

Na tomto místě bude oficiální zadání vaší práce

- Toto zadání je podepsané děkanem a vedoucím katedry,
- musíte si ho vyzvednout na studijním oddělení Katedry počítačů na Karlově náměstí,
- v jedné odevzdané práci bude originál tohoto zadání (originál zůstává po obhajobě na katedře),
- ve druhé bude na stejném místě neověřená kopie tohoto dokumentu (tato se vám vrátí po obhajobě).

České vysoké učení technické v Praze
Fakulta elektrotechnická
Katedra počítačů



Bakalářská práce

Ruby - Access Control List

Jan Šírl

Vedoucí práce: Ing. Pavel Strnad

Studijní program: Softwarové technologie a management, Bakalářský

Obor: Softwarové inženýrství

20. května 2012

Poděkování

Chtěl bych především poděkovat panu Ing. Pavlu Strnadovi, že se mě ujal jako vedoucí jak semestrálního projektu tak Bakalářské práce a poskytl mi rady a motivaci. Poděkování také patří mé přítelkyni a rodině za nemalou podporu při tvorbě bakalářské práce a v průběhu celého studia.

Prohlášení

Prohlašuji, že jsem práci vypracoval samostatně a použil jsem pouze podklady uvedené v příloženém seznamu.

Nemám závažný důvod proti užití tohoto školního díla ve smyslu §60 Zákona č. 121/2000 Sb., o právu autorském, o právech souvisejících s právem autorským a o změně některých zákonů (autorský zákon).

V Praze dne 25. 5. 2012

.....

Abstract

This paper presents the design and implementation of library management access rights in the programming language Ruby.

The library is designed for object XML database and is solved using the Access Control List. Library used for storing and querying the database itself, which communicates with the protocol for remote procedure call and for quering uses current query languages. The functionality of the library was developed and tested on eXistDB.

Abstrakt

Tato práce prezentuje návrh a realizaci knihovny pro správu přístupových práv v programovacím jazyku Ruby.

Knihovna je určena pro objektovou XML databázi a je řešena pomocí Access Control Listu. Knihovna využívá pro ukládání a dotazování samotnou databázi, se kterou komunikuje pomocí protokolu pro vzdálené volání procedur a pro dotazování používá aktuální dotazovací jazyky. Funkčnost knihovny byla vyvíjena a testována pomocí databáze eXist-db.

Obsah

1	Úvod	1
1.1	Úvod do problematiky	1
1.2	Motivace	2
2	Popis problému, specifikace cíle	5
2.1	Popis řešeného problému	5
2.2	Vymezení cílů a požadavků	5
2.3	Popis struktury bakalářské práce	5
2.4	Existující řešení	7
2.4.1	Firemní řešení	7
2.4.2	Dostupné knihovny	7
2.4.2.1	Acl9	7
2.4.2.2	iq-acl	8
2.4.2.3	ActiveACLPlus	8
3	Analýza a návrh řešení	9
3.1	Analýza	9
3.1.1	Existující řešení	9
3.1.1.1	Obecné řešení	9
3.1.1.2	Oracle	10
3.1.1.3	phpGACL	10
3.2	Návrh implementace	11
3.2.1	Use Case Scénáře	11
3.2.1.1	Ověřování oprávnění k objektu	11
3.2.2	Class diagram	11
3.2.2.1	Zadávání pravidla (ACE)	12
3.2.3	Rozhraní	12
3.2.3.1	Rozhraní mezi Ruby-ACL a databází	12
3.2.3.2	Rozhraní mezi Ruby-ACL a uživatelskou aplikací	15
3.2.4	Ukázka použití	15
3.2.4.1	Příklad nastavení práv	15
3.2.4.2	Příklad kontroly práv	16
3.3	Popis logiky vyhodnocování pravidel	17
3.3.1	ACL Objekty	17
3.3.1.1	Zmocnitel (Principal)	18

3.3.1.2	Oprávnění (Privilege)	18
3.3.1.3	Zdrojový objekt (Resource object)	19
3.3.1.4	Pravidlo (ACE)	20
3.3.2	Pravidla rozhodování	20
3.3.2.1	Priorita rozhodování	20
3.3.3	Složitost rozhodování	21
4	Realizace	23
4.1	Průběh realizace	23
4.2	Programy použité při vývoji	24
4.3	Databáze	25
4.3.1	Sedna	25
4.3.2	eXist-db	25
4.3.2.1	eXistAPI	25
4.4	Implementace	26
4.4.1	Třída RubyACL	26
4.4.1.1	Metoda check	26
4.4.1.2	Příklad	26
4.4.2	ACL_Object	28
4.4.3	Vyjimky	29
4.4.4	Práce s pamětí	29
5	Testování	31
5.1	RubyACL	31
5.2	eXistAPI	31
6	Závěr	33
7	Moje pracovní poznámky	35
A	Pokyny a návody k formátování textu práce	39
A.1	Vkládání obrázků	39
A.2	Kreslení obrázků	40
A.3	Tabulky	40
A.4	Odkazy v textu	41
A.4.1	Odkazy na literaturu	41
A.4.2	Odkazy na obrázky, tabulky a kapitoly	43
A.5	Rovnice, centrovaná, číslovaná matematika	43
A.6	Kódy programu	44
A.7	Další poznámky	44
A.7.1	České uvozovky	44
B	Ukázky zdrojového kódu	45
B.1	Výpis veřejných metod Ruby-ACL	45
C	Seznam použitých zkratk	47

D	UML diagramy	49
D.1	Use Case Diagram	49
D.2	eXistAPI	49
E	Instalační a uživatelská příručka	53
F	Obsah přiloženého CD	55

Seznam obrázků

3.1	Class diagram	13
3.2	Diagram znázorňující model rozhraní	14
3.3	Diagram znázorňující sekvenci komunikace	17
4.1	Diagram znázorňující chod metody check	27
A.1	Popiska obrázku	40
D.1	UseCases	50
D.2	Diagram tříd eXistAPI	51
F.1	Seznam přiloženého CD — příklad	55

Seznam tabulek

2.1	Tabulka funkčních a obecných požadavků. priorita = (povinný, volitelný, nepovinný)	6
3.1	Jednoduchý příklad obecného řešení modelu práv pomocí matice	9
3.2	Parametry a návratové hodnoty metody check	15
A.1	Ukázka tabulky	40

Kapitola 1

Úvod

1.1 Úvod do problematiky

Tato bakalářská práce navazuje na můj semestrální projekt. Zabývá se správou - administrací řízení přístupu, jakožto procesu autorizování uživatelů, skupin a počítačů pro přístup k objektům. Tento proces pracuje s pojmy: oprávnění, uživatelská práva a audit objektů. Tato jednotlivá přiřazená oprávnění začleňuje jako položky řízení přístupu (ACE¹) a jejich celé sady začleňuje do seznamu přístupových práv (ACL²). Úkolem bakalářské práce bylo vytvořit, navrhnout a realizovat v jazyce Ruby model uživatelských přístupových práv určenou pro objektovou XML³ databázi.

Součástí práce byl návrh knihovny. Navržená knihovna realizuje správu řízení přístupu pomocí ACL. Knihovnu nazývám Ruby-ACL. Protože moje bakalářská práce je prací implementační, včetně testů navrženého knihovny, zaměřil jsem se na specifikaci rozhraní knihovny a na příklady jejího použití. Výsledkem je nejen samotná realizace knihovny, ale i podrobná programátorská dokumentace.

Je-li potřeba zabezpečit zdroje a jeho prostředky, je nutné vzít v úvahu, jakými právy budou disponovat ti, kteří k nim budou přistupovat. Zabezpečit objekt, například dokument či kolekci, lze přidělením oprávnění, která umožňují uživatelům nebo skupinám provádět u tohoto objektu určité akce. Řízení přístupů je činnost zabývající se přidáváním a zamítáním oprávnění přistupujícím.

Ruby je objektově orientovaný, interpretovaný skriptovací programovací jazyk. Díky své jednoduché syntaxi je poměrně snadný, přesto však dostatečně výkonný, aby dokázal konkurovat známějším jazykům jako je Python a Perl. Převzato a upraveno z wikipedie[9]

ACL je seznam pro řízení přístupů. Seznam určuje, kdo nebo co má povolení přistupovat k objektu a jaké operace s ním může nebo nesmí provádět. V typickém ACL specifikuje každý záznam v seznamu uživatele a operaci[10].

¹Access Control Entry

²Access Control List

³Extensible Markup Language

1.2 Motivace

Zaujala mě problematika práv, databází a pro mě neznámého jazyka Ruby. Jádro celé mé motivace, bylo projít si vývojem softwaru, v tomto případě knihovny, od návrhu přes implementaci k testování a dokumentaci a ověřit si získané poznatky z předmětu softwarového inženýrství. Soustředil jsem se na vlastní nápady. Nechtěl jsem skládat a kopírovat polotovary a "lepit" je dohromady.

TODO prepsat vsechny xQuery, xUpdate, xPath na velky X TODO odkaz na prvni vyskyt RubyGEM

Kapitola 2

Popis problému, specifikace cíle

2.1 Popis řešeného problému

Databáze, pro kterou byla knihovna určena, nemá žádný model uživatelských přístupových práv. Bylo potřeba tento nedostatek vyřešit knihovnou implementovanou v jazyce Ruby. Jazyk Ruby byl vybrán z důvodu jeho rozšířenosti a z důvodu kompatibility s budoucími částmi databáze, které budou taktéž naimplementované v Ruby. Mnou naimplementovaná knihovna Ruby-ACL řeší problém se spravováním přístupových práv.

2.2 Vymezení cílů a požadavků

Cílem bakalářská práce bylo navrhnout, realizovat a otestovat knihovnu v jazyce Ruby, která bude spravovat uživatelská přístupová práva pro objektovou XML databázi. Cílem bylo vytvořit co nejjednodušší knihovnu, která by splňovala všechna kritéria zadání. Nechtěl jsem používat nebo skládat dohromady existující moduly a knihovny, které danou problematiku řeší, protože jsem si za cíl dal vytvořit něco svého a projít si vývojem softwaru od požadavků, analýzy, návrhu přes realizaci k testování a dokumentaci. I když Ruby-ACL je určena pro XML databázi, chtěl jsem, aby byla použitelná i pro reálné přístupy do budov apod. Předsevzal jsem si, že by bylo přínosné, kdyby knihovna umožňovala jemně nastavit přístupy (v angličtině se používá výraz "fine-grained").

Seznam požadavků je popsán v tabulce [2.1](#)

2.3 Popis struktury bakalářské práce

Nejpodstatnější částí bakalářské práce z pohledu vytyčených cílů je sekce Rozhraní a sekce Příklady užití v kapitole Analýza a dále celá kapitola Testování.

Kapitola 1 nás uvádí do Bakalářské práce. Vysvětluje, co vlastně řízení práv je, popisuje jeho význam a vysvětluje nejdůležitější pojmy.

V kapitole 2 se hovoří o důvodech implementace Ruby-ACL, vytyčují se cíle a prezentují požadavky. Dále obsahuje stručný popis existujících řešeních.

id	Specifikace požadavků na software	priorita
FUNKČNÍ POŽADAVKY		
0	Ruby-ACL bude umožňovat řízení přístupů pomocí ACL	povinný
1.0	Ruby-ACL bude umožňovat definovat, editovat a mazat zmocnitele (principals)	povinný
1.1	Ruby-ACL bude umožňovat definovat, editovat a mazat oprávnění (privileges)	povinný
1.2	Ruby-ACL bude umožňovat definovat, editovat a mazat zdrojové objekty (resource objects)	povinný
1.3	Ruby-ACL bude umožňovat definovat, editovat a mazat pravidla přístupu (ACE)	povinný
2.0	Ruby-ACL bude umožňovat vytvářet ACL	povinný
2.1	Ruby-ACL bude umožňovat načítat a ukládat ACL z a do XML souboru	povinný
2.2	XML soubor bude definovaný pomocí DTD or XML Schema	volitelný
2.3	XML soubor bude "well formated" podle W3C doporučení	povinný
3.0	Ruby-ACL bude nabízet pouze Default-Deny politiku	povinný
4.0	Ruby-ACL bude testována na eXist-db	povinný
OBECNÉ POŽADAVKY		
1.0	Ruby-ACL bude naprogramována v jazyce Ruby	povinný
2.0	Ruby-ACL bude vydaná jako RubyGem	volitelný
3.0	Ruby-ACL potřebuje databázi podporující xQuery, xPath technologie	povinný

Tabulka 2.1: Tabulka funkčních a obecných požadavků. priorita = (povinný, volitelný, nepovinný)

Kapitola 3 je nejpodstatnější z celé práce. Jedná se o Analýzu, ve které bylo popsáno z čeho jsem vycházel při návrhu. Nejdůležitější část je Rozhraní a Ukázka použití, protože přímo splňují zadání/požadavky práce. Sekce Popisu logiky vyhodnocování pravidel je důležitá pro uživatele knihovny. Nejen že vysvětluje pojmy jako ACL objekt, principal, ale hlavně popisuje jakým způsobem knihovna Ruby-ACL rozhoduje, které pravidlo má vyšší váhu.

Kapitola 4 se zaměřuje na postup vývoje při implementaci a problémy při realizaci.

Kapitole 5 je rozdělená na dvě části. Jedna se zabývá pomocným komunikačním rozhraním eXistAPI a druhá samotnou knihovnou Ruby-ACL. Obě části jsou však zaměřeny na vysvětlení způsobu testování a jejich výsledky.

Kapitola 6 se zaměřuje na zhodnocení splnění cílů Bakalářské práce a rozebírá možné nedostatky a případné pokračování v práci na knihovně.

V příloze se nacházejí diagramy, které nebyly potřeba pro vysvětlení funkcionality knihovny. Součástí přílohy je i postup instalace knihovny a CD s knihovnou a dalšími soubory.

2.4 Existující řešení

Existující řešení lze rozdělit na dva druhy. První jsou firemní řešení a druhé jsou knihovny v Ruby zabývající se stejnou nebo podobnou problematikou.

2.4.1 Firemní řešení

Vybral jsem tři ukázková řešení. Oracle má nejpropracovanější model řízení práv. Podporuje integrování LDAP¹ a WebDAV².

PhpGACL je bezplatný jednodušší systém ve srovnání Oracle spravující přístupy ke zdrojům. PhpGACL je opensource využívaný pro webové aplikace.

Obecné řešení se skládá z dvojrozměrné tabulky, kde jeden rozměr je tvořen všemi, kdo vyžadují přístup a druhý rozměr obsahuje objekty, ke kterým je vyžadován přístup. Oprávnění je v buňce, která se nachází na průsečíku os zmíněných dvou rozměrů. Podrobnější zpracování se nachází v sekci 3.1.1 Existující řešení, která se nachází v kapitole Analýza.

2.4.2 Dostupné knihovny

V následujících podsekcích jsou dostupné knihovny napsané v Ruby, které by mohly být použity, kdybych si za cíl nedal vytvořit Ruby-ACL sám.

2.4.2.1 Acl9

Acl9 je další řešení autorizace založené na rolích v Ruby on Rails³. Skládá se ze dvou subsystémů, které mohou být použity samostatně. Subsystém kontroly rolí umožňuje nastavovat a dotazovat se na uživatelské role pro různé objekty. Subsystém řízení přístupu umožňuje určit různá přístupová pravidla založená na rolích uvnitř řadičů. Text byl převzat a upraven z [6]

¹Lightweight Directory Access Protocol

²Web-based Distributed Authoring and Versioning

³framework pro pohodlné a rychlé vytváření moderních webových aplikací

2.4.2.2 iq-acl

Cílem tohoto rubygemu je poskytnout serii tříd, které umí zacházet s běžnými požadavky na řízení práv. V současné době poskytuje třídu `IQ::ACL::Basic`, která přestože je velmi jednoduchá je také velmi schopná. Více o použití se můžete dočíst zde [\[5\]](#).

2.4.2.3 ActiveACLPlus

Plugin `ActiveAclPlus` realizuje flexibilní, rychlý a snadno použitelný obecný systém kontroly přístupu. Systém je založen na `phpgacl.sourceforge.net`, přidáním objektu orientace, polymorfismu a dvou úrovní paměti. `PhpGacl` tvrdí, že v reálné pracovní verzi s mnoha přidávanými vrstvami složitosti podporuje více než 60.000 účtů, 200 skupin a 300 ACO. "Testy provedené na vyvojářském notebooku ukazují 10 - 30 krát větší zlepšení výkonnosti ve srovnání s `active_rbac`. Plugin používá ukládání do mezipaměti. Používá instanční paměť a v případě potřeby ukládá výsledky oprávnění do paměti s použitím časového omezení. Text byl převzat a upraven z [\[8\]](#)

Kapitola 3

Analýza a návrh řešení

Tato kapitola se dělí na analýzu a návrh. V analýze jsem se zaměřil na prostudování tří existujících řešení. Z informací získaných z dokumentace jsem sestavil návrh pro Ruby ACL.

3.1 Analýza

Protože práce byla velmi přesně zadána, nezbylo příliš prostoru na různé alternativy. Ruby bylo zadáno jako implementační prostředí. Způsob zpracování byl zadán pomocí ACL. Hlavním úkol bylo zjistit, jakým způsobem implementovat samotné ACL a řádně implementaci zdokumentovat a otestovat.

3.1.1 Existující řešení

Při řešení vlastního návrhu na model řízení přístupových práv jsem vycházel ze dvou zdrojů – známých řešení a jednoho obecného řešení.

3.1.1.1 Obecné řešení

Obecným řešením je držet si tabulku, kde ve sloupcích budou objekty, ke kterým je možno přistupovat a v řádcích jsou přístupující. V poli pak jsou hodnoty boolean, které vyjadřují buď allow nebo deny. Příklad tabulkového řešení je v tabulce [3.1](#).

Kdo / Kam	Operační sál	Ambulance	Pokoj pacienta
Chirurg	1	1	1
Sestřička	X	1	1
Pacient	X	X	1

Tabulka 3.1: Jednoduchý příklad obecného řešení modelu práv pomocí matice

3.1.1.2 Oracle

Jedním z řešení je firemní řešení prezentované v Oracle® XML DB Developer's Guide, 11g Release 1 (11.1), Part Number B28369-04 na stránkách Oracle dokumentace [11]. Ve stati Access Control Lists and Security Classes je popsán koncept firmy ORACLE.

Text popisuje několik podmínek a pojetí řízení přístupu. Každá z popsaných entit, uživatel, role, privilegia, bezpečnostní třídy, Access Control List (ACL) a Access Control Entry (ACE), je realizována deklarativně jako XML dokument nebo fragment.

Bezpečnostní autorizace vyžaduje definovat, kteří uživatelé, aplikace nebo funkce mohou mít přístup k jakým datům nebo jaké druhy operací mohou provádět. Existují tedy tři dimenze: (1) kteří uživatelé mohou, (2) vykonávat jaké činnosti, (3) na jakých datech. V souvislosti s každou jednotlivou dimenzí hovoříme o (1) principals - zmocnitelích, (2) privileges - oprávněných, a (3) objektech, které korespondují s těmito třemi dimenzemi. Principals mohou být uživatelé nebo role/skupiny.

Principals a privileges (dimenze 1 a 2) jsou deklarativním způsobem spojeni v definovaných seznamech řízení přístupu - ACL. Ty jsou pak spojené s třetí dimenzí - daty, různými způsoby. Například úložiště zdrojů nebo tabulky dat Oracle XML DB mohou být ochráněny pomocí PL / SQL procedury DBMS\XDB.setACL nastavením jeho řídicího ACL.

3.1.1.3 phpGACL

Druhým ze zdrojů, z nichž jsem vycházel, je řešení prezentované v Generic Access Control List with PHP - phpGACL na [12].

Nástroj phpGACL je sada funkcí, která umožňuje použít řízení přístupu na libovolné objekty (webové stránky, databáze, atd.), jiným libovolným objektům (uživatelé, vzdálené počítače, atd.). Stejně jako Oracle nabízí jemně nastavitelnou kontrolu přístupu s jednoduchou a velmi rychlou správou. Je napsán v populárním dynamickém skriptovacím jazyku PHP.

Nástroj phpGACL vyžaduje relační databáze pro ukládání informací k řízení přístupu. Přistupuje k databázi prostřednictvím tzv. abstraktního obalu ADOdb. Je kompatibilní s databázemi, jako PostgreSQL, MySQL a Oracle.

Nástroj phpGACL používá pojmy jako ACO a ARO:

- Access Control Objects (ACO), jsou věci, ke kterým chceme ovládat přístup (např. webové stránky, databáze, pokoje, atd.).
- Access Request Objects (ARO), jsou věci, které žádají o přístup (např. osoby, vzdálené počítače, atd.)
- ARO stromy definují hierarchii skupin a ARO. Skupiny mohou obsahovat jiné skupiny a ARO.
- Výchozí "catch-all" politikou stromu ARO je vždy "DENY ALL".
- Chceme-li přiřadit přístupovou politiku ve stromu směrem dolů, explicitně přiřazujeme oprávnění skupinám a ARO pro každou ACO, pro kterou je potřeba.

3.2 Návrh implementace

Při návrhu implementace jsem se inspiroval jak Oraclem tak phpGACL. Oba modely řízení přístupových práv mají podobnou strukturu nebo stejnou s jiným pojmenováním. Z Oraclu jsem převzal koncept a pojmenování dimenzí: principals, privileges, objects, ze kterých jsem vytvořil hlavní třídy.

Jemně nastavitelných přístupových práv se docílí pomocí Access Control Listu. ACL obsahuje seznam pravidel jednotlivých přístupů. Pravidlo se nazývá Access Control Entry (ACE). V ACE je uloženo **kdo**, nebo **co** má jaká **práva** přistupovat k jakým **objektům**. Těmto třem rozměrům se v problematice přístupových práv říká: principals - zmocnitelé, privileges - oprávnění, resource objects - zdrojové objekty. Pokud mluvím o ACL objektu, myslím tím ACE nebo principal nebo privilege nebo resource object.

3.2.1 Use Case Scénáře

3.2.1.1 Ověřování oprávnění k objektu

Uživatel má vytvořenou instanci třídy RubyACL, která pracuje s již vytvořenými ACL objekty. Následující scénář popisuje jednotlivé kroky při kontrolování pravidla.

Hlavní úspěšný scénář:

1. Uživatel zavolá metodu `check`. Přes tuto metodu se dotáže knihovny, jestli uživatel/skupina mají nebo nemají oprávnění ke zdrojovému objektu.
2. a) Knihovna vrátí `true` v případě, že uživatel/skupina má specifikované nebo vyšší oprávnění.
2. b) Knihovna vrátí `false` v případě, že uživatel/skupina nemají specifikované nebo vyšší oprávnění.

Rozšíření:

0. Pokud knihovna nenašla žádné vyhovující pravidlo, systém vrátí `false`.

3.2.2 Class diagram

Class diagram 3.1 znázorňuje třídy knihovny a jejich vazby. Diagram je zjednodušen - nejsou vypsány všechny metody a parametry z důvodů přehlednosti a čitelnosti.

Hlavní třídou knihovny je třída RubyACL. Pomocí ní a jejích veřejných metod pracuje uživatelská aplikace s ACL objekty. Třída RubyACL obsahuje jen jednu instanci od každé třídy dědící z ACL_Object. Tyto třídy jsou pomocné třídy, které "znají" strukturu dokumentu uloženého v databázi. Například s pomocí instance třídy ACE lze přidávat, měnit a mazat jednotlivá pravidla, které jsou v databázi v XML souboru reprezentována jako uzly.

Druhou podstatnou třídou je třída ACL_Object, ze které dědí všechny pomocné třídy. Třidu ACL_Object jsem vytvořil, protože se velká část kódu tříd Principal, Privilege, ResourceObject, ACE, Group, Individual shodovala nebo byla velmi podobná. Vyjmenované

podtřídy využívají zděděné metody a případně je konkretizují. V textu tyto třídy nazývám pomocné třídy. Třída `ACL_Object` obsahuje metody, které pomocí `xQuery` a `xPath`¹ obsluhují rozhraní API, v tomto případě `eXistAPI`.

Z diagramu je patrné, že všechna data jsou uložena v databázi ve formě XML dokumentů, kde se na ně dotazuje rozhraní API.

3.2.2.1 Zadávání pravidla (ACE)

Uživatel má vytvořenou instanci `RubyACL`, která obsahuje pravidla. Hlavní úspěšný scénář:

1. Uživatel zavolá metodu `create_ace` a specifikuje údaje (Uživatel/skupina, typ přístupy (allow/deny), oprávnění, zdrojový objekt
2. Knihovna nastaví pravidlo a vrátí identifikační číslo pravidla, když vše proběhlo v pořádku, a vyvolá výjimku když nastala chyba.

Poznámka: Jde vlastně o přiřazování oprávnění uživatelům na objekt.

3.2.3 Rozhraní

S knihovnou `Ruby-ACL` lze komunikovat prostřednictvím rozhraní. Knihovna `Ruby-ACL` se nachází mezi databází (kde jsou uložena data o přístupech) a uživatelskou aplikací (která knihovnu používá). Z tohoto důvodu jsem tuto sekci rozdělil na dvě podsekce. Jedna podsekce popisuje rozhraní mezi knihovnou a databází a druhá podsekce popisuje rozhraní mezi knihovnou a uživatelskou aplikací.

3.2.3.1 Rozhraní mezi Ruby-ACL a databází

Rozhraní mezi `Ruby-ACL` a databází je zprostředkováno pomocí API². `Ruby-ACL` bylo testováno s databází `eXist-db`, se kterou komunikovalo pomocí mnou naimplementovaného rozhraní `eXistAPI`. K používání `Ruby-ACL` s jinou XML databází než `eXist-db` je nutné komunikační rozhraní, které nahradí `eXistAPI`. V ukázkové třídě API je výčet všech potřebných metod, které `Ruby-ACL` používá, včetně popisu parametrů a výstupů. Pro funkčnost knihovny s jinou XML databází je potřeba podle vzorové třídy API naimplementovat nové komunikační rozhraní. Podrobnější popis rozhraní se nachází v příloženém CD jako prázdná třída zdokumentovaná pomocí RDOC³. Na této části rozhraní záleží bezpečnost. Linka mezi API a databází je potenciálně nebezpečné místo k útoku.

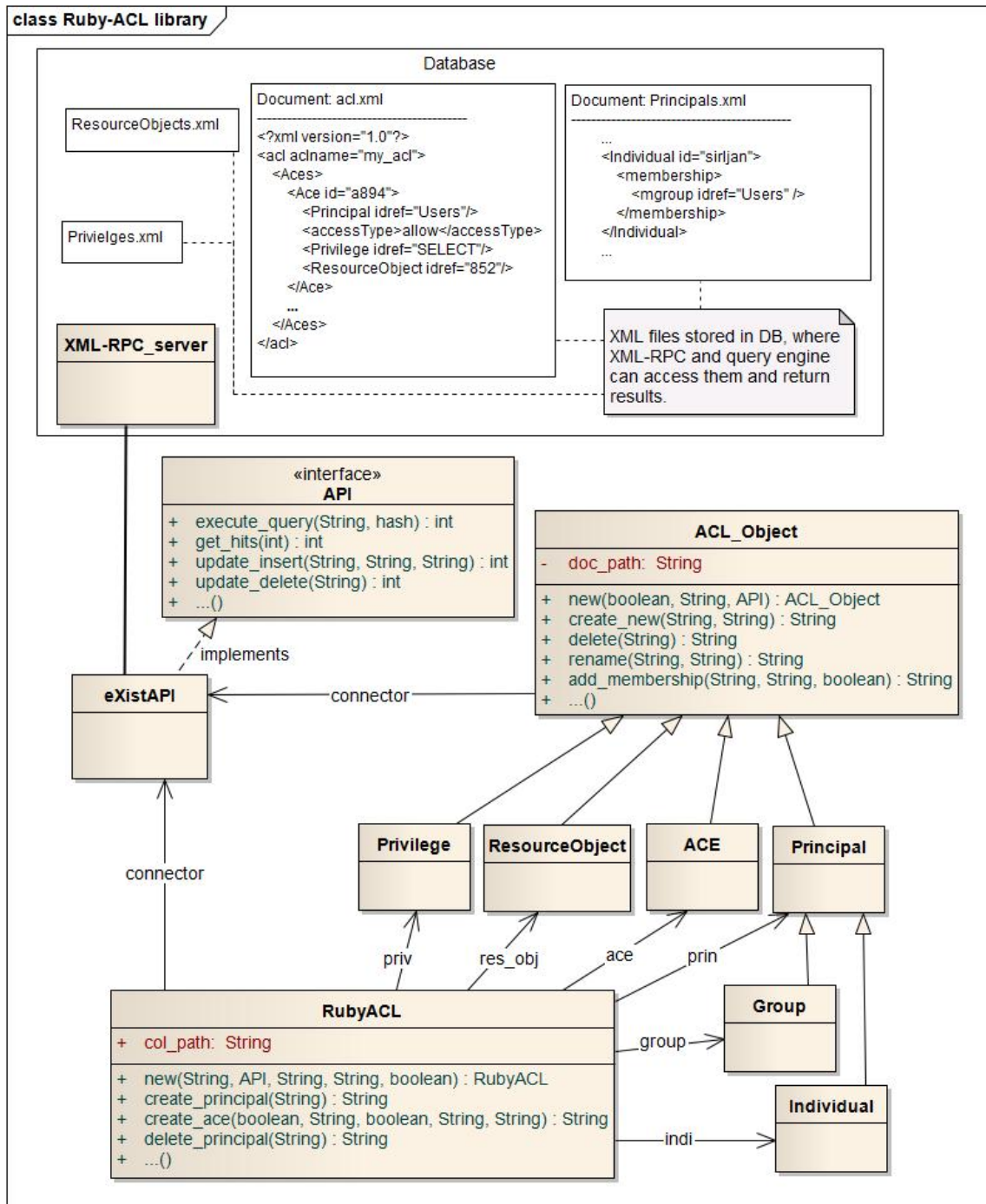
Rozhraní mezi `Ruby-ACL` a databází je patrné na obrázku diagramu komunikace 3.3 v krocích 2 a 5.

Obrázek 3.2 zobrazuje model tříd rozhraní API.

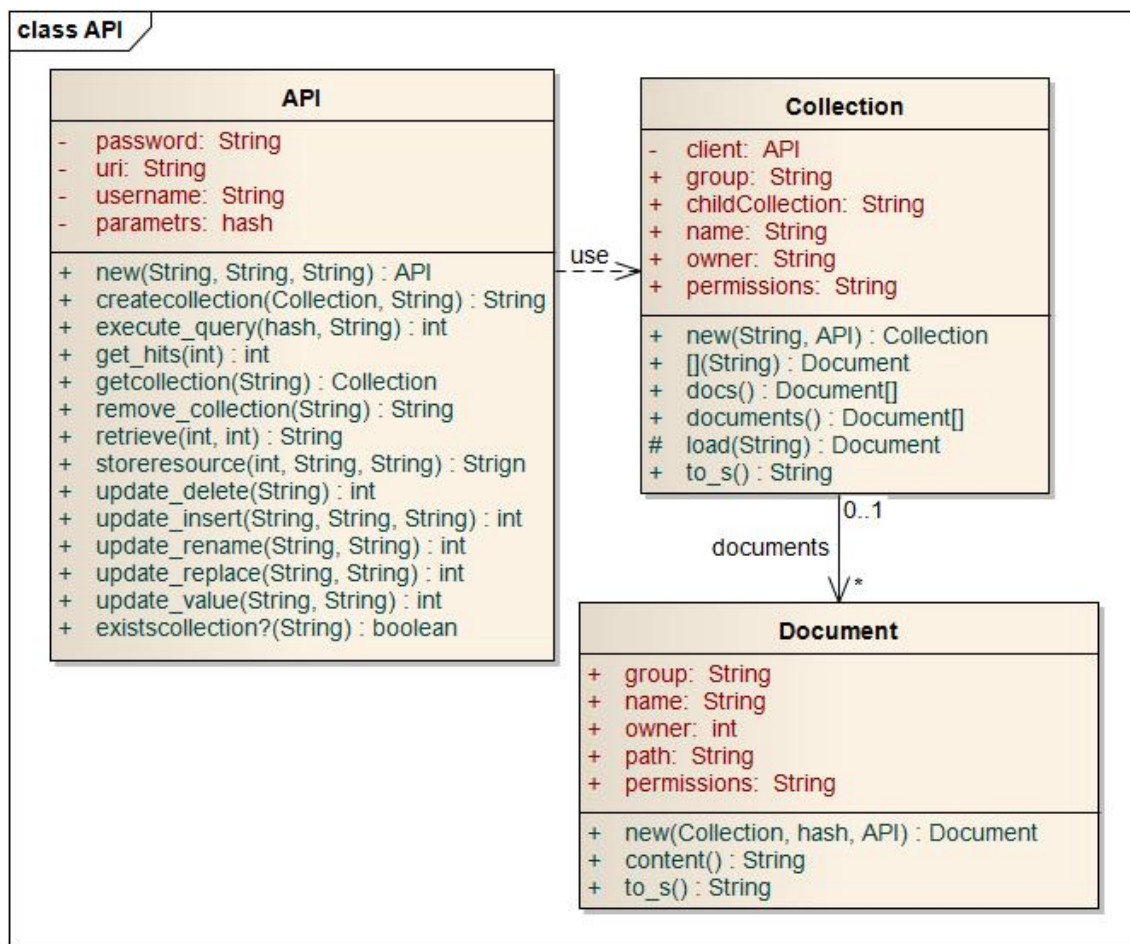
¹XML Path Language

²Application Programming Interface

³Dokumentace pro zdrojové kódy v Ruby



Obrázek 3.1: Class diagram



Obrázek 3.2: Diagram znázorňující model rozhraní

jméno	typ	popis
Vstupy		
principal	string	jméno zmocnitele
privilege	string	název oprávnění
resource object type	string	typ - druh zdrojového objektu
resource object address	string	adresa zdrojového objektu
Výstupy		
access	boolean	true = přístup povolen, false = přístup zakázán

Tabulka 3.2: Parametry a návratové hodnoty metody check

3.2.3.2 Rozhraní mezi Ruby-ACL a uživatelskou aplikací

Rozhraní mezi knihovnou a uživatelskou aplikací tvoří všechny veřejné (public) metody třídy RubyACL. Pomocí instance této třídy a instance třídy API (která je předaná jako parametr) a následném volání veřejných metod může uživatel zavádět zmocnitele, oprávnění, zdrojové objekty a pravidla a pracovat s nimi. Výčet veřejných metod se nachází v příloze B.1.

Nejčastěji používanou částí knihovny bude patrně metoda `check`. Mimo správu ACL objektů je hlavní účel knihovny rozlišit, jestli někdo nebo něco má oprávnění na nějaký zdrojový objekt. Pokud zmocnitel má nebo nemá přístup se uživatel dozví podle výstupu. Výstupem je `true` nebo `false` hodnota. Stručný popis nejčastějších vstupů a výstupů je v tabulce 3.2

3.2.4 Ukázka použití

Tato sekce prezentuje stručné ukázky použití.

3.2.4.1 Příklad nastavení práv

První příklad ukazuje, základní funkčnost knihovny a vytvoření pravidla. Protože knihovna Ruby-ACL vyžaduje jako jeden z parametrů instanci rozhraní API, byla v příkladě použita implementace rozhraní eXistAPI. Pro vytvoření pravidla musí existovat instance RubyACL. Ke správnému vytvoření pravidla musí být všechny ACL objekty vytvořené. Pokud nebudou vytvořené, Ruby-ACL vyhodí výjimku, nebo ACL Objekt založí. Metodě `create_ace` je potřeba předat uživatelské jméno zmocnitele, typ přístupu ("allow" nebo "deny"), oprávnění a požadovaný objekt identifikovaný pomocí typu objektu a adresy.

```

1 require 'eXistAPI'      #must require 'eXistAPI' to comunicated with eXist
  -db
2 #creates instance of ExistAPI
3 @db = ExistAPI.new("http://localhost:8080/exist/xmlrpc", "admin", "
  password")
4 @col_path = "/db/test_acl/"      #sets the collection where you want
  to have ACL in db
5 @src_files_path = "./src_files/" #path to source files
6 @my_acl = RubyACL.new("my_acl", @db, @col_path, @src_files_path, true)

```

```

7 #it's good to create some principals at the begging
8 @my_acl.create_principal("Sheldon")
9 @my_acl.create_privilege("SIT")
10 @my_acl.create_resource_object("seat", "/livingroom/couch/Sheldon's_spot", "Sheldon") #(type, address, owner) of resource object
11 @my_acl.create_ace("Sheldon", "allow", "SIT", "seat", "/livingroom/couch/Sheldon's_spot") #(principal, acc.type, privilege, resOb type, adr)

```

3.2.4.2 Příklad kontroly práv

Následující příklad prezentuje možné použití knihovny a její metody `check`. V případě, že metoda vrátí hodnotu `true`, přístup je povolen. Pokud vrátí hodnotu `false`, tak je přístup zamítnut.

```

12 #Next method in if returns deny
13 if(@my_acl.check("Penny", "SIT", "seat", "/livingroom/couch/Sheldon's_spot"))
14     puts "Access allowed. You may retrieve resource object."
15 else
16     puts "Access denied."

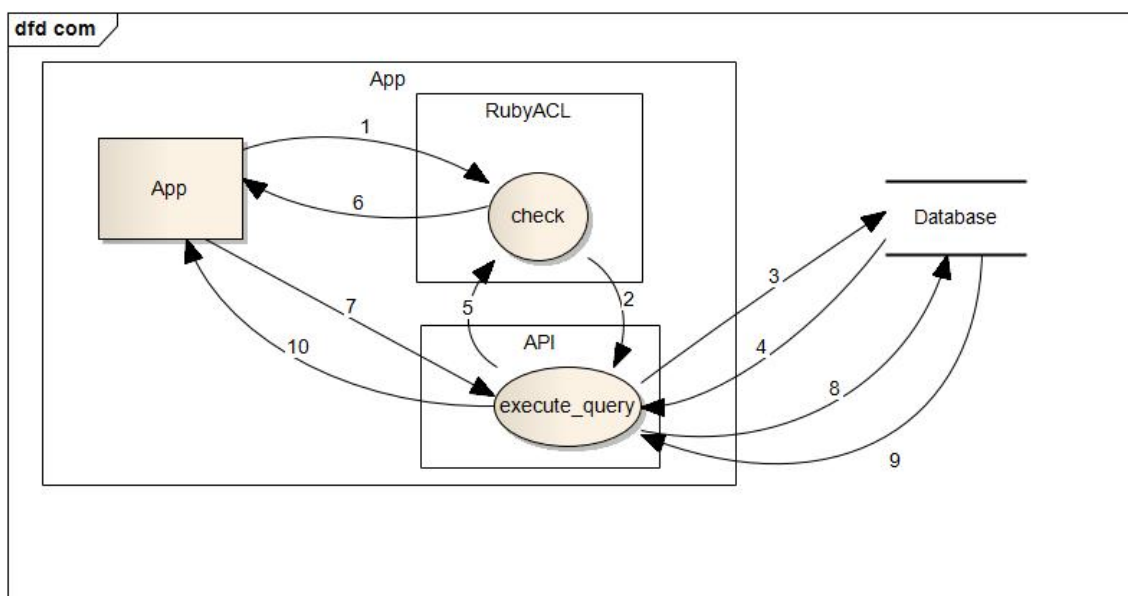
```

Co se děje při volání metody `check` názorně vysvětluje obrázek 3.3 zobrazující sekvenci komunikace. Kroky 7 a vyšší jsou volitelné.

1. Výkonná část uživatelské aplikace zavolá metodu `check` třídy `RubyACL`
2. Metoda `check` vytvoří xQuery⁴ dotaz a předá ho implementaci rozhraní API zavoláním metody `execute_query`
3. API se pomocí XML-RPC⁵ protokolu dotáže databáze
4. databáze vrátí výsledek/výsledky
5. API předá výsledek/výsledky metodě `check`
6. Metoda `check` z výsledků rozhodne, jestli principal má oprávnění na zdrojový objekt a podle toho vrátí boolean hodnotu. Více o prioritě rozhodování se lze dočíst v sekci 3.3.2.1 - Priorita rozhodování.
7. Uživatelská aplikace zavolá `execute_query`
8. `execute_query` pomocí XML-RPC zažádá o objekt

⁴Dotazovací jazyk nad XML dokumenty

⁵XML Remote Procedure Call protokol



Obrázek 3.3: Diagram znázorňující sekvenci komunikace

9. databáze vrátí objekt

10. `execute_query` předá objekt uživatelské aplikaci

3.3 Popis logiky vyhodnocování pravidel

V této kapitole je popsáno, jakým způsobem Ruby-ACL rozhoduje, jestli je přístup povolen či nikoliv. Nejkonkrétněji se tímto zabývá sekce "Pravidla rozhodování", nicméně k pochopení celku je dobré vědět vlastnosti jednotlivých ACL objektů.

3.3.1 ACL Objekty

Jak již bylo v předchozí části textu řečeno. ACL objekt je principal, privilege, resource object, ACE. Principal se dále dělí na individual a group. Všechna data uložená v databázi ve formě XML dokumentů jsou vlastně ACL objekty. Každý ACL objekt má dedikovaný soubor vyjma individual a group. Tyto ACL objekty se ukládají do `Principals.xml`, protože jsou podmnožinou principal.

Ruby-ACL pracuje s daty tak, že za pomoci jedné ze tříd, které dědí ze třídy `ACL_Object`, edituje XML soubory v databázi. Upravuje je takovým způsobem, že přidává, maže nebo mění jednotlivé uzly.

Struktura XML souborů je popsána pomocí stenojmenných DTD⁶ souborů v příloze na CD. Strukturu pro každý ACL objekt vyjadřují v příslušné sekci i slovně.

⁶Document Type Definition

3.3.1.1 Zmocnitel (Principal)

Principal nebo-li zmocnitel může být jednotlivec nebo skupina. Jednotlivec může být člověk jako uživatel nebo proces, aplikace, připojení, zkratka cokoliv, co vyžaduje přístup.

Skupiny a jednotlivci se ukládají do souboru `Principals.xml`. Vyjádření struktury slovy, je následující. `Principals.xml` obsahuje kořenový uzel `Principals`, ten obsahuje pouze dva uzly `Groups` a `Individuals`. V uzlu `Groups` je neomezené množství uzlů `Group`, které reprezentují skupiny. V uzlu `Individuals` je neomezené množství uzlů `Individual`.

Jednotlivci a skupiny se mohou sdružovat do skupin. Každý jednotlivec nebo skupina má v sobě uzel `membership`, ve kterém je neomezené množství uzlů `mgroup` s atributem `idref`, který odkazuje na danou skupinu. Nelze, aby jednotlivec byl členem v jednotlivci.

```

1 <Principals>
2   <Groups>
3     <Group id="ALL">
4       <membership/>
5     </Group>
6     <Group id="Users">
7       <membership>
8         <mgroup idref="ALL"/>
9       </membership>
10    </Group>
11  </Groups>
12  <Individuals>
13    <Individual id="sirljan">
14      <membership>
15        <mgroup idref="Users" />
16        <mgroup idref="Developers" />
17      </membership>
18    </Individual>
19  </Individuals>
20 </Principals>

```

3.3.1.2 Oprávnění (Privilege)

Ruby-ACL nabízí výchozí oprávnění, které jsem převzal od Oracle a MySQL. To jestli je uživatel použije, záleží na něm. Výchozí privilegia jsou: 'ALL PRIVILEGES', 'ALTER', 'CREATE', 'DELETE', 'DROP', 'FILE', 'INDEX', 'INSERT', 'PROCESS', 'REFERENCES', 'RELOAD', 'SELECT', 'SHUTDOWN', 'UPDATE' a 'USAGE'. Pravidla lze vytvářet a seskupovat do stromové struktury.

Oprávnění se ukládají do souboru `Privileges.xml`. Ten obsahuje kořenový uzel `Privileges`, ve kterém je neomezené množství uzlů `Privilege`. Stejně jako `Individual` nebo `Group`, `Privilege` obsahuje uzel `membership` a v něm neomezené množství uzlů `mgroup`, které pomocí `idref` odkazují na nadřazené oprávnění.

```

1 <Privileges>
2   <Privilege id="SELECT">

```

```

3      <membership>
4          <mgroup idref="ALL_PRIVILEGES" />
5      </membership>
6  </Privilege>
7 </Privileges>

```

3.3.1.3 Zdrojový objekt (Resource object)

Zdrojové objekty jsou ukládány do souboru `ResourceObjects.xml`. Struktura je opět podobná předchozím ukázkám. Kořenový uzel je `ResourceObjects` a v něm je neomezené množství uzlů `ResourceObject`.

```

1 <ResourceObjects>
2   <ResourceObject id="r753951654">
3     <type>doc</type>
4     <address>/db/cities/cities.xml/cities</address>
5     <owner idref="sirljan"/>
6   </ResourceObject>

```

Zdrojový objekt se skládá ze tří položek: typ, adresa, vlastník. Typ může být dokument, nebo reálné objekty jako dveře, místnosti apod. Adresa společně s typem identifikuje objekt. Zdrojový objekt jasně identifikuje i id v parametru každého `ResourceObject` uzlu, ale uživatel velmi pravděpodobně nebude vědět id zdrojových objektů. Uživatel by sice mohl id zjistit pomocí metody `find_res_ob` třídy `ResourceObject`, ale stejně by musel zadat typ a adresu. Vlastník objektu může být jakýkoliv principál.

Při zadávání adresy je potřeba dodržet jediné pravidlo. V adrese oddělovat každý jednotlivý objekt dopředným lomítkem (/).

Pokud je zdrojový objekt typu "doc", adresa může obsahovat klauzuli ve formátu '("/kořen/větev/list-soubor_s_příponou")' a pokud je nutné jemnější řízení přístupu uvnitř dokumentu, následuje klauzule "/jiný_kořen/jiná_větev". Nicméně do databáze se ukládá sloučená adresa bez závorek a uvozovek ("").

Příklad: '("/db/data/cities.xml")/cities/city' v databázi je uložen pod adresou: "/db/data/cities.xml/cities/city". Tento způsob byl zvolen kvůli jednoduššímu rozhodování a parsování.

Při zakládání nového zdrojového objektu je potřeba typ objektu oddělit od adresy. Například vkládám-li objekt 'doc("/db/cities/cities.xml")/cities', rozdělím objekt na dvě části: typ, v tomto případě "doc", a zbylou adresu '("/db/cities/cities.xml")/cities'. Klauzule "/"* (lomítko hvězdička) vyjadřuje všechny podřadné objekty.

Například objekt "/db/data/*" vybere všechny podřadné objekty objektu data.

Vlastník/Owner může být jednotlivec, množina jednotlivců i skupina. Množina jednotlivců je v případě, pokud se od kořene zdrojových objektů k listu mění vlastník. Vlastník nejnadřazenějšího zdroje má největší práva. Vlastník má veškerá práva na daný objekt a všechny jeho podobjekty.

Vlastník objektu `"/db/data"` má vlastnictví tohoto objektu i všech podřízených, například i takového objektu `"/db/data/e-books"`.

Vlastník objektu `"/db/data/*"` má vlastnictví pouze podřízených objektů.

3.3.1.4 Pravidlo (ACE)

Pokud přijde žádost o vytvoření ACE s neexistujícím zdrojovým objektem, knihovna objekt vytvoří. Z toho vyplývá, že v databázi existují všechny zdrojové objekty z pravidel sjednocené s objekty, které byly vytvořeny přímo. Vlastníkem takového objektu se automaticky stává principal uvedený v pravidlu. Je třeba si proto dávat pozor na to, jestli objekt už existuje, nebo ne. Pokud objekt neexistuje, principal tím automaticky získává veškeré oprávnění.

Pravidla (ACE) se ukládají do souboru `acl.xml`. V kořenovém uzlu `acl` je atribut `aclname`, ve kterém je uloženo jméno seznamu. Uvnitř uzlu `acl` je pouze jeden uzel `Aces` a vněm je libovolné množství uzlů `Ace`. Každý uzel `Ace` má svoje id uložené v atributu a uvnitř `Ace` jsou uzly: `Principal` - reference na zmocnitel, `accessType` - přístupový typ, `Privilege` - reference na oprávnění a `ResourceObject` s referencí na zdroj.

```

1 <acl aclname="my_first_acl">
2   <Aces>
3     <Ace id="a894">
4       <Principal idref="Users"/>
5       <accessType>allow</accessType>
6       <Privilege idref="SELECT"/>
7       <ResourceObject idref="852"/>
8     </Ace>
9   </Aces>
10 </acl>

```

`AccessType` rozhoduje o tom, jestli oprávnění bude povoleno, nebo zakázáno. Je to obdoba příkazu `revoke` a `grant` z Oracle a MySQL, kde `allow` = `grant` = povolit přístup a `deny` = `revoke` = zakázat/odebrat přístup.

3.3.2 Pravidla rozhodování

3.3.2.1 Priorita rozhodování

Nejnižší číslo v seznamu má největší prioritu. Slovy se dá následující výpis pravidel vyjádřit jako: Je-li zmocnitel vlastníkem objektu, knihovna okamžitě vrátí `true`. Konkrétní zákaz má přednost před konkrétním povolením. Zděděný zákaz má přednost před zděděným povolením. Konkrétní mají přednost před zděděnými. Pokud není nalezeno pravidlo, knihovna vrátí `false`. Pokud knihovna podle zmíněné priority nalezne pravidlo s `deny`, vrátí `false`. Pokud nalezne `allow`, vrátí `true`.

1. Owner
2. Explicit Deny

3. Explicit Allow
4. Inherited Deny
5. Inherited Allow
6. If not found > Deny

- allow - knihovna vrátí **true** - přístup povolen
- deny - knihovna vrátí **false** - přístup zakázán
- nenalezeno - knihovna vrátí **false** - přístup zakázán

3.3.3 Složitost rozhodování

Složitost rozhodování závisí na metodě **check**. Rozhodování probíhá tak, že si knihovna připraví všechny zdrojové objekty, které se mohou týkat objektu, u kterého se zjišťuje přístup stejně tak všechny oprávnění a zmocnitelé. Ve všech třech případech se jedná o pole obsahující samotný ACL objekt a všechny jeho nadřazené ACL objekty. Pro každý principal se vytvoří xQuery dotaz, který se následně pošle databázi na vyhodnocení. Metoda **check** tedy obsahuje jednu vnořený cyklus, proto je složitost přibližně e^2 .

Kapitola 4

Realizace

Popis implementace/realizace se zaměřením na nestandardní části řešení.

4.1 Průběh realizace

Průběh realizace jsem si zpětně rozdělil do několika fází, tak jak šly chronologicky po sobě. Každá fáze představuje určitou etapu ve vývoji knihovny.

Fáze 1

Po analýze a návrhu byla naimplementována zkušební část kvůli ověření návrhu tříd a rozhraní. Na tuto verzi byla napsána první velká skupina unit testů. Verze nepracovala vůbec s databází, byla pouze instanční a od současného modelu tříd se velmi odlišovala. Tato část práce byla součástí mého softwarového projektu.

Fáze 2

Po dokončení instanční verze jsem pokračoval seznámením se s databází eXist-db, která mi byla doporučena vedoucím práce. V této fázi nastaly největší potíže. Nemohl jsem najít způsob, jak komunikovat s databází. V Ruby neexistovalo žádné komunikační rozhraní, které by umožňovalo měnit obsah a dotazovat se na něj. Dokumentace přímo na webu [13] popisovala jenom protokoly, kterými lze s databází komunikovat. Mezi ně patří například REST¹, SOAP² a XML-RPC, který jsem si zvolil pro vývoj API. Velice mi pomohlo, nalezení ukázky knihovny pro XML-RPC v instalačním adresáři eXist-db.

Komplikace tím neskončily. Uměl jsem sice pomocí XML-RPC pracovat s kolekcemi a dokumenty, ale nedokázal jsem pomocí xUpdate upravovat dokumenty: vkládat, upravovat a mazat uzly a atributy. Dodržel jsem dokumentaci XML-RPC, specifikace metody `XUpdate` [2] a dokumentaci `XUpdate` [1], na kterou se eXist odvolává. V této fázi se mi delší dobu nedařilo zporvoznit upravování dokumentů. Proto jsem začal vyhledávat alternativní možnosti. Hledal jsem XML databázi, která by měla komunikační rozhraní v Ruby. Našel jsem jedinou databázi jménem Sedna. Bohužel ani s ní jsem nedospěl pokroku. Problém byl se zprovozněním knihoven, které byly napsané v jazyku C.

¹Representational State Transfer

²Simple Object Access Protocol

Významnou pomocí v této fázi pro mě byla rada vedoucího mojí bakalářské práce Ing. Strnada, který mě nasměroval na použití technologie xQuery Update Facility.

Fáze 3

Přepracoval jsem první verzi implementace RubyACL z lokální/instanční na databázovou, což představovalo přepsání tříd na pseu-singlotonské managery. Třídy nejsou čistými single-tony, ale knihovna obsahuje pouze jednu instanci od každé pomocné třídy. Všechny informace jsou uloženy v databázi. RubyACL pouze vkládá nové záznamy, upravuje a maže staré. Na existujících záznamech provádí dotazy a rozhodování o přístupu. Při úplném naimplementování eXistAPI, jsem kompletně předělal třídní model. Do této doby byly třídy jako Principal, Privilege, ResourceObject a ACE používány pro ukládání dat do instancí a polí instancí. Ve fázi 3 jsem tyto třídy předělal na pomocné třídy, které upravují pomocí poskytnutého rozhraní obsah databáze. Přidal jsem třídy Individual a Group. Dalším nezdarem v této části bylo vytvoření nějakého referenčního systému mezi ACL objekty. Pro identifikaci a propojení jsem uvažoval mezi xLink a idref. Idref nabízí jednoduchý systém, ale xLink nabídl podrobné specifikace W3C, velké množství návodů a turtoriálů a především se jedná o novější technologii, jejíž osvojení jsem považoval za výhodné. Rozhodl jsem se proto implementovat xLink. Problém nastal při některých vkládání textů a dotazování. eXist-db měla problém s jmeným prostorem xLink i přes skutečnost, že jmený prostor byl uveden jak v samotném XML dokumentu, tak ve vkládaném uzlu. I když jsem se zabýval proč použití xLink nejde, nedospěl jsem k rozřešení a tak jsem raději přešel na jednoduchý idref. Implementace idref proběhla rychle a bez problémů.

Fáze 4

Druhý návrh tříd nebyl zcela správný a i implementace se odchylovala od návrhu. Znovu jsem musel zasáhnout do modelu tříd a předělat jej. Vytvořil jsem nadřazenou třídu `ACL_Object`, ze které dědí ostatní pomocné třídy. V této fázi jsem velmi intenzivně testoval. Opakované ověřování správné funkčnosti při neustálém upravování knihovny, mi zabíralo příliš mnoho času. V této fázi jsem docenil význam unit testů a jejich schopnost odhalovat chyby.

Fáze 5

Dokončil jsem rozhodovací metodu `check` a dotáhl do finální podoby funkcionality knihovny. V této fázi jsem opravil chyby a odstranil nedostatky. Přidal jsem další velkou část unit testů, převážně zaměřených na metodu `check`. Začal jsem tvořit dokumentaci.

Fáze 6 TODO Finalizování - rubygem. Dokumentace. Codecoverage

4.2 Programy použité při vývoji

V následujícím seznamu jsou uvedeny všechny programy, které jsem při vývoji knihovny použil.

- NetBeans s Ruby platformou od Thomase Enebo
<<http://blog.enebo.com/>>
<<http://plugins.netbeans.org/plugin/38549/ruby-and-rails>>

- GIT <<https://github.com/sirljan/Ruby-ACL>>
- eXist-db <<http://www.exist-db.org>>
- Enterprise Architect
- MikTex

4.3 Databáze

Protože XML databáze, která má RubyACL používat, není dosud plně funkční, nebylo možné implementaci testovat přímo na ní. Za tímto účelem se musela vybrat jiná podobná databáze.

4.3.1 Sedna

Databáze Sedna poskytuje rozhraní v Ruby. Klientské rozhraní je Ruby rozšíření, které používá ovladač jazyka C, který je dodáván jako součást distribuce Sedna. Použití má být jednoduché a snadno použitelné. Ovšem zprovoznění knihovny Sedna je nedostatečně popsané a nekompatibilní se současnou verzí Ruby 1.9.3.

4.3.2 eXist-db

Open source systém eXist-db je systém pro správu databáze vytvořené pomocí technologie XML. Databáze eXist-db ukládá XML data podle datového modelu XML a nabízí efektivní a XQuery zpracování založené na indexování. Podporuje velké množství technologií. Dále zde uvádím pouze ty, které se týkají RubyACL nebo eXistAPI: XPath 2.0, XQuery, XML-RPC, XQuery Update Facility 1.0.

eXist-db se podobá XML databázi, která má RubyACL používat, a byla doporučena vedoucím práce jako ideální. Přesto se vyskytly komplikace s XUpdate navzdory přesné interpretaci dokumentace. Z tohoto důvodu bylo nutné přejít na XQuery Update Facility.

4.3.2.1 eXistAPI

ExistAPI je komunikační rozhraní, jehož implementaci si vynutilo ladění a testování knihovny RubyACL na databázi eXist-db. Jednalo se o trochu nestandardní část realizace, protože eXist-db nemá knihovnu rozhraní pro jazyk Ruby.

eXistAPI komunikuje s databází prostřednictvím XML-RPC protokolu a technologií XQuery a XPath. XQuery Update Facility se používá pro editování dat v dokumentech kolekce databáze. Pokus o naimplementování XUpdate nebyl úspěšný. XUpdate není podporován konzorciem W3C³ a tudíž není jednoznačně nadefinovaný. Zde nejspíše vznikl problém, kdy eXist-db použila jinou interpretaci XUpdate, než uvádí dokumentace XUpdate [1].

Interface eXistAPI považuji za přínos pro uživatele eXist-db, kteří programují v Ruby nebo to mají v plánu. Rozhraní eXistAPI je ošetřené výjimkami, otestované a zdokumentované tak, jak to vyžaduje balíčkový systém **RubyGem**, kde ho lze najít pod jménem

³World Wide Web Consortium

"eXistAPI". Lze jej tedy stáhnout příkazem `gem install eXistAPI`. Model tříd je v příloze [D.2](#).

4.4 Implementace

V této sekci se zabývám popisem implementace metody `check` a rozepsání a popsání příkladu založení uživatele.

4.4.1 Třída RubyACL

Hlavní účel třídy `RubyACL` je zprostředkovávat funčnost pomocných tříd dědicích z třídy `ACL_Object`. Dále sama nabízí možnost ukládání a načítání ACL ze souborů a hlavně obsahuje metodu `check`, která slouží k vyhodnocování pravidel.

4.4.1.1 Metoda check

Metoda `check` obsatarává rozhodování, jestli zmocnitel má oprávnění k objektu nebo nemá. Stručně a výstižně vysvětluje algoritmus obrázek [4.1](#).

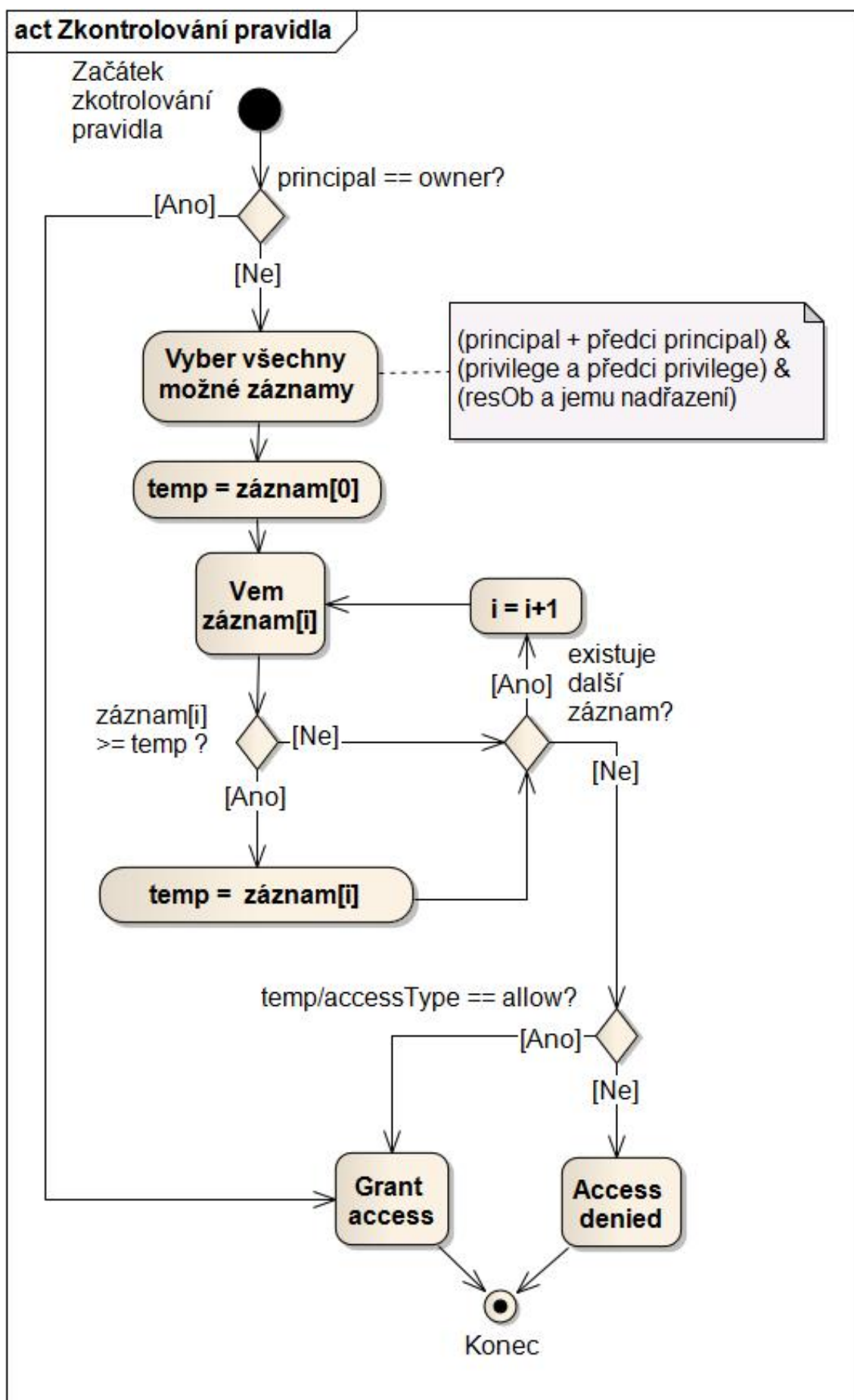
Metoda `check` nejprve zjistí, jestli daný zmocnitel není vlastník daného objektu nebo jakéhokoliv nadřazeného objektu. Pokud je ihned vrátí `true` a ukončí běh. Pokud zmocnitel není vlastníkem, metoda `check` nashromáždí všechny potenciální pravidla, která by mohly ovlivňovat přístup. Udělát to tak, že pro daného zmocnitele najde všechny jeho nadřezené zmocnitele. Jedná se tedy o pole skupin a pokud je daným zmocnitelem jednolivec, tak i on. Obdobným způsobem vytvoří i pole privilegií a nadřazených privilegií. zdrojových objektů a nadřazených zdrojových objektů. Dále z těchto polí vytvoří XQuery dotaz, který vybere takové pravidlo, které obsahuje jednoho ze zmocnitelů a zároveň jedno z pravidel a zároveň jeden zdrojový objekt. XQuery se odešle pomocí `eXistAPI` a přijme se množina pravidel. V této množině pravidel hledá metoda `check` pravidlo, které má nejvyšší prioritu. Metoda určuje prioritu podle seznamu v sekci [3.3.2.1](#) Priorita rozhodování. Metoda prochází pravidla a srovnává pravidlo uložené ve pomocné proměnné s aktuálním pravidlem. Pokud aktuální má větší prioritu zapíše ho do pomocné proměnné. Pokud není žádný další záznam, metoda přečte `accessType` a podle něj rozhodne, co vrátí. Pokud `accessType` obsahuje "allow" vrátí `true`, pokud obsahuje "deny" vrátí `false`.

4.4.1.2 Příklad

Uvedu zde jeden příklad vytvoření zmocnitele a na něm vysvětlím `TODO`.

```
@my_acl.create_principal("Sheldon", "4th_Level")
```

Je zřejmé, že příkaz volá metodu `create_principal`. Příkaz má vytvořit zmocnitele jménem "Sheldon", který bude patřit do skupiny "4th_Level". `RubyACL` má instanci od každé pomocné třídy. `RubyACL` tedy pouze zavolá metodu `create_new` instance `@indi`, což je instance třídy `Individual`. Metoda `create_new` ničím nekonkrétizuje chování supertřídy



Obrázek 4.1: Diagram znázorňující chod metody check

`ACL_Object`, takže se zavolá metoda této třídy. Zde se provede kontrola, jestli jméno není prázdné. Pokud jméno je prázdné vyhodí se výjimka. Dále se pokračuje kontrolou, jestli jméno už není obsazené. Pokud je, vyhodí se výjimka. Pomocí metody `generate_expr` se vygeneruje uzel, který bude následně skládat. Metoda `generate_expr` není ve třídě `Individual` přepsaná (anglicky `overridden`), takže se v tomto případě volá metoda supertřídy. Výsek kódu tvoření uzlu je v následujícím výpisu:

```

    expr = <<END
    <#{self.class.name} id="#{name}">
      <membership>

      </membership>
    </#{self.class.name}>
END

```

Kód ukazuje, že jméno uzlu je utvořeno podle jména třídy (jedná se o část `self.class.name`), takže jméno uzlu je v tomto případě "Individual". Dalším krokem je vytvoření části příkazu, který rozhoduje kam se uzel bude vkládat a opět se využívá jména třídy.

```

expr_loc = "#{@doc} // #{self.class.name}s / #{self.class.name}[last()]"

```

Zbývá už pouze zavolat metodu `update_insert` instance třídy `eXistAPI`, která vloží připravený uzel na připravenou pozici. Následně se zkontroluje, pokud byl uzel vložen, pokud ne, vyvolá se výjimka.

Následuje větvení, které rozhoduje zda se bude zmocnitel přidávat do nějaké skupiny. V tomto případě se zmocnitel do skupiny bude přidávat. Přidávání se provádí pomocí metody `add_membership`, kterou lze volat i samostatně. Parametry metody jsou: "kdo má být", "v jaké skupině". Metoda zjednodušeně řečeno vkládá do zmocnitele odkaz na skupinu. `add_membership` funguje podobně jako `create_new` - připraví se uzel na vkládání, připraví se lokace vložení jako XPath výraz a předtím než se zavolá `update_insert` se zkontroluje, jestli se členství necyklí. Nakonec se zkontroluje, pokud byl uzel vložen, pokud ne, vyvolá se výjimka.

4.4.2 ACL_Object

Třída `ACL_Object` je rodičem tříd: `Principal`, `Privilege`, `ResourceObject`, `ACE` a nepřímým rodičem `Individual` a `Group`. Třída sdružuje obecné metody, které jsou společné pro všechny výše vyjmenované třídy. Takové metody například jsou: `create_new`, `delete`, `rename`.

`ACL_Object` také obsahuje důležitou proměnnou `doc`, která obsahuje cestu k datovému dokumentu v databázi. Tuto proměnnou taktéž třídy dědí. Jednolivé třídy vyjmenované v této sekci pracují se svým dokumentem. Dokument a uzly v něm mají různou strukturu. Například oprávnění má jeden kořenový uzel a pak už jen list oprávnění, ve kterých jsou odkazy na nadřazené oprávnění. Čili se jedná o poměrně jednoduchou strukturu. Každá ze tříd má tuto strukturu v sobě uloženou a podle ní tvoří dotazy na editování dat.

4.4.3 Vyjímky

Vyjímky pro knihovnu RubyACL vyvolává třída `RubyACLError`. Knihovna buď vyjímky vyvolává nebo je jen propaguje dále. Seznam vyjímek, jejich kódů a metod, které je vyvolávají jsou vypsány v příloze TODO ?? a na přiloženém CD přímo pod zdrojovým kódem třídy `RubyACLError`.

Vyjímky pro rozhraní eXistAPI vyvolává třída `ExistException`.

4.4.4 Práce s pamětí

Na začátku mi bylo jasné, že bych neměl zanedbat práci s pamětí, protože aplikace s knihovnou bude spuštěna v řádech dnů a měsíců. Špatné zacházení s pamětí by, proto bylo kritické pro server, na kterém aplikace s Ruby-ACL knihovnou bude spuštěna. Ruby-ACL veškeré data ukládá do databáze. Jediné obsazení paměti je instancí samotné třídy `RubyACL` a instancemi pomocných tříd. Nicméně v Ruby uvolňování paměti programátorem nemá význam, protože v dynamickém prostředí Ruby se o uvolňování stará garbage collector. Ruby používá garbage collector, který nepoužívané objekty vystopuje a odstraní [4].

Kapitola 5

Testování

- Způsob, průběh a výsledky testování.
- Srovnání s existujícími řešeními, pokud jsou známy.

Testování bylo zaměřeno pouz na funkcionalitu. Vynecháno bylo testování rychlosti, práce s operační pamětí, spotřeby procesorového času. Testoval jsem unit testama za pomoci Ruby modulu `Test::Unit` a informací z knihy Ruby - kompendium znalostí [4] a online tutoriálu [7]. Test Coverage TODO

5.1 RubyACL

Testoval jsem každou veřejnou metodu u RubyACL. Celkem jsem vytvořil 43 testů s 119 asercema. Testy jsem rozdělil do 4 částí-souborů podle charakteru metod, které testují. Set1 testuje metody přímo spjaté s ACL, například load, save and rename. Set2 testuje metody, které vytváří ACL objekty. Set3 testuje metody, které upravují ACL objekty. Set4 testuje pouze metodu check. Testování probíhá neobvykle delší dobu, protože před každým spuštěním nového testu se do databáze nahrávají testovací dokumenty a po každém testu se maže kolekce s testovacími daty. Hlavně nahrávání dokumentů probíhá poměrně dlouhou dobu. Čas provedení testů trvá asi 80 sekund.

5.2 eXistAPI

Testoval jsem každou veřejnou metodu eXistAPI. Celkem jsem vytvořil 14 testů s 21 asercema.

Kapitola 6

Závěr

- Zhodnocení splnění cílů DP/BP a vlastního přínosu práce (při formulaci je třeba vzít v potaz zadání práce).
- Diskuse dalšího možného pokračování práce.

Časově nejnáročnější operací je komunikace s databází. Zrychlení by se dalo docílit optimalizací dotazování tak, aby se dotazovalo, co nejméně. Záporom tohoto řešení by byla větší paměťová náročnost aplikace, která by knihovnu Ruby-ACL používala. Současný kód knihovny obsahuje opakované dotazování na stejnou věc místo ukládání výsledku do paměti pro případné následující použití. Tento model byl zvolen kvůli jednoduchosti a faktu, že vykonání dotazu a vrácení výsledku trvá řádově milisekundy. Je ale těžké odhadnout, kolik času by zabralo dotazování při plné databázi a použití více uživatelů ve stejný čas. Navíc eXist-db si po určitou dobu v paměti uchovává výsledky předchozích dotazů, čímž se ještě více snižuje náročnost opakovaných dotazů. Vše je ale závislé na lince mezi knihovnou a databází.

Kapitola 7

Moje pracovni poznamky

- id u resob. Pokud to bude poradi, mohlo byt snaze napadnutelne nez kdyby id nedavalo smysl.

- eXist xpath chyba. Na dotaz

```
doc("/db/test_acl/ResourceObjects.xml")/ResourceObjects/descendant::*[type="Rybnik"]
```

ale na dotaz

```
doc("/db/test_acl/ResourceObjects.xml")/ResourceObjects/descendant::*[type="Rybnik"]
```

- res_ob se muzou vytvorit dopredu v seznamu res_obs. Res_ob musi byt kazdopadne v seznamu (vytvori se pri vytvoreni pravidla).
- Use “all” or “*” to match any address.
- vysledek rozhodnuti jde nechranenou cestou. pokud ma napadajici pristup ke kanalu mezi db a rubyacl potazmo db aplikaci, tak muze zmenit vysledek dotazu a tim i rozhodnuti.
- to_s jsem vynechal, protoze knihovna bude pouzita aplikaci a ne clovekem, který by si mohl precist stringovy obsah.
-

Literatura

- [1] Andreas Laux, Lars Martin. xUpdate documentation, 2012. Dostupné z: <<http://xmldb-org.sourceforge.net/xupdate/xupdate-wd.html>>.
- [2] eXist authors. eXist-db, Developer's Guide - XML-RPC - XUpdate, 2012. Dostupné z: <<http://exist-db.org/exist/devguide/xmlrpc.xml#d41536e1573>>.
- [3] FOWLER, M. *Destilované UML*. Grada Publishing, 1st edition, 2010. In Czech.
- [4] FULTON, H. *RUBY, kompendium znalostí*. Zoner Press, 1st edition, 2010. In Czech.
- [5] Jamie Hill, SonicIQ Ltd. *iq-acl* [online]. 2012. [cit. 16. 5. 2012]. Dostupné z: <<https://github.com/soniciq/iq-acl>>.
- [6] Oleg Dashevskii. *acl9* [online]. 2012. [cit. 16. 5. 2012]. Dostupné z: <<https://github.com/be9/acl9>>.
- [7] Pat Eyler. *Test-first programming with Ruby* [online]. 2012. [cit. 16. 5. 2012]. Dostupné z: <<http://www.ibm.com/developerworks/opensource/tutorials/os-ruby1/section4.html>>.
- [8] Peter Schrammel, Gregor Melhorn. *Active Access Control Lists Plus (ActiveAclPlus)* [online]. 2012. [cit. 16. 5. 2012]. Dostupné z: <<http://activeaclplus.rubyforge.org/>>.
- [9] Příspěvatelé Wikipedie. *Ruby (programovací jazyk)* [online]. 2012. [cit. 16. 5. 2012]. Dostupné z: <[http://cs.wikipedia.org/wiki/Ruby_\(programovac%C3%AD_jazyk\)](http://cs.wikipedia.org/wiki/Ruby_(programovac%C3%AD_jazyk))>.
- [10] Příspěvatelé Wikipedie. *Access control list* [online]. 2012. [cit. 16. 5. 2012]. Dostupné z: <http://cs.wikipedia.org/wiki/Access_control_list>.
- [11] web:oracle. ORACLE — Oracle® XML DB Developer's Guide, 11g Release 1 (11.1), Part Number B28369-04.
http://docs.oracle.com/cd/B28359_01/appdev.111/b28369/xdb21sec.htm, stav z 15. 5. 2012.
- [12] web:phpGACL. phpGACL — Generic Access Control List with PHP.
phpgac1.sourceforge.net/manual.pdf, stav z 15. 5. 2012.
- [13] Wolfgang Meier. *eXist-db web* [online]. 2012. [cit. 16. 5. 2012]. Dostupné z: <<http://exist-db.org>>.

Příloha A

Pokyny a návody k formátování textu práce

Tato příloha samozřejmě nebude součástí vaší práce. Slouží pouze jako příklad formátování textu.

Používat se dají všechny příkazy systému L^AT_EX. Existuje velké množství volně přístupné dokumentace, tutoriálů, příruček a dalších materiálů v elektronické podobě. Výchozím bodem, kromě Googlu, může být stránka CSTUG (Czech Tech Users Group) [?]. Tam najdete odkazy na další materiály. Většinou dostačující a přehledně organizovanou elektronikou dokumentaci najdete například na [?] nebo [?].

Existují i různé nadstavby nad systémy T_EX a L^AT_EX, které výrazně usnadní psaní textu zejména začátečníkům. Velmi rozšířený v Linuxovém prostředí je systém Kile.

A.1 Vkládání obrázků

Obrázky se umísťují do plovoucího prostředí **figure**. Každý obrázek by měl obsahovat **název** (`\caption`) a **návěští** (`\label`). Použití příkazu pro vložení obrázku `\includegraphics` je podmíněno aktivací (načtením) balíku `graphicx` příkazem `\usepackage{graphicx}`.

Budete-li zdrojový text zpracovávat pomocí programu `pdflatex`, očekávají se obrázky s příponou `*.pdf`¹, použijete-li k formátování `latex`, očekávají se obrázky s příponou `*.eps`.²

Příklad vložení obrázku:

```
\begin{figure}[h]
\begin{center}
\includegraphics[width=5cm]{figures/LogoCVUT}
\caption{Popiska obrazku}
\label{fig:logo}
```

¹pdflatex umí také formáty PNG a JPG.

²Vzájemnou konverzi mezi snad všemi typy obrázku včetně změn velikostí a dalších vymožeností vám může zajistit balík ImageMagick (<http://www.imagemagick.org/script/index.php>). Je dostupný pod Linuxem, Mac OS i MS Windows. Důležité jsou zejména příkazy `convert` a `identify`.



Obrázek A.1: Popiska obrázku

DTD	construction	elimination
	in1 A B a:sum A B in1 A B b:sum A B	case([_:A]a)([_:B]a)ab:A case([_:A]b)([_:B]b)ba:B
+	do_reg:A -> reg A	undo_reg:reg A -> A
*, ?	the same like and + with empty_el:empty	the same like and + with empty_el:empty
R(a,b)	make_R:A->B->R	a: R -> A b: R -> B

Tabulka A.1: Ukázka tabulky

```
\end{center}
\end{figure}
```

A.2 Kreslení obrázků

Zřejmě každý z vás má nějaký oblíbený nástroj pro tvorbu obrázků. Jde jen o to, abyste dokázali obrázek uložit v požadovaném formátu nebo jej do něj konvertovat (viz předchozí kapitola). Je zřejmě vhodné kreslit obrázky vektorově. Celkem oblíbený, na ovládání celkem jednoduchý a přitom dostatečně mocný je například program Inkscape.

Zde stojí za to upozornit na kreslicí programe Ipe [?], který dokáže do obrázku vkládat komentáře přímo v latexovském formátu (vzroce, stejné fonty atd.). Podobné věci umí na Linuxové platformě nástroj Xfig.

Za pozornost ještě stojí schopnost editoru Ipe importovat obrázek (jpg nebo bitmap) a krelit do něj latexovské popisky a komentáře. Výsledek pak umí exportovat přímo do pdf.

A.3 Tabulky

Existuje více způsobů, jak sázet tabulky. Například je možno použít prostředí `table`, které je velmi podobné prostředí `figure`.

Zdrojový text tabulky 2.1 vypadá takto:

```

\begin{table}
\begin{center}
\begin{tabular}{|c|l|l|l|}
\hline
\textbf{DTD} & \textbf{construction} & \textbf{elimination} & \\
\hline
 $\mid$  &  $\verb+in1|A|B$  a:sum A B+ &  $\verb+case([_:A]a)([_:B]a)ab:A+\backslash\backslash$  &  $\verb+in1|A|B$  b:sum A B+ &  $\verb+case([_:A]b)([_:B]b)ba:B+\backslash\backslash$  \\
\hline
 $\&\&$  &  $\verb+do\_reg:A \rightarrow reg A+\&\verb+undo\_reg:reg A \rightarrow A+\backslash\backslash$  & & \\
\hline
 $\ast,?\$$  & the same like  $\mid$  and  $\&\&$  & the same like  $\mid$  and  $\&\&\backslash\backslash$  & & with  $\verb+empty\_el:empty+$  & with  $\verb+empty\_el:empty+\backslash\backslash$  \\
\hline
 $R(a,b)$  &  $\verb+make\_R:A\rightarrow B\rightarrow R+$  &  $\verb+a: R \rightarrow A+\backslash\backslash$  & &  $\verb+b: R \rightarrow B+\backslash\backslash$  \\
\hline
\end{tabular}
\end{center}
\caption{Ukázka tabulky}
\label{tab:tab100}
\end{table}
\begin{table}

```

A.4 Odkazy v textu

A.4.1 Odkazy na literaturu

Jsou realizovány příkazem `\cite{odkaz}`.

Seznam literatury je dobré zapsat do samostatného souboru a ten pak zpracovat programem bibtex (viz soubor `reference.bib`). Zdrojový soubor pro bibtex vypadá například takto:

```

@Article{Chen01,
  author   = "Yong-Sheng Chen and Yi-Ping Hung and Chiou-Shann Fuh",
  title    = "Fast Block Matching Algorithm Based on
              the Winner-Update Strategy",
  journal  = "IEEE Transactions On Image Processing",
  pages    = "1212--1222",
  volume   = 10,
  number   = 8,
  year     = 2001,
}

@Misc{latexdocweb,

```

```

author = "",
title = "{\LaTeX} --- online manuál",
note = "\verb|http://www.cstug.cz/latex/lm/frames.html|",
year = "",
}
...

```

Pozor: Sazba názvů odkazů je dána BibTeX stylem (`\bibliographystyle{abbrv}`). BibTeX tedy obvykle vysází velké pouze počáteční písmeno z názvu zdroje, ostatní písmena zůstanou malá bez ohledu na to, jak je napíšete. Přesněji řečeno, styl může zvolit pro každý typ publikace jiné konverze. Pro časopisecké články třeba výše uvedené, jiné pro monografie (u nich často bývá naopak velikost písmen zachována).

Pokud chcete BibTeXu napovědět, která písmena nechat bez konverzí (viz `title = "{\LaTeX} --- online manuál"` v předchozím příkladu), je nutné příslušné písmeno (zde celé makro) uzavřít do složených závorek. Pro přehlednost je proto vhodné celé parametry uzavírat do uvozovek (`author = "..."`), nikoliv do složených závorek.

Odkazy na literaturu ve zdrojovém textu se pak zapisují:

```

Podívejte se na \cite{Chen01},
další detaily najdete na \cite{latexdocweb}

```

Vazbu mezi soubory `*.tex` a `*.bib` zajistíte příkazem `\bibliography{}` v souboru `*.tex`. V našem případě tedy zdrojový dokument `thesis.tex` obsahuje příkaz `\bibliography{reference}`.

Zpracování zdrojového textu s odkazy se provede postupným voláním programů `pdflatex <soubor>` (případně `latex <soubor>`), `bibtex <soubor>` a opět `pdflatex <soubor>`.³

Níže uvedený příklad je převzat z dříve existujících pokynů studentům, kteří dělají svou diplomovou nebo bakalářskou práci v Grafické skupině.⁴ Zde se praví:

```

...
j) Seznam literatury a dalších použitých pramenů, odkazy na WWW stránky, ...
Pozor na to, že na veškeré uvedené prameny se musíte v textu práce
odkazovat -- [1].
Pramen, na který neodkazujete, vypadá, že jste ho vlastně nepotřebovali
a je uveden jen do počtu. Příklad citace knihy [1], článku v časopise [2],
statí ve sborníku [3] a html odkazu [4]:
[1] J. Žára, B. Beneš;, and P. Felkel.
    Moderní počítačová grafika. Computer Press s.r.o, Brno, 1 edition, 1998.
    (in Czech).

```

³První volání `pdflatex` vytvoří soubor s koncovkou `*.aux`, který je vstupem pro program `bibtex`, pak je potřeba znovu zavolat program `pdflatex` (`latex`), který tentokrát zpracuje soubory s příponami `.aux` a `.tex`. Informaci o případných nevyřešených odkazech (cross-reference) vidíte přímo při zpracovávání zdrojového souboru příkazem `pdflatex`. Program `pdflatex` (`latex`) lze volat vícekrát, pokud stále vidíte nevyřešené závislosti.

⁴Několikrát jsem byl upozorněn, že web s těmito pokyny byl zrušen, proto jej zde přímo necituji. Nicméně příklad sám o sobě dokumentuje obecně přijímaný konsensus ohledně citací v bakalářských a diplomových pracích na KP.

- [2] P. Slavík. Grammars and Rewriting Systems as Models for Graphical User Interfaces. *Cognitive Systems*, 4(4--3):381--399, 1997.
- [3] M. Haindl, Š. Kment, and P. Slavík. Virtual Information Systems. In *WSCG'2000 -- Short communication papers*, pages 22--27, Pilsen, 2000. University of West Bohemia.
- [4] Knihovna grafické skupiny katedry počítačů:
<http://www.cgg.cvut.cz/Bib/library/>

... abychom výše citované odkazy skutečně našli v (automaticky generovaném) seznamu literatury tohoto textu, musíme je nyní alespoň jednou citovat: Kniha [?], článek v časopisu [?], příspěvek na konferenci [?], [www odkaz \[? \]](#).

Ještě přidáme další ukázkou citací online zdrojů podle české normy. Odkaz na wiki o frameworkích [?] a ORM [?]. Použití viz soubor `reference.bib`. V seznamu literatury by nyní měly být živé odkazy na zdroje. V `reference.bib` je zcela nový typ publikace. Detaily dohledal a dodal Petr Dlouhý v dubnu 2010. Podrobnosti najdete ve zdrojovém souboru tohoto textu v komentáři u příkazu `\thebibliography`.

A.4.2 Odkazy na obrázky, tabulky a kapitoly

- Označení místa v textu, na které chcete později čtenáře práce odkázat, se provede příkazem `\label{navesti}`. Lze použít v prostředích `figure` a `table`, ale též za názvem kapitoly nebo podkapitoly.
- Na návěští se odkážeme příkazem `\ref{navesti}` nebo `\pageref{navesti}`.

A.5 Rovnice, centrováná, číslovaná matematika

Jednoduchý matematický výraz zapsaný přímo do textu se vysází pomocí prostředí `math`, resp. zkrácený zápis pomocí uzavření textu rovnice mezi znaky `$`.

Kód `$ S = \pi * r^2 $` bude vysázen takto: $S = \pi * r^2$.

Pokud chcete nečíslované rovnice, ale umístěné centrováně na samostatné řádky, pak lze použít prostředí `displaymath`, resp. zkrácený zápis pomocí uzavření textu rovnice mezi znaky `$$`. Zdrojový kód: `$$$ S = \pi * r^2 $$$` bude pak vysázen takto:

$$S = \pi * r^2$$

Chcete-li mít rovnice číslované, je třeba použít prostředí `equation`. Kód:

```
\begin{equation}
  S = \pi * r^2
\end{equation}
```

```
\begin{equation}
  V = \pi * r^3
\end{equation}
```

je potom vysázen takto:

$$S = \pi * r^2 \quad (\text{A.1})$$

$$V = \pi * r^3 \quad (\text{A.2})$$

A.6 Kódy programu

Chceme-li vysázet například část zdrojového kódu programu (bez formátování), hodí se prostředí *verbatim*:

```
(* nickname2 *)
Lego> Refine in1
      (do_reg (nickname1 h));
Refine by in1 (do_reg (nickname1 h))
  ?4 : pcddata
  ?5 : pcddata
      (* surname2 *)
Lego> Refine surname1 h;
Refine by surname1 h
  ?5 : pcddata
      (* email2 *)
Lego> Refine undo_reg (email1 h);
Refine by undo_reg (email1 h)
*** QED ***
```

A.7 Další poznámky

A.7.1 České uvozovky

V souboru `k336_thesis_macros.tex` je příkaz `\uv{}` pro sázení českých uvozovek. „Text uzavřený do českých uvozovek.“

Příloha B

Ukázky zdrojového kódu

B.1 Výpis veřejných metod Ruby-ACL

V této sekci se nachází výpis veřejných metod. Všechny metody i s parametry a návratovými hodnotami je možné najít v RDOC dokumentaci na CD.

- new
- add_membership_principal
- add_membership_privilege
- change_of_res_ob_address
- change_of_res_ob_owner
- change_res_ob_type
- check
- create_ace
- create_group
- create_principal
- create_privilege
- create_resource_object
- del_membership_principal
- del_membership_privilege
- delete_ace
- delete_principal
- delete_privilege

- delete_res_object
- delete_res_object_by_id
- rename
- rename_principal
- rename_privilege
- save
- show_permissions_fo

Příloha C

Seznam použitých zkratk

ACL Access Control List

ACE Access Control Entry

XML Extensible Markup Language

RDOC RubyDOCumentation

ACO Access Control Objects

ARO Access Request Objects

API Application Programming Interface

phpGACL Generic Access Control Lists with PHP

DTD DTD

XML-RPC XML Remote Procedure Call protokol

xQuery Dotazovací jazyk nad XML dokumenty

XPath XML Path Language

W3C World Wide Web Consortium

Příloha D

UML diagramy

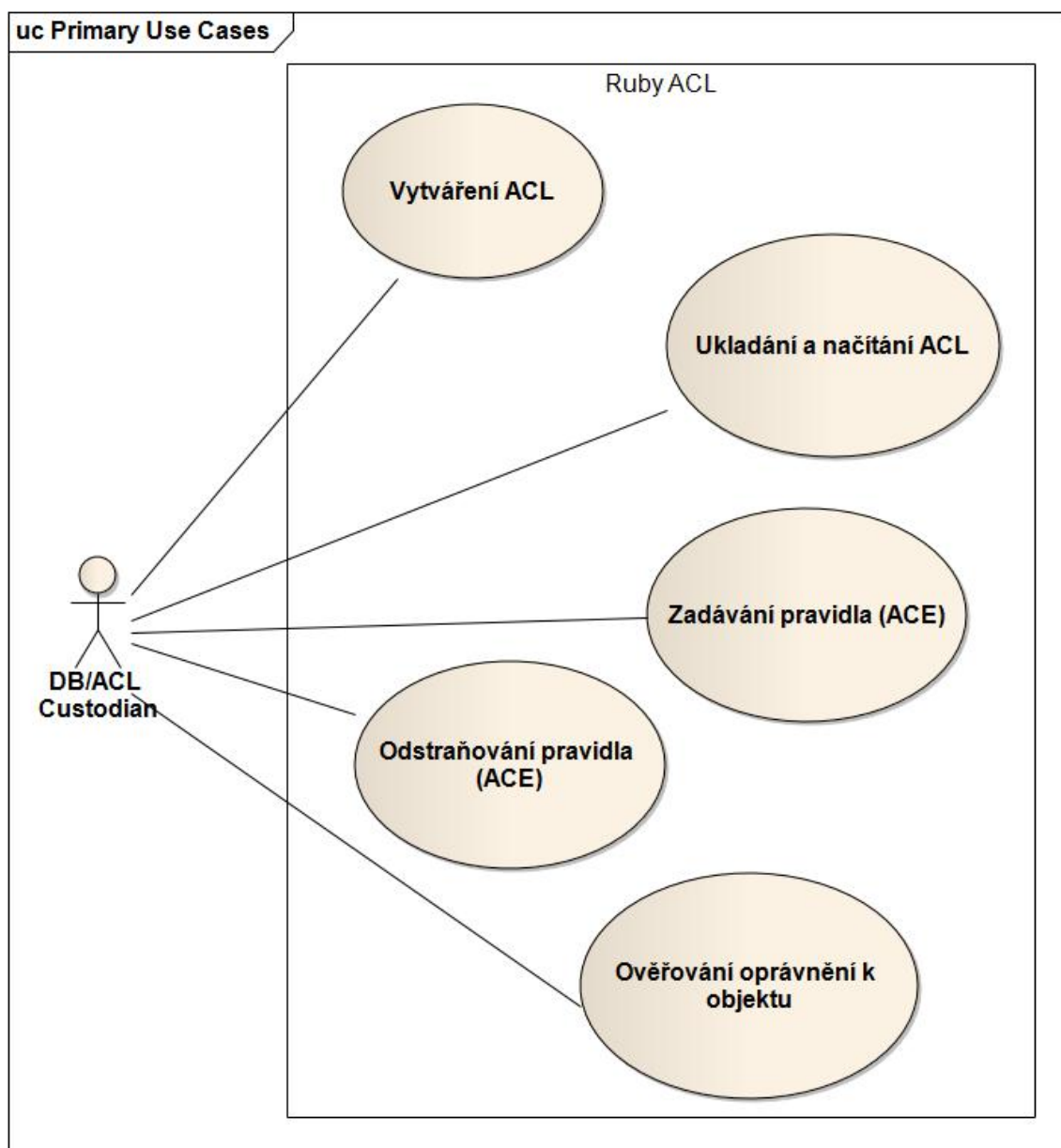
Vytvořené diagramy jsem tvořil podle knihy Destilované UML [\[3\]](#)

D.1 Use Case Diagram

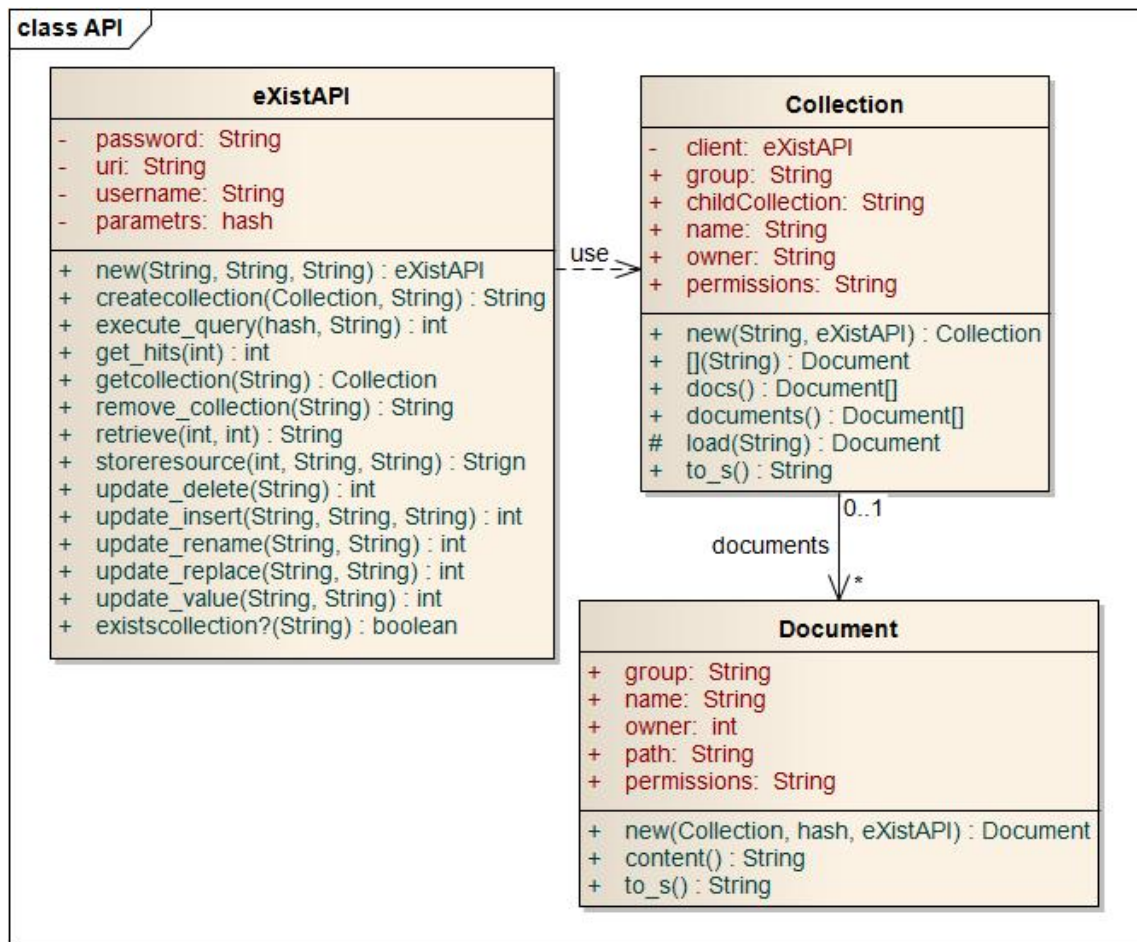
Znázorňuje roli uživatele vůči knihovně. Ruby ACL definuje jednoho aktéra, kterým je uživatel/administrátor ACL viz obrázek [D.1](#). Jedná se vlastně o vývojáře DB Aplikace. Všechny případy užití předpokládají vytvořenou instanci RubyACL, která má vytvořená nějaká pravidla.

D.2 eXistAPI

Obrázek popisuje třídní model rozraní eXistAPI.



Obrázek D.1: UseCases



Obrázek D.2: Diagram tříd eXistAPI

Příloha E

Instalační a uživatelská příručka

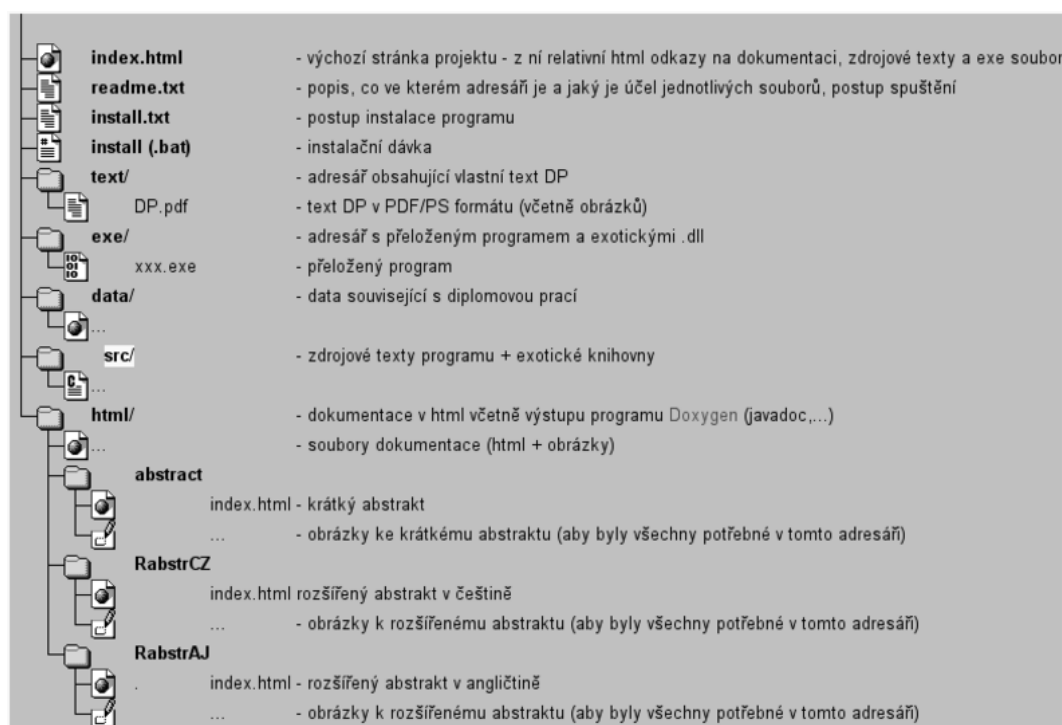
Tato příloha velmi žádoucí zejména u softwarových implementačních prací.

Příloha F

Obsah přiloženého CD

Tato příloha je povinná pro každou práci. Každá práce musí totiž obsahovat přiložené CD. Viz dále.

Může vypadat například takto. Váš seznam samozřejmě bude odpovídat typu vaší práce. (viz [?]):



Obrázek F.1: Seznam přiloženého CD — příklad

Na GNU/Linuxu si strukturu přiloženého CD můžete snadno vyrobit příkazem:

```
$ tree . >tree.txt
```

Ve vzniklém souboru pak stačí pouze doplnit komentáře.

Z **README.TXT** (případně index.html apod.) musí být rovněž zřejmé, jak programy instalovat, spouštět a jaké požadavky mají tyto programy na hardware.

Adresář **text** musí obsahovat soubor s vlastním textem práce v PDF nebo PS formátu, který bude později použit pro prezentaci diplomové práce na WWW.