České vysoké učení technické v Praze
Fakulta elektrotechnická

katedra počítačů

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

Student: **Ondřej Svoboda**

Studijní program: Softwarové technologie a management
Obor: Softwarové inženýrství

Název tématu: **Refaktoring a transakční modul pro RedXML**

Pokyny pro vypracování:

Záměrem práce je tvorba modulu pro refaktorizaci a transakční zpracování kódu v RedXML. Navažte na projekt RedXML, především pak na diplomové práce [1,2]. Proveďte revizi stávajících modulů. Stávající implementaci upravte tak, aby splňovala architekturu klient-server. Dále projekt rozšiřte o podporu transakčního zpracování. Vše důkladně otestujte pomocí jednotkových testů a zdokumentujte.

Seznam odborné literatury:

[1] Pavel Jíra, Transformace XML do key-value databáze Redis, https://dip.felk.cvut.cz/browse/details.php?f=F8&d=K102&y=2012&a=jirapave&t=dipl

[2] Martin Kostolný, XQuery modul pro dotazování nad databází Redis, https://dip.felk.cvut.cz/browse/details.php?f=F8&d=K102&y=2012&a=kostomar&t=dipl

Vedoucí: Ing. Pavel Strnad, Ph.D.

Platnost zadání: do konce letního semestru 2015/2016

[redacted]

doc. Ing. Filip Železný, Ph.D.
vedoucí katedry

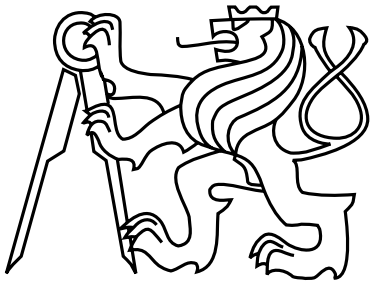
L.S.

[redacted]

prof. Ing. Pavel Ripka, CSc.
děkan

V Praze dne 31. 10. 2014

České vysoké učení technické v Praze
Fakulta elektrotechnická
Katedra počítačů



Bakalářská práce

Refaktoring a transakční modul pro RedXML

Ondřej Svoboda

Vedoucí práce: Ing. Strnad Pavel, Ph.D.

Studijní program: Softwarové technologie a management, Bakalářský

Obor: Softwarové inženýrství

4. ledna 2015

Poděkování

Rád bych poděkoval svému vedoucímu Ing. Pavlu Strnadovi ze jeho aktivní vedení a velmi hodnotné rady během vytváření této práce. Také bych rád poděkoval své rodině, která mi poskytovala zázemí a tím umožnila vytvoření této práce.

Prohlášení

Prohlašuji, že jsem předloženou práci vypracoval samostatně a že jsem uvedl veškeré použité informační zdroje v souladu s Metodickým pokynem o dodržování etických principů při přípravě vysokoškolských závěrečných prací.

V Praze dne 4. 1. 2015 .....

Abstract

This work deals with implementation of a native XML database storing XML documents in a key-value database and implementation of transaction processing. First part is focused on methods of refactorization of original RedXML project and describing individual tools used. Second part focuses on implementation of client-server architecture, processing of XML documents via XQuery and XPath queries and support of transactional processing with taDOM2 protocol.

Abstrakt

Tato práce se zabývá implementací nativní XML databáze ukládající XML dokumenty do key-value databáze a implementací transakčního zpracování. V první části jsou rozebrány metody refaktorizace původního projektu RedXML a nástrojů k tomu použitých. V druhé části je pak rozebrána implementace architektury klient-server, práce s XML dokumenty pomocí XQuery a XPath dotazů a podpory transakčního zpracování protokolem taDOM2.

Obsah

Seznam obrázků

Seznam tabulek

Kapitola 1

Úvod

Projekt RedXML je nativní XML databáze napsaná v jazyce Ruby. RedXML je unikátní tím, že používá key-value databázi pro ukládání dat. Databáze podporuje základní správu dokumentů pomocí kolekcí. Disponuje transakčním zpracováním a umožňuje dotazování pomocí jazyka XQuery. Projekt RedXML tak navazuje na stávající koncept, který je implementován jako knihovna umožňující nahrávání a získávání XML dokumentů díky mapování do key-value databáze Redis. Nad XML dokumenty je možné provádět jednoduché XPath a XQuery dotazy. Takto implementovaná databáze ale nelze dobře použít v produkčním prostředí neboť neumožňuje připojení ke vzdálenému serveru, nebo vytvořit více spojení a pracovat tak s více dokumenty zároveň. Mnoho dnešních webových aplikací pracuje ve více vláknech, nebo procesech a aby bylo možné pracovat efektivně, každý proces/vlákno potřebuje alespoň jedno nebo i více spojení k databázi zároveň. Proto je potřeba stávající implementaci rozšířit tak, aby splňovala architekturu klient-server. Díky této architektuře bude možné aby se více klientů mohlo připojit k jednomu serveru a bylo tedy možné zpracovávat tak více dokumentů zároveň.

Kapitola 2

Refaktorizace

2.1 Co je to refaktorizace?

Refaktorování je disciplinovaný proces provádění změn v softwarovém systému takovým způsobem, že nemají vliv na vnější chování kódu, ale vylepšují jeho vnitřní strukturu s minimálním rizikem vnášení chyb. Při refaktoringu provádíme malé až primitivní změny, ale celkový efekt je velký a to v podobě čistšího, průhlednějšího a čitelnějšího kódu, kód se také lépe udržuje a rozšiřuje. Zlepšuje se také celková kvalita kódu a architektura, snižuje se počet chyb a tím i zvyšuje rychlost vývoje programu. Refaktoring nám pomáhá pochopit a více si ujasnit kód, což je vhodné zejména, pokud upravujeme zdrojový kód po někom jiném.[22, 17]

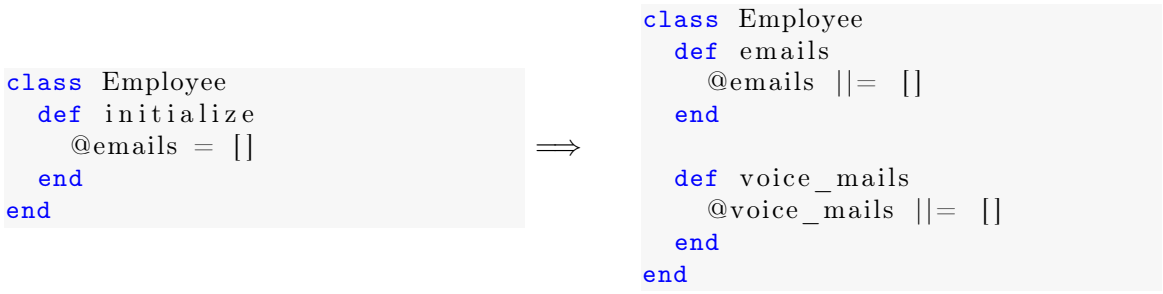
Důvod proč chceme kód refaktorovat je, že chceme zlepšit kvalitu kódu, jeho čitelnost a učinit kód lépe udržovatelným. Pokud chceme do stávajícího kódu přidat novou funkcionalitu, nebo opravit chyby, refaktorizace je dobrým způsobem jak se seznámit se stávajícím kódem a upravit ho do srozumitelnější podoby aby další práce s kódem byla jednodušší. Bez refaktorizace bychom pracovali s kódem, který může mít špatně navržený základ a veškeré další změny budou trpět stejnými neduhy jako stávající kód.

Refaktoring provádíme většinou když potřebujeme nějakým způsobem pracovat s už napsaným kódem, který je buďto nesrozumitelný, špatně navržený, nebo obsahuje chyby. Běžný postup refaktorizace začíná porozuměním kódu (ať už vlastního, nebo cizího) k tomu abychom ho mohli upravovat.

2.1.1 Metody refaktorizace

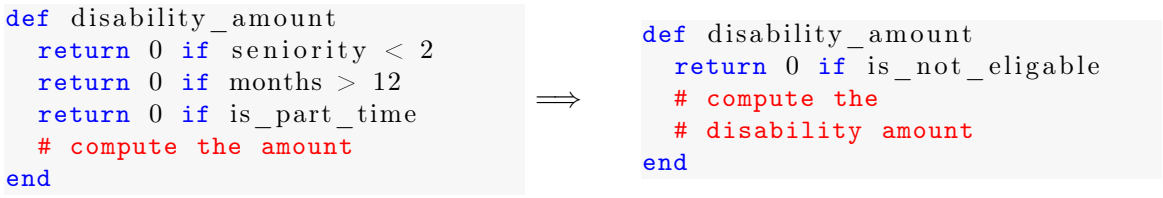
Úprava kódu pak probíhá v malých krocích. Tyto kroky jsou popsány v katalogu Martina Fowlera o refactoringu[16]. Většina kroků je nzaávislá na použitém jazyku, ale pár zde uvedených je specifických pro jazyk Ruby. Několik těchto kroků použitých při práci na RedXML projektu jsou popsány v následujícím textu.

- **Lazy initialized attribute** je jeden z kroků specifických pro jazyk ruby. Ruby má operátor `||=`, který načte hodnotu z proměnné a pokud je hodnota vyhodnocena jako `false`, tak uloží hodnotu na pravé straně operátoru. Kód upravíme tak, aby se argumenty inicializovali až při přístupu k nim.



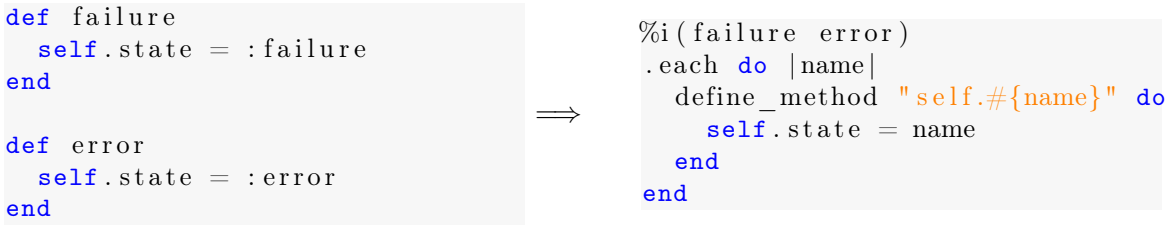
Obrázek 2.1: Lazy initialized attribute

- **Consolidate conditional expression** - Pokud máme v metodě více podmínek se stejným výsledkem, zkombinujeme tyto podmínky do jedné metody. To zlepší čitelnost a srozumitelnost kódu.



Obrázek 2.2: Consolidate Conditional Expression

- **Dynamická definice metod** je vhodná pokud se v kódu opakuje kód metod, které by mohli být jednoduše definovány pomocí define_method. Tím se vyhneme duplikaci kódu. Problémem s dynamickou definicí metod je ten, že jsou definice těchto metod těžko dohledatelné. Běžně se dá vyhledat "`def method_nam`" k nalezení definice, ale při použití dynamických definic se takto vyhledat nedají. Řešením by bylo použít vlastní speciální metodu pro dynamickou definici (například `def_each`), pro jednoduché dohledání takových metod.



Obrázek 2.3: Dynamic function definition

- **Extrakce třídy** - Pokud máme třídu, která realizuje nesouvisející funkcionalitu, oddělíme jednotlivé funkcionality do dvou tříd.

- **Extrakce metody** - Část kódu, který se dá seskupit dohromady, oddělíme do samostatné metody, která svým jménem odpovídá své funkcionalitě.

```
def print_owning
  print_banner

  # print details
  puts "name #{name}"
  puts "amount #amount"
end
```

⟹

```
def print_owning
  print_banner
  print_details
end

def print_details
  puts "name #{name}"
  puts "amount #amount"
end
```

Obrázek 2.4: Extrakce metody

- **Exktrace modulu** - Máme dvě, nebo více tříd, které mají stejný kód, vytvoříme z nich modul a vložíme do tříd jako mixin.[1]

```
class Bid...
  before_save :cap_account
  def cap_account
    account = buyer.account
  end
end
```

⟹

```
class Bid
  include NumberCap
end

module NumberCap
  def self.included(klass)
    klass.class_eval do
      before_save :cap_account
    end
  end

  def cap_account
    account = buyer.account
  end
end
```

Obrázek 2.5: Extrakce modulu

- **Exrakce obklopující metody** - Dvě nebo více metod, které používají skoro stejný kód, ale liší se pouze uprostřed. Vytvoříme metodu, která přebírá blok a vykoná ho na místě, kde se kód liší.

Tyto kroky jsou nejčastěji použité při refaktorování a samotném vývoji celého projektu. Díky nim, je kód "čistčí", neboť neobsahuje duplicitu kódu a je lépe testovatelný. To se projevuje lepší pokrytím kódu automatickými testy a méně chyb jak od nástroje `Rubocop`, tak `Code Climate`, které jsou popsané níže v kapitole 2.6 o použitých nástrojích.

