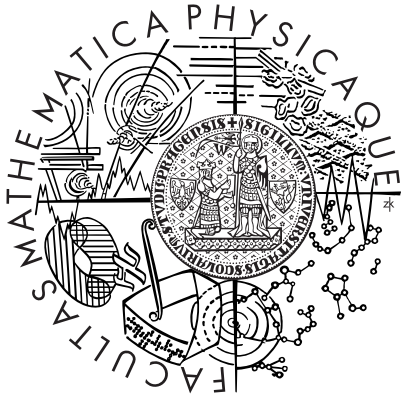
Univerzita Karlova v Praze
Matematicko-fyzikální fakulta

BAKALÁŘSKÁ PRÁCE



Ondřej Kunčar

SVN Proxy

Katedra softwarového inženýrství

Vedoucí bakalářské práce: RNDr. David Bednárek

Studijní program: Informatika
Studijní obor: Obecná informatika

PRAHA 2006

Rád bych poděkoval mé rodině za to, že mi umožnila studovat, vedoucímu mé práce Davidu Bednárkovi za několik cenných připomínek a Jiřímu Janíčkovi za jazykovou korekturu textu.

Prohlašuji, že jsem svou bakalářskou práci napsal samostatně a výhradně s použitím citovaných pramenů. Souhlasím se zapůjčováním práce a jejím zveřejňováním.

V Praze dne 10. 8. 2006 Ondřej Kunčar

Obsah

Název práce: SVN Proxy
Autor: Ondřej Kunčar
Katedra (ústav): Katedra softwarového inženýrství
Vedoucí bakalářské práce: RNDr. David Bednárek
e-mail vedoucího: david.bednarek@mff.cuni.cz

Abstrakt: SVN Proxy je aplikace, která se chová jako proxy SVN repository. Obsahuje lokální repository, do kterého uživatel provádí commity. Lokální repository je schopné se na požádání synchronizovat se vzdáleným SVN repository. Lokální revize mohou být číslovány tak, aby nedošlo k rozsynchronizování číslování se vzdáleným repository. Data v lokálním repository jsou ukládána v jazyku XML. Program obsahuje implementaci hledání rozdílů mezi verzemi a umí verze spojovat a detekovat konflikty. SVN Proxy je aplikace pro příkazovou řádku a její rozhraní je odvozeno z rozhraní standardního SVN klienta. V návrhu aplikace je kladen důraz na snadnost budoucího rozvoje. Návrh umožňuje přidání dalších přístupových metod k repository a vytvoření jiných uživatelských rozhraní včetně GUI. Aplikace je určena pro vývojáře zdrojových textů, kteří nechtějí nebo nemohou posílat každý commit do vzdáleného repository.

Klíčová slova: správa verzí, proxy, SVN

Title: SVN Proxy
Author: Ondřej Kunčar
Department: Department of Software Engineering
Supervisor: RNDr. David Bednárek
Supervisor's e-mail address: david.bednarek@mff.cuni.cz

Abstract: SVN Proxy is an application that behaves like a proxy of a SVN repository. It contains a local repository, which is used by a user for local commits. The local repository enables to synchronize itself with a remote SVN repository on demand. Local revisions can be numbered in a way they never become unsynchronized with the remote repository. The data of the local repository is stored in XML. The application contains the implementation of a diff algorithm and enables to merge versions and to detect conflicts. SVN Proxy is a command line client and its interface is derived from the common SVN client interface. The main emphasis is laid on the easy future development in the design of the application. The design enables to add other new connections to the repository and create other user interfaces including GUI. The application is intended for source text developers who do not want or are not able to send every commit into the remote repository.

Keywords: revision control, proxy, SVN

Kapitola 1

Úvod

1.1 Cíl práce

Cílem této práce je vytvořit dvouvrstevný systém pro správu verzí. Bude obsahovat místní repository[1], které bude sloužit k lokálnímu ukládání spravovaných verzí. Systém bude schopen tento místní server synchronizovat se vzdáleným standardním SVN repository. Aby systém vzbuzoval iluzi standardního SVN, bude jeho rozhraní odvozeno od rozhraní SVN klienta pro příkazovou řádku. Systém bude primárně určen pro správu zdrojových textů.

1.2 Motivace

Motivací pro použití tohoto systému je situace, kdy uživatel nechce nebo nemůže posílat každý commit do vzdáleného repository. Důvodů k tomuto počínání může být několik:

- Uživatel má pomalé nebo žádné připojení k Internetu, které je potřebné pro přístup k vzdálenému repository. To může například nastat, pokud používá k práci notebook, se kterým nemá vždy přístup k Internetu. Proto si ukládá své změny lokálně, a až mu bude Internet dostupný, provede jeden větší commit.
- Uživatel má připojení k Internetu, ale nechce do vzdáleného repository odesílat své velmi malé změny, nýbrž souhrn těchto změn. Modelovým příkladem může být vývoj open source projektu komunitou vývojářů, která hlasuje o navržených změnách. Uživatel pak bude chtít navrhnout ucelenou změnu, o které nechá hlasovat, nikoliv dílčí změny, které se mohou vyskytovat v několika variantách a pokusech.

[1] repository se autor rozhodl nepřekládat a bude skloňováno jako *to* centrální úložiště

- Uživatel projekt vyvíjí pro více platforem a přirozeně jej na těchto více platformách testuje. Pak může použít lokální repository, které je dostupné z těchto více platforem (například je na sdíleném disku). Commity provádí do tohoto repository a kód testuje na druhé platformě (po commitu třeba potřebuje přebootovat počítač nebo používá jiný). Až je kód přiměřeně funkční na všech požadovaných platformách, provede uživatel commit do vzdáleného repository.

1.3 Kontext zadání

K pochopení struktury dalšího popisu je třeba uvést kontext vývoje SVN Proxy. Aplikace byla původně vyvíjena v rámci ročníkového projektu jako jednodušší správa verzí s možností budoucího rozšíření. Tímto rozšířením se pak v rámci tvorby bakalářské práce stala možnost synchronizace s SVN repository motivovaná situacemi popsanými v oddílu 1.2. Proto také v dalších kapitolách bude popis návrhu a řešení odpovídat popisu lokální části projektu a způsobu, jakým byla začleněna možnost synchronizace. Průběh vzniku konečné podoby aplikace bylo potřeba uvést, neboť je jím ovlivněna architektura systému a některá zvolená řešení. Vzhledem k faktu, že vzdálené SVN repository je implementováno pomocí SVN samotného, tvoří realizace synchronizace jen menší část celého díla, čemuž odpovídá i pozornost věnovaná v textu.

Kapitola 2

Návrh

Zadané téma je velmi rozsáhlé a výsledná aplikace by mohla být vyvíjena v čase několikráte přesahujícím čas dostupný k vývoji aplikace stávající. Proto se pozornost při návrhu soustředila na vytvoření systému, který by se dal v budoucnu rozumně rozvíjet, nikoliv na to, aby obsáhnul co nejvíce rozličných funkcí často ne až tak důležitých. Protože však půjde o systém, který bude běžet lokálně a v řadě případů jednouživatelsky, měl by být návrh i přiměřeně jednoduchý.

2.1 Výběr modelu správy verzí

Základní úlohou systému pro správu verzí je umožnit sdílení a modifikování dat více uživateli. K naplnění tohoto cíle může být zvoleno několik modelů. Jedním z nich je model copy-modify-merge popsaný například v [Cou89, CSFP04]. V tomto modelu se každý uživatel připojí k repository a stáhne si pracovní kopii do pracovního adresáře. Tento pracovní adresář je lokální podobou souborů a adresářů v repository. Uživatelé mohou pracovat paralelně a měnit svoje soukromé pracovní adresáře. Nakonec jsou všechny změny sloučeny v novou konečnou verzi. Systém správy verzí může asistovat u slučování, ale hlavní díl práce a odpovědnosti je ponechán na člověku. Ten musí zajistit, aby výsledná verze byla korektní.

Copy-modify-merge model se také někdy nazývá multiple checkout model a odlišuje se od jiného modelu, lock-modify-unlock modelu (též exclusive checkout model). V modelu lock-modify-unlock repository dovoluje jen jedné osobě měnit soubor v daném čase. Tato výlučnost je zajištěna pomocí mechanismu locků (zámků). Než uživatel může změnit nějaký soubor, musí ho zamknout (lock). Ostatním uživatelům je pak znemožněna práce na tomto souboru a musí vyčkat, dokud první uživatel tento soubor neodemkne (unlock). V [CSFP04] jsou popsány problémy, které tento model může přinést:

- Uživatel, který soubor zamkne, na něj může zapomenout a ostatní uživatelé musejí zbytečně čekat.

- Uživatel A chce editovat začátek souboru a uživatel B konec souboru. Jejich změny se nepřekrývají, přesto je jim znemožněno editovat soubor současně.

- Zámky způsobují falešný pocit bezpečnosti. Pokud jsou zároveň editovány dva soubory, které na sobě závisejí, nemusejí jejich nové verze být navzájem kompatibilní. Uživatelé si však toto riziko neuvědomují, neboť si představují, že zamknutím souboru jsou jejich změny bezpečné.

Z důvodu výše popsaných problémů, a proto, že model copy-modify-merge je běžně používaný a široce přijímaný model, byl modelem správy verzí SVN Proxy zvolen právě model copy-modify-merge. Z toho plyne nutnost implementovat postupy, které budou umožňovat spojování souborů (merge) a případně detekovat vzniklé konflikty. Spojování různých verzí se může zdát být slabým místem zvoleného modelu, nicméně v [Ber90] se tvrdí, že konflikty jsou poměrně vzácné a v mnoha případech jejich vyřešení nepředstavuje žádnou obtíž.

2.2 Datový model

Při diskuzi o datovém modelu byly diskutovány dvě otázky: Jak budou obecně data ukládána v repository a jaký zvolit formát nejen dat v repository, ale i ostatních pomocných dat. Způsob uložení pomocí databáze byl zamítnut hned napočátku, protože nám jde také o jednoduchost celého systému a především bychom mohli narazit na problémy s přenositelností takové databáze na jiné platformy. Proto budou data ukládána do datových souborů.

Formát těchto souborů byl druhou otázkou. SVN například používá v repository poněkud nepřehledný polobinární formát[1]. Snahou bylo vyhnout se tvorbě nějakého dalšího ad hoc vytvořeného formátu. Proto volba padla na jazyk XML. Přínosů (a zároveň i důvodů rozhodnutí) této volby je celá řada: Je to formát pro člověka čitelný, je standardizovaný, multiplatformní (přenositelný), samovysvětlující, textový, dokument lze validovat a lze použít znakovou sadu Unicode. Nevýhodou jsou prostorové nároky XML a režie spojená se zpracováním. Velké prostorové nároky lze částečně odstranit pomocí komprese tak, jak je navrhováno v oddílu 4.2. Zvýšenou režii spojenou se zpracováním lze chápat jako daň za předešlé výhody. Nicméně režie zpracování obsahuje i čas strávený validací dokumentu, což je žádaná vlastnost, neboť to pak zjednodušuje a zpřehledňuje vlastní kód aplikace. Nemusíme už řadu věcí kontrolovat, pokud je dokument validní. K popisu struktury dokumentu bude sloužit definice typu dokumentu (DTD). Jednotlivé dokumenty pak budou během běhu aplikace parsovány z datových souborů a reprezentovány pomocí stromového modelu DOM v paměti počítače. Změněná struktura pak bude zpět ukládána do datových souborů. Další podrobnosti o datovém modelu repository jsou uvedeny v oddílu 2.3.2.

[1] pokud se použije backend fsfs

Použití XML umožňuje jednodušší návrh aplikací, které by operovaly na těchto datových souborech, od jiných tvůrců. V neposlední řadě lze taktéž využít jazyka XSL a jednoduše například data uložená v repository transformovat do jiných formátů třeba za účelem jejich efektní prezentace. Všechny tyto úkony se zjednodušují a jsou dobře definovatelné ve srovnání se situací, kdy bychom použili nějaký svůj vlastní formát.

2.3 Architektura systému

Již samotné zadání práce naznačuje, že systém bude obsahovat centrální repository, nepůjde tedy o decentralizovaný typ správy verzí. Proto systém obsahuje standardní centrální repository a uživatelé z něj stahují soubory a do něj nahrávají vytvořené změny tak, jak je to obvyklé v modelu copy-modify-merge. Architektura je zobrazena na obrázku 2.1. Jednotlivé části systému budou dále podrobněji popsány v příslušných oddílech.

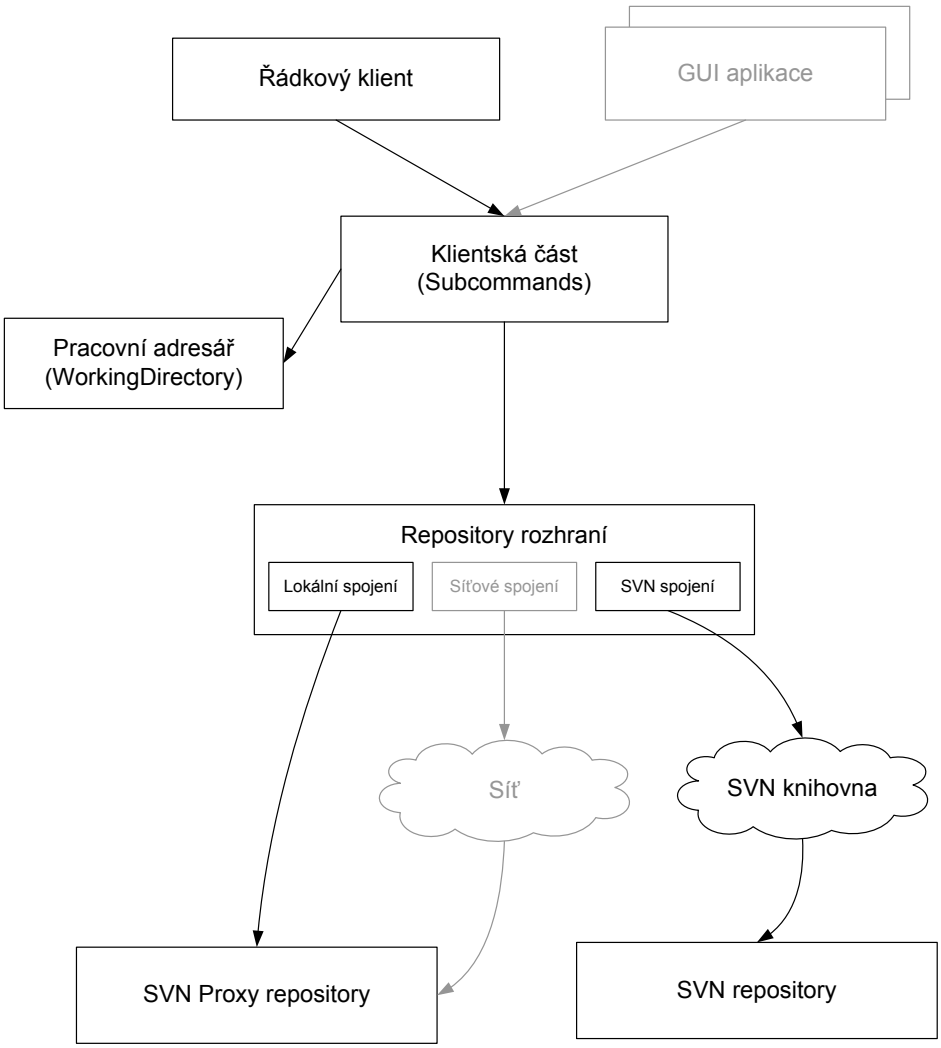
2.3.1 Přístup k repository

Ve snaze udělat systém co možná nejvíce flexibilní bylo při návrhu vytvořeno repository rozhraní, které umožňuje skrýt přemostění mezi klientem a skutečným repository. Repository rozhraní, tak jak ho vidíme na obrázku 2.1, je návrhovým vzorem Bridge. Snažíme se skrýt implementaci přístupu k repository (zde nazýváno *spojení*) a zároveň v budoucnu umožnit vytvoření dalších nových spojení bez zásahů do existujícího kódu. Vytvořená spojení musejí implementovat rozhraní `RepositoryInterface`.

