Univerzita Pardubice

Fakulta elektrotechniky a informatiky

Testování úloh v Java

Daniel
Dvořáček

Bakalářská práce

2014

Univerzita Pardubice
Fakulta elektrotechniky a informatiky
Akademický rok: **2013/2014**

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

(PROJEKTU, UMĚLECKÉHO DÍLA, UMĚLECKÉHO VÝKONU)

Jméno a příjmení:	**Daniel Dvořáček**
Osobní číslo:	**I11036**
Studijní program:	**B2646 Informační technologie**
Studijní obor:	**Informační technologie**
Název tématu:	**Testování úloh v Java**
Zadávající katedra:	**Katedra informačních technologií**

Zásady pro vypracování:

Navrhněte a implementuje systém automatického ověřování odevzdaných úloh v jazyce Java. V teoretické části bakalářské práce budou popsány metodiky a organizace testování. Dále bude sestaven přehled nástrojů pro testování a zmíněna problematika automatického vytváření testovacích případů.
V realizační části bude popsán návrh a implementace aplikace, která bude kontrolovat odevzdané úlohy v jazyce Java. V příloze bude projekt s aplikací na testování úloh.

Rozsah grafických prací:

Rozsah pracovní zprávy:

Forma zpracování bakalářské práce: **tištěná/elektronická**

Seznam odborné literatury:

[1]Pecinovský, R. Návrhové vzory. Computer Press, 2007. ISBN 978-80-251-1582-4.
[2]Arlow, J., Neustadt, I. UML a unifikovaný proces vývoje aplikací. Praha: Computer Press. ISBN 80-7226-947-X.
[3]Meilir Page-Jones. Základy objektově orientovaného návrhu v UML. Grada, 2001. ISBN 80-247-0210-X.
[4]Stephens, M.; Rosenberg, D. Testování softwaru řízené návrhem, Computer Press, ISBN 9788025136072

Vedoucí bakalářské práce: **Ing. Karel Šimerda**
Katedra softwarových technologií

Datum zadání bakalářské práce: **20. prosince 2013**
Termín odevzdání bakalářské práce: **9. května 2014**

L.S.

prof. Ing. Simeon Karamazov, Dr.
děkan

Ing. Lukáš Čegan, Ph.D.
vedoucí katedry

V Pardubicích dne 31. března 2014

Prohlašuji:
Tuto práci jsem vypracoval samostatně. Veškeré literární prameny a informace, které jsem v práci využil, jsou uvedeny v seznamu použité literatury.
Byl jsem seznámen s tím, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorský zákon, zejména se skutečností, že Univerzita Pardubice má právo na uzavření licenční smlouvy o užití této práce jako školního díla podle § 60 odst. 1 autorského zákona, a s tím, že pokud dojde k užití této práce mnou nebo bude poskytnuta licence o užití jinému subjektu, je Univerzita Pardubice oprávněna ode mne požadovat přiměřený příspěvek na úhradu nákladů, které na vytvoření díla vynaložila, a to podle okolností až do jejich skutečné výše.

Souhlasím s prezenčním zpřístupněním své práce v Univerzitní knihovně.

V Pardubicích dne 5. 5. 2014

Daniel Dvořáček

Poděkování

Chtěl bych poděkovat vedoucímu bakalářské práce Ing. Karlu Šimerdovi za cenné rady svému domácímu zázemí za podporu při tvorbě této bakalářské práce.

ANOTACE

Cílem práce je navrhnout a implementovat systém automatického ověřování odevzdaných úloh v jazyce Java. V teoretické části bakalářské práce budou popsány metodiky a organizace testování. Dále bude sestaven přehled nástrojů pro testování a zmíněna problematika automatického vytváření testovacích případů. V realizační části bude popsán návrh a implementace aplikace, která bude kontrolovat odevzdané úlohy v jazyce Java. V příloze bude projekt s aplikací na testování úloh.

KLÍČOVÁ SLOVA

Java, testy, automatická kontrola, subversion

TITLE

Test of projects in Java

ANNOTATION

The goal is to design and implement system for automatic validation of tasks in Java. Theoretical part of bachelor's thesis will contain description of testing methods and its organization. There will be also summary of tools for testing and problems of the automatic creation of test cases. In practical part, application’s design and implementation will be described. This application will examine submitted tasks in Java language. Application project for task tests will be in attachment.

KEYWORDS

Java, tests, automatic check, subversion

OBSAH

SEZNAM ILUSTRACÍ A TABULEK

SEZNAM ZKRATEK A ZNAČEK

SCCS – Source Code Control System

CVS – Concurrent Version Systém

SVN - Subversion

Úvod

Trendem současné moderní doby je začleňování výpočetní techniky do každodenního života, a to ve velké míře. Ať už se jedná o nahrazení manuální činnosti člověka, obsluhu abstraktnější činnosti (vědeckých výpočtů, manažerských systémů, účetních systémů, …), či jakéhokoliv jiného využití, je zde potřeba vývoje a implementace softwaru, který bude zajišťovat správný chod takové techniky. Již od počátku jde ruku v ruce s vývojem takových softwarů nutná potřeba testování. Tak jako stoupá úroveň a náročnost používaných programů, vyvíjejí se i postupy praktikované při testování těchto programů. Testování je tedy nutně součástí samotného postupu při vývinu každého softwaru.

V teoretické části této práce budou popsány některé metodiky programování softwaru, spolu s jejich klady i zápory. Tak jak se vyvíjí technika, vyvíjejí se i nové metodiky programování, z toho důvodu není možné dostatečně obsáhnout všechny existující metodiky.

Při vývinu programů se ve většině případů používá nástrojů poskytujících ulehčení realizace spolupráce mezi jednotlivými programátory v týmu. Jednou z možností jsou systémy správy verzí. Funkcionalita těchto systémů, konkrétně systému Subversion bude v práci detailně popsána. V praktické části je pro automatické spouštění testů využito možností právě tohoto systému správy verzí.

Použité konvence

Kurzíva – pojmy v anglickém jazyce, pro které není ustálený český překlad nebo výraz není překládán například z důvodu srozumitelnosti.

`Neproporcionální styl písma` – programové příkazy, názvy programových částí apod.

1 Metodiky a organizace testování

Metodika jako taková je ustáleným postupem při určitém pracovním postupu. Zvolená metodika má za úkol nastavit jistý rámec, který mimo jiné bude definovat návaznosti jednotlivých dílčích činností vývoje softwaru, vstupy a výstupy takových činností, případně podmínky pro opakování jednotlivých částí. Metodika vývoje softwaru také nutně obsahuje způsob a provedení testování vyvíjeného produktu. Metodiku vývoje softwaru lze také nazývat modelem životního cyklu takového vývoje.

1.1 Model velkého třesku

Tento model je možné považovat za historického předchůdce dnešních metodik, který vznikl při první potřebě vývoje softwaru. Jedná se v podstatě o nejjednodušší možnou metodiku. Tento postup se skládá z počátečního nahromadění všech potřebných zdrojů (finančních prostředků, lidí), vynaložení energie na vývoj, po němž z postupu vyjde výsledný produkt, který může, ale nemusí, ideálně splňovat počáteční požadavky.

Metodika velkého třesku nevyžaduje téměř žádné plánování a rozdělování jednotlivých činností při vývoji. Důležitá část, kterou testování bezesporu je, u tohoto modelu zcela chybí. Tato skutečnost může výrazně ovlivnit výslednou efektivnost a úspěšnost vyvíjeného softwaru. [1]

1.2 Model vodopádu

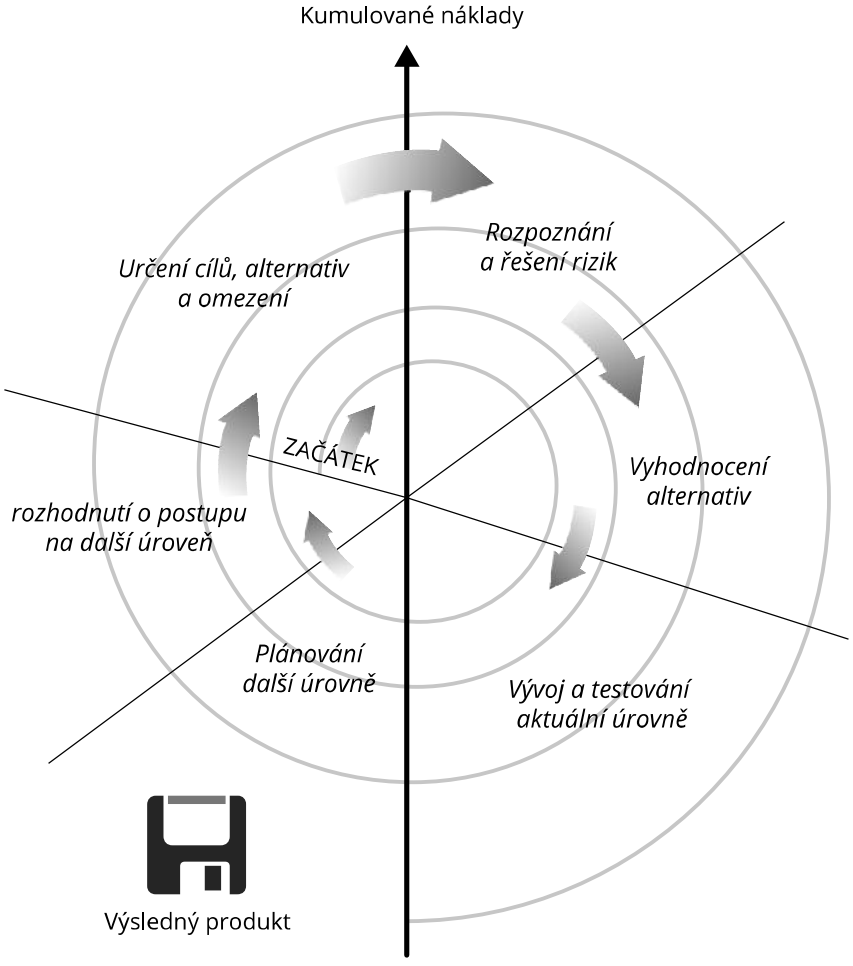
Model vodopádu neboli kaskádový postup je jedním z nejznámějších existujících vývojových postupů. Na rozdíl od předchozího modelu již definuje jednotlivé rámce dílčím činnostem při vývoji. Vývoj softwaru je zde rozdělen do šesti oblastí. Jako první je sběr informací (nápad), který udává základní myšlenku celého projektu. Následuje analýza, návrh, vývoj a závěrečné testování. Po průchodu těmito částmi vzniká výsledný produkt. Jednotlivé fáze na sebe navazují a to bez jakéhokoliv překrytí. Při postupu jsou používány výstupy předchozí části jako vstupy části nadcházející.

Nevýhodou modelu vodopádu je nutnost striktního dodržení všech návazností, což znemožňuje například úpravu vstupních požadavků pro celý projekt po opuštění fáze sběru požadavků. Další zápornou vlastností kaskádového pojetí je nemožnost vrácení se do jakékoliv předchozí části postupu. V takovém případě je nutné dosavadní postup „zahodit“ a začít znovu od začátku.

Díky zmíněným vlastnostem není tento model nikterak flexibilním. Nicméně pro jednodušší projekty, u kterých jsou známy všechny vstupní požadavky, a možnost změn při průběhu vývoje je nízká, lze tento postup s výhodou pro jeho jednoduchost uplatnit. Struktura a pojetí tohoto modelu tvoří základ pokročilejších metodik vývoje softwaru. [1]

1.3 Iterativní přístup – Spirálový model

Spirálový model (viz Obrázek 1) s sebou mimo jiné přináší řešení nedostatků předchozích metodik. Jedná se o jednu z implementací iterativního přístupu, u které je podobně jako u kaskádového postupu, rozdělen vývoj projektu do jednotlivých částí. První v posloupnosti je stejně jako u předchozího modelu analýza, určení cílů a omezení. V další fázi jsou rozpoznávána a řešena případná rizika, následně se vyhodnotí alternativy k těmto řešením.



