VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INTELLIGENT SYSTEMS

SADA TESTŮ ROZHRANÍ XML-RPC PRO SYSTÉM SPACEWALK

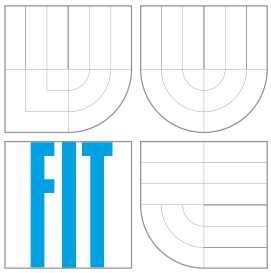
BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE
AUTHOR

JAN VLČEK

BRNO 2011

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INTELLIGENT SYSTEMS

SADA TESTŮ ROZHRANÍ XML-RPC PRO SYSTÉM SPACEWALK

A TEST SUITE OF SPACEWALK XML-RPC INTERFACE

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE
AUTHOR

JAN VLČEK

VEDOUCÍ PRÁCE
SUPERVISOR

Ing. ALEŠ SMRČKA, Ph.D.

BRNO 2011

Abstrakt

Práce se zabývá vytvářením sady testů pro XML-RPC rozhraní systému Spacewalk a to převážně knihovny, která má vytváření usnadnit. Řešení vychází z analýzy systému Spacewalk z pohledu komunikace pomocí XML-RPC. Práce také prezentuje několik metod, jak lze vyhodnocovat kvalitu testů. Popsána je implementace metody pokrytí větví zdrojového kódu.

Abstract

This thesis covers the creation of test suite for Spacewalk XML-RPC interface. It focuses on library which is intended to simplify such a creation. The solution is based on analysis of Spacewalk from the perspective of XML-RPC communication. The thesis also presents several methods which could be used for quality of tests evaluation. Implementation of the method of branch-code coverage is demonstrated.

Klíčová slova

Spacewalk, XML-RPC, testování, pokrytí kódu, pokrytí větví kódu, Cobertura, coverage.py.

Keywords

Spacewalk, XML-RPC, testing, code coverage, branch coverage, Cobertura, coverage.py.

Citace

Jan Vlček: Sada testů rozhraní XML-RPC pro systém Spacewalk, bakalářská práce, Brno, FIT VUT v Brně, 2011

Sada testů rozhraní XML-RPC pro systém Spacewalk

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením pana Ing. Aleše Smrčky, Ph.D. Další informace mi poskytl Jan Hutař. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

.....
Jan Vlček
18. května 2011

Poděkování

Poděkování patří vedoucímu práce panu Ing. Aleši Smrčkovi, Ph.D. a Janu Hutařovi z firmy Red Hat za pomoc při vytváření této práce a za cenné a konstruktivní připomínky vedoucí k úspěšnému řešení.

Obsah

Kapitola 1

Úvod

Cílem této bakalářské práce bylo vytvořit sadu testů rozhraní XML-RPC pro systém Spacewalk. Systém Spacewalk je projekt s otevřeným zdrojovým kódem určený ke správě systémů založených na operačním systému Linux. Je vyvíjen komunitně a firma Red Hat z něj odvozuje svůj komerční produkt zvaný Red Hat Network Satellite.

Systém Spacewalk se jako celek skládá z různých nástrojů, které spolu potřebují komunikovat a které se navzájem řídí. Komunikace mezi nimi probíhá pomocí technologie XML-RPC. Nejdůležitějším článkem v celé infrastruktuře je server a testování jeho XML-RPC rozhraní je jádrem této práce.

Informace předávané v systému dosahují rozličné úrovně důležitosti. Jedním z komunikačních kanalů jsou například přenášeny údaje, kterých je následně využito pro úpravu nastavení vzdáleného operačního systému. Protože jde o velmi citlivá data, která by poškozená mohla dokonce vyřadit spravovaný operační systém, je vyžadována vysoká spolehlivost. Testy XML-RPC rozhraní serveru jsou jedním ze způsobů, jak spolehlivost zaručit a jak ji automatizovaně ověřovat.

Problematika je rozebrána v kapitolách vyjadřujících jednotlivé fáze při vytváření sady testů. Celému vývoji předchází analýza problému a technologií, na kterých staví návrh knihovny usnadňující samotný návrh testů. Návrh knihovny je obsahem kapitoly druhé. Kapitola třetí popisuje implementaci knihovny. Na implementované knihovně následně staví kapitola poslední, která využívá vytvořené knihovny k návrhu a implementaci testů.

Kapitola 2

Analýza problému

Analýza problému nastíněného v zadání bakalářské práce se dělí do několika částí, ze kterých vychází kapitoly další. Část první analyzuje funkčnosti testovaného systému Spacewalk. Dle instalační příručky[1] byl systém Spacewalk – verze 1.2 – nainstalován do operačního systému Fedora 14 a bylo prozkoumáno vše, co by se dalo v dalších fázích vývoje využít. Analýza se zaměřuje převážně na části související s rozhraním XML-RPC.

Aby bylo možné vyhodnocovat kvalitu testů, pojednává druhá část o modelech a kritériích testování.

Poslední část shrnuje znalosti získané analýzou a ve vztahu k bodům zadání bakalářské práce nastiňuje další vývoj.

2.1 Systém Spacewalk

Systém Spacewalk je jedním z komunitních projektů zaštiťovaných firmou Red Hat. Oblast uplatnění nachází ve správě linuxových operačních systémů. Spravované systémy lze sdružovat do libovolných skupin a provádět s nimi různé operace. Mezi základní operace patří vzdálená instalace a aktualizace balíčků s aplikacemi. Další operace umožňují například distribuci konfiguračních souborů nebo monitorování stavu systému a některých běžících služeb. Všechny operace lze provádět na pokyn administrátora nebo plánovat na konkrétní data a časy.

2.1.1 Třívrstvá architektura

Systém je založen na třívrstvé architektuře. Třívrstvá architektura se skládá ze tří částí – datové, procesní a prezentační. Datová vrstva funguje jako centrální, společné úložiště dat využívané více nástroji zároveň. Procesní vrstva obsahuje logiku spojenou se změnami dat a od prezentační vrstvy je oddělena z důvodu odlehčení klientských aplikací. Ty nemusí vědět nic o způsobu ukládání dat a mají pouze dvě starosti. Převod uživatelských vstupů do formátu, kterému rozumí procesní vrstva, a zpětný převod výsledků získaných od procesní vrstvy do formátu vhodného pro uživatele. Tím jsem ve stručnosti shrnul popis třívrstvé architektury popsané v knize [20].

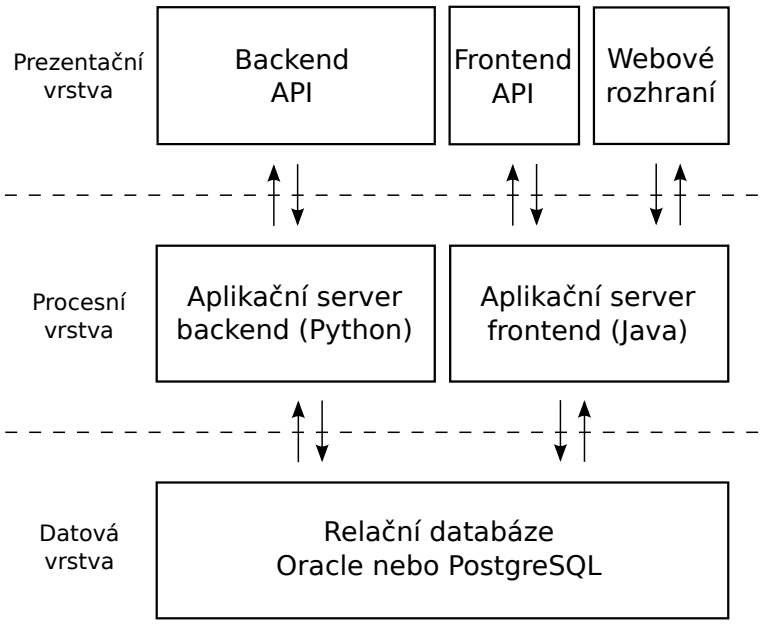
V systému Spacewalk odpovídá datové vrstvě relační databáze. Ve verzi Spacewalku 1.2, byla doporučenou relační databází databáze Oracle verze 10g nebo 11g. Lze použít i relační databázi PostgreSQL, ale její podpora není stoprocentní a mohou se při jejím použití objevit

[1] Viz https://fedorahosted.org/spacewalk/wiki/HowToInstall

chyby, ke kterým s databází Oracle nedochází. Dle doporučení v instalačních instrukcích byla nainstalována relační databáze Oracle verze 10g.

Procesní vrstvě odpovídají aplikační servery postavené nad relační databází. Záměrně je zde použito množné číslo, protože lze zjednodušeně říci, že tato vrstva obsahuje dva aplikační servery. Jeden napsaný v programovacím jazyce Java obsluhující frontend systému. A druhý v jazyce Python obsluhující backend. Pro pojmy frontend a backend neexistují jednoznačné české ekvivalenty, proto budou v tomto textu uváděny v původní formě v anglickém jazyce. Význam těchto pojmů v kontextu systému Spacewalk vychází z typu procesů, které aplikační server nabízí. Frontend procesy slouží správci systémů ke změnám na spravovaných operačních systémech a k získávání informací o nich. Nejčastěji jsou frontend procesy iniciovány skrz webové rozhraní pomocí prohlížeče. Backend procesy nejsou oproti frontend procesům správci dostupné přímo, ale jsou využívány dalšími součástmi systému Spacewalk – například terminálovými utilitami určenými k registraci operačních systémů.

Prezentační vrstva v systému Spacewalk obsahuje primárně webové rozhraní využívající technologie HTML, CSS a JavaScript. Webové rozhraní ovládá správce pomocí webového prohlížeče. Kromě webového rozhraní obsahuje prezentační vrstva i XML-RPC rozhraní, se kterým komunikují jiné služby a skripty. Celé schéma třívrstvé architektury v systému Spacewalk je znázorněno na obrázku 2.1.



Obrázek 2.1: Třívrstvá architektura v systému Spacewalk

2.1.2 Rozhraní XML-RPC

Technologie XML-RPC je sadou nástrojů, které s využitím existujících standardů vytvořených pro komunikaci na Internetu nabízí možnost komunikace mezi dvěmi aplikacemi. Existujícími standardy je myšlen značkovací jazyk XML (zkratka Extensible Markup Language), protokol HTTP (zkratka Hypertext Transfer Protocol) a standard RPC (zkratka Remote Procedure Call). XML poskytuje slovník pro popis vzdálených volání RPC přenášených mezi aplikacemi protokolem HTTP [11]. Díky tomu mohou být komunikující aplikace implementovány v různých programovacích jazycích a na různých platformách – stačí když

podporují protokol HTTP a umí zpracovávat jazyk XML. Důležitou vlastností XML-RPC je fakt, že abstrahuje datové typy jednotlivých programovacích jazyků a dovoluje tak přenos dat mezi dvěma aplikacemi, které využitý datový typ implementují rozdílně.

V komunikaci pomocí XML-RPC hraje vždy jedna z aplikací roli klienta a druhá roli serveru. Sekvence akcí prováděných během jednoho volání vzdálené procedury klientem je následující [11]:

1. Klientská aplikace zavolá proceduru pomocí XML-RPC klienta předávajíc název vzdálené procedury, její parametry a adresu vzdáleného XML-RPC rozhraní.
2. XML-RPC klient zabalí název vzdálené procedury s jejími parametry do XML a výsledek vloží jako obsah do HTTP POST dotazu. Dotaz poté odešle na adresu vzdáleného XML-RPC rozhraní.
3. HTTP server obsluhující adresu vzdáleného rozhraní přijme POST dotaz a jeho obsah předá XML-RPC serveru.
4. XML-RPC server zpracuje předaná data v jazyce XML a získá z nich jméno procedury a parametry. Parametry využije při volání procedury daného jména.
5. Procedura vrátí výsledek XML-RPC serveru, který výsledek zabalí pomocí XML a zabalený pošle HTTP serveru.
6. Z HTTP serveru se odešle klientovi HTTP POST odpověď obsahující data v jazyce XML.
7. XML-RPC klient zpracuje odpověď a výsledek volání vzdálené procedury vrátí zpět klientské aplikaci.
8. Klientská aplikace pokračuje dál ve vykonávání programu pracujíc se získaným výsledkem.