Stávající návrh obsahuje dvě spojení. Je to lokální spojení, které je velmi jednoduše implementováno a pouze deleguje požadavky přímo na lokální repository. Druhým spojením je SVN spojení. Toto spojení plní navíc úlohu návrhového vzoru Adapter, protože přizpůsobuje rozhraní SVN repository na rozhraní SVN Proxy repository.

Návrh a implementace SVN spojení je složitější než návrh a implementace lokálního spojení – počínaje složitou inicializací SVN knihovny, dále skrytím vlastní správy pamětí pomoci apr_pool_t a konče naprogramováním mnoha callbacků. Především v případě commitu obsahuje složitější mapování požadavků na úroveň SVN knihovny. K tomuto účelu byla navržena pomocná třída `SVNEditor`. Tato třída například překlenuje rozdíl, kdy v SVN repository k smazání adresáře stačí zavolat příslušnou funkci a není potřeba mazat soubory v tomto adresáři. Proto v SVNEditoru je potřeba tato nadbytečná mazání souborů odchytávat, neboť po smazání adresáře již nesmí být provedena. SVN repository také očekává určité pořadí provádění změn při commitu. I toto je zde řešeno.

Repository rozhraní také umožňuje, jak bylo výše zmíněno, přidávání dalších implementací spojení. Nabízí se vytvořit síťové spojení k repository, jak je na



Obrázek 2.1: Rozhraní repository

obrázku 2.1 naznačeno šedou barvou. Toto spojení by obsahovalo složitou serializaci objektů a jejich posílání po síti. Dále ošetření všech obtíží, které s sebou síťový přenos přináší. Serializace posílaných objektů by mohla využít už současného návrhu, neboť téměř každý z objektů, který je použit při komunikaci s repository, se umí uložit do XML a také se z XML vytvořit. Zřejmě by bylo vhodné jen dodat nějakou kompresi těchto XML struktur, aby se zmenšil datový tok po síti.

Repository rozhraní se volí za běhu programu pomocí návrhového vzoru Abstract Factory. Třída `RepositoryConnectionFactoryInterface` představuje rozhraní factory na vytváření repository spojení, konkrétní implementace od něj dědí. Konkrétní implementace pak musejí být schopny určit, jaké spojení vrátit na základě cesty zadané při volání `make_connection()`. Současný návrh obsahuje dvě takové factory:

- `BaseRepositoryConnectionFactory` – tato továrna vrací jenom lokální spojení.
- `SVNRepositoryConnectionFactory` – potomek předešlé factory, umí rozhodnout, jestli vytvořit SVN spojení, nebo lokální spojení. V druhém případě volá metodu předka.

Dvě výše uvedené factory navíc vyráběná spojení hašují. Pokud tedy klient požádá o spojení, které je již vytvořeno, je mu vráceno toto existující spojení. Pomocí počítání referencí je dosaženo korektnosti tohoto přístupu – pokud již spojení není používáno, uzavře se.

2.3.2 Repository

Je spravováno pomocí třídy `Repository`, která by měla implementovat rozhraní `RepositoryInterface`. Repository ukládá verzovaný adresářový strom. K tomuto účelu se částečně používá souborového systému operačního systému, na kterém repository běží. V souborovém systému daného systému je vytvořen adresářový strom, který odpovídá verzovanému adresářovému stromu.

Všechny verze souboru jsou uloženy v souboru, který má stejný název jako verzovaný soubor plus přípona xml. Tento soubor se nachází na stejném místě (myšleno podadresáři) jako ve verzovaném stromu. Popsaný způsob ukládání dat byl zvolen především díky své jednoduchosti. Představuje přehledný a přirozený způsob, jak tato data ukládat. Možnost ukládat celý strom do jednoho souboru byla zamítnuta. Jelikož volba formátu datových souborů padla na XML, přístup k datům by v případě jednoho souboru byl příliš drahý díky režii spojené s parsováním.

Služby pro práci se souborem v repository poskytuje třída `VersionFile`. Soubor je uložen ve formátu XML a obsahuje text poslední verze souboru. Další verze jsou zachyceny pomocí souhrnu editačních změn, které je potřeba provést na nejnovější text, abychom dostali jeho starší verzi.

Pokud dojde ke smazání souboru ve verzovaném stromě, jemu odpovídající soubor se fyzicky nemaže. Informace o smazání se do něj pouze zaznamená jako speciální druh popisu rozdílu mezi verzemi. Byly navrženy následující služby, které by třída `VersionFile` měla poskytovat:

- nastavení a získání textu nejnovější verze souboru
- nastavení a získání čísla nejnovější verze
- zjištění existence největší z menších existujících verzí než je zadaná verze
- získání rozdílu mezi dvěma po sobě jdoucími verzemi

Editace ve stromové struktuře jsou zaznamenány ve speciálním souboru, který je obsažen v každém adresáři. Tento soubor má stejný formát jako ostatní soubory (tedy opět je spravován třídou `VersionFile`). Pochopitelným rozdílem je fakt, že neobsahuje žádné textové informace.

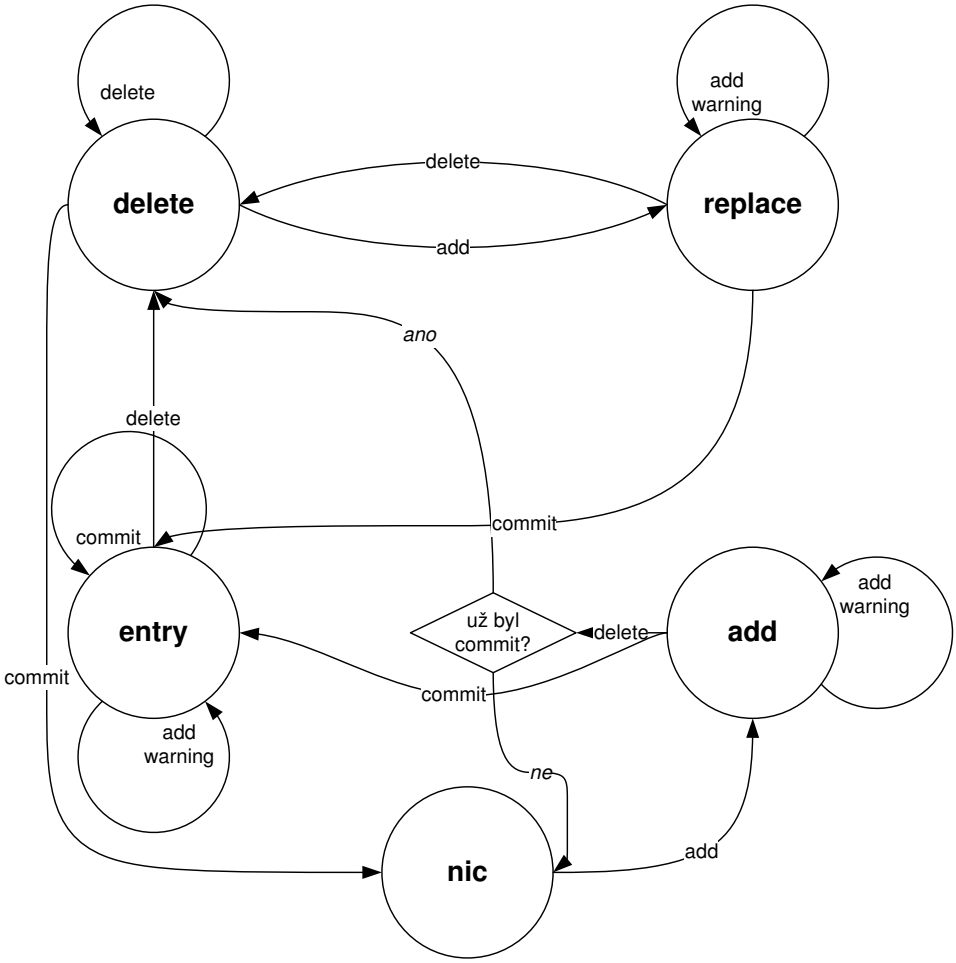
Protože názvy souborů a adresářů mohou někdy kolidovat, jsou v tomto případě názvy komoleny, aby byla tato kolize odstraněna. Služby pro komolení názvů taktéž poskytuje třída `VersionFile`. Repository si navíc ukládá logovací záznamy. Potřebnou funkčnost poskytuje třída `LogFile`.

Snahou bylo navrhnout co možná nejjednodušší repository. Proto je repository bezestavové a jediným stavem, který si uchovává, jsou zámky. Před každým čtením je nutné repository zamknout pro čtení, nebo pro zápis. Zamknout pro čtení lze repository libovolněkrát, pro zápis jen jednou. Je-li repository zamknuté pro čtení, nelze jej zamknout pro zápis. Je-li zamknuté pro zápis, lze jej zamknout pro čtení jen v případě, že požadavek přišel od stejného klienta. Klienti se identifikují pro ně neprůhlednou třídou `Locker`. V současném návrhu `Locker` zapouzdřuje cestu pracovního adresáře, který představuje klienta. Přidáváním dalších vlastností se může obsah třídy `Locker` změnit: Může zahrnovat jméno uživatele (při víceuživatelském systému), identifikaci počítače (při rozšíření o síťové spojení) apod.

2.3.3 Pracovní adresář

Pracovní adresář je spravován třídou `WorkingDirectory`. Z programátorského hlediska je zajímavý způsob správy souborů v pracovním adresáři. Informace v pracovním adresáři jsou ukládány do administrativní části, která se nachází v podadresáři .ver. Administrátorský podadresář se nachází v každém podadresáři pracovního adresáře. V souboru WDItems.xml jsou uloženy informace o adresářích a souborech. Informace o pracovním adresáři, ke kterému WDItems.xml patří, jsou uloženy pod názvem „.“. Soubor WDItems.xml je spravován třídou `WDItemsList`.

Na obrázku 2.2 vidíme, jakých stavů mohou položky nabývat. Jsou to tyto stavy:



Obrázek 2.2: Stavy položek v pracovním adresáři

- **add** – položka je naplánována pro přidání
- **delete** – položka je naplánována pro smazání
- **entry** – položka je v normálním stavu
- **replace** – položka bude vyměněna za jinou
- **nic** – položka se vůbec nenachází pod správou verzí

Hrany se šipkami pak představují přechody mezi těmito stavy. Jsou to akce, které s nimi lze provádět – přidání, smazání a commit. Pokud je u přechodu připsáno warning, je vydáno varování, ale přechod proběhne. Zajímavá je situace, pokud chceme smazat položku, která je naplánována pro přidání. Pokud ještě nebyla podrobena commitu (to znamená, že byla přidána až po posledním provedeném commitu), je úplně odstraněna, tj. záznam o ní je smazán a je fyzicky odstraněna z disku.

Další informací uloženou u souborů je čas posledního zápisu do tohoto souboru. Porovnáním se skutečným časem posledního zápisu do souboru lze určit, zda byl soubor modifikován. Poslední, co je v souboru WDItems.xml uloženo, je verze, ve které se položka nachází v pracovním adresáři. Záznam o položce může také obsahovat názvy pomocných souborů, pokud je položka v konfliktním stavu.

Dalším zajímavým souborem je soubor WDPaths.xml, ve kterém je uložena cesta k repository, ze které pocházejí data, a relativní cesta vzhledem ke kořenu repository odpovídajícího tomuto pracovnímu adresáři.

2.3.4 Klientská část

Klientská část, jejíž kód je obsažen ve třídě `Subcommands`, poskytuje metody, které odpovídají základním příkazům SVN Proxy, jako například checkout, commit, status, ... Tyto metody představují nejvyšší API pro ostatní aplikace. Touto aplikací může být třeba řádkový klient (jak je tomu v současné implementaci), ale mohou to být i další aplikace, například s grafickým rozhraním. Výstup v klientské části je realizován pomocí potomka třídy `OutputInterface`. Ta poskytuje rozhraní pro výstup chyby, varování (warning), informačního textu a souboru. Různé aplikace si mohou implementovat vlastní formu výstupu (konzole, okno, ...). Pomocí návrhového vzoru Decorator pak lze výstup dále uzpůsobit vlastním potřebám. Může jít o různé kombinace chování, které by se nevyplatilo realizovat pouhým děděním od potomka `OutputInterface`. Především je řeč o různých přesměrováních do logů, potlačeních varování apod.

2.4 Synchronizace s SVN

Požadavek na synchronizaci s SVN přišel až dodatečně jako rozšíření v rámci bakalářské práce. Tedy v době, kdy byl celý systém už navržen (jako systém pro správu verzí) a velká část implementována. Synchronizace vyžaduje mnoho postupů používaných při běžné práci s lokálním repository, jako je update, commit, řešení konfliktů atd. Proto bylo rozhodnuto implementovat další spojení k SVN repository a využít z velké části již existující kód. Tím je operace synchronizace poskládána z již existujících operací, jak je znázorněno na obrázcích 2.3 a 2.4.

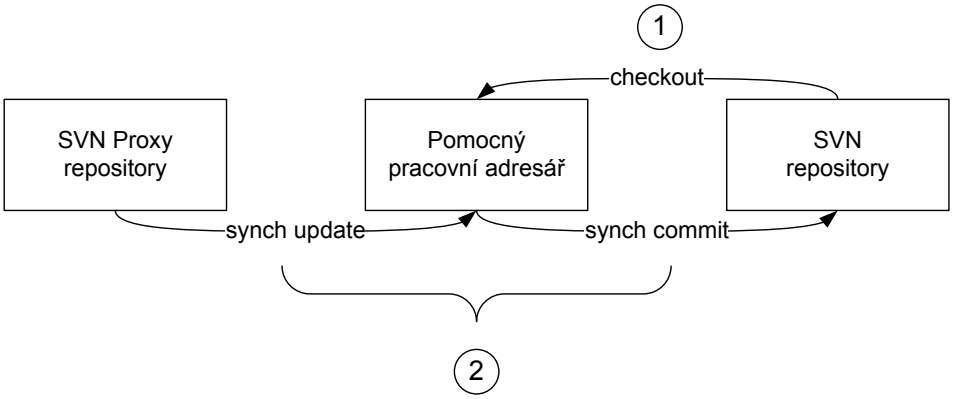
Operace prováděné při synchronizačním updatu nejsou úplně totožné s operacemi při regulérním updatu. Rozdíl je v tom, co se stane s položkou, která je updatována. Při regulérním updatu je při požadavku na smazání položka fyzicky smazána a odstraněna ze správy verzí. Při synchronizačním updatu je jenom naplánována pro smazání. Při přidání je při regulérním updatu přidána a mění se na plnohodnotnou položku, kdežto při synchronizaci je jenom naplánována pro přidání. Toto chování se liší až na úrovni jednotlivých položek, proto je třída `WorkingDirectory` parametrizována factory, která vytváří konkrétní položku podle toho, zda jde o regulérní nebo synchronizační update. Oba typy položek mají stejné rozhraní, proto je zbytek kódu nedotčen a je použit pro oba případy synchronizace. Při commitu též dochází k mírné změně chování, ale opět jen na úrovni položek. Při regulérním commitu je zakázáno, aby se jeden typ položky vyměnil za druhý, například soubor za adresář. K tomu jsou při běžné práci zapotřebí commity dva. Při synchronizaci však takový požadavek může vzniknout v rámci jednoho commitu. Proto se položky vytvářené při synchronizaci umějí s tímto problémem vyrovnat.

