VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

NASAZENÍ KONCEPTU PRŮBĚŽNÉ INTEGRACE V PROJEKTU OPENSCAPE 4000 MANAGER

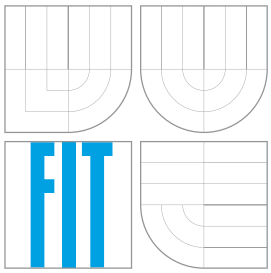
BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE
AUTHOR

PETR JIROUT

BRNO 2014

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

NASAZENÍ KONCEPTU PRŮBĚŽNÉ INTEGRACE V PROJEKTU OPENSCAPE 4000 MANAGER

DEPLOYMENT OF CONTINUOUS INTEGRATION IN SOFTWARE PROJECT

OPENSCAPE 4000 MANAGER

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE
AUTHOR

PETR JIROUT

VEDOUCÍ PRÁCE
SUPERVISOR

Ing. PETR MATOUŠEK, Ph.D.

BRNO 2014

Abstrakt

Tato práce popisuje obecné principy Průběžné integrace. Některé z těchto principů jsou následně implementovány a nasazeny v prostředí reálného projektu OpenScape 4000 Manager, který je vyvíjen společností Unify s.r.o. Vybranými koncepty jsou: zrychlení kompilačního procesu, integrační otestování změn pomocí provedení kompilace a na závěr pak přehledné hlášení výsledků tohoto procesu. Byla provedena analýza stávajícího stavu a poté byl pro podporu nasazení vytvořen nástroj, jehož návrh a implementace je v této práci popsán.

Abstract

This thesis describes the common principles of Continuous Integration process. Some of the principles were implemented and deployed in the existing project, OpenScape 4000 Manager, which is being developed by Unify GmbH & Co. KG. The selected concepts are: speed up the compilation, integration testing of changes via compilation and the process results reporting. An analysis of a current state had been undertaken and a tool had been developed to support process deployment.

Klíčová slova

Průběžná integrace, metodika, softwarové inženýrství, Unify, OpenScape 4000 Manager.

Keywords

Continuous Integration, Methodology, Software Engineering, Unify, OpenScape 4000 Manager.

Citace

Petr Jirout: Nasazení konceptu Průběžné integrace v projektu OpenScape 4000 Manager, bakalářská práce, Brno, FIT VUT v Brně, 2014

Nasazení konceptu Průběžné integrace v projektu OpenScape 4000 Manager

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením pana Ing. Petra Matouška, Ph.D. a konzultanta pana Ing. Petra Jelena.

.....
Petr Jirout
20. května 2014

Poděkování

Chtěl bych poděkovat mému vedoucímu Ing. Petrovi Matouškovi, Ph.D. za rady a návrhy na zlepšení. Dále bych chtěl poděkovat mému konzultantovi Ing. Petrovi Jelenovi, který mě vedl ke zdárnému cíli, za jeho neocenitelné odborné rady. Můj další dík patří všem pracovníkům oddělení R&D 2 brněnské pobočky společnosti Unify s.r.o. za jejich asistenci. V neposlední řadě jsem vděčný všem těm, kteří mě během tvorby této práce jakkoliv podporovali.

Obsah

Kapitola 1

Úvod

V této práci je řešena problematika *Průběžné integrace* (*Continuous Integration*), jakožto praktiky používané při vývoji software.

V dnešní době masivního rozšíření informačních technologií do každého kouta lidské činnosti, kde usnadňují úlohy dříve neproveditelné nebo žádající si značné množství manuální práce, jsou na nabízené produkty v počítačovém odvětví kladeny stále větší nároky a očekávání. K tomu se přidává i rostoucí konkurenci mezi společnostmi a obecný trend zrychlování naší civilizace. Není se potom divit, že roste tlak společností na parametry vytváření software. Ať už jde o efektivitu, přímé a nepřímé náklady či schopnost dodržovat časové termíny.

Po prvních selháních při vývoji „klasickými“ technikami v šedesátých letech minulého století, kdy se hovoří o tzv. „softwarové krizi“ [5, kap. 1.7], vznikl kompletně nový obor softwarového inženýrství, který měl tyto problémy řešit. Tento obor přispěl k zavedení osvědčených postupů pro řešení opakujících se problémů, ať už z hlediska architektury programů (návrhové vzory) či z pohledu vývoje software (nové metodiky pro vývoj, údržbu či testování). Na přelomu milénia pak s příchodem technik *Extrémního programování* [1], které dávají důraz na maximální efektivitu (ekonomickou, časovou, kvality), tento trend ještě zesílil. *Průběžnou integraci* lze pak vnímat jako pokračovatele, který má pozvednout výše zmiňované aspekty na vyšší úroveň.

1.1 Cíl a postup práce

Cílem projektu je začlenění vybraných částí konceptu *Průběžné integrace* do vývojového projektu *OpenScape 4000 Manager*. Tyto části slouží k zefektivnění a urychlení vývojových procesů, především v oblasti programování a zahrnování nové funkcionality do stávajících systémů. Přínos vede k ušetření nákladů jak přímo při vývoji, tak při následném testování.

Důraz práce je kladen na tři následující složky: zrychlení kompilace a sestavení produktu, integrační testování z hlediska možnosti sestavení (kompilace) a prezentace výsledků těchto procesů.

Složka první — zrychlení kompilace a sestavení produktu — je dosažena pomocí vytvořeného analyzátoru závislostí mezi soubory, který umožňuje spustit sestavení pouze nezbytně nutných částí komponent. Tato možnost v projektu doposud chyběla, z důvodu dlouhé doby vývoje a složitých závislostí mezi komponentami.

Část druhá — integrační testování — spočívá v ověření správnosti provedených změn zdrojové základny projektu pomocí testu úspěšného přeložení a sestavení dotčených kom-

ponent. Jakmile je tento test úspěšný, je možné prohlásit, že nové změny nezanesly do projektu chybu.

Poslední část — prezentace výsledků — poskytuje přehledné a dostatečně podrobné rozhraní pro efektivní přehled a práci s výsledky integračního testování. Umožňuje vývojářovi rychle reagovat na nastalou situaci, ať už tato reakce spočívá v opravě kódu nebo možnosti přejít na jiný úkol s vědomím, že dokončená práce neobsahuje chyby.

1.2 Členění práce

V první části této práce (kapitola 2) je nastíněn obecný koncept problematiky *Průběžné integrace*. Ve druhé části (kapitola 3) je popsán výchozí stav projektu z hlediska aspektů důležitých pro tuto práci. Následující část (kapitola 4) popisuje návrh a implementace softwarové podpory pro vybrané aspekty *Průběžné integrace*. Poslední část (kapitola 5) se zabývá konceptem a průběhem nasazení vyvinutého nástroje do reálného prostředí projektu *OpenScape 4000 Manager*.

1.3 OpenScape 4000 Manager

Produkt *OpenScape 4000* (dříve *HiPath 4000*) je komunikační platforma vhodná pro společnosti od 300 do 100 000 zaměstnanců. Je vyvíjen společností *Unify* (dříve *Siemens Enterprise Communications*), která zajišťuje integrovaná komunikační řešení pro cca 75% společností z *Global 500*[1]. Tento produkt nabízí širokou škálu komunikačních služeb, od sdílené pracovní plochy, přes hlasové a video konference, až po telefonování skrze VoIP nebo analogové přístroje.

OpenScape 4000 Manager pak zajišťuje správu, konfiguraci a administraci veškerých částí systému *OpenScape 4000*. Produkt je vyvíjen již 13 let — od verze 1.0 k současné verzi 7.0. Nejprve byl postaven na operačním systému ReliantUnix, poté na UnixWare a nyní využívá 64-bit SUSE Linux Enterprise Server. Kódová základna obsahuje přes 10 miliónů řádek zdrojového kódu, převážně v jazycích C/C++, Java, Perl a shell skript.

Během dlouholetého vývoje se na produktu podílelo přes 100 vývojářů z více než 14 zemí světa. V současné době je vývoj koncentrován v České republice (Brno), Německu (Mnichov), Maďarsku (Brašov) a Turecku (Ankara). Produkt *OpenScape 4000* je nyní nainstalován na přibližně 40 tisících systémech ve více než 80 zemích světa, kde spravuje přes 25 miliónů portů (komunikačních zařízení).

Právě na projektu *OpenScape 4000 Manager* bude nasazován koncept *Průběžné integrace*. Tento projekt je vyvíjen v brněnské pobočce společnosti *Unify s.r.o.*

[1] Viz http://www.unify.com/us/~/media/internet-2012/about/Facts-Unify-2014.pdf

Kapitola 2

Průběžná integrace

V této kapitole je popsán obecný koncept *Průběžné integrace* z pohledu vývoje software. Ačkoliv by bylo možné tento princip zobecnit pro využití v jiných oblastech, nebudeme tak v této práci činit.

O praktikách *Průběžné integrace* jakožto celku se poprvé zmiňuje Kent Beck v rámci svého konceptu *Extrémního programování* [1]. Dalším milníkem byl článek Martina Fowlera [3], který sestavil používané principy a představil v rámci metodiky.

Základním cílem *Průběžné integrace* je minimalizovat náklady, ať už se jedná o zdroje lidské, finanční nebo časové, při vytváření software. Zaměřuje se na jednu specifickou fázi vývoje — fázi integrační. Integrace znamená zahrnování změn (kupř. nové funkcionality, opravy chyb) do již existujícího systému. Systém nemusí být dostupný pro použití zákazníkem, může se nacházet stále v počáteční fázi vývoje. Zjednodušeně řečeno, jakmile vznikne první soubor se zdrojovým kódem programu a položí tak základ systému, jakákoliv další přidaná řádka do programu již prochází procesem *integrace*. Složitost tohoto procesu je vzrůstající s velikostí celého projektu — jak z hlediska počtu vývojářů, tak z pohledu obsáhlosti zdrojového kódu. Od určité velikosti pak přestává být udržitelná tzv. „integrace na koleně“, kdy každý vývojář musí manuálně kontrolovat dopady každé změny.

Mezi dalšími benefity uvádí Duvall [2, s. 29-32] například redukci rizika, omezení opakujících se manuálních procesů či větší důvěru vývojářů ve vyvíjený projekt.

2.1 Problémy integrace

Některé obecné metodiky vývoje software (například vodopádový model) staví integrační část až za implementaci jednotlivých komponent, kdy se odděleně vyvíjejí jednotlivé součásti systému a poté, v rámci integrace, se dávají dohromady. Je zjevné, že tento postup je kriticky závislý na dostatečně kvalitní a hlavně neměnné definici všech rozhraní mezi jednotlivými komponentami, které se pak v průběhu implementace nesmí již jakkoliv upravovat.

Velice často se však v průběhu vytváření systémů mění požadavky na jeho vlastnosti, funkcionalitu a chování, což nutně přináší potřebu změny v návrhu. Jakmile je ukončena po četných návrhových změnách fáze implementace, většinou jsou z původních dohodnutých rozhraní pouze trosky (obecně je tento jev značně závislý na kvalitě analýze požadavků a návrhu systému). Tehdy je třeba změnit již „dokončené“ komponenty a začít je upravovat tak, aby byly schopny fungovat v rámci nových rozhraní. Problémem nemusí být pouze použitá rozhraní, ale i například očekávané chování samotných komponent, kdy se například přesune zodpovědnost za autentizaci uživatele do jiné části, než původní návrh komponenty

počítal. Jelikož se systém nikdy nesestává pouze z jedné komponenty, tyto změny se přenesou do ostatních částí. Pokud navíc nejsou komponenty dokončeny ve stejnou dobu a někteří vývojáři se již věnují něčemu jinému, ztratíme přehled v této komponentě a bude potřeba „znovu objevovat“ svůj kód. Toto přináší další zdržení.

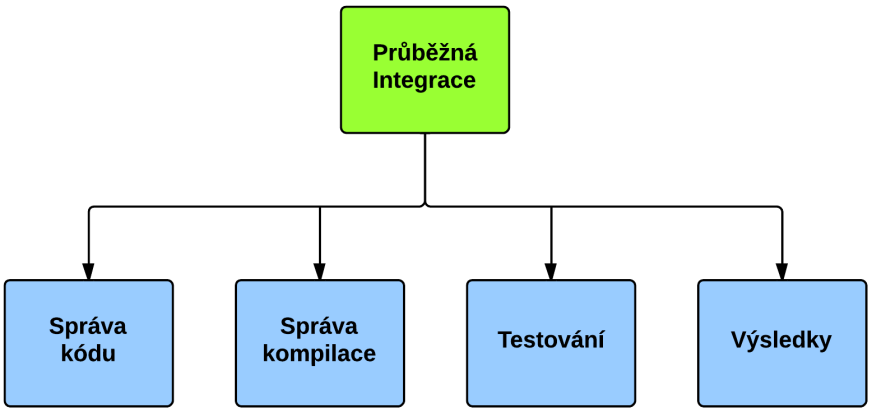
Řešením těchto potíží je provádět integrační fázi již během vývoje, ideálně co nejčastěji. Právě problematikou co možná nejčastějšího integrování se zabývá koncepce *Průběžné integrace*.

2.2 Cíle Průběžné integrace

Účelem *Průběžné integrace* je zmenšit náklady (ať už jsou v jakékoliv formě) vložené do procesu vývoje software. Tohoto se dosahuje pomocí co nejčastějšího integrování s minimální aktivitou požadovanou od vývojáře. Tento požadavek je pochopitelný a vychází z lidské přirozenosti — pokud je po člověku vyžadováno opakované rutinní vykonávání jedné činnosti (byť se zřejmým užitkem), klesá jeho motivace dodržovat tento postup a objevuje se snaha tyto požadavky obejít. Dalším negativním důsledkem je tříštění jeho pozornosti, kdy neustále musí myslet na to, zda už některou část integračního postupu provedl, opakovaně kontrolovat, zda je již integrace dokončena a po jejím dokončení ručně analyzovat její úspěšnost. K tomu navíc musí stále udržovat svoje integrační prostředí v korektním stavu.

