Univerza v Ljubljani
Fakulteta za računalništvo in informatiko

Jan Keber

Sistemi za nadzor različic za strukturirane datoteke

diplomsko delo

visokošolski strokovni študijski program prve stopnje računalništvo in informatika

Mentor: dr. Andrej Brodnik

Ljubljana, 2017

Besedilo je oblikovano z urejevalnikom besedil LaTeX.

Fakulteta za računalništvo in informatiko izdaja naslednjo nalogo:

Tematika naloge:

Ena storitev, ki jo sodobne arhitekture ponujajo je souporaba različnih datotek. Zgodovinsko gledano je slednje zelo podobno souporabi iste programske kode v razvoju programske opreme. Najbolj temeljna razlika je v tem, da so bile datoteke v razvoju programske opreme besedilne, medtem ko so dandanes datoteke veliko bogateje strukturirane. V diplomski nalogi naredite pregled orodij za souporabo datotek v procesu razvoja programske opreme. Nato preučite možnost, kako lahko te sisteme nadgradimo, da bomo dobili podobne funkcionalnosti ne samo za besedilne datoteke, ampak tudi za drugače strukturirane datoteke. Vse skupaj pilotno implementirajte za primer datotek odt (Open Document Format for Office Applications).

Zahvaljujem se mentorju dr. Andreju Brodniku za pomoč pri izdelavi diplomskega dela. Posebno zahvalo za vso izkazano podporo tekom študija pa namenjam staršem, vsem ostalim domačim, prijateljem in moji Piji.

Kazalo

Seznam uporabljenih kratic

kratica	angleško	slovensko
ODT	OpenDocument Text	odprtokodni besedilni dokument
CVS	Concurrent Versioning System	sistem za sočasen nadzor različic
RCS	Revision Control System	sistem za upravljanje z različicami
SVN	Subversion	sistem za nadzor različic
XML	Extensible Markup Language	razširljivi označevalni jezik
HTML	Hyper Text Markup Language	jezik za označevanje hiperbesedila

Povzetek

Naslov: Sistemi za nadzor različic za strukturirane datoteke

Povzetek: Namen diplomskega dela je nadgraditi sistem za nadzor različic. Cilj nadgradnje je možnost primerjave različic datotek zapisanih v strukturiranem zapisu. Na kratko so predstavljene posplošene ključne funkcionalnosti sistemov, njihov razvoj skozi čas in delitev sistemov na podskupine. V diplomskem delu je prikazan postopek primerjave OpemDocument Text dokumentov. Razložen je postopek primerjave besedilnega dela dokumenta, v nadaljevanju pa je podan primer primerjave besedila vključno z njegovo obliko. Predstavljen je še postopek, kako na zelo preprost način povezati svoj program za primerjavo datotek s sistemom za nadzor različic.

Ključne besede: razlike, primerjava, strukturirane datoteke, različice.

Abstract

Title: Version control systems for structured files

The aim of this thesis is to upgrade a version control system. The goal of the upgrade is to compare versions of files stored in a structured format. Some of the key functionalities of version control systems, development of systems over time and basic groups are described. The thesis contains step by step explanation and practical examples of OpenDocument Text file comparison. The process that compares textual part of document is explained for starter. Later, text formatting is also included in compare process. Thesis also includes an example of how very simple it is to use your own program for file comparing with version control systems.

Keywords: diff, comparison, structured documents, versions.

Poglavje 1

Uvod

Spremljanje sprememb, ki jih naredimo tekom razvoja obsežnejše programske opreme, je brez uporabe sistemov za nadzor različic zelo težavno. Ti sistemi nam med drugim omogočajo pregled nad spremembami vsake datoteke vse od njenega nastanka.

Zakaj bi sploh želeli imeti evidentirane vse spremembe, ki so bile narejene tekom razvoja? Eden izmed razlogov je, da prihaja do sprememb, ki problemov ne rešujejo ampak jih povzročajo. V takšnem primeru lahko razvijalec brez težav s pomočjo sistema iz shrambe pridobi delujočo različico, ki še nima vključenega slabega popravka. V primeru, ko želi spremembo pred njeno potrditvijo preveriti, da ne bi prišlo do neželjenih napak, mu sistem omogoča pregled sprememb med njegovo različico in zadnjo potrjeno.

Problem nastane, ko sprememba ni narejena nad besedilno, ampak nad binarno ali drugače strukturirano datoteko. Orodja, ki v sistemih za nadzor različic omgočajo pregled razlik, ponavadi ne znajo razlikovati takih datotek. Cilj diplomske naloge je opis vseh potrebnih korakov, podprtih s praktičnimi primeri, da lahko v poljubnem sistemu za nadzor različic izvedemo primerjavo dveh strukturiranih binarnih datotek.

1.1 Struktura naloge

V uvodnem delu je na kratko predstavljen namen uporabe sistemov za nadzor različic in cilj, ki ga želimo z izdelavo diplomske naloge doseči. V drugem poglavju bodo posplošeno predstavljene ključne funkcionalnosti sistemov. V začetku tretjega poglavja bodo na kratko predstavljene tri glavne skupine sistemov za nadzor različic, ki so se skozi čas oblikovale, in ključne lastnosti, ki jih razlikujejo. V nadaljevanju bo predstavljenih nekaj sistemov, ki so svojim nastankom zaznamovali zgodovino. V četrtem poglavju bo opisan celoten postopek primerjave strukturiranih datotek na besedilnem in oblikovnem nivoju. Opisan bo postopek povezovanje svojega programa za primerjanje v sistem za nadzor različic SVN. Na koncu, pri sklepnih ugotovitvah, bodo predstavljene funkcionalnosti, ki tekom diplomskega dela niso bile dokončno izpopolnjene. Opisani bodo tudi načini, kako je mogoče sisteme še dodatno izboljšati.

Poglavje 2

Osnovne operacije

Vsi sistemi za nadzor različic pokrivajo približno enak nabor osnovnih operacij. V nadaljevanju je predstavljena večina operacij, ki so del procesa dela z novodobnim sistemom za nadzor različic.

2.1 Delo s shrambo

Ustvarjanje nove, prazne shrambe je možno narediti z ukazom **ustvari** (ang. *create*). Shramba je prostor, kjer je shranjeno vse naše delo. Kar jo razlikuje od mrežnega datotečnega sistema je to, da je v njej shranjena celotna zgodovina sprememb datotek.

Posledica tega je, da se nobena sprememba nikoli ne izgubi. Vsakič, ko se zgodi sprememba v shrambi (tudi, če nekaj izbrišemo), ta postaja večja, saj je tudi zgodovina sprememb daljša. Vsaka sprememba se v zgodovino vedno dodaja in iz nje ne moremo nikoli ničesar izbrisati. Operacija **ustvari** je ena izmed prvih uporabljenih operacij. Od nas ponavadi zahteva, da izberemo mesto, kjer naj se ustvarjena shramba nahaja, in kako naj se imenuje.

Izvoz delovne kopije shrambe, ki že obstaja, je možno narediti s pomočjo ukaza **izvozi** (ang. *checkout*). Izvoženo kopijo sestavlja posnetek stanja shrambe, ki je uporabljena s strani razvijalca kot prostor, kjer lahko dela spremembe. Shramba je deljena s celotno razvojno ekipo, ampak člani ne delajo sprememb direktno na njej temveč nad svojo delovno lokalno kopijo. Ideja delovne kopije je, da razvijalca izolira od ostalih in mu daje občutek da delo opravlja sam, nemoteno.

Posodobitev služi za posodabljanje delovne kopije glede na trenutno stanje shrambe. V primeru, da so bile na delovni kopiji, nad katero izvajamo operacijo **posodobi** (ang. *update*), narejene spremembe, ki še niso potrjene v shrambi, sistem poskrbi za združevanje naših sprememb s pridobljenimi. Z uporabo te operacije lahko pride do neskladja (ang. *conflict*). To se zgodi takrat, ko nekdo drug naredi in potrdi spremembe na istem delu kode kot mi, sistem pa jih ne zna združiti, saj ne mora določiti čigava sprememba je pomembnejša.

Potrjevanje služi za posredovanje sprememb, narejenih nad lokalno kopijo, v shrambo. To je operacija, ki v shrambi ustvari novo različico datoteke. Večina modernih sistemov za nadzor različic opravlja to operacijo atomarno (ang. *atomic commit*). Z drugimi besedami to pomeni, da ne glede na to, koliko individualnih sprememb želimo narediti, bodo v shrambi pristale vse (v primeru, ko je operacija uspešna) ali pa nobena (v primeru, ko je operacija neuspešna).

