VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

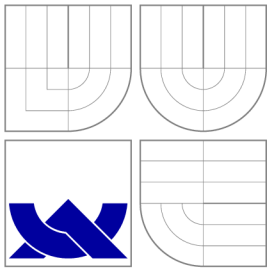
JAZYK PRO DOTAZOVÁNÍ JAVA AST

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

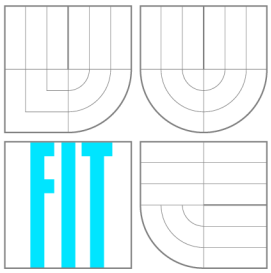
AUTOR PRÁCE
AUTHOR

Bc. JIŘÍ BÍLEK

BRNO 2015



VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

JAZYK PRO DOTAZOVÁNÍ JAVA AST

JAVA AST QUERY LANGUAGE

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE
AUTHOR

Bc. JIŘÍ BÍLEK

VEDOUCÍ PRÁCE
SUPERVISOR

Ing. ZBYNĚK KŘIVKA, Ph.D.

BRNO 2015

Abstrakt

Cílem této práce je návrh dotazovacího jazyka nad abstraktním syntaktickým stromem Java kódu a implementace nástroje, který využívá tento dotazovací jazyk. V práci se nachází průzkum dostupných grafových databází a podrobnější studium grafových databází Neo4J a Titan. Následuje průzkum dostupných nástrojů pro analýzu Java bajtkódu a opět podrobnější zkoumání nástrojů Procyon a BCEL. Dále práce obsahuje návrh jazyka a detailní popis implementace nástroje společně s popisem uložení jednotlivých entit do grafové databáze. Závěrem se práce zabývá experimenty s vytvořeným nástrojem a vyhodnocením časové složitosti knihovny.

Abstract

The purpose of this thesis is to design a Java AST query language and implement tool that uses the query language. This work overviews graph databases and their libraries with focus on Neo4J and Titan. This thesis overviews tools Java bytecode analysis as well. Libraries Procyon and BCEL are described in detail. The work includes a proposal the query language and detailed description of the tool implementation, together with the detailed description of the way how Java entities are stored into the graph databases. In the end, the work deals with experiments and the evaluation of the time complexity of the library.

Klíčová slova

Java, AST, BCEL, Procyon, Neo4J, Titan, Grafové databáze, Dekompilace, Dotazovací jazyk, XPath, Frames, ANTLR.

Keywords

Java, AST, BCEL, Procyon, Neo4J, Titan, Graph database, Decompilation, Query language, XPath, Frames, ANTLR.

Citace

Jiří Bílek: Jazyk pro dotazování Java AST, diplomová práce, Brno, FIT VUT v Brně, 2015

Jazyk pro dotazování Java AST

Prohlášení

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně pod vedením pana Ing. Zbyňka Křivky, Ph.D.

.....
Jiří Bílek
25. května 2015

Poděkování

Děkuji svému vedoucímu Ing. Zbyňku Křivkovi Ph.D., a konzultantu Ing. Ondřeji Žižkovi ze společnosti Red Hat za cenné rady, odbornou pomoc a vedení, jenž mi poskytli při řešení tohoto projektu. Dále děkuji svým rodičům za veškerou podporu, bez které by vypracování této práce nebylo možné. Díky patří také mé sestře za motivaci a inspiraci nejen při psaní této práce. Rád bych takto také poděkoval celé své rodině za modlitbu a psychickou podporu. V neposlední řadě bych chtěl poděkovat své přítelkyni za trpělivost a cenné rady.

Obsah

Kapitola 1

Úvod

V případě ztráty zdrojových kódů aplikací, které máme ve zkompilovaném stavu, je dekompilace jediná možnost k opětovnému získání těchto zdrojových kódů. Další možné využití dekompilace může být při hledání škodlivého kódu v aplikacích. U velkých projektů, které mohou mít například řádově tisíce souborů (tříd, rozhraní a pod.), je po dekompilaci problém zorientovat se ve zdrojových kódech a vyhledávání je tak velice obtížné. Z tohoto důvodu je zapotřebí vytvořit dotazovací jazyk, který ve spustitelných souborech Java aplikací nalezne potřebné části implementace.

Základním kritériem vytvořené aplikace pro interpretaci tohoto jazyka musí být samozřejmě rychlost a intuitivnost. Ke zvýšení rychlosti může přispět dekompilace pouze do formy abstraktního syntaktického stromu (AST) a provádění dotazů nad touto strukturou. K dalšímu zrychlení lze dospět pomocí detekce typu informací požadovaných dotazem a dle této znalosti použití vhodného nástroje pro dekompilaci. Tímto se nebude provádět úplná dekompilace, pokud bude dotaz požadovat pouze povrchové informace. Intuitivnost dotazovacího jazyka bude zajištěna inspirací návrhu jazyka již existujícím a používaným jazykem.