Obrázek 1 Grafické znázornění spirálového modelu [1]

Dále přichází právě fáze testování a kontroly požadovaných vlastností právě vyvíjeného fragmentu celého projektu. Po realizaci testů se naplánuje další úroveň, případně další postup.

Tyto jednotlivé fáze jsou opakovány ve spirále do té doby, než je vyvíjený produkt kompletní. Každá iterace přináší funkční prototyp, který sice neobsahuje řešení všech požadovaných problémů, ale poskytuje možnost průběžné kontroly. Tato vlastnost je velmi cennou, neboť cena odstranění případně objevené chyby s postupem času od základní myšlenky až po finální distribuci softwaru stoupá exponenciálně.

Model spirály také umožňuje průběžné úpravy vstupních požadavků, což u předchozího kaskádového pojetí nebylo možné. [1]

1.4 Agilní metodiky

Trendem současnosti jsou agilní metodiky. Dosud zmíněné postupy by se daly označit také jako metodiky rigidní, tedy ztuhlé, těžkopádné a nepoddajné. Naproti tomu metody agilní jsou tomu naprostým opakem. Dají se považovat za další vývojový stupeň v rámci metod vývoje softwaru. U těchto metod je dáván důraz na spokojenost zákazníka a s ní související zapojení zákazníka do postupu vývoje. Typickou vlastností agilních metod je také iterativní postup s co možná nejkratšími časovými délkami jednotlivých iterací. Z tohoto důvodu jsou zmiňované metodiky vhodné pro menší vývojářské týmy, pracující v jedné společné místnosti. Příkladem agilních metod je metodika extrémního plánování nebo metodika *Scrum*. [4]

1.5 Scrum

Scrum je jedna z nejznámějších agilních metod vývoje softwarových řešení. Vytvořil ji Jeff Sutherland v roce 1993 na základě svých předchozích zkušeností s vývojovými postupy softwaru. *Scrum* je tedy metodikou vývoje programů, lze ji však úspěšně uplatnit na jakýkoliv podobný proces. Jedním ze základních kamenů přístupu *Scrum* je spolupracující tým, který je schopen samostatné organizace. V týmu má každý jednotlivec stanovenou svoji roli a podle toho také své dovednosti aplikuje. [2]

1.5.1 Rozdělení pracovního týmu

Základní rozdělení týmu je následující:

- *Scrum master* – jehož role by se dala přirovnat k roli projektového manažera.
- *Product owner* – zastupuje zadavatele, tedy zákazníka zadávajícího práci.
- *Tým* – tým samotných vývojářů.

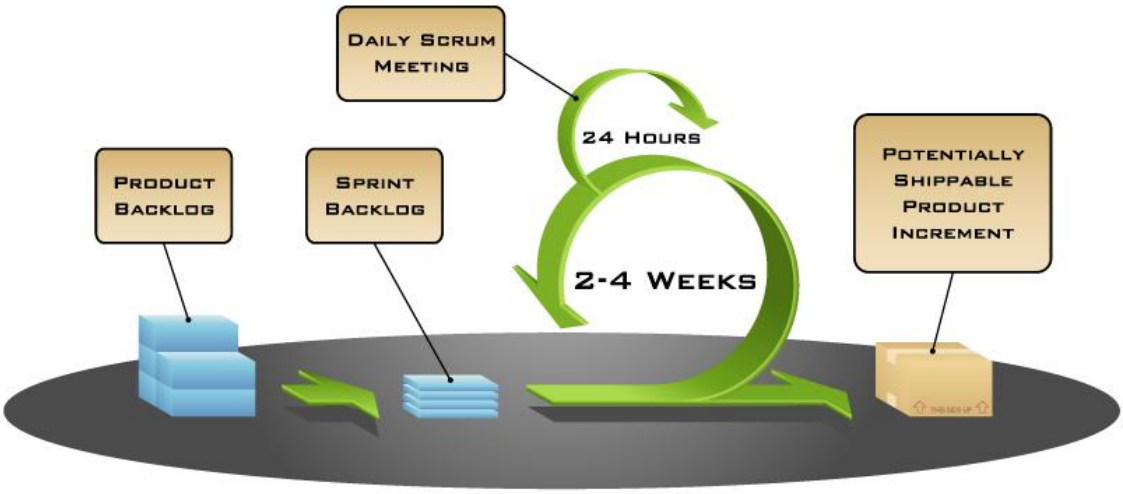
Hlavní zodpovědnost za management projektu zaštiťuje zmíněný *scrum master*. Další položkou na poli jeho působnosti je dozor nad existencí samotné myšlenky metodiky *Scrum* a odstraňování vzniknuvších problémů v okolí týmu. *Scrum master* má na starost mimo jiné i komunikaci mezi jednotlivými členy týmu a ochranu před nepříznivými vlivy zvenčí.

Product owner vytváří soupis jednotlivých požadovaných kroků s prioritami, které sám určí. Tento soupis tvoří tzv. *product backlog*, z něhož se vybírají činnosti pro jednotlivé iterace. *Product owner* také případně upravuje vlastnosti produktu v porovnání s konkurencí. A v neposlední řadě je zodpovědný za průběžné předvádění hotového postupu zákazníkovi.

Tým vývojářů by měl být spíše menšího počtu – pět až devět členů. Metodika *Scrum* je zaměřena na univerzálnost a nahraditelnost v týmu, proto by se tým pokud možno měl skládat z členů bez striktního specifického zaměření, což umožňuje nahrazení nepřítomného člena týmu jiným. [2]

1.5.2 Průběh modelu Scrum

Průběh modelu *Scrum* znázorňuje Obrázek 2. Nejdříve je vytvořen *product backlog*, který obsahuje požadavky na funkcionalitu ze strany zákazníka, případně doplňkové informace ze strany týmu. Jednotlivé iterace, po kterých vývoj postupuje, se nazývají sprinty. Před průběhem každého sprintu je *product backlog* seřazen dle priority, následně jsou z něho vybrány části pro nadcházející sprint. Určí se také časový interval sprintu, nenechávají se zde žádné rezervy, avšak v případě dřívějšího splnění sprintu je také nutná úprava časových odhadů u dalších sprintů.



Obrázek 2 Grafické znázornění základů metodiky SCRUM [3]

Po dokončení každého sprintu je produkt předváděn zákazníkovi. Nedodělky, které v tu chvíli v produktu existují, jsou skryty, vystupuje pouze funkcionalita, která je již hotova. V tento moment je zde prostor pro případné změny požadavků ze strany zákazníka. V průběhu jednotlivých sprintů není změna přípustná.

Další vlastností modelu *Scrum* je tzv. každodenní *Scrum*. Jedná se o ranní schůzi, trvající obvykle kolem patnácti minut. Účastnit by se měl vždy celý tým a celá schůze by měla být pokud možno ve stoje, což vede ke snaze diskutované problémy rychleji vyřešit. Probírán je zde dosavadní postup, výskyt a řešení případných překážek, co by bylo vhodné změnit a jaký bude další postup.

Směrodatným výstupem jednotlivých sprintů je tzv. *burndown chart*. Je to graf, ukazující průběžný postup ve sprintu. Jeho veličinou je zbývající potřebný čas na dokončení sprintu. Umožňuje také lépe předvídat celý průběh sprintu, odhadovaný pomocí dosavadního postupu.

Metoda *Scrum* je v dnešní době hojně používanou velkým počtem společností po celém světě. Správné dodržování jejích zákonitostí poskytuje velkou flexibilitu při vývoje softwaru, možnost úprav požadavků ze strany zákazníka i v pozdějších fázích projektu bez větších složitostí. Průběžné testování elementárních částí celého projektu zvyšuje efektivitu a snižuje chybovost výsledného projektu jako celku. [2]

1.6 Extrémní programování

Extrémní programování je další ze známých agilních metod vývoje. První projekt vytvářený za pomocí pravidel extrémního programování vznikl v roce 1996. Extrémní programování se podobně jako zmíněná metod *Scrum* zaměřuje na uspokojení přání zákazníka. Zaměřuje se na průběžné výstupy dílčích částí projektu, které umožňuje změny v zadání od zákazníka a to i v pozdějších fázích vývoje.

1.6.1 Principy extrémního programování

Extrémní programování je postaveno na několika základních principech, které zaručují efektivní vývoj projektu:

- jednoduchost,
- komunikace,
- zpětná vazba,
- respekt,
- odvaha.

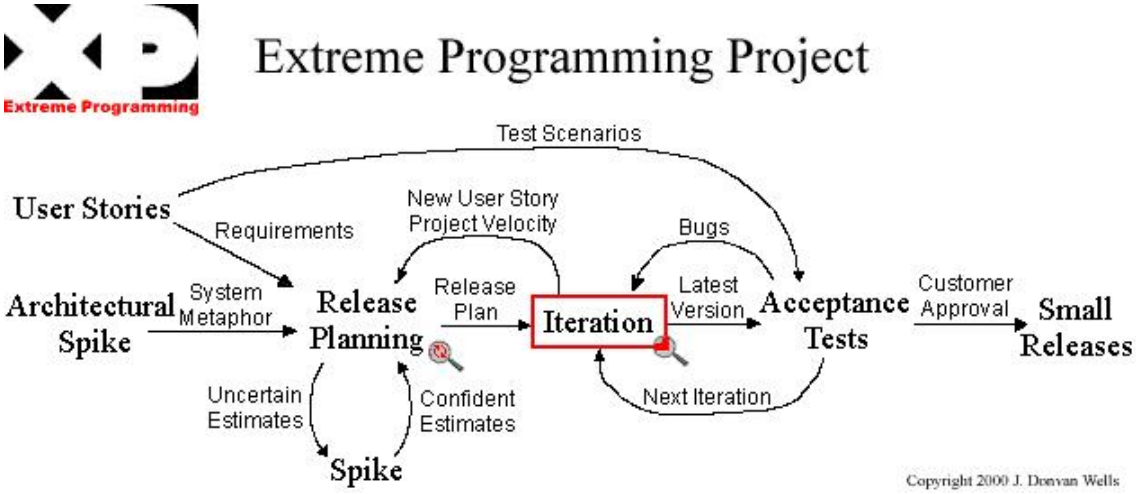
Jednoduchost je důležitou veličinou z důvodu přehlednosti. Pravidlo také říká, že v každé iteraci bude vytvořeno jen to, co se předem určí, nic navíc. Maximalizuje se tak schopnost odevzdání vypracovávaného segmentu v pořádku a včas.

Komunikace a zpětná vazba jsou dalšími důležitými elementy. Při dobré fungující komunikaci je řešení problému rychlejší a efektivnější. Zpětná vazba je potřebná pro flexibilitu vývoje, přináší možnost reagovat na přání zákazníka.

Respekt zde staví všechny členy vývojářského týmu, manažery i zákazníka na stejnou úroveň cenného člena týmu, který si zaslouží příslušný přístup. S tím souvisí i poslední princip, odvaha, kdy se např. nastavované termíny vyhotovení vždy řídí skutečností a v případě nezdarou nejsou vytvářeny výmluvy zaštiťující daný neúspěch.

1.6.2 Průběh vývoje

Stejně jako každá jiná metodika má i model extrémního programování rámcově stanovený postup vývoje. Návaznosti jednotlivých kroků jsou, popisuje Obrázek 3. Jedná se o implementaci iterativního přístupu, na základě čehož je postup stavěn.



Obrázek 3 Návaznosti jednotlivých akcí v extrémním programování [4]

1.6.2.1 Sestavení seznamu požadavků

Na začátku je nejdříve sestaven seznam požadavků, na základě tzv. scénáře (*User stories*), které určují elementární případy použití výsledné produktu stanovené přáními zákazníka. Dalším postupem je tvorba krátkých řešení oblastí, která jsou vyhodnocena jako riziková. Jednoduché a rychlé řešení takových oblastí přináší možnost zhodnocení dalšího postupu a případných alternativ pro zákazníka.