V systému Spacewalk existují dvě XML-RPC rozhraní, ke kterým lze vzdáleně přistupovat. Jedno obsluhuje frontend aplikační server a druhé backend aplikační server. Označují se proto jako frontend API a backend API[2]. Nachází se v prezentační vrstvě systému a komunikují s jinými aplikacemi a službami. Jejich poloha je naznačena na obrázku 2.1.

Aby bylo možné frontend API a backend API rozlišovat na straně klienta, má každé rozhraní svou vlastní adresu:

- Frontend API: `http://domena/rpc/api`
- Backend API: `http://domena/XMLRPC`

Konkrétní doména, v adrese nahrazená řetězcem `domena`, na které Spacewalk běží, závisí na nastavení stroje na němž je systém nainstalován.

Frontend API funguje jako alternativa k obsluze systému Spacewalk webovým prohlížečem. Lze říct, že téměř všechny akce proveditelné prostřednictvím webového prohlížeče lze také vykonat vhodnou sekvencí volání Frontend API. Volání se sdružují do jmenných prostorů dle oblasti, se kterou pracují. Příkladem budiž jmenný prostor `api` sloužící k přístupu k metadatům Frontend API – informacím jako seznam všech dostupných

[2]Zkratka API (z angl. Application Programming Interface) v překladu znamená rozhraní pro programování aplikací.

volání nebo verze systému. Je vhodné zmínit, že u většiny volání se mezi povinnými parametry vyskytuje parametr `sessionKey`. Tento „klíč sezení“ lze získat přihlášením pomocí volání `auth.login` (metody `login` z jmenného prostoru `auth` a je paralelou k autentizaci uživatele přihlašovacím formulářem při přístupu webovým prohlížečem. Dokumentaci jednotlivých volání obsahuje každá instalace systému Spacewalk a je dostupná webovým prohlížečem bez nutnosti přihlášení. Obsahuje krátký popis volání se seznamem povinných i nepovinných parametrů. Ve verzi Spacewalku 1.2 se dokumentace nachází na adrese `http://domena/rhn/apidoc/index.jsp`.

Jak už bylo zmíněno, backend nabízí procesy sloužící k interní komunikaci nástrojů systému. Rozhraní backend API má dokumentaci pouze v komentářích samotného obslužného kódu a neposkytuje metadata o voláních.

2.1.3 Aplikační servery

Aplikační servery jsou v systému dva. Server Apache Tomcat (dále pouze Tomcat) zpracovávající požadavky na frontend kódem v jazyce Java a Apache HTTP Server Project (dále pouze Httpd) předavající požadavky na backend ke zpracování kódem v jazyce Python. Oba ale nemohou zároveň naslouchat na portech 80 (standardní port protokolu HTTP) a 443 (standardní port protokolu HTTPS, zabezpečeného HTTP). Instalace v operačním systému Fedora 14 řeší problém modulem `mod_jk` serveru Httpd, který umí vybrané HTTP požadavky protokolem AJPv13[3] předávat serveru Tomcat [5]. Server Tomcat naslouchá na jiných portech a zpracovává příchozí požadavky přijaté protokolem AJPv13.

Na obrázku 2.2 s výpisem všech naslouchajících služeb lze oba spuštěné servery identifikovat. Červeně jsou zvýrazněny procesy serveru Httpd a modře procesy serveru Tomcat. Server Httpd naslouchá na portech 80 a 443. Pokud detekuje HTTP/HTTPS požadavek pro Tomcat, pošle mu ho protokolem AJPv13 na port 8009. Server Tomcat zároveň přijímá požadavky na portu 8080, nicméně je pouze přesměrovává na port 80.

Server Tomcat naslouchá ještě na třetím portu, tzv. „shutdown portu“, nesoucím číslo 8005. Název „shutdown port“ vychází z jeho chování. Pokud přijme zprávu `SHUTDOWN`, zastaví server. A jediným omezením této funkčnosti je IP adresa odesílatele, která musí být lokální. Konkrétní chování podléhá konfiguraci. Podrobnější informace o portu obsahuje dokumentace [3].

Kromě organizace samotných serverů je důležité i zmínit, jak je spouštěn samotný obslužný kód – ať už v programovacím jazyce Java nebo Python.

Kód v jazyce Java je serverem Tomcat načten při startu ve skompilované podobě a navíc v Java archivu. Java archiv (zkráceně JAR) je formát založený na rozšířeném formátu ZIP určený k agregaci skompilovaných Java tříd a dalších souvisejících dat do jednoho souboru [17]. Konkrétní obslužný kód vykonávaný pro příchozí požadavek určují částečně konfigurační soubory a částečně samotný kód.

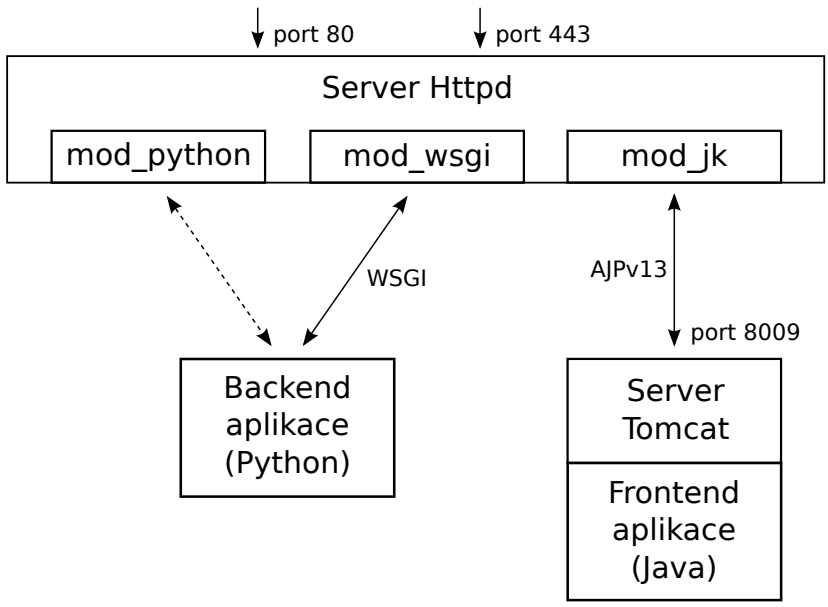
S kódem v jazyce Python je to trochu složitější. Aby mohl HTTP/HTTPS požadavky zpracovávat kód jazyka Python, musí mít server Httpd některý z vhodných modulů. V případě mé instalace byl využit modul `mod_wsgi`, který vykonává kód aplikace dle specifikace WSGI [9]. Ten v praxi ale pouze překládá přicházející požadavky do formátu, se kterým pracuje modul `mod_python`. Aplikace tedy umí reagovat na požadavky z obou modulů. Důvodem je funkčnost systému Spacewalk na více platformách – při instalaci v OS RHEL 5 se použije modul `mod_python` a v OS Fedora modul `mod_wsgi` [21].

[3] Viz http://tomcat.apache.org/connectors-doc/ajp/ajpv13a.html

```
Active Internet connections (only servers)
Proto Recv-Q Send-Q Local Address               Foreign Address             State       PID/Program name
tcp        0      0 0.0.0.0:5347                0.0.0.0:*                   LISTEN      1479/router
tcp        0      0 0.0.0.0:5222                0.0.0.0:*                   LISTEN      1493/c2s
tcp        0      0 0.0.0.0:111                 0.0.0.0:*                   LISTEN      880/rpcbind
tcp        0      0 0.0.0.0:34704               0.0.0.0:*                   LISTEN      1323/xe_d000_XE
tcp        0      0 0.0.0.0:1521                0.0.0.0:*                   LISTEN      1296/tnslsnr
tcp        0      0 0.0.0.0:57362               0.0.0.0:*                   LISTEN      969/rpc.statd
tcp        0      0 0.0.0.0:5269                0.0.0.0:*                   LISTEN      1501/s2s
tcp        0      0 127.0.0.1:631               0.0.0.0:*                   LISTEN      1078/cupsd
tcp        0      0 127.0.0.1:25                0.0.0.0:*                   LISTEN      1355/sendmail: acce
tcp        0      0 127.0.0.1:25151             0.0.0.0:*                   LISTEN      1696/python
tcp        0      0 0.0.0.0:9055                0.0.0.0:*                   LISTEN      1296/tnslsnr
tcp        0      0 127.0.0.1:32000             0.0.0.0:*                   LISTEN      1576/java
tcp        0      0 ::ffff:127.0.0.1:8005       :::*                        LISTEN      1424/java   Shutdown port
tcp        0      0 ::ffff:127.0.0.1:8009       :::*                        LISTEN      1424/java   AJPv13
tcp        0      0 ::ffff:127.0.0.1:2828       :::*                        LISTEN      1576/java
tcp        0      0 :::111                      :::*                        LISTEN      880/rpcbind
tcp        0      0 ::ffff:127.0.0.1:8080       :::*                        LISTEN      1424/java   HTTP
tcp        0      0 :::80                       :::*                        LISTEN      1465/httpd  HTTP
tcp        0      0 ::1:631                     :::*                        LISTEN      1078/cupsd
tcp        0      0 :::60856                    :::*                        LISTEN      969/rpc.statd
tcp        0      0 :::443                      :::*                        LISTEN      1465/httpd  HTTPS
```

Obrázek 2.2: Zjištění naslouchajících služeb pomocí příkazu `netstat -tlnp`.

Celé schéma organizace aplikačních serverů včetně výše popsaných vazeb k aplikacím (tj. vazeb ke kódu) je na obrázku 2.3.



Obrázek 2.3: Schéma organizace aplikačních serverů a vazeb k aplikacím

Jistým druhem komunikace s aplikačním serverem jsou také soubory s chybovými záznamy. Informace se v této komunikaci sice pohybují jen směrem od aplikačního serveru, ale je to důležitý doplňující zdroj údajů o případné chybě. Pokud k chybě v obslužném kódu dojde, většinou se protokolem HTTP/HTTPS navrátí pouze chybový kód s omluvným textem a podrobnosti problému jsou zapsány do onoho chybového záznamu. Implicitně instalace v systému Fedora 14 umísťuje záznamy do následujících cest:

- Aplikační server Httpd: `/var/log/httpd/error_log`
- Aplikační server Tomcat: `/var/log/tomcat6/catalina.out`

2.2 Modely a kritéria testování

Text v této kapitole vychází z knihy [1]. Kniha obsahuje mimo jiné i formální definice všech pojmů použitých dále.

Testování je jednou s technik, jak ověřovat funkčnost a správnost chování programů. Testování lze provádět jak ručně, tak automatizovaně. Pro ruční testování není zapotřebí žádných speciálních nástrojů a tester si s programem pouze „hraje“ a sám vyhodnocuje, zda se program chová správně. Automatizované testování naproti tomu vyžaduje specifikaci chování ve strojově čitelném formátu tak, aby se při spuštění testů dala ověřovat.

Ač jde tvořit testy (resp. specifikace chování) bez nějakého plánu nebo cíle, nemají většinou takové testy smysl. Mohou sice ověřovat nějakou důležitou část programu, nelze o nich ale prohlásit, zda jsou k něčemu dobré nebo ne. Zda testují vše nebo třeba jen několik procent funkčnosti. Aby něco takového bylo možné prohlásit, je nutné testy a sady testů spojit s *pokrytím*.

Testy mohou pokrývat například případy užití programu, části formální specifikace nebo zdrojový kód programu. K vyhodnocování pokrytí tyto součásti abstrahujeme do modelů, se kterými můžeme pracovat pomocí příslušných pravidel.