2.3 Fáze Průběžné integrace

V této části bude popsáno několik praktik, které dohromady pokrývají problematiku integrace a ověření její úspěšnosti. Tyto fáze a principy vycházejí z článku Martina Fowlera [3] a knihy Paula Duvalla [2]. Samozřejmě není vyžadováno nasazení všech praktik, což se zejména u projektů s dlouhou historií a značným rozsahem ukazuje jako náročný úkol. Je třeba přijmout fakt, že nasazené prostředky nevyužijí plně potenciál tohoto konceptu. Názvy těchto praktik vychází z originálního článku Martina Fowlera [3] a jsou rozčleněny do částí dle oblasti, kterou se zabývají — *správa kódu*, *správa kompilace*, *testování* a *výsledky integrace*. Schéma je možné vidět na obrázku 2.1.



Obrázek 2.1: Součásti *Průběžné integrace*.

Dále v textu budu používat výraz *kompilace* ve významu sestavení spustitelného programu. Toto sestavení obnáší nejen kompilaci objektových souborů, ale i zavedení knihoven a spojení jednotlivých komponent do jednoho fungujícího systému.

2.3.1 Správa kódu

Zde budou popsány části, které se dotýkají správy kódu a principů spojených s prací s repozitářem verzovacího systému.

Udržování jednoho repozitáře

Projekt by měl být celý uložen v jednom repozitáři, aby se zamezilo problémům typu, že se nějaká součást musí stahovat nebo ukládat na nějaké další místo (repozitář). Kdokoliv si může kdykoliv z repozitáře stáhnout zdroje projektu a začít s nimi pracovat. S tím souvisí pravidlo, že by v repozitáři měly být *všechny* zdroje. Jedinou výjimku tvoří části systému příliš velké či komplikované na instalaci, jako je třeba operační systém nebo virtualizační prostředí.

Zároveň by se v repozitáři nemělo objevit nic, co se vytváří při kompilaci (obecně sestavením) projektu. Tento fakt by mohl naznačovat, že nejsme schopni spolehlivě sestavit dané části a proto je máme v repozitáři. Výjimku mohou tvořit pouze neměnné či stabilní komponenty třetích stran, nad jejichž zdrojovými texty nemáme žádnou kontrolu a nekompilujeme je (například knihovna OpenSSL).

Časté ukládání změn do repozitáře

Abychom zamezili hledání integračních problémů v objemných a nově přidaných částech, je zapotřebí, aby každý vývojář ukládal své změny do repozitáře co možná nejčastěji.

Díky tomuto pravidlu jsou vývojáři „nuceni“ plánovat si práci na menší části (aby mohli každý den uložit změny), což umožňuje lepší správu a sledování vývoje projektu.

2.3.2 Správa kompilace

V této části budou rozebrána doporučení pro sestavení a kompilaci projektu. Pro kompilaci je potřeba udržovat si zvláštní stroj (server), který bude mít na starosti správu integrace v tom smyslu, že bude provádět sestavování projektu s cílem ověřit vznik integračních chyb. Pro tento stroj použijeme pojem *integrační stroj*.

Existence integračního stroje usnadňuje používání procesu *Průběžné integrace* tak, že vývojář se nemusí starat o ruční ověřování existence integračních problémů. Tato zodpovědnost je přesunuta na tento stroj, potažmo jeho správce.

Automatizace kompilace

V kapitole 2.3.1 jsme se bavili o pravidelném ukládání změn na integrační stroj, který následně provede ověření, zda integrace byla úspěšná. Pro tuto funkcionalitu je kritické mít k dispozici nástroj, který dokáže *automaticky* provést kompilaci a sestavení celého systému.

Pokud není v možnostech společnosti zajistit kompilaci po každém uložení změn, je často v jejím nejlepším zájmu provádět alespoň nějakou formu pravidelných kompilací (na denní bázi). Se vzrůstajícím intervalem mezi kompilacemi se zvětšuje množství změn zanesených do systému a s tím doba potřebná na opravu případných integračních konfliktů.

Integrační ověření každého balíku změn

Každá *logická* změna, kterou vývojář udělá, by měla projít procesem ověření integrace. Důležité je slovo *logická*, neboť například při opravě chyby v kódu může být potřeba upravit několik zdrojových souborů. Tento „balíček změn“ (samozřejmě se může jednat i o jediný soubor), které nesou jednu logickou změnu, budu nadále nazývat *balíkem změn* (odpovídá anglickému termínu *commit*).

Důležitým aspektem je, aby vývojář před uložením balíku změn vyzkoušel lokální kompilaci na svém stroji. Pokud je tato kompilace úspěšná (kód je tzv. *compile-clean*), je možné tyto změny odeslat na integrační stroj. Pokud by toto vývojář neudělal, integrační stroj by pak sloužil k ověření kompilovatelnosti a nikoliv pro detekci integračních konfliktů. Toto však není cílem *Průběžné integrace*.

Rychlá kompilace

Kompilace a sestavení kompletního systému může u rozsáhlého projektu trvat i několik hodin, což z důvodu požadované reaktivity při ukládání změn vývojářem (který chce vědět co nejdříve, zda jeho kód způsobí integrační konflikt nebo ne) není akceptovatelné. Z toho důvodu mnoho projektů používá „přes den“ (doba, kdy jsou vývojáři v práci) pouze minimální kompilace, kdy se znovu sestaví a zkompilují pouze soubory, které daná změna ovlivnila.

Oproti tomu jednou denně, zpravidla „v noci“ (doba, kdy vývojáři nepracují), se provede kompletní kompilace a sestavení všech souborů a komponent. Nutno podotknout, že při masivním používání sdílených knihoven a „ručních postupů“[1] se otázka minimální kompilace stává netriviální záležitostí, kterou mnohdy dokáže vyřešit pouze změna sestavovacího schématu.

2.3.3 Testování

I pokud je kompilace úspěšně dokončena, nemůžeme prohlásit, že daná změna nezanesla do programu chybu. Mnoho chyb se v programu projeví až za běhu programu — ať už jsou způsobeny například špatnou sémantikou operací, nedokonalou synchronizací při práci se sdílenými prostředky či nedostatečnou kontrolou uživatelského vstupu.

Kontrola úspěšnosti integrace však nemá suplovat roli testovacího oddělení ve společnosti. Není to jejím účelem — naopak to jde z části proti jejímu smyslu, kdy se snažíme mít co nejrychlejší odezvu na provedené změny. Klasický průběh kompletního testování trvá řádově týdny až měsíce v různých fázích (interní testování, cvičný provoz, ...) a z toho důvodu není vhodné pro testovací potřeby *Průběžné integrace*. Naším cílem je provést pouze testování v rámci ověření úspěšnosti integrace, což znamená, že nemáme velké nároky na penetraci testů nebo jejich kvalitu. Naším cílem je pokrýt co nejvíce funkcionality za přiměřenou cenu (testování v řádech jednotek minut). Z tohoto důvodu se v rámci integračního testování testuje pouze základní funkcionalita — například že komponenty odpovídají na dotazy, naslouchají na požadovaných portech a podobně.

Další možností je zavést vícevrstvou architekturu testování. Při této variantě se po úspěšné kompilaci spustí nejprve základní (a rychlé testy), jejichž výsledek se předá vývojářovi. Ten tak v brzké době získá jistotu o tom, že jím provedené změny nenarušují

[1] Příkladem může být kopírování zdrojových souborů v rámci přípravy kompilace, kdy se potřebné hlavičkové soubory ručně kopírují do příslušných adresářů. Ačkoliv tato praktika není z hlediska návrhu ideální, v reálných projektech je často používaná.

funkcionalitu projektu. Po dokončení základních testů se spouští testy komplexnější, které již mohou trvat řádově delší dobu. Opět je třeba zvážit, zda je k dispozici dostatečná kapacita testovacích strojů a času — pokud bychom za den zvládli komplexní testy tří až čtyř balíků změn, je nasnadě spouštět tyto komplexní testy pouze v pevně stanovený čas (několikrát za den) a v rámci testování jednotlivého balíku změn spouštět pouze základní testy.

Automatizace testování

Po dokončení kompilace je záhodno ověřit zachování funkcionality i spuštěním testů, které ověří běhovou korektnost, nikoliv pouze tu kompilační. Jak bylo zmíněno výše, je třeba pečlivě vybírat testy, které se budou spouštět, z hlediska časové náročnosti.

Funkcionalita nemusí být jediným aspektem, který lze testovat. Můžeme například testovat míru shody zapsaného zdrojového kódu s definovanou štábní kulturou, jež může přinést snadnější udržování projektu v budoucnu. Dále lze například provádět skeny kódu pomocí statických analyzátorů[2], které mohou odhalit potenciální chybové stavy programu.

Testování v produkčním prostředí

Pokud spouštíme na integračním stroji i testy, je značně výhodné snažit se na něm udržovat tzv. *produkční prostředí*. To je takové prostředí, ve kterém bude náš systém reálně nasazen (například u zákazníků).Toto nám pomůže v časné detekci chyb, které by byly způsobené rozdílným prostředím a přišlo by se na ně až v rámci komplexního testování nebo až u zákazníka. Zamezíme tím stavu, kdy náš produkt funguje pouze na našem, léty pečlivě ručně vyladěném, stroji.

2.3.4 Výsledky integrace

Jakmile vývojář dokončí a uloží změnu do repozitáře, je záhodno ho informovat o výsledku integračního procesu. Nejčastější způsob je zprávou pomocí e-mailu, který je zaslán buď přímo vývojářovi, který změnu provedl (pokud to integrační systém podporuje) nebo pevně přiřazenému pracovníkovi. Druhý postup v sobě nese tu nevýhodu, že tento pracovník musí případný integrační konflikt analyzovat a předat zodpovědnému vývojářovi.

Další možností je mít tzv. „webovou nástěnku“, kde budou vystavené všechny výsledky — ať už se jedná o výsledek kompilace nebo testování. Na vyžádání jsou pak dostupné podrobnější informace, například soubor s kompilačním logem nebo přehled neúspěšných testů.

Snadno dostupné výsledky kompilace

Pokud je při vývoji software použita metodika, která počítá s častou změnou a upravováním požadavků (kupř. RAD, spirálový model, agilní techniky), je užitečné, aby byla k dispozici aktuální verze produktu ve spustitelné verzi. Tuto verzi můžete využít například při prezentování zákazníkovi. („Ano, tuto funcionalitu jsme implementovali minulý týden.“)

[2]Například nástroj *Coverity*.

Přehled o stavu integrace

Jak bylo řečeno výše, je důležité, aby byl vždy dostupný výsledek integračního sestavení a testování. Každý se poté může snadno podívat, zda může svojí změnu odeslat na integrační server. Pokud je systém v nestabilním stavu (při selhání integrace), nemá smysl se pokoušet integrovat nějakou další část — dokonce by to mohlo přinést další zpoždění. S tím souvisí i pravidlo, že integrační stav systému (kompilace a testování) by měl vždy být v korektním stavu. Pokud tomu tak není, je třeba ho s nejvyšší prioritou uvést opět do fungujícího stavu. Toto pravidlo plyne z logiky věci — nemá cenu vyvíjet nové součásti, pokud ty staré společně nefungují.

Automatizace nasazení

Tato praktika souvisí s požadovanou dostupností poslední vývojové verze ve spustitelné podobě, zmíněné v sekci 2.3.4. Mít spustitelné aplikace rychle k dispozici nebude mít takový užitek (spíše minimální), pokud budeme muset trávit spoustu času instalací a následnou konfigurací systému. Ideálně by tento postup fungoval na dvě kliknutí — jedno by zajistilo stažení aktuální verze, druhé kliknutí by jí následně nainstalovalo a korektně nastavilo.

Kapitola 3

Popis projektu OpenScape 4000 Manager

V této části je popsán počáteční stav prostředí a *Průběžné integrace* v projektu *OpenScape 4000 Manager*.

Ve výchozím stavu projektu je nasazen tzv. *Nightly Build*, což není nic jiného než pravidelná kompletní kompilace a sestavení všech komponent. Jeho provádění je započato každý večer těsně po půlnoci. Pokud je kompilace dokončena úspěšně, jsou spuštěny základní testy, které otestují základní funkčnost. Sestavení zabere cca 2,5 h, testy pak další hodinu. Výsledky tohoto procesu jsou zasílány na mailing list, k jehož odběru jsou přihlášeni všichni vývojáři.

Dále je k dispozici online kompilační log, ze kterého lze extrahovat potřebné detaily, pokud nebyla kompilace úspěšně dokončena. Vždy se provádí kompletní kompilace. Neexistuje kompilace pouze potřebných komponent (*minimální kompilace*) — vždy se provede vyčištění (smazání) všech souborů vytvořených předchozí kompilací.

Kompilace je spouštěna a řízena pomocí shell skriptu, jehož spuštění je naplánováno pomocí `cron`. Lze ho spustit i manuálně na vyžádání, přičemž se vždy provede kompletní kompilace.

3.1 Popis prostředí

V této kapitole bude popsáno softwarové prostředí, v jehož rámci pracuje vývojář a zároveň prostředí, ve kterém funguje výsledné řešení *Průběžné integrace*, jako jsou například pomocné skripty a celý implementovaný systém.

3.1.1 IBM Rational ClearCase

V rámci projektu je používán systém IBM Rational ClearCase® (dále jen *ClearCase*). Je využíván pro správu verzí i pro podporu kompilace. Pro názornější popis vlastností jeho části pro správu verzí ho porovnáme s verzovacím systémem *Git*, který je v současné době jeden z nejpoužívanějších, zejména v Open Source oblasti. Používání *ClearCase* je podmíněno zaplacením licenčních poplatků, nejedná se tudíž o volný software.