1.6.2.2 Plánování

Následuje fáze plánování, která se opakuje před každou iterací postupu. Délky iterací obvykle bývají od jednoho do tří týdnů. Ve fázi plánování jsou vybrány za spolupráce se zákazníkem scénáře, které budou zařazeny do nadcházející iterace. Ohled se zde bere i na neúspěšné kontroly přijatelnosti ze zákazníkova hlediska. Tyto položky jsou rozřazeny do úkolů, jež je budou řešit. Úkoly jsou sestaveny podobného seznamu vstupních požadavků, s tím rozdílem že jsou psány v technickém jazyce srozumitelném vývojářům. Vývojáři si z vytvořeného seznamu vyberou jednotlivé úkoly a na základě svých zkušeností stanoví časové odhady. Suma těchto odhadů potom určuje výsledný odhad délky iterace.

1.6.2.3 Návrh a programování

Další fází je fáze návrhu, kde je oceňována zejména jednoduchost. Prováděny jsou zde také zmiňované „*spiked solutions*“ – jednoduchá řešení rizikových oblastí.

Po návrhu probíhá již fáze samotného programování. Souběžně s tvorbou řešení se programují i jednotkové testy, což maximalizuje účinnost takových testů. Programování

zde probíhá v páru, kdy v jednu chvíli danou dílčí oblast vyvíjí jeden programátor, zatímco druhý je v roli pozorovatele. Tímto postupem se zamezí situacím, kdy zdrojový kód viděl pouze jeden člověk, sníží se také celková chybovost. Role obou programátorů jsou často střídány.

1.6.2.4 Jednotkové testy

Po naprogramování jsou prováděny jednotkové testy, kterými část vyvíjená v dané iteraci musí bezpodmínečně projít. V případě nálezu chyby jsou na nalezenou chybu vytvořeny další jednotkové testy na základě znalostí funkcionality testované části.

1.6.2.5 Testy akceptace

Následují testy akceptace, které kontrolují produkt z pohledu zákazníka. Je zde porovnávaná praktická funkcionalita programu oproti vstupním scénářům. Zákazník v tuto chvíli může scénáře upravit či přidat nové, zapříčiní tím však vznik další placené iterace.

Tato podkapitola byla zpracována podle zdroje [4]

1.7 Další vývojové metodiky

V případě vývoje softwaru existuje v dnešní době poměrně velké množství možností, jakou metodiku k vývoji využít. Mimo zmíněné jsou to například *Crystal*, *Vývoj řízený testy* či *Vývoj řízený vlastnostmi*. Každá z metod má své výhody i nevýhody. Při výběru aplikované metody je vhodné zvážit velikost a umístění pracovního týmu, objemnost projektu, podle čehož je možné vybrat vhodnou metodiku.

2 Testovací metody

K testování softwaru jako takovému lze přistupovat několika cestami. Každá má svá pro i proti a je vhodná pro rozdílné situace. Účinnější je pak kombinace několika způsobů.

2.1 Jednotkové testy

Jednotkové testy, neboli „*Unit testy*“, jsou nástrojem pro verifikaci a validaci softwaru. Touto metodou je možné otestovat veškeré funkční části kódu samostatně a prověřit tak průběh jednotlivých částí. Jednotkové testy mohou být metodou naprogramovanou ve stejném jazyce jako je testovaný program, kontrolující funkce testovaného programu jak jednotlivě, tak i kombinovaně.

2.2 Testování černé a bílé skříňky

V případě testování černé skříňky analyzujeme data programu na základě vlastností, které vystupují na povrch. Porovnávají se vstupní a výstupní data bez jakékoliv znalosti vnitřní implementace programu. Tento způsob je jednoduchým testováním, které však neumožňuje cílené hledání chyb a nenapomáhá při opravě nalezených chyb. Nejvíce se jím však přiblížíme způsobu konečného používání programu

Testy bílé skříňky neboli skříňky transparentní naopak probíhají se znalostmi vnitřního fungování programu. Tato skutečnost umožňuje lepší cílení testů a jejich testovacích dat, vedoucí k důkladnějšímu prověření funkcionality testovaného produktu.

2.3 Regresní testování

Provádění regresního testování prověřuje, zda byla objevená chyba odstraněna, či nikoliv. Regresní testy mohou být spouštěny po každé dílčí změně, pro prověření funkcionality jak nové části, tak i částí předchozích. Kontrolovány jsou tak situace, kdy by připsání nového kódu programu mělo záporně ovlivnit již existující části. Z důvodu nutnosti častého vykonávání takových testů je vhodné regresní testy zautomatizovat.

2.4 Integrační testování

Při integračním testování jsou jednotlivé části vyvíjeného produktu poskládány do jednoho a testovány jako celek. Integračnímu testování ve většině případů předchází jednotkové testování. V této metodě je otestována funkcionalita celého projektu, bezchybnost jeho navazujících částí a soudržnosti.

Tato kapitola byla zpracována podle zdroje [1]

3 Nástroje pro testování

Manuální testování vyvíjených programů neumožňuje ani zdaleka testování počtu situací a stavů blížícím se všem možným situacím. Pro účely zjednodušení testování byly již od počátku vytvářeny nástroje, které testování zjednoduší a zefektivní. Nástroje pro testování lze rozdělit do několika základních skupin.

3.1 Obecné rozdělení

Prohlížeče a monitory

Tyto nástroje umožňují zejména neinvazivní pozorování běhu testovaného programu. Umožňují pozorování vnitřního chodu programů, který je z vnějšího prostředí nepřístupný. Může se jednat například o pozorování toku informací v síti. Prohlížeči v tomto smyslu se dají považovat i ladící nástroje, poskytované obvykle v dokonalejších vývojových prostředích většiny programovacích jazyků.

Ovladače

Ovladače v případě testování jsou nástroji pro ovládání testovaného softwaru. Používají se pro simulaci vstupních akcí přicházejících z vnějšího prostředí. Použitím ovladačů se docílí neinvazivního zásahu, který může být pro určité testovací situace nutností.

Makety a emulátory

Nástroji pro testování výstupních dat testovaného programu jsou makety a emulátory. Na rozdíl od ovladačů, které nastavují vstupní data, se zde pouze zpracovávají data, která jsou programem posílána na výstup. Díky těmto nástrojům je tak možné otestovat například výstup do tiskárny, při kterém by za normálních okolností analýza výstupních dat na reálné tiskárně byla méně přesná z důvodu kontroly pouhým okem.

Nástroje pro zátěžové testy

Po ověření plné funkcionality programu provedené testy zaměřenými pouze na daný software, je důležité otestovat chování běhu programu za podmínek, které jsou pro jeho užití typické. Pokud se jedná o běžný uživatelský program, souběžně s jeho chodem bude zabírat systémové prostředky řada dalších programů. Proto je nutné užití takových nástrojů pro kontrolu chování.

Nástroje pro automatické testy

Z důvodů usnadnění provádění testů a umožnění kontroly velkého množství variant vstupních dat, jsou zde právě nástroje pro automatické testy. V současnosti existuje široká nabídka volných i komerčních nástrojů. Při automatických testováních mohou být používána programová makra i propracovanější testovací programy.

Tato podkapitola byla zpracována podle zdroje [1]

3.2 Příklady nástrojů pro testování

Na trhu v současnosti existuje řada komerčních i nekomerčních nástrojů pro aplikaci různých způsobů testování. Často jsou nástroje umožňující testování vyvíjeny v spolupráci s vývojovým prostředím

3.2.1 Nástroje pro jednotkové testování

Pro většinu programovacích jazyků existuje velká řada různých nástrojů pro provádění jednotkových testů. Mezi tyto nástroje se mimo jiné řadí:

- *JUnit* – asi nejznámější nástroj pro testování programů v jazyce Java. Jedná se o nekomerční nástroj poskytující jednoduchý přístup k vytváření testovacích programů.
- *NUnit* – nástroj s porovnatelnou funkcionalitou jako *JUnit*, pracující s programy v jazycích Microsoft .NET. Tento nástroj je také nekomerční.
- *Jasmine – open source* nástroj pro testování dynamických částí webových stránek psaných v jazyce JavaScript.
- *PHPUnit* – vytvořený pro provádění jednotkových testů skriptovacího jazyka PHP. Výstupem spuštěných testů můžou být různé typy forem – XML, JSON a další.
- *SQL Developer* – program pro tvorbu databázových příkazů PL/SQL obsahující také nástroje pro jednotkové testování.

3.2.2 Nástroje pro sledování dat

- *Wireshark* – nástroj pro sledování síťového protokolu. Umožňuje efektivně sledovat typy průchozích paketů, zobrazovat data z hlavičkových informací a mnoho dalšího.
- *System explorer* – jedná se v podstatě o klasického správce úloh v systému Windows, obsahující informace o využití zdrojů jednotlivých běžících programů.

3.2.3 Nástroje pro automatické testování

Efektivní použití automatického testování přináší jednak úsporu času, který by byl potřebný pro manuální spouštění testů a také zvyšuje efektivitu testování jako takového uplatněním například regresního testování.

- *Mercury QuickTest Professional* – komerční nástroj pro zavedení automatického testování širokého spektra aplikací a prostředí. Používá jazyka VBScript, jehož znalost je při užívání tohoto systému prospěšná, ne však nutná.
- *WET Web Tester* – nekomerční nástroj pro testování webových aplikací. Je výtvorem dvou produktových manažerů společnosti Google.
- *Canoo WebTest* – další nástroj pro automatické prověřování funkčností webových aplikací. Jedná se také o *open source* nástroj.

4 Automatické testování

Automatické testování, tedy automatické spouštění předem připravených testů, samo o sobě může a nemusí být vždy tou správnou volbou. Před jeho aplikováním je nutné zvážit a nadále mít na paměti několik skutečností.

Software se neustále vyvíjí

Tento fakt je potřeba mít při vytváření automatických testů na paměti. Důležité je vytvářet flexibilní testy, které budou schopny co možná nejjednoduššího přizpůsobení novým částím programu, stejně tak úpravám provedených na již existujících částech.

Automatické testy nezastoupí lidskou kontrolu

Ať už jsou vytvářené automatické kontroly jakkoliv promyšlené, nikdy nezastoupí manuální testování člověkem zcela. Stejně tak verifikace se velmi obtížně vykonává automatickou cestou z důvodů strojového porovnávání dat.

Počáteční investice

V případě zvolení robustních komerčních produktů, obsluhujících veškerou funkcionalitu automatického testování je třeba počítat i s výší investice do takového nástroje. Před použitím je dobré zvážit nutnost použití, například ve vztahu s velikostí testovaného projektu.

Tvorba testových programů

Náročnost tvorby takových programů přináší další náklady na čas i peníze. Program, který bude spouštěn při automatických kontrolách, může být i větší a složitější než testovaný program sám.

5 Systémy správy verzí

Pro úspěšné uplatnění zmiňovaných metod vývoje softwaru, případně provádění následného testování, a jakéhokoliv programování vůbec, je téměř nutné nejen při práci více programátorů na stejném projektu použití některého systému správy verzí.

Jednou z hlavních výhod používání takového systému je uchovávání historických verzí. Díky uchovávání změn v minulosti, je poté hledání problému, který zapříčinil chybu v poslední verzi programu, daleko jednodušší. Pokud je pak nutnost ukázat v danou chvíli program, který funguje bez chyb, není nic jednoduššího než nahrát poslední fungující verzi. Vytvářet takové zálohové struktury manuálně je samozřejmě možné, systém správy verzí však tuto funkcionalitu zastává daleko spolehlivěji a efektivněji.

Další z mnoha přínosů používání systému správy verzí je jednoduché uskutečnění spolupráce na projektech, tedy sdílení programových kódu jednotlivých částí programu. Není zde nutnost kopírování souborů z jednoho místa na druhé, stačí pouze pomocí nějakého klienta systému správy verzí provést nahrání souborů, případně pouze změn do požadované složky.