Ve následující kapitole této práce jsou stručně probrány vztahy mezi třídami a rozhraními. Dále je kapitola zaměřena na grafové databáze, kde po popisu principu těchto databází následuje průzkum dostupných knihoven pro práci s těmito databázemi. Nejvhodnější databáze jsou následně popsány blíže. Dalším bodem teoretického rozboru je studium nástrojů pro analýzu bajtkódu, které je započato shrnutím dostupných dekompilátorů jazyka Java. Tento souhrn dále pokračuje podrobnějším popisem vybraných nástrojů. Třetí kapitola obsahuje specifikaci navrhované aplikace v rámci způsobu použití nástrojů pro analýzu bajtkódu a způsobu uložení AST do grafové databáze. Součástí návrhu implementace je také návrh dotazovacího jazyka nad Java AST a sada komplexních dotazů, které dovede tento jazyk zpracovat. Čtvrtá kapitola se podrobně zabývá implementací dotazovacího jazyka včetně uvedení použitého nástroje ANTLR. Další kapitola je věnována bližšímu zkoumání prvků jazyka Java a jejich interpretaci odpovídajících prvků v grafové databázi. Šestou kapitolou je implementace shrnuta. Jsou zde sepsány veškeré použité knihovny. Dále zde lze nalézt popis struktury aplikace v pohledu časového zařazení zpracování dotazu a zpracování vstupních souborů. V předposlední kapitole jsou komentovány experimenty, které byly s výslednou aplikací prováděny. Zejména se jedná o experimenty pozorující rychlost a složitost algoritmů. Lze zde najít grafy znázorňující časovou složitost subsystému aplikace.

Kapitola 2

Teoretický rozbor

V následující kapitole jsou probrány znalosti nezbytně nutné pro návrh dotazovacího jazyka nad Java AST. V první řadě se jedná o shrnutí vztahů tříd a objektů v jazyce Java důležitých pro návrh uložení v databázi a také pro specifikaci požadovaného druhu databáze. Protože je předem zjevné, že nejpřirozenější uložení těchto vztahů bude v grafových databázích, následuje přehled těchto databází se stručným vysvětlením principu. Vybrané databáze jsou zkoumány podrobněji v podkapitolách.

Závěrem této sekce je výsledek průzkumu dostupných dekompilátorů se stručnou charakteristikou. Dále je více rozebrán vybraný dekompilátor. Navíc je u dekompilátorů podrobněji popsána knihovna BCEL, která poskytuje jenom částečnou dekompilaci, přesto však může být pro projekt zajímavá.

2.1 Vztahy tříd

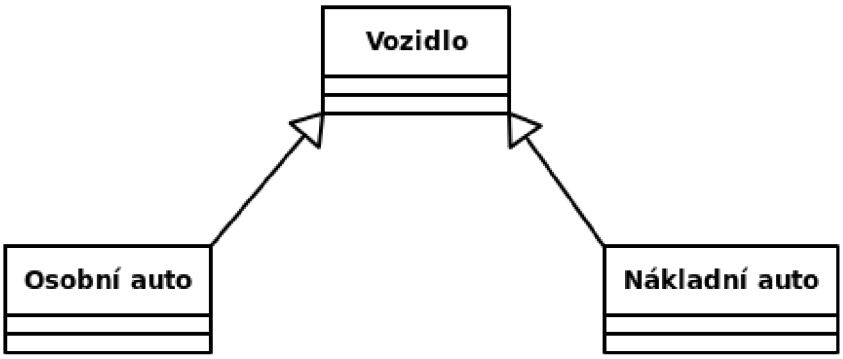
Vztahy mezi třídami lze rozdělit následovně.

- Vztahy mezi třídami jako datovými typy:
  - generalizace,
  - realizace.
- Vztahy mezi třídami jako instancemi:
  - asociace,
  - agregace,
  - kompozice.

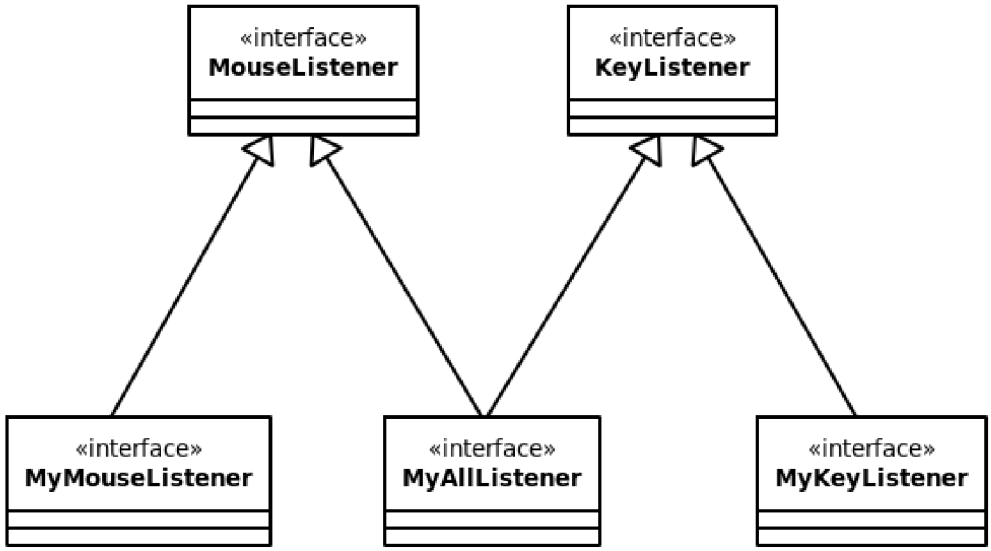
Generalizace

Generalizace neboli specializace nastává při dědění třídy. V Javě se pro tento vztah používá klíčové slovo `extends`. Třída tímto zdědí metody a atributy třídy, od které je děděno.