ClearCase je narozdíl od *Git* centralizovaný systém s architekturou klient-server, kdy se jednotliví uživatelé připojují k centrálnímu repozitáři, ze kterého si stahují pouze upravované soubory k sobě na stroj, či s nimi pracují vzdáleně na serveru. Oproti tomu *Git* ukládá u každého uživatele kompletní kopii repozitáře včetně historie.

Jakmile chce uživatel systému *ClearCase* upravovat soubor, je mu tento soubor *rezervován* v centrálním repozitáři a žádný další uživatel ho již nemůže upravovat. Nedojde tedy ke konfliktu změn, kdy by dva uživatelé změnili ten samý soubor.

ClearCase nabízí integrovanou podporu kompilace, kdy jeho část `ClearMake` nabízí stejné možnosti jako unixový nástroj `make`, včetně několika rozšíření, jako je například podpora sledování použitých souborů při procesu sestavení.

3.1.2 Prostředí vývojáře

Vývojář se ze svého osobního počítače (sloužící prakticky jako terminál), kde používá operační systém Windows, připojuje pomocí SSH klienta na vzdálené stroje, na kterých probíhá veškerá pracovní činnost a komunikace s repozitářem. Na těchto strojích (nazývaných taktéž „kompilační stroje“) běží operační systém SUSE Linux Enterprise System (dále jen SLES) ve verzích 10 a 11.

3.1.3 Systém Jenkins

Pro koncepci *Průběžné integrace* existují podpůrné nástroje. Nejpokročilejší z nich se nazývají integrační servery, a zahrnují v sobě pomůcky pro zrychlení integračního procesu. Umožňují například automatické spouštění kompilace při detekování nové verze souboru, časově plánované kompilace či spouštění jedné kompilace po dokončení jiné. Jedním z takových nástrojů je i systém Jenkins[1].

V rámci mojí práce nebylo možné podobné nástroje použít z následujících důvodů, neboť tyto nástroje spoléhají na správně napsané makefile (ať už ve formátu `make` nebo například `Ant`). Z důvodu dlouhé historie projektu *OpenScape 4000 Manager* a z důvodu existence rozsáhlé základny zdrojových kódů jsou v současné době závislosti uvedené v makefile neaktuální, neúplné nebo naopak přebytečné. Z toho důvodu se při kompilaci používá zásadně varianta plné kompilace, tzn. vyčištění všech souborů po předchozí kompilaci a následné spuštění nové kompilace a sestavení.

Ačkoliv *Jenkins* nabízí základní podporu *ClearCase* skrze plugin[2], nepřináší žádné další výhody kromě spouštění kompilace a sestavení přímo pomocí `cleartool`. V projektu *OpenScape 4000 Manager* je zapotřebí před samotnou kompilací provést ještě několik konfiguračních kroků a současně je celé schéma sestavení orientováno na *make cíle* (více v části 3.2), nikoliv na automatické hierarchické struktury závislostí, které jsou zapotřebí pro využití *Jenkins* pluginu.

3.2 Kompilace a sestavení

Samotný projekt tvoří celkově 68 komponent. Každá komponenta má ve svém hlavním makefile právě sedm cílů — jejich výčet je uveden v tabulce 3.1. Vzhledem k existenci kruhových závislostí mezi jednotlivými cíli v rámci různých komponent je pořadí kompilace komponent pevně stanoveno (odpovídá pořadí v tabulce, shora dolů) — toto pořadí se historicky ukázalo jako fungující a proto se nemění.

Komponenty jsou rozděleny dvě části — `PLATFORM` a `APPLICATIONS`. Komponenty části `PLATFORM` poskytují podporu a základní služby (komunikace mezi systémy) pro komponenty

[1] Více na: http://jenkins-ci.org/

[2] Viz https://wiki.jenkins-ci.org/display/JENKINS/ClearCase+Plugin

Název cíle	Popis cíle
`export_inc`	Export hlavičkových souborů.
`export_lib`	Export sdílených knihoven.
`export_jlib`	Export sdílených knihoven Java.
`devel`	Kompilace vývojářských prostředků.
`all`	Kompilace funkčních částí.
`javagui`	Kompilace grafického rozhraní Java.
`pkgs`	Vytvoření `RPM` balíčku.

Tabulka 3.1: Seznam `make` cílů

části `APPLICATIONS`, které mají na starosti služby orientované na uživatele (zprostředkovávání informací uživateli atp.). Kompilace v rámci části `PLATFORM` probíhá postupně „po cílech“. V pořadí tabulky 3.1 se postupně provádí jeden cíl pro všechny komponenty a teprve tehdy, je-li daný cíl dokončen pro všechny komponenty, spouští se další cíl. Kupříkladu se tedy nejprve provede cíl `export_inc` a jakmile je dokončen pro všechny komponenty, spouští se cíl `export_lib` opět pro všechny komponenty.

Část `APPLICATIONS` je kompilována po komponentách, kdy se nejprve provedou všechny cíle pro jednu komponentu a až poté se pokračuje na další komponentu.

3.3 Pracovní postup vývojáře

Vývojář většinu času pracuje na zadaném úkolu, ať už je to implementace nové funkcionality nebo oprava chyby. Jakmile dokončí část úkolu, kterou chce uložit do repozitáře, zpravidla provede kompilaci a sestavení celé komponenty nebo pouze provedení jednoho *make cíle*. V některých případech neprovede ani toto. Zde vzniká veliké nebezpečí porušení pravidla o odevzdávání pouze kompilovatelného (*compile-clean*) kódu do repozitáře, což vede k okamžitému uvedení dané komponenty (ba i celého projektu) do nevalidního stavu.

V závislost na výsledku této lokální kompilace (vývojář ji dělá ve svých pracovních adresářích a nikoliv na integračním stroji) poté uloží změny na vzdálený repozitář. Každou noc je pak proveden *Nightly Build* (více v úvodu této kapitoly), který zkompiluje a sestaví všechny komponenty v projektu. Pokud byly v průběhu dne do projektu uloženy změny, které způsobí integrační chyby, jsou detekovány právě během *Nightly Build*. Ráno pak vývojář odpovědný za analýzu logů podá zprávu o chybě zodpovědnému pracovníkovi, který dané změny do repozitáře uložil. Ten ji pak neprodleně opraví.

V závislosti na závažnosti chyby se pak opakuje celá plná kompilace a sestavení nebo kompilace pouze dotčené komponenty. Poté je zpravidla systém nainstalován a jsou spuštěny testy. V případě, že daný problém vyžadoval rekompilaci všech komponent, následnou instalaci a spuštění testů, je celý tento proces dokončen až za čtyři hodiny od jeho počátku. A to pouze za předpokladu, že se neobjeví nové chyby.

Postup integrace změn

1. Dokončení úkolu, je doporučeno provést lokální kompilaci dotčené komponenty.
2. Uložení změn na vzdálený repozitář.
3. Automatické provedení pravidelné kompilace (*Nightly build*).

4. Automatické spuštění testů.

5. Instalace systému.

Pokud není noční kompilace úspěšná (například vznikem integračního konfliktu), musí se provést ruční analýza problému. Jakmile je problém detekován a vyřešen, spouští se celý proces od kroku č. 3.

3.3.1 Zhodnocení potřeby Průběžné integrace

Z výše uvedeného popisu plyne, že chyby zanesené do repozitáře jsou detekovány až následující den od jejího vytvoření. To by samo o sobě nebyl tak závažný problém, ale tato chyba vždy způsobí selhání noční plné kompilace (*Nightly Build*) a následné instalace systému. Důsledek tohoto je pak fakt, že ostatní vývojáři nemohou následující den testovat či jinak pracovat s aktuální verzí projektu. Toto prodlužuje a zdražuje celý vývoj projektu, protože tím, jak je vývojář nucen pracovat delší čas ve svém izolovaném prostředí (protože noční instalace aktuální verze systému je poškozená), přímo úměrně narůstá riziko vzniku dalšího integračního konfliktu.

Cílem této práce je minimalizovat toto riziko za pomoci principů *Průběžné integrace*. Tyto principy budou nasazeny v rámci vývojového oddělení projektu. Nebudou mít tudíž přesah do testovacího či produkčního oddělení. Tento přesah není potřebný z důvodu, že tato oddělení používají delší kontrolní intervaly (tzn. čas, za který se začíná znovu od začátku testovat nová verze projektu). Dalším důvodem je skutečnost, že produkce určité verze spočívá v „zamrazení“ projektu v daném stavu a jeho produkcí. Není zde proto potřeba ověřovat možnost vzniku integračních konfliktů, neboť všechny změny projdou nejdříve přes vývojové oddělení a tudíž i přes vyvinutý nástroj.

Kapitola 4

Návrh a implementace podpory Průběžné integrace

Jelikož je problematika *Průběžné integrace* poměrně rozsáhlá a její nasazení do velkého a komplexního projektu (jakým *OpenScape 4000 Manager* bezesporu je) není triviální, bylo třeba vymezit cíle a prvky *Průběžné integrace*, na které se tato práce zaměří.

Po analýze jsme dospěli k tomu, že primárním cílem by měly být aspekty týkající se kompilace a sestavení (viz část 2.3.2), neboť právě tyto praktiky mají z hlediska efektivity celého procesu největší dopad (především díky úspoře času a tím i nákladů) a proto i největší potenciální přínos.

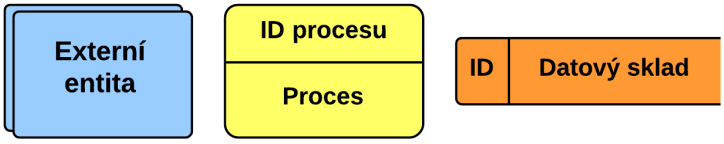
Hlavními uživatelskými požadavky a tudíž i cíli této práce byly následující záležitosti. Důraz byl kladen na co nejmenší zatížení vývojáře z hlediska práce s nástrojem pro podporu *Průběžné integrace*. Z tohoto pramení potřeba pro co největší nezávislost a automatickou funkčnost nástroje. Dále pak je žádoucí vytvoření, kromě webového rozhraní pro práci se změněnými soubory, i rozhraní pro práci z příkazové řádky. Mezi dalšími požadavky pak figurovalo automatické podávání hlášení na e-mail, které bude obsahovat i informace o typu chyby (například část logu nepodařené kompilace), nebo udržování statistik o provedených operacích (kolik akcí bylo úspěšných, kdo má jakou úspěšnost odevzdaných balíků změn atp.).

Požadavky na systém

- Minimální zátěž vývojáře.
- Minimální správa — automatické generování závislostí.
- Automatická notifikace zodpovědných osob (vedoucího, vývojáře).
- Možnost ručně zadat ignorované soubory a autory změn.
- Přehledná webová prezentace výsledků kompilace s dostupnými logy.
- Zrychlení kompilace a sestavení pomocí spouštění pouze nutných *make cílů*.
- Dostupné rozhraní příkazové řádky.

Pro znázornění architektury systému jsou použity diagramy datových toků (*Data Flow Diagram*), které poskytují především přehled o struktuře z hlediska vstupů a výstupů.

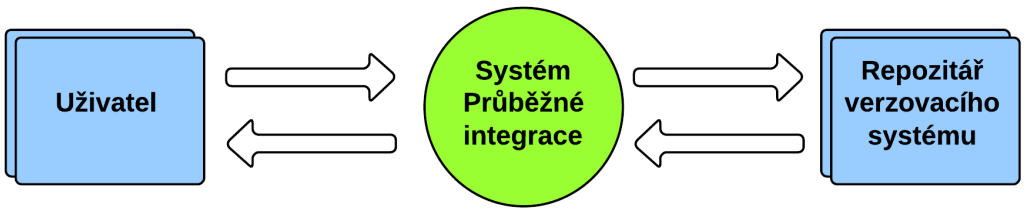
Barvy v diagramech jsou zvoleny účelově a jsou konzistentní skrze celou práci. Jejich ukázka je zobrazena na obrázku 4.1.



Obrázek 4.1: Ukázka použité notace diagramů datových toků.

Světle modrá barva ve spojení s obdélníkem se zdvojenou hranou reprezentuje terminátor, tzn. externí entitu, která stojí vně daného systému a komunikuje s ním. Světle žlutý obdélník se zakulacenými rohy reprezentuje proces, který obstarává změnu vstupů na výstupy. V horní části obdélníku se nachází identifikátor procesu. V tomto identifikátoru je zahrnuta jeho příslušnost a hierarchie v rámci celého systému. Například identifikátor procesu „4.3.2“ znamená, že proces je zahrnut v komponentě *Hlídač repozitáře* (č. 4), v rámci subsystému *Kompilace* (č. 3) a jeho číslo je 2. Oranžový obdélník s jednou neuzavřenou stranou má význam datového skladu, kam se ukládají data pro pozdější využití. Posledním prvkem diagramů jsou pak datové toky, které jsou zachyceny černou šipkou.

Vzhledem k velikosti vytvořeného systému není nutné dokumentovat implementační detaily (např. metody a členské proměnné tříd). Umístění nástroje v rámci prostředí je znázorněno na obrázku 4.2.