5.1 Historie

Systémy správy verzí byly s výhodou využívány již na přelomu 70. a 80. let. První používaný SCCS (*Source Code Control System*) pocházel z Bellových laboratoří. Dalším postupem v této oblasti byl vývoj systému CVS (*Concurrent Version System*). Začátky CVS systému se datují do roku 1986, kdy Dick Grune zaslal do jisté diskusní skupiny sadu skriptů v *shellu*. Základní algoritmy vyvinuté v tuto chvíli jsou používány i v aktuálních verzích CVS. V roce 1989 byl uskutečněn návrh systému CVS Briana Berlinera, později na systému spolupracoval také Jeff Polk.

Současníkem v aktuálních systémech správy verzí, který je možno považovat za jeden z nejpoužívanějších je SVN (*Subversion*). Tento systém se začal rozšiřovat na začátku 21. Století. Jeho základy položila firma CollabNet v roce 2000, když potřebovala nahradit a zdokonalit systém správy verzí který používala. SVN byl pak dokončen konce srpna roku 2001.

Pojem *subversion* se velmi často nahrazuje pouze zkratkou SVN, z důvodu významu slova *subverison* (česky podvracení, zkáza, svržení), což u vedoucích pracovníků při přechodu na tento systém správy verzí vyvolává nežádoucí asociace.

Subversion byl po dlouhou dobu jedním z nejpoužívanějších systémů správy verzí. V dnešní době ho ale začínají nahrazovat některé z distribuovaných systémů (*GNU arch*, *Git*, *Darcs*, *Monotone*…) [5]

5.2 Subversion

Subversion je tedy nekomerční, otevřený systém, který umožňuje správu souborů a jejich jednotlivých verzí. Vychází systému CVS, z něhož má některé základní vlastnosti. SVN je však jednodušší pro běžné používání a odstraňuje jisté neflexibilní oblasti systému CVS. SVN například obvykle nepoužívá zamykání souborů při úpravách. Paralelně prováděné změny automaticky spojuje, když se nepřekrývají. Při překrytých změnách vznikne hlášení konfliktů, které je nutné ručně vyřešit. Ke konfliktům by nemělo při správné organizaci vývoje docházet.

5.2.1 Podpora operačních systémů

Subversion je možné provozovat na všech typech operačních systémů s příslušnými instancemi serveru systému správy verzí.

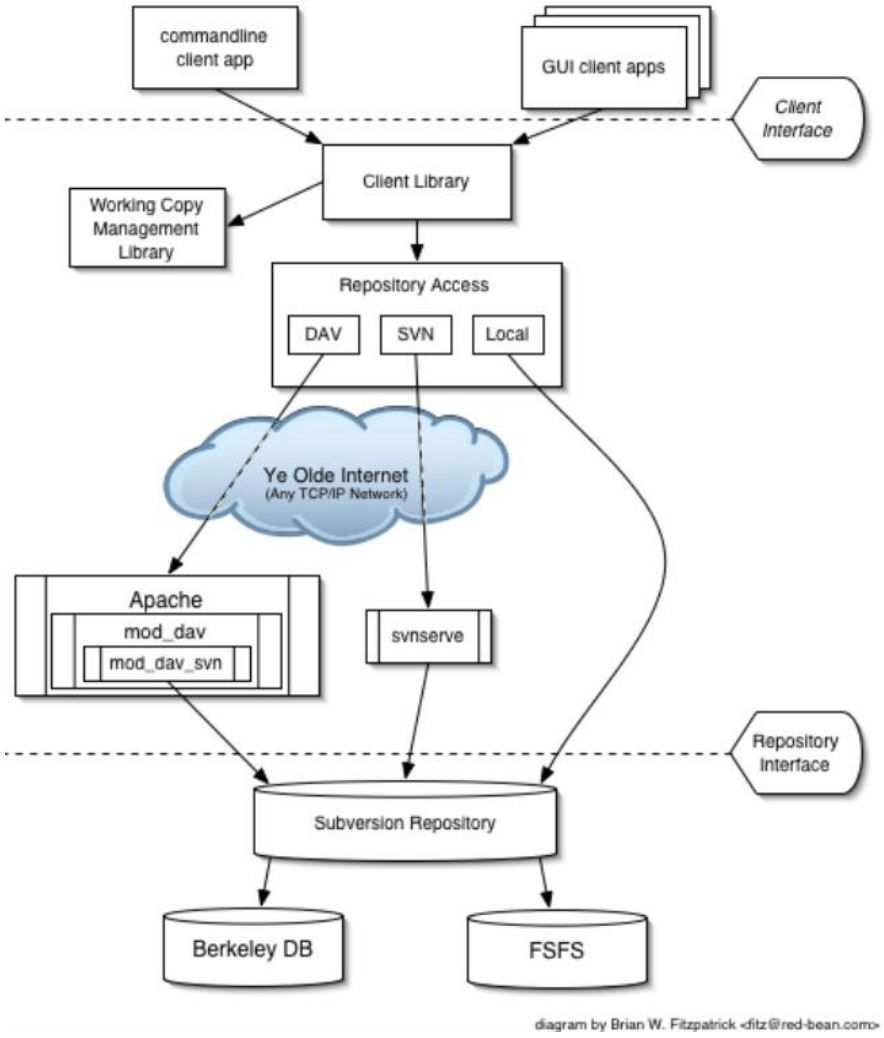
Pro Windows je to *Visual SVN Server*. Po intuitivní instalaci je možné provádět veškerá nastavení pomocí jednoduchého grafického rozhraní. Ve stejném rozhraní se vytvářejí a upravují jednotlivé repositáře, vytvářejí uživatelé, případně skupiny.

Pro Linuxové systémy bývá *Subversion* často součástí základních instalačních balíčků a pro platformy Mac existují obdoby těchto služeb.

5.2.2 Architektura SVN

SVN je ve své podstatě centralizovaný systém, což znamená, že funkcionalita je zde založena na vztahu server-klient. Na straně serveru jsou jednotlivé archivy, neboli repositáře, které obsahují veškerá verzovaná data. Na druhé straně je klient, který přes pomocí svého SVN klientského programu pracuje se svou pracovní kopií, která je odrazem verzovaných dat na serveru.

Mezi těmito částmi je několik cest (viz Obrázek 4), kterými se data dostávají ze serveru ke klientovi a obráceně. Jedná se o cesty přes obslužné protokoly, které pomocí serverů přistupují k repositářům, nebo přímé cesty přístupu k repositářům, například při lokálním přístupu.



Obrázek 4 Funkcionalita systému správy verzí SVN [6]

5.2.3 Nástroje SVN

Pro obsluhu serveru systému správy verzí disponuje SVN několika základními nástroji:

- `svn` – klientský program příkazové řádky.
- `svnversion` – program pro zjišťování stavu pracovní kopie, poskytuje také informace o jednotlivých revizích.
- `svnlook` – nástroj pro přímou práci s repositáři.

- `svnadmin` – používá se pro vytváření, úpravy či opravování jednotlivých repositářů, umožňuje také mimo jiné úpravu logových zpráv k jednotlivým verzím souborů.
- `mod_dav_svn` – modul pro používaný *Apache HTTP Server*, sloužící k umožnění přístupu k repositářům prostřednictvím sítě.
- `svnserve` – samostatný serverový program poskytující další přístupovou cestu k repositářům pomocí sítě.
- `svnsync` – program usnadňující udržování kopie repositářů pomocí sítě.
- `svnrdump` – pro zobrazování historie repositářů, případně jejich obsahu.

5.2.4 Základní pojmy SVN

SVN je systém, který uchovává informace o jednotlivých verzích (neboli revizích) souborů, případně celých adresářů. Ke každé revizi existuje také záznam o uživatelích, či skupinách uživatelů, kteří se na revizích podílejí. Díky těmto záznamům je umožněna možnost procházení jednotlivých historických verzí souborů, což je nejen při skupinovém vývoji velice cenné.

Repositáře

Repositáře jsou základními datovými strukturami, ve kterých se data uchovávají na straně serveru. Repositáře jsou obvykle organizovány do stromové struktury souborového systému, jenž obsahuje hierarchii adresářů a souborů. Ke každému repositáři může mít přístup neomezený počet uživatelů – klientů, kteří mohou číst jejich obsah či zapisovat nový obsah.

Repositář je v podstatě souborovým serverem, který si ale na rozdíl od klasických souborových serverů uchovává informace o jednotlivých verzích souborů. Z repositáře lze zrekonstruovat jakoukoliv předchozí verzi souboru. Z důvodu uspoření místa je v repositáři uložena vždy poslední celá verze souboru a předchozí verze pouze jako rozdíly změn. Tento mechanismus je velmi efektivní pro textové soubory, ale méně pro binární soubory.

Pracovní kopie

Pracovní kopie je v podstatě lokální obraz obsahu repositáře. V operačním systému na pracovní stanici vystupuje jako obyčejný adresář se soubory a umožňuje tak práci s těmito

soubory bez nutnosti zpracovávání informací o verzích na straně klientského operačního systému.

Tyto lokální pracovní kopie umožňují bezpečné úpravy sdílených souborů, které zamezují nechtěnému přepisování dat mezi jednotlivými spolupracujícími uživateli.

Checkout

Operace, při které se vytvoří lokální obraz repositáře – pracovní kopie na pracovní stanici. *Checkout* je v podstatě vyzvednutí projektu, kdy je zkonstruován lokální pracovní prostor.

Commit

Commit je pojem pro odeslání lokální pracovní kopie z pracovní stanice, respektive změn provedených od poslední revize. *Commit* jako takový je z hlediska SVN atomickou operací. V případě že není úspěšný, nevytváří se nová kopie a obsah repositáře na serverové straně zůstává neměnný.

Upload

Operace, při které se z SVN serveru aktualizuje lokální pracovní kopie na pracovní stanici. Lokální nezveřejněné změny v pracovní kopii nejsou přepsány, aktualizovány jsou pouze části souborů, které jsou ve starší revizi.

Lock

Uzamčení části, nebo případně celého repositáře, za účelem zamezení konfliktních situací způsobených paralelně probíhající úpravou souborů více uživateli. V případě uzamčení může danou část upravovat prostřednictvím odesílání pracovní kopie pouze uživatel, který část uzamkl. Po odemčení se části vrací běžná funkcionalita.

Revize

Při každém úspěšném odeslání pracovní kopie na serverovou stranu systém vytváří novou revizi adresářů. Každá revize je opatřená unikátním číslem revize, které začíná od nuly a je inkrementováno s každým odesláním změn z klientské strany.

Merge

Informativní stav, který označuje výsledek průběhu operace *commit*. V tomto případě byly jednotlivé změny v souborech automaticky sloučeny do výsledného korektního stavu.

Konflikt

Konflikt je stav, signalizující problém při odesílání lokální pracovní kopie repositáře. Důvodem konfliktu bývá nemožnost automatické operace *merge*, v případě, kdy jsou provedené změny v textových souborech provedeny v identických částech souboru, např. na stejném řádku. V tomto případě není tedy možné jednoznačně určit finální stav, je poté nutné manuální vyřešení situace.

Adresář .svn

V každém lokálním obrazu repositáře je automaticky vytvářena složka `.svn`. V této složce jsou informace pro systém SVN, týkající se mimo jiné lokálně provedených změn v souborech, které zatím nebyly odeslány. Dále jsou zde informace o souborech, které nejsou aktuální vůči poslední revizi repositáře.