Dále byl zaveden speciální stav repository – synch mode. V tomto módu se jednotlivé revize počítají odlišným způsobem. Revize se stává dvojrozměrným číslem. Tedy například po revizi 14 následují revize 14.1, 14.2, 14.3, . . . Revizi $x.y$ budeme říkat podrevize revize x. Tedy například revize 14.2 je podrevize revize 14. Revize 15 bude následovat až po synchronizaci. Důvod pro existenci tohoto odlišného číslování je zřejmý. Je jím snaha nerozsynchronizovat číslování na lokálním repository a na vzdáleném repository.

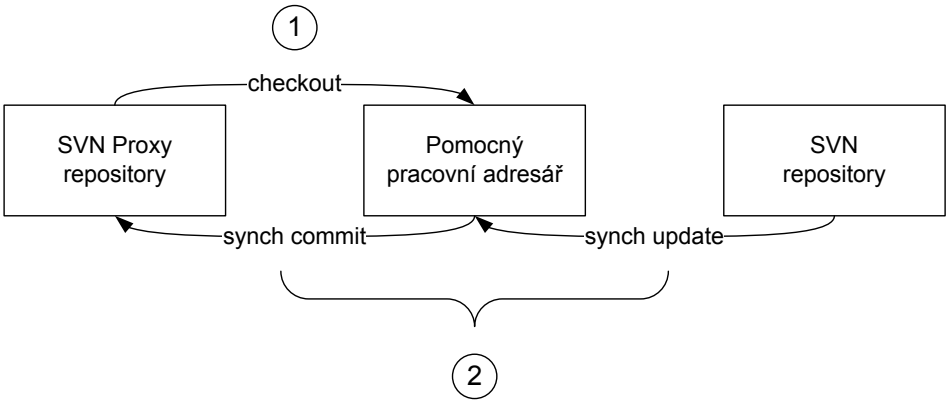
2.4.1 Export

Export je vybrán, pokud head revize lokálního repository je ostře větší než head revize vzdáleného repository. V prvním kroku, jak vidíme na obrázku 2.3, se provede checkout head revize vzdáleného repository do pomocného pracovního adresáře. Následuje opakující se krok 2, tedy update z lokálního repository a následný commit do vzdáleného repository, dokud je head revize lokálního repository ostře větší než head revize vzdáleného repository.

Nyní popíšeme, na které revize z lokálního repository se bude pomocný adresář updatovat. Výsledek updatu se pak commituje do vzdálené repository pod



Obrázek 2.3: Implementace exportu



Obrázek 2.4: Implementace importu

stejnou revizí. Budeme uvažovat posloupnost revizí lokálního repository, které jsou novější než head revize vzdáleného repository. Z této série jsou pro update vybrány jen revize, které nejsou podrevizemi. Pokud série končí podrevizí, je tato podrevize chápána jako další následující revize, která není podrevizí. Například pro revizi 12.3 je to revize 13. Máme-li tedy sérii novějších revizí 10, 11, 12, 12.1, 12.2, 12.3, výsledná série updatů bude 10, 11, 12, 13 (=12.3). Série commitů bude stejná. Po exportu je zaručeno, že se head revize synchronizovaných repository sobě rovnají.

2.4.2 Import

Import je vybrán, pokud head revize lokálního repository je ostře menší než head revize vzdáleného repository. Prováděné operace lze sledovat na obrázku 2.4. Mohou nastat dva případy:

- **Head revize lokálního repository není podrevize** – provede se checkout head revize lokálního repository do pomocného adresáře. Pak import vyústí v posloupnost updatů a commitů revizí vzdáleného repository, které jsou mladší než head revize lokálního repository. Například, je-li head revize lokálního repository 10 a head revize vzdáleného repository 14, posloupnost commitů je 11, 12, 13 a 14. Princip je podobný jako v případě exportu. Po tomto typu importu je zaručeno, že se head revize synchronizovaných repository sobě rovnají.

- **Head revize lokálního repository je podrevize** – pak nastala situace, kdy head revize lokálního repository není aktuální. Provede se checkout revize, která odpovídá revizi head podrevize lokálního repository, do pomocného adresáře. Například je-li head revize lokálního repository 14.3, provede se checkout revize 14. Pak se provede stejná posloupnost updatů a commitů jako v předešlém případě. Na konci ovšem přibude ještě jeden commit navíc. Vezme se původní neaktuální head revize lokálního repository a provede se její update na head revizi vzdáleného repository. Stejně jako při základní práci se SVN Proxy může i zde dojít k úspěšnému sloučení revizí, ale může se objevit i konflikt. V případě konfliktu se synchronizace zastaví a uživatel musí konflikt vyřešit tak, jak je zvyklý ze základní práce se správou verzí. Rozdíl bude v tom, že toto řešení konfliktů bude provádět v dočasném pracovním adresáři, který se během synchronizace vytváří. Po vyřešení konfliktů se synchronizace restartuje a dokončí. V poslední části této varianty importu je na sloučenou verzi proveden commit do lokálního repository. Head revize synchronizovaných repository se sobě nerovnají, head revize lokálního repository je navýšena oproti head revizi vzdáleného repository o poslední commit sloučené revize.

2.5 Použití návrhových vzorů

Při návrhu systému byla použita celá řada návrhových vzorů popsaných v [GHJV95]. Zde je jejich krátký výčet:

- **Abstract Factory** – je použita při implementaci přístupu k repository, kdy vytváří spojení k repository na základě adresy tohoto repository. Také je použita k vyrábění XML builderu a XML writeru. Třídy, které se umí serializovat do XML nebo z něj vytvořit, jsou parametrizovány továrnou, která writery a buildery vytváří. Pracovní adresář je parametrizován factory, která vytváří položky, které adresář obsahuje. Používají se různé pro běžnou práci a pro synchronizaci s SVN repository.

- **Factory** – potomci třídy `PlatformDependentInterface` se chovají jako factory, protože vytvářejí platformově závislého potomka třídy `ThroughDirectoryInterface`. Tento potomek implementuje procházení adresářů.

- **Singleton** – řada tříd je Singleton, například třída `Output` pro získání implementace výstupu nebo třída `RepositoryInterfaceFactory` pro získání implementace factory na vytváření spojení.

- **Adapter** – třída `SVNRepositoryConnection` je Adapter, protože přizpůsobuje rozhraní SVN repository na rozhraní SVN Proxy repository.

- **Bridge** – spojení k repository jsou návrhové vzory Bridge. Oddělují a skrývají implementaci přístupu k repository a umožňují přidání dalších implementací v budoucnu bez velkých zásahů do existujícího kódu.

- **Decorator** – používá se pro vytvoření žádoucí kombinace vlastností implementace výstupu (potomci třídy `OutputInterface`). Například pro přesměrování do logů, potlačení výpisu varování apod.

- **Proxy** – Například implementace garbage collectoru ProxyCountable, který počítá reference na svěřený pointer. Dále spousta jiných tříd sloužících k uchování pointeru na nějakou třídu zpravidla kvůli volání virtuálních funkcí.

- **Iterator** – třída `DirTree`, která uchovává položky adresářového stromu, obsahuje vlastní implementaci Iteratoru pro procházení položek v sekvenčním pořadí.

- **Memento** – třída `LockState` je Memento, vzniká při zamknutí repository a je potřeba ji uvést jako parametr při odemykání, aby repository mohlo obnovit svůj původní stav.

- **Visitor** – například použitý pro výpis obsahu adresáře v repository, implementací Visitoru lze určit množství a formát vypisovaných informací. Stejný princip je použit pro výpis logovacích záznamů.

Kapitola 3

Implementace

3.1 Výběr jazyka a vývojového nástroje

Aplikace SVN Proxy nijak kriticky nezávisí na rychlosti implementace. Důraz je spíš kladen na robustnost, flexibilnost a snadnou rozšiřitelnost. V návrhu také byla zmíněna celá řada návrhových vzorů. Proto byl výběr zúžen na programovací jazyky podporující objektově orientované paradigma. Z těchto dostupných jazyků bylo zvoleno C++, protože v době volby to byl jazyk, v kterém autor práce dosáhl nejvyšších dovedností, a protože je to jazyk velmi rozšířený a široce podporovaný v mnoha ohledech.

Protože autor používá operační systém Windows, padla volba vývojového nástroje na produkt Microsoft Visual Studio .NET 2005, který je autorovi jakožto studentovi dostupný prostřednictvím MSDN Academic Alliance pro nekomerční účely [msda]. Autor především ocenil novou vlastnost této verze, která při ladění *přímo* umožňuje zobrazovat obsah STL kontejnerů. Tím se fáze ladění významně zjednodušila.

3.2 Algoritmus na hledání rozdílů

Jak bylo popsáno v oddílu 2.3.2, v repository se ukládají starší verze souborů pomocí souhrnu editačních změn, které se musejí provést na nejnovější verzi. Přirozeným požadavkem je, aby tento souhrn editačních změn byl co nejkratší, neboli aby bylo provedeno co nejméně editací. Toto je ekvivalentní problému nalezení nejkratšího editačního skriptu – v angličtině shortest edit script (SES). Což je duální úloha k úloze nalezení nejdelší společné podsekvence dvou sekvencí A a B. V angličtině se tento problém nazývá longest common subsequence (LCS). V SVN Proxy je použita vlastní implementace algoritmu na hledání LCS tak, jak ji popsal Myers v [Mye86].

Tento algoritmus hledá cestu v editačním grafu dvou posloupností A a B a využívá přístupu dynamického programování. Algoritmus má časovou složitost

$O(ND)$, kde N je součet délek posloupností A a B a D je délka výsledného editačního skriptu. Funguje tedy velmi rychle pro posloupnosti, které se od sebe příliš neliší, což vyhovuje i naší aplikaci. V článku je taky popsáno vylepšení pomocí přístupu „rozděl a panuj“ navržené Hirschbergem tak, aby algoritmus měl lineární $O(N)$ prostorovou složitost. Toto vylepšení bylo taktéž implementováno. Implementovaný algoritmus je zapouzdřen ve třídě `Diff`.

Zjištěný editační skript je schován ve třídě `DiffItemsList`. Tato třída se chová jako `::std::vector` a k popisu změn používá tři základní editační primitiva:

- smazání řádku
- vložení řádku
- náhrada řádku za jiný

Z optimalizačních důvodů jsou po sobě jdoucí stejná primitiva slučována. Dostáváme tak rozšířené operace, které odpovídají původním primitivům, pracují však s více řádky zároveň.

3.3 Algoritmus slučování

Volba modelu správy verzí copy-modify-merge vyžaduje implementaci slučovacího algoritmu a detekce konfliktů. Byl implementován velmi jednoduchý algoritmus, který byl pro tyto účely přímo vytvořen. Nyní bude popsáno, jak tento algoritmus funguje.

Předem je dobré si uvědomit, že to, co od tohoto algoritmu požadujeme, není spojení dvou obecných souborů, ale dvou souborů, které vznikly ze společného předka. Mějme soubor x, z něj editací vznikly soubory y a z. Algoritmem na hledání rozdílů můžeme najít editační skripty E_{xy} a E_{xz}, které popisují změny vzniklé přechodem od souboru x k souborům y a z. Problém sloučení souborů y a z se nám povedlo převést na problém sloučení odpovídajících editačních skriptů E_{xy} a E_{xz}. Pokud by se nám tyto skripty povedlo sloučit do skriptu $E_{xx'}$, jeho provedením na soubor x získáme sloučený soubor x'.

Pro každou editační operaci O zavedeme její počátek $\mathrm{begin}(O)$ a konec $\mathrm{end}(O)$:

- Operace O je operace vkládání řádků a vkládá řádky za řádek r, pak $\mathrm{begin}(O) := r$ a $\mathrm{end}(O) := r$.
- Operace O je operace smazání řádků a maže řádky r až s, pak $\mathrm{begin}(O) := r - 1$ a $\mathrm{end}(O) := s$.
- Operace O je operace náhrada řádku za jiný a nahrazuje řádky r až s, pak $\mathrm{begin}(O) := r - 1$ a $\mathrm{end}(O) := s$.

Nyní budeme definovat $E_{xx'}$ a množinu konfliktů C postupně pro $\forall O \in E_{xy}$ a $\forall P \in E_{xz}$:

- Na počátku $E_{xx'} := \emptyset$ a $C := \emptyset$.
- Jsou-li O a P operace vkládání řádků a $\mathrm{begin}(O) = \mathrm{begin}(P)$, pak, pokud obě operace nevkládají stejné řádky, je $C := C \cup \{(O, P)\}$. Pokud obě operace vkládají stejné řádky, je $E_{xx'} := E_{xx'} \cup \{O\}$.
- Jsou-li O a P operace mazání a je-li $\mathrm{begin}(O) < \mathrm{end}(P) \vee \mathrm{begin}(P) < \mathrm{end}(O)$, pak $E_{xx'} := E_{xx'} \cup \{R\}$, kde R je operace mazání, která maže řádky $\min(\mathrm{begin}(O) + 1, \mathrm{begin}(P) + 1)$ až $\max(\mathrm{end}(O), \mathrm{end}(P))$. Jinak $E_{xx'} := E_{xx'} \cup \{O, P\}$.
- V ostatních případech, pokud $\mathrm{begin}(O) < \mathrm{end}(P) \vee \mathrm{begin}(P) < \mathrm{end}(O)$, pak $C := C \cup \{(O, P)\}$, jinak $E_{xx'} := E_{xx'} \cup \{O, P\}$.

Protože editační operace se v jednom skriptu nemohou překrývat, ve vlastním algoritmu stačí mít operace v editačních skriptech seřazené podle jejich počátků a oba editační skripty procházet sekvenčně. Není tedy potřeba výše uvedené podmínky testovat pro všechny dvojice operací. Po skončení vytváření $E_{xx'}$ a C, máme v $E_{xx'}$ množinu editačních operací, které spolu nekolidují (ve smyslu výše uvedených podmínek), a množinu dvojic editačních operací C, které spolu kolidují.