V rámci abstraktního modelu pak definujeme *kritéria* a *požadavky* na splnění kritérií. S požadavky se dá následně prohlásit, zda je testy splnily a zda některé testy nesplňují požadavky duplicitně.

Existují obecně 4 typy abstraktních modelů do kterých můžeme některou z částí programu převést. Pro každý model uvedu v jim určených kapitolách ta nejdůležitější kritéria ve vztahu k testovanému systému. Čtyři typy abstraktních modelů jsou tyto:

- Grafové modely,
- modely s logickými výrazy,
- modely dělící množinu vstupních hodnot
- a modely se syntaktickými popisy.

2.2.1 Grafové modely

Grafové modely jsou obecně nejrozšířenějším a nejčastěji využívaným typem abstraktního modelu. Jde o vytvoření grafu z některého pohledu na testovaný program a stanovení požadavků na jeho pokrytí. Pohledem na program není myšlen vždy jen zdrojový kód, ale třeba i specifikace, prvky systému nebo případy užití. Stačí, když z takového pohledu lze vytvořit orientovaný graf.

Graf se skládá z uzlů a hran mezi uzly. V orientovaném grafu nese navíc každá hrana informaci o tom, který z hranou spojených uzlů je výchozí a který cílový. V grafovém modelu programu uzly často reprezentují stavy programu a orientované hrany přechody mezi nimi.

Kritérií pro pokrytí grafu existuje mnoho. Dvě základní jsou: *pokrytí všech uzlů* (angl. node coverage) a *pokrytí všech hran* (angl. edge coverage). Další vyžadují definice nových pojmů a pracují s různými průchody grafem. Pro potřeby této bakalářské práce si ale vystačím jen s těmi základními.

Pro splnění kritéria pokrytí všech uzlů grafu musí být stanoveny takové požadavky na testy, aby alespoň jednou navštívily každý dostupný uzel grafu. Dostupným uzlem je myšlen uzel, který lze navštívit alespoň jedním průchodem grafu z jednoho z počátečních uzlů do jednoho z koncových uzlů. Kritérium pokrytí všech hran grafu je analogicky splněno

v případě, že jsou požadavky na testy stanoveny tak, aby navštívily všechny dostupné hrany grafu.

2.2.2 Modely s logickými výrazy

Pokud jsem se u grafových modelů zabýval abstrakcí do grafů, budu se v této podkapitole snažit popsat abstrakci k logickým výrazům. V praxi to například může znamenat získání všech logických výrazů ze zdrojového kódu a následné vytvoření testů dle požadavků vycházejících z vybraného kritéria pokrytí.

U logických výrazů existují tři základní kritéria. *Pokrytí predikátů*, *pokrytí klauzulí* a *kombinatorické pokrytí*.

Predikátem je myšlen celý logický výraz složený z klauzulí spojených logickými operátory. *Klauzule* je část logického výrazu, která je z logického hlediska dále nedělitelná (tj. neobsahuje další logické operátory). Pokud není v predikátu použit žádný operátor, může predikát obsahovat pouze jednu klauzuli. V takovém případě lze nazvat logický výraz jak klauzulí, tak predikátem. V případě logického výrazu $(A \wedge B) \vee C$ je predikátem celý výraz, klauzule však obsahuje tři: proměnné A, B a C.

Kritérium pokrytí predikátů klade požadavky na testy tak, aby byly všechny predikáty vyhodnoceny s oběma logickými výsledky – pravdou i nepravdou. Druhé kritérium pokrytí klauzulí pracuje se stejnou podmínkou jako kritérium pokrytí predikátů, jen jsou předmětem požadavků klauzule. Všechny klauzule musí být v testech jak s logickým výsledkem pravda, tak nepravda.

Poslední kritérium kombinatorického pokrytí rozšiřuje předešlé kritérium tím, že ke splnění požaduje vyhodnocení všech predikátů se všemi kombinacemi logických hodnot jejich klauzulí.

2.2.3 Modely dělící množinu vstupních hodnot

Cílem přístupu dělícího množinu vstupních hodnot je rozdělit konkrétní vstupní hodnoty do skupin, ve kterých si jsou hodnoty ve vztahu k chování systému rovny. Pro pokrytí všech možných vstupních hodnot pak stačí vybrat jen jednu hodnotu z každé skupiny.

Model, ze kterého se v tomto případě vychází, se nazývá *model vstupních hodnot* (angl. input domain model). Pro jeho vytvoření je nutné definovat *charakteristiky* dělící rozsahy vstupních hodnot do *bloků*. Skupina hodnot patřící do bloku si je vůči definované charakteristice rovna. Bloky získané rozdělením tvoří *oddíl* a je po nich požadováno, aby byly v rámci oddílu vzájemně disjunktní a aby při sjednocení vyplňovaly celý rozsah vstupních hodnot rozdělených charakteristikou.

Testování funguje na základě výběru sady vstupních hodnot. Sada je tvořena jednou hodnotou z každé charakteristiky. Pro získání všech sad k testování se musí kombinovat hodnoty ze všech bloků jednotlivých charakteristik. Kritérium požadující testy se všemi takto získanými sadami vstupních hodnot se nazývá *pokrytí všech kombinací* (angl. all combinations coverage). Jeho splnění je však kvůli obrovskému množství sad velmi náročné. Další kritéria proto podmínky zjednodušují a omezují tím počet sad vstupních hodnot, které je nutné testovat.

Počet sad vstupních hodnot ovlivňuje ale také metoda použitá pro identifikaci charakteristik. Existují dvě: jedna vycházející z rozhraní (angl. interface-based input domain modelling) a druhá vycházející z funkcionality (angl. functionality-based input domain modelling).

První z metod pro identifikaci charakteristik analyzuje testovaná rozhraní programu a pro každý parametr rozhraní vytváří jednu charakteristiku. Druhá metoda vychází z funkčnosti popsané ve specifikaci nebo případech užití a generuje charakteristiky z definovaného chování.

2.2.4 Modely se syntaktickými popisy

Poslední typ abstraktního modelu vychází ze syntaktických popisů. Těmi je zde myšlena *gramatika*. Gramatika se získává jak ze zdrojového kódu programu, tak například z jeho specifikací.

Kritéria pro gramatiku vychází z odvozovacích pravidel, které ji definují. Jedno z kritérií zvané *pokrytí terminalních symbolů* požaduje aby testy obsahovaly všechny terminální symboly gramatiky. Jiné, zvané *pokrytí odvozovacích pravidel*, například vyžaduje v testech použití všech odvozovacích pravidel.

2.3 Shrnutí analýzy

Část zabývající se problematikou modelů a kritérií testování naznačila, že pro vytváření testů je nejdříve potřeba měřítko. Měřítko, podle kterého může být hodnocena kvalita testů a díky kterému se dají najít neotestované části systému. Ač bakalářská práce v zadání zmiňuje návrh pouze a jen testů, považuji za nezbytné nejdříve navrhnout a implementovat sadu nástrojů (dále jen *knihovnu*), která by vykonávala roli zmiňovaného měřítka.

Požadavek na knihovnu je pouze jeden, zato velmi důležitý: umět vyhodnotit vybrané kritérium pro vytvořené testy.

Požadavky na testy, které vycházejí ze zadání bakalářské práce a znalostí nabytých během analýzy jsou následující:

1. Testy musí být začleněné do tématicky zaměřených množin.
2. Musí být umožněno spuštění celé množiny nebo i podmnožiny kompletní sady testů.
3. Výstup testů by měl být srozumitelný pro člověka, ale také strojově čitelný pro další zpracování.
4. Uchovávat dobu trvání každého jednotlivého testi pro pozdější zpracování.
5. Možnost spouštět některé testy jen na určitých verzích systému Spacewalk.
6. Sledovat při testování přírůstky v záznamech chyb aplikačních serverů.

Kapitola 3

Návrh knihovny

Kapitola návrhu knihovny se zabývá vytvořením nástrojů a prostředí, ve kterém by bylo možné následně testy vytvářet. Protože se práce s testy dá obecně rozdělit do dvou fází – v první testy vytváříme a druhé je opakovaně spouštíme – vycházím z takového rozdělení i při návrhu knihovny. Informaci, zda testy splňují požadavky kritéria pokrytí, totiž není potřeba znát při spouštění testů a vyhodnocování jejich úspěšnosti. Požadavky na kritéria pokrytí jsou praktické hlavně ve fázi vytváření.

Další popis návrhu je proto rozdělen do dvou podkapitol. Každá reflektující jeden modul, jednu část knihovny. První modul se stará jen o prostředí pro testy, jejich spouštění a vyhodnocování výsledků. Modul druhý se zaobírá zprvu výběrem vhodného kritéria pokrytí a následně návrhem jeho implementace.

3.1 Modul s testy

Modul s testy má obsahovat pouze samotné testy a splňovat požadavky sepsané v kapitole 2.3. Je nutné navrhnout prostředí, které usnadní tvorbu testů a skripty obsluhující jejich spouštění.

3.1.1 Knihovna pro psaní testů

Výběr knihovny pro psaní testů ovlivnil převážně fakt, že bude výsledek dále využíván ve firmě Red Hat. Ta si vyvinula vlastní testovací knihovnu na úrovni interpretu linuxových příkazů pro Bash poskytující funkce projekt jednodušší psaní, spouštění a analýzu integračních a *black-box* testů s názvem BeakerLib [14].

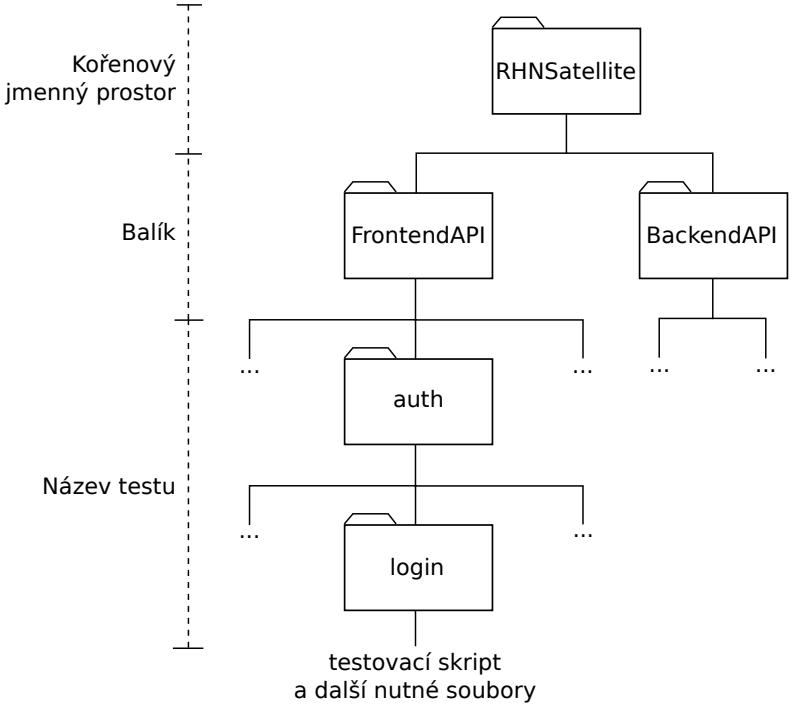
Nevýhodou knihovny je absence XML-RPC klienta skrz který by se dalo testovat rozhraní systému Spacewalk. Například ve standardní knihovně jazyka Python existuje modul `xmlrpclib` zajišťující funkcionalitu XML-RPC klienta a při použití testovací knihovny `unittest` by tak byl dostupný. Knihovna BeakerLib však nabízí možnost rozšíření a s vědomím, že z kódu interpretu Bash lze volat skripty v jiných jazycích, existuje cesta, jak využít implementaci klienta XML-RPC právě z jazyka Python.

Z uvedené nevýhody se ale může stát i výhoda. Oddělením skriptů zapouzdřujících XML-RPC volání od testů se nabízí možnost použití i mimo tento projekt. Skripty pouze propagují, stávají se adaptérem, XML-RPC rozhraní.

Kromě uvedené „nevýhody“ se ale knihovna BeakerLib k řešení hodí. Základní myšlenka testů založených na knihovně dělí test do fází. Předpřipraveny jsou funkce pro tři fáze: přípravnou (angl. setup phase), testovací (angl. test phase) a úklidovou (angl. cleanup

phase). V přípravné fázi má dojít k zálohám a nastavením nutným pro dosažení prostředí, ve kterém se bude testovat. Následují testovací fáze, kde se ověřuje chování testovaného programu. Fáze úklidová nakonec vrátí změny provedené ve fázích předchozích do stavu před spuštěním. Další funkčnost je popsána ve vztahu k požadavkům uvedeným v kapitole 2.3 – ty knihovna buď přímo řeší nebo nabízí cestu, jak požadavek splnit rozšířením nebo externím skriptem.

Požadavek číslo 1 zmiňuje rozdělení testů do tématických množin. Knihovna BeakerLib pracuje s termínem balík a dovoluje více testů zahrnout do jednoho balíku. V rámci balíku se navíc dají testy členit adresářovou strukturou. Jeden test (resp. jeden testovací skript) je vždy v samostatném adresáři. Cestu do adresáře s testem určuje název testu. Název včetně názvu balíku je pak uveden ve vyhodnocení. Ve vztahu k rozhraní XML-RPC systému Spacewalk se nabízí rozdělení testů do dvou balíků podle toho, zda jde o frontend nebo backend rozhraní. V rámci balíku by první úroveň adresářové struktury mohla vyjadřovat název jmenného prostoru testovaného volání. Druhou úroveň by tvořil název volání. Oba balíky by nakonec zastřešoval kořenový jmenný prostor podle testované aplikace. Na obrázku 3.1 je ilustrována adresářová struktura s umístěním testu pro frontend volání `auth.login` (volání `login` z jmenného prostoru `auth`).



Obrázek 3.1: Umístění testovacího skriptu pro frontend volání `auth.login`

Splnit požadavek číslo 2 s knihovnou také není problém, protože každý test je spustitelný skript. Vytvořením externího skriptu pouze spouštějícího testy lze zaručit spuštění celé množiny stejně jako jen určité podmnožiny kompletní sady testů.

V průběhu vyhodnocování testu knihovnou BeakerLib se zaznamenávají informace do tzv. *žurnálu*. Jsou v něm víceméně zaznamenány všechny kroky provedené v testu a informace o jejich úspěšnosti. Po vykonání všech kroků testu se z žurnálu generuje výstup ve ze dvou formátech – textovém a v jazyce XML – a je uložen v dočasném úložišti závislém na nastavení operačního systému. Textový formát je lidsky čitelný, zatímco formát XML jde

s jednotnou strukturou naproti strojovému zpracování. Tím knihovna BeakerLib splňuje i požadavek číslo 3.

Požadavek číslo 4 zmiňuje uchovávání doby trvání testu pro další použití. Žurnál knihovny BeakerLib měří dobu, jak dlouho se test vykonával. Externí skript spouštějící sady testů musí tedy hodnotu doby trvání z každého testu ukládat a například vytisknout na výstup.

Poslední dva požadavky s čísly 5 a 6 jsou ty, pro které nemá knihovna BeakerLib zabudovanou podporu. Vhodným rozšířením se ale dá uvedených funkčností dosáhnout.

Pro kontrolu verze je nutné znát verzi testovaného systému Spacewalk. Ač je tato informace zjistitelná frontend voláním `api.systemVersion`, zavádělo by to závislost i testů backend rozhraní na volání z frontendu. Verze testovaného systému Spacewalk tedy musí být konfigurovatelná. Podmínka rozhodující o spuštění testu může následně porovnat verzi, pro kterou je test určen, s testovanou verzí.

Sledování záznamů chyb aplikačních logů v praxi znamená dvě akce. První akce uloží aktuální stav záznamů na začátku testu a druhá provedená na konci testu získá přírůstek porovnáním stavu aktuálního se stavem uloženým. Neprázdný přírůstek by měl být označen za chybu a zanesen do záznamu o vyhodnocení testu.

3.1.2 Společná konfigurace

Kromě zmíněné verze testovaného systému Spacewalk existují ještě další tři údaje, které jsou na konkrétním testovaném systému závislé: doménové jméno a administrátorské uživatelské jméno a heslo. Doménovým jménem je myšlena část URL patřící zařízení na síti s testovaným systémem a slouží k jeho identifikaci. Přihlašovací údaje administrátora jsou nutné pro většinu frontend volání vyžadujících přihlášení platného uživatele.

Formát konfiguračního souboru musí být jednoduše čitelný jak z testů v jazyce interpretu Bash, tak ze skriptů s XML-RPC klientem v jazyce Python. V testech je potřeba informace o verzi systému a ve skriptech doménové jméno a přihlašovací údaje administrátora.

Jazyk Python má pro konfigurační soubory vlastní modul ve standardní knihovně nazvaný `ConfigParser`. Interpret Bash nic konkrétního nevyužívá, ale dokáže zpracovat soubory s jednodušší syntaxí pomocí terminálových utilit. Vhodnou syntaxí by proto mohla být podmnožina možností popsaných v dokumentaci modulu `ConfigParser` [18].

3.2 Modul vyhodnocující pokrytí

Na modul vyhodnocující pokrytí je z hlediska použití kladen pouze jediný, z jeho názvu zřejmý požadavek – aby bylo možné vyhodnotit pokrytí dle jednoho z kritérií. První část návrhu modulu se proto zabývá výběrem kritéria s důvody, které k výběru vedly. S vybraným kritériem pracuje druhá část popisující návrh implementace nástrojů pro jeho vyhodnocování.

3.2.1 Výběr kritéria pokrytí

Aby mohlo být vybráno kritérium, jehož pokrytí bude možné následně vyhodnocovat, je nejdříve nutné vybrat typ abstraktního modelu. Každý z typů modelů představený v podkapitole 2.2 tvoří svůj model a vyhodnocuje jeho pokrytí jiným způsobem. Záleží proto, z čeho se při vytváření modelu vychází a jak je náročné vyhodnocení míry pokrytí.

Jak už bylo zmíněno v analýze, rozhraní XML-RPC nemá specifikaci ani případy užití. Abstrahovat model tak lze jen přímo ze zdrojového kódu. Tento fakt přímo nevyřazuje

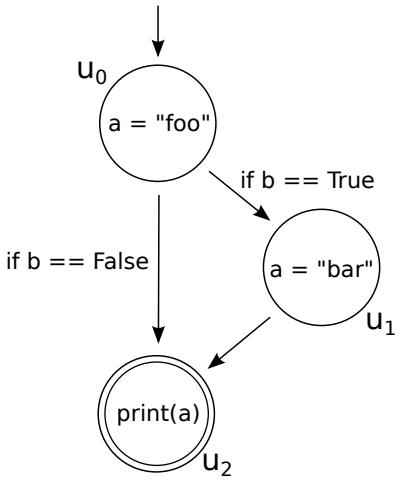
žádný z typů modelů, ale souvisí s náročností vyhodnocení míry pokrytí.

Vyhodnocení musí být proveditelné automatizovaně a jednoduše, aby při psaní testů pomáhalo a nepřekáželo. A tomu napomáhají nástroje. Pro nejrozšířenější typ abstraktních modelů – grafové modely – se hledají nástroje snadno. Pro ostatní typy modelů s logickými výrazy a modelů se syntaktickým popisem by bylo nutné nástroj vytvořit. Proti použití modelu s děděním vstupních hodnot stojí navíc ještě fakt, že chování jednotlivých XML-RPC volání ovlivňují nejen jim předané parametry, ale i mnoho globálních nastavení. Problémem by bylo najití všech vstupních parametrů. Grafové modely proto vyhovují podmínkám nejlépe.

Konkrétní výběr kritéria z grafových modelů je dán tzv. sílou kritérií a, podobně jako výběr typu modelu, možnostmi nástrojů. Sílou kritéria je myšlena identifikace cest vykonávání programu. Příkladem budiž krátký úsek zdrojového kódu 3.1. Jeho abstraktní grafový model ilustruje obrázek 3.2, ve kterém uzly u_0, u_1 a u_2 reprezentují jednotlivé příkazy úryvku zdrojového kódu a orientované hrany možný postup vykonávání příkazů. Při aplikaci kritéria pokrytí všech uzlů grafu stačí graf projít jednou s logickou hodnotou pravda v proměnné `b`, tj. cestou u_0, u_1, u_2. Pro pokrytí všech hran grafu jsou ale nutné dva průchody lišící se hodnotami v proměnné `b`. Jeden průchod cestou u_0, u_1, u_2 a druhý cestou u_0, u_1. Na tomto případu je vidět, že kritérium pokrytí všech hran je silnější než pokrytí všech uzlů, tj. že pokrývá více možností, jak může být zdrojový kód vykonáván.

```
1 a = "foo"
2 if b:
3     a = "bar"
4 print(a)
```

Zdrojový kód 3.1: Krátký úsek kódu pro demonstraci síly kritérií



Obrázek 3.2: Abstraktní grafový model zdrojového kódu 3.1

Existují i další kritéria, která jsou silnější než pokrytí všech hran, ty ale často nejsou podporovány nástroji pro vyhodnocování pokrytí. Pro jazyk Python doporučují sami vývojáři jazyka Python nástroj `coverage.py` [19], který měří pokrytí z pohledu kritéria všech uzlů grafu (tj. všech příkazů, angl. statement coverage) nebo z pohledu kritéria všech hran grafu (tj. všech větví programu, angl. branch coverage) [6].

Z nekomerčních nástrojů pro jazyk Java jsou nejčastěji zmiňovány nástroje Emma a Cobertura. Nástroj Cobertura se však z pohledu měřitelných kritérií jeví lepší, protože nabízí měření pokrytí všech hran grafu [8].

Vhodným kritériem se proto stalo pokrytí všech hran grafu zdrojového kódu.

3.2.2 Návrh měření pokrytí kódu jazyka Python

K měření pokrytí nedochází automaticky při každém vykonávání obslužného kódu jazyka Python. Doporučovaný nástroj `coverage.py` se musí explicitně instruovat k započetí měření. Protože se ale nespouští obslužný kód při XML-RPC volání přímo, ale prostřednictvím aplikačního serveru, nelze před voláním jednoduše měření zapnout a po obdržení výsledku zase vypnout. Knihovna nabízí dvě možnosti jak tento problém vyřešit. První možností je upravit obslužný kód tak, aby na začátku vykonávání začal s měřením a na konci skončil a výsledky uložil. Druhá možnost spočívá v instalaci krátkého kusu kódu do inicializační rutiny interpretu jazyka Python, který automaticky zapne měření v případě, že je detekována speciální proměnná prostředí.

Ze zmíněných možností preferuji úpravu obslužného kódu, protože nezasahuje do globálního nastavení interpretu jazyka Python. Nelze sice přímo upravit části kódu systému, ale díky modulům `mod_python` a `mod_wsgi` aplikačního serveru Httpd spouštějícím aplikaci se nabízí řešení implementace měření jako *middleware*.

Pojem middleware, který nemá svůj jednoznačný český ekvivalent, je popsán ve specifikaci WSGI [9] jako objekt, který vzhledem k aplikaci hraje roli serveru a vzhledem k serveru roli aplikace. Přítomnost objektu je pro server i aplikaci transparentní a vyžaduje nastavení serveru tak, aby místo aplikace spustil middleware. Middleware následně převezme roli serveru a spustí původní aplikaci. Aplikace může být i další middleware zapouzdřující aplikaci a tvořící tzv. *middleware zásobník* (angl. middleware stack). Obrázek 3.3 ilustruje middleware zásobník nad aplikací komunikující s aplikačním serverem pomocí WSGI.

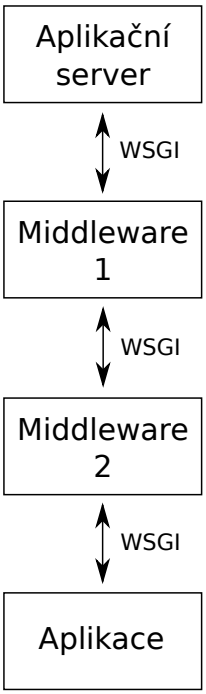
Ačkoli se popis middleware nachází ve specifikaci WSGI, je takové řešení aplikovatelné i při komunikaci skrz modul `mod_python`. Liší se pouze v protokolu, kterým si mezi sebou aplikační server a aplikace předávají informace.

Při instalaci modulu pro měření pokrytí se proto změní konfigurace serveru tak, aby byl spouštěn middleware starající se o pokrytí. Při přijetí požadavku zapne měření a pošle požadavek aplikaci. Aplikace vykoná potřebné akce a vrátí odpověď zpět do middleware. Middleware nakonec vypne měření, uloží naměřená data pro další zpracování a pošle odpověď serveru.

3.2.3 Návrh měření pokrytí kódu jazyka Java

Měření pokrytí kódu jazyka Java funguje na jiném principu. Místo zapínání a vypínání měření se do samotného obslužného kódu vloží měřící příkazy. Měřící příkazy zaznamenávají průběh vykonávání kódu a umí získaná data ukládat mimo aplikaci pro další analýzu. Proces úpravy existujícího kódu za účelem sběru dat se nazývá *instrumentace* [16].

V případě knihovny Cobertura použité pro měření pokrytí se instrumentace provádí s již skompilovaným kódem (tzv. bytecode). V instalaci systému Spacewalk se všechen skompilovaný kód nachází v jednom JAR archivu. Proces instalace modulu pro měření pokrytí tedy musí zahrnovat instrumentaci daného JAR archivu. Dostupné jsou tři cesty jak instrumentační nástroj použít: pomocí skriptu, pomocí *úkolu* (angl. task) nástroje Apache Ant nebo pomocí rozšíření nástroje Apache Maven. Protože RPM balíček `cobertura` v OS Fedora 14 obsahuje rozšíření nástroje Apache Ant (dále jen jako Ant) definující závislosti na jiných



Obrázek 3.3: Middleware zásobník mezi aplikačním serverem a aplikací

knihovnách, zdá se použití nástroje Ant k instalaci vhodné. Ant je knihovna jazyka Java a terminálový nástroj sloužící ke spouštění procesů, které jsou popsané v konfiguračních souborech [2]. Stačí implementovat konfigurační soubor popisující proces instrumentace. Při instrumentaci skriptem by totiž musely být explicitně určeny všechny knihovny, které Cobertura pro svůj běh vyžaduje.

Problém, který ale použití knihovny Cobertura přináší, se projeví při samotném měření. Instrumentovaný kód ukládá své statistiky do struktur v rámci aplikace a aby mohly být statistiky uloženy do souboru, musí se aplikační server Tomcat restartovat.

3.2.4 Společná konfigurace

Podobně jako pro modul s testy i pro modul pro měření pokrytí potřebuje konfiguraci nastavení, která se mohou v různých instalacích měnit. Jde především o cesty ke všem ovlivněným souborům – instrumentovaný JAR archiv, konfigurace serveru Httpd, nebo datové soubory, kam mají být data a statistiky z měření ukládány.

Nastavení z konfiuračního souboru jsou potřeba ve dvou prostředích. Jedním je popis procesů prováděných nástrojem Ant a druhým kód middleware v jazyce Python. Ant podporuje konfigurační soubory z jazyka Java (angl. properties files) [4]. Jejich formát je podobný tomu, který popisuje standardní modul jazyka Python `ConfigParser` [18], ale nedá se říci, že by jeden z formátů byl podmnožinou toho druhého. Rozhodl jsem se proto pro použití podmnožiny obou formátů tak, aby byla konfigurace čitelná z obou prostředí.

3.2.5 Obsluha modulu

Pod pojmem obsluha modulu se skrývá sadu nástrojů, které mají umožnit spouštět akce spojené s měřením pokrytí. Akce jsou následující:

1. Instalace součástí pro měření pokrytí.
2. Odinstalace součástí a návrat systému do stavu před instalací.
3. Spuštění všech testů s vyhodnocením pokrytí.
4. Vynulování datových souborů měřících pokrytí.
5. Autonomní vyhodnocení pokrytí.
6. Generování HTML výstupu pokrytí

Pro instrumentaci byl již vybrán konfigurační soubor nástroje Ant, proto ho lze rozšířit i akcemi pro instalaci middleware. Procesy spojené s instalací by pak měly logicky doplňovat i procesy pro návrat ke stavu před instalací. Akce číslo 1 a 2 jsou tedy v kompetenci procesů nástroje Ant.

Akce číslo 3 využívá dvě další akce a akci z modulu pro testy. V první fázi vynuluje datové soubory zaznamenávající pokrytí (akce číslo 4), v druhé provede testy voláním skriptu z modulu pro testy a nakonec vyhodnotí pokrytí (akce číslo 5). Pro akci proto stačí pouze skript vyvolávající sekvenci zmíněných třech akcí.

Nulování datových souborů pokrytí je nutné spojit s restartem aplikačního serveru Tomcat, protože nástroj Cobertura si data z měření pokrytí ukládá do struktur v paměti a do souboru je ukládá pouze při vypnutí serveru. Instrumentace nástroje Cobertura navíc vytváří prázdný datový soubor obsahující přehled všech instrumentovaných tříd zdrojového kódu. Při nulování je onen prázdný datový soubor nahrazen za ten, do kterého se ukládá. Datový soubor nástroje `coverage.py` stačí vymazat a je obnoven při dalším měření. Pojem prázdný datový soubor pro nástroj `coverage.py` znamená opravdu prázdný nebo i žádný datový soubor.

Akce číslo 5 vyhodnocující pokrytí musí podobně jako akce nulování restartovat server Tomcat k donucení nástroje Cobertura, aby uložil data z měření do datového souboru. Datové soubory včetně toho z nástroje `coverage.py` se následně analyzují a získává se z nich hodnota pokrytí kritéria všech větví programu v procentech.

Generování HTML výstupu pokrytí jako akce číslo 6 znamená vytvoření souhrnné statistiky pokrytí z naměřených dat. Oba nástroje použité pro měření nabízí možnost vygenerování výstupu v HTML pro webový prohlížeč. Vizuálně se liší, ale zobrazovaná data jsou stejná – pro každý sledovaný kus kódu je dostupná míra pokrytí a dokonce lze zobrazit kód s řádky zvýrazněnými různými barvami podle toho, zda jsou pokryty nebo ne. Nástroj `coverage.py` zvýrazněný kód generuje automaticky. Nástroji Cobertura musí být při generování předána cesta ke zdrojovým souborům, ze kterých byl instrumentovaný kód skompilován. Zdrojové kódy jsou dostupné buď z repozitáře Spacewalku nebo ze zdrojových balíčků[1].

[1] Viz https://fedorahosted.org/spacewalk/wiki/DownloadIt

Kapitola 4

Implementace knihovny

Kapitola implementace knihovny ctí rozdělení dle modulů stanovené v kapitole návrhu. V první části bude rozebráno vytvoření modulu pro testy a v druhém modulu vyhodnocující pokrytí.

Celý projekt byl implementován v adresáři systému `/opt`, který je pro rozšiřující balíčky nespadající do výchozí instalace určen [15]. Konkrétně v adresáři `spacewalk-xmlrpc-tests`.

4.1 Modul pro testy

Základní rozvržení modulu pro testy do adresářů a skriptů vychází z návrhu. Adresáře jsou čtyři: `beakerlib_plugins`, `conf`, `RHNSatellite` a `xmlrpc_scripts`. A skripty dva: `runtests.sh` a `analyze_beakerlib_journal.py`.

- Adresář `beakerlib_plugins` obsahuje rozšíření knihovny BeakerLib.
- Konfigurace modulu se nachází v adresáři `conf`.
- Adresář `RHNSatellite` obsahuje samotné testy pro něž je `RHNSatellite` nejvyšším, kořenovým jmenným prostorem v jejich hierarchii.
- V posledním adresáři `xmlrpc_scripts` jsou skripty zapozdřující jednotlivá XML-RPC volání pro možnost volání z interpretu Bash.
- Skript `runtests.sh` umí hromadně spouštět testy a vypisovat jejich úspěšnost.
- Druhý skript `analyze_beakerlib_journal.py` dokáže analyzovat XML výstup BeakerLib žurnálu a získávat z něj potřebné údaje.

4.1.1 Rozšíření knihovny BeakerLib

Rozšíření knihovny BeakerLib je napsáno v jazyce interpretu Bash a respektuje styl psaní kódu použitý v implementaci knihovny samotné. Ten se vyznačuje především prefixováním všech poskytovaných funkcí dvěma znaky `rl` a dokumentací se syntaxí *Pod*. Pod (zkratka Plain Old Documentation) je jednoduchý značkovací jazyk pro psaní dokumentace jazyka Perl a v jazyce Perl psaných programů a modulů [22]. Aby nebyla část s dokumentací interpretována při spuštění skriptu, využívá se konstrukce *here document* interpretu Bash.

```
: <<=cut
=pod

=head3 rlSatelliteXmlRpcFrontendRun

Run the given XML-RPC script from library (actual path is affected by version
of RHN Satellite) against Satellite frontend API.

    rlSatelliteXmlRpcFrontendRun script

Output of the script is accessible in file of name specified in $rlRun_LOG
variable. Caller is responsible for removing the file.

=over

=item script

The script which should be run from library, e.g. "auth.login.py admin redhat
    ".

=back

Returns 0 and asserts PASS if the return value of script is 0 (FAIL otherwise
    ).

=cut

rlSatelliteXmlRpcFrontendRun(){
    __INTERNAL_SatelliteXmlRpcRun "frontend" $1

    return $?
}
```

Zdrojový kód 4.1: Syntax dokumentace Pod v knihovně BeakerLib

Příklad použití je vidět v úryvku zdrojového kódu rozšíření 4.1. Úryvek dokumentace vždy začíná dvojtečkou, příkazem interpretu Bash, který „nic nedělá“. Následuje here document (dvě uhlové závorky) s řetězcem, který pokud se objeví na vstupu na samostatném řádku tak ukončí vstup právě tím řádkem. Syntax Pod dokumentace má končit řetězcem `=cut` a proto je pro ukončení vstupu použit právě on. Při zpracování nástrojem `perlpod` se naopak ignoruje kód interpretu Bash a dokumentace se generuje pouze z textu mezi dvojicemi řetězců `=pod` a `=cut`.

Rozšíření poskytuje několik funkcí přidávajících knihovně chybějící funkčnost. Funkce `rlSatelliteXmlRpcFrontendRun` volá skript z adresáře `xmlrpc_scripts/frontend`. Konkrétní jméno skriptu včetně předávaných parametrů je nutné funkci předat v parametru jako řetězec. Aby bylo možné odlišit XML-RPC volání v různých verzích Spacewalku, zkontroluje se nejprve, zda neexistuje skript právě pro nakonfigurovanou verzi. U takového skriptu se předpokládá umístění v podadresáři nazvaném označením verze. Pro Spacewalk verze 1.2 by se tedy zprvu zkontrolovalo umístění `xmlrpc_scripts/frontend/1.2` a pokud by tam skript nebyl nalezen, pokusila by se funkce hledat skript v `xmlrpc_scripts/frontend`. Funkce `rlSatelliteXmlRpcBackendRun` se chová analogicky až na umístění skriptu – místo `xmlrpc_scripts/frontend` pracuje s `xmlrpc_scripts/backend`.

Dvojice funkcí `rlSatelliteSaveTomcat6Log` a `rlSatelliteTomcat6LogNotDiffer` za-

jišťuje uložení záznamu chyb aplikačního serveru Tomcat na začátku testu a kontrolu přírůstku s případným vyvoláním chyby na konci testu. Výchozí nastavení počítá s existencí chybového záznamu v umístění `/var/log/tomcat6/catalina.out`. Nastavení lze však změnit definicí proměnné prostředí `TOMCAT6_LOG_FILE`. Podobně se chová i dvojice funkcí `rlSatelliteSaveHttpdLog` a `rlSatelliteAssertHttpdErrorLogNotDiffer`, která se stará o záznam chyb aplikačního serveru Httpd. Výchozí umístění `/var/log/httpd/error_log` je nastavitelné proměnnou prostředí `HTTPD_ERROR_LOG_FILE`.

Funkce poslední s názvem `rlSpacewalkVersionIs` požaduje v jednom parametru číslo verze systému Spacewalk jako řetězec. Ten je porovnán s verzí v konfiguraci a v případě, že je verze stejná, vrací logickou hodnotu pravda (v interpretu Bash hodnotu 0) a v případě verze odlišné logickou hodnotu nepravda (v interpretu Bash hodnotu 1).

4.1.2 Skripty volání XML-RPC

Jak jsem zmínil u popisu funkce `rlSatelliteXmlRpcFrontendRun` v rozšíření BeakerLib, v adresáři `xmlrpc_scripts` se nachází dva další adresáře `frontend` a `backend` dělící XML-RPC skripty podle rozhraní, ke kterému patří. V těch adresářích jsou už skripty nazvané dle zapouzdřovaného XML-RPC volání. Například skript `auth.login.py` z adresáře `frontend` zapouzdřuje volání `auth.login` z frontend XML-RPC rozhraní.

Kromě jednotlivých skriptů nazvaných dle XML-RPC volání obsahují adresáře `frontend` a `backend` moduly jazyka Python utilizující práci s XML-RPC rozhraním v souborech `frontend.py`, resp. `backend.py`. Oba exportují objekt `client` instancovaný ze třídy `xmlrpclib.ServerProxy`, který reprezentuje klienta XML-RPC. Modul `frontend.py` poskytuje navíc dvě proměnné `spacewalkLogin` a `spacewalkPassword` – přihlašovací údaje administrátora z konfigurace.

Ukázkový zdrojový kód 4.2 představuje příklad implementace skriptu pro XML-RPC volání `auth.login`. Skript vytváří rozhraní kopírující definici XML-RPC volání. Uživatelské jméno (angl. username) a heslo (angl. password) jsou parametry povinné, resp. pokud nejsou uvedeny užijí se automaticky ty administrátorské. Parametr trvání přihlášení (angl. duration) je naproti tomu nepovinný. V rámci propagace rozhraní musí skript navíc převádět vstupní hodnoty na správné typy. V ukázkovém zdrojovém kódu se na řádku 23 přetypovává parametr trvání přihlášení na celé číslo, protože všechny vstupní parametry skript získá v proměnných typu řetězec.

4.1.3 Testy s BeakerLib

Každý test založený na knihovně BeakerLib se nachází ve vlastním adresáři a skládá se ze třech souborů: `Makefile`, `PURPOSE` a `runtest.sh`.

Soubor `Makefile` obsahuje předpis kroků pro různé cíle. Nástroj `make` s parametrem cíle pak umí předepsané kroky postupně vykonat. Nástroj `make` je primárně určen ke generování spustitelných a jiných nezdrojových souborů programu ze souborů zdrojových [10]. U testu není z důvodu generování, ale z důvodu spuštění testovacího skriptu se správnými parametry a proměnnými prostředí.

Účel testu je popsán v souboru `PURPOSE`. Jeho obsah se začleňuje do výpisu žurnálu.

Ve třetím souboru `runtest.sh` jsou definovány jednotlivé fáze testu s využitím knihovny BeakerLib. Příklad testovacího skriptu ukazuje zdrojový kód 4.3. Na řádku druhém se vkládá společné nastavení všech testovacích skriptů, které zahrnuje vložení funkcí knihovny BeakerLib včetně rozšíření a které také zpracovává konfigurační soubor pro testovanou verzi systému Spacewalk. Všechny fáze zastřešuje žurnál zaznamenávající průběh – funkce

```
1  import sys
2  import xmlrpclib
3  import getopt
4  from frontend import client, spacewalkLogin, spacewalkPassword
5
6  def main(argv):
7      """
8      The main function called when this script is executed.
9      """
10     username = spacewalkLogin
11     password = spacewalkPassword
12     duration = None
13
14     try:
15         # there is one optional parameter "duration"
16         opts, args = getopt.getopt(argv, "d:", ["duration="])
17     except getopt.GetoptError:
18         usage()
19         sys.exit(2)
20     for opt, arg in opts:
21         if opt in ("-d", "--duration"):
22             # duration should be Integer param
23             duration = int(arg)
24
25     if len(args) != 2 and len(args) != 0:
26         usage()
27         sys.exit(2)
28
29     if len(args) == 2:
30         username = args[0]
31         password = args[1]
32
33     if duration is None:
34         session_key = client.auth.login(username, password)
35     else:
36         session_key = client.auth.login(username, password, duration)
37
38     print(session_key)
39
40 def usage():
41     """
42     Prints the usage information.
43     """
44     print("Python script calling the XML-RPC auth.login of local Spacewalk
           server.")
45     print("If both username and password are omitted, default satellite login
           and")
46     print("password is used.")
47     print("")
48     print("Usage:")
49     print(" ./auth.login.py [-d optional_duration] [username password]")
50     print("")
51
52 if __name__ == "__main__":
53     main(sys.argv[1:])
```

Zdrojový kód 4.2: Skript zapouzdřující XML-RPC frontend volání `auth.login`.

```
1  # Include the common setup
2  . ./../../../setup.sh
3
4  rlJournalStart
5
6  # ================================================================
7  # Do the testing
8  # ================================================================
9  if rlSpacewalkVersionIs "1.2"; then
10   rlPhaseStartTest "Testing auth.login of default administrator"
11     rlSatelliteSaveTomcat6Log
12
13     rlSatelliteXmlRpcFrontendRun "auth.login.py"
14
15     # Expect the session key of length 36
16     rlAssertGrep "[a-z0-9]\{36\}" "$rlRun_LOG"
17     rlRun "rm -f $rlRun_LOG"
18
19     rlSatelliteAssertTomcat6LogNotDiffer
20   rlPhaseEnd
21 fi
22
23 rlJournalEnd
24 rlJournalPrintText
```

Zdrojový kód 4.3: Testovací skript volání `auth.login`.

`rlJournalStart` a `rlJournalEnd` – vypisující výsledek v textovém formátu na standardní výstup funkcí `rlJournalPrintText`. Jediná testovací fáze je podmíněna shodou verze systému Spacewalk 1.2 s testovanou verzí na řádku 9. Fáze pak testuje úspěch přihlášení administrátora údaji z konfigurace voláním příslušného skriptu (řádek 13) a kontrolou výsledku (řádek 16), kterýmž by měl být klíč sezení v podobě řetězce o délce 36 znaků. Na konci fáze je navíc kontrola přírůstku záznamu chyb aplikačního serveru Tomcat (řádek 19) porovnávající obsah záznamu se stavem uloženým na začátku fáze (řádek 11).

4.1.4 Skript pro spouštění testů

Pro hromadné spouštění testů byl vytvořen skript `runtests.sh`, který ve výchozím nastavení spustí všechny testy v adresáři `RHNSatellite`. Adresář, ve kterém se mají testy vyhledat a spustit, se však dá změnit předáním jeho umístění v prvním parametru skriptu. Výstup skriptu je velmi stručný:

```
[vlki@vlki-laptop-fedora14 tests]$ ./runtests.sh
Passed 2 out of 2 tests
Duration: real 17s; tests 9s
```

První řádek výstupu zobrazuje počet úspěšných testů z celkového počtu všech spuštěných. Druhý řádek zobrazuje čas a to jak celkový čas běhu skriptu, tak i čas běhu pouze testů.

V případě, že dojde v některém z testů k chybě, je vytisknuto celé jméno testu a cesta k textovému výstupu žurnálu, aby bylo možné dohledat příčinu. Příkladem takového výstupu je následující výňatek z historie terminálu:

```
[vlki@vlki-laptop-fedora14 tests]$ ./runtests.sh
FAIL of /RHNSatellite/FrontendAPI/auth/login: See /tmp/beakerlib-Fq1aPpZ/jo
urnal.txt
Passed 1 out of 2 tests
Duration: real 23s; tests 13s
```

Hodnoty úspěšnost, doba trvání a celý název testu testu se zjišťují z XML výstupu žurnálu skriptem v jazyce Python `analyze_xmlrpc_journal.py`. Cesta k analyzovanému výstupu žurnálu se předává v prvním parametru a přepínače ovlivňují žádaný typ hodnoty. Bez přepínače je výstupem řetězec `FAIL` nebo `PASS` podle úspěšnosti testu. S přepínačem `-d` se zobrazí doba trvání jako počet sekund a přepínač `-t` zobrazí celý název testu.

Návratový kód skriptu `runtests.sh` je ovlivněn úspěšností testů. Pokud nedojde k chybě v žádném z provedených testů, je vrácena hodnota 0. Při detekci chyby pak hodnota 1.

4.2 Modul vyhodnocující pokrytí

Implementace modulu vyhodnocujícího pokrytí kritéria všech větví programu sestává ze dvou hlavních částí. Části instalace a odinstalace, kdy je systém připravován na měření nebo čištěn od měřících součástí, a části měření, při kterém se předpokládá, že již byl systém připraven pro měření a dochází pouze k vyhodnocování.

4.2.1 Instalace a odinstalace

Instalace a odinstalace se provádí pomocí procesů nakonfigurovaných pro nástroj Ant. Jejich vykonání vyžaduje práva superuživatele, protože ovládají aplikační servery a upravují instalaci systému Spacewalk. Veřejné procesy jsou pouze dva: jeden instalační nazvaný `install` a odinstalační `uninstall`. Pro jednodušší manipulaci jsem vytvořil i skripty `install.sh`, resp. `uninstall.sh`, zapouzdřující spouštění nástroje Ant s konfiguračním souborem a názvem procesu k vyvolání. Termínálová utilita `ant` nástroje Ant spouštějící procesy implicitně očekává konfigurační soubor pojmenovaný `build.xml` v adresáři, ze kterého je spuštěna.

Procesy se dále dělí na instalaci nástrojů měřících pokrytí pro kód jazyka Java (nástroj Cobertura) a pro kód jazyka Python (nástroj `coverage.py`). Instalace nástroje Cobertura spočívá v nahrazení nainstalovaného JAR archivu se skompilovaným kódem JAR archivem s instrumentovaným skompilovaným kódem. JAR archiv se v instalaci nacházel v umístění `/usr/share/rhn/lib/rhn.jar`. Instalační proces v kroku prvním zkopíruje původní JAR archiv z důvodu zálohy. V kroku druhém rozbalí JAR archiv do dočasného umíštění. V kroku třetím instrumentuje všechny nalezené třídy a nakonec znovu zabalí do JAR archivu, který umístí do stejného adresáře jako původní JAR archiv. V tomto stavu tedy existují tři JAR archivy:

- Neinstrumentovaný v původním umístění: `/usr/share/rhn/lib/rhn.jar`
- Neinstrumentovaný v záložním umístění: `/usr/share/rhn/lib/rhn.backup.jar`
- Instrumentovaný: `/usr/share/rhn/lib/rhn.cobertura.jar`

Aby bylo možné se při odinstalaci vrátit zpět k neinstrumentovanému JAR archivu, původní se odstraní a na jeho místo se vloží symbolický odkaz na JAR archiv instrumentovaný.

Instrumentace pomocí nástroje Cobertura navíc vytvoří prázdný datový soubor, který je přemístěn do adresáře projektu s proměnlivými daty – v operačních systémech vycházejících z Linuxu se taková data nacházejí v adresáři `/var` [15]. Protože se celý projekt nachází v adresáři `/opt/spacewalk-xmlrpc-tests`, umístění proměnlivých dat je od něj odvozeno: `/var/opt/spacewalk-xmlrpc-tests`.

Fakt, že instrumentace není nastavitelná, ovlivňuje i umístění, do kterého se ukládá datový soubor – do domovského adresáře serveru Tomcat (proměnná prostředí `$TOMCAT_HOME`). Pro dosažení ukládání do adresáře `/var/opt/spacewalk-xmlrpc-tests` instalace přidává symbolický odkaz z domovského adresáře serveru Tomcat právě do adresáře s proměnnými daty.

Instalace nástroje `coverage.py` upravuje konfigurační soubory modulů `mod_wsgi` a `mod_python`. Konkrétně přidává do seznamu umístění, kde se hledají při importu třídy, adresář `python_lib`. Následně pak přepíše názvy tříd, které se starají o zpracování požadavku vlastními nacházejícími se právě v adresáři `python_lib`. V adresáři `python_lib` je vytvořen speciální modul jazyka Python nazvaný `spacewalkcoverage` s middleware zajišťujícím měření pokrytí.

Konfigurační soubory obou modulů serveru Httpd se zálohují do stejného umístění jako soubory původní, navíc postfixované řetězcem `-backup`. V mé instalaci se konfigurační soubory nacházely v následujících umístěních:

- Modul `mod_python`:
  `/etc/rhn/satellite-httpd/conf/rhn/spacewalk-backend-xmlrpc.conf`
- Modul `mod_wsgi`: `/etc/httpd/conf.d/zz-spacewalk-server-wsgi.conf`

Aby si server Httpd nahrál upravené konfigurační soubory a server Tomcat instrumentovaný JAR archiv, je nutné je pro dokončení instalace restartovat.

Proces odinstalace v případě vyvolání pouze nahrazuje upravené soubory jejich zálohami, maže instalací vytvořené soubory a restartuje oba servery.

4.2.2 Měření pokrytí

Po dokončení instalace přichází na řadu skripty pracující s datovými soubory měření. Skript `reset.sh` je prvním, kterým začíná samotné měření. Vymaže datové soubory v adresáři `/var/opt/spacewalk-xmlrpc-tests` a restartuje server Tomcat.

Mnohem složitějším je skript `evaluate.sh`, který analyzuje datové soubory a získává z nich procentuální hodnotu pokrytí. Pokud je zavolán ihned po nulování skriptem `reset.sh`, vrací vždy 0 % tak jak je vidět na následujícím výpisu:

```
[vlki@vlki-laptop-fedora14 coverage]$ sudo ./reset.sh && sudo ./evaluate.sh
Coverage data files were successfully reset
Branch coverage: Java 0%, Python 0%
```

Postup vyhodnocení pokrytí se skládá z několika kroků. V prvním kroku se datové soubory nástrojů Cobertura a `coverage.py`, jejichž formát se liší, analyzují a generují se výstupy v jazyce XML. Výstup obou nástrojů se v tomto případě shoduje a je jím dokument definovaný pomocí DTD na stránkách projektu Cobertura[1]. Na vygenerování XML dokumentu jsou potřeba dva odlišné skripty pro každý nástroj zvlášť. Na zpracování

[1] Viz http://cobertura.sourceforge.net/xml/coverage-04.dtd

XML dokumentu je však potřeba už jen jeden skript. Ten čte z XML dokumentu hodnotu pokrytí a předává ji skriptu `evaluate.sh`, který ji pouze zobrazí na výstupu pro každý jazyk zvlášť.

Generování XML dokumentu z datového souboru nástroje Cobertura zajišťuje nástroj Ant s konfiguračním souborem v `scripts/report_java.xml`. Konfigurační soubor obsahuje dva procesy na generování výstupu v XML, resp. HTML. Při volbě procesu generujícího XML, se výstup vytvoří v adresáři `/tmp`.

Datový soubor nástroje `coverage.py` zpracovává skript `scripts/report_python.py` jež výsledný XML dokument také ukládá pouze dočasně do adresáře `/tmp` a navíc vrací cestu k dokumentu. Volající má v obou případech generování XML dokumentu povinnost dočasné soubory po vlastní analýze vymazat.

Vytvoření skriptu pro analýzu XML dokumentu a získání hodnoty pokrytí nebylo tak jednoduché, jak se mohlo na první pohled zdát. Ač je potřebná informace v atributu kořenového elementu, XML dokument musí být nejdříve zpracován. První implementace využívající modul jazyka Python `xml.dom.minidom` zpracovávala dokument z nástroje Cobertura velikosti 12,7 MiB asi 45 sekund. Modul `xml.dom.minidom` totiž nahrává celý XML dokument do paměti. Z porovnání různých knihoven určených ke zpracování XML vyšly lépe moduly `lxml` a `xml.etree.cElementTree` [12]. Udělal jsem proto krátké měření se zpracováním dokumentu z nástroje Cobertura a výsledky zaznamenal do tabulky 4.1. Měření bylo prováděno pomocí linuxové utility `time` pětkrát pro každou implementaci skriptu a délka zpracování v tabulce vyjadřuje součet hodnot `user` a `sys` z výpisu utility `time` – to je doba, po kterou bylo jádro využito pouze a jen procesem skriptu.

Použitá knihovna	Měření č.1 [s]	č.2	č.3	č.4	č.5	Průměr [s]
`xml.dom.minidom`	45,3	44,3	43,6	45,3	44,4	44,6
`lxml`	2,5	2,0	2,0	2,0	2,0	2,1
`xml.etree.cElementTree`	2,2	2,3	2,2	2,3	2,3	2,3

Tabulka 4.1: Měření doby zpracování XML dokumentu různými knihovnami jazyka Python

Ve skriptu `scripts/fetch_branch_coverage.py` jsem pro zpracování XML na základě měření použil knihovnu `lxml`. Skript přečte hodnotu z atributu `branch-rate` kořenového elementu, převede ji na celé číslo a vytiskne na svém výstupu.

Skript kopírující jméno skriptu z modulu pro testy `runtests.sh` pouze zjednodušuje vyhodnocení pokrytí testů tím, že nejdříve vykoná skript `reset.sh` nulující datové soubory, následně skript `runtests.sh` spouštějící testy a vyhodnotí výsledek voláním skriptu `evaluate.sh`.

Posledním nezmíněným skriptem je `report.sh` jež dokáže generovat statistiky pokrytí v HTML formátu zobrazitelném webovým prohlížečem. Podobně jako ostatní skripty také restartuje server Tomcat a vyžaduje proto spuštění s právy superuživatele. Pracuje s konfiguračním souborem nástroje Ant `scripts/report_java.xml` a skriptem v jazyce Python `scripts/report_python.py`, které se správnými parametry umí místo XML výstupu generovat výstup v HTML. Jako parametry je nutné předat cesty k adresářům, ve kterých se mají HTML dokumenty vytvářet a navíc umístění zdrojových souborů jazyka Java. Absence správného umístění ovlivní výstup pokrytí kódu jazyka Java nedostupností zvýrazněného zdrojového kódu.

Kapitola 5

Návrh a implementace sady testů

Po vytvoření knihovny se konečně dostávám k cíli bakalářské práce. A to k vytvoření sady testů. Sada, kterou budu na navrhovat a následně implementovat, bude spíše demonstrační. Vytváření sady rozsáhlejší by totiž překročilo rozsah vypsaný pro bakalářskou práci.

Vybral jsem si jedno volání z frontend rozhraní tak, abych ukázal možnosti, které knihovna vytvořená v předchozích krocích nabízí. Jde o `errata.clone`, které jsem vybral z důvodu rozvětvenosti obslužného kódu. Budu na obslužném zdrojovém kódu demonstrovat vytvoření testů se stoprocentním pokrytím.

5.1 Návrh testů

Část návrhu testů se zabývá opravdu jen návrhem testů tak, aby pokryly požadavky vycházející z kritéria pokrytí všech větví programu.

O obsluhu volání `errata.clone` z frontend rozhraní se stará metoda `clone` nacházející se v třídě `ErrataHandler` balíku `com.redhat.rhn.frontend.xmlrpc.errata`. I když obslužná metoda volá další metody, budu se v tomto návrhu zabývat jen pokrytím samotné obslužné metody. Implementace metody je vidět ve vloženém zdrojovém kódu 5.1.

Graf vytvořený ze zdrojového kódu metody `clone` je na obrázku 5.1. Čísla v uzlech grafu vyjadřují řádky kódu vloženého zdrojového kódu 5.1. Hrany grafu jsou popsány popsány podmínkou, která určí další postup vykonávání programu. Není-li hrana popsána, předpokládá se podmínka opačná než u druhé hrany vycházející z uzlu.

Pro pokrytí grafu metody jsou potřeba čtyři testy:

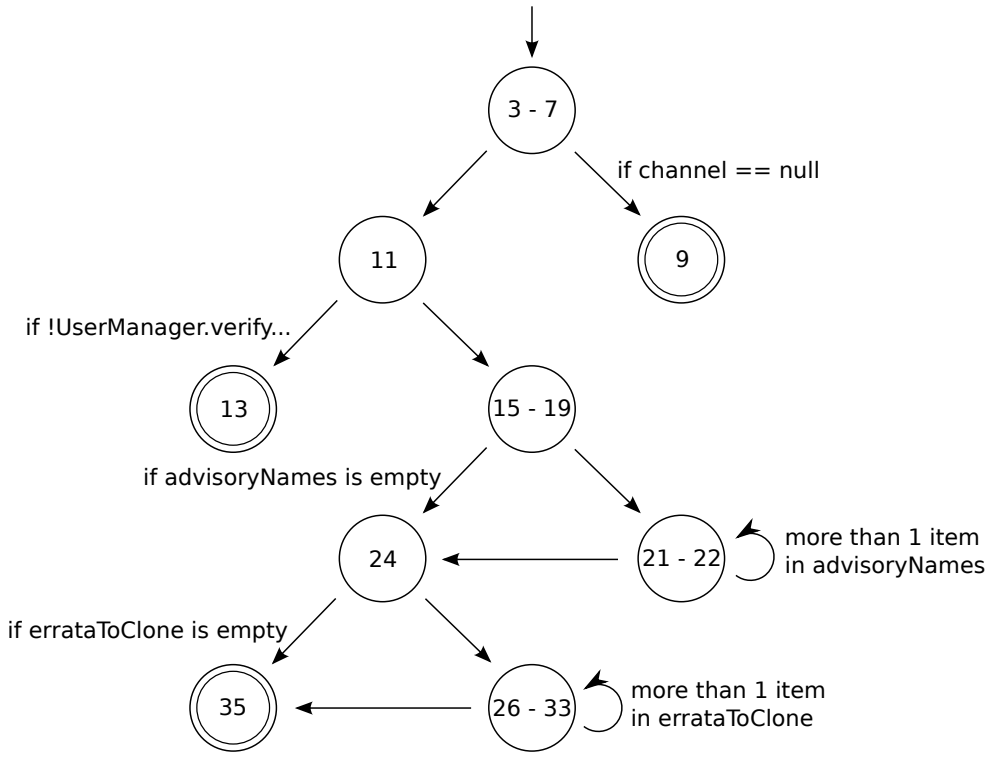
1. Test vyhodnocující podmínku `channel == null` jako pravdivou. Musí volat rozhraní s neexistujícím názvem kanálu.

2. Test vyhodnocující podmínku `if (!UserManager.verify ...` jako pravdivou a podmínku `channel == null` jako nepravdivou. Volání musí proto provádět s existujícím názvem kanálu a přihlášený jako uživatel, který nemá právo ke kanálu přistupovat.

3. Test vyhodnocující podmínky `channel == null` a `if (!UserManager.verify ...` jako nepravdivé a způsobující zároveň provedení s prázdnými seznamy `advisoryNames` a `errataToClone`. Takový test bude tedy volán s názvem existujícího kanálu, s přihlášeným uživatelem, který má ke kanálu přístup, a prázdným seznamem chyb ke klonování.

```
1     public Object[] clone(String sessionKey, String channelLabel,
2             List advisoryNames) throws InvalidChannelRoleException {
3         User loggedInUser = getLoggedInUser(sessionKey);
4
5         Channel channel = ChannelFactory.lookupByLabelAndUser(channelLabel,
6                                 loggedInUser);
7
8         if (channel == null) {
9             throw new NoSuchChannelException();
10        }
11
12        if (!UserManager.verifyChannelAdmin(loggedInUser, channel)) {
13            throw new PermissionCheckFailureException();
14        }
15
16        List erratasToClone = new ArrayList();
17        List toReturn = new ArrayList();
18
19        //We loop through once, making sure all the errata exist
20        for (Iterator itr = advisoryNames.iterator(); itr.hasNext();) {
21            Errata toClone = lookupErrata((String)itr.next(), loggedInUser.
                  getOrg());
22            erratasToClone.add(toClone);
23        }
24        //now that we know its all valid, we clone everything.
25        for (Iterator itr = erratasToClone.iterator(); itr.hasNext();) {
26            Errata cloned = ErrataManager.createClone(loggedInUser, (Errata)
                  itr.next());
27            Errata publishedClone = ErrataManager.publish(cloned);
28
29            publishedClone = ErrataFactory.publishToChannel(publishedClone,
                  channel,
30                      loggedInUser);
31            ErrataFactory.save(publishedClone);
32
33            toReturn.add(publishedClone);
34        }
35        return toReturn.toArray();
36    }
```

Zdrojový kód 5.1: Oslužná metoda frontend volání `errata.clone`.



Obrázek 5.1: Graf zdrojového kódu metody `clone`, který má být pokryt.

4. Poslední test se liší od třetího testu tím, že seznamy `advisoryNames` a `erratoToClone` budou mít alespoň dva prvky. Toho lze dosáhnout předáním seznamu alespoň dvou chyb ke klonování.

5.2 Implementace testů

Prvním krokem implementace bylo vytvoření všech XML-RPC skriptů v adresáři modulu pro testy `xmlrpc_scripts/frontend`. Kromě skriptu pro testované volání bylo nutné vytvořit i skripty další, využité při přípravě systému Spacewalk k testování a následném úklidu.

Aby mohl být implementován test první, je nutný skript volání `auth.login` pro přihlášení jako administrátor. Test druhý vyžaduje volání dalších čtyř: dvojici volání pro vytvoření a vymazání testovacího kanálu `channel.software.create` a `channel.software.delete` a dvojici `user.create` a `user.delete` pro vytvoření testovacího uživatele. Poslední dvojici volání přidává ještě čtvrtý test: `errata.create` a `errata.delete` k vytvoření objektů chyb, které budou následně naklonovány.

Krok druhý zahrnoval vytvoření samotného testovacího skriptu. Umístěn je dle názvu testovaného volání v `RHNSatellite/FrontendAPI/errata/clone`. V přípravné fázi se přihlásí administrátor, vytvoří se testovací kanál se dvěma objekty chyb a vytvoří se testovací uživatel, který je následně také přihlášen.

Testovací fáze skriptu jsou čtyři a implementují navržené testy.

V poslední fázi úklidu se odstraní všechna testovací data vytvořená ve fázi přípravné. Tím se zajistí možnost opakovaného spouštění testu bez problémů s konflikty existujících

jmen objektů v systému.

5.3 Vyhodnocení pokrytí testů

Pro vyhodnocení pokrytí jsem nepoužil skript měřící procentní pokrytí všech zdrojových kódů, ale vygenerovaný HTML výstup. Procentní pokrytí je totiž při testech pouze malé podmnožiny volání zkreslené a nemá žádnou vypovídající hodnotu.

Aby mohl být HTML výstup jazyka Java se zvýrazněnými řádky dle pokrytí, musel jsem v prvé řadě stáhnout zdrojové soubory systému Spacewalk verze 1.2. Použil jsem k tomu distribuovaný verzovací nástroj git. Stáhl jsem celý repozitář pomocí příkazu `git clone` a příkazem `git checkout remotes/origin/SPACEWALK-1.2` se přepnul do větve systému Spacewalk verze 1.2

Následně jsem spustil testy s analýzou pokrytí:

```
[vlki@vlki-laptop-fedora14 coverage]$ sudo ./runtests.sh
Coverage data files were successfully reset
Passed 3 out of 3 tests
Duration: real 78s; tests 68s
Branch coverage: Java 4%, Python 1%
```

A nakonec vygeneroval HTML výstup:

```
[vlki@vlki-laptop-fedora14 coverage]$ sudo ./report.sh -j ~/java_html_report/
-s ~/spacewalk-src/java/code/src/
report.sh: Java HTML report generated into /home/vlki/java_html_report/
```

Obrázek 5.2 zachycuje HTML výstup pokrytí zdrojového kódu metody `clone` třídy `ErrataHandler`. Zelená barva potvrzuje, že testy kompletně pokrývají metodu. Vyhovují tedy požadavkům vycházejícím z kritéria pokrytí všech větví programu pro vybranou část systému Spacewalk.

Obrázek 5.2: HTML výstup potvrzující kompletní pokrytí zdrojového kódu 5.1.

Kapitola 6

Závěr

Původním cílem bakalářské práce bylo navržení a implementace sady testů rozhraní XML-RPC systému Spacewalk. Z analýzy systému a způsobu testování ale vyšel důležitější požadavek a to najití cesty, jak vyhodnocovat kvalitu testů. Práce proto dala vzniknout komplexní knihovně, která má testování XML-RPC rozhraní systému Spacewalk zastřešovat. V rámci práce byly také navrženy testy pro malou část systému Spacewalk a tím demonstrován způsob využití vytvořené knihovny. Důležitým získaným poznatkem je fakt, že při testování nejde pouze o vyhodnocování úspěšnosti testů, ale stejně tak i o vyhodnocování kvality samotných testů vůči některému aspektu testovaného systému.

Aktuální řešení některých částí knihovny není ideální a například nutnost restartování serveru Tomcat při každé manipulaci s datovými soubory měření vyžaduje po uživateli použití superuživatelských privilegií a navíc procesy zpomaluje. Řešením problému by mohlo být použití druhého, vestavěného serveru Tomcat jak jej popisuje dokumentace [13]. Ten by byl nainstalován do serveru původního a umožňoval by jednodušší restartování. Jiná řešení popisuje i část oficiálních webových stránek nástroje Cobertura s častými otázkami a odpověďmi [7], ze kterých považuji za nejpraktičtější opakované ukládání s dostatečně krátkou časovou periodou.

Dalším nedostatkem práce je velmi stručná sada testů. Tak jak byl v kapitole 5 popsán návrh a implementace testů pro jedno z volání, tak by se měly projít všechny části kódu a podle nich navrhnout testy. Vytvoření kompletní sady testů je ale příliš komplikované vzhledem k jednoletému bakalářskému projektu.

Zajímavým rozšířením by nakonec mohlo být vytvoření nástrojů pro analýzu pokrytí požadavků kritéria z jiného než grafového typu abstraktního modelu. Kritérium pokrytí všech větví programu není rozhodně nejlepší a proto předpokládám, že použitím jiného kritéria by se mohlo dosáhnout vyšší efektivity testů. Výběru kritéria by však musela předcházet důkladná analýza ve vztahu k testovanému systému.

Literatura

[1] Ammann, P.; Offutt, J.: *Introduction to Software Testing*. Cambridge University Press, 2008, ISBN 978-0-521-88038-1.

[2] Apache Software Foundation: *Apache Ant* [online]. [cit. 2011-05-02].
URL http://ant.apache.org/

[3] Apache Software Foundation: *Apache Tomcat Configuration Reference — The Server Component* [online]. [cit. 2011-04-24].
URL http://tomcat.apache.org/tomcat-6.0-doc/config/server.html

[4] Apache Software Foundation: *Properties files* [online]. [cit. 2011-05-01].
URL http://commons.apache.org/configuration/howto_properties.html

[5] Apache Software Foundation: *The Apache Tomcat Connector — Webserver HowTo — Apache HowTo* [online]. [cit. 2011-04-24].
URL
http://tomcat.apache.org/connectors-doc/webserver_howto/apache.html

[6] Batchelder, N.: *Branch coverage measurement* [online]. Last modification: July 25th, 2011, [cit. 2011-04-30].
URL http://nedbatchelder.com/code/coverage/branch.html

[7] Doliner, M.: *Cobertura: FAQ* [online]. [cit. 2011-04-30].
URL http://cobertura.sourceforge.net/faq.html

[8] Doliner, M.: *Cobertura* [online]. [cit. 2011-04-30].
URL http://cobertura.sourceforge.net/

[9] Eby, P.: *Python Web Server Gateway Interface v1.0.1* [online]. Last modification: January 16th, 2011, [cit. 2011-05-01].
URL http://www.python.org/dev/peps/pep-3333/

[10] Free Software Foundation, Inc.: *GNU Make* [online]. Last modification: July 3rd, 2010, [cit. 2011-05-02].
URL http://www.gnu.org/software/make/

[11] Laurent, S. S.; Johnston, J.; Dumbill, E.: *Programming Web Services with XML-RPC*. O'Reilly Media, 2001, ISBN 0-596-00119-3.

[12] Lundh, F.: *The cElementTree Module* [online]. Last modification: January 30th, 2005, [cit. 2011-05-02].
URL http://effbot.org/zone/celementtree.html

[13] McClanahan, C. R.: *Apache Tomcat 6.0.32 API Documentation — Class Embedded* [online]. Last modification: April 29th, 2010, [cit. 2011-05-03].
URL http://tomcat.apache.org/tomcat-6.0-doc/api/org/apache/catalina/startup/Embedded.html

[14] Müller, P.; Hudlický, O.; Hutař, J.; aj.: *BeakerLib* [online]. Last modification: June 24th, 2010, [cit. 2011-04-27].
URL https://fedorahosted.org/beakerlib/

[15] Nguyen, B.: *Linux Filesystem Hierarchy: The Root Directory* [online]. Last modification: July 30th, 2004, [cit. 2011-05-02].
URL http://www.tldp.org/LDP/Linux-Filesystem-Hierarchy/html/the-root-directory.html

[16] Oracle: *Instrumentation (Java Platform SE 6)* [online]. [cit. 2011-05-01].
URL http://download.oracle.com/javase/6/docs/api/java/lang/instrument/Instrumentation.html

[17] Oracle: *JAR File Overview* [online]. [cit. 2011-04-24].
URL http://download.oracle.com/javase/6/docs/technotes/guides/jar/jarGuide.html

[18] Python Software Foundation: *ConfigParser — Configuration file parser* [online]. [cit. 2011-04-29].
URL http://docs.python.org/library/configparser.html

[19] Python Software Foundation: *Increase Test Coverage — Python Developer's Guide* [online]. [cit. 2011-04-30].
URL http://docs.python.org/devguide/coverage.html

[20] Reese, G.: *Database Programming with JDBC & Java, Second Edition*. O'Reilly Media, 2000, ISBN 1-56592-616-1.

[21] Sherrill, J.: *Mail archive of spacewalk-devel: mod_wsgi support merged with master* [online]. Last modification: January 7th, 2010, [cit. 2011-04-24].
URL http://www.mail-archive.com/spacewalk-devel@redhat.com/msg01948.html

[22] Wall, L.; Burke, S. M.: *Perl Programming Documentation: perlpod - the Plain Old Documentation format* [online]. [cit. 2011-05-01].
URL http://perldoc.perl.org/perlpod.html