavku – lze tření minimalizovat přiřazením nejvyšší priority těmto revizím mezi všemi úkoly v rámci týmu. Poté je ale od posuzovatelů vyžadováno časté přepínání kontextů mezi právě rozpracovanou prací a prováděním revizí, což může být vyčerpávající a neefektivní, nebo je třeba čekat na dokončení právě rozpracované práce jednoho z možných posuzovatelů, který se dané revize po dokončení práce ujme.

Minimalizace tření na druhé úrovni – doby provádění revize – závisí na kontextu, ale obecně jej do určité míry minimalizovat lze [14]. Prvním způsobem je určení metodiky provádění daných revizí a poučit s ní všechny posuzovatele. Dále je vhodné, pokud se dané revizi věnuje ten člen týmu, který má k dané oblasti projektu nejblíže a nejrychleji se tedy zorientuje. Minimalizace tření na druhé úrovni dále také probíhá přirozeně – pokud se revize provádějí pravidelně, tak si posuzovatelé postupně na provádění revizí zvykají a jsou efektivnější. S přirozenou minimalizací také souvisí postupně zkvalitňující se artefakty od přispěvatelů z pohledu zavedených standardů kvality, díky poučování se z konstruktivních poznámek v rámci předchozích revizí.

I přes aplikování popsaných minimalizací však může integrační tření v rámci předintegrační revize stále narušovat efektivitu vývoje do takové míry, kdy může být vhodné praktikovat spíše alternativy k předintegrační revizi, kterým se tato sekce dále věnuje.

Alternativa: Pointegrační revize

Jednou z alternativ je požádání o revizi až po integraci práce do hlavní linie vývoje. Případné připomínky od posuzovatele se zapracovávají stejným způsobem, jako původní změna – zapracovaná připomínka se tedy ihned integruje a následně je požádáno o revizi.

Tímto způsobem je tedy integrační tření z pohledu přispěvatele minimalizováno vzhledem k tomu, že všichni členové týmu mohou na dané změně stavět ihned po jejím dokončení, což s sebou ale samozřejmě nese riziko, že daná změna bude z pohledu posuzovatele koncepčně špatně a bude vyžadováno její přepracování. Frekvenci nastávání této situace lze minimalizovat – jako v případě předintegrační revize – prioritizací provádění revizí, aby byla doba od původní problematické integrace a integrace zapracování připomínek co nejkratší.

Tuto metodu je tedy vhodné používat v prostředí, kde jsou v rámci týmu již zkušenější členové a je v ně vkládána dostatečná důvěra.

Alternativa: Párové programování

Další alternativou je tzv. párové programování. Jedna se o metodu, kdy nad danou funkcionalitou v daném čase nepracuje jeden člen týmu, ale skupina dvou členů zároveň [44]. Oba členové paralelně kolaborují nad danou funkcionalitou a typicky se střídají v rolích implementátora a posuzovatele. Revize tedy probíhá kontinuálně, paralelně s vývojem.

Prvotní reakce na alokování času dvou členů na jeden úkol může být, že to je plýtvání lidských prostředků. Ale v některých případech může být opak pravdou vzhledem k tomu, že některé úlohy vyžadují spíše přemýšlení nad řešením daného problému, což může být v mnoha případech mnohem efektivnější v páru [44]. Navíc – jak již bylo řečeno – se zároveň s realizací provádí revize implementace.

Párové programování může tedy být – při správném praktikování – efektivní praktikou realizace funkcionality a jejich revizí. Jedná se ale o dovednost,

která se při špatném praktikování může stát naopak velmi neefektivní [44]. Je také nutno zmínit, že se s párovým programováním úzce pojí sociální aspekt a někteří členové se mohou stát méně efektivní při takto intenzivním sociálním kontaktu.

Zhodnocení metod revizí

Bylo představeno několik metod revizí, jejich výhody a rizika. Volba konkrétní metody závisí na několika faktorech, jako je složení týmu, důvěra v schopnosti jednotlivých členů a alokace času členů pro daný projekt. Tyto faktory se mohou v průběhu času samozřejmě měnit. Lze tedy například začít s praktikováním předintegrační revize, a po získání dostatků zkušeností a důvěry v jednotlivé členy přejít na pointegrační revizi.

4.4 Vydání produkční verze

Doposud se tato kapitola zabývala spíše prací nad hlavní vývojovou linií, konkrétně integrací změn a udržování dané větve ve stabilním stavu. V této sekci jsou dále představeny a zhodnoceny metody týkající se sestavení produkční verze softwaru vycházející z hlavní vývojové linie.

4.4.1 Přímé vydání z hlavní vývojové linie

Nejjednodušší metodou – čistě z pohledu komplexnosti procesu verzování – je vydávání produkční verze přímo z hlavní vývojové linie.

Vydání v tomto případě neznamená nic jiného, než sestavení výsledného software z aktuálního commitu hlavní vývojové linie a prohlášení tohoto sestavení za produkční verzi [40]. S vydáním verze se také často pojí i přidání tzv. *tagu* ke commitu, ze kterého se daná verze vydala, což je pouze jakési smysluplné označení daného commitu.

Tato metoda je z pohledu verzovacího procesu velmi jednoduchá, ale je samozřejmě požadováno důrazného kladení důrazu na kontinuální udržování hlavní vývojové linie ve stavu, ze kterého je možno produkční verzi produktu sestavit. To lze považovat jak za nevýhodu, tak za výhodu. Udržování hlavní linie vývoje v takovémto stavu totiž podporuje disciplínu jednotlivých vývojářů a důsledkem tedy může být ještě stabilnější a kvalitnější stav produktu [41].

Je třeba brát v potaz, že v případě zajištění stabilního stavu v rámci všech commitů v hlavní vývojové linii nemusí vždy platit, že by každý tento commit měl automaticky znamenat vydání nové produkční verze. S touto problematikou se pojí dvě praktiky – *continuous delivery* a *continuous deployment*. Při praktikování continuous deployment je s každým novým commitem, přijatým do hlavní vývojové linie, vydána nová produkční verze, zatímco při praktikování continuous delivery je před vydáním produkční verze třeba provést manuální krok ve formě rozhodnutí, zda z tohoto commitu produkční verzi vydat [41]. Lze tedy říci, že continuous deployment je podmnožinou continuous delivery.

4.4.2 Úrovně vyzrálosti

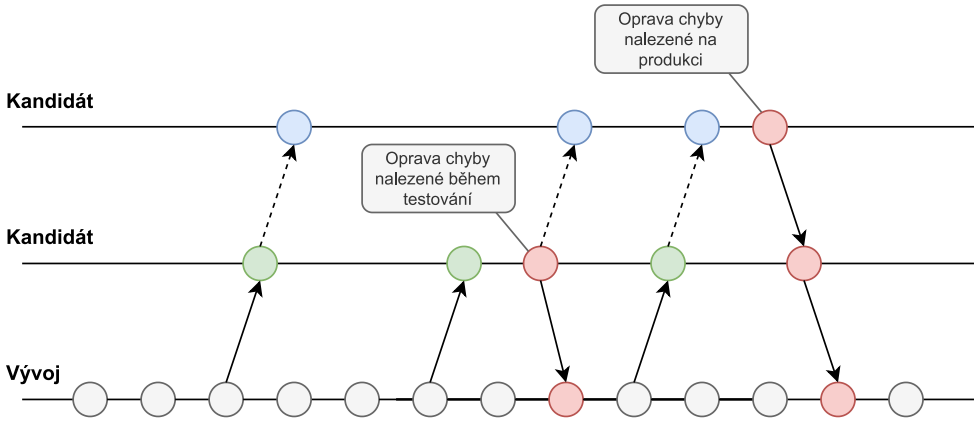
Další metoda je založena na několika několika paralelně existujících větví představujících jednotlivé úrovně vyzrálosti. Vyzrálost změny je definována tím, v jaké větvi se právě nachází. Počet úrovní této vyzrálosti si může každý tým zvolit sám tak, jak uzná za vhodné. Konkrétním příkladem může být realizace znázorněná na obrázku 4.11, kde jsou použity celkem tři úrovně – *vývoj*, *kandidát* a *produkce* – s následujícími významy [46]:

- **Vývoj**: Tato větev představuje hlavní linii vývoje, v rámci níž by se měla ideálně praktikovat kontinuální integrace. Pokud se používají feature toggles, měla by verze vycházející z této větve být testována s takovou konfigurací, kde jsou dané funkcionality povolené.

- **Kandidát**: V případě usouzení, že současný stav hlavní vývojové větve je vhodný k vydání nové verze softwaru, se hlavní vývojová větev sloučí s větví kandidát. V rámci této větve se poté provádí předprodukční testování se správně nastavenými feature toggles. Může totiž nastat situace, kdy přesná konfigurace feature toggles potřebná pro současnou verzi nebyla v rámci testování nad hlavní vývojovou linií otestována. V případě nalezení chyby se tato chyba opraví přímo do větve kandidát a následně se sloučí zpět s hlavní vývojovou linií.

- **Produkce**: Je-li větev kandidát stabilizována, lze vydat novou produkční verzi sloučením větve kandidát do větve produkce a následným provedením sestavení z této větve. Lze se tedy spolehnout, že větev produkce vždy reflektuje současný produkční stav. Chyby nalezené v produkční verzi se opravují přímo do větve produkce a následně se provádí slučování s větvemi představující nižší úroveň vyzrálosti – tedy

nejdříve větev produkce s větví kandidát a následně větev kandidát s větví vývoj.



Obrázek 4.11: Příklad realizace úrovní vyzrálosti.

Větve vyzrálosti tedy umožňují modelovat proces vydávání verzí v rámci daného týmu a zároveň umožňují jednoduché nahlédnutí do současného stavu produktu v jednotlivých fází tohoto procesu. V rámci této metody lze také zavést automatizaci svázanou s konkrétní větví – například automatizovaný proces může nasadit verzi do produkce kdykoliv, kdy se ve větvi produkce objeví nový commit. Použití tohoto modelu je tedy vhodné v případě, kdy je proces vydávání verzí komplexnější a přímé vydávání z hlavní vývojové linie není vyhovující.

4.4.3 Větve vydání

Poslední zde zmíněná metoda je založena na vytváření zvláštních větví pro každou verzi produktu. Typická větev vydání vychází z aktuální hlavní linie vývoje a v rámci této větve je poté povolena pouze práce týkající se přípravy na vydání nové verze a případná stabilizace [40]. Nové funkcionality – které budou součástí dalších verzí produktu – se dále vyvíjí pouze nad hlavní vývojovou linií.

Případné opravy v rámci stabilizace lze provádět několika způsoby. Tím prvním je aplikování oprav přímo nad danou větví vydání s následným sloučením do hlavní linie vývoje. Druhým způsobem je integrace opravy do hlavní vývojové linie a následně s commitem obsahující danou opravu provést tzv. *cherry-pick* do dané větve vydání, což je operace kopírující daný commit do cílové větve bez nutnosti kompletního sloučení větví. Druhá metoda je označována termínem *upstream first policy* a je používána například v rámci

procesů *Google* [54]. Může ale nastat situace, kdy mezi hlavní vývojovou větví a větví vydání vznikne příliš velká vzdálenost a danou opravu nelze aplikovat do obou větví. Poté je nutno chybu opravit přímo do větve vydání a v hlavní linii vývoje řešit danou chybu zvlášť.

Větve vydání mohou přijít vhod například v situacích, kdy tým nedokáže udržet hlavní vývojovou linii ve stabilním stavu a je počítáno s tím, že bude třeba provést stabilizaci dané verze, která by měla být oddělena od dalšího vývoje. Tuto situaci lze ale řešit i větvemi vyzrálosti, viz předchozí sekce 4.4.2. Kde jsou ale větve vydání nezbytné, jsou prostředí, kde je třeba zároveň spravovat několik verzí najednou. Mohou také existovat prostředí, kde je s vydáváním verzí spjato určité tření [40]. Toto tření by mělo být ideálně odstraněno, což ale v některých prostředích není možné. Příkladem může být vydávání softwaru na aplikační platformy, jako jsou obchody s mobilními aplikacemi [40].

5 Návrh procesu verzování pro vývoj videoher

V předchozích kapitolách byly představeny a zanalyzovány specifika vývoje her týkající se verzování softwaru a prvky verzovacích procesů obecného softwarového vývoje. Cílem této kapitoly je dát poznatky z těchto kapitol dohromady a navrhnout proces verzování vhodný pro vývoj her s využitím Gitu.

5.1 Požadavky

V rámci této sekce budou představeny jednotlivé požadavky na následně navržený proces verzování. Na začátek je ale třeba zmínit, že jednotlivé požadavky nelze označit za univerzální. Jak již bylo v rámci předchozích kapitol několikrát zmíněno, jednotlivé týmy vyvíjí v rámci různých kontextů, které mohou tyto požadavky zásadně ovlivňovat.

Následuje výčet jednotlivých požadavků, které vyplývají zejména z analyzovaných specifik herního vývoje, provedeného průzkumu, specifik Gitu a analýzy verzovacích procesů v obecném softwarovém vývoji:

- **Podpora verzování velkých binárních souborů:** Ve vývoji her tvoří velké binární soubory významnou část projektu a samotný Git není pro verzování takovýchto souborů vhodný, viz kapitola 3.1.2, která se věnuje i řešení tohoto úskalí. Ve výsledném procesu by toto mělo být zohledněno.

- **Zamezení konfliktů nad binárními soubory:** Soubory spravované umělci jsou často binární soubory, pro které platí, že slučování paralelně nad nimi prováděných změn je obtížné až nemožné a ve většině případů ani nedává smysl. Této problematice se do detailu věnuje kapitola 3.1.3. Navržený proces by tedy měl zajišťovat zamezení paralelní práce nad takovými soubory.

- **Minimalizace četnosti a složitosti konfliktů nad ostatními soubory:** Předchozí požadavek se týkal konfliktů nad velkými binárními

soubory. Konflikty – zejména velké a špatně řešitelné – by se ale měly ideálně minimalizovat nad všemi soubory. Tato problematika je detailně probrána zejména v kapitole 4.3.

- **Jednoduchost procesu:** Jak vyplývá zejména z průzkumu popsaném v kapitole 3.2, netechnické profese často mají s používáním systémů správy verzí – Gitu zejména – potíže. Není tedy vhodné přidávat další vrstvu komplexity složitým procesem verzování, kterého by měli být součástí i právě netechničtí členové týmu. Výsledný proces verzování by tedy měl být pokud možno co nejjednodušší. Jednoduchost procesu ale samozřejmě neprospívá pouze netechnickým členům, nýbrž celému týmu.

- **Dlouhodobá podpora verzí:** Ze zmíněného průzkumu dále vyplývá častá nutnost podpory několika verzí her, což je tedy při návrhu také nutné vzít v potaz.

- **Podpora provádění revizí:** Míra důvěry v kvalitu artefaktů produkovaných jednotlivými členy se může lišit a je tedy vhodné v procesu počítat s praktikováním jejich revizí. Problematika revizí byla probrána v kapitole 4.3.4. Z průzkumu navíc vyplývá, že je tato praktika v herním vývoji hojně používána.

5.2 Návrh procesu verzování

Tato sekce se bude zabývat návrhu jednotlivých prvků verzovacího procesu. Bude nejdříve představen návrh způsobu integrace změn, revizí těchto integrací a bude i nastíněno, jak sledovat stav integrace dané funkcionality. Dále se sekce zaměří na způsob vydávání verzí. Nakonec budou zmíněna specifika prostředí.

5.2.1 Verzování velkých binárních souborů

V rámci kapitoly 3.1.2 byla zmíněna metoda řešení jednoho z úskalí Gitu při použití ve vývoji her – jehož řešení je zároveň jedním z vytyčených požadavků – tedy neefektivní verzování velkých binárních souborů. Řešením je konfigurace a zavedení používání rozšíření Git LFS. Tomuto rozšíření je věnována kapitola 2.3.3.

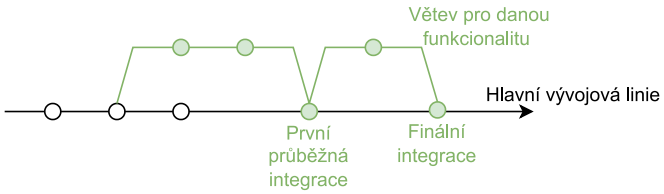
Vzhledem k tomu, že po nakonfigurování již rozšíření Git LFS spravuje ver-

zování binárních souborů zcela transparentně, není dále zmiňováno. Jedinou výjimkou je využití další jeho funkce v rámci navrženého procesu – zamykání souborů.

5.2.2 Integrace změn

V sekci 4.3 byly porovnány dva způsoby integrací změn – tématické větve a kontinuální integrace. Ze zmíněného porovnání vyplynulo několik výhod kontinuální integrace v případě klasického vývoje – konflikty jsou méně časté a jednoduší a existuje průzkumy ověřená korelace s návratností investic a ziskovostí produktu. Od technických uživatelů je ale požadována větší pečlivost, psaní automatických testů a znalost dodatečných technik, jako feature toggles a branch by abstraction zejména k zajištění stabilního stavu hlavní vývojové linie.

Pro navržený proces byla tedy z popsaných důvodů zvolena kontinuální integrace. Jsou tedy zavedeny dva předpoklady pojící se s praktikováním kontinuální integrace. První byl zmíněn v předchozím odstavci – jednotliví techničtí členové musí být schopni udržovat hlavní vývojovou linii ve stabilním stavu a v případě porušení této stability musí mít navrácení se do stabilního stavu nejvyšší prioritu. Druhým předpokladem je dostatečná důvěra mezi jednotlivými členy týmu. Tato důvěra může v průběhu času narůstat, ale měla by vždy dosahovat dostatečné úrovně. Jinými slovy – vývoj by měl probíhat v rámci klasického, například komerčního, týmu, nikoliv jako open-source.



Obrázek 5.1: Navržená metoda integrací.

Navržená metoda integrací změn je znázorněna na obrázku 5.1. Proces integrace změny je tedy následující:

1. Vytvoření větve z hlavní vývojové linie s vhodným pojmenováním.

2. Provádění průběžných integrací s hlavní vývojovou linií. Ideálně by se mělo před každou integrací provést lokální ověření, zda dané změny nenaruší stabilitu hlavní vývojové linie – například provedením lokálního

sestavení a spuštěním testů. Lze vidět, že ne každý commit musí nutně znamenat integraci. Do větve tedy lze například přidávat commity pro ucelený kus práce, který by ale narušoval stabilitu hlavní vývojové linie. Integrace se provede, až když je větev ve stavu vhodném k integraci.

Lze vidět, že se v práci nad větví pokračuje i po průběžných integracích. Z určitého úhlu pohledu tedy lze na tuto metodu nahlížet jako na kombinaci tématických větví a kontinuální integrace.

3. Finální integrace dané funkcionality.

S integracemi se pojí i další prvky procesu verzování – revize těchto integrací a zamezení konfliktů nad binárními soubory – kterým se věnují následující sekce.

5.2.3 Integrační revize

V kapitole 4.3.4 byly představeny tři metody revizí – předintegrační revize, pointegrační revize a párové programování. Bylo vyhodnoceno, že volba konkrétní metody závisí na kontextu, v jakém týmy pracují – zejména na míře důvěry mezi jednotlivými členy. Tento kontext se může v průběhu času měnit a tím tedy i vhodnost použití jednotlivých metod. Vhodnost jednotlivých metod se také může lišit na základě povahy úlohy – pokud seniorní vývojář provede marginální změnu typu opravy překlepu v dokumentaci, tak je zřejmé, že by předintegrační revize v tomto případě zavedla do integrace naprosto zbytečné tření.

V navržené metodě byla snaha tyto poznatky reflektovat zahrnutím celkem tří typů integrací:

- **S předintegrační revizí:** Jedná se o integraci s nejvyšší mírou tření – každá integrace musí být nejdříve schválena posuzovatelem, který samotnou integraci provádí. Je vhodná zejména pro nové a nezkušené členy týmu. Lze ji ale použít i pro integraci zásadnějších změn, na jejíchž základě se budou v nejbližší době vyvíjet další funkcionality. Další vhodné použití může být v případě, kdy si danou změnou není autor jist a přišel by mu vhod druhý pohled na věc.

- **S pointegrační revizí:** Autor provádí integrace sám a žádá o revizi již integrovaných změn. Tím tedy odpadá integrační tření z pohledu

autora dané funkcionality, kterou může plynule kontinuálně integrovat. Použití tohoto typu se očekává od již zkušenějších členů týmu, u kterých se předpokládá, že jejich změny budou ve většině případů odpovídat zavedeným standardům a budou v dostatečné kvalitě.

- **Bez revize:** Autor provádí integraci sám bez následného vyžádání revize. Je vhodná pro integrace, u kterých je zřejmé, že by revize nepřinesla žádnou přidanou hodnotu. Příkladem použití může být přidání menší funkcionality seniorním členem týmu s použitím zavedených standardů nebo provedení marginální změny typu oprava překlepu. Dále je tento typ vhodný použít v případě, kdy je daná funkcionalita implementována v rámci párového programování, kdy se samotná revize provádí paralelně s vývojem.

Konkrétní metoda volby typu integrace je na posouzení daného týmu. Volba může být ponechána kompletně v režii autora dané funkcionality, případně se mohou konkrétním členům omezit možnosti integrace na základě jejich zkušeností a s přibývající mírou důvěry se jim mohou zpřístupňovat možnosti nové.

Je také vhodné zmínit, že v rámci implementace jedné funkcionality se může použít několik typů integrací. Několik prvních integrací lze tedy například provést s pointegrační revizí a poslední předintegrační revizí. Je tedy vhodné nějakým způsobem evidovat aktuální stav integrace, čemuž se věnuje sekce 5.2.5.

5.2.4 Zamezení konfliktů nad binárními soubory

Metodám splňujícím další požadavek – zamezení konfliktů nad binárními soubory – se věnovala kapitola 3.1.3. Ze zhodnocení těchto metod vyšla nejlépe metoda explicitního zamykání pomocí rozšíření Git LFS. Jak ale bylo v kapitole zmíněno, samotné zamykání a odemykání zamezení konfliktů nezajišťuje. Pokud se totiž soubor odemkne před integrací změny, riziku konfliktu se zcela nepředejde. Je tedy vhodné zavést proces i na úrovni zamykání souborů skládající se z následujících kroků:

1. Zamčení souborů.

2. Úprava souborů a případné průběžné integrace.

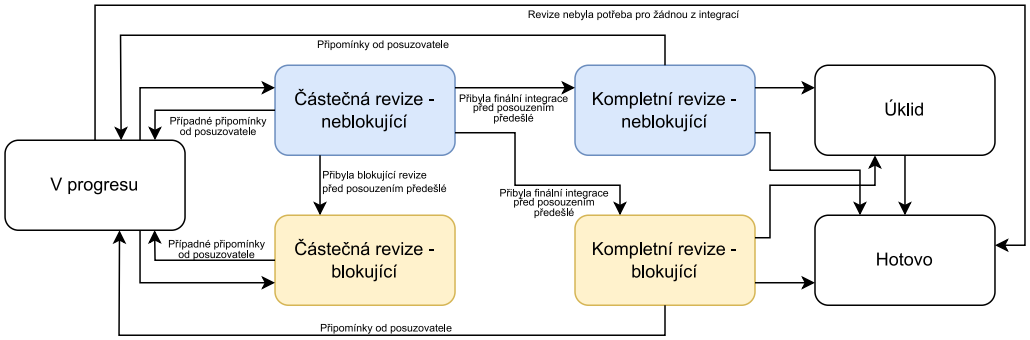
3. Finální integrace:

   (a) Bez revize: Integrace a odemčení souborů.

   (b) S revizí: Pokud nemá posuzovatel v rámci revize žádné připomínky, dá po integraci autorovi funkcionality vědět, že soubory může odemknout, viz stav *úklid* v následující sekci.

5.2.5 Sledování stavu integrace

V rámci vývoje softwaru jsou často používány nástroje umožňující sledování stavu jednotlivých úkolů – příkladem může být Jira [18]. Zadefinováním vhodných stavů a jejich integrováním do těchto nástrojů lze sledovat průběh integrace dané funkcionality. Příklad základních stavů týkající se integrace a přechodů mezi nimi je znázorněn na obrázku 5.2, kde jsou reflektovány zejména prvky procesu z předchozích dvou sekcí. Je třeba zmínit, že se jedná pouze o nástin základních stavů týkajících se integrace. Přidání dalších stavů záleží na konkrétních potřebách týmu.



Obrázek 5.2: Stavy kontinuální integrace funkcionality.

Tyto stavy by měly být brány v potaz v rámci prioritizace práce – například částečná blokující revize by měla mít pokud možno co nejvyšší prioritu vzhledem k tomu, že blokuje právě rozdělanou práci.

5.2.6 Vydávání verzí

Vzhledem k analýze v rámci kapitoly 4.4 a požadavku na dlouhodobou podporu vícero verzí vyplývá, že by měly být do návrhu procesu integrovány větve vydání. Proces vydání produkční verze by tedy měl být následující:

1. V případě usouzení, že by se měla vydat nová verze produktu, se z hlavní vývojové linie vytvoří nová větev s vhodným označením, například `release/X.Y`, kde `X.Y` představuje danou verzi produktu na prvních dvou úrovních.

2. V dané větvi se přidají commity v rámci nichž se provedou případné nutné kroky před vydáním verze. Počet těchto kroků by se měl pokud možno co nejvíce minimalizovat. Ideálně by neměla být potřeba žádných takovýchto kroků. V tom případě nepředstavuje větev vydání není nic jiného, než pouhý ukazatel na commit v hlavní linii vývoje a daná větev se od hlavní vývojové linie odchyluje až při případných opravách odpovídající verze.

3. Následné sestavení konkrétní verze by se mělo ideálně provádět v rámci automatizovaného procesu s využitím build serveru, viz sekce 4.2.2. V rámci tohoto procesu by se mělo provést i přidání tagu označující konkrétní verzi, například přidáním další další úrovně – tedy `X.Y.z`.

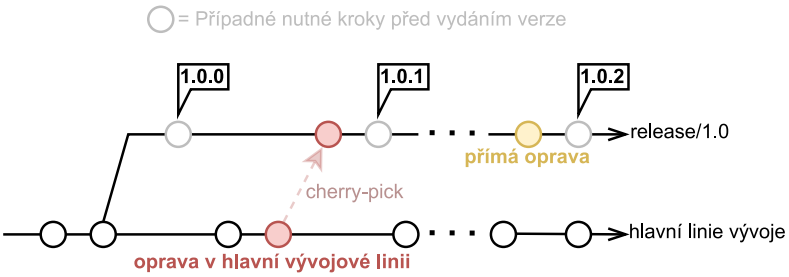
4. V případě nalezení a opravení chyby se vrací k bodu 2. Opravu lze provést dvěma způsoby:

   - **Oprava v hlavní vývojové linii:** Oprava se provede do hlavní vývojové linie a commit obsahující danou opravu se poté pomocí operace cherry-pick zkopíruje do odpovídající větve vydání. Jedná se o preferovaný způsob opravy.

   - **Přímá oprava:** Pokud nelze opravu provést prvním způsobem – například z důvodu již velké odlišnosti větve vydání a hlavní vývojové linie – může se oprava aplikovat přímo do odpovídající větve vydání. Následně je třeba zkontrolovat, zda se chyba projevuje také v hlavní vývojové linii a případně v provést opravu i v rámci této větve.

5. V případě ukončení podpory dané verze se odpovídající větev může smazat.

Příklad procesu vydávání verzí je znázorněn na obrázku 5.3, kde je zobrazen případ s nezbytnými kroky před vydáním verze. V případě, kdy by tyto kroky nutné nebyly, by dané commity zmizely a jednotlivé tagy by se posunuly na předešlé commity.



Obrázek 5.3: Navržená metoda vydávání verzí.

5.2.7 Specifika prostředí

Předchozí sekce v rámci této kapitoly se zabývaly návrhu obecného procesu pro vývoj počítačových her. Aby ale bylo verzování co nejefektivnější a s minimem různého tření, je často vhodné zavést i určité praktiky závisející na specifickém prostředí. Takovým prostředím může být například herní engine.

Jako jednoduchý příklad lze uvést verzování scén v herním enginu Unity, kde je vhodné – zjednodušeně řečeno – rozdělit scénu do hierarchie podscén, kde verzování podléhá každá z těchto podscén zvlášť. Je poté mnohem menší šance na konflikt při paralelní úpravě jedné scény a v případě zamykání scén není třeba uzamykat celou scénu, ale pouze dané podscény [60].

6 Jazyk pro popis verzovacích procesů

I přes jednoduchost navrženého procesu může být v určitých kontextech vhodné přidat další zjednodušující vrstvu odstiňující uživatele od příkazů Gitu a usnadňující používání daného procesu. Takováto vrstva může přinášet hodnotu nejen pro netechnické uživatele, ale případně i pro méně zkušené technické uživatele.

Příkladem takového zjednodušení může být rozšíření Gitu *gitflow* [38], jehož cílem je zjednodušení používání stejnojmenného modelu větvení. Zjednodušení spočívá v poskytnutí příkazů pro příkazovou řádku, které odstiňují uživatele od konkrétních Git příkazů. Namísto nich totiž poskytuje příkazy odpovídající zavedenému názvosloví v rámci daného modelu větvení. Například započetí práce na nové funkcionalitě se provádí příkazem `git flow feature start <jméno>` a její ukončení poté `git flow feature finish <jméno>`. Jak již ale bylo řečeno, rozšíření je limitované na použití konkrétního procesu verzování.

Jedním z cílů této práce je zvolit či navrhnout jazyk umožňující definici libovolného verzovacího procesu. Tento jazyk by poté měl být zpracovatelný aplikací, která na základě popisu v daném jazyce poskytne uživateli relevantní akce v grafickém uživatelském rozhraní. Realizace takové aplikace je součástí této práce a věnuje se jí následující kapitola 7.

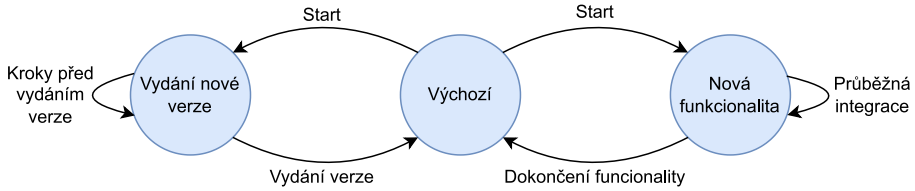
Použití takového jazyka má i několik pozitivních vedlejších důsledků, jako nutnost důkladného zamyšlení se nad daným procesem a jeho následnou pevnou definicí. Popis procesu verzování pomocí daného jazyka lze zároveň případně pokládat za jeho dokumentaci.

6.1 Základní požadavky

Daný jazyk by měl být zejména lehce čitelný a naučitelný pro osoby znalé Gitu, které by měly být schopni – po krátkém seznámení se s daným jazykem – namodelovat v daném jazyce libovolný verzovací proces.

V rámci verzovacích procesů jsou často nutné uživatelské vstupy různých typů – například pro specifikaci popisu commitu nebo pojmenování či vybrání konkrétní větve. Podpora definice uživatelských vstupů vhodných typů je tedy dalším požadavkem.

Na verzovací proces lze nahlížet jako na orientovaný graf, kde jednotlivé hrany představují jeden nebo několik Git příkazů, viz velmi zjednodušený proces na obrázku 6.1. V jazyce by tedy mělo být jednoduché namodelovat orientovaný graf, včetně přiřazení Git příkazů k jednotlivým hranám.



Obrázek 6.1: Zjednodušený verzovací proces namodelovaný orientovaným grafem.

6.2 Možnosti

6.2.1 DOT

Mezi kandidáty byl zařazen jazyk *DOT*, což je jazyk pro popis libovolných grafů [6]. Jazyk umožňuje definici jednotlivých grafů, uzlů a hran, včetně přidávání atributů k jednotlivým entitám.

V jazyce lze definovat jak orientovaný, tak neorientovaný graf [6]. Pro definici orientovaného grafu se používá klíčové slovo `digraph` a pro následnou definici hran se používá kombinace znaků `->`. Atributy se poté uvádějí do hranatých závorek za odpovídající entity ve formátu klíč a hodnota, kde jednotlivé atributy mohou mít pro různé aplikace různý význam.

Pomocí těchto atributů tedy lze na hranách definovat jednotlivé vstupy od uživatelů a Git příkazy. Příklad možné realizace procesu z obrázku 6.1, včetně vysvětlujících komentářů, je uveden ve fragmentu kódu 6.1. Pro zjednodušení je realizována pouze část daného procesu týkající se nové funkcionality a byly vynechány některé Git příkazy. V příkladu je také uvedena proměnná `puvodni_vetev`, což představuje větev, na které se uživatel nacházel v době před přechodem přes danou hranu.

Z příkladu je vidět, že samotná definice orientovaného grafu, modelujícího

verzovací proces, je velmi jednoduchá a přehledná. Při zhodnocení jazyka v kontextu dalších požadavků ale již vyvstávají určité nedostatky. První kategorie nedostatků se týká uživatelských vstupů – jejich definice je nepřehledná, jelikož je rozdělena do několika atributů a v případě nutnosti vícero vstupů se tato nepřehlednost ještě stupňuje. Tato nepřehlednost je dána zejména absencí hierarchie atributů – všechny atributy dané entity jsou na jedné a té samé úrovni. Dále lze označit za nepřehledné i samotné definování jednotlivých Git příkazů v případě, kdy daná hrana zastřešuje vícero příkazů.

Pro popis verzovacího procesu lze samozřejmě jazyk použít i jiným způsobem, ale vždy je nutné zavést určitý kompromis – buď zastřešit jedním atributem vícero informací nebo použít jeden atribut pro jednu informaci, čímž se ale zhoršuje škálovatelnost a čitelnost.

```
bigraph nazevgrafu {
    // Definice stavu jejich atributu
    v [nazev="Vychozi"];
    f [nazev="Nova funcionalita"];

    // Definice prechodu vcetne definice uzivatelskeho vstupu a
        jeho pouziti.
    v -> f [nazev="Start" vstup1_typ="text" vstup1_text="Zadejte
        nazev" gitPrikazy="git checkout -b ${vstup1}; git push -u
        origin ${vstup1};" ];

    f -> f [nazev="Prubezna integrace" vstup1_typ="text"
        vstup1_text="Zadejte popis" gitPrikazy="git commit -am
        ${vstup1}; git push;" ]

    f -> v [nazev="Dokonceni funkcionality" gitPrikazy="git
        checkout master; git merge <puvodni_vetev>; git push" ];
}
```

Fragment kódu 6.1: Popis procesu jazykem DOT.

6.2.2 SCXML

SCXML (State Chart XML) je jazyk založený na XML umožňující definici stavového automatu, včetně možnosti spouštění skriptů v rámci stavů či přechodů [27]. Jazyk, ve kterém lze zmíněné skripty psát, závisí na konkrétní platformě implementující tento jazyk. Mezi podporované jazyky patří

například JavaScript, Lua nebo C++ [27][24].

Příklad realizace zjednodušeného procesu z předchozí sekce je znázorněn ve fragmentu kódu 6.2, kde je pro psaní skriptů zvolen JavaScript standard ECMAScript.

```
<?xml version="1.0" encoding="UTF-8"?>
<scxml datamodel="ecmascript" name="Test" initial="Vychozi"
    version="1.0" xmlns="http://www.w3.org/2005/07/scxml">
   <script src=":git.js"/>
   <state id="Vychozi">
        <transition event="start" target="NovaFunkcionalita"
           nazev="Start">
         <script>
           var nazevVetve = textovyVstup("Zadejte nazev");
           git("checkout -b " + nazevVetve);
           git("push -u origin " + nazevVetve);
         </script>
      </transition>
    </state>
   <state id="NovaFunkcionalita">
        <transition event="prubeznaIntegrace"
           target="NovaFunkcionalita" nazev="Prubezna integrace">
         <script>
           var commitZprava = textovyVstup("Zadejte popis");
           git("commit -am " + commitZprava);
           git("push");
         </script>
      </transition>
      <transition event="konec" target="Vychozi" nazev="Dokonceni
          funkcionality">
         <script>
           var puvodniVetev = ziskejSoucasnouVetev();
           git("checkout master");
           git("merge " + puvodniVetev);
           git("push");
         </script>
      </transition>
    </state>
</scxml>
```

Fragment kódu 6.2: Popis procesu jazykem SCXML.

Podpora skriptů značně rozšiřuje možnosti jazyka a z příkladu lze vidět, že čitelnost v případě Git příkazů a uživatelských vstupů je – i na takto krátkém příkladě – značně lepší. Jako nevýhodu lze uvést, že jsou na autora kladeny vyšší požadavky vzhledem k tomu, že musí být obeznámen se syntaxí dvou jazyků – samotného SCXML a poté daného jazyka pro psaní skriptů, u kterého je ale potřeba znát pouze základní syntaxi. Dále neodpadává nedostatek zmíněný u jazyka DOT – atributy jednotlivých entit nejsou hierarchické. Pokud by tedy měly mít jednotlivé přechody vícero atributů tvořící určitou hierarchii, mohla by se snížit čitelnost.

6.2.3 Nový doménově specifický jazyk

Kromě vybrání již existujícího obecného jazyka se lze vydat jinou cestou – vytvoření jazyka vlastního, který bude navrhnut tak, aby co nejvíce vyhovoval používání v dané doméně, v tomto případě tedy popisu verzovacího procesu. Takovéto jazyky, specializované na konkrétní doménu, jsou nazývány doménově specifické jazyky [43]. V literatuře se často označují zkratkou *DSL* z anglického názvu *domain specific language*.

DSL se dělí na dva typy:

- **Interní:** Jako interní DSL se označuje zvláštní forma rozhraní definována v hostitelském obecném programovacím jazyce [43]. Program v interním DSL je poté validní kód tohoto obecného jazyka, ale používá pouze dané rozhraní. V mnoha případech je interní DSL implementován jako tenká vrstva nad abstrakcí obecného programovacího jazyka.

- **Externí:** Externí DSL mají naopak vlastní syntaxi a je pro něj třeba vytvořit parser v obecném programovacím jazyce, kterým se poté daný externí DSL zpracovává. Příkladem známých externích DSL může být SQL, CSS nebo také již zmíněný jazyk DOT.

Vytvořením externího DSL speciálně pro definování procesu verzování tedy mohou odpadnout – při správném návrhu – veškeré nevýhody zmíněné u dříve zmíněných jazyků. Daný jazyk navíc může mít přidanou hodnotu například v tom, že budou jeho klíčová slova úzce spjatá s procesem verzování a může být navržen speciálně pro potřeby aplikace, která jej bude dále zpracovávat.

Je samozřejmě nutné počítat s tím, že na vytvoření DSL, a nástrojů pro jeho podporu, je potřeba vynaložit určité úsilí. Je tedy vhodné zvážit, zda přínosy daného jazyka převažují nad daným množství úsilí.

Pro tvorbu jazyka lze použít několik nástrojů. Jako příklady lze uvést ANTLR, Eclipse Xtext nebo JetBrains Meta-Programming System [43].

6.2.4 Zhodnocení možností

Byly představeny dva existující jazyky, ale jazyků umožňující modelování orientovaných grafů je samozřejmě více – jako příklad lze uvést UML [20]. Výhoda použití již existujícího jazyka je zřejmá – není potřeba vynaložit žádné úsilí na návrh jazyka, tvorbu parseru a různých dalších nástrojů podporující psaní v daném jazyce. Pokud je totiž daný jazyk populární, tak často platí, že pro něj existují různé nástroje a plug-iny vytvořeny buď přímo autorem jazyka nebo komunitou [24][6].

Nevýhody existujících jazyků byly zmíněny v jednotlivých kapitolách věnujících se konkrétním jazykům. Obecně lze ale říci, že dané jazyky jsou až příliš obecné – v porovnání s vytvořením vlastního jazyka – a je nutné modelování procesu přizpůsobit danému jazyku, což vyúsťuje v horší čitelnost.

V rámci této práce tedy bude navržen externí DSL specifický pro popis procesů verzování systémem správy verzí Git vzhledem k usouzení, že výhody vytvoření nového DSL převažují nad zmíněnými úskalími existujících jazyků a nutným úsilím požadovaného na vytvoření takového jazyka.

Je také vhodné zmínit, že konkrétní jazyky byly zhodnoceny pouze z hlediska základních požadavků. Při zvolení cesty nového DSL lze jazyk navrhnou na míru vůči jak základním, tak i vedlejším požadavkům a přizpůsobit jej tak, aby byl lehce čitelný pro uživatele znalé Gitu a snadno rozšiřitelný v rámci dané domény. Nenastává tedy situace, kdy by bylo třeba vynechat volitelný požadavek z důvodu, že by byla jeho realizace v daném jazyce složitá nebo neproveditelná. V rámci následující sekce – zabývající se návrhem DSL – bude významnost této výhody zásadní, například při řešení alternativních toků.

6.3 Návrh jazyka

V rámci této sekce bude navrhnut externí DSL na základě zhodnocení možností v předchozí sekci. Základní syntaxe jazyka je inspirována datovým formátem *JSON* (JavaScript Object Notation) vzhledem k jeho čitelnosti a popularitě. V následujících sekcích budou postupně probrány jednotlivé konstrukce jazyka.

6.3.1 Základní konstrukce

Základní konstrukce jazyka slouží pro základní modelování verzovacího procesu. Řadí se mezi ně celkem čtyři entity – *command*, *step*, *action* a *state*. Jednotlivé entity a vztahy mezi nimi jsou dále popsány.

Command

Command představuje právě jeden Git příkaz. Lze v rámci něj používat proměnné a volání funkcí, viz dále. Syntaxe konstrukce command je znázorněna na jednoduchém příkladu vytvoření commitu, viz fragment kódu 6.3. Lze si povšimnout klíčového slova git na počátku, který lze v současné době považovat za zbytečný. Toto klíčové slovo je součástí příkazu z důvodu možné rozšiřitelnosti. Tímto způsobem lze jazyk rozšířit i o jiné typy příkazů, než pouze příkazy Gitu. Příkladem může být zaslání notifikací pomocí e-mailu nebo jiných platforem.

```
git "commit -am \"Commit zprava\"";
```

Fragment kódu 6.3: Syntaxe jazykové konstrukce command.

Step

Hlavním účelem konstrukce step je zastřešení několika konstrukcí command. Jeden step je poté považován za atomickou operaci – počítá se tedy s tím, že proběhnou buď všechny zastřešené příkazy, nebo žádný. Základní syntaxe je znázorněna ve fragmentu kódu 6.4, kde jsou zastřešeny celkem dva příkazy – commit a push. Jak již bylo řečeno, v příkladu je uvedena pouze základní syntaxe konstrukce. Step lze dále navíc parametrizovat a lze přidat další sady příkazů, které jsou v určitém smyslu inverzní k původním příkazům. Tyto rozšířené konstrukce jsou detailněji popsány dále.

```
commitAndPush() {
    commands [
        git "commit -am \"Commit zprava\"";
        git "push";
    ]
}
```

Fragment kódu 6.4: Syntaxe jazykové konstrukce step.

Action

Konstrukce action již představuje spustitelnou akci, u které se předpokládá, že bude určitým způsobem zpřístupněna uživateli. Základní definice akce se skládá ze tří částí.

První částí je definice posloupnosti stepů, které se mají v rámci dané akce provést. Na první pohled se může zdát, že je konstrukce step zbytečná a mohly by se namísto nich uvádět přímo jednotlivé příkazy. Step je ale součástí jazyka pro případy, kdy se složitější posloupnosti příkazů v rámci procesu verzování několikrát opakují. Bez této konstrukce by se musely jednotlivé posloupnosti duplikovat a v případě úpravy dané posloupnosti by se úprava musela provést na několika místech. Při použití reverzibilních stepů – viz sekce 6.3.5 – je výhoda rozdělení na dvě konstrukce ještě znatelnější.

Další část se týká definice vlastností zobrazení. Jazyk podporuje celkem tři takové vlastnosti akcí – název, stručný popis a případně i přiřazení ikony k dané akci.

Poslední část, týkající se základní definice akce, je reference na stav, do kterého se má v rámci procesu přesunout po provedení příkazů. Stavy procesu jsou poslední základní konstrukcí jazyka a jsou popsány dále v této sekci.

Příklad jednoduché definice ukázkové akce je znázorněn ve fragmentu kódu 6.5.

```
dokoncitFunkcionalitu {
    steps [
        commitAllAndPush {}
        checkoutMaster {}
    ]
    display {
        icon: "dokoncitFunkcionalitu.png";
        text: "Dokoncit funkcionalitu";
        tooltip: "Dokoncit a zverejnit funkcionalitu";
    }
    nextState: "vychozi";
}
```

Fragment kódu 6.5: Syntaxe jazykové konstrukce action.

I pro tuto konstrukci platí, že byly v rámci této sekce představeny pouze základní části. Pro akce totiž dále lze definovat podmínku dostupnosti a politiky řešení alternativních toků, viz sekce 6.3.4 a 6.3.5.

Akci lze také použít pro jiný účel, než jen pro vykonání určitých příkazů. Pokud se totiž neuvede žádný step, docílí se toho, že se po vykonání dané akce pouze změní stav na stav následující. Této vlastnosti lze využít například ve stavu, který slouží pouze jako jakýsi rozcestník do ostatních stavů.

State

Poslední základní konstrukcí je state, který představuje jeden stav procesu. V rámci stavu se definují akce, které by v něm měly být dostupné. Dále lze – stejně jako u akcí – definovat vlastnosti zobrazení. Jeden z těchto stavů je nutné označit jako výchozí nastavením hodnoty atributu `initialState`.

Zjednodušený příklad neúplné definice dvou stavů je uveden ve fragmentu kódu 6.6.

```
statesSettings {
    states [
        /* Definice prvniho stavu */
        vychozi {
            actions [
                zacitNovouFunkcionlaitu {
                    steps [ ... ]
                    display { ... }
                    nextState: "novaFunkcionalita";
                }
            ]
            display {
                text: "Vychozi stav";
            }
        }

        /* Definice druheho stavu */
        novaFunkcionalita {
            actions [ ... ]
            display {
                text: "Nova funkcionalita";
            }
        }
    ]
    initialState: "vychozi";
}
```

Fragment kódu 6.6: Syntaxe jazykové konstrukce state.

6.3.2 Proměnné a parametry

Jazyk umožňuje v rámci procesu definovat a referencovat několik typů proměnných. Dále je umožněno parametrizovat jednotlivé stepy. Proměnné a parametry lze referencovat obalením identifikátoru proměnné mezi kombinaci znaků `${` a `}`. Příkladem reference proměnné tedy může být `${promenna}`. Jednotlivé typy proměnných a parametry stepů jsou detailněji popsány dále.

Globální proměnné

Jedním z typů proměnných jsou tzv. globální proměnné. Hodnoty těchto proměnných definuje autor procesu verzování a jsou tedy pro každého člena týmu stejné. Sloužit mohou zejména pro definování konstant, které se napříč procesem několikrát opakují. Syntaxe definování globálních proměnných je znázorněna ve fragmentu kódu 6.7.

```
globalVariables [
    prefixTematickeVetve: "feature/";
    hlavniVyvojovaLinie: "master";
]
```

Fragment kódu 6.7: Definice globálních proměnných.

Uživatelské proměnné

Dalším typem proměnných jsou proměnné uživatelské. Ty umožňují přizpůsobit určité prvky procesu konkrétnímu uživateli. Příkladem použití je uchování jména uživatele v proměnné, kterou lze následně použít pro pojmenovávání tématických větví. Aby měly uživatelské proměnné význam, je nutné zajistit, aby je mohl uživatel sám definovat. Toho je docíleno jednak tak, že je v definici proměnné povoleno používání funkcí – viz sekce 6.3.3 – čímž lze od uživatele vyžádat vstup například při prvním spuštěním aplikace nad daným repozitářem. Výslednou hodnotu je poté vhodné uložit na disk, aby nemusela být zadávána stále dokola.

Syntaxe definování uživatelské proměnné je znázorněna ve fragmentu kódu 6.8. V ukázce jsou použity i funkce, kterým se bude za okamžik věnovat sekce 6.3.3. Dále si lze všimnout, že v porovnání s globálními proměnnými přibyla možnost definice popisu dané proměnné. Toho lze v aplikaci využít pro možnost změny hodnoty dané proměnné, kdy se uživateli nezobrazí pouhý identifikátor proměnné, ale smysluplný text ve formě daného popisu.

```
userVariables [
    username {
        value: input(type: "text", message: "Zadejte Vase jmeno");
        description: "Uzivatelske jmeno";
    }
]
```

Fragment kódu 6.8: Definice uživatelské proměnné.

Proměnné akce

Proměnné spadající do předposledního typu – tedy do proměnných akce – se definují na úrovni akce, v rámci níž jsou poté dostupné. Tento typ proměnných má opět význam spíše v kombinaci s použitím funkcí umožňující uživatelský vstup. Příklad definice a referencování proměnné akce je uveden ve fragmentu kódu 6.9.

```
novaFunkcionalita {
    actionVariables [
        nazevVetve: input(type: "text", message: "Zadejte nazev: ");
    ]
    steps [
        vytvorVetev {
            vetev: ${nazevVetve};
        }
    ]
    display { ... }
    nextState: "dalsiStav";
}
```

Fragment kódu 6.9: Definice uživatelské proměnné.

Systémové proměnné

Posledním typem proměnných jsou tzv. systémové proměnné. Pomocí nich může aplikace předávat procesu verzování různé informace. Aplikace navržená v rámci kapitoly 7 poskytuje mimo jiné proměnnou `StepContext.startBranch` pro předání informace jednotlivým stepům o větvi, která byla zvolena na počátku daného stepu. To umožňuje přepnout se v rámci daného stepu na jinou větev a stále si udržovat referenci na větev původní pro případnou potřebu.

Parametry stepů

Jak již bylo řečeno, jednotlivé stepy lze také parametrizovat. Parametry se uvádějí do hlavičky daného stepu a následně je lze referencovat jako klasické proměnné. Příklad definice parametru a jeho následné použití je uveden ve fragmentu kódu 6.10. Příklad použití parametrizovaného stepu z akce je poté znázorněn ve fragmentu kódu 6.11.

```
commitVseho(popisCommitu) {
    commands [
        git "commit -am \"" + ${popisCommitu} + "\"";
    ]
}
```

Fragment kódu 6.10: Příklad parametrizace stepu.

```
akceCommitVseho {
    steps [
        commitVseho {
            popisCommitu: "Predani parametru";
        }
    ]
    display { ... }
    nextState: "nasledujiciStav";
}
```

Fragment kódu 6.11: Příklad použití parametrizovaného stepu v rámci akce.

6.3.3 Funkce

Jazyk dále umožňuje volání funkcí. Definice funkcí je plně v režii aplikace sloužící jako platforma pro daný jazyk. Aplikace navržená v rámci této práce využívá funkcí pro umožnění uživatelských vstupů, viz sekce 7.5.1. Syntaxe provolání funkce umožňující uživatelský vstup, včetně předání argumentů, byl již zahrnut v rámci předchozí sekce ve fragmentu kódu 6.8.

Volání funkcí samozřejmě není limitováno pouze pro definici uživatelské proměnné. Lze je použít také v definici příkazu v rámci stepu a při předávání step argumentů.

6.3.4 Podmínky akcí

V procesu verzování mohou nastat situace, kdy v současném stavu není vhodné provést jednu z akcí i přes to, že ji daný stav poskytuje. Příkladem může být akce vytvářející commit – tato akce může v daném stavu dávat smyl, ale jen v případě, kdy uživatel provedl nějaké změny. Jazyk tedy poskytuje možnost zpřístupnit akci pouze v případě splnění určité podmínky.

Podmínky lze definovat pomocí logického výrazu. Podporované operátor jsou

AND (`&&`), OR (`||`) a unární logická negace (`!`). Poskytnutí operandů je opět kompletně v režii aplikace. Aplikace navržená v rámci této práce poskytuje například operand `changesExist`, pomocí kterého lze zjistit, zda uživatel provedl změnu některého souboru.

Příklad definice podmínky akce je uveden ve fragmentu kódu 6.12. Z příkladu lze vidět, že v objektu podmínky lze definovat i popis. Ten může v rámci aplikace sloužit, v případě nesplnění podmínky, pro předání informace uživateli, proč daná akce nelze spustit.

```
akceCommitVseho {
    steps [ ... ]
    display { ... }
    condition {
        expression: (changesExist && operand2) || !operand3;
        description: "Musi byt provedena alespon jedna zmena
            souboru";
    }
    nextState: "nasledujiciStav";
}
```

Fragment kódu 6.12: Příklad definice podmínky akce.

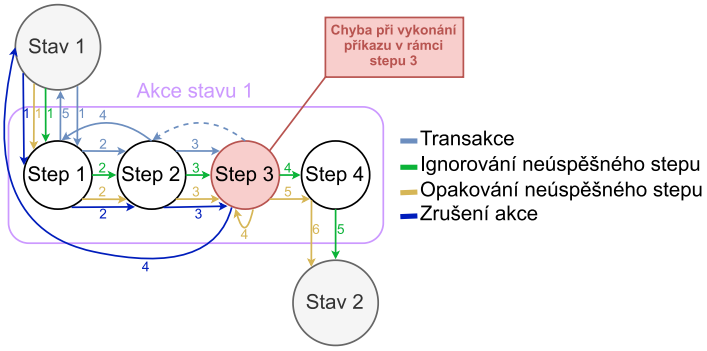
6.3.5 Řešení selhání příkazu

V rámci procesu mohou nastat situace, kdy se určité příkazy neprovedou očekávaným způsobem. Příčiny lze zařadit do dvou kategorií. První kategorie se týká selhání příkazu – příkladem může být nevalidní příkaz nebo ztráta připojení při pokusu o synchronizaci. Touto kategorií se bude zabývat tato sekce. Druhá kategorie – kterou se zabývá sekce následující – představuje situace, kdy daný příkaz způsobí konflikt.

Jazyk poskytuje celkem čtyři možnosti, jak se na úrovni akce vypořádat s případným selháním jednoho z příkazů. Konkrétně se jedná o zrušení akce, ignorování selhání, opakování neúspěšného stepu a transakční běh akce. Jednotlivým možnostem se tato sekce detailně věnuje dále. Všechny možnosti jsou také vizualizovány na obrázku 6.2, kde je znázorněn jednoduchý příklad o dvou stavech a jedné akci.

Pro zvolení konkrétní možnosti slouží atribut `executionPolicy`, kterému lze nastavit jednu z hodnot `abortOnFailure`, `repeatStepOnFailure`, `continueOnFailure` nebo `transactional`, kde každá z hodnot koresponduje s

jednou ze zmíněných možností.



Obrázek 6.2: Vizualizace možností pro řešení selhání příkazu.

Zrušení akce

Metoda zrušení akce, s kterou je spjat identifikátor `abortOnFailure`, je výchozí možností řešení selhání příkazů.

Na obrázku 6.2 lze vidět, že v případě selhání stepu 3 se následující stepy již nevykonají a zároveň se neprovede přechod do dalšího stavu. Aplikace by ale měla zajistit, že se uživatel o dané chybě dozví, aby na ní mohl případně reagovat.

Ignorování selhání

Ignorování selhání je metoda, která i přes selhání stepu pokračuje vykonáním následujících stepů. V rámci jazyka je metoda označována jako `continueOnFailure`.

Z obrázku 6.2 lze vidět, že i přes vyskytnutí chyby v rámci stepu 3, se pokračuje na step 4 a následně se přechází do stavu 2. Aplikace by opět měla zajistit, že bude uživatel o dané chybě informován.

Opakování neúspěšného stepu

Další metodou je opakování stepu, v rámci něhož k chybě došlo. Tato metoda je vhodná zejména pro stepy obsahující příkazy s uživatelskými vstupy. Identifikátor této možnosti je `repeatStepOnFailure`.

Jako příklad vhodného použití lze uvést akci obsahující step vykonávající příkaz pro vytvoření větve. Na obrázku 6.2 by jím mohl být step 3. S vytvořením větve se totiž často pojí uživatelský vstup sloužící pro zvolení názvu

dané větve. A pokud uživatel zvolí název, se kterým se ale již pojí jiná větev, tak daný Git příkaz selže. Poté je v některých případech vhodnější poskytnout uživateli možnost zvolit název jiný, než rušit celou akci.

Je nutné si uvědomit, že – vzhledem ke konceptu jazyka, kde je step považován za atomickou operaci – všechny zmíněné metody operují až na úrovni stepů, nikoliv na úrovni jednotlivých příkazů. Pokud tedy v rámci daného stepu předchází problematickému příkazu další příkazy, tak se provedou i při opakování.

Transakce

Poslední možností jsou tzv. transakční akce. Transakční akce umožňují v případě selhání jednoho ze stepů provést tzv. *rollback*, tedy navrácení do stavu před spuštěním dané akce. Identifikátorem metody je `transactional`.

Aby šlo takového chování docílit, je třeba umožnit, aby šly jednotlivé stepy v rámci akce zvrátit – tedy umožnit definici inverzních příkazů k příkazům provádějících se při vykonávání daného stepu. K definici inverzních příkazů slouží atribut `reverseCommands`.

Ve fragmentu kódu 6.13 je uveden příklad jednoduchého reverzibilního stepu přidávající soubory do oblasti připravených změn a následného vytvoření commitu. V rámci inverzních příkazů jsou poté uvedeny příkazy, které nejdříve vytvořený commit odstraní a následně se vybrané soubory vrátí z oblasti připravených změn do oblasti pracovního adresáře. V příkladu je také naznačena důležitá informace – inverzní příkazy se musí uvádět v opačném pořadí, než příkazy původní.

```
pridejAVytvorCommit(soubory) {
    commands [
        git "add " + ${soubory}; // 1. prikaz
        git "commit -m \"commit zprava\""; // 2. prikaz
    ]
    reverseCommands [
        git "reset --soft HEAD~1"; // inverzni prikaz k 2. prikazu
        git "restore --staged " + ${soubory}; // inverzni prikaz k
                                              // 1. prikazu
    ]
}
```

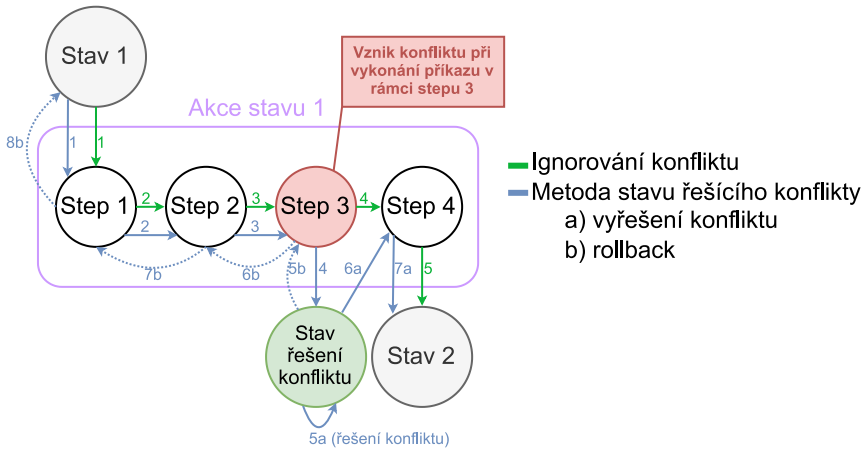
Fragment kódu 6.13: Příklad reverzibilního stepu.

Transakční akce poté v případě selhání jednoho ze stepů zajišťují postupné provolání inverzních příkazů předchozích stepů. Zároveň zamezí přechodu do následujícího stavu. Transakční akce je znázorněna také na obrázku 6.2, kde se po vyskytnutí chyby ve stepu 3 vyvolá postupný rollback od stepu 2. Inverzní příkazy se tedy neprovádějí od stepu, v rámci něhož příkaz selhal, nýbrž od stepu předchozího.

Je nutno zmínit, že při definování reverzibilních stepů je třeba dbát zvýšené opatrnosti, aby definované reverzní příkazy byly ve všech případech opravdu inverzní k původním příkazům.

6.3.6 Řešení konfliktů

Jazyk dále poskytuje možnost řešit situace, kdy jeden ze stepů v rámci akce způsobí konflikt. V určitých případech je vhodné se s tímto konfliktem nejdříve vypořádat, a až poté pokračovat v procesu. Jazyk poskytuje dva způsoby vypořádání se s konflikty v rámci akcí – ignorování konfliktu nebo přesunutí se do stavu řešícího konflikty. Obě metody jsou dále popsány a zároveň jsou vizualizovány na obrázku 6.3.



Obrázek 6.3: Vizualizace možností pro řešení konfliktů v rámci akcí.

Pro zvolení konkrétní možnosti slouží atribut akce `conflictResolutionPolicy`, kterému lze nastavit buď hodnotu `continue` nebo `stopAndResolve`.

Ignorování konfliktu

Jedním z možných chování akce je ignorování konfliktu a pokračování v akci. S touto metodou je spjat identifikátor `continue`. V příkladu na obrázku 6.3 lze vidět, že i přes vznik konfliktu ve stepu 3 se pokračuje stepem 4 a následně

se přechází do následujícího stavu. Ve stepech následujících po stepu v rámci něhož může vzniknout konflikt je tedy třeba Pokud tedy na step, v rámci které může vzniknout konflikt, navazují další stepy, pak je v nich třeba při použití této metody počítat s potenciální existencí konfliktu.

Tato metoda by se tedy měla ideálně používat pouze v případě, kdy je step obsahující příkaz, který může konflikt způsobit, posledním stepem dané akce. Konflikt se poté může vyřešit v dalším stavu mimo akci.

Metoda stavu řešení konfliktů

Druhou možností je zavedení stavu, do kterého se proces přesune v případě vzniku konfliktu. V rámci tohoto stavu je poté aplikací umožněno konflikt vyřešit. Po vyřešení konfliktu je možno pokračovat stepem bezprostředně následujícím po problematickém stepu, případně přechodem do následujícího stavu, pokud je problematický step v rámci akce stepem posledním. Identifikátorem této možnosti je `stopAndResolve` a jedná se o výchozí metodu.

Tok metody je v příkladu na obrázku 6.3 znázorněn modrou barvou. Lze vidět, že je tok od stavu řešení konfliktu rozdělen na dva. První tok, v obrázku označen písmenem *a*, vizualizuje případ popsaný v předchozím odstavci – v rámci stavu řešení konfliktu se konflikt vyřeší a pokračuje se dalším stepem. V případě transakčních akcí, viz předchozí sekce, lze v rámci tohoto stepu inicializovat i rollback, pokud například uživatel nechce daný konflikt řešit. Tok rollbacku je na obrázku označen písmenem *b*.

K rollbacku ze stavu řešení konfliktů je třeba přistupovat jinak, než v případě řešení selhání příkazů. Jak lze vidět na obrázku 6.3, rollback je inicializován již od problematického stavu, nikoliv až od stepu předchozího, jako tomu je u transakcí v rámci řešení selhání příkazu.

V rámci stepu, který je reverzibilní a zároveň může zapříčinit konflikt, se tedy musí počítat s tím, že může být součástí dvou rollback scénářů. První scénář se netýká konfliktů, nýbrž selhání příkazů v rámci následujících stepů. V tomto případě se provedou příkazy definované v atributu `reverseCommands`, viz předchozí sekce 6.3.5. Druhý scénář nastává v případě, kdy daný step způsobil konflikt a ve stavu řešení konfliktu se následně inicializoval rollback, viz již zmíněný tok označený písmenem *b* na obrázku 6.3. V tomto případě nelze použít příkazy `reverseCommands` určené pro scénář selhání příkazu, ale je nutno definovat další sadu příkazů obstarávající zotavení se z konfliktu.

Pro tyto příkazy slouží atribut `conflictReverseCommands`.

Příklad reverzibilního stepu, v rámci něhož může zároveň vzniknout konflikt, je uveden ve fragmentu kódu 6.14, kde lze vidět, že se příkazy pro oba scénáře liší. V případě `reverseCommands` se totiž počítá s tím, že step proběhl úspěšně a slučovací commit byl vytvořen, takže je potřeba jej odstranit. Naopak, v případě `conflictReverseCommands` slučovací commit vytvořen ještě nebyl. Je ale třeba zajistit zrušení operace `merge`.

```
slucDoMainline() {
    commands [
        git "checkout master";
        git "merge --no-ff" + ${StepContext.startBranch};
    ]
    reverseCommands [
        git "reset --hard HEAD~1";
        git "checkout " + ${StepContext.startBranch};
    ]
    conflictReverseCommands [
        git "merge --abort";
        git "checkout "+ ${StepContext.startBranch};
    ]
}
```

Fragment kódu 6.14: Příklad reverzibilního stepu pro sloučení větví.

6.3.7 Podpora Git LFS zamykání

Navržený jazyk je určen zejména pro použití v rámci aplikace navržené v následující kapitole, jejíž hlavním cílem je zjednodušení používání Gitu pro netechnické profese v herním vývoji. A jak bylo v rámci této práce několikrát zmíněno, určitá specifika herního vývoje lze vyřešit používáním možnosti zamykání rozšíření Git LFS. Předpokládá se tedy, že daná aplikace bude zamykání podporovat.

Do jazyka je tudíž zabudována podpora zamykání pomocí Git LFS ve formě možnosti specifikování, jaké soubory se mohou zamykat. Pomocí této informace poté aplikace může uživateli zobrazit seznam souborů k uzamčení a odemčení. Jazyk podporuje celkem dvě možnosti specifikování uzamykatelných souborů.

Čtení souboru .gitattributes

První možnost je založena na čtení souboru `.gitattributes` v kořenovém adresáři projektu. Soubor `.gitattributes` je jednoduchý soubor přiřazující souborům atributy. Jedním z takových atributů je atribut `lockable`, který slouží zejména pro rozšíření Git LFS. Na základě tohoto atributu totiž Git LFS rozhoduje, zda má být daný soubor dostupný pouze pro čtení v případě, kdy uživatel nedrží zámek daného souboru [2]. A právě pomocí atributu `lockable` se zároveň v rámci první metody definují i uzamykatelné soubory pro aplikaci.

Jednoduchý příklad souboru `.gitattributes` je uveden ve fragmentu kódu 6.15, kde jsou atributem `lockable` označeny všechny soubory ve formátu `bmp` a `jpg`.

```
*.bmp lockable
*.jpg lockable
```

Fragment kódu 6.15: Příklad souboru .gitattributes.

```
lfs {
    locks {
        lockableFilePatternsSource: "gitattributes";
    }
}
```

Fragment kódu 6.16: Zvolení metody čtení souboru .gitattributes.

V jazyce se tato možnost zvolí pomocí identifikátoru `gitattributes`, viz fragment kódu 6.16

Manuální specifikace

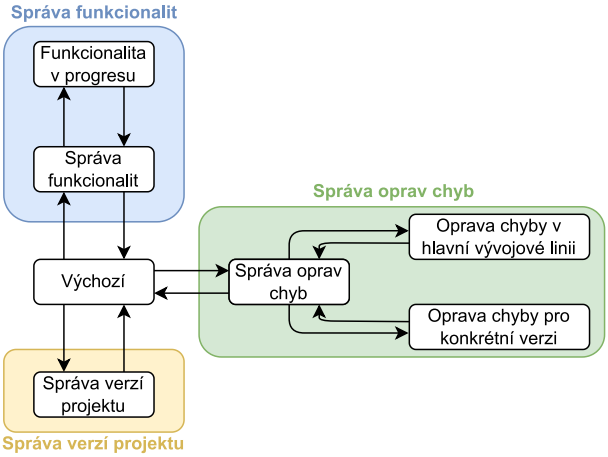
Další možností je manuální specifikace výčtu uzamykatelných souborů. V tomto případě se tedy nečte soubor `.gitattributes`, ale výčet uzamykatelných souborů je přímou součástí definice procesu. Této možnosti je přidělen identifikátor `custom`. Příklad zvolení této metody je uveden ve fragmentu kódu 6.17.

```
lfs {
    locks {
        lockableFilePatternsSource: "custom";
        lockableFilePatterns ["*.bmp", "*.jpg"]
    }
}
```

Fragment kódu 6.17: Zvolení metody manuální specifikace uzamykatelných souborů.

6.4 Realizace navrženého procesu

V rámci této sekce bude popsána realizace výsledného procesu z kapitoly 5 v navrženém jazyce. Před samotnou realizací je vhodné vytvořit model procesu ve formě orientovaného grafu, což je základní stavební kámen daného jazyka, kde jednotlivé přechody představují konkrétní akce. Zjednodušený model – obsahující všechny stavy, ale pouze základní přechody – je znázorněn na obrázku 6.4.



Obrázek 6.4: Základní model navrženého procesu.

Lze vidět, že je proces rozdělen do několika částí a stav `Výchozí` pro ně slouží pouze jako rozcestník – s jeho přechody tedy nejsou spjaty žádné příkazy. Ostatní části modelu, včetně všech přechodů a jejich významů, jsou podrobně popsány dále. V popisech jednotlivých částí modelu procesu jsou pro jednoduchost většinou vynechány příkazy zajišťující synchronizace repozitářů, na což je ale ve výsledné realizaci procesu myšleno.

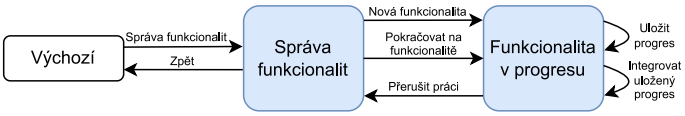
Kompletní realizace procesu v navrženém jazyce je přiložena v souboru

`Vstupni_data/Realizace_procesu/navrzeny_proces.gitsy`. V rámci realizace byly maximálně využity možnosti jazyka. Je například hojně využita možnost transakčních akcí za účelem celistvosti dané akce s tím, že je dodržena konzistence v rámci daného procesu – tedy i v případě selhání akce v průběhu vykonávání. V realizaci jsou dále použity systémové proměnné, funkce a operandy pro podmínky, které jsou poskytnuty implementací popsané v následující kapitole, viz sekce 7.5.

Je také nutné zmínit, že se daná realizace týká pouze prvků procesu bezprostředně týkajících se Gitu. Ostatní prvky – jako například vyžádání revizí, sestavení produktu a kontrolu stability hlavní vývojové linie – je nutno řešit nad rámec realizace procesu v navrženém jazyce. Dále není v procesu reflektováno zamykání souborů, s nímž se pojí zvláštní proces, viz sekce 5.2.4. Pro řešení těchto prvků procesu je vhodné do celkového procesu verzování zakomponovat další nástroje. Příkladem je build server kontrolující stabilitu hlavní vývojové linie a sestavující verze produktu. Dále je vhodné do procesu zakomponovat nástroj umožňující sledování stavu jednotlivých úkolů – příkladem může být Jira [18].

6.4.1 Správa funkcionalit

Na obrázku 6.5 jsou znázorněny všechny stavy a přechody týkající se správy funkcionalit.



Obrázek 6.5: Model správy funkcionalit.

Prvním stavem je `Správa funkcionalit`, v rámci něhož jsou dostupné celkem tři akce, konkrétně:

- **Zpět**: Navrácení se zpět do stavu `Výchozí`. S akcí nejsou spjaty žádné příkazy.

- **Nová funkcionalita**: Vytvoření nové větve s vhodným pojmenováním a přepnutí se na ni. Předpokládá se, že je s danou funkcionalitou spjat tzv. ticket, v rámci něhož se funkcionalita vyvíjí. Finální název větve se poté skládá z prefixu pro funkcionalitu reprezentovaného globální proměnou, uživatelského jména reprezentovaného uživatelskou

proměnnou a označením ticketu, který je vyžádán od uživatele. Příklad pojmenování tedy může být `feature/jirakv/T-123`.

- **Pokračovat na funkcionalitě**: Práci na funkcionalitě lze pozastavit, viz akce `Přerušit práci` stavu `Funkcionalita v progresu`. Zvolením této akce se inicializuje pokračování na dané funkcionalitě. Uživateli jsou dány na výběr větve představující jím vyvíjené funkcionality, tedy například ty, jejichž název začíná na `feature/jirakv/`. Po zvolení jedné z nich se nejdříve daná aktualizuje vzhledem ke stavu vzdáleného repozitáře a následně se na danou větev přepne.

Dalším stavem je `Funkcionalita v progresu` představující právě probíhající práci na dané funkcionalitě. Poskytuje také tři akce:

- **Uložit progres**: Slouží pro vytvoření commitu z vybraných změn ve větvi odpovídající dané funkcionalitě. Tato větev je následně synchronizována se vzdáleným repozitářem. Nejedná se tedy o integraci do hlavní vývojové linie, nýbrž pouze o průběžné uložení práce.

  Pokud se uložený progres na funkcionalitě dostane do stavu vhodného k integraci, nastávají dvě možnosti. První možností je přímá integrace do hlavní vývojové linie, viz akce `Integrovat uložený progres`. Následně lze buď požádat o pointegrační revizi, nebo revizi kompletně vynechat.

  Druhou možností je požádání o předintegrační revizi, použít akci `Přerušit práci` a ponechat případnou integraci na posuzovateli. Pokud se nejedná o finální integraci v rámci dané funkcionality, lze – po integraci daných změn – na funkcionalitě pokračovat pomocí akce `Pokračovat na funkcionalitě` předchozího stavu.

- **Integrovat uložený progres**: Po docílení stavu na funkcionalitě vhodného k integraci do hlavní vývojové linie lze použít akci `Integrovat uložený progres`. Jak již bylo řečeno v rámci popisu předchozí akce – tato akce je spjata s pointegrační revizí nebo s vynecháním revize.

  V rámci akce jsou zastřešeny kroky potřebné pro integraci. Nejdříve se tedy větev pro funkcionalitu sloučí do hlavní vývojové linie. Následně je třeba posunout ukazatel větve pro funkcionalitu na daný slučovací commit z předchozího kroku, aby v ní byl reflektován současný stav

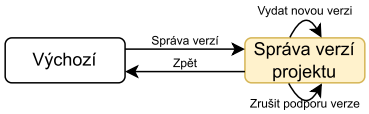
hlavní vývojové linie. Posledním krokem je synchronizace obou větví – větve pro funkcionalitu a hlavní vývojové linie – operací `push`, čímž se integrace dokončí.

Akce je definována jako transakční s použitím stavu řešení konfliktů. Při případném nastání konfliktu tedy lze buď konflikt vyřešit, nebo integraci zvrátit.

- **Přerušit práci**: Poslední akce slouží pro přerušení práce na dané funkcionalitě a přepnutí se zpět do hlavní vývojové linie. Použití akce se předpokládá při dokončení funkcionality nebo při čekání na předintegrační revizi. Dále ji lze použít v případě, kdy je třeba vyřešit úkol s vyšší prioritou.

6.4.2 Správa verzí projektu

Část modelu procesu týkající se správy verzí projektu je znázorněna na obrázku 6.6. Pro jednoduchost se v rámci modelu počítá s tím, že nejsou potřeba žádné dodatečné kroky spjaté s vydáním verze. Pokud by tyto kroky třeba byly, do modelu by stačilo přidat další stav pro správu konkrétní verze, do kterého by se přešlo akcí `Vydat novou verzi`. Dodatečné kroky by dále bylo potřeba reflektovat i ve správě oprav chyb, viz další sekce.



Obrázek 6.6: Model správy verzí projektu.

Celá správa verzí projektu se tedy sestává pouze z jednoho stavu `Správa verzí projektu`, v rámci něhož jsou poskytnuty celkem tři akce:

- **Zpět**: Navrácení se zpět do stavu Výchozí. S akcí nejsou spjaty žádné příkazy.

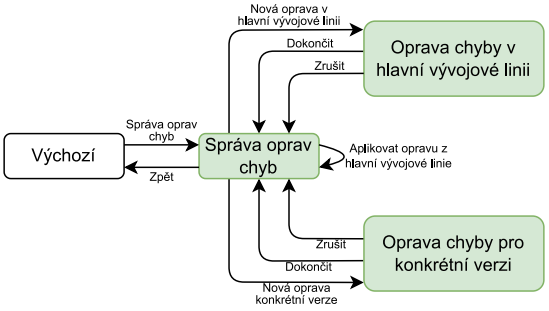
- **Vydat novou verzi**: Vytvoření nové větve vydání s označením `release/X.X`, kde `X.X` je zadáno uživatelem. Tato větev je následně vytvořena i ve vzdáleném repozitáři operací `push`.

- **Zrušit podporu verze**: Při ukončení podpory jedné z verzí produktu je vhodné odpovídající větev smazat, aby se například při aplikování oprav chyb, viz dále, zobrazovaly pouze podporované verze. V rámci

akce je po uživateli nejdříve vyžádáno vybrání větve, pro kterou skončila podpora. Tato větev se následně odstraní jak v lokálním, tak ve vzdáleném repozitáři.

6.4.3 Správa oprav chyb

Poslední částí modelu se týká oprav chyb jednotlivých verzí. Všechny stavy a akce jsou znázorněny na obrázku 6.7.



Obrázek 6.7: Model správy oprav chyb.

Jedním ze stavů je `Oprava chyby v hlavní vývojové linii` představující preferovanou metodu opravy chyb – tedy provedení opravy v hlavní vývojové linii a následné aplikování opravy operací `cherry-pick` do větví vydání. V rámci akce jsou k dispozici celkem dvě akce:

- **Dokončit**: Dokončí práci na opravě vytvořením commitu sestávající se z provedených změn. Na tomto commitu je následně vytvořen tag, aby na něj byla vytvořena čitelná reference pro případ aplikování opravy do konkrétní verze, viz akce `Aplikovat opravu z hlavní vývojové linie`. Tag je pojmenován podle vzoru `mainline-hotfix/<označení>`, kde `označení` je zadáno uživatelem.

- **Zrušit**: Akce sloužící pro zrušení provádění opravy chyby formou opuštění tohoto stavu. S akcí nejsou spjaty žádné příkazy.

Další stav – `Oprava chyby pro konkrétní verzi` – představuje druhou metodu opravy chyb, tedy přímou opravu konkrétní verze. V rámci stavu se již předpokládá, že se uživatel nachází na odpovídající větvi vydání, viz akce `Nová oprava konkrétní verze` stavu `Správa oprav chyb`. Stav poskytuje dvě akce:

- **Dokončit**: Dokončí práci na opravě vytvořením commitu a následnou synchronizací větve vydání se vzdáleným repozitářem. Nakonec se změní aktuální větev na hlavní vývojovou linii.

- **Zrušit**: Pokud je potřeba provádění opravy zrušit, lze použít tuto akci. S akcí je spjat jeden příkaz – přepnutí se na hlavní vývojovou linii.

Posledním stavem je `Správa oprav chyb` umožňující provést celkem čtyři akce:

- **Zpět**: Navrácení se zpět do stavu Výchozí. S akcí nejsou spjaty žádné příkazy.

- **Nová oprava v hlavní vývojové linii**: Akce inicializující preferovanou metodu opravy chyb – tedy provedení opravy v hlavní vývojové linii. V rámci akce se nevykonávají žádné příkazy. Slouží pouze pro přechod do stavu `Oprava chyby v hlavní vývojové linii`.

- **Aplikovat opravu z hlavní vývojové linie**: Po provedení opravy do hlavní vývojové linie v rámci stavu `Oprava chyby v hlavní vývojové linii` je třeba tuto opravu aplikovat do konkrétních verzí. Právě k tomu slouží tato akce. Od uživatele je nejdříve vyžádáno zvolení větve ze seznamu větví vydání a tagu ze seznamu tagů odpovídajících jednotlivým commitům obsahující opravy. Na základě těchto informací se do zvolené větve operací `cherry-pick` zkopíruje daný commit obsahující opravu.

- **Nová oprava konkrétní verze**: Inicializace přímé opravy konkrétní verze. Od uživatele je nejdříve vyžádáno zvolení větve ze seznamu větví vydání, která se následně nastaví jako větev aktuální. Tím tato akce končí a samotná oprava se řeší v následujícím stavu `Oprava chyby pro konkrétní verzi`.

7 Git klient

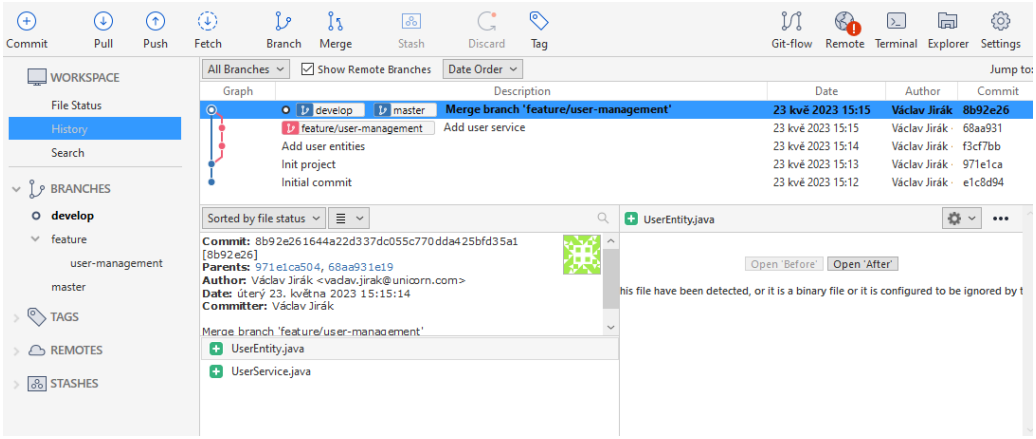
V rámci práce je implementována aplikace ve formě desktopového Git klienta s grafickým uživatelským rozhraním, zaměřující se zejména na netechnické uživatele. Hlavním požadavkem tedy je jednoduchost používání, odstínění uživatele od Git příkazů a zjednodušení používání různých procesů verzování. Většina těchto požadavků je vyřešena implementací jazyka navrženém v předchozí kapitole. Aplikace byla pojmenována *Gitsy*, sloučením názvu Git a anglického slova *easy*.

7.1 Existující nástroje

Git klientů s grafickým uživatelským rozhraním existuje celá řada, ale v rámci analýzy nebyl nalezen žádný, který by splňoval všechny požadavky – zejména zjednodušení používání různých procesů verzování.

Dále budou představeny celkem dva nástroje – Sourcetree a integrovaný klient v IntelliJ IDEA. Součástí analýzy bylo vícero nástrojů – například GitKraken [11] a Git Extensions [9] – ale bylo uváženo, že není třeba představovat všechny, ale pouze ty, které jsou zajímavé z pohledu zmíněných požadavků.

7.1.1 Sourcetree



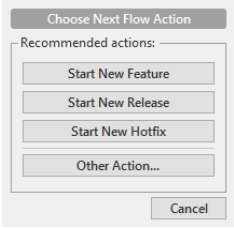
Obrázek 7.1: Grafické uživatelské rozhraní nástroje Sourcetree.

Sourcetree je populární Git klient ve formě samostatného desktop klienta,

umožňující používání Git příkazů skrze grafické uživatelské rozhraní, viz obrázek 7.1.

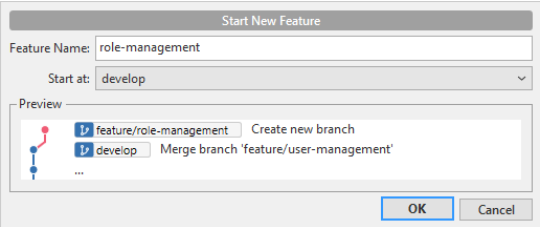
Sourcetree je zde zmíněn zejména kvůli jeho podpoře rozšíření gitflow, které zjednodušuje používání stejnojmenného modelu větvení [38]. Tato podpora nástroj přibližuje ke splnění požadavku zjednodušení používání různých procesů verzování. Nesplňuje jej však zcela, jelikož je rozšíření limitováno pouze na jeden konkrétní proces verzování. Podpora tohoto rozšíření samozřejmě není specialita nástroje Sourcetree, je integrována například i nástroj GitKraken [11], ale princip podpory je víceméně stejný napříč nástroji.

Dané zjednodušení spočívá v poskytnutí relevantních tlačítek v současném stavu, které zastřešují potřebné git příkazy, viz obrázek 7.2.



Obrázek 7.2: Podpora rozšíření gitflow nástrojem Sourcetree.

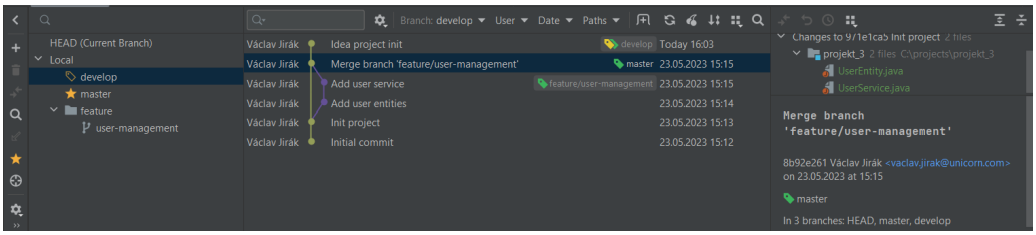
Po vybrání konkrétní akce se nástroj uživatele doptá na potřebné informace, jako například pojmenování větve, viz obrázek 7.3. Na základě těchto informací se poté provedou odpovídající Git příkazy.



Obrázek 7.3: Konkrétní akce v rámci podpory rozšíření gitflow nástrojem Sourcetree.

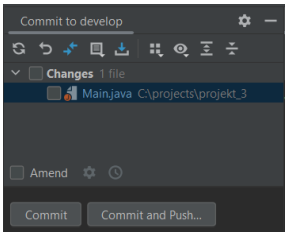
7.1.2 IntelliJ IDEA

V rámci této sekce nebude představen samostatný desktopový Git klient, nýbrž Git podpora v rámci IntelliJ IDEA, což je vývojové prostředí pro programování například v jazycích Java, Kotlin a dalších [15]. Podpora Gitu je zde také implementována ve formě grafického uživatelského rozhraní, viz obrázek 7.4



Obrázek 7.4: Git podpora v IntelliJ IDEA.

Nástroj IntelliJ IDEA je zmíněn zejména kvůli jeho přístupu ke změnám. Nástroj totiž díky *changelist* konceptu [16] uživatele odstiňuje od rozdělení změn do dvou pracovních oblastí – tedy pracovního adresáře a oblasti připravených změn. Všechny změny jsou prezentovány v jednom seznamu, viz obrázek 7.5, což značně zvyšuje uživatelskou přívětivost, zejména pro nezkušené uživatele.



Obrázek 7.5: Changelist koncept v IntelliJ IDEA.

7.2 Zvolené technologie

7.2.1 Qt Framework

Qt Framework je open-source software pro vývoj multiplatformních aplikací s grafickým uživatelským rozhraním [23]. Aplikace vytvořené pro grafické uživatelské prostředí používají nativní vzhled operačního systému, takže vyvinuté aplikace se vždy přizpůsobí vzhledu používaného prostředí. Jedná se o *C++* framework, ale lze jej použít i například v rámci jazyka *Python* [23]. Výsledná aplikace byla implementována v Qt Framework verze 6.2.3 a v jazyce C++ verze 17.

Qt Framework byl zvolen zejména kvůli nativnímu vzhledu aplikací v něm vyvinutých, kvalitně zpracované dokumentaci a jeho vyzrálosti, s čímž souvisí dostupnost spousty dodatečných materiálů.

7.2.2 ANTLR

ANTLR je nástroj umožňující generování vlastního parseru na základě gramatiky v tzv. rozvinuté Backusově–Naurově formě [52]. Nástroj podporuje generování parserů pro mnoho jazyků, mimo jiné právě i pro C++.

Nástroj byl zvolen zejména k jeho jednoduchosti, kvalitní dokumentaci [53] a existenci dalších nástrojů podporujících ANTLR – například rozšíření [50] do editoru *Visual Studio Code*.

7.3 Obrazovky

Aplikace se skládá z několika obrazovek, které budou dále představeny. Zároveň na nich budou vysvětleny některé ze základních funkcionalit.

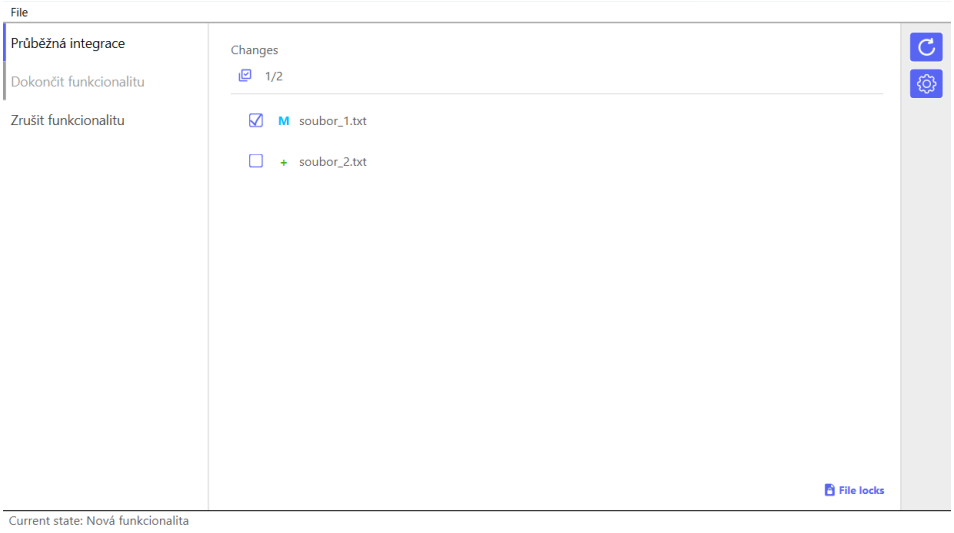
7.3.1 Volba projektu

Na obrázku 7.6 je zobrazena první obrazovka sloužící ke správě projektů. Projekty lze přidávat, odstraňovat a otevírat. S otevřením projektu se pojí posloupnost kroků, kterým je věnována sekce 7.4.

Obrázek 7.6: Obrazovka volby projektu.

7.3.2 Otevřený projekt

Obrázek 7.7 obsahuje hlavní okno aplikace představující otevřený projekt. Hlavním účelem obrazovky je zprostředkování možnosti spouštění akcí dostupných v současném stavu. Název současného stavu je uveden v levé dolní části obrazovky.



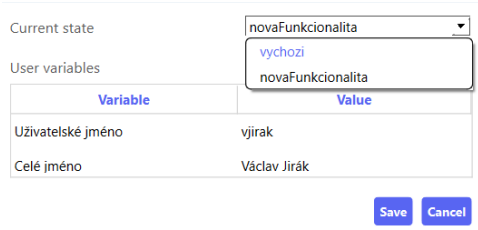
Obrázek 7.7: Obrazovka otevřeného projektu.

Dále je poskytnuto vybrání změněných souborů. U jednotlivých změn je vizualizován i typ změny – například zda se jedná o nový soubor nebo soubor pozměněný. Lze si všimnout, že jsou změny prezentovány v zjednodušené formě – na úrovni prezentace se totiž nerozlišuje mezi změnami nacházející se v pracovním adresáři a oblasti připravených změn. Na úrovni procesu je ale samozřejmě informace o typu pracovní oblasti k dispozici, viz sekce 7.5.3.

Z této obrazovky se lze také dostat na obrazovky další – konkrétně na obrazovku nastavení procesu a správy zámků, viz dále.

7.3.3 Nastavení procesu

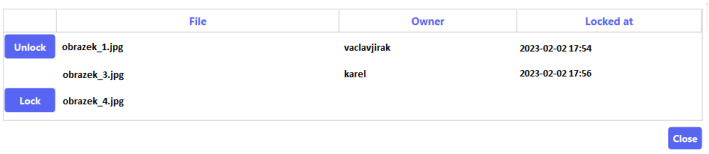
Další obrazovka slouží pro správu vlastností procesu specifických pro daného uživatele, viz obrázek 7.8. Konkrétně zde lze pozměnit hodnoty jednotlivých uživatelských proměnných a případně lze i manuálně přepnout stav procesu.



Obrázek 7.8: Obrazovka nastavení procesu.

7.3.4 Správa zámků

Poslední obrazovka slouží pro správu zámků rozšíření Git LFS, viz obrázek 7.9. Tato obrazovka je zpřístupněná pouze v případě povolení zamykání v rámci procesu, viz sekce 6.3.7. Na obrazovce jsou k dispozici všechny uzamykatelné soubory, včetně případných informací o současných držitelích zámků.

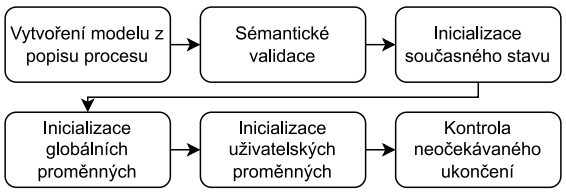


Obrázek 7.9: Obrazovka správy zámků.

7.4 Otevření projektu

S otevřením projektu je spjata posloupnost několika kroků zajišťující mimo jiné tvorbu interního modelu procesu, vyhodnocení proměnných a validace. Všechny kroky jsou znázorněny na obrázku 7.10 a dále popsány.

Aplikace si v průběhu otvírání projektu – a následných spouštění akcí – udržuje pomocné soubory ve složce `.gitsy/internal`, jejíž cesta je uvedena relativně ke kořenovému adresáři repozitáře. Soubory v této složce jsou pro každého uživatele unikátní a je ji tedy nutné vyloučit z verzování. Toho se docílí přidáním řádky `.gitsy/internal` do souboru `.gitignore`.



Obrázek 7.10: Proces otevření projektu.

Prvním krokem je parsování souboru `.gitsy/process.gitsy` obsahující definici procesu v jazyce z předchozí kapitoly, během něhož se tvoří interní model daného procesu a zároveň se provádí validace syntaxe jazyka. U tohoto souboru se očekává, že bude zahrnut ve verzování, aby jej měli k dispozici všichni členové týmu. Parser jazyka je vytvořen nástrojem ANTLR na základě gramatiky nacházející se v souboru `Aplikace_a_knihovny/antlr_grammar/GitProcess.g4`, který je součástí této práce.

Následuje další sada validací zaměřující se spíše na sémantiku. Součástí této sady je zejména ověření, zda uvedené reference na stavy, stepy a proměnné jsou v rámci daného modelu validní.

Dalším krokem je inicializace současného stavu procesu. Při prvotním spuštění projektu se současný stav vyhodnotí na základě atributu `initialState` v definici procesu. Tento stav se následně uloží do souboru `.gitsy/internal/currentState`, který je následně aktualizován při každé změně stavu a je čten při následujících otevření projektu.

Následuje vyhodnocení hodnot globálních a uživatelských proměnných. Uživatelské proměnné se po vyhodnocení ukládají do souboru `.gitsy/internal/userVariables`. Vzhledem k tomu, že vyhodnocení uživatelských proměnných může vyžadovat uživatelské vstupy, je tento soubor čten při každém následujícím otevření projektu a není tedy nutno vyhodnocení provádět pokaždé.

Posledním krokem je kontrola, zda aplikace neočekávaně neskončila při vykonávání příkazu v rámci daného projektu. Tento krok je detailněji popsán v sekci 7.5.5.

7.5 Implementace procesu

Specifikace jazyka, navrženého v předchozí kapitole, pevně definuje určité prvky procesu verzování, ale u některých prvků nechává volnost pro konkrétní implementaci. Tato sekce se bude věnovat zejména právě takovým prvkům procesu.

7.5.1 Uživatelské vstupy

Jazyk umožňuje konkrétním implementacím poskytnout různé funkce v rámci definice procesu. Této možnosti je využito pro podporu uživatelských vstupů.

Je implementována podpora celkem tří typů uživatelských vstupů. Každý vstup je implementován funkcí `input`, která má minimálně dva povinné parametry – `type` pro specifikaci typu vstupu a `message` představující zprávu, která se v rámci požadavku na vstup zobrazí uživateli.

Uživatelům je při vyžádání vstupu umožněno tento požadavek odmítnout. Pokud je uživatelský vstup odmítnut během vykonávání akce – například při vyžádání zadání názvu větve v rámci stepu – a uživatel požadavek odmítne, daný step bude v tomto případě pokládán za nezdařilý a inicializuje se vybraná metoda pro řešení selhání příkazu, viz sekce 6.3.5.

Ve většině případů je tedy vhodné řešit uživatelské vstupy potřebné pro akci již na úrovni proměnných akce, viz sekce 6.3.2. Pokud totiž uživatel odmítne požadavek na vstup při vyhodnocování proměnné akce, daná akce se nezačne vykonávat.

Textový vstup

Prvním typem je vstup textový, pro který je vyhrazen identifikátor `text`. Textovém vstupu lze předat dodatečný parametr `inputPattern` ve formě regulárního výrazu, kterým se definuje vzor, jenž musí uživatelský vstup splňovat. Toho lze využít například pro limitaci rozsahu textu nebo množiny povolených znaků. Příklad je uveden ve fragmentu kódu 7.1, kde je ukázka vyžádání textového vstupu pro zadání názvu větve včetně jednoduchého omezení vstupu.

```
input(type: "text", message: "Zadejte nazev vetve",
       inputPattern: "[0-9A-Za-z/]*")
```

Fragment kódu 7.1: Příklad textového uživatelského vstupu.

Výběr větve

Další typ vstupu slouží pro výběr existující větve. Identifikátorem tohoto typu je `branchSelect` a má dva volitelné parametry. Prvním je parametr `pattern` sloužící pro filtraci větví, které jsou uživateli následně dány na výběr. Nastaví-li se tento parametr například na `feature/*`, tak se uživateli nabídnou pouze větve, které danému vzoru odpovídají.

Dalším parametrem je parametr `omitPart`. Tímto parametrem lze z jednotlivých větví, než se uživateli nabídnou k výběru, odstranit část názvu. Projde-li například dříve zmíněným filtrem větev `origin/feature/funkc`

ionalita1 s nastaveným parametrem omitPart na hodnotu origin/, tak se uživateli nenabídne větev s celým názvem, ale část origin/ se ořízne. Uživateli je tedy ve výsledku nabídnuta větev feature/funkcionalita1. Tohoto parametru lze využít například právě pro případy, kdy není žádáno, aby se uživateli nabízely vzdálené větve.

Příklad použití je uveden ve fragmentu kódu 7.2, kde jsou všechny volitelné parametry použity.

```
input(type: "branchSelect", message: "Vyberte funkcionalitu",
    pattern: "feature/*", omitPart: "origin/")
```

Fragment kódu 7.2: Příklad výběru větve.

Výběr tagu

Poslední typ uživatelského vstupu – s identifikátorem tagSelect – slouží pro výběr konkrétního tagu. Má jeden volitelný parametr pattern, který slouží – stejně jako v případě výběru větve – k filtraci tagů, které jsou uživateli následně dány na výběr. Příklad použití je uveden ve fragmentu kódu 7.3.

```
input(type: "tagSelect", message: "Vyberte tag",
    pattern: "bugfix/*")
```

Fragment kódu 7.3: Příklad výběru tagu.

7.5.2 Podmínky akcí

V rámci sekce 6.3.4 byla představena možnost jazyka pro podmínění dostupnosti akce. Poskytnutí jednotlivých operandů je ale v režii implementace. Aplikace poskytuje celkem tři operandy, které lze libovolně kombinovat ve výsledném logickém výrazu:

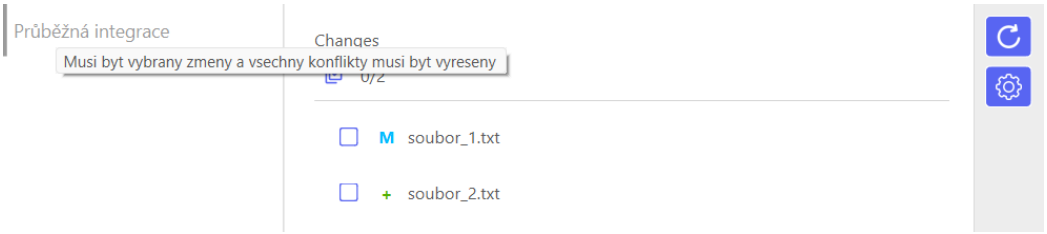
- Operand changesExist nabývá hodnoty true v případě, kdy má uživatel ve svém repozitáři nějaké změny buď v pracovním adresáři, nebo v oblasti připravených změn.
- Operand selectedChangesExist nabývá hodnoty true v případě, kdy je v seznamu změn alespoň jedna změna vybrána.
- Operand conflictsExist nabývá hodnoty true v případě, kdy v da-

ném čase existují nevyřešené konflikty.

Příklad použití podmínek v rámci akce je uvedena ve fragmentu kódu 7.4 a příklad nesplnění této podmínky v aplikaci je poté uveden na obrázku 7.11.

```
prubeznaIntegrace {
    steps [ ... ]
    display {
        text: "Prubezna integrace";
    }
    condition {
        expression: selectedChangesExist && !conflictsExist;
        description: "Musi byt vybrany zmeny a vsechny konflikty
            musi byt vyreseny";
    }
    nextState: "nasledujiciStav";
}
```

Fragment kódu 7.4: Příklad definice podmínky akce.



Obrázek 7.11: Příklad nesplněné podmínky v aplikaci.

7.5.3 Systémové proměnné

V rámci definice procesu lze referencovat proměnné, které jsou přímou součástí jazyka – tedy globální a uživatelské proměnné a parametry stepů, viz sekce 6.3.2. Při návrhu jazyka ale bylo počítáno i s možností poskytnutí dodatečných, tzv. systémových, proměnných danou implementací.

Aplikace poskytuje celkem tři typy systémových proměnných – proměnné uživatelského rozhraní, kontextové proměnné stepu a kontextové proměnné akce.

Proměnné uživatelského rozhraní

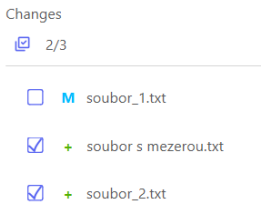
Proměnné uživatelského rozhraní slouží pro předání informací procesu o současném stavu uživatelského rozhraní. Aplikace definuje celkem dvě takové proměnné – `Ui.selectedUnstagedFiles` obsahující uživatelem vybrané pozměněné soubory z pracovního adresáře a `Ui.selectedStagedFiles` obsahující vybrané soubory z oblasti připravených změn. Obě proměnné nabývají hodnot ve formě seznamu názvu souborů oddělených mezerou, což je formát vhodný pro použití v rámci Git příkazů.

Jak již bylo řečeno, uživateli jsou změny souborů nacházející se v obou oblastech – v pracovním adresáři a oblasti připravených změn – prezentovány ve sloučeném seznamu. Uživatel je tedy odstíněn od tohoto rozdělení změn do dvou oblastí. V rámci procesu je ale tato informace již podstatná a právě proto aplikace vybrané změny rozděluje do dvou proměnných. Příklad použití proměnné uživatelského rozhraní je uveden v příkazu ve fragmentu kódu 7.5.

```
git "add " + ${Ui.selectedUnstagedFiles};
```

Fragment kódu 7.5: Příklad použití proměnné uživatelského rozhraní.

Pokud se tedy akce spustí například se stavem uživatelského rozhraní znázorněném na obrázku 7.12, kde se obě vybrané změny nacházejí v pracovní oblasti, tak se příkaz z fragmentu kódu 7.5 vyhodnotí v příkaz `git add "soubor s mezerou.txt" soubor_2.txt`.



Obrázek 7.12: Příklad vybrání změn v uživatelském rozhraní.

Kontextové proměnné stepu

Dalším typem systémových proměnných jsou proměnné nabývající hodnot na základě kontextu daného stepu. Jsou definovány dvě systémové proměnné spadající do této kategorie – `StepContext.startBranch` a `StepContext.startCommit` – uchovávající větev a commit, které byly aktuální

v době začátku vykonávání daného stepu. Příklad použití je znázorněn ve fragmentu kódu 7.6 obsahující definici jednoduchého stepu.

```
sloucitDoVetveMaster() {
    commands [
        git "checkout master";
        git "merge " + ${StepContext.startBranch};
    ]
}
```

Fragment kódu 7.6: Příklad použití kontextové proměnné stepu.

Kontextové proměnné akce

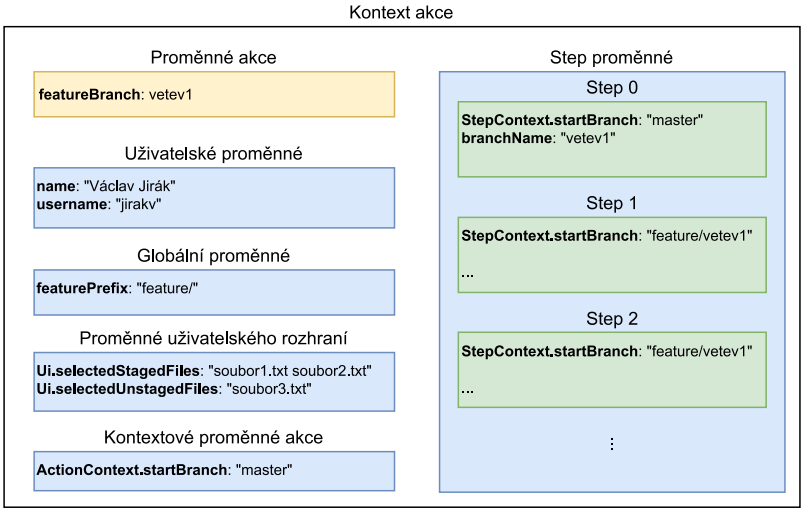
Obdobné proměnné jsou dostupné i v kontextu akce, nazývané kontextové proměnné akce. Jsou také definovány celkem dvě takové proměnné – `ActionContext.startBranch` a `ActionContext.startCommit` – uchovávající větev a commit, které byly aktuální v době začátku vykonávání dané akce.

7.5.4 Kontext akce

Pří každém spuštění akce si aplikace vytváří kontext zastřešující relevantní proměnné, který se propaguje postupně každým stepem dané akce. V rámci kontextu se udržují jednak proměnné společné pro všechny stepy – tedy globální proměnné, uživatelské proměnné, proměnné uživatelského rozhraní a kontextové proměnné akce. Dále jsou uchovávány proměnné relevantní pouze pro konkrétní stepy, tedy kontextové proměnné stepu a případné argumenty stepu. Poslední částí kontextu akce jsou proměnné akce, které jsou dostupné pouze na úrovni akce a jednotlivé stepy k ním tedy nemají přístup.

Step má tedy přístup ke všem společným proměnným a k odpovídajícím step proměnným. Příklad kontextu je znázorněn na obrázku 7.13, kde například první step dané akce má přístup ke všem modře označeným proměnným a ke Step 0 proměnným.

Kontext se průběžně ukládá do souboru `.gitsy/internal/actionContext` pro případy, kdy se aplikace ukončí v průběhu vykonávání akce. K takové situaci může dojít buď očekávaně – viz stav řešení konfliktu v sekci 7.5.6 – nebo neočekávaně, čemuž se věnuje sekce 7.5.5. Pokud by se kontext akce neukládal do souboru, nebylo by při opětovném spuštění možné získat hodnoty některých proměnných, například proměnné `Step-`



Obrázek 7.13: Vizualizace kontextu akce.

`Context.startBranch`.

7.5.5 Zajištění konzistence

Při návrhu jazyka a aplikace bylo myšleno i na zajištění konzistence v rámci procesu. Zajištění konzistence na úrovni jazyka se věnovala sekce 6.3.5 pojednávající o dostupných metodách řešení selhání jednoho z příkazů, což lze do určité míry považovat za očekávanou událost.

Aplikace ale řeší i situace neočekávané – tedy takové, kdy se aplikace neočekávaně ukončí v průběhu vykonávání jednoho z příkazů. Takováto situace může nastat v případě pádu aplikace kvůli neodhalené chybě nebo vynuceném vypnutí aplikace uživatelem, například kvůli jeho netrpělivosti během vykonávání dlouho trvajícího příkazu.

Takové případy jsou do jisté míry ošetřeny zavedením následujícího procesu vykonávání jednotlivých příkazů:

1. **Vyhodnocení příkazu**: Aplikace nejdříve vyhodnotí příkaz do podoby interpretovatelné Gitem.

2. **Vytvoření žurnálu**: Před spuštěním daného příkazu se vytvoří soubor `.gitsy/internal/commandJournal` obsahující informace o právě prováděném příkazu. Ukázka souboru, včetně vysvětlujících komentářů, je uvedena ve fragmentu kódu 7.7.

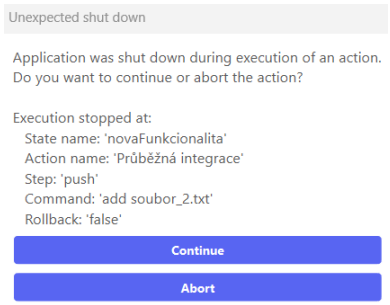
3. **Provedení příkazu**: Provedení vyhodnoceného příkazu z prvního kroku.

4. **Odstranění žurnálu**: Smazání souboru vytvořeného v rámci druhého kroku, čímž se dává najevo úspěšné vykonání daného příkazu.

```
{
    " actionIdentifier ": "addFiles", // Identifikator akce
    "stepIndex": 0, // Poradi stepu v ramci akce
    "commandIndex": 0, // Poradi prikazu v ramci stepu
    "evaluatedCommand": "add soubor1.txt soubor2.txt", // Vyhodnoceny prikaz
    "isRollback": false // Zda se jedna o klasicky nebo rollback prikaz
}
```

Fragment kódu 7.7: Příklad žurnálu.

Aplikace poté provádí kontrolu existence souboru `.gitsy/internal/commandJournal` při spuštění aplikace nad daným projektem. Tato kontrola již byla zmíněna v popisu procesu otvírání projektu – viz sekce 7.4 – kde představuje poslední krok dané posloupnosti.



Obrázek 7.14: Dialogové okno informující o předešlém neočekávaném ukončení aplikace.

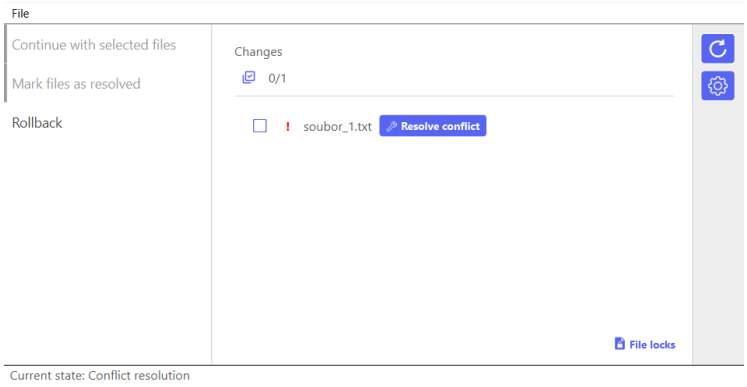
Pokud se při otevření projektu zjistí existence daného souboru, je uživatel o neočekávaném vypnutí informován a jsou mu nabídnuty dvě možnosti, viz obrázek 7.14. Jednou z možností je *Abort*, při jejíž zvolení se celá akce přeruší. Druhá možnost je *Continue*, pomocí které lze v akci pokračovat od problematického příkazu daného stepu. Lze si povšimnout, že je v tomto případě porušen koncept atomických stepů, jelikož se přistupuje přímo ke konkrétnímu příkazu daného stepu.

Je nutno říci, že přidáním této funkcionality se také porušuje jeden ze základních požadavků na aplikaci – tedy uživatelská přívětivost pro netechnické profese a odstínění od Git příkazů. Neočekává se ale, že by tato situace

nastávala často. Ideálně by se s ní daný uživatel neměl setkat nikdy. Její ošetření je ale vhodné, aby byla zajištěna konzistence v rámci procesu.

7.5.6 Řešení konfliktů

V sekci 6.3.6 byly představeny dvě metody vypořádání se s konflikty – pokračování v akci i přes výskyt konfliktu nebo přechod do stavu, v rámci něhož se konflikt vyřeší. Implementace druhé metody v rámci aplikace je znázorněna na obrázku 7.15, kde je zobrazen případ, kdy se proces právě nachází ve stavu řešení konfliktů.



Obrázek 7.15: Stav řešení konfliktu.

Lze vidět, že v rámci daného stavu jsou k dispozici celkem tři akce:

- *Continue with selected files*: Tato akce provede potřebné Git příkazy pro označení konfliktu jako vyřešený a následně zajistí pokračování akce, v rámci niž konflikt vznikl. Akce je podmíněna vyřešením všech konfliktů.

- *Mark files as resolved*: Pokud se konflikt vyřešil bez použití aplikace – nebo přímo Gitu – je o tom nutné Git informovat. Právě k tomu slouží tato akce, která konflikty u vybraných souborů označí jako vyřešené.

- *Rollback*: V případě transakční akce je v rámci stavu řešení konfliktů zpřístupněn rollback, viz sekce 6.3.6.

V seznamu souborů na obrázku 7.15 si lze všimnout tlačítka *Resolve conflict* u souboru obsahující konflikt. Tlačítko slouží pro spuštění externího nástroje pro řešení konfliktu. Nástrojů pro řešení konfliktu existuje několik. Jedním z nich je *meld* [36], na kterém bude předvedena základní konfigurace potřebná pro to, aby bylo tlačítko *Resolve conflict* dostupné.

Základní konfigurace se skládá z několika příkazů příkazové řádky, které jsou, včetně vysvětlivek, uvedeny ve fragmentu kódu 7.8. Příkazy je nutné pouštět buď přímo v daném repozitáři, nebo je možné všechny příkazy pustit s přepínačem `--global`, čímž se zajistí, že se daná konfigurace aplikuje v rámci celého systému.

```
// Konfigurace cesty k nastroji meld
git config --global mergetool.meld.path "<cesta-k-nastroji-meld>"
// Nastaveni nastroje meld jako vychozi nastroj pro reseni konfliktu
git config --global merge.tool meld
// Vypnuti nutnosti potvrdit otevreni externiho nastroje
git config --global mergetool.prompt false
// Povoleni duvery ve vystupni kodu nastroje ve vsech pripadech
git config mergetool.meld.trustExitCode true
// Odstraneni pomocnych souboru po ukonceni nastroje
git config mergetool.keepbackup false
```

Fragment kódu 7.8: Konfigurace nástroje pro řešení konfliktu.

7.5.7 Logování

I přes to, že jedním z požadavků na aplikaci je odstínění uživatele od příkazů Gitu, je vhodné vykonané příkazy zaznamenávat, například pro případnou analýzu problémů.

Aplikace proto všechny vykonávané příkazy zaznamenává do souboru `.git sy/internal/gitCommands.log`, kam se zaznamenávají jednotlivé příkazy, jejich výstupy a čas provedení.

8 Testování a zhodnocení

8.1 Automatické testování aplikace

Za účelem ověření správného fungování aplikace byly vytvořeny automatické testy. Testy byly implementovány za pomocí frameworku `QtTest`. Testy ověřují správnost zejména následujících funkcionalit – tvorba a validace modelu, vykonávání jednotlivých akcí a stepů včetně alternativních toků a vyhodnocování výrazů.

8.2 Uživatelské testování

Dále bylo provedeno uživatelské testování. Testování proběhlo na dvou úrovních. Cílem první úrovně bylo zhodnotit navržený jazyk a jeho implementace aplikací. Účastníky první části tedy byli techničtí uživatelé. Cílem druhé úrovně bylo zejména ověření, zda je výsledná aplikace vhodná pro používání netechnickými uživateli, kteří tedy byli cílovými účastníky této části.

Testerům byl nejdříve stručně vysvětlen účel a princip aplikace a byl jim poskytnut instalátor aplikace, včetně české nebo anglické verze uživatelské příručky, viz příloha A. Všechno testování proběhlo na počítačích účastníků s operačním systémem Windows 10 nebo Windows 11, pro které je v současnosti aplikace dostupná.

Testeři na obou úrovních byli po testování požádáni, zda by byli ochotni aplikaci otestovat i na reálném herním projektu s tím, že by se následně ozvali se zpětnou vazbou. Žádný účastník se ale – z různých důvodů – této části testování nakonec nezúčastnil.

8.2.1 Testování jazyka pro popis verzovacích procesů

Požadavky kladené na účastníky první části byly zejména dobrá znalost Gitu a zkušenosti s herním vývojem. Této části se zúčastnilo celkem sedm účastníků. Všech sedm testerů se následně zúčastnilo i části druhé, viz následující sekce 8.2.2, aby se získaly názory na použitelnost aplikace i od technických uživatelů.

Účastníkům byly poskytnuty, kromě instalátoru a uživatelské příručky, také ukázky realizace procesů. Některé z těchto ukázek jsou přiložené k práci na cestě `Vstupni_data/Realizace_procesu`.

Tato část testování se sestávala ze dvou fází. V rámci první fáze se účastníkům rozeslaly zmíněné materiály, tedy instalátor aplikace, uživatelská příručka a ukázky realizace procesů. Na seznámení se s materiály bylo uživatelům dáno libovolné množství času – až několik dní. Ve druhé fázi byl účastníkům rozeslán popis jednoduchého procesu verzování a krátký dotazník, viz příloha B. Úkolem poté bylo proces v navrženém jazyce realizovat na základě daného popisu. Účastníci byli také požádáni o sdílení obrazovky v průběhu realizace, aby se případně zjistily určité vzorce používání.

Z testování vzešlo několik připomínek a návrhů na vylepšení. Řešením jednotlivých bodů se poté věnuje sekce 8.3. Připomínky a návrhy lze shrnout do následujících bodů:

- **Podpora komentářů**: Několik respondentů uvedlo, že by uvítali možnost psaní komentářů v rámci definice procesu.

- **Přidání možnosti limitace dostupnosti akcí**: Dále bylo zmíněno, že by bylo vhodné přidat možnost limitovat dostupnost akcí, aby například ne všichni měli možnost vydávat verze produktu.

- **Připomínky k syntaxi**: Respondenti měli také drobné připomínky k syntaxi jazyka, například k nekonzistentnosti v používání dvojteček.

- **Zamezení syntaktickým chybám**: Bylo zpozorováno, že účastníci často dělali menší syntaktické chyby, na které přišli až při spuštění aplikace. Chybu na základě vypsané hlášky aplikací sice opravili relativně rychle, ale ověření správné syntaxe až při otevření projektu zásadně snižuje uživatelskou přívětivost a efektivitu realizace procesu.

Až na zmíněné výtky byl jazyk hodnocen ve všech tázaných aspektech – tedy syntaxe jazyka, uplatnění jazyka ve herním vývoji a jednoduchost realizace procesu – pozitivně. Nejpozitivněji byla hodnocena zejména základní myšlenka, tedy usnadnění používání libovolného verzovacího procesu a odstínění uživatelů od příkazů Gitu.

Je třeba zmínit, že respondentům dělal největší problém koncept reverzi-

bilních akcí, u kterého zpravidla platilo, že se s ním seznamovali relativně dlouho. Dále relativně dlouho strávili hledáním některých příkazů, například pro zvrácení operace `rebase`, k čemuž byla některým účastníkům poskytnuta pomoc. Konečné reakce lze ale shrnout tak, že se jedná o užitečnou vlastnost, se kterou se ale musí zacházet obezřetně.

8.2.2 Testování aplikace a navrženého procesu

Další úroveň testování byla určena, jak již bylo řečeno, zejména pro netechnické uživatele s cílem ověřit aplikaci z pohledu použitelnosti. Testování se zúčastnilo celkem pět netechnických uživatelů. Tohoto testování ale byli součástí také účastníci z předchozí části, aby se aplikace zhodnotila i z pohledu technických uživatelů. Celkem se této části tedy zúčastnilo dvanáct testerů.

Jediný požadavek na netechnické účastníky byl, aby se pohybovali v herním vývoji a měli alespoň minimální zkušenosti s verzováním softwaru. Nebyl tedy kladen požadavek na znalost Gitu.

Testování proběhlo ve formě plnění scénáře, přiloženém v příloze C, a následném vyplnění krátkého dotazníku. Byly vytvořeny také anglické verze scénáře a dotazníku pro anglicky hovořící respondenty. V rámci scénáře byla použita realizace verzovacího procesu ze sekce 6.4, který byl uživatelům nejdříve stručně představen. Tato část testování se tedy zároveň zabývala zhodnocením navrženého procesu. Respondenti byli opět požádáni o sdílení obrazovky při plnění daného scénáře.

Před započetím práce byla uživatelům poskytnuta pomoc s nastavením systému do stavu vhodného k testování – tedy zejména s instalací Gitu a rozšíření Git LFS, instalací a konfigurací nástroje pro řešení konfliktů a naklonování testovacího repozitáře. Dále byly uživatelům v případě potřeby vysvětleny některé principy a terminologie, jako například co je to konflikt, kdy může nastat a jak ho řešit pomocí nástroje Meld. Dalšími často vysvětlovanými principy bylo zamykání souborů, vydávání verzí produktu a opravování jednotlivých verzí.

Z testování opět vyplynulo několik připomínek a návrhů na vylepšení, jejichž řešení se také věnuje následující sekce 8.3. Připomínky a návrhy lze shrnout do následujících bodů:

- **Odmítnutí uživatelského vstupu**: V testované verzi aplikace nebylo uživateli umožněno odmítnout uživatelský vstup. Uživatel tedy byl blokován do té doby, dokud vstup nezadal.

- **Ošetření změny procesu**: Bylo odhaleno, že není žádným způsobem ošetřen případ, kdy se změní soubor realizující proces v průběhu používání aplikace nad daným repozitářem. V důsledku nastávali situace, kdy se musel v projektu smazat i celý `.gitsy/internal` adresář, aby šel projekt v aplikaci opět otevřít. Tento bod byl odhalen v rámci předchozí části testování. Je ale zmíněn zde, jelikož se týká aplikace a nikoliv samotného jazyka.

- **Ošetření operací provedených mimo aplikaci**: Dále byla vznesena připomínka, že aplikace neošetřuje případy, kdy se provedou některé Git příkazy měnící stav procesu mimo aplikaci. Příkladem může být změna větve z příkazové řádky. Aplikace se poté může nacházet ve stavu procesu, který ale neodpovídá realitě.

- **Změna prezentace souborů při zamykání**: Byla také vznesena připomínka k formě uzamykání souborů, kdy danému respondentovi nevyhovovala současná prezentace uzamykatelných souborů – tedy jednoduchý seznam souborů. Dle jeho názoru by byla vhodnější prezentace souborů v adresářové struktuře.

- **Vizualizace změn**: Aplikace v současné době uživateli prezentuje pouze názvy změněných souborů, na což si stěžovali zejména techničtí uživatelé, kteří jsou z ostatních nástrojů zvyklí zároveň vidět i konkrétní změny provedené v daných souborech.

- **Menší odstínění od příkazů**: Někteří techničtí uživatelé měli problém s tím, že se v aplikaci nezobrazují konkrétní příkazy spjaté s danou akcí a často se koukali do souboru definice procesu za účelem zjištění těchto informací.

- **Zobrazování více informací v daném stavu**: Jeden z respondentů uvedl, že by bylo vhodné přidat možnost přidání vhodných informací ke konkrétním stavům. Jako příklad uvedl stav `Funkcionalita v progresu`, kde by bylo vhodné vizualizovat například počet uložených, ale neintegrovaných změn – v terminologii Gitu tedy počet commitů, které se nacházejí pouze ve zvláštní větvi pro funkcionalitu a nebyly ještě

sloučeny do hlavní vývojové linie.

- **Absence běžných funkcionalit**: Někteří účastníci by také uvítali implementaci funkcionalit běžně dostupných v ostatních Git klientech s uživatelským rozhraním. Často byla zmiňována například absence vizualizace větví.

- **Přidání podpory v herních enginech**: Dále zaznělo, že by bylo vhodnější řešení implementovat spíše jako rozšíření do herních enginů, než jako samostatnou aplikaci.

- **Terminologie navrženého verzovacího procesu**: Jak již bylo řečeno, testování probíhalo s použitím realizovaného verzovacího procesu ze sekce 6.4. I přes krátké představení procesu měli někteří účastníci problém s použitou terminologií. Téměř všichni netechničtí uživatelé byli zmateni z dostupnosti dvou akcí `Uložit progres` a `Integrovat uložený progres` ve stavu `Funkcionalita v progresu`, jelikož jim nebylo zcela jasné, co přesně jednotlivé akce znamenají. Po krátkém dodatečném vysvětlení se ale rozdíl podařilo vždy vysvětlit.

Zhodnocení aplikace se značně lišilo v závislosti na tom, zda byl uživatel technické nebo netechnické povahy.

Techničtí účastníci

Techničtí uživatelé většinou zmiňovali, že by aplikaci v současném stavu spíše nepoužívali, a to z různých důvodů. Jedním z důvodů bylo nepřinesení žádné přidané hodnoty v porovnání se současně používanými nástroji, jelikož nemají s používáním Gitu v kombinaci se zavedeným procesem žádný problém. Koncept aplikace byl ale hodnocen kladně a se splněním testovacího scénáře neměli problém.

Techničtí uživatelé byli dále požádání o zhodnocení použitého verzovacího procesu, který všichni zhodnotili také kladně.

Netechničtí účastníci

Reakce od netechnických uživatelů byla pozitivnější. Sice většina účastníků také zmínila, že by v současném stavu aplikaci spíše nepoužívala vzhledem k absenci některých funkcionalit – například vizualizace konkrétních změn v textových souborech – ale reakce na koncept aplikace byla velice kladná,

zejména od těch, kteří Git používají. Většina také zmínila, že by po přidání některých funkcionalit používání aplikace určitě zvážila.

Jak již bylo uvedeno ve výčtu připomínek výše, netechničtí uživatelé měli občas problém s vykonáváním scénáře, zejména kvůli použité terminologii v rámci navrženého procesu verzování. Po krátkém dovysvětlení scénář ale splnili všichni.

8.3 Limitace a možná rozšíření

Z obou částí uživatelského testování vzešlo několik připomínek a návrhů na vylepšení týkajících se jak jazyka, tak aplikace. O některých limitacích a možnostech vylepšení – včetně několika zmíněných v rámci testování – se vědělo již před provedením uživatelského testování. Shrnutí všech limitací, připomínek a možností vylepšení se věnuje tato sekce. Některé body z testování byly již zapracovány, což bylo zároveň reflektováno v odpovídajících kapitolách této práce. U nezapracovaných bodů je poté popsán návrh řešení.

Podpora autentizace

Jedna z největších limitací současného stavu aplikace je absence podpory autentizace. Pokud je pro komunikaci se vzdáleným repozitářem potřeba zadání hesla vzhledem k nutnosti autentizace – například přes protokoly *SSH* nebo *HTTPS* – a je v rámci aplikace vykonán příkaz inicializující právě komunikaci se vzdáleným repozitářem, tak tento příkaz selže.

Je tedy nutné mimo aplikaci zajistit, aby heslo nebylo nikdy vyžádáno. Toho lze docílit konfigurací tzv. *Credential Storage* [36], což je systém v rámci něhož se daná hesla ukládají a není je tedy třeba zadávat při každém příkazu. Existuje několik implementací – například *Git Credential Manager* pro Windows nebo *osxkeychain* pro macOS.

Jedním z možných rozšíření tedy je integrovat podporu autentizace přímo do aplikace buď ve formě podpory konfigurace Credential Storage, nebo alespoň zamezit selhání příkazů v případě, kdy je uživatel Gitem požádán o zadání hesla.

Zjednodušení konfigurace prerekvizit aplikace

Dalším vhodným rozšířením aplikace – vzhledem k tomu, že je určená zejména pro netechnické uživatele – by mohlo být zjednodušení konfigurace prerekvizit aplikace, konkrétně instalace Gitu a rozšíření Git LFS a konfigurace nástroje pro řešení konfliktů.

Toho by šlo docílit jednak integrováním podpory konfigurace zmíněných prerekvizit do aplikace, jednak zahrnutím Gitu a vybraného nástroje pro řešení konfliktů do instalace aplikace.

Editor jazyka

Dále by bylo vhodné implementovat editor usnadňující realizaci procesu v navrženém jazyce. V současnosti totiž lze správnou syntaxi zkontrolovat pouze otevřením projektu v aplikaci.

Editor by mohl být implementován buď ve formě přímé integrace do aplikace nebo ve formě rozšíření do již existujícího editoru, například *Visual Studio Code*.

Připomínky k jazyku

V rámci testování byli vzneseny některé drobné připomínky k syntaxi jazyka – například nekonzistence v používání dvojteček. Všechny připomínky byly vyhodnoceny jako opodstatněné a byly zapracovány.

Co se jazyka týče, byl vznesen ještě jeden návrh na vylepšení – a to podpora komentářů, což bylo vyhodnoceno také jako užitečná funkce, ale realizována v současnosti není. Mohlo by se tedy jednat o navazující práci.

Přidání možnosti limitace dostupnosti akcí

Další, z testování vzešlý, návrh na vylepšení jazyka bylo přidání možnosti limitace dostupnosti akcí. Může být totiž žádáno, aby ne každému uživateli byly dostupné všechny akce.

Implementaci této připomínky lze realizovat několika způsoby. Prvním způsobem je přidání podpory několika definic procesů verzování v rámci jednoho projektu, kde by každý proces byl definován ve zvláštním souboru. V aplikaci by se poté vybral soubor odpovídající definici procesu určeného pro daného člena týmu.

Dále lze tuto připomínku realizovat na úrovni samotného jazyka přidáním možnosti definice rolí a následném přidání možnosti zpřístupnění akcí pouze pro určené role. Uživatel by si tedy jednu nebo více z těchto rolí vybral a na základě této volby by mu byly nabízeny pouze jemu určené akce.

Přidání možnosti odmítnutí uživatelského vstupu

Dále jeden z testerů poukázal na fakt, že v testované verzi aplikace nebyla ošetřena situace situace, kdy byl uživatel při vykonávání akce požádán o uživatelský vstup, ale z nějakého důvodu nechtěl daný vstup zadat. Aplikace totiž neumožňovala uživatelský vstup odmítnout a uživatel tedy byl nucen buď vstup i přesto zadat, nebo vypnout aplikaci.

Tato připomínka byla zapracována jak do aplikace, tak do jazyka. V aplikaci byla zpřístupněna možnost uživatelský vstup odmítnout. Pokud je uživatelský vstup součástí stepu a je odmítnut, tak se daný step považuje za neúspěšný a inicializuje se zvolená metoda řešení selhání příkazu, viz sekce 6.3.5.

Dále byly na základě této připomínky do jazyka přidány proměnné akce, viz sekce 6.3.2, které se vyhodnocují před samotným spuštěním akce. Ve většině případů tedy může být vhodné o všechny potřebné uživatelské vstupy požádat právě na úrovni proměnných akcí. Pokud se totiž uživatelský vstup odmítne v rámci vyhodnocení proměnné akce, tak se spuštění dané akce přeruší.

Ošetření změny definice procesu

Dále bylo v rámci testování odhaleno, že není ošetřena změna definice procesu po tom, co se nad daným projektem již aplikace používala. Pokud se totiž aplikace vypne ve chvíli, kdy se uživatel nachází v procesním stavu `A` a následně se změní definice procesu tak, že se stav `A` odstraní nebo přejmenuje, tak aplikaci nad daným projektem již nelze otevřít, jelikož soubor `.gitsy/internal/currentState` stále odkazuje na stav `A`.

Tuto situaci lze vyřešit smazáním zmíněného souboru `.gitsy/internal/currentState` a následným manuálním přesunem do odpovídajícího stavu v aplikaci.

Další rozšíření aplikace by tedy mohlo být ošetření i takovéto situace například tím, že by se ve složce `.gitsy/internal` uchovávala kopie poslední

definice procesu, se kterým byla aplikace nad daným projektem otevřena. Po otevření projektu by se poté kontrolovalo, zda se aktuální definice procesu rovná té uložené. Pokud ne, tak by aplikace o tom uživatele informovala a adekvátně reagovala, například smazáním právě souboru `.gitsy/internal/currentState`.

Ošetření operací provedených mimo aplikaci

Dále bylo odhaleno, že aplikace neřeší situace, kdy se mimo aplikaci provedou Git příkazy měnící stav procesu, jako například změna větve.

Tato situace by šla ošetřit zavedením dalšího interního souboru, v rámci něhož by se například uchovávaly informace o aktuálním commitu a větvi. Po otevření projektu v aplikaci by se poté porovnala současná situace s tou, která je uložena v souboru. Pokud by se informace lišily, tak by se opět informoval uživatel a byla by mu případně nabídnuta možnost změnit stav.

Změna prezentace souborů při zamykání

Další připomínka z testování se týkala prezentace uzamykatelných souborů. Bylo zmíněno, že by vhodnější formou prezentace byla adresářová struktura, než současný jednoúrovňový seznam.

Tato připomínka byla usouzena jako adekvátní, ale zapracována v současnosti není. Jedná se tedy o možnou budoucí modifikaci aplikace.

Přidání běžně dostupných funkcionalit z ostatních Git klientů

Zejména z testování technickými uživateli vyšlo, že by do aplikace bylo vhodné implementovat běžné funkcionality z ostatních Git klientů, jako například vizualizace větví a konkrétních změn jednotlivých souborů.

Vizualizace změn by mohla být jedním z budoucích rozšíření aplikace. U přidání vizualizace větví ale bylo usouzeno, že by se tím mohl porušit jeden ze základních konceptů aplikace – tedy jednoduchost a odstínění od Gitu. Vizualizace větví by tedy případně mohla být do aplikace integrována způsobem, kdy by nebyla součástí hlavní obrazovky aplikace, nýbrž by se k ní dostali pouze uživatelé, kteří by o ní měl zájem.

Další běžně dostupná funkcionalita v ostatních Git klientech je přesun změn do oblasti odložené práce pomocí operace `stash`. Tuto funkcionalitu lze nyní

do určité míry implementovat akcí v definici procesu. V rámci aplikace by ale šla podpora odkládání práce zlepšit například zavedením dalšího typu uživatelského vstupu. Git totiž umožňuje odloženou práci pojmenovat. Nový typ uživatelského vstupu – například s identifikátorem `stashSelect` – by poté sloužil k výběru konkrétní odložené práce.

Menší odstínění od příkazů

U některých technických uživatelů bylo zpozorováno, že by pro ně bylo užitečné, kdyby aplikace poskytovala možnost zjištění konkrétních příkazů spjatých s danou akcí. Zobrazením informace o konkrétních příkazech by se ovšem opět porušil základní koncept aplikace. Stejně jako u vizualizace větví by se ale tato informace případně mohla prezentovat takovým způsobem, aby se k ní dostali pouze uživatelé, pro které je relevantní – například přidáním možnosti zobrazení detailu akce.

Přidání možnosti zobrazit více informací ve stavu

V rámci testování bylo dále upozorněno, že by se v rámci stavů mohlo zobrazovat více informací, jako například počet ještě neintegrovaných commitů ve stavu `Funkcionalita v progresu`.

Tuto funkcionalitu by bylo možné realizovat například formou rozšíření jazyka, kde by v rámci jednotlivých stavů byla možnost definovat, jaké informace by byly v rámci daného stavu relevantní. Zobrazení daných informací by poté bylo na konkrétní implementaci.

Přidání podpory více platforem

V současnosti je aplikace dostupná pouze pro platformu Windows. Podpora platforem Linux a macOS je téměř hotova, ale je třeba vyřešit některé menší nedostatky, takže instalátory pro dané platformy ještě nejsou zpřístupněny.

Přidání podpory v herních enginech

Bylo také navrhnuto vytvoření implementace ve formě rozšíření do konkrétních herních enginů. Implementace takových rozšíření by mohla případná navazující práce.

8.4 Zhodnocení

Z poznatků získaných z uživatelského testování lze učinit určité závěry jak o navrženém jazyku, tak o implementované aplikaci.

Celkový koncept, včetně navrženého jazyka, lze zhodnotit kladně. Jak techničtí, tak netechničtí uživatelé většinou usoudili, že má daný koncept potenciál a mohl by ve vývoji her najít uplatnění. Co se čistě jazyka týče, bylo vzneseno několik připomínek a návrhů na vylepšení. Některé z připomínek byly zapracovány. Mezi nejzásadnější nezapracované připomínky týkající se jazyka lze zařadit absenci možnosti limitace dostupnosti akcí a absenci editoru jazyka.

Co se aplikace týče, tak zejména z testování vzešlo několik připomínek a možností rozšíření. Některé byly zapracovány a pro ty, které zapracovány nebyly, bylo navrhnuto řešení a jejich realizace by mohla být navazující prací. Zhodnocení použitelnosti aplikace je nutné rozdělit na technické a netechnické uživatele. Názory technických uživatelů lze shrnout tak, že by aplikaci nejspíše nepoužívali ani při implementaci některých zmíněných rozšíření. Je ale třeba zmínit, že všichni techničtí účastníci měli velmi dobré zkušenosti s Gitem a nejsou tedy cílovou skupinou.

Většina netechnických účastníků by po implementaci některých rozšíření naopak používání aplikace zvážila. Nejkladněji aplikaci hodnotili ti netechničtí uživatelé, kteří mají zkušenosti s Gitem a ocenili tedy odstínění od komplexity Gitu.

Na aplikaci tedy lze v současné době nahlížet jako na referenční implementaci navrženého jazyka s tím, že by po implementaci některých rozšíření a odstranění limitací měla v rámci herního vývoje potenciál, zejména u cílové skupiny – tedy netechnických uživatelů. Potenciál by se mohl ještě zvýšit vytvořením implementace ve formě rozšíření do herních enginů nebo nástrojů často používaných netechnickými profesemi v rámci herního vývoje.

Dále lze z testování učinit některé závěry o navrženém procesu. Techničtí uživatelé neměli s plněním scénáře v druhé části testování žádný problém. Netechnickým účastníkům ale dělalo menší problém pochopit některé koncepty daného procesu verzování. Po krátkém dovysvětlení se ale vše vyjasnilo. Těmto problémům se ale lze snadno vyhnout tím, že se každý člen nejdříve obeznámí se zavedenou terminologií v rámci procesu verzování.

9 Závěr

První dvě kapitoly se týkají představení problematiky verzování softwaru. Nejprve byla představena obecná problematika verzování softwaru a následně proběhla analýza specifik vývoje videoher týkajících se verzování softwaru pomocí nástroje Git. Pro jednotlivá specifika byla představena možná řešení. V rámci této analýzy byl proveden i průzkum za účelem zjištění dodatečných informací a současného stavu verzování v herním vývoji.

Dále se práce zaměřila na procesy verzování. Nejprve byly představeny běžně používané prvky verzování, na jejichž základě se navrhl – v kombinaci s předchozí analýzou specifik verzování ve vývoji videoher – proces verzování pro vývoj videoher.

V následující kapitole proběhla analýza jazyků vhodných pro popis verzovacích procesů. Bylo usouzeno, že žádný z existujících jazyků zcela nesplňuje kladené požadavky a byl tedy navržen jazyk nový. Tento jazyk byl implementován Git klientem s grafickým uživatelským rozhraním, jehož implementace je také součástí této práce.

Nakonec proběhlo zhodnocení řešení, zejména ve formě uživatelského testování, ze kterého vzešlo, že celkový koncept, včetně navrženého jazyka, má potenciál a mohl by ve vývoji her najít uplatnění. Hodnocení samotné aplikace z pohledu použitelnosti cílovou skupinou – tedy netechnickými uživateli – lze také považovat za relativně kladné. Aplikace všem uživatelům přišla intuitivní a její používání by většina netechnických respondentů – po vyřešení některých současných limitací a implementaci některých rozšíření – zvážila.

V rámci testování bylo vzneseno několik připomínek a návrhů na vylepšení jak pro jazyk, tak pro aplikaci. Některé byly zapracovány a pro ty, které zapracovány nebyly, bylo navrhnuto řešení a jejich realizace by tedy mohla být případnou navazující prací.

Uživatelské testování probíhalo s navrženým procesem pro vývoj videoher a lehce se tedy týkalo i zhodnocení samotného procesu. Všichni dotázaní respondenti hodnotili proces kladně a neměli k němu žádné připomínky.

Seznam obrázků

Seznam tabulek

Literatura

[1] *Forking Workflow* [online]. Atlassian Corporation. [cit. 2023/05/15]. Dostupné z: https://www.atlassian.com/git/tutorials/comparing-workflows/forking-workflow.

[2] *Git Large File Storage (LFS)* [online]. Atlassian Corporation. [cit. 2023/04/26]. Dostupné z: https://www.atlassian.com/git/tutorials/git-lfs.

[3] *What is version control?* [online]. Atlassian Corporation. [cit. 2023/03/21]. Dostupné z: https://www.atlassian.com/git/tutorials/what-is-version-control.

[4] *Use Git LFS with Bitbucket* [online]. Atlassian Corporation. [cit. 2023/04/26]. Dostupné z: https://support.atlassian.com/bitbucket-cloud/docs/use-git-lfs-with-bitbucket/.

[5] *CVS — Concurrent Versions System* [online]. Free Software Foundation. [cit. 2023/04/16]. Dostupné z: https://www.gnu.org/software/trans-coord/manual/cvs/cvs.html.

[6] *DOT Language* [online]. AT&T Labs. [cit. 2023/05/29]. Dostupné z: https://www.graphviz.org/doc/info/lang.html.

[7] *The benefits of a distributed version control system* [online]. GitLab Inc. [cit. 2023/04/18]. Dostupné z: https://about.gitlab.com/topics/version-control/benefits-distributed-version-control-system/.

[8] *GUI Clients* [online]. Git-Scm.com. [cit. 2023/05/01]. Dostupné z: https://git-scm.com/downloads/guis/.

[9] *Git Extensions* [online]. Git Extensions. [cit. 2023/06/02]. Dostupné z: https://gitextensions.github.io/.

[10] *Configuring Git Large File Storage* [online]. GitHub, Inc. [cit. 2023/04/26]. Dostupné z: https://docs.github.com/en/repositories/working-with-files/managing-large-files/configuring-git-large-file-storage.

[11] *GitKraken Client* [online]. GitKraken. [cit. 2023/06/02]. Dostupné z: https://www.gitkraken.com/git-client.

[12] *Git Large File Storage (LFS)* [online]. GitLab Inc. [cit. 2023/04/25]. Dostupné z: https://docs.gitlab.com/ee/topics/git/lfs/.

[13] *Google docs - Find what's changed in a file* [online]. Google LLC. [cit. 2023/03/22]. Dostupné z: https://support.google.com/docs/answer/190843.

[14] *Code Review Developer Guide* [online]. Google LLC. [cit. 2023/05/20]. Dostupné z: https://google.github.io/eng-practices/review/.

[15] *IntelliJ IDEA – the Leading Java and Kotlin IDE* [online]. JetBrains s.r.o. [cit. 2023/06/02]. Dostupné z: https://www.jetbrains.com/idea/.

[16] *Manage changelists* [online]. JetBrains s.r.o. [cit. 2023/06/02]. Dostupné z: https://www.jetbrains.com/help/idea/managing-changelists.html.

[17] *What is a CI Server?* [online]. JetBrains s.r.o. [cit. 2023/05/17]. Dostupné z: https://www.jetbrains.com/teamcity/ci-cd-guide/ci-cd-tools/servers/.

[18] *What is Jira Software?* [online]. Atlassian Corporation. [cit. 2023/05/24]. Dostupné z: https://www.atlassian.com/software/jira/guides/getting-started/introduction.

[19] *Newzoo Global Games Market Report 2021* [online]. Newzoo. [cit. 2023/04/29]. Dostupné z: https://newzoo.com/resources/trend-reports/newzoo-global-games-market-report-2021-free-version.

[20] *OMG Unified Modeling Language* [online]. Object Management Group. [cit. 2023/05/30]. Dostupné z: https://www.omg.org/spec/UML/2.2/Superstructure/PDF.

[21] *What is open source?* [online]. OpenSource.com. [cit. 2023/05/18]. Dostupné z: https://opensource.com/resources/what-open-source.

[22] *Perforce Helix Core Pricing and Plans* [online]. Perforce Software, Inc. [cit. 2023/05/10]. Dostupné z: https://www.perforce.com/resources/vcs/helix-core-pricing.

[23] *Qt Framework* [online]. The Qt Company. [cit. 2023/06/02]. Dostupné z: https://www.qt.io/product/framework.

[24] *Qt SCXML Overview* [online]. The Qt Company. [cit. 2023/05/30]. Dostupné z: https://doc.qt.io/qt-6/qtscxml-overview.html.

[25] *GNU RCS* [online]. Free Software Foundation. [cit. 2023/03/22]. Dostupné z: https://www.gnu.org/software/rcs/rcs.html.

[26] *Chapter 5 SCCS Source Code Control System* [online]. Oracle Corporation. [cit. 2023/03/22]. Dostupné z: https://docs.oracle.com/cd/E19504-01/802-5880/6i9k05dhp/index.html.

[27] *State Chart XML (SCXML): State Machine Notation for Control Abstraction* [online]. W3C. [cit. 2023/05/30]. Dostupné z: https://www.w3.org/TR/scxml/.

[28] *2022 Developer Survey* [online]. Stack Overflow. [cit. 2023/04/23]. Dostupné z: https://survey.stackoverflow.co/2022/.

[29] *SonarQube Documentation* [online]. SonarSource. [cit. 2023/05/17]. Dostupné z: https://docs.sonarqube.org/latest/.

[30] *Blueprints Visual Scripting* [online]. Epic Games. [cit. 2023/06/18]. Dostupné z: https://docs.unrealengine.com/5.0/en-US/blueprints-visual-scripting-in-unreal-engine/.

[31] *Page history* [online]. Wikimedia Foundation, Inc. [cit. 2023/03/22]. Dostupné z: https://en.wikipedia.org/wiki/Help:Page_history.

[32] Appleton, B. et al. Streamed Lines: Branching Patterns for Parallel Software Development. *PLoP '98 conference*. 1998. Dostupné z: https://ntrs.nasa.gov/citations/19860021622.

[33] Arbel, Y. *Unleashing Git for the Game Development Industry* [online]. The New Stack. [cit. 2023/04/30]. Dostupné z: https://thenewstack.io/unleashing-git-for-the-game-development-industry/.

[34] Austin, J. Git for games: current problems and solutions. Dostupné z: https://www.youtube.com/watch?v=K3zOhU3NdWA. Git Merge, 2019.

[35] Baum, T. et al. A Faceted Classification Scheme for Change-Based Industrial Code Review Processes. In *2016 IEEE International Conference on Software Quality, Reliability and Security (QRS)*, s. 74–85, 2016. doi: 10.1109/QRS.2016.19.

[36] Chacon, S. *Pro Git, 2nd Edition*. Apress, 2014. ISBN 978-1484200773.

[37] Daffurn, R. W. *Process of a painting: "Glen Gillard"* [online]. [cit. 2023/05/04]. Dostupné z: https://www.rwdaffurn.com/blog/2016/10/7/process-of-a-painting.

[38] Driessen, V. *Gitflow* [online]. [cit. 2023/05/28]. Dostupné z: https://github.com/nvie/gitflow.

[39] Forsgren, N. – Humble, J. – Kim, G. *Accelerate*. IT Revolution Press, 2018. ISBN B07B9F83WM.

[40] Fowler, M. *Patterns for Managing Source Code Branches* [online]. [cit. 2023/05/16]. Dostupné z: https://martinfowler.com/articles/branching-patterns.html.

[41] Fowler, M. *Continuous Delivery* [online]. [cit. 2023/05/21]. Dostupné z: https://martinfowler.com/bliki/ContinuousDelivery.html.

[42] Fowler, M. *Continuous Integration* [online]. [cit. 2023/05/17]. Dostupné z: https://martinfowler.com/articles/continuousIntegration.html.

[43] Fowler, M. *Domain-Specific Languages*. Addison-Wesley Professional, 2010. ISBN 978-0321712943.

[44] Fowler, M. *Pair Programming* [online]. [cit. 2023/05/20]. Dostupné z: https://martinfowler.com/bliki/PairProgramming.html.

[45] Hammant, P. *Trunk-Based Development And Branch By Abstraction*. Vydáno samostatně, 2023.

[46] Hilton, R. *A branching strategy simpler than GitFlow: Three-Flow* [online]. [cit. 2023/05/21]. Dostupné z: https://www.rodhilton.com/2017/04/09/a-different-branching-strategy/.

[47] Hodgson, P. *Feature Toggles* [online]. [cit. 2023/05/18]. Dostupné z: https://martinfowler.com/articles/feature-toggles.html.

[48] Hudson, G. *The benefits of a distributed version control system* [online]. [cit. 2023/04/22]. Dostupné z: http://web.mit.edu/ghudson/thoughts/undiagnosing.

[49] Kahn, S. *Git Source Control for game development* [online]. [cit. 2023/05/01]. Dostupné z: https://kahncode.com/2019/11/05/git-source-control-for-game-development/.

[50] Lischke, M. *ANTLR4 grammar syntax support* [online]. [cit. 2023/06/02]. Dostupné z: https://marketplace.visualstudio.com/items?itemName=mike-lischke.vscode-antlr4.

[51] Okamoto, Y. *Support for Locking feature of Git LFS* [online]. [cit. 2023/04/29]. Dostupné z: https://jira.atlassian.com/browse/SRCTREE-4663.

[52] Parr, T. *ANTLR* [online]. [cit. 2023/06/02]. Dostupné z: https://www.antlr.org/.

[53] Parr, T. *Getting Started with ANTLR v4* [online]. [cit. 2023/06/02]. Dostupné z: https://github.com/antlr/antlr4/blob/master/doc/getting-started.md.

[54] Potvin, R. – Levenberg, J. Why Google Stores Billions of Lines of Code in a Single Repository. *Commun. ACM.* jun 2016, 59, 7, s. 78–87. ISSN 0001-0782. doi: 10.1145/2854146. Dostupné z: https://doi.org/10.1145/2854146.

[55] Santacroce, F. *Git Essentials - Second Edition.* Packt Publishing, 2017. ISBN 9781787120723.

[56] Sharma, P. *Git Working Areas* [online]. Knoldus Inc. [cit. 2023/04/25]. Dostupné z: https://blog.knoldus.com/git-working-areas/.

[57] Sink, E. *Version Control by Example.* Pyrenean Gold Press, 2011. ISBN 978-0983507901.

[58] Toftedahl, M. *Which are the most commonly used Game Engines?* [online]. Game Developer. [cit. 2023/04/30]. Dostupné z: https://www.gamedeveloper.com/production/which-are-the-most-commonly-used-game-engines-.

[59] Dijck, T. *Support for compressing files in the lfs folder?* [online]. [cit. 2023/04/28]. Dostupné z: https://github.com/git-lfs/git-lfs/issues/260.

[60] Weimann, J. *Unity3D HowTo: Avoid Scene Merge Conflicts with GIT* [online]. [cit. 2023/05/25]. Dostupné z: https://www.youtube.com/watch?v=YgoCp2tzRh0.

[61] Yurichev, D. *Some of git internals* [online]. Dennis Yurichev. [cit. 2023/04/24]. Dostupné z: https://yurichev.com/news/20201220_git/.

A Uživatelská příručka

Uživatelská příručka jazyka pro popis verzovacích procesů

V rámci této sekce bude poskytnut základní popis jazyka sloužícího pro definici procesů verzování. Jednoduché příklady realizací procesů v daném jazyce lze nalézt na cestě `Vstupni_data/Realizace_procesu`.

Základní konstrukce

Command

Command představuje právě jeden příklad. Syntaxe konstrukce command je znázorněna na jednoduchém příkladu vytvoření commitu, viz fragment kódu A.1.

```
git "commit -am \"Commit zprava\"";
```

Fragment kódu A.1: Syntaxe jazykové konstrukce command.

Step

Hlavním účelem konstrukce step je zastřešení několika konstrukcí command. Jeden step je poté považován za atomickou operaci – počítá se tedy s tím, že proběhnou buď všechny zastřešené příkazy, nebo žádný. Základní syntaxe je znázorněna ve fragmentu kódu A.2.

```
commitAndPush() {
    commands [
        git "commit -am \"Commit zprava\"";
        git "push";
    ]
}
```

Fragment kódu A.2: Syntaxe jazykové konstrukce step.

Action

Konstrukce action již představuje spustitelnou akci, u které se předpokládá, že bude určitým způsobem zpřístupněna uživateli. Základní definice akce se skládá ze tří částí – posloupnost stepů, vlastnosti zobrazení a reference na následující stav.

```
dokoncitFunkcionalitu {
    steps [
        commitAllAndPush {}
        checkoutMaster {}
    ]
    display {
        text: "Dokoncit funkcionalitu";
    }
    nextState: "vychozi";
}
```

Fragment kódu A.3: Syntaxe jazykové konstrukce action.

State

State představuje jeden stav procesu, v rámci něhož se definují dostupné akce a vlastnosti zobrazení. Jeden z těchto stavů je nutné označit jako výchozí nastavením hodnoty atributu `initialState`. Zjednodušený příklad neúplné definice dvou stavů je uveden ve fragmentu kódu A.4.

```
statesSettings {
    states [
        /* Definice prvniho stavu */
        vychozi {
            actions [
                zacitNovouFunkcionlaitu {
                    steps [ ... ]
                    display { ... }
                    nextState: "novaFunkcionalita";
                }
            ]
            display {
                text: "Vychozi stav";
            }
        }

        /* Definice druheho stavu */
        novaFunkcionalita {
            actions [ ... ]
            display {
                text: "Nova funkcionalita";
            }
        }
    ]
    initialState : "vychozi";
}
```

Fragment kódu A.4: Syntaxe jazykové konstrukce state.

Proměnné a parametry

Jazyk umožňuje v rámci procesu definovat a referencovat několik typů proměnných. Dále je umožněno parametrizovat jednotlivé stepy. Proměnné a parametry lze referencovat obalením identifikátoru proměnné mezi kombinaci znaků `${` a `}`. Příkladem reference proměnné tedy může být `${promenna}`. Jednotlivé typy proměnných a parametry stepů jsou detailněji popsány dále.

Globální proměnné

Hodnoty globálních proměnných definuje autor procesu verzování a jsou tedy pro každého člena týmu stejné. Syntaxe definování globálních proměnných je znázorněna ve fragmentu kódu A.5.

```
globalVariables [
    prefixTematickeVetve: "feature/";
    hlavniVyvojovaLinie: "master";
]
```

Fragment kódu A.5: Definice globálních proměnných.

Uživatelské proměnné

Uživatelské proměnné umožňují přizpůsobit určité prvky procesu konkrétnímu uživateli. Uživatelské proměnné mají smysl pouze v kombinaci s uživatelskými vstupy, viz dále. U těchto proměnných je také očekáváno, že je bude možné měnit – je k nim tedy přidán popis, který se případně může zobrazit v uživatelském rozhraní místo identifikátoru. Příklad je uveden ve fragmentu kódu A.6.

```
userVariables [
    username {
        value: input(type: "text", message: "Zadejte Vase jmeno");
        description: "Uzivatelske jmeno";
    }
]
```

Fragment kódu A.6: Definice uživatelské proměnné.

Proměnné akce

Proměnné akce se definují na úrovni akce, v rámci níž jsou poté dostupné. Tento typ proměnných má opět význam spíše v kombinaci s použitím funkcí umožňující uživatelský vstup. Příklad definice a referencování proměnné akce je uveden ve fragmentu kódu A.7.

```
novaFunkcionalita {
    actionVariables [
        nazevVetve: input(type: "text", message: "Zadejte nazev: ");
    ]
    steps [
        vytvorVetev {
            vetev: ${nazevVetve};
        }
    ]
    display { ... }
    nextState: "dalsiStav";
}
```

Fragment kódu A.7: Definice uživatelské proměnné.

Systémové akce

Systémové akce umožňují aplikaci předávat procesu verzování různé aplikace. Implementovaná aplikace poskytuje následující proměnné:

- `Ui.selectedUnstagedFiles`: Vybrané `unstaged` soubory.
- `Ui.selectedStagedFiles`: Vybrané `staged` soubory.
- `StepContext.startBranch`: Větev, která byla větví aktuální v době začátku vykonávání daného stepu. Tato proměnná je tedy dostupná pouze v rámci stepů.
- `StepContext.startCommit`: SHA1 hash commitu, který byl commitem aktuálním v době začátku vykonávání daného stepu. Tato proměnná je tedy dostupná pouze v rámci stepů.
- `ActionContext.startBranch`: Větev, která byla větví aktuální v době začátku vykonávání dané akce. Tato proměnná je tedy dostupná pouze na úrovni akce.

- `ActionContext.startCommit`: SHA1 hash commitu, který byl commitem aktuálním v době začátku vykonávání dané akce. Tato proměnná je tedy dostupná pouze na úrovni akce.

Parametry stepů

Jednotlivé stepy lze parametrizovat. Parametry se uvádějí do hlavičky daného stepu a následně je lze referencovat jako klasické proměnné. Příklad definice parametru a jeho následné použití je uveden ve fragmentu kódu A.8. Příklad použití parametrizovaného stepu z akce je poté znázorněn ve fragmentu kódu A.9.

```
commitVseho(popisCommitu) {
    commands [
        git "commit -am \"" + ${popisCommitu} + "\"";
    ]
}
```

Fragment kódu A.8: Příklad parametrizace stepu.

```
akceCommitVseho {
    steps [
        commitVseho {
            popisCommitu: "Predani parametru";
        }
    ]
    display { ... }
    nextState: "nasledujiciStav";
}
```

Fragment kódu A.9: Příklad použití parametrizovaného stepu v rámci akce.

Uživatelské vstupy

Aplikace podporuje celkem tři typy uživatelských vstupů, které jsou popsány dále.

Uživatelům je při vyžádání vstupu umožněno tento požadavek odmítnout. Pokud je uživatelský vstup odmítnut během vykonávání akce – například při vyžádání zadání názvu větve v rámci stepu – a uživatel požadavek odmítne, daný step bude v tomto případě pokládán za nezdařilý a inicializuje se vybraná metoda pro řešení selhání příkazu.

Ve většině případů je tedy vhodné řešit uživatelské vstupy potřebné pro akci již na úrovni proměnných akce. Pokud totiž uživatel odmítne požadavek na vstup při vyhodnocování proměnné akce, daná akce se nezačne vykonávat.

Textový vstup

Ve fragmentu kódu A.10 je uveden příklad textového vstupu. Parametry `type` a `message` jsou povinné. Pomocí volitelného parametru `inputPattern` lze vstup limitovat ve formě regulárního výrazu.

```
input(type: "text", message: "Zadejte nazev vetve",
       inputPattern: "[0-9A-Za-z/]*")
```

Fragment kódu A.10: Příklad textového uživatelského vstupu.

Výběr větve

Ve fragmentu kódu A.11 je uveden příklad vstupu pro výběr jedné z existujících větví. Parametry `type` a `message` jsou povinné. Volitelný parametr `pattern` slouží pro filtraci větví, které jsou uživateli následně dány na výběr. Nastaví-li se tento parametr například na `feature/*`, tak se uživateli nabídnou pouze větve, které danému vzoru odpovídají.

Dalším parametrem je parametr `omitPart`. Tímto parametrem lze z jednotlivých větví, než se uživateli nabídnou k výběru, odstranit část názvu. Projde-li například dříve zmíněným filtrem větev `origin/feature/funkcionalita1` s nastaveným parametrem `omitPart` na hodnotu `origin/`, tak se uživateli nenabídne větev s celým názvem, ale část `origin/` se ořízne. Uživateli je tedy ve výsledku nabídnuta větev `feature/funkcionalita1`. Tohoto parametru lze využít například právě pro případy, kdy není žádáno, aby se uživateli nabízely vzdálené větve.

```
input(type: "branchSelect", message: "Vyberte funkcionalitu",
   pattern: "feature/*", omitPart: "origin/")
```

Fragment kódu A.11: Příklad výběru větve.

Výběr tagu

Ve fragmentu kódu A.12 je uveden příklad vstupu pro výběr jedné z existujících tagů. Volitelný parametr `pattern` slouží k filtraci tagů, které jsou

uživateli následně dány na výběr. Příklad použití je uveden ve fragmentu kódu 7.3.

```
input(type: "tagSelect", message: "Vyberte tag",
    pattern: "bugfix/*")
```

Fragment kódu A.12: Příklad výběru tagu.

Podmínky akcí

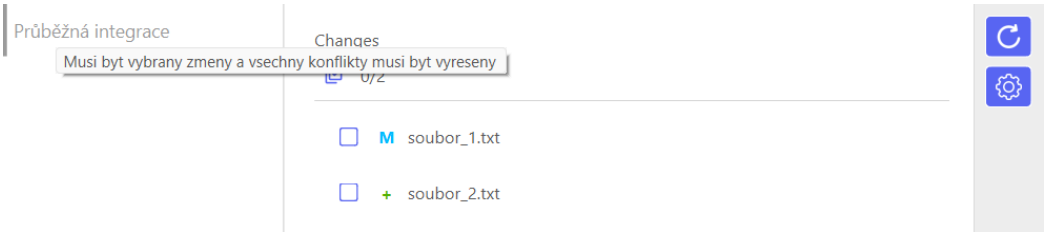
Dostupnost jednotlivých akcí lze podmínit. Aplikace poskytuje celkem tři operandy, které lze libovolně kombinovat do výsledného logického výrazu:

- `changesExist`: nabývá hodnoty `true` v případě, kdy má uživatel ve svém repozitáři nějaké změny buď v pracovním adresáři, nebo v oblasti připravených změn.
- `selectedChangesExist`: nabývá hodnoty `true` v případě, kdy je v seznamu změn alespoň jedna změna vybrána.
- `conflictsExist`: nabývá hodnoty `true` v případě, kdy v daném čase existují nevyřešené konflikty.

```
prubeznaIntegrace {
    steps [ ... ]
    display {
        text: "Prubezna integrace";
    }
    condition {
        expression: selectedChangesExist && !conflictsExist;
        description: "Musi byt vybrany zmeny a vsechny konflikty
            musi byt vyreseny";
    }
    nextState: "nasledujiciStav";
}
```

Fragment kódu A.13: Příklad definice podmínky akce.

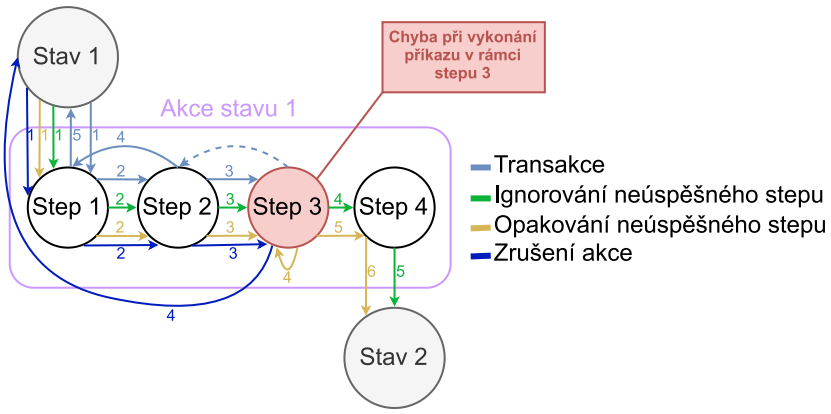
Příklad použití podmínek v rámci akce je uvedena ve fragmentu kódu A.13 a příklad nesplnění této podmínky v aplikaci je poté uveden na obrázku A.1.



Obrázek A.1: Příklad nesplněné podmínky v aplikaci.

Řešení selhání příkazu

Jazyk poskytuje celkem čtyři možnosti, jak se na úrovni akce vypořádat s případným selháním jednoho z příkazů. Všechny možnosti jsou vizualizovány na obrázku A.2.



Obrázek A.2: Možností pro řešení selhání příkazu.

Pro zvolení konkrétní možnosti slouží atribut akce `executionPolicy`, kterému lze nastavit odpovídající identifikátor. Jedná se konkrétně o:

- **Zrušení akce**: Identifikátor akce je `abortOnFailure` a jedná se o výchozí metodou řešení selhání příkazů. Tato metoda zajišťuje, že se v případě selhání příkazu následující stepy již nevykonají a neprovede se přechod do dalšího stavu.

- **Ignorování selhání**: Identifikátor akce je `continueOnFailure`. Případné selhání příkazu se ignoruje a pokračuje se dalším stepem.

- **Opakování neúspěšného stepu**: Identifikátor akce je `repeatStepOnFailure`. Metoda zajišťuje opakování stepu, v rámci něhož step selhal.

- **Transakce**: Identifikátor akce je `transactional`. Transakční akce umožňují v případě selhání jednoho ze stepů provést tzv. *rollback*, tedy navrácení do stavu před spuštěním dané akce.

  Aby šlo takového chování docílit, je třeba umožnit, aby šly jednotlivé stepy v rámci akce zvrátit – tedy umožnit definici inverzních příkazů k příkazům provádějících se při vykonávání daného stepu. K definici inverzních příkazů slouží atribut `reverseCommands`.

  Ve fragmentu kódu A.14 je uveden příklad jednoduchého reverzibilního stepu přidávající soubory do oblasti připravených změn a následného vytvoření commitu. V rámci inverzních příkazů jsou poté uvedeny příkazy, které nejdříve vytvořený commit odstraní a následně se vybrané soubory vrátí z oblasti připravených změn do oblasti pracovního adresáře. V příkladu je také naznačena důležitá informace – inverzní příkazy se musí uvádět v opačném pořadí, než příkazy původní.

```
pridejAVytvorCommit(soubory) {
    commands [
        git "add " + ${soubory}; // 1. prikaz
        git "commit -m \"commit zprava\""; // 2. prikaz
    ]
    reverseCommands [
        git "reset --soft HEAD~1"; // inverzni prikaz k
                                   // 2. prikazu
        git "restore --staged " + ${soubory}; // inverzni
                                     // prikaz k 1. prikazu
    ]
}
```

Fragment kódu A.14: Příklad reverzibilního stepu.

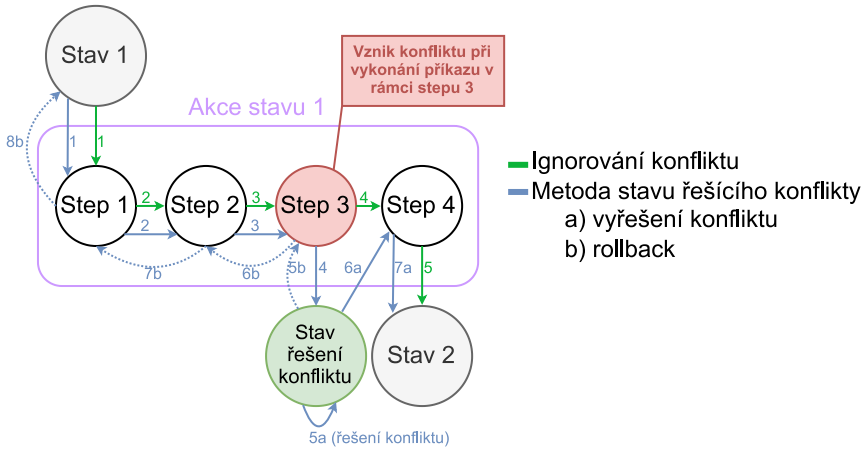
Transakční akce poté v případě selhání jednoho ze stepů zajišťují postupné provolání inverzních příkazů předchozích stepů. Zároveň zamezí přechodu do následujícího stavu. Transakční akce je znázorněna také na obrázku A.2, kde se po vyskytnutí chyby ve stepu 3 vyvolá postupný rollback od stepu 2. Inverzní příkazy se tedy neprovádějí od stepu, v rámci něhož příkaz selhal, nýbrž od stepu předchozího.

Je nutno zmínit, že při definování reverzibilních stepů je třeba dbát zvýšené opatrnosti, aby definované reverzní příkazy byly ve všech pří-

padech opravdu inverzní k původním příkazům.

Řešení konfliktů

Jazyk poskytuje dvě možnosti řešení konfliktu. Pro zvolení konkrétní možnosti slouží atribut akce `conflictResolutionPolicy`. Obě metody jsou vizualizovány na obrázku A.3 a jsou dále popsány.



Obrázek A.3: Vizualizace možností pro řešení konfliktů v rámci akcí.

Ignorování konfliktu

S touto metodou je spjat identifikátor `continue`. V případě této metody je konflikt ignorován a pokračuje se v akci.

Metoda stavu řešení konfliktů

Identifikátorem této možnosti je `stopAndResolve`. Druhou možností je zavedení stavu, do kterého se proces přesune v případě vzniku konfliktu. V rámci tohoto stavu je poté aplikací umožněno konflikt vyřešit. Po vyřešení konfliktu je možno pokračovat stepem bezprostředně následujícím po problematickém stepu, případně přechodem do následujícího stavu, pokud je problematický step v rámci akce stepem posledním. Jedná se o výchozí metodu.

Tok metody je v příkladu na obrázku A.3 znázorněn modrou barvou. Lze vidět, že je tok od stavu řešení konfliktu rozdělen na dva. První tok, v obrázku označen písmenem *a*, vizualizuje případ popsaný v předchozím odstavci – v rámci stavu řešení konfliktu se konflikt vyřeší a pokračuje se dalším stepem. V případě transakčních akcí, viz předchozí sekce, lze v rámci tohoto stepu

inicializovat i rollback, pokud například uživatel nechce daný konflikt řešit. Tok rollbacku je na obrázku označen písmenem *b*.

K rollbacku ze stavu řešení konfliktů je třeba přistupovat jinak, než v případě řešení selhání příkazů. Jak lze vidět na obrázku A.3, rollback je inicializován již od problematického stavu, nikoliv až od stepu předchozího, jako tomu je u transakcí v rámci řešení selhání příkazu.

V rámci stepu, který je reverzibilní a zároveň může zapříčinit konflikt, se tedy musí počítat s tím, že může být součástí dvou rollback scénářů. První scénář se netýká konfliktů, nýbrž selhání příkazů v rámci následujících stepů. V tomto případě se provedou příkazy definované v atributu `reverseCommands`. Druhý scénář nastává v případě, kdy daný step způsobil konflikt a ve stavu řešení konfliktu se následně inicializoval rollback, viz již zmíněný tok označený písmenem *b* na obrázku A.3. V tomto případě nelze použít příkazy `reverseCommands` určené pro scénář selhání příkazu, ale je nutno definovat další sadu příkazů obstarávající zotavení se z konfliktu. Pro tyto příkazy slouží atribut `conflictReverseCommands`.

Příklad reverzibilního stepu, v rámci něhož může zároveň vzniknout konflikt, je uveden ve fragmentu kódu A.15, kde lze vidět, že se příkazy pro oba scénáře liší. V případě `reverseCommands` se totiž počítá s tím, že step proběhl úspěšně a slučovací commit byl vytvořen, takže je potřeba jej odstranit. Naopak, v případě `conflictReverseCommands` slučovací commit vytvořen ještě nebyl. Je ale třeba zajistit zrušení operace `merge`.

```
slucDoMainline() {
    commands [
        git "checkout master";
        git "merge --no-ff" + ${StepContext.startBranch};
    ]
    reverseCommands [
        git "reset --hard HEAD~1";
        git "checkout " + ${StepContext.startBranch};
    ]
    conflictReverseCommands []
        git "merge --abort";}
        git "checkout " + ${StepContext.startBranch};
    ]
}
```

Fragment kódu A.15: Příklad reverzibilního stepu pro sloučení větví.

Podpora Git LFS zamykání

Do jazyka je zabudována podpora zamykání pomocí Git LFS ve formě možnosti specifikování, jaké soubory se mohou zamykat. Pomocí této informace poté aplikace může uživateli zobrazit seznam souborů k uzamčení a odemčení. Jazyk podporuje celkem dvě možnosti specifikování uzamykatelných souborů.

Čtení souboru .gitattributes

První možnost je založena na čtení atributů `lockable` ze souboru `.gitattributes` v kořenovém adresáři projektu, což je jednoduchý soubor přiřazující souborům atributy. V jazyce se tato možnost zvolí pomocí identifikátoru `gitattributes`, viz fragment kódu A.17. Jednoduchý příklad souboru `.gitattributes` je uveden ve fragmentu kódu A.16, kde jsou atributem `lockable` označeny všechny soubory ve formátu `bmp` a `jpg`.

```
*.bmp lockable
*.jpg lockable
```

Fragment kódu A.16: Příklad souboru .gitattributes.

```
lfs {
    locks {
        lockableFilePatternsSource: "gitattributes";
    }
}
```

Fragment kódu A.17: Zvolení metody čtení souboru .gitattributes.

Manuální specifikace

Další možností je manuální specifikace výčtu uzamykatelných souborů. V tomto případě se tedy nečte soubor `.gitattributes`, ale výčet uzamykatelných souborů je přímou součástí definice procesu. Této možnosti je přidělen identifikátor `custom`. Příklad zvolení této metody je uveden ve fragmentu kódu A.18.

```
lfs {
    locks {
        lockableFilePatternsSource: "custom";
        lockableFilePatterns ["*.bmp", "*.jpg"]
    }
}
```

Fragment kódu A.18: Zvolení metody manuální specifikace uzamykatelných souborů.

Uživatelská příručka aplikace

Instalace aplikace

Pro platformu Windows byl vytvořen instalátor, který tedy stačí spustit a následovat kroky zobrazené instalátorem. Ostatní platformy v současné době nejsou podporovány.

Prerekvizity aplikace

Instalace Git a Git LFS

Aplikace předpokládá, že budou v systému nainstalovány nástroje Git a případně Git LFS. Nástroje lze stáhnout na následujícím odkazu `https://git-scm.com/downloads`.

Pro instalaci Git LFS je následně potřeba spustit následující příkaz v příkazové řádce: `git lfs install`.

Nastavení autentizace

Jedna z největších limitací současného stavu aplikace je absence podpory autentizace. Je tedy mimo aplikaci nutné zajistit, aby v rámci aplikace nikdy nebylo nutné zadání hesla přes příkazovou řádku při komunikaci se vzdáleným repozitářem.

Toho lze docílit konfigurací tzv. *Credential Storage*, čemuž se detailně věnuje následující článek `https://git-scm.com/book/en/v2/Git-Tools-Credential-Storage`.

Nastavení nástroje pro řešení konfliktů

Dále lze volitelně konfigurovat nástroj pro řešení konfliktů. Konfigurace bude předvedena na nástroji meld, který je volně dostupný ke stažení na následujícím odkazu `https://meldmerge.org/`.

Základní konfigurace nástroje se poté skládá z několika příkazů příkazové řádky, které jsou, včetně vysvětlivek, uvedeny ve fragmentu kódu A.19. Příkazy je nutné pouštět buď přímo v daném repozitáři, nebo je možné všechny příkazy pustit s přepínačem `--global`, čímž se zajistí, že se daná konfigurace aplikuje v rámci celého systému.

```
// Konfigurace cesty k nastroji meld
git config --global mergetool.meld.path "<cesta-k-nastroji-meld>"
// Nastaveni nastroje meld jako vychozi nastroj pro reseni konfliktu
git config --global merge.tool meld
// Vypnuti nutnosti potvrdit otevreni externiho nastroje
git config --global mergetool.prompt false
// Povoleni duvery ve vystupni kodu nastroje ve vsech pripadech
git config mergetool.meld.trustExitCode true
// Odstraneni pomocnych souboru po ukonceni nastroje
git config mergetool.keepbackup false
```

Fragment kódu A.19: Konfigurace nástroje pro řešení konfliktu.

Konfigurace projektu

Aby bylo možné projekt bez problému v aplikaci otevřít, musí autor definice procesu nejprve provést několik kroků.

Prvním krokem je vytvoření definice procesu v jazyku pro popis verzovacích procesů. Tento soubor je poté nutné umístit do složky `.gitsy/process.gitsy`, kde je cesta uvedena relativně ke kořenovému adresáři projektu. U tohoto souboru se očekává, že bude zahrnut ve verzování, aby jej měli k dispozici všichni členové týmu.

Aplikace si v průběhu otvírání projektu – a následných spouštění akcí – udržuje pomocné soubory ve složce `.gitsy/internal`, jejíž cesta je uvedena relativně ke kořenovému adresáři repozitáře. Soubory v této složce jsou pro každého uživatele unikátní a je ji tedy nutné vyloučit z verzování. Toho se docílí přidáním řádky `.gitsy/internal` do souboru `.gitignore`.

Používání aplikace

Samotné používání aplikace je poté velice jednoduché, sestává se pouze z několika obrazovek, které budou postupně představeny.

Volba projektu

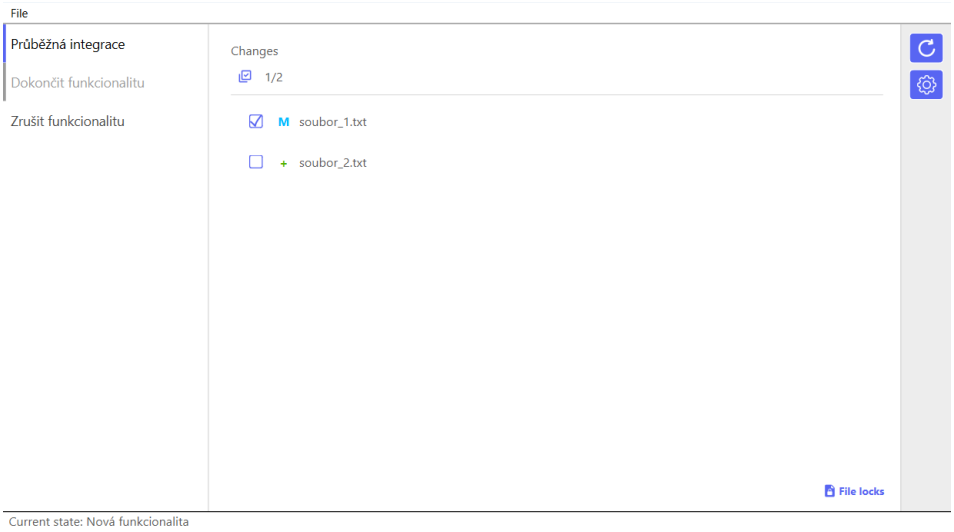
Na obrázku A.4 je zobrazena první obrazovka sloužící ke správě projektů. Projekty lze přidávat, odstraňovat a otevírat.

Obrázek A.4: Obrazovka volby projektu.

Otevřený projekt

Obrázek A.5 obsahuje hlavní okno aplikace představující otevřený projekt. Hlavním účelem obrazovky je zprostředkování možnosti spouštění akcí dostupných v současném stavu. Název současného stavu je uveden v levé dolní části obrazovky.

Dále je poskytnuto vybrání změněných souborů. U jednotlivých změn je vizualizován i typ změny – například zda se jedná o nový soubor nebo soubor pozměněný. Uživatelé znalí Gitu si mohou všimnout, že jsou změny prezentovány v zjednodušené formě – na úrovni prezentace se totiž nerozlišuje mezi změnami nacházející se v pracovním adresáři a oblasti připravených změn.



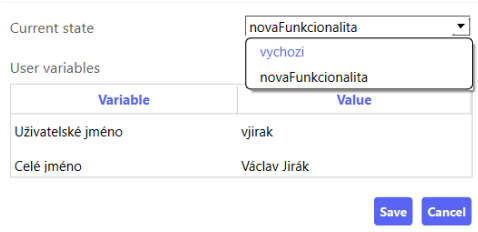
Obrázek A.5: Obrazovka otevřeného projektu.

Na úrovni procesu je ale samozřejmě informace o typu pracovní oblasti k dispozici.

Z této obrazovky se lze také dostat na obrazovky další – konkrétně na obrazovku nastavení procesu a správy zámků, viz dále.

Nastavení procesu

Další obrazovka slouží pro správu vlastností procesu specifických pro daného uživatele, viz obrázek A.6. Konkrétně zde lze pozměnit hodnoty jednotlivých uživatelských proměnných a případně lze i manuálně přepnout stav procesu.

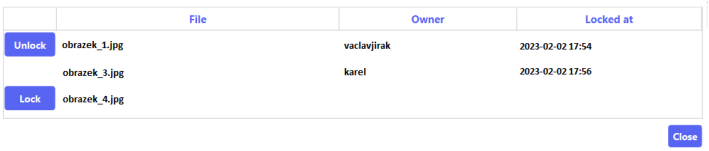


Obrázek A.6: Obrazovka nastavení procesu.

Správa zámků

Poslední obrazovka slouží pro správu zámků rozšíření Git LFS, viz obrázek A.7. Tato obrazovka je zpřístupněná pouze v případě povolení zamykání v

rámci procesu. Na obrazovce jsou k dispozici všechny uzamykatelné soubory, včetně případných informací o současných držitelích zámků.



Obrázek A.7: Obrazovka správy zámků.

B Scénář pro testování jazyka pro popis verzovacích procesů

Prerekvizity

- Nainstalovaná aplikace.
- Nainstalovaný Git.

Popis procesu

Jedná se o velmi jednoduchý proces, jehož model větvení se sestává pouze ze dvou typů větví – z jedné větve master a ze zvláštních větví pro vývoj funkcionalit.

Větve pro vývoj funkcionalit by měly splňovat následující:

- Započetí vývoje nové funkcionality by mělo vycházet vždy z aktuální větve `master`.
- Název pro větev funkcionality by měl odpovídat formátu `feature/<uzivatelske_jmeno>/<ticket>` (uživatelské jméno by nemělo být od uživatele vyžádáno při každé funkcionalitě).
- Mělo by být možné commitovat průběžnou práci.
- Dokončení funkcionality se skládá z následujících kroků:
    1. Aktualizace větev master vůči vzdálenému repozitáři.
    2. Provedení operace rebase nad hlavní větev master (v rámci této operace je možné, že vzniknout konflikty, což je v definici procesu třeba ošetřit).

3. Změny z větve pro vývoj funkcionality se sloučí do větve master.

4. Provedení synchronizace vzdálené větve master s lokální větví (operace `push`).

- Dokončení funkcionality by měla být atomická operace – buď proběhnou všechny kroky, nebo žádné.

Dotazník

- **Jak hodnotíte syntaxi jazyka?**
  - **Hodnocení:** 1 – 5 (1 = Velmi dobrá, 5 = Velmi špatná)
- **Jak hodnotíte celkový koncept daného jazyka? Myslíte, že by jeho implementace měly v herním vývoji uplatnění?**
  - **Hodnocení:** 1 – 5 (1 = Pozitivně, 5 = Negativně)
- **Jak se Vám daný proces realizoval na základě poskytnutého popisu?**
  - **Hodnocení:** 1 – 5 (1 = Velmi dobře, 5 = Velmi špatně)
- Následovala volná diskuze.

C Scénář pro testování aplikace

Prerekvizity

- Nainstalovaná aplikace.
- Nainstalovaný Git a Git LFS.
- Nakonfigurovaný nástroj pro řešení konfliktů.
- Naklonovaný testovací repozitář.

Scénář

1. V aplikaci otevřete testovací projekt.
2. Započtěte práci na nové funkcionalitě. Je předpokládáno, že je pro danou funkcionalitou vytvořen ticket s označením TASK-1.

   2.1 Funkcionalitu lze rozdělit na dvě části. Proveďte jednotlivé části v uvedeném pořadí a před započetím práce na druhé částí nejprve integrujte část první. Jedná se o následující části:

      2.1.1 Přidání libovolné řádky do textového souboru `text/text_file_1.txt`.

      2.1.2 Libovolná úprava obrázku `img/image_1.jpg` (v libovolném grafickém editoru, například Malování).

         - *Pozn.: Před úpravou `jpg` souboru je třeba daný soubor nejprve uzamknout.*

   2.2 Dokončete práci na funkcionalitě. Nezapomeňte, že práce na souboru `img/obrazek_1.jpg` je již dokončena a lze jej tedy odemknout.

3. Započtěte práci na nové funkcionalitě odpovídající ticketu TASK-2, v rámci něhož se má upravit pouze soubor `text/text_file_2.txt`.

    3.1 První část úkolu je přidání libovolné řádky do daného souboru. Tuto část po vykonání ihned integrujte.

    3.2 Nyní prosím moment vyčkejte – bude nasimulována situace, v rámci níž nastane konflikt (bude upravena první řádka daného souboru). Hned, jak bude situace nasimulována, budete informování a budete moct pokračovat dalším krokem.

    3.3 Druhá část úkolu spočívá v úpravě první řádky souboru `text/text_file_2.txt`. Tuto část po vykonání také ihned integrujte.

    3.4 Po integraci by měl nastat konflikt nad souborem `text/text_file_2.txt`. Vyřešte ho pomocí tlačítka `Resolve conflict`.

4. Nyní bylo usouzeno, že je vhodné vytvořit novou verzi produktu s označením `1.0`. Prosím, vytvořte ji.

5. Během testování byla odhalena chyba – soubor `text/text_file_3.txt` obsahuje 1+1=3, namísto 1+1=2. Je tedy třeba danou chybu opravit a aplikovat ji do verze `1.0`.

    - *Pozn.: Akci* ***`Create release-specific hotfix`*** *prosím ignorujte.*

Dotazník

- **Zvážil/a byste používání aplikace v současném stavu?**
    - **Hodnocení:** 1 – 5 (1 = Určitě ano, 5 = Určitě ne)
- **Jak hodnotíte celkový koncept dané aplikace? Myslíte, že má uplatnění v herním vývoji?**
    - **Hodnocení:** 1 – 5 (1 = Určitě ano, 5 = Určitě ne)
- **Jak se Vám dařilo provádět kroky scénáře?**
    - **Hodnocení:** 1 – 5 (1 = Velmi lehce, 5 = Velmi těžce)

- **Jak hodnotíte použitý verzovací proces? (U netechnických profesí bylo možno odpověď vynechat)**
  - **Hodnocení:** 1 – 5 (1 = Velmi dobrý, 5 = Velmi špatný)
- Následovala volná diskuze.

D Obsah přílohy

- `Text_prace/`
  - `src/` - zdrojové soubory textu práce.
  - `DP_A22N0125P.pdf` - výsledný text práce ve formátu `.pdf`.
- `Poster/`
  - `Jirak_Vaclav_2023.pdf` - poster ve formátu `.pdf`.
  - `Jirak_Vaclav_2023.pub` - poster ve formátu `.pub`.
- `Aplikace_a_knihovny/`
  - `windows_installer.exe` - instalátor aplikace pro platformu Windows.
  - `qt_project/` - adresář obsahující Qt projekt aplikace.
    - `gitsy.pro` - soubor Qt projektu, pomocí kterého lze nástrojem `qmake` sestavit aplikaci. Tento soubor lze případně také otevřít ve vývojovém prostředí `Qt Creator`.
    - `src/` - zdrojové soubory aplikace.
    - `test/` - zdrojové soubory testů.
    - `thirdparty/` - knihovny třetích stran.
  - `antlr_grammar/`
    - `GitProcess.g4` - ANTLR gramatika navrženého jazyka.
    - `generateParser.bat` - skript pro platformu Windows generující C++ parser na základě dané gramatiky. Pro správné fungování je třeba ve skriptu adekvátně upravit proměnou

antlr_jar.

    * copyGenerated.bat - skript pro platformu Windows kopírující parser vygenerovaný předchozím skriptem na správné místo do projektu.

  – deploy/

    * windows/ - adresář obsahující skript (a další potřebné soubory) pro tvorbu Windows instalátoru pomocí nástroje Qt Installer Framework. Součástí složky je i Readme.txt soubor s návodem.

- Vstupni_data/

  – Realizace_proces/ - ukázkové definice procesů verzování. V adresáři se také nachází soubor navrzeny_proces.gitsy obsahující definici verzovacího procesu, který byl navržen v této práci.

- Vysledky/

  – pruzkum_odpovedi.csv - všechna shromážděná data z provedeného průzkumu ve formátu .csv.

- Readme.txt - detailní popis aktuální adresářové struktury.