Pokud je množina kolizí prázdná, můžeme editační skript $E_{xx'}$ použít na soubor x a získat tak sloučený soubor x'. V případě, že byly nějaké konflikty detekovány, jsou vhodným způsobem vypsány tak, jak je popsáno v uživatelské dokumentaci. Klíčovou otázkou je volba pravidel určujících, kdy jsou operace v konfliktu. Celkem jednoduchá je otázka konfliktu dvou operací vkládání řádku, které vkládají jiné řádky za tentýž řádek. Toto je zřejmý konflikt. Obtížnějším úkolem je rozhodnout, zda je konfliktem případ, ve kterém se dvě operace smazání řádku překrývají. Nakonec bylo rozhodnuto, že je přirozenější tuto situaci za konflikt nepovažovat a do výsledného editačního skriptu vložit sloučenou operaci smazání řádku.

Otevřenou otázkou zůstává, zda jsou konfliktní operace, které se jenom dotýkají, nikoliv překrývají. Může se například jednat o situaci, kdy jeden uživatel přidá řádky za řádky, které jiný uživatel smazal. Nelze jednoznačně rozhodnout, zda o konflikt jde, či nikoliv. Proto bylo zvoleno použití benevolentnějšího přístupu a rozhodnuto konflikt v těchto případech nehlásit. Hlavní díl odpovědnosti při slučování souborů leží na uživatelích, kteří musejí ověřit, zda byly soubory správně sloučeny, a zda toto sloučení je sémanticky správné. Při paralelním editování souborů je potřeba klást důraz na vzájemnou komunikaci, kterou žádný systém nemůže plně nahradit.

Navržený algoritmus rozhodně nemá ambice podávat nejoptimálnější výsledky v každé situaci. Je pouze přímočarou implementací jednoduché myšlenky,

která říká, že konflikt nastane při průniku konkurenčních editačních operací. V budoucnu se jistě nabízí široký prostor pro rozšiřování a vylepšování tohoto algoritmu.

3.4 Zapouzdření platformově závislých operací

Při návrhu aplikace byl dodržen požadavek, aby platformově závislé operace byly zřetelně odděleny. Proto byla navržena třída `Path`, která reprezentuje platformově nezávislou cestu v sytému tím, že odstraňuje především závislost na oddělovačích v této cestě. Třída se chová jako `::std::vector` a jejími prvky jsou objekty typu `NameItem`. Třída `NameItem` definuje pomocí operátorů porovnání uspořádání a ekvivalenci prvků, které jsou taktéž platformově závislé. Na některých systémech jsou totiž názvy souborů a adresářů citlivé na velikost písmen, a na některých ne.

Dále bylo definováno rozhraní `PlatformDependentInterface`, které obsahuje následující skupiny metod:

- operátory na porovnávání třídy `NameItem`
- převod třídy `Path` na platformově závislý řetězec a také opačný převod
- metody pro práci s časem a datem
- metody pro manipulaci s adresáři a soubory – mazání, kopírovaní, vytváření, . . .
- metody pro získávání unikátních jmen
- metody konverze kódování `::std::wstring` na jiná kódování
- získání úplné cesty z relativní cesty a získání cesty aplikace

Součástí je také rozhraní `ThroughDirectoryInterface`. Potomci tohoto rozhraní by měli implementovat procházení adresářů. V současné době jsou implementovány dvě implementace: pro operační systémy řady Windows NT a pro Linux. Implementace pro Linux je pouze experimentální a nebyla tak důkladně testována jako implementace pro řadu Windows NT. Slouží pouze jako praktická ukázka toho, že rozhraní platformově závislých operací bylo navrženo dostatečně obecně, aby obsáhlo i implementace pro jiné platformy než je Windows. Implementace se volí během kompilace pomocí podmínkových direktiv preprocesoru.

3.5 Použité knihovny

Kromě standardní knihovny C++ byly použity další knihovny třetích stran.

3.5.1 Xerces

Protože bylo v návrhu aplikace rozhodnuto ukládat data ve formátu XML, vynořila se přirozená otázka, jakou máme použít knihovnu pro práci s XML. Knihovna by měla umět parsovat XML soubor a s takto rozparsovaným XML stromem by mělo být možné manipulovat. Především je řeč o operacích, jako je přidání a smazání uzlů, vytváření nových, procházení stromu atd. Knihovna by měla mít rozhraní v C++. Dalšími požadavky byla volná dostupnost a podpora více platforem.

V konečné fázi se výběr zúžil na dvě dostupné knihovny: Xerces a libxml++. Libxml++ zapouzdřuje knihovnu libxml napsanou v jazyce C. Libxml++ se zdála z počátku jednodušší na použití a pro její výběr hovořila i oblíbenost knihovny libxml v unixové komunitě. Další vlastností zapadající do konceptu vyvíjené aplikace byl fakt, že se libxml++ nevyhýbá řadě moderních vlastností C++, jako jsou prostory jmen (namespaces), STL kontejnery nebo RTTI (Runtime Type Identification) [Cum]. Proto byla vybrána knihovna libxml++.

Avšak během prvních stádií vývoje se ukázala tato volba jako chybná. Co se z počátku jevilo jako snadnost použití, to se později ukázalo jako nedostatečná funkčnost. Také později se objevily vážné problémy se stabilitou, které se projevovaly stále častějšími pády aplikace. V době použití (srpen 2005 – verze 2.10) se libxml++ ukázalo jako nepoužitelné pro seriózní práci.

Kvůli výše popsaným problémům bylo původní rozhodnutí přehodnoceno a jako knihovna pro práci s XML byl nakonec zvolen Xerces. Je to mnohem robustnější knihovna než libxml++, splňuje řadu standardů konsorcia W3C [xer]: XML 1.0 a 1.1, DOM Level 2 Core Specification a dalších. Také obsahuje neúplnou implementaci některých částí DOM Level 3.0 (např. zajímavá je podpora DOMBuilderu a DOMWriteru). Xerces nepoužívá některé z moderních vlastností C++ (šablony, RTTI) a pro reprezentaci řetězců používá ukazatel na XMLCh. V současné implementaci je použita verze 2.7. Podle vývojářů Xercesu měla ke konci roku 2005 vyjít nová verze 3.0. Při vývoji se počítalo s pozdějším použitím této nové verze. Doposud (srpen 2006) však verze 3.0 nebyla uvolněna.

3.5.2 Boost.Program_options

Klienti pro příkazovou řádku často používají pro získání vstupních informací parametry a přepínače zadávané z příkazové řádky. SVN Proxy obsahuje celou řadu přepínačů, které lze použít v krátké i dlouhé formě, a variabilní počet zadávaných parametrů. Aplikace tedy musejí obsahovat nástroje, které umožní rozparsovat vstupní přepínače a parametry, a především odhalit celou řadu chybně zadaných vstupů. Tyto nástroje musejí tedy být dostatečně robustní. Není zrovna lehké naprogramovat takovouto funkčnost tak, aby byla opravdu spolehlivá, často se jedná o úmornou práci, která se s každým projektem tvoří „na koleně“ znovu a znovu. Proto bylo rozhodnuto použít už nějaký existující nástroj.

V unixovém světě se používá funkce getopt pro krátké varianty přepínačů a getopt_long, která podporuje i dlouhé přepínače. Problémem je však dostupnost těchto funkcí na operačních systémech Windows, a navíc funkce getopt_long není standardizována. Proto volba padla na knihovnu Program_options, která je součástí C++ knihovny Boost. Knihovna Boost je volně dostupná pro celou řadu platforem, velmi dobře spolupracuje se standardní knihovnou C++ a vyznačuje se vysokou kvalitou [boo]. V SVN Proxy je použita verze 1.33.

3.5.3 SVN knihovna

Součástí specifikace SVN Proxy je synchronizace s SVN repository. Implementace vlastního přístupu k tomuto repository by byl až příliš náročný úkol. Proto bylo učiněno rozhodnutí použít oficiální knihovnu SVN. Tato knihovna je napsána v jazyce C a existuje pro ni knihovna RapidSVN, která původní knihovnu zapouzdřuje do C++. Zapouzdření ještě není dokončeno a RapidSVN se v současném stavu vývoje zaměřilo na zapouzdření klientské části API. Protože však SVN Proxy potřebuje využívat část, která implementuje přístup k repository – modul svn_ra, který do klientské vrstvy nepatří, používá se původní knihovna s rozhraním v C.

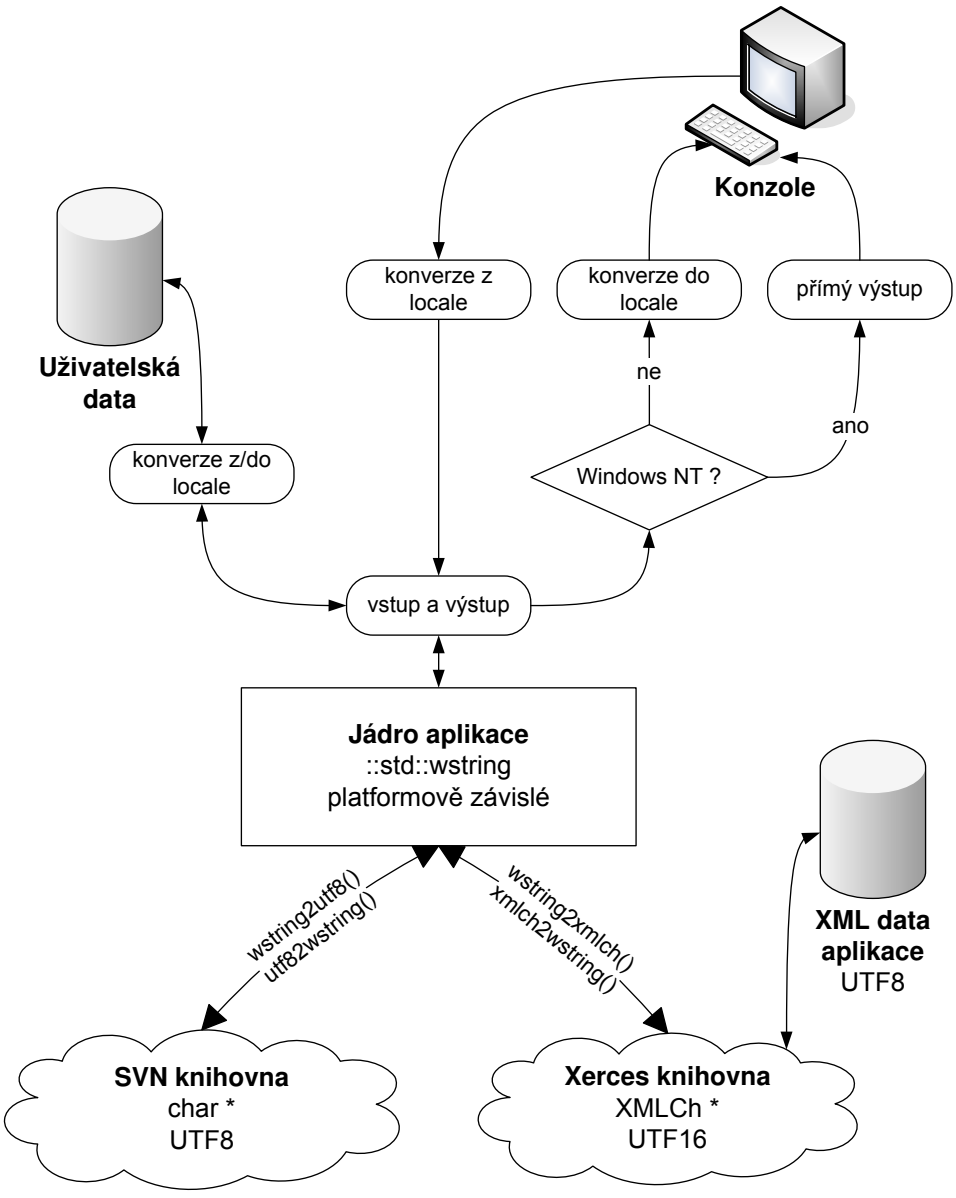
Při začlenění SVN knihovny byla celá řada funkčností zapouzdřena do C++ tříd. Jsou to následující třídy:

- `SVNEditor` – slouží k provádění commitu
- `SVNPool` – zapouzdřuje apr_pool_t sloužící ke správě paměti
- `SVNROStream` a `SVNWOStream` – streamy pro vstup a výstup
- `SVNCallbacks` – rozhraní SVN callbacků, které je potřeba v potomkovi implementovat
- `SVNCmdCallbacks` – implementace pro klienta pro příkazovou řádku
- `SVNURL` – obsahuje konverzní funkce mezi SVN URL a třídou `Path`
- `SVNUtils` – další pomocné funkce

SVN knihovna závisí na řadě dalších knihoven. V SVN Proxy byla použita SVN knihovna verze 1.3.1 a tyto pomocné knihovny: Apache Portable Runtime 0.9.7, Neon 0.25.5, Berkeley DB 4.4.20 a OpenSSL 0.9.8b.

3.6 Práce s řetězci

Při implementaci práce s řetězci byl kladen důraz na použití znakové sady Unicode. Motivací byla snaha vyhnout se problémům při používání nekompatibilních tradičních znakových sad a snaha zvýšit tak robustnost aplikace.



Obrázek 3.1: Schéma použitých formátů řetězců a jejich konverze

Na obrázku 3.1 je znázorněno, jak tato snaha byla implementována. Pro jádro aplikace byl zvolen datový typ `::std::wstring` z STL. Proti použití jiných řešení třetích stran hovořila právě přítomnost tohoto typu v STL, a tedy z tohoto faktu vyplývající výhody. Především podpora v ostatních datových typech STL (jako třeba streamy) a způsob práce používaný v STL. Nevýhodou je, že norma C++ [Int03] přímo nespecifikuje, jaké má být použito kódování, které je tak platformově, respektive implementačně závislé. To při použití na více systémech a při potřebě překódovat obsah proměnné typu `::std::wstring` do jiného kódování přináší zbytečné komplikace.

Ve spodní části obrázku 3.1 vidíme, jakým způsobem dochází k výměně řetězců mezi použitými knihovnami. Na druhém řádku u popisu knihoven je uvedeno, jaký datový typ se používá pro reprezentaci řetězců, a na třetím řádku v jakém kódování se tak děje. Na přechodech k těmto knihovnám jsou vyznačeny funkce, které se používají pro převod z jednoho kódování do druhého. Tyto převody jsou platformově závislé. Například na operačním systému řady Windows NT je konverze mezi jádrem aplikace a knihovnou Xerces triviální, neboť obě používají stejné kódování UTF16 a knihovna Xerces zaručuje, že jejich XMLCh * je roven wchar_t *.

Veškerá vnitřní data aplikace jsou ukládána ve formátu XML prostřednictvím knihovny Xerces, a ta je ukládá ve standardním kódování XML UTF8. Data, která jsou používána v komunikaci s uživatelem a jsou ukládána do souborů, jsou kódována z a do locale, které je v daný okamžik nastaveno v uživatelově operačním systému. Při výstupu na konzoli hraje roli, zda uživatelův systém je řady Windows NT. Tyto systémy umožňují přímý výpis na konzoli ve znakové sadě Unicode nezávisle na locale systému. Pokud operační systém nepatří do zmíněné řady, je použita konverze do locale (což může a nemusí být nějaké kódování Unicode). Na vstupu z konzole je vždy použita konverze z uživatelova locale.

Za zmínku ještě stojí správa řetězcových konstant v programu. Všechny řetězcové konstanty jsou uloženy jako konstantní statické položky třídy `Resources`. Jejich typ je `wchar_t *`. Řetězcové konstanty tak byly zřetelně odděleny od zbytku programu, což odstraňuje potencionální nekonzistence při modifikaci konkrétních řetězcových konstant. Taktéž je usnadněna lokalizace do jiného jazyka. Implementovaný způsob není úplně ideální, například při změně některého řetězce se musí aplikace překompilovat. Nicméně představuje dobrý základ pro budoucí sofistikovanější systémy správy řetězcových konstant.

Kapitola 4

Závěr

4.1 Splnění cíle

V rámci této bakalářské práce byla vytvořena aplikace SVN Proxy. Bylo navrženo a implementováno místní repository využívající k uložení změn rozdílů oproti starší verzi a ukládající tyto informace v XML. Taktéž byla navržena a implementována správa pracovního adresáře. K implementaci synchronizace s SVN repository bylo využito SVN knihovny a vlastní synchronizace byla implementována pomocí upravených postupů z lokální části projektu.

Při návrhu byl kladen důraz na budoucí rozšiřitelnost. Lze efektivně přidávat nová spojení k repository. Taktéž lze tvořit vlastní uživatelská rozhraní. Při návrhu byla využita celá řada návrhových vzorů za účelem zvýšení flexibility a robustnosti programu.

Uživatelské rozhraní klienta pro příkazovou řádku je odvozeno od rozhraní standardního SVN klienta. Byl tak splněn požadavek, aby SVN Proxy vzbuzoval iluzi standardního SVN.

Systém vnitřně používá znakovou sadu Unicode, čímž předchází problémům s tradičními znakovými sadami, ke konverzi dochází jen na rozhraní s uživatelem. Taktéž byly identifikovány platformově závislé operace a byly zřetelně odděleny. Aplikace obsahuje implementace těchto operací pro operační systémy řady Windows NT a Linux.

Podařilo se vytvořit plnohodnotnou aplikaci SVN Proxy, která je dobrým základem pro budoucí rozšiřování funkcí a optimalizace, které jsou jistě zapotřebí vzhledem k rozsáhlosti díla.

4.2 Možnosti dalšího vývoje

V několika bodech budou popsány možnosti dalšího vývoje aplikace:

- Implementace síťového spojení k repository, jak bylo diskutováno v oddílu 2.3.1.

- Úprava uložení informací v repository. V současné implementaci se změny v adresářové struktuře ukládají do souboru „..xml“, který je v každém adresáři. Tento způsob je z okolí viditelný a pracovní adresář toho využívá. Mnohem lepší by bylo udělat způsob ukládání informací o adresáři transparentní. Tento nedostatek způsobený nedomyšleným návrhem byl odhalen v pozdější fáze implementace. Bylo zvažováno jeho odstranění, avšak z časových důvodů k němu nedošlo. Zdržení by především způsobil zásah do již hotové části kódu pracovního adresáře, což by vedlo k ladění kódu, který již byl odladěn.

- Současná implementace diffu je pro velké soubory, které se od sebe velmi liší, pomalá. Nabízí se prostor pro celou řadu optimalizací.

- Způsob synchronizace by šel taktéž zoptimalizovat. Například vytvořením nějakého abstraktního rozhraní, tak aby synchronizační pracovní adresář mohl přímo přistupovat k souborům v repository, které by se mu jevily jako soubory v pracovním adresáři. Nemuselo by tak docházet k některým kopírováním souborů do pracovního adresáře.

- Zvážit, zda by nebylo vhodné nahradit `::std::wstring` v jádru aplikace něčím, o čem víme, jaké používá kódování.

- Soubory ukládané v XML mají zvýšené prostorové nároky. V této situaci by bylo vhodné použít nějakou metodu komprese XML. V současné době existuje mnoho způsobů komprese XML, konkrétní volba by závisela na požadavcích a praktických zkouškách.

- Z časových důvodů nebyla implementována funkce blame, která pro každý řádek souboru vypíše, z které verze pochází.

- Podpora více uživatelů. Vzhledem k robustnímu návrhu aplikace stačí jenom přidat logování uživatelů při commitu a případně nějakou správu uživatelů, pakliže bude potřeba.

- Při synchronizaci je potřeba opakovaně uvádět cestu k místnímu i vzdálenému repository. Pro větší uživatelské pohodlí by bylo vhodné si tyto dvě cesty po prvním zadání zapamatovat.

- Systém je určen pro správu textových souborů, v budoucnu by mohla být přidáná podpora binárních souborů.

- Další drobná rozšíření, jako: nápověda v programu (prozatím nahrazena podrobnou uživatelskou dokumentací), možnost zadávat verze pomocí data, příkazy copy, mkdir a move, které nejsou v podstatě nic jiného, než jenom volání již existujících operací add a delete.

Literatura

[Ber90] B. Berliner. CVS II: Parallelizing software development. In *Proceedings of the USENIX Winter 1990 Technical Conference*, stránky 341–352, Berkeley, CA, 1990. USENIX Association. Dostupné z: `http://citeseer.ist.psu.edu/berliner90cvs.html`.

[boo] Boost C++ Libraries [online, citováno 25. 7. 2006]. Dostupné z: `http://www.boost.org/`.

[Cou89] William Courington. *The Network Software Environment*. Sun Microsystems, Inc., 1989.

[CSFP04] Ben Collins-Sussman, Brian W. Fitzpatrick, and C. Michael Pilato. *Version Control with Subversion*. O'Reilly & Associates, Sebastopol, California, 2004.

[Cum] Murray Cumming. libxml++ – An XML Parser for C++ [online, citováno 25. 7. 2006]. Dostupné z: `http://libxmlplusplus.sourceforge.net/docs/manual/html/index.html`.

[EA03] Bruce Eckel and Chuck Allison. *Thinking in C++, Volume 2: Practical Programming, Second Edition*. Prentice Hall, 2003.

[Eck00] Bruce Eckel. *Thinking in C++, Volume 1: Introduction to Standard C++ (2nd Edition)*. Prentice Hall, 2000.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software (Addison-Wesley Professional Computing Series)*. Addison-Wesley Professional, 1995.

[Int03] International Organization for Standardization. *ISO/IEC 14882:2003: Programming languages — C++*. International Organization for Standardization, 2003.

[msda] MSDN-AA MEMBERSHIP [online, citováno 25. 7. 2006]. Dostupné z: `http://msdn.microsoft.com/academic/program/eula/default.aspx`.

[MSDb] The Microsoft Developer Network (MSDN) [online]. Dostupné z: http://msdn.microsoft.com/.

[Mye86] Eugene W. Myers. An O(ND) difference algorithm and its variations. *Algorithmica*, 1(2):251–266, 1986. Dostupné z: http://citeseer.ist.psu.edu/myers86ond.html.

[xer] Xerces C++ Parser [online, citováno 25. 7. 2006]. Dostupné z: http://xml.apache.org/xerces-c/.

Příloha A

Uživatelská dokumentace

A.1 Úvod

Text této uživatelské dokumentace je vytvořen na základě knihy Version Control with Subversion napsané autory Ben Collins-Sussman, Brian W. Fitzpatrick a C. Michael Pilato. Kniha je dostupná v elektronické podobě na adrese `http://svnbook.red-bean.com/` pod licencí Creative Commons Attribution License, jejíž plné znění je dostupné na adrese `http://creativecommons.org/licenses/by/2.0/`. Tato licence umožňuje knihu volně kopírovat a distribuovat a stejně tak vytvářet další díla z ní vycházející. Některé části výše zmíněné knihy byly přímo přeloženy, jiné části posloužily jenom jako inspirace svou strukturou výkladu.

Dokumentace obsahuje čtyři další kapitoly. V kapitole Základní návrh se uživatel seznámí s architekturou systému SVN Proxy a s některými obecnými zásadami týkajícími se správy verzí. Tato kapitola je obzvláště vhodná pro uživatele, kteří se nikdy nesetkali s žádným systémem pro správu verzí. Další kapitola, Základní práce, provádí uživatele jedním běžným cyklem vytvoření nové revize. Kapitola Synchronizace s SVN popisuje pokročilou vlastnost systému a její název mluví sám za sebe. Poslední kapitola je referencí a popisuje všechny přepínače a příkazy SVN Proxy. Při čtení předešlých kapitol je vhodné do reference nahlížet, neboť jednotlivé příkazy a přepínače nejsou v těchto kapitolách podrobně rozebírány.

A.2 Základní návrh

Tato krátká kapitola vysvětluje základní návrh správy verzí použitý v SVN Proxy a je určena především nováčkům ve správě verzí.

Repository

SVN Proxy je centralizovaný systém pro správu zdrojových textů. Jeho jádrem je repository, což je centrální úložiště dat. Repository uchovává data ve formě adresářového stromu. Klienti se mohou připojit k repository a číst a zapisovat do uložených souborů. Zápisem dat klient tato data činí dostupnými pro ostatní klienty. Čtením dat pak informace získává.

Zatím tento popis vypadá jako obyčejný souborový server. Repository je ve skutečnosti druhem serveru, ale není to obyčejný souborový server. Tím, čím se odlišuje, je schopnost *zapamatovat si* každou změnu způsobenou zápisem do souboru nebo změnu adresářového stromu.

Když klient čte data z repository, zpravidla čte poslední verzi adresářového stromu. Ovšem klient má také schopnost pracovat s jakýmkoliv předešlým stavem uloženého adresářového stromu.

Copy-modify-merge model

Základní úlohou systému pro správu verzí je umožnit sdílení a modifikování dat více uživateli. K naplnění tohoto cíle může být zvoleno několik modelů. Jedním z nich je model copy-modify-merge, což bychom mohli přeložit jako „model zkopíruj-změň-sluč“.

V tomto modelu se každý uživatel připojí k repository a stáhne si pracovní kopii. V SVN Proxy má pracovní kopie podobu pracovního adresáře. Tento pracovní adresář je lokální podobou souborů a adresářů v repository. Uživatelé mohou pracovat paralelně a měnit svoje soukromé pracovní adresáře. Nakonec jsou všechny změny sloučeny v novou konečnou verzi. Systém správy verzí může asistovat u slučování, ale hlavní díl práce a odpovědnosti je ponechán na člověku. Ten musí zajistit, že výsledná verze bude korektní.

Možná, že copy-modify-merge model zní složitě, v praxi ale funguje velmi hladce. Uživatelé mohou pracovat paralelně, nemusejí na sebe čekat. Pokud pracují na jednom souboru a snaží se vyhnout změnám, které by se překrývaly, je výskyt konfliktů vzácný. Čas strávený řešením konfliktů je menší, než kdyby na stejném souboru nemohlo zároveň pracovat více uživatelů a museli by na sebe čekat.

Alfou a omegou tohoto přístupu je komunikace mezi uživateli. Pokud uživatelé komunikují málo, množství konfliktů roste. Žádný systém nemůže donutit uživatele k ideální komunikaci.

SVN Proxy je v současné podobě zamýšlen jako jednouživatelský systém s lokálním repository. Nabízí se tedy otázka, proč je zde model copy-modify-merge zmiňován. Odpovědí může být několik. Systém byl navržen tak, aby tento model splňoval, což umožňuje jeho budoucí rozšíření na víceuživatelský se vzdáleným repository. Nicméně i v současné podobě může být takto používán. Repository může být na síťovém disku a může k němu přistupovat i více uživatelů. Sys-

tém tomu žádným způsobem nebrání. Omezením však je, že prozatím nejsou nikde zaznamenávány změny podle uživatelů, kteří je provedli. Avšak hlavním důvodem, proč je zde model copy-modify-merge zmiňován, je jeho schopnost synchronizace se vzdáleným SVN repository. Při tomto způsobu práce se nám již všechny výše popsané situace objevují. Uživatel při synchronizaci se vzdáleným repository, které je typicky používáné mnoha dalšími uživateli, musí zahrnout případné změny, které nastaly od poslední synchronizace. Může vzniknout konflikt a obecně je k tomuto způsobu práce potřeba přistupovat stejně jako v případě copy-modify-merge modelu.

Pracovní adresář

O pracovním adresáři jsme se už zmínili. Pracovní adresář je běžný adresář v souborovém systému lokálního počítače. Obsahuje soubory a další podadresáře. Soubory mohou být editovány, a pokud jde o zdrojové soubory, můžeme z nich kompilovat tak, jak jsme zvyklí. Pracovní adresář by se dal přirovnat k osobnímu písečku: Můžeme si na něm provádět své změny, a dokud nechceme, ostatní se o nich nedozví. Můžeme mít i více pracovních adresářů patřících k jednomu repository.

Až usoudíme, že provedené změny jsou zralé k publikování, systému to oznámíme a změny budou odeslány do repository.

Pracovní adresář obsahuje i některé speciální soubory, které SVN Proxy pomáhají spravovat pracovní adresář. Tyto soubory jsou uloženy v adresáři .ver, který se nachází v každém podadresáři pracovního adresáře.

Je běžným jevem, že repository obsahuje několik různých projektů. Každý projekt obvykle sídlí ve svém podadresáři. V takovémto rozvržení projektů bude typicky pracovní adresář odpovídat nějakému podadresáři v repository, nikoliv celému repository.

Revize

Pokaždé, kdy repository přijme sadu změn, vytvoří se nový stav adresářového stromu v repository, tento stav se nazývá *revize*. Ke každé revizi je přiřazeno jednoznačné číslo, které je větší než číslo přiřazené předešlé starší revizi. Prvotní revize má číslo 0 a obsahuje pouze prázdný kořenový adresář.

Je dobré si uvědomit, že číslo revize přísluší *celému stromu*, nikoliv jednotlivým souborům. Každé číslo revize tak odpovídá konkrétnímu obrazu celého adresářového stromu po provedení jedné série změn. Pokud uživatel mluví o revizi 5 souboru foo.c, ve skutečnosti myslí soubor foo.c tak, jak vypadá v revizi 5. Není pravda, že by se soubor v revizi N a M musel od sebe lišit.

Stejně tak není nutné, aby pracovní adresář obsahoval soubory, které by odpovídaly jedné revizi. Ve skutečnosti může obsahovat mnoho souborů z různých revizí. Například se tak může stát, pokud uživatel změní jeden soubor a odešle

jen tento soubor do repository. Pak daný soubor má vyšší revizi než všechny ostatní.

Každý soubor v pracovním adresáři se může nacházet v jednom z těchto čtyř stavů:

nezměněný a aktuální Soubor v pracovním adresáři nebyl změněn a ani žádná změna nebyla odeslána do repository. Příkaz **ver commit** nedělá nic a **ver update** taktéž.

lokálně změněný a aktuální Soubor byl změněn v pracovním adresáři, žádná změna však nebyla odeslána do repository. Příkaz **ver commit** úspěšně odešle změny do repository a **ver update** nedělá nic.

nezměněný a neaktuální Soubor nebyl změněn v pracovním adresáři, ale byl změněn v repository. Příkaz **ver commit** nedělá nic a **ver update** stáhne poslední změny a zaktualizuje soubor v pracovním adresáři.

lokálně změněný a neaktuální Soubor byl změněn jak v pracovním adresáři, tak v repository. **ver commit** selže, protože neumí zpracovat změněný neaktuální soubor. Soubor musí být nejdříve aktualizován; **ver update** se pokusí nové změny sloučit s našimi. Může se mu to povést, ale výsledkem může být i konflikt.