Generalizace je jednosměrná, takže vytváří hierarchii tříd (rodič → potomek). Třída může dědit pouze od jedné třídy (obrázek 2.1), ale rozhraní může dědit od více rozhraní (obrázek 2.2).



Obrázek 2.1: UML diagram s ukázkou dědičnosti tříd.



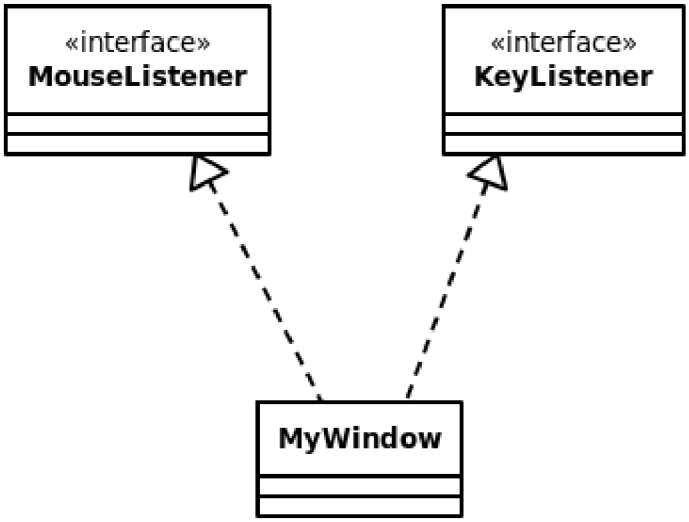
Obrázek 2.2: UML diagram s ukázkou dědičnosti rozhraní.

Realizace

Zde se jedná o implementaci rozhraní. Rozhraní (interface) definuje metody, které je potom nutné v realizační třídě implementovat. Odkaz na rozhraní se u této třídy provádí klíčovým slovem `implements`. Realizace je jednosměrný vztah, stejně jako generalizace. Realizovat je možné více rozhraní. Omezením tohoto vztahu je, že vždy na jedné straně vztahu musí být rozhraní a na druhé straně třída. Z toho vyplývá, že graf realizace musí mít vždy právě dvě úrovně (rozhraní a třída). Na obrázku 2.3 je příklad UML diagramu znázorňujícího realizaci.

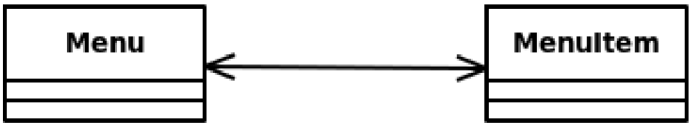
Asociace

Jedná se o poměrně obecný způsob vztahu mezi dvěma instancemi. Příkladem může být použití jedné třídy v jiné třídě jako parametru metody, atributu třídy a podobně. Následující vztahy (agregace a kompozice) jsou specifičtější.



Obrázek 2.3: UML diagram s ukázkou realizace.

Tento vztah může (ale nemusí) být obousměrný. Tedy pokud zakreslíme tento vztah mezi třídami do grafu, můžeme v grafu získat cykly (obrázek 2.4). Z toho vyplývá, že grafem nemusí být hierarchický graf, jak tomu bylo v předchozích případech. Tato vlastnost platí i pro agregaci a kompozici.



Obrázek 2.4: UML diagram s ukázkou obousměrné asociace tříd.

Případ na obrázku může nastat, pokud položka menu (MenuItem) obsahuje zpátky referenci na menu, do kterého patří.

Agregace

Tento druh vztahu vytváří mezi instancemi volnou vazbu - mohou bez sebe existovat. Například objekt `PC` používá objekt `Tiskarna`. Pokud bude vztah ukončen, stále mohou jednotlivé objekty fungovat.

Kompozice

Kompozice je opak agregace. Tímto vztahem rozumíme silnou vazbu, tedy že jeden objekt bez druhého nemůže existovat, neboť ztrácí smysl. Jako příklad si můžeme vzít objekt obdélník a bod. Body definují umístění a velikost obdélníku. Bez tohoto objektu nemůže obdélník existovat.

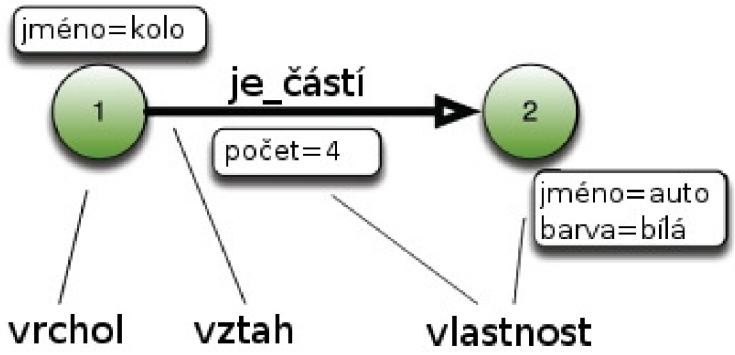
2.2 Grafové databáze

Databázový koncept označovaný zkratkou NoSQL[1] je specifický svým alternativním přístupem k ukládání dat a dotazováním nad nimi. S daty se nepracuje jako v klasických relačních databázích. Možným příkladem jsou grafové databáze, kde jsou data uložena v orientovaných grafech.