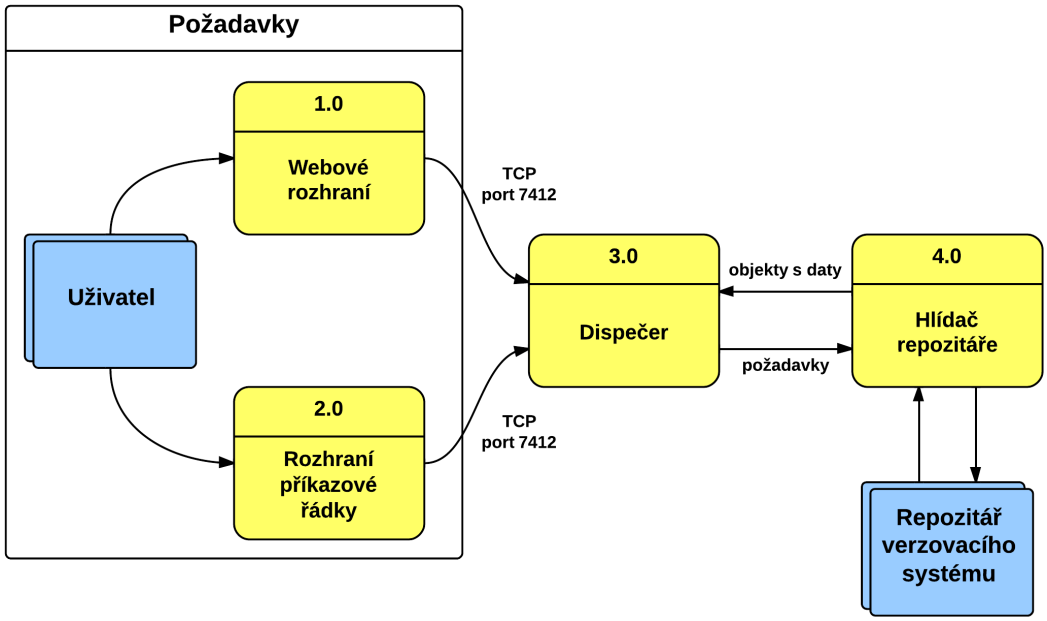


Obrázek 4.2: Kontextový diagram systému.

4.1 Architektura systému

Zde bude popsána celková architektura nástroje, který slouží jakožto aplikační podpora *Průběžné integrace*. První částí nástroje je *Analyzátor závislostí*, který má za úkol získat a uložit závislosti mezi soubory, v rámci jednotlivých komponent. Z těchto informací pak *Hlídač repozitáře* extrahuje potřebné závislosti pro změněné soubory. Tyto soubory zprostředkovává *Dispečer požadavků* buď skrze webové rozhraní nebo skrze CLI (*Command Line Interface*) nástroj. *Dispečer* pracuje jako démon, který naslouchá na TCP portu číslo 7412 a sbírá příchozí požadavky od uživatele, které následně předává dále. Komunikuje jak

s částí zodpovědnou za vytváření webové prezentace, tak i s částí starající se o samotné provedení kompilace a následném hlášení o výsledcích.



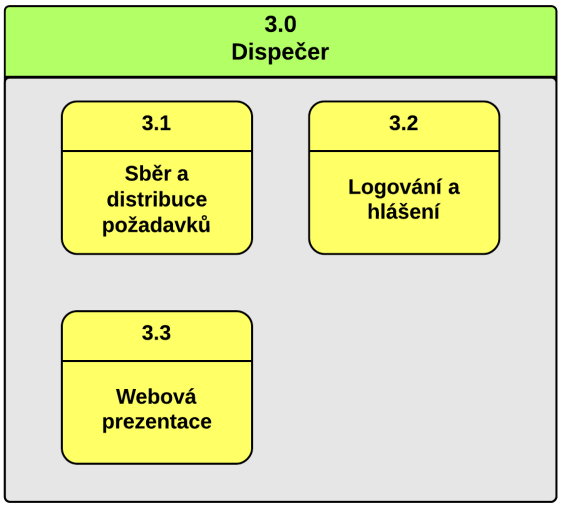
Obrázek 4.3: Schéma architektury systému.

Nástroj je implementován v programovacím jazyce Python[1] verze 2.4 (novější verze není na vývojářském serveru k dispozici). Pro ukládání cache souborů je použit Python modul `pickle`[2], který využívá vlastního formátu serializace a deserializace pro ukládání objektových struktur na disk. Vzhledem k relativně malým velikostem ukládaných objektů (řádově jednotky MB) není třeba používat složitější prostředky, jako například databázový systém *SQLite*.

4.2 Dispečer

Dispečer má na starosti sběr požadavků od uživatele a jejich distribuci k dalším částem systému *Průběžné integrace*. Mezi jeho další zodpovědnosti patří logování a hlášení všech událostí. Dále pak má na starosti generování webové prezentace, kde je zobrazen aktuální stav celého systému. Z důvodu těchto požadavků a záměru centralizování kontroly celého systému byla pro jeho vnitřní implementaci použita variace návrhového vzoru Prostředník (*Mediator*). Tento návrhový vzor zapouzdřuje interakci různorodých objektů pod jedno rozhraní, tudíž můžeme ovládat více částí systému skrze jeden objekt. Tímto chceme docílit zjednodušení vnitřní komunikace mezi jednotlivými částmi, které jsou implementovány množstvím tříd, a sjednocení dostupné funkcinality pod jedno zastřešující rozhraní. Dle Gammy et al. [4, str. 277] použití tohoto návrhového vzoru navíc abstrahuje koncept komunikace jednotlivých částí systému.

[1] Více na: https://www.python.org/
[2] Viz https://docs.python.org/2.6/library/pickle.html#module-pickle



Obrázek 4.4: Přehled komponent Dispečera.

4.2.1 Sběr a distribuce požadavků

Dispečer přijímá požadavky skrze dvě vstupní rozhraní — webové rozhraní a rozhraní příkazové řádky. Jak tato rozhraní vypadají se můžete podívat v příloze C. Tyto požadavky jsou následně analyzovány dle jejich typu a jsou z nich extrahovány potřebné informace. Například je-li požadavek typu `REQ_COMMIT` (tzn. vytvoř ze změněných souborů balík změn), je třeba získat informace o tom, které všechny soubory jsou relevantní vůči požadavku.

Obě rozhraní pracují stejně — vývojář musí označit, které ze všech změněných souborů chce zahrnout do balíku změn, jinými slovy určit soubory, u nichž chce ověřit validitu z hlediska integrace. Poté, co vytvoří tento balík změn, může vidět, jakých všech *make cílů* se daná změna dotýká. Následně spustí pro tento balík kompilaci.

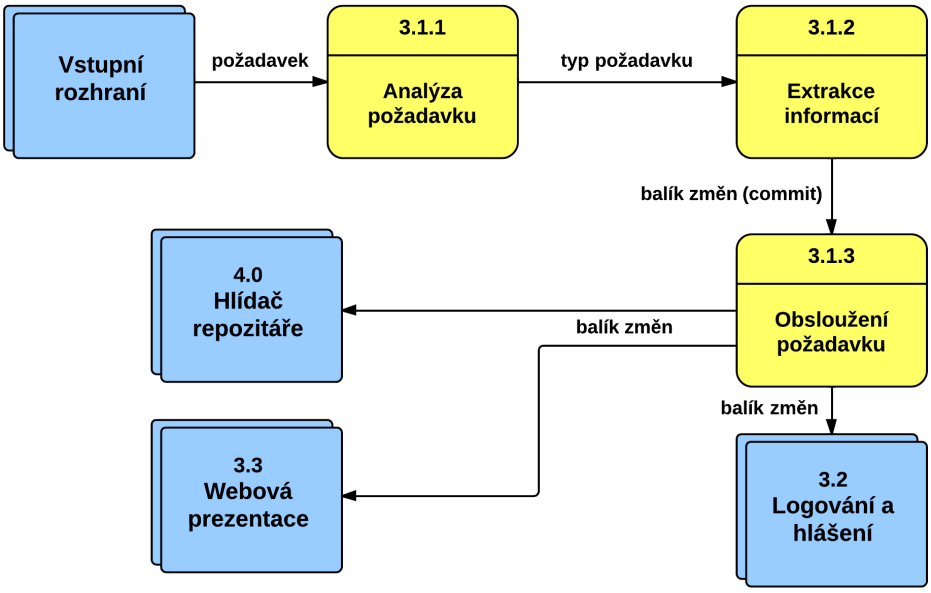
Jakmile je požadavek analyzován a potřebné informace jsou shromážděny, zašle ho odpovídající komponentě — ať už ostatním částem *Dispečera* nebo *Hlídači repozitáře*. Příklad zpracování požadavků je zobrazen na obrázku 4.5.

Sběr požadavků běží nezávisle na ostatních akcích jím vykonávaných (ve vlastním vlákně) a přijaté požadavky jsou ukládány na zásobník požadavků (ošetřeno výlučným přístupem). Tím je zaručeno, že klient bude moci odeslat svůj požadavek i tehdy, je-li systém zaneprázdněn zpracováváním jiného požadavku.

Pokud po odeslání požadavku uživatel zjistí, že odeslal špatný požadavek, je možné tento požadavek odstranit. Toto je možné pouze před tím, než uživatel odeslal příkaz ke zpracování požadavku — v tom případě se špatný požadavek vyhodnotí (zpravidla vyústí v integrační konflikt).

4.2.2 Webová prezentace a hlášení

Další funkcionalitou zastřešovanou *Dispečerem* je tvorba webové prezentace procesu *Průběžné integrace*. V této prezentaci jsou obsaženy informace o změněných, ale ještě nezpracovaných souborech, o balících změn čekajících na kompilaci, stavu aktuálně probíhající kompilace, autoři této změny atd. Dále pak uchovává historii o všech provedených akcích, včetně



Obrázek 4.5: *Dispečer* — příklad zpracování požadavků.

kompilačních logů pro uskutečněné kompilace. Ukázka webové prezentace je k nalezení v příloze C.1.

Informace poskytované webovým rozhraním

- Seznam změněných souborů.
- Seznam a detaily odeslaných balíků změn.
- Historie kompilace balíků změn.
- Logy z průběhu kompilace (výstupy kompilátoru).
- Čas a datum vykonání vše výše uvedeného.

Neméně důležitým úkolem je hlášení o provedených akcích. Výsledek kompilace je ihned po jejím dokončení zaslán zodpovědnému vývojářovi. Na pravidelné bázi (např. týden) jsou zasílány souhrnné statistiky vedoucímu pracovníkovi, který může vyhodnotit úspěšnost integrace za uplynulé období.

Informace obsažené v hlášení o výsledku

- Čas a datum odeslání a zpracování balíku změn.
- Seznam souborů, které byly v daném balíku změn.
- Seznam vykonaných *make cílů* včetně jejich statutu.
- Odkaz na místo, kde je uložen výsledný `RPM` soubor pro danou komponentu.
- Celkový výsledek kompilace a sestavení — v pořádku či selhal.

4.2.3 Rozhraní příkazové řádky

Jeden z požadavků (viz kapitolu 4) byl vytvoření rozhraní příkazové řádky (dále jen CLI). Rozhraní podporuje i omezenou formu zobrazování dat z *Dispečera* (které jinak obstarává hlavně webová prezentace). CLI je navržen jako tenký klient, který všechny informace získává z démona běžícího nad repozitářem (*Dispečer*) a řeší pouze zobrazování těchto informací a odesílání požadavků. V tabulce 4.1 je uveden seznam dostupných přepínačů. Pokud není uveden žádný přepínač, nástroj vypíše seznam neintegrovaných souborů.

Přepínač	Význam
`-h, --help`	Vypsání nápovědy.
`-b ID, --build=ID`	Spuštění integrační kompilace vybraného balíku změn.
`-d ID, --delete=ID`	Odstranění vybraného balíku změn.
`-l, --list`	Vypsání seznamu neintegrovaných souborů. (Výchozí možnost.)
`-s FILE, --submit=FILE`	Vytvoření balíku změn, seznam změn je ve vstupním souboru (FILE). Pokud je FILE `-` (spojovník), je čten standardní vstup.
`-r, --run`	V kombinaci s `-s` spouští integrační kompilaci odeslaného balíku změn.
`-c, --commits`	Vypsání všech balíků změn.
`-u USER, --user=USER`	Zobraz balíky změn či změny pouze od daného uživatele.
`-t SINCE, --time=SINCE`	Zobraz balíky změn či změny pouze od určitého data.
`-q, --queue`	Vypsání fronty požadavků na *Dispečerovi*.

Tabulka 4.1: Přepínače rozhraní příkazové řádky.

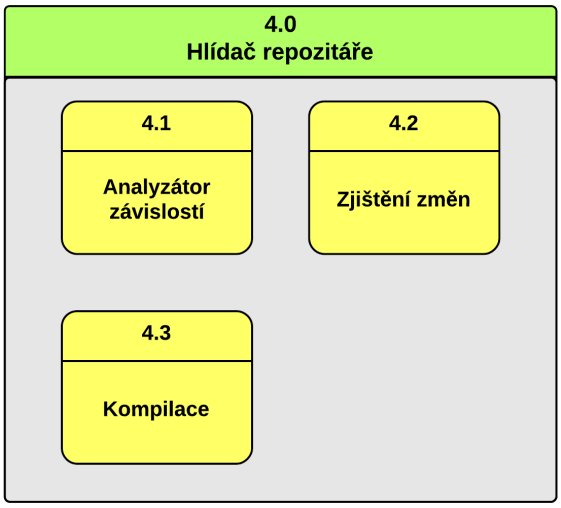
4.3 Hlídač repozitáře

Tato komponenta zodpovídá za veškerou práci s repozitářem verzovacího systému, v našem případě *ClearCase*. Má na starosti analýzu závislostí mezi soubory, zjišťování nových událostí nad repozitářem (nový soubor, nová verze) a samotné spouštění kompilace potřebných cílů. Uživatel s ním přímo nekomunikuje, požadavky jsou mu zasílány (resp. jeho funkcionalita je vyvolávána) skrze *Dispečer*.

4.3.1 Analyzátor závislostí

Pro všechny binární soubory (aplikace), knihovny a objektové soubory se generují jejich závislosti (i vzájemné). Závislosti mohou mohou být přímé (daná aplikace pro svojí kompilaci potřebuje konkrétní soubor) nebo zřetězené (aplikace potřebuje pro kompilaci knihovnu, která se skládá ze souborů — tyto soubory pak tvoří závislosti i původní aplikace). Ukázku závislostí pro konkrétní knihovnu lze nalézt v příloze B.

Hlavní sledované vlastnosti výše zmíněných objektů jsou: *make cíl*, kterým daný objekt vznikl, dále pak komponenta, ke které náleží a v neposlední řadě závislosti na zdrojových souborech, knihovnách a objektových souborech. *Make cíl* znamená, v rámci kterého `makefile` a jeho cíle (*target*, např. `all`, `export_inc`, ...) byl daný objekt vytvořen.



Obrázek 4.6: Model hlídače repozitáře.

Výsledkem analýzy je orientovaný graf mezi objekty (uzly). Hrany reprezentují vztah kompilační závislosti (rodič-potomek), uzly reprezentují sledované objekty (zdrojové soubory, knihovny, binární soubory). Tento graf je implementován pomocí asociativního pole, kdy klíčem je absolutní cesta souboru v repozitáři, tudíž je zaručeno, že bude vždy unikátní. Pro reprezentaci hran (vztahů) mezi uzly má každý prvek ve své datové struktuře obsažen i seznam odkazů na prvky, se kterými je propojen, plus příslušný směr.

Architektura analyzátoru závislostí

Po úspěšném provedení buildu, jehož výsledkem je skupina `RPM` balíčků, jsou tyto balíčky jeden po druhém analyzovány. Z každého je extrahován seznam binárních spustitelných souborů a knihoven. Tento seznam je pravidelně ukládán (jakožto *cache* soubor, viz datový sklad č. 1 na diagramu 4.7) a v případě potřeby je možné získat data přímo z něj, čímž se ušetří čas potřebný pro prohledání `RPM` balíčků.

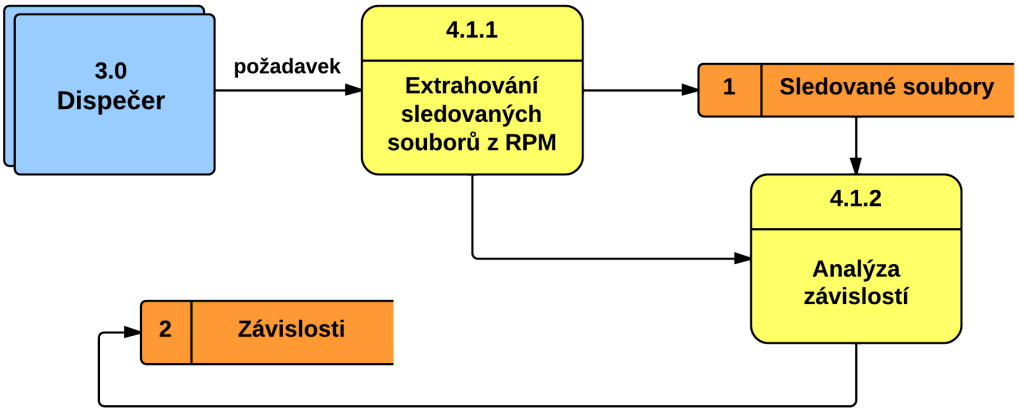
Poté jsou tyto soubory vyhledány v repozitáři (ve kterém musí po kompilaci zůstat, nesmí být smazány). Nyní přichází na řadu využití systému *IBM ClearCase*, který pro každý zkompilovaný objekt pomocí nástroje `clearmake` (jehož jádrem je klasický systém `make`) udržuje ve speciální datové struktuře MVFS[3] přehled objektů, ze kterých byl daný objekt kompilován.

Pro daný objekt jsou tyto závislosti uloženy. Takto jsou postupně analyzovány všechny objekty ze všech `RPM` balíčků. Jakmile je tento proces hotov, pro každý z objektových souborů (nikoliv tedy pro zdrojové či hlavičkové soubory, kde by to nedávalo smysl) jsou z *kompilačního logu* extrahovány informace o *make cíli*, kterým byl daný objekt vytvořen. Všechna tato data jsou přiřazena konkrétních prvkům (nyní již uzlům) v orientovaném grafu, který je implementován pomocí asociativních polí. Každý uzel obsahuje informaci o tom, jakého je typu (objektový soubor, binární soubor, knihovna atp.), jeho ID (ve skutečnosti plná cesta v repozitáři, čímž je zajištěna jeho unikátnost), seznam závislostí (list

[3]IBM Rational MultiVersion File System.

odkazů na uzly) a seznam „rodičů“ (tzn. list uzlů, které jsou na daném prvku závislé) pro snazší implementaci průchodu grafem.

Závěrem jsou všechny závislosti v příslušných datových strukturách uloženy na disk (viz datový sklad č. 2 na diagramu 4.7), čímž je dosaženo značného zrychlení systému, neboť při dotazu na závislosti daných souborů není třeba znovu spouštět analýzu (plná analýza trvá cca 40 minut).



Obrázek 4.7: Architektura analyzátoru závislostí.

4.3.2 Zjištění změn

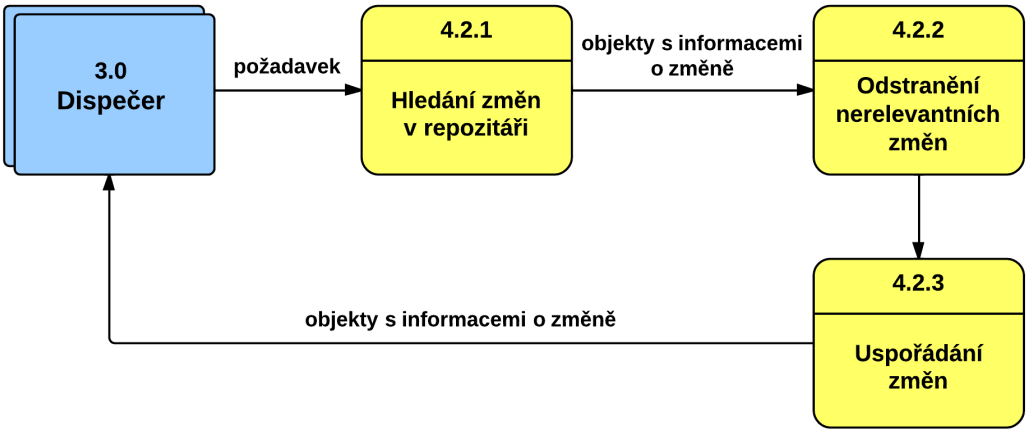
Jelikož nástroj pracuje jako mezivrstva mezi repozitářem a uživatelem, je třeba v nástroji řešit zjišťování změn (nové verze, nové soubory) v repozitáři. Bod, od kterého jsou změny při prvním spuštění zjišťovány, je čas poslední úspěšné kompletní kompilace (všechny komponenty). Jakmile je nástroj spuštěn a jsou načteny všechny změny k aktuálnímu času, je tento seznam změn periodicky aktualizován (každou minutu) pomocí dotazů na změny od data poslední kontroly. Tuto kontrolu lze cíleně vynutit. Změny jsou aktualizovány vždy přírůstkově. Tím se zvýší rychlost, sníží datová zátěž a redundance celého procesu.

Jakmile je přijat požadavek na zjištění změn (buď přímo od uživatele nebo v rámci periodické kontroly), odešle tento modul dotaz repozitáři na změny od poslední kontroly. *ClearCase* nepodporuje přímé hlášení změn souborů, ale pouze takzvaných *událostí* (*events*). Mezi *události* patří například zamknutí souboru, import a export souborů mezi vzdálenými repozitáři či události týkající se správy repozitáře. Z všech událostí je třeba vybrat pouze ty, které mají co dočinění se změnou souboru nebo jeho vytvořením. Tyto události jsou dvě: `mkbranch` a `checkin`.

Dále je třeba odstranit další nadbytečné záznamy. Nadbytečné jsou třeba změny dočasných souborů, které si *ClearCase* vytváří, či změny od předem specifikovaných uživatelů — například uživatel, který zapisuje do repozitáře číslo dané kompilace (tyto informace jsou z hlediska *Průběžné integrace* naprosto irelevantní). Poté jsou změny uspořádány podle času jejich výskytu a jsou vráceny *Dispečerovi*. Na obrázku 4.8 je znázorněno schéma systému.

Kroky zjišťování změn v repozitáři

1. Zjištění všech událostí (*events*) od poslední plné kompilace či poslední kontroly.
2. Selekce relevantních událostí (vytvoření a změnění souborů).
3. Odstranění ignorovaných souborů (dočasné soubory, změny od určitého uživatele).
4. Uspořádání podle času výskytu události.
5. Předání *Dispečerovi*.



Obrázek 4.8: Postup zjišťování změn v repozitáři.

4.3.3 Kompilace

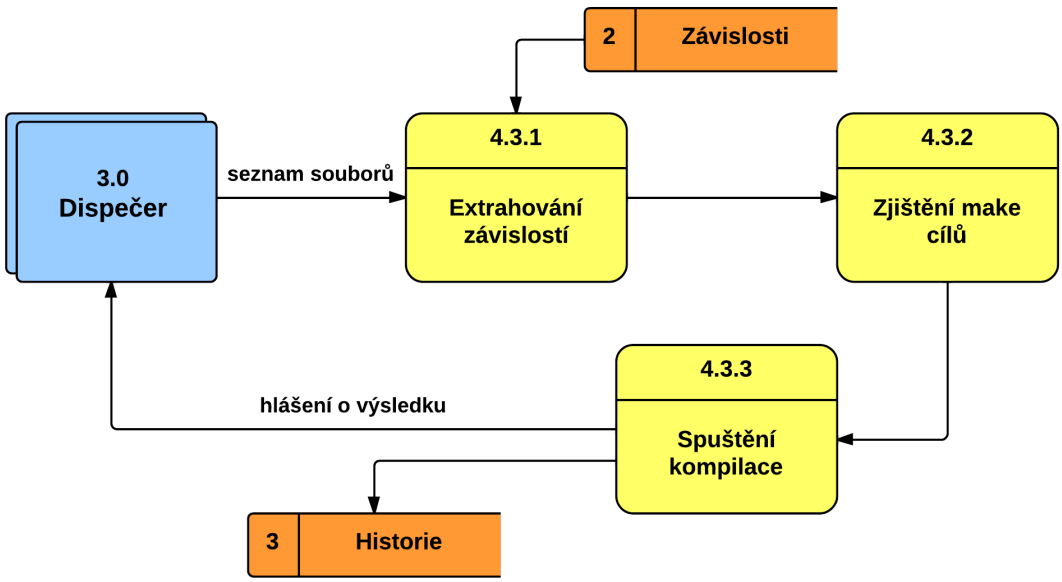
Dispečer odešle balík změn obsahující seznam souborů, které byly změněny a s nimiž chceme provést integrační kompilaci. Pro dotčené soubory jsou z datového skladu č. 2 (*Závislosti*) extrahovány jejich závislosti. Tyto závislosti musely být dříve zpracovány, analyzovány a uloženy (více v sekci 4.3.1).

Jakmile jsou k dispozici všechny závislosti, jsou zjištěny z log souborů poslední plné kompilace odpovídající *make cíle*, kterými dané soubory procházejí. Tyto cíle jsou následně seřazeny do správného pořadí a poté je postupně spouštěna jejich kompilace. Pro každý cíl vytvoří odpovídající soubor s logy. Jakmile jsou všechny cíle zkompilovány, je výsledek (včetně log souborů) předán *Dispečerovi*. Na obrázku 4.9 je znázorněno schéma systému.

Kompilace se provádí v předem připraveném prostředí, které je neměnné. V rámci *ClearCase* je toto oddělené prostředí realizováno pomocí tzv. dynamického pohledu (*dynamic view*). Různé pohledy se mezi sebou nemohou ovlivnit. Jsou tak dokonale oddělené — ať už z hlediska konfigurace a nastavení, tak z hlediska výsledků kompilace. Tyto pohledy sdílí stejnou kódovou základnu (všechny zdrojové kódy).

Kroky kompilace balíku změn

1. Přijetí seznamu souborů jejichž změny budou integrovány pomocí kompilace.
2. Extrahování závislostí těchto souborů.
3. Zjištění *make cílů* z kompilačních logů z plné kompilace.
4. Kompilace zjištěných *make cílů*.
5. Vytvoření souborů s logy kompilace.
6. Předání výsledků a souborů s logy *Dispečerovi*.



Obrázek 4.9: Průběh kompilace balíku změn.

Kapitola 5

Výsledky nasazení

V této části je rozebráno nasazení a praktické použití vyvinutého nástroje. Dále budou probrány potenciální chybové scénáře a rizika z nich plynoucí.

5.1 Potenciálně problémové scénáře

Vyvinutý nástroj se musí v reakci na okolní prostředí a vstupy od uživatele potýkat s nejrůznějšími chybovými stavy. V této kapitole budou ukázány ty nejčastější a také bude popsáno chování nástroje, včetně potenciálních rizik.

Systém samozřejmě nepracuje v izolovaném prostředí, ale v reálném světě, kde ke svému správnému fungování potřebuje funkční nástroje nižší vrstvy. Příkladem mohou být nedostupné síťové prostředky. Tento aspekt je navíc posílen faktem, že systém *ClearCase* se při *každém* volání jeho nástrojů (například pro kompilaci nebo zjišťování informací o souborech v repozitáři) dotazuje licenčního serveru na to, zda má daný uživatel právo využít dané služby. Tento licenční server se navíc pro brněnskou pobočku firmy *Unify* nachází mimo Českou republiku.

V této kapitole se proto zaměříme na scénáře vycházející pouze z vyvinutého nástroje pro podporu *Průběžné integrace*, ignorujíce tak problémy plynoucí z nefunkčních částí okolních prostředí.

5.1.1 Scénář 1: Nevhodné nastavení prostředí ClearCase

Celý systém pracuje v dynamickém pohledu nástroje *ClearCase*. V této souvislosti pak je důležité zajistit, že tento dynamický pohled je správně nastaven. Je třeba, aby vyvinutý nástroj (a uživatel, který ho spouští) měl například správně nastavená práva k souborovému systému.

Dalším aspektem je nutnost zajistit správné nastavení prostředí, například nejrůznější vnitřní proměnné, které pak ovlivňují samotnou kompilaci — například výběrem kompilátoru, přidáním dalších přepínačů pro kompilaci, změna adresářů pro linkování knihoven atp. Nastavení těchto atributů se provádí pomocí předpřipravených konfiguračních skriptů.

V neposlední řadě je třeba zajistit správnou výchozí *konfigurační specifikaci*, která určuje, jaké verze konkrétních elementů (souborů) se zobrazují. Ve výsledku je tedy „pouze“ třeba zajistit, aby nikdo nesprávnou manipulací s výše uvedeným neuvedl prostředí nástroje do nevalidního stavu. Pokud by se tak stalo, mohly by se začít objevovat nepředvídatelné problémy, jako například selhávání kompilace pro jednu konkrétní dynamickou knihovnu.

V současné době nástroj nezajišťuje správné nastavení prostředí z hlediska *ClearCase* atributů z důvodu příliš širokého záběru — nástroj nemá sloužit jako všeobjímající komplexní systém, který bude řídit všechny aspekty kompilace. Pro použití nástroje je tedy nutné manuálně správně nakonfigurovat běhové prostředí (*runtime*). V našem případě bylo nastavení prostředí odvozeno od pravidelné noční plné kompilace (*Nightly Build*).

5.1.2 Scénář 2: Porušení datových souborů

Může nastat situace, kdy se poškodí datové (*cache*) soubory, například soubory s uloženými, dříve vygenerovanými, závislostmi. Jakmile nástroj detekuje poškození tohoto souboru (například nejde otevřít nebo má špatný formát), spustí automaticky opětovné vygenerování závislostí pro dotčenou komponentu. Bohužel tento fakt může značně prodloužit reakční dobu celého nástroje, neboť analýza závislostí je časově nejnáročnější operace.

Nástroj nemá šanci poznat *nekompletní* závislosti, tzn. situaci, kdy sice datový soubor je v pořádku a ve správném formátu, ale nebyly do něj zapsány všechny informace. Tehdy může docházet ke kompilačním chybám z důvodu, že nebyly spuštěny všechny potřebné *make cíle*. Nástroj tím ztrácí svou základní funkcionalitu. Jedinou prevencí je pravidelně obnovovat závislosti a následně kontrolovat, zda některé soubory nemají výrazně menší velikost než předtím. Pokud mají, je třeba tuto situaci ručně analyzovat — ideálně spustit generování závislostí několikrát a porovnávat výsledky.

5.1.3 Scénář 3: Více současně vložených změn

V reálném prostředí může samozřejmě nastat situace, kdy několik vývojářů uloží změněné soubory (balíky změn) do repozitáře, přičemž tyto změny spolu logicky nesouvisí. Tato situace může nastat tehdy, pokud vývojář odešle ke zpracování jeden balík změn a současně (nebo v průběhu kompilace) jiný vývojář odešle nové (jiné) změny. Pokud by teď došlo k integračnímu konfliktu, nevíme, která část ho způsobila — jestli změny z prvního balíku nebo změny vložené při jeho zpracovávání.

Pro řešení tohoto problému jsou použity tzv. *značky* (*label*). Jakmile nástroj detekuje změněný soubor, okamžitě na jeho stabilní verzi[1] vloží *značku*. Takto se zachová pro všechny změny. Jakmile vývojář odstartuje kompilaci, jsou na soubory, které jsou v daném balíku změn, umístěny nové značky s vyšší prioritou. V konfigurační specifikaci (*config spec*) dynamického pohledu je uvedeno pravidlo, které ustanovuje, že integrační značky mají vyšší prioritu než značky označující nově změněné soubory (které jsou v balíku změn). Tímto je zajištěno, že kompilace proběhne pouze se zahrnutím námi požadovaných souborů, které uživatel označil v rámci balíku změn.

Pokud při zpracování jednoho balíku změn bude přijat další, postaví se do fronty a za předpokladu úspěšné integrace předchozího balíku bude spuštěn ihned po něm.

5.2 Použití vyvinutého nástroje

V této části jsou popsány příklady typického použití vyvinutého nástroje pro podporu *Průběžné integrace*. Nejedná se o uživatelskou příručku, tudíž zde nejsou k nalezení postupy a návody ve stylu popisu ovládacích prvků. Naopak se zde nachází popis použití nástroje

[1] Stabilní verze je ta, se kterou byla úspěšně provedena kompilace – je jedno, jestli v rámci integrační nebo plné kompilace.

z hlediska technologického — jaké akce vyvolá daná událost, jaká data a kam jsou posílána atp.

V tabulce 5.1 je uveden přehled doby zpracování vybraných operací. Doba trvání je určena pomocí střední hodnoty z patnácti měření, kdy byl daný úkon skriptem spouštěn za sebou. Analýza volaných příkazu ukázala, že největším zpomalením pro dobu vykonávání je čekání na výsledek příkazů *ClearCase*.

Název operace	Přibližná doba trvání
Generování závislostí	40 minut
Inicializace nástroje	4 minuty
Zjištění závislostí pro balík změn (z datového skladu)	12 sekund
Aktualizace změněných souborů	7 sekund

Tabulka 5.1: Doba zpracování vybraných operací.

5.2.1 Integrační ověření daných souborů

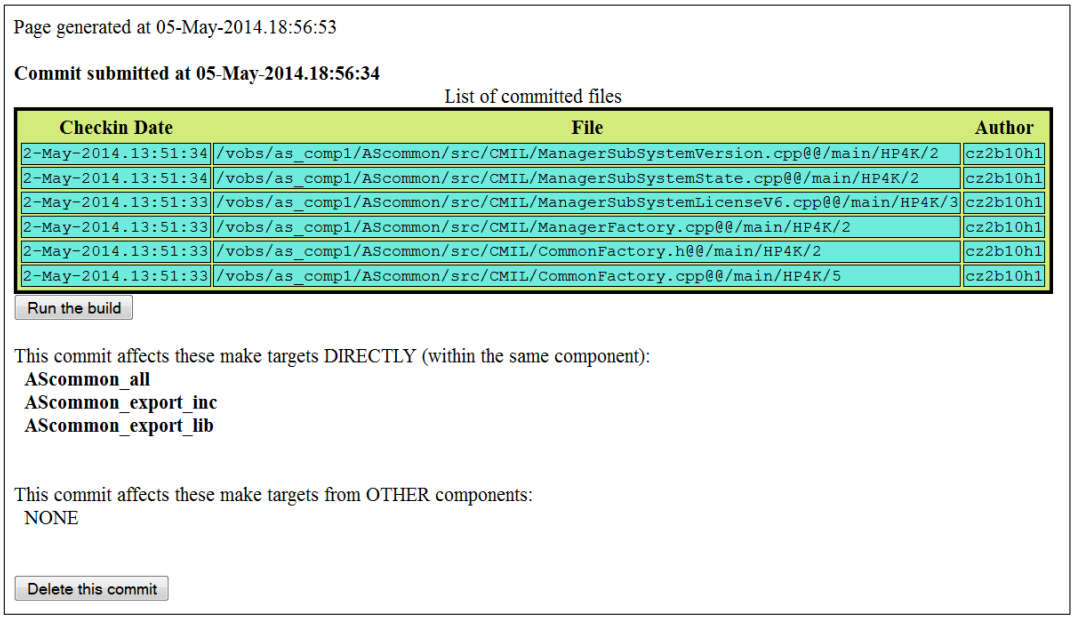
Toto je nejčastější případ užití vyvinutého nástroje — vývojář dokončí úkol, na kterém pracoval, a je připraven ověřit, zda tyto změny nezpůsobí integrační konflikt. Odešle tudíž požadavek na vytvoření balíku změn z jím změněných souborů. Toto může učinit skrze rozhraní příkazové řádky (odešle přímo jména souborů, včetně jejich verze), pomocí příkazu `cli_tool.py -s FILE` kde soubor `FILE` obsahuje změněné soubory. Dále je možné balík vytvořit pomocí webového rozhraní, kde se soubory jednoduše označí. Ukázku je možné vidět na obrázku 5.1, kde je označeno šest vybraných souborů, ze kterých chceme vytvořit balík změn. Vidíme jejich název, čas změny a autora změny (`cz2b10h1`).

☑	2-May-2014.13:51:34	/vobs/as_comp1/AScommon/src/CMIL/ManagerSubSystemVersion.cpp@@/main/HP4K/2	cz2b10h1
☑	2-May-2014.13:51:34	/vobs/as_comp1/AScommon/src/CMIL/ManagerSubSystemState.cpp@@/main/HP4K/2	cz2b10h1
☑	2-May-2014.13:51:33	/vobs/as_comp1/AScommon/src/CMIL/ManagerSubSystemLicenseV6.cpp@@/main/HP4K/3	cz2b10h1
☑	2-May-2014.13:51:33	/vobs/as_comp1/AScommon/src/CMIL/ManagerFactory.cpp@@/main/HP4K/2	cz2b10h1
☑	2-May-2014.13:51:33	/vobs/as_comp1/AScommon/src/CMIL/CommonFactory.h@@/main/HP4K/2	cz2b10h1
☑	2-May-2014.13:51:33	/vobs/as_comp1/AScommon/src/CMIL/CommonFactory.cpp@@/main/HP4K/5	cz2b10h1

Obrázek 5.1: Označené soubory do balíku změn.

V tomto okamžiku *Dispečer* přijímá požadavek na vytvoření balíku změn z daných souborů. Vytvoří tedy tento balík, přesune dotčené soubory ze své vnitřní struktury „neintegrovaných souborů“ do struktury balíků změn (dále jen balík) připravených pro kompilaci. Nyní zjistí všechny závislosti a tím i nutné *make cíle* pro daný balík. Tyto informace extrahuje z dříve vytvořených cache souborů. Příklad zpracovaného balíku změn je možné spatřit na obrázku 5.2. Zde vidíme seznam souborů v tomto balíku, čas jeho vytvoření a detekované *make cíle* (`all`, `export_inc` a `export_lib`), které se spustí při integračním ověření. Také je zde možnost tento balík změny odstranit (`Delete this commit`). Pomocí CLI si lze tyto informace zobrazit příkazem `cli_tool.py -c`.

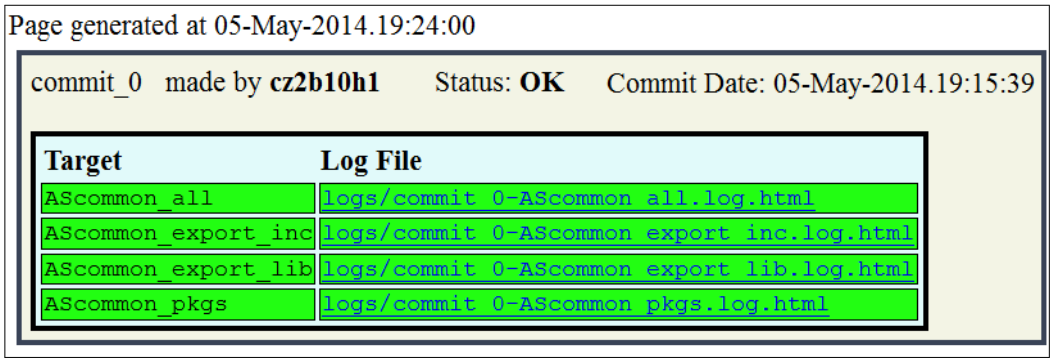
Teď jeho práce končí a čeká se na reakci vývojáře. Nyní musí zadat příkaz pro spuštění kompilace a sestavení — tento příkaz je možné v rozhraní příkazové řádky sloučit s prvním,



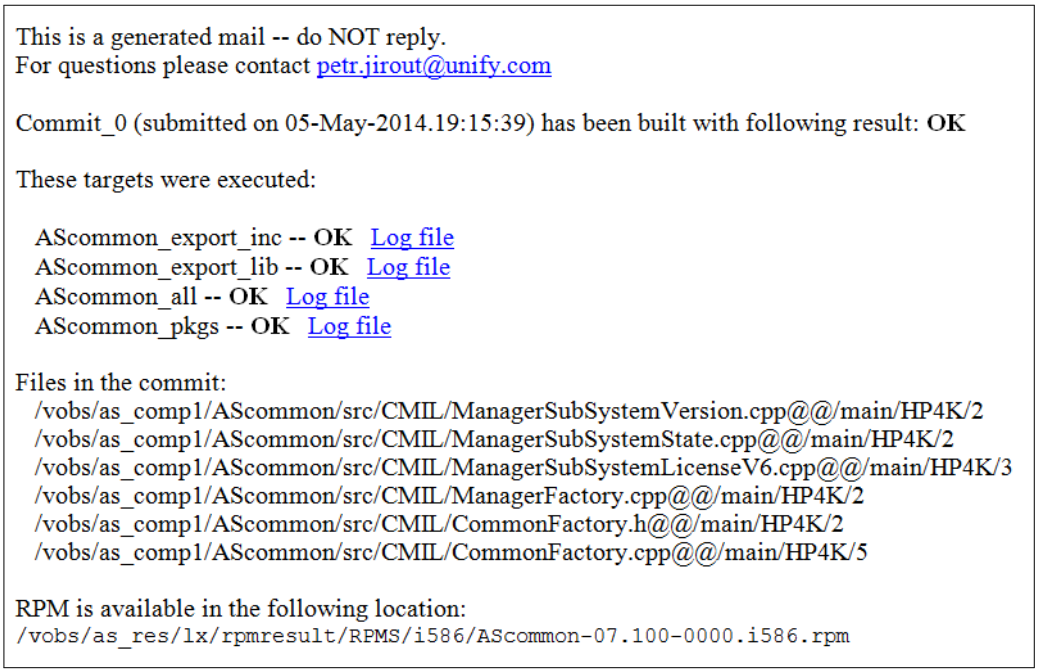
Obrázek 5.2: Připravený balík změn.

který odesílá názvy změněných souborů (sémantika akce je pak následující: vytvoř balík změn a hned ho zkus zkompilovat) — a v tomto momentu se již nemusí o nic starat. V tomto momentu nástroj identifikuje dotčený balík a postupně ve správném pořadí volá jím vygenerované *make cíle*, ukládajíce si výsledky těchto jednotlivých kompilací ve formě později dostupných logů, viz obrázek 5.3. Zde vidíme autora tohoto balíku změn (`cz2b10h1`), výsledný status (`OK`), čas vytvoření balíku (5. května) a ještě informaci, že všechny *make cíle* byly dokončeny úspěšně. Dále je zde odkaz na webovou stránku, kde je možné nalézt kompilační logy. V případě neúspěšné kompilace by se zelená barva změnila na červenou a status by se změnil na `FAILED`. Soubory s logy nejsou dostupné skrze rozhraní CLI, zobrazen je pouze celkový status integrace.

Nakonec vytvoří `RPM` soubor dané komponenty a odešle hlášení na mail příslušnému uživateli. Obsah tohoto hlášení je možné vidět na obrázku 5.4. Toto hlášení obsahuje stejné informace jako stránka s logy plus cestu k výslednému `RPM` souboru.



Obrázek 5.3: Log soubory zpracovaného balíku změn.


This is a generated mail -- do NOT reply.
For questions please contact petr.jirout@unify.com

Commit_0 (submitted on 05-May-2014.19:15:39) has been built with following result: **OK**

These targets were executed:

AScommon_export_inc -- **OK** Log file
AScommon_export_lib -- **OK** Log file
AScommon_all -- **OK** Log file
AScommon_pkgs -- **OK** Log file

Files in the commit:
/vobs/as_comp1/AScommon/src/CMIL/ManagerSubSystemVersion.cpp@@/main/HP4K/2
/vobs/as_comp1/AScommon/src/CMIL/ManagerSubSystemState.cpp@@/main/HP4K/2
/vobs/as_comp1/AScommon/src/CMIL/ManagerSubSystemLicenseV6.cpp@@/main/HP4K/3
/vobs/as_comp1/AScommon/src/CMIL/ManagerFactory.cpp@@/main/HP4K/2
/vobs/as_comp1/AScommon/src/CMIL/CommonFactory.h@@/main/HP4K/2
/vobs/as_comp1/AScommon/src/CMIL/CommonFactory.cpp@@/main/HP4K/5

RPM is available in the following location:
/vobs/as_res/lx/rpmresult/RPMS/i586/AScommon-07.100-0000.i586.rpm

Obrázek 5.4: Obsah zprávy o kompilaci balíku změn.

Obsah hlášení je možné si prohlédnout v sekci 4.2.2. V případě, že kompilace nebyla úspěšná, je vývojářovi podána zpráva a on je povinen tento integrační konflikt co nejrychleji opravit. Pokud všechno proběhlo v pořádku, je mu opět zaslána zpráva, ale nevyžaduje se od něj žádná další akce.

5.2.2 Administrace nástroje

Pro nasazení vyvinutého nástroje je potřeba nejprve správně vytvořit a následně nastavit *ClearCase* dynamický pohled. Například nastavení konfigurační specifikace pohledu může existovat několik. Je doporučeno řídit se nastavením, které se běžně používá pro plnou kompilaci projektu — v našem případě specifikace pro *Nightly Build*.

Dále je potřeba pravidelně generovat a obnovovat závislosti mezi soubory (pomocí skriptu `generate_deps.py`). Praxe ukazuje, že stačí toto generování provést jednou denně, ideálně po plné noční kompilaci. Ačkoliv je možné provádět generování častěji, z hlediska zatížení systému a potenciálního přínosu je výsledný zisk minimální. Pro speciální případy změny závislostí může administrátor nástroje dané závislosti přegenerovat manuálně.

Jakmile jsou závislosti vygenerovány, je potřeba spustit *Dispečera*, tj. démona, který bude udržovat stav systému a sbírat požadavky. Název příslušného skriptu je `CI_watch.py`. Po jeho spuštění je třeba vyčkat několik minut, než jsou provedeny všechny inicializační úkony.

Konfigurace nástroje se provádí úpravou příslušných konfiguračních souborů, hlavním souborem je `CI_common.py` pro nastavení chování *Dispečera* a *Hlídače repozitáře*, dále pak `web_common.py` pro nastavení webového rozhraní a sbírání požadavků a `cli_common.py` pro nastavení rozhraní příkazové řádky.

Pro funkčnost webového rozhraní je nutné oprávnění pro nástroj k vytváření souborů v příslušné webové složce a povolit vykonávání `CGI` (*Common Gateway Interface*) skriptů.

Kapitola 6

Závěr

V rámci této práce byl úspěšně vytvořen a nasazen nástroj pro zlepšení pro podporu *Průběžné integrace* v projektu *OpenScape 4000 Manager*. Z důvodu efektivity a časové složitosti se vyplatilo vybudovat tento nástroj s využitím již existujících nástrojů (skriptů) projektu a prostředí. Nebylo tudíž nutné měnit zaběhlé postupy a metody kompilace projektu. Tímto nástrojem byly pokryty nejdůležitější aspekty *Průběžné integrace* ve vztahu k projektu *OpenScape 4000 Manager* — urychlení kompilace, včasná detekce integračních konfliktů a hlášení stavu odpovědným osobám.

Tato práce na příkladu reálného komplexního projektu ukazuje, že princip *Průběžné integrace* je možné aplikovat i na projekty s dlouhou historií, rozsáhlou základnou zdrojových souborů a mnohačetnými závislostmi, které při jejich založení s tímto konceptem nepočítaly.

Reálné nasazení pak ukazuje i přínos samotné metodiky *Průběžné integrace*, který tkví ve zrychlené detekci chyb, jejich rychlejší opravě a v konečném důsledku i v ušetřených přímých i nepřímých nákladech.

Jako další rozšíření je možné implementovat simultánní kompilaci různých balíků změn. Dále pak zavést víceúrovňové integrační ověření, při současné existenci více integračních strojů, kdy nejprve první integrační stroj ověří kompilační integrační konflikty a další stroj pak provede ověření integrace pomocí spuštění vybraných testů a instalace systému. Další možností je využít získané závislosti mezi soubory k detekci nevyužívaných souborů a knihoven a tím přispět k vyčištění zdrojové základny od nepotřebných souborů.

Vyvinutý nástroj bude začleněn do vývojového procesu daného projektu a tím bude přispívat k ušetření nákladů na vývoj.

Literatura

[1] Beck, K. a Andres, C. *Extreme Programming Explained: Embrace Change*. 2. vyd. Boston, USA: Addison-Wesley, 2005. ISBN 987-0-321-27865-4.

[2] Duvall, P. M., Matyas, S. a Glover, A. *Continuous Integration: Improving Software Quality and Reducing Risk*. 1. vyd. Boston, USA: Addison-Wesley, 2007. ISBN 987-0-321-33638-5.

[3] Fowler, M. *Continuous Integration* [online]. 2006-04-01 [cit. 2014-03-28]. Dostupné na: <`http://martinfowler.com/articles/continuousIntegration.html`>.

[4] Gamma, E. et al. *Design Patterns: Elements of Reusable Object-Oriented Software*. 8. vyd. Boston, USA: Addison-Wesley, 1994. ISBN 978-0-201-63361-0.

[5] Mishra, J. a Mohanty, A. *Software Engineering*. 1. vyd. Boston, USA: Addison-Wesley, 2011. ISBN 978-8-131-75869-4.

Příloha A

Uživatelská příručka

V této sekci je popsáno použití nástroje pro oba potenciální uživatele — vývojáře, který bude nástroj používat pro detekci integračních konfliktů a administrátora, který bude nástroj spravovat a udržovat.

A.1 Vývojář

A.1.1 Příprava systému

Webovou část není třeba nijak nastavovat, pouze zjistit adresu webového rozhraní a poté na něj přistoupit pomocí libovolného webového prohlížeče, který podporuje HTML a JavaScript.

Pro použití CLI rozhraní je třeba zajistit přístup ke skriptu `cli_tool.py`. Je vhodné si vytvořit alias na spuštění toho skriptu z adresáře administrátora systému *Průběžné integrace*. Skript si poté všechna nastavení čte z konfiguračního souboru, který udržuje administrátor. Tento soubor obsahuje například IP adresu integračního serveru a další informace potřebné pro připojení.

A.1.2 Použití systému

Webové rozhraní je intuitivní a bylo popsáno v sekci 5.2.1. V této sekci je popsán nejčastější případ užití — ověření integračních změn — pomocí CLI rozhraní.

Integrační ověření změn pomocí CLI

1. Odeslání změn na integrační server, pomocí `cli_tool.py -s FILE`. Je možné zadat soubory na standardním vstupu i s využitím dalších nástrojů příkazové řádky, například `cli_tool.py -l -u MY_LOGIN | grep cpp | cli_tool.py -s -`. Příkaz vrátí ID daného balíku změn, v našem případě například `commit_21`.

2. Zjištění, které *make cíle* jsou dotčeny. Příkaz `cli_tool.py -c` a následné nalezení daného balíku změn (`commit_21`).

3. Spuštění integračního ověření balíku změn. Příkaz `cli_tool.py -b commit_21` nebo `cli_tool.py -b 21`.

4. Do e-mailové schránky po dokončení přijde hlášení na mail a při vypsání balíků změn skrze CLI (`cli_tool.py -c`) je zobrazen výsledný status kompilace `OK` nebo `FAILED`.

Pokud se odeslaný balík změn neobjeví v seznamu balíků, je nutné si nechat vypsat frontu požadavků integračního démona (pomocí `cli_tool.py -q`) — zde se může objevit a bude zpracován, až na něj dojde řada.

A.2 Administrátor

A.2.1 Příprava systému

Je třeba zajistit pravidelnou plnou kompilaci ve stanovený čas (ideálně v noci). Tento čas je třeba zadat do `CI_watch.py` skriptu. Jakmile je plná kompilace úspěšně dokončena, vždy je třeba automaticky generovat závislosti — spustit skript `generate_deps.py`. Jakmile je dokončeno generování závislostí, spustit hlavního démona — skript `CI_watch.py`.

Pro správné fungování rozhraní CLI je třeba nastavit práva ke čtení a spouštění pro všechny uživatele systému *Průběžné integrace*. Dále je třeba zadat do konfiguračního souboru `cli_config.py` IP adresu stroje a příslušný port, na kterém je spuštěn integrační démon (*Dispečer*).

Je potřeba nastavit oprávnění ke spouštění CGI skriptů na stroji, kde běží integrační démon.

Dále je třeba zajistit správné mapování uživatelských loginů na e-mailové adresy. Příslušné mapování je uvedeno v souboru `user_mails.txt`.

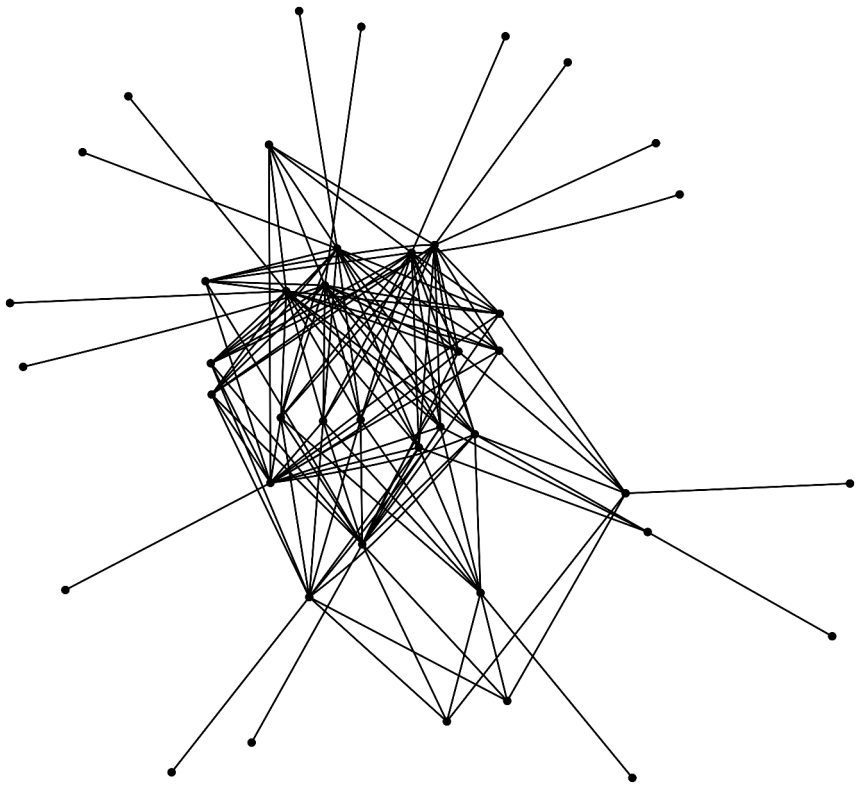
A.2.2 Údržba systému

Je třeba průběžně kontrolovat logy, které jsou zapisovány do souboru `CI_logs.txt`. Je nutné kontrolovat dostupnost integračního stroje a webového rozhraní.

Příloha B

Ukázka závislostí

Jako příklad jsou na obrázku B.1 pomocí programu Graphviz zobrazeny závislosti statické knihovny `libxie`. Tato knihovna je tvořena 105 hlavičkovými soubory a 11 zdrojovými soubory. K tomu používá knihovnu OpenSSL, která je zde zobrazena jako jeden uzel. Při překladu je vytvořeno 11 objektových souborů. Tato knihovna má 134 vztahů závislostí (například vztah zdrojový soubor – hlavičkový soubor), které jsou reprezentovány hranami. Soubory jsou reprezentovány body, kde každý bod představuje jeden unikátní soubor. Z této ukázky jasně plyne, že závislosti byť i jedné knihovny nejsou zpracovatelné manuálně. Pro větší přehlednost byly odstraněny názvy souborů.



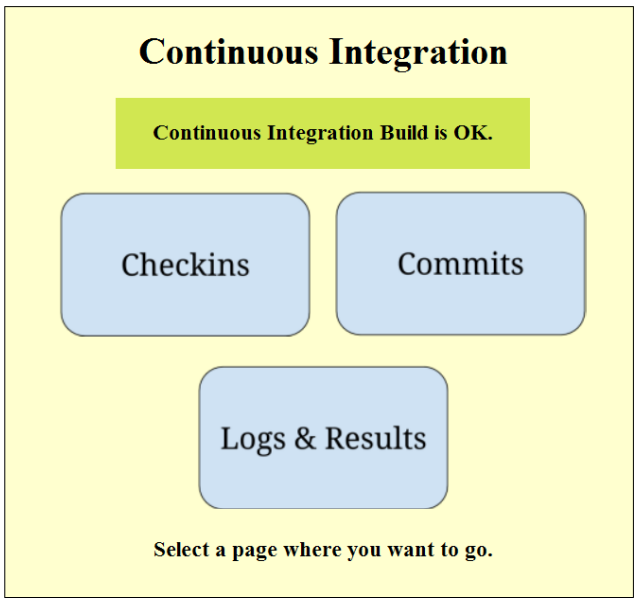
Obrázek B.1: Závislosti knihovny `libxie`.

Příloha C

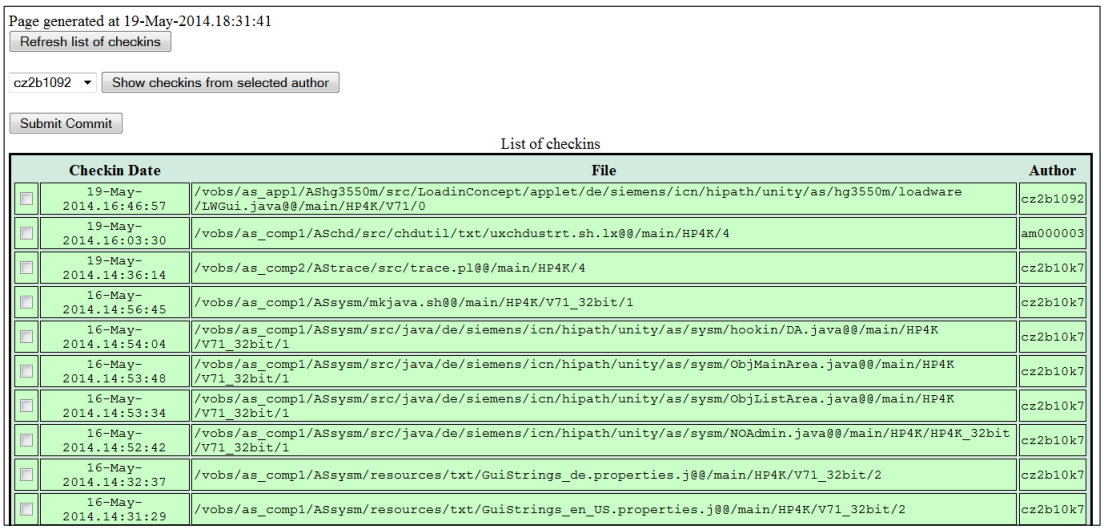
Uživatelské rozhraní

Některé dialogy byly již představeny v sekci 5.2.1, nebudou zde tudíž zobrazeny znovu. V této části se nachází obrázky webového rozhraní a rozhraní příkazové řádky.

C.1 Webové rozhraní



Obrázek C.1: Úvodní stránka systému.



Page generated at 19-May-2014.18:31:41

Refresh list of checkins

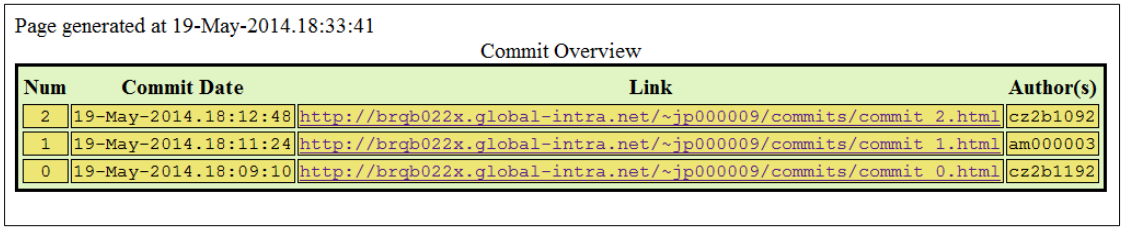
cz2b1092 Show checkins from selected author

Submit Commit

List of checkins

	Checkin Date	File	Author
☐	19-May-2014.16:46:57	/vobs/as_appl/AShg3550m/src/LoadinConcept/applet/de/siemens/icn/hipath/unity/as/hg3550m/loadware/LWGui.java@@/main/HP4K/V71/0	cz2b1092
☐	19-May-2014.16:03:30	/vobs/as_comp1/ASchd/src/chdutil/txt/uxchdustrt.sh.lx@@/main/HP4K/4	am000003
☐	19-May-2014.14:36:14	/vobs/as_comp2/AStrace/src/trace.pl@@/main/HP4K/4	cz2b10k7
☐	16-May-2014.14:56:45	/vobs/as_comp1/ASsysm/mkjava.sh@@/main/HP4K/V71_32bit/1	cz2b10k7
☐	16-May-2014.14:54:04	/vobs/as_comp1/ASsysm/src/java/de/siemens/icn/hipath/unity/as/sysm/hookin/DA.java@@/main/HP4K/V71_32bit/1	cz2b10k7
☐	16-May-2014.14:53:48	/vobs/as_comp1/ASsysm/src/java/de/siemens/icn/hipath/unity/as/sysm/ObjMainArea.java@@/main/HP4K/V71_32bit/1	cz2b10k7
☐	16-May-2014.14:53:34	/vobs/as_comp1/ASsysm/src/java/de/siemens/icn/hipath/unity/as/sysm/ObjListArea.java@@/main/HP4K/V71_32bit/1	cz2b10k7
☐	16-May-2014.14:52:42	/vobs/as_comp1/ASsysm/src/java/de/siemens/icn/hipath/unity/as/sysm/NOAdmin.java@@/main/HP4K/HP4K_32bit/V71_32bit/1	cz2b10k7
☐	16-May-2014.14:32:37	/vobs/as_comp1/ASsysm/resources/txt/GuiStrings_de.properties.j@@/main/HP4K/V71_32bit/2	cz2b10k7
☐	16-May-2014.14:31:29	/vobs/as_comp1/ASsysm/resources/txt/GuiStrings_en_US.properties.j@@/main/HP4K/V71_32bit/2	cz2b10k7

Obrázek C.2: Přehled změněných souborů.

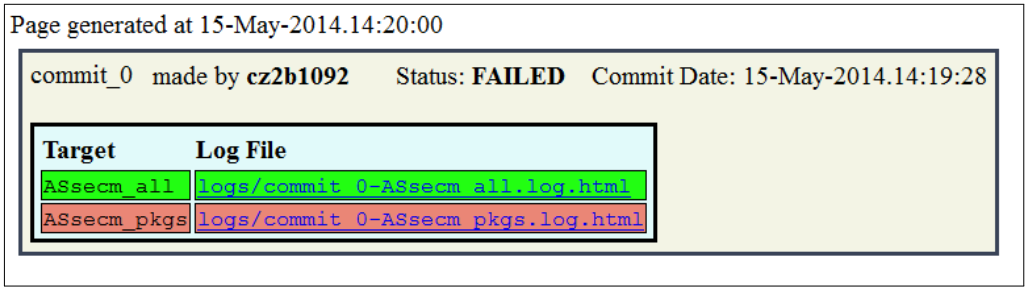


Page generated at 19-May-2014.18:33:41

Commit Overview

Num	Commit Date	Link	Author(s)
2	19-May-2014.18:12:48	http://brqb022x.global-intra.net/~jp000009/commits/commit_2.html	cz2b1092
1	19-May-2014.18:11:24	http://brqb022x.global-intra.net/~jp000009/commits/commit_1.html	am000003
0	19-May-2014.18:09:10	http://brqb022x.global-intra.net/~jp000009/commits/commit_0.html	cz2b1192

Obrázek C.3: Přehled balíků změn.



Page generated at 15-May-2014.14:20:00

commit_0 made by **cz2b1092** Status: **FAILED** Commit Date: 15-May-2014.14:19:28

Target	Log File
ASsecm_all	logs/commit_0-ASsecm_all.log.html
ASsecm_pkgs	logs/commit_0-ASsecm_pkgs.log.html

Obrázek C.4: Log neúspěšné kompilace.

C.2 Rozhraní příkazové řádky

```
SLES10_SP3 ~/CI > ./cli_tool.py -u cz2b10r4 -t 07-May-2014.16:16:42
09-May-2014.15:21:08    /vobs/as_comp2/ASswa2/src/SwaCommon.pm@@/main/HP4K/45   cz2b10r4
09-May-2014.15:21:08    /vobs/as_comp2/ASswa2/src/swa@@/main/HP4K/20    cz2b10r4
09-May-2014.15:21:08    /vobs/as_comp2/ASswa2/src/PP_RLC_activate@@/main/HP4K/44        cz2b10r4
07-May-2014.17:58:44    /vobs/as_comp2/ASswa2ui/txt/SWAtxt_de.properties@@/main/HP4K/11 cz2b10r4
07-May-2014.16:16:43    /vobs/as_comp2/ASswa2ui/src/gui/con@@/main/HP4K/25      cz2b10r4
```

Obrázek C.5: Vypsání změněných souborů pro konkrétního uživatele a od určitého času.

```
SLES10_SP3 ~/CI > ./cli_tool.py -u cz2b1192 -t 19-May-2014.16:56:19 | grep AShg | ./cli_tool.py -s -
Submitting file: /vobs/as_appl/AShg3550m/src/LoadinConcept/daemon/LWRefreshCgi.cpp@@/main/HP4K/HP4K_32bit/V71_32bit/1
Submitting file: /vobs/as_appl/AShg3550m/src/LoadinConcept/client/lw_update_client.cpp@@/main/HP4K/HP4K_32bit/V71_32bit/1
Submitting file: /vobs/as_appl/AShg3550m/src/LoadinConcept/daemon/LWUpdateDaemon.cpp@@/main/HP4K/HP4K_32bit/V71_32bit/2
Submitting file: /vobs/as_appl/AShg3550m/src/LoadinConcept/libs/liblwXML.cpp@@/main/HP4K/HP4K_32bit/V71_32bit/1
Submitting file: /vobs/as_appl/AShg3550m/src/LoadinConcept/libs/liblwXML.h@@/main/HP4K/HP4K_32bit/V71_32bit/1
Submitting file: /vobs/as_appl/AShg3550m/src/backup/interface/hg_interf.pm@@/main/HP4K/HP4K_32bit/V71_32bit/2
Submitting file: /vobs/as_appl/AShg3550m/src/LoadinConcept/daemon/LWUpdate_common.h@@/main/HP4K/HP4K_32bit/V71_32bit/1
Submitting file: /vobs/as_appl/AShg3550m/src/LoadinConcept/daemon/LWUpdate_common.cpp@@/main/HP4K/HP4K_32bit/V71_32bit/1
Created commit_0
```

Obrázek C.6: Vytvoření balíku změn pomocí vypsání změn a následného čtení dat ze vstupu.

```
SLES10_SP3 ~/CI > ./cli_tool.py -c
Unprocessed commits:

commit_0        (19-May-2014.18:09:10)
        Authors: cz2b1192
        Targets: AShg3550m_export_lib AShg3550m_all AShg3550m_pkgs
        Files:  /vobs/as_appl/AShg3550m/src/LoadinConcept/daemon/LWUpdate_common.cpp@@/main/HP4K/HP4K_32bit/V71_32bit/1
                /vobs/as_appl/AShg3550m/src/LoadinConcept/daemon/LWUpdate_common.h@@/main/HP4K/HP4K_32bit/V71_32bit/1
                /vobs/as_appl/AShg3550m/src/backup/interface/hg_interf.pm@@/main/HP4K/HP4K_32bit/V71_32bit/2
                /vobs/as_appl/AShg3550m/src/LoadinConcept/libs/liblwXML.h@@/main/HP4K/HP4K_32bit/V71_32bit/1
                /vobs/as_appl/AShg3550m/src/LoadinConcept/libs/liblwXML.cpp@@/main/HP4K/HP4K_32bit/V71_32bit/1
                /vobs/as_appl/AShg3550m/src/LoadinConcept/daemon/LWUpdateDaemon.cpp@@/main/HP4K/HP4K_32bit/V71_32bit/2
                /vobs/as_appl/AShg3550m/src/LoadinConcept/client/lw_update_client.cpp@@/main/HP4K/HP4K_32bit/V71_32bit/1
                /vobs/as_appl/AShg3550m/src/LoadinConcept/daemon/LWRefreshCgi.cpp@@/main/HP4K/HP4K_32bit/V71_32bit/1

Processed commits:

commit_1        (19-May-2014.18:11:24)
        Authors: am000003
        Targets: AScomwin_pkgs AScomwin_all
        Build status: OK
        Files:  /vobs/as_comp1/AScomwin/src/access/ComWinAccessImpl.cpp@@/main/HP4K/HP4K_32bit/1

commit_2        (19-May-2014.18:12:48)
        Authors: cz2b1092
        Targets: AShttp_pkgs AShttp_all
        Build status: OK
        Files:  /vobs/as_comp2/AShttp/src/mod_unity/module/mod_unity.h@@/main/HP4K/V71/0
```

Obrázek C.7: Vypsání balíků změn dostupných na serveru.

```
SLES10_SP3 ~/CI > ./cli_tool.py -b 1
Igniting build of commit_1
Build started
```

Obrázek C.8: Spuštění integrační kompilace vybraného balíku změn.

Příloha D

Obsah CD

Elektronická verze textu této bakalářské práce se nachází v kořenovém adresáři přiloženého CD. Na disku se nacházejí také následující adresáře:

Adresář `src`

- Zdrojové kódy vyvinutého nástroje pro podporu systému *Průběžné integrace*.

Adresář `tex`

- Zdrojové kódy textu této práce ve formátu LaTeX.