A.3 Základní práce

Prvotní checkout

Ve většině případů začneme naše používání SVN Proxy stažením – *checkout* – projektu. Checkout vytvoří lokální kopii na našem počítači. Tato kopie bude obsahovat HEAD revizi (tj. nejaktuálnější revizi) repository, které jsme specifikovali v příkazu checkout:

```
$ ver checkout /tmp/repos
A  trunk/file
A  trunk/file2
A  trunk/subdir
...
Checked out revision 10
```

Ačkoli výše uvedený příklad stáhne kořenový adresář, můžeme si stáhnout kterýkoliv podadresář tím, že jej uvedeme v zadávané cestě:

```
$ ver checkout /tmp/repos/subdir
A  subdir/readme-dblite.html
A  subdir/dtd
A  subdir/dtd/dblite.dtd
...
Checked out revision 10.
```

Nyní nám již nic nebrání v práci s pracovním adresářem. Pracovní adresář není nic jiného než skupina souborů a adresářů. Můžeme ji měnit, mazat, můžeme dokonce smazat celý pracovní adresář a už se o něj nestarat. Je ale potřeba mít stále na paměti, že když provádíme změnu struktury uvnitř pracovního adresáře, musíme o tom SVN Proxy dát vědět. Proto v situaci, kdy chceme třeba nějakou položku smazat, měli bychom použít **ver delete**.

Dokud nechceme provést commit nových souborů nebo adresářů, nebo už existujících, ale změněných, není potřeba informovat repository o jakýchkoliv změnách, které jsme v pracovním adresáři udělali.

K čemu je .ver adresář

Každý pracovní adresář obsahuje .ver podadresář. Jedná se o administrativní prostor, ve kterém SVN Proxy skladuje všechny potřebné informace, aby mohlo spravovat pracovní adresář. Pamatuje si, v jaké revizi byl soubor stažen a kdy se tak stalo. Také si pamatuje, které položky jsou naplánovány pro smazání nebo přidání, nebo zda je nějaký soubor v konfliktním stavu. V žádném případě by neměl být tento administrativní adresář smazán a měl by být modifikován jen ve výjimečných případech.

Základní rozvrh práce

V následujících částech postupně probereme běžný postup práce tak, jak se vyskytuje při každodenním používání.

Typický postup probíhá takto:

- Aktualizace pracovního adresáře
  - **ver update**
- Provádění změn
  - **ver add**
  - **ver delete**
- Prohlížení změn

  - **ver status**
  - **ver diff**

- Zahrnutí jiných změn do pracovního adresáře
  - **ver update**
  - **ver resolved**

- Commit – odeslání změn
  - **ver commit**

Aktualizace pracovního adresáře

Když pracujeme na projektu, který je vyvíjen týmem několika lidí, bude často potřeba aktualizovat pracovní adresář, aby obsahoval změny ostatních vývojářů, které se objevily od našeho posledního updatu. Pokud na projektu pracujeme sami, nebudeme update používat tak často. Nutnost aktualizovat pracovní adresář se může objevit, pokud používáme synchronizaci našeho lokálního repository se vzdálenou SVN repository.

Použití **ver update** může vypadat třeba takto:

```
$ ver update
U  foo.c
U  bar.c
Updated to revision 2
```

V tomto příkladu někdo změnil soubory foo.c a bar.c od posledního updatu. SVN Proxy tyto změny přeneslo, aby náš pracovní adresář tyto změny obsahoval.

Vytváření změn v pracovním adresáři

Nyní se můžeme dát do práce a měnit náš pracovní adresář. Obvykle bývá nejvýhodnější rozmyslet si určitou sérii změn, jako třeba přidání nové vlastnosti, opravu chyby apod. Změny, které můžeme provést v pracovním adresáři, jsou tyto:

Změna souboru Je to nejjednodušší změna. Nepotřebujeme SVN Proxy jakkoliv informovat o tomto kroku.

Změna ve stromové struktuře Můžeme SVN Proxy požádat, aby některé adresáře nebo soubory označil pro smazání nebo přidání. Zatímco se tyto

změny odehrají v pracovním adresáři okamžitě, repository zůstane nedotčeno, dokud neprovedeme commit. Označení pro přidání nebo smazání se provádí příkazy **ver add** a **ver delete**. Změny ve stromové struktuře by měly být prováděny *výhradně* tímto způsobem.

Varování

Systém je určen pouze pro správu textových souborů, proto bychom neměli pomocí **ver add** přidávát binární soubory. Repository pravděpodobně nebude schopno uchovat tento soubor ve správné podobě.

Prohlížení změn

V okamžiku, kdy jsme dokončili všechny změny a budeme chtít provést commit, obvykle vyvstane potřeba prohlédnout si, co všechno jsme změnili. Díky prohlédnutí všech změn lze uvést mnohem přesnější logovací záznam. Můžeme také objevit, že jsme na něco zapomněli. Přehled změněných položek můžeme získat příkazem **ver status** a přehled konkrétních změn pak příkazem **ver diff**.

ver status

Pokud spustíme **ver status** v kořenu pracovního adresáře bez dodatečných argumentů, zobrazí všechny změněné soubory a změny ve struktuře adresáře. Následuje příklad:

```
$ ver status wc
M       file
A       file2
D       dir
C       dir/file
R       dir/dir2/file
```

První písmeno každého řádku určuje stav položky. V mnoha případech je účelné použít přepínač --verbose pro zobrazení detailnějších informací. Další podrobnosti viz reference.

ver diff

Dalším způsobem, jak můžeme zkoumat naše změny, je použití příkazu **ver diff**. Pomocí něj můžeme *přesně* zjistit, co všechno jsme změnili, pokud příkaz pustíme na nějaký soubor v pracovním adresáři bez dalších parametrů. Výstup používá běžný formát diff utilit:

```
$ ver diff main.cpp
33c33
<             locale::global(locale(”“));
---
>             locale::global(locale(”cs_CZ”));
52,53d51
<             callbacks = 0;
<             throw;
```

Tento výstup vznikl porovnáním souboru v pracovním adresáři s head revizí tohoto souboru v repository. Přirozeně nemusíme porovnávat jen s head revizí, ale i s jakoukoli jinou revizí specifikovanou přepínačem --revision.

Řešení konfliktů

Při práci se SVN Proxy může dojít k nepříjemné situaci – konfliktu. Prvním předpokladem je následující situace: změníme ve svém pracovním adresáři nějaký soubor a někdo jiný tak učiní také a tento změněný soubor nahraje do repository. V tento okamžik jsme nuceni použít update, abychom mohli na náš změněný soubor použít commit. Během updatu se SVN Proxy pokusí změny provedené naším kolegou zapracovat do našeho (taktéž změněného) souboru v pracovním adresáři. Můžeme dostat dva různé výsledky:

```
$ ver update
U  INSTALL
G  README
C  bar.c
Updated to revision 46
```

Písmeno G znamená, že změny byly úspěšně zaneseny (soubor byl spojen – merGed). Spolu s obyčejným updatem obsahu souboru (písmeno U) výsledek spojení neznamená žádný problém. Soubory absorbovaly požadované změny. V případě G to znamená, že požadované změny nijak nekolidovaly s našimi lokálními změnami.

Ovšem písmeno C značí konflikt. To znamená, že změny z repository kolidují s našimi změnami. Bude potřeba ručního zásahu.

Během konfliktu se stanou tři věci:

- SVN Proxy zobrazí C během updatu a zapamatuje si, že se soubor dostal do konfliktního stavu.

- Do souboru jsou umístěny speciální *označovače konfliktu*, které viditelně označují překrývající se oblasti.
- Pro každý soubor v konfliktním stavu jsou vytvořeny tři speciální soubory, které nejsou pod správou verzí:

  filename.mine Toto je soubor, který se nacházel v pracovním adresáři před započetím updatu. Obsahuje naše lokální změny.

  filename.rOLDREV Tento soubor odpovídá souboru, jehož zeditováním vznikl předešlý soubor.

  filename.rNEWREV Toto je soubor, který se nachází v repository a obsahuje změny, které kolidovaly s našimi.

  OLDREV je revize dotyčného souboru v pracovním adresáři a NEWREV je revize souboru v repository.

V následujícím přikladu náš kolega změnil soubor file.txt a změny commitem nahrál do repository. My jsme tento soubor také změnili a po vyvolání updatu jsme dostali konflikt:

```
$ ver update
C  file.txt
Updated to revision 2
$ ls -1
file.txt
file.txt.mine
file.txt.r1
file.txt.r2
```

V tento okamžik nám SVN Proxy *nedovolí* provést commit dokud je soubor v konfliktním stavu – to je ekvivalentní existenci tří výše zmíněných souborů.

```
$ ver commit --message ”Add a few more things”
Repository was locked
error during checking files to commit: file.txt: \
remains in conflict
fatal error occurred: we can’t continue
Repository was unlocked
```

Pokud chceme vyřešit konflikt, musíme udělat jednu z následujících věcí:

- Sloučit konfliktní text „ručně“ (prozkoumáním a zeditováním označovačů konfliktu v konfliktním souboru).

- Překopírovat jeden z pomocných souborů přes konfliktní soubor v našem pracovním adresáři.

V okamžiku, kdy jsme konflikt vyřešili, je vhodné o tom informovat SVN Proxy pomocí příkazu **ver resolved**. Tento příkaz odstraní tři pomocné soubory a odstraní tak konfliktní stav souboru:

```
$ ver resolved file.txt
Resolved conflicted state of file.txt
```

Odeslání změn

A je to tu. Poslední fáze jednoho pracovního cyklu. Jakmile jsou změny dokončeny a sloučili jsme všechny změny z repository, nastává čas naše změny odeslat do repository.

Příkaz **ver commit** pošle všechny naše změny do repository. Když provádíme commit, musíme uvést *logovací záznam*, který popisuje, jaké jsme udělali změny. Náš logovací záznam bude připojen k nové revizi, která se commitem vytvoří. Zanesené logovací záznamy si lze prohlédnout pomocí **ver log**. Logovací záznam se uvádí pomocí parametru --message (nebo -m):

```
$ ver commit --message \
”Corrected number of cheese slices.”
Repository was locked
Journal was saved
Sending        file.txt
Committed revision 3
Journal was deleted
Repository was unlocked
```

Repository nezajímá, jestli naše změny dávají nějaký smysl, jenom se ověří, že nikdo jiný nezměnil některý z posílaných souborů a my jsme si toho nevšimli. Pokud se tak *stalo*, commit neproběhne a zobrazí se informace o této skutečnosti:

```
$ ver commit --message ”Add another rule”
Repository was locked
error during checking files to commit: .\rules.txt: \
out of date, use update
Repository was unlocked
fatal error occurred: we can’t continue
```

V takovém případě musíme použít **ver update**, uvážit sloučení, konflikty a pokusit se o commit znovu.

Protože je commit jediná funkce, která mění repository a vytváří novou revizi, je potřeba v SVN Proxy použít dodatečné mechanismy, které zajistí bezproblémovost této operace. Především se musíme vyvarovat dvou problémů:

Během provádění commitu by se někdo jiný snažil také o commit

SVN Proxy k řešení tohoto problému zavádí mechanismus zámků. Repository je před každým pokusem o commit zamčeno pro zápis. Dokud není tento zámek zrušen, nemůže nikdo další s repository pracovat. Zámek pro zápis je po úspěšném commitu zrušen. Průběh zamykání a odemykání můžeme vidět na výše uvedených příkladech commitů.

Při přerušení commitu zámek pro zápis zůstává. Zaručuje tak, že repository zůstane ve stejném stavu, dokud se uživatel nepokusí přerušený commit obnovit. Může však nastat situace, kdy se uživateli z rozličných důvodů commit obnovit nepodaří. To je velmi nepříjemná situace, neboť se repository nachází pravděpodobně ve stavu, kdy obsahuje jen část zamýšlených změn. Zde je nutná dobrá komunikace mezi uživateli k dodatečnému zajištění korektnosti obsahu změn. Opravný commit může provést původní uživatel nebo i jiný, pokud původní má problémy, které mu úplně znemožňují provádět commity. Během řešení tohoto problému je potřeba zrušit zámek pro zápis, aby byly možné opravné úpravy repository. K tomu lze použít příkaz **ver unlock**, který zruší všechny zámky, které repository má:

```
$ ver commit -m”some fixies”
error during locking a repository for writing:
d:\rep3:already locked from <unknown>
fatal error occurred: we can’t continue

$ ver unlock
```

Repository se také zamyká pro čtení při jakékoli operaci, která přistupuje k repository, ale nemodifikuje ji. Repository může být pro zápis zamknuto paralelně mnoha uživateli. Účelem tohoto locku je zamezit někomu před zamknutím pro zápis, dokud někdo z repository čte.

Commit je přerušen a repository obsahuje jen část změn

K řešení tohoto problému byl zaveden mechanismus journalu. To je soubor, který se vytváří při commitu v pracovním adresáři. Obsahuje seznam souborů

a adresářů, které budou odesílány do repository. V průběhu commitu se vymazávají položky, které už byly odeslány. Journal nám umožňuje obnovit přerušený commit tím, že při restartu commitu odešle položky, které ještě nebyly zpracovány. Restart commitu se provádí příkazem **ver recommit**. Pouští se na stejný adresář, pro který byl vyvolán přerušený commit.

Přítomnost journalu nám znemožňuje vyvolat nový commit, dokud není starý dokončen. V případě nemožnosti commit dokončit může být journal prostě smazán pomocí příkazu **ver delete_journal**. Odstraněním journalu už můžeme provádět nové commity, musíme mít však na paměti, že se v repository pravděpodobně nachází jen část zamýšlených změn. Řešíme tak stejný problém, který byl už zmíněn v předešlém bodě.

SVN Proxy umožňuje synchronizaci s SVN. Vedlejším efektem této vlastnosti je možnost používat přímo (ne přes synchronizaci) SVN repository jako repository systému SVN Proxy. Systém SVN umožňuje provést commit, který je atomický, proto jsou dva výše uvedené mechanismy zbytečné. Při provádění commitu do SVN Proxy nedochází k žádnému zamykání, a i když je journal vytvořen (z důvodu vnitřní architektury systému), je při přerušení commitu automaticky smazán. Recommit tedy není možný, a v tomto případě ani nedává smysl. Přímé používání SVN Proxy pro přístup k SVN repository není příliš vhodné.

A.4 Synchronizace s SVN

Pokročilou funkcí SVN Proxy je synchronizace se vzdáleným SVN repository. Uživatel má lokální repository na svém lokálním počítači. Commity provádí do tohoto repository. Po určité době provede synchronizaci se vzdálenou SVN repository, kdy se jeho série změn projeví jako jedna velká změna v tomto vzdáleném repository. Do jeho lokální repository jsou případně nataženy změny ze vzdáleného SVN repository. Synchronizace se provádí příkazem **ver synchronize**. Příkaz sám rozhodne, zda se provede export, import, nebo není potřeba žádné synchronizace.

Synch mode

Synch mode je stav repository. V tomto módu se jednotlivé revize počítají odlišným způsobem. Revize se stává dvojrozměrným číslem. Tedy například po revizi 14 následují revize 14.1, 14.2, 14.3, ... Revizi $x.y$ budeme říkat podrevize revize x. Proto revize 14.2 je podrevize revize 14. Revize 15 bude následovat až po synchronizaci. Důvod pro existenci tohoto odlišného číslování je zřejmý. Je jím snaha nerozsynchronizovat číslování na lokálním repository a na vzdáleném repository. Pokud se rozhodneme používat synchronizaci, je nanejvýš vhodné synch mode zapnout.

Synch mode se mění pomocí příkazu **ver change_synch_mode**. Aktuální stav lze zjistit příkazem **ver rep_info**.

Export

Export je vybrán, pokud head revize lokálního repository je ostře větší než head revize vzdáleného repository. Budeme uvažovat posloupnost revizí lokálního repository, které jsou novější než head revize vzdáleného repository. Z této série jsou commitovány jen revize, které nejsou podrevizemi. Pokud série končí podrevizí, je tato podrevize chápána jako další následující revize, která není podrevizí. Například pro revizi 12.3 je to revize 13. Máme-li tedy sérii novějších revizí 10, 11, 12, 12.1, 12.2, 12.3, výsledná série commitů bude 10, 11, 12, 13 (=12.3). Po exportu je zaručeno, že se head revize synchronizovaných repository sobě rovnají.

V následujícím příkladu provedeme export. Head revize lokálního repository je 10.3, vzdáleného repository 10:

```
$ ver rep_info
head revision: 10.3
synch mode: enabled

$ ver synchronize d:\x_temp d:\rep3 file:///d:/rep4
creating a new synch dir
start of export
checking out from a ver repository the revision 10
A       d:\x_temp\dir2
A       d:\x_temp\dir2\file
A       d:\x_temp\file
synch to the revision: 11
A       d:\x_temp\dir
A       d:\x_temp\dir\file
U       d:\x_temp\file
Repository was locked
Journal was saved
Adding          d:\x_temp\dir
Sending         d:\x_temp\file
Adding          d:\x_temp\dir\.
Adding          d:\x_temp\dir\file
Committed revision 11
Journal was deleted
Repository was unlocked
deleting the synch dir
all is done
```

Vidíme, že se do vzdáleného repository exportovala revize 11, což je revize obsahem odpovídající revizi 10.3 lokálního repository.

Import

Import je vybrán, pokud head revize lokálního repository je ostře menší než head revize vzdáleného repository. Mohou nastat dva případy:

- **Head revize lokálního repository není podrevize** – pak import vyústí v posloupnost commitů revizí vzdáleného repository, které jsou mladší než head revize lokálního repository. Například, je-li head revize lokálního repository 10 a head revize vzdáleného repository 14, posloupnost commitů je 11, 12, 13 a 14. Po tomto typu importu je zaručeno, že se head revize synchronizovaných repository sobě rovnají.
- **Head revize lokálního repository je podrevize** – pak nastala situace, kdy head revize lokálního repository není aktuální. Provede se stejná posloupnost commitů jako v předešlém případě. Na konci ovšem přibude ještě jeden commit navíc. Vezme se původní neaktuální head revize lokálního repository a provede se její update na head revizi vzdáleného repository. Stejně jako při základní práci se SVN Proxy zde může dojít k úspěšnému sloučení revizí, ale může se objevit i konflikt. V případě konfliktu se synchronizace zastaví a uživatel musí konflikt vyřešit tak, jak je zvyklý ze základní práce se správou verzí. Rozdíl bude v tom, že toto řešení konfliktů bude provádět v dočasném pracovním adresáři, který se během synchronizace vytváří – viz reference jednotlivých příkazů. Po vyřešení konfliktů můžeme synchronizaci restartovat a dokončit. V poslední části této varianty importu je na sloučenou verzi proveden commit do lokálního repository. Head revize synchronizovaných repository se sobě nerovnají, head revize lokálního repository je navýšena oproti head revize vzdáleného repository o poslední commit sloučené revize.

Na příkladu si ukážeme, jak vypadá import, kdy head revize lokálního repository je podrevize (druhý případ importu) a kdy nám vznikl konflikt:

```
$ ver synchronize d:\x_temp d:\rep3 file:///d:/rep4
creating a new synch dir
start of import
checking out from a ver repository the revision 8
A       d:\x_temp\dir
A       d:\x_temp\dir2
```

```
A       d:\x_temp\dir2\file
A       d:\x_temp\file
synch to the revision: 9
D       d:\x_temp\dir\.
D       d:\x_temp\dir
Repository was locked
Journal was saved
Deleting        d:\x_temp\dir
Deleting        d:\x_temp\dir\.
Committed revision 9
Journal was deleted
synch to the revision: 10
U       d:\x_temp\file
Repository was locked
Journal was saved
Sending         d:\x_temp\file
Committed revision 10
Journal was deleted
We must merge
checking out from a ver repository the revision 8.1
A       d:\x_temp\dir
U       d:\x_temp\file
merging with the revision 10
C       d:\x_temp\file
D       d:\x_temp\dir\.
D       d:\x_temp\dir
Repository was locked
error during checking files to commit: \
d:\x_temp\file: remains in conflict
fatal error occurred: we can’t continue

$ ver resolved d:\x_temp\file
Resolved conflicted state of file

$ ver synchronize d:\x_temp
start of import
We must merge
Repository was locked
Journal was saved
Sending         d:\x_temp\file
Committed revision 10.1
Journal was deleted
deleting the synch dir
all is done
```

Ve výše uvedeném příkladu se importovaly do lokálního repository mladší revize 9 a 10 ze vzdáleného repository. Import pokračuje spojením neaktuální head revize 8.1 lokálního repository s head revizí 10 vzdáleného repository. Při slučování naneštěstí došlo ke konfliktu. Uživatel musí tento konflikt vyřešit v pomocném pracovním adresáři d:\x_temp, což je adresář, který zadal jako parametr při spuštění příkazu **ver synchronize**. Po vyřešení konfliktu můžeme synchronizaci restartovat zápisem zkrácené syntaxe příkazu **ver synchronize** a import dokončit commitem sloučené revize do lokálního repository.

Omezení synchronizace

Základní omezení synchronizace s SVN repository plynou z návrhu lokálního repository. Lokální repository bylo navrženo pro skladování zdrojových textů, a proto není schopno uchovávat binární soubory. Synchronizace binárního souboru buď skončí chybou, nebo se podaří, ale pak je pravděpodobně tento soubor uložen v nesprávné podobě. Taktéž lokální repository vůbec nezná pojem *properties*, které se používají v SVN. Omezení SVN Proxy jsou tedy zřejmá: Nelze synchronizovat binární soubory a nelze pracovat s properties.

A.5 Reference SVN Proxy

Použití klienta pro příkazovou řádku je jednoduché, napíšeme **ver**, následuje příkaz, který chceme použít, přepínače a případně cesty, na kterých bude zadaný příkaz provádět příslušné operace.

Přepínače

Použité přepínače mají globální charakter, což neznamená nic jiného, než že u každého příkazu znamenají totéž. Například: --verbose (-v) vždy znamená „rozšířený výpis“ nezávisle na použitém příkazu. Proto budou všechny přepínače popsány na začátku. U jednotlivých příkazů je už jenom uvedeno, které přepínače příkaz podporuje.

--directory (-d) Pokud je v cestě uveden adresář, bude uvažován tento adresář jako položka sídlící v nadřazeném adresáři nikoliv jako položka obsahující další položky. Dojde tak k potlačení rekurzivního chování.

--force Vnutí určité chování. Při normálním použití jsou některé akce zakázány, použitím force přepínače říkáte: „ano, vím, co dělám, a vím, jaké to bude mít důsledky, chci pokračovat“.

--message (-m) ***MESSAGE*** Umožňuje specifikovat logovací záznam commitu. Například:

```
$ ver commit -m ”Něco jsem změnil”
```

--non-recursive (-N) Vypne rekurzivní chování příkazu.

--revision (-r) ***REV*** Určuje revizi, s kterou bude příkaz pracovat. Revize může být zadána číslem, slovem HEAD a u některých příkazů i rozsahem revizí. Rozsah se zadává zadáním dvou revizí oddělených dvojtečkou. Například:

```
$ ver log -r 1729
$ ver log -r 1729:HEAD
$ ver log -r 1729:1744
$ ver log -r 14.3
```

--svn_url Říká, že tam, kde je očekávána cesta k repository, je uvedena URL odpovídající SVN repository. Pokud chceme pracovat se SVN repository, je nutné tento přepínač uvést.

--verbose (-v) Požaduje, aby klient vytiskl „rozšířený výpis“. To může vést k výpisu dalších položek nebo detailních informací o každém souboru.

Příkazy

Název

ver add – Přidá soubory nebo adresáře.

Předpis

```
ver add PATH...
```

Popis

Přidá soubory nebo adresáře do pracovního adresáře a naplánuje jejich přidání do repository. Přidání se provede při následujícím commitu.

Alternativní názvy

žádné

Mění

Pracovní adresář

Přistupuje k repository

ne

Přepínače

```
--non-recursive (-N)
```

Příklady

Přidá soubor do pracovního adresáře:

```
$ ver add foo.c
A         foo.c
```

Při přidávání adresáře pomocí **ver add** se standardně používá rekurze:

```
$ ver add testdir
A         testdir
A         testdir/a
A         testdir/b
A         testdir/c
A         testdir/d
```

Tomuto chování se lze vyhnout a přidat adresář bez jeho obsahu:

```
$ ver add --non-recursive otherdir
A         otherdir
```

Název

ver cat – Vypíše obsah uvedeného souboru.

Předpis

```
ver cat TARGET...
```

Popis

Vypíše obsah souboru o dané cestě, která odpovídá buď souboru v pracovním adresáři, nebo je to cesta k souboru v repository. Pro výpis obsahu adresáře viz **ver list**.

Alternativní názvy

žádné

Mění

nic

Přistupuje k repository

ano

Přepínače

```
--revision (-r) REV
--svn_url
```

Příklady

Vypíše soubor readme.txt v pracovním adresáři:

```
$ ver cat readme.txt
This is a README file in a working directory.
You should read this.
```

Soubor readme.txt můžeme ale také nechat vypsat z repository. V tomto příkladu ze SVN repository:

```
$ ver cat --svn_url \
http://svn.red-bean.com/repos/test/readme.txt
This is a README file.
You should read this.
```

Název

ver checkout – Stáhne pracovní kopii z repository do pracovního adresáře.

Předpis

```
ver checkout REP_PATH [PATH]
```

Popis

Stáhne pracovní kopii z repository do pracovního adresáře. Pokud *PATH* není uvedena, použije se poslední položka z cesty *REP_PATH* vůči aktuálnímu adresáři.

Alternativní názvy

co

Mění

Vytváří pracovní adresář.

Přistupuje k repository

ano

Přepínače

```
--revision (-r) REV
--non-recursive (-N)
--svn_url
```

Příklady

Stáhne pracovní kopii z repository do adresáře mine:

```
$ ver checkout /tmp/repos/test mine
A  mine/a
A  mine/b
Checked out revision 2
$ ls
mine
```

Pokud je checkout přerušen před úspěšným dokončením, může být obnoven vyvoláním stejného příkazu checkout nebo použitím příkazu update k aktualizaci nekompletního pracovního adresáře:

```
$ ver checkout /tmp/repos/test test
A  test/a
A  test/b
^C

$ ver checkout /tmp/repos/test test
A  test/c
A  test/d
^C

$ cd test
$ ver update
A  test/e
A  test/f
Updated to revision 3
```

Název

ver commit – Odešle změny z pracovního adresáře do repository.

Předpis

```
ver commit [PATH...]
```

Popis

Odešle změny z pracovního adresáře do repository. Logovací záznam je nutné uvést pomocí parametru --message, který je povinný.

Alternativní názvy

ci (zkratka za „check in“; ne „co“, což je zkratka za „checkout“)

Mění

Pracovní adresář, repository

Přistupuje k repository

ano

Přepínače

```
--message (-m) TEXT
--non-recursive (-N)
```

Příklady

Provede commit jednoduché změny souboru. Jako cíl je použit implicitní cíl, pracovní adresář („.“):

```
$ ver commit -m ”added howto section.”
Repository was locked
Journal was saved
Sending        ./a
Committed revision 3
Journal was deleted
Repository was unlocked
```

Provede commit souboru určeného pro smazání:

```
$ ver commit -m ”removed file ’c’.”
Repository was locked
```

```
Journal was saved
Deleting        ./c
Committed revision 4
Journal was deleted
Repository was unlocked
```

Název

ver create – Vytvoří novou repository.

Předpis

```
ver create REP_PATH
```

Popis

Vytvoří nové prázdné repository. Pokud zadaný adresář neexistuje, je vytvořen. Pokud je uveden přepínač --svn_url, výjimečně se uvádí cesta, nikoliv url. Pokud je vytvořeno SVN repository, bude mít fsfs backend.

Alternativní názvy

žádné

Mění

Vytváří nové repository.

Přistupuje k repository

ano

Přepínače

```
--svn_url
```

Příklady

Vytvoření nového repository je opravdu jednoduché:

```
$ ver create /tmp/new_repos
```

Název

ver delete – Smaže položku z pracovního adresáře.

Předpis

```
ver delete PATH...
```

Popis

Položka určená parametrem *PATH* je naplánována k smazání, které se provede během dalšího commitu. Soubory (a adresáře, které ještě nebyly ani jednou odeslány commitem) jsou ihned smazány z pracovního adresáře. Příkaz nesmaže položky, které nejsou pod správou verzí anebo které byly změněny. Toto chování lze potlačit přepínačem --force.

Alternativní názvy

del, remove, rm

Mění

Pracovní adresář

Přistupuje k repository

ne

Přepínače

```
--force
```

Příklady

Smaže soubor myfile z pracovního adresáře a naplánuje jeho smazaní z repository. Po vyvolání příkazu commit je tento soubor smazán i z repository:

```
$ ver delete myfile
D         myfile

$ ver commit -m ”Deleted file ’myfile’.”
Repository was locked
Journal was saved
Deleting        myfile
Committed revision 5
Journal was deleted
Repository was unlocked
```

Následující příklad ukazuje, jak lze vynutit smazání souboru, který byl změněn, pomocí přepínače --force:

```
$ ver delete over-there
error during deleting: over-there:has local \
modifications
fatal error occurred: we can’t continue

$ ver delete --force over-there
D         over-there
```

Název

ver delete_journal – Odstraní journal.

Předpis

```
ver delete_journal [PATH]
```

Popis

Odstraní journal v zadaném pracovním adresáři. Pokud není zadán žádný pracovní adresář, je použit aktuální adresář.

Alternativní názvy

žádné

Mění

Pracovní adresář

Přistupuje k repository

ne

Přepínače

žádné

Příklady

Použití je opravdu jednoduché:

```
$ ver delete_journal
Journal was deleted
```

Název

ver diff – Zobrazí rozdíly mezi soubory v pracovním adresáři a repository.

Předpis

```
ver diff PATH...
```

Popis

Zobrazí rozdíly mezi souborem v pracovním adresáři a nějakou jeho revizí v repository. Přepínač --revision určuje, vůči které revizi souboru v repository se porovnání provádí. Pokud není uveden, porovnání probíhá vůči head revizi.

Alternativní názvy

žádné

Mění

nic

Přistupuje k repository

ano

Přepínače

```
--revision (-r) REV
```

Příklady

Můžeme tak jednoduše zjistit, jaké lokální změny jsme v souboru main.cpp v pracovním adresáři provedli:

```
$ ver diff main.cpp
6c6,8
<    p->left = q->right;
---
>    if(p) {
>       p->left = q->right;
>    }
20a22
>    check();
300c303
<    return 0;
---
>    return SUCCESS;
```

Formát výstupu je stejný jako u běžné utility diff. Číslice udávají řádek, případně rozsah řádků, pokud je mezi nimi čárka. Písmena specifikují prováděnou akci (**a**dd, **d**elete, **c**hange). Číslice před písmenem se týkají prvního souboru, číslice za písmenem druhého souboru. Výstup je snad dostatečně intuitivní. V našem příkladu vidíme, že řádek 6 z prvního souboru je nahrazen řádky 6 až 8 z druhého souboru. Dále za řádek dvacet z prvního souboru je přidán řádek 22 z druhého souboru. Řádek 300 je nahrazen řádkem 303.

Název

ver change_synch_mode – Změní synch mode repository.

Předpis

```
ver change_synch_mode [TARGET]
```

Popis

Změní synch mode repository. Pokud není parametr *TARGET* uveden, použije se aktuální adresář. Pokud je *TARGET* cesta k pracovnímu adresáři, synch mode se změní u repository, které je s tímto pracovním adresářem svázáno.

Alternativní názvy

žádné

Mění

Repository

Přistupuje k repository

ano

Přepínače

```
--svn_url
```

Příklady

Zapne synch mode zadané repository:

```
$ ver rep_info /tmp/repos
head version: 15
synch mode: disabled

$ ver change_synch_mode /tmp/repos
synch mode: enabled
```

Tip

Pokud chcete zjistit současný stav synch mode, můžete použít **ver rep_info**.

Název

ver list – Vypíše obsah repository nebo pracovního adresáře.

Předpis

```
ver list [TARGET...]
```

Popis

Vylistuje každý soubor, případně obsah adresáře uvedeného v *TARGET*. *TARGET* může obsahovat položky pracovního adresáře nebo cesty k repository. V případě neuvedení žádné cesty se použije implicitní cíl, pracovní adresář („.“).

Při uvedení parametru --verbose se vypíšou následující informace pro každou položku:

- Revize odpovídající poslednímu commitu
- Datum a čas poslední změny

Alternativní názvy

ls

Mění

nic

Přistupuje k repository

ano

Přepínače

```
--revision (-r) REV
--verbose (-v)
--directory (-d)
--svn_url
```

Příklady

ver list je velmi užitečné, pokud chceme vědět, jaké položky se nacházejí v repository, aniž bychom volali checkout:

```
$ ver list --svn_url \
http://svn.red-bean.com/repos/test/support
README.txt
INSTALL
examples/
```

Použitím přepínače --verbose můžeme získat další informace:

```
$ ver list --verbose /tmp/repos
14.2            17.7.2006 3:03:37       bbb/
1               14.7.2006 20:50:18      dir/
14.3            17.7.2006 3:04:09       file
10              15.7.2006 15:01:00      file2
```

Název

ver log – Zobrazí logovací záznamy.

Předpis

```
ver log [TARGET...]
```

Popis

Implicitní cíl je aktuální pracovní adresář. Pokud nejsou uvedeny žádné další dodatečné parametry, **ver log** zobrazí logovací záznamy všech souborů a podadresářů v aktuálním pracovním adresáři. Toto chování lze upravit uvedením nějaké cesty, rozsahem zobrazených revizí nebo kombinací obojího. Standardní rozsah revizí je HEAD:1.

Jako cestu lze uvést i cestu k repository, pak se logovací záznamy zobrazují vzhledem k souborům a adresářům v uvedeném repository.

Každý logovací záznam je zobrazen jen jednou pro jednu zadanou cestu, i když přísluší více položkám v této cestě. Při více zadaných cestách se záznamy opakují tak, jak se prochází přes tyto zadané cesty.

Alternativní názvy

žádné

Mění

nic

Přistupuje k repository

ano

Přepínače

```
--revision (-r) REV
--svn_url
```

Příklady

Zobrazí všechny logovací záznamy pro položky v pracovním adresáři zadáním příkazu **ver log** v kořenu pracovního adresáře:

```
$ ver log
------------------------------------------------
r10 | 15.7.2006 20:24:20

Synchronization to the version 10
------------------------------------------------
r5.2 | 15.7.2006 15:14:05

pridan adresar dir2
------------------------------------------------
...
```

Můžeme si nechat zobrazit logovací záznamy třeba jen pro jeden soubor:

```
$ ver log foo.c
------------------------------------------------
r32 | 15.7.2006 15:10:06
```

```
Added defines.
------------------------------------------------
r30 | 14.7.2006 20:00:08

Some fixies
------------------------------------------------
...
```

Nemusíme se však omezovat jen na pracovní adresář a můžeme si nechat vypsat logovací záznamy pro repository a stejně tak můžeme omezit objem výpisu zadáním rozsahu revizí:

```
$ ver log -r 1:2 --svn_url \
http://svn.collab.net/repos/svn
------------------------------------------------
r1 | 31.8.2001 8:24:14

Initial import.
------------------------------------------------
r2 | 31.8.2001 8:26:09

Update notice to indicate self-hosting.
------------------------------------------------
```

Může se stát, že když **ver log** zadáme konkrétní cestu a revizi, nedostane žádný výstup:

```
$ ver log -r 20 /tmp/repos/trunk/untouched.txt
------------------------------------------------
```

To neznamená nic jiného, než že tato cesta nebyla v zadané revizi změněna.

Název

ver recommit – Dokončí předešlý neúspěšný commit.

Předpis

```
ver recommit [PATH...]
```

Popis

Dokončí předešlý neúspěšný commit. Podmínkou je přítomnost Journalu z předešlého commitu. Journal se neukládá, pokud předešlý commit byl do SVN repository, jinak ano.

Alternativní názvy

žádné

Mění

Repository

Přistupuje k repository

ano

Přepínače

žádné

Příklady

Nejjednodušší případ jak vyvolat recommit:

```
$ ver commit -m”oprava kritické chyby”
Repository was locked
Journal was saved
^C

$ ver recommit
Repository lock was verified
Committed revision 14.6
Journal was deleted
Repository was unlocked
```

Při commitu do SVN repository se Journal smaže při přerušení commitu, a proto recommit nelze použít:

```
$ ver recommit
```

```
error during recommitting: .:has not a journal
fatal error occurred: we can’t continue
```

Název

ver rep_info – Zobrazí informace o repository.

Předpis

```
ver rep_info [REP_PATH]
```

Popis

Zobrazí informace o repository. Pokud není parametr *REP_PATH* uveden a nacházíme se v pracovním adresáři, jsou zobrazeny informace o repository, které je s tímto pracovním adresářem svázáno. Zobrazené informace obsahují tyto položky:

- head revision – nejnovější revize
- synch mode – zda je zapnutý synch mode

Alternativní názvy

žádné

Mění

nic

Přistupuje k repository

ano

Přepínače

```
--svn_url
```

Příklady

Zjistí, jaká je nejnovější verze SVN:

```
$ ver rep_info --svn_url \
http://svn.collab.net/repos/svn/trunk/
head version: 20698
synch mode: disabled
```

Název

ver resolved – Odstraní konfliktní stav souboru v pracovním adresáři.

Předpis

```
ver resolved PATH...
```

Popis

Odstraní konfliktní stav souboru v pracovním adresáři. Tento příkaz neodstraní konflikt jako takový, pouze smaže pomocné soubory, které vznikly při odhalení konfliktu. Soubor tak bude možné znovu zpracovat commitem.

Alternativní názvy

žádné

Mění

Pracovní adresář

Přistupuje k repository

ne

Přepínače

žádné

Příklady

Pokud vznikne konflikt během updatu, objeví se v pracovním adresáři tři nové soubory:

```
$ ver update
C  foo.c
Updated to revision 31
$ ls
foo.c
foo.c.mine
foo.c.r30
foo.c.r31
```

Až konflikt vyřešíme a soubor foo.c bude připraven ke commitu, spustíme **ver resolved** na znamení, že problém byl vyřešen.

Varování

Pomocné soubory *mohou* být odstraněny bez použití **ver resolved**, ale použití **ver resolved** je pohodlnější a spolehlivější.

Název

ver status – Zobrazí stav souborů a adresářů v pracovním adresáři.

Předpis

```
ver status [PATH...]
```

Popis

Zobrazí stav souborů a adresářů v pracovním adresáři. Bez uvedení přepínače --verbose vypíše jenom položky, které byly lokálně změněny, nepřistupuje tedy k repository. S uvedením přepínače --verbose poskytne plný výpis.

První sloupec udává, jestli byla položka přidána, smazána nebo nějak jinak změněna.

' ' Žádná změna.

'A' Položka je naplánována pro přidání.

’D’ Položka je naplánována pro smazání.

’M’ Položka byla změněna.

’R’ Položka byla nahrazena v pracovním adresáři. To znamená, že soubor byl nejdříve naplánován pro smazání a pak byl přidán jiný nový soubor se stejným jménem.

’C’ Položka je v konfliktním stavu.

’?’ Položka není pod kontrolou správy verzí.

’!’ Položka chybí. Pravděpodobně byla smazána bez použití **ver**.

’~’ Položka je spravována jako jeden typ (soubor, adresář), ale byla nahrazena jiným typem.

Druhý sloupec uvádí, jestli je položka aktuální, nebo ne.

’ ’ Položka v pracovním adresáři je aktuální.

’*’ V repository existuje novější revize než v pracovním adresáři.

Třetí sloupec obsahuje revizi, ve které se položka nachází v pracovním adresáři.

Čtvrtý sloupec pak obsahuje informaci, ve které revizi byl naposledy proveden commit.

Alternativní názvy

stat, st

Mění

nic

Přistupuje k repository

Pokud se použije --verbose

Přepínače

```
--verbose (-v)
--non-recursive (-N)
```

Příklady

Toto je nejjednodušší cesta, jak zjistit, které položky byly v pracovním adresáři změněny:

```
$ ver status wc
M       wc/file
A       wc/file2
```

Pokud se chceme dovědět, které položky jsou neaktuální a další informace, musíme použít přepínač --verbose. Neaktuální položkou je v tomto příkladu soubor foo.c, který se změnil v repository od doby, kdy jsme naposledy prováděli update pracovního adresáře:

```
$ ver status --verbose
                14      1       .
                14      1       dir\.
        *       4       1       foo.c
M               14.3    14.3    file
```

Název

ver synchronize – Synchronizuje SVN Proxy repository se SVN repository.

Předpis

```
ver synchronize TMP_PATH PROXY_REP_PATH SVN_REP_PATH

ver synchronize [TMP_PATH]
```

Popis

Synchronizuje SVN Proxy repository se SVN repository. První varianta příkazu slouží k prvotnímu zahájení synchronizace.

Druhá varianta slouží k obnovení synchronizace, pokud předešlý pokus byl z jakéhokoliv důvodu přerušen. Je nutné uvést stejnou *TMP_PATH* jako při prvním spuštění. Pokud není *TMP_PATH* uvedena, použije se aktuální adresář.

Alternativní názvy

synch

Mění

Repository

Přistupuje k repository

ano

Přepínače

žádné

Příklady

Provede synchronizaci SVN Proxy repository se SVN repository:

```
$ ver synchronize d:\x_temp d:\rep3 file:///d:/rep4
creating a new synch dir
start of export
checking out from a ver repository the version 0
synch to the version: 1
A       d:\x_temp\file
Repository was locked
Journal was saved
Sending         d:\x_temp\.
Adding          d:\x_temp\file
Committed revision 1
Journal was deleted
Repository was unlocked
deleting the synch dir
all is done
```

Synchronizace může být přerušena například kvůli chybě, anebo proto, že vznikl konflikt. Po vyřešení problému či konfliktu lze použít druhou variantu příkazu **ver synchronize** pro obnovení přerušené synchronizace.

Název

ver unlock – Zruší všechny locky repository.

Předpis

```
ver unlock [REP_PATH]
```

Popis

Zruší všechny locky repository *REP_PATH*. Pokud není uvedena žádná cesta a nacházíme-li se v pracovním adresáři, použije se repository, které je s tímto pracovním adresářem svázáno.

Alternativní názvy

žádné

Mění

Repository

Přistupuje k repository

ano

Přepínače

```
--svn_url
```

Příklady

Odemkne zadané repository:

```
$ ver unlock /tmp/repos
```

Název

ver update – Provede update pracovního adresáře.

Předpis

```
ver update [PATH...]
```

Popis

ver update přenese změny z repository do pracovního adresáře. Pokud není uvedena žádná revize, update zajistí aktuálnost vůči HEAD revizi repository. V opačném případě je update proveden oproti revizi uvedené v parametru --revision.

Pro každou aktualizovanou položku je na začátku uvedeno písmeno odpovídající akci, která se při aktualizaci vykonala. Použitá písmena mají následující význam:

A Added – položka byla přidána

D Deleted – položka byla smazána

U Update – byl aktualizován obsah položky

C Conflict – položka nabyla konfliktního stavu

G Merged – změny byly sloučeny s lokálními změnami v pracovním adresáři

Alternativní názvy

up

Mění

Pracovní adresář

Přistupuje k repository

ano

Přepínače

```
--revision (-r) REV
--non-recursive (-N)
```

Příklady

Provede změny, které se v repository udály od posledního update:

```
$ ver update
U       .\dir\file
A       .\newdir\toggle.c
A       .\newdir\disclose.c
A       .\newdir\launch.c
D       .\newdir\README
Updated to revision 32
```

Můžeme samozřejmě aktualizovat pracovní adresář vzhledem k nějaké starší revizi:

```
$ ver update -r30
U       .\dir\file
A       .\newdir\README
D       .\newdir\toggle.c
D       .\newdir\disclose.c
D       .\newdir\launch.c
U       .\foo.c
Updated to revision 30
```

Tip

Pokud chcete prohlížet starší revize jednotlivých souborů, můžete použít **ver cat**.

Příloha B

Obsah přiloženého CD

K této bakalářské práci je přiloženo CD, které obsahuje popisovanou aplikaci SVN Proxy. Obsah tohoto CD bude nyní stručně uveden:

- `doc/` – dokumentace projektu.
- `doc/bak.pdf` – tato bakalářská práce.
- `doc/INSTALL.txt` – popis kompilace a postupu, jak se má SVN Proxy nainstalovat.
- `doc/reference/` – programátorská reference vytvořená Doxygenem ze zdrojových souborů. Kořenovým dokumentem této reference je soubor `index.html`.
- `install/` – zkompilovaný program pro Windows, který lze nainstalovat. Více informací lze nalézt v souboru `doc/INSTALL.txt`.
- `projects/` – pomocné soubory Visual Studia a potřebné knihovny a headery pro kompilaci na operačním systému Windows.
- `src/` – zdrojové soubory.
- `Makefile` – makefile pro kompilaci na Linuxu.
- `README.txt` – tento soubor obsahuje obdobné informace jako tento přehled.
- `ver.sln` – solution pro MS Visual Studio 8 sloužící ke kompilaci na Windows.