5.2.5 Hook skripty

Běžnou funkcionalitu systému SVN je možné obohatit o spouštění libovolných programů na základě probíhajících akcí v rámci SVN, jak na straně serveru, tak na straně některých klientů. Díky tzv. *hook* skriptům je možné automaticky provádět jakékoliv akce, spouštěné na základě uživatelských událostí (například odeslání souborů na server). SVN poskytuje dostatečnou škálu událostí, které jednotlivé *hook* skripty spouštějí. Několik nejpoužívanějších typů *hook* skriptů:

- `pre-commit` – je spouštěn před samotným odesláním pracovní verze na serverovou část. Přiklad použití – kontrola vyplnění zprávy popisující provedené změny.
- `post-commit` – spouštěny po provedení operace *commit*. Užitečný například pro vytváření kopie obsahu repositáře v jiném repositáři či aktualizaci dat na jiném běžícím zařízení.
- `pre-lock` – spouštěný před operací *lock*. Může sloužit například pro nastavování atributů pro odemčení.
- `pre-revprop-change` – spouštěný v momentě, kdy jsou přidány, upraveny či odebrány popisující atributy revize. V případě potřeby změny zprávy popisující provedené změny při operaci *commit* je možné použít tento skript.

SVN poskytuje několik dalších variant *hook* skriptů. Jejich názvy analogicky popisují události, při kterých jsou spouštěny.

Tyto *hook* skripty se nacházejí v adresáři `hooks` v příslušném repozitáři na serveru. Popisy jednotlivých spouštěcích událostí, jsou ve většině případů popsány v souborech se šablonami v adresáři `hooks`. Dále jsou zde také vypsány parametry, které jsou při odlišných událostech přístupné při spouštění daného *hook* skriptu.

Pro úspěšné uplatnění *hook* skriptu musí mít daný skript jeden z nabízených názvů a musí se jednat o spustitelný soubor. Pro napsání *hook* skriptu s požadovanou funkcionalitou je tedy možná volba jakéhokoliv programovacího jazyka, ze kterého je možné zkompilovat spustitelný soubor.

Tato podkapitola byla zpracována podle zdroje [6]

5.3 Klientské programy SVN

Pro používání SVN na pracovních stanicích existuje velké množství klientských aplikací. Jednotlivé aplikace se pak liší podporovaným operačním systémem, licencemi, podporovanými protokoly a v neposlední řadě šíří poskytovaných funkcí pro práci s SVN.

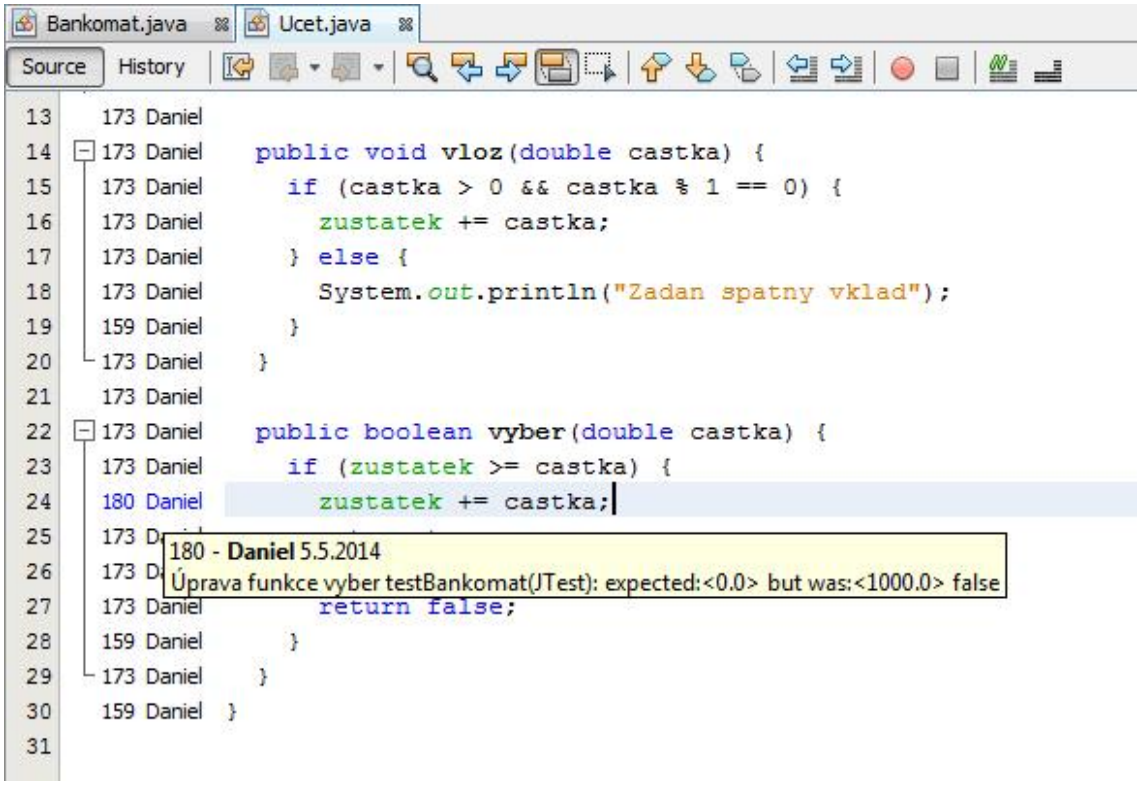
5.3.1 Klient příkazové řádky

Jedním z nejjednodušších klientských programů je klient příkazové řádky. V případě *VisualSVN Server* je dodáván se samotným SVN serverem. Pomocí klienta příkazové řádky je možné vykonávat veškeré operace, které umožňuje SVN. Při práci v příkazové řádce jsou pro jednotlivé operace používány mimo jiné i tyto nástroje: `svn`, `svnadmin`, `svnserve`, `svnsync` a další. Vstupní argumenty je možné zadávat pomocí přepínačů. Podporována je také nápověda, kde je možné naleznout veškeré informace o používání jednotlivých nástrojů.

5.3.2 Vestavěný klient vývojového prostředí NetBeans

Další možností, jak SVN používat je vestavěný klient ve vývojovém prostředí *NetBeans*. Jednotlivé operace jsou přístupné v položce hlavní nabídky `Team`. Před prvním použití je potřeba nastavit správně přístupové parametry k SVN serveru na požadovanou složku.

V nabídce `Team` se zde nacházejí veškeré možné operace (*commit*, *upload*, *checkout*…), spolu s možnostmi, které zobrazí požadované informace přímo v okně s kódem programu. Pomocí možnosti `Show Annotations` je možné zobrazit číslo revize a jejího autora u každého řádku programu. Dále je také možné zobrazit datum vytvoření revize a zprávu, která jí popisuje (viz Obrázek 5).



Obrázek 5 Popisky k jednotlivým revizím v prostředí NetBeans

5.3.3 Další klientské programy

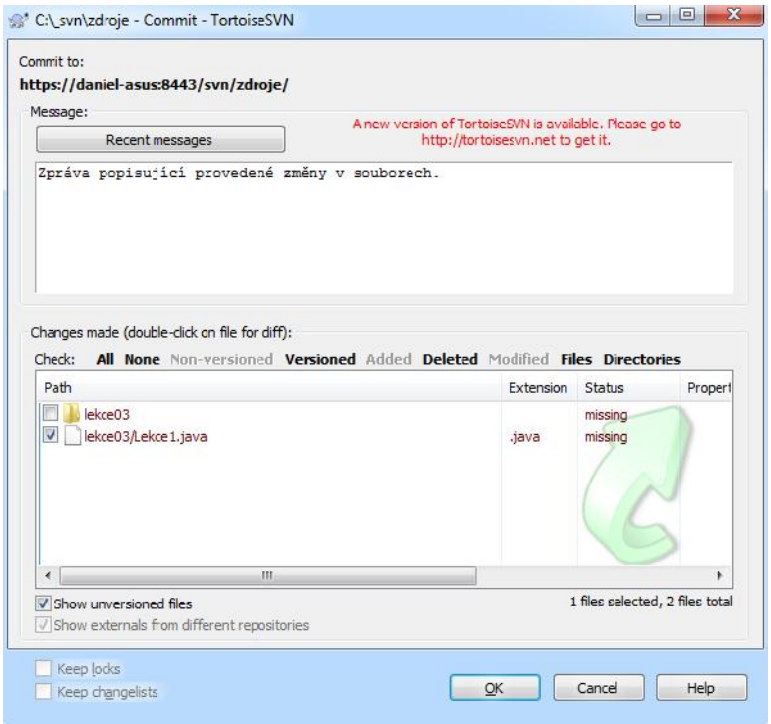
V současné době existuje mnoho programů, které umožňují práci se systémem SVN. Vestavěné klientské programy jsou podporovány například ve vývojových prostředích *Eclipse* a *IntelliJIDEA*.

5.4 TortoiseSVN

Dalším z SVN klientských programů, v tomto případě pro operační systém Windows, je *TortoiseSVN*. Jedná se o nekomerční aplikaci, s intuitivním a jednoduchým grafickým rozhraním, které usnadňuje rutinní obsluhu SVN. První verze byla veřejně vydána již na začátku roku 2003. V současnosti se program nachází ve verzi 1.8.5 a podporuje více než 35 jazykových mutací.

5.4.1 Základní vlastnosti

TortoiseSVN umožňuje jednoduché používání SVN. Veškeré příkazy jsou přístupné přímo z prohlížeče souborů, pomocí kontextového menu. V případě označení souborů nebo složek, jsou pak nabízeny pouze ty příkazy, které v daném případě mají smysl. Pro lepší orientaci stavu jednotlivých souborů a adresářů, integruje *TortoiseSVN* do systému ikony, které překrývají klasické ikony a jednoznačně tak označují aktuální stav – jestli jsou soubory aktualizovány, zda jsou oproti revizi na serveru nové apod.



Obrázek 6 Dialogové okno pro operaci commit

Poskytuje také výkonného správce pro operace *commit* (Obrázek 6). Přehledně se zde dají vybrat všechny potřebné soubory k odeslání. Nabízí také zabudovanou kontrolu pravopisu a automatické dokončování slov, týkajících se názvů odesílaných souborů.

Pomocí nástrojů *TortoiseSVN* je také možné vygenerovat graf, který popisuje jednotlivé revize repositáře. Generovaný graf tak poskytuje jednoduchý přehled historických verzí, souběžně s daty úprav.

5.4.2 Užitečné nástroje

Pro efektivnější spravování a používání SVN nabízí tento klientský program několik užitečných nástrojů.

TortoiseMerge

Nástroj pro zobrazení rozdílných částí pracovní kopie souboru a jeho poslední revize. Poskytuje přehledný pohled na obsahy obou těchto verzí, s barevným označím rozdílných řádků a provedených změn. V případě potřeby je možné obsah souborů upravit a následně uložit. Napomáhá také při řešení vzniklých konfliktů souborů, kdy je nutné manuálně určit, která pozměněná část bude považována za správnou a bude aktualizována.

TortoiseBlame

Používá příkazu `blame` systému SVN a uspořádává jeho výstupy do přehlednějšího formátu. Příkaz `blame` v podstatě slouží k získání informací o provedených změnách v jednotlivých revizích souboru. Spolu s provedenými změnami také poskytuje jméno uživatele, kterým byla daná změna provedena. Nástroj *TortoiseBlame* tyto informace navíc barevně zvýrazňuje změny provedené v rozdílných revizích. Záznamy o změnách souboru doprovází obsah popisující zprávy, vytvořené při odesílání souboru.

TortoiseIDiff

Nástroj pro správu změn provedených v souborech, které nejsou v textové formě, a není je tak možné zpracovat běžnými nástroji pro zobrazení změn. Použití tohoto nástroje může být užitečné například při práci s obrázky, kdy přehledně zobrazuje provedené změny. [7]

6 JUnit

JUnit je jednoduchým rámcovým modelem pro vytváření a používání jednotkových testů, ověřujících funkcionalitu programů psaných v jazyce Java. Jedná se o zástupce z řad *xUnit* modelů pro jednotkové testování.