Pri takih sistemih je nemogoče, da pride do stanja, kjer so operacije polovično zaključene. Tipično je, da ob izvajanju operacije **potrdi** (ang. *commit*) podamo sporočilo ali komentar, ki na kratko opiše spremembe, ki smo jih naredili. Sporočilo postane del zgodovine repozitorija.

Zgodovina sprememb različic, ki so kadarkoli obstajale so zabeležene v shrambi. Operacija **prikaz zgodovine** (ang. *log*) nam omogoča prikaz teh zapisov. Služi tudi za prikaz dodatnih informacij, ki so del posamezne spremembe kot so:

- avtor spremembe,
- čas spremembe in
- sporočilo, ki je bilo podano ob spremembi.

Večina sistemov za nadzor različic nam omogoča iskanje po zgodovini sprememb s pomočjo te operacije (npr. prikaz vseh sprememb, ki jih je naredil uporabnik Marko od leta 2015 dalje).

2.2 Delo z datotekami

Spreminjanje datoteke je najpogostejša operacija pri uporabi sistemov za nadzor različic. Ko si izvozimo lokalno kopijo, ta ponavadi vsebuje ogromno datotek iz shrambe. Te datoteke spreminjamo v pričakovanju, da bodo v prihodnosti postale del shrambe. Operacija spreminjanja ni orodje ali akcija v sistemih za nadzor različic.

Gre za preprost proces spreminjanja datoteke, ki je v večini dosežen s pomočjo poljubnega urejevalnika besedila (Notepad, Vim, Sublime,...) ali razvojnega okolja (Eclipse, NetBeans, Code::Blocks,...). Ko željeno datoteko spremenimo, sistem to spremembo zazna in datoteko doda na seznam sprememb, ki čakajo na potrditev.

Pri nekaterih sistemih je potrebno biti celo bolj natančen, saj vse datoteke v delovni kopiji označijo z značko „le za branje“. Preden želimo datoteko spremeniti, moramo to sporočiti sistemu, da željeno datoteko označi z možnostjo za zapisovanje.

Dodajanje in vključevanje nove datoteke, ustvarjene znotraj naše delovne kopije, v sistem za nadzor različic izvedemo s pomočjo operacije **dodaj** (ang. *add*). S to operacijo dodamo datoteko le na seznam sprememb, ki čakajo na potrditev, novo dodana datoteka pa bo v shrambo shranjena šele z operacijo **potrjevanja**.

Odstranjevanje uporabljamo, kadar želimo odstraniti datoteko ali imenik iz shrambe. Tipično bo operacija **odstranjevanja** (ang. *remove*) takoj odstranila delovno kopijo datoteke, zahteva za izbris datoteke iz shrambe pa bo dodana na seznam sprememb, ki čakajo na potrditev. Potrebno je vedeti, da datoteka iz shrambe nikoli ne bo resnično izbrisana. Ko potrdimo čakajoči seznam sprememb, ki vsebuje akcijo brisanja, se v repozitoriju vzpostavi nova različica drevesa, ki ne vsebuje izbrisane datoteke. Predhodnja različica drevesa, ki vsebuje izbrisano datoteko, pa ostane (kot neaktivna).

Preimenovanje se uporablja za spreminjanje imena datotek. Ob izvedbi ukaza **preimenuj** (ang. *rename*) se akcija doda na čakajoči seznam sprememb, datoteka, katere ime spreminjamo, pa je ponavadi v okviru delovne kopije preimenovana takoj. Nekateri (Bazaar, Veracity) imajo ime datoteke ali imenika predstavljen kar v obliki atributa, ki je mišljen, da se skozi čas spreminja.

Premikanje se uporablja, ko želimo premakniti datoteko ali imeniško strukturo iz enega mesta na drugega. Tudi ta akcija je dodana na čakajoči seznam sprememb, znotraj lokalne kopije pa je akcija izvedena takoj po ukazu. Nekatera orodja obravnavajo operaciji **preimenuj** in **premakni** (ang. *move*) enako (povzeto po Unix sistemih - absolutna pot datoteke je njeno ime), ostali pa ju obravnavajo ločeno (ime datoteke in imenik, v katerem se nahaja, sta ločena atributa).

Stanje sprememb, ki so bile narejene nad lokalno kopijo in dodane na seznam sprememb, lahko pogledamo z ukazom **pregled stanja** (ang. *status*). Operacijo si lahko razlagamo tudi tako, da nam **pregled stanja** prikaže spremembe, ki bi bile aplicirane v shrambo ob **potrjevanju**.

Prikazovanje razlik Operacija **pregled stanja** prikaže samo spremembe iz čakajočega seznama. Če želimo videti tudi podrobnosti o spremembah, ki so bile narjene na nivoju datotek, pa moramo uporabiti operacijo **prikazovanja razlik** (ang. textitdiff). Sistemi za nadzor različic imajo orodja za prikaz razlik implementirana na različne načine. Nekateri, ki delujejo v ukazni vrstici, ponavadi izpišejo razliko kar na standardni izhod, medtem ko nam ostali sistemi lahko prikažejo razlike v grafičnem vmesniku.

Razveljavitev sprememb (ang. *revert*) pride v poštev, ko naredimo spremembe nad lokalno kopijo, ki jih v nadaljevanju ne želimo obdržati. Lahko jo uporabljamo tudi, če ustvarimo napako, ali pa nam naše delo ni sprejemljivo in želimo spremembe zavreči. Izvedba operacije nad datoteko ali imenikom povzroči, da se spremembe odstranijo iz čakajočega seznama, lokalna kopija pa se posodobi na stanje, kakršno je bilo pred spremembami.

2.3 Pripomočki za vzporeden razvoj

Vejitev nam omogoča razcep razvojnega procesa na dve ločeni veji. V primeru, ko izdamo novo različico programa (V2.0), je smiselno ustvariti dve **veji** (ang. textitbranch). Prva bo namenjena za popravke izdane različice (V2.0x-popravki), druga pa za razvoj nove (V3.0).

Združevanje sledi operaciji **vejitve**, saj želimo ponavadi v neki točki razvojni proces, ki je bil ločen na dve veji, vsaj delno združiti. V primeru, ko smo naredili ločeno vejo za popravke (V2.0x-popravki), želimo da se popravki

upoštevajo tudi na glavni veji razvoja (tam, kjer je potekal razvoj različice V3.0).

Brez operacije **združevanje** (ang. textitmerge) bi bilo to mogoče izvesti s pripravo popravkov na obeh vejah. Z njeno pomočjo pa lahko narejene popravke iz specifične veje (V2.0x-popravki) karseda avtomatizirano združimo z glavno vejo razvoja (V3.0).

Reševanje neskladja (ang. *conflict*) po uporabi operacije **združevanje** je včasih potrebno s strani uporabnika. Operacija se namreč avtomatsko loti združevanja tam, kjer je to mogoče narediti na varen način, brez da bi povzročil škodo. V primeru, ko temu ni ugodeno, pride do t.i. konfikta. V primeru, ko datoteka `test.js`, ki je bila na eni veji spremenjena, na drugi pa odstranjena, se sistem ne more odločiti, katera veja je pomembnejša, in tako nastane neskladje. Operacija **reševanje** (ang. *resolve*) je namenjena za pomoč uporabniku pri reševanju tovrstnih neskladij.

Zaklepanje (ang. *lock*) je uporabno v primeru, ko želimo imeti samo mi pravico do urejanja določene datoteke. Ta funkcija ni del vseh sistemov za nadzor različic. Določeni sistemi zaklepanje podpirajo, s predpostavko, da bo le redko uporabljeno. Za datoteke, ki so preprostega besedilnega formata, je ponavadi najbolje prepustiti probleme vzporednosti kar sistemu za nadzor različic. Binarne datoteke, ki ne morejo biti avtomatsko združene, pa je v določenih primerih smiselno zakleniti.

Označevanje nam omogoča poimenovanje specifične različice v zgodovini repozitorija s smiselnim imenom. To lahko naredimo s pomočjo operacije **označevanja** (ang. *tag*).

Poglavje 3

Zgodovina in razvoj sistemov

Vsem, ki jih zanima poglobljeno znanje o sistemih za nadzor različic, svetujemo branje dela Moshe Bara in Karla Fogela[1], v katerem so prikazani tudi konkretni primeri z uporabo sistema CVS.

3.1 Generacije sistemov za nadzor različic

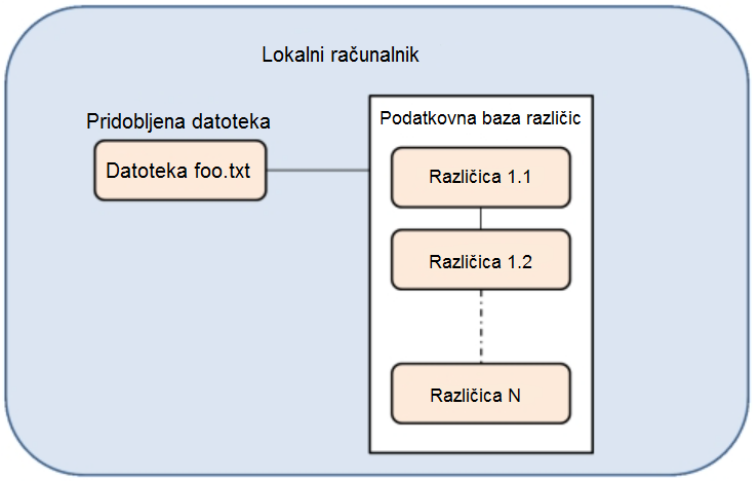
Sistemi so deljeni na tri različne skupine, ki opisujejo njihovo strukturo in način delovanja. Izoblikovale so se zaradi potreb po različnih funkcionalnostih.

3.1.1 Lokalni

Po ugotovitvi, da vzdrževanje različic datotek v obliki kopij z različnimi imeni (`datoteka-v1.0.txt`, `datoteka-v1.1.txt`, `datoteka-v1.2.txt`,...) lahko prinese veliko napak, je bil lokalni sistem za nadzor različic prvi uspešen način, ki je težavo rešil. Revision Control System je bil eden izmed prvih popularnih sistemov, ki je to omogočal.

To orodje deluje tako, da vzdržuje serijo popravkov (razlike med datotekami skozi čas) z uporabo posebnega formata v sledilcu različic, ki je shranjen na disku lokalnega računalnika. Sistem lahko potem natančno poustvari datoteko v katerikoli časovni točki zgodovine sprememb. To naredi s pomočjo združevanja in apliciranja vseh potrebnih popravkov nad izvorno različico v pravilnem vrstnem redu.

Sledilec različic ni nič drugega kot datoteka s svojim datotečnim formatom, ki vsebuje strukturirano vsebino, s pomočjo katere lahko izvaja različne akcije. Ko je datoteka vnešena v RCS, sistem ustvari novega sledilca različic, ki bo vseboval podatke o konfiguraciji za to konkretno datoteko, številko različice, datum, čas, ime avtorja, podatke o veji in povezavo do naslednjega stanja datoteke v obliki posebno formatiranega zapisa. Po končanju tega procesa je datoteka izbrisana, poustvariti jo je možno z uporabo sistema. Na sliki 3.1 je prikazana struktura lokalnih sistemov za nadzor različic.



Slika 3.1: Model lokalnih sistemov za nadzor različic

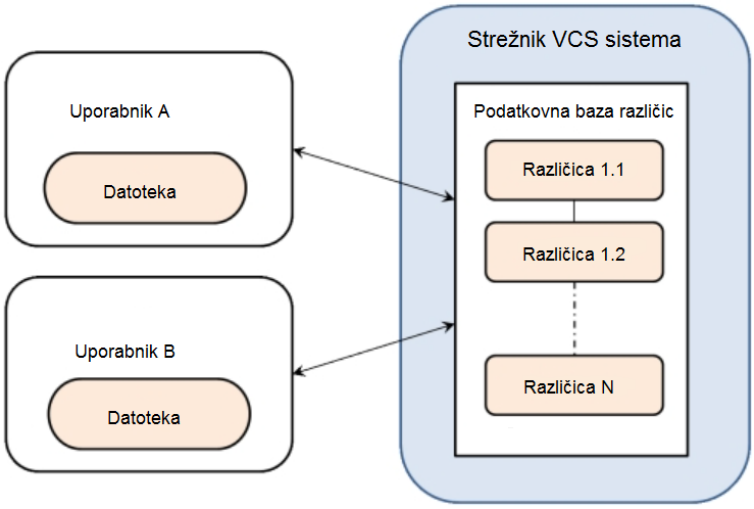
3.1.2 Centralizirani

Kot pri vseh drugih programskih paketih ali konceptih so se potrebe stopnjevale in uporabniki so spoznali, da jih sistem, ki omogoča le lokalno delo,

omejuje. Sistemi so nastali zaradi potrebe sodelovanja večih razvijalcev na istem projektu.

Problem lokalnih sistemov za nadzor različic je bil rešen tako, da so vse datoteke shranjevali na isti lokaciji (strežnik), do katere so imeli vsi uporabniki dostop s svojih lokalnih računalnikov (odjemalci). Kadarkoli je želel uporabnik urediti datoteko, je sistem poskrbel, da je pridobil zadnjo različico. Ta sistem poleg dostopa do datotek ponuja tudi pregled nad aktivnostmi drugih ljudi. Ker so datoteke shranjene na eni lokaciji, preko katere si vsi uporabniki delijo datoteko, so vse spremembe avtomatsko posredovane tudi ostalim uporabnikom. Na sliki 3.2 je prikazana struktura centraliziranih sistemov za nadzor različic.

Ta sistem pa ima nekaj slabosti. Najočitnejša med njimi je onemogočeno shranjevanje sprememb v primeru nedelovanja strežnika. Če sistem odpove, je izgubljena tudi vsa zgodovina različic.



Slika 3.2: Model centraliziranih sistemov za nadzor različic

3.1.3 Porazdeljeni

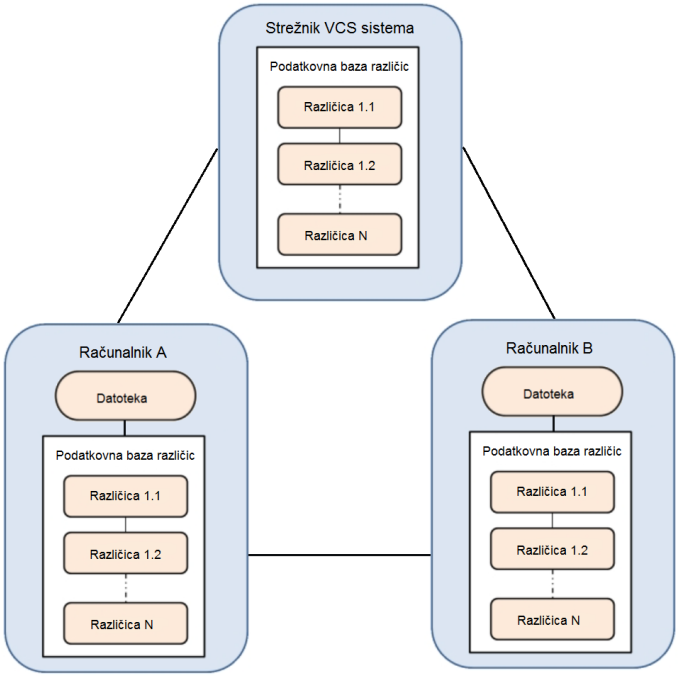
Uporabnike je pri modelu centraliziranega sistema za nadzor različic motilo to, da so ogrožali izgubo vseh podatkov v primeru odpovedi sistema, saj so bili ti shranjeni na eni sami lokaciji. Če združimo prednosti lokalnih in centraliziranih sistemov, dobimo hibridni model ki se imenuje porazdeljeni sistem za nadzor različic (glej sliko 3.3). Porazdeljeni sistemi za nadzor različic vsebujejo prednosti lokalnih sistemov kot so:

- ustvarjanje lokalnih sprememb brez neprekinjene povezave do strežnika in
- ni potrebe po zanašanju na kopije datotek, ki so shranjene na strežniku.

v kombinaciji s prednostmi centraliziranih sistemov za nadzor različic kot so:

- ponovna uporaba dela,
- skupinsko delo in
- ni se potrebno zanašati na zgodovino, shranjeno na posameznih računalnikih.

Ti sistemi so oblikovani tako, da se lahko obnašajo na oba načina. Vsak računalnik v verigi ima pri sebi shranjeno celotno kopijo, medtem ko spremembe na njem sistem usklajuje s strežnikom vsakič, ko je to potrebno. Velika prednost tega sistema je, da v primeru odpovedi strežnika lahko za postavitev novega strežnika uporabimo kar eno izmed uporabniških lokalnih shramb. Poleg tega ima sistem tudi nekaj drugih prednosti, ki so povezane z odzivnostjo, administracijo in preprostostjo uporabe. Porazdeljeni sistemi naj bi podpirali vse funkcionalnosti centraliziranih, nekatere pa celo v izboljšanih različicah.



Slika 3.3: Model porazdeljenih sistemov za nadzor različic

3.2 Orodji `diff` in `patch`

`Diff` je standardni Unix program, katerega funkcionalnost je prikaz razlik na podlagi primerjave dveh besedilnih datotek. Če program `diff` uporabimo nad besedilno datoteko pred spremembo in isto datoteko po spremembi, bo rezultat programa `diff` ravno ta sprememba. Nespremenjeni deli, ki niso del spremembe, ne bodo del rezultata.

Nekdo, ki je dobro usposobljen za pregled razlik programa `diff`, lahko pregleda rezultat in približno oceni, kaj se je dogajalo z datoteko, medtem ko lahko dober program točno pove, kaj se je zgodilo. Kmalu po programu `diff` je Larry Wall napisal odprtokodni program `patch`, pri katerem je primerjava s programom `diff` enaka primerjavi operaciji integriranja in odvajanja.

Če vzamemo za primer razliko med datotekama A in B (datoteka B je pogosto le datoteka A po določenih spremembah) in jo v kombinaciji z datoteko A

uporabimo kot vhodni parameter v program `patch`, bo program kot rezultat vedno tvoril datoteko B.

V svetu programiranja ponavadi na enem projektu sodeluje več razvijalcev, njihovo delo je potrebno na določeni točki združiti v smiselno celoto. Prispevek h kodi je največkrat množica sprememb v raznoraznih datotekah, ki sestavljajo projekt. Vzdrževalec projekta želi točno vedeti, kaj so te spremembe bile, oziroma so in kako je do njih prišlo. Ob predpostavki, da so vse spremembe sprejemljive, jih bo vzdrževalec želel vključiti v projekt s čim manj truda. Idealni način za doseganje tega cilja je, da vzdrževalec prejme serije popravkov, jih pregleda in potem s pomočjo programa `patch` vključi v izvorno kodo tekočega projekta.

Z uporabo programov `diff` in `patch` je nastala standardna pot za združevanje prispevkov kode v t.i. popravke. Programerji so hitro spoznali, da to ne bo dovolj, saj je občasno prišlo tudi do kakšnega hrošča, ki ga je bilo potrebno odstraniti iz kode. Seveda je bilo težje ugotoviti, čigav prispevek h kodi je napako povzročil in kdaj je prišlo do namestitve tega popravka. Kot odgovor na problematiko so začeli nastajati sistemi za vzdrževanje zgodovine sprememb datotek, ki so omogočali pridobivanje predhodnih verzij datoteke za primerjavo s trenutno verzijo. Prvi tak sistem je *Revision Control System* (RCS).

3.3 Revision Control System

Sistem RCS je za tiste čase zadostoval, kljub temu da so mu manjkale ključne funkcionalnosti. Projekte je obravnaval kot skupek datotek brez kakršnih koli povezav med njimi, čeprav so bile datoteke med seboj odvisne ali pa celo v isti imeniški strukturi. Deloval je po principu „zakleni-spremeni-odkleni“, kar je pomenilo, da je moral razvijalec pred vsako spremembo datoteko zakleniti, jo urediti in ob končanem delu odkleniti oz. sprostiti. Ko je bila datoteka zaklenjena, je ni mogel spreminjati nihče drug razen tistega, ki jo je

zaklenil. Seveda je prihajalo tudi do primerov, ko je nekdo datoteko pozabil zaklenjeno. Če je nekdo poskusil zakleniti datoteko, ki je bila že zaklenjena s strani drugega uporabnika, je moral počakati, da jo je trenutni uporabnik odklenil, ali pa odstraniti zaklenjenost. Kot posledica je pred delom na skupni datoteki postal dogovor med razvijalci nujen, pa čeprav so ti delovali na drugem predelu kode. Glavna pomanjkljivost je bila, da je RCS sistem lahko deloval le lokalno v okviru enega računalnika, saj ni imel omrežne podpore. Posledično so morali vsi razvijalci kodo razvijati na istem računalniku, kjer je bila beležena datotečna zgodovina, ali pa si spisati svoje skripte, ki so skrbele za prenos datotek med RCS strežnikom in ostalimi računalniki. Sistem RCS uvrščamo v generacijo lokalnih sistemov za nadzor različic (glej poglavje 3.1.1).

3.4 Concurrent Versions System

Kot odgovor na probleme, ki so nastajali z uporabo RCS, je nastal sistem CVS. Njegova zasnova je bila skupek skript, ki jih je napisal Dick Grune v letu 1986, z namenom narediti sistem RCS preprostejši za uporabo. Leta 1989 je Brian Berliner te skripte prepisal v programskem jeziku C, Jeff Polk pa je kasneje dodal nekaj ključnih funkcionalnosti.

CVS je nadaljeval z uporabo izvornega RCS formata za shranjevanje zgodovine podatkov. Od začetka je bil CVS celo odvisen od pripomočkov sistema RCS za razčlenjevanje formata teh podatkov, vendar z malo več zmožnostmi. Pomembno je izpostaviti, da se je za razliko od svojega predhodnika CVS zavedal imeniške strukture. Imel je celo mehanizem za poimenovanje skupin imenikov, preko katerega jih je bilo možno pridobiti. To je omogočalo sistemu obravnavanje projekta kot smiselne celote, podobno kot si to predstavljamo ljudje. Sistem ni imel potrebe po principu „zakleni-spremeni-odkleni“, kar je omogočalo razvijalcem hkratno urejanje kode in delno potrjevanje sprememb v shrambi (prostor, kjer se hrani koda projekta in zgodovina spre-

memb). Sistem je znal poskrbeti za shranjevanje vseh sprememb na datoteki, združevanje hkratnih sprememb iste datoteke in obveščanje razvijalcev o zaznanih konfliktih (prekrivanjih kode dveh razvijalcev).

V začetkih leta 1990 je Jim Kingdon omogočil sistemu delovanje v omrežju. Razvijalci so po novem lahko dostopali do kode projekta od kjerkoli preko interneta. To je omogočilo dostop do baze kode komurkoli, ki ga je ta zanimala. Ker je sistem znal pametno združevati spremembo ene datoteke s stranih več razvijalcev hkrati, jim je bilo le redko treba skrbeti za organizacijo dela nad istim naborom datotek. Z novo pridobitvijo, se je sistem uvrstil v generacijo centraliziranih sistemov za nadzor različic (glej poglavje 3.1.2. Nekako je bil CVS za kodo podoben razmerju banke proti denarju. Večina ljudi danes več ne skrbi, kako dostopati do denarja v oddaljenih krajih, ga zaščititi pred krajo, beležiti vse transakcije, istočasno dostopati do njega ali celo zapraviti več kot ga imajo. Banke namreč danes za vse to poskrbijo oz. nas opozorijo, ko ga zmanjka.

Internetni dostop do kode, sočasni razvoj in pameten način za združevanje sprememb več razvijalcev so postale privlačne funkcionalnosti pri zaprtokodnih in odprtokodnih projektih. Takrat sta oba svetova (odprtokodni in zaprtokodni) pogosto uporabljala ta sistem. CVS je bil v svojih časih eden izmed bolj uporabljenih odprtokodnih sistemov za nadzor različic.

3.5 Subversion

V začetku leta 2000 je podjetje CollabNet, Inc (http://www.collab.net) začelo iskati razvijalce, ki bi razvili zamenjavo za CVS. Podjetje je v tem času ponujalo programski paket, ki se je imenoval *CollabNet Enterprise Edition* (CEE). Ena izmed komponent je bil tudi sistem za nadzor različic, ki je izhajal iz CVS. Podjetje se je zavedalo omejitev CVS sistema in jim je bilo zato jasno, da bodo v prihodnosti potrebovali nekaj boljšega. Ker je uporaba CVS v tem času postala nenapisan standard v svetu odprtokodne programske opreme,

saj takrat ni bilo boljšega sistema z brezplačno licenco, se je podjetje odločilo, da bodo za začetek razvili novo programsko rešitev za nadzor različic, ki bo vsebovala osnovne ideje CVS, vendar brez napak in zavajajočih funkcionalnosti.

Februarja leta 2000 je podjetje stopilo v stik z avtorjem knjige „Open Source Development with CVS (Coriolis, 1999)“, Karlom Fogelom, s ponudbo o sodelovanju pri razvoju novega projekta. Čisto naključno je Karl v tistem času s svojim prijateljem Jimom Blandy gradil strukturo novega sistema za nadzor različic. V letu 1995 sta s skupnimi močmi ustvarila podjetje Cyclic Software, ki je ponujalo storitve podpore sistemom za nadzor različic CVS. Čeprav sta kasneje podjetje prodala, sta vseeno uporabljala CVS za službene namene. Njuno razočaranje nad CVS sistemom je spodbudilo Jima, da je skrbno premislil o boljših načinih za nadzor različic, ki ga je pripeljalo do osnovne strukture sistema in celo do njegovega imena. Karl se je na ponudbo podjetja CollabNet nemudoma odzval, Jim pa je podjetje, v katerem je bil tisti čas zaposlen, pripravil do tega, da so njegov delavnik poklonili projektu za nedoločen čas. Poglobljeno oblikovanje strukture sistema se je začelo maja istega leta, ko je podjetje CollabNet najelo Karla in Bena Collins-Sussman. S pomočjo nekaj usposobljenih ljudi pod okriljem Briana Behlendorf in Jasona Robbins iz podjetja CollabNet ter samostojnega razvijalca Grega Steina, ki je ta čas deloval na WebDAV/DeltaV procesih, je projekt Subversion hitro pridobil skupnost aktivnih razvijalcev.

Ekipa, ki je skrbela za obliko in strukturo, se je osredotočila na nekaj preprostih ciljev, saj niso želeli odkrivati tople vode, ampak samo izboljšati obstoječ CVS. Odločili so se, da bo Subversion obdržal funkcionalnosti CVS-ja in ohranil razvojni model, medtem ko bodo odpravljene najočitnejše napake in pomanjkljivosti. Podobnosti in novosti so želeli dodati in ohraniti do te mere, da bi se lahko vsak uporabnik CVS sistema le z malo truda naučil uporabljati sistem Subversion.

Po štirinajstih mesecih razvoja, 31. avgusta leta 2001, je postal sistem Sub-

version sam svoj gost. Razvijalci so opustili uporabo sistema CVS za nadzor različic nad projektom Subversion. CVS so nadomestili kar s tedanjo različico sistema SVN. Sistem SVN prav tako kot njegov prednik spada v skupino centraliziranih sistemov za nadzor različic 3.1.2.

SVN je imel pred sistemom CVS kar nekaj prednosti:

- deloval je hitreje, saj je po omrežju pošiljal veliko manj informacij,
- podpiral je več opreacij za delo v načinu brez povezave,
- omogočal je, da se vsaki datoteki dodaja opisne (ang. *meta*) podatke,
- obvladoval je delo z vsemi datotekami ne glede na njihov zapis,
- imel je podporo transakcij po principu „vse ali nič“,
- struktura kode je bila preglednejša, kar je omogočalo lažji dorazvoj.

3.6 GIT

GIT je eden izmed sistemov, ki je zaradi svoje strukture uvrščen v skupino porazdeljenih sistemov za nadzor različic (glej poglavje 3.1.3). Razvit je bil na podlagi izkušenj z uporabo sistema Bitkeeper. Kot za večino velikih dogodkov v zgodovini je bil povod za nastanek GIT kanček destruktivnosti in ognjevite polemike.

Jedro Linux sistemov je zelo obsežen odprtokodni projekt. Ob vrhuncu vzdrževanja jedra Linux sistema (1991-2002) so se spremembe programske opreme prenašale v obliki popravkov in arhiviranih datotek. Leta 2002 so za projekt Linux jedra začeli uporabljati lastniški porazdeljen sistem za nadzor različic, imenovan BitKeeper.

Leta 2005 je razmerje med skupnostjo, ki je razvijala Linux jedro, in komercialnim podjetjem, ki je razvijal BitKeeper, propadlo in orodje je postalo plačljivo. To je spodbudilo skupnost Linux razvijalcev (in konkretno Linusa Torvalds, avtorja sistema Linux) k razvoju lastnega orodja za nadzor različic, ki naj bi nastal na izkušnjah, ki so se jih naučili z uporabo sistema BitKee-

per.

Nekateri izmed ciljev novega sistema so bili:

- hitrost,
- preprosta osnova/oblika,
- močna podpora za nelinearen razvoj (tisoče paralelnih vej),
- popolna porazdeljenost in
- zmožnost upravljanja z velikimi projekti, kot je Linux jedro (hitrost in velikost podatkov).

Od prve izdaje se je sistem GIT razvijal in dozoreval v enostavnosti uporabe, vseeno pa je obdržal svoje začetne kvalitete. Sistem deluje neverjetno hitro, je efektiven za delo z obsežnimi projekti in ima dobro podporo za paralelni razvoj.

Poglavje 4

Primerjava strukturiranih datotek

Za primerjavo dveh datotek je več možnih načinov. Prvi izmed njih je, da si predstavljamo nabor izbrisanih, dodanih, ali spremenjenih vrstic v izvorni datoteki kot podlago za ustvarjanje ciljne datoteke. GNU `diff` primerja dve datoteki vrstico po vrstico, da najde skupine vrstic, ki so si različne, in jih prikaže kot rezultat. Programu `diff` lahko vnaprej določimo, katere spremembe so za nas nepomembne in tekom razlikovanja dveh datotek ne bodo upoštevane (npr. VELIKE-male črke, ponavljajoči presledki).

Drugi izmed možnih načinov je, da si dve datoteki predstavljamo kot zaporedje bajtov, ki je lahko bodisi enako ali različno. Orodje `cmp` prikaže razlike dveh datotek bajt za bajtom, za razliko od programa `diff`, ki se problema loteva vrstico po vrstico. Uporaba `cmp` pride najbolj do izraza takrat, ko želimo izvedeti ali sta datoteki identični oz. ali predstavlja začetek prve datoteke predpono druge. Za lažjo predstavo razlike delovanja primerjave bajt za bajtom in vrstice po vrstici vzemimo za primer dodajanje nove prazne vrstice, v besedilni datoteki. Program `diff` bo izpisal, da je bila v datoteko dodana prazna vrstica, medtem ko bo `cmp` prikazal, da se je skoraj vsak bajt v datoteki spremenil.

Če želimo z uporabo `diff` programov primerjati dve datoteki, ki sta drugače strukturirani, je potrebno razumevanje zapisa, na podlagi katerega je datoteka zgrajena. Ker orodje deluje na nivoju vrstic, je potrebno dokument smiselno preoblikovati v vrstični zapis.

4.1 Format ODT

Za primerjavo dveh binarnih datotek formata `.odt` (*OpenDocument Text*) moramo vedeti, kaj točno želimo primerjati, saj gre v tem primeru za strukturirano datoteko. V tem strukturiranem zapisu je možno poleg besed beležiti tudi njihovo obliko (stil pisave, barva besedila,...).

Dotični format temelji na XML in HTML strukturnih jezikih. Dokument sestavlja več datotek, ki so stisnjene v formatu ZIP. Ključne XML datoteke, ki sestavljajo `.odt` dokument so:

- **content.xml**: vsebina dokumenta in samodejni stili,
- **styles.xml**: stili, uporabljeni v vsebini dokumenta, in samodejni stili, ki so uporabljenih znotraj stilov samih,
- **meta.xml**: opisni (ang. *meta*) podatki dokumenta kot so avtor ali čas zadnjega shranjevanja datoteke in
- **settings.xml**: aplikativno specifične nastavitve kot so velikost okna ali informacije o tiskalniku.

4.2 Primerjava na besedilnem nivoju

Če nas zanimajo le spremembe, ki se nanašajo na besedilo, je primerjava dveh ODT dokumentov sorazmerno preprosta. Za primerjavo je treba najprej pretvoriti oba dokumenta, ki jih želimo primerjati, v besedilni format. To je najlažje narediti z uporabo programa `odt2txt`. Sicer je za primerjavo ODT dokumentov na besedilnem nivoju že veliko rešitev, nekatere so

celo vključene v določene urejevalnike takih dokumentov (npr. LibreOffice, OpenOffice,...).

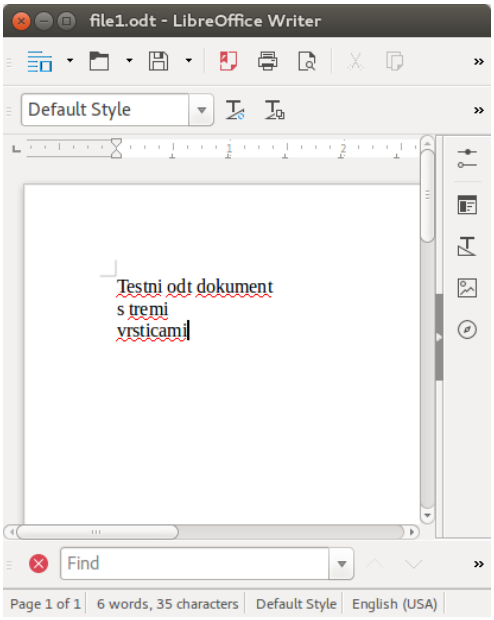
Program `odt2txt` je orodje, ki deluje v ukazni lupini. Služi kot preprost pretvornik `.odt` dokumentov v besedilne datoteke. Napisan je v programskem jeziku `C` in izdan pod licenco GPL-2. Omogoča nam, da iz `.odt` dokumenta (ustvarjenega s pomočjo Libre Office, OpenOffice, StarOffice, KOffice,...) izluščimo besedilni del.

V naslednjem koraku je potrebno primerjati izhodiščni datoteki besedilnega formata. Slednje je mogoče preprosto izvesti z uporabo Unix programa `diff`.

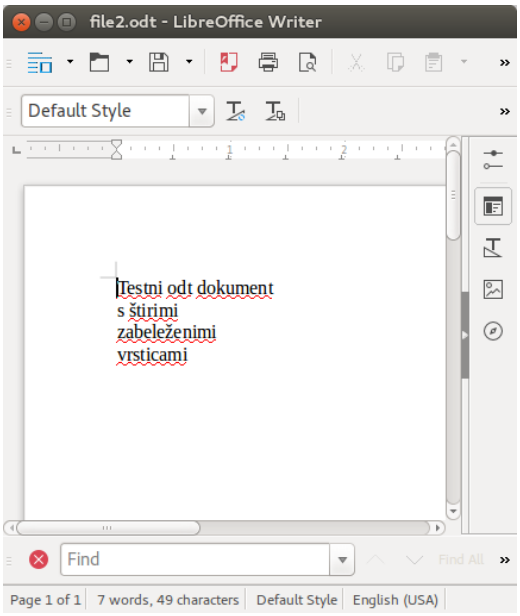
Skupek vseh potrebnih korakov lahko združimo v `bash` skripto in jo poimenujemo `odtCmp` (glej izvorno kodo 4.1), ki kot vhodne parametre sprejme datoteki, ki ju želimo primerjati.

Izvorna koda: 4.1: skripta `odtCmp`

```
1  #!/bin/bash
2  file1=${1};
3  file2=${2};
4  txtFile1='/tmp/odtDiff_f1';
5  txtFile2='/tmp/odtDiff_f2';
6  odt2txt $file1 > $txtFile1;
7  odt2txt $file2 > $txtFile2;
8  diff $txtFile1 $txtFile2;
9  rm $txtFile1;
10 rm $txtFile2;
```



Slika 4.1: Pregled dokumenta `file1.odt` z LibreOffice



Slika 4.2: Pregled dokumenta `file2.odt` z LibreOffice

Za pregled razlik med datotekama `file1.odt` (glej sliko 4.1) in `file2.odt` (glej sliko 4.2) izvedemo ukaz za primerjavo s pomočjo skripte `odtCmp`(glej izvorno kodo 4.2). Prikaz razlike zaradi programa `odt2txt` ni popolnoma pravilen, saj ta doda za vsako vrstico eno prazno. Posledica je, da se ob dodajanju nove vrstice pred njo izpiše tudi ena prazna, kar sploh ni tako moteče - morda je celo preglednejše.

Izvorna koda: 4.2: Zagon skripte `odtCmp`

```
jank@x220:~/odtShell$ ./odtCmp file1.odt file2.odt
4c4,6
< s tremi
---
> s štirimi
>
> zabeleženimi
```

4.3 Primerjava na oblikovnem in besedilnem nivoju

V nekaterih primerih pa primerjava na besedilnem nivoju ne zadostuje. Če želimo določen del besedila iz ene na drugo revizijo dodatno izpostaviti, jo lahko odebelimo, označimo v drugi barvi, spremenimo tipografijo ali podobno. Skripta `odtCmp` takih sprememb ne bi zaznala, saj je njen namen le primerjava sprememb črk in ne oblike besedila.

Kot je bilo rečeno na začetku poglavja 4.1, se moramo za primerjavo oblike besedila zavedati strukture ODT dokumentov. Ena izmed možnosti za primerjavo različice dokumeta, ki je shranjen v shrambi SVN in se razlikuje v obliki besedila, je izvedljiva v naslednjih treh korakih.

1. Pridobitev željene različice dokumenta
2. Izluščevanje oblike in besedila iz dokumentov
3. Primerjava izluščenih oblik in besedila

4.3.1 Pridobitev željene različice datoteke

Pridobitev željene različice datoteke iz shrambe `SVN` je v ukazni vrstici mogoča z ukazom `svn cat` (glej izvorno kodo 4.3).

Izvorna koda: 4.3: Primer klica ukaza `svn cat`

```
svn cat -r 12 file1.odt > file1.odt_r12
```

- Število `12` predstavlja identifikacijsko številko različice datoteke, ki jo želimo iz repozitorija `SVN` pridobiti,
- `file1.odt` predstavlja ime datoteke, pod katerim je v repozitoriju shranjena,
- `file1.odt_r12` pa predstavlja ime datoteke, v katero želimo shraniti pridobljeno različico.

4.3.2 Izluščevanje stila in besedila

Za izluščevanje stila in besedila iz datotek, ki jih bomo primerjali, moramo izdelati program, ki bo po standardih ODT iz željene datoteke izluščil stil in besedilo. Primer, preko katerega bo implementacija programa predstavljena, je spisan v Java programskem jeziku. Zaznavanje v primeru je narejeno le za stile, ki so določeni samodejno (samodejni stili). Taki stili so ustvarjeni, ko uporabnik izbere del besedila in pritisne gumb za odebeljevanje, podčrtavanje, obarvanje besedila,... Če bi želeli zaznavo tudi vnaprej definiranih stilov (npr. Naslov 1, Naslov 2,...), bi bilo potrebno v iskanje vključiti tudi datoteko `styles.xml`. Vhodni parametri programa so imena stilov, ki jih želimo izluščiti. Če ti niso podani, program za izluščevanje uporabi privzet nabor lastnosti:

- font-weight,
- background-color,
- color,
- font-name,
- font-weight in
- font-size.

Najprej je treba `.odt` datoteko, iz katere želimo izluščiti oblikovne značke in besedilo, odpreti in prebrati z uporabo knjižnice, ki je namenjena za delo z datotekami `.zip` formata (v našem primeru `java.util.zip`). Na poljuben način moramo iz datoteke `content.xml`, ki se nahaja znotraj podanega `.odt` dokumenta, poiskati vozlišče, v katerem so zapisani vsi samodejo tvorjeni stili, ki so uporabljeni v dokumentu (`office:document-content -> office:automatic-styles`). Vsak samodejno tvorjen stil je predstavljen s svojim vozliščem `style:style`, ki ima ime zabeleženo v lastnosti `style:name`. Ta vozlišča imajo lahko podrejeno vozlišče `style:text-properties`, kjer so shranjeni oblikovni podatki samodejno tvorjenega stila.

V naslednjem koraku iz datoteke `content.xml` znotraj vozlišča `office:body`

izluščimo vsa besedilna vozlišča `text`. Vsako tako vozlišče ima v lastnosti `text:style-name` zabeleženo ime stila, na katerega se referencira. Besedilna vozlišča je možno gnezditi, kar pomeni, da se gnezdijo tudi oblikovne lastnosti vozlišč. Proces pridobivanja besedila in vseh oblikovnih lastnosti na poti do vozlišča je najenostavneje narediti rekurzivno. Vsakič, ko pridemo do končnega besedilnega vozlišča, ustvarimo nov objekt z besedilom vozlišča in vsemi najdenimi iskanimi oblikovnimi lastnostmi na poti.

Zadnji korak izluščevanja obsega izpis celotnega dokumenta na standardni izhod (**stdout**) ukazne vrstice. Ker bo v nadaljevanju narejena primerjava izhodnih podatkov s standardnim Linux `diff` orodjem, ki deluje na osnovi primerjave vrstic, bo zaradi preprostejšega razumevanja vsaka beseda oz. najmanjši člen, za katerega velja določena oblika, izpisan v svoji vrstici. Program na standardni izhod izpiše vsako besedo oz. najmanjši člen (npr. beseda *testna***Beseda** bo prikazana v dveh vrsticah, zaradi različnih stilov uporabljenih znotraj besede). Strukturna maska izpisa je sestavljena iz dveh delov.

```
(font-weight=bold;background-color=#ffff00;)besedilo
```

Prvi del v oklepaju predstavlja vse stile in njihove vrednosti, ki so bili zaznani na poti do besedilnega vozlišča, ločene z znakom „;“. Drugi del pa je izpisan neposredno za zaklepajem in predstavlja besedilni del. Izvorna koda Java programa, ki izvede vse opisane korake, je v diplomskem delu dodana kot priloga (glej izvorno kodo A.1).

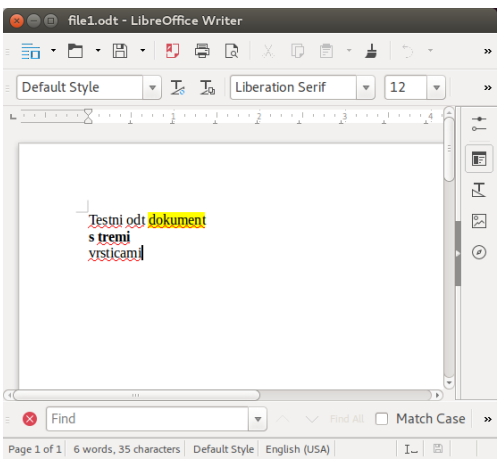
4.3.3 Primerjava izluščenih stilov in besedila

Ko imamo program, ki na standardni izhod izpisuje podatke o besedilu in njegovem stilu, lahko z uporabo `diff` orodja med seboj primerjamo dva dokumenta. Za primerjavo moramo standardni izhod programa `sodt2txt.jar` preusmeriti v začasno datoteko, ki bo vsebovala izvlečke besedila in oblike željenega `.odt` dokumenta. Primerjava začasnih datotek z orodjem `diff`,

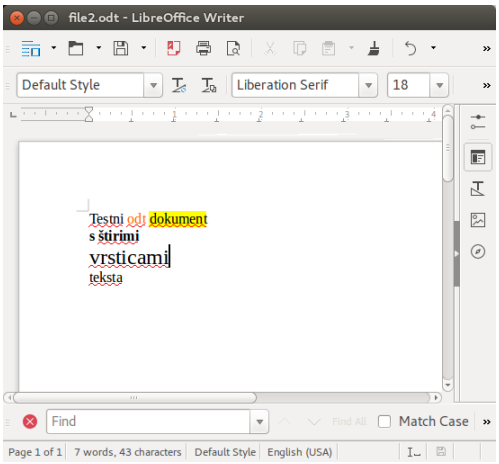
ki predstavljata izvlečke stilov in besedila `.odt` dokumentov, je narejena v okviru programa `sodtCmp` (glej izvorno kodo priloge 4.4).

Izvorna koda: 4.4: skripta `sodtCmp`

```
1  #!/bin/bash
2  file1=${1};
3  file2=${2};
4  styles="";
5  for i in ${@:3}
6  do
7      styles+="$i ";
8  done
9  txtFile1='/tmp/odtDiff_f1';
10 txtFile2='/tmp/odtDiff_f2';
11 java -jar sodt2txt.jar $file1 $styles > $txtFile1;
12 java -jar sodt2txt.jar $file2 $styles > $txtFile2;
13 diff $txtFile1 $txtFile2;
14 rm $txtFile1;
15 rm $txtFile2;
```



Slika 4.3: Pregled dokumenta `sFile1.odt` z LibreOffice



Slika 4.4: Pregled dokumenta `styleFile2.odt` z LibreOffice

Za pregled oblikovnih in besednih razlik med dokumentoma `sFile1.odt` (glej sliko 4.3) in `sFile2.odt` (glej sliko 4.4) izvedemo ukaz za primerjavo s

pomočjo skripte `sodtCmp` (glej izvorno kodo 4.5). Iz primerjave je razvidno, da so zaznane spremembe:

- barve pisave pri besedi „odt“,
- beseda „tremi“ se spremeni v „štirimi“,
- besedi „vrsticami“ se spremeni velikost pisave in
- na koncu se doda beseda „teksta“.

Izvorna koda: 4.5: Zagon skripte `sodtCmp`

```
jank@jx220:~sodtShell$ ./sodtCmp sFile1.odt sFile2.odt
2c2
< ()odt
---
> (color=#ff6600;)odt
5,6c5,7
< (font-weight=bold;)tremi
< ()vrsticami
---
> (font-weight=bold;)štirimi
> (font-size=18pt;)vrsticami
> ()teksta
```

4.3.4 Integracija v sistem SVN

Sedaj lahko vse programe in skripte združimo v orodje, ki bo služilo za primerjavo trenutne delovne različice dokumenta s poljubnim, ki je shranjen v shrambi `SVN` sistema. Da to dosežemo, moramo vse dosedanje korake smiselno izvesti znotraj nove skripte `revCmp` (glej izvorno kodo 4.6), ki bo povezovala pridobivanje željene različice dokumenta iz shrambe, in klic programa `sodtCmp` nad pridobljenim dokumentom in trenutno kopijo dokumenta. Integracija naše skripte v sistem bi bilo mogoče s spreminjanjem izvore kode *Apache Subversion*. V odseku kode, ki je namenjen za primerjavo, bi bilo potrebno vgraditi mehanizem, ki bi v primeru `.odt` datoteke za primerjavo uporabil našo skripto `sodtCmp`.

Izvorna koda: 4.6: skripta `revCmp`

```
1  #!/bin/bash
2  file=${1};
3  revis=${2};
4  fileTmp=$(printf "%s_r%s" "$file" "$revis");
5  svn cat -r $revis $file > $fileTmp
6  styles="";
7  for i in ${@:3}
8  do
9      styles+="$i ";
10 done
11 bash sodtCmp $fileTmp $file $styles;
12 rm $fileTmp;
```

Poglavje 5

Sklepne ugotovitve

Sistemi za nadzor različic so v svetu programiranja prinesli veliko olajšanje pri organizaciji datotek. S pomočjo slednjih si v današnjih časih težko predstavljamo učinkovito vodenje velikih programerskih projektov. Ker je nabor strukturiranih datotek že zelo obsežen, je sistem, ki ga uporabljamo za vodenje, smiselno prilagoditi najbolj uporabljeni strukturi dokumenta.

S pomočjo skript in programov, ki so predstavljeni v diplomski nalogi, je mogoče v poljuben sistem za nadzor različic vpeljati program za izračun razlik dveh strukturiranih datotek. Če bi želeli primerjati drug tip strukturiranih datotek, je potrebno v skripti `sodtCmp` spremeniti le klic `sodt2text.jar`, s poljubnim programom, ki zna strukturiran dokument pretvoriti v besedilni vrstični zapis.

Za izboljšano uporabniško izkušnjo bi bilo smiselno dograditi obstoječe delo z grafičnim uporabniškim vmesnikom, ki bi rezultate primerjave prikazoval na prijaznejši in bolj čitljiv način.

Da bi poljuben sistem za nadzor različic še dodatno izboljšali, bi bilo potrebno spremeniti način skladiščenja strukturiranih datotek. Sistemi namreč nepoznane tipe dokumentov shranjujejo v binarni obliki, kar zahteva veliko prostora v shrambi. Vsaka različica je v primeru takih dokumentov hranjena v

celoti in ne v obliki množice sprememb, na podlagi katere je mogoče s pomočjo sistema za nadzor različic poustvariti dokument. Da bi omogočili specifično možnost shranjevanja strukturiranih datotek, bi bilo potrebno zgraditi mehanizem za poustvarjanje strukturiranih dokumentov na podlagi izvornega dokumenta in množice sprememb (pri besedilnih datotekah nam to omogoča orodje `patch` - glej poglavje 3.2). Mehanizem bi bilo potrebno uporabiti pri pridobivanju datoteke iz shrambe, v njej pa bi morali biti dokumenti shranjeni kot množica sprememb (rezultat programa `sodtCmp`).

Literatura

[1] Moshe Bar, Karl Fogel. *Open Source Development with CVS*. Paraglyph Press, 2003.

[2] Ben Collins-Sussman, Brian W. Fitzpatrick, C. Michael Pilato. *Version Control with Subersion*. O'Reilly, 2011.

[3] Scott Chacon, Ben Straub. *Pro Git*. Apress, 2014.

[4] Ravishankar Somasundaram. *Git: Version Control for Everyone*. Packt Publishing Ltd., 2013.

[5] Eric Sink. *Version Control by Example*. Pyrenean Gold Press, 20011.

[6] Open Document Format for Office Applications (OpenDocument) Version 1.2. 29 September 2011. OASIS Standard.

[7] GNU diffutils. [Online]. Dosegljivo: http://www.gnu.org/software/diffutils/manual/diffutils.html. [Dostopano 9. 2. 2017].

[8] README datoteka orodja odt2txt. [Online]. Dosegljivo: https://github.com/dstosberg/odt2txt/blob/master/README.md. [Dostopano 9. 2. 2017].

[9] How to read XML file in Java / Kako prebrati XML datoteko s pomočjo Jave. [Online]. Dosegljivo: https://www.mkyong.com/java/how-to-read-xml-file-in-java-dom-parser/ [Dostopano 21. 2. 2017]

[10] Basic SVN Commands / Osnovne SVN operacije. [Online]. Dosegljivo: http://www.linuxfromscratch.org/blfs/edguide/chapter03.html [Dostopano 21. 2. 2017]

[11] The OpenDocument Format / Format OpenDocument. [Online]. Dosegljivo: https://mashupguide.net/1.0/html/ch17s03.xhtml [Dostopano 21. 2. 2017]

[12] Download LibreOffice / Prenesi LibreOffice. [Online]. Dosegljivo: https://www.libreoffice.org/download/download/ [Dostopano 21. 2. 2017]

[13] Steps to Using JDOM / Koraki za uporabo knjižnjice JDOM. [Online]. Dosegljivo: https://www.tutorialspoint.com/java_xml/java_jdom_parse_document.htm [Dostopano 21. 2. 2017]

[14] GNU General Public Licence. [Online]. Dosegljivo: https://www.gnu.org/copyleft/gpl.html. [Dostopano 20. 2. 2017].

[15] SVN vs CVS / SVN proti CVS. [Online]. Dosegljivo: http://www.pushok.com/software/svn-vscvs.html. [Dostopano 28. 2. 2017].

Priloge

Dodatek A

Izvorna koda

Izvorna koda: A.1: Izvorna koda Java programa `sodt2txt.jar`

```
1  package styleParser;
2
3  import org.jdom2.Attribute;
4  import org.jdom2.Content;
5  import org.jdom2.Document;
6  import org.jdom2.Element;
7  import org.jdom2.Text;
8  import org.jdom2.input.SAXBuilder;
9  import java.util.zip.ZipFile;
10 import java.util.zip.ZipEntry;
11 import java.util.ArrayList;
12 import java.util.Arrays;
13 import java.util.Enumeration;
14 import java.util.HashMap;
15 import java.util.Iterator;
16 import java.util.List;
17 import java.util.Map;
18
```

```
19 public class FileParser {
20
21   public class StyleString {
22
23     private Map<String,String> styles;
24     private String text;
25
26
27     public StyleString(Map<String, String> styles, String text){
28       this.text = text;
29       this.styles = styles;
30     }
31
32     public String getStyles(){
33       StringBuffer sb = new StringBuffer();
34       sb.append('(');
35       for (Map.Entry<String,String> entry : styles.entrySet()) {
36           String key = entry.getKey();
37           String value = entry.getValue();
38           sb.append(key);
39           sb.append('=');
40           sb.append(value);
41           sb.append(';');
42       }
43       sb.append(')');
44       return sb.toString();
45     }
46
47     @Override
48     public String toString() {
49       return this.getStyles() + text;
```

```
50      }
51
52      public String[] toWords() {
53        String style = getStyles();
54        ArrayList<String> words = new ArrayList<String>();
55
56        for(String word : text.replaceAll("\\s+", " ").split(" ")){
57          words.add(style + word);
58        }
59        return words.toArray(new String[words.size()]);
60
61      }
62
63    }
64
65      private List<Element> stylesList;
66      private ArrayList<StyleString> styleStringList;
67      private String[] searchedAttributes;
68      private String fileName;
69
70
71      /*CONSTANTS*/
72      private static final String[] defaultATTRIBS =
73          new String[]{
74                  "font-weight",
75                  "background-color",
76                  "color",
77                  "font-name",
78                  "font-weight",
79                  "font-size"
80                  };
```

```
81
82      /*Creates a new instance of OpenOfficeParser */
83      public FileParser(String [] args) {
84        if(args.length == 0 || args[0].equals("--help")){
85          printHelp();
86          System.exit(2);
87        }
88
89        fileName = args[0];
90        if(args.length > 1)
91          searchedAttributes = Arrays.copyOfRange(args, 1, args.length);
92      else searchedAttributes = defaultATTRIBS;
93      }
94
95      //return applied attributes from style list (styles inherit each other!)
96      public Map<String, String> getTextAttributes(List<String> elementStyles){
97        Map<String,String> attributeValues = new HashMap<String,String>();
98
99
100       //NOTE: List of elementStyles must be ordered
101       //from least significant to most significant
102       for(String styleName : elementStyles){
103         for(Element style : stylesList){
104           boolean nameMatch = false;
105           Map<String,String> tmpAttVals = new HashMap<String,String>();
106
107           //Check if attribute name matches with currently treated style
108           for(Attribute attrib: style.getAttributes()){
109                   if(attrib.getName().equals("name")){
110                     //if style-name attribute is not found
111                     // within that style, then skip it
```

```
112                     if(attrib.getValue().equals(styleName)){
113                       nameMatch = true;
114                       break;
115                     }
116                   }
117             }
118
119                 /*if style name is OK, dig deeper into textProperties*/
120                 if(nameMatch){
121                   for(Element textProperties : style.getChildren()){
122                     if(textProperties.getQualifiedName().equals(
123                         "style:text-properties")){
124                       for(Attribute attrib: textProperties.getAttributes()){
125                         //loop through attributes that we are searching for
126                         //and set value to map if att is found
127                         for(String searchedAttrib : searchedAttributes){
128                           if(attrib.getName().equals(searchedAttrib))
129                             tmpAttVals.put(searchedAttrib, attrib.getValue());
130                         }
131                             }
132                     }
133                   }
134                 }
135                 //we got ourselves potential attributes from current style,
136                 //if attributes value is null - nothing about it was found
137                 //if attributes value is NOT null- attribute was found!
138                 for(String attributeName : searchedAttributes){
139                   String attVal = tmpAttVals.get(attributeName);
140                   if(attVal != null)
141                     attributeValues.put(attributeName, attVal);
142                 }
```

```
143
144             }
145         }
146         //return latest(most important) attributes found.
147         return attributeValues;
148     }
149
150
151
152     //Process text elements recursively
153     public void processElement(Object o, List<String> styles) {
154
155         if (o instanceof Element) {
156
157             Element e = (Element) o;
158             String elementName = e.getQualifiedName();
159
160             //create custom style level for nodes deeper from that recursion point
161             ArrayList<String> sty = new ArrayList<String>();
162             sty.addAll(styles);
163
164             //find if node has attribute we are searching for, add it to sty
165             for(Attribute attrib: e.getAttributes()){
166               if(attrib.getName().equals("style-name"))
167                 sty.add(attrib.getValue());
168             }
169
170             if (elementName.startsWith("text")) {
171
172                 if(!elementName.equals("text:s") &&
173                     !elementName.equals("text:tab")) {
```

```
174                     List<Content> children = e.getContent();
175                     Iterator<Content> iterator = children.iterator();
176
177                     while (iterator.hasNext()) {
178
179                         Object child = iterator.next();
180                         //If Child is a Text Node, then append the text
181                         if (child instanceof Text) {
182                             Text t = (Text) child;
183                             styleStringList.add(
184                               new StyleString(
185                                 getTextAttributes(sty),
186                                 t.getValue()));
187                         }
188                         else
189                         processElement(child , sty); // Recursively process
190                     }
191                 }
192             }
193             else {
194                 List<Content> non_text_list = e.getContent();
195                 Iterator<Content> it = non_text_list.iterator();
196                 while (it.hasNext()) {
197                     Object non_text_child = it.next();
198                     processElement(non_text_child, sty);
199                 }
200             }
201         }
202     }
203
204     public void printStyledText() throws Exception {
```

```
205         //Unzip the openOffice Document
206         ZipFile zipFile = new ZipFile(fileName);
207         Enumeration<?> entries = zipFile.entries();
208         ZipEntry entry;
209
210         while(entries.hasMoreElements()) {
211             entry = (ZipEntry) entries.nextElement();
212
213             if (entry.getName().equals("content.xml")) {
214
215                 SAXBuilder sax = new SAXBuilder();
216                 Document doc = sax.build(zipFile.getInputStream(entry));
217                 Element rootElement = doc.getRootElement();
218
219                 ArrayList<String> styles = new ArrayList<String>();
220                 List<Element> childs = rootElement.getChildren();
221                 Element bodyElement = rootElement;
222                 styleStringList = new ArrayList<StyleString>();
223
224                 for(Element el : childs){
225                   if(el.getQualifiedName().equals("office:automatic-styles"))
226                     stylesList = el.getChildren();
227                   if(el.getQualifiedName().equals("office:body"))
228                     bodyElement = el;
229                 }
230
231                 processElement(bodyElement, styles);
232                 break;
233             }
234         }
235         zipFile.close();
```

```
236         /*loop through all styled elements found in odt file*/
237         for(StyleString stylesString : styleStringList){
238           /*loop through &print all words in styled elements*/
239           for(String parsedWord : stylesString.toWords()){
240             System.out.println(parsedWord);
241           }
242         }
243     }
244
245     public void printHelp(){
246       System.out.print("sodt2txt.jar input arguments:");
247       System.out.print("fileName [style-attribute1] [style-attribute2] [...]");
248       System.out.println("\n");
249       System.out.print("In case no arguments are passed in,");
250       System.out.print("defaults will be used:");
251       for(String st:defaultATTRIBS)
252         System.out.println(st);
253     }
254
255     public static void main(String args[]) throws Exception
256     {
257         new FileParser(args).printStyledText();
258     }
259 }
```