[1] Mixin je modul, obsahující metody z několika tříd a je vkládán do tříd pomocí `include`

```ruby
def charge(amount)
  conn = Server.connect(...)
  conn.send(amount)
rescue IOError => e
 log "Can't charge"
ensure
  conn.close
end
```

$\Longrightarrow$

```ruby
def charge(amount)
  connect do |conn|
    conn.send(amount)
  end
end

def connect
  conn = Server.connect(...)
  yield(conn)
rescue IOError => e
 log "Can't charge"
ensure
  conn.close
end
```

Obrázek 2.6: Extrakce obklopující metody

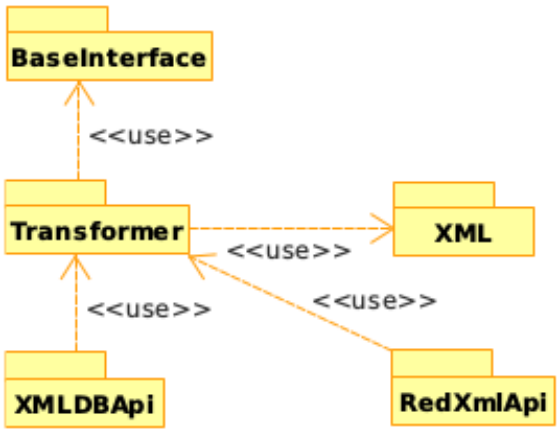
2.2 Koncept RedXML

Koncept RedXML byl projekt, který si kladl za cíl vytvořit základ pro nativní XML databázi, která ukládá XML dokumenty do key-value databáze redis.

2.2.1 Struktura

Původní projekt se skládá ze tří základních vrstev: Databázová, Transformační a Aplikační. Tyto vrstvy jsou obsaženy v pěti modulech znázorněné na obrázku 2.7.

Obrázek 2.7: Diagram balíčků RedXML



Databázová vrstva zajišťuje základní komunikaci s databází a umožňuje vykonávat jednoduché příkazy v podobě Create Read Update Delete (CRUD) operací na úrovni datových struktur Redisu. Poskytuje rozhraní, které podporuje potřebné operace, jež jsou delegovány na Ruby klienta Redis-Rb. Tato vrstva je realizována v podobě menšího modulu

`BaseInterface`. Modul je tvořen dvěma třídami, kde `DbInterface` zajišťuje samotnou komunikaci a `Command` slouží k ukládání volaných operací na databázovém rozhraní pro potřebu jejich pozdějšího přehrání.

Tuto vrstvu bude potřeba rozšířit o další modul, aby bylo možné použít i jiné key-value databáze než Redis a upravit stávající třídu `BaseInterface` aby byla vhodně použitelná pro implementaci transakčního spracování.

Transformační vrstva má na starosti kompletní převod XML dokumentu a jeho částí do formátu vhodném pro uložení pomocí databázové úrovně a naopak. Vrstva je rozdělena na dva menší moduly Transformer a XML. Modul Transformer zajišťuje převod mezi aplikační a databázovou vrstvou a využívá k tomu pomocné struktury modulu XML. Dále jsou zde třídy `KeyBuilder` a `KeyElementBuilder`. Tyto třídy se starají o sestavování klíču pro databázovou vrstvu.

Dále je zde vrstva aplikační. Ta představuje uživatelské rozhraní pro práci s celým systémem. Rozhraní je implementováno jako standard XML:DB API.

2.3 Ruby gem

Mnoho jazyků má vlastní balíčkovací manažery pro jednoduché distribuování programů a knihoven a správu závislostí. Jazyk Ruby není vyjímkou a jeho balíčkovací systém se jmenuje RubyGems, který je hojně využíván v mnoha projektech, proto je použit i v projektu RedXML. RubyGems je tedy balíčkovací manažer pro jazyk Ruby, vytvořen v roce 2003 a nyní je součástí standartní knihovny ruby od verze 1.9. Tento manažer poskytuje standartní formát pro distribuci Ruby programů a knihoven, které jsou zabaleny do tzv. gemů, a nástroje pro jednoduchou instalaci a správu gemů a jejich distribuci pomocí RubyGems serveru.[10]

2.3.1 Struktura gemu

Každý gem obsahuje jméno, verzi a platformu. Například `rake` gem[8] má nyní verzi 10.4.2. Platforma tohoto gemu je `ruby`, což znamená, že funguje na všech platformách na kterých funguje jazyk ruby.

Platforma je složená z architektury CPU, typu operačního systému a někdy verze operačního systému (např. “x86-mingw32”, nebo “java”). Platforma gemu říká, že gem funguje pouze pro konkrétní verzi ruby vytvořenou pro danou platformu.

Každý gem se skládá z jednotlivých komponent:

- Zdrojový kód (včertně testů a přídavných knihoven, nástrojů, . . .)
- Dokumentace
- gemspec

Každý gem musí mít stejnou standartní souborovou strukturu kódu a obsahovat tyto hlavní komponenty:

- Složku `lib` obsahující zdrojové kódy gemu.

- Složku `test` nebo `spec` obsahující testy. Název složky závisí na použitém testovacím frameworku.
- Gem většinou obsahuje soubor `Rakefile`, který použit programem `rake` k automatizování testů, generování kódu nebo dokumentace a provádení jiných úloh.
- Pokud gem obsahuje spustitelné soubory, jsou umístěny ve složce `bin`, která bude načtená do proměnné `PATH` po té co je gem nainstalován. Tím je umožněno uživateli spouštět tyto soubory.
- Dokumentace většinou bývá v souboru `README` a přímo v kódu. Tato dokumentace se automaticky generuje při instalaci gemu. Většina gemů používá `RDoc` dokumentaci, nebo `YARD`.
- Nakonec každý gem obsahuje soubor `gemspec`, který obsahuje informace o gemu. Nachází se zde informace o autorovo jméně, emailu, dále pak verze gemu, platforma, které soubory gem obsahuje a informace o testech.

```
example_gem/
|-- example_gem.gemspec
|-- Gemfile
|-- lib
|   |-- example_gem
|   |   `-- version.rb
|   `-- example_gem.rb
|-- LICENSE.txt
|-- Rakefile
`-- README.md
```

Obrázek 2.8: Struktura gemu

Struktura většiny gemů vypadá jako na obrázku 2.8. Veškeré kódy jsou ve složce `lib`, která je přidána do proměnné prostředí `$LOAD_PATH` odkud Ruby načítá soubory pro příkaz `require`. Složka lib tedy obsahuje dvě věci: .rb soubor a složku pojmenované stejně jako gem. Soubor `example_gem.rb` vypadá následovně:

```
require "example_gem/version"

module ExampleGem
  # Your code goes here...
end
```

Pokud je ve složce `lib/example_gem` více souborů, jsou v tomto souboru vložené pomocí příkazu `require`. Toto umožní rozložit projekt do mnoha souborů, libovolně pojmenované bez vzájemných kolizí. Například pokud by dva gemy měli stejný soubor "`help.rb`", nebylo by možné určit který z těchto souboru má být načten. Touto strukturou je tedy zabráněno takovýmto kolizím.

Soubor `version.rb` vypadá následovně:

```
module ExampleGem
  VERSION = "0.0.1"
end
```

Tento soubor obsahuje konstantu `ExampleGem::VERSION`, která je použita v gemspec souboru. Je dobré mít definovanou verzi ve zvláštním souboru abychom mohli jednoduše navýšit verzi gemu když vydáváme novou verzi.

RubyGems používají sémantické verzování [19]. Verze se udává jako řetězec tří nezáporných čísel oddělených tečkou. Jednotlivé čísla udávají "major" verzi, "minor" verzi a "build number". Každé číslo se mění podle toho, jaké změny proběhli v kódu.

První "build number" se navýší pokud se provedou nevýznamné změny, popřípadě opravy programátorských chyb. Druhá "minor" verze se navyšuje pokud se přidává nová funkcionalita, ale neovlivňuje chování aplikací, které knihovnu používají. Třetí "major" verze se navyšuje když se rozhraní knihovny mění tak, že to ovlivňuje chování existujících aplikací a není zpětně kompatibilní s nižší verzí. Při navýšení major verze se nuluje minor a build number.

Složka `test` nebo `spec` obsahuje soubory testů. Testování je popsáno v kapitole 2.5 Testování.

Složka `bin` obsahuje spoustitelné soubory. Většinou jsou to ruby scripty, ale mohou zde být i binární soubory a různé podpůrné scripty. Typický ruby script v `bin` složce začíná "shebang" řádkem, říkající, že se soubor má spustit jako ruby script:

```
#!/usr/bin/env ruby

begin
  require 'example_gem'
rescue LoadError
  require 'rubygems'
  require 'example_gem'
end

# more code goes here
```

Tento idiom je často použit v ruzných gemech. Nejprve se pokusíme načíst náš gem bez použití rubygems, neboť uživatel může používat jiný manažer gemů než rubygems a tudíž nechceme uživatele nutit aby používal právě tento manažer. Rescue sekce zajistí, že pokud náš gem nebyl nalezen v `$LOAD_PATH`, pokusíme se rubygems použít místo aby náš script skončil s chybou.

2.3.2 gemspec

Gemspec je speciální ruby soubor, který obsahuje informace o gemu jako je jméno gemu, jméno autora, popis, licence a další závislosti na jiné gemy. Dále také obsahuje informace o cestách, která specifikuje jaké soubory budou zahrnuty při balení gemu. Zároveň je zde cesta, která se vloží do load path při prvním načtení gemu, takže nejsou potřeba žádné absolutní cesty pro načtení (příkaz `require`) jakýchkoli souborů z gemu.

První skupinou příkazů je specifikace jména gemu, verze, autora, apod. Další skupina (`files`, `executables`, `test_files` a `require_paths`) je generována automaticky. Většinou

```
Gem::Specification.new do |spec|
  spec.name          = "example_gem"
  spec.version       = ExampleGem::VERSION
  spec.authors       = ["Ondřej Svoboda"]
  spec.email         = ["svoboo18@fel.cvut.cz"]
  spec.summary       = %q{TODO: Write a short summary. Required.}
  spec.description   = %q{TODO: Write a longer description. Optional.}
  spec.license       = "MIT"

  spec.files         = `git ls-files -z`.split("\x0")
  spec.executables   = spec.files.grep(%r{^bin/}) {|f| File.basename(f)}
  spec.test_files    = spec.files.grep(%r{^(test|spec|features)/})
  spec.require_paths = ["lib"]

  spec.add_development_dependency "bundler", "~> 1.6"
  spec.add_development_dependency "rake"
end
```

Obrázek 2.9: Ukázkový gemspec soubor example_gem.gemspec

není potřeba tyto řádky nijak upravovat, pouze v případě, že máme například více složek, které obsahuje spustitelné soubory, potom musíme upravit příslušné řádky aby náš kód fungoval správně anebo správně zabaleny.

Poslední skupinou jsou definice závislostí na jiné gemy (`add_development_dependency`). Zde jsou závislosti na další gemy, které jsou použité v kódu, nebo použité různé nástroje. Příkladem jsou `bundler` a `rake` gemy, které jsou zde vloženy automaticky. Je možné definovat dvě různé závislosti. První je vývojová závislost (`add_development_dependency`), která je použita pro vývoj, nebo testování gemu. Další možností je definovat závislost jako "runtime", pomocí `add_runtime_dependency`. Ta je použita vždy a je automaticky instalována pokud je gem použit v aplikaci třetí strany, kdežto vývojové závislosti jsou ignorovány.

2.3.3 Bundler

Bundler je program pro správu gemů a jejich závislostí pro Ruby projekty. Díky bundleru můžeme říct, které gemy projekt potřebuje a v jakých verzí. Dále se stará o instalaci a update jednotlivých gemů.

Díky bundleru není potřeba jednotlivé gemy instalovat ručně pomocí příkazu `gem`, stačí tedy jeden příkaz `bundle install`. Tímto se nemusíme starat o jednotlivé gemy sami, ale bundler zajistí vše za nás.

Jednotlivé gemy se definují ve speciálním souboru `Gemfile`. Zde jsou definovány názvy gemů a jejich verzí, popřípadě i zdroj (např. adresa git repozitáře, nebo složka).

Od verze 1.0 přidává Bundler příkaz `bundle gem`, který vytvoří kostru gemu. Tímto jsou vytvořeny všechny části projektu RedXML.

```
source 'https://rubygems.org'
gem 'nokogiri'
gem 'rack', '~>1.1'
gem 'rspec', :require => 'spec'
```

Obrázek 2.10: Ukázkový Gemfile

2.3.4 Automatická kompilace C rozšíření

V projektu RedXML je použitá C knihovna `SWIG` pro parsování XQuery a XPath dotazů. Tato knihovna se pro použití musí nejprve zkompilovat pro danou platformu. Tento proces se skládá z vytvoření souboru `extconf.rb`, který je použit pro generování `Makefile` pro C kompilátor a samotné kompilování. Tento proces se díky gemu `rake-compiler` dá zautomatizovat.