Grafové databáze využívají následující tři stavební bloky:

- vrchol,
- hrana,
- vlastnost.

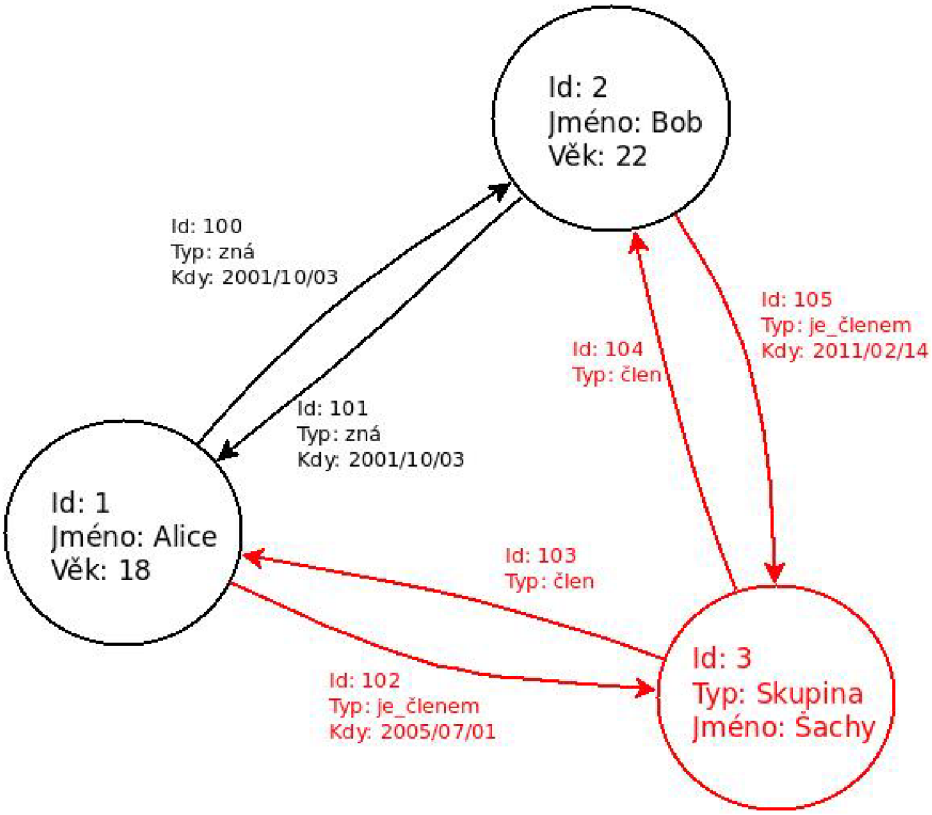
Vrchol grafu představuje záznamy databáze. Například vrchol 1 a 2 v následujícím obrázku 2.5. Hrany představují vztahy mezi vrcholy v databázi. Nutným parametrem hrany je směr a typ. Podle následujícího obrázku 2.5 je tak zřejmé, že vrchol 1 je ve vztahu s vrcholem 2. Tento vztah je otypován jako `je_částí`, tedy objekt zastoupený záznamem vrchol 1 je pravděpodobně součástí jiného objektu zastoupeného záznamem vrchol 2. Posledním stavebním blokem je vlastnost. Lze takto blíže specifikovat jak vrchol, tak i hranu. Dle následujícího příkladu (obrázek 2.5) je tak snadno pochopitelné, že vrchol 1 jménem kolo je součástí vrcholu 2 jménem auto, které je mimochodem bílé barvy, a že tyto součásti jsou zapotřebí čtyři (počet=4).



Obrázek 2.5: Stavební bloky grafové databáze [29].

Výhodou grafových databází je přirozený popis dat s podobnou strukturou. Příkladem může být mapování struktury objektově orientovaných aplikací bez pevně stanoveného schématu [25]. Dalším častým využitím grafových databází je popis vztahů mezi lidmi, tedy tzv. sociální síť (Facebook, Google+). Z následujícího komplexnějšího příkladu (obrázek 2.6) lze vyčíst, že se Alice a Bob znají (obousměrný vztah) a že oba jsou členové skupiny šachových hráčů. Je vhodné si povšimnout, že vztah mezi například vrcholem interpretující Alici a vrcholem interpretující šachový kroužek není obousměrný vztah, nýbrž dva rozdílné vztahy (`člen` a `je_člen`) s opačnou orientací. Tímto je zachována sémantika dat, aby bylo zřejmé kdo je příjemcem vztahu a kdo poskytovatelem. Mnohé knihovny grafových databází však umožňují procházení grafu v obou možných orientacích vztahů.

[1] Tato zkratka bývá často vysvětlována jako „no only SQL“ [25].



Obrázek 2.6: Příklad grafové databáze[2].

