Strategije in izzivi pri prehodu dolgo živečih projektov iz SVN na GIT

Gregor Novak, Grega Krajnc, Izidor Pokrivač

SRC d.o.o., Ljubljana, Slovenija
gregor.novak@src.si, grega.krajnc@src.si, izidor.pokrivac@src.si

Migracija starejših, a še aktivnih projektov iz sistema za nadzor različic Subversion (SVN) na Git je lahko zahteven, a dolgoročno zelo koristen proces. V prispevku predstavljamo praktično izkušnjo selitve večletnih projektov iz SVN na Git in opišemo izzive, s katerimi smo se srečali pri preslikavi zgodovine revizij, kot so ohranjanje metapodatkov in pravilne strukture vej, oznak ter združitev. Opišemo uporabo orodij, kot so git-svn, git filter-repo in lastni skripti, s katerimi smo avtomatizirali in poenostavili korake urejanja zgodovine. Selitev smo izkoristili tudi za prenovo razvojnega procesa. Uvedli smo delovni tok na osnovi uveljavljenih praks za Git, integracijo zahtev za združitev in novo CI/CD infrastrukturo. Posebej naslavljamo vzporedno podporo dveh aplikacijskih platform z ločenimi procesi znotraj Git okolja. Prehod na Git nam je omogočil uporabo bolj prilagodljive, pregledne in sodobne razvojne infrastrukture, lažjo integracijo z orodji za sledenje nalogam ter lažjo vključenost pregledov kode v vsakdanji proces. Migracijo razumemo kot izhodišče za uvajanje naprednejših praks, saj Git s svojo fleksibilnostjo podpira nenehno izboljševanje delovnega toka in sledenje sodobnim tehnologijam ter pristopom razvoja programske opreme.

Ključne besede:
nadzor različic,
Subversion,
SVN,
GIT,
delovni proces,
modernizacija,
razvojna orodja.

Univerzitetna založba Univerze v Mariboru
ISBN 978-961-299-037-4
DOI https://doi.org/10.18690/um.feri.7.2025.4

1 Uvod

V sodobnem razvoju programske opreme je učinkovito upravljanje različic kode ključnega pomena za uspešno sodelovanje in hiter razvoj. Medtem ko se je Git uveljavil kot prevladujoče orodje za nadzor različic, številne ekipe še vedno uporabljajo starejše sisteme, kot je Subversion (v nadaljevanju SVN), zlasti pri starejših, a še vedno aktivnih projektih.

Git je bil razvit leta 2005 za potrebe razvoja jedra Linuxa [1]. Gre za distribuiran sistem za upravljanje različic, pri katerem ima vsak razvijalec lokalno kopijo celotne zgodovine projekta. To omogoča delo tudi brez stalne internetne povezave. Nasprotno pa SVN temelji na centraliziranem modelu, kjer se zgodovina sprememb nahaja na strežniku. Posledično so določene operacije počasnejše in odvisne od omrežne povezave.

Ena izmed ključnih prednosti Gita je učinkovito delo z vejami (angl. branches). Razvijalci lahko z njimi preprosto ločijo nove funkcionalnosti, eksperimentalne spremembe ali popravke napak od glavne kode, ne da bi ogrozili stabilnost projekta. Čeprav tudi SVN omogoča uporabo vej, je delo z njimi bolj zapleteno in manj učinkovito kot pri Gitu. Veje v SVN se tako po navadi uporabljajo le izjemoma, medtem ko je njihova uporaba v Gitu stalnica.

Poleg tehničnih prednosti Git izstopa tudi po močni podpori gostiteljskih platform, kot so GitHub, GitLab in Bitbucket. Te ponujajo sodobna orodja za pregledovanje kode, sodelovanje, avtomatizacijo CI/CD procesov in upravljanje projektov, kar izboljšuje razvojni proces. SVN sicer ostaja zanesljiv sistem, a njegova manj prilagodljiva arhitektura in nižja razširjenost sodobnih orodij je razlog, da vse bolj izgublja stik s potrebami sodobnih razvojnih ekip.

1.1. Motivacija za selitev starejših projektov

V podjetju SRC d.o.o. za vse novejše projekte uporabljamo Git kot privzeti sistem za upravljanje različic, hkrati pa še vedno vzdržujemo starejše projekte, pri katerih se je do sedaj uporabljal SVN. Kljub njihovi starosti so ti projekti v vsakodnevni produkcijski uporabi, njihova stabilnost pa je kritična za delo končnih uporabnikov. Ti projekti niso zgolj pasivno vzdrževani – redno jih nadgrajujemo in razvijamo naprej. Ravno zaradi tega je selitev iz SVN na Git imela smisel: kljub njihovi starosti, so projekti živi, uvedba Gita pa zanje pomeni tudi uvedbo sodobnega razvojnega okolja, boljša orodja in višjo prilagodljivost razvojnega cikla.

O migraciji na Git smo razmišljali že dlje časa, a selitev ni bila prioriteta, saj je delovni proces s SVN-jem deloval stabilno, robustno in varno. Zaradi drugih nalog vzdrževanja in nadgrajevanja projektov pa selitvi na Git tudi nismo uspeli nameniti primernega časovnega okna, da bi se migracije lotili na temeljit in strukturiran način, kljub temu da bi nam uporaba Gita lahko že v preteklosti prinesla dodano vrednost k razvojnemu procesu.

Projekti, pri katerih se je uporabljal SVN, so v večini javanski zaledni sistemi. Tekom njihovega vzdrževanja se je pojavila potreba po selitvi zalednih sistemov iz ene aplikacijske platforme na drugo (v nadaljevanju platforma A in platforma B). V času prehoda smo morali zagotavljati hkratno izdajo posameznih različic, kot tudi pravilno delovanje sistemov za obe platformi. Git ponuja učinkovito upravljanje z vejami, hkrati pa omogoča tudi pripravo kopij celotnih repozitorijev. Zaradi tega je delovni proces, ki smo ga morali vzpostaviti, lažje doseči z Gitom kot s SVN-jem. Menjava aplikacijskih platform in vzporedno vzdrževanje sistemov na obeh platformah je bila tako naravna prelomnica projektov, pri kateri je postala selitev na Git ne le smiselna, temveč tudi potrebna za lažje upravljanje razvoja.

2 Naš pristop k selitvi na Git

Ob prehodu iz SVN na Git smo si zadali jasen cilj: selitev mora biti premišljena, varna in brez vpliva na stabilnost projektov. Poseben poudarek smo namenili ohranjanju celotne zgodovine sprememb, saj se ta vsakodnevno uporablja pri analizah, odpravljanju težav in razumevanju konteksta odločitev iz preteklosti. Naša zgodovina obsega več let razvoja, številne spremembe in tudi menjave članov ekip. Podatki v zgodovini, kot so denimo številke nalog v revizijskih sporočilih, so tako pogosto edina vez z razvojnimi odločitvami iz preteklosti.

Zavedali smo se tudi, da selitev ni le tehnični prenos repozitorija, temveč vključuje spremembo razvojne kulture in orodij, ki zajema drugačen potek razvoja, drugačno integracijo CI/CD in predvsem drugačen način vsakodnevnega dela. Želeli smo, da je ob izvedbi selitve vse pripravljeno za nemoten prehod – brez potrebe po vrnitvi v SVN-okolje ali ročnih prilagoditev v zadnjem trenutku. V nadaljevanju poglavja tako na kratko predstavljamo ključne faze, ki so zaznamovale naš prehod na Git.

2.1. Izbira primernega trenutka za selitev

Selitve nismo izvedli impulzivno. Odločitev o prehodu smo sprejeli premišljeno in v pravem trenutku – ko so se potrebe po razvejanem razvoju in večji fleksibilnosti ujemale s prelomnico v razvojnem ciklu: selitvijo projektov na novo aplikacijsko platformo. Želeli smo, da bo v trenutku prehoda na Git vzpostavljeno vse potrebno: prenesena zgodovina repozitorijev, preizkušena integracija CI/CD in jasno definiran delovni proces. Le na ta način smo lahko zagotovili, da se bo razvoj lahko nadaljeval brez prekinitev ali dodatnih tveganj. Prehod smo torej načrtovali vnaprej in na način, da bi bila selitev za ekipo čim manj moteča, razvojna okolja pa kar se da usklajena.

2.2. Prenos zgodovine projektov na Git

Zgodovina razvoja je za nas izjemno dragocena. Vsebovala je več let sprememb, pogosto kompleksne in povezane z različnimi zahtevami. Ena ključnih zahtev pri selitvi je bila torej ohraniti čim bolj dosledno, pregledno in uporabno zgodovino v Gitu. Pri prenosu zgodovine smo se soočili z izzivi – zaradi arhitekturnih razlik SVN-ja in Gita je avtomatska migracija pogosto prinesla popačen potek zgodovine. Za ustrezno predstavitev zgodovine v Gitu smo zato izvajali ročne posege, preuredili strukturo commitov, da izraža dejansko stanje iz SVN-ja in pripravili skripte za avtomatizacijo prilagoditev. S tem smo zagotovili, da migrirana zgodovina izraža dejansko stanje poteka razvoja iz preteklosti, kar omogoča njeno nadaljnjo uporabnost v Gitu.

2.3. Načrtovanje delovnega procesa

Migracija je bila tudi priložnost za prenovo razvojnega procesa. Eden od ciljev je bil, da ne podvajamo navad iz SVN-ja v Gitu, temveč se premaknemo k uveljavljenim praksam za Git, kot sta razvejano razvijanje in združevanje kode preko zahtev za združitev. Ker smo morali vzdrževati dve vzporedni različici aplikacije – za platformo A in platformo B – smo za čas hkratne podpore oblikovali proces, ki izkorišča zmožnost podvajanja Git-repozitorijev s celotno zgodovino, s čimer je tudi CI/CD proces izoliran zgolj na dotično platformo. Z načrtnim pristopom smo tako ustvarili okolje, v katerem lahko razvoj poteka učinkoviteje, pregledneje in z boljšimi orodji kot prej, predvsem pa smo s prenovljenim delovnim tokom zadostili novim potrebam razvojnega procesa.

3 Selitev zgodovine

Selitev zgodovine revizij iz SVN-ja na Git se je izkazala za najzahtevnejši in časovno najintenzivnejši del celotnega procesa. Začetna preslikava revizij iz SVN-ja v Git-commite je bila sicer preprosta, a smo hitro ugotovili, da rezultati niso popolnoma ustrezni. Pridobljena zgodovina v Gitu je bila v več primerih popačena ne glede na izbrano orodje za migracijo. Razlika med arhitekturama SVN-ja in Gita se namreč odraža tudi v tem, kako se interpretirajo veje in oznake (angl. tags). Posledično prenesen repozitorij ni v celoti izražal stanja zgodovine, kot bi ga naravno pričakovali v Gitu.

Za doseganje zgodovine, ki izraža pravilno stanje in pričakovan potek smo morali izvajate ročne posege v strukturo commitov. Kljub temu, da je bilo takšno dodatno urejanje zgodovine časovno zamudno, smo se za to odločili, ker smo se s selitvijo na Git želeli znebiti odvisnosti od SVN-ja, a hkrati tudi ohraniti pravilnost podatkov, ki nam jih nudi obstoječa zgodovina. V tem poglavju tako podrobneje predstavljamo orodja, ki smo jih uporabili pri prenosu in urejanju zgodovine ter korake, ki smo jih izvedli, da smo dosegli repozitorij s pravilno strukturo vej, oznak in metapodatkov commitov.

Zaradi obsežnosti in časovne zahtevnosti postopka smo ugotovili, da je prenos zgodovine smiselno izvesti dovolj zgodaj – še preden so pripravljeni vsi drugi deli migracije. Glavni del zgodovine lahko namreč prenesemo vnaprej, aktiven razvoj pa do dejanskega prehoda na Git še vedno poteka v SVN-ju. Kasnejše revizije iz SVN-ja lahko še vedno uvozimo in jih dodamo k že preneseni zgodovini, vse do zaključka migracije na Git.

3.1. Uporabljena orodja

git-svn

git-svn je orodje, ki omogoča kloniranje SVN-repozitorijev v Git in preslikavo SVN-revizij v Git-commite [2]. Gre za ukaz, vgrajen v Git, ki omogoča delo z obstoječimi SVN-repozitoriji kar znotraj Gitove infrastrukture. Preizkusili smo tudi druga orodja, kot sta svn2git [3] in SubGit [4], vendar se je git-svn izkazal za najprimernejšo izbiro. Orodje je enostavno za uporabo, ne zahteva dodatnih odvisnosti in je v celoti brezplačno. Na našo odločitev pa je prav tako vplivalo dejstvo, da smo tudi pri ostalih orodjih naleteli na podobne težave pri kakovosti migrirane zgodovine. Zato smo git-svn prepoznali kot rešitev, ki je najbolj preprosta za uporabo, hkrati pa nam še vedno omogoča zadostno preglednost in prilagodljivost pri nadaljnji obdelavi.

Lastni namenski skripti

Za poenostavitev postopkov čiščenja zgodovine in rekonstrukcijo logične strukture repozitorija smo pripravili lastne skripte. Njihov namen je bil odprava potrebe po ročnem izvajanju ponavljajočih, daljših operacij, s čimer smo skrajšali čas, potreben za urejanje zgodovine.

git filter-repo

Orodje git filter-repo je sodoben in zmogljiv pripomoček za prečiščevanje zgodovine repozitorijev v Gitu. Predstavlja uradno priporočeno zamenjavo zastarelega ukaza `git filter-branch` [5]. Orodje smo uporabili predvsem za popravljanje metapodatkov po operacijah, kot je rebase, da smo v končnem repozitoriju ohranili dosledne in točne podatke o zgodovini sprememb. Drug razlog za uporabo orodja pa je bila prilagoditev datotečne strukture repozitorija, saj so bile v starejših delih zgodovine zaradi preteklih praks prisotne binarne datoteke, ki jih v Gitu nismo želeli ohranjati. Git namreč ni optimalen za verzioniranje binarnih vsebin, saj ob vsaki spremembi shrani celotno vsebino datoteke. Ohranjanje takšnih datotek v zgodovini bi povzročilo nesorazmerno povečanje velikosti repozitorija na disku in otežilo njegovo vsakodnevno uporabo.

Grafični vmesnik za Git

Za hitrejši pregled in primerjavo commitov, vizualizacijo grafa in preverjanje metapodatkov smo uporabljali tudi grafični vmesnik za Git. Ta ni bil nujno potreben, vendar nam je prihranil precej časa predvsem zato, ker nam je omogočil neposreden pregled stanja brez vnašanja dodatnih ukazov v ukazni vrstici.

Uporabljali smo integriran vmesnik za delo z Gitom znotraj okolja IntelliJ IDEA [6], ki je sicer enoten za vsa JetBrainsova orodja [7]. Ta izbira je bila predvsem stvar priročnosti, saj to okolje uporabljamo že pri samem razvoju. Seveda pa enake funkcionalnosti nudijo tudi številna druga grafična orodja za Git. Ključno je bilo, da smo z vmesnikom hitro preverjali:

- strukturo grafa commitov (veje, oznake in združitve),
- razlike med vsebino commitov ter
- metapodatke commitov (avtorji, časi, sporočila).

3.2. Začetni prenos zgodovine

Priprava datoteke s podatki o avtorjih

Za prenos zgodovine iz SVN-ja v Git smo uporabili ukaz `git svn clone`. Pred tem je bilo potrebno pripraviti datoteko z avtorji, saj git-svn brez nje ne more pravilno preslikati avtorjev iz SVN-ja.

Za pridobitev osnovnega seznama avtorjev iz SVN-repozitorija smo uporabili naslednji cevovod v Bashu.

```
svn log -q <svn-repo-url> | \
    awk -F '|' '/^r/ {gsub(/ /, "", $2); sub(" $", "", $2); \
    print $2" = todo-username <todo-email@todo.com>"}' | \
    sort -u > authors.txt
```

Cevovod na mestu, kjer smo ga pognali, ustvari datoteko z vsebino avtorjev iz SVN-ja v obliki, kot jo prikazuje spodnji primer.

```
1  MartinKrpan = todo-username <todo-email@todo.com>
2  PeterKlepec = todo-username <todo-email@todo.com>
3  peterklepec = todo-username <todo-email@todo.com>
4  VeronikaCeljska = todo-username <todo-email@todo.com>
5  VeronikaDeseniska = todo-username <todo-email@todo.com>
```

Datoteko smo pred njeno uporabo vsebinsko uredili, da smo uskladili imena avtorjev in določili njihove dejanske elektronske naslove. Pri tem smo uporabili isto obliko podatkov, kot jo razvijalci že uporabljamo pri drugih projektih v Gitu.

```
1  MartinKrpan = MartinKrpan <martin.krpan@src.si>
2  PeterKlepec = PeterKlepec <peter.klepec@src.si>
3  peterklepec = PeterKlepec <peter.klepec@src.si>
4  VeronikaCeljska = VeronikaCeljska <veronika.celjska@src.si>
5  VeronikaDeseniska = VeronikaCeljska <veronika.celjska@src.si>
```

Prikazana primera prenesene in urejene datoteke prikazujeta tudi možnost povezovanja različnih uporabniških imen iz SVN-ja z enotnim avtorjem v Gitu. S takšno ureditvijo datoteke dosežemo, da so commiti iz SVN-ja v Gitu pripisani pravim osebam ne glede na to, s kakšnim uporabniškim imenom so v preteklosti delale v SVN-ju.

Kloniranje SVN-repozitorija z git-svn

Ko je bila datoteka z avtorji pripravljena, smo jo podali ukazu `git svn clone` skupaj z naslovom SVN-repozitorija in imenom lokalnega direktorija, kamor smo želeli klonirati projekt.

```
git svn clone -s --authors-file=<authors-file-path> <svn-repo-url> <cloned-repo-dir-name>
```

Sami smo uporabili zastavico `-s` (sinonim za `--stdlayout`), ker so bili naši SVN-repozitoriji organizirani po standardni strukturi z mapami trunk, branches in tags. Če projekt odstopa od standardne postavitve, je potrebno poti do posameznih delov določiti ročno z zastavicami `--trunk`, `--branches` in `--tags`.

Prenos lahko pri obsežnejših projektih z daljšo zgodovino traja več ur ali celo dni. Med prenosom lahko pride do napak, recimo v primeru prekinjene omrežne povezave. V takšnih primerih lahko postopek ponovno zaženemo z enakim ukazom – git-svn bo nadaljeval tam, kjer je prenos prekinilo, pri tem pa ne potrebujemo izvesti dodatnih ukrepov nad že ustvarjenimi datotekami.

Commiti preneseni, na tak način, so vsebovali metapodatke v obliki, kot je predstavljena z naslednjim primerom.

```
$ git log -1 cc215e71
commit cc215e71797ae4a22df036f1c26907fe0c3bf9be
Author: MartinKrpan <martin.krpan@src.si>
Date:   Wed Jan 1 08:51:10 2025 +0000

    Initial version

    git-svn-id: https://example.svn.repo/trunk@1111 198ea1cc-1dab-4807-93c9-788d0729d413
```

Na dnu sporočila vsakega commita je viden dodatni zapis git-svn-id, ki predstavlja povezavo s pripadajočo SVN-revizijo. Ta metapodatek omogoča nadaljnjo sinhronizacijo z izhodiščnim repozitorijem – na primer za prenos novejših revizij ali dvosmerne integracije.

Če metapodatkov ne želimo ohraniti, lahko pri zagonu ukaza uporabimo še zastavico `--no-metadata`. Vendar v tem primeru izgubimo možnost nadaljnje integracije z izvornim SVN-repozitorijem. Če nam je ta integracija pomembna, metapodatkov pa v končnem repozitoriju vseeno ne želimo, zastavice ne uporabimo, prisotne metapodatke pa nato odstranimo z orodjem git filter-repo, ko integracije več ne potrebujemo. Mi se te integracije sicer nismo posluževali, smo pa metapodatke vseeno obdržali, da lahko jasneje ločimo med commiti, ki so nastali še v SVN-ju in kasnejšimi commiti, ki so bili dodani neposredno v Gitu.

3.3. Koraki po začetnem prenosu

Priprava lokalnega repozitorija brez reference na SVN

Ker povezave do izvornega SVN-repozitorija nismo več potrebovali, smo pred nadaljnjo obdelavo zgodovine najprej pripravili lokalni repozitorij, ki reference na SVN ne vsebuje več. Po prenosu zgodovine z git-svn je v pridobljenem repozitoriju izvorni SVN-repozitorij obravnavan podobno kot navaden oddaljen Git-repozitorij, z določenimi razlikami. Medtem ko lahko dostopamo do vseh commitov preko referenc na oddaljene veje, samega oddaljenega repozitorija ne moremo odstraniti enako, kot če bi šlo za navaden oddaljen repozitorij. Zaradi tega smo se reference na SVN znebili tako, da smo na novo klonirali obstoječ lokalni git-svn-repozitorij. Nov klon tako za oddaljen repozitorij ne bo več imel reference na SVN-repozitorij, temveč bo imel referenco na repozitorij, ki se nahaja v našem lokalnem datotečnem sistemu.

Najprej smo v kloniranem git-svn-repozitoriju za vse oddaljene veje iz SVN-ja ustvarili lokalne veje. S tem smo poskrbeli, da bodo vse veje vidne v klonu, ki je kloniran iz lokalnega git-svn-repozitorija.

```
git branch -r --no-color | grep -v '\->' | \
    while read remote_branch; do git branch "${remote_branch#origin/}" "$remote_branch"; done
```

Za tem smo izvedli kloniranje lokalnega git-svn-repozitorija.

```
1  git clone <local-git-svn-clone> <new-clone>
2  cd <new-clone>
3  git branch -r --no-color | grep -v '\->' | \
4      while read remote_branch; do git branch "${remote_branch#origin/}" "$remote_branch"; done
5  git remote rm origin
```

Pri tem smo v novem klonu morali ponovno izvesti ukaz za kreiranje lokalnih referenc vej, saj so pri svežem klonu veje ponovno dostopne zgolj kot oddaljene reference, s tem da je bil oddaljen repozitorij v tem primeru lokalni git-svn-klon. Na koncu smo odstranili še referenco oddaljenega repozitorija na lokalni git-svn-klon in s tem pridobili repozitorij, ki nima nobene reference oddaljenih repozitorijev.

Prvotni git-svn-klon smo obdržali kot rezervno kopijo, ker je lahko nadaljnje urejanje zgodovine destruktivno glede na prvotno preneseno zgodovino. Z ohranitvijo originalnega klona pa imamo možnost začeti znova brez ponovnega prenašanja z git-svn, če bi ob urejanju zgodovine morebiti izvedli nepopravljivo napako.

Zamenjava vej za oznake z dejanskimi oznakami

Zaradi načina predstavitve oznak v SVN-ju so te pri prenosu repozitorija z git-svn predstavljene kot veje, ki v imenu vsebujejo še predpono s potjo do oznake v datotečni strukturi SVN-repozitorija. Ker so v našem primeru imeli SVN-repozitoriji standardno strukturo, so veje, ki so predstavljale oznake, v imenu vsebovale predpono »tags/«. Oznake smo v Gitu želeli predstaviti kot dejanske oznake, ne kot veje s posebnimi imeni. Zato smo pripravili funkcijo v Pythonu, ki za vsako takšno vejo ustvari ustrezno oznako. Ime oznake pridobi iz imena veje, oznako pa postavi na isti commit, na katerega kaže veja. Po ustvarjeni oznaki funkcija vejo še izbriše, saj ta ni več potrebna.

```python
import subprocess


def tags_from_branches():
    # Ensure the current branch is main
    subprocess.run(["git", "checkout", "main"], check=True)

    # Get all local branches that include 'tags/' in their name
    output = subprocess.check_output(["git", "branch", "--list", "tags/*"], text=True)
    branches = [b.strip() for b in output.splitlines()]

    for tag_branch in branches:
        # Example: "tags/v1.0/v1.0.5" -> "v1.0.5"
        tag_name = tag_branch.split("/")[-1]

        print(f"Creating tag '{tag_name}' from branch '{tag_branch}'")
        subprocess.run(["git", "tag", tag_name, tag_branch], check=True)
        print(f"Deleting branch '{tag_branch}'")
        subprocess.run(["git", "branch", "-D", tag_branch], check=True)
```

Odločili smo se, da oznake iz SVN-ja v Gitu kreiramo kot lahke oznake (angl. lightweight tags), kar pomeni, da oznake ne vsebujejo anotacij s podatki o avtorju, sporočilu in času kreiranja oznake.

3.4. Ureditev zgodovine

Če bi bila prenesena zgodovina brez napak, bi bil repozitorij po kreiranju lokalne kopije in ustvarjanju pravih oznak že pripravljen za nadaljnjo uporabo. V našem primeru pa so se pojavile nepravilnosti v zgodovini, ki so izvirale iz načina, kako git-svn pretvori SVN-strukturo v Git. Zato smo preneseno zgodovino dodatno uredili. Ta faza je bila najzahtevnejši del selitve zgodovine in glavni razlog, da je ta trajala več časa. Kljub temu je bil ta korak nujen, da smo zagotovili Git-repozitorij s pregledno zgodovino, ki ohranja dejanski potek razvoja. V nadaljevanju opisujemo najpomembnejše korake, ki smo jih izvedli za pridobitev urejene zgodovine prenesenega repozitorija v Gitu.

Premik oznak na ustrezne commite

V SVN-ju so oznake mape, ki vsebujejo kopijo vsebine iz določene revizije repozitorija. Oznako se ustvari s commitom, ki zabeleži dejanje kopiranja. Pri prenosu v Git se to izrazi kot samostojen commit, ki v resnici ne vsebuje nobenih sprememb – tako imenovani prazen commit (angl. empty commit).

V Gitu pa oznake niso mape ali commiti, temveč kazalci (angl. pointers) na že obstoječe commite. Zato je oznake primerneje usmeriti neposredno na commit, ki odraža dejansko vsebino določene prelomnice – torej tisti commit,

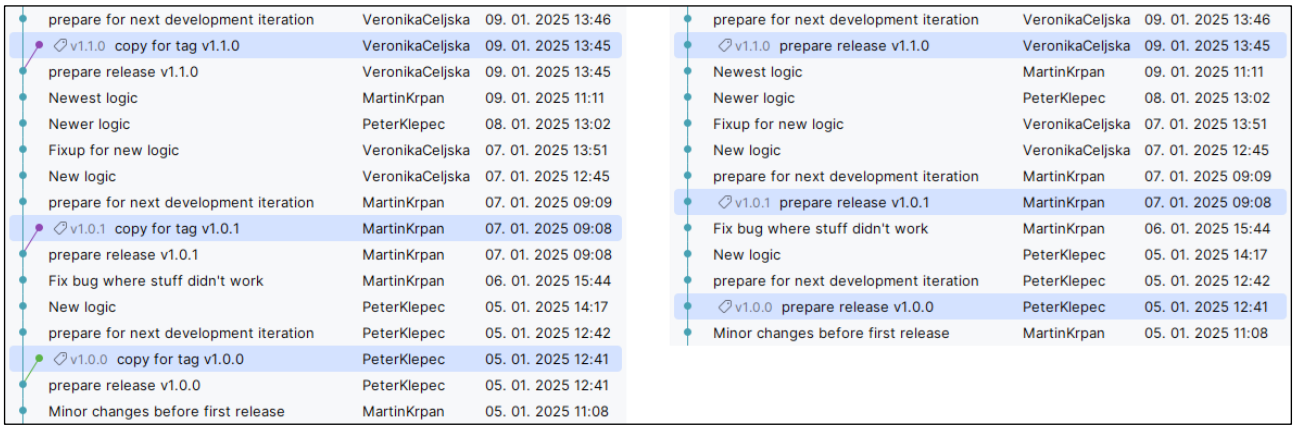
iz katerega je bila v SVN-ju ustvarjena kopija. Če oznake ostanejo na omenjenih umetnih praznih commitih, graf repozitorija v Gitu postane nepregleden, saj so oznake prikazane na stranskih vejah, ločeno od glavne zgodovine. S premikom oznak na pravilnejši commit izboljšamo berljivost grafa in ohranimo logičen potek zgodovine.

Naši projekti so zaradi daljše zgodovine vsebovali večje število oznak, zato smo za njihovo premikanje pripravili funkcijo v Pythonu.

```
1  import subprocess
2
3
4  def move_tags_to_nth_parent_commit(tags, n):
5      for tag in tags:
6          nth_parent_commit_hash = subprocess.check_output(
7              ["git", "rev-list", f"--skip={n}", "-1", tag], text=True
8          ).strip()
9          subprocess.run(["git", "tag", "-f", tag, nth_parent_commit_hash], check=True)
```

Funkcija prejme seznam oznak za premik in število *n*, ki predstavlja globino predhodnega commita, na katerega je potrebno prestaviti vsako podano oznako. Možnost poljubne globine za določen seznam oznak smo podprli zato, ker je bilo v določenih primerih potrebno oznake prestaviti za več kot en predhodni commit, npr. če so bili prisotni dodatni prazni commiti, ki so nastali zaradi sprememb lokacije oznak v datotečni strukturi.

Na Sliki 1 je prikazan primer grafa repozitorija pred in po premiku oznak. Levo vidimo stanje pred premikom, kjer se oznake nahajajo na ločenih commitih, ki se vejijo iz glavne veje repozitorija. Desno pa je prikazano stanje po popravku. Oznake so premaknjene na primernejše commite na glavni veji, v tem primeru na njihove direktne predhodnike. Graf ima sedaj znižano kompleksnost in pravilnejši potek glede na način uporabe oznak v Gitu. V obeh primerih so za lažjo primerjavo označene vrstice commitov z oznakami.



Slika 1: Primerjava grafa repozitorija pred (levo) in po (desno) premiku oznak na ustrezne commite.

Obnovitev pravilnega poteka grafa zgodovine

Po začetnem prenosu zgodovine iz SVN-ja pogosto ni bila pravilno ohranjena struktura vejitev in združitev (angl. branching and merging). Pogosto se je namreč zgodilo, da določene združitve niso bile pravilno predstavljene z združitvenimi commiti, ali pa da veje niso izvirale iz commita, ki je predstavljal njihov pravi izhodiščni commit (angl. base commit) v SVN-ju. Zgodovina je bila posledično neustrezna za dolgoročno vzdrževanje in preglednost. Takšno stanje smo morali ročno popraviti, kar je vključilo rekonstrukcijo poteka vej z operacijo rebase in rekreacijo združitvenih commitov, kjer so bili ti napačno predstavljeni kot navadni commiti.

Takoj ko spremenimo potek grafa, moramo izvesti operacijo rebase še za vse commite od točke spremembe do aktualnega HEAD-commita. To velja tudi za veje in združitvene commite, ki so sicer pravilno zaznani. Ker so commiti v Gitu nespremenljivi, z operacijo rebase ustvarimo novo različico commita, če z operacijo spremenimo vsaj eno lastnost commita, kot je denimo njegov predhodni commit. Zaradi tega je izvajanje operacije rebase, še posebej globoko v zgodovini, lahko zelo zamudno, saj pomeni rekreiranje commitov, skupaj z vejitvami in

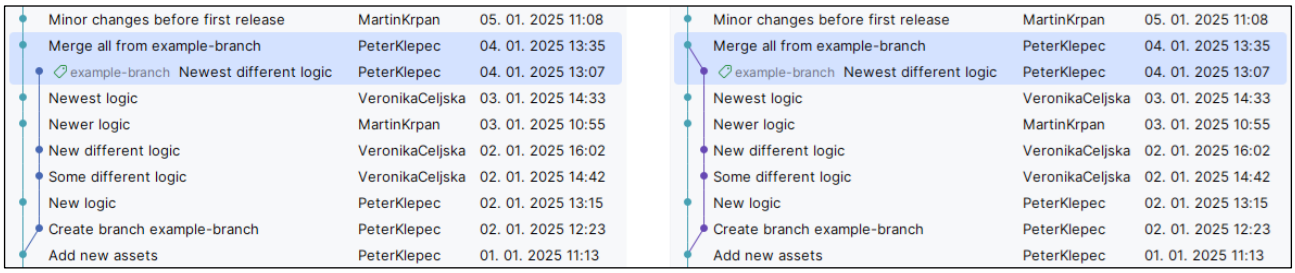
združitvami, na celotni poti do HEAD-commita. Če se na spremenjeni poti grafa nahajajo oznake, jih je potrebno po spremenjenem poteku grafa prestaviti na ustrezne nove commite, ki smo jih ustvarili z operacijo rebase, saj se lokacije oznak samodejno ne spreminjajo.

Operacijo rebase smo vedno izvajali z zastavico `--committer-date-is-author-date`, saj bi novi commiti v nasprotnem primeru za datum ustvarjanja commita imeli nastavljen datum, ko smo izvedli rebase. Na podoben način Git ločuje tudi med avtorjem commita in tistim, ki je commit dejansko izvedel (angl. committer). Slednji podatek se ob izvedbi operacije rebase za nove commite nastavi na vrednosti iz naših lokalnih nastavitev imena in elektronskega naslova za repozitorij. Za ohranjanje enakosti med avtorjem in committerjem ne potrebujemo skrbeti sproti, saj lahko to dosežemo za celotno zgodovino z uporabo git filter-repo, kar je tudi opisano kasneje v poglavju.

Nezaznani združitveni commiti

Ob prenosu zgodovine smo srečevali več ponavljajočih anomalij, ki smo jih morali odpravljati. V nadaljevanju pa podrobneje opisujemo napako, ki je bila po našem mnenju nabolj kritična Pri določenih združitvah git-svn ob prenosu ni zaznal, da gre za združitveni commit. Čeprav je stanje projekta pravilno ustrezalo rezultatu združitve, je bil commit za združitev predstavljen kot običajen linearen commit, kar pomeni, da se podrobnejša zgodovina združene veje na glavni veji ni ohranila. Commiti združene veje so bili sicer normalno preneseni z git-svn, a ti niso bili povezani s celoto. Rešitev je bila, da te združitvene commite ročno rekreiramo, nato pa vse nadaljnje commite z operacijo rebase prekopiramo na nov združitveni commit.

Nepravilno stanje združitve je prikazano na levi strani Slike 2. Na desni strani Slike 2 pa je prikazano stanje na grafu, ki smo ga želeli doseči z rekreacijo združitvenega commita. Razlika na samem grafu vizualno ni drastična, vendar pa je vsebinsko pomembna, da lahko na glavni veji pravilno prepoznamo dejanski izvorni commit posameznih sprememb preko ukazov, kot je `git blame`.



Slika 2: Primerjava grafa zgodovine pred (levo) in po (desno) rekreaciji združitvenega commita.