Zdrojové kódy knihovny jsou umístěné ve složce `ext` včetně souboru `extconf.rb`, který obsahuje informaci potřebné ke kompilování knihovny. Jelikož je knihovna SWIG velmi jednoduchá z pohledu kompilace, obsahuje soubor extconf.rb pouze základní příkaz pro vytvoření Makefile souboru.

```
require 'mkmf'

create_makefile('Parsers')
```

Obrázek 2.11: Obsah souboru extconf.rb

Gem `rake-compile` předpokládá standartní stromovou strukturu pro práci se zdrojovými kódy tak jak je ukázáno na obrázku 2.12.

```
|-- ext
|   `-- parser
|       |-- extconf.rb
|       |-- parser.h
|       |-- parser.i
|       `-- parser_wrap.cxx
|-- lib
|   `-- redxml
`-- Rakefile
```

Obrázek 2.12: Základní stromová struktura externí knihovny v gemu

Dále je potřeba upravit soubor `Rakefile`, který slouží pro definování různch pomocných úkolů. Přidáním těchto pár řádků, definujeme nový úkol pro zkompilování SWIG knihovny:

Výsledkem je příkaz `rake compile`, který se postará o celý proces kompilování knihovny.

```
require 'bundler/gem_tasks'
require 'rake/extensiontask'

Rake::ExtensionTask.new('Parsers') do |ext|
  ext.ext_dir = 'ext/parser'
  ext.lib_dir = 'lib/redxml/server/xquery'
end
```

Obrázek 2.13: Rake task pro kompilaci C knihovny

2.4 Ruby style guide

Styl zápisu kódu je soubor pravidel, které se používají k psaní zdrojového kódu. Tyto pravidla umožňují programátorům snažší orientaci v kódu a jeho pochopení. Také zlepšují přenositelnost kódu z osoby na osobu, resp. z programátora na programátora. Zároveň pomáhají vyvarovat se různým chybám při jeho tvorbě.

Každý programátor má vlastní styl psaní kódu. Pokud používá nějaké vývojové prostředí (IDE, integrated development environment), např. Visual Studio, píše kód podle stylu vynuceného editorem. Pokud používá jednoduchý textový editor, vytvoří si vlastní styl, podle toho jak se mu kód nejlépe čte. Tento styl se mění s použitým jazykem. Rozhodnutí, která fungují pro JavaScript nemusí fungovat v CSS. Například v JavaScriptu bychom měli používat dvojité uvozovky, zatímco v CSS uvozovky jednoduché.

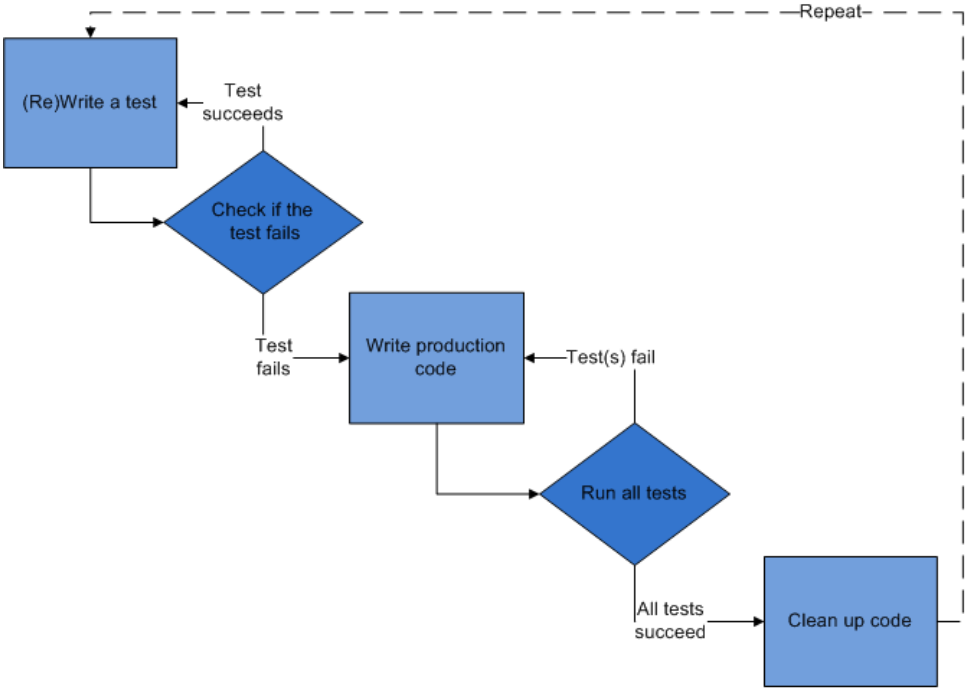
Styl psaní kódu se skládá z několika malých rozhodnutí založených na konkrétním jazyce:

- jak a kdy používat komentáře,
- odsazení pomocí tabulátorů, nebo mezer,
- správné použití bílých znaků,
- pojmenování tříd, proměnných a funkcí,
- seskupování kódu a jeho organizace,
- použité vzory,
- a vzory kterým se vyhnout.

Aby se styl v projektu sjednotil a zlepšila se ním i čitelnost a porozumitelnost kódu, zavádí se různé “style guide” pro jednotlivé jazyky. Například Google má style guide pro většinu jazyků, které používá[4]. Python má oficiální style guide definovanou v PEP-8 [6].

Bohužel ruby žádný oficiální style guide nemá, proto byl vytvořen ruby style guide[11]. Tento style guide byl vytvořen Bozhidarem Batsovem na základě jeho zkušeností, návrhů od Ruby komunity a knížek Programming Ruby 1.9[21] a The Ruby Programming Language[15].

Tento Ruby style guide se stal po čase jakýmsi standardem pro styl psaní kódu v Ruby. Velkou část komunity tvoří lidé kolem web frameworku Ruby on Rails, kteří udávají směr různých konvencí a pravidel pro psaní Ruby kódu. Samotný style guide tak začali používat známe firmy jako GitHub[3], nebo thoughtbot[12].



Obrázek 2.14: Test driven development

2.5 Testování

Testování je proces, při kterém spouštíme program, nebo aplikaci za účelem hledání chyb. Také může sloužit jako proces ověřování kvality a funkcionality programu, aplikace, nebo celého produktu. Testování je důležité, neboť chyby děláme všichni. Některé chyby nejsou tak kritické, ale některé mohou být nebezpečné, případně nákladné na opravu.[13]

2.5.1 Programování řízené testy

Programování řízené testy, anglicky test-driven development (TDD) je přístup k vývoji softwaru, kdy nejprve vytváříme testy, a až poté samotný kód, který prochází temito testy a refactoringem. Hlavním cílem TDD je specifikace, nýbrž ověřování funkčnosti.[14] Což znamená, že je nutné si nejprve vytvořit návrh a uvědomit si, jakou činnost má daný kód vykonávat.

TDD se skládá z pěti kroků[23]:

1. **Napsat test**
   Jako první vytvoříme test. Tím vlastně napíšeme definici požadavků na danou funkcionalitu. Programátor se tím ujistí, že přesně chápe požadavky na onu funkcionalitu, čímž se při psaní samotného kódu eliminuje odchýlení se od původního záměru a cíle. Testy mohou být psány rozdílně, a to podle use-case diagramu, user-stories, nebo jiných materiálů. Tento rozdíl se může zdát jako nevýznamný, nicméně je podstatnou odlišností.

2. **Spustit testy a ujistit se, že všechny neprojdou**
   Tímto ověříme, že máme správně nastavené prostředí (syntaktické chyby v testu, stuby, ...) a že testy skončí neúspěchem. Toto vylučuje, že nová funkcionalita není dokončena ještě dřív, než by někdo vůbec začal s její implementací. Test by také měl skončit s chybou, kterou očekáváme kvůli neexistující funkcionalitě.

3. **Napsat vlastní kód**
   Dalším krokem je psaní vlastního kódu. V tomto kroku děláme rychlé přírůstky, tak aby nově napsané testy procházely. Cílem není psát co nejefektivnější, nebo nejelegantnější kód, to je předmětem dalších kroků, zde se jedná pouze o jednoduché splnění testů.

4. **Kód testy prochází**
   Kód který jsme napsali v minulém bodě prochází testy, což znamená, že kód plní definované požadavky a nerozbíjí již existující funkcionality. Pokud testy neprochází, kód musí být kód upravován dokud všechny testy neprojdou.

5. **Refaktorizace**
   Rostoucí kód musí být pravidelně upravován do čitelnější a srozumitelnější podoby. Nový kód, který jsme psali pro splnění testů můžeme teď přesunout na vhodnější místo kam se logicky hodí více. Také musíme odstranit duplikaci kódu. Názvy objektů, tříd, modulů, proměnných a metod by měly správně reprezentovat svůj účel a použití. Jak přidáváme novou funkcionalitu, těla metod se mohou být delší a jednotlivé objekty větší. Takto dlouhé metody můžeme rozdělit do kratších a velké objekty rozdělit do několika menších, což zlepší čitelnost a udržitelnost kódu jako celku později v životním cyklu vývoje. Metod jak toho dosáhnout je popsáno v kapitole 2 Refaktorizace.

2.5.2 Programování řízené chováním

Programování řízené chováním, anglicky behaviour driven development (BDD), je metoda založená na TDD, ale narozdíl od TDD klade důraz na chování než na strukturu. Při psaní jednotkových testů stylem TDD se programátor často zaměřuje na testování konkrétních struktur, kde testujeme například, že metoda `register()` objektu `Registrator` vloží hodnotu do proměnné `registers`, a že tato proměnná je pole. Tyto detaily vytváří závislost testu na vnitřní struktuře testovaného objektu. Později, pokud změníme například typ proměnné `registers` z typu Array na typ Hash, test na tento objekt selže i přesto, že jsme chování objektu nikterak nezměnili a objekt funguje stejně. Tato závislost na vnitřních strukturách objektu dělá test špatně udržovatelný a je pravdepodobnější, že test bude ignorován a eventuálně i vyřazen. Problém s testováním interní struktury objektu je, že testujeme co objekt je a ne co objekt dělá. BDD se zaměřuje na chování místo struktury na každé úrovni vývoje.[14]

2.5.3 RSpec

RSpec je BDD framework pro jazyk Ruby, vytvořený v roce 2005 Stevenen Bakerem jako učební pomůcka TDD. RSpec může být považována za doménově specifický jazyk (DSL), který se podobá přirozenému jazyku.[9]

```
RSpec.describe Object do
  it 'do something' do
    expect(subject).to be_a Object
  end
end
```

Obrázek 2.15: Ukázka testu v RSpec

2.6 Nástroje

V této kapitole budou rozebrány nástroje použité při vývoji projektu RedXML. Tyto nástroje pomáhají sledovat stav projektu, kvalitu kódu a automatické testování vůči různým verzím jazyka Ruby.

2.6.1 Rubocop

Rubocop je statický analyzátor kódu pro Ruby založen na ruby style guide. Díky tomuto nástroji je možné odhalit různé chyby jako překlepy, špatné názvy tříd, nebo špatně použitých bílých znaků. V projektu RedXML je rubocop použit pro úpravu a refaktorizaci původních modulů a jejich snadné začlenění.

Rubocop umožňuje komplexní nastavení přes soubor `.rubocop.yml`. Zde můžeme povolit, nebo zakázat jednotlivé kontroly, nebo upravit jejich defaultní hodnoty.

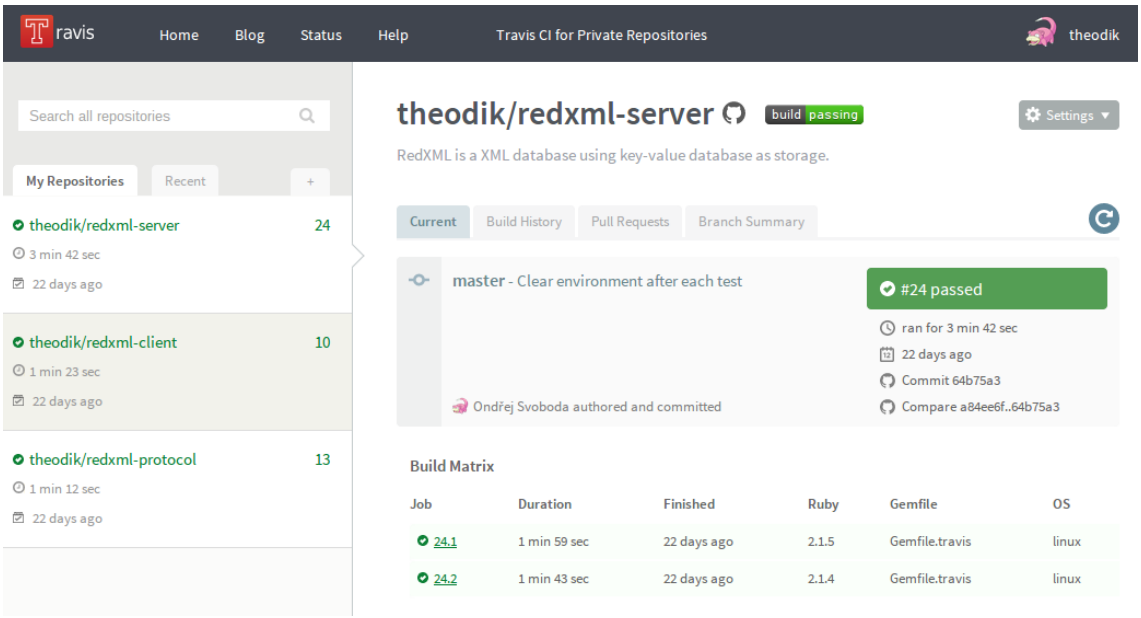
```
$ rubocop
Inspecting 107 files
..................CC.C.................................C.....................

Offenses:

spec/redxml/xquery/xquery_spec.rb:7:27: C: Space between { and | missing.
      results = result.map{|i| i.respond_to?(:to_html) ? i.to_html : i }
                           ^^
spec/redxml/xquery/update_spec.rb:7:27: C: Space between { and | missing.
      results = result.map{|i| i.respond_to?(:to_html) ? i.to_html : i }
                           ^^
spec/redxml/xquery/update_spec.rb:13:27: C: Space between { and | missing.
      results = result.map{|i| i.respond_to?(:to_html) ? i.to_html : i }
                           ^^
spec/redxml/xquery/xpath_spec.rb:7:27: C: Space between { and | missing.
      results = result.map{|i| i.respond_to?(:to_html) ? i.to_html : i }
                           ^^
lib/redxml/server/executors.rb:27:20: C: Space between { and | missing.
          xml.map{|i| i.respond_to?(:to_html) ? i.to_html : i }.join(',')
                   ^^

107 files inspected, 5 offenses detected
```

Obrázek 2.16: Nalezené chyby v RedXML-Server

Před začátkem refaktorování původního kódu z konceptu RedXML, našel `Rubocop` spoustu chyb. Některé byli pouze špatné odsazení, nebo špatně použité závorky, ale některé byli významnější, např. chybějící písmenka v proměnných, které původní testy neodhalili. Detailní popis průběhu refaktorování je popsán v sekci 2.7 Výsledný stav projektu po refaktorizaci.

2.6.2 Travis CI

Travis CI je webová služba umožňující průběžnou integraci projektů hostujících repositáře na GitHubu. Průběžná integrace (anglicky continuous integration) je metoda vývoje softwaru, kde každý vývojář integruje svoji část práce průběžně. Každá integrace je ověřena automatickými testy, k co nejrychlejšímu nalezení chyb.



Obrázek 2.17: TravisCI

Projekt RedXML používá TravisCI pro automatické testování v různých prostředích a verzích ruby. Nastavení se provádí v souboru `.travis.yml`, kde se specifikuje jazyk projektu, různé verze Ruby, v kterých mají testy proběhnout a další kroky potřebné pro běh testů.

```
language: ruby
rvm:
  - 2.1.5
  - 2.1.4
gemfile:
  - Gemfile.travis
services:
  - redis-server
before_script: bundle exec rake compile
```

Obrázek 2.18: Soubor .travis.yml pro nastavení TravisCI

Na obrázku 2.18 je uveden soubor `.travis.yml` pro gem `redxml-server`. Zde bylo potřeba definovat, že projekt používá databázi Redis (část `services`) a že pro běh testů potřebuje nejprve zkompilovat parser knihovnu (část `before_script`).

2.6.3 Code climate

Code climate je webová služba, která poskytuje statickou a automatickou analýzu kódu Ruby on Rails, JavaScript a PHP projektů. Code climate automaticky naklonuje git (github) repozitář a analyzuje každý commit. Výsledkem jsou pak změny, které se odehrály v čase, kvalita kódu a upozornění na bezpečnostní problémy.

Při analyzování kódu se Code Climate zaměřuje na:

- Složitost kódu.
- Duplicita.
- Bezpečnostní zranitelnosti.
- Zobrazuje, které části kódu se nejvíce mění. Kód který je komplexní a je vhodný pro refaktorování.
- Test coverage.

2.6.4 Coveralls

Pokrytí kódu testy je metrika, která určuje jak velká část kódu je pokryta automatickými testy. Pro sledování pokrytí kódu v projektu RedXML je použita webová služba Coveralls. Ta umožňuje sledovat postupný vývoj kódu a jeho pokrytí.

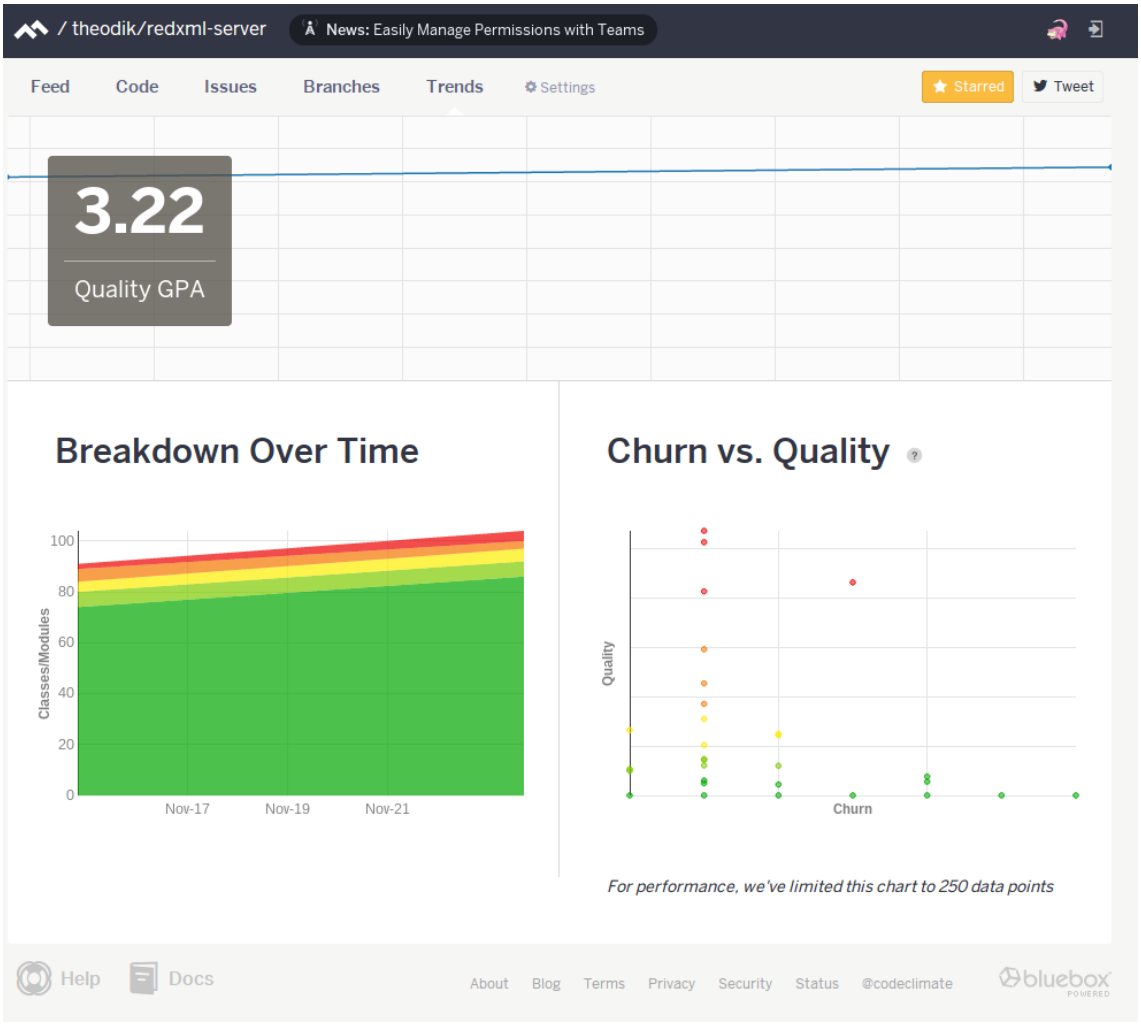
Projekt RedXML aktulně má pokrytí kódu přes 80%. Některé části kódu nejsou otestovány kvůli jejich závislosti na původních modulech, které v současném stavu nejsou použity, ale nelze je odstranit, neboť by to vyžadovalo kompletní přepsání původních modulů, včetně jejich testů. Proto byly zachovány.

2.7 Výsledný stav projektu po refaktorizaci

Cílem refaktorizace bylo vytvořit kód, který je srozumitelný, modulární a jednoduchý na úpravy. K tomu byli použity výše zníměné nástroje.

Aby bylo možné začlenit původní moduly, bylo nejprve nutné kód upravit do podoby, která bude vyhovovat pravidlům z Ruby style guide. Za pomoci nástroje `Rubocop` byl opraven styl kódu, dále pak začlenění kódu do logických modulů a odstranění většiny závislostí na ostatních modulech aby bylo možné je používat nezávisle na použití konkrétních modulech. Příkladem takových změn je odstranění závorek kolem podmínky v příkazu if, nebo změna složených závorek obalující ruby blok s víceřádkovým kódem jak je ukázáno na obrázcích 2.20 a 2.21.

Hlavní změna byla provedena u načítání jednotlivých souborů, kde byla použita funkce `autoload`. Tato funkce umožňuje dynamické načítání souborů až v době, kdy jsou potřebné pro vykonání kódu. Soubor tak už nejsou složitě načítány z jednotlivých souborů, ale definice jsou uvedeny v na jednom místě, což umožňuje lepší organizaci načítaných souborů.



Obrázek 2.19: Code Climate

```
def prepare_results(results)
  final_results = []
  results.each { |result|
    if(result.kind_of?(ExtendedKey))
      final_results << @path_solver.path_processor.get_node(result)
    else
      final_results << result
    end
  }

  return final_results
end
```

Obrázek 2.20: Ukázka kódu před refaktorizací

```
def prepare_results(results)
  results.map do |result|
    if result.kind_of? ExtendedKey
      @path_solver.path_processor.get_node(result)
    else
      result
    end
  end
end
```

Obrázek 2.21: Ukázka kódu po refaktorizaci

```
require_relative "flwor_solver"
require_relative "path_solver"
require_relative "../expression/expression_module"
require_relative "../../transformer/key_builder"
require_relative "../../transformer/key_element_builder"

module XQuery
  class XQuerySolver
  ...
```

Obrázek 2.22: Načítání souborů pomocí relativních cest

V původní podobě byli jednotlivé soubory načítány z relativních cest, to vedlo k problémům při práci s jednotlivými třídami, které byli načtené z těchto souborů. Také je duležité načítat soubory ve správném pořadí.

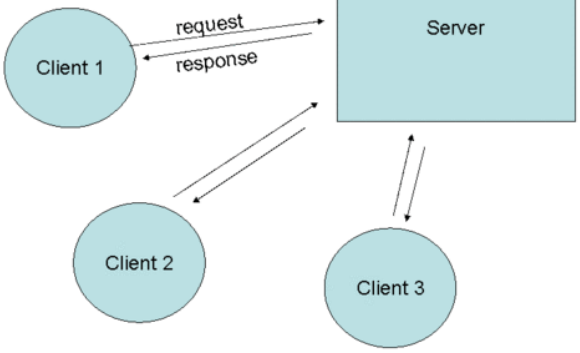
Tyto problémy byly odstraněny funkcí `autoload`, díky které je o načítání souborů stará samo Ruby.

```
module RedXML
  module Server
    module XQuery
      module Solvers
        autoload :Comparison, 'redxml/server/xquery/solvers/comparison'
        autoload :Delete,     'redxml/server/xquery/solvers/delete'
        autoload :FLWOR,      'redxml/server/xquery/solvers/flwor'
        ...
```

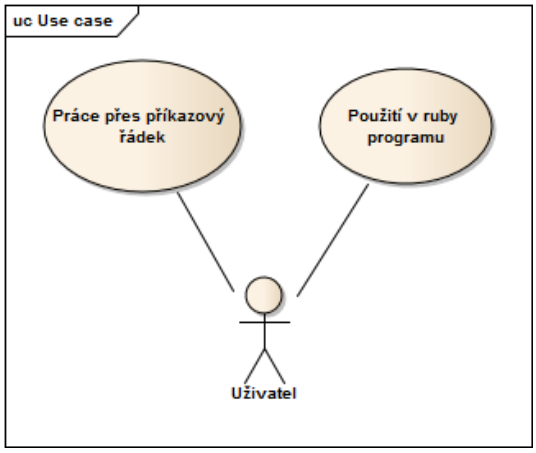
Obrázek 2.23: Použití funkce autoload

Kapitola 3

Architektura



Obrázek 3.1: Architektura klient server



Obrázek 3.2: Use case klienta

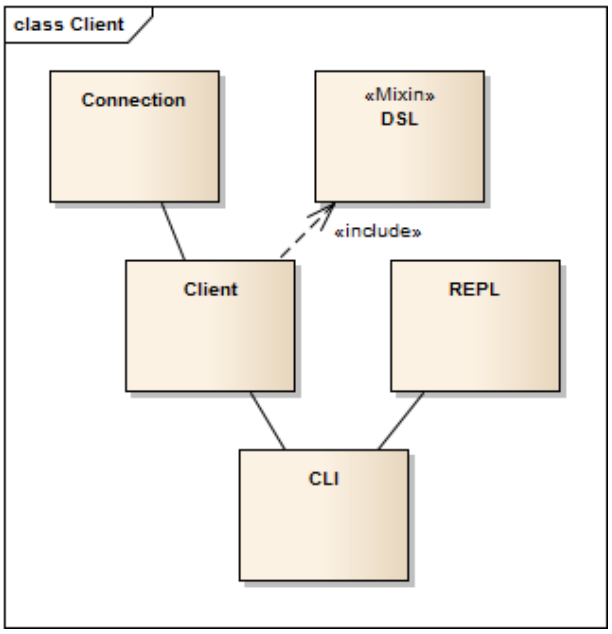
Jelikož je projekt RedXML koncipován jako nativní XML databáze, byla použita architektura klient-server. Architektura klient-server je tzv. dvouvrstvá architektura, ve které jsou dvě části. První z nich je klient. Klient je aktivní část, která posílá žádnosti serveru a očekává odpověď. Je to také část, která komunikuje přímo s uživateli pomocí uživatelského prostředí. Druhou částí je server, který je pasivní, a který naslouchá a reaguje na žádosti připojených klientů. Po přijetí požadavku klienta, server požadavek zpracuje a odešle odpověď.

Projekt RedXML je rozdělen na tři části: server, klient a protokol. Každá část je vytvořena jako samostatný ruby gem.

3.1 Klient

Klientská část databáze RedXML je implementována v gemu `redxml-client`. Tento gem je možné instalovat pomocí příkazu `gem`, nebo vložit do souboru Gemfile a nainstalovat pomocí `bundle install`.

Od klienta očekáváme, že bude jednoduchý na použití v ruby scriptech a zároveň chceme mít možnost ovládat server přes příkazovou řádku.



Obrázek 3.3: Class diagram klienta

Toho je docíleno strukturou kterou vidíme na obrázku 3.3. Základem je třída `Client`, která poskytuje API pro připojení k serveru a práci s XML dokumenty. Tato třída je detailně popsána v sekci 3.1.1 Ruby klient. Dále tento gem poskytuje příkaz `redxml-client`, který nám poskytne interaktivní rozhraní příkazového řádku (CLI) pro práci s RedXML klientem. Tento příkaz je popsán v sekci 3.1.2 CLI klient.

3.1.1 Ruby klient

Abychom mohli klienta používat v ruby projektu, potřebujeme nejprve nainstalovat `redxml-client` gem:

```
$ gem install redxml-client-0.0.1.gem
Successfully installed redxml-client-0.0.1
1 gem installed
```

nebo pokud je použit bundler pro správu gemů, přidáme do `Gemfile` řádek:

```
gem 'redxml-client'
```

Od této chvíle můžeme klienta používat.

Připojení k databázi provedeme vytvořením `RedXML::Client` objektu.

```
require 'redxml/client'

client = RedXML::Client.new
```

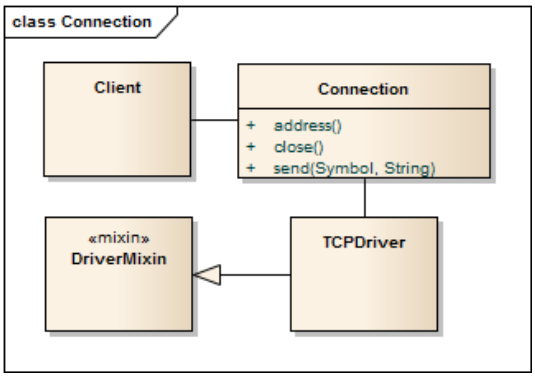
Bez uvedení jakýchkoli parametrů, klient předpokládá, že server byl spuštěn s defaultním nastavením, běží na stejném stroji a poslouchá na portu 33965. Pokud se potřebujeme připojit na jiný, než defaultní port, použijeme parametr `port`:

```
client = RedXML::Client.new(host: '10.0.0.1', port: 12345)
```

Třída `RedXML::Client` také poskytuje metodu `connect`, které předáme blok, v kterém bude náš kód pro práci s klientem. Po skončení bloku metoda connect automaticky uzavře klienta. Metoda přebírá stejné parametry jako metoda `new`:

```
RedXML::Client.connect(host: '10.0.0.1') do |client|
  # kód
end
```

Při vytváření instance třídy `Client`, je automaticky vytvořena třída `Connection`. Tato třída má za úkol vytvořit paket s informacemi pro server, který je následně odeslán pomocí třídy `Driver`. Jak takový paket vypadá je popsáno v kapitole 3.3 Protokol. `Driver` obstarává samotné připojení k serveru. V tuto chvíli je implementován pouze `TCPDriver` pro komunikaci po TCP protokolu a do budoucna je počítáno s implementací unix socketu. Celou strukturu můžeme vidět na obrázku 3.4.



Obrázek 3.4: Struktura tříd Client, Connection, Driver

Klient podporuje tyto příkazy pro práci s databází a XML dokumenty:

server_version Tento příkaz vrací verzi serveru ke kterému je klient právě připojen. Příkaz nepřebírá žádné argumenty.

ping Ping odesílá na server prázdný požadavek díky němuž si tak můžeme ověřit, že jsme připojeni k serveru a server odpovídá.

use Příkaz `use` nastaví aktuálně pužité prostředí a kolekci. Každý XML dokument uložený v RedXML databázi má rodičovskou kolekci a kolekce má rodičovské prostředí. Příkaz přebírá dva argumenty, první je dané prostředí a druhým je kolekce:

```
client.use('env', 'col')
# další příkazy...
```

save_document Tímto uložíme celý XML dokument do databáze. Příkaz přebírá dva argumenty, prvním je řetězec s daným xml dokumentem a druhým je název tohoto dokumentu. Je možné předat pouze jméno souboru s XML dokumentem, v tom případě není nutné předávat název.

```
# jako xml
client.save_document('<xml></xml>', 'test.xml')
# jako soubor
client.save_document('document.xml')
```

Dokument se ukládá pod současně vybraným prostředím a kolekcí. Pokud neexistují automaticky se vytvoří.

load_document Načtení celého XML dokumentu z databáze. Příkaz přebírá pouze argument s názvem dokumentu.

execute Spuštění XQuery a XPath dotazu v daném prostředí a kolekci.

```
client.execute("doc('test.xml')/root")
```

begin Zahájí transakční zpracování. Všechny příkazy po `begin` jsou zpracovány v jedné transakci dokud není transakce potvrzena příkazem `commit`.

commit Ukončí a uloží data z právě prováděné transakce.

close Odpojí se od serveru.

3.1.2 CLI klient

Po instalaci gemu je k dispozici systémový příkaz `redxml-client`, který nám poskytuje interaktivní prostředí pro komunikaci s RedXML serverem. Stejně jako klient Redisu, nebo PostgreSQL, `redxml-client` používá pro vstup knihovnu readline pro editování přikazů uživatele.

```
$ redxml-client --help
redxml-client [options]
  -s, --scheme SCHEME              tcp/unix
  -h, --host HOST                  Host to connect to
  -p, --port INT                   Port to connect
  -u, --unix INT                   Unix socket to connect
  --help                           Show help
```

Obrázek 3.5: Možnosti příkazu redxml-client

Použitá knihovna umožňuje rozšířené úpravy příkazového řádku. Přidává podporu pro klávesové zkratky známé z unixového shellu GNU Bash, nebo lze nakonfigurovat tak, aby používala zkratky z editoru Emacs, nebo Vi. Její hlavní výhodou je podpora historie příkazů. Příkazy zadané uživatelem jsou uloženy a dají se později vyvolat pomocí klávesy šipky nahoru. V historii lze také vyhledávat.

Ctrl-C Zruší právě psaný příkaz a skočí na nový řádek.

Ctrl-D Vloží znak EOF. Toto způsobí ukončení interaktivního režimu a ukončí spojení k serveru. Stejný jako příkaz close.

Ctrl-W Smaže jedno slovo.

Ctrl-U Smaže vše od současné pozice na začátek.

Ctrl-K Smaže vše od současné pozice na konec.

Ctrl-R Vyhledává v historii příkazů.

Obrázek 3.6: Přehled základních klávesových zkratek

3.1.3 Testování

Testování klientské části je víceméně přímočarý proces. Abychom nemuseli pro testování mít spuštěný server, který by testy výrazně zpomalil, vytvoříme mock-up objekty pro komunikaci se serverem. Takto můžeme ověřit, zda naše knihovna funguje podle specifikace popsané v předchozí kapitole.

Testy jsou rozděleny do jednotlivých souborů podle vrstev, kromě třídy `Client`, která je rozdělena do dvou souborů `client_spec` a `dsl_spec`. V prvním souboru ověřujeme, že naše knihovna umožňuje vytvoření instance a připojí se k databázi. Také, že lze použít metoda `connect`, která uzavře spojení po vykonání předaného bloku. Ve druhém souboru je ověřováno, že klient správně zpracovává jednotlivé příkazy pro práci s XML dokumenty a odesílá je ve správném formátu.

Pro testování třídy `Connection` si definujeme speciální testovací třídu `TestDriver`, kterou použijeme místo třídy `TCPDriver`. Tato třída tedy není explicitně otestována, ale protože pouze obaluje objekt `TCPSocket`, který je součástí standartní knihovny Ruby, můžeme test vynechat. Jelikož se zaměřujeme na funkčnost klienta nikoliv na správnost odpovědí od serveru, speciální testovací třída tedy zachytává požadavky a místo aby je posílala na server přes TCP, vytvoří a vrátí testovací mock-up odpovědi.

Třída starající se o zpracování vstupů pro interaktivní příkazovou řádku, není otestována z důvodu použití knihovny readline. Tato knihovna manipuluje se standartním vstupem a výstupem takovým způsobem, že neexistuje jednoduchý způsob jak posílat příkazy a číst odpovědi.

Testy jsou rozděleny do 5 souborů po 26 jednotlivých testovacích scénářů.

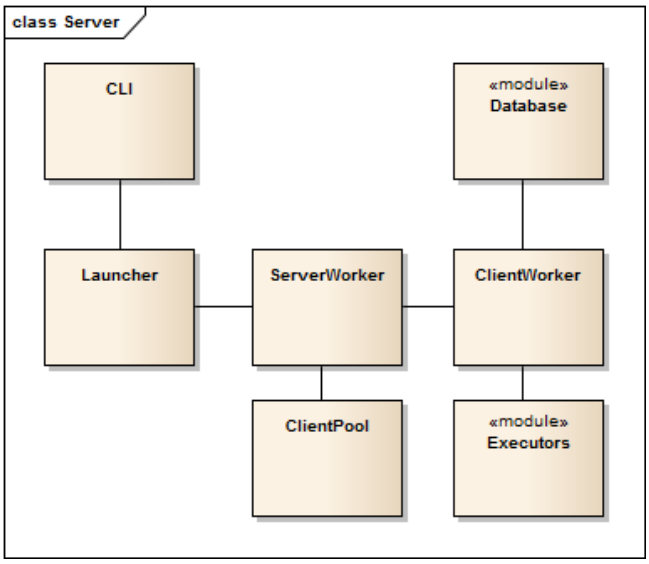
```
Run options: include {:focus=>true}

All examples were filtered out; ignoring {:focus=>true}
...........***.............

Pending:
  RedXML::Client::REPL sends xquery
    # Temporarily skipped with xdescribe
    # ./spec/redxml/repl_spec.rb:20
  RedXML::Client::REPL sends ping
    # Temporarily skipped with xdescribe
    # ./spec/redxml/repl_spec.rb:49
  RedXML::Client::REPL exits and closes connection
    # Temporarily skipped with xdescribe
    # ./spec/redxml/repl_spec.rb:34

Finished in 0.05202 seconds (files took 1.28 seconds to load)
26 examples, 0 failures, 3 pending
```

Obrázek 3.7: Test klienta

3.2 Server

Serverová část databáze RedXML je implementována v gemu `redxml-server`. Stejně jako u klientské části, tento gem poskytuje příkaz `redxml-server`, který spouští RedXML databázi.

3.2.1 Struktura



Obrázek 3.8: Class diagram serveru

RedXML server se spouští příkazem `redxml-server`, což je ruby script, který získá instanci třídy `CLI` a zavolá metodu `run`.

Třída `CLI` je singleton a stará se o parsování argumentů z příkazové řádky, nastavení logování a spuštění samotného serveru. Pro parsování argumentů je použita třída `OptionParser` ze standartní knihovny Ruby. Pomocí OptionParseru je možné jednoduše definovat, které

argumenty program přebírá a napsat k nim jednoduchou dokumentaci, kterou lze pak vytisknout (například při zadání argumentu pro nápovědu, nebo při špatně zadaných argumentech). Dále třída `CLI` inicializuje logger. Logger je objekt starající se o formát a výpis jednotlivých informací o běhu aplikace. Jelikož server pracuje ve více vláknech, chceme aby nám logger vypisoval i informace o vlákně, v kterém se událost odehrála, proto je definována vlastní formátovací třída, která tuto informaci doplní. Po zparsování argumentů a inicializování loggeru je spuštěn vlastní server.

Server se spouští pomocí třídy `Launcher`, která se stará o inicializaci tříd potřebných k běhu serveru. Obsahuje pouze dvě metody: `run`, která spouští hlavní smyčku serveru a `stop`, která server zastaví.

3.2.2 Obsluha klientů

Hlavní smyčka serveru je v třídě `ServerWorker` a vypadá následovně:

```
1 start_server_socket
2 while running?
3   next unless IO.select([socket], nil, nil, 0.5)
4   client = socket.accept
5   client_pool.que(client) do |cli|
6     RedXML::Server::ClientWorker.new(cli).process
7   end
8 end
```

Na řádku 1 vytvoříme serverový socket. V tuto chvíli je implementován pouze protokol TCP, ale do budoucna se počítá s implementací alespoň unix socketu. Pro práci se socketem obsahuje Ruby knihovna třídu `TCPServer`, která reprezentuje TCP/IP server socket. Tento server socket po vytvoření poslouchá defaultně na portu 33965.

Na řádce 2 začíná hlavní smyčka, a poté se čeká na nové připojení klienta. Připojení se přijímá pomocí metody `accept`, ale protože je tato metoda blokující bez možnosti timeoutu, což znamená, že by se server nedal vypnout nastavením proměnné running, je využita funkce select. Funkce select je systémové volání unixu. Tato funkce ověřuje, které ze zadaných socketů jsou připraveny pro příjem dat (resp. obsahují požadavek o spojení), zápis dat (resp. úspěšně navázaly spojení) nebo se nacházejí v chybovém stavu. Také tato funkce umožňuje specifikovat timeout, po kterém funkce vrátí, nehledně na stav zadaných socketů.

Po přijetí nového klienta, zařadíme ho do fronty zpracování pomocí objektu `client_pool`. Tento objekt je zodpovědný za správu klientů a jejich zpracování ve vláknech.

3.2.2.1 Správa vláken

Správa vláken je implementována v třídě `ClientPool`. Jelikož nechceme pro každý nově přijaté spojení klienta vytvářet nové vlákno, což by při větším počtu spojení mohlo způsobit větší zátěž než samotné zpracování klientů, `ClientPool` vytváří pouze omezený počet vláken v kterých se jednotlivé požadavky klientů zpracovávají postupně.

Základem je metoda `que`, která zařadí klienta do fronty zpracování a případně vytvoří nové vlákno. Každé nové vlákno obsahuje smyčku, v které se z fronty vybírá první volný klient pro zpracování. Klient je pak zpracován v bloku předaném metodě `que`. Pokud je fronta prázdná, vlákno je ukončeno.

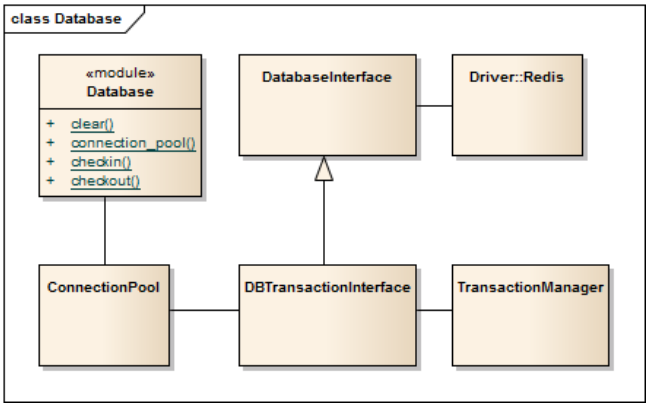
O zpracování klientů se stará třída `ClientWorker`.

3.2.2.2 Zpracování požadavku

Jako první, po přijetí spojeni, se server identifikuje názvem a verzí, kterou pošle klientovi a dále pak čeká na jeho požadavek. Každý požadavek je složen z názvu příkazu a parametru. Příkazy jsou delegovány do "executor" tříd, kde je příkazy zpracují a vrátí výsledek. V současné době je implementováno 6 příkazů, které jsou již popsány v kapitole 3.1.1 Ruby client, a to: ping, execute, save_document, load_document, begin a commit. Detailní popis příkazů a parametrů je podrobně popsán v kapitole 3.3 Protokol.

3.2.3 Databáze

Jako úložiště používá databáze RedXML key-value databázi Redis, do které mapuje XML dokumenty. Zde si popíšeme databázový modul, který se stará o jednotlivá připojení k databázi a transakční zpracování příkazů.



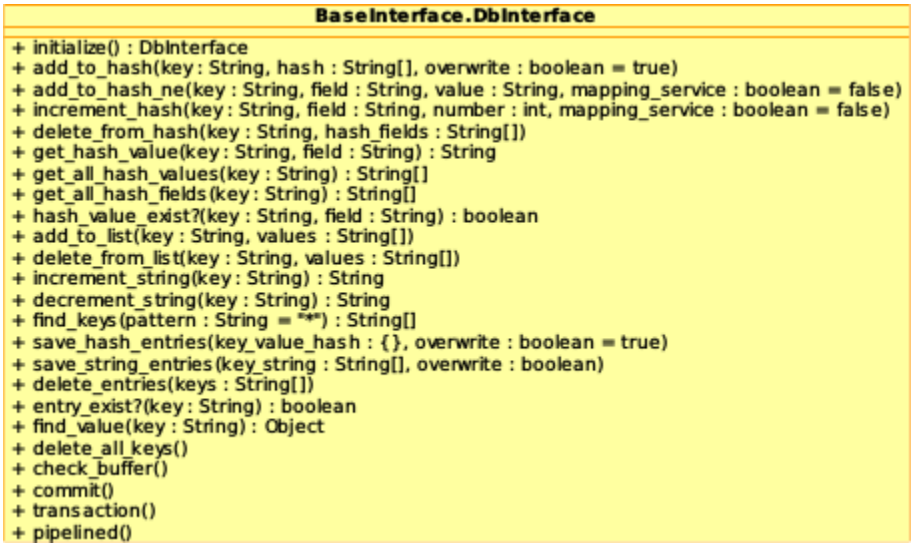
Obrázek 3.9: Class diagram modulu database

Základem pro práci s databází je modul `Database`. Tento modul obsahuje dvě základní metody pro získání spojení z poolu a jeho následné vrácení zpět. Pro správu databázových spojení se používá `ConnectionPool`, který je vytváří a udržuje pro další použití, aby se spojení nevytvářela pokaždé znova. Každé vlákno zpracovávající požadavek od klienta potřebuje vlastní spojení do databáze, to dostane zavoláním metody `checkout`. Connection pool je plně thread-safe, což znamená, že lze používat libovolně z různých vláken současně. Tohoto je docíleno synchronizací jednotlivých metod přes `Monitor`. Monitor je vysokoúrovňové synchronizační primitivum pro řízení přístupu ke sdíleným prostředkům, typicky implementované pomocí jiného synchronizačního primitiva. V Ruby se monitor používá vložením mixinu `MonitorMixin` do naší třídy, kterou chceme synchronizovat. `MonitorMixin` nám poskytne metodu `synchronize`, která přebírá blok, který je vykonán vždy jen jedním vláknem v jeden okamžik. Po dokončení práce s databázovým spojením, by mělo být vráceno zpátky do poolu metodou `checkin`. `ConnectionPool` neimplementuje žádné limity na počet spojení, neboť každé vlákno, zpracovávající požadavky klienta, vlastní vždy jen jedno spojení a počet vláken je sám limitován třídou `ClientPool`.

Spojení s databází je reprezentováno třídou `DatabaseInterface`, která poskytuje jednotné rozhraní pro komunikaci. Toto rozhraní je převzato tak, jak bylo navrhnuto v rámci práce Pavla Jíry [18] a lehce modifikováno kvůli podpoře transakcí. Základní sada method je zachována kvůli kompatibilitě s XQuery modulem. Jeho strukturu můžeme vidět na obrázku 3.10.

Nad tímto interfacem je postavený transakční modul `DBTransactionInterface`. Tento transakční modul pracuje nad jednotlivými klíči už namapovaného XML dokumentu. Využívá k tomu `TransactionManager`, který je popsán v kapitole 4 Transakce.

`DatabaseInterface` zapouzdřuje objekt `driver`, který reprezentuje konkrétní připojení k databázi. V tuto chvíli je implementován pouze driver `Redis`, který používá gem `redis-rb` pro připojení k Redis databázi. Dále je možné implementovat drivery pro podporu i jiných



Obrázek 3.10: Rozhraní DatabaseInterface

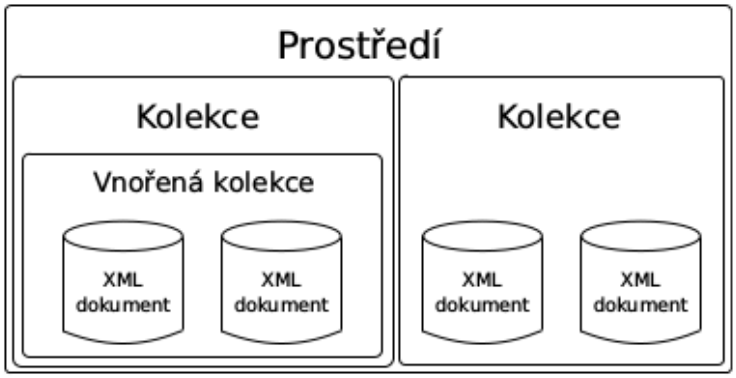
databází. Příkladem takové databáze by mohl být `FoundationDB`[2], `Dynamo`[1], nebo postgres, který podporuje typ `hstore`[5] pro ukládání key-value dat.

3.2.4 Mapování XML dokumentů

Mapování XML dokumentů je realizováno tak, jak je detailně popsáno v práci Pavla Jíry [18]. Zde si popíšeme pouze základy mapování a strukturu použitých tříd.

3.2.4.1 Mapování organizačních struktur

XML dokumenty jsou logicky uspořádány do prostředí a kolekcí.



Obrázek 3.11: Logická struktura databáze

Prostředí je nejvyšší logické jednotka v rámci databáze, která obsahuje jednotlivé kolekce a dokumenty společně s generátorem pro jejich ID. Prostředí lze vytvořit libovolné množ-

ství, ale jejich jména musí být unikátní. V rámci key-value databáze jsou jak prostředí, tak kolekce mapovány na ID, což umožňuje jejich snadné přejmenování či přesun dané struktury. Jednotlivá prostředí jsou generována pomocí inkrementace potřebného klíče. Pro názornost jsou zde uvedeny klíče, které obsahují informace týkající se prostředí:

- **info** - Obsahuje hash se základními informacemi o databázi, aktuálně obsahuje pouze pole <**iterator**>, které obsahuje string s číslem ke generování ID nových prostředí.
- **environments** - Obsahuje hash s mapováním jmen veškerých prostředí v rámci databáze na ID.
- **IDprostředí<info** - Podobně jako klíč info obsahuje hash s informacemi o daném prostředí, obsahuje pole <**iterator**> ke generování ID nových kolekcí a dokumentů v rámci prostředí.
- **IDprostředí:collections** - Obsahuje mapování jmen kolekcí na ID (např. "jméno kolekce" => "ID").

Pro organizaci daného prostředí jsou základní strukturou kolekce. Ty lze vnořovat do sebe a vytvářet tak stromovou strukturu. Každá kolekce může obsahovat libovolný počet dalších kolekcí či dokumentů. Kolekce musí být unikátní v rámci dané úrovně vnoření, lze tedy vytvořit kolekci "test" a vní vnořenou další kolekci se stejným názvem "test", ale už nelze vytvořit dvě stejně pojmenované kolekce vedle sebe. Zde je přehled použitých klíčů k reprezentaci kolekcí:

- **IDprostředí:IDkolekce:documents** - Podobně jako klíč kolekcí obsahuje tento klíč hash s mapováním jmen dokumentů na ID.
- **IDprostředí:IDkolekce:collections** - Obsah jako předchozí klíč, ovšem místo dokumentů obsahuje mapování kolekcí
- **IDprostředí:IDkolekce<info** - Obsahuje hash s informacemi o dané kolekci, aktuálně obsahuje:
  - <**name**> Pole obsahující název dané kolekce.
  - <**parent_id** Pole obsahující ID nadřazené kolekce, pokud existuje.

Nakonec samotné mapování dokumentů:

- **IDprostředí:IDkolekce:IDdokumentu<namespaces** - Obsahuje hash s deklarovanými jmennými prostory v rámci dokumentu, ukázka obsahu:

```
{"a" => "http://www.domain.com/a"}
```

- **IDprostředí:IDkolekce:IDdokumentu<info** - Obsahuje hash s kompletním obsahem XML dokumentu. Mezi pole, která může hash obsahovat patří:
  - **IDkořenovéhoElementu:IDelementu>pořadí>c>pořadí** - Obsahuje textový obsah komentáře.

– **IDkořenovéhoElementu:IDelementu>pořadí>d>pořadí** - Analogicky pak toto pole obsahuje textový obsah sekce CDATA taktéž bez uvozujících znaků.
– **IDkořenovéhoElementu:IDelementu>pořadí** - Toto pole nyní obsahuje většinu informací elementu, včetně atributů, textového obsahu a klíčů pro přímé potomky, jejichž pořadí je zachováno. Obsah klíče může být například následovný (pro lepší čitelnost přidány mezery kolem oddělovače):

```
1 < idAtributu < hodnotaAtributu < 1:2>1 <
1-Obyčejný text < 2-1:2>1>c>1 < 3-1:2>1>d>1
```

Kde první číslo uvádí počet atributů, následují samotné atributy a nakonec smíšený obsah elementu v podobě klíčů přímých potomků (element, CDATA, komentář) a obyčejného textu. Oddělovačem je znak "<", jelikož nemůže být obsahem XML kromě CDATA sekcí a komentářů, proto jejich mapování zůstalo beze změny a mají stále vlastní klíč. Přesto vidíme, že v případě jakéhokoliv textu (obyčejný, komentář či CDATA) je uveden typ v podobě prefixu "IDtypu-", který umožňuje rychlejší pocházení obsahu.

Práci s tímto mapováním XML dokumentů zajišťuje transformační vrstva. Tato vrstva byla z větší části převzána z původní práce a refaktorována, aby bylo možné použít nezávisle na původních `XMLDBapi` a `RedXmlApi` modulech, které již nejsou potřeba protože se veškeré api přesunulo do klientské části.

Mapování se provádí pomocí tříd `KeyBuilder` (mapování prostředí a kolekcí) a `KeyElementBuilder` (mapování dokumentů). Tyto třídy jsou použité ve službách `EnvironmentService`, `CollectionService` a `DocumentService`, které poskytují abstraktní vrstvu pro práci s XML dokumenty. Služba `MappingService` se stará o veškeré mapování názvů (ať už dokumentů, kolekcí či například elementů a atributů) na ID a naopak.

3.2.5 XQuery

Stejně jako mapování, tak modul XQuery byl převzat z původního konceptu RedXML a bylo provedeno pár nutných úprav, aby modul reflektoval strukturu projektu a styl kódu.

3.2.5.1 Parser knihovna

Základem modulu XQuery je třída `Executor`, která poskytuje základní rozhraní pro dotazování. Parsování samotného dotazu probíhá v externí C knihovně `SWIG`, která se automaticky kompiluje pomocí příkazu `rake compile`, jak je popsáno v kapitole 2.3.4.

Při testování této knihovny jsem zjistil, že při zadání syntakticky špatného dotazu knihovna vyhodí vyjímku v rámci C++ kódu, což způsobí pád celé aplikace včetně ruby interpretu a není tak možné tento chybový stav nijak odchytit. Toto chování je velmi nežádoucí, neboť jeden špatně napsaný XQuery dotaz ukončil celou aplikaci okamžitě bez jakýchkoliv chyb. Abych tuto chybu vyřešil musel jsem knihovnu upravit.

Nejprve bylo nutné upravit "interface file" `parser.i` pro generátor SWIG tak, aby překládal C++ vyjímky na ruby vyjímky přidáním posledních dvou řádek následovně:

```
%module Parsers
%{
/* Put header files here or function declarations like below */
#include "parser.h"
%}

%include parser.h
%exceptionclass MalformedInputException;
%exceptionclass ParseException;
```

Aby bylo možné tyto chyby odchytit musel být upraven soubor `parser.h` tak aby v C++ kódu bylo deklarováno, že dané vyjímky tyto funkce vyhazují. Například funkce `parse_XQuery`:

```
const char* parse_XQuery(const char* orig)
```

Byla upravena do této podoby:

```
const char* parse_XQuery(const char* orig) throw(ParseException)
```

Následně byl zovu vygenerován soubor `parser_wrap.cxx`, který přizpůsobuje `parser.h` pro kompilaci do binárního kódu kompatibilního s Ruby.

Od této chvíle veškeré chyby, které nastanou v této knihovně jsou posílány do ruby kódu, kde jsou jednoduše odchyceny a zpracovány pomocí begin rescue bloku.

3.2.5.2 Solver a Processor třídy

Další částí XQuery modulu jsou `Solver` a `Processor` třídy, které se starají o samotné vykonání dotazu. Třída `Solver` dostane expression tree vygenerovaný Parser knihovnou a vyřeší daný dotaz. Při řešení dotazu využívá tříd Processor pro komunikaci s databázovou vrstvou.

Výsledek dotazu je vrácen jako XML reprezentované v DOM stromu.

3.2.6 Instalace a spuštění

Stejně jako klient, je server instalován jako gem `redxml-server`. Gem poskytuje příkaz `redxml-server` kterým server spustíme. Tomuto příkazu můžeme předat několik parametrů pro nastavení hodnot jako je host, nebo port pro naslouchání. Zde je jejich přehled:

3.2.7 Testování

Jak již bylo zmíněno, projekt RedXML používá pro testování framework RSpec, ale původní koncept používá framework `Test::Unit`. Aby bylo monžé použít stávající unit testy, umožňuje RSpec použít asserty z jiných frameworků.

Testy jsou rozděleny do 20 souborů a 230 jednotlivých testovacích scénářů.

```
$ redxml-server --help
redxml-server [options]
  -v, --verbose                    Print more verbose output
  -L, --logfile PATH               Path to logfile
  -P, --pidfile PATH               Path to pidfile
  -c, --config PATH                Path to YAML config file
  -C, --concurency INT             Processor threads to use
  -p, --port INT                   Listening port
  -b, --host HOST                  Host to start listening on
  -h, --help                       Show help
```

Obrázek 3.12: Přehled parametrů serveru

3.3 Protokol

RedXML databáze používá vlastní binární protokol zasílání zpráv pro komunikaci mezi serverem a klientem, který je inspirován protokolem pro databázi PostgreSQL[7]. Jednotlivé zprávy posílané mezi klientem a serverem se nazývá paket. Kromě prvního paketu, který posílá server ihned po připojení klienta, klient vždy posílá dotazy, nebo další příkazy serveru a server na ně odpovídá výsledkem dotazu, nebo jinou odpovědí. Spojení vždy uzavírá klient příkazem `quit`.

3.3.1 Struktura paketu

První 4 byty paketu obsahují délku celého paketu, následují 4 byty obsahující verzi protokolu. Těchto 8 bytů není obsaženo v celkové délce. Následuje samotný příkaz ukončený nulovým bytem a parametr také ukončený nulovým bytem. Aby se předešlo problémům se synchronizací proudu, jak server tak klient načítají celý paket do paměti (za použití hodnoty v prvních 4 bytech – délky celého paketu) před samotným zpracováním. Tím je možné odchytit chybu při zpracovávání obsahu paketu a odeslat chybový paket druhé straně. V extrémních případech, kdy není možné celý paket načíst do paměti (např. nedostatek paměti), je možné pomocí celkové délky určit, kolik bytů lze přeskočit před pokračnováním čtení paketu. Pokud klient nebo server pošle protistraně neúplný paket, je paket zahozen a spojení ukončeno. Tím se předejde chybám se synchronizací zpráv.

3.3.1.1 Příkazy

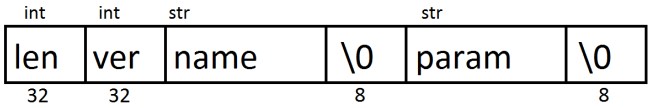
Příkaz je reprezentován jedním znakem (1 byte), následován 4 byty obsahující délku parametru a ukončen nulovým bytem. Parametr je pak sekvence znaků ukončené nulovým bytem. Pokud je parametr prázdný (délka parametru je 0), celý paket je ukončen dvěma nulovými byty ihned za příkazem.

Znak příkazu (tag), je malé písmeno. Například klient posílá XQuery dotaz do databáze, výsledný paket vypadá následovně:

50	1	e	34	0	\1env\1coll\1doc('test.xml')/root	0

```
Run options: include {:focus=>true}

All examples were filtered out; ignoring {:focus=>true}
.....................................................................
...*.................................................................
.....................................................*...............
.....................

Pending:
  RedXML::Server::Executors RedXML::Server::Executors::Execute returns string as resut
    # Temporarily skipped with xdescribe
    # ./spec/redxml/executors_spec.rb:26
  RedXML::Server::ClientPool#que delete inactive threads
    # Move threads to server worker
    # ./spec/redxml/client_pool_spec.rb:27

Finished in 4.15 seconds (files took 1.97 seconds to load)
230 examples, 0 failures, 2 pending

Top 3 slowest examples (0.52229 seconds, 12.6% of total time):
  RedXML::Server::XQuery::Executor::execute update insert node "two text"
  after doc("catalog.xml")/catalog/product[3]/name
    0.25465 seconds ./spec/redxml/xquery/update_spec.rb:5
  Transactions sx x nr
    0.13469 seconds ./spec/redxml/transactions_spec.rb:72
  Transactions cx x sx
    0.13295 seconds ./spec/redxml/transactions_spec.rb:65

Top 3 slowest example groups:
  Transactions
    0.11208 seconds average (0.67246 seconds / 6 examples) ./spec/redxml/transactions_spec.rb:3
  RedXML::Server::XQuery::Executor
    0.05563 seconds average (1.67 seconds / 30 examples) ./spec/redxml/xquery/update_spec.rb:3
  RedXML::Server::Launcher
    0.03259 seconds average (0.06517 seconds / 2 examples) ./spec/redxml/launcher_spec.rb:3
```

Obrázek 3.13: Test serveru



Obrázek 3.14: Struktura paketu

Server tento dotaz vyhodnotí a odešle odpověď jako parametr stejného příkazu, například odpověď na předchozí dotaz vypadá takto:

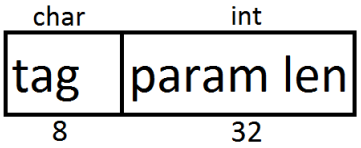
27	1	e	21	0	<root><child/></root>	0

Pokud při vyhodnocování dotazu nastane chyba, například syntakticky špatný dotaz, neexistující prostředí, kolekce, nebo dokument, tag příkazu se mění na velké písmeno a jako parametr se uvede text nastalé chyby. Pokud odpověď neobsahuje žádná data, je parametr prázdný a délka parametru je nastavena na 0. Přehled všech příkazů a jejich příslušné tagy jsou uvedeny v tabulce 3.1.

3.3.2 Implementace

Jelikož je kód implementující protokol společný jak pro server tak klienta, byl oddělen do samostatného gemu `redxml-protocol`, aby se odstranila duplicita kódu. Gem je pak použit jako závislost v obou gemech pro server tak klienta.

Pro práci z protokolem, vytváření jednotlivých paketů a jejich parsování, je modul `Protocol`. Tento modul obsahuje dvě základní třídy: `Packet` a `PacketBuilder`.



Obrázek 3.15: Struktura příkazu

`Packet` slouží pro reprezentaci paketů a obsahuje tyto informace:

- **data** - Binární data paketu.
- **length** - Délka paketu. Odpovídá číslu v prvních 4 bytech.
- **protocol** - Verze protokolu.
- **command** - Název příkazu. Reprezentován jako symbol.
- **command_tag** - Tag příkazu.
- **param_length** - Délka parametru.
- **param** - Parametr.
- **error?** - Vrací true pokud je paket nastaven jako chybový.

Dále `Packet` obsahuje pomocné metody `error` a `response`. `error` vytvoří nový paket a označí packet jako chybový, jako parametr se předává chybová zpráva. `response` je metoda, která vytvoří z aktuálního paketu odpověď.

3.3.3 Testování

Testování protokolu je provedeno tak, že pro každý definovaný příkaz je vytvořen paket a poté porovnány jeho vytvořené data s daty předpřipravenými.

Testy jsou rozděleny do 3 souborů po 34 testovacích scénářů.

```
Run options: include {:focus=>true}

All examples were filtered out; ignoring {:focus=>true}
................................

Finished in 0.03428 seconds (files took 0.68326 seconds to load)
34 examples, 0 failures
```

Obrázek 3.16: Testy protokolu

Tag	Název	Parametry	Popis
h	hello	Verze serveru	Příkaz, který posílá pouze server při připojení klienta. Parametr obsahuje string s verzí serveru. Přklad: "`RedXML-0.0.1`"
p	ping		Prázdný požadavek serveru, server odpovídá stejným paketem. Slouží jako ověření, že spojení je navázáno a že server odpovídá.
q	quit		Požadavek na ukončení spojení. Odesílá vždy klient serveru před odopjením.
e	execute	XQuery	XQuery dotaz na databázi na daný dokument. Parametr obsahuje prostředí, kolekci a samotný dotaz oddělený bytem 1 (znak \1). Příklad: "`\1env\1coll\1doc('test.xml')/root`"
l	load_document	Název dok.	Načte celý XML dokument z databáze. Parametrem je opět prostředí, kolekce a název dokumentu, např: "`\1env\1coll\1test.xml`"
s	save_document	XML dok.	Uloží XML dokument do databáze. Parametr je složen z prostředí, kolekce, názvu dokumentu a xml dat. Příklad: "`\1env\1coll\1doc.xml\1<root></root>`"
b	begin		Zahájí transakční zpracování. Nemá žádné parametry.
c	commit		Ukončí a uloží data z právě prováděné transakce. Nemá žádné parametry.

Tabulka 3.1: Příkazy a jejich parametry

Kapitola 4

Transakce

Transakce je skupina příkazů, které převedou databázi z jednoho konzistentního stavu do druhého. S transakcí se zachází jako s jedním celkem, který pro přístup k jednotlivým objektům transakce používá zámky. Aby byla transakce dobře formovaná musí splňovat následnující požadavky[20]:

- Transakce zamyká objekt chce-li k němu přistupovat.
- Transakce nezamyká objekt, který je již touto transakcí zamčený.
- Transakce neodmyká objekt, který není touto transakcí zamčený.
- Po ukončení transakce všechny objekty, které byly touto transakcí uzamčeny, jsou odemknuty.

Jednotlivé transakce také musí splňovat tzv. vlastnosti ACID.

- **Atomicita** (Atomicity) - transakce se tváří jako jeden celek (atomická operace), buď proběhne celá, nebo vůbec
- **Konzistence** (Consistency) - transakce transformuje databázi z jednoho konzistentního stavu do jiného konzistentního stavu
- **Nezávislost** (Isolation) - jednotlivé efekty transakce nejsou viditelné jiným transakcím
- **Trvanlivost** (Durability) - efekty potvrzené úspěšně dokončené transakce jsou uloženy do databáze

4.1 Protokol taDOM2

Aby byla transakce izolovaná, tj. pokud jedna transakce nevidí změny provedené jinou konkurenční transakcí, je nejčastěji dosaženo pomocí zámků. Zamykání probíhá podle uzamykacího protokolu, který je sada pravidel, která nám zajistí dodržení požadavků kladené na danou transakci.

Zámek na konetx-tovém uzlu	**Zámek na rodiči**	**Význam zámku**
IR	IR	Intention read: Alespoň jeden z potomků kontextového uzlu je čten.
NR	IR	Node read: Kontextový uzel je čten.
LR	IR	Level read: Kontextový uzel a všechny jeho dědi jsou čteny.
SR	IR	Subtree read: Kontextový uzel a všechny jeho potomcu jsou čteny.
IX	IX	Intention exclusive: Alespoň jeden potomek kontextového uzlu, ale ne přímý potomek, je měněn.
CX	IX	Child exclusive: Alespoň jeden přímý potomkek je měněn.
SU	IR	Subtree update: Subtree update: Kontextový uzel a všechny jeho potomci jsou čteny. Možnost konverze na SR nebo SX.
SX	CX	Subtree exclusive: Kontextový uzel a všechny jeho potomci budou smazány.

Tabulka 4.1: Typy zámků protokolu taDOM2

Protokol použitý v této práci se nazývá taDOM2. Tento protokol pracuje nad upraveným DOM (Document Object Model) stromem. Reprezentaci XML dokumentu rozšiřuje DOM o tři nové typy uzlů: `document root`, `attribute root`, a `string`. Tímto je dosaženo větší strukturalizaci objektového modelu a umožňuje tak zamykat menší počet uzlů k dosažení stejného výsledku.

Protokol taDOM2 používá 8 typů zámků, které jsou popsány v tabulce 4.1. Při situaci, kdy již daná transakce zámek už má a požaduje zámek jiný, dochází ke konverzi zámku podle konverzní matice.

```
def lock_request(node, lock)
  if lock.compatible? node.lock
    set_lock(node, lock)
  else
    wait_for(node)
  end
end

while node
  lock_request node, lock
  node = node.parent
  lock = lock.parent
end
```

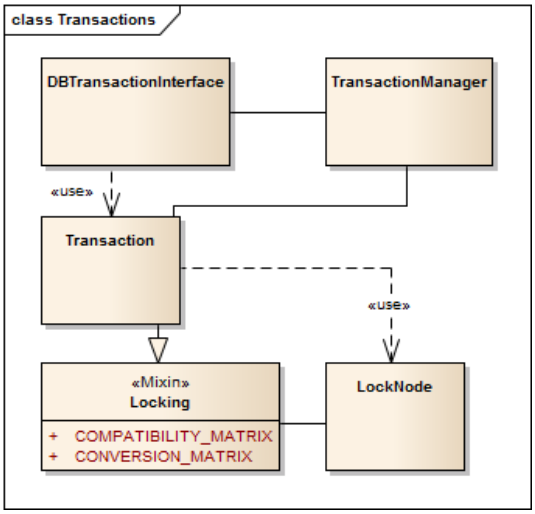
Obrázek 4.1: Algoritmus taDOM2

Algoritmus zamykání protokolu taDOM2 je znázorněn na obrázku 4.1. Algoritmus je založen na dvou základních operacích:

- `set_lock(node, lock)` - přidělení zámku `lock` na uzel `node`. Součástí je i operace konverze zámku podle konverzní matice.
- `lock1.compatible?(lock2)` - kontrola kompatibility zámku `lock1` a `lock2`.

4.2 Implementace

Základem pro implementaci transakcí se stala třída `DatabaseInterface`. Tato třída se stará o veškerou komunikaci s databází a je tak vhodným místem. Byla tak vytvořena třída `DBTransactionInterface`, která rozšiřuje stávající interface transakčním zpracováním nad jednotlivými uzly XML dokumentu.



Obrázek 4.2: Struktura transakčního modulu

Pro každou metodu, která nějakým způsobem pracuje s XML dokumentem bylo doplněno zamykání daného uzlu. Například zamknutí uzlu pro čtení je prováděno v metodě `get_hash_value`:

```
def get_hash_value(key, field)
  return super unless @transaction
  if is_content? key
    node = ContextNode.new(key, field)
    @transaction.acquire_lock node, :NR
  end
  super
end
```

Aby bylo možné takto pracovat s jednotlivými transakcemi byl vytvořen modul `TransactionManager`, který uchovává informace o všech transakcích, které právě probíhají. Tato třída obsahuje dvě metody pro práci s transakcemi: `transaction` a `release`. První metoda vytvoří nový objekt `Transaction`, který nese informaci o probíhající transakci. Druhá metoda pak transakci uvolní - tedy uvolní všechny zámky získané v rámci dané transakce.

	nr	sx
ir	x	x
nr	-	x
ix	-	x
cx	-	x
sx	x	-

Tabulka 4.2: Vybrané testovací scénáře transakcí

Třída `Transaction` se stará o veškeré zamykání jednotlivých uzlů, u kterých si drží informaci o typu zámku. Samotné zamykání zajišťuje algoritmus taDOM. Aby byla zajištěna atomicita i v rámci jednotlivých vláken, je třída koncipována jako thread-safe za pomoci zamykání jednotlivých metod přes Monitor objekt. K zamykání uzlů je použita pomocná třída `LockMode`, která reprezentuje pouižitý zámek díky které, lze jednoduše konvertovat zámky mezi sebou pomocí konverzní matice, nebo lze porovnávat dva zámky jestli jsou kompatibilní podle matice kompatibility.

Kvůli chybějícím informacím o právě prováděné akci s daným uzlem ve třídě `DatabaseInterface`, nebylo tak možné implementovat akci rollback, která vrátí veškeré změny zpět. Implementace této fuknce by vyžadovalo přepsání veškerého kódu, který pracuje s databázovou vrstvou, což je z největší části modul XQuery. Ten využívá přímo metod třídy `DatabaseInterface` bez informace jak s daným uzlem hodlá pracovat. Tudíž není možné uložit právě prováděnou akci pro pozdější navrácení do původního stavu.

4.3 Testování

Pro testování bylo nutné vybrat testovací scénáře tak aby se pokryli nejčastější případy. Tyto scénáře jsou otestovány v souboru `transactions_spec.rb`.

Základními scénáři je tak skupina těchto 6 případů, které zajišťují co největší pokrytí.

```
Run options: include {:focus=>true}

All examples were filtered out; ignoring {:focus=>true}

Transactions
  sx x nr
  ir x sx
  cx x sx
  ix x sx
  nr x sx
  ir x nr

Finished in 0.71193 seconds (files took 0.4351 seconds to load)
6 examples, 0 failures

Top 3 slowest examples (0.41025 seconds, 57.6% of total time):
  Transactions sx x nr
    0.13829 seconds ./spec/redxml/transactions_spec.rb:72
  Transactions ir x sx
    0.13829 seconds ./spec/redxml/transactions_spec.rb:37
  Transactions ix x sx
    0.13367 seconds ./spec/redxml/transactions_spec.rb:58
```

Obrázek 4.3: Testy transakčního modulu

Kapitola 5

Závěr

Cílem této práce bylo vytvoření nativní XML databáze, která ukládá XML dokumenty pomocí key–value databáze. Základem se staly diplomové práce Pavla Jíry a Martina Kostolného, které se zabývaly mapováním XML dokumentů a provádění XQuery dotazů, které jsem zrefaktoroval tak, abych je mohl začlenit do RedXML databáze. Výsledkem byla databáze typu klient–server s podporou transakcí.

Celý projekt jsem psal s ohledem na čitelnost a znovupoužitelnost kódu, za pomoci různých nástrojů pro vývoj. Původní projekt RedXML se mi podařilo refaktorovat tak, aby splňoval standardy psaní Ruby kódu a bylo tak možné jednoduše na tento projekt v budoucnu navázat a implementovat nové funkcionality.

Díky architektuře klient-server jsem vytvořil databázi použitelnou nejen v projektech psaných v jazyce Ruby, ale i v jiných jazycích. Součástí projektu je i klient pro příkazovou řádku s podporou historie příkazů a pokročilé editace a knihovna pro jazyk Ruby pro komunikaci se serverem pomocí API.

Navázáním na původní projekt RedXML jsem tak vytvořil server, který ukládá XML dokumenty do key-value databáze s podporou transakčního zpracování, které bylo implementováno pomocí protokolu taDOM2.

Celý projekt jsem implementoval pomocí metodiky testy řízeného vývoje, kdy jsem pro každou novou funkcionalitu vytvořil testy a poté implementoval samotnou funkcionalitu. Tím jsem vytvořil projekt plně otestovaný jednotkovými a integračními testy.

Všechny vytyčené cíle byly splněny, takže soudím, že práce plně splňuje zadání a doufám, že se projekt bude dále rozvíjet.

Literatura

[1] Amazon DynamoDB. Dostupné z: <http://aws.amazon.com/dynamodb/>.

[2] FoundationDB. Dostupné z: <https://foundationdb.com/>.

[3] Github Ruby Style Guide, 2014. Dostupné z: <https://github.com/styleguide/ruby>.

[4] google-styleguide, 2014. Dostupné z: <https://code.google.com/p/google-styleguide/>.

[5] PostgreSQL Documentation - hstore, 2014. Dostupné z: <http://www.postgresql.org/docs/9.3/static/hstore.html>.

[6] PEP 8 - Style Guide for Python Code, 2014. Dostupné z: <https://www.python.org/dev/peps/pep-0008/>.

[7] PostgreSQL Documentation - Frontend/Backend Protocol. Dostupné z: <http://www.postgresql.org/docs/current/static/protocol.html>.

[8] Rake, 2014. Dostupné z: <https://github.com/ruby/rake>.

[9] RSpec, 2014. Dostupné z: <https://github.com/rspec/rspec/wiki/Background>.

[10] What is a gem, . Dostupné z: <http://guides.rubygems.org/what-is-a-gem/>.

[11] A community-driven Ruby coding style guide, . Dostupné z: <https://github.com/bbatsov/ruby-style-guide>.

[12] thoughtbot. Dostupné z: <http://thoughtbot.com/>.

[13] Why is software testing necessary? Dostupné z: <http://istqbexamcertification.com/why-is-testing-necessary/>.

[14] CHELIMSKY, D. *The RSpec book : behaviour-driven development with RSpec, Cucumber, and Friends*. Lewisville, Tex : Pragmatic, 2010. ISBN 1934356379.

[15] FLANAGAN, D. *The Ruby programming language*. Beijing Sebastopol, CA : O'Reilly, 2008. ISBN 978-0596516178.

[16] FOWLER, M. Catalog of Refactorings, 2014. Dostupné z: <http://refactoring.com/catalog/>.

[17] FOWLER, M. *Refactoring* [online]. 2014. [cit. 16.12.2014]. Dostupné z: <http://refactoring.com/>.

[18] JíRA, B. P. Transformace XML do key-value databáze Redis. 2012.

[19] PRESTON-WERNER, T. *Semantic Versioning 2.0.0* [online]. 2014. [cit. 16.12.2014]. http://semver.org/. Dostupné z: <http://semver.org/>.

[20] STRNAD, P. Rozvrhovač transakcí v projektu CellStore. 2007.

[21] THOMAS, D. *Programming Ruby 1.9 & 2.0 : the pragmatic programmers' guide*. Dallas, Texas : The Pragmatic Bookshelf, 2013. ISBN 978-1937785499.

[22] WIKIPEDIE, P. *Refaktorování* [online]. 2014. [cit. 16.12.2014]. Dostupné z: <http://cs.wikipedia.org/wiki/Refaktorování>.

[23] WIKIPEDIE, P. *Programování řízené testy* [online]. 2014. Dostupné z: <http://cs.wikipedia.org/wiki/Programování_řízené_testy>.

Příloha A

Uživatelská a instalační příručka

Spuštění RedXML databáze spočívá v nainstalování gemů `redxml-server`, `redxml-protocol` a `redxml-client`, které jsou na přiloženém CD.

```
gem install redxml-protocol-0.0.1.gem
gem install redxml-server-0.0.1.gem
gem install redxml-client-0.0.1.gem
```

Spuštění serveru se provádí příkazem `redxml-server`.

```
$ redxml-server
```

Stejně tak klient příkazem `redxml-client`.

```
$ redxml-client
```

Příloha B

Obsah přiloženého CD

Tato příloha obsahuje seznam elektronických příloh k této práci - CD. Jedná se o zdrojové kódy celého celého projektu i samotný text této práce.

- **redxml-client** - Adresář obsahující zdrojové kódy klientské části RedXML.
- **redxml-server** - Adresář obsahující zdrojové kódy serverové části RedXML.
- **redxml-protocol** - Adresář obsahujcí implementaci protokolu použitého pro komunikaci mezi klientem a serverem.
- **rexml-text** - Adresář obsahující text této práce ve formě zdrojových souborů TEX a dalších souborů potřebných pro sazbu.
- **redxml-protocol-0.0.1.gem** - Instalační soubor gemu `redxml-protocol`.
- **redxml-server-0.0.1.gem** - Instalační soubor gemu `redxml-server`.
- **redxml-client-0.0.1.gem** - Instalační soubor gemu `redxml-client`.