Na trhu je k dispozici velké množství knihoven pro práci s grafovými databázemi. Příkladem komerčních knihoven může být produkt firmy Oracle s názvem Oracle Spatial and Graph [31] nebo produkt Oracle NoSQL Database [30]. Další známé produkty jsou například InfiniteGraph [16], GraphBase [15] nebo Sqrrl Enterprise [34]. Relevantní databáze se však řadí především mezi produkty s open source filozofií. Příkladem lze uvést Neo4J [23], Titan [24], OrientDB [28] a InfoGrid [17].

Článek na portálu db-engines [2] uvádí srovnání různých knihoven pro grafové databáze. Výsledky porovnání jsou uvedeny v tabulce 2.1. Srovnání v této tabulce je prováděno na základě hodnocení oblíbenosti. Do tohoto hodnocení se zahrnuje počet vyhledávání relevantních slov v internetových vyhledávačích, frekvence diskuzí na populárních portálech (Stack Owerflow, DBA Stack Exchange), množství pracovních pozic s požadavkem na znalost patřičné databáze a diskutovatelnost na sociálních sítích.

V následujících podkapitolách jsou podrobněji zkoumány dvě nejlépe hodnocené knihovny pro grafové databáze z této tabulky, tedy Neo4J a Titan.

2.2.1 Neo4J

Knihovna pro práci s grafovými databázemi Neo4J je implementována v programovacím jazyce Java. Tato grafová knihovna využívá dotazovacího jazyka Cipher a její domovská stránka je na adrese [23]. Na těchto stránkách je taktéž k dispozici stažení knihovny určené

[2] Obrázek byl převzat z `http://en.wikipedia.org/wiki/Graph_database`.

Pořadí	DBMS	Skóre
1.	Neo4J	25.17
2.	Titan	2.85
3.	OrientDB	2.18
4.	Sparksee	0.91
5.	Giraph	0.56
6.	ArangoDB	0.48
7.	InfiniteGraph	0.28
8.	Sqrrl	0.19
9.	InfoGrid	0.14
10.	FlockDB	0.12
11.	GraphBase	0.03
12.	HyperGraphDB	0.03
13.	Amisa Server	0.00
13.	GlobalDB	0.00

Tabulka 2.1: Srovnání knihoven pro grafové databáze podle článku na db-engines [2].

pro komunitu a tedy zdarma pod licencí Apache 2.0. Neo4J je možné získat i v komerční verzi.

Používat grafovou knihovnu Neo4J lze několika způsoby. Jedním ze způsobů je spuštění databázového serveru a používání pomocí prohlížeče (viz Cipher). Další možností je Java API.

Cipher

Po stažení balíku s databází lze v adresáři aplikace zadat následující příkaz, který spustí databázový server.

```
./bin/neo4j console
```

Následně lze v prohlížeči navštívit stránku http://localhost:7474/, kde je k dispozici manuál, rychlý tutoriál a hlavně terminál pro příkazy (dotazy) v jazyce Cipher.

Jazyk Cipher je deklarativního typu. Jeho syntaxe byla pro větší intuitivnost inspirována jazykem relačních databází SQL. V tomto jazyce se nezadávají příkazy způsobem „jak chci hledat“, ale „co chci najít“.

Vytvoření vrcholu se provádí pomocí následujícího příkazu.

```
CREATE (ee:Person { name: "Emil", from: "Sweden" })
```

Tímto příkazem je pomocí jazyka Cipher v databázi vytvořen vrchol označený jako „Person“. Tento vrchol má vlastnosti `name` s hodnotou `Emil` a `from` s hodnotou `Sweden`.

Pro vytvoření hrany je potřeba zavolat příkaz níže, jehož součástí jsou dva dotazy.

