VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

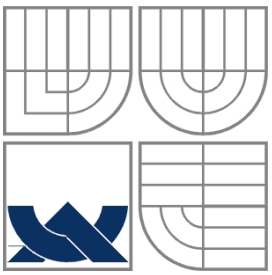
FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

KLIENTSKÁ ČÁST SYSTÉMU PRO SPRÁVU PROJEKTOVÉ DOKUMENTACE

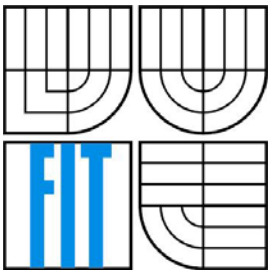
DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE Bc. ONDŘEJ BÝM
AUTHOR

BRNO 2008



VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

KLIENTSKÁ ČÁST SYSTÉMU PRO SPRÁVU PROJEKTOVÉ DOKUMENTACE

CLIENT PART OF THE PROJECT DOCUMENTATION MANAGEMENT SYSTEM

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE
AUTHOR

Bc. ONDŘEJ BÝM

VEDOUCÍ PRÁCE
SUPERVISOR

Ing. ZBYNĚK KŘIVKA, Ph.D.

BRNO 2008

Zadání diplomové práce/5972/2007/xbymon00

Vysoké učení technické v Brně - Fakulta informačních technologií

Ústav informačních systémů Akademický rok 2007/2008

Zadání diplomové práce

Řešitel: **Bým Ondřej, Bc.**
Obor: Informační systémy
Téma: **Klientská část systému pro správu projektové dokumentace**
Kategorie: Softwarové inženýrství

Pokyny:

1. Seznamte se s problematikou verzování a systémů na správu verzí a správu změn. Zaměřte se zejména na požadavky kladené na uživatelské rozhraní a na synchronizaci lokálních dat na klientské stanici s daty na serveru.
2. Navrhněte architekturu systému, který bude poskytovat podporu pro správu projektové dokumentace, zejména správu verzí. Při návrhu spolupracujte s Radkem Černobilou, který tvoří serverovou část systému. Vaším úkolem bude navrhnout především koncepci uživatelského rozhraní a navrhnout rozhraní umožňující použití zásuvných modulů specializovaných na porovnání určitých typů souborů.
3. Proveďte návrh uživatelského rozhraní a návrhněte algoritmy porovnání a způsob vizualizace změn souborů různých typů.
4. Klientskou část systému navrženou v bodech 2 a 3 implementujte v prostředí platformy .NET.
5. Funkčnost ověřte na ukázkovém projektu.
6. Zhodnoťte dosažené výsledky.

Literatura:

- Conradi, R.: Westfechtel, B.: Version models for software configuration management. ACM Computing Surveys (CSUR), Vol. 30, No. 2. 1998. pp. 232 - 282.
- Baudiš, P.: Výlet do říše verzí. Seriál online článků dostupný na http://www.root.cz/serialy/vylet-do-rise-verzi/.
- Dokumentace k platformě .NET a jazyku C# dostupná na stránce Microsoft MSDN http://msdn2.microsoft.com/en-us/default.aspx.

Při obhajobě semestrální části diplomového projektu je požadováno:

- Body 1 až 3.

Podrobné závazné pokyny pro vypracování diplomové práce naleznete na adrese http://www.fit.vutbr.cz/info/szz/

Technická zpráva diplomové práce musí obsahovat formulaci cíle, charakteristiku současného stavu, teoretická a odborná východiska řešených problémů a specifikaci etap, které byly vyřešeny v rámci ročníkového a semestrálního projektu (30 až 40% celkového rozsahu technické zprávy).

Student odevzdá v jednom výtisku technickou zprávu a v elektronické podobě zdrojový text technické zprávy, úplnou programovou dokumentaci a zdrojové texty programů. Informace v elektronické podobě budou uloženy na standardním nepřepisovatelném paměťovém médiu (CD-R, DVR-R, apod.), které bude vloženo do písemné zprávy tak, aby nemohlo dojít k jeho ztrátě při běžné manipulaci.

Vedoucí: **Křivka Zbyněk, Ing., Ph.D.**, UIFS FIT VUT
Datum zadání: 24. září 2007
Datum odevzdání: 19. května 2008

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
Fakulta informačních technologií
Ústav informačních systémů
612 66 Brno, Božetěchova 2

L.S.

doc. Ing. Jaroslav Zendulka, CSc.
vedoucí ústavu

LICENČNÍ SMLOUVA
POSKYTOVANÁ K VÝKONU PRÁVA UŽÍT ŠKOLNÍ DÍLO

uzavřená mezi smluvními stranami

1. Pan

Jméno a příjmení: **Bc. Ondřej Bým**
Id studenta: 88409
Bytem: Národní 112/4, 460 07 Liberec
Narozen: 23. 06. 1983, Liberec
(dále jen "autor")

a

2. Vysoké učení technické v Brně

Fakulta informačních technologií
se sídlem Božetěchova 2/1, 612 66 Brno, IČO 00216305
jejímž jménem jedná na základě písemného pověření děkanem fakulty:

.....
(dále jen "nabyvatel")

Článek 1
Specifikace školního díla

1. Předmětem této smlouvy je vysokoškolská kvalifikační práce (VŠKP):
diplomová práce

Název VŠKP: Klientská část systému pro správu projektové dokumentace
Vedoucí/školitel VŠKP: Křivka Zbyněk, Ing., Ph.D.
Ústav: Ústav informačních systémů
Datum obhajoby VŠKP: .....

VŠKP odevzdal autor nabyvateli v:

tištěné formě počet exemplářů: 1
elektronické formě počet exemplářů: 2 (1 ve skladu dokumentů, 1 na CD)

2. Autor prohlašuje, že vytvořil samostatnou vlastní tvůrčí činností dílo shora popsané a specifikované. Autor dále prohlašuje, že při zpracovávání díla se sám nedostal do rozporu s autorským zákonem a předpisy souvisejícími a že je dílo dílem původním.
3. Dílo je chráněno jako dílo dle autorského zákona v platném znění.
4. Autor potvrzuje, že listinná a elektronická verze díla je identická.

Článek 2
Udělení licenčního oprávnění

1. Autor touto smlouvou poskytuje nabyvateli oprávnění (licenci) k výkonu práva uvedené dílo nevýdělečně užít, archivovat a zpřístupnit ke studijním, výukovým a výzkumným účelům včetně pořizování výpisů, opisů a rozmnoženin.
2. Licence je poskytována celosvětově, pro celou dobu trvání autorských a majetkových práv k dílu.
3. Autor souhlasí se zveřejněním díla v databázi přístupné v mezinárodní síti:
   - ☐ ihned po uzavření této smlouvy
   - ☐ 1 rok po uzavření této smlouvy
   - ☐ 3 roky po uzavření této smlouvy
   - ☐ 5 let po uzavření této smlouvy
   - ☐ 10 let po uzavření této smlouvy

   (z důvodu utajení v něm obsažených informací)
4. Nevýdělečné zveřejňování díla nabyvatelem v souladu s ustanovením § 47b zákona č. 111/ 1998 Sb., v platném znění, nevyžaduje licenci a nabyvatel je k němu povinen a oprávněn ze zákona.

Článek 3
Závěrečná ustanovení

1. Smlouva je sepsána ve třech vyhotoveních s platností originálu, přičemž po jednom vyhotovení obdrží autor a nabyvatel, další vyhotovení je vloženo do VŠKP.
2. Vztahy mezi smluvními stranami vzniklé a neupravené touto smlouvou se řídí autorským zákonem, občanským zákoníkem, vysokoškolským zákonem, zákonem o archivnictví, v platném znění a popř. dalšími právními předpisy.
3. Licenční smlouva byla uzavřena na základě svobodné a pravé vůle smluvních stran, s plným porozuměním jejímu textu i důsledkům, nikoliv v tísni a za nápadně nevýhodných podmínek.
4. Licenční smlouva nabývá platnosti a účinnosti dnem jejího podpisu oběma smluvními stranami.

V Brně dne: .....

....

Nabyvatel Autor

Klientská část systému pro správu projektové dokumentace

Prohlášení

Prohlašuji, že jsem tuto práci vypracoval samostatně pod vedením Ing. Zbyňka Křivky, PhD. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

……
Ondřej Bým
19. května 2008

Poděkování

Na tomto místě bych rád poděkoval vedoucímu mé diplomové práce Ing. Zbyňku Křivkovi, Ph.D. za věcné připomínky, podněty a pečlivou kontrolu při vedení mé diplomové práce. Dále bych chtěl poděkovat Bc. Radku Černobilovi za spolupráci při vývoji systému.

Abstrakt

Cílem této práce je navrhnout obecně použitelný verzovací systém pro správu elektronických dokumentů různých typů, detailněji navrhnout a implementovat klientskou část tohoto systému (založeném na modelu klient-server). Implementace je postavena na platformě .NET. Tento text také popisuje obecné přístupy k verzování v různých systémech a poskytuje přehled nad principy existujících verzovacích systémů s důrazem na interakci s uživatelem.

Klíčová slova

verzovací systém, verzování, správa verzí, verze souboru, revize souboru, LCS, nejdelší společná podposloupnost, problém vzdálenosti změn, XML serializace .NET, pracovní prostor verzovacího systému

Abstract

The goal of this work is to design a generally useful versioning system for the administration of different types of electronics documents, to design in detail and to implement the client part of this system (based on the client-server model). The implementation is built on .NET platform. This text also describes general approaches to versioning in different systems and shows a survey over the principles of the existing versioning systems with respect to the interaction with user.

Keywords

versioning system, versioning, revision control, file version, file revision, LCS, longest common subsequence, edit distance problem, XML .NET serialization, version system working space

Citace

Bým Ondřej: Klientská část systému pro správu projektové dokumentace. Brno, 2008, diplomová práce, FIT VUT v Brně.

Obsah

1 Úvod

Společnosti ale i jednotlivci zabývající se vývojem software dříve nebo později začali pociťovat potřebu systémů pro jednoduchou správu a uchovávání zdrojových kódů jimi vyvíjených softwarových produktů. S růstem velikosti programátorských týmů rostla i potřeba spolupráce a kooperace při vývoji software, kde je jedním ze zásadních problémů sdílení právě těchto zdrojových kódů. Dále do spolupráce vstupuje potřeba paralelního vývoje na různých variantách jednotlivých algoritmů, ale i celých systémových komponent. Proto softwarové společnosti velmi brzy začaly vyvíjet a používat tzv. verzovací systémy. Ty byly sestaveny tak, aby si programátorské týmy v mnoha směrech ulehčily práci a do jisté míry zautomatizovaly rutinní činnosti. Historie verzovacích systémů sahá až do dob počátků programování. V době překotného vývoje programovacích jazyků a platforem prošly i verzovací systémy řadou změn, měnila se i celková koncepce. Vždy se ale tyto systémy vyráběly na míru programátorům podle jejich specifických požadavků.

V dnešní době jsou však elektronické dokumenty v různých formách používány prakticky v každém odvětví lidské činnosti. Většina dokumentů, které dříve vznikaly v čistě listinné podobě, jsou dnes produkovány v podobě elektronické. V závislosti na velikosti instituce pak vzniká denně od jednotek po tisíce nových dokumentů. Převod těchto záznamů do elektronické podoby s sebou nese i problematiku jednotného uchovávání. Vznikají také vysoké nároky na správu a zabezpečení. Na rozdíl od listinné podoby s sebou přinášejí elektronické dokumenty možnost znovupoužití a reprodukce s možností různě rozsáhlých úprav. Proto zde vyvstává potřeba verzovacích a archivačních systémů, které se nesoustředí pouze na potřeby softwarových vývojářů, ale přistupují k verzování z obecného hlediska. Takový systém by měl být široce použitelný a přizpůsobitelný konkrétní potřebě koncového uživatele. Zároveň by měl být jednoduchý na ovládání, zejména z toho důvodu, že systém bude denně využíván lidmi, kteří mají jen základy práce s osobním počítačem.

Vzniká tedy motivace k vytvoření programového systému pro ukládání, správu a sdílení obecných dokumentů. V takovém systému by měl být kladen důraz především na přehled nad manipulací s jednotlivými dokumenty a jejich časovými a variantními verzemi. Dále musí systém umožnit práci s celými množinami souborů či projektovými adresářovými strukturami. S tím také souvisí možnost aktualizace těchto projektových struktur či jednotlivých dokumentů. V neposlední řadě je třeba poskytnout jednoduchou definici a kontrolu oprávnění přístupu jak k jednotlivým dokumentům, tak k celým projektovým celkům.

Tato práce se shrnuje problematiku verzování a verzovacích systémů zejména z pohledu uživatelského rozhraní a podle způsobu použití. Dále se zabývá návrhem obecného verzovacího systému, založeného na modelu klient-server, a to včetně implementace klientské části tohoto systému. První kapitola zahrnuje základní přehled principů verzování a z nich vyplývajících architektur verzovacích systémů. Jsou zde také popsány způsoby organizace pracovních prostorů

uživatelů. Dále jsou v této kapitole představeni typičtí zástupci existujících verzovacích systémů, kteří jsou zhodnoceni z hlediska uživatelského přístupu a jejich základních vlastností. V další kapitole jsou rozebrány možnosti synchronizace adresářových struktur, včetně problému zjištění diferencí mezi verzemi souborů. Třetí kapitola se zabývá návrhem verzovacího systému, jeho architekturou a komunikačním protokolem. Klientská část systému, včetně několika modulů pro porovnání diferencí mezi jednotlivými verzemi souborů, je navržena ve čtvrté kapitole. V předposlední kapitole jsou zaznamenány výsledky provedených testů. V závěru je pak zhodnocen přínos práce a nastíněn další možný vývoj.

2 Systémy pro správu verzí dokumentů

Během poměrně krátké historie systémů pro správu verzí dokumentů vznikla řada systémů vycházejících z několika základních přístupů k verzování, nebo tyto přístupy kombinujících. V této kapitole je popsán přehled základních principů verzování a nastíněna filozofie těchto systémů.

2.1 SCM - Software Configuration Management

Systémy pro správu verzí dokumentů, neboli verzovací systémy, souvisí s tzv. SCM (Software Configuration Management). SCM je část řízení vývoje software zabývající se identifikací, organizací a správou změn částí softwarového produktu (viz [1]). SCM pokrývá celý životní cyklus vývoje software.

2.1.1 Definice

CM (Configuration Management) je oblast zabývající se řízením změn ve velkých komplexních systémech. Hlavním účelem je spravovat a řídit množství oprav, rozšíření či adaptací, které provázejí systém během celého jeho životního cyklu. SCM je pak aplikací CM na softwarové systémy (viz [2]). Cílem SCM je zabezpečit systematicky a přehledně záznamy o všech vývojových procesech tak, aby byly zpětně identifikovatelné a zároveň zajistit neustálou dostupnost software v jeho správné aktuální formě. Samotné SCM se pak dělí na několik oblastí, přičemž jedna z nich se zaměřuje na správu projektových souborů a adresářů. Většina vývojových nástrojů stále spoléhá na standardní souborový systém a předává starost nad správou projektových dokumentů samotnému vývojáři. Také z tohoto důvodu vzniklo množství externích nástrojů, které se to snaží řešit. Někdy se nástroje zahrnující tuto oblast označují také zkratkou SCM avšak ve významu Source Code Management. Další kapitoly se již budou zabývat pouze touto užší problematikou, a to pod pojmem verzovací systémy.

2.2 Verzovací systémy a verzování

Verzovací systém (také systém pro správu verzí), je program či skupina programů umožňující uživateli oproti práci s běžným souborovým systémem náhled do historie změn jednotlivých souborů. Zároveň umožňuje jednotlivé soubory a adresáře sdružovat do skupin nazývajících se projekt. Na ten pak lze nahlížet v časových milnících, což umožňuje návrat do historie projektu jako celku, bez potřeby znát konkrétní verze souborů.

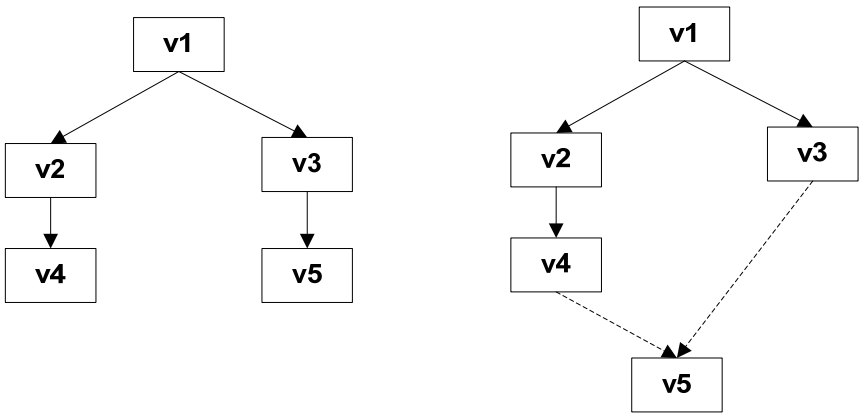
Systémy pro správu verzí by měly zejména ulehčovat práci ve vícečlenném týmu, práci na několika verzích software současně a také mohou sloužit jako archivační úložiště. Verzovací systém je vlastně správcem tzv. *repositáře*, což je složka na lokálním disku či serveru, v níž je uložena

adresářová struktura projektu. Každý soubor, příp. i adresář pak může obsahovat různé verze, přičemž tyto verze se mohou dělit do dalších. Více o způsobech verzování je uvedeno v následujících odstavcích.

2.2.1 Klasické verzování

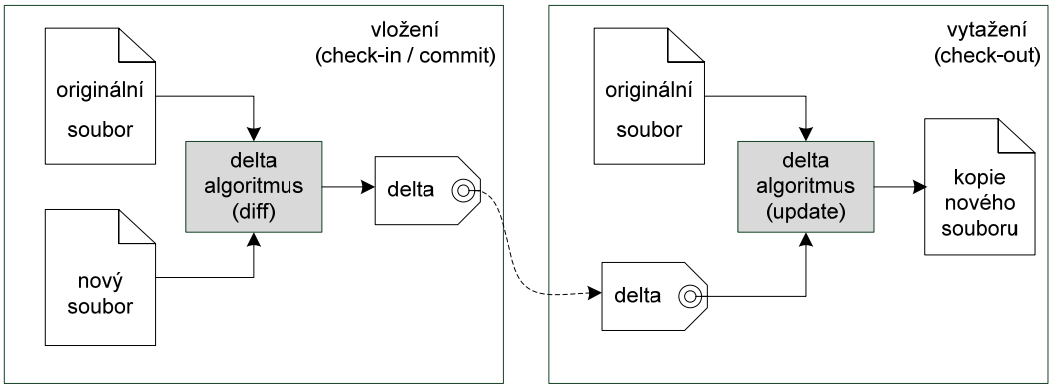
Obvyklá technika je označit každou změnu nějakým identifikátorem. Samotné záznamy změn jsou však nedostačující pro určení vztahů mezi verzemi a reálným modelem. Klasické verzování tedy přináší techniku, při které se příbuzné položky seskupují do množin zvaných verzovací skupina (jinak též verzovací graf, komponenta), kde se pak spravuje vývoj těchto množin. Položky jsou organizovány do orientovaného acyklického grafu (obr. 1), kde hrany reprezentují vztah následník. Rozlišujeme tři typy vztahu následník:

1) Vztah **revize (revision-of)** odkazuje na přímou historii vývoje konkrétní položky.
2) Vztah **varianta (variant-of)** vzniká mezi paralelními položkami, které jsou prakticky totožné, avšak liší se v určité konfiguraci (implementační, funkční). Na obr. 1 je tento vztah např. mezi položkami *v2* a *v3*, případně *v4* a *v3*.
3) Vztah **sloučení (merge)** reprezentuje provedení změn v několika variantách, které se kombinují dohromady.



Obrázek 1- Příklady možných grafů

Tato technika verzování se dle [2] nazývá klasické verzování a je implementována prakticky ve všech verzovacích nástrojích. Při delším vývoji však vznikají stovky až jednotky tisíc revizí pro každou položku, přičemž rozdíly jsou prakticky minimální. Objevuje se zde tedy problém s prostorem úložiště změn. Dané položky není možné ukládat vcelku, ale nějakým vhodným způsobem komprimovat. Změny se pak při požadavku na zobrazení jednotlivé verze položky musí z těchto komprimovaných dat zrekonstruovat. Mechanismus zabývající se těmito optimalizacemi se nazývá delta technika, delta komprese či delta algoritmus (viz obr. 2). Výsledkem těchto metod jsou data nazvaná právě delta (více v kapitole 2.2.5).



Obrázek 2 - Princip delta mechanismu

2.2.2 Pokročilé verzování

Klasické verzování je limitováno svými pevně danými vztahy, které jsou vedeny mezi konkrétními fyzickými soubory a konfiguračními položkami systému. Pro mnoho účelů je toto postačující, avšak existují i techniky pro uživatele více flexibilní. Některé jsou obecnější, některé specializované pro konkrétní využití, např. konkrétních typů souborů, programovacích jazyků, či vývojových nástrojů.

2.2.2.1 Prostor produktu

Jedna taková technika se nazývá *prostor produktu* (angl. *product space*). *Prostor produktu* aplikace (softwarového produktu) je vlastně zobecněním konfiguračních položek a pravidel. Systém zahrnuje veškeré entity, které slouží k „vyrobení“ aplikace. Těmito entitami jsou např. soubory nutné k sestavení aplikace, návrhové dokumenty a obecně dokumentace k vývoji, dále pak vztahy mezi návrhy a konkrétními zdrojovými kódy, včetně požadavků na změny a množiny již provedených změn. Metadata všech těchto konfigurací jsou uložena a spravována pomocí hlavního repositáře, kde pro jednotlivé typy položek existují verzovací komponenty, které mají specifické chování a sémantiku vhodnou pro tyto dané typy položek.

V metadatech jsou tedy uložena pravidla sestavení produktu. V určité konfiguraci mohou být zaměňovány jednotlivé komponenty, jejich verze apod. Celkový popis prostoru produktu závisí na aktuálním datovém schématu (vztahy komponent a položek). Odkazy pak mohou popisovat vztahy i na základě fází životního cyklu software a mohou nabývat různých sémantik, např. „část z“ nebo „závisí na“.

Tento způsob verzování je značně obecný a do velké míry se týká oblasti návrhu software. Hodí se k popisu vztahů mezi návrhovými komponentami a zdrojovými kódy. Toho lze využít i při testování správnosti implementace dle návrhu.

2.2.2.2 Prostor verze

Prostor verze (angl. *version space*) je množina všech možných verzí komponent systému. Každá verze (zastoupená konfigurační položkou) má jedinečný identifikátor a množinu atributů (např. OS = Unix, Language = English, …). Těchto atributů se využívá při alternativním sestavení globální konfigurace aplikace. Lze tedy poté vytvářet pravidla na základě vlastností jednotlivých verzí a jejich revizí. K tomu je ovšem nutná podpora verzovacího systému a zejména správce repositáře.

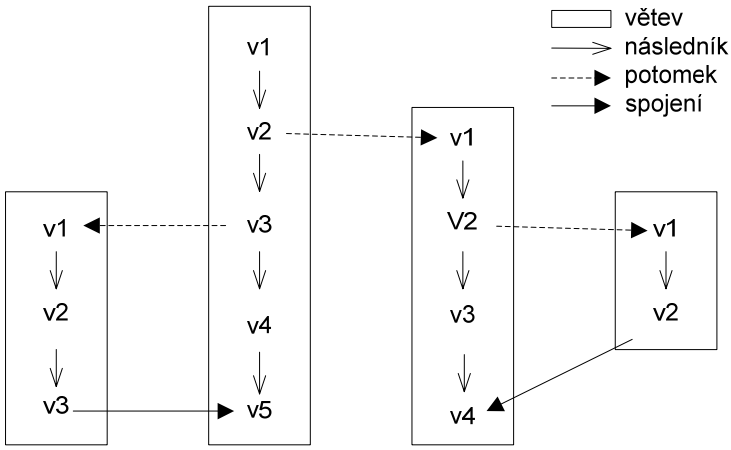
2.2.2.3 Změnová množina

Tento verzovací model se dle [2] anglicky nazývá *change-set versioning*. Česky by se dalo nazvat verzování pomocí množiny změn, nebo pomocí změnové množiny (dále bude uváděn pod zavedeným anglickým názvem). Hlavní myšlenkou je přístup ke změnám entity první třídy (nejmenší možná položka), obrácením vztahů mezi konfigurační položkou a její změnou. Místo toho, aby se každá verze konfigurační položky uchovávala a ponechávala dostupná (s nebo bez použití delta optimalizačních technik), *change-set* verzování přistupuje k uložení změny jako nezávislé na jiných změnách. Verze konfigurační položky je pak zkonstruována použitím množiny požadavků na změnu od základní verze. V klasickém verzování tedy s entitami první třídy pracuje přímo uživatel a změny jsou rekonstruovány nepřímo (např. pomocí diff nástroje), zatímco v druhém případě uživatel manipuluje se změnami a konfigurační položky musí být odvozeny nepřímo od základní verze.

Během vývoje těchto systémů však v historii docházelo k problémům, kdy rekonstrukce byly velmi složité, např. u dokumentů systémů jako je Microsoft Word. Dané techniky se tedy částečně kombinují tím způsobem, že je po určitém počtu (či velikosti) změn vytvořen kontrolní bod (angl. *checkpoint*), který vytváří novou výchozí položku.

2.2.3 Více úrovní verzování

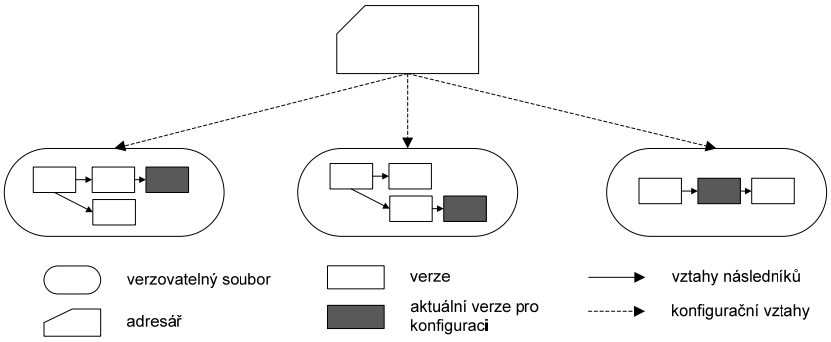
Na větvení verzí, tedy vytváření variant konfiguračních entit, lze nahlížet několika směry a v několika úrovních. Zpravidla jde o větvení na základě různých konfigurací větších celků, nebo naopak o požadavek na změnu menší entity v systému. Pohled první úrovně je vyobrazen již na obr. 1, kde je zachycen vývojový graf jedné konkrétní entity, pohled druhé úrovně již zahrnuje nejen vývojový graf entity, ale zároveň příslušnost ke konkrétní vývojové větvi vyšší úrovně (obr. 3). Dle [3] existují 3 základní náhledy na verzování softwarového produktu.



Obrázek 3 - Graf verzí (pohled 2. úrovně)

2.2.3.1 Komponentní verzování

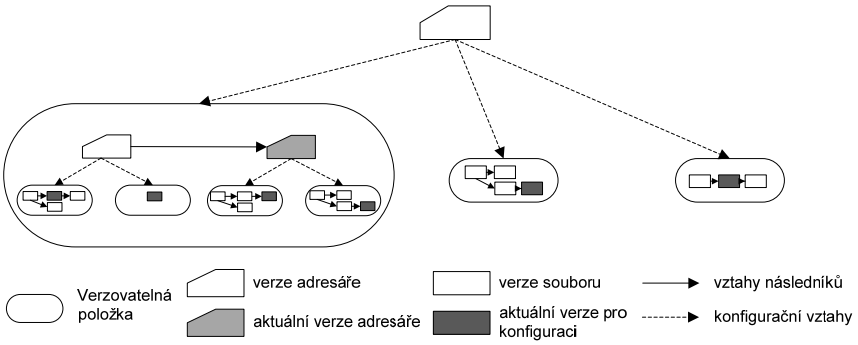
V komponentním verzování (angl. *component versioning*) jsou typicky verzovány jednotlivé položky do verzovacích grafů a zpravidla se využívá klasické verzovací techniky. Konfigurace systému je pak seskládána z jednotlivých atomických komponent. Verzovací grafy jednotlivých entit si jsou velmi málo podobné. Problém může nastat při výběru položek do konkrétní konfigurace. Prostor verze se totiž skládá z entit stejných jmen, ale různých revizí, kterých bývá velké množství.



Obrázek 4 - Komponentní verzování

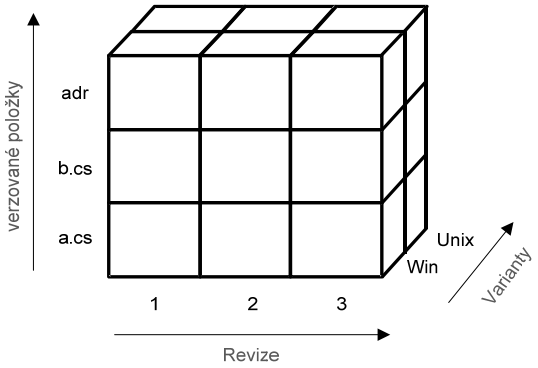
2.2.3.2 Totální verzování

Totální verzování (angl. *total versioning*) oproti komponentnímu, kde se verzují pouze listy konfiguračních vztahů, verzuje všechny objekty. Dále zde vystupují všechny objekty jako entity a jsou tak verzovány všechny objekty zvlášť (tedy např. včetně adresářů). Sestavování konfigurace pak probíhá dynamicky dle konfiguračních pravidel.



2.2.3.3 Verzování produktu

Verzování produktu (angl. *product versioning*) se dle [3] liší od totálního verzování tím, že se všechny entity nacházejí uspořádaně v jednotném globálním verzovacím prostoru. Poté je na položky nahlíženo zjednodušeně, vždy jako na výběr určité vrstvy, či skupiny entit. Dále umožňuje náhled na projekt vždy jako by byl jednou verzí, která skrývá detaily verzí jednotlivých entit a jejich vztahy. Verzovací prostor v tomto případě zahrnuje i tzv. modelový prostor produktu. Při verzování produktu se ztrácí výhoda verzovatelnosti jednotlivých entit a jednotlivé změny jsou změnami globálními. Aplikací této úrovně verzování je verzování pomocí *prostoru produktu*.



Obrázek 5 – Verzování produktu

2.2.4 Pracovní prostor verzovacích systémů

Způsob organizace pracovního prostoru výrazně ovlivňuje architekturu celého systému a spolupráci uživatelů. Jde o organizaci přístupu jednotlivých uživatelů k systému, projektům a jednotlivým souborům.

Verzovací systémy musí nějakým uniformním způsobem zajistit poskytnutí konkrétních souborů, případně jejich verzí uživateli, který si o to zažádá. Pokud tedy mají být konkrétní soubory modifikovány, je potřeba je nějakým způsobem přímo nebo nepřímo poskytnout vývojovému prostředí, nebo jen přenést k uživateli. Této operaci se říká vytažení souboru (verze), anglicky *check-out,* do pracovního prostoru. Tento pracovní prostor může však být různorodě implementován.

Může být sdílený přímo v repositáři více uživateli a o dostupnosti souboru pak rozhodují zámky, nebo podobné techniky. Případně mohou být vytvářeny kopie souborů pro každého uživatele zvlášť. Dle [18] pracovní prostor obvykle poskytuje tyto funkce:

- Pískoviště (*sandbox*) – pracovní prostor poskytuje místo, kde se dají jednotlivé soubory modifikovat.
- Sestavení (*building*) – v pracovním prostoru probíhá kompilace programů a jsou zde uloženy nově vzniklé soubory (binární soubory). Pracovní prostor také drží přístupné všechny potřebné soubory pro sestavení včetně vkládaných hlavičkových souborů a knihoven.
- Izolace (*isolation*) – typicky vytváří oddělený pracovní prostor pro každého uživatele zvlášť. Ten pak může modifikovat, kompilovat, testovat atd. bez zásahu do pracovních prostorů ostatních uživatelů, kterých se pracovní změny nijak nedotknou.

Ve starších systémech byly pracovní prostory vytvářeny pomocí skriptů, které zkopírovaly potřebné soubory z repositáře do pracovního adresáře a byly vlastně mimo systém. Modernější nástroje poskytují vytváření pracovních prostorů přímo v systému, nebo dokonce v repositáři a poskytují správu i nad těmito soubory.

2.2.4.1 Fyzické pracovní prostory

Při vývoji rozsáhlých softwarových produktů musí mít kompilátor přístup k mnoha tisícům zdrojových objektů (souborů). Tyto soubory musí být extrahovány a dekomprimovány ze souborových úložišť a vloženy do pracovního prostoru. U těchto rozsáhlých projektů existují také stovky individuálních pracovních prostorů pro jednotlivé vývojáře, což logicky vede ke stovkám tisíc kopií souborů.

Jako řešení se tedy používají sdílené prostory, nebo se prostory hierarchicky dělí, přičemž každý vývojář vlastní jeden z adresářů, a právě v něm provádí své změny. Při překladech se pak spoléhá na podporu překladačů, které dohledají příslušně cesty ve sdílených prostorech, kde se nacházejí plné kopie sdílených souborů. Izolovány jsou tedy modifikované soubory a povinně i „makefile“ soubory pro jednotlivé části vyvíjeného systému.

2.2.4.2 Virtuální pracovní prostory

Zcela jiný pohled přinášejí virtuální pracovní prostory, které se též označují pojmem pohledy. Tyto pohledy jsou implementovány přímo uvnitř operačního systému hostitelského počítače. Veškeré soubory (z projektu), které uživatel vidí, fyzicky neexistují, s výjimkou souborů editovaných. Všechny ostatní soubory jsou zpřístupněny přímo z verzovacího systému. Pokud tedy kompilátor nebo jiný nástroj vyžaduje zpřístupnění souboru, musí být tento soubor nejprve dekomprimován a vytažen z repositáře. V nutném případně musí být také zkonstruována správná revize zdrojového souboru. Z pohledu uživatele soubory vypadají, jako by byly součástí adresářové struktury

operačního systému a rozlišují se pouze pomocí atributů (pokud je soubor zamčen, má atribut pouze pro čtení, apod.). Virtuální pracovní prostory jsou rychle stvořitelné a jsou nenáročné na velikost.

Pokud bychom uvažovali přenos těchto souborů po síti, je možné vytvořit vyrovnávací paměť a často používané a neměnné soubory přenášet pouze z vyrovnávací paměti. Neposlední výhodou je možnost práce nad virtuálním pracovním prostorem z různých stanic bez nutnosti přenášet s sebou fyzické soubory. Tedy například pokud vývojář vyvíjí v zaměstnání a večer doma na svém počítači, stačí si virtuální prostor přenést na druhý počítač bez nutnosti přenášet celý obsah, tak jak by to bylo nutné při vytažení celého fyzického pracovního prostoru.

2.2.4.3 Izolace a sdílení

Rozsáhlé projekty se obvykle dělí na podprojekty, které se skládají z různorodých individuálních úkolů, nebo úkolů značně oddělených. Tento typ vývoje může přímo obsahovat několik různých pracovních prostorů, které jsou uspořádány hierarchicky. Každý uživatel má svůj jedinečný prostor, kde provádí lokální změny, a každý podprojekt (úkol) má další pracovní prostor, kde se tyto lokální změny slučují dohromady pomocí kombinačních metod. Celý projekt je pak sestaven pomocí hlavní větve, která vybírá konkrétní aktuální verze podprojektů (hlavní větve podprojektů). Výběr a správné rozvržení sdílení je rozhodující v projektech obsahujících milióny řádků kódu.

Výběrové mechanismy musí být schopny vybrat správné verze v jednotlivých úrovních projektu, a proto je dosti důležité umět správně vytvořit hierarchii těchto sdílených prostorů. V systémech založených na verzování pomocí množiny změn (*change-set*), které vlastně specifikují hlavní vývojovou větev a jen od ní se odvíjející změny, lze lehce vytvořit vždy zvláštní sdílený prostor pro každou tuto větev. U jiných, více obecných mechanismů pro rekonstruování aktuální verze, je výběr a vytvoření správných sdílených pracovních prostorů náročnější.

Virtuální pracovní prostory by se neměly hierarchicky dělit. Mechanismus pro výběr aktuální verze projektu by měl zkombinovat lokální změny, verze podprojektů a hlavní větev. Mechanismus by tedy měl zobrazit kombinaci množiny verzí jako jednu adresářovou strukturu. Vznikají tedy vyrovnávací paměti celých adresářových struktur, které jsou specifické pro konkrétní konfigurace celých projektů. Tento princip je do jisté míry typický pro verzování na úrovni produktu.

2.2.4.4 Reintegrace revizí

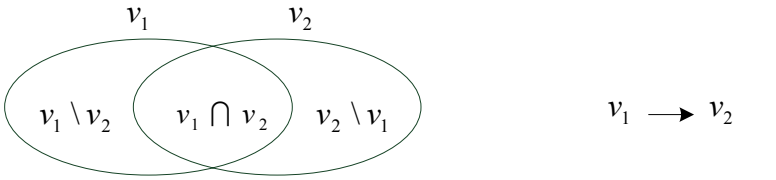
Společně se vznikem pracovních prostorů jednotlivých uživatelů přichází problém zpětné integrace soukromých revizí do vývojových větví. Je potřeba provedené změny (v izolovaných souborech) nějakým způsobem zahrnout zpět do projektu a sloučit je s originálními soubory. Většinou jde o jednoduché sloučení tím, že je vytvořena nová revize souboru (hlavní větve), a ta je považována za aktuální. Problém nastává při změně jednoho souboru více uživateli. V takovém případě je použita některá z metod sloučení (s asistencí, nebo bez asistence uživatele). Některé systémy vícenásobné

změny ani nepovolují a pro jednotlivé soubory používají zámky, pak ovšem částečně odpadá potřeba vytváření pracovních prostorů.

2.2.5 Delta technika

Delta technika (delta mechanismus) je poměrně široký pojem v oblasti verzovacích systémů. Základním rozlišením mezi výsledky delta techniky by mohlo být rozdělení na fyzickou a logickou deltu. Fyzická delta je používána v datových úložištích, kde potřebujeme zaznamenat jen nejnutnější informace o změnách mezi soubory. Logická delta může být tvořena na více úrovních a slouží k zobrazení rozdílů mezi soubory (příp. stromovými strukturami souborů). Dle [3] je v některých systémech užívána tzv. jednotná delta, kde se fyzická a logická delta reprezentuje jako jedna. Při vytváření delty se používá dvou přístupů:

- *Symetrická delta* by se dala pomocí množin definovat jako doplněk ke společnému prostoru mezi předchůdcem a následníkem. Poté je možné rekonstruovat následníka z předchůdce, ale i předchůdce z následníka. Symetrická delta také podporuje vznik tzv. *vestavěné delty* (angl. *embedded delta*), kde jsou všechny verze umístěné v jednou prostoru a mají sdílené společné sekce (viz obr. 6a).
- *Přímá delta* reprezentuje jen změnu od předchůdce k následníkovi. Tedy ukládá pevně daný počet operací, které byly nutné ke změně z podoby předchůdce do podoby následníka (viz obr. 6b). Přímá delta je na příklad použita v RCS (Revision control system) [19], kde je vlastně celý systém postaven na operacích v různých úrovních. Rekonstrukce verzí v odlišných větvích probíhá pomocí zpětné delty v hlavní vývojové větvi stromu (k bodu rozdělení) a dopředné delty v rámci konkrétní větve.



Obrázek 6 - Fyzická delta

Delta mechanismy jsou velice obsáhlou kapitolou výzkumu v SCM a vyznačují se řadou metod přístupu, jako např. kontextový, operačně orientovaný, syntakticky orientovaný atd. Tyto přístupy jsou voleny s ohledem na co nejefektivnější práci při hledání rozdílů jak při porovnání, tak při slučování větví. Součástí delta techniky je také převod textových souborů na soubory binární atp.

2.3 Přehled existujících verzovacích systémů a jejich vlastností

V předchozí části kapitoly je uveden teoretický pohled na verzování a možnosti architektur verzovacích systémů. V této části jsou porovnány vlastnosti existujících a v současné době významných verzovacích systémů, přičemž každý z nich určitým způsobem zahrnuje výběr z výše uvedených vlastností a principů.

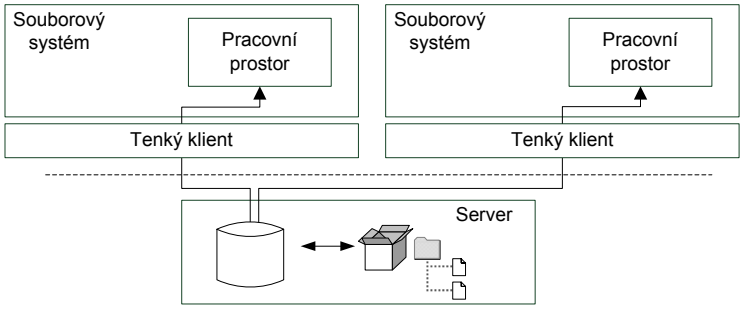
2.3.1 Typy verzovacích systémů

V současné době je k dispozici mnoho různě složitých, uživatelsky více či méně příjemných verzovacích systémů, přičemž se dělí na dvě základní skupiny, a to dle koncepce použité architektury.

2.3.1.1 Centrální

V centrálních (centralizovaných) systémech se o jednotlivé pracovní kopie repositáře (pracovní prostory) stará stále jeden centrální správce. Tyto systémy jsou většinou založeny na modelu klient-server, kde server je správcem a poskytuje služby nad repositářem pro klientské aplikace (viz obr. 7). Ke správě takového repositáře je poté možno přistupovat pomocí dvou způsobů.

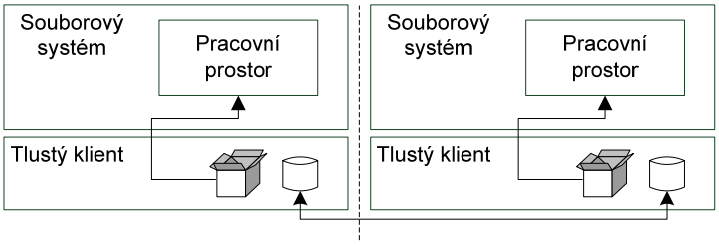
- Prvním, náročnějším na implementaci i celou správu, avšak pro uživatele neomezujícím je tzv. „Copy-Modify-Merge“ [20], kde při poskytnutí souboru klientovi není na daný soubor kladeno žádné omezení. Při zavádění nové verze však musí systém automaticky detekovat, zda soubor nebyl zároveň změněn vícekrát, a s pomocí uživatele, případně automaticky pak tyto dvě změněné verze sloučit dohromady. Je zde tedy uplatněn princip izolovaných pracovních prostorů s podporou vícenásobného sloučení.
- Druhý přístup se označuje pojmem „Lock-Modify-Unlock“. Uplatňuje se zde také princip izolovaných pracovních prostorů, ale poskytnutý soubor je uzamčen pro modifikaci a ostatním uživatelům je poskytován pouze pro čtení. Zpět odemčen je až po úspěšné aktualizaci v repositáři [8].



Obrázek 7 – Centralizovaný verzovací systém

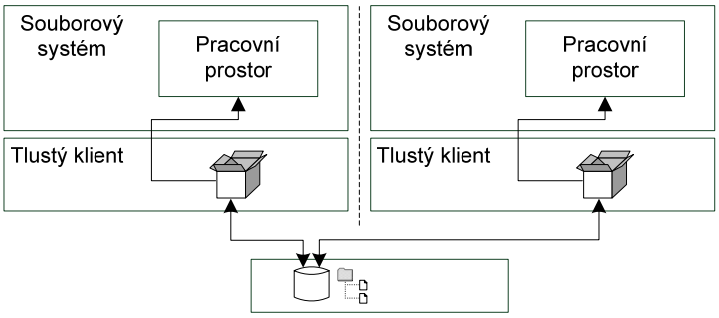
2.3.1.2 Distribuované

V distribuovaných verzovacích systémech je pracovní obsah repositáře umístěn u každého uživatele. Ten pak při změnách pomocí peer-to-peer spojení svůj repositář synchronizuje s ostatními vlastníky projektu. Na pracovní kopii tedy může uživatel provádět změny, aniž by omezoval ostatní vývojáře. Synchronizace pak musí poskytovat slučování více verzí zejména z principu, že v systému nelze uplatnit zamykání souboru. Nevystupuje zde ani centrální správce, který by přiděloval pracovní prostory (viz obr. 8).



Obrázek 8 - Distribuovaný verzovací systém

Takto výhradně distribuované systémy neposkytují dostatečnou míru automatizované spolupráce, a proto se většinou kombinují s centralizovanými prvky. Pracovní prostory jsou sice oddělené a distribuované mezi uživatele, ale existuje centrální správce, který u sebe obsahuje alespoň aktuální verze a případně umí zprostředkovat starší (viz obr. 9).



Obrázek 9 - Distribuovaný verzovací systém s centralizovaným správcem

2.3.2 Základní vlastnosti verzovacích systémů

Mimo základní typové rozdělení (viz odstavec 2.3.1) lze verzovací systémy porovnávat i dle funkčních vlastností. Těmito vlastnostmi jsou např.:

- **atomické vložení** (angl. *atomic commit)* – vlastnost systému garantující vložení a sloučení verzí vždy plně automaticky (i při vícenásobných paralelních změnách).
- Schopnost **přejmenování** jednotlivých souborů bez dopadu na historii verzí a s tím související schopnost **slučovaní přejmenovaných souborů.** Tyto systémy poté umožňují sloučit soubory i v takovém případě, kdy jeden z nich byl přejmenován pouze v jedné vývojové větvi, a zároveň je tento soubor ve druhé větvi veden pod původním jménem.
- **Podpora symbolických odkazů** (*symbolic links)* – podpora systému přistupovat k symbolickým odkazům tak, jako by se jednalo o soubory.
- **Vytváření tzv. tagů**, tedy značek revizí. Revizí se rozumí označení všech aktuálních verzí číslem revize. Kdykoli v budoucnu lze na tuto revizi nahlížet globálně, jako by byla právě poslední aktuální verzí celého systému.
- **Sledování sloučení** (*merge tracking*) – schopnost systému rozlišit a pamatovat si trasování rozdělení a sloučení jak v případech, kdy je jedna vývojová větev sloučena s druhou, tak v případech, kdy je jedna větev sloučena do jiné.
- Existují i mnohé další vlastnosti, kterými se vyznačují různé verzovací systémy a které více či méně ulehčují vývojářům práci. Např. nezávislost klientských platforem u textových souborů (ignorace různě interpretovaných konců řádků apod.), či podpora open source dokumentačních standardů [9]. Mimo funkční vlastnosti lze verzovací systémy rozdělit dle typu implementace, či užití různých komunikačních protokolů. Zatímco některé systémy využívají pouze vlastního komunikačního protokolu, jiné podporují hned několik standardů, jako je HTTP, FTP, SSH, rsync a další.

2.3.3 Příklady verzovacích systémů

Jak již bylo naznačeno výše, vzniklo v minulosti poměrně velké množství verzovacích nástrojů a systémů. Mezi větší počet uživatelů se jich však dostalo jen několik, nebo vycházejí z velmi podobných principů. V následujících odstavcích budou představeni typičtí zástupci.

2.3.3.1 CVS - Concurrent Versions System

CVS je pravděpodobně nejznámějším verzovacím systémem. Přes své stáří patří stále k oblíbeným a většina open-source projektů i komerčních firem jej používá. Má však řadu vlastností, které způsobují práci poněkud komplikovanější, než by si vývojář přál. Některé vlastnosti CVS:

- Každý soubor je číslován zvlášť.
- Verze lze označit (tag), existují dynamické a statické tagy.

- Rozlišuje mezi adresářem a souborem.
- Implicitně pracuje s textovými soubory, binární soubory je nutné označit.
- Náročnější slučování větví.
- Obtížný přesun adresářů či přejmenování souborů.
- Samotný program CVS je dodáván v základní verzi pro použití v příkazovém řádku, existují však i nadstavby pro použití v grafickém režimu, jako je třeba WinCVS, apod.

2.3.3.2 SVN – SubVersion

SVN je novějším zástupcem CVS, který se snaží pokrýt vlastnosti, pro které je CVS kritizováno. Asi největší rozdíl je v číslování verzí. CVS čísluje každý soubor zvlášť, naproti tomu SVN čísluje vždy celý repositář. Další vlastnosti:

- Nerozlišuje mezi souborem a adresářem.
- Více možností volby přístupu (ssh, svnserver, apache2 mod_dav_svn module).
- Pro ukládání používá Berkeley Data Base, případně nativní souborový systém.
- Neznačkuje jednotlivé soubory, ale vytváří větve (používá levné kopie souborů/adresářů).
- Jednodušší slučování větví.
- Snadný přesun souborů.
- Na uživatelské úrovni nerozlišuje textové a binární soubory.

2.3.3.3 GNU Arch

Asi největším rozdílem Archu od CVS je velmi jednoduchá komunikace přes síť. Lze totiž použit prakticky jakýkoliv protokol pro přenos souborů (HTTP, FTP, rsync,…). Díky této vlastnosti je však v systému značně neefektivně řešeno ukládání repositáře a celá komunikace je velmi pomalá. Další vlastnosti:

- Podporuje vícenásobné slučování souborů.
- Plně podporuje distribuovaný vývoj.

2.3.3.4 Microsoft Visual SourceSafe

Je zástupcem komerčního verzovacího systému. Tato aplikace je plně integrovaná s vývojovými prostředími aplikace Visual Studio a s aplikacemi sady Microsoft Office. Dále umožňuje práci se všemi typy souborů vytvořenými jakýmkoli vývojovým jazykem, nástrojem nebo aplikací. Uživatelé mohou pracovat na úrovni souborů nebo projektů. Dalšími vlastnostmi je správa celých návrhů WWW serverů, či kontrola aktuálnosti hypertextových odkazů [10]. Celý systém je pak určen spíše pro jednotlivce, či malé vývojové týmy. Pro větší týmy je pak určeno speciální vývojové prostředí s již zabudovaným verzovacím systémem Visual Studio Team System (viz [11]).

2.4 Práce s verzovacími systémy

Již z předchozí kapitoly vyplývá, že existující verzovací systémy přistupují ke komunikaci s uživatelem velmi různorodě. Prakticky všechny starší systémy nabízejí své služby pouze přes příkazový řádek a jsou velmi složitě parametrizovatelné. Tento přístup poměrně znesnadňuje přímou interakci uživatele a systému, naopak je však lze velice dobře použít při vytváření skriptů, díky kterým lze zefektivnit běžnou rutinní činnost. Naproti tomu jiné systémy povolují obsluhu pouze pomocí grafického rozhraní, což je pro obyčejného uživatele jistě mnohem příjemnější, avšak chybí zde podpora automatizace činností. Některé systémy ovšem již automatizaci obsahují přímo ve volbách GUI aplikace. Poslední a poměrně rozšířenou skupinou jsou grafické nadstavby nad staršími, ale mezi uživateli rozšířenými systémy. Ty zachovávají správcům možnost vytvářet skripty, pro obyčejného uživatele pak umožňují komfortní ovládání pomocí grafického uživatelského rozhraní. V následujících odstavcích bude podrobněji představeno několik zástupců verzovacích systémů právě z hlediska UI.

2.4.1 Verzovací systémy bez grafického uživatelského rozhraní

V této skupině verzovacích systémů se objevují především zástupci koncepčně starších systémů. Většina z nich byla vytvořena pro potřeby vývoje open source software a jsou tvořeny jako konzolové aplikace unixových systémů. Veškerá interakce uživatele a systému probíhá pomocí příkazů a parametrů daného programu. Při práci s těmito systémy je třeba znát jejich koncepci, způsob organizace repositáře apod. Tyto vědomosti jsou potřebné k práci se systémem zejména proto, že většina příkazů a parametrů vychází právě z tohoto konceptu. V této práci není nutné detailně popisovat práci s jednotlivými systémy, avšak pro názornost je níže uvedeno několik základních příkazů nejznámějšího zástupce této kategorie, a to systému CVS.

```
# --- CVS základní příkazy ---
# vytažení souboru/adresáře z repozitáře
$ cvs checkout soubor
# vytažení starší verze (1.3) souboru/adresáře z repozitáře,
$ cvs checkout –r soubor-1-3 soubor
# přidaní souboru/adresáře do repozitáře
$ cvs add adr/soubor2
# aktualizace lokální pracovní kopie projektu z repozitáře (verze 1.5)
$ update –r adr-1-5
# sloučení pracovní kopie a repozitáře
$ cvs commit
```

Sledovat změny ve verzích souborů a adresářů pak lze pouze pomocí kontrolních výpisů či logů. Rozdíly mezi jednotlivými soubory jsou zobrazovány pomocí externích nástrojů (např. pomocí programu diff, který je obsažen v unixových systémech).

```
# --- CVS - zkrácený výpis informací o značkách a větvých ---
$ cvs status –v ChangeLog
File: ChangeLog         Status: Needs Patch

 Working revision:    1.25.2.27
 Repository revision: 1.25.2.48     /cvs/proj/ChangeLog,v
 Sticky Tag:          soubor-1-2 (branch: 1.25.2)
 Sticky Date:         (none)
 Sticky Options:      (none)

 Existing Tags:
      soubor-2-0                        (branch: 1.1085.2)
      soubor-2-0-branchpoint            (revision: 1.1085)
      SOUBOR_2_0_1                      (revision: 1.1079)
      SOUBOR_2_0_0                      (revision: 1.1060)
      SOUBOR_2_0_0_RC1                  (revision: 1.1055)
      …
```

2.4.2 Verzovací systémy založené na grafickém uživatelském rozhraní

Verzovací systémy založené na grafickém uživatelském rozhraní jsou většinou vyvíjeny komerčními společnostmi pro běžné uživatele, mnohem více však pro početné vývojářské týmy. Běžný uživatel ocení jednoduchost a přehlednost ovládání, velké týmy se potom neobejdou bez přehledného a uniformního prostředí. Rozdíly mezi jednotlivými verzemi, pohledy do historie atp. jsou zobrazovány v různých nástrojích velmi podobně. Diference mezi jednotlivými verzemi souboru se zobrazují ve dvou, či více vedle sebe umístěných textových oknech. Chybějící (přebývající) text je rozlišen barvou textu, či podbarvením pozadí. Některé programy pak celé vypuštěné bloky pouze označují pomocí barevných čar, naznačujících vsunutí daného textu v druhé verzi (viz příloha B obr. B-1 a B-2). Většina grafických nadstaveb je určena pro konkrétní programovací jazyky, a tak pro ještě lepší názornost umí zvýraznit i syntaxi daného jazyka.

2.4.3 Grafické nadstavby klasických systémů

Tato skupina je v dnešní době asi nejrozšířenější formou verzovacích systémů pro běžné uživatele. Přestože většinou vychází jen ze tří základních systémů, a to CVS, SVN a GNU Arch, je těchto nadstaveb poměrně velké množství. Některé byly vytvořeny za určitým specifickým účelem, většina však pouze různým způsobem přistupuje k zobrazení historie, vývoje jednotlivých větví (viz např. příloha B obr. B-3) a samozřejmě také k zobrazení rozdílů mezi jednotlivými soubory. Některé grafické nadstavby v sobě také nesou přizpůsobení pro jinou platformu či operační systém.

Nedílnou součástí této, ale i předchozí skupiny verzovacích systémů je integrace do IDE jednotlivých programovacích jazyků. Programové systémy jsou tedy zpravidla určeny pro konkrétní programovací jazyk a některé jeho vývojové nástroje. Daný systém při instalaci vloží zásuvný modul do daného IDE, kde je pak pomocí hlavní nabídky, či nástrojové lišty možno volat externí verzovací nástroj, či nástroj pro zobrazení diferencí verzí. Někdy jsou tyto zásuvné moduly vyvíjeny samostatně, nezávisle na verzovacím systému. Příkladem je produkt VssConneXion společnosti EPocalipse Software, který integruje Visual SourceSafe do IDE nástrojů společnosti Borland (viz příloha B obr. B-4, více o VssConneXion v [15]).

3 Synchronizace adresářové struktury

Při synchronizaci lokální adresářové struktury a struktury repositáře je třeba uživateli nabídnout přehledné zobrazení diferencí jak mezi jednotlivými strukturami, tak mezi jednotlivými soubory (verzemi souborů).

3.1 Porovnání a synchronizace adresářů

Výsledkem porovnání adresáře by měl být seznam souborů čtyř možných stavů. Soubor může být na jedné straně přebývajícím, chybějícím, změněným nebo totožným, přičemž změna či totožnost se může porovnávat podle několika atributů současně. U souboru jde typicky o porovnání dle jména, velikosti a obsahu. Porovnání dle jednotlivých dat a atributů souboru se neuvažuje. Pro porovnání celých adresářových struktur je možné použít rekurentně metody určené k porovnání rozdílů mezi řetězci. Každý soubor je v tomto případě abstrahován na úroveň znaku řetězce.

Se synchronizací adresářů úzce souvisí i zobrazení rozdílů mezi soubory. Přehledné zobrazení těchto rozdílů umožní lepší náhled na to, který soubor je aktuální a který má být při synchronizaci uvážen jako novější. U verzovacích systémů je tato možnost ještě rozšířena o zakládání nových verzí souboru.

Pokud bychom uvažovali plně automatickou synchronizaci adresářových struktur, bez zásahu uživatele, není možné určovat aktuálnost souboru dle jeho obsahu (neuvažujeme shodné soubory), ale dle data poslední změny. To s sebou ovšem nese požadavek aktuálnosti času na všech entitách systému. Toho se dá dosáhnout například pomocí časové synchronizace s časovými servery, či užitím nějakého časově-synchronizačního algoritmu.

Poměrně mocným nástrojem při samotné synchronizaci adresářů je použití programu či pouze algoritmu založeném na protokolu rsync. Rsync je unixová aplikace založená na modelu klient-server, která slouží (nejen) k přímé synchronizaci adresářů. Jeho síla je především v množství přenášených dat. Kvůli úspoře dat je vždy pomocí programu rdiff vygenerován delta rozdíl a pouze ten je poté přenesen. Zároveň je podporována další komprimace přenášených dat a komunikace pomocí zabezpečeného spojení (SSH). Více o rsync v [6].

Synchronizace adresářů se však netýká jen SCM a verzovacích systémů, ale je poměrně důležitou součástí každodenní činnosti i obyčejného uživatele. Využívá se zejména při zálohování důležitých dat, dokumentů a v neposlední řadě i fotografií. Proto lze nalézt velké množství nástrojů umožňujících automatizovat tuto činnost a bývá již obvykle i součástí ovladačů externích pevných disků či USB flash disků.

3.2 Porovnání řetězců

Porovnání dvou řetězců je v mnoha prostředích důležitým a někdy dokonce hlavním problémem výzkumu. Příkladem mohou být programy pro vyhledávání chyb v psaném textu, v molekulární biologii pak porovnání DNA řetězců či nacházení podobností v proteinových sekvencích. Ve verzovacích systémech je tento problém klíčový při hledání rozdílů mezi verzemi jednotlivých souborů. Hlavním kritériem v porovnání dvou souborů je nalezení největšího počtu shodných znaků v závislosti na jejich uspořádání. Obecně se tomu říká problém nalezení nejdelší společné podposloupnosti (angl. *Longest Common Subsequence – LCS*).

3.3 LCS - Nejdelší společná podposloupnost

Dle [4] se LCS definuje takto:

Mějme dva vstupní řetězce $X[1..m]$ a $Y[1..n]$, které jsou elementy množiny Σ^*, kde Σ označuje *vstupní abecedu* složenou ze symbolů σ. *Podposloupnost* (*p*) $S[1..s]$ z $X[1..m]$ je získána vymazáním $m - s$ symbolů z X. *Společná podposloupnost* (*cs*) $X[1..m]$ a $Y[1..n]$, značená $cs(X, Y)$, je podposloupnost, kterou obsahují oba řetězce. *Nejdelší společná podposloupnost* (*lcs*) řetězců $X[1..m]$ a $Y[1..n]$ je tedy společná podposloupnost největší délky.

LCS problém je speciálním případem tzv. *problému vzdálenosti změn* (angl. *edit distance problem*). Vzdálenost mezi řetězci *X* a *Y*, *dist(X, Y)*, je definována jako minimální počet elementárních operací potřebných k transformaci řetězce *X* na řetězec *Y*. V praxi je považováno za elementární operaci vložení, smazání a substituce symbolu řetězce (u souborů se pak může jednat o řádky).

Při pokusu řešit LCS pomocí „naivního“ přístupu hledání dosáhneme časové složitosti $O(n\,2^m)$. Pro řešení LCS jsou však známy dva základní způsoby řešení pomocí dynamického programování, při nichž lze dosáhnout složitosti polynomiální.

3.3.1 Výpočet LCS pomocí tabulky

Jde o tradiční techniku při řešení LCS, která funguje na principu zjištění všech možných kombinací prefixů vstupních řetězců. Rekurentní ohodnocující funkce pro prodloužení LCS je pro každý pár prefixů ($X[1..i]$ a $Y[1..j]$) následující [5]:

$$LCS[i,j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0 \\ LCS[i-1, j-1] + 1 & \text{if } X[i] = Y[j] \\ \max\{LCS[i-1, j], LCS[i, j-1]\} & \text{if } X[i] \neq Y[j] \end{cases} \quad [1]$$

Algoritmicky se pak tento problém řeší pomocí tabulky, kde se vynulovaná tabulka o rozměrech $i \times j$ vyplňuje dle rovnice 1. Výpočet tedy probíhá tak, že se zároveň počítá cena od počátku řetězce a zároveň se označuje cesta postupu. Cesta zpět je pak zrekonstruována tak, že se

vychází od poslední položky (vždy má nejvyšší ohodnocení) a postupuje dle šipek. Pokud je nalezena šipka odkazující nahoru vlevo, je znak přidán na začátek výsledného řetězce. Časová složitost algoritmu je $O(n\,m)$, prostorová složitost odpovídá dvěma pomocným polím, tedy $O(2n\,m)$.

Algoritmus:

```
function LCS (X, Y){
     m=length(X); n=length(Y);
     for (i=0;i<=m;i++) a[i,0] = 0;
     for (j=0;j<=n;j++) a[0,j] = 0;
     for (i=0;i<=m;i++)
       for(j=0;j<=n;j++)
         if (X[i]==[j]){
           a[i,j]=a[i-1,j-1]+1;
           b[i,j]="↑←";}
         else
           if (a[i-1,j]>=a[i,j-1]){
             a[i,j]=a[i-1,j];
             b[i,j]="↑";}
           else{
             a[i,j]=a[i,j-1];
             b[i,j]="←";}
return a,b;
}
```

Příklad 1:

Zadání: $\Sigma = \{\text{ACTG}\},\ X = \text{AGTCAACGTT}, Y = \text{GTTCGACTGTG}.$

		A	G	T	C	A	A	C	G	T	T
	0	0	0	0	0	0	0	0	0	0	0
G	0	0	↖**1**	← 1	← 1	← 1	← 1	← 1	↖ 1	← 1	← 1
T	0	0	↑ 1	↖**2**	← 2	← 2	← 2	← 2	← 2	↖ 2	↖ 2
T	0	0	↑ 1	↖ 2	← 2	← 2	← 2	← 2	← 2	↖ 3	↖ 3
C	0	0	↑ 1	↑ 2	↖**3**	← 3	← 3	↖ 3	← 3	← 3	← 3
G	0	0	↖ 1	↑ 2	↑ 3	← 3	← 3	← 3	↖**4**	← 4	← 4
A	0	↖ 1	← 1	↑ 2	↑ 3	↖**4**	↖ 4	← 4	← 4	← 4	← 4
C	0	↑ 1	← 1	↑ 2	↖ 3	↑ 4	← 4	↖**5**	← 5	← 5	← 5
T	0	↑ 1	← 1	↖ 2	↑ 3	↑ 4	← 4	↑ 5	← 5	↖ 6	↖ 6
G	0	↑ 1	↖ 2	↑ 2	↑ 3	↑ 4	← 4	↑ 5	↖**6**	← 6	← 6
T	0	↑ 1	↑ 2	↖ 3	← 3	↑ 4	← 4	↑ 5	↑ 6	↖**7**	↖ 7
G	0	↑ 1	↖ 2	↑ 3	← 3	↑ 4	← 4	↑ 5	↖ 6	↑ 7	← 7

Výsledkem je tedy podposloupnost $\text{LCS} = \{\text{GTCACGT}\}$.

3.3.2 Výpočet LCS pomocí LIS

Druhou možností je aplikovat při výpočtu problém hledání *nejdelší rostoucí podposloupnosti* (*LIS – Longest Increasing Subsequence problem*). V LIS problému je posloupnost Z složená ze z celočíselných hodnot. V tomto případě je úkolem najít nejdelší podposloupnost v Z, která je přísně rostoucí. Vstup do LIS je získán ze vstupu LCS následovně: Vezmeme $X[1]$ a najdeme všechny pozice j v Y, pro které platí $X[1] = Y[j]$. Pozice zaznamenáme a seřadíme sestupně. Poté získáme hodnoty pro $X[2]$ a přidáme výsledky k prvním. Toto provedeme pro všechny prvky v X. Z výsledků vznikne graf sestupných hodnot, připomínající zuby pily. V tomto grafu pak nalezneme nejdelší přísně rostoucí posloupnost (viz příklad 2 a obr. 10). Tento způsob řešení je výhodný zejména v případech, kdy se poměrně malá množina vstupních symbolů vyskytuje v posloupnosti vícekrát.

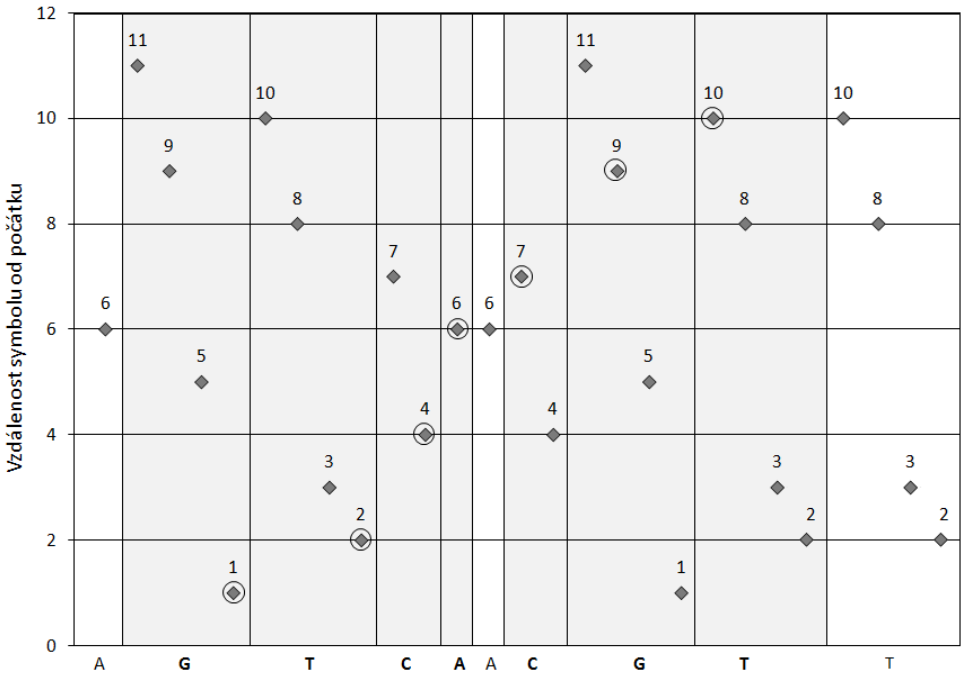
Příklad 2:

Zadání: $\Sigma = \{\text{ACTG}\},\ X = \text{AGTCAACGTT}, Y = \text{GTTCGACTGTG}.$

A	6			
G	11	9	5	**1**
T	10	8	3	**2**
C	7	**4**		
A	**6**			
A	6			
C	**7**	4		
G	11	**9**	5	1
T	**10**	8	3	2
T	10	8	3	2

Řešení:

$\text{LCS} = \{\text{GTCACGT}\}$



Obrázek 10 - Užití LIS při řešení LCS

3.3.3 Optimalizace LCS

Základní algoritmus LCS je použitelný pro kratší řetězce, tedy při porovnání souborů pro menší soubory, resp. soubory s menším počtem řádků. Zpravidla se totiž textové soubory porovnávají nejprve dle řádků, případně až poté po znaku. A to pouze u případů, kdy je daný řádek substituován za jiný. Algoritmus je náročný jak prostorově, tak časově.

Pokud však vycházíme z předpokladu, že zejména u zdrojových kódů programů se začátky a konce souboru prakticky neliší a změny probíhají pouze uvnitř, je možné před zahájením výpočtu zredukovat počet porovnávaných řádků, čímž můžeme výrazně zredukovat velikost matice. Absolutní počet porovnání sice klesá, avšak časová složitost zůstává stejná.

Časovou složitost můžeme snížit pomocí zakódování jednotlivých řádků textu do kontrolního součtu, či hash kódu. To je ovšem výhodné pouze v případech, kdy průměrná délka řádku přesahuje cca 60 znaků. Porovnáván je tedy pouhý hash, kde je velmi málo pravděpodobné, že dojde k zakódování dvou řádků do stejného kódu.

Více o různých typech optimalizací a algoritmů pro určité konkrétní problémy např. v [4] a [7].

4 Návrh verzovacího systému

V praktické části práce byl navržen a implementován obecně použitelný verzovací systém orientovaný na projekty strukturované standardním souborovým systémem. Architektura systému jako celku a komunikačního protokolu byla navrhována ve spolupráci s R. Černobilou [23]. V [23] je detailněji popsána architektura serverové části.

4.1 Požadavky na systém

Cílem bylo vytvořit centralizovaný verzovací systém, který nebude primárně určen pro zdrojové kódy programů. Naopak měl být co nejobecněji použitelný a přizpůsobitelný konkrétnímu nasazení. Dále měl systém sloužit jako archivační úložiště s možností oddělení logické a fyzické struktury souborového systému a měl podporovat i rozdělení z hlediska výpočetní a prostorové kapacity. Systém měl podporovat víceuživatelský a vzdálený přístup, definovat přístupová práva a oprávnění. Z hlediska verzovacích služeb měl poskytovat základní operace vytvoření a sloučení větve projektu (*fork*, *merge*), vytvoření nové verze souboru a poskytnutí projektu a souborů v něm (*check-in*, *check-out*). Z hlediska uživatele se mělo jednat o přizpůsobitelný systém pro práci s různými typy projektů, či různé úrovně složitosti aplikace.

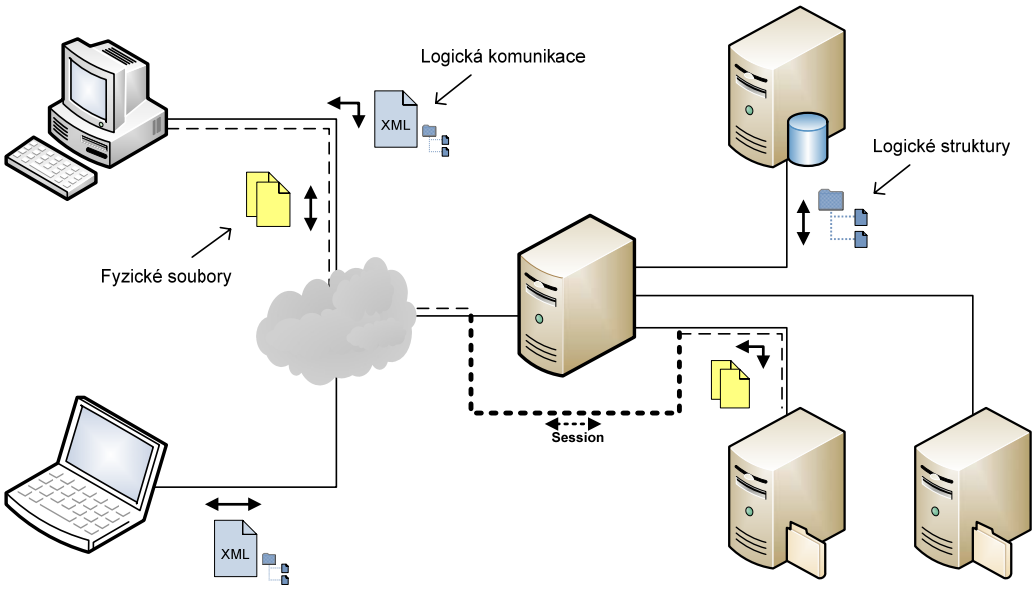
4.2 Návrh a implementace

Při návrhu se vycházelo z kritérií, které byla vznesena při určování požadavků na systém. Dalším ze základních předpokladů bylo provést návrh tak, aby bylo možné jednotlivé části systému vyvíjet značně nezávisle.

4.2.1 Architektura systému

Celá architektura je založena na modelu klient-server, kde klientská část poskytuje uživatelské rozhraní celému systému a je značně nezávislá na části serverové (počítá se s různou implementací klientské aplikace přizpůsobené konkrétní potřebě použití koncového uživatele). Serverová část pak obstarává veškerou logiku verzování a organizace fyzických úložišť. Více dle obr. 11.

Klientská aplikace komunikuje pomocí XML souborů se serverem v rámci logického schématu projektů, požadavků na soubory apod. Samotné poskytnutí, resp. nahrání souboru poté zajišťuje komunikace klienta přímo se souborovými servery. Logicky se tedy serverová část tváří jako by byla jedním serverem. Ve skutečnosti je však zcela oddělena databázová část spravující logické schéma projektů, komunikační server pro verzovací službu a server s datovými úložišti. Databázových a souborových serverů může být v případě potřeby i několik.



Obrázek 11 - Koncepce systému

4.2.2 Komunikační protokol

Komunikační protokol je založen na dvou typech spojení:

1. Logickém – sloužícím k logické komunikaci mezi klientem a serverem. Jak již bylo naznačeno výše, logická komunikace probíhá plně v rámci XML streamů s přesně definovanou strukturou. Tato varianta je využita zejména z důvodu kompatibility s případně jinými klienty, či systémy, které se lehko na daný protokol mohou modifikovat. Struktura XML souborů pak plně vychází ze serializovaných objektů pomocí .NET XML serializace (viz zdrojový kód 1).

```
namespace Server
{
    [Serializable, XmlRoot("Request")]
    public class RequestLogin
    {
        public string Login = "";
        public string Password = "";
    }


    [Serializable, XmlRoot("Answer")]
    public class AnswerLogin
    {
        public bool IsLogged = false;
        public string Result = "";
    }
}
```

```
<?xml version="1.0" ?>
<Request xmlns:xsi="http://www.w3.org/... "
         xmlns:xsd="http://www.w3.org/...">
      <Login>host</Login>
      <Password>heslo</Password>
</Request>

<Answer xmlns:xsi="http://www.w3.org/... "
        xmlns:xsd="http://www.w3.org/...">
      <IsLogged>true</IsLogged>
      <Result>Uživatel zalogován.</Result>
</Answer>
```

Zdrojový kód 1 - Příklad XML komunikační třídy a její instance

2. Fyzickém (přenosovém) – zajišťujícím přenos konkrétních fyzických souborů. Datové přenosy, resp. přenosy fyzických souborů jsou realizovány pomocí „přemostění“ na jiný server.

Poskytnutí přístupu k datům na souborovém serveru probíhá v následujícím pořadí.

Komunikace s logickým serverem:

1. Přihlášení, stažení struktur apod. – logická komunikace předcházející požadavku.
2. Vyžádání souboru – klient si vyžádá množinu souborů.
3. Předání identifikátoru – server ověří dostupnost, práva apod. a předá identifikátor (ten by se dal přirovnat k souborovému deskriptoru v OS) a session, která slouží k zabezpečení datového přenosu a úložiště.

Komunikace se souborovým serverem:

4. Vyžádání fyzického souboru – klient požaduje konkrétní soubor na odpovídající URL adrese (identifikátoru) a autentizuje se pomocí session.
5. Server poskytuje soubor.
6. Odpojení klientské části od souborového serveru.

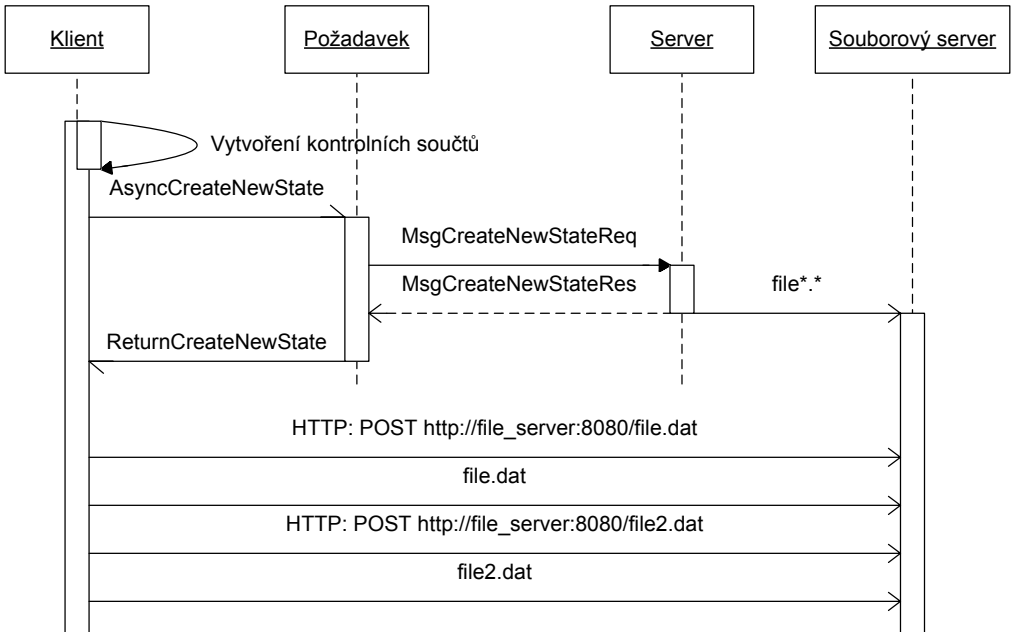
7. Klientská část dále komunikuje s logickým serverem a pokračuje v dalších činnostech, případně vyžaduje po souborovém serveru další soubor.

Veškerá komunikace probíhá v rámci protokolu HTTP. Je třeba tedy při vývoji dbát na všechny vlastnosti, které HTTP protokol má, zejména na bezstavovost. Ta je však uměle potlačena, tak jak je tomu např. u klasických webových HTTP serverů, a to pomocí session.

4.2.2.1 Implementace jednotlivých operací v protokolu

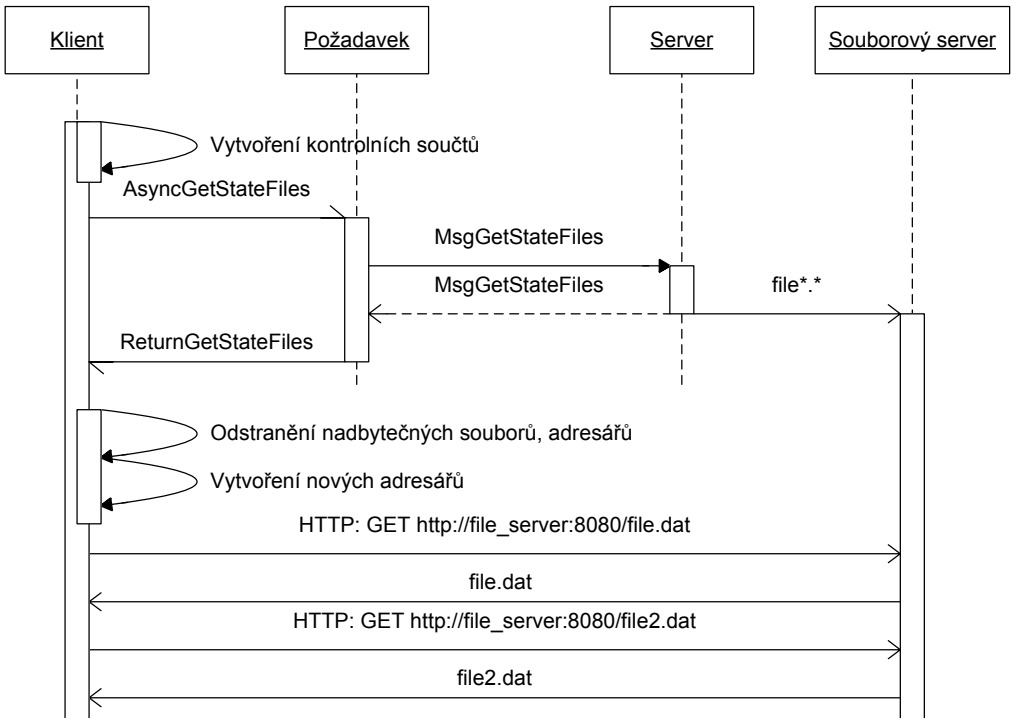
Jak je naznačeno výše, je většina operací v rámci protokolu založena na principu požadavku a provedení/neprovedení na straně serveru. Dále by se jednotlivé požadavky daly rozdělit na požadavky zjišťovací a funkční příkazy. U příkazů jde například o vytvoření nového projektu či větve projektu. U zjišťovacích jde o poskytnutí informací o projektech, větvích či konkrétních adresářových strukturách stavů větví projektů.

Zvláštním případem komunikace je synchronizace strany serverové a lokální. Ta je vždy založena na porovnání adresářových struktur. V případě aktualizace repositáře (vytvoření nového stavu větve) jsou vytvořeny kontrolní součty jednotlivých souborů a pak společně s celou adresářovou strukturou odeslány na server s požadavkem na vznik nové větve. Server provede pomocí synchronizačního algoritmu porovnání a vrátí seznam souborů, které je třeba aktualizovat. Klientská část společně se seznamem souborů obdrží i URL adresy pro odeslání souborů, na které jednotlivé soubory odešle (viz obr. 12).



Obrázek 12 - Synchronizace: Vytvoření nového stavu

V případě aktualizace lokální části je postup obdobný (viz obr. 13). Aby byla dodržena vlastnost jednoduchosti klienta a zejména vlastnost, že logiku verzování obstarává server, je výpočet rozdílu opět ponechán na straně repositáře. Postup je tedy následující. Nejprve odešle klientská část obraz adresářové struktury společně s kontrolními součty na stranu serveru a poté obdrží tzv. synchronizační seznamy. V těchto seznamech jsou uvedeny adresáře a soubory, které jsou určeny pro vymazání. Další seznam obsahuje adresáře, které naopak mají být vytvořeny, a v posledním je uveden výčet souborů společně s URL adresami, odkud je možné dané soubory momentálně aktualizovat.



Obrázek 13 - Synchronizace: Aktualizace lokálního projektu

4.3 Použité principy verzování v systému

V navrhovaném systému byla uvažována řada přístupů k verzování na různých úrovních. Výsledně byl vybrán princip, který nejlépe reflektuje potřeby a organizaci obecných projektů. V úvahu nebyla brána specifická hlediska pro vývoj softwarových produktů, jako je možnost integrace s vývojovými prostředími, či napojení pracovních prostorů na specifické kompilátory.

4.3.1 Verzování na úrovni projektu

Na nejvyšší úrovni pohledu v systému se vyskytují **projekty**. Tyto projekty jsou dostupné konkrétním uživatelům vlastnícím odpovídající oprávnění. Pro konfiguraci těchto nejvyšších položek v systému je využíváno tzv. **větví**. Jak název napovídá, je každá tato větev vývojovou větví projektu. Tyto větve jsou vytvářeny explicitně uživatelem pro konkrétní potřebu a jsou verzovány pomocí klasického verzování. Jsou tedy rozdělitelné, slučitelné a podporují vytváření vlastních revizí. Revize větve je vytvořena vždy při aktualizaci adresářové struktury projektu. Pokud je při aktualizaci zjištěna nová verze některého ze souborů, je vytvořena tato nová revize. Jednotlivé větve mohou také nabývat různých manipulačních stavů. Mohou vystupovat jako větve pozastavené, ukončené apod. Na těchto větvích již není možné pracovat, tzn. vytvářet nové revize, samozřejmě jsou však přístupné jejich adresářové struktury pro čtení.

4.3.2 Verzování na úrovni objektu

Systém samozřejmě podporuje verzování i nejmenších entit systému (souborů), avšak pouze na úrovni revizí. Každý soubor je při své aktualizaci prohlášen za novou revizi a vystupuje jako výchozí revize souboru v té dané větvi. Vytváření variantních vztahů je ponecháno pouze na úrovni větví. Revidování souborů je v systému implementováno pouze pro možnosti sledování změn souboru ve větvi.

4.3.3 Organizace pracovních prostorů

Jak je již patrné z návrhu architektury systému a požadavků na systém, je pracovní prostor navržen jako decentralizovaný fyzický pracovní prostor. Každý uživatel vlastní svoji vlastní kopii pracovní adresářové struktury, se kterou může libovolně manipulovat. Systém tedy nesleduje práci v pracovním prostoru, a tak vystupuje spíše jako archivační verzovací systém. Tento přístup je ideální z hlediska obecnosti použití a zároveň pro vzdálenou a centralizovanou správu repositáře. Hlavní nevýhodu fyzických pracovních prostorů, tedy nutnost přenášení celých adresářových struktur, systém částečně potlačuje tím, že fyzický prostor je vně systému. Není tedy potřeba vždy při zahájení práce přenášet celou adresářovou strukturu, ale pouze aktualizovat změněné soubory.

5 Návrh a implementace klientské části systému

V této kapitole je popsán návrh a vybrané implementační detaily klientské části systému. Z důvodu přehlednosti a srozumitelnosti architektury je popis systému do jisté míry abstrahován. Také z důvodu většího rozsahu implementace zde nejsou popisovány všechny detaily vytvořené aplikace a knihoven.

5.1 Požadavky na klientskou část systému

Hlavním účelem klientské aplikace bylo uživatelsky přívětivým způsobem poskytnout správu verzovaného projektu. Pod pojmem správa se očekává poskytnutí kompletních operací pro manipulaci s projektovým stromem i jednotlivými soubory a jeho verzemi. V základním okně aplikace mělo být umožněno procházení adresářové struktury projektu jak na straně lokální (pracovní) kopie projektu, tak na straně repositáře (serveru). Vhodným způsobem pak mělo umožnit vytvoření a správu více nezávislých projektů. Aplikace také měla poskytovat automatické sledování změn v projektovém stromu, či jednotlivých souborech.

Další podstatnou částí měl být modul pro porovnávání jednotlivých souborů, či jejich verzí a zobrazení diferencí mezi nimi. Tento modul měl pracovat samostatně, nezávisle na aplikaci, s možností rozšíření o specifické zásuvné moduly pro jednotlivé typy souborů.

5.2 Implementační prostředí

Implementace klientské aplikace a zároveň celého systému je vybudována na platformě .NET 2.0 pomocí programovacího jazyka C#. Toto rozhodnutí bylo učiněno hned z několika důvodů. Aplikace by měla mít silné grafické prostředí, což .NET platforma s využitím technologie WinForms poměrně jednoduše umožňuje. Veškerá síťová komunikace (viz kapitola 4) je pak díky vlastnostem platformy realizována pouze jako XML serializace .NET objektů. To je velmi dobrým předpokladem pro usnadnění vývoje a jednoduchou rozšiřitelnost (i s ohledem na určitou nezávislost vývoje obou částí systému). Dalšími výhodami jsou poměrně dobrá podpora standardních komunikačních protokolů a šifrovacích či hashovacích algoritmů.

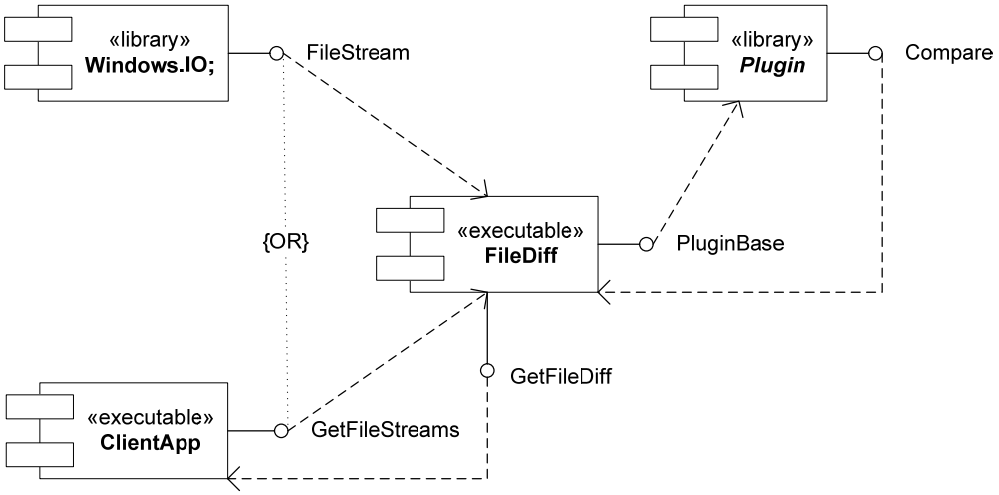
Jako implementační prostředí bylo zvoleno Microsoft Visual Studio 2005 s využitím verzovacího nástroje Microsoft Visual SourceSafe.

5.3 Návrh klientské části systému

Návrh klientské části systému, stejně jako celého systému, vznikal do značné míry na základě vlastností použité platformy a technologií. Snaží se proto využít všechny jejich výše zmiňované výhody. Jednotlivé části systému (aplikace i zásuvné moduly) jsou navrhovány jako samostatné assembly. Vnitřní struktury pak vycházejí z vlastností programovacího jazyka C# a technologie WinForms.

5.3.1 Architektura klientské části systému

Ze základních požadavků na klientskou aplikaci plyne, že nástroj pro porovnání a zobrazení diferencí mezi soubory bude vyvíjen jako samostatná aplikace (dále pod názvem FileDiff). Tento nástroj však musí být dostupný z hlavní aplikace (dále pod názvem ClientApp), a proto bude dále uvažován jako samostatný externí modul (viz obr. 14). Podrobnosti jednotlivých komponent jsou uvedeny v následujících odstavcích.



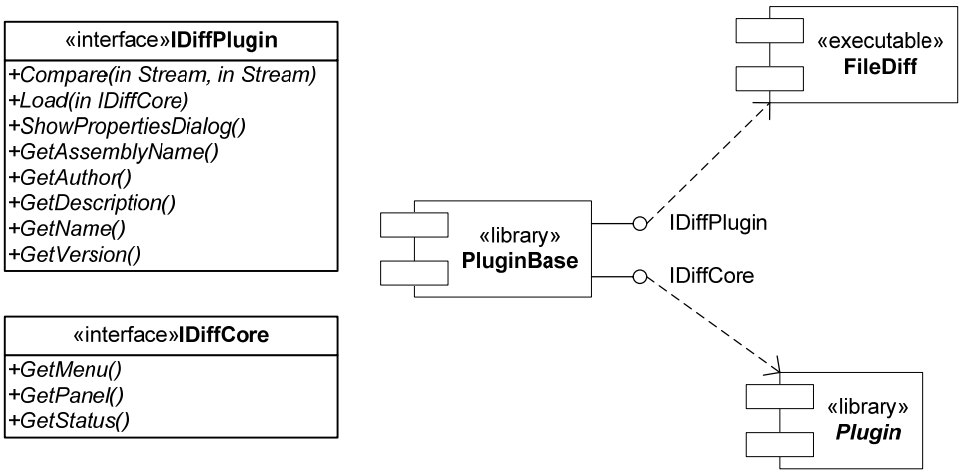
Obrázek 14 - Architektura klientské části - model komponent

5.3.2 Architektura modulu FileDiff

Samotný modul pro zobrazení diferencí slouží pouze jako rozhraní pro uživatele, externí aplikace a porovnávací moduly (viz obr 15). Jeho úkolem je poskytnutí uživatelského rozhraní a datových proudů (souborů) pro jednotlivé zásuvné moduly a zpřístupnění správy těchto modulů uživateli.

Aplikace FileDiff je tedy vlastníkem delegátů pro získání souborových proudů. Je schopna rozlišit, zda se jedná o proudy z fyzických disků či ze serveru, avšak pouze z důvodů rozhraní externích aplikací. Dále je vlastníkem zásuvných modulů, kterým poskytuje rozhraní *IDiffCore*, čímž daným modulům zpřístupňuje své komponenty uživatelského rozhraní, a to podnabídku hlavního aplikačního menu, stavový řádek a kontejner pro umístění komponent pro zobrazení výsledků. Zda

bude k zobrazení diferencí použito standardních komponent ze jmenného prostoru Windows.Forms či vlastních komponent, je již plně v kompetenci konkrétního modulu. Pomocí rozhraní *IDiffPlugin* naopak zásuvné moduly poskytují operace aplikaci. Tyto operace zahrnují inicializaci (*Load*), samotné porovnání (*Compare*) a případně vyvolání dialogu pro správu specifických nastavení daného modulu (*ShowPopertiesDialog*). Další operace pak slouží k identifikaci daných modulů a jejich vlastností.

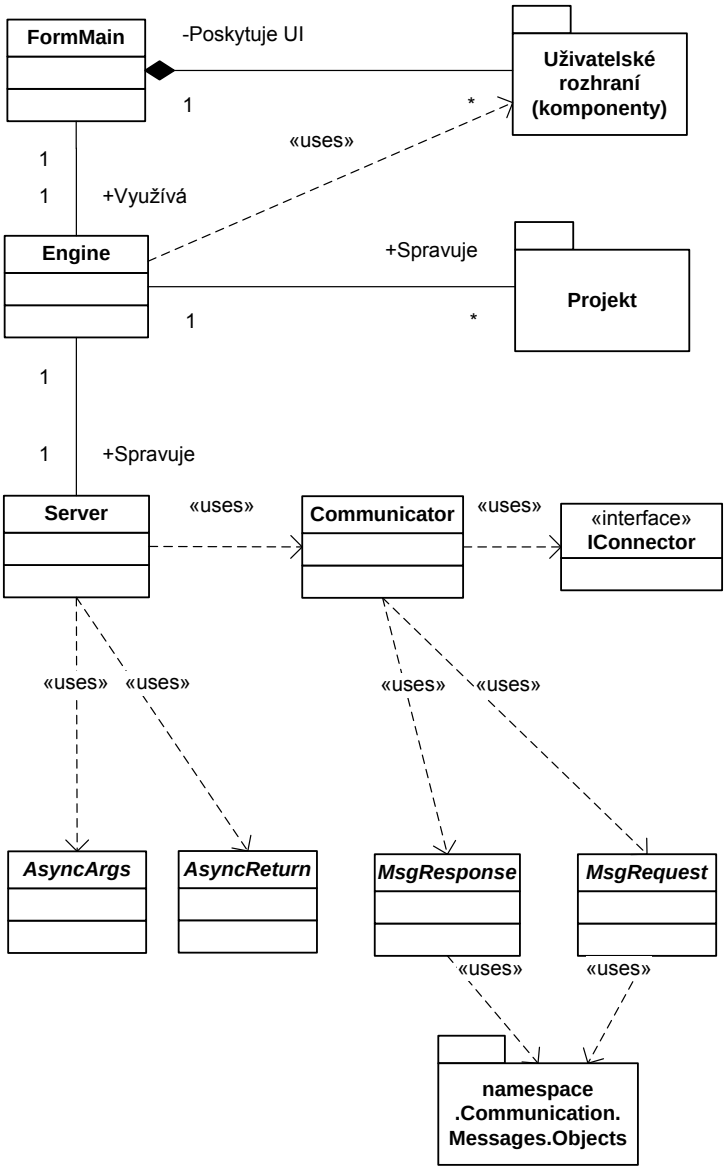


Obrázek 15- Architektura modulu FileDiff - model komponent

5.3.3 Architektura klientské aplikace

Klientská aplikace by neměla žádným způsobem zasahovat do souborového systému jednotlivých projektů. Proto si veškeré informace o lokálních obrazech projektů musí získat vždy pouze z adresářové struktury. Aplikace tedy neslouží jako správce nějakého lokálního projektu, ale jako správce a prohlížeč projektů a větví projektů na straně centrálního úložiště (serveru) a poskytuje možnosti synchronizace a porovnání s lokální adresářovou strukturou.

Při vytváření architektury byla uvažována největší možná abstrakce operací na jednotlivých úrovních komunikace a obecně práce s projekty. Takto abstrahované části aplikace byly vytvořeny z důvodů lehké udržovatelnosti a případné rozšiřitelnosti. Ačkoli v tomto návrhu je počítáno pouze s připojením k jednomu serveru, zároveň je pamatováno i na možnost současné komunikace s více servery. Více na základě obr. 16 v následujícím popisu.



Obrázek 16 - ClientApp - diagram tříd

Třída FormMain

Aplikace je reprezentována hlavním formulářem (*FormMain*), který pomocí svých komponent uživateli poskytuje ostatní prvky uživatelského rozhraní. Tento formulář je také vlastníkem instance třídy *Engine*, která je centrálním správcem projektů a slouží jako poskytovatel všech operací nad nimi, a to včetně komunikace se serverem. Typickým příkladem je vyvolání nějaké akce nad projektem z kontextového menu, což způsobí pouze zavolání odpovídající metody třídy *Engine* s danými parametry.

Třídy Engine a Server

Instance třídy *Engine* může spravovat libovolné množství projektů, jejichž struktura je popsána dále. Vlastní také instance třídy *Server*, který představuje nejvyšší stupeň abstrakce komunikace se serverem. Třída *Server* je navržena tak, aby umožňovala synchronní i asynchronní komunikaci se

serverem (vzhledem k výkonnému vláknu aplikace). Implementováno bude v této práci pouze rozhraní asynchronní, protože synchronní nemá žádný zvláštní význam, avšak návrh jej nevylučuje. Asynchronní požadavky využívají .NET třídy *BackgroundWorker* ze jmenného prostoru *System.ComponentModel* a data si předávají pomocí specializovaných tříd zděděných z abstraktních tříd *AsyncArgs* a *AsyncReturn*. K vlastní komunikaci se serverem je poté využito třídy *Communicator*, která využívá dalších abstraktních tříd a rozhraní (*IConnector, MsgRequest* a *MsgResponse*).

Třída Communicator a komunikační třídy MsgRequest a MsgResponse

Princip činnosti třídy *Communicator* a souvisejících tříd a rozhraní je následující. Nejprve se kdekoli během činnosti aplikace (typicky při nějaké události uživatelského rozhraní) vytvoří objekt s argumenty konkrétního požadavku. Ten je reprezentován pomocí konkrétní podoby třídy *AsyncArgs*. Instance této třída je předána pomocí metody *AsyncReturn Server.AsyncServerRequest (BackgroundWorker bw, object args)* společně s instancí třídy *BackgroundWorker* (ta je potřeba pro možnost předávání mezistavových informací) objektu *Server*. Toto předání je implementováno jako zavolání delegáta dané metody v již nově vytvořeném vlákně (viz zdrojový kód 2).

```
server.AsyncWorker = new BackgroundWorker();
server.AsyncWorker.WorkerReportsProgress = true;
server.AsyncWorker.WorkerSupportsCancellation = true;
server.AsyncWorker.DoWork += new DoWorkEventHandler(AsyncDoWork);
server.AsyncWorker.RunWorkerCompleted += new RunWorkerCompletedEventHandler(AsyncCompleted);
server.AsyncWorker.ProgressChanged += new ProgressChangedEventHandler(AsyncProgress);
server.AsyncWorker.RunWorkerAsync(new ArgsLogin(dlg.Login, dlg.Password));

void AsyncDoWork(object sender, DoWorkEventArgs e)
{
   BackgroundWorker bw = sender as BackgroundWorker;
   e.Result = server.AsyncServerRequest(bw, e.Argument);
}
```

Zdrojový kód 2 - Vytvoření asynchronního dotazu

Tato metoda, na základě výše zmíněné konkrétní podoby třídy *AsyncArgs,* zavolá odpovídající metodu komunikace, ve které je vytvořen taktéž odpovídající objekt typu *MsgRequest*. Tento objekt je po naplnění požadovanými daty pomocí statické metody *static MsgResponse QuerySend (IRepositoryConnector connector, MsgRequest req)* použit ke komunikaci se serverem. Po obdržení odpovědi ve formě *MsgResponse* je na základě obdržených dat vytvořena odpovídající instance třídy *AsyncReturn* a ta je navrácena jako výsledek operace třídy *BackgroundWorker* v původním vlákně.

```
public AsyncReturn AsyncServerRequest(BackgroundWorker bw, object args)
{
   if (Communicator == null)
      return new ReturnError(ConnectorException.GetExceptionString(...));
   try
   {
      if (args.GetType() == typeof(ArgsLogin))
          return LoginRequest(args);
      ...
   }
   catch (ConnectorException conEx)
   {...}
   catch
   {...}
}

private AsyncReturn LoginRequest(object args)
{
   ArgsLogin loginArgs = (args as ArgsLogin);
   MsgLoginReq req = new MsgLoginReq();
   req.Login = loginArgs.Login;
   req.Password = loginArgs.Password;
   MsgLoginRes res = Communicator.QuerySend(Communicator.Connector, req) as MsgLoginRes;
   return new ReturnLogin(res.LoginStatus);
}
```

Zdrojový kód 3 - Příklad rozpoznání dotazu a jeho vykonání

Objektu *Communicator* je poskytnuto spojení pomocí konektoru splňující rozhraní *IConnector*. Komunikační třídy *MsgRequest* a *MsgResponse* se přímo využívají k XML serializaci a XML deserializaci, a tak jsou přímým nositelem přenášené informace. K sjednocení komunikace na straně serveru a klienta je používán jmenný prostor *Communication.Messages.Objects*, ve kterém se vyskytují konkrétní třídy a kolekce tříd používané k reprezentaci vnitřních dat systému.

Abstrakce asynchronního rozhraní

K zjednodušení implementace jednotlivých požadavků na server a zároveň k zatraktivnění uživatelského rozhraní je implementován speciální komunikační formulář *FormAsyncWork*, který dále abstrahuje asynchronní komunikaci. Je využíván zejména tam, kde je potřeba pozdržet další činnost uživatele, zároveň však umožnit reakce aplikace na systémové události a v neposlední řadě informovat uživatele o průběžném stavu vyvolané činnosti. Rozhraní s formulářem je pak velice jednoduché. Formuláři je v konstruktoru předán již naplněný objekt třídy *AsyncArgs* a po dokončení metody *DialogResult ShowDialog()* je v případě správné návratové hodnoty ve vlastnosti *Return* dostupná odpovídající instance třídy *AsyncReturn*. Demonstrace použití je v následujícím fragmentu kódu.

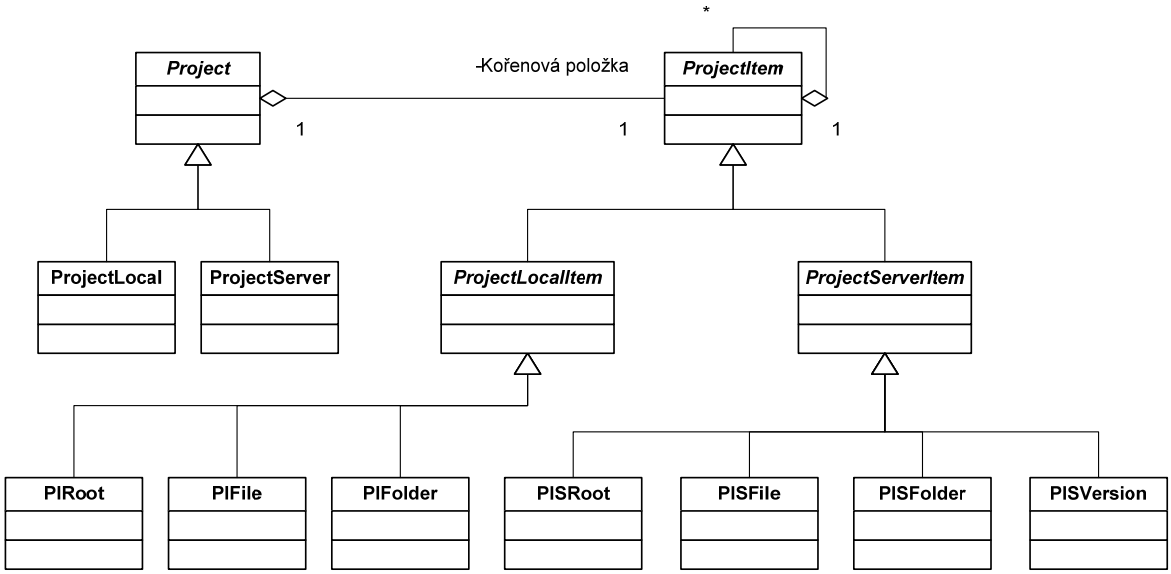
```
FormAsyncWork form = new FormAsyncWork(new ArgsLogin(dlg.Login, dlg.Password),
                                       Engine.Server, "Připojení", true);

if (form.ShowDialog() != DialogResult.Cancel)
{
   AsyncReturn ret = form.Return;
   if (ret is ReturnError)
   {
      statusStrip.Items[0].Text = "Přihlášení zamítnuto: " + (ret as ReturnError).Result;
   }
   else
   {
      ReturnLogin retLogin = (ret as ReturnLogin);
      if (retLogin.IsLoged())
      {
         statusStrip.Items[0].Text = "Přihlášení proběhlo úspěšně";
      }
      else
      {
         statusStrip.Items[0].Text = "Přihlášení zamítnuto: " + ret.Result;
      }
   }
}
```

Zdrojový kód 4 - Ukázka použití asynchronního formuláře

Reprezentace projektu

Každý projekt je v systému reprezentován jedním objektem, který popisuje veškeré jeho vlastnosti a vlastní výchozí bod adresářové struktury (tedy jednu položku struktury). Tato položka pak odkazuje na další položky v adresářové struktuře podobně, jako je to tomu v souborovém sytému. Jednotlivé projekty se dělí na projekty lokální (*ProjectLocal*) a serverové (*ProjectServer*), a to z důvodu různých



Obrázek 17 - Reprezentace projektu v systému - diagram tříd

funkčních vlastností. Příkladem může být načítání adresářové struktury (lokální projekt je v interakci přímo s adresářovou strukturou operačního systému, naopak vzdálený projekt vždy komunikuje se serverem).

Rozdělení na jednotlivé typy projektových položek (*ProjectItem*) má důvod analogický. Výrazně zjednodušuje práci se strukturou jako celkem a umožňuje polymorfně přetěžovat operace

nad těmito jednotkami. Dalším důvodem pro podrobné rozdělení položek je zjednodušení způsobu vizualizace struktury pomocí třídy *ProjectBrowser* (více v kapitole o implementaci UI). Položka projektu tedy obsahuje veškeré relevantní informace a sdružuje kolekci položek podřazených. Každá podřazená položka (mimo instancí třídy *PIRoot* a *PISRoot*) odkazuje i na svého vlastníka, tedy položku nadřazenou. Tím je vytvořena obecná stromová struktura, která zjednodušuje průchody jak směrem k listům stromu, tak ke kořenu. V jedné úrovni také drží rovnoprávné položky a umožňuje je například řadit pomocí specifického komparátoru (implementovaného jako .NET rozhraní *System.Collection.IComparer*). Diagram tříd reprezentujících projekt je zobrazen na obr. 17.

5.3.4 Návrh uživatelského rozhraní

Klientská aplikace má být dle požadavků co nejjednodušší a přehledná na ovládání. Veškerá interakce se systémem probíhá pomocí grafických komponent. Z tohoto důvodu je uživatelské rozhraní navrženo tak, aby vytvářelo dojem práce s klasickým souborovým systémem, obohaceným o správu verzí. V základní navrhované verzi aplikace podporuje současnou práci pouze s jedním projektem, nicméně do budoucna je počítáno s rozšířením uživatelského rozhraní na práci s více projekty zároveň.

Výchozím bodem aplikace je pohled na adresářové struktury jednotlivých částí projektu. Výběr lokální části je založen na zvolení konkrétního adresáře na disku. Výběr serverového projektu pak vybírá v samostatném formuláři z nabídky dostupných projektů, z jejich vývojových větví, případně verzí (stavů větve). K zobrazení struktur je použito komponenty *ProjectBrowser* vycházející z volně dostupné komponenty *Aga.TreeViewAdvance* [16]. Jednotlivá nastavení a správa projektů pak probíhá v příslušných formulářích a dialozích. Základní formulář také obsahuje typické prvky Windows aplikací, jako je hlavní nabídka, či stavový řádek.

Uživatelské rozhraní modulu FileDiff vychází přímo ze specifikace architektury. Samotná aplikace FileDiff obsahuje pouze hlavní nabídku, stavový řádek a panel pro umístění výsledků. Jak bude vypadat zobrazení výsledků a jaké rozšiřující komponenty budou jednotlivé zásuvné moduly používat je již zcela v jejich kompetenci.

Zásuvné moduly jsou do aplikace zaváděny dvěma způsoby. Prvním způsobem je zavedení assembly přímo, a to ve formě souboru s příponou „dll“. Takto zaváděný modul musí obsahovat všechny popisné informace (verze, název, atd.) přes definované rozhraní *IDiffPlugin* a název třídy splňující toto rozhraní musí nést stejný název jako assembly. Druhým způsobem je zavedení pomocí konfiguračního XML souboru, který popisuje všechny potřebné informace. Toto řešení se doporučuje pouze v případě, kdy jméno hlavní třídy zásuvného modulu není totožné s názvem assembly. V konfiguračním souboru je možnost toto jméno nastavit libovolné. Příklad konfiguračního XML souboru je uveden ve zdrojovém kódu 5.

```
<?xml version="1.0"?>
<PluginSettings xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
                xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <arrSuffix/>
  <AssemblyName>DiffPluginXML.DiffPluginXML</AssemblyName>
  <Description>Plugin pro porovnání souborů s XML obsahem.</Description>
  <Name>XML Plugin</Name>
  <Author>Ondřej Bým</Author>
  <Location></Location>
  <LocationXml></LocationXml>
  <Version>0.1</Version>
</PluginSettings>
```

Zdrojový kód 5 - Příklad konfiguračního souboru zásuvného modulu

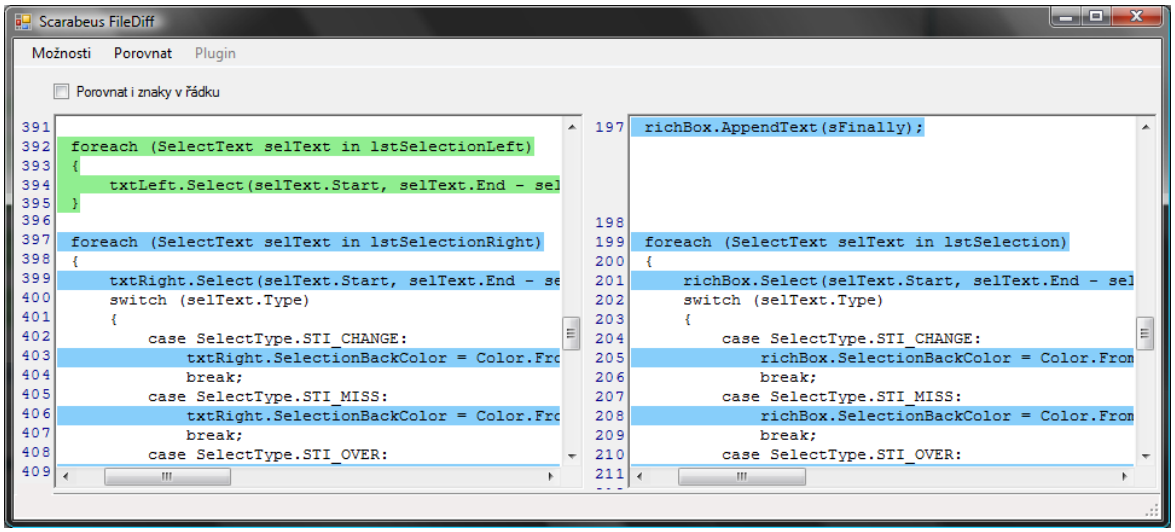
5.4 Principy porovnání souborů a způsob vizualizace změn

Přestože každý ze zásuvných modulů určených pro porovnání souborů může pracovat na jiném principu, či s využitím jinak optimalizovaných algoritmů, jsou v rámci této práce užívány vždy algoritmy založené na problému LCS. Samotné využití LCS se však u jednotlivých modulů liší. V základní verzi aplikace budou přístupné 3 moduly, dva obecné a jeden specializovaný. První z obecných bude uzpůsoben k porovnání standardních textových souborů (*DiffPluginPlain*), druhý k porovnání binárních souborů (*DiffPluginBinary*). Poslední bude představovat specializaci na konkrétní vnitřní strukturu souboru a bude jím modul k porovnání souborů XML (*DiffPluginXML*).

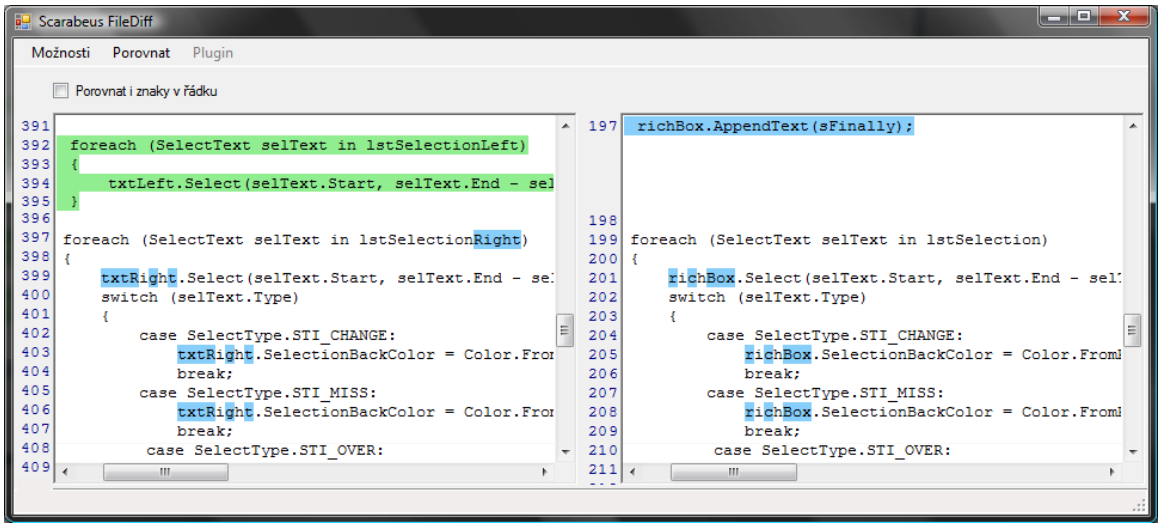
5.4.1 DiffPluginPlain

Zásuvný modul DiffPluginPlain je navržen tak, aby porovnával rozdíly orientované na řádky souboru a následně na složení jednotlivých řádků. Každý řádek souboru představuje jeden symbol v LCS a je porovnáván jako celek. Pokud je řádek vyhodnocen jako rozdílný, resp. editovaný, je provedeno LCS volitelně i na všechny jeho znaky.

Výsledek je zobrazen ve dvou vedle sebe umístěných oknech. Pro snazší orientaci ve výsledku porovnání jsou jednotlivá okna zarovnána vždy podle odpovídajících řádků. Tzn., pokud nějaký řádek chybí, je nahrazen v druhém okně řádkem prázdným. Chybějící řádky jsou pro názornost barevně odlišeny (viz obr. 18). Stejně tak jsou dle zvoleného způsobu porovnání barevně odlišeny jednotlivé změny v řádcích editovaných (viz obr. 19).

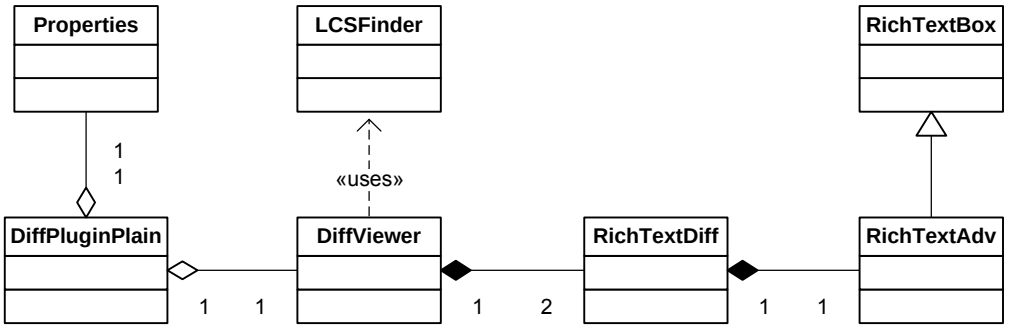


Obrázek 18 - Výsledek porovnání s rozlišením na řádek (DiffPluginPlain)



Obrázek 19 - Výsledek porovnání s rozlišením na znak řádku (DiffPluginPlain)

K zobrazení rozdílů mezi soubory je využito volně dostupné komponenty *RichTextAdv* [22], která je rozšířením standardní komponenty *System.Windows.Forms.RichTextBox*. Tato třída slouží pouze k zobrazení textu, další prvky rozhraní, jako zobrazení řádků apod., obstarává třída *RichTextDiff*. Dvě instance komponenty *RichTextDiff* jsou pak obsaženy v kontejnerech komponenty *DiffViewer*, která je zásuvným modulem předávána hlavní aplikaci. Diagram tříd je znázorněn na obr. 20.

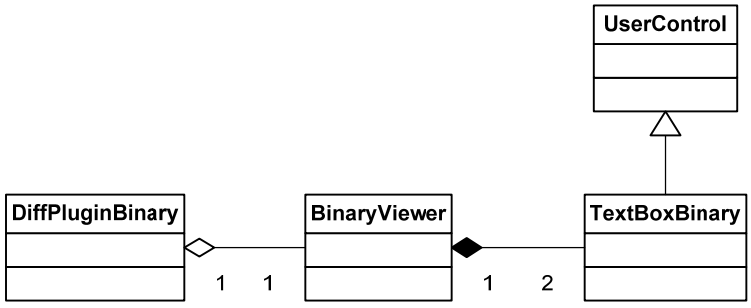


Obrázek 20 - Diagram tříd zásuvného modulu DiffPluginPlain

5.4.2 DiffPluginBinary

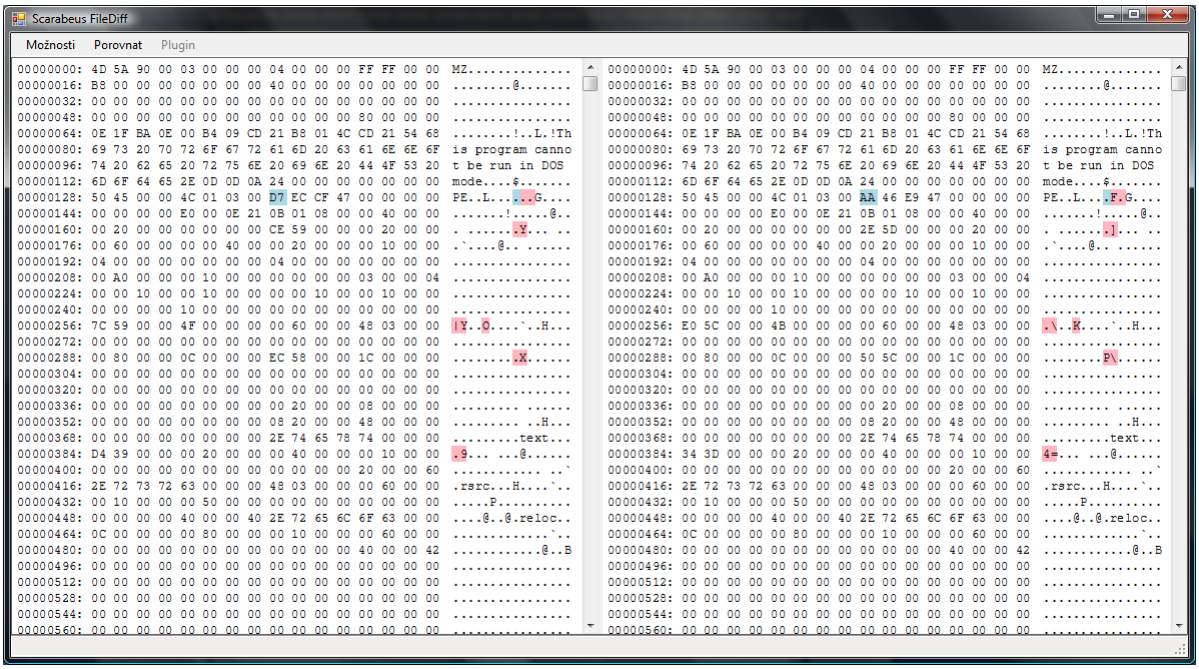
Tento zásuvný modul žádným způsobem soubor nedělí a považuje každý byte jako symbol. Výsledky jsou zobrazeny ve dvou oknech, podobně jako v předchozím případě. Tato okna jsou však rozdělena na další dvě podokna, kde je pro větší názornost uvedena reprezentace jak hexadecimální, tak textová. Na každém řádku je vždy uveden stejný počet bytů, tak aby byla zachována vizuální podoba s editory binárních souborů. Tento zásuvný modul neporovnává soubory pomocí standardní metody LCS, ale porovnává pouze hodnoty bytů na příslušné pozici v souboru.

Pro snazší orientaci je po kliknutí na konkrétní znak zvýrazněna hexadecimální i ASCII podoba příslušného bytu v obou porovnávaných souborech. Příklad zobrazení výsledku porovnání dvou binárních souborů je znázorněn na obr. 22.



Obrázek 21 - Diagram tříd zásuvného modulu DiffPluginBinary

Uživatelské rozhraní je implementováno obdobně jako v případě předchozího zásuvného modulu. K zobrazení jednotlivých souborů byla vytvořena komponenta *TextBoxBinary* zajišťující zobrazení souboru v textové i hexadecimální podobě. Dvě instance této třídy vlastní komponenta *BinaryViewer*, která je předávána do hlavní aplikace (viz obr. 21).

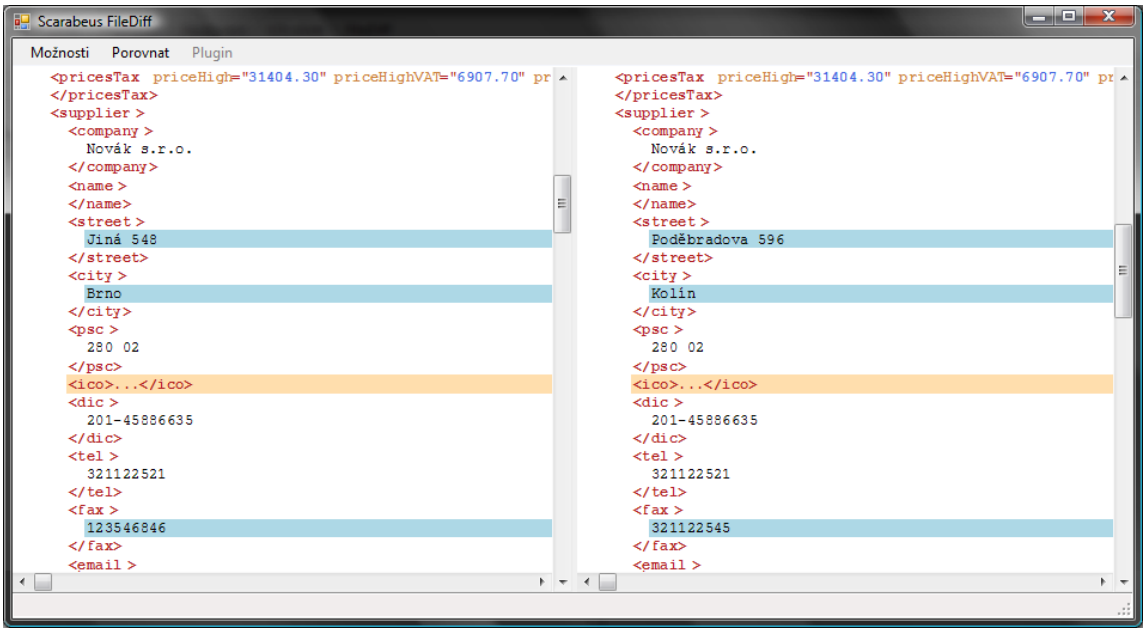


Obrázek 22 - Porovnání dvou hexadecimálních souborů pomocí zásuvného modulu DiffPluginBinary

5.4.3 DiffPluginXML

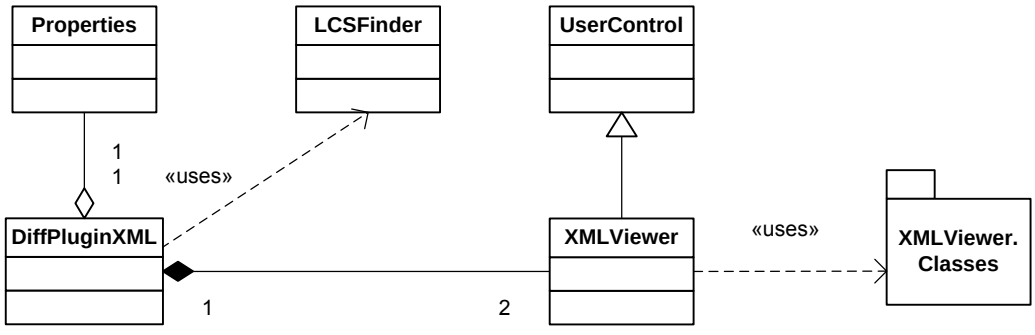
Zásuvný modul DiffPluginXML je příkladem specializovaného porovnávání. Na soubor se nedívá standardním způsobem, ale jako na datovou strukturu. V tomto případě se jedná o DOM model a reprezentaci stromové struktury. Během porovnání soubor dělí na jednotlivé XML elementy a ty porovnává, jako by šlo o symboly. Jelikož se jedná o stromovou strukturu, prochází se daný strom do hloubky za použití metody preorder. Za symbol do LCS je považován také čistý text umístěný vně tagů a tím vytváří listy stromu. Pokud mezi otevíracím a uzavíracím tagem elementu není žádný čistý text, je list považován za prázdný a v porovnávání je uvažován také.

V rámci této práce je navržen algoritmus, který porovnává jednotlivé elementy pomocí booleovské logiky, a proto není žádným způsobem reflektována váha rozdílu mezi jednotlivými elementy. Algoritmus, který by přikládal váhu jednotlivým rozdílům (u stejných elementů na stejné úrovni), by mohl mít nastavitelné váhy pro jednotlivé typy diferencí. Takovými diferencemi jsou například rozdílné hodnoty atributů, rozdílné atributy, či rozdíly vnitřních elementů na různých úrovních zanoření. Pokud bychom vytvořili takovýto algoritmus, je třeba počítat s vysokou časovou složitostí, protože při situaci mnoha stejných elementů s vysokou úrovní zanoření by docházelo k mnoha porovnáním celých stromových struktur. Při použití s LCS by pak složitost ještě narůstala o režii zpracování výsledků porovnání, protože klasické použití LCS počítá pouze s výsledky ve dvouhodnotové logice.



Obrázek 23- DiffPluginXML - Ukázka porovnání

Barevné odlišení výsledků je rozdílné od předchozích dvou zásuvných modulů (viz obr. 23). Důraz se klade na přehledné zobrazení stromu a zarovnání dle hloubky zanoření. V souborech jsou také ignorovány tzv. bílé znaky a je tedy zobrazen jen čistý DOM model dle specifikace XML (více např. v [21]). K zobrazení výsledků byla vytvořena komponenta *XMLViewer*, která převádí DOM model do grafické podoby s orientací na řádek. Stromová struktura je naznačena odsazením textu daného elementu. K snazší orientaci ve výsledcích jsou jednotlivé chybějící (změněné) elementy barevně označeny. Výsledky obou oken jsou obdobně jako v zásuvném modulu DiffPluginPlain zarovnány na odpovídající řádky. V případě potřeby lze dvojklikem na otevírací tag elementu tento element sbalit do podoby jednořádkového zobrazení, čímž je dále naznačena stromová struktura a je usnadněna orientace v delších souborech. Diagram tříd je zobrazen na obr. 24.



Obrázek 24 - Diagram tříd zásuvného modulu DiffPluginXML

6 Testování

V rámci ověření funkčnosti vytvořeného systému (dále jako testovaný systém) bylo provedeno několik testů. Testy se týkaly jak samotného ověření funkčnosti, tak porovnání vlastností a rychlosti práce s existujícím volně dostupným verzovacím systémem. Testy byly provedeny společně pro tuto práci a práci zabývající se serverovou částí [23].

6.1 Metoda a podmínky testování

Testování bylo prováděno na 2, resp. 4 nezávislých pracovních stanicích v následujících konfiguracích (viz tabulka 1). Propojení pracovních stanic bylo provedeno pomocí jediného aktivního prvku typu přepínač ve 100Mb/s síti.

	Sestava A	**Sestava B**
Procesor	Intel Pentium M 1,6Ghz	Intel Pentium D 2,8Ghz
Operační paměť	1GB	2x 1GB
Pevný disk	SATA 30GB volných fragmentovaných	SATA II 120GB volných nefragmentovaných
Operační systém	Windows Vista Business SP1	Windows Vista Business SP1

	Sestava C	**Sestava D**
Procesor	Intel Core Duo T2350 D 1,86Ghz	AMD Athlon 1,8Ghz
Operační paměť	1GB	2x 256MB
Pevný disk	SATA 40GB volných fragmentovaných	SATA 40GB volných fragmentovaných
Operační systém	Windows Vista Home Premium SP1	Windows XP SP2

Tabulka 1 - Konfigurace stanic využitých při testování

6.1.1 Metoda testů

Jako zástupce volně dostupného verzovacího systému byl vybrán systém SubVersion. Repositář byl instalován ve verzi 1.4.6, klientská část byla zastoupena grafickou nadstavbou TortoiseSVN verze 1.0.8 [24]. Měření bylo prováděno zvlášť na serverové i klientské straně pomocí Nástroje systému Windows Sledování spolehlivosti a výkonu, kde lze sledovat s přesností na desetiny vteřiny procesorovou činnost, přístup k disku a síti jednotlivých procesů. Přesnost byla ověřena pomocí porovnání naměřených hodnot tímto nástrojem a hodnot zaznamenaných vnitřní komponentou klientské části systému určenou pro logování událostí v systému.

Celkově byly provedeny dva typy testů. Prvním byl test přímé propustnosti, kdy komunikace probíhala mezi jednou instancí serveru a jednou instancí klienta. Druhý test se týkal měření rychlosti při distribuovaném rozložení systému.

Testovány byly vždy dva případy použití, a to načtení a přenos celého projektu směrem do repositáře (*check-in*) a vytažení celého projektového stromu na lokální disk (*check-out*).

6.2 Výsledky testů

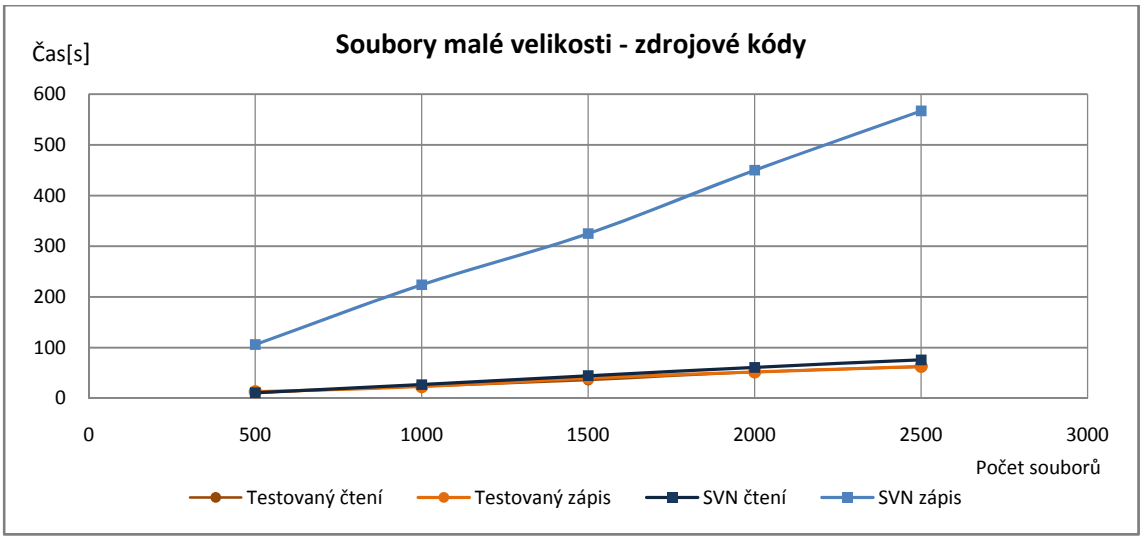
6.2.1 Test přímé propustnosti

Instalace hlavního repositáře byla provedena na stanici **A** včetně souborového serveru. Klientská část byla instalována na stanici **B**.

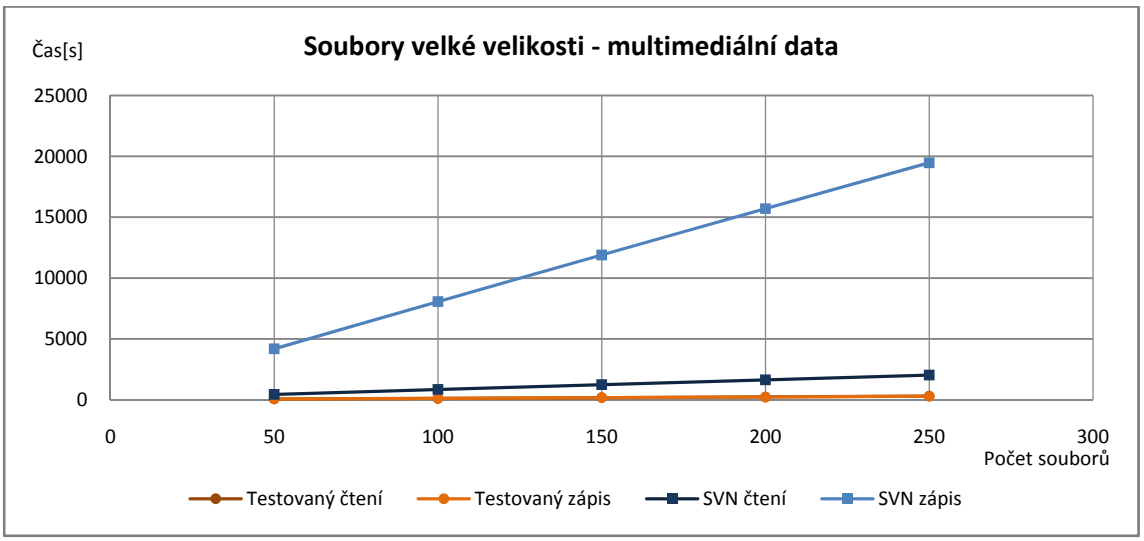
V uvedených časech (viz tabulka 2) jsou zahrnuty výsledky všech fází komunikace. Mezi fáze komunikace se počítá zahájení (vytvoření kontrolních součtů apod.), přenos a ukončení (načtení struktur, obnovení souborového systému apod.). Grafy naměřených hodnot jsou uvedeny v obr. 25 a 26.

	Testované struktury		Check-out		Check-in	
	Počet souborů	Průměrná velikost [kB]	Testovaný [s]	SVN [s]	Testovaný [s]	SVN [s]
A1	500	3,6	11	11	13	106
A2	1000	3,6	24	27	23	224
A3	1500	3,6	36	45	40	325
A4	2000	3,6	52	61	52	450
A5	2500	3,6	62	76	63	567
B1	50	9010	59	435	58	4177
B2	100	9010	104	838	98	8061
B3	150	9010	179	1235	158	11892
B4	200	9010	218	1632	209	15703
B5	250	9010	294	2023	278	19477

Tabulka 2 - Tabulka naměřených hodnot - přímé spojení



Obrázek 25 - Graf naměřených hodnot - malé soubory - přímé spojení



Obrázek 26 - Graf naměřených hodnot - velké soubory - přímé spojení

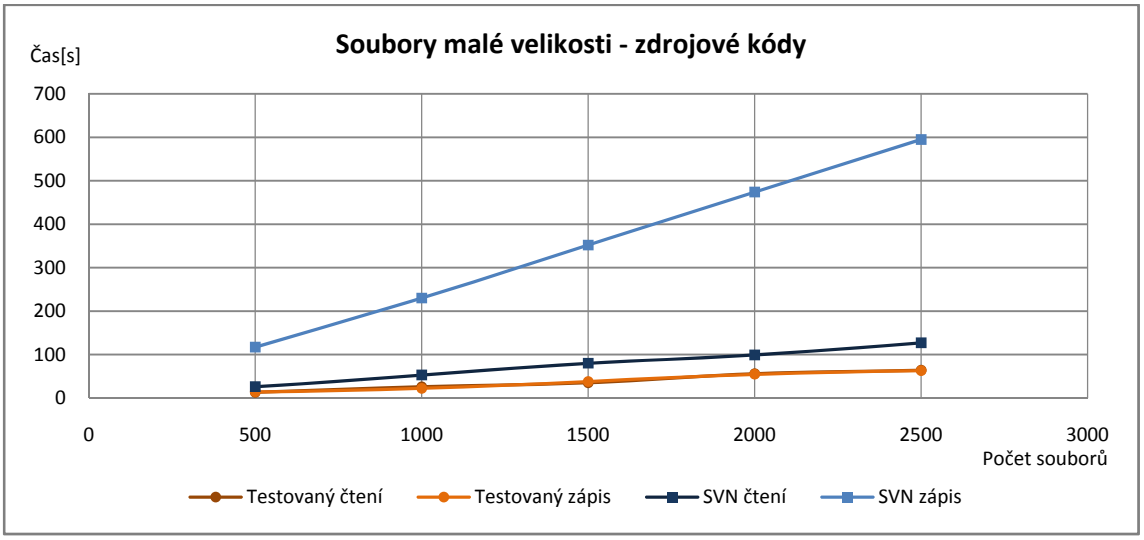
6.2.2 Test distribuovaného rozložení

Klientské části systému byly umístěny na stanicích **B** a **C**. Instalace repositáře a prvního souborového serveru byla provedena na stanici **A,** druhý souborový server byl umístěn na stanici **D**. V případě použití SVN byla instalace repositáře provedena na stanici **A**. K simulaci distributivity byly spuštěny dvě instance SVN repositáře (rozlišené komunikačním portem) a k nim příslušné datové adresáře byly zřízeny na stanicích **A** a **D**.

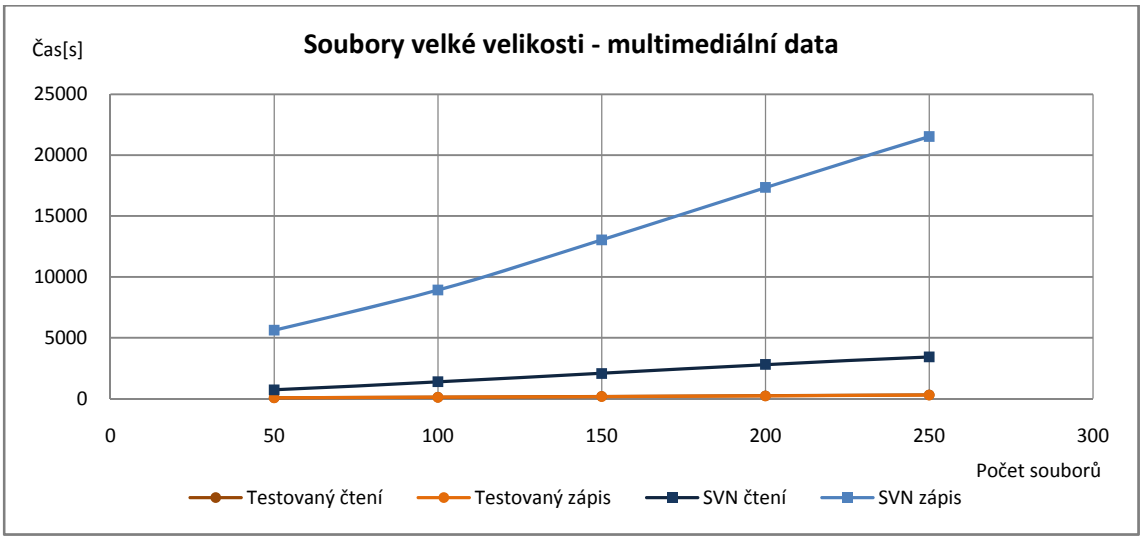
Nahrání, resp. vytažení projektů bylo spuštěno vždy současně na obou klientských stanicích. V uvedených časech (viz Tabulka 3) jsou zahrnuty výsledky všech fází komunikace, podobně jako v předchozích případech. Jelikož bylo prováděno měření na dvou stanicích zároveň a rozdíl mezi dokončením ani v jednom případě nedosáhl 5% celkového času, je uveden pouze vyšší z údajů. Grafy výsledných hodnot jsou uvedeny na obr. 27 a 28.

	Testované struktury		Check-out		Check-in	
	Počet souborů	Průměrná velikost [kB]	Testovaný [s]	SVN [s]	Testovaný [s]	SVN [s]
C1	500	3,6	13	26	14	117
C2	1000	3,6	26	53	22	230
C3	1500	3,6	35	80	38	352
C4	2000	3,6	56	99	55	474
C5	2500	3,6	64	127	63	595
D1	50	9010	58	715	54	5613
D2	100	9010	105	1382	100	8915
D3	150	9010	179	2055	161	13035
D4	200	9010	220	2803	206	17344
D5	250	9010	299	3420	282	21533

Tabulka 3 - Tabulka naměřených hodnot - distribuovaná úložiště



Obrázek 27 - Graf naměřených hodnot - malé soubory - distribuovaná úložiště



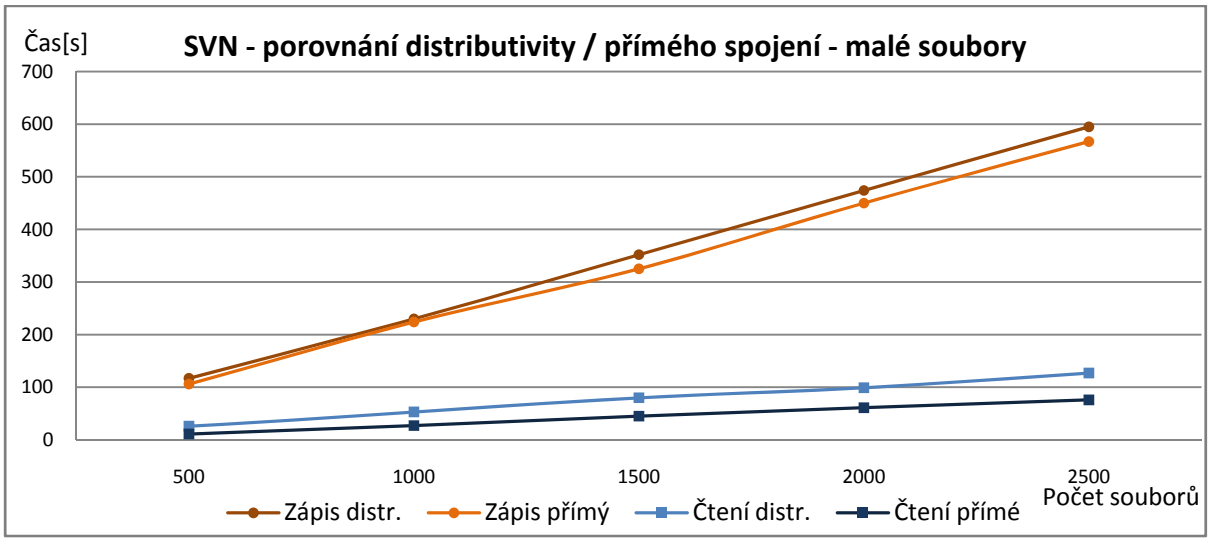
Obrázek 28 - Graf naměřených hodnot - velké soubory - distribuovaná úložiště

6.3 Zhodnocení testů

Výsledky testů dopadly dle očekávání. Všechny závislosti mají lineární charakter. Jediným poměrně překvapivým jevem je délka trvání nahrání projektu v systému SubVersion. Prodloužení doby nahrání je zřejmě způsobeno principem uchování informací o jednotlivých souborech v projektu. Systém SubVersion totiž vytváří v každém adresáři při nahrání vlastní kontrolní soubory, kde pomocí textových informací vytváří log změn v projektu (i na straně klienta). Počet těchto souborů pak několikanásobně přesahuje počet samotných projektových souborů. Vytvoření a režie obsluhy těchto souborů je zřejmě hlavní důvod propastného rozdílu v čase komunikace.

6.3.1 Výsledky testu distribuovaného rozložení systému

Pokud zhodnotíme časové závislosti u měření s přístupem k jednomu souborovému serveru (testy A1-A5 a B1-B5) a měření k dvěma souborovým serverům (C1-C5 a D1-D5), zjistíme, že v testovaném systému se doba operací prakticky nezměnila. V systému SVN je u malých i velkých souborů čas mezi distribuovaným a přímým spojením v případě čtení zhruba o 80% vyšší. Pokud ovšem přihlédneme k faktu, že prakticky stejné rozdíly jsou i u zápisu, který je několikanásobně delší, je pravděpodobně tato prodleva způsobena režií systému při kontrole jednotlivých souborů na straně serveru (viz obr. 29). Režie u testovaného systému je jen minimální čas na počátku transakce, proto nemá na celkový čas operace vliv.



Obrázek 29 - Graf časů distribuovaného a přímého spojení pro SVN - malé soubory

V těchto základních demonstračních testech sice vychází testovaný systém mnohem lépe, avšak je třeba přihlédnout k faktu, že systém SubVersion neuchovává informace o změnách pouze v jednom místě a nechová se transakčně jako testovaný systém. SubVersion tedy má mnohem lepší možnost zotavení při pádech systému či přerušení spojení apod.

7 Závěr

Cílem této práce bylo zpracování teoretického přehledu o existujících verzovacích systémech a principech verzování v těchto systémech uplatněných a na základě teoretického rozboru navrhnout verzovací systém pro obecné použití.

Práce tedy ve své první části popisuje jednotlivé způsoby verzování a principy funkčnosti existujících verzovacích systémů. Je zde kladen důraz zejména na potřeby uživatele, rozhraní systému s uživatelem a organizaci pracovních prostorů. Z těchto výsledků byly vybrány potřebné vlastnosti pro návrh a vytvoření vlastního systému. Vytvořený systém je navržen s důrazem na co největší obecnost použití. Není tedy přizpůsoben jen pro správu zdrojových kódů softwarových produktů, ale pro jakýkoliv obecný projekt založený na elektronických dokumentech.

Celková architektura je založena na modelu klient-server, kde server slouží jako hlavní repositář verzovacího systému a navenek působí jako jedno centrální úložiště. Vnitřně je však distribuována výpočetní část a datová úložiště. Klientská část systému pak vystupuje jako jednotné uživatelské rozhraní pro celý systém. V rámci této práce byla detailně navržena a implementována právě klientská část.

Návrh klientské části systému se snaží o co největší obecnost a nezávislost na způsobu použití. Také žádným způsobem nezasahuje do operačního systému nebo adresářových struktur definovaných projektů a je pouze jednoduchým způsobem napojena na konkrétní projektové adresáře. Jako součást klientské části systému byla navržena aplikace pro porovnání a vizualizaci rozdílů mezi jednotlivými soubory či verzemi souborů. V této aplikaci je implementováno rozhraní pro externí zásuvné moduly umožňující porovnání souborů různých typů.

V závěru práce byl vytvořený systém testován z hlediska splnění funkčních požadavků a zároveň byl porovnán se zástupcem existujícího verzovacího systému. Z testů vyplývá, že distribuované rozložení souborových serverů v architektuře systému zefektivňuje paralelní práci. Uchování veškerých informací o verzích souborů a repositáři v jednom centrálním úložišti a zároveň přenechání logiky verzování ve výkonné databázi ukazuje další výhodu navrženého řešení oproti porovnávanému systému. Průměrné rychlosti přenášených souborů pak klesají s velikostí, a to zejména z důvodu režie navázaní spojení v rámci zvoleného protokolu HTTP.

7.1 Stav vývoje a plánovaná rozšíření

V současné době je systém implementován a otestován z hlediska základní funkčnosti. Zbývá doplnění doprovodných funkcionalit, jako je například komplexní správa uživatelů apod. Jedním z prvních plánovaných rozšíření je také přehledné grafické zobrazení historie vývoje projektu na základě větvení projektu či jiných specifik. Další součástí systému by se měly stát statistické přehledy

nad prací jak z hlediska uživatelů systému, tak například četností použití jednotlivých souborů či celých projektů. Tato plánovaná rozšíření jsou z hlediska implementace díky navržené architektuře poměrně jednoduše dostupná.

Klientská aplikace je v současné době plně funkční v rámci navrženého řešení a zbývá vytvořit doplňující operace a funkčnosti uživatelského rozhraní. Dále je třeba vytvořit jednotné grafické prvky, jako jsou ikony apod.

Uvažovaným rozšířením celého systému je podpora dalších komunikačních protokolů, včetně zabezpečeného protokolu HTTPS. Rozšíření klientské aplikace počítá s vytvořením jednoduchého aplikačního rozhraní. Navržená aplikace je totiž samostatně funkční i bez grafických komponent a je schopna pracovat i na základě jednotlivých komunikačních příkazů s potřebnými parametry.

Literatura

[1] RENDER, H., CAMPBELL, R.: An object-oriented model of software configuration management. *Proceedings of the 3rd international Workshop on Software Configuration Management*. New York: ACM Press, 1991. s. 127-139. ISBN: 0-897914-429-5

[2] TICHY, W. et al.: Impact of software engineering research on the practice of software configuration management. ACM Trans. Softw. Eng. Methodol. 14. New York, ACM Press, 2005. s. 383-430, ISSN: 1049-331X

[3] CONRADI, R., WESTFECHTEL, B.: Version models for software configuration management. *ACM Computing Surveys*. New York: ACM Press 1998, s. 232-282, ISSN: 0360-0300

[4] BERGROTH, L., HAKONEN, H., RAITA, T.: A Survey of Longest Common Subsequence Algorithms. *Seventh International Symposium on String Processing Information Retrieval*. Washington: IEEE Computer Society, 2000. s. 39-49. ISBN: 0-7695-0746-8.

[5] WAGNER, R.A., FISCHER, M.J.: The String-to-String Correction Problem. *Journal of the ACM.* ACM Press: 1975. s. 168-173, ISSN: 0004-5411

[6] BAUER, M.: Paranoid penguin: rsync. *Linux Journal.* Seattle: Specialized Systems Consultants Inc., 2003. ISSN: 1075-3583

[7] LIU, W., CHEN, Y., CHEN, L., QIN, L.: A Fast Parallel Longest Common Subsequence Algorithm Based on Pruning Rules. *Proceedings of the First international Multi-Symposiums on Computer and Computational Sciences*. Washington: IEEE Computer Society, 2006. s. 27-34. ISBN:0-7695-2581-4-01

[8] STRMISKA, L.: Správa verzí. [Online] [Citace: 10. Srpen 2007]. Dostupný z WWW: <http://www.fi.muni.cz/~kas/p090/referaty/2006-podzim/st/xstrmisk-verze.html>.

[9] *Wikipedia:* Comparison of revision control software. [Online] 18. Srpen 2007. [Citace: 22. Srpen 2007].
Dostupný z WWW: <http://en.wikipedia.org/wiki/Comparison_of_revision_control_software>.

[10] *Microsoft Corporation*: Microsoft Visual SourceSafe 6.0 Standard Edition. *Středisko informací o produktech společnosti Microsoft.* [Online] Microsoft Corporation. [Citace: 21. Srpen 2007]. Dostupný z WWW: <http://www.microsoft.com/products/info/product.aspx?view=4&pcid=d53566f4-2274-408b-9f11-a3d1d21f85a6&crumb=catpage&catid=515c9859-958b-4433-b4f9-91f37258ca2f>.

[11] *Microsoft Corporation*: Visual Studio Team System. *Microsoft.com.* [Online] Microsoft Corporation. [Citace: 21. Srpen 2007].
Dostupný z WWW: <http://msdn2.microsoft.com/en-us/teamsystem/default.aspx>.

[12] *Araxis Ltd.:* Instant Overview of Folder Comparison and Synchronization. *Araxis.* [Online] 30. Srpen 2007. [Citace: 4. Říjen 2007] .
Dostupný z WWW: <http://www.araxis.com/merge/overview2.html>.

[13] *Zend Technologies Ltd.:* Zend Feature List. [Online] Zend Technologies Ltd. [Citace: 20. Srpen 2007], Dostupný z WWW: <http://www.zend.com/de/products/zend_studio/feature_list>

[14] *WinCVS:* CvsGui. *WinCVS.* [Online] [Citace: 19. Srpen 2007].
Dosupný z WWW: <http://www.wincvs.org/shots.html>.

[15] *EPocalipse Software:* Product. [Online] EPocalipse Software. [Citace: 20. Srpen 2007].
Dostupný z WWW: <http://www.epocalipse.com/vcxscr.htm>.

[16] GLIZNETSOV, A.: Advanced TreeView for .NET. *The Code Project.* [Online] [Citace: 10. Srpen 2007]. Dostupný z WWW: <http://www.codeproject.com/cs/miscctrl/treeviewadv.asp>.

[17] *Microsoft Corporation:* Visual SourceSafe - Architecture. *Microsoft.com.* [Online] [Citace: 06. říjen 2007] .
Dostupný z WWW: <http://msdn2.microsoft.com/en-us/library/ms181039(VS.80).aspx>.

[18] ESTUBLIER, J., et al: Impact of the research community on the field of software configuration management: summary of an impact project report, New York: ACM SIGSOFT Software Engineering Notes, 2002 s. 31-39, ISSN:0163-5948

[19] TICHY, W. F.: RCS - A system for version control. *Software - Practice & Experience.* New York: John Wiley & Sons, Inc., *1985*, s. 637-654. ISSN:0038-0644

[20] COLLINS-SUSSMAN, B., FITZPATRICK, B. W., PILATO, C. M.: *Version Control with Subversion* [Online]. c2005 [Citace: 28. Leden 2008].
Dostupný z WWW: <http://svnbook.red-bean.com/en/1.1/index.html>.

[21] W3C. *Extensible Markup Language (XML)* [online]. c2003, 2008/01/30 [Online] [Citace: 15. Březen 2008]. Dostupný z WWW: <http://www.w3.org/XML/>.

[22] Bennett, D.: *David's .net tales: Extending RichTextBox*. April 05, 2005 [Online] [Citace: 24. Září 2007]. Dostupný z WWW:
<http://davidnetfrog.blogspot.com/2005_04_01_davidnetfrog_archive.html>.

[23] Černobila, R.: Serverová část systému pro správu projektové dokumentace. Brno, 2008, diplomová práce, FIT VUT v Brně.

[24] *Tigris.org:* Open Source Software Engineering Tools [online]. c2006 [Citace: 24. Duben 2008]. Dostupný z WWW: <http://subversion.tigris.org/>.

Seznam příloh

Příloha A - Pojmy ve verzovacích systémech

Branch – Funguje stejně jako *Tag*, ale dále se v něm pracuje. Po dokončení změn se spojí s hlavním stromem. Vlastní *branch*, neboli větev projektu, je vhodné použít pro dlouhodobější práce, které zásadněji ovlivňují nebo ruší funkčnost modulu, příp. algoritmu (funkční verze v *Trunk* je mezitím k dispozici pro drobnější změny).

Check out – Vytažení určené verze z archivu, neboli vytvoření pracovní kopie.

Commit – Odeslání změn v pracovní kopii do archivu s komentářem o účelu změn.

Conflict – Nelze provést *Commit* automaticky a nová verze se musí určit ručně.

Export – Vytažení souborů z archivu bez pomocných dat, která se normálně vytvářejí při založení pracovní kopie.

Import – Vložení neverzovaných dat do archivu bez nutnosti vytvářet pracovní kopii.

Log – Výpis revizí s odpovídajícími komentáři popisujícími změny a se jmény uživatelů, kteří změny provedli.

Merge – Spojení dvou paralelních (nebo podobných) verzí. Při operaci *Commit* probíhá ve valné většině případů automaticky (spojuje se *working copy* a aktuální verze v archivu).

Repository – Archiv obsahující všechny verze a log. Zpřístupněný serverem.

Revision – Číslo verze. Při každé změně se zvýší. Je společné pro celý archiv.

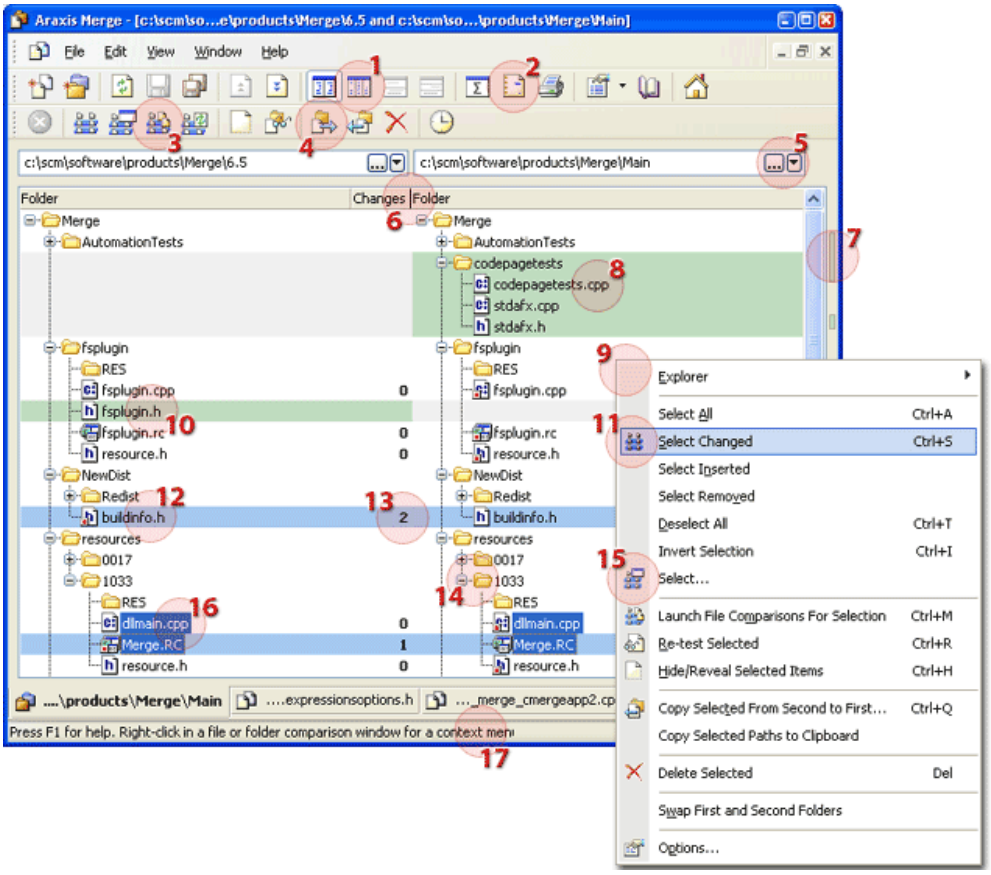
Tag – Kopie stromu nebo podstromu projektu v archivu označující specifickou verzi v čase. Dále se nemění, existuje jen pro informaci.

Trunk – Hlavní vývojová větev v archivu (trunk česky - „kmen“). Je to nejnovější funkční „oficiální“ verze.

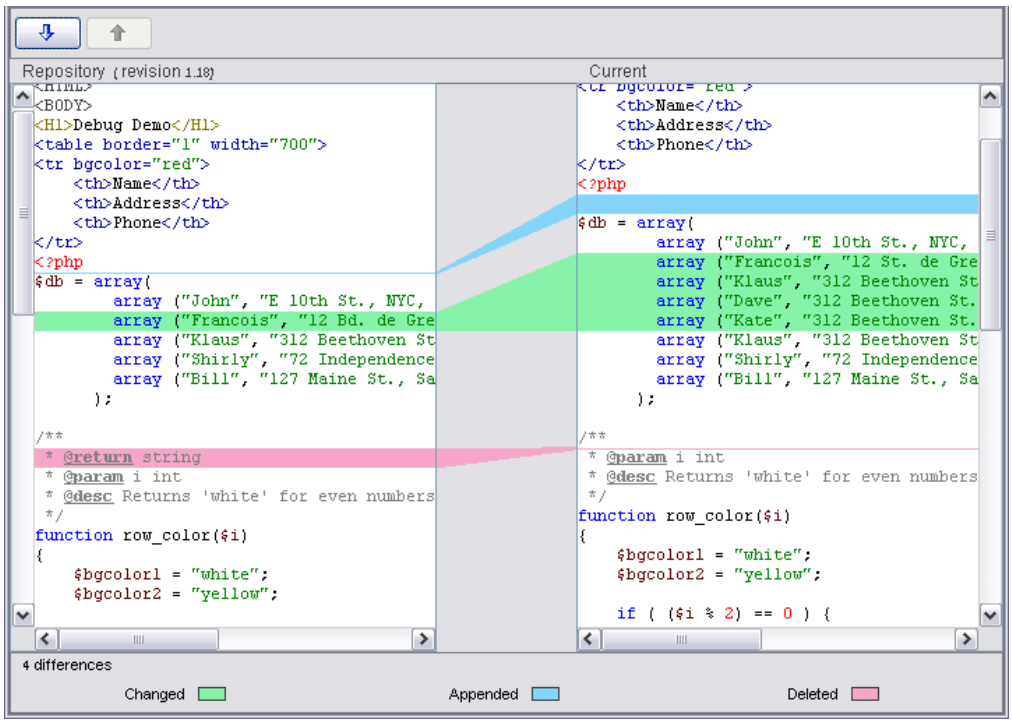
Update – Aktualizace pracovní kopie podle nejnovější verze v archivu. Nezničí místní změny provedené v pracovní kopii (varuje při konfliktu – *Conflict*).

Working copy – Pracovní kopie, obsahující verzi z archivu plus případné změny.

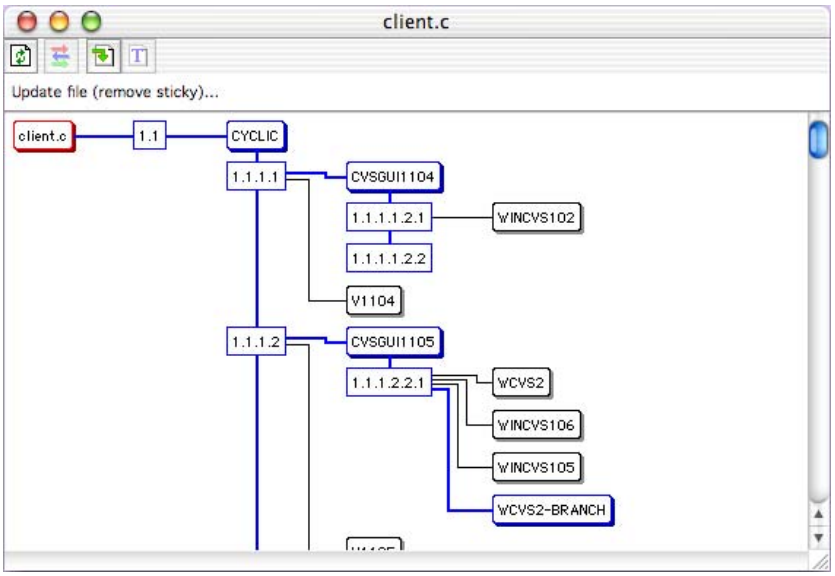
Příloha B - Příklady uživatelských rozhraní



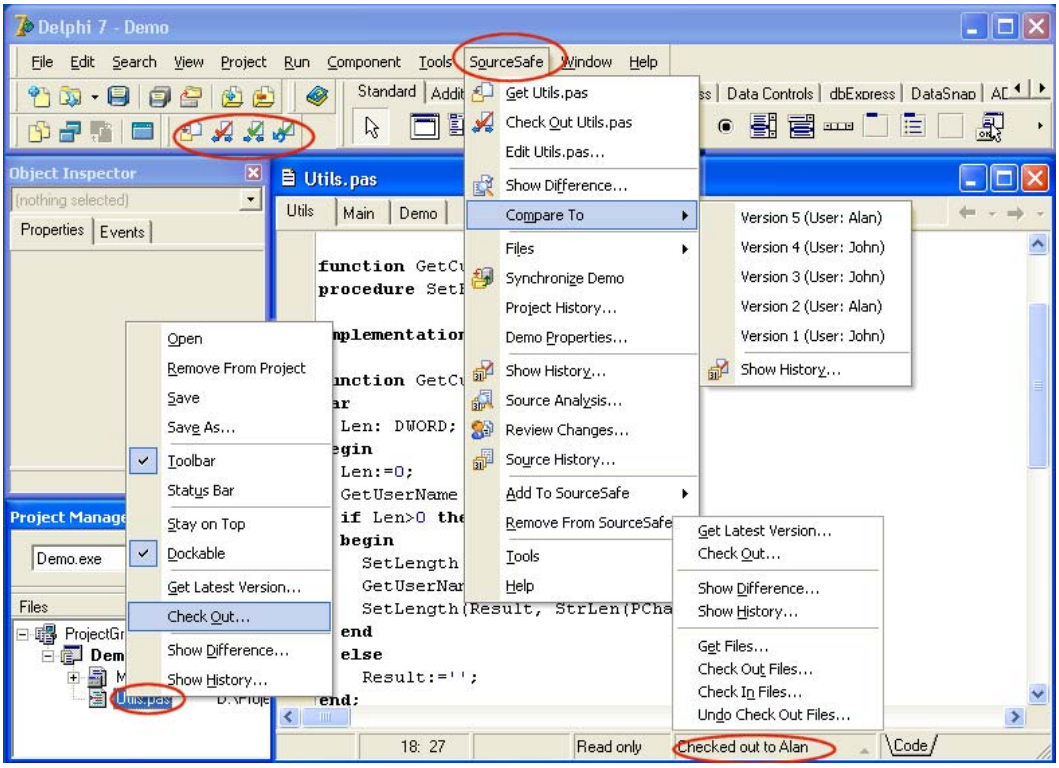
Obrázek B-1 - Araxis Merge - Zobrazení rozdílů mezi adresářovými strukturami [12]



Obrázek B-2 - Zend Studio (část grafická nadstavba CVS) - Zobrazení diference mezi verzemi [13]



Obrázek B-3 - MacCVS - Vývoj větví [14]



Obrázek B-4 - VssConneXion - Integrace Visual SourceSafe do IDE Delphi 7 [15]

Příloha C - Obsah přiloženého CD-ROM

- doc\ – elektronická verze tohoto textu
- exec\ – spustitelná verze klientské aplikace, spustitelná verze FileDiff aplikace
- sources\ – kompletní UML model ve formátu MS Visio
- sources\ScarabeusClient\ – zdrojové kódy jednotlivých aplikací, zásuvných modulů a knihoven potřebných pro sestavení aplikace včetně zavedeného projektu v Microsoft Visual Studio 2005