Pri rekreaciji združitvenega commita je bilo potrebno razrešiti tudi vse morebitne konflikte. Ker je vsak nepravilno zaznan združitveni commit, predstavljen kot navaden commit, vseboval pravilno stanje ob posamezni združitvi, je tako vseboval tudi stanje razrešenih konfliktov. Ker je nemogoče zagotavljati, da bi ob vsaki rekreaciji združitve pravilno razrešili konflikte, smo vsebino novega združitvenega commita z ukazom `git read-tree --reset -u` vedno nastavili na zgodovino prvotnega nepravilno zaznanega združitvenega commita. Ker je bila vsebina novega združitvenega commita s tem identična izvornemu linearnemu commitu, so bili vsi konflikti razrešeni na enak način, kot so bili razrešeni v prvotnem commitu. Pri ustvarjanju novega združitvenega commita pa smo prav tako morali vse metapodatke, kot so avtor, čas in sporočilo, nastaviti na vrednosti, ki so prisotne na prvotnem commitu. Pri tem smo uporabljali `git commit --amend`. Za avtomatizacijo opisanega postopka smo implementirali skript v Pythonu, ki glede na izvoren commit rekreira združitveni commit z enako vsebino in metapodatki na novi lokaciji.

```
1  import argparse
2  import os
3  import subprocess
4
5
6  def run_git_command(cmd, *, capture_output=False, text=True, env=None):
7      std = subprocess.PIPE if capture_output else None
8      return subprocess.run(["git"] + cmd, stdout=std, stderr=std, text=text, env=env, check=True)
```

```
 9
10
11  def get_git_output(cmd):
12      return subprocess.check_output(["git"] + cmd, text=True).strip()
13
14
15  def amend_commit(original_commit):
16      author = get_git_output(["log", "-1", "--pretty=%an <%ae>", original_commit])
17      date = get_git_output(["log", "-1", "--pretty=%ci", original_commit])
18
19      env = os.environ.copy()
20      env["GIT_COMMITTER_DATE"] = date
21      run_git_command(
22          ["commit", "--amend", "--no-edit", f"--author={author}", f"--date={date}"],
23          env=env
24      )
25
26
27  def recreate_merge(what_to_merge, where_to_merge, original_merge_commit, branch_to_reset=None):
28      # Use commit hash to avoid updating branch HEAD if 'where_to_merge' is a branch name
29      where_to_merge_hash = get_git_output(["rev-parse", where_to_merge])
30      run_git_command(["checkout", where_to_merge_hash])
31
32      # Merge but don't commit yet
33      run_git_command(["merge", "--no-commit", "--no-ff", what_to_merge])
34      # Replace working tree and index to match the original merge commit exactly
35      run_git_command(["read-tree", "--reset", "-u", original_merge_commit])
36
37      orig_commit_message = get_git_output(["log", "-1", "--pretty=%B", original_merge_commit])
38      run_git_command(["commit", "-m", orig_commit_message])
39      amend_commit(original_merge_commit)
40
41      # Reset working tree to clean state
42      run_git_command(["reset", "--hard", "HEAD"])
43
44      if branch_to_reset is not None:
45          run_git_command(["checkout", "-B", branch_to_reset])
46
47
48  if __name__ == "__main__":
49      parser = argparse.ArgumentParser(description="Recreate a merge commit at a new base.")
50
51      parser.add_argument("what", help="The branch or commit to merge (the 'merged' side)")
52      parser.add_argument("into", help="The branch or commit to merge into (the base)")
53      parser.add_argument("orig", help="The original merge commit to recreate")
54      parser.add_argument(
55          "--B", help="Branch name to update to the new merge result (optional)", default=None
56      )
57
58      args = parser.parse_args()
59      recreate_merge(args.what, args.into, args.orig, branch_to_reset=args.B)
```

Skript prejme tri parametre. S prvim parametrom povemo, kaj želimo združiti. To je v večini primerov HEAD-commit veje, ali pa ime veje, ki jo želimo združiti. Drug parameter pove, kam združujemo oz. kje želimo izvesti združitev. Ta commit predstavlja predhodnika združitvenega commita na veji, v katero združujemo željeno drugo vejo. Kot tretji parameter podamo referenco na obstoječ združitveni commit, iz katerega se bo črpalo vsebino sprememb in metapodatke commita. V danem primeru je to združitveni commit, ki je nepravilno predstavljen kot linearen commit. Skript prejme še dodaten opcijski parameter za ime veje, za katero se bo izvedlo njeno kreiranje ali posodobitev lokacije na novo ustvarjen združitveni commit.

Skript je sposoben rekreacije združitve tako iz linearnih commitov kot tudi iz pravih združitvenih commitov. Zaradi tega smo ga uporabljali tudi ob rekreaciji vseh pravilno zaznanih združitvenih commitov na poteh, za katere je bilo potrebno izvesti operacijo rebase. Operacija rebase sicer podpira rekreiranje združitev z uporabo zastavice `--rebase-merges`, vendar je bila slednja za nas neuporabna, saj moramo z njeno uporabo vse konflikte ob združevanjih vseeno razreševati ročno.

Premik oznak na nove commite

Ko smo z operacijo rebase zaključili urejanje celotne zgodovine, smo morali obstoječe oznake prestaviti na ustrezna mesta v novi strukturi repozitorija. V ta namen smo pripravili dve funkciji v Pythonu. Prva v datoteko JSON shrani podatek o trenutni lokaciji oznak, druga pa na podlagi te datoteke določi ustrezne nove commite, dostopne iz podane veje, na katere prestavi oznake.

```python
import json
import subprocess


def extract_svn_revision(commit_msg):
    return commit_msg.split("git-svn-id:")[1].split()[0].split("@")[-1]


def save_revisions_tags_json(json_save_path):
    tags = subprocess.check_output(["git", "tag"], text=True).splitlines()
    tags = [t.strip() for t in tags if t.strip()]

    revisions_tags = {}
    for t in tags:
        log_output = subprocess.check_output(["git", "log", "-1", t], text=True)
        revision = extract_svn_revision(log_output)
        revisions_tags[revision] = t

    with open(json_save_path, "w") as f:
        json.dump(revisions_tags, f, indent=4)


def tags_from_revisions_json(target_branch, revisions_json_file):
    subprocess.run(["git", "checkout", target_branch])

    with open(revisions_json_file) as f:
        revisions_tags = json.load(f)

    target_revisions = set(revisions_tags)
    found_revisions = set()

    git_log = subprocess.check_output(["git", "log"], text=True).split("commit ")
    git_log = [commit for commit in git_log if commit]

    for commit in git_log:
        svn_revision = extract_svn_revision(commit)
        if svn_revision not in target_revisions:
            continue

        commit_hash = commit.splitlines()[0]
        tag = revisions_tags[svn_revision]
        subprocess.run(["git", "tag", "-f", tag, commit_hash], check=True)

        found_revisions.add(svn_revision)
        if len(found_revisions) == len(target_revisions):
            break

    print("Done.")
```

Za podatek o lokaciji oznak smo izkoristili SVN-metapodatke o revizijah v sporočilih commitov, ki se ohranijo tudi v novih commitih, ustvarjenih z operacijo rebase. Če bi se odločili, da SVN-metapodatkov ob prenosu zgodovine z git-svn ne uporabimo, pa bi za enolično določanje commitov za lokacije oznak morali uporabiti drug nabor metapodatkov, kot je recimo avtor commita v kombinaciji s časom in sporočilom commita.

Odstranitev nepotrebnih vej

V SVN-ju so veje predstavljene kot mape, zato v zgodovini ostane podatek o veji tudi po tem, ko je bila njena mapa izbrisana. Posledično se pri prenosu v Git ohranijo vse veje, ki so kadar koli obstajale v razponu prenesenih revizij – tudi tiste, katerih mape v trenutnem stanju repozitorija ne obstajajo več. Zato smo po zaključku ureditve grafa repozitorija v Gitu izbrisali vse veje, ki jih nismo več potrebovali. S tem smo zagotovili, da se te ne prenašajo naprej in ne ustvarjajo zmede pri nadaljnjem razvoju.

Dodatna ureditev s pomočjo git filter-repo

S predhodnimi koraki smo zgodovino repozitorija že skoraj v celoti uredili. Obstaja pa še nekaj dodatnih korakov, ki smo jih po končanem urejanju izvedli z orodjem git filter-repo, da smo dosegli zares čisto zgodovino. Pri uporabi ukaza `git filter-repo` smo vedno uporabljali zastavico `--force`, saj orodje git filter-repo ne dopušča destruktivnih akcij v repozitorijih, za katere zazna, da niso sveži. V naših primerih smo zgodovino že ročno urejali, tako da ni šlo več za sveže klonirane repozitorije. Kljub temu smo pred vsako uporabo orodja ustvarili varnostno kopijo repozitorija, da nismo izgubili celotnega napredka v primeru, da z uporabljenim ukazom ne bi pridobili željenega rezultata.

Prazni commiti

Omenili smo že, da so oznake in veje v SVN-ju predstavljene kot mape. Kreira se jih tako, da se določeno revizijo kopira na potrebno mesto, kar se izvede z novim commitom. Ker v Gitu oznak in vej ne predstavljamo z mapami, so takšni commiti, kjer se zgolj ustvarja novo mesto v strukturi SVN-repozitorija, prisotni, a so vsebinsko prazni (angl. empty commits). Takšne commite za oznake smo že odstranili s premikom oznak. V repozitoriju pa so lahko še vedno prisotni drugi prazni commiti, kot so commiti za ustvarjanje vej. S spodnjim ukazom takšne commite odstranimo iz repozitorija.

```
git filter-repo --force --prune-empty always
```

Ukaz filter-repo bo poskrbel, da je potek grafa ohranjen, kljub temu, da so prazni commiti odstranjeni. Zaradi odsotnosti praznih commitov pa bodo vsi commiti, ki jim sledijo, v resnici novi commiti z drugo hash-vrednostjo, podobno kot če bi izvedli operacijo rebase. Prednost uporabe git filter-repo pa je, da se vse združitve, lokacije oznak in metapodatki ohranijo, ne da bi jih potrebovali dodatno urejati sami.

Enakost med avtorjem in committerjem

Git ločuje med avtorjem commita in tistim, ki je dejansko ustvaril commit (angl. committer). Ob izvedbi operacije rebase se lahko za novo nastale commite podatka o avtorju in committerju razlikujeta, saj je podatek o committerju nastavljen na lokalne podatke tistega, ki je izvedel rebase, medtem ko se vrednost avtorja ohrani. Takšni metapodatki pa ne predstavljajo realnega stanja prenesene zgodovine iz SVN, zato smo s pomočjo git filter-repo vrednosti za committerja nastavili nazaj na vrednosti avtorja za celoten repozitorij, pri čemer smo uporabili spodnji ukaz.

```
git filter-repo --force --commit-callback '
    commit.committer_name = commit.author_name;
    commit.committer_email = commit.author_email;'
```

Manipulacija datotečne strukture projekta

Ker gre pri projektih, ki smo jih prenašali na Git, za starejše projekte, so se v preteklosti zaradi takratnih tehnoloških omejitev uporabljale prakse, ki danes niso več aktualne. Binarne datoteke, potrebne za delovanje aplikacije, se je tako dodajalo v repozitorij, njihovo vsebino pa se je skupaj z razvojem aplikacije tudi spreminjalo. Git spremembe binarnih datotek obravnava tako, da ob vsaki spremembi shrani celotno vsebino datoteke. To lahko privede do visoke velikosti repozitorija na disku, kar otežuje vsakodnevno delo razvijalcev, saj imamo v Gitu celotno zgodovino repozitorija preneseno lokalno. Zaradi tega smo se odločili, da sledenje takšnim datotekam v Gitu odstranimo skozi celotno zgodovino. Spodaj sta primera ukazov, ki jih lahko uporabimo za odstranitev datoteke ali direktorija iz zgodovine.

```
# Odstranitev datoteke
git filter-repo --force --path path/to/file --invert-paths
# Odstranitev direktorija
git filter-repo --force --path path/to/dir/ --invert-paths
```

Orodje git filter-repo bo sledenje podani datoteki ali direktoriju odstranilo skozi celotno zgodovino, hkrati pa se bodo odstranili tudi vsi commiti, ki so se nanašali izključno samo na to datoteko ali direktorij.

Ker smo določene binarne datoteke odstranili iz zgodovine, smo lahko prilagodili tudi datotečno strukturo projekta, da je bila ta primerna za posodobljen nabor datotek. Z orodjem git filter-repo smo tako določenim direktorijem spremenili lokacijo znotraj projekta. Naslednja primera ukazov prikazujeta dve možnosti premika – prvi primer predstavlja premik vsebine mape na popolnoma novo pot, drug primer pa predstavlja premik vsebine mape na korensko pot repozitorija.

```
# Premik na novo mesto
git filter-repo --force --path path/to/dir/ --path-rename path/to/dir/:new/path/to/dir/
# Premik na koren repozitorija
git filter-repo --force --path path/to/dir/ --path-rename path/to/dir/:
```

3.5. Objava oddaljenega Git-repozitorija

S predhodnimi koraki smo dosegli čisto zgodovino repozitorija s pravilnim potekom grafa. Takšen repozitorij smo nato objavili na interni platformi za gostovanje Git-repozitorijev. Na platformi najprej ustvarimo nov prazen repozitorij, v lokalnem repozitoriju pa zanj dodamo novo referenco oddaljenega repozitorija. Na dodan oddaljen repozitorij izvedemo push vseh vej in vseh oznak.

```
1  git remote add origin <new-remote-repo-url>
2  git push origin --all
3  git push origin --tags
```

Oddaljen repozitorij lahko ustvarimo tudi, če zaenkrat razvojnega procesa še nimamo namena v celoti preseliti na Git. Zgodovino nastalega repozitorija lahko namreč do končne migracije sproti posodabljamo, hkrati pa nam bo nov repozitorij omogočal testiranje vseh CI/CD procesov, ki jih moramo posodobiti, da ti pravilno delujejo v Gitu.

3.6. Posodabljanje prenesene zgodovine

Če po začetni selitvi zgodovine razvoj še vedno poteka v SVN-ju, lahko zgodovino v Gitu posodabljamo z novejšimi revizijami iz SVN-ja, ki so nastale po začetnem prenosu zgodovine. Tako zagotovimo, da bo repozitorij v Gitu ob prehodu že vseboval celotno zgodovino, ne da bi bilo potrebno čakati še na njeno dokončno pripravo.

Če bi datotečna struktura repozitorija ostala neobdelana, bi lahko repozitorij posodabljali kar preko orodja git-svn. Pogoj za to je sicer, da sporočila v commitih vsebujejo potrebne metapodatke o številki revizije posameznega commita in da lokalne veje v Gitu sledijo oddaljenim vejam v SVN-repozitoriju. Ker pa smo v našem primeru datotečno strukturo spreminjali, to ni bilo več mogoče, ker stanje projekta v Gitu ni bilo več skladno s stanjem projekta v SVN-ju. Zato smo zgodovino do časa prehoda na Git posodabljali tako, da smo delno prenašali novo nastalo zgodovino, jo uredili na enak način kot obstoječo zgodovino in jo po ureditvi dodali k skupni zgodovini.

Delno zgodovino smo z ukazom `git svn clone` prenesli tako, da smo z uporabo zastavice `-r` podali razpon revizij za prenos.

```
git svn clone -s -r <revision-number>:HEAD --authors-file=<authors-file-path> \
    <svn-repo-url> <cloned-repo-dir-name>
```

Za začetno revizijo smo izbrali takšno, ki je že prisotna v prvotni zgodovini, da smo lahko s tem primerjali vsebino commitov in se prepričali, da je nova zgodovina po njenem urejanju v pravilnem vsebinskem stanju. Za končno revizijo smo podali HEAD, torej smo zgodovino prenesli vse od podane do zadnje nastale revizije. Uporabili smo obstoječo datoteko avtorjev, ki smo jo ustvarili že ob prvotnem prenosu zgodovine. Pomembno je, da v takšnem primeru v datoteko dodamo vse morebitne avtorje, ki so od zadnjega prenosa naredili svoj prvi prispevek v določenem projektu in zato še niso bili prisotni v prvotni različici datoteke.

Ko je bila nova zgodovina urejena, smo jo vključili v obstoječ repozitorij s pomočjo začasne oddaljene povezave na lokalni repozitorij z novo zgodovino. Nato smo pridobili oznake in vse relevantne veje, za slednje pa ustvarili

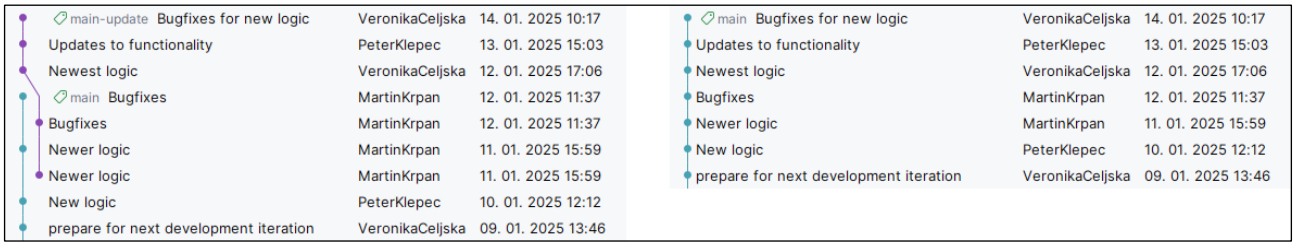
še lokalne reference. Po kreiranju lokalnih vej smo referenco oddaljene povezave odstranili, ker je nismo več potrebovali.

```
git remote add update <local-repo-with-new-history>
git fetch update main --tags
git checkout -b main-update update/main
git remote rm update
```

Zgornji nabor ukazov prikazuje ustvarjanje lokalne veje samo za nove commite na glavni veji, ki je v našem primeru poimenovana »main«. Če so v novi zgodovini nastale nove veje, ali pa so bili dodani novi commiti na obstoječe veje, moramo ustvariti lokalne reference tudi za takšne veje.

Novo ustvarjene veje bodo v obstoječem repozitoriju prikazane kot osamele veje (angl. orphan branches), saj zaradi delnega prenosa nimajo skupnega izhodišča (angl. base) z obstoječo zgodovino, temveč je izhodišče commit za tisto revizijo, od katere smo prenesli delno zgodovino. Nove commite dodamo k obstoječi zgodovini tako, da izvedemo rebase. Če smo prenesli tudi takšne commite, ki so v obstoječi zgodovini že prisotni, pazimo, da takšnih commitov ob izvedbi operacije rebase ne ohranjamo.

Slika 3 na levi strani prikazuje graf repozitorija, kjer je v repozitorij že dodana veja z novo zgodovino kot osamela veja. Ta osamela veja vsebuje tudi dva commita, ki sta na obstoječi veji že prisotna. Na desni strani pa je prikazan graf repozitorija, kjer se je z operacijo rebase commite iz nove veje dodalo na obstoječo vejo. Z operacijo rebase so bili dodani zgolj trije commiti, ki na obstoječi veji predhodno niso obstajali.



Slika 3: Dodajanje nove zgodovine k obstoječi zgodovini.

4 Priprava delovnega procesa

Selitev na Git smo izkoristili kot priložnost za prenovo razvojnega procesa. Ker Git temelji na drugačnih konceptih kot SVN, smo si prizadevali zasnovati delovni tok, ki izkorišča prednosti novega sistema, namesto da bi vanj preslikali stare prakse. Če Git obravnavamo na enak način kot SVN, hitro izgubimo njegove ključne prednosti in s tem tudi smisel same selitve. Pri snovanju novega delovnega procesa smo se zgledovali po uveljavljenih tokovih za Git, kot so GitFlow [8], GitHub Flow [9], GitLab Flow [10] in trunk-based development [11]. Te pristope smo nato prilagodili lastnim potrebam in značilnostim projektov. Ključno vodilo je bilo, da izkoristimo lahkotnost in fleksibilnost, ki jo Git ponuja pri delu z vejami, hkrati pa izkoriščamo njegove naprednejše možnosti, kot je operacija rebase, kjer je to smiselno ali mogoče.

4.1. Uporaba zahtev za združitev za integracijo pregleda kode

Najpomembnejša razlika med Gitom in SVN-jem je izredno učinkovito in hitro upravljanje z vejami. V Gitu ustvarjanje, preklapljanje in združevanje vej predstavlja osnovo vsakodnevnega dela, medtem ko je bilo v SVN-ju to pogosto nepraktično in rezervirano za večje spremembe. Ta lastnost je upoštevana v večini najpogosteje uporabljanih delovnih tokovih, saj omogoča izoliran razvoj funkcionalnosti in popravkov, večjo preglednost sprememb in lažje sodelovanje med člani ekipe.

Večina sodobnih platform za Git, kot so GitHub, GitLab in Bitbucket, dodatno omogočajo zahteve za združitev sprememb iz ene veje v drugo vejo (angl. pull requests ali merge requests). V sklopu zahteve za združitev je možno pregledati vse spremembe, ki bi se z združitvijo dodale, pred samo združitvijo pa zahtevati popravke, ali pa zahtevo celo zavrniti. To nam omogoča neposreden pregled kode ob vsaki zahtevi za združitev. Ker smo se odločili, da naš delovni tok osnujemo glede na že uveljavljene tokove, smo s tem tudi začeli uporabljati ločene veje za razvoj sprememb. Njihovo združevanje v glavno vejo smo izvajali izključno preko zahtev za združitev.

Z začetkom uporabe zahtev za združitev smo uvedli tudi sistemsko zahtevo po pregledu vsake spremembe pred vključitvijo v glavno vejo razvoja. Pregledi kode tako niso bili več prepuščeni dosledni disciplini sprotnega pregledovanja novih revizij, temveč so postali obvezen in naravno vgrajen korak v razvojnem ciklu. S tem smo povečali robustnost procesa, zmanjšali možnost spregledanih napak ter izboljšali skupno kakovost kode.

4.2. Definicija delovnega toka in CI/CD procesa

V uvodnem delu prispevka smo že omenili, da je bila selitev projektov iz platforme A na platformo B pomembna prelomnica, ob kateri smo se zaradi praktičnih razlogov odločili tudi za selitev na Git. Glavni razlog je bil v tem, da smo morali v prehodnem obdobju istočasno podpirati delovanje aplikacij na obeh platformah. Nove funkcionalnosti so se razvijale na platformi A, saj so aplikacije še vedno delovale na njej, a so morale biti hkrati pripravljene tudi za delovanje na platformi B.

Prenova delovnega toka pa je zahtevala tudi prilagoditev CI/CD-procesa. Že v času SVN-ja smo uporabljali avtomatizirano gradnjo in nameščanje, zato smo želeli to prakso ohraniti in nadgraditi. Ker smo vedeli, da bo podpora platforme A v prihodnosti ukinjena, smo se odločili, da CI/CD proces za platformo B v celoti prenovimo, medtem ko za platformo A ni bilo smiselno uvajati večjih sprememb. V Gitu smo to lahko bistveno lažje izvedli z uporabo ločenih razvojnih kopij repozitorija za vsako platformo posebej. Tako je imel vsak repozitorij svoj lasten CI/CD proces, popolnoma ločen in prilagojen specifičnim zahtevam platforme, ne da bi en proces vplival na drugega.

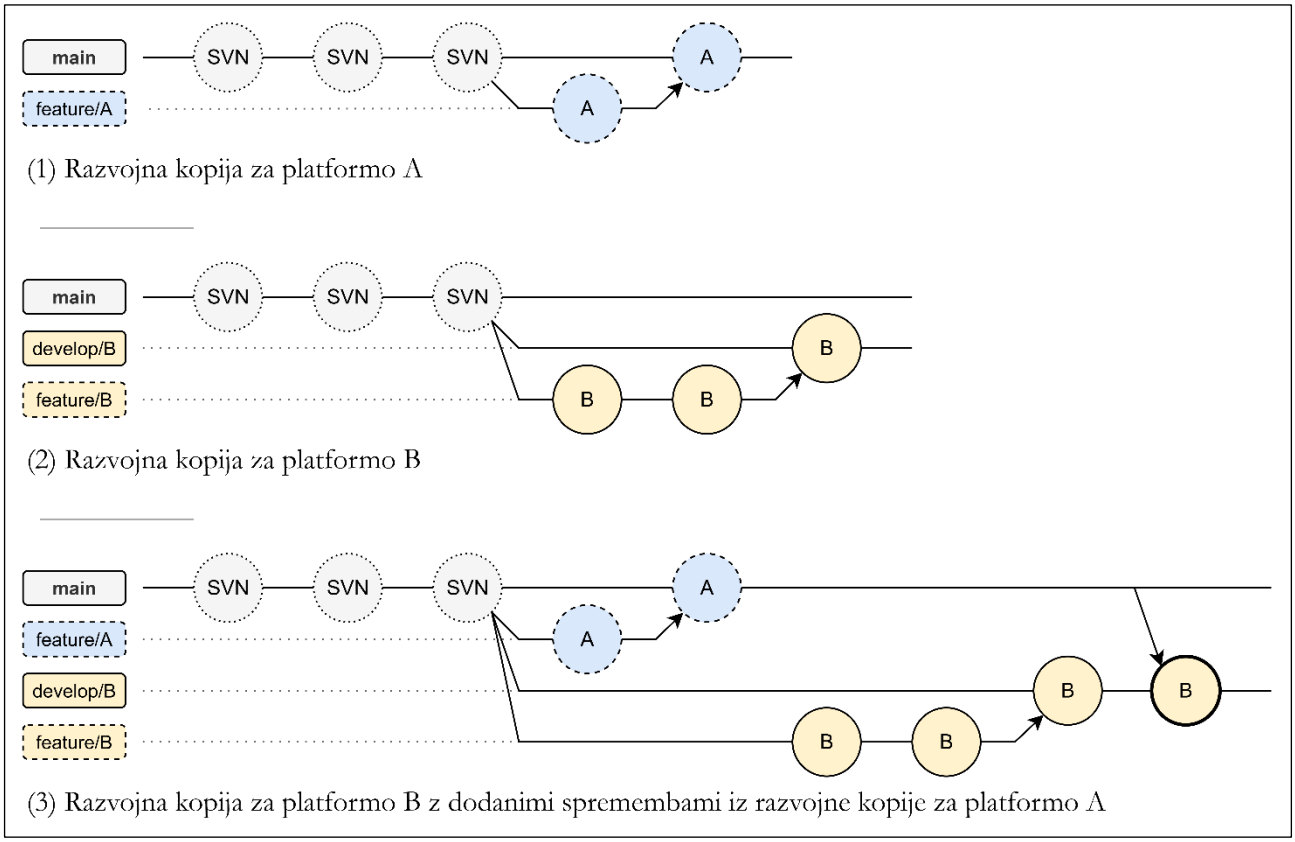
Za platformo A smo CI/CD proces prilagodili zgolj minimalno – toliko, da je ta pravilno deloval z Git-repozitorijem. Za delo z Gitom smo uporabili delovni tok, ki temelji na pristopu, imenovanem trunk-based development, podobno kot pri delu s SVN. Kljub temu smo v Gitu uvajali ločene veje za posamezne funkcionalnosti, ki smo jih nato dodajali v glavno vejo s pomočjo zahtev za združitev. Na ta način smo obdržali prednosti, ki jih Git prinaša, kot je vgrajen pregled kode, ne da bi pretirano spreminjali obstoječ proces razvoja na platformi A. Na ta način smo omogočili lažji prehod z minimalnim vložkom, saj platforma A dolgoročno ni bila predvidena za razvoj.

Za platformo B pa smo uvedli bolj strukturiran delovni tok, ki temelji na pristopu GitFlow. Platforma B je imela svojo ločeno razvojno vejo, v katero smo dodajali tehnične spremembe, potrebne za pravilno delovanje aplikacij na novi platformi. Poslovna logika se je še vedno v celoti razvijala na platformi A, deljena izvorna zgodovina obeh kopij repozitorijev pa nam je omogočila, da smo spremembe poslovne logike periodično sinhronizirali s tehničnimi spremembami za platformo B. CI/CD proces smo za platformo B vzpostavili povsem na novo, skladno z izhodišči pripravljenega delovnega toka. Postavili smo novo strukturo cevovodov in poskrbeli za testiranje vseh ključnih funkcionalnosti.

Tak pristop nam je omogočil stabilno in pregledno upravljanje sprememb v obeh platformah, hkrati pa smo lahko CI/CD procese za posamezno platformo neodvisno razvijali, vzdrževali in testirali. Tovrstna prilagodljivost razvojnih tokov in CI/CD procesov bi bila bistveno težje izvedljiva v SVN-ju, kjer delo z več vejami in več repozitoriji ni podprto na enako učinkovit način kot v Gitu. Primer opisanega postopka prav tako ponazoruje Slika 4, ki predstavlja tri dele postopka:

- pod (1) je prikazana razvojna kopija repozitorija za platformo A,
- pod (2) razvojna kopija repozitorija za platformo B,

- pod (3) pa je prikazan prenos sprememb iz razvojne kopije za platformo A v razvojno kopijo za platformo B.



Slika 4: Hkratni razvoj za obe platformi v ločenih kopijah repozitorija.

4.3. Testiranje CI/CD procesa

Ker je bil CI/CD proces pomemben del prehoda na Git in hkrati ključen za zanesljiv potek razvoja, smo ga morali ustrezno preveriti, še preden smo omogočili dejanski razvoj v Gitu. V ta namen smo pripravili testne kopije repozitorijev za platformi A in B, s katerimi smo simulirali običajen potek razvoja, objavljanja in izdaje novih različic glede na posamezen delovni tok.

Testni repozitoriji so bili pripravljeni z namenom omogočanja izvajanja vseh CI/CD korakov, ne da bi s tem vplivali na produkcijsko kodo. Vanje smo lahko dodajali poljubne spremembe, ki so sprožile cevovode, kar nam je omogočilo preizkus vseh ključnih točk v procesu tako za platformo A kot tudi platformo B.

S predhodnim testiranjem smo zmanjšali tveganje za napake pri dejanskem prehodu, saj so bili vsi postopki testirani v okolju, ki je posnemalo dejansko uporabo. CI/CD procesi so bili tako v trenutku prehoda pripravljeni na uporabo in integrirani v prenovljen delovni tok. To nam je omogočilo gladek prehod na Git brez zastojev v razvoju in brez potrebe po dodatnem usklajevanju po začetku dela v novem sistemu.

5 Zaključek

Migracija iz SVN na Git je bila za naše projekte več kot le tehnična posodobitev – šlo je za strateško odločitev, ki je vplivala na razvojne procese, orodja in sodelovanje v ekipi. Postopek selitve je bil zaradi zgodovine, ki je zajemala več let razvoja in veliko število revizij, tehnično zahteven. Kljub temu smo s skrbnim načrtovanjem uspeli selitev delovnega procesa izvesti z ohranitvijo celotne zgodovine projektov in postavili temelje za učinkovit razvojni proces.

V prispevku smo opisali celoten postopek selitve zgodovine – od začetnega prenosa s pomočjo orodja git-svn, urejanja avtorjev in oznak, do obsežnih korakov za rekonstrukcijo logične strukture repozitorija, ki je bila zaradi razlik med sistemoma SVN in Git pogosto neustrezno interpretirana. Pri tem smo uporabili lastne skripte, s katerimi smo poskrbeli za avtomatizacijo ponavljajočih se nalog. Migracijo zgodovine smo izvedli dovolj zgodaj v procesu, saj gre za časovno najzahtevnejši korak, ki pa ne ovira nadaljnjega razvoja v SVN-ju. To nam je omogočilo vzporedno pripravo novih CI/CD procesov, postavitev delovnega toka in testiranje na dejanskem repozitoriju, še preden je bil izveden dejanski prehod razvojnega procesa.

Z uporabo Gita smo uvedli prenovljen delovni tok, ki temelji na preizkušenih praksah. Uvedli smo izolirano delo na funkcionalnih vejah in integrirane preglede kode preko obveznih zahtev za združitev. S tem smo povečali kakovost kode, zmanjšali možnost napak in omogočili večjo odgovornost posameznikov znotraj ekipe. Poleg tega smo lahko Git z njegovo fleksibilnostjo in distribuirano naravo učinkovito izkoristili tudi za vzporedno vzdrževanje dveh aplikacijskih platform, ki sta imeli povsem ločena CI/CD procesa in razvojna poteka, ne da bi pri tem ogrožali stabilnost drug drugega.

S prehodom na Git smo vzpostavili temelje, ki omogočajo dolgoročno razširljivost in večjo agilnost ekipe. Git zaradi svoje razširjenosti in priljubljenosti omogoča lažje uvajanje sodobnih orodij in pristopov, ki se skupaj s tehnologijami nenehno razvijajo. Fleksibilnost Gita pomeni, da lahko sproti prilagajamo delovni tok glede na nove potrebe projekta in ekipe, pri tem pa ne potrebujemo spreminjati osnovnega orodja ali strukture repozitorija. Prehod se je tako izkazal kot pomembna naložba v prihodnost projektov – izboljšal je preglednost, omogočil boljšo integracijo z drugimi orodji, olajšal uvajanje novih razvijalcev ter poenotil način dela z ostalimi ekipami v podjetju. Mlajši kader, ki že pozna delo z Gitom, se po novem lažje vključuje v obstoječe procese na teh projektih, kar zmanjšuje stroške uvajanja in omogoča hitrejši začetek produktivnega dela.

S prispevkom smo delili izkušnje, ki smo jih pridobili med selitvijo na Git, glede na dosežen rezultat pa bi selitev priporočili tudi drugim ekipam, ki še vedno uporabljajo SVN ali druge starejše rešitve. Čeprav je selitev lahko zahtevna, se njen doprinos hitro obrestuje – tako skozi izboljšano kakovost razvoja kot tudi skozi večjo preglednost, avtomatizacijo in podporo sodobnim pristopom razvoja programske opreme.

Literatura

[1] git-scm.com/book/ms/v2/Getting-Started-A-Short-History-of-Git, A Short History of Git, obiskano 26. 6. 2025.

[2] git-scm.com/docs/git-svn, git-svn Documentation, obiskano 30. 6. 2025.

[3] github.com/nirvdrum/svn2git, svn2git: Ruby tool for importing existing svn projects into git., obiskano 30. 6. 2025.

[4] subgit.com, SVN to Git Migration – TMate SubGit, 30. 6. 2025.

[5] github.com/newren/git-filter-repo, git filter-repo, obiskano 30. 6. 2025.

[6] www.jetbrains.com/idea, IntelliJ IDEA – the IDE for Pro Java and Kotlin Development, obiskano 10. 7. 2025.

[7] git-scm.com/book/en/v2/Appendix-A%3A-Git-in-Other-Environments-Git-in-IntelliJ-/-PyCharm-/-WebStorm-/-PhpStorm-/-RubyMine, Git in Intellij / PyCharm / WebStorm / PhpStorm / RubyMine, obiskano 10. 7. 2025.

[8] nvie.com/posts/a-successful-git-branching-model, A successful Git branching model, obiskano 11. 7. 2025.

[9] docs.github.com/en/get-started/using-github/github-flow, GitHub flow, obiskano 11. 7. 2025.

[10] about.gitlab.com/topics/version-control/what-is-gitlab-flow, What is GitLab Flow?, obiskano 11. 7. 2025.

[11] trunkbaseddevelopment.com, Trunk Based Development, obiskano 11. 7. 2025.