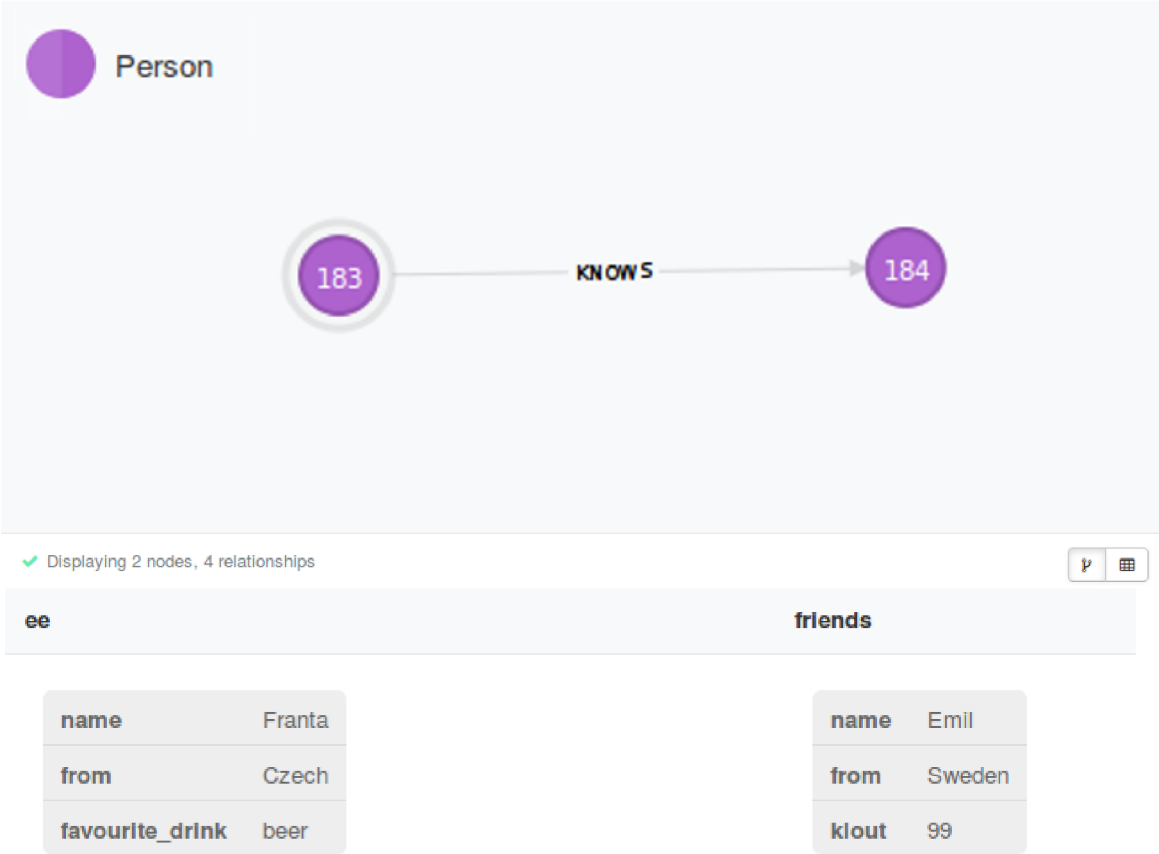
```
MATCH (ee:Person) WHERE ee.name = "Emil"
MATCH (ef:Person) WHERE ef.name = "Franta"
CREATE (ee)-[:KNOWS {since:1989}]->(ef)
```

Zmíněný dotaz provádí vyhledání vrcholu s vlastností `name`, která má hodnotu `Emil`. Tento vrchol je uložen do proměnné `ee`. Podobně je do proměnné `ef` vložen vrchol s vlastností `name`, který má hodnotu `Franta`. Následně se vytváří vztah s typem `KNOWS` a vlastností `since` s hodnotou `1989`.

Dotaz na Frantu a jeho známé se potom provádí následovně.

```
MATCH (ee:Person)-[:KNOWS]-(friends)
WHERE ee.name = "Franta" RETURN ee, friends
```

Díky tomuto dotazu je možné vyhledat všechny uzly typu `Person`, které mají vztah s uzlem, jež má vlastnost `name` s hodnotou `Franta`. Dalším specifikem musí být typ vztahu `KNOWS`. Dále se vykreslí příp. vypíše výsledek příkazu, který je možné vidět na obrázcích 2.7.



Obrázek 2.7: Výsledek dotazu Neo4J v grafu nebo v tabulkovém vyjádření.

Java API

Pro vytvoření databáze Neo4J s využitím Java API je potřeba instanciovat třídu `GraphDatabaseService`. Vytvoření databáze tak i spustí databázový server. Při instanciaci je potřeba zadat parametr, který značí adresu v souborovém systému k úložišti. Na následujícím příkladu lze vidět definici adresy takového úložiště, spuštění databázového serveru a případné vytvoření databáze. Příkazem `shutdown()` se databázový server zase ukončí.

```
private static final String DB_PATH = "/home/uziv/Plocha/test-data";

GraphDatabaseService graphDb;
graphDb = new GraphDatabaseFactory().newEmbeddedDatabase( DB_PATH );

graphDb.shutdown();
```

Veškerá manipulace s databází probíhá v transakcích a proto je potřeba po každém bloku příkazů pracujících s databází zadat speciální příkaz pro potvrzení. Navíc je nutné tuto manipulaci s ukončovacím potvrzením uzavřít do klauzule pro odchytávání výjimek kvůli možným chybám.

```
try ( Transaction tx = graphDb.beginTx() )
{
    // Database operations go here
    tx.success();
}
```

Vytvoření vrcholu a hrany je potom vidět až na komplexnějším příkladu. Dále je přiložen funkční jednoduchý příklad, který vytvoří dva vrcholy a mezi nimi jednu hranu. Nakonec příkladový program vyhledá vrchol, který má vlastnost `name` nastavenou na hodnotu `Karel` a vypíše všechny vrcholy, které jsou s ním ve vztahu `KNOWS`.

```
import org.neo4j.graphdb.*;
import org.neo4j.graphdb.factory.GraphDatabaseFactory;

public class Main {

 private static enum RelTypes implements RelationshipType {
     KNOWS,
 }

 public static void main(String[] args) {

  GraphDatabaseService graphDb;
  Node firstNode;
```

```
  Node secondNode;
  String DB_PATH = "/tmp/neo4j/file";
  graphDb = new GraphDatabaseFactory().newEmbeddedDatabase( DB_PATH );
  Label label = DynamicLabel.label( "User" );

  try ( Transaction tx = graphDb.beginTx() )  {
    //create data
    firstNode = graphDb.createNode(label);
    firstNode.setProperty( "name", "Karel" );
    secondNode = graphDb.createNode(label);
    secondNode.setProperty( "name", "Josef" );
    firstNode.createRelationshipTo(secondNode,RelTypes.KNOWS);

    // Database operations go here
    tx.success();
  }

  // ...

  try ( Transaction tx = graphDb.beginTx() )  {
    // select data
    for(Node a: graphDb.findNodesByLabelAndProperty(label,"name","Karel")){
      for (Relationship r : a.getRelationships(RelTypes.KNOWS)){
        System.out.println(r.getOtherNode(a).getProperty("name"));
      }
    }

    tx.success();
  }

  graphDb.shutdown();
 }
}
```