6.1 Historie

Potřeba opakovaného testování vedoucímu k prověření a dokázání funkčnosti vyvíjeného programu vedla dva vývojáře k vytvoření použitelného modelu pro takové testování. V roce 1997 Erich Gamm a Kent Beck vytvářejí jednoduchý, ale efektivní rámcový model pro testování programů v jazyce Java. Vzniká tzv. *framework* s názvem *JUnit*, kdy základy tvoří zkušenosti z vývoje předchozího *frameworku* pro jazyk *Smalltalk*, vytvořeného Kentem Beckem.

JUnit vznikal jako volný software, publikovaný pod licencí, která umožňuje volné používání i pro komerční účely bez větších omezení. *JUnit* se rychle stal takřka standardním *frameworkem* pro vytváření jednotkových testů v jazyce Java. V dnešní době existuje velké množství modelů postavených na myšlence *xUnit* pro většinu programovacích jazyků.

6.2 Filosofie testování s JUnit

Jak již bylo zmíněno, *JUnit* slouží pro jednotkové testování funkcionalit programu. Jednotkou jako takovou je v případě jazyku Java ve většině případů myšlena metoda.

Testování metody je samozřejmě možné i bez pomoci prostředků *JUnit*, je však méně efektivní. Pokud bychom chtěli testovat metody manuálně, je nutné vytvářet rozhraní pro vstupní data, na jejichž základě by byla metoda testována. Dále by bylo nutné vytvářet chybové výstupy pro případy, kdy se metoda zachová nesprávně. Analýza takových výstupů by v případě jednoduchých metod nebyla nikterak složitá, v případě testování složitějších funkcionalit by však byla o mnoho pracnější.

Pro zjednodušení provádění testování a jeho případné automatizace, je tedy vhodné využít prostředků *JUnit*. Odpadá zde nutnost vytváření rozhraní pro vstupní parametry, spouštěný test bude sám o sobě volat prověřovanou metodu s požadovanými parametry. [8]

6.3 Hlavní prvky rozhraní JUnit

Pro provádění testování pomocí *JUnit* je poskytnuto obsáhlé rozhraní pro programování testů. Rozhraní obsahuje mimo jiné velké množství tříd pro vytváření potřebných funkcionalit programovaných testů. Nejdůležitější základní třídy jsou v balíčku `junit.framework`.

6.3.1 Třída Assert

Metody třídy `Assert` jsou používány zejména pro porovnávání výstupních hodnot testovaných metod. Jedná se o metody bez návratové hodnoty, jejich výsledek však ovlivní celkový výsledek testu. Často používanými metodami jsou:

- `assertEquals` – má dva vstupní parametry, porovnává se pak, jestli jsou oba stejné. Prvním parametrem je očekávaná hodnota, druhým pak hodnota testovaná.
- `assertFalse` – jediným vstupním parametrem této metody je podmínkový výraz. Porovnávána je poté nepravdivost výrazu.
- `assertNotNull` – vstupním parametrem je jakýkoliv potomek třídy `Object`. Testuje se, zda hodnota vstupního parametru není `NULL`.
- `fail` – metoda pro ukončení testu se záporným výsledkem bez jakékoliv zprávy.

6.3.2 Třída TestCase

Tato třída umožňuje spouštění více testovacích metod v rámci jednoho testu. Disponuje taktéž několika standardními metodami, které poskytují informace o stavu probíhajícího testu. Jedná se ve většině případů o bezparametrické metody, z nichž hlavní jsou:

- `countTestCases` – metoda, jejíž návratovou hodnotou je počet spouštěných testovacích případů.
- `createResult` – slouží k vytvoření objektu třídy `TestResult`.
- `run` – metoda pro spuštění testu, výsledky testu jsou sbírány a vráceny objektem `TestResult`
- metody `setUp` a `tearDown` – připravují, případně ukončují prostředky potřebné pro běh testu. Příkladem takového prostředku může být připojení k síti.

6.3.3 Třída TestResult

Údaje z průběhu testů a jejich výsledky jsou shromažďovány v objektech třídy `TestResult`. Rozlišuje se zde mezi dvěma úrovněmi chyb. První z nich jsou selhání (*failure*), která jsou očekávaná, a jejich výskyt signalizuje nekorektní chování testované funkce. Druhým typem chyb jsou neočekávané chyby – například při přetečení pole apod. Pro práci s objektem třídy `TestResult` je zde několik metod:

- `addError` – přidává chybu předanou vstupním parametrem do seznamu chyb u příslušného testu.
- `addFailure` – obdobně jako `addError`, přidává poskytnuté selhání do seznamu selhání.
- `endTest` – informuje objekt typu `TestResult` o ukončení probíhajícího testu.
- `failureCount` – návratovou hodnotou této metody je počet zaznamenaných selhání.

6.3.4 Třída TestSuite

Tato třída zapouzdřuje více testů do jedné soustavy. Místo spouštění testů jednotlivě můžeme použít právě třídy `TestSuite`, do které jednotlivé testy přidáme a následně spustíme jako celek. Nejdůležitější metody třídy `TestSuite`:

- `addTest` – přidá test poskytnutý ve vstupním parametru do soustavy testů.
- `countTestCases` – vrací počet testovacích případů, které budou spuštěny
- `testAt` – návratovou hodnotou je objekt třídy Test na poskytnutém indexu.
- `testCount` – vrací počet testů, které jsou v soustavě testů.
- `run` – spuštění soustavy testů.

Tato podkapitola byla zpracována podle zdroje [9]

6.4 Použití JUnit

Testování pomocí *JUnit* je možné uskutečnit několika způsoby. V moderních vývojových prostředích, jako jsou *NetBeans*, *Eclipse*, *JCreator* a dalších je *framework JUnit* buď přímo implementovaný, nebo je možné jej přidat prostřednictvím rozšíření. V případě vývojového prostředí *NetBeans* je zde přístupný i grafické zpracování výsledků spouštěných testů.

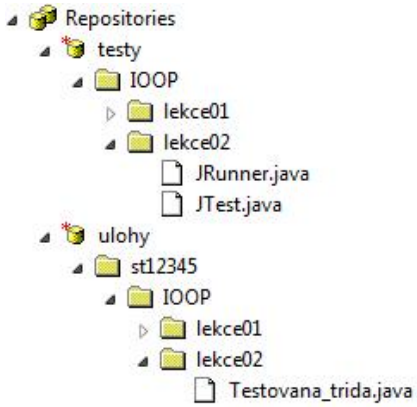
V případě potřeby vkládání výstupů testů v textové podobě do souborů je možné použití *JUnit* na nižší úrovni, a to pomocí spouštění samotných testů například pomocí příkazové řádky. Příklad takového testu je uveden v Příloze D.

7 Praktická část

Cílem praktické části je navrhnout a implementovat systém, který by umožňoval automatické testování odevzdávaných úloh v jazyce Java do repositáře SVN. Akcí, která spuštění testů vyvolá, je událost odeslání souborů programu pomocí SVN. Výsledky spouštěných testů jsou pak v textové podobě přidány do zprávy popisující odesílané soubory.

7.1 Návrh adresářové struktury

Pro potřeby automatického spouštění testů pro příslušné úlohy byla navržena následující struktura repositářů a adresářů:



Obrázek 7 Struktura repositářů a adresářů (výpis z VisualSVN Server)

V repositáři `testy` budou pak v adresářích souvisejících s jednotlivými lekcemi umístěny příslušné testy. Spolu s testy zde bude také spouštěč, zajišťující spuštění všech požadovaných testů. Tento repositář bude běžnému uživateli nepřístupný.

Běžnému uživateli (studentovi) přístupný pak bude pouze repositář `ulohy` a v něm složka s jeho identifikátorem. V této složce budou podobně organizované jednotlivé složky pro rozdílné lekce. Na základě adresáře, ze kterého jsou soubory odesílány, bude následně rozhodováno, jaké testy se budou spouštět.

Tato struktura není jedinou možnou, ale byla vyhodnocena jako nejoptimálnější, vzhledem k další implementaci automatického spouštění testů. Další možností by mohlo být uchovávání testů neorganizovaně v jednom adresáři a rozhodovat o výběru konkrétního testu na základě jeho názvu. Tato možnost by však byla velice omezující vzhledem k různorodosti testů, činila by taktéž vybírání konkrétního testu obtížnějším a méně jednoznačným.

7.2 Spouštění připravených testů

Požadovanou akcí, která bude spouštět připravené testy je odeslání souborů s programovým kódem do repositáře pomoci SVN. Pro zajištění spouštění připravených testů byly vytvořeny *hook* skripty, které budou aktivovány právě v požadované momenty.

Pro vytvoření potřebných *hook* skriptů byla vybrána z důvodů zdánlivé jednoduchosti cesta dávkových souborů. Dalším důvodem je operační systém Windows, který je operačním systémem na zařízení, na kterém bude aplikace uplatněna.

7.2.1 Skript Pre-commit

Před samotným odesláním souborů s programem je nejdříve kontrolována přípona všech odesílaných souborů. K vykonání kontroly je využito zmíněného `pre-commit` *hook* skriptu, jenž se spouští v momentě, kdy je provedena kompletace souborů k odeslání a odeslání požadavku na operaci *commit*. Po provedení tohoto skriptu následuje samotná operace nahrání a vytvoření nové revize repositáře na serverové straně.

SVN poskytuje při provádění tohoto skriptu dva argumenty. Jedním z nich je adresa repositáře, do kterého se mají soubory posílat. Druhým je pak jméno transakce, která právě probíhá. Nástroj `svnlook changed` poskytuje seznam adres souborů, které byly při dané transakci v zadaném repositáři změněny. Spolu s adresami souborů vypisuje taktéž znak, případně dva znaky, které určují, o jaký typ změny se jednalo (A – přidaný soubor/složka, U – změna obsahu souboru a další). Pro získání potřebných informací o právě probíhající operaci jako argumenty programu `svnlook changed` tedy použijeme argumenty, které jsou přístupné při provádění `pre-commit` skriptu.

```
10 REM ********Set of temporary directory, possible directories and extensions
11 SET DIRS=lekce01;lekce02;lekce05;
12 SET EXTS=.java .jpg
13 SET TEMP_DIR=C:\_svn_data\
14
15 CD %TEMP_DIR%
16 SET CHECK=
17 REM ********Check all committed files
18 svnlook changed -t %REV% %REPOS%>%TEMP_DIR%tmpFile
19
20 FOR /f "delims=*" %%f IN (%TEMP_DIR%tmpFile) DO (
21 REM ********Check for allowed extensions
22 SET var=!EXTS:%%~xf=!
23 IF NOT "!var!"=="%EXTS%" (
24   SET file_row=%%f
25 REM ********Skip if this is Delete action
26   IF NOT "!file_row:~0,1!"=="D" (
27 REM ********Find folder name
28     SET folder_name=!file_row:*   =!
29     IF "!folder_name!"=="!file_row!" ECHO Bad Commit>&2 &GOTO :err
30     FOR /f "tokens=3delims=/" %%a IN ("!folder_name!") DO SET "folder_name=%%~a"
31   ) ELSE REM ECHO erasing>&2
32 ) ELSE (
33   ECHO You can commit only %EXTS% files.>&2
34   GOTO :err
```

Obrázek 8 Část kódu skriptu kontrolující přípony souborů a jejich rodičovské adresáře

Díky výstupu tohoto nástroje tak můžeme získat celé adresy posílaných souborů. Z adres posílaných souborů je pak možné extrahovat příponu a rodičovský adresář, tedy všechny údaje potřebné pro kontrolu (viz Obrázek 1) před povolením nahrání souborů. Pomocí stavového znaku můžeme z kontroly vyloučit ty soubory, u kterých je prováděna operace odstranění.

Kontrola je prováděna oproti obsahu proměnné, která obsahuje seznam podporovaných přípon. V podstatě se jedná pouze o příponu java a případně o další typy souborů, potřebné ke správnému otestování odesílaných úloh. Tyto přípony je možné kdykoliv do proměnné dopsat pro rozšíření akceptovaných typů.

Tato kontrola je prováděna zejména z důvodu zamezení nestandardního chování v dalších krocích. V případě nálezu souboru, který není uvedený mezi podporovanými je zamezeno provedení operace *commit* a je vypsána chybová hláška popisující situaci.

Po kontrole přípony nahrávaných souborů následuje kontrola rodičovské složky, ze které jsou soubory nahrávány. Umožněn je *commit* pouze souborů z těch složek, které jsou vypsány v iniciační proměnné. Povoleno je také pouze nahrávání souborů jednoho rodičovského adresáře v rámci jedné operace *commit*. Tato kontrola zamezuje dalšímu nestandardnímu jednání v dalších fázích, kdy by pro zasílané soubory nebylo možné najít a určit správné testy.

V případě výskytu chybové události je hláška vypsána na výstup pomocí přesměrování do chybového kanálu `stderr` a běh *hook* skriptu je ukončen s chybovým parametrem. V případě bezchybného průběhu je uskutečněna operace *commit* a soubory jsou uloženy do serverových repositářů.

7.2.2 Skript Post-commit

Pro realizaci automatického spuštění testů je použito *hook* skriptu `post-commit`. Aktivační událostí tohoto typu skriptu je dokončení nahrání posílaných souborů do serverového repositáře.Tento moment je pro spouštění testů vhodný zejména proto, že zasílané soubory budou uloženy na serveru, nehledě na výsledek testu, což umožní odevzdávání i nekompletních úloh.

Při vykonávání tohoto skriptu systém SVN poskytuje tři argumenty. Prvním z nich je název repositáře, ve kterém se právě provedenou akcí *commit* soubory aktualizovaly. Druhým je číslo určující momentální pořadové číslo revize repositáře a třetím je název provedené transakce.

Hook skripty jsou z bezpečnostních důvodů spouštěny s prázdnými proměnnými prostředí. Proměnné prostředí jsou proměnnými operačního systému a obsahují například absolutní cesty vedoucí ke spustitelným souborům. Z tohoto důvodu je nejdříve do proměnné `PATH` uložit cestu do domovského adresáře instalace Java. Do proměnné `CLASSPATH` je pak nutné vypsat cesty k souborům, potřebných při vytváření a spouštění samotných testů *JUnit*.

Po inicializační části nastavení potřebných proměnných je do dočasného souboru uložen výstup z nástroje `svnlook changed`. Z těchto informací je posléze vybrán název adresáře a další potřebné části z adres odesílaných souborů.

Následuje nakopírování všech souborů z cílového adresáře (Obrázek 9 řádek 31). Pro získání názvů souborů je použito nástroje `svn list`, který poskytuje seznam souborů ve složce zadané v parametru. V cyklu jsou pak postupně kopírovány jednotlivé soubory. V těle cyklu je použito nástroje `svnlook cat`. Tento nástroj poskytuje na základě poskytnutého čísla revize, názvu repositáře a názvu požadovaného souboru textový obsah souboru. Přesměrováním výstupu z `svnlook cat` do dočasného adresáře, je vytvářena kopie každého souboru.

Po zkopírování souborů, jsou do dočasné složky zkopírovány také soubory s připravenými testy z příslušné složky dané lekce (Obrázek 9 řádek 42). Minimálně se zde jedná o soubor

```
31 REM ********Copy all files from commited dir into temp dir
32 IF NOT EXIST %TEMP_DIR%!folder_name!%TXN_NAME% MD !folder_name!%TXN_NAME%
33
34 SET file_row=!file_row:%file_name%=!
35 svn list %REPOS_PATH%!file_row!>tmpAllFiles
36
37 FOR /f "delims=*" %%f IN (%TEMP_DIR%tmpAllFiles) DO (
38 svnlook cat -r %REV% %REPOS% !file_row!%%~nxf\>%TEMP_DIR%!folder_name!%TXN_NAME%\%%~nxf
39 )
40
41
42 REM ********Copy test files
43 cd %folder_name%%TXN_NAME%
44 svnlook cat %TEST_REPOS% \IOOP\%folder_name%\JTest.java>%TEMP_DIR%%folder_name%%TXN_NAME%\JTest.java
45 svnlook cat %TEST_REPOS% \IOOP\%folder_name%\JRunner.java>%TEMP_DIR%%folder_name%%TXN_NAME%\JRunner.java
46
47 REM ********Compilation state check "2" - for redirecting stderr output to file
48 javac JTest.java JRunner.java 2>compile_output
49 SET /p COMPILE_OUTPUT=<compile_output
50 IF NOT "%COMPILE_OUTPUT%"=="" (
51   ECHO Compiling has not been successfull. Output is in version log.>&2
52   svnlook log %REPOS%>compile_output
53   ECHO Compile error.>> compile_output
54   ECHO %COMPILE_OUTPUT%>>compile_output
55   svnadmin setlog %REPOS% -r %REV% compile_output
56   EXIT 1
57 )
58
59 REM ********JUnit test run
60 svnlook log %REPOS%>test_output
61 ECHO Test Results:>>test_output
62 java JRunner>>test_output
63 svnadmin setlog %REPOS% -r %REV% test_output
64 ECHO Commit has been successfull, test has runned. Results are in version log.>&2
```

Obrázek 9 Část skriptu zajišťující překopírování souborů, kompilaci a spuštění testů

s testem samotným a dále o soubor se spouštěčem testu.

Následně je spuštěna kompilace souboru testu a souboru programu, který test aplikuje (Obrázek 9 řádek 47). Výstup z chybového kanálu kompilátoru je přesměrován do dočasného souboru. Prázdnost tohoto souboru signalizuje, že kompilace proběhla bez problémů. V opačném případě je vypsána chybová hláška a obsah chybového hlášení z průběhu kompilace je připsán k doprovodné zprávě nahrávaných souborů.

Po úspěšném průběhu kompilace je spuštěn samotný test (Obrázek 9 řádek 60). Jeho výstup popisující průběh a výsledky testu je podobně jako chybové hlášení při kompilaci připsán ke zprávě příslušných souborů.

Na závěr je vyprázdněna složka s používanými dočasnými soubory.

7.2.3 Výstup z průběhu testů

Při používání *hook* skriptů jsou možnosti, jak vypsat jakákoliv výstupní data značně omezené. Dialogové okno popisující průběh operace *commit* má pouze jediný použitelný výstup, a to je výstup chybový. Výpis průběhu testu v podobě hlášení, které je chybového

typu, ale svým obsahem chybové není, protože by se v něm nejednalo o zprávu o selhání operace systému SVN. Z těchto důvodů byl tento způsob zobrazení výstupní zprávy o průběhu testu shledán nevhodným.

Další možnou variantou, jak poskytnout uživateli přístup k výstupu proběhnutých testů by bylo vytvoření nového souboru v repositáři na straně serveru. Vytvářené soubory by však zbytečně rozšiřovaly souborovou strukturu repositáře. Dalším záporem by pak byla nutnost provedení operace update pro získání souboru s výsledky testů. Varianta vytváření souborů s výsledky testů byla také shledána nevhodnou.

Dle výsledků zkoumání, se varianta připsání výsledku testů ke zprávě, která je přístupná ke každé revizi, jeví jako nejvíce vhodnou. K textu této zprávy, vytvořeném uživatelem před odesláním souborů, jsou po provedení testů připsány jejich výsledky. Pro provedení této operace je použito nástroje `svnadmin setlog`. Jako vstupní parametr je vloženo číslo revize a jméno repositáře. Posledním parametrem je soubor, jehož obsah přepíše obsah zprávy související s danou revizí. Pro možnost změny této zprávy je nutné přepsat *hook* skript spouštěný při úpravě vlastností souborů.

7.2.4 Skript Pre-revprop-change

Změna tohoto *hook* skriptu je minimální, ale pro umožnění změny zprávy doprovázející soubory každé revize nutná. Ve standardním nastavení jsou totiž veškeré změny vlastností zakázány. Pro povolení provádění změn stačí jen přepsat tento skript tak, aby byl ukončen s nechybovým signálem ukončení (viz Obrázek 10). Pomocí tohoto *hook* skriptu je také možné kontrolovat konkrétní měněné vlastnosti a například zakázat změnu autora, v tomto případě je však prosté přepsání hodnoty ukončovacího signálu dostatečné.

```
1 @echo off
2 exit /B 0
3
4
5
```

Obrázek 10 Hook skript pre-revprop-change

7.3 Instalace a použití skriptů

Pro aplikování automatického spouštění testů při odeslání souborů na SVN je zapotřebí nejdříve vytvořit zmíněnou strukturu. Základem jsou dva repositáře – `testy` a `ulohy`. V obou repositářích je pak nutné vytvořit zmíněnou adresářovou strukturu (viz kapitola 7.1) se všemi požadovanými složkami jednotlivých lekcí.

Následně je potřeba umístit `pre-commit`, `post-commit` a `pre-revprop-change`, skripty do adresáře `hooks` v adresáři s nastavením repositáře `ulohy`.

Dalším krokem je úprava několika málo proměnných ve skriptu `pre-commit`. První z nich je proměnná `DIRS`, která obsahuje názvy adresářů – lekcí, pro které jsou připraveny testy a které je tedy možné na server SVN odeslat. Další proměnnou, kterou je vhodné upravit je proměnná `EXTS`. Tato proměnná obsahuje seznam přípon souborů, které jsou pro odeslání přijatelné. Poslední proměnnou, kterou je třeba nastavit je proměnná `TEMP_DIR`. Pro provádění vstupní kontroly odesílaných souborů je nutné vytvářet dočasné soubory se zkoumanými informacemi. Tyto soubory se pak vytvářejí ve složce, ke které je uvedena absolutní cesta právě v proměnné `TEMP_DIR` .

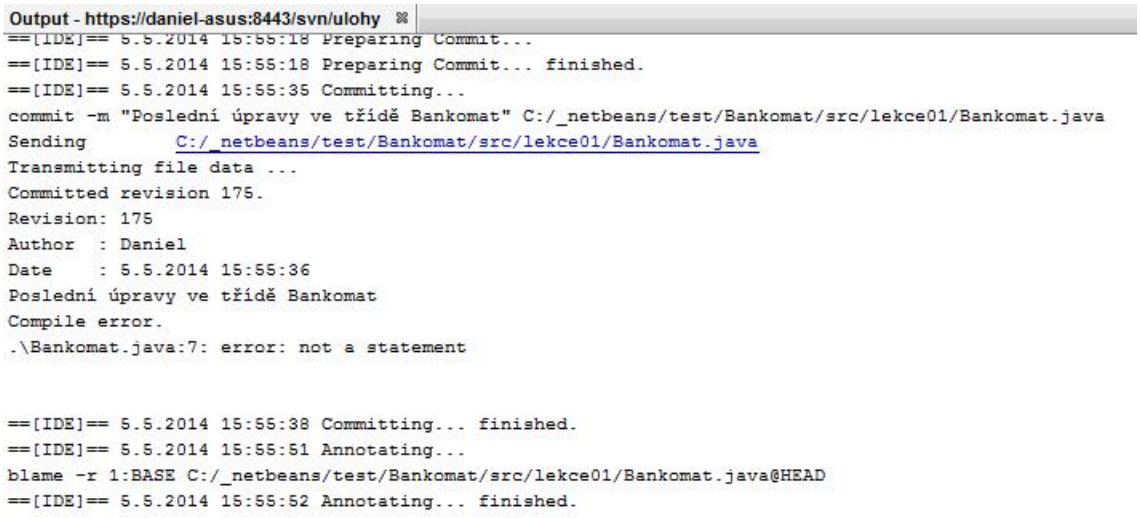
Poslední ze změn, nutných při instalaci skriptů, jsou změny v proměnných ve skriptu `post-commit`. Zde je obdobně jako v předchozím skriptu zapotřebí nastavení proměnné `TEMP_DIR`. *Hook* skripty jsou v rámci SVN z bezpečnostních důvodů spouštěny s prázdnou systémovou proměnnou `PATH`, která obsahuje cesty ke spustitelným programům. Proto je nutné si do této proměnné vypsat absolutní cesty k potřebným programům. Proměnná `CLASSPATH` je využívána při práci s kompilací a spouštění souborů java. Do této proměnné je třeba vypsat absolutní cesty k souborům *JUnit*. Jednou z posledních z používaných proměnných, které je třeba upravit je proměnná `TEST_REPOS`. Obsahuje absolutní cestu k adresáři s nastavením repositáře s testy. Proměnná `REPOS_PATH` pak musí obsahovat adresu repositáře s úlohami studentů.

7.4 Pokyny pro uživatele (studenty)

Pro správné spouštění připravených testů je nutné nejdříve provést operaci `checkout` na složku, která je připravená pro zdrojové soubory. Tuto operaci je možné provést jak pomocí externího klienta (*TortoiseSVN*), tak i pomocí vestavěných klientů vývojových

prostředí (např. *NetBeans*). Cílovou adresou operace `checkout` je pak adresa repositáře `ulohy` doplněná identifikátorem studenta, názvem předmětu a názvem příslušné lekce.

Odesílání vytvořených souborů je možné provádět jak pomocí externích SVN klientů, tak i vestavěných. Na Obrázek 11 je výstup odeslání souboru, s chybou při kompilaci.



Obrázek 11 Odeslání souborů ve vývojovém prostředí NetBeans – chyba při kompilaci

V případě bezchybného průběhu kompilace je spuštěn předem připravený test, jehož výstup je taktéž viditelný ve zprávě o odeslání souborů (Obrázek 12). Po informacích o momentální revizi souborů je ve výstupu vypsána zpráva, která příslušnou revizi doprovází. K této zprávě je připojen výstup průběhu testů.

Obrázek 12 Odeslání souborů ve vývojovém prostředí NetBeans – výstup testu

8 Závěr

V této práci byly popsány jednotlivé metodiky vývoje programů od těch nejjednodušších, které však pro určité typy programů mohou být dostatečné, po složitější, zahrnující vícečlenný vývojový tým, ve kterém jsou určeny role a zodpovědnosti každého člena. Dále pak byly popsány rozdílné typy testování spolu s nástroji pro testování.

V praktické části byla úspěšně sestavena funkcionalita zajišťující automatické spouštění testů programů psaných v jazyce Java. Samotná kódová část skriptů, které se starají o spouštění testů, není nikterak obsáhlá, avšak nalezení optimálního postupu při vytváření těchto skriptů bylo složitější. Důvodem výskytu dalších složitostí, byla volba skriptovacího jazyku příkazové řádky v systému Windows, ve kterém jsou skripty napsány. Tato možnost byla zvolena zejména z toho důvodu, že prostředím na které mají být skripty aplikovány, je právě operační systém Windows. Jedním z problémů byla skutečnost, že většina příkladů skriptů, které bylo možné nalézt, byla vytvářena pro systémy typu Linux. Dalším z obtíží byla také má neznalost příkazů a složitosti ve struktuře jednotlivých příkazů. Po překonání těchto obtíží byla nakonec vytvořena požadovaná funkcionalita automatického spouštění testů pro úlohy v jazyce Java. Instalace vytvořených skriptů není nikterak složitá, jde pouze o nakopírování skriptů do určené složky a případně další menší úpravy pro nastavení povolených přípon a adresářů.

Literatura

[1] PATTON, Ron. *Testování softwaru*. Praha: Computer Press, 2002, Programování. ISBN 80-722-6636-5.

[2] AICHINGER, Michal. *Prezentace aplikování metodiky Scrum ve firmě Skype.* [cit. 2014-04-27].

[3] FERNANDES, Eduardo. SCRUM explained. In: *CodeProject* [online]. © 1999-2014 [cit. 2014-04-26]. Dostupné z: http://www.codeproject.com/Articles/704720/SCRUM-explained

[4] WELLS, Don. *Extreme Programming* [online]. 2013 [cit. 2014-04-27]. Dostupné z: http://www.extremeprogramming.org

[5] KUČERA, František. Distribuované verzovací systémy: Úvod. In: *AbcLinuxu* [online]. © 1999-2013 [cit. 2014-04-27]. Dostupné z: http://www.abclinuxu.cz/clanky/distribuovane-verzovaci-systemy-uvod-1

[6] PILATO, C, Brian FITZPATRICK a Ben COLLINS-SUSSMAN. *Version control with subversion* [online]. 2nd ed. Beijing: O'Reilly, 2008, [cit. 2014-04-27]. ISBN 978-0-596-51033-6. Dostupné z: http://svnbook.red-bean.com/en/1.7/svn-book.pdf

[7] THE TORTOISESVN TEAM. *TortoiseSVN* [online]. ©2004-2014 [cit. 2014-04-27]. Dostupné z: http://tortoisesvn.net/

[8] MASSOL, Vincent a Ted HUSTED. *JUnit in action*. Greenwich, Conn.: Manning, ©2004, ISBN 19-301-1099-5.

[9] JUnit API. In: TUTORIALSPOINT. *Tutorialspoint* [online]. © 2014 [cit. 2014-04-27]. Dostupné z: http://www.tutorialspoint.com/junit/junit_api.htm

Obsah přiloženého CD

Přiložené CD obsahuje:

- tento dokument v pdf formátu,
- soubory vytvořených *hook* skriptů
- příklad možného *JUnit* testu
- soubor se spouštěčem tohoto testu

Přílohy

Příloha A *Skript pre-commit*

```
@ECHO OFF & SETLOCAL ENABLEDELAYEDEXPANSION
REM ********PRE-COMMIT hook that check extensions and parent
REM ********directory of commited files


REM ********Arguments supplied by Subversion
SET REPOS=%1
SET REV=%2

REM ********Set of temporary directory, possible directories and extensions
SET DIRS=lekce01;lekce02;lekce05;
SET EXTS=.java .jpg
SET TEMP_DIR=C:\_svn_data\

CD %TEMP_DIR%
SET CHECK=
REM ********Check all committed files
svnlook changed -t %REV% %REPOS%>%TEMP_DIR%tmpFile

FOR /f "delims=*" %%f IN (%TEMP_DIR%tmpFile) DO (
REM ********Check for allowed extensions
SET var=!EXTS:%%~xf=!
IF NOT "!var!"=="%EXTS%" (
  SET file_row=%%f
REM ********Skip if this is Delete action
  IF NOT "!file_row:~0,1!"=="D" (
REM ********Find folder name
    SET folder_name=!file_row:*   =!
    IF "!folder_name!"=="!file_row!" ECHO Bad Commit>&2 &GOTO :err
    FOR /f "tokens=3delims=/" %%a IN ("!folder_name!") DO SET "folder_name=%%~a"
  ) ELSE REM ECHO erasing>&2
) ELSE (
  ECHO You can commit only %EXTS% files.>&2
  GOTO :err
)
)

REM ********Check for no parent directory
IF "%folder_name%"=="" ECHO Bad folder.>&2 & GOTO :err

REM ********Check for supported parent directories
SET var=!DIRS:%folder_name%=!
IF "!var!"=="%DIRS%" (
 ECHO "You are commiting in not supported directory.(%folder_name%)">&2
 GOTO :err
 )


ENDLOCAL
EXIT 0

:err
ENDLOCAL
EXIT 1
```

Příloha B *Skript post-commit*

```
@ECHO OFF & SETLOCAL ENABLEDELAYEDEXPANSION
REM ******** POST-COMMIT hook which compile and run JUnit tests, results are put into log
REM ******** Arguments supplied by Subversion
SET REPOS=%1
SET REV=%2
SET TXN_NAME=%3
REM ******** Directories variables set
SET TEST_REPOS=D:\Repositories\testy
SET REPOS_PATH=https://Daniel-Asus:8443/svn/ulohy/
SET PATH=%PATH%;C:\tools\;C:\tools\jdk1.7.0_17\bin\;
SET CLASSPATH=C:\tools\junit.jar;C:\tools\hamcrest-core-1.3.jar;
SET TEMP_DIR=C:\_svn_data\

CD %TEMP_DIR%
REM ********Get paths from commited files
svnlook changed %REPOS%>%TEMP_DIR%tmpFile
FOR /f "tokens=1delims=*" %%f IN (%TEMP_DIR%tmpFile) DO (
SET file_row=%%f
SET file_row=!file_row:*   =!

REM ********Get file name and folder name
FOR /f "tokens=3delims=/" %%a IN ("!file_row!") DO (
    SET "folder_name=%%~a"
    SET "file_name=%%~nxf"  )
)
REM ********Copy all files from commited dir into temp dir
IF NOT EXIST %TEMP_DIR%!folder_name!%TXN_NAME% MD !folder_name!%TXN_NAME%
SET file_row=!file_row:%file_name%=!
svn list %REPOS_PATH%!file_row!>tmpAllFiles
FOR /f "delims=*" %%f IN (%TEMP_DIR%tmpAllFiles) DO (
svnlook cat -r %REV% %REPOS% !file_row!%%~nxf\>%TEMP_DIR%!folder_name!%TXN_NAME%\%%~nxf)
REM ********Copy test files
cd %folder_name%%TXN_NAME%
svnlook cat %TEST_REPOS%
\IOOP\%folder_name%\JTest.java>%TEMP_DIR%%folder_name%%TXN_NAME%\JTest.java
svnlook cat %TEST_REPOS%
\IOOP\%folder_name%\JRunner.java>%TEMP_DIR%%folder_name%%TXN_NAME%\JRunner.java
REM ********Compilation state check "2" - for redirecting stderr output to file
javac JTest.java JRunner.java 2>compile_output
SET /p COMPILE_OUTPUT=<compile_output
IF NOT "%COMPILE_OUTPUT%"=="" (
  ECHO Compiling has not been successfull. Output is in version log.>&2
  svnlook log %REPOS%>compile_output
  ECHO Compile error.>> compile_output
  ECHO %COMPILE_OUTPUT%>>compile_output
  svnadmin setlog %REPOS% -r %REV% compile_output
  EXIT 1
)
REM ********JUnit test run
svnlook log %REPOS%>test_output
ECHO Test Results:>>test_output
java JRunner>>test_output
svnadmin setlog %REPOS% -r %REV% test_output
ECHO Commit has been successfull, test has runned. Results are in version log.>&2
REM ********Final cleaning...
GOTO :end
:err
CD %TEMP_DIR%
RMDIR /s /q "!folder_name!%TXN_NAME%"
DEL tmpFile
DEL tmpAllFiles
ECHO Error.>&2
ENDLOCAL
EXIT 1
:end
CD %TEMP_DIR%
RMDIR /s /q "!folder_name!%TXN_NAME%"
DEL tmpFile
DEL tmpAllFiles
ENDLOCAL
ECHO Ok.>&2
EXIT 1
```

Příloha C *Skript pre-revprop-change*

```
@echo off
exit /B 0
```

Příloha D *Příklad JUnit testu*

```
import static org.junit.Assert.*;
import org.junit.Test;

public class JTest {
    Bankomat b1, b2;

    public JTest() {
        b1 = new Bankomat();
        b2 = new Bankomat();
    }

    @Test
    public void testBankomat() {
        Ucet u1 = new Ucet();
        u1.vloz(1000);

        assertEquals(0, b1.vyberPenize(1500), 0.0);

        b1.nastavUcet(u1);
        b2.nastavUcet(u1);
        assertEquals (0, b1.vyberPenize(6.50), 0.0);
        assertEquals(600, b1.vyberPenize(600), 0.0);
        assertEquals(400, b2.vyberPenize(400), 0.0);
        assertEquals(0, b1.vyberPenize(1000), 0.0);
    }

    @Test
    public void testBankomatPrazdnost(){
        Ucet u1 = new Ucet();
        u1.vloz(2000);
        b1.nastavUcet(u1);

        b1.vyberPenize(850);
        assertFalse(b1.prazdnyUcet());

    }
}
```