**Univerzita Hradec Králové**
**Fakulta informatiky a managementu**
**Katedra informatiky a kvantitativních metod**

# Systémy pro správu verzí se zaměřením na Git

Bakalářská práce

Autor: Daniel Maixner
Studijní obor: Aplikovaná informatika

Vedoucí práce: doc. Ing. Filip Malý, Ph.D.

Hradec Králové duben 2016

**Prohlášení:**

Prohlašuji, že jsem bakalářskou práci zpracoval samostatně a s použitím uvedené literatury.

V Hradci Králové dne Daniel Maixner

**Poděkování:**

Na tomto místě bych chtěl poděkovat především vedoucímu mé bakalářské práce doc. Ing. Filipu Malému, Ph.D., za ochotu, trpělivost a cenné rady při psaní této práce.

## Anotace

Tato bakalářská práce pojednává o systémech pro správu verzí. Konkrétně řeší, k čemu tyto systémy slouží, proč by se měly používat a jak mohou uživatelům, kteří je používají, pomoci a usnadnit práci. Po seznámení se s problémem systémů pro správu verzí následuje bližší popis systému pro správu verzí s názvem Git. Systém Git je zpracován jak z teoretického hlediska, tak i pomocí praktického projektu, který pokrývá několik kapitol v jeden ucelený reálný příklad využití systému Git.

## Annotation

### Title: Version Control Systems with Focus on Git

This bachelor thesis deals with version control systems. In particular, the thesis describes the purpose of these systems, why they should be used and how they can facilitate work for people who use them. After introducing version control systems, the thesis focuses in detail on version control system called Git. The Git system is described both from theoretical point of view and in practical project. Both parts form a complete example of the use of Git system.

# Obsah


1. Úvod ..... 1
   1.1. Konvence ..... 1
2. Systémy pro správu verzí ..... 3
   2.1. Systém pro správu verzí ..... 3
   2.2. Rozdělení verzovacích systémů ..... 4
      2.2.1. Centralizované systémy ..... 4
      2.2.2. Decentralizované (distribuované) systémy ..... 5
   2.3. Historie verzovacích systémů ..... 6
3. Verzovací systém Git ..... 7
   3.1. Historie ..... 7
   3.2. Architektura ..... 7
   3.3. Kategorie souborů v Gitu ..... 8
   3.4. Instalace ..... 10
   3.5. Popis praktického projektu ..... 11
   3.6. Vytvoření pracovního adresáře projektu ..... 12
   3.7. Vytvoření repozitáře projektu ..... 12
   3.8. Složka .git ..... 13
   3.9. Git status ..... 14
   3.10. Vytvoření prvního souboru ..... 15
   3.11. Přidání nesledovaných souborů do indexu ..... 15
   3.12. Uložení změn do historie projektu ..... 17
   3.13. Git log ..... 18
4. Princip Gitu ..... 19
   4.1. Objekty ..... 19
      4.1.1. Identifikátor objektu ..... 19
      4.1.2. Objektová databáze ..... 20
      4.1.3. Commit ..... 20
      4.1.4. Strom ..... 21
      4.1.5. Blob ..... 22
   4.2. Přidání souborů praktického projektu ..... 22
   4.3. Virtuální strom indexu ..... 24
   4.4. Reference ..... 25
      4.4.1. Master ..... 26
      4.4.2. Větve ..... 26
      4.4.3. HEAD ..... 27
   4.5. Štítek ..... 29
5. Příkazy Gitu a související akce ..... 31
   5.1. Diff ..... 31
   5.2. Základní obsah souborů praktického projektu ..... 33
   5.3. Konvence pro psaní popisů změn ..... 34
   5.4. Průběh práce ..... 35
   5.5. Checkout ..... 38

5.6. Merge ... 38
5.6.1. Fast-forward strategie ... 38
5.6.2. Rekurzivní strategie ... 40
5.7. Grafické zobrazení historie ... 41
5.8. Vzdálený repozitář (remote) ... 42
5.9. Push ... 42
5.10. Vzdálené větve ... 43
5.11. Sledování vzdálených větví ... 43
5.12. Zkratky referencí v Gitu ... 44
5.13. Clone ... 45
5.14. Fetch ... 46
5.15. Spolupráce v týmu ... 47
5.16. Rebase ... 50
5.17. Amend ... 52
5.18. Tag ... 52
5.19. Odstranění souborů (rm) ... 53
5.20. Revert ... 54
5.21. Show ... 54
5.22. Alias ... 55
5.23. Pull ... 55
5.24. Describe ... 56
5.25. Ignorování souborů ... 56
5.26. Implementace funkcí praktického projektu ... 57
5.27. Implementace hratelnosti ... 60
5.28. Začlenění do větve master ... 60
5.29. Konflikt ... 62
5.30. Cherry-pick ... 64
5.31. Přejmenování/přesunutí souborů (mv) ... 65
5.32. Závěrečné začlenění do větve master ... 66
5.33. Blame ... 67
5.34. Garbage collector (gc) ... 67
6. Shrnutí výsledků ... 68
7. Závěr ... 70
8. Seznam použitých zdrojů ... 71

# Seznam obrázků

2.1. Ukázka vývoje projektu v čase [autor] ... 3
2.2. Schéma centralizovaného verzovacího systému [autor] ... 4
2.3. Schéma decentralizovaného verzovacího systému [autor] ... 6
3.1. Lineární vývoj větve projektu v čase [autor] ... 8
3.2. Kategorie, které nabývají soubory projektu [autor] ... 10
3.3. Kategorie souborů (git status) [autor] ... 14
3.4. Kategorie souborů (git status) [autor] ... 16
3.5. Kategorie souborů (git status) [autor] ... 17
3.6. Kategorie souborů (git status) [autor] ... 18
4.1. Objektová databáze (zjednodušeno) [autor] ... 22
4.2. Objektová databáze (zjednodušeno) [autor] ... 25
4.3. Historie větve master [autor] ... 27
4.4. Historie větví master a develop [autor] ... 27
4.5. Reference HEAD [autor] ... 28
4.6. Rozdělení historie větví projektu [autor] ... 29
5.1. Historie před začleněním [autor] ... 39
5.2. Historie po začlenění [autor] ... 39
5.3. Historie před začleněním [autor] ... 40
5.4. Historie po začlenění [autor] ... 41
5.5. Historie před rebase [autor] ... 51
5.6. Historie po rebase [autor] ... 51
6.1. Ukázka výsledného praktického projektu [autor] ... 68

# Seznam příkladů

1.1. Ukázka formátu konzole ... 2
3.1. Ověření správného nainstalování Gitu ... 10
3.2. Nastavení uživatelského jména a e-mailu ... 11
3.3. Zobrazení nápovědy Gitu ... 11
3.4. Vytvoření pracovního adresáře projektu ... 12
3.5. Vytvoření repozitáře projektu ... 13
3.6. Výpis obsahu pracovního adresáře projektu ... 13
3.7. Výpis obsahu složky .git ... 13
3.8. Git status v „prázdném“ pracovním adresáři projektu ... 14
3.9. Vytvoření README souboru ... 15
3.10. Git status s nesledovanými soubory ... 15
3.11. Přidání souboru do indexu ... 16
3.12. Git status se soubory v indexu ... 16
3.13. Git commit ... 17
3.14. Git status s čistým pracovním adresářem projektu ... 18
3.15. Git log ... 18
4.1. Vyhledání souborů v objektové databázi ... 20
4.2. Zobrazení commitu ... 21
4.3. Zobrazení stromu ... 21
4.4. Zobrazení blobu ... 22
4.5. Přidání souborů praktického projektu ... 23
4.6. Přidání všech .htm a .js souborů do indexu ... 23
4.7. Vyhledání souborů v objektové databázi ... 23
4.8. Vytvoření základních souborů hry ... 24
4.9. Přidání HTML5 hlavičky ... 25
4.10. Vypsání historie větve master ... 26
4.11. Vytvoření a vypsání větví projektu ... 27
4.12. Zobrazení HEAD ... 27
4.13. Změna aktuální větve ... 28
4.14. Přidání základního HTML kódu v play.htm ... 29
4.15. Přidání štítku pro první commit projektu ... 30
5.1. Zobrazení rozdílu, který lze přidat do indexu ... 31
5.2. Zobrazení rozdílu, který lze přidat do indexu ... 32
5.3. Přidání základních objektů hada ... 34
5.4. Ukázka doporučení pro psaní popisů změn [19] ... 35
5.5. Vytvoření feature větví a přepnutí na větev feature/animate ... 36
5.6. Implementace animací hry a možnosti pozastavit hru ... 37
5.7. Přepnutí na větev master ... 38
5.8. Začlenění větve ... 39
5.9. Přepnutí na větev feature/snake-grow ... 40
5.10. Rekurzivní začlenění ... 41
5.11. Zobrazení historie projektu pomocí grafu ... 41

5.12. Vytvoření hlavního repozitáře ... 42
5.13. Odeslání větví do hlavního repozitáře ... 43
5.14. Vypsání vzdálených větví ... 43
5.15. Sledování vzdálených větví ... 44
5.16. Výpis štítků, hlav a vzdálených referencí ... 44
5.17. Vytvoření repozitáře Boba ... 45
5.18. Horizontální vycentrování plátna na stránce ... 46
5.19. Stažení aktuálních dat ze vzdáleného repozitáře ... 46
5.20. Vytvoření větve pro ovládání entity ... 47
5.21. Ovládání hada šipkami ... 48
5.22. Vytvoření větve pro ukončení hry ... 49
5.23. Přidání ověření pro konec hry ... 50
5.24. Začlenění změn od Dana ... 50
5.25. Rebase změn od Dana ... 51
5.26. Změna posledního commitu ... 52
5.27. Začlenění větve develop do větve master ... 52
5.28. Hotfix přemazávání plátna ... 53
5.29. Odstranění souboru ... 53
5.30. Vrácení změn commitu ... 54
5.31. Vytvoření globálního aliasu ... 55
5.32. Git pull ... 55
5.33. Generování verze z git describe ... 56
5.34. Ignorování složky images ... 56
5.35. Vytvoření větve pro implementaci kolizí objektů ... 57
5.36. Přidání metod pro kolizi objektů ... 58
5.37. Přidání jídla na náhodné pozici ... 59
5.38. Vytvoření větve pro ztížení hry ... 60
5.39. Ztížení hry po každých 5 bodech ... 60
5.40. Vydání verze 1.0.0 ... 61
5.41. Přidání textů UI ... 62
5.42. Konflikt ... 63
5.43. Vyřešení konfliktu ... 64
5.44. Oprava chyby v pohybu hada ... 64
5.45. Omezení nepovoleného pohybu hada ... 65
5.46. Přesunutí/přejmenování souboru ... 66
5.47. Sjednocení větví develop a master ... 66
5.48. Spuštění garbage collectoru ... 67
6.1. Výsledný graf projektu GitSnake ... 69

# 1. Úvod

Každý jistě někdy při používání počítače využil možnost vrátit se o krok zpět. Například jsme při psaní textu udělali chybu a po zjištění této chyby jsme klikli na tlačítko *ZPĚT* pro vrácení do stavu před chybou. Pokud jsme chybu objevili až po sérií úprav od napsání dané chyby, případně po znovuotevření souboru, nejspíše jsme narazili na problém, že na tlačítko *ZPĚT* lze kliknout pouze několikrát, případně vůbec. V takovéto chvíli jsme si jistě přáli mít neomezený počet vrácení historie souboru. Systém pro správu verzí nás nejen v takovýchto situacích může zachránit.

Systém pro správu verzí umožňuje ukládat změny provedené v souboru (souborech) v průběhu času. Mezi další přednosti patří možnost prohlížet a vracet se k původním verzím souborů, porovnávat změny v souborech, spolupráci v týmu lidí a mnoho dalšího. Systém pro správu verzí umožňuje uživateli upravit soubor bez obav o ztrátu předchozí verze. Nechtěné úpravy, případně odstranění lze snadno navrátit nejen do stavu před úpravou či odstraněním, ale do jakéhokoli stavu, který je v systému uložen.

Systém pro správu verzí je nepostradatelnou součástí vývoje každého softwaru, pro tento účel byly tyto systémy primárně navrženy, ale i například pro designéra vytvářejícího grafiku nebo skladatele hudby je neocenitelným pomocníkem ve sledování historie (vývoje) digitálního projektu. Systém pro správu verzí je použit i při psaní této práce.

Cílem této práce je podat čtenáři obecné informace o systémech pro správu verzí a následně bližší seznámení se systémem Git.

V kapitole 2 budou vysvětleny základní pojmy, rozdělení a krátká historie systému pro správu verzí. V kapitole 3 bude představen systém Git a základní práce v tomto systému. V kapitole 4 bude podrobněji rozebrán princip systému Git. V kapitole 5 budou probrány příkazy Gitu a související akce pro dokončení praktického projektu.

## 1.1. Konvence

Použité konvence v této práci:

- Konzole – pro příkazový řádek či terminálový emulátor bude použito slovo konzole.
- Git – pokud je v této práci odkazováno na program jako na celek, bude použito slovo Git (s velkým písmenem na začátku).
- Git – použití výrazu git (s malým písmenem na začátku) bude odkazovat na spouštěcí soubor Gitu pro použití v konzoli, pokud nebude gramatika češtiny vyžadovat použití velkého počátečního písmena.

Formát konzole v této práci používá následující konvence:

- $ – všechny příkazy, které je třeba napsat do konzole, mají na začátku řádku symbol „$“. Příkazy se potvrzují stiskem klávesy Enter (Return) po napsání posledního znaku příkazu.
- # – symbol „#“ na začátku řádku značí, že daný řádek je komentář. Text na řádku začínajícím tímto symbolem není třeba opisovat do konzole.

Pro zobrazení „obrázku“ konzole bude v této práci použit následující formát:

**Příklad 1.1. Ukázka formátu konzole**

```
$ příkaz --přepínač parametr
# řádek s komentářem

Běžný vstup či výstup konzole, ❶
který může být i na více řádku.
```

❶ Popis, který slouží pro bližší vysvětlení, co se na označeném řádku událo.

V této práci jsou využity následující typy konzolových příkazů:

1. Git příkazy – příkazy, které začínají slovem git, například `git status`. [1]
2. Unix příkazy – základní programy definované v *IEEE Std 1003.1-2008*, například `cat`. [2]
3. Příkazy interpretu konzole – vnitřní příkazy interpretu konzole, například `pwd`. V této práci je použit konzolový interpret Bourne Again Shell (Bash)[1], který je nastaven jako výchozí konzolový interpret na většině konzolí.

[1] http://www.gnu.org/software/bash/

# 2. Systémy pro správu verzí

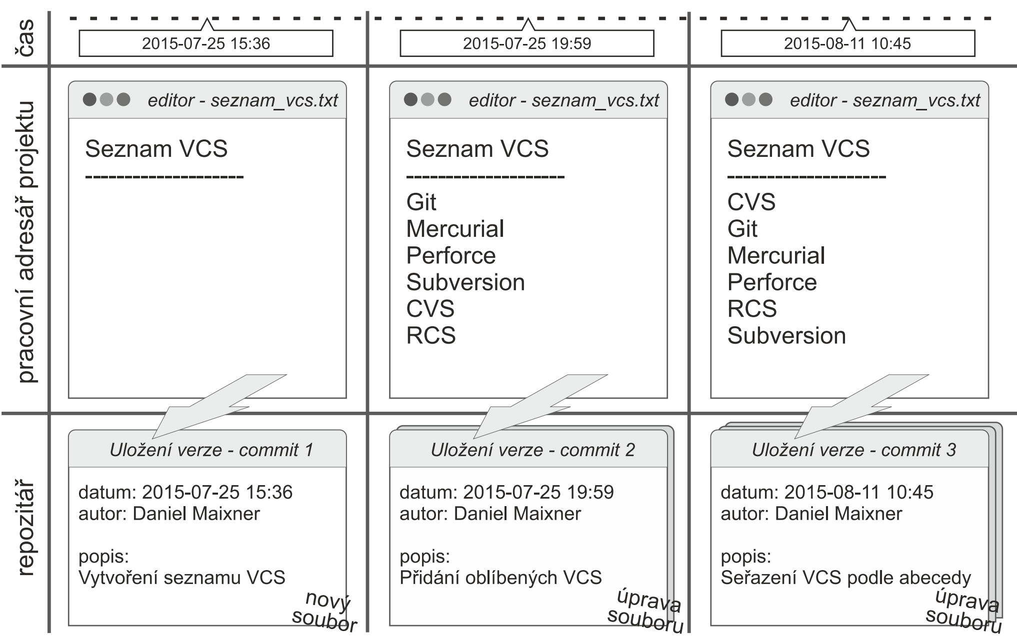
Systémy pro správu verzí se začaly objevovat již v 60. letech 20. století. Základní princip systému pro správu verzí však zůstal stejný a v této kapitole bude vysvětlen spolu s rozdělením a krátkou historií systémů pro správu verzí.

## 2.1. Systém pro správu verzí

Systém pro správu verzí (zkratka VCS[1]) je počítačový program pro ukládání verzí (někdy též označovaných revizí) skupiny souborů. Takováto skupina souborů se nazývá **projekt**. Projekt je udržovaný ve složce označované **pracovní adresář projektu**. Verzovací systém ukládá veškeré uložené verze projektu do struktury nazývané **repozitář**[2].

Systém pro správu verzí (dále jen **verzovací systém**) pracuje na principu ukládání verzí projektu v čase. Tyto verze ovšem verzovací systém ukládá pouze, pokud je mu implicitně řečeno, na rozdíl od v úvodu zmiňovaného tlačítka *ZPĚT*, které ukládá veškeré změny například po stisku klávesy v okně aplikace. Verzovací systém se nejčastěji používá pro ukládání textových souborů (souborů kódovaných pomocí standardu pro text), ale dnešní moderní verzovací systémy umožňují verzovat i soubory binární[3] (aplikace, videa a další netextové soubory).

Verzovací systém u každé uložené verze, které se říká **commit**, zaznamená, **kdo změnu provedl**, **datum** a uživatelem definovaný **popis** této změny.




**Obrázek 2.1. Ukázka vývoje projektu v čase [autor]**

[1] Z anglického version control system.

[2] Lze se setkat i s výrazem databáze. V této práci je však výhradně použit výraz repozitář.

[3] Verzovací systémy však nejsou primárně navrženy k verzování většího počtu binárních souborů, k tomuto účelů existují speciální programy pro zálohování dat.

## 2.2. Rozdělení verzovacích systémů

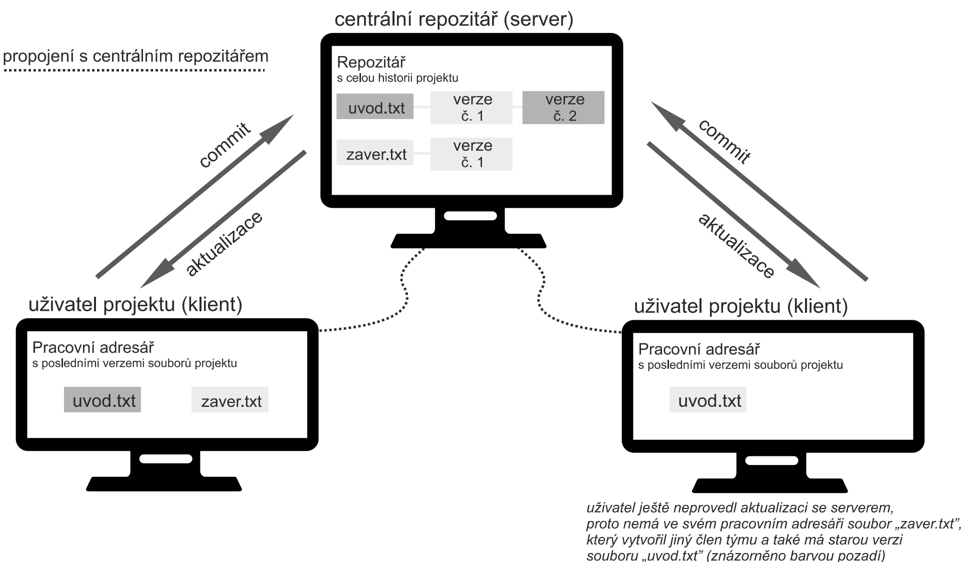
Verzovací systémy se dělí na dvě hlavní kategorie (řešení) podle přístupu k repozitáři.

1. Centralizované systémy
2. Decentralizované (distribuované) systémy

### 2.2.1. Centralizované systémy

Centralizované řešení je starší, používá **pouze jeden** (centrální) repozitář, ke kterému mají přístup všichni zúčastnění na projektu (nejčastěji pomocí úložiště v síti). Centralizovaný systém pracuje na principu **klient-server**, kde server je onen **centrální repozitář s kompletní historii projektu** a klienti jsou uživatelé pracující na projektu. Klienti mají **pouze poslední verze souborů projektu** uložené v pracovním adresáři projektu na svém disku, případně rozpracované nové verze čekající na odeslání do repozitáře.

Práce s centralizovaným systémem probíhá tak, že když chceme pracovat na projektu, provedeme nejprve aktualizaci s centrálním repozitářem. Po této aktualizaci se stáhnou **pouze poslední verze všech souborů** uložené v centrálním repozitáři (pokud již nemáme pracovní adresář projektu aktuální) a následně se pomocí procesu začlenění (**merge**) tyto změny přenesou do našeho pracovního adresáře projektu. Po začlenění můžeme upravovat jednotlivé soubory projektu a v případě potřeby *commit*nout novou verzi na server (například pro zpřístupnění dat kolegovi nebo uložení současného stavu do historie projektu).




**Obrázek 2.2. Schéma centralizovaného verzovacího systému [autor]**

Nevýhodou centralizovaného řešení je, že pokud nejsme připojeni k síti, nemůžeme pracovat s repozitářem (prohlížet, stahovat či nahrávat změny). Kvůli nutnosti posílat data po síti je také práce s repozitářem pomalejší. A posledním problémem je, že například v případě selhání disku serveru či jiném vážném poškození, pokud nemáme vytvořené zálohy, ztratíme veškerou historii projektu, neboť historie projektu se ukládá pouze na serveru.

## 2.2.2. Decentralizované (distribuované) systémy

Distribuované řešení je vylepšením centralizovaného přístupu. Hlavní rozdíl je, že nejsou závislé pouze na jednom repozitáři, ale fungují na principu **peer-to-peer** (rovný s rovným). V distribuovaném přístupu má každý člen projektu **na svém disku uložen svůj vlastní (osobní) repozitář s kompletní historii projektu**, který si je rovný se zbylými repozitáři. Rovný znamená, že každý člen projektu může stahovat z repozitáře, případně nahrávat do repozitáře kteréhokoli jiného člena (každý může vystupovat zároveň jako „klient“ či „server“), pokud je pro něj daný repozitář dostupný (přístup po síti, lokální práva pro přístup k souborům). S přibývajícím počtem uživatelů zúčastněných na projektu však nutnost stahovat veškeré změny od všech členů týmu se stává zdržováním, a proto se většinou zvolí pouze jeden[4] repozitář, který je dobře přístupný (nejčastěji úložiště v síti) a tento repozitář se označí jako **hlavní**.[5] S tímto hlavním repozitářem se ostatní členové týmu synchronizují.

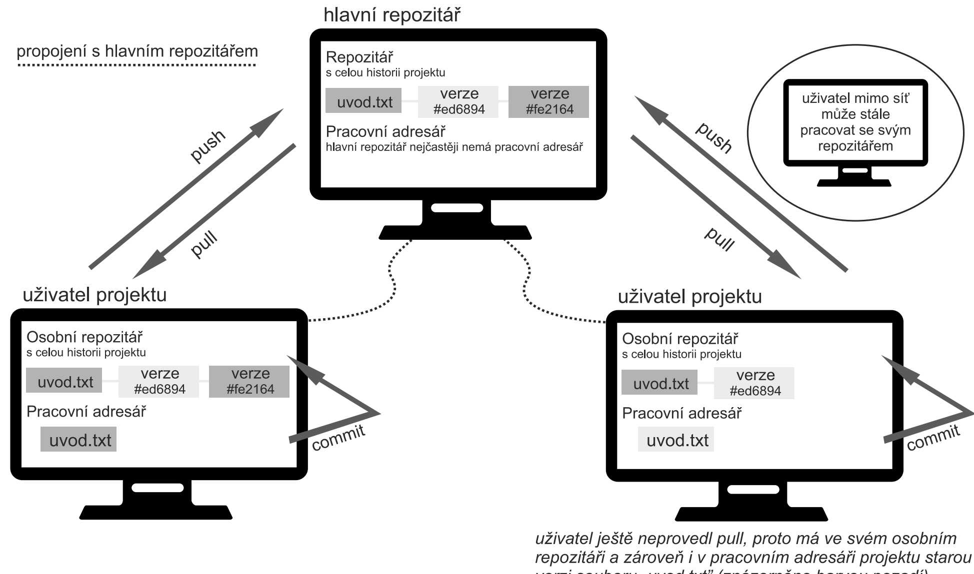
Každý, kdo chce na projektu pracovat, si vytvoří repozitář nový, případně si z již existujícího repozitáře zkopíruje (**naklonuje**) svůj osobní repozitář obsahující celou historii projektu, která byla uložená v repozitáři ze kterého byl klon vytvořen. Z takto vytvořeného repozitáře lze vytvářet další klony (repozitáře).

Postup práce v distribuovaném systému začíná tak, že si nejdříve „natáhneme“ (**pull**) veškerou historii z některého z dostupných repozitářů (nejčastěji je to repozitář, ze kterého jsme náš repozitář klonovali, ale není to pravidlo) do našeho osobního repozitáře. *Pull* po stažení veškerých dat provede proces začlenění dat do pracovního adresáře projektu. Nyní můžeme provést změny v projektu a v případě potřeby *commit*nout novou verzi do našeho osobního repozitáře. Pokud chceme změny odeslat i do repozitáře jiného člena týmu či do hlavního repozitáře, stačí „natlačit“ (**push**) změny do daného repozitáře. Vzhledem k dynamičnosti propojení repozitářů v distribuovaném řešení lze vytvořit nekonečně mnoho schémat zapojení, jedno takové schéma ukazuje obrázek 2.3.

Výhodou distribuovaného řešení je, že každý klon obsahuje veškerou historii projektu, takže v případě výpadku některého repozitáře (třeba hlavního) lze zvolit repozitář libovolného člena týmu (nejčastěji toho, kdo má nejaktuálnější historii projektu), který nahradí funkci hlavního repozitáře. Další výhodou je, že náš osobní repozitář je uložen na našem lokálním disku, tudíž operace s repozitářem probíhá velice rychle (v porovnání s centralizovaným systémem). Také odpadá nutnost pracovat s repozitářem pouze pokud jsme připojeni k síti. V distribuovaném řešení můžeme ukládat změny průběžně do svého repozitáře bez přístupu k síti, a jakmile se připojíme k síti, můžeme *push*nout změny do repozitáře kolegy či hlavního repozitáře projektu.

[4] U větších projektů bývá členění daleko složitější hlavně kvůli rozdělení do menších logických částí projektů (modulů) a kvůli omezení přístupových práv k repozitáři.

[5] Většinou se tento repozitář nazývá centrální, ale aby nedocházelo k záměně s centralizovanými systémy, je v této práci použit výraz hlavní.




**Obrázek 2.3. Schéma decentralizovaného verzovacího systému [autor]**

## 2.3. Historie verzovacích systémů

Jelikož historie verzovacích systémů je dosti složitá, na následujících řádcích budou vypsány pouze stručně milníky tohoto tématu. Historie bude rozdělena do tří generací podle [3][4].

Téma verzovacích systémů se stalo populární v 70. letech 20. století. Vše začal systém s názvem *Source Code Control System (SCCS)*, vytvořený v Bellových laboratořích roku 1972. Jeho autorem je *Marc Rochkind*. [5] Tento systém je považován za první populární verzovací systém, i když první svého druhu nebyl. Koncept vln a jednoznačně identifikovaných verzí systému SCCS lze najít i v dnešních verzovacích systémech. [4] Na úspěch systému SCCS navázal v roce 1982 systém *Revision Control System (RCS)*, jehož autorem je *Walter F. Tichy*. [6] Uvedený systém byl alternativou pro populární *SCCS*. Tyto dva systémy jsou zástupci tzv. první generace verzovacích systémů. První generace je známá pro verzování jednotlivých souborů, centralizovaným přístupem k repozitáři a možnosti upravovat jednotlivé soubory pouze jedním uživatelem pomocí techniky zámku souboru. [3][4]

Za průkopníka druhé generace je považován Concurrent Version System (CVS), který vytvořil v roce 1985 Dick Grune. [7] CVS bylo považováno de facto za lídra verzovacích systému té doby, a to až do roku 2000, kdy vznikl systém Subversion (SVN), vytvořený firmou CollabNet. [8] Druhá generace se vyznačuje prací více uživatelů na stejných souborech, podporou pro nelineární vývoj a centralizovaným řešením přístupu k repozitáři.

Třetí generace používá distribuované řešení přístupu k repozitáři a také vylepšuje podporu pro nelineární vývoj projektu. Mezi hlavní zástupce patří Git, kterému se blíže věnuje tato práce. Git vznikl v roce 2005 a jeho autorem je Linus Torvalds. [9] Dalším zástupcem je Mercurial, který také vznikl v roce 2005, autorem je Matt Mackall. [10]

# 3. Verzovací systém Git

V minulé kapitole byly probrány základní informace o verzovacích systémech. V této kapitole začneme pracovat s verzovacím systémem Git.

Git je zdarma dostupný počítačový program s otevřeným zdrojovým kódem, licencovaný podle GPL-2.0 pro správu verzí souborů. [11] Git se zaměřuje především na správu textových souborů, ale díky jeho pevné vazbě na souborový systém lze v něm verzovat cokoli, co lze uložit do souborového systému. Git je distribuovaný verzovací systém s důrazem na rychlost, zachování integrity dat projektu a silnou podporu nelineárního vývoje. [11]

Přestože existuje velké množství programů s grafickým uživatelským rozhraním postavených nad Gitem, nejlepší způsob jak využívat Git ve většině případů je pomocí konzole, která poskytuje univerzální rozhraní. Konzole byla proto zvolena jako hlavní prostředí pro praktickou práci s Gitem v této práci.

## 3.1. Historie

Historie Gitu je úzce spojena s vývojem jádra Linux. V roce 2002 přechází vývoj Linuxu na komerční verzovací systém s názvem BitKeeper. BitKeeper má pro komunitu vývojářů Linuxu výjimku – za jeho používání komunita nemusí nic platit. Tato spolupráce trvá až do roku 2005, kdy je Linuxovému repozitáři tato výjimka odebrána.[1] Na tuto událost zareagoval Linus Torvalds, který se začal[2] rozhlížel po alternativním verzovacím systému. Výsledky hledání náhrady však byly neúspěšné, a tak Linus přišel s vlastním verzovacím systémem, který pojmenoval Git. Git „spatřil světlo světa“ 7. dubna 2005, když prokázal svoji vyspělost tím, že jeho zdrojový kód začal verzovat sám Git. [9] Krátce poté přešel vývoj Linuxu pod správu Gitu. [12]

> Git je druhé nejslavnější „dítě“ Linuse Torvalda, který Gitu svěřil verzovat miliony řádků zdrojových kódů jádra Linux.
>
> — volně přeloženo z [13]

Linus zvolil jméno Git, protože chtěl třípísmennou zkratku, která se dobře vyslovuje (/gɪt/) a není použita pro žádný standartní unixový program. [9][14] Nakonec zvolit slovo Git, které pochází z anglického slangu a znamená hloupý (u Gitu je to však myšleno ve smyslu krásně jednoduchý). [9][14]

## 3.2. Architektura

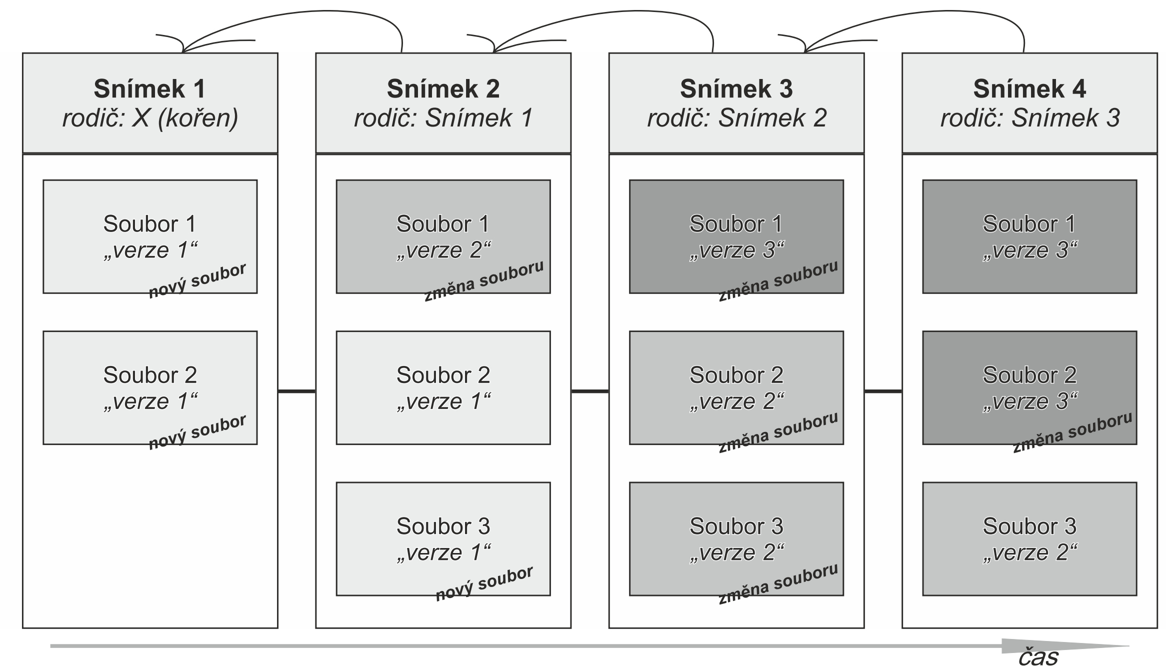
Git byl prvotně napsán jako sada konzolových skriptů. Z počátku se nepředpokládalo vyvinutí do kompletního systému, spíše se předpokládalo, dle definice unixových programů, že bude v budoucnu vyvinut program postavený nad jádrem Gitu pro interakci s uživatelem. Nakonec se však vývojáři rozhodli k příkazům v jádru (v Gitu označované **plumbing**) přidat příkazy uživatelského rozhraní (v Gitu označované **porcelain**), a vytvořit tak kompletní systém. Jádro Gitu je však stále přístupné, a je tak možné pracovat s repozitářem i na nižší úrovni přístupu.

[1] https://lkml.org/lkml/2005/4/6/121

[2] Hledání nového verzovacího systému pro Linux započalo již před touto událostí, především kvůli faktu, který trápil mnoho vývojářů Linuxu, a sice používání komerčního softwaru pro vývoj nekomerčního softwaru.

Git ukládá obsahy všech souborů v okamžik commitu jako jednu velkou verzi. Commity v Gitu se proto často označují jako **snímky** (Git při commitu „vyfotí“ a uloží kompletní stav projektu v daném čase do historie projektu). Toto chování je velice odlišné od tradičních verzovacích systému, které se zaměřují na změny v jednotlivých souborech.

Všechny commity v Gitu jsou závislé na commitech předešlých, a vytváří se tak vztah **rodič-dítě**[3] (parent-child) mezi commity. Git řadí commity do posloupnosti nazývané **větev (branch)**. Projekt může mít i více než jednu větev a podobně také commit může mít i více rodičů nebo žádného. V případě, že commit nemá rodiče, je označován jako **kořen (root-commit)**.




**Obrázek 3.1. Lineární vývoj větve projektu v čase [autor]**

Pro vytváření vztahu *rodič-dítě* je nutné, aby si Git ukládal odkaz na **„poslední“ commit**, který se v Gitu označuje **HEAD**. Rodič následujícího commitu tak bude commit, na který *HEAD* odkazuje. Po vytvoření nového commitu se *HEAD* aktualizuje, aby odkazoval na nově vytvořený commit, a proces se opakuje.

## 3.3. Kategorie souborů v Gitu

Git ve výchozím nastavení ukládá repozitář projektu do pracovního adresáře projektu, konkrétně do složky ve výchozím nastavení pojmenované *.git*.[4] V pracovním adresáři projektu tak jsou nejčastěji uložena data projektu spolu se složkou *.git* obsahující data repozitáře tohoto projektu.

[3] Lze se setkat i s pojmem předek-potomek.

[4] Pro odkazování na složku repozitáře Gitu v této práci bude použit výraz *.git*.

```
.
└── složka_projektu (pracovní adresář projektu)
    ├── .git (repozitář projektu)
    ├── soubory
    ├── a_složky
    ├── projektu
```

Soubory projektu z pohledu Gitu se mohou nacházet v jedné ze dvou hlavních kategorií:

1. Tracked (sledované) – soubory, které jsou uloženy v *HEAD* nebo ve staging area. Sledované soubory se mohou nacházet v jednom z následujících stavů:

    a. Modified (změněné)[5] – znamená, že se daný soubor v pracovním adresáři projektu změnil v porovnání se souborem uloženým v *HEAD* nebo v porovnání se staging area. Tuto změnu lze uložit do historie větve.

    b. Unmodified (nezměněné) – znamená, že soubor je ve stejné verzi, jaká je uložená v *HEAD*, a zároveň i ve staging area.

2. Untracked (nesledované) – ostatní soubory[6], které nejsou uloženy v *HEAD* nebo ve staging area. Tyto soubory nelze uložit do historie projektu. Git ukládá verze pouze u souborů projektu, u kterých mu je implicitně řečeno, aby je ukládal. Každý nově vytvořený soubor projektu je proto automaticky zařazen do kategorie nesledované soubory.

Pro práci se sledovanými soubory Git obsahuje dvě hlavní struktury udržované ve složce *.git*:

1. Staging area (dále jen **index**[7]) – index slouží jako mezistupeň mezi pracovním adresářem projektu a příštím commitem. Index ukazuje, jaké soubory budou uloženy v příštím commitu (čeká se na příkaz commit pro trvalé uložení).

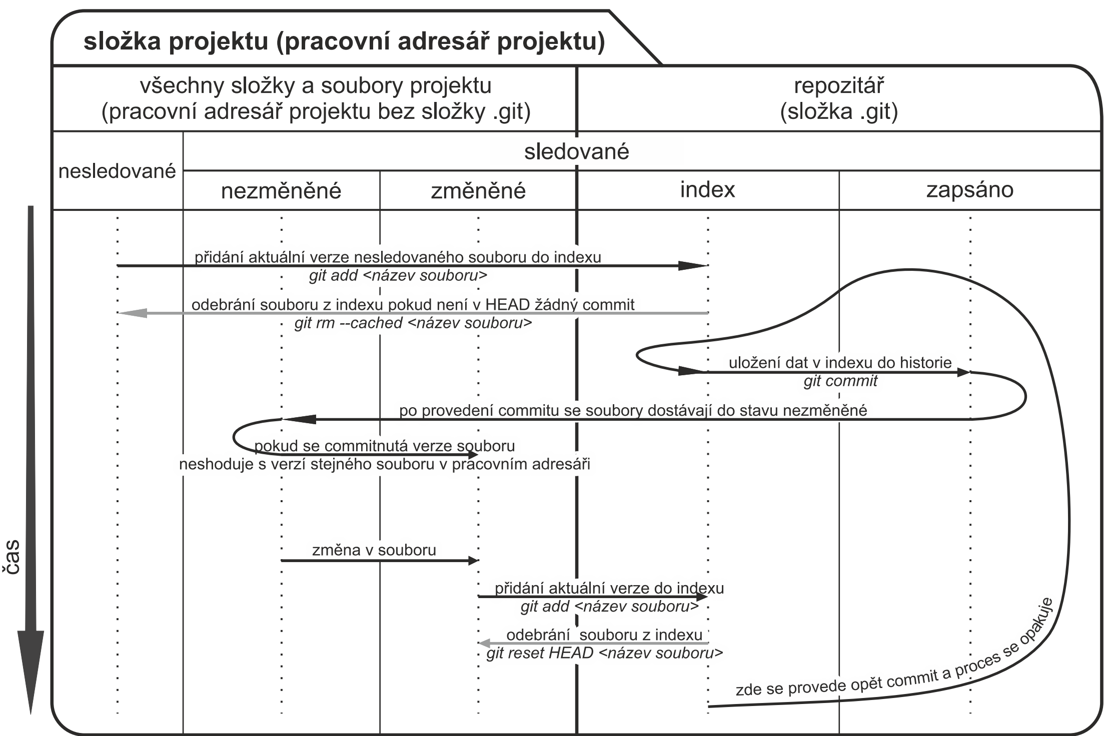
2. Commited (zapsáno) – soubory, které byly v indexu, jsou bezpečně uloženy v historii projektu (proběhl commit).

Tuto sekci zakončuje obrázek 3.2, který graficky shrnuje změny v souborech projektu z pohledu Gitu. V následujících sekcích bude rozebráno, jak lze měnit kategorie, ve kterých se soubory projektu nachází, pomocí příkazů Gitu.

[5] Změnou souboru Git rozumí kromě změny v obsahu souboru i vytvoření, smazání či přejmenování souboru.

[6] Soubor může být i ve stavu ignorován (ignored). Tento stav je specializací nesledovaných souborů. Více o ignorovaných souborech v kapitole 5.25.

[7] Pokud bude v této práci psáno o staging area, bude použit výraz index především kvůli lepšímu použití v české větě. Správně by se však měl používat výraz staging area, i když použití slova index není špatně. Občas se také lze setkat s výrazem cache, který by se ale neměl v tomto kontextu používat.




**Obrázek 3.2. Kategorie, které nabývají soubory projektu [autor]**

## 3.4. Instalace

Git je dostupný zdarma pro všechny současné majoritní operační systémy (Windows NT, OS X a Linux) a jeho instalace je jednoduchá. [11] Pro nainstalování Gitu stačí přejít na domovskou stránku Gitu http://git-scm.com/, která by automaticky měla rozpoznat operační systém návštěvníka a nabídnout možnost stažení (**download**) programu. Domovská stránka Gitu poskytuje podrobný návod jak Git stáhnout a následně nainstalovat.

Tato práce předpokládá práci v konzoli schopné emulovat unixové prostředí. OS X a Linux mají konzoli splňující tento požadavek přímo v základní instalaci operačního systému. Pro Windows NT je nutné používat program **Git Bash**, který se nainstaloval spolu s Gitem, případně lze využít jiný emulátor unixového prostředí pro Windows NT, například Cygwin[8] apod.

Po nainstalování Gitu otevřeme konzoli a zadejme příkaz:

**Příklad 3.1. Ověření správného nainstalování Gitu**

```
$ git --version
git version 2.8.1
```

Pokud jsme neinstalovali Git kompilací zdrojových kódů, mělo by se nám vypsat něco podobného výstupu příkladu 3.1 (verze Gitu může být odlišná). Pokud se tak stalo, znamená to, že Git je správně nainstalovaný a připraven k použití, v opačném případě je třeba konzultovat návod Gitu.

[8] https://www.cygwin.com/

Po nainstalování Gitu je třeba vyplnit naše jméno a e-mail. Tyto údaje Git použije pro vyplnění autora změny v commitu. Pro nastavení použijeme příkaz:

**Příklad 3.2. Nastavení uživatelského jména a e-mailu**

```
$ git config --global user.name "Daniel Maixner"
$ git config --global user.email "daniel-maixner@seznam.cz"
```

`Git config <klíč> <hodnota>` slouží pro úpravu nastavení Gitu ve třech úrovních:

- Systémové – nastavení pro všechny repozitáře uložené na daném počítači.
- Globální – nastavení pro všechny repozitáře, se kterými daný uživatel operačního systému pracuje.
- Lokální – nastavení pro aktuální repozitář (repozitář, ve kterém je příkaz proveden).

V příkladu 3.2 jsme použili globální nastavení (přepínač *--global*), které uložilo zadané uživatelské jméno a e-mail jako výchozího autora změn pro všechny commity vytvořené pomocí aktuálně přihlášeného uživatelského účtu operačního systému.

Pokud si nebudeme vědět rady jak použít libovolný příkaz Gitu, případně k čemu slouží, vždy je možnost využít nápovědy Gitu. Nápověda Gitu je dostupná na domovské stránce Gitu nebo přímo v konzoli pomocí příkazu `git help <název příkazu>` nebo pomocí přepínače *--help* pro daný příkaz.[9]

**Příklad 3.3. Zobrazení nápovědy Gitu**

```
# zobrazení nápovědy pro příkaz Gitu s názvem config
$ git help config
# případně
$ git config --help
```

Příklady v této práci jsou realizovány pomocí Gitu ve verzi 2.8.1, vydané 3. dubna 2016. Příkazy v této práci patří mezi základní a jejich syntaxe se nezměnila již od verze 2.0.0 a pravděpodobně se v blízké době ani nezmění.

## 3.5. Popis praktického projektu

Od následující sekce začneme tvořit zdrojové soubory pro praktickou ukázku využití nástroje Git – vlastní verzi známé hry Had[10], kterou nazveme **GitSnake**. Hra bude tvořena od samého začátku až do finální verze hry, ve které bude hratelná v libovolném moderním webovém prohlížeči.

Projekt bude velice jednoduchý a slouží především jako ukázka použití Gitu než jako plnohodnotná hra. Zdrojové soubory projektu, jejímž autorem je autor této práce, budou jednoduché a měli by je být schopni

[9] Pro zobrazení stručného návodu jak lze použít určitý příkaz, Git nabízí příkaz `git <příkaz> -h`.
[10] https://en.wikipedia.org/wiki/Snake_(video_game)

pochopit i uživatelé, kteří nemají předešlé zkušenosti s programováním. Pro následování příkladu v této práci však není nutnost rozumět každému řádku zdrojového kódu, nejdůležitější je pochopení Gitu, což je hlavním cílem této práce.

## 3.6. Vytvoření pracovního adresáře projektu

Prvním krokem je vytvoření pracovního adresáře projektu pro uchování dat projektu. Složku projektu pojmenujeme *git_snake*. Tuto složku vytvoříme na *Ploše* našeho operačního systému následovně:

Změníme pracovní adresář konzole na zvolené umístění pro ukládání dat projektu – *Plocha* operačního systému. Ke změně pracovního adresáře konzole využijeme příkaz konzolového interpretu – `cd`. Nejjednodušší způsob jak se dostat na *Plochu* je pomocí **~** (vlnovka nebo též tilda). Tento znak zastupuje domovský adresář. Nejčastěji je *Plocha* (**Desktop**) umístěna právě v domovském adresáři, proto se na *Plochu* dostaneme pomocí příkazu `cd ~/Desktop`. Jakmile je pracovní adresář konzole na *Ploše*, stačí zde vytvořit novou složku *git_snake* pomocí příkazu `mkdir`, která bude sloužit jako pracovní adresář projektu *GitSnake*. Jakmile je složka vytvořená, změníme pracovní adresář konzole na nově vytvořenou složku pomocí příkazu `cd`.

**Příklad 3.4. Vytvoření pracovního adresáře projektu**

```
# Pracovní adresář konzole změníme na Plochu
$ cd ~/Desktop
# Vytvoříme složku git_snake
$ mkdir "git_snake"
# Změníme pracovní adresář konzole
$ cd git_snake
```

Uvozovky u "git_snake" nejsou povinné, pokud název neobsahuje mezery nebo speciální znaky rozeznatelné konzolí. Obecně není doporučeno používat mezery, speciální znaky a českou diakritiku v názvech souborů (či adresářů) projektu.

Další příklady předpokládají, pokud není uvedeno jinak, že pracovní adresář konzole je vždy v tomto umístění (*/Users/dan/Desktop/git_snake*), pokud si čtenář zvolí jiné umístění, je nutné zaměnit */Users/dan/Desktop/git_snake* za čtenářem definované umístění. Pro přehled však v každé ukázce konzole bude na prvním řádku použit příkaz `pwd`, který slouží k výpisu absolutní cesty k aktuálnímu pracovnímu adresáři konzole.

## 3.7. Vytvoření repozitáře projektu

Minulou sekci jsme zakončili vytvořením pracovního adresáře projektu a změnou pracovního adresáře konzole na toto nové umístění. V této sekci vytvoříme repozitář projektu pomocí příkazu Gitu. Pro vytvoření repozitáře stačí naklonovat existující repozitář nebo vytvořit repozitář nový. V tomto případě budeme vytvářet repozitář nový.

Pro vytvoření repozitáře stačí zadat příkaz `git init`. Takto napsaný příkaz vytvoří repozitář v současném pracovním adresáři konzole.

**Příklad 3.5. Vytvoření repozitáře projektu**

```
$ pwd
/Users/dan/Desktop/git_snake
$ git init ❶
Initialized empty Git repository in /Users/dan/Desktop/git_snake/.git/ ❷
```

❶ Pomocí příkazu `git init` řekneme Gitu, aby v aktuálním pracovním adresáři konzole (složka *git_snake* uložená na *Ploše*) vytvořil a zinicializoval nový repozitář. Pokud by zde repozitář již existoval, Git by načetl inicializační sekvenci znovu (existující soubory projektu by zůstaly nedotčené).

❷ Výsledkem příkazu je potvrzení od Gitu o vytvoření a zinicializování Git repozitáře (vytvoření složky *.git*) v umístění */Users/dan/Desktop/git_snake/.git/*.

## 3.8. Složka .git

Pokud se podíváme na obsah pracovního adresáře projektu pomocí příkazu `ls`, zjistíme, že ve složce *git_snake* není žádná složka *.git*, což nesouhlasí s tím, co vypsal Git v příkladu 3.5.

**Příklad 3.6. Výpis obsahu pracovního adresáře projektu**

```
$ pwd
/Users/dan/Desktop/git_snake
$ ls
# příkaz ls vrátí prázdný výsledek (prázdný adresář)
```

Složka *.git* se ve výpisu nezobrazuje, neboť je ve výchozím nastavení skrytá. Hlavní důvod, proč Git skrývá složku *.git*, je ten, že tato složka je repozitář projektu. Repozitář mimo jiné obsahuje i veškerou historii projektu, proto je lepší, že je tato složka skrytá, aby ji uživatel omylem nesmazal, a nepřišel tak o veškerou historii projektu. Pro zobrazení skrytých souborů (složek) musíme použít například přepínač *-A* příkazu `ls`.

**Příklad 3.7. Výpis obsahu složky .git**

```
$ pwd
/Users/dan/Desktop/git_snake
$ ls -A
.git
$ ls -p .git
HEAD  config  description  hooks/  info/  objects/  refs/
```

Příkaz `ls -A` již vrátil hledanou složku *.git*. Pokud se podíváme pomocí `ls -p .git` na obsah složky *.git*, zjistíme, že zde jsou soubory *HEAD, config, description* a složky *hooks, info, objects, refs*.

Přepínač *-p* příkazu `ls` slouží k vizuálnímu oddělení typu souborů. Složky mají na konci názvu lomítko, soubory nikoli. Pro zobrazení nápovědy unixových příkazů lze použít přepínač *--help* obdobně jako u příkazů Gitu, například `ls --help`.

Obsah složky *.git* bude rozebrán podrobněji v následujících kapitolách. V tuto chvíli je dobré pouze vědět, že tato složka existuje a jak lze vypsat její obsah.

## 3.9. Git status

Pro zobrazení stavu projektu z pohledu Gitu Git nabízí příkaz `git status`. Pokud zadáme tento příkaz, tak pokud je pracovní adresář projektu „prázdný“ (ve smyslu, že kromě skryté složky *.git* v něm nic jiného není) Git vypíše následující:

**Příklad 3.8. Git status v „prázdném“ pracovním adresáři projektu**

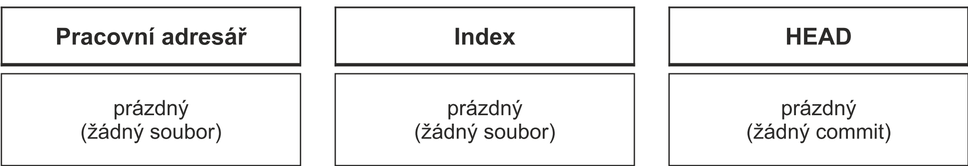
```
$ pwd
/Users/dan/Desktop/git_snake
$ git status
On branch master ❶

Initial commit ❷

nothing to commit (create/copy files and use "git add" to track) ❸
```

❶ *On branch master* znamená, že Git je nyní fixován na výchozí větev, která se v Gitu jmenuje **master**. Tuto větev budeme „rozrůstat“ (přidávat commity).

❷ *Initial commit* znamená, že Git čeká na vytvoření prvního commitu projektu ve větvi *master*. Tento commit bude *kořen* (prvotní rodič všech následujících commitů v *master* větvi).

❸ Tento řádek říká, že nemáme nic, co by Git mohl uložit (*commit*nout), což je logické, neboť v pracovním adresáři projektu není žádný soubor. Git nám dává radu, co bychom měli udělat – v tomto případě nám radí, ať vytvoříme, nebo vložíme nějaký soubor do složky projektu a použijeme příkaz `git add`, aby mohl Git začít sledovat změny v tomto souboru.

Přesněji příkaz `git status` slouží k vypsání změn mezi pracovním adresářem projektu, indexem a *HEAD*. Git zjistil, že v pracovním adresáři projektu nejsou žádné soubory. V indexu Git také nenalezl žádné soubory a ani v *HEAD* Git nenalezl žádné soubory, neboť zde žádný commit dosud nebyl vytvořen. Vše ukazuje obrázek 3.3. Obdobné obrázky budou použity i v dalších sekcích pro lepší pochopení toho, jak Git porovnává změny v těchto kategoriích.




**Obrázek 3.3. Kategorie souborů (git status) [autor]**

## 3.10. Vytvoření prvního souboru

Dáme na radu Gitu a začneme tvořit projekt (soubory projektu). Jako první, co by měl mít každý projekt, je soubor, který popisuje, čím se projekt zabývá a jak funguje. Pro naše účely to bude jednoduchý soubor *README.txt*, který vytvoříme následovně:

Otevřeme oblíbený textový editor a vytvoříme nový soubor, který uložíme jako *README.txt* do složky projektu (/Users/dan/Desktop/git_snake). Obsah souboru lze napsat vlastní nebo použít stejný (bez diakritiky) jako v příkladu 3.9. Po uložení souboru se vraťme zpátky do konzole a proveďme kontrolu, zdali zde přibyl námi vytvořený soubor pomocí příkazu `ls`. Pro zobrazení obsahu souboru použijeme příkaz `cat`, který vypíše text zadaný v textovém editoru (lze využít pro opsání textu).

**Příklad 3.9. Vytvoření README souboru**

```
# Otevřeme oblíbený textový editor
# napíšeme/opíšeme obsah souboru
# uložíme do složky projektu jako README.txt
$ pwd
/Users/dan/Desktop/git_snake
$ ls
README.txt
$ cat README.txt
GitSnake

Vlastni implementace zname hry Snake.
Projekt slouzi jako ukazka prace v systemu Git.
```

## 3.11. Přidání nesledovaných souborů do indexu

Po provedení `git status` dostaneme následující výpis:

**Příklad 3.10. Git status s nesledovanými soubory**

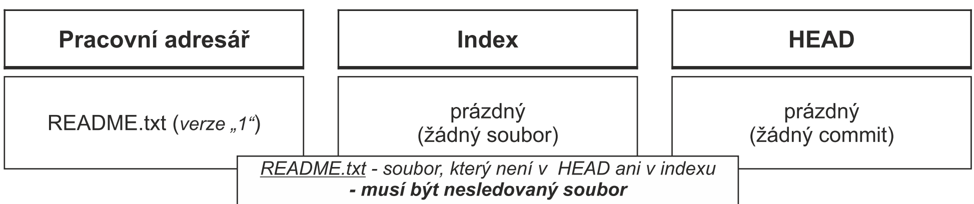
```
$ pwd
/Users/dan/Desktop/git_snake
$ git status
On branch master

Initial commit

Untracked files:
  (use "git add <file>..." to include in what will be committed) ❶

        README.txt ❷

nothing added to commit but untracked files present (use "git add" to track) ❸
```

Oproti výpisu statusu v příkladu 3.8 se výpis trochu zvětšil (přibyla sekce **nesledované soubory** a došlo k drobným změnám na posledním řádku výpisu).

❷ Nově vytvořený soubor v pracovním adresáři projektu *README.txt* se nyní nachází v kategorií nesledované soubory.

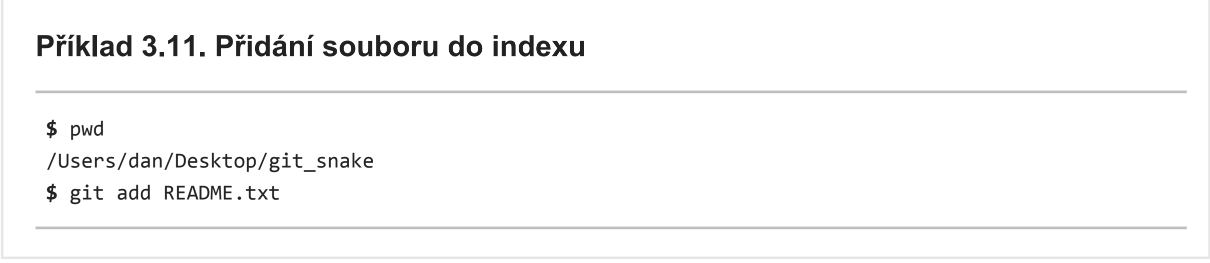
❶ Git napovídá, že pokud chceme tyto soubory (v tomto případě zatím pouze soubor *README.txt*) začít verzovat, máme použít příkaz `git add <název souboru>`, který daný soubor v pracovním adresáři projektu zkopíruje do indexu.

❸ Poslední řádek říká, že Git neví o žádných souborech, které by mohl uložit do historie větve (v indexu nejsou žádné soubory), ale ví o souborech, které je možné přidat do indexu pomocí již zmiňovaného příkazu `git add`.




**Obrázek 3.4. Kategorie souborů (git status) [autor]**

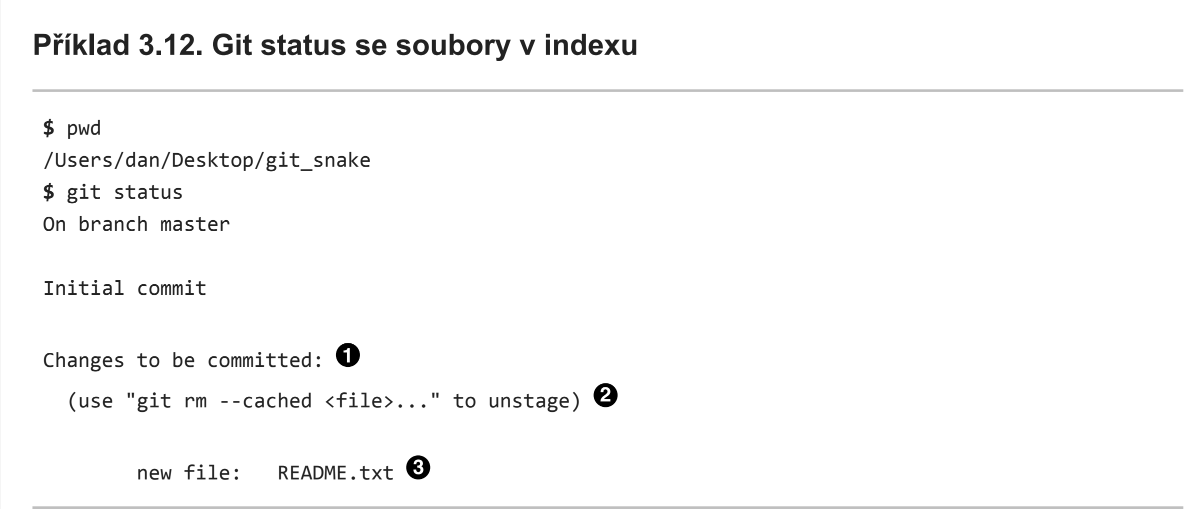
Opět dáme na radu Gitu a použijeme příkaz `git add <název souboru>` pro přidání do indexu. Vzhledem k tomu, že zatím máme pouze jeden soubor – *README.txt*, použijeme příkaz:



**Příklad 3.11. Přidání souboru do indexu**

```
$ pwd
/Users/dan/Desktop/git_snake
$ git add README.txt
```

Po provedení příkazu Git nic nevypsal, což značí, že přidání do indexu bylo úspěšné. Pro kontrolu aktuálního stavu zadejme příkaz `git status`.



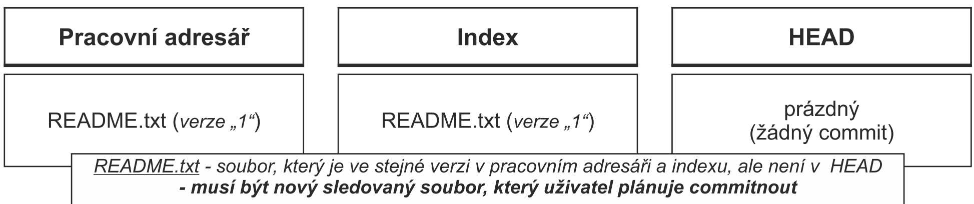
**Příklad 3.12. Git status se soubory v indexu**

```
$ pwd
/Users/dan/Desktop/git_snake
$ git status
On branch master

Initial commit

Changes to be committed: ❶
  (use "git rm --cached <file>..." to unstage) ❷

        new file:   README.txt ❸
```

❶ Sekce s nesledovanými soubory zmizela a místo ní je zde sekce s názvem změny připravené k zapsání (**index**). Soubory v indexu (zatím stále pouze soubor *README.txt*) ukazují, jaké nové verze souborů budou uloženy do historie větve, pokud bude proveden commit.

❷ Git nabízí možnost, kterou lze využít. Jedná se o příkaz `git rm --cached <název souboru>`, který odstraní daný soubor z indexu.

❸ V indexu je soubor *README.txt* připraven pro zapsání, který Git označil jako nový, neboť v *HEAD* nebyl nalezen soubor s tímto názvem.




**Obrázek 3.5. Kategorie souborů (git status) [autor]**

## 3.12. Uložení změn do historie projektu

> Commit je jediný způsob jak lze uložit změnu do historie a každá změna v historii musí být uvedena commitem.
>
> — volně přeloženo z [17]

Když máme připravené změny pro uložení v indexu, stačí provést příkaz `git commit`. Po provedení příkazu Git uloží data v indexu do historie aktuální větve (*master*).

**Příklad 3.13. Git commit**

```
$ pwd
/Users/dan/Desktop/git_snake
$ git commit -m "Přidání README projektu" ❶
[master (root-commit) d3173a9] Přidání README projektu ❷
 1 file changed, 4 insertions(+) ❸
 create mode 100644 README.txt ❹
```

❶ Příkazu `git commit` byl předán přepínač *-m*, který slouží k specifikování popisu změny. Jak správně psát popisy změn bude ukázáno v kapitole 5.3.

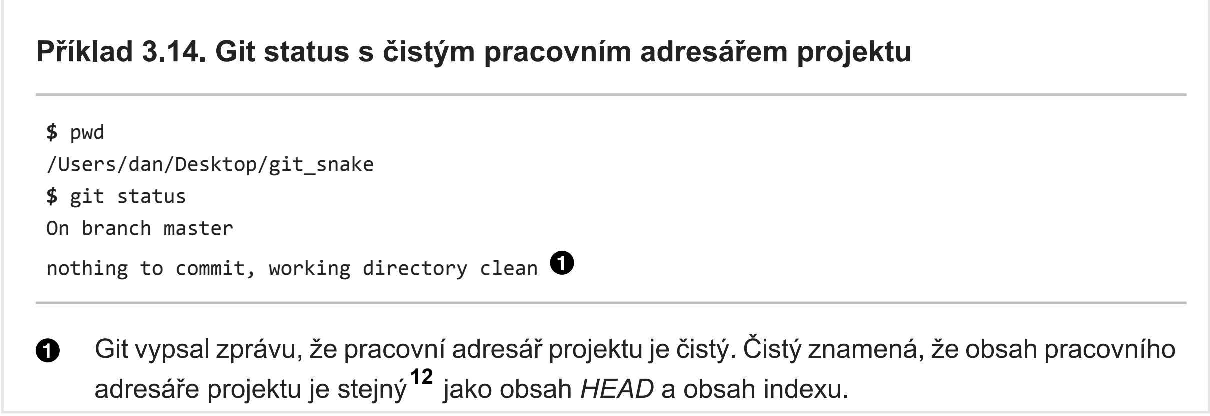
❷ Tento řádek vypisuje informace o nově vytvořeném commitu. Konkrétně, že commit byl proveden na *master* větvi a také, že tento commit je kořen **(root-commit)**. Poslední slovo mezi hranatými závorkami je „d3173a9“, což je prvních 7 znaků **commit-id**. *Commit-id* bude podrobněji vysvětleno v následující kapitole. Pokud následujeme příkazy v této práci, je pravděpodobné, že se nám vypsaly znaky jiné, toto chování je normální a bude vysvětleno v následující kapitole. Za hranatými závorkami je vypsán první řádek popisu změny.

❸ Souhrn toho, co se v commitu událo, 1 soubor byl změněn (*README.txt*) a byly přidány (+) 4 řádky (počet řádků v souboru *README.txt*).

❹ Poslední řádek vypisuje informace o nových a smazaných souborech obsažených v commitu. Soubor *README.txt* nebyl v předchozí *HEAD*, proto byl nově vytvořen (create). Režim

100644 je spojení dvou čísel, čísla 100, což značí klasický (nelinkový) soubor a čísla 644, které označuje podmnožinu[11] unixových přístupových práv k souboru.

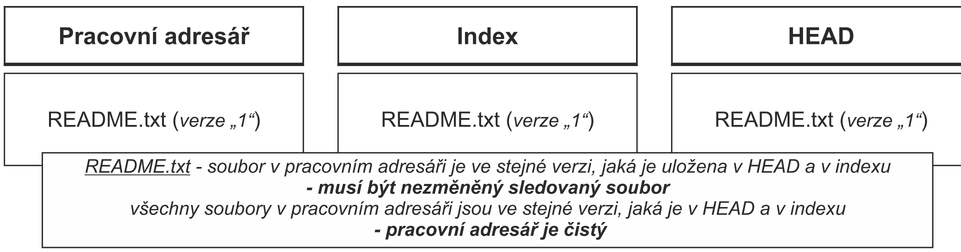
Po provedení příkazu `git status` dostaneme následující výpis:



**Příklad 3.14. Git status s čistým pracovním adresářem projektu**

```
$ pwd
/Users/dan/Desktop/git_snake
$ git status
On branch master
nothing to commit, working directory clean ❶
```

❶ Git vypsal zprávu, že pracovní adresář projektu je čistý. Čistý znamená, že obsah pracovního adresáře projektu je stejný[12] jako obsah *HEAD* a obsah indexu.




**Obrázek 3.6. Kategorie souborů (git status) [autor]**

## 3.13. Git log

Pro zobrazení historie větve projektu použijeme příkaz `git log <název větve>`:

**Příklad 3.15. Git log**

```
$ pwd
/Users/dan/Desktop/git_snake
$ git log master
commit d3173a992e609a9b03f73239e15610987f11935e
Author: Daniel Maixner <daniel-maixner@seznam.cz>
Date:   Sat Mar 12 11:53:58 2016 +0100

    Přidání README projektu
```

Příkazy Gitu mají nastavený výstup do stránkovacího programu, kterým je ve výchozím nastavení `less`. Pokud se tedy nevejde výstup příkazu na jednu obrazovku konzole, dojde ke stránkování.

[11] Git ve starších verzích podporoval plná unixová práva k souboru, ovšem tato vlastnost přinášela více obtíží než užitku, a proto byla práva k souboru zredukována na podmnožinu unixových práv.

[12] Za předpokladu, že v pracovním adresáři projektu nejsou ignorované soubory.

# 4. Princip Gitu

Předchozí kapitola byla zakončena vytvořením prvního snímku projektu pomocí příkazu `git commit`. V této kapitole bude podrobněji vysvětlen princip Gitu.

> Git je v podstatě obsahově adresovatelný souborový systém s uživatelským rozhraním verzovacího systému postaveném nad tímto souborovým systémem.
>
> — volně přeloženo z [15]

## 4.1. Objekty

Objektem v Gitu je myšleno vše, co je uloženo v **objektové databázi**, která se ve výchozím nastavení nachází ve složce *.git/objects/*. Objektová databáze funguje na principu **key-value (klíč-hodnota)**, kde klíčem je **identifikátor objektu** a hodnotou je **obsah objektu**.

### 4.1.1. Identifikátor objektu

Obdobně jako například knihy mají ISBN[1] kód, který jednoznačně identifikuje určitou knihu, Git používá podobnou identifikaci pro odkazování na **objekty** v Gitu. Tato identifikace objektu se v Gitu označuje **otisk**. Git vytváří otisk z **Gitem přidané hlavičky a obsahu objektu** (dále bude pod pojmem **obsah objektu myšlena hlavička + obsah objektu**).[2]

Otisky Git realizuje pomocí hašovací funkce SHA-1, která z libovolných vstupních dat vytvoří sekvenci 160 bitů. [16] Vlastnost funkcí je ta, že pro stejný vstup (obsah) generují stejný výstup (otisk), jinými slovy soubory s identickým obsahem mají stejné otisky.[3] Pracovat s výstupem 160 nul a jedniček funkce SHA-1 je nepraktické, proto se tato sekvence v Gitu reprezentuje pomocí čtyřiceti hexadecimálních[4] znaků. SHA-1 otisk může vypadat například následovně:

```
763fe41beb3a113bd09fd9bd16762067bf7f1cdb
```

Pro odkazování na objekt pomocí 40 hexadecimálních znaků je pro uživatele nepraktické, proto lze v Gitu použít pouze tolik prvních hexadecimálních znaků z otisku, kolik je třeba, aby Git nalezl právě jeden objekt s otiskem začínajícím touto sekvencí znaků, minimálně však musejí být napsány znaky čtyři.

[1] https://cs.wikipedia.org/wiki/International_Standard_Book_Number
[2] Formát obsahu objektu je následující: *<typ objektu><mezera><velikost obsahu v bajtech><null byte><obsah>*.
[3] Kvůli pevně danému výstupu 160 bitů z funkce SHA-1 při libovolném vstupu jsou identifikátory objektů považovány za efektivně jednoznačné. Efektivně jednoznačné znamená, že sice je zde pravděpodobnost kolize identifikátoru dvou a více objektů s odlišným obsahem, ovšem tato pravděpodobnost je tak malá, že ji prostě ignorujeme.
[4] https://cs.wikipedia.org/wiki/Šestnáctková_soustava

### 4.1.2. Objektová databáze

Všechny objekty v Gitu jsou neměnné[5] a jednoznačně se identifikují pomocí zmiňovaného SHA-1 otisku obsahu daného objektu.[6] Objekty v Gitu se dělí na čtyři typy:

1. Commit
2. Strom (tree)
3. Blob[7]
4. Komentovaný štítek (annotated tag)[8]

Pokud vyhledáme všechny soubory v objektové databázi, dostaneme výstup podobný příkladu 4.1.

**Příklad 4.1. Vyhledání souborů v objektové databázi**

```
$ pwd
/Users/dan/Desktop/git_snake
$ find .git/objects/ -type f ❶
.git/objects/5e/560c018223107e71f65cdb46fabd65d2889454
.git/objects/7b/04950e85ed3557721a0ec6a7b94359488cfb0b
.git/objects/d3/173a992e609a9b03f73239e15610987f11935e
```

❶ Výsledkem vyhledání souborů v objektové databázi jsou tři soubory. Otisky objektů (**názvy souborů**) máme pravděpodobně odlišné.

Aby v objektové databázi časem nebylo příliš mnoho souborů, Git vždy vezme první dva hexadecimální znaky otisku, ze kterých Git vytvoří adresář, a zbylých 38 znaků Git použije jako název souboru v daném adresáři. Tímto způsobem se tak v objektové databázi jednotlivé objekty rozdělí do maximálně 256 podsložek. V Gitu se však na otisky odkazuje pomocí celého otisku (rozdělování do podadresářů je pouze optimalizace pro souborový systém). V následujících sekcích bude rozebráno, k čemu jednotlivé typy objektů slouží a které typy objektů máme aktuálně uloženy v objektové databázi.

### 4.1.3. Commit

V kapitole 3.12 jsme vytvořili první commit projektu *GitSnake*, konkrétně v příkladu 3.13 nám Git po provedení commitu vypsal *commit-id*. *Commit-id* je termín používaný v Gitu pro otisky objektů typu commit. Pravděpodobně tušíme, že jeden z oněch tří souborů v příkladu 4.1 bude nejspíše typu commit. Pokud se podíváme na *commit-id* z příkladu 3.13, opravdu ve výstupu příkladu 4.1 soubor s „názvem" začínajícím „d3173a9" je. Obsahem všech objektů je kvůli ušetření místa na disku *DEFLATE* komprese samotného obsahu objektu, kvůli této kompresi musíme pro zjištění typu a zobrazení obsahu objektu použít například příkaz z jádra Gitu – `git cat-file`, který s přepínačem *-t* vrátí typ objektu s daným otiskem jako parametrem a s přepínačem *-p* vypíše formátovaný obsah objektu (souboru). [15]

[5] Neměnné objekty znamenají, že nelze změnit jejich obsah, aniž by nedošlo ke změně otisku.
[6] https://en.wikipedia.org/wiki/Content-addressable_storage
[7] https://cs.wikipedia.org/wiki/BLOB
[8] Komentovaný štítek bude probrán v kapitole 4.5.

### Příklad 4.2. Zobrazení commitu

```
$ pwd
/Users/dan/Desktop/git_snake
$ git cat-file -t d3173a9
commit ❶
$ git cat-file -p d3173a9 ❷
tree 5e560c018223107e71f65cdb46fabd65d2889454
author Daniel Maixner <daniel-maixner@seznam.cz> 1457780038 +0100
committer Daniel Maixner <daniel-maixner@seznam.cz> 1457780038 +0100

Přidání README projektu
```

❶ Objekt s otiskem „d3173a9“ je podle očekávání typu commit.

❷ Obsahem commitu, kromě již známých[9] údajů (popis, autor a datum změny[10]), je také odkaz na objekt typu **strom** s otiskem „5e560c01“.

## 4.1.4. Strom

Strom v Gitu reprezentuje hierarchickou adresářovou strukturu, konkrétně jeden strom reprezentuje obsah jedné úrovně složky. Obsahem stromu jsou názvy souborů a adresářů v dané úrovni stromu spolu s metadaty (přístupová práva, typ objektu, otisk objektu).[11] Vzhledem k hierarchii, kterou stromy vytvářejí, stačí ukládat v commitu pouze strom, který reprezentuje nejvyšší úroveň adresáře (pracovní adresář projektu), tento strom se označuje **kořenový**. Ostatní úrovně zanoření se z tohoto kořenového stromu dají již sestavit. Pro zobrazení obsahu kořenového stromu z příkladu 4.2 využijeme otisk z příkladu 4.2. Otisk můžeme použít celý *5e560c018223107e71f65cdb46fabd65d2889454* anebo jen pár prvních znaků, které jednoznačně určí, který objekt chceme zobrazit, většinou prvních 8 znaků je dostačujících.

### Příklad 4.3. Zobrazení stromu

```
$ pwd
/Users/dan/Desktop/git_snake
$ git cat-file -t 5e560c018223107e71f65cdb46fabd65d2889454
tree
$ git cat-file -p 5e560c01
100644 blob 7b04950e85ed3557721a0ec6a7b94359488cfb0b    README.txt ❶
```

❶ V době commitu z příkladu 3.13 byl v indexu pouze soubor *README.txt* uložený v kořenovém adresáři. Výstupem příkazu je jeden řádek s otiskem objektu typu **blob** *7b04950e85ed3557721a0ec6a7b94359488cfb0b*, režimem *100644* a názvem *README.txt*.

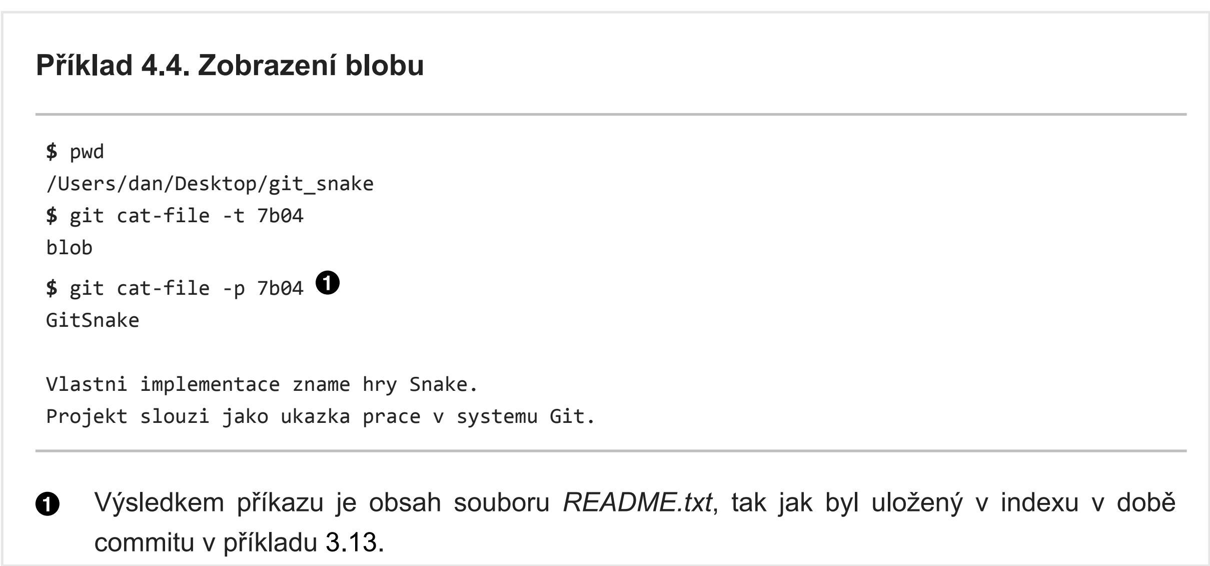
[9] Ještě by zde měl být uložen otisk rodiče tohoto commitu, ale jelikož tento commit je kořen, tak zde žádný otisk rodiče uložen není.

[10] Git rozlišuje mezi autorem a datem změny (author) a autorem a datem commitu (committer). Datum je ukládán v obecném (Unix epoch) formátu: *<počet vteřin od 1.1.1970 utc> <časový posun pásma>*.

[11] https://en.wikipedia.org/wiki/Merkle_tree

### 4.1.5. Blob

Blob v Gitu reprezentuje soubor, přesněji **obsah** souboru. Pro ověření typu a zobrazení obsahu objektu získaného v příkladu 4.3 použijeme opět příkaz `git cat-file`:

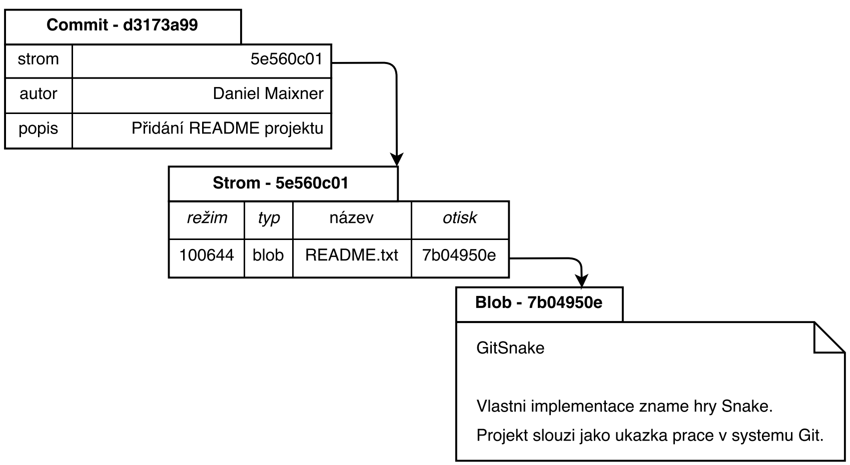


**Příklad 4.4. Zobrazení blobu**

```
$ pwd
/Users/dan/Desktop/git_snake
$ git cat-file -t 7b04
blob
$ git cat-file -p 7b04 ❶
GitSnake

Vlastni implementace zname hry Snake.
Projekt slouzi jako ukazka prace v systemu Git.
```

❶ Výsledkem příkazu je obsah souboru *README.txt*, tak jak byl uložený v indexu v době commitu v příkladu 3.13.

Pokud následujeme příklady v této práci, tak pokud jsme použili stejný obsah a název souboru *README.txt* jako v této práci, pak bychom měli mít stejné otisky jak objektu typu blob, tak i objektu typu strom. Pro objekt typu commit však naše otisky budou pravděpodobně odlišné. Otisk commitu je odlišný především kvůli datu commitu. Zjednodušené schéma objektů v objektové databázi ukazuje obrázek 4.1




**Obrázek 4.1. Objektová databáze (zjednodušeno) [autor]**

## 4.2. Přidání souborů praktického projektu

V minulé sekci jsme si rozebrali objekty, které byly vytvořeny v kapitole 3. V této sekci přidáme další soubory hry *GitSnake* a také rozšíříme objektovou databázi. Jak již bylo řečeno, hra bude hratelná ve webovém prohlížeči, proto byla zvolena technologie pro publikování webových stránek, konkrétně

značkovací jazyk **HTML**[12] (ve verzi 5) a programovací jazyk **ECMAScript**[13] (ve verzi 6) v implementaci **JavaScript**[14]. Začneme vytvořením základních souborů hry v pracovním adresáři projektu:

**Příklad 4.5. Přidání souborů praktického projektu**

```
$ pwd
/Users/dan/Desktop/git_snake
$ touch play.htm
$ mkdir js
$ touch js/Point2D.js js/SquareObject.js js/Snake.js js/Game.js
```

Všechny vytvořené soubory v pracovním adresáři projektu přidáme do indexu. Pro přidání do indexu využijeme zástupné znaky konzolového interpretu:

**Příklad 4.6. Přidání všech .htm a .js souborů do indexu**

```
$ pwd
/Users/dan/Desktop/git_snake
$ git add *.htm *.js ❶
```

❶ Příkazu `git add` byly předány „dva“ parametry, které konzolový interpret převedl na aktuální cesty k souborům. První parametr specifikuje všechny *HTML* soubory (přípona *.htm*) v pracovním adresáři projektu a druhý parametr všechny soubory *JavaScriptu* (přípona *.js*).[15]

Při přidání do indexu Git vytváří objekty typu blob přidaných souborů. Vytvoření objektů typu blob můžeme ověřit:

**Příklad 4.7. Vyhledání souborů v objektové databázi**

```
$ pwd
/Users/dan/Desktop/git_snake
$ find .git/objects/ -type f
.git/objects/5e/560c018223107e71f65cdb46fabd65d2889454
.git/objects/7b/04950e85ed3557721a0ec6a7b94359488cfb0b
.git/objects/d3/173a992e609a9b03f73239e15610987f11935e
.git/objects/e6/9de29bb2d1d6434b8b29ae775ad8c2e48c5391 ❶
$ git cat-file -t e69d
blob ❷
```

❶ V objektové databázi oproti výstupu v příkladu 4.1 přibyl pouze jeden objekt.

❷ Nově vytvořený objekt je typu blob.

[12] https://www.w3.org/html/

[13] http://www.ecma-international.org/publications/standards/Ecma-262.htm

[14] https://cs.wikipedia.org/wiki/JavaScript

[15] Pro převedení zástupných znaků na aktuální cesty samotným Gitem je nutné použít uvozovky okolo parametrů.

Výhoda tvorby otisku je ta, že pokud chceme uložit objekt, jehož otisk je již uložen v objektové databázi (obsahy jsou stejné), Git nemusí tento objekt ukládat znovu[16], stačí, když Git odkáže na již vytvořený objekt pomocí otisku. Tímto způsobem dokáže Git efektivně ukládat soubory se stejným obsahem. Vzhledem k tomu, že všechny soubory byly vytvořeny pomocí příkazu `touch`, který mimo jiné vytvoří soubory s prázdným obsahem, pokud soubory s cestou předanou jako parametrem neexistují. Tedy všechny nově vytvořené soubory pomocí příkazu `touch` mají stejný (prázdný) obsah, proto se vytvořil pouze jeden objekt typu blob, na který odkazují dané cesty ve **virtuálním stromu v indexu**.

## 4.3. Virtuální strom indexu

Git sleduje primárně změny v obsazích souborů a k těmto obsahům přiřazuje názvy souborů jako sekundární údaj. Pro přiřazování názvů souborů k objektu typu blob, jak již víme, Git používá stromy. Kvůli častým změnám v indexu Git vytváří virtuální strom, který se ukládá spolu s dalšími informacemi ve zdrojovém souboru indexu (*.git/index*) a až v případě commitu, Git tento virtuální strom projde a vytvoří z něj objekty typu strom, které zapíše do objektové databáze spolu s objektem typu commit, který odkazuje na kořenový strom. Pro přibližné zobrazení virtuálního stromu indexu lze použít příkaz `git ls-files --stage`.

Pomocí příkazu z jádra Gitu – `git write-tree` můžeme instruovat Git, aby aktuální virtuální strom v indexu převedl a uložil do objektové databáze. Výsledkem příkazu `git write-tree` je otisk kořenového stromu. Pomocí otisku kořenového stromu můžeme vytvořit commit, například pomocí příkazu opět z jádra Gitu – `git commit-tree`. Pro zjednodušení ovšem použijeme rovnou uživatelský příkaz `git commit`, který provede vše v jednom příkazu:

**Příklad 4.8. Vytvoření základních souborů hry**

```
$ pwd
/Users/dan/Desktop/git_snake
$ git commit -m "Vytvoření základních souborů hry"
[master ccf57ac] Vytvoření základních souborů hry
 5 files changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 js/Game.js
 create mode 100644 js/Point2D.js
 create mode 100644 js/Snake.js
 create mode 100644 js/SquareObject.js
 create mode 100644 play.htm
```

Commit proběhl v pořádku, bylo přidáno 5 souborů a nepřibyly žádné řádky (všechny přidané soubory mají prázdný obsah). Pro lepší ukázku znovu použitelnosti objektů se stejným obsahem změníme pouze obsah souboru *play.htm* v kořenovém adresáři a provedeme commit. Pro commit použijeme možnost předat příkazu `git commit` jako parametr názvy sledovaných změněných souborů, které chceme *commit*nout. Takto definovaný příkaz ignoruje index, tedy v commitu budou změněny pouze soubory v parametru tak, jak jsou uloženy v pracovním adresáři projektu:

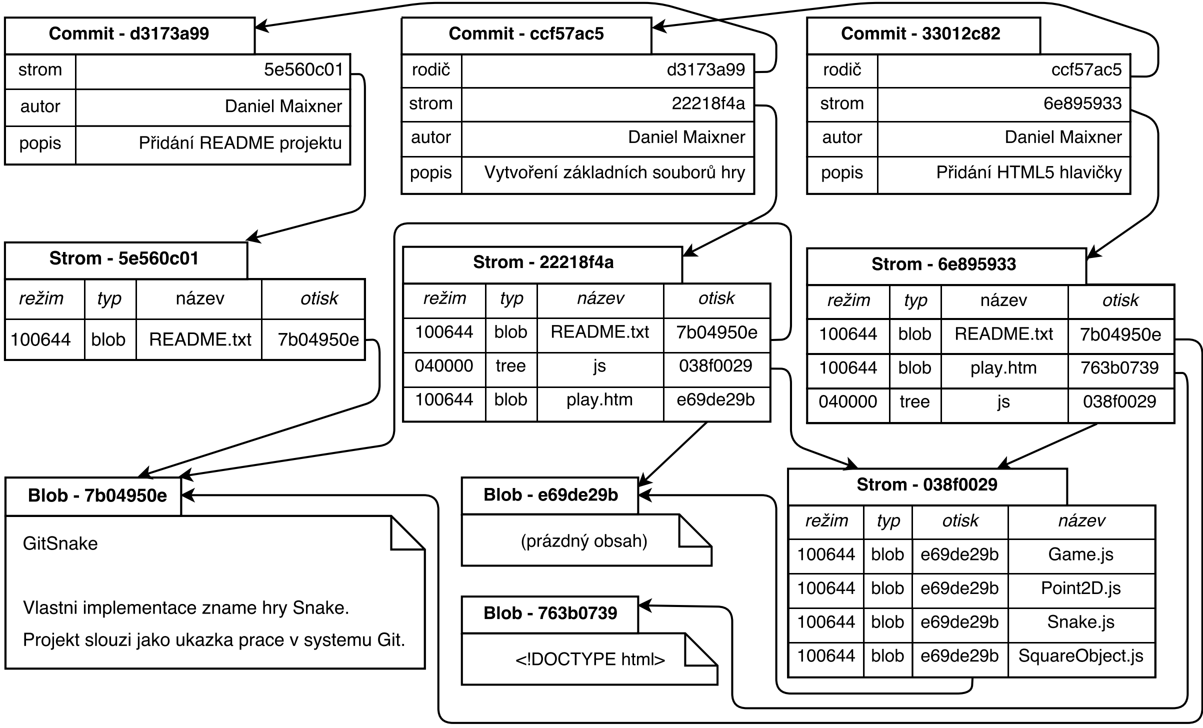
[16] Git tento objekt ani znovu uložit nemůže, neboť otisk je názvem souboru.

### Příklad 4.9. Přidání HTML5 hlavičky

```
$ pwd
/Users/dan/Desktop/git_snake
$ echo -n '<!DOCTYPE html>' > play.htm ❶
$ git commit -m "Přidání HTML5 hlavičky" play.htm
[master 33012c8] Přidání HTML5 hlavičky
 1 file changed, 1 insertion(+)
```

❶ Uložení textu *<!DOCTYPE html>* do souboru *play.htm*.

V příkladu 4.9 došlo ke změně pouze jednoho blob objektu a daného kořenového stromu. Ve složce *js* se v commitu z příkladu 4.9 nic nezměnilo, proto otisk stromu adresáře *js* je stejný jako v commitu předešlém a nově vytvořený kořenový strom pouze odkáže na objekty již uložené v objektové databázi (viz obrázek 4.2).




**Obrázek 4.2. Objektová databáze (zjednodušeno) [autor]**

Pro získání příslušné adresářové struktury v commitu stačí Gitu pouze odkaz na commit, pomocí kterého již Git nalezne všechny objekty související s tímto commitem. V následujících kapitolách proto budou v grafickém vyjádření historie projektu pro zjednodušení použity pouze objekty typu commit.

## 4.4. Reference

Odkazovat na objekty v Gitu pouze pomocí otisku objektu je pro uživatele nepraktické, a proto Git nabízí možnost použití referencí. Reference jsou v Gitu **pojmenované odkazy na objekty**. Reference se v Gitu dělí na dvě hlavní kategorie:

1. Absolutní reference – reference na objekt.
   - Hlavy (složka *.git/refs/heads/*) – lokální větve.
   - Štítky (složka *.git/refs/tags/*) – odkazy na objekty, nejčastěji typu commit.
   - Vzdálené (složka *.git/refs/remotes/*) – vzdálené větve.
2. Symbolické reference – reference na jinou referenci.

Dosud jsme se v této práci setkali se dvěma referencemi – *HEAD* a *master*.

### 4.4.1. Master

Pro udržování záznamu o historii projektu musí Git udržovat sekvenci posloupnosti commitů, která vytváří historii. Protože projekt může mít více nezávislých linií historie, Git používá koncept větví. Výchozí větev, kterou Git automaticky vytvoří při prvním commitu, se jmenuje *master*. Větev *master* je většinou považováno za hlavní linií projektu, ale není to pravidlo.

### 4.4.2. Větve

> Vytvoření větve není nic víc než uložení 40 znaků do souboru.
>
> — volně přeloženo z [18]

Již jsme viděli, že v commitu samotném není uložen údaj, do jaké větve commit patří. V commitu je však uložen údaj o rodiči, kde již bylo řečeno, že commit může mít i více rodičů. Commity v Gitu vytváří **orientovaný acyklický graf**[17]. Díky vztahu rodič-dítě mezi commity stačí, aby větev v Gitu odkazovala na určitý commit a od tohoto commitu se již daná historie větve dopočítá pomocí všech generací rodičů tohoto commitu. Historie větve tvoří všechny commity, na které se dá pomocí *následování šipek* (orientovaný graf) dostat z commitu, na nějž větev odkazuje, aniž bychom se dostali zpátky do stejného commitu (acyklický graf). V projektu *GitSnake* máme zatím pouze lineární historii projektu se třemi commity (viz obrázek 4.3). Pro zobrazení otisku, na který odkazuje větev *master*, použijeme příkaz `cat` a pro vypsání všech commitů ve větvi *master* použijeme například příkaz `git log --oneline master`:

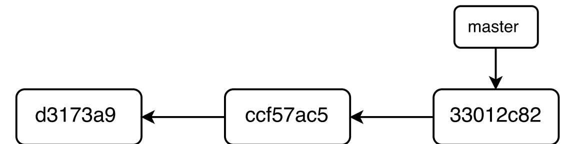
**Příklad 4.10. Vypsání historie větve master**

```
$ pwd
/Users/dan/Desktop/git_snake
$ cat .git/refs/heads/master ❶
33012c824e321a467c298891013f280ce502db02
$ git log --oneline master ❷
33012c8 Přidání HTML5 hlavičky
ccf57ac Vytvoření základních souborů hry
d3173a9 Přidání README projektu
```

❶ Lokální větve jsou ukládány ve složce *.git/refs/heads/<název větve>*. Po vypsání obsahu souboru *master* dostáváme absolutní odkaz na commit „33012c82“.

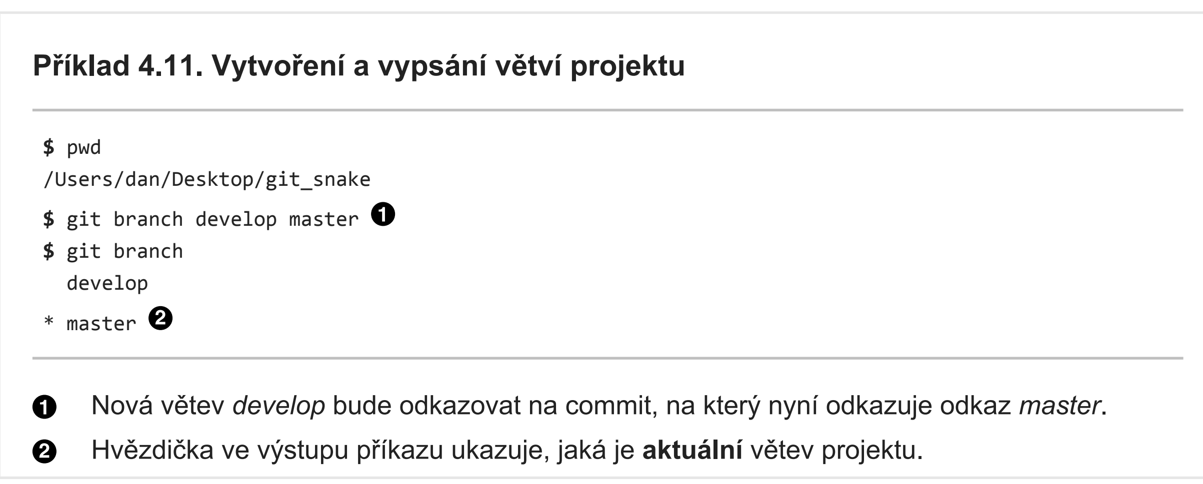
[17] https://en.wikipedia.org/wiki/Directed_acyclic_graph

❷ Výpis historie větve *master* od commitu, na který *master* odkazuje, až po *root-commit*.




**Obrázek 4.3. Historie větve master [autor]**

Jelikož projekt *GitSnake* je stále ve vývoji a větev *master* má podle konvencí obsahovat hlavní („stabilní“) větev projektu, vytvoříme větev novou, kterou pojmenujeme *develop*. Vytvoření větve provedeme pomocí příkazu `git branch <název nové větve> <commit, na který nová větev bude odkazovat>`. Samotný příkaz `git branch` vypíše seznam lokálních větví projektu.



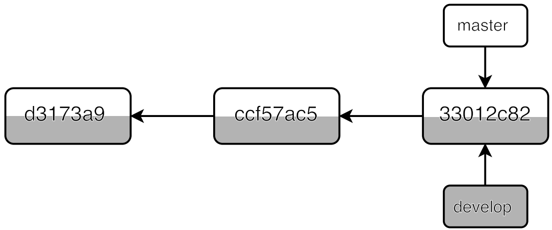
**Příklad 4.11. Vytvoření a vypsání větví projektu**

```
$ pwd
/Users/dan/Desktop/git_snake
$ git branch develop master ❶
$ git branch
  develop
* master ❷
```

❶ Nová větev *develop* bude odkazovat na commit, na který nyní odkazuje odkaz *master*.

❷ Hvězdička ve výstupu příkazu ukazuje, jaká je **aktuální** větev projektu.

Historie větví *master* a *develop* jsou stejné, neboť obě větve odkazují na stejný commit (viz obrázek 4.4).




**Obrázek 4.4. Historie větví master a develop [autor]**

## 4.4.3. HEAD

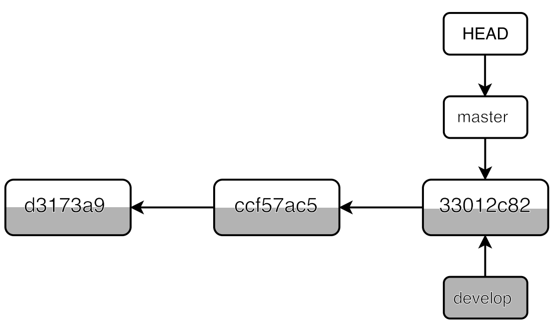
Již víme, že *HEAD* nejčastěji odkazuje na „poslední“ commit v aktuální větvi a také že *HEAD* je vždy rodič následujícího commitu. Pokud si zobrazíme obsah reference *HEAD* ukládané v souboru *.git/HEAD*, dostaneme následující:

**Příklad 4.12. Zobrazení HEAD**

```
$ pwd
/Users/dan/Desktop/git_snake
```

```
$ cat .git/HEAD
ref: refs/heads/master ❶
```

❶ Výstupem je symbolický zápis používaný v Gitu. *HEAD* je v tomto případě symbolický odkaz na lokální větev *master* (viz obrázek 4.5).




**Obrázek 4.5. Reference HEAD [autor]**

Příkaz `git commit` automaticky nastavuje referenci *HEAD*, aby odkazovala na nově vytvořený commit. Proto stačí, když změníme referenci *HEAD*, aby odkazovala na nově vytvořenou větev *develop*, a následující commity budou součástí historie pouze větve *develop*. Pro změnu aktuální větve můžeme přímo upravit obsah souboru *.git/HEAD*, ale ve složce *.git* je lepší pouze číst nežli zapisovat, proto použijeme příkaz `git checkout <název větve>`, který slouží k nastavení větve v parametru jako aktuální. Historii reference *HEAD* lze zobrazit pomocí příkazu `git reflog`.

### Příklad 4.13. Změna aktuální větve

```
$ pwd
/Users/dan/Desktop/git_snake
$ git checkout develop
Switched to branch 'develop'
$ git branch
* develop
  master
$ cat .git/HEAD
ref: refs/heads/develop
```

Pro přidání commitu ve větvi *develop* otevřeme v textovém editoru soubor *play.htm* uložený v pracovním adresáři projektu a upravíme obsah, aby vypadal následovně (**tučným** písmem je označen nový obsah):

**play.htm**

```
<!DOCTYPE html><html>
<head><meta charset="UTF-8"><title>GitSnake</title>
<style>#canvas { border: 2px solid black; }</style>
</head>
<body>
    <canvas width="600" height="300" id="canvas">Váš prohlížeč není podporován</canvas>
</body>
```

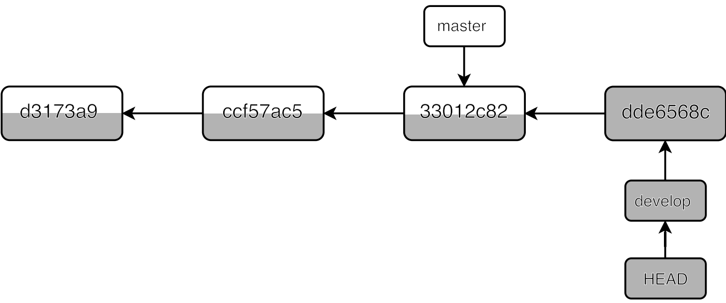
```
</html>
```

Po úpravě uložíme a vrátíme se do konzole, kde provedeme commit a následně pomocí přepínačů příkazu `git log` ověříme, že se historie větví projektu rozdělila.

**Příklad 4.14. Přidání základního HTML kódu v play.htm**

```
$ pwd
/Users/dan/Desktop/git_snake
$ git add play.htm
$ git commit -m "Přidání základního HTML kódu v play.htm"
[develop dde6568] Přidání základního HTML kódu v play.htm
 1 file changed, 8 insertion(+), 1 deletion(-)
$ git log --oneline --decorate HEAD ❶
dde6568 (HEAD -> develop) Přidání základního HTML kódu v play.htm
33012c8 (master) Přidání HTML5 hlavičky
ccf57ac Vytvoření základních souborů hry
d3173a9 Přidání README projektu
```

❶ Přepínač *--decorate* vypisuje názvy referencí, které odkazují na dané commity.




**Obrázek 4.6. Rozdělení historie větví projektu [autor]**

Pro práci s odkazy Git poskytuje příkaz `git rev-parse`, který mimo jiné dokáže z předaného parametru vrátit absolutní referenci. Tento příkaz Git vnitřně spouští na všech parametrech, kde očekává otisk. Díky tomu lze používat reference místo otisku. Většina příkazů Gitu očekávající v parametru otisk commitu dosadí *HEAD*, pokud tento parametr nevyplníme.

## 4.5. Štítek

Štítek v Gitu slouží pro označení objektu. Nejčastěji se označuje (**taguje**) objekt typu commit. Štítky slouží pro přiřazení názvu pro „významný“ commit, například nová verze projektu apod. Štítky v Gitu jsou dvojího typu:

1. Odlehčený štítek – absolutní reference ukládaná ve složce *.git/refs/tags/*.
2. Komentovaný štítek – objekt typu tag ukládaný v objektové databázi.

Odlehčené štítky se vzhledem k jejich možnosti změny moc nepoužívají, naproti tomu komentované štítky se hojně využívají pro tagování specifické verze historie. Populární systém tagování projektu

je *sémantické verzování*[18], ovšem toto schéma je pouze doporučení a je vždy na vývojářích projektu, na jakých konvencích se domluví. Pro otagování prvního commitu projektu *GitSnake* pomocí komentovaného štítku použijeme příkaz `git tag -a -m <komentář> <identifikace> <commit>`:

**Příklad 4.15. Přidání štítku pro první commit projektu**

```
$ pwd
/Users/dan/Desktop/git_snake
$ git rev-parse HEAD~3 ❶
d3173a992e609a9b03f73239e15610987f11935e
$ git tag -a -m "První commit GitSnake" v0.1.0 HEAD~3 ❷
$ git log --oneline --decorate
dde6568 (HEAD -> develop) Přidání základního HTML kódu v play.htm
33012c8 (master) Přidání HTML5 hlavičky
ccf57ac Vytvoření základních souborů hry
d3173a9 (tag: v0.1.0) Přidání README projektu ❸
$ git cat-file -t v0.1.0 ❹
tag
$ git tag ❺
v0.1.0
```

❶ Pro získání otisku prvního commitu jsme použili *HEAD~3*. Tento zápis znamená, že chceme třikrát prvního předka commitu, na který *HEAD* odkazuje.[19] Vzhledem k tomu, že zatím máme lineární historii projektu (všechny commity mají maximálně jednoho rodiče), je to třetí commit před *HEAD*.

❷ Pro vytvoření komentovaného štítku pro commit *HEAD~3* jsme předali příkazu `git tag` přepínač *-a* pro specifikování komentovaného štítku a přepínač *-m* pro komentář štítku.

❸ Komentované štítky se také objevují ve výstupu příkazu `git log`, pokud použijeme přepínač *--decorate*.

❹ Typ objektu, na který odkazuje „reference“ *v0.1.0*, je typu *tag*. Pro vypsání formátovaného obsahu štítku lze použít příkaz `git cat-file -p v0.1.0`.

❺ Samotný příkaz `git tag` slouží pro vypsání všech štítků projektu.

Štítky slouží pro označení milníku projektu. V příkladu 4.15 jsme použili označení *v0.1.0* pro první commit projektu *GitSnake*, díky tomuto štítku můžeme kdykoli v budoucnu odkazovat na commit „d3173a99“ jako na *verzi 0.1.0* namísto hledání a použití otisku prvního commitu. Komentované štítky a commity se dají digitálně podepsat pomocí GnuPG[20]. [15]

[18] http://semver.org/
[19] https://git-scm.com/docs/gitrevisions
[20] https://www.gnupg.org/

# 5. Příkazy Gitu a související akce

V minulé sekci byl podrobně rozebrán princip Gitu. V této kapitole budou probrány základní příkazy pro práci s Gitem.

V textovém editoru otevřeme soubor *Point2D.js* ze složky *js* projektu *GitSnake* a upravíme obsah, aby vypadal následovně:

**js/Point2D.js**

```
"use strict";
GitSnake.Point2D = class Point2D {
    constructor(x, y) {
        this.x = x; this.y = y;
    }
}
```

## 5.1. Diff

Program `diff` slouží, ve výchozím nastavení, pro vypsání rozdílů jednotlivých **řádků** mezi dvěma soubory. Program `diff` je základní program definovaný v [2], ale kvůli lepší podpoře operačních systému a spolupráce s Gitem je program `git diff` součástí Gitu.[1] Pro zobrazení rozdílu mezi sledovanými soubory v pracovním adresáři projektu a sledovanými soubory v indexu použijeme příkaz `git diff`. Po zobrazení rozdílu (**diffu**) přidáme soubor *Point2D.js* do indexu.

**Příklad 5.1. Zobrazení rozdílu, který lze přidat do indexu**

```
$ pwd
/Users/dan/Desktop/git_snake
$ git diff
diff --git a/js/Point2D.js b/js/Point2D.js ❶
index e69de29..3b8b34b 100644 ❷
--- a/js/Point2D.js
+++ b/js/Point2D.js
@@ -0,0 +1,6 @@
+"use strict";
+GitSnake.Point2D = class Point2D {
+    constructor(x, y) {
+        this.x = x; this.y = y; ❸
+    }
+}
\ No newline at end of file ❹
$ git add js/Point2D.js
```

❶ Spuštění programu `git diff` pro dva soubory (verze *A* a *B* souboru *js/Point2D.js*).

[1] Pro spuštění externího nástroje pro zobrazení rozdílu lze použít příkaz `git difftool`.

❷ Verze *A* je blob „e69de29“ (prázdný obsah) v indexu a verze *B* je blob „3b8b34b“ (nově přidaný obsah) v pracovním adresáři projektu.

❸ Ve výstupu příkazu `git diff` jsou jednotlivé **přidané** řádky označeny na začátku symbolem **+** a **odstraněné** řádky symbolem **-**.

❹ Upozornění, že na konci souboru není odřádkováno. Obecně je doporučeno, pokud to zdrojový soubor dovoluje, na konci souboru odřádkovat pro předcházení konfliktům (více o konfliktu v kapitole 5.29).

V pracovním adresáři projektu jsme změnili pouze sledovaný soubor *Point2D.js*, proto ve výstupu příkladu 5.1 je vypsán změněný obsah tohoto souboru oproti verzi tohoto souboru uložené v indexu. Nyní opět změníme obsah souboru *Point2D.js* v pracovním adresáři projektu a provedeme opět příkaz `git diff`:

**js/Point2D.js**

```
    constructor(x, y) {
        this.x = x; this.y = y;
    }
    copy() {
        return new GitSnake.Point2D(this.x, this.y);
    }
}
```

### Příklad 5.2. Zobrazení rozdílu, který lze přidat do indexu

```
$ pwd
/Users/dan/Desktop/git_snake
$ git diff
diff --git a/js/Point2D.js b/js/Point2D.js
index 3b8b34b..20c9bf3 100644
--- a/js/Point2D.js
+++ b/js/Point2D.js
@@ -3,4 +3,7 @@ GitSnake.Point2D = class Point2D {
     constructor(x, y) {
         this.x = x; this.y = y;
     }
+    copy() {
+        return new GitSnake.Point2D(this.x, this.y);
+    }
 }
\ No newline at end of file
```

Výstupem příkazu 5.2 je nově přidaný obsah (metoda *copy*). Příkaz `git diff --staged` slouží pro zobrazení rozdílu mezi indexem a *HEAD*, tedy tento příkaz ukazuje změny, které budou v commitu, pokud se provede příkaz `git commit` oproti commitu předešlém. `Git diff` lze použít k porovnání libovolných dvou objektů, například k zobrazení *diffu* mezi commity, na které ukazují větve *master* a *develop*, lze použít příkaz `git diff master develop`. Program `diff` zobrazuje změny souboru *B* oproti souboru *A*, záleží tedy na pořadí parametrů, proto je dobré jako první parametr *A* použít objekt,

který je „starší“ než objekt *B*. Pro vytváření „pěkných“ *diffů* a přehledného kódu je doporučeno oddělovat jednotlivé části zdrojových kódů pomocí nového řádku, ovšem v této práci kvůli ušetření místa nebude toto doporučení striktně dodržováno. Nápomocný je také přepínač *--check* příkazu `git diff`, který slouží k zjištění přebytečných bílých znaků v *diffu*.

## 5.2. Základní obsah souborů praktického projektu

Upravíme následující soubory:

**js/SquareObject.js**

```
"use strict";
GitSnake.SquareObject = class SquareObject {
    constructor(pos) {
        this.position = pos; this.color = "red"; this.size = this.constructor.size;
        this.velocity = new GitSnake.Point2D(0, 0); this.collidable = true;
    } draw(ctx) {
        ctx.fillStyle = this.color; var tp = this.position; ctx.strokeStyle = "white";
        ctx.fillRect(tp.x * this.size, tp.y * this.size, this.size, this.size);
        ctx.strokeRect(tp.x * this.size, tp.y * this.size, this.size, this.size);
    }
}
GitSnake.SquareObject.size = 1;
```

**js/Snake.js**

```
"use strict";
GitSnake.Snake = class Snake extends GitSnake.SquareObject {
    constructor(pos) {
        super(pos); this.velocity.x = 1;
        this.tail = [new GitSnake.SquareObject(new GitSnake.Point2D(pos.x - 1, pos.y))];
        this.length = 1 + this.tail.length; this.color = "green";
    } draw(ctx) {
        this.tail.forEach(function (chunk) {
            chunk.draw(ctx);
        }); super.draw(ctx);
    }
}
```

**js/Game.js**

```
"use strict";
GitSnake.Game = class Game {
    constructor(canvas, tileSize, drawFPS = 60) {
        this.width = canvas.width; this.height = canvas.height;
        this.ctx = canvas.getContext("2d"); this.animationSpeed = 100;
        this.drawingSpeed = Math.ceil(1000 / drawFPS); this.drawingId;
        this.tileX = Math.floor(this.width / tileSize); this.animationId;
        this.tileY = Math.floor(this.height / tileSize);
        GitSnake.SquareObject.size = tileSize; this.gameObjects = [];
        this.isPlaying = false; this.isGameOver = false; this.score = 0;

        this.snake = new GitSnake.Snake(new GitSnake.Point2D(0, 0));
        this.addEntity(this.snake);
```

```
    } addEntity(entity) {
        this.gameObjects.push(entity);
    } draw() {
        this.gameObjects.forEach(function (entity) {
            entity.draw(this.ctx);
        }, this);
    }
}
```

**play.htm**

```
    <canvas width="600" height="300" id="canvas">Váš prohlížeč není podporován</canvas>
    <script>window.GitSnake = {};</script><script src="js/Point2D.js"></script>
    <script src="js/SquareObject.js"></script><script src="js/Snake.js"></script>
    <script src="js/Game.js"></script>
    <script>
        (function () {
            var game = new GitSnake.Game(document.getElementById("canvas"), 20, 30);
            game.draw();
        })();
    </script>
</body>
```

Po úpravě a uložení všech souborů otevřeme soubor *play.htm* v moderním webovém prohlížeči a zkontrolujeme, zdali se v okně webového prohlížeče zobrazil jeden zelený čtverec (hlava hada) v levé horní části černého rámečku (plátno). Po kontrole provedeme commit, kde využijeme přidání přepínače *-a* příkazu `git commit`, který přidá všechny změněné sledované soubory projektu do indexu a provede commit:

**Příklad 5.3. Přidání základních objektů hada**

```
$ pwd
/Users/dan/Desktop/git_snake
$ git commit -a -m "Přidání základních objektů hada"
[develop b588ff6] Přidání základních objektů hada
 5 files changed, 63 insertions(+)
```

## 5.3. Konvence pro psaní popisů změn

Dosud jsme vždycky v této práci pro specifikování popisu změny použili přepínač *-m* příkazu `git commit`. Přepínač *-m* se hodí pro krátké popisy změn, ovšem popis změn může (a měl by) být podrobný, tedy na více řádcích. Pro vytvoření popisu změny na více řádcích můžeme použít více přepínačů *-m* nebo úplně vynechat přepínač *-m*, přičemž Git otevře textový editor pro napsání popisu změny. Git otevře textový editor definovaný v nastavení Gitu (klíč *core.editor*) nebo editor definovaný v proměnných prostředí operačního systému (*$GIT_EDITOR*, *$EDITOR*, *$VISUAL*), případně editor *vim*[2], pokud v předešlých umístěních Git nic nenalezne. Pro rychlý kurz práce v editoru *vim* můžeme spustit tutoriál příkazem `vimtutor`.

[2] http://www.vim.org/

Popis změny v Gitu je nepovinný, i když doporučený, ovšem pokud napíšeme jeden znak nebo tisíc už Git nezajímá a je jen pouze na nás, jak kvalitně popíšeme změny v commitu.[3] Pro popis změn je důležité na první řádek napsat hlavní shrnutí změny (**předmět**). Tento první řádek se vypisuje v mnoha příkazech Gitu, například `git log --oneline`. Po napsání předmětu změny je nutné dvakrát odřádkovat a napsat podrobnější popis změny (viz příklad 5.4). [19]

**Příklad 5.4. Ukázka doporučení pro psaní popisů změn [19]**

```
Summarize changes in around 50 characters or less

More detailed explanatory text, if necessary. Wrap it to about 72
characters or so. In some contexts, the first line is treated as the
subject of the commit and the rest of the text as the body. The
blank line separating the summary from the body is critical (unless
you omit the body entirely); various tools like `log`, `shortlog`
and `rebase` can get confused if you run the two together.

Explain the problem that this commit is solving. Focus on why you
are making this change as opposed to how (the code explains that).
Are there side effects or other unintuitive consequenses of this
change? Here's the place to explain them.

Further paragraphs come after blank lines.

 - Bullet points are okay, too

 - Typically a hyphen or asterisk is used for the bullet, preceded
   by a single space, with blank lines in between, but conventions
   vary here

If you use an issue tracker, put references to them at the bottom,
like this:

Resolves: #123
See also: #456, #789
```

## 5.4. Průběh práce

Průběh práce (**workflow**) v Gitu je často diskutované téma. Git nevyžaduje používání určitého workflow a je tedy na uživateli, aby si vybral, jaké workflow mu nejvíce vyhovuje. Populární workflow je **GitFlow**[4], které používá pro každou vyvíjenou část projektu (**feature**) vlastní větev. Upravené *GitFlow* bude použito v této práci.

Projekt *GitSnake* kromě vykreslení hlavy hada zatím moc neumí, proto by bylo dobré přidat další funkce hry, například růst hada a základní animaci objektů. Pro růst hada vytvoříme novou větev s názvem *feature/snake-grow* a pro implementování animací objektů vytvoříme novou větev s názvem *feature/animate* a do této nově vytvořené větve se přepneme:

[3] Ukázky špatných popisů změn lze najít například na adrese http://www.commitlogsfromlastnight.com/.
[4] http://nvie.com/posts/a-successful-git-branching-model/

### Příklad 5.5. Vytvoření feature větví a přepnutí na větev feature/animate

```
$ pwd
/Users/dan/Desktop/git_snake
$ git branch feature/snake-grow
$ git checkout -b feature/animate ❶
Switched to a new branch 'feature/animate'
```

❶ Příkaz `git checkout -b <název větve>` je zkratka pro spuštění příkazu `git branch <název větve>` a `git checkout <název větve>` po sobě.

Po přepnutí na nově vytvořenou větev upravíme následující soubory projektu:

#### js/Point2D.js

```
    copy() {
        return new GitSnake.Point2D(this.x, this.y);
    } setTo(point) {
        this.x = point.x; this.y = point.y;
    } equals(point) {
        return (this.x === point.x && this.y === point.y);
    }
}
```

#### js/SquareObject.js

```
        ctx.strokeRect(tp.x * this.size, tp.y * this.size, this.size, this.size);
    } animate() {
        this.position.x += this.velocity.x; this.position.y += this.velocity.y;
    }
}
GitSnake.SquareObject.size = 1;
```

#### js/Snake.js

```
    } draw(ctx) {
        this.tail.forEach(function (chunk) {
            chunk.draw(ctx);
        }); super.draw(ctx);
    } animate() {
        var lastChunk = this.tail.pop(); this.tail.unshift(lastChunk);
        lastChunk.position.setTo(this.position); super.animate();
    }
}
```

#### js/Game.js

```
    } draw() {
        this.gameObjects.forEach(function (entity) {
            entity.draw(this.ctx);
        }, this);
    } animate() {
```

```
        this.gameObjects.forEach(function (entity) {
            entity.animate();
        }, this);
    } play() {
        if (!this.isPlaying && !this.isGameOver) {
            this.isPlaying = true; var game = this;
            this.drawingId = setInterval(function () {
                game.draw();
            }, this.drawingSpeed);
            this.animationId = setInterval(function () {
                game.animate();
            }, this.animationSpeed);
        }
    } pause() {
        this.isPlaying = false;
        clearInterval(this.drawingId); clearInterval(this.animationId);
    }
}
```

### play.htm

```
    (function () {
        var game = new GitSnake.Game(document.getElementById("canvas"), 20, 30);

        window.addEventListener("keydown", function (e) {
                switch (e.keyCode) {
                    case 27: case 32:
                        if (game.isPlaying) {
                            game.pause();
                        } else {
                            game.play();
                        }
                } e.preventDefault();
        });
    })();
</script>
```

Po úpravě a vyzkoušení, že se hra při stisku klávesy *mezerník* nebo *Esc* spustí, případně pozastaví a že se had pohybuje doprava v okně webového prohlížeče, provedeme commit. Commity by ovšem měly být **atomické**, tzn., že by měly přinášet jednotnou změnu. Aktuální změněné soubory ovšem přidaly kromě souvislé změny v logice hry i ovládání pomocí klávesnice, a proto provedeme commity dva. První commit se všemi změněnými soubory kromě souboru *play.htm* a druhý commit pouze se souborem *play.htm*. Díky tomu, že Git obsahuje index, si můžeme do indexu přidat pouze všechny soubory ze složky *js* a ty *commit*nout a v dalším commitu přidat do indexu pouze soubor *play.htm*:

#### Příklad 5.6. Implementace animací hry a možnosti pozastavit hru

```
$ pwd
/Users/dan/Desktop/git_snake
$ git add js/
$ git commit -m "Implementace animací objektů"
[feature/animate d9b576f] Implementace animací objektů
```

```
 4 files changed, 26 insertions(+)
$ git add play.htm
$ git commit -m "Play/Pause hry klávesou mezerník nebo ESC"
[feature/animate 38f1466] Play/Pause hry klávesou mezerník nebo ESC
 1 file changed, 11 insertions(+), 1 deletions(-)
```

## 5.5. Checkout

Již víme, že příkaz `git checkout <commit>` slouží pro změnu odkazu *HEAD*. Kromě změny odkazu *HEAD* ještě příkaz `git checkout` při spuštění zajistí, že pracovní adresář projektu bude vypadat tak, jak byl uložen v commitu (stromě) předaném v parametru. Pokud provedeme *checkout* větve *master*, tak se všechny sledované soubory v pracovním adresáři projektu smažou a nahradí se soubory, které jsou uloženy v commitu, na který *master* odkazuje.

**Příklad 5.7. Přepnutí na větev master**

```
$ pwd
/Users/dan/Desktop/git_snake
$ git checkout master
Switched to branch 'master'
$ cat play.htm
<!DOCTYPE html> ❶
```

❶ Obsah souboru *play.htm* v pracovním adresáři projektu je stejný jako v commitu, na který *master* odkazuje.

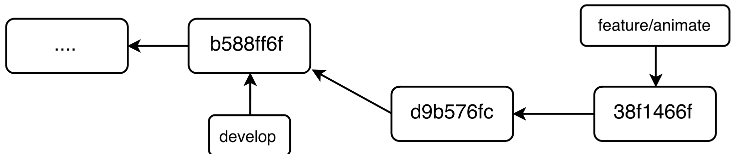
Mizení souborů z pracovního adresáře projektu spoustu lidí vystraší, ale musíme si uvědomit, že pracovní adresář projektu je pouze **pracovní kopie** a důležitá jsou data v repozitáři. Kvůli nahrazení souborů v pracovním adresáři projektu je dobré spouštět příkaz `git checkout` v čistém pracovním adresáři projektu, i když Git vypíše varování a zabrání přepnutí na specifikovaný commit, pokud by mělo dojít k přepsání změn v souborech. V případě špinavého pracovního adresáře projektu lze využít příkaz `git stash` nebo `git worktree`.

## 5.6. Merge

Pokud chceme určitou větev začlenit (sjednotit) do větve jiné, použijeme příkaz `git merge`. Sjednocení (**merge**) větví je v Gitu jednoduché, neboť větev v Gitu je pouze ukazatel na commit a commity vytváří orientovaný acyklický graf. Přesto v Gitu existuje několik možností (strategií) pro začlenění a také je možné vytvářet vlastní strategie.

### 5.6.1. Fast-forward strategie

Nejjednodušší sjednocovací strategie je *fast-forward*, která funguje nejlépe pro lineární vývoj a je to také doporučená strategie, pokud chceme začlenit naše změny do jiného repozitáře. *Fast-forward* strategii lze použít, pokud začleňujeme větev *F* do větve *D*, kde commit *D* je libovolný předchůdce commitu *F*. Pro ukázku *fast-forward* začlenění použijeme větev *develop* a z této větve „pokračující“ větev *feature/animate* (viz obrázek 5.1).




**Obrázek 5.1. Historie před začleněním [autor]**

### Příklad 5.8. Začlenění větve

```
$ pwd
/Users/dan/Desktop/git_snake
$ git checkout develop ❶
Switched to branch 'develop'
$ git merge feature/animate ❷
Updating b588ff6..38f1466
Fast-forward ❸
 js/Game.js         | 17 +++++++++++++++++
 js/Point2D.js      |  4 ++++
 js/Snake.js        |  3 +++
 js/SquareObject.js |  2 ++
 play.htm           | 12 +++++++++++-
 5 files changed, 37 insertions(+), 1 deletions(-) ❹
$ git branch -d feature/animate ❺
Deleted branch feature/animate (was 38f1466).
```

❶ Pro začlenění je nutné se nejprve přepnout na větev, kterou chceme aktualizovat.

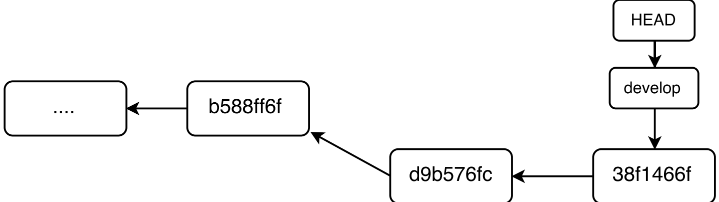
❷ Příkazu `git merge` jsme předali větev, kterou chceme začlenit do větve *develop*.

❸ Začlenění proběhlo pomocí *fast-forward* strategie.

❹ Souhrn, jaké změny byly provedeny ( `git diff --stat b588ff6 38f1466` ). Tento výpis se označuje *diffstat*[5].

❺ Po začlenění větve *feature/animate* do větve *develop* můžeme tuto větev smazat (*-d*), neboť již není potřeba (Git smaže **pouze** referenci této větve).

Po provedení začlenění větev *develop* odkazuje na stejný commit, na který odkazovala větev *feature/animate* před smazáním (viz obrázek 5.2).




**Obrázek 5.2. Historie po začlenění [autor]**

[5] http://invisible-island.net/diffstat/diffstat.html

## 5.6.2. Rekurzivní strategie

Rekurzivní strategie se používá pro sjednocení **dvou** větví, kde jedna větev není předchůdce druhé větve. V takovém případě nelze použít *fast-forward* strategii, ale je nutno vytvořit nový commit, který bude mít **dva** rodiče (commity, na které odkazují obě sjednocované větve). Výhodou rekurzivní strategie oproti *fast-forward* je, že ve vytvořeném commitu jsou uloženy odkazy na rodiče, tedy je zde uložena informace o tom, že tento commit vznikl spojením dvou větví a po smazání odkazů původních větví tyto větve můžeme v budoucnu kdykoli znovu vytvořit a pokračovat ve vývoji těchto větví. Dokonce můžeme tyto větve v budoucnu znovu sloučit. Nevýhodou rekurzivní strategie je, že již nemáme pouze lineární historii projektu.

Pro ukázku rekurzivní strategie použijeme větev *feature/snake-grow* vytvořenou v příkladu 5.5:

**Příklad 5.9. Přepnutí na větev feature/snake-grow**

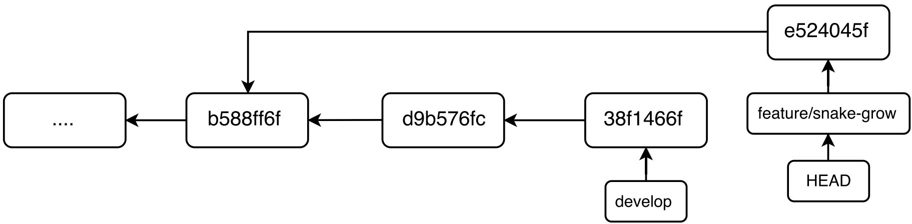
```
$ pwd
/Users/dan/Desktop/git_snake
$ git checkout feature/snake-grow
Switched to branch 'feature/snake-grow'
```

Upravíme následující soubor:

**js/Snake.js**

```
        this.tail = [new GitSnake.SquareObject(new GitSnake.Point2D(pos.x - 1, pos.y))];
        this.length = 1 + this.tail.length; this.color = "green";
    } grow(size) {
        this.length += size; var lastChunk = this.tail[this.tail.length - 1];
        for (var i = 1; i <= size; i++) {
            this.tail.push(new GitSnake.SquareObject(lastChunk.position.copy()));
        }
    } draw(ctx) {
        this.tail.forEach(function (chunk) {
}
```

Po uložení provedeme commit a následně se přepneme na větev *develop*, kde provedeme sjednocení s větví *feature/snake-grow*. Obrázek 5.3 ukazuje stav po provedení commitu.




**Obrázek 5.3. Historie před začleněním [autor]**

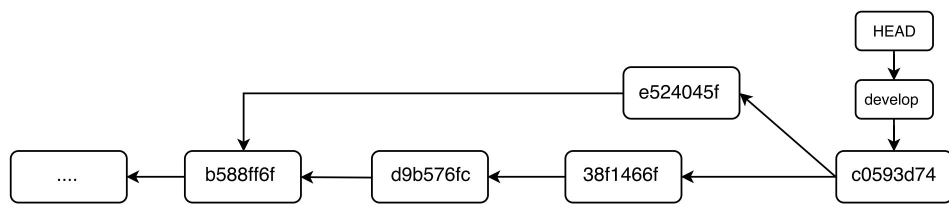
### Příklad 5.10. Rekurzivní začlenění

```
$ pwd
/Users/dan/Desktop/git_snake
$ git commit -m "Přidání grow(size) metody hada" js/Snake.js
[feature/snake-grow e524045] Přidání grow(size) metody hada
 1 file changed, 5 insertions(+)
$ git checkout develop
Switched to branch 'develop'
$ git merge feature/snake-grow ❶
Auto-merging js/Snake.js ❷
Merge made by the 'recursive' strategy. ❸
 js/Snake.js | 5 +++++
 1 file changed, 5 insertions(+)
$ git branch -d feature/snake-grow
Deleted branch feature/snake-grow (was e524045).
```

❶ Příkazu `git merge` jsme nepředali přepínač *-m* pro specifikování popisu změny, proto Git otevřel editor a vyplnit výchozí popis změny *„Merge branch 'feature/snake-grow' into develop“*. Tento výchozí popis změny můžeme ponechat tak, jak je, anebo upravit. Jakmile jsme spokojeni s popisem změny, uložíme soubor a zavřeme okno editoru.

❷ Soubor *js/Snake.js* byl změněn jak ve větvi *feature/snake-grow*, tak také ve větvi *develop* oproti verzi tohoto souboru ve společném předkovi těchto větví, proto došlo ke sloučení obsahu tohoto souboru z obou větví.[6] Vzhledem k tomu, že v obou větvích došlo k úpravě jiné části (řádků) souboru, nedošlo ke konfliktu.

❸ Začlenění bylo provedeno pomocí rekurzivní strategie.




**Obrázek 5.4. Historie po začlenění [autor]**

## 5.7. Grafické zobrazení historie

Historie projektu v Gitu tvoří graf, proto je běžné zobrazovat tento graf pro lepší orientaci v historii projektu. Pro zobrazení grafu aktuální větve lze využít příkaz:

### Příklad 5.11. Zobrazení historie projektu pomocí grafu

```
$ pwd
/Users/dan/Desktop/git_snake
```

[6] V případě sjednocování binárních souborů většinou není doporučeno použití sjednocení dvou verzí, ale použití celého souboru z jedné nebo druhé větve.

```
$ git log --oneline --decorate --graph ❶
*   c0593d7 (HEAD -> develop) Merge branch 'feature/snake-grow' into develop
|\
| * e524045 Přidání grow(size) metody hada
* | 38f1466 Play/Pause hry klávesou mezerník nebo ESC
* | d9b576f Implementace animací objektů
|/
* b588ff6 Přidání základních objektů hada
* dde6568 Přidání základního HTML kódu v play.htm
* 33012c8 (master) Přidání HTML5 hlavičky
* ccf57ac Vytvoření základních souborů hry
* d3173a9 (tag: v0.1.0) Přidání README projektu
$ gitk ❷
```

❶ Přepínač *--graph* vytvoří textový graf. Pro zobrazení historie všech referencí lze použít přepínač *--all*.

❷ Pro lepší zobrazení grafického vyjádření lze použít například příkaz `gitk`.

## 5.8. Vzdálený repozitář (remote)

Dosud jsme pracovali v našem osobním repozitáři jako uživatel *Daniel Maixner* (**Dan**). Nyní je čas sdílet projekt *GitSnake* s dalšími vývojáři, proto vytvoříme „**hlavní**“ repozitář, který bude sloužit k synchronizaci mezi vývojáři projektu *GitSnake*. Hlavní repozitář můžeme vytvořit v libovolném sdíleném umístění. Git pro přístup ke vzdáleným repozitářům podporuje protokoly SSH, HTTP(S), FILE a GIT. Pro naše potřeby vytvoříme hlavní repozitář na lokálním disku ve složce *hlavniRepozitar.git* uložené na ploše našeho operačního systému. V této složce můžeme nastavit čtecí a zapisovací práva pro všechny uživatele operačního systému. Hlavní repozitář nejčastěji nemá pracovní adresář projektu, protože v tomto repozitáři nikdo přímo nepracuje (všichni vývojáři mají vlastní repozitáře). Repozitář bez pracovního adresáře projektu se označuje **bare** a je zvykem pojmenovat složku s *bare* repozitářem ve tvaru *<název projektu>.git*. Pro vytvoření *bare* repozitáře použijeme například příkaz `git init --bare <cesta k repozitáři>`:

### Příklad 5.12. Vytvoření hlavního repozitáře

```
$ pwd # Konzole Dana ❶
/Users/dan/Desktop/git_snake
$ git init --bare /Users/dan/Desktop/hlavniRepozitar.git
Initialized empty Git repository in /Users/dan/Desktop/hlavniRepozitar.git
$ git remote add hlavni /Users/dan/Desktop/hlavniRepozitar.git ❷
```

❶ Pro rozlišení konzole uživatele projektu bude použita poznámka za příkazem `pwd`.

❷ Přidáme vzdálený repozitář jako referenci *hlavni* s adresou nově vytvořeného repozitáře.

## 5.9. Push

Nově vytvořený hlavní repozitář je nyní „prázdný“. Pro odeslání dat do hlavního repozitáře využijeme příkaz `git push <adresa repozitáře> <data k odeslání>`. Pokud jako <data> použijeme větev,

pak Git odešle všechny objekty, které souvisejí se specifikovanou větví. Hlavní repozitář je veřejně dostupný, proto je dobré odeslat pouze větve, které chceme sdílet. V našem případě chceme sdílet větve *master* a *develop*.

**Příklad 5.13. Odeslání větví do hlavního repozitáře**

```
$ pwd # Konzole Dana
/Users/dan/Desktop/git_snake
$ git push hlavni master develop ❶
Counting objects: 39, done.
Delta compression using up to 4 threads.
Compressing objects: 100% (33/33), done.
Writing objects: 100% (39/39), 4.83 KiB | 0 bytes/s, done.
Total 39 (delta 14), reused 14 (delta 3)
To /Users/dan/Desktop/hlavniRepozitar.git
 * [new branch]      master -> master
 * [new branch]      develop -> develop
```

❶ Namísto specifikování adresy cílového repozitáře použijeme referenci *hlavni*.

## 5.10. Vzdálené větve

Po první komunikaci s hlavním repozitářem se v našem repozitáři vytvořily nové vzdálené větve s názvem *hlavni/master* a *hlavni/develop*. Vzdálené větve jsou pouze pro čtení, tedy nelze se na ně standartní cestou přepnout. Git tyto větve synchronizuje s větvemi uloženými ve vzdáleném repozitáři *hlavni*. K synchronizaci těchto větví dochází pouze při komunikaci se vzdáleným repozitářem pomocí již známého příkazu `git push` nebo příkazu `git fetch`, který bude probrán v kapitole 5.14.

**Příklad 5.14. Vypsání vzdálených větví**

```
$ pwd # Konzole Dana
/Users/dan/Desktop/git_snake
$ git branch -r ❶
  hlavni/develop
  hlavni/master
```

❶ Přepínač *-r* příkazu `git branch` vypíše pouze vzdálené větve. Pro vypsání lokálních i vzdálených větví lze využít přepínač *-a*.

## 5.11. Sledování vzdálených větví

Když máme nastavený a aktualizovaný hlavní repozitář, chtěli bychom propojit naše lokální větve *master* a *develop* se vzdálenými větvemi *hlavni/master* a *hlavni/develop*. Díky tomuto propojení (**sledování**) nebudeme muset psát dlouhé příkazy jako `git push hlavni <větev>` ale pouze `git push` pro odeslání aktuální větve do nastaveného vzdáleného repozitáře. Pro nastavení sledování využijeme

například příkaz `git branch -u <vzdálená větev> <lokální větev>`, který upraví lokální konfigurační soubor Gitu (*.git/config*):

**Příklad 5.15. Sledování vzdálených větví**

```
$ pwd # Konzole Dana
/Users/dan/Desktop/git_snake
$ git branch -u hlavni/master master
Branch master set up to track remote branch master from hlavni.
$ git branch -u hlavni/develop develop
Branch develop set up to track remote branch develop from hlavni.
```

Pro zobrazení detailních informací o vzdálených repozitářích Git nabízí příkaz `git remote show <repozitář>`.

## 5.12. Zkratky referencí v Gitu

V Gitu je mnoho referencí (viz kapitola 4.4). Již víme, že příkaz `git log master` vypíše historii lokální větve *master*. Ve skutečnosti ovšem dojde k výpisu historie větve *refs/heads/master*. Git prohledává následující složky v tomto pořadí pro nalezení reference s daným názvem:

1. .git/<název reference>
2. .git/refs/<název reference>
3. .git/refs/tags/<název reference>
4. .git/refs/heads/<název reference>
5. .git/refs/remotes/<název reference>
6. .git/refs/remotes/<název reference>/HEAD

Pokud bychom tedy měli v repozitáři štítek s názvem *master* a zároveň lokální větev *master*, Git by vypsal varování a my bychom museli použít specifičtější cestu pro rozlišení, se kterou referencí chceme pracovat.

**Příklad 5.16. Výpis štítků, hlav a vzdálených referencí**

```
$ pwd # Konzole Dana
/Users/dan/Desktop/git_snake
$ git show-ref ❶
c0593d7407187d3a0eaa39164d4887ffb99a2a19 refs/heads/develop
33012c824e321a467c298891013f280ce502db02 refs/heads/master
c0593d7407187d3a0eaa39164d4887ffb99a2a19 refs/remotes/hlavni/develop
33012c824e321a467c298891013f280ce502db02 refs/remotes/hlavni/master
bdc6059afa3b50994df52b7ed94faea24519948b refs/tags/v0.1.0
```

❶ Výpis aktuálních štítků, hlav a vzdálených referencí spolu s otisky, na které odkazují.

Aktuálně máme lokální větev *master* (*refs/heads/master*) a vzdálenou větev *„master“* (*refs/remotes/hlavni/master*). Vzdálená větev *„master“* je ale uložena ve složce *refs/remotes/hlavni/* a Git prohledává vzdálené větve ve složce *refs/remotes/<název reference>*, proto musíme pro specifikování vzdálené větve použít *hlavni/master*, aby Git nalezl správnou cestu *refs/remotes/hlavni/master*. Pro zobrazení celých názvů referencí ve výpisu historie lze použít přepínač *--decorate=full* příkazu `git log`.

## 5.13. Clone

Do projektu *GitSnake* se rozhodl přidat další vývojář, kterého nazveme **Bob**. Bob si chce naklonovat repozitář *GitSnake* a provést úpravy.

Pro práci v roli Boba můžeme vytvořit nový uživatelský účet v našem operačním systému anebo zůstat přihlášeni pod současným uživatelským účtem a vydávat se za Boba. Vytvoření repozitáře Boba provedeme pomocí příkazu `git clone <zdrojový repozitář> <cílový repozitář>`:

**Příklad 5.17. Vytvoření repozitáře Boba**

```
$ pwd # Konzole Boba
/Users/bob
$ git clone /Users/dan/Desktop/hlavniRepozitar.git /Users/bob/Desktop/bobGitSnake
Cloning into '/Users/bob/Desktop/bobGitSnake'...
done.
$ cd /Users/bob/Desktop/bobGitSnake
$ git config user.name Bob ❶
$ git config user.email bob@uhk.nm
$ git checkout -b develop origin/develop ❷
Branch develop set up to track remote branch develop from origin.
Switched to a new branch 'develop'
$ git remote ❸
origin
```

❶ Upravíme lokální nastavení repozitáře pro rozlišení uživatele.

❷ Přepneme se na novou větev *develop* vytvořenou z reference *origin/develop*. Git automaticky nastavil pro tuto větev sledování vzdálené větve *origin/develop*.

❸ Pro vypsání referencí na vzdálené repozitáře použijeme příkaz `git remote`.

Příkaz `git clone` naklonoval[7] všechny objekty a reference z klonovaného repozitáře a také provedl *checkout* větve, na kterou odkazuje *HEAD* v klonovaném repozitáři (*master*) do složky specifikované Bobem. Příkaz `git clone` také vytvořil vzdálený repozitář, který se ve výchozím nastavení jmenuje **origin**. Bob má nyní staženou celou historii projektu *GitSnake* a také odkaz na originální repozitář, ze kterého byl tento klon vytvořen (*origin*).

[7] Díky tomu, že klonovaný repozitář je na stejném lokálním disku jako cílový repozitář, Git použil optimalizaci souborového systému. Pro skutečné klonování bychom museli použít *FILE* protokol nebo přepínač *--no-local* příkazu `git clone`.

Bob se rozhodl upravit estetickou stránku projektu *GitSnake*, a proto změnil soubor *play.htm* a provedl commit.

**play.htm**

```
<head><meta charset="UTF-8"><title>GitSnake</title>
<style>#canvas { border: 2px solid black; display: block; margin: 0 auto; }</style>
</head>
```

**Příklad 5.18. Horizontální vycentrování plátna na stránce**

```
$ pwd # Konzole Boba
/Users/bob/Desktop/bobGitSnake
$ git add play.htm
$ git commit -m "Horizontální vycentrování plátna na stránce"
[develop c100627] Horizontální vycentrování plátna na stránce
 1 file changed, 1 insertion(+), 1 deletion(-)
$ git remote rename origin hlavni ❶
$ git push --quiet ❷
```

❶ Pro přehlednost přejmenujeme referenci *origin* na *hlavni*.

❷ Pro ušetření místa bude pro určité příkazy použit přepínač *--quiet*, který slouží k zamezení výpisu daného příkazu. Pro čtenáře je ovšem doporučeno psát příkazy bez přepínače *--quiet*. Díky nastavení sledovaných větví lze použít samotný příkaz `git push`, který odešle nové objekty **aktuální** větve do vzdáleného repozitáře a provede *fast-forward* začlenění.

## 5.14. Fetch

Pro stažení nových změn ze vzdáleného repozitáře do našeho repozitáře použijeme příkaz `git fetch`. `Git fetch` aktualizuje pouze vzdálené větve. Vzhledem k tomu, že repozitář Dana má propojené lokální větve *master* a *develop* se vzdálenými větvemi z hlavního repozitáře, stačí zadat pouze samotný příkaz `git fetch`, pokud jsme ve větvi *master* nebo *develop* pro stažení aktualizací z hlavního repozitáře.

**Příklad 5.19. Stažení aktuálních dat ze vzdáleného repozitáře**

```
$ pwd # Konzole Dana
/Users/dan/Desktop/git_snake
$ git status
On branch develop
Your branch is up-to-date with 'hlavni/develop'. ❶
nothing to commit, working directory clean
$ git fetch --quiet
$ git status
On branch develop
Your branch is behind 'hlavni/develop' by 1 commit, and can be fast-forwarded. ❷
  (use "git pull" to update your local branch)
```

```
nothing to commit, working directory clean
$ git diff --unified=1 HEAD hlavni/develop ❸
diff --git a/play.htm b/play.htm
index 849fff2..d33048d 100644
--- a/play.htm
+++ b/play.htm
@@ -2,3 +2,3 @@
 <head><meta charset="UTF-8"><title>GitSnake</title>
-<style>#canvas { border: 2px solid black; }</style>
+<style>#canvas { border: 2px solid black; display: block; margin: 0 auto; }</style>
 </head>
$ git diff --unified=0 --word-diff HEAD hlavni/develop ❹
diff --git a/play.htm b/play.htm
index 849fff2..d33048d 100644
--- a/play.htm
+++ b/play.htm
@@ -3 +3 @@
<style>#canvas { border: 2px solid black; {+display: block; margin: 0 auto;+} }</style>
$ git merge --quiet hlavni/develop ❺
```

❶ Díky sledování vzdálené větve *hlavni/develop* příkaz `git status` vypisuje, že podle poslední aktualizace (`git fetch`) lokální větev *develop* odkazuje na stejný commit jako vzdálená větev *hlavni/develop*.

❷ Po provedení příkazu `git fetch` už `git status` vypisuje, že větev *develop* je o jeden commit pozadu oproti *hlavni/develop* a také že lze provést začlenění metodou *fast-forward*.

❸ Použití příkazu `git diff` pro zobrazení rozdílu mezi *HEAD* a *hlavni/develop*. Přepínač *--unified=<N>* specifikuje kolik řádků kontextu se bude v *diffu* zobrazovat.

❹ Zobrazení *diffu* jednotlivých slov (*--word-diff*) namísto řádků.

❺ Změny od Boba se nám líbí, a proto provedeme začlenění změn z větve *hlavni/develop* do aktuální větve (*develop*).

## 5.15. Spolupráce v týmu

Bob se rozhodl intenzivně spolupracovat na projektu *GitSnake*, proto jsme se sešli na schůzi a domluvili jsme se, že Bob provede implementaci funkce v případě ukončení hry a my (Dan) vytvoříme ovládání hada pomocí ovládacích šipek klávesnice. Začneme našimi změnami, tedy vytvořením nové větve *feature/entity-control*:

**Příklad 5.20. Vytvoření větve pro ovládání entity**

```
$ pwd # Konzole Dana
/Users/dan/Desktop/git_snake
$ git branch
* develop
  master
$ git checkout -b feature/entity-control
Switched to a new branch 'feature/entity-control'
```

Upravíme následující soubory:

**js/SquareObject.js**

```
    } animate() {
        this.position.x += this.velocity.x; this.position.y += this.velocity.y;
    } turn(velX, velY) {
        this.velocity.x = velX; this.velocity.y = velY;
    }
}
```

**play.htm**

```
    var game = new GitSnake.Game(document.getElementById("canvas"), 20, 30);
    var snake = game.snake;

    window.addEventListener("keydown", function (e) {
        switch (e.keyCode) {
            case 38: return game.isPlaying && snake.turn( 0, -1);
            case 40: return game.isPlaying && snake.turn( 0,  1);
            case 37: return game.isPlaying && snake.turn(-1,  0);
            case 39: return game.isPlaying && snake.turn( 1,  0);
            case 27: case 32:
```

Po uložení a zkontrolování ve webovém prohlížeči, že lze ovládat hada pomocí ovládacích šipek klávesnice, provedeme opět rozdělení na dva commity pro každý soubor zvlášť:

### Příklad 5.21. Ovládání hada šipkami

```
$ pwd # Konzole Dana
/Users/dan/Desktop/git_snake
$ git add -u ❶
$ git reset HEAD play.htm ❷
Unstaged changes after reset:
M       play.htm ❸
$ git commit -m "Přidání turn metody ve SquareObject.js" ❹
[feature/entity-control 07e8f32] Přidání turn metody ve SquareObject.js
 1 file changed, 2 insertions(+)
$ git commit -a -m "Ovládání hada pomocí šipek klávesnice"
[feature/entity-control ba70595] Ovládání hada pomocí šipek klávesnice
 1 file changed, 5 insertions(+)
$ git checkout --quiet develop
$ git merge --quiet feature/entity-control
$ git branch --quiet -d feature/entity-control
$ git push --quiet
```

❶ Příkazu `git add` jsme předali přepínač *-u*, který přidá všechny sledované změněné soubory do indexu.

❷ Příkaz `git reset HEAD play.htm` zkopíruje soubor *play.htm* z *HEAD* do indexu, kde přepíše dříve přidanou změněnou verzi souboru *play.htm*. Příkaz *reset* v tomto případě

nezmění nic v pracovním adresáři projektu, kde máme stále uloženou změnu v souboru *play.htm*.

❸ Git vypisuje informaci, že po nahrazení verze souboru *play.htm* v indexu verzí z *HEAD* je stále soubor *play.htm* v pracovním adresáři projektu změněný (**M**) oproti verzi v indexu. [8]

❹ V commitu bude změněný pouze soubor *SquareObject.js*, pro kontrolu následujícího commitu lze využít příkaz `git diff --staged`. Tento příkaz je dobré spouštět před každým commitem.

Pro pokročilou práci s indexem lze využít příkaz `git add --patch` pro přidávání části souborů (**hunk**), případně obecnější verzi `git add --interactive`. Pro odebírání částí souborů z indexu Git nabízí příkaz `git reset --patch`.

Bob pracoval paralelně s námi na implementaci ukončení hry. Bob vytvoří novou větev *feature/game-over* z commitu, na který odkazuje větev *develop*, která vyřeší ukončení hry v případě nárazu hlavy hada do okraje plátna (zdi) nebo v případě dokončení hry.

### Příklad 5.22. Vytvoření větve pro ukončení hry

```
$ pwd # Konzole Boba
/Users/bob/Desktop/bobGitSnake
$ git branch
*  develop
   master
$ git checkout -b feature/game-over
Switched to a new branch 'feature/game-over'
```

Bob upraví soubor *Game.js* následovně:

#### js/Game.js

```
    } animate() {
        this.gameObjects.forEach(function (entity) {
            entity.animate();
        }, this);
        this.checkGameOverConditons();
    } checkGameOverConditons() {
        if (this.snake.length >= this.tileX * this.tileY - 1)
            return this.gameOver("Vyhráli jste!");
        var spos = this.snake.position;
        if (spos.x < 0 || spos.x > this.tileX - 1)
            return this.gameOver();
    } gameOver(text = "Prohráli jste!") {
        this.pause(); this.isGameOver = true;
        alert( text + "\nVaše skóre: " + this.score );
        location.reload();
    } play() {
        if (!this.isPlaying && !this.isGameOver) {
```

[8] S obdobným formátem výstupu se lze setkat například při použití příkazu `git status -s`.

Po uložení a zkontrolování toho, že had po přejetí plátna narazí do zdi a hra se ukončí, Bob provede commit.

**Příklad 5.23. Přidání ověření pro konec hry**

```
$ pwd # Konzole Boba
/Users/bob/Desktop/bobGitSnake
$ git add js/Game.js
$ git commit -m "Přidání ověření pro konec hry"
[feature/game-over 1a02b1b] Přidání ověření pro konec hry
 1 file changed, 11 insertions(+)
```

Bob ovšem nemůže s hadem hýbat a tedy ověřit, zdali nová funkce funguje podle očekávání, protože na této části pracuje Dan. Naštěstí Dan prý již nahrál své změny do větve *develop* v hlavním repozitáři a Bob si je tedy může stáhnout a začlenit.

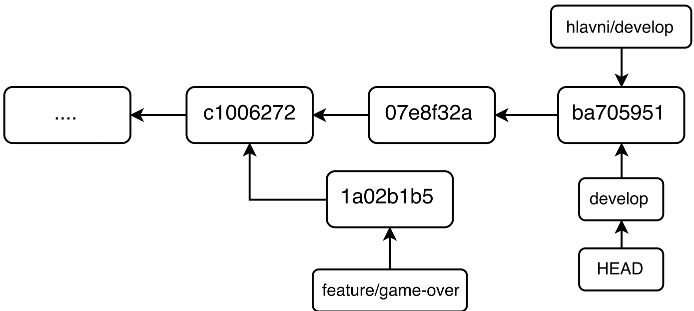
**Příklad 5.24. Začlenění změn od Dana**

```
$ pwd # Konzole Boba
/Users/bob/Desktop/bobGitSnake
$ git checkout develop
Switched to branch 'develop'
Your branch is up-to-date with 'hlavni/develop'.
$ git fetch --quiet
$ git log --oneline HEAD..hlavni/develop ❶
ba70595 Ovládání hada pomocí šipek klávesnice
07e8f32 Přidání turn metody ve SquareObject.js
$ git merge --quiet hlavni/develop
```

❶ Zápis *<A>***..***<B>* slouží pro získání objektů, které jsou v *B* navíc oproti *A*.

## 5.16. Rebase

Změny od Dana jsou nyní v repozitáři Boba ve větvi *develop* a *hlavni/develop* (viz obrázek 5.5). Začlenění změn z větve *develop* do větve *feature/game-over* (nebo obráceně) můžeme provést pomocí klasického začlenění (`git merge`), ovšem v tomto případě by došlo k *rekurzivní* strategii začlenění a historie projektu by se zbytečně zkomplikovala. Proto použijeme příkaz `git rebase`, který „vezme“ změny v commitech aktuální větve oproti společnému předku aktuální větve a větve nad kterou se provádí *rebase* a tyto změny aplikuje jako nové commity nad danou větví. `Git rebase` také nastaví referenci aktuální větve, aby odkazovala na nově vytvořené commity (viz obrázek 5.6). Příkaz `git rebase` vytváří nové commity, proto tyto commity mají jiné otisky než commity původní. Pro interaktivní použití *rebase* lze použít příkaz `git rebase --interactive`. Tento příkaz se hodí pro „zkrášlení“ historie větve, která nebyla odeslána do vzdáleného repozitáře.



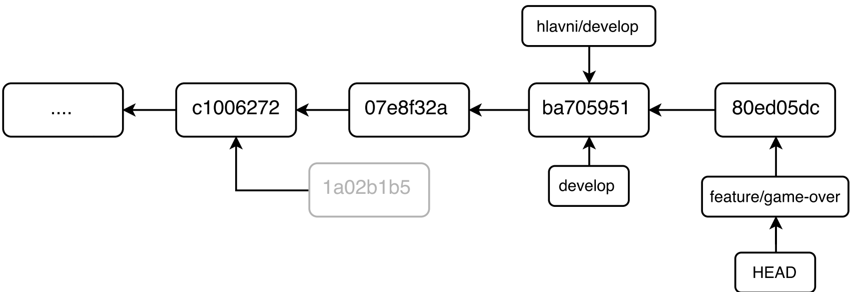

**Obrázek 5.5. Historie před rebase [autor]**

### Příklad 5.25. Rebase změn od Dana

```
$ pwd # Konzole Boba
/Users/bob/Desktop/bobGitSnake
$ git checkout - ❶
Switched to branch 'feature/game-over'
$ git rebase develop ❷
First, rewinding head to replay your work on top of it...
Applying: Přidání ověření pro konec hry
```

❶ Přepneme se na větev, kterou chceme aktualizovat. Pro přepnutí na předchozí aktuální větev lze předat parametr **-** příkazu `git checkout`.

❷ Příkazu `git rebase` jsme předali odkaz, nad kterým chceme vytvořit nové commity. Zde bychom mohli použít i odkaz *hlavni/develop*, neboť tento odkaz odkazuje na stejný commit jako *develop*.




**Obrázek 5.6. Historie po rebase [autor]**

Nyní má Bob ve větvi *feature/game-over* jak změněné soubory od Dana s ovládáním hada, tak také změnu pro ukončení hry. Díky použití *rebase* namísto *merge* jsou commity lineární. Po vyzkoušení ve webovém prohlížeči Bob zjistil, že hra se ukončí pouze při nárazu do levé nebo pravé stěny. Tuto chybu Bob opraví upravením souboru *js/Game.js*:

**js/Game.js**

```
        var spos = this.snake.position;
        if (spos.x < 0 || spos.x > this.tileX - 1 || spos.y < 0 || spos.y > this.tileY - 1)
            return this.gameOver();
```

```
} gameOver(text = "Prohráli jste!") {
```

Bob je v týmu projektu *GitSnake* nový, a proto nechce vytvářet další commit, který by navždy uložil v historii jeho chybu, ale chce „upravit“[9] commit předešlý, který by „sloučil“ s aktuální změnou v pracovním adresáři. Naštěstí Bob ještě neprovedl odeslání změn do hlavního repozitáře a všechny změny jsou zatím pouze v Bobově lokálním repozitáři.

## 5.17. Amend

Pro změnu posledního provedeného commitu Git poskytuje například přepínač *--amend* příkazu `git commit`. Přidáme do indexu soubor *js/Game.js* a provedeme příkaz `git commit --amend`, který sloučí změny v posledním commitu (*HEAD*) s aktuálním indexem a vytvoří commit nový, na který přesune referenci *feature/game-over*.

### Příklad 5.26. Změna posledního commitu

```
$ pwd # Konzole Boba
/Users/bob/Desktop/bobGitSnake
$ git add js/Game.js
$ git commit --amend -m "Restart hry v případě výhry či prohry"
[feature/game-over d448862] Restart hry v případě výhry či prohry
Date: Fri Apr 22 14:35:46 2016 +0100
 1 file changed, 11 insertions(+)
$ git checkout --quiet develop
$ git fetch ❶
$ git merge --quiet feature/game-over
$ git branch --quiet -d feature/game-over
$ git push --quiet
```

❶ Spuštění příkazu `git fetch` pro kontrolu zdali nedošlo ke změnám v hlavním repozitáři od poslední aktualizace. Příkaz `git fetch` je dobré spouštět před každým vytvořením nové větve ze sledované větve a také před každým sjednocením či *rebase*.

## 5.18. Tag

Nyní je ve větvi *develop* v Bobově repozitáři, a také v hlavním repozitáři implementované jak ovládání pomocí ovládacích šipek klávesnice, tak i ukončení hry při nárazu hlavy hada do zdi nebo v případě dokončení hry. Projekt *GitSnake* pomalu začíná připomínat skutečnou hru had. V tuto chvíli se můžeme rozhodnout, že hra prošla důležitým milníkem a my tento milník chceme uchovat v historii projektu. Pro ukázku Bob začlení větev *develop* do větve *master* a vytvoří štítek *v0.2.0*.

### Příklad 5.27. Začlenění větve develop do větve master

```
$ pwd # Konzole Boba
/Users/bob/Desktop/bobGitSnake
```

[9] Již víme, že nelze upravit obsah objektu, aniž by nedošlo ke změně otisku, proto Git nemění objekty, ale vytváří objekty nové.

```
$ git checkout --quiet master
$ git merge --quiet develop
$ git tag -a -m "Had ovládán šipkami a ukončení hry" v0.2.0
$ git push --quiet --tags ❶
$ git push --quiet
```

❶ Pro odeslání pouze všech štítků musíme specifikovat přepínač *--tags*.

Po vydání projektu *GitSnake* ve verzi *0.2.0* testerům se zjistil velký nedostatek projektu a sice, že had neustále „roste", i když by měl mít zatím pouze hlavu a jednu část ocasu. Tento nedostatek je nutno rychle opravit a vydat novou verzi pro testování. Bob upraví soubor *Game.js* ve větvi *master* následovně:

**js/Game.js**

```
} draw() {
    this.ctx.clearRect(0, 0, this.width, this.height);
    this.gameObjects.forEach(function (entity) {
        entity.draw(this.ctx);
```

Po vizuálním ověření ve webovém prohlížeči a spuštění automatizovaných testů Bob provede commit a označení nové verze:

**Příklad 5.28. Hotfix přemazávání plátna**

```
$ pwd # Konzole Boba
/Users/bob/Desktop/bobGitSnake
$ git add js/Game.js
$ git commit --quiet -m "Oprava mazání plátna v Game.js:draw()"
$ git tag -a -m "Oprava překreslení plátna" v0.2.1
$ git push --quiet --tags
$ git push --quiet
$ git checkout --quiet develop
$ git merge --quiet master
$ git push --quiet
```

## 5.19. Odstranění souborů (rm)

Pro odstranění sledovaného souboru stačí tento soubor smazat v indexu a provést commit, který daný soubor smaže ve snímku projektu. Takto smazaný soubor je však stále dostupný v historii projektu.[10] Pro ukázku Bob odstraní soubor *play.htm*:

**Příklad 5.29. Odstranění souboru**

```
$ pwd # Konzole Boba
```

[10] Pro odstranění souboru z celé historie projektu lze použít příkaz `git filter-branch`, ovšem tento příkaz patří mezi pokročilejší příkazy.

```
/Users/bob/Desktop/bobGitSnake
$ git rm play.htm
rm 'play.htm' ❶
$ git commit -m "Odstranění souboru play.htm"
[develop e838400] Odstranění souboru play.htm
 1 file changed, 32 deletions(-)
 delete mode 100644 play.htm
$ git push --quiet
```

❶ Příkaz `git rm <název>` kromě odstranění daného souboru z indexu také spustí unixový příkaz `rm <název>`, který odstraní daný soubor z pracovního adresáře projektu.

## 5.20. Revert

V kapitole 5.19 Bob provedl odstranění souboru *play.htm*. Toto odstranění, konkrétně commit z příkladu 5.29, byl omyl, a proto by Bob tento commit chtěl „vrátit“. Pokud by byl commit pouze v Bobově repozitáři, nebyl by problém commit „upravit“, ale jelikož byl tento commit odeslán do hlavního repozitáře, je zde možnost, že si již tento commit někdo stáhl, a proto Bob nemůže tento commit v hlavním repozitáři nahradit jiným, neboť by to negativně ovlivnilo všechny uživatele, kteří si commit již stáhli. Pro vrácení commitu můžeme použít příkaz `git revert <commit>`, který commit v parametru zneguje. Příkaz `git revert` ovšem nenahradí negovaný commit, pouze vytvoří nový commit, který zneguje změny commitu v parametru, tedy přidání nahradí odstraněním a obráceně.

**Příklad 5.30. Vrácení změn commitu**

```
$ pwd # Konzole Boba
/Users/bob/Desktop/bobGitSnake
$ git revert --no-edit HEAD ❶
[develop c8ff334] Revert "Odstranění souboru play.htm"
 1 file changed, 32 insertions(+) ❷
 create mode 100644 play.htm
$ git push --quiet
```

❶ Přepínač *--no-edit* slouží k ponechání výchozího popisu změny, který vytvořil Git automaticky.

❷ Počet vložených řádků je stejný jako počet odstraněných řádků v příkladu 5.29.

## 5.21. Show

Příkaz `git show` je základní příkaz pro zobrazování informací o objektech v Gitu. Pokud zadáme pouze příkaz `git show`, který automaticky dosadí *HEAD* jako referenci, dostaneme informace o posledním commitu, včetně vypsání *diffu*. Příkaz `git show <reference>:<cesta k souboru>` zase zobrazí obsah souboru v dané cestě, tak jak byl uložen v dané referenci. Pro zobrazení komentovaného štítku příkaz `git show <štítek>` zobrazí jak informace o štítku, tak i o objektu, na který štítek odkazuje, proto musíme použít symbolickou referenci. Například pro zobrazení commitu, na který odkazuje štítek *v0.2.0*, použijeme příkaz `git show "v0.2.0^{commit}"`. Obdobně můžeme zobrazit kořenový strom větve *master* `git show "master^{tree}"`.

## 5.22. Alias

Git umožňuje vytvoření aliasů pro příkazy. Například v příkladu 5.11 jsme použili příkaz `git log --oneline --decorate --graph`, který poskytuje přehled o historii projektu. Abychom nemuseli tento příkaz neustále psát, můžeme pro něj vytvořit alias v konfiguračním souboru Gitu.

**Příklad 5.31. Vytvoření globálního aliasu**

```
$ pwd # Konzole Boba
/Users/bob/Desktop/bobGitSnake
$ git config --global alias.l "log --oneline --decorate --graph"
$ git l HEAD -2 ❶
* c8ff334 (HEAD -> develop, hlavni/develop) Revert "Odstranění souboru play.htm"
* e838400 Odstranění souboru play.htm
```

❶ Aliasům lze přidávat i další parametry a přepínače. Zde jsme předali parametr *HEAD* pro referenci odkud má Git začít logovat, a přepínač *-2* pro specifikování vypsání pouze dvou posledních commitů historie.

Aliasy v Gitu jsou výborné pro ušetření psaní dlouhých příkazů, ovšem stále je nutné psát příkaz `git <alias>`. Pro pokročilejší definování aliasů lze využít například aliasy konzolového interpretu.

## 5.23. Pull

Příkaz `git pull` slouží pro provedení aktualizace a následně případné začlenění nových commitů do aktuální větve podle parametrů definovaných v nastavení repozitáře. `Git pull` je tedy zkratka pro spuštění příkazu `git fetch` následovaném příkazem `git merge`. Pro ukázku se vrátíme do role Dana a provedeme příkaz `git pull`:

**Příklad 5.32. Git pull**

```
$ pwd # Konzole Dana
/Users/dan/Desktop/git_snake
$ git log --oneline --decorate -1
ba70595 (HEAD -> develop) Ovládání hada pomocí šipek klávesnice
$ git pull --quiet --ff-only ❶
$ git log --oneline --decorate -5
c8ff334 (HEAD -> develop, hlavni/develop) Revert "Odstranění souboru play.htm"
e838400 Odstranění souboru play.htm
a0a0d97 (tag: v0.2.1, hlavni/master) Oprava mazání plátna v Game.js:draw()
d448862 (tag: v0.2.0) Restart hry v případě výhry či prohry
ba70595 Ovládání hada pomocí šipek klávesnice
```

❶ Přepínač *--ff-only* znamená, že chceme provést začlenění pouze pokud lze provést *fast-forward* začlenění.

Pro použití *rebase* namísto *merge* lze použít přepínač *--rebase* příkazu `git pull`.

## 5.24. Describe

Příkaz `git describe <commit>` slouží pro výpis Gitem generované verze v závislosti na posledním dosažitelném štítku z commitu definovaném v parametru.

**Příklad 5.33. Generování verze z git describe**

```
$ pwd # Konzole Dana
/Users/dan/Desktop/git_snake
$ git describe
v0.2.1-2-gc8ff334 ❶
$ git describe v0.2.0~3
v0.1.0-9-gc100627
```

❶ Výstupem příkazu je formát: *<první dosažitelný štítek z commitu v parametru>-<počet commitů od tohoto štítku do commitu v parametru>-g<část otisku commitu v parametru>*

## 5.25. Ignorování souborů

Při vytváření projektu se lze setkat s mnoha soubory, které nechceme verzovat. Mezi soubory, které bychom neměli verzovat, patří soubory nesouvisející s projektem (dočasné soubory operačního systému či editoru), binární soubory (které lze vygenerovat ze zdrojových souborů) apod. Aby se tyto soubory stále neobjevovaly jako nesledované soubory, lze tyto soubory přidat mezi ignorované. Git pro práci s ignorovanými soubory nabízí tři možnosti jak lze soubory ignorovat:

- .gitignore – seznam ignorovaných souborů pro všechny uživatele projektu. Tento soubor je součástí historie projektu.[11] Soubor *.gitignore* lze použít i v podsložkách projektu.
- .git/info/exclude – seznam ignorovaných souborů pouze pro projekt, ve kterém je tento soubor uložen.
- config – seznam ignorovaných souborů definovaných v souboru specifikovaném v nastavení Gitu (klíč *core.excludesFile*). Lze použít přepínače příkazu `git config` pro specifikování úrovně použití.

V projektu *GitSnake* nemáme žádné soubory, které bychom měli v tuto chvíli ignorovat pro potřeby projektu, ovšem jako vývojáři projektu si můžeme definovat globálně, které soubory chceme ignorovat v závislosti na tom, jaké nástroje používáme. Například většina textových editorů ukládá zálohy (kopie) aktuálně editovaných souborů do stejné složky, ve které je uložený originální soubor.

Pro ukázku použití souboru *.gitignore* vytvoříme složku *images* v pracovním adresáři projektu, která může představovat složku pro ukládání obrázků projektu, které nechceme verzovat:

**Příklad 5.34. Ignorování složky images**

```
$ pwd # Konzole Dana
/Users/dan/Desktop/git_snake
$ mkdir images
```

[11] Výjimka nastane, pokud do obsahu souboru *.gitignore* přidáme samotný název souboru *.gitignore*.

```
$ git status ❶
On branch develop
Your branch is up-to-date with 'hlavni/develop'.
nothing to commit, working directory clean
$ touch images/.gitkeep ❷
$ git status -s ❸
?? images/
$ git add images/.gitkeep
$ git commit --quiet -m "Vytvoření složky pro ukládání obrázků"
$ echo -n "images" > .gitignore ❹
$ git add .gitignore
$ git commit --quiet -m "Přidání .gitignore pro složku images" ❺
$ touch images/ignorovany_soubor.txt
$ git status -s ❻
```

❶ Git sleduje pouze soubory, tedy prázdný adresář *images* se ve výstupu příkazu `git status` nevypisuje ani jako nesledovaný soubor.

❷ Pro zaznamenání složky *images* musíme vytvořit alespoň jeden soubor ve složkce *images*. Je zvykem tento soubor pojmenovat *.gitkeep*.

❸ Přepínač *-s* příkazu `git status` slouží k výpisu statusu ve stručném formátu.

❹ Přidáme název složky, kterou chceme ignorovat do souboru *.gitignore*.

❺ Soubor *.gitignore* přidáme do historie projektu, aby se vytvoření a ignorování obsahu složky *images* projevilo i u ostatních vývojářů projektu, kteří si tento commit stáhnou.

❻ Nově vytvořený soubor *images/ignorovany_soubor.txt* se neobjevuje jako nesledovaný soubor.

## 5.26. Implementace funkcí praktického projektu

Pro ukázku dalších funkcí Gitu vytvoříme pár commitů v roli Dana. Začneme vytvořením větve *feature/collidable-entity* a následně větve *feature/game-food*.

**Příklad 5.35. Vytvoření větve pro implementaci kolizí objektů**

```
$ pwd # Konzole Dana
/Users/dan/Desktop/git_snake
$ git checkout --quiet -b feature/collidable-entity
```

Upravíme následující soubory projektu:

**js/SquareObject.js**

```
    } turn(velX, velY) {
        this.velocity.x = velX; this.velocity.y = velY;
    } hasSamePositionAs(point) {
        return (this.collidable && this.position.equals(point));
    } resolveCollisionWithSnake(snake, game) {
        game.gameOver();
```

```
    }
}
```

### js/Snake.js

```
        lastChunk.position.setTo(this.position); super.animate();
    } resolveCollisionWithSnake(snake, game) {
        this.tail.forEach(function (chunk) {
            if (chunk.hasSamePositionAs(snake.position))
                game.gameOver();
        });
    } hasSamePositionAs(point) {
        return this.tail.some(function (chunk) {
            return chunk.hasSamePositionAs(point);
        });  // not snake head itself
    }
}
```

### js/Game.js

```
    } animate() {
        this.gameObjects.forEach(function (entity) {
            if (entity.hasSamePositionAs(this.snake.position)) {
                entity.resolveCollisionWithSnake(this.snake, this);
            }
            entity.animate();
        }, this);
}
```

### Příklad 5.36. Přidání metod pro kolizi objektů

```
$ pwd # Konzole Dana
/Users/dan/Desktop/git_snake
$ git add js/SquareObject.js js/Snake.js
$ git commit --quiet -m "Přidání metod pro kolizi objektů"
$ git commit --quiet -a -m "Kolize entit s hlavou hada"
$ git checkout --quiet -b feature/game-food HEAD^1 ❶
$ touch js/Food.js
```

❶ Zápis **<commit>^N** znamená, že chceme *N*-tého rodiče uloženého v daném commitu.

Upravíme následující soubory (symbol **[…]** označuje vynechání části kódu ve výpisu):

### js/Food.js

```
"use strict";
GitSnake.Food = class Food extends GitSnake.SquareObject {
    constructor(pos) {
        super(pos); this.score = 1; this.color = "gray";
    } resolveCollisionWithSnake(snake, game) {
        this.position.setTo(game.getRandomFreeCell());
        snake.grow(this.score); game.increaseScore(this.score);
```

```
    }
}
```

### js/Game.js

```
        this.addEntity(this.snake);
        this.addEntity(new GitSnake.Food(this.getRandomFreeCell()));
    } addEntity(entity) {
[...]
    } pause() {
        this.isPlaying = false;
        clearInterval(this.drawingId); clearInterval(this.animationId);
    } getRandomFreeCell() {
        var cellPoint = new GitSnake.Point2D(
            Math.floor(Math.random() * this.tileX), Math.floor(Math.random() * this.tileY)
        );
        if (this.gameObjects.every(function (entity) {
            return entity.hasSamePositionAs(cellPoint) === false;
        })) {
            return cellPoint;
        } return this.getRandomFreeCell();
    } increaseScore(score) {
        this.score += score;
    }
}
```

### play.htm

```
    <script src="js/Game.js"></script>
    <script src="js/Food.js"></script>
    <script>
```

#### Příklad 5.37. Přidání jídla na náhodné pozici

```
$ pwd # Konzole Dana
/Users/dan/Desktop/git_snake
$ git add js/Food.js ❶
$ git commit --quiet -a -m "Přidání jídla na náhodné pozici"
$ git checkout --quiet develop
$ git fetch
$ git l feature/game-food feature/collidable-entity -4 ❷
* e764ba8 (feature/game-food) Přidání jídla na náhodné pozici
| * bce0dc8 (feature/collidable-entity) Kolize entit s hlavou hada
|/
* 3103eff Přidání metod pro kolizi objektů
* 7efeb7e (HEAD -> develop) Přidání .gitignore pro složku images
$ git merge --quiet --no-edit feature/collidable-entity feature/game-food ❸
$ git branch --quiet -d feature/collidable-entity feature/game-food
$ git push --quiet
```

❶ Do indexu přidáme nesledovaný soubor *js/Food.js*.

❷ Použijeme alias z příkladu 5.31, který vytvoříme také pro uživatele Dana. Aliasu předáme reference, které chceme zahrnout v logu a také omezíme výstup na poslední 4 commity.

❸ Příkaz `git merge` zvládá sloučit i více větví. V tomto případě dojde k začlenění pomocí *octopus* strategie.

## 5.27. Implementace hratelnosti

Bob se rozhodl zapracovat na hratelnosti hry, a proto vytvořil novou větev pro ztížení hry:

**Příklad 5.38. Vytvoření větve pro ztížení hry**

```
$ pwd # Konzole Boba
/Users/bob/Desktop/bobGitSnake
$ git pull --quiet --ff-only
$ git checkout --quiet -b feature/game-difficulty
```

Bob upraví následující soubor:

**js/Game.js**

```
    } increaseScore(score) {
        this.score += score;
        if (this.score % 5 === 0) {
            this.animationSpeed = Math.max(50, this.animationSpeed - 5);
            this.pause(); this.play();
        }
    }
}
```

Po úpravě a uložení Bob provede commit. Bob si ovšem není jistý, zdali neprovede ještě další změny ve větvi *feature/game-difficulty*, a proto zatím nebude tuto větev začleňovat do větve *develop* a následně sdílet do hlavního repozitáře.

**Příklad 5.39. Ztížení hry po každých 5 bodech**

```
$ pwd # Konzole Boba
/Users/bob/Desktop/bobGitSnake
$ git commit --quiet -m "Ztížení hry po každých 5 bodech" js/Game.js
```

## 5.28. Začlenění do větve master

Verze hry *GitSnake* ve větvi *develop* v hlavním repozitáři již dost připomíná originální hru Had, proto provedeme začlenění větve develop do větve master a vytvoříme nový štítek. Po vydání verze *v1.0.0* pokračujeme ve vývoji příjemnějšího uživatelského rozhraní (ui[12]) hry.

[12] https://en.wikipedia.org/wiki/User_interface

### Příklad 5.40. Vydání verze 1.0.0

```
$ pwd # Konzole Dana
/Users/dan/Desktop/git_snake
$ git checkout --quiet master
$ git pull --quiet --rebase
$ git merge --quiet develop
$ git tag -a -m "Kolize objektů a nárůst hada po jídle" v1.0.0
$ git push --quiet --tags
$ git push --quiet
$ git checkout --quiet -b feature/game-ui
$ touch js/Text.js
```

Upravíme následující soubory:

### js/Text.js

```
"use strict";
GitSnake.Text = class Text extends GitSnake.SquareObject {
    constructor(pos, text) {
        super(pos); this.text = text; this.color = "gray"; this.collidable = false;
        this.attr = {
            overlay: { color: "white", opacity: 0 },
            font: "18px Arial", textAlign: "left"
        };
    } draw(ctx) {
        if (this.text === undefined)
            return;

        ctx.fillStyle = this.attr.overlay.color;
        ctx.globalAlpha = this.attr.overlay.opacity;
        ctx.fillRect(0, 0, ctx.canvas.width, ctx.canvas.height);
        ctx.globalAlpha = 1; ctx.textAlign = this.attr.textAlign;
        ctx.fillStyle = this.color; ctx.font = this.attr.font;
        ctx.fillText(this.text, this.position.x * this.size, this.position.y * this.size);
    }
}
```

### js/Game.js

```
        this.addEntity(this.snake);
        this.addEntity(new GitSnake.Food(this.getRandomFreeCell()));
        var pos = new GitSnake.Point2D(1, this.tileY - 1);
        this.scoreText = new GitSnake.Text(pos, "Skóre: " + this.score);
        this.scoreText.attr.font = "15px Arial"; this.addEntity(this.scoreText);
        pos = new GitSnake.Point2D(this.tileX / 2, this.tileY / 2)
        this.pauseText = new GitSnake.Text(pos);
        this.pauseText.attr.textAlign = "center"; this.pauseText.attr.overlay.opacity = 0.8;
    } setPauseText(text) {
        this.pauseText.text = text; this.pauseText.draw(this.ctx);
    } addEntity(entity) {
[...]
```

```
    } increaseScore(score) {
        this.score += score;
        this.scoreText.text = "Skóre: " + this.score;
    }
}
```

**play.htm**

```
    <script src="js/Game.js"></script>
    <script src="js/Food.js"></script><script src="js/Text.js"></script>
    <script>
        (function () {
            var game = new GitSnake.Game(document.getElementById("canvas"), 20, 30);
            game.setPauseText("Pro zahájení hry stiskněte mezerník nebo Esc");
            var snake = game.snake;
[...]
                        if (game.isPlaying) {
                            game.pause();
                            game.setPauseText("Hra pozastavena");
                        } else {
```

### Příklad 5.41. Přidání textů UI

```
$ pwd # Konzole Dana
/Users/dan/Desktop/git_snake
$ git add js/Text.js
$ git commit --quiet -a -m "Přidání textů UI"
$ git checkout --quiet develop
$ git merge --quiet --no-ff feature/game-ui ❶
$ git branch --quiet -d feature/game-ui
$ git push --quiet
```

❶ Přepínač *--no-ff* specifikuje, že nechceme použít metodu *fast-forward*, ale vytvořit nový (*merge*) commit.

## 5.29. Konflikt

Konflikt vzniká, pokud sjednocujeme větve, ve kterých došlo ke změně stejného řádku. V takovém případě Git neví, která varianta souboru je ta „správná“, a oznámí tedy konflikt a vyzve uživatele, aby konflikt vyřešil. Pro vyznačení konfliktu Git používá ve výchozím nastavení standartní značky konfliktů:

```
<<<<<<< HEAD
varianta v aktuální větvi (HEAD)
=======
varianta ve sjednocované větvi
>>>>>>> sjednocovaná větev
```

Bob se rozhodl, že jeho lokální větev *feature/game-difficulty* je dokončená, a tedy připravená pro začlenění do větve *develop* a odeslání do hlavního repozitáře:

### Příklad 5.42. Konflikt

```
$ pwd # Konzole Boba
/Users/bob/Desktop/bobGitSnake
$ git checkout --quiet develop
$ git pull --quiet --rebase
$ git merge feature/game-difficulty
Auto-merging js/Game.js
CONFLICT (content): Merge conflict in js/Game.js ❶
Automatic merge failed; fix conflicts and then commit the result.
$ tail -12 js/Game.js ❷
    } increaseScore(score) {
        this.score += score;
<<<<<<< HEAD ❸
        this.scoreText.text = "Skóre: " + this.score;
======= ❹
        if (this.score % 5 === 0) {
            this.animationSpeed = Math.max(50, this.animationSpeed - 5);
            this.pause(); this.play();
        }
>>>>>>> feature/game-difficulty  ❺
    }
}
```

❶ Git oznámil konflikt v souboru *js/Game.js*.[13]

❷ Pro zobrazení konfliktu můžeme otevřít soubor v textovém editoru nebo použít příkaz `tail <soubor>` s přepínačem *-12* pro zobrazení posledních 12 řádek souboru v parametru.

❸ Varianta (níže po oddělovač), kterou vytvořil Dan (*HEAD*).

❹ Oddělovač variant.

❺ Varianta (výše po oddělovač), kterou vytvořil Bob (*feature/game-difficulty*).

Pro vyřešení konfliktu otevřeme soubor *js/Game.js* v textovém editoru a upravíme následovně:[14]

### js/Game.js

```
    } increaseScore(score) {
        this.score += score; this.scoreText.text = "Skóre: " + this.score;
        if (this.score % 5 === 0) {
            this.animationSpeed = Math.max(50, this.animationSpeed - 5);
            this.pause(); this.play(); // refresh loops
        }
    }
}
```

[13] Pro zrušení **probíhajícího** začlenění a vrácení se do stavu před začleněním lze použít přepínač *--abort* příkazu `git merge`. Přepínač *--abort* lze použít i pro další příkazy například `git rebase` a další.

[14] Pro vyřešení konfliktu lze použít externí program pomocí příkazu `git mergetool`.

V tomto případě jsme vyřešili konflikt sloučením a úpravou změn z obou větví. Nyní musíme ještě říci Gitu, že jsme konflikt vyřešili přidáním souboru do indexu a provedením commitu:

**Příklad 5.43. Vyřešení konfliktu**

```
$ pwd # Konzole Boba
/Users/bob/Desktop/bobGitSnake
$ git status -s
UU js/Game.js
$ git add js/Game.js
$ git commit --quiet --no-edit ❶
$ git branch --quiet -d feature/game-difficulty
$ git push --quiet
```

❶ Po vyřešení konfliktu je dobrým zvykem popsat v popisu změny, jak jsme konflikt vyřešili spolu s dalšími informacemi. Zde ovšem pro jednoduchost použijeme již známý přepínač *--no-edit*, který použije popis vygenerovaný Gitem.

## 5.30. Cherry-pick

Příkaz `git cherry-pick <commit>` slouží k aplikování změny, kterou přinesl daný commit na aktuální větev. Tento příkaz se tak výborně hodí pokud udržujeme více větví projektu. Například ve větvi *develop* od posledního sjednocení s větví *master* přibylo mnoho commitů, které ještě nechceme sjednotit do větve *master*, ale jeden z těchto mnoha commitů opravuje velkou chybu, a tedy bychom změnu v tomto specifickém commitu chtěli aplikovat nad větví *master*.

V projektu *GitSnake* byla objevena chyba v pohybu hada. Opravením chyby byl pověřen Bob, který chybu diagnostikoval a vymyslel řešení problému:

**Příklad 5.44. Oprava chyby v pohybu hada**

```
$ pwd # Konzole Boba
/Users/bob/Desktop/bobGitSnake
$ git fetch
$ git checkout --quiet -b hotfix/snake-move
```

**js/SquareObject.js**

```
        this.velocity = new GitSnake.Point2D(0, 0);
        this.lastVelocity = this.velocity.copy();
    } draw(ctx) {
[...]
    } animate() {
        this.position.x += this.velocity.x; this.position.y += this.velocity.y;
        this.lastVelocity.x = this.velocity.x; this.lastVelocity.y = this.velocity.y;
    } turn(velX, velY) {
```

**js/Snake.js**

```
        this.tail.push(new GitSnake.SquareObject(lastChunk.position.copy()));
    }
} turn(velX, velY) {
    if (this.lastVelocity.x !== -(velX) || this.lastVelocity.y !== -(velY))
        super.turn(velX, velY);
} draw(ctx) {
    this.tail.forEach(function (chunk) {
```

**Příklad 5.45. Omezení nepovoleného pohybu hada**

```
$ pwd # Konzole Boba
/Users/bob/Desktop/bobGitSnake
$ git commit -a -m "Omezení nepovoleného pohybu hada"
[hotfix/snake-move 61fe963] Omezení nepovoleného pohybu hada
 2 files changed, 5 insertions(+)
$ git checkout --quiet develop
$ git merge --quiet --no-ff --no-edit hotfix/snake-move
$ git branch --quiet -d hotfix/snake-move
$ git push --quiet ❶
$ git checkout --quiet master
$ git pull --quiet --rebase
$ git cherry-pick 61fe963 ❷
[master 24b8edd] Omezení nepovoleného pohybu hada
 Date: Fri Apr 22 16:22:37 2016 +0100
 2 files changed, 5 insertions(+)
$ git tag -a -m "Omezení nepovoleného pohybu hada" v1.1.0
$ git push --quiet hlavni v1.1.0
$ git push --quiet ❸
$ git checkout --quiet develop
```

❶ Oprava chyby ve větvi *develop* byla odeslána do hlavního repozitáře pro stažení ostatními vývojáři.

❷ Po přepnutí na větev *master* aplikujeme commit (změnu v commitu) „61fe963“ nad větví *master*.

❸ Odešleme a začleníme změny větve *master* do hlavního repozitáře.

## 5.31. Přejmenování/přesunutí souborů (mv)

Git na rozdíl od jiných verzovacích systémů neukládá informace o přejmenování souborů. Pokud přejmenujeme soubor s názvem *A* na *B*, Git toto přejmenování chápe tak, že se smazal soubor s názvem *A* a obsahem *O* a zároveň přibyl nový soubor s názvem *B* a obsahem *O*. Pokud ovšem změníme obsah souboru *A* a zároveň provedeme přejmenování z *A* na *B*, situace je trochu komplikovanější a Git musí použít heuristiku pro odhad, zdali se jedná o nový soubor, nebo přejmenování souboru původního. Tato heuristika funguje tak, že pokud je shoda obsahu *X* % a více (kde *X* se dá uživatelsky definovat) je tato operace rozpoznána jako přejmenování, v opačném případě se jedná o smazání původního souboru a přidání souboru nového. Nejlepší je proto, pokud měníme obsah souboru a zároveň chceme soubor přejmenovat, vytvořit dva commity, ovšem ne vždy je toto možné (například pokud se obsah souboru

váže s názvem daného souboru). Pro ukázku Bob upraví soubor *README.txt* do Markdown[15] syntaxe následovně:

**README.txt**

```
# GitSnake

- Vlastní implementace známé hry Snake.
- Projekt slouží jako ukázka práce v systému Git.
```

Po uložení a zavření souboru v editoru se vrátíme do konzole, kde provedeme dva commity.

**Příklad 5.46. Přesunutí/přejmenování souboru**

```
$ pwd # Konzole Boba
/Users/bob/Desktop/bobGitSnake
$ git commit --quiet -a -m "README převedeno do Markdown syntaxe" ❶
$ git mv README.txt README.md ❷
$ git commit --quiet -m "README.txt přejmenováno na README.md"
$ git push --quiet
```

❶ První commit se změnou obsahu.

❷ Přejmenování souboru *README.txt* v pracovním adresáři projektu i v indexu na *README.md* pomocí příkazu `git mv <starý název> <nový název>`. Příkaz `git mv` slouží jak pro přejmenování, tak i pro přesunutí souboru obdobně jako unixová verze `mv`.

Git nikde neukládá informaci o přejmenování, a proto pokud chceme zjistit například historii souboru „*README.md*" ve větvi *HEAD*, musíme použít `git log --follow README.md`. Přepínač *--follow* instruuje Git, aby aplikoval heuristiku pro detekování přejmenování souboru.

## 5.32. Závěrečné začlenění do větve master

Projekt *GitSnake* již v této práci nebudeme upravovat, a proto provedeme závěrečné začlenění větve *develop* do větve *master*. Také si zobrazíme porovnání s první verzí projektu.

**Příklad 5.47. Sjednocení větví develop a master**

```
$ pwd # Konzole Boba
/Users/bob/Desktop/bobGitSnake
$ git checkout --quiet master
$ git merge --quiet --no-edit develop
Removing README.txt
$ git push --quiet
$ git diff --stat v0.1.0 HEAD
 .gitignore         |  1 +
 README.md          |  4 +++
```

[15] https://daringfireball.net/projects/markdown/

```
README.txt         |  4 ---
images/.gitkeep    |  0
js/Food.js         |  9 +++++++
js/Game.js         | 77 +++++++++++++++++++++++++++++++++++++++++++++++++++++
js/Point2D.js      | 13 +++++++++
js/Snake.js        | 32 ++++++++++++++++++++++
js/SquareObject.js | 22 ++++++++++++++++
js/Text.js         | 20 ++++++++++++++
play.htm           | 35 ++++++++++++++++++++++++
11 files changed, 213 insertions(+), 4 deletions(-)
```

## 5.33. Blame

Příkaz `git blame <reference> <název souboru>` slouží pro vypsání v jakém commitu a kdo v dané referenci **naposledy** změnil každý řádek daného souboru. Například pro zobrazení *blame* souboru *README.md* v referenci *HEAD* použijeme příkaz `git blame README.md`.

## 5.34. Garbage collector (gc)

Pokud vyhledáme všechny soubory v objektové databázi podobně jako v příkladu 4.1, zjistíme, že se objektová databáze poměrně rozrostla. Pro ušetření místa obsahuje Git mechanismus, který komprimuje objektovou databázi a také reference. Příkaz `git repack` slouží pro vytvoření tzv. zabalených (*packed*) objektů, které výrazně šetří úložný prostor a zároveň i datový přenos, například při klonování repozitáře po síti. `Git repack` se spouští jako subrutina garbage collectoru (příkaz `git gc`), který slouží pro údržbu repozitáře. Hlavním úkolem garbage collectoru je smazat objekty, které jsou nedosažitelné[16], a provést zabalení (*pack*) objektů a referencí. Nedosažitelné objekty jsou všechny objekty, na které se nedá dostat z libovolné reference nebo štítku.[17]

**Příklad 5.48. Spuštění garbage collectoru**

```
$ pwd # Konzole Boba
/Users/bob/Desktop/bobGitSnake
$ du -sh .git/ ❶
226K     .git/
$ git gc
Counting objects: 150, done.
Delta compression using up to 4 threads.
Compressing objects: 100% (127/127), done.
Writing objects: 100% (150/150), done.
Total 150 (delta 56), reused 55 (delta 20)
$ du -sh .git/
136K     .git/
```

❶ Příkaz `du` slouží k zjištění, kolik daný soubor (složka) zabírá místa na disku.

[16] Ve výchozím nastavení Git maže pouze nedosažitelné objekty starší než 2 týdny.

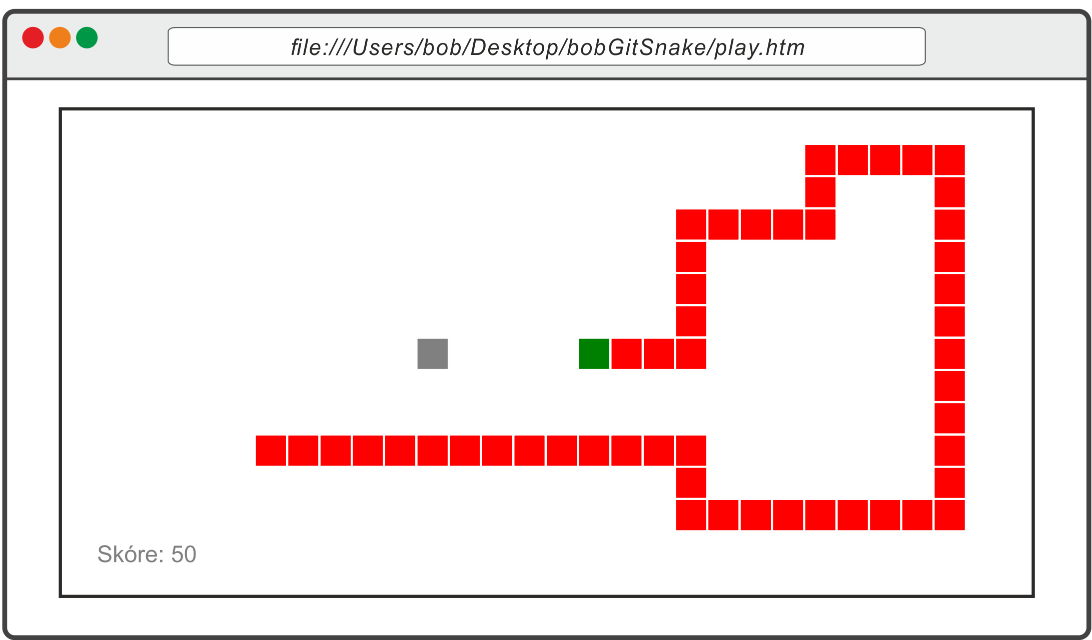
[17] Pro vypsání všech nedosažitelných objektů lze použít příkaz `git fsck --unreachable`.

# 6. Shrnutí výsledků

Teoretická část této práce byla věnována obecným informacím o verzovacích systémech, a především pak teorii okolo verzovacího systému Git.

S teoretickou částí úzce souvisela praktická část, která se zaměřila na práci v systému Git od prvního otevření konzole a ověření správnosti nainstalování systému Git přes provedení 32 commitů až po závěrečné spuštění garbage collectoru pro údržbu repozitáře v roli vývojáře Boba.

Celou tuto práci provázel praktický projekt *GitSnake*, který po přečtení práce a spuštění z pracovního adresáře projektu Boba vypadá například následovně:




**Obrázek 6.1. Ukázka výsledného praktického projektu [autor]**

Vytvořením tohoto praktického projektu vznikl následující graf historie projektu:

### Příklad 6.1. Výsledný graf projektu GitSnake

```
$ pwd # Konzole Boba
/Users/bob/Desktop/bobGitSnake
$ git l
*   a6eb3d8 (HEAD -> master, hlavni/master, hlavni/HEAD) Merge branch 'develop'
|\
| * c5d8060 (hlavni/develop, develop) README.txt přejmenováno na README.md
| * 6116ead README převedeno do Markdown syntaxe
| *   cecdaa4 Merge branch 'hotfix/snake-move' into develop
| |\
| | * 61fe963 Omezení nepovoleného pohybu hada
| |/
| *   411ce84 Merge branch 'feature/game-difficulty' into develop
| |\
| | * 13711f4 Ztížení hry po každých 5 bodech
| * |   b62eae9 Merge branch 'feature/game-ui' into develop
| |\ \
| | |/
| |/|
| | * 80fdf41 Přidání textů UI
| |/
* | e433a7c (tag: v1.1.0) Omezení nepovoleného pohybu hada
|/
*   deb3b55 (tag: v1.0.0) Merge branches 'feature/collidable-entity' and 'feature/game-food'
 into develop
|\
| * e764ba8 Přidání jídla na náhodné pozici
* | bce0dc8 Kolize entit s hlavou hada
|/
* 3103eff Přidání metod pro kolizi objektů
* 7efeb7e Přidání .gitignore pro složku images
* dd290f0 Vytvoření složky pro ukládání obrázků
* c8ff334 Revert "Odstranění souboru play.htm"
* e838400 Odstranění souboru play.htm
* a0a0d97 (tag: v0.2.1) Oprava mazání plátna v Game.js:draw()
* d448862 (tag: v0.2.0) Restart hry v případě výhry či prohry
* ba70595 Ovládání hada pomocí šipek klávesnice
* 07e8f32 Přidání turn metody ve SquareObject.js
* c100627 Horizontální vycentrování plátna na stránce
*   c0593d7 Merge branch 'feature/snake-grow' into develop
|\
| * e524045 Přidání grow(size) metody hada
* | 38f1466 Play/Pause hry klávesou mezerník nebo ESC
* | d9b576f Implementace animací objektů
|/
* b588ff6 Přidání základních objektů hada
* dde6568 Přidání základního HTML kódu v play.htm
* 33012c8 Přidání HTML5 hlavičky
* ccf57ac Vytvoření základních souborů hry
* d3173a9 (tag: v0.1.0) Přidání README projektu
```

# 7. Závěr

Cílem této práce bylo poskytnout základní informace o verzovacích systémech, jak fungují, proč a k jakému účelu byly primárně vytvořeny. Po seznámení se s verzovacími systémy následoval popis verzovacího systému Git, kterému se věnovala většina této práce. Práce s Gitem byla psaná pro uživatele, kteří s verzovacími systémy (Gitem) začínají, a byly popsány každodenní případy užití Gitu, se kterými se uživatelé setkají nejčastěji. Výstupem těchto každodenních případů vznikl praktický projekt – hra Had.

Tato práce nerozebírá Git do největších detailů především kvůli omezení počtu stránek práce, ale slouží spíše jako začátečnická příručka. Pro pokročilejší práci s Gitem bylo doporučeno mnoho zdrojů, kde se o problematice Gitu píše podrobněji. Zde si dovolíme opakovat se a znovu doporučíme jako zdroj oficiální nápovědu Gitu, dostupnou například na domovské stránce Gitu, která je hned po zkoumání zdrojového kódu samotného Gitu nejlepším místem pro porozumění tak mocného nástroje, jakým Git je.

# 8. Seznam použitých zdrojů


[1] Git [online]. In: *Git-scm*. [cit. 2016-03-07]. Dostupné z: https://git-scm.com/docs/

[2] The Open Group Base Specifications Issue 7 IEEE Std 1003.1™, 2013 Edition. In: *The Open Group Publications Server* [online]. [cit. 2016-04-01]. Dostupné z: http://pubs.opengroup.org/onlinepubs/9699919799/

[3] SINK, Eric. Version control by example. 1st ed. Champaign, Ill.: *Pyrenean Gold Press*, 2011, ix, 277 p. ISBN 9780983507901.

[4] Understanding Version-Control Systems (DRAFT). In: *Eric S. Raymond's Home Page* [online]. [cit. 2016-04-01]. Dostupné z: http://www.catb.org/esr/writings/version-control/version-control.html

[5] ROCHKIND, Marc J. *The source code control system. IEEE Transactions on Software Engineering* [online]. 1975, SE-1(4): 364-370 [cit. 2016-04-01]. DOI: 10.1109/tse.1975.6312866

[6] TICHY, Walter F. *Rcs - a system for version control. Software: Practice and Experience* [online]. 1985, 15(7): 637-654 [cit. 2016-04-01]. DOI: 10.1002/spe.4380150703.

[7] Concurrent Versions System CVS. In: *Dick Grune* [online]. [cit. 2016-04-01]. Dostupné z: http://www.dickgrune.com/Programs/CVS.orig/

[8] Apache™ Subversion®. In: *Subversion* [online]. [cit. 2016-04-01]. Dostupné z: https://subversion.apache.org/

[9] Initial revision of "git", the information manager from hell. In: *Kernel.org git repositories* [online]. [cit. 2016-03-07]. Dostupné z: http://git.kernel.org/cgit/git/git.git/commit/?id=e83c5163316f89bfbde7d9ab23ca2e25604af290

[10] Matt Mackall. In: *Mercurial* [online]. [cit. 2016-04-01]. Dostupné z: https://www.mercurial-scm.org/wiki/mpm

[11] Git [online]. In: *Git-scm*. [cit. 2016-03-07]. Dostupné z: https://git-scm.com/

[12] Linux-2.6.12-rc2. In: *Kernel.org git repositories* [online]. [cit. 2016-04-01]. Dostupné z: http://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/commit/?id=1da177e4c3f41524e886b7f1b8a0c1fc7321cac2

[13] SANTACROCE, Ferdinando. *Git Essentials*. Packt Publishing, 2015. ISBN 978-1785287909.

[14] Meaning of "git" in the English Dictionary. In: *Cambridge Dictionaries Online* [online]. [cit. 2016-03-07]. Dostupné z: http://dictionary.cambridge.org/dictionary/english/git

[15] CHACON, Scott a Ben STRAUB. *Pro Git*. 2nd ed. 2014. ISBN 978-1484200773.

[16] US Secure Hash Algorithm 1 (SHA1). In: *IETF Tools* [online]. [cit. 2016-04-01]. Dostupné z: https://tools.ietf.org/html/rfc3174

[17] LOELIGER, Jon a Matthew MCCULLOUGH. *Version control with Git*. 2nd ed. 2012. ISBN 978-1449316389.

[18] CHACON, Scott. *Git Internals* [online]. 2008 [cit. 2016-03-07]. Dostupné z: https://github.com/pluralsight/git-internals-pdf/raw/master/drafts/peepcode-git.pdf

[19] How to Write a Git Commit Message. *Chris Beams* [online]. [cit. 2016-03-07]. Dostupné z: http://chris.beams.io/posts/git-commit/

Univerzita Hradec Králové
Fakulta informatiky a managementu
Akademický rok: 2015/2016

Studijní program: Aplikovaná informatika
Forma: Prezenční
Obor/komb.: Aplikovaná informatika (ai3-p)

**Podklad pro zadání BAKALÁŘSKÉ práce studenta**

| PŘEDKLÁDÁ: | ADRESA | OSOBNÍ ČÍSLO |
|---|---|---|
| Maixner Daniel | Mírová 1541, Rychnov nad Kněžnou | I1301256 |

**TÉMA ČESKY:**

Systémy pro správu verzí se zaměřením na Git

**TÉMA ANGLICKY:**

Version Control Systems with Focus on Git

**VEDOUCÍ PRÁCE:**

doc. Ing. Filip Malý, Ph.D. - KIKM

**ZÁSADY PRO VYPRACOVÁNÍ:**

Cílem práce je podat obecné informace o verzovacích systémech spolu s bližším seznámením se systémem Git.

Osnova:
1. Úvod
2. Historie verzovacích systémů
3. Systémy pro správu verzí
4. Systém Git
5. Závěr
6. Literatura

**SEZNAM DOPORUČENÉ LITERATURY:**


CHACON, Scott a Ben STRAUB. Pro Git. 2nd ed. 2014. ISBN 978-1484200773.
LOELIGER, Jon a Matthew MCCULLOUGH. Version control with Git. 2nd ed. 2012. ISBN 978-1449316389.


**Podpis studenta:** ........................................ **Datum:** 25.11.2015

**Podpis vedoucího práce:** ........................................ **Datum:** 25.11.2015