2.2.2 Titan

Tato knihovna pro grafové databáze je implementována v jazyce Clojure. Jazyk Clojure je dialekt jazyka Lisp. Domovská stránka knihovny Titan je na adrese [24]. Titan je šířený jako open source projekt a spadá pod licenci Apache 2.0. Grafová knihovna Titan umožňuje použití několika úložných back-endů. Lze zde využít databáze Apache Cassandra, Apache HBase nebo Oracle BerkeleyDB.

Pro ovládání databáze vytvořené pomocí knihovny Titan neexistuje speciální databázový jazyk, jako je tomu u Neo4J. Databázi můžeme ovládat pomocí Blueprints nebo pomocí přiložené aplikace, která využívá grafového dotazovacího jazyka Gremlin. Blueprints je uváděno jako generické java API pro grafové databáze. Lze tak tímto API ovládat různé databáze (Neo4J, OrientDB, Titan a další) [9].

Gremlin

Jestliže se nacházíme v adresáři aplikace, potom se zavoláním následujícího příkazu spustí terminál Gremlin, ve kterém je již možné spouštět dotazy a příkazy ve stejnojmenném dotazovacím jazyce.

```
./bin/gremlin.sh
```

Dále, pro připojení k databázi a případnému vytvoření databáze, je nutné zadat adresu ke konfiguračnímu souboru. Příkazy pro připojení databáze a k ukončení připojení jsou následující.

```
g=TitanFactory.open('/directory/file')
g.shutdown()
```

Podrobný popis tvorby obsahu konfiguračního souboru, nutného pro vytoření databáze, lze nalézt v dokumentu `titan-cassandra-es.properties`, který se nachází v adresáři `conf`. Jednoduchým příkladem funkční konfigurace může být následující text.

```
storage.backend=berkeleyje
storage.directory=db/berkeley
index.search.backend=elasticsearch
index.search.directory=db/es
index.search.elasticsearch.client-only=false
index.search.elasticsearch.local-mode=true
```

Klíčovou informací z konfiguračního souboru je použití úložného back-endu, který je v tomto případě BerkleyDB (berkeleyje).

Pro vytvoření vrcholu se použije funkce `addVertex()`, která vrací vytvořený vrchol. Následně se pomocí funkce `setProperty()` nastaví vrcholu požadované vlastnosti.

```
prom1 = g.addVertex()
prom1.setProperty('name','Jirka')
```

K vytvoření hrany se potom použije funkce `addEdge()`, kde se spojí například vrchol `prom1` a `prom2` a nastaví se typ resp. popisek. Dále je samozřejmě možné dát hraně další vlastnosti podobně jako vrcholu.

```
g.addEdge(null,prom1,prom2,'KNOWS')
```

Jednoduchý dotaz může vypadat následovně. Zde se dotazuje grafové databáze na vrchol, který má vlastnost `name` s hodnotou „Jirka“.

```
g.V.has('name','Jirka')
```

Blueprints API

Aplikaci Gremlin a toto generické API vyvíjí stejná skupina lidí (TinkerPop). Díky tomu je použití Blueprints v Javě velice podobné příkazům v aplikaci Gremlin. Zpracování databáze je zde transakční, proto je vhodné operace nad databází uzavírat do klauzule pro odchytávání výjimek. Po ukončení bloku příkazů je nutné jej zakončit příkazem `shutdown()`.

Za zmínku už zde stojí asi jen fakt, že proměnná `g`, která se používala i v aplikaci Gremlin (zde má stejný význam), je datového typu `TitanGraph`. Vrcholy vytvořené funkcí `addVertex()` jsou datového typu `Vertex` a hrany jsou typu `Edge`.

```
import com.thinkaurelius.titan.core.TitanFactory;
import com.thinkaurelius.titan.core.TitanGraph;
import com.tinkerpop.blueprints.*;

public class test {

 public static void main(String[] args) {

  TitanGraph g = TitanFactory.open("/tmp/titan/file");

  Vertex oz = g.addVertex(null);
  oz.setProperty("name", "Jiri Bilek");
  oz.setProperty("age", "24");

  Vertex sz = g.addVertex(null);
  sz.setProperty("name", "Tatka Bilek");

  Edge e = g.addEdge( null, sz, oz, "parent");

  g.commit();

  if (g.getVertices("name", (Object)"Jiri Bilek").iterator().hasNext()){
   System.out.println("Jiri Bilek is "+g.getVertices("name",
(Object)"Jiri Bilek").iterator().next().getProperty("age")+" years old");
  }else{
   System.out.println("Zadny Jiri Bilek v databazi neni");
  }
  g.shutdown();
 }
}
```

2.3 Dekompilátory Java

Žádné nástroje pro tvorbu abstraktního syntaktického stromu z byte kódu nejsou k dispozici. Lze však využít dekompilátor, který z byte kódu vytváří přímo zdrojový kód. Pokud by tento nástroj byl s otevřeným kódem (open source), byla by možnost si nástroj upravit pro pouhou tvorbu AST nebo nástroj použít tak, aby tento AST vracel. Z toho vyplývá první nárok na dekompilační nástroj. Dalším nárokem je implementační jazyk, neboť potřebujeme nástroj modifikovat resp. používat v programovacím jazyce Java.

Dekompilátorů Java kódu je poměrně velké množství. Mnohé již ale nejsou v současné době vyvíjeny a další postrádají kvalitu. V následujícím přehledu je charakterizována většina známějších dekompilátorů. Obsáhlejší seznam dekompilátorů lze nalézt na blogu [36].

Je zřejmé, že je k dispozici velké množství dekompilátorů pro Java bajtkód. Avšak většina nástrojů neodpovídá požadavkům této práce a další jsou buď dosud nedokončené (CRF), nebo jejich nedokončený vývoj skončil (Candle, EDJC). Jediným odpovídajícím použitelným nástrojem je tedy Procyon. Tomuto nástroji se budu dále věnovat podrobněji.

2.3.1 Procyon

Balík Procyon je open source projekt vyvíjený pod licencí Apache 2.0. Vývoj započal koncem roku 2012 hlavním vývojářem Mikem Strobelem. Domovské stránky s možností stažení zdrojových kódů, drobným popisem a návodem lze nalézt na adrese [35].

Procyon je balík nástrojů pro generování a analýzu zdrojového kódu. Hlavní části (knihovny) jsou Core framework, Reflection framework, Expression framework, Compiler toolset a Java decompiler.

Je nutno podotknout, že dekompilace je poměrně náročná operace a vyžaduje hodně výpočetního času procesoru. Dekompilace velkého projektu může na průměrném stroji trvat několik jednotek až desítek minut.

Použití

Pokud je stažen již zkompilovaný `procyon-decompiler.jar`, lze si spuštěním tohoto programu vypsat nápovědu.

```
java -jar procyon-decompiler.jar -?
```

Nebo je možné se rovnou pokusit dekompilovat nějaký soubor. Pro otestování funkčnosti lze vyzkoušet dekompilovat standardní knihovny Javy jako například `java.lang.String`. Soubor pro dekompilaci se zadává jako parametr programu.

```
java -jar procyon-decompiler.jar java.lang.String
```

V případě použití dekompilátoru v terminálech, které neposkytují obarvení textu (vyznačení syntaxe) bude uživatel značně ochuzen. Proto se doporučují terminály s možností zabarvování textu a pokud přesto detekce ANSI selže, je možné toto vynutit přidáním argumentu `-DAnsi=true`.

```
java -DAnsi=true -jar procyon-decompiler.jar java.lang.String
```

Název	Licence	Poznámky
Procyon [35]	Apache License 2.0	Nový, poměrně kvalitní dekompilátor, který je stále vyvíjen.
CRF (Class File Reader) [11]	MIT License	Vytvořen v Java 6. Podporuje Java 6, Java 7 i Java 8. Stále vyvíjen. Občas ještě vypisuje kontrolní výpisy.
Krakatau [21]	GNU GPL	Stále vyvíjen. Vytvořen v jazyce Python (Python 2.7). Částečná podpora Java 8.
Candle [10]		Dekompilátor programovaný v Javě. Vývoj pravděpodobně skončil v roce 2013. Přestože je vývoj opřen o firmu Red Hat, není dekompilátor úplně dokončený.
JBVD (Java Bytecode Viewer & Decompiler) [19]	Academic Free License (AFL)	Vývoj skončil v roce 2011. Přestože označovaný jako open source, nepodařilo se mi na internetu najít zdrojový kód.
EDJC (Emilio's Java Decompiler) [13]	GNU GPL	Dekompilátor vytvářen v jazyce Java. Vývoj skončil alfa verzí v roce 2011.
JD [20]	MIT License, uzavřený kód	Součásti JD-Code a JD-Gui programovány v jazyce C++. Podpora jdk1.1.8, jdk1.3.1, jdk1.4.2, jdk1.5.0, jdk1.6.0 a jdk1.7.0. Tento dekompilátor poskytuje plugin k vývojovému prostředí Eclipse.
FernFlower [14]	Freeware License 1.0	Vytvořeno v jazyce Java. K projektu není žádná dokumentace ani bližší informace.
JaD (JAva Decompiler) [18]	Komerční program (pro nekomerční použití zdarma)	Dekompiler podporující dekompilaci Java 7. K dispozici je také plugin pro vývojové prostředí Eclipse s názvem JadClipse.
DJ [12]	Nekomerční použití zdarma	Určený pro operační systém Windows. Součástí je i editor.
Mocha [22]		V roce 1996 vydána betaverze, poté vývoj skončil.

Tabulka 2.2: Seznam dostupných dekompilátorů Java kódu.

Balík Procyon umožňuje také dekompilaci celých `.jar` souborů. K tomu je potřeba přidat dekompilátoru argument `-jar`, oznamující typ příchozího souboru a dále je možné si výsledek uložit do vybraného umístění.

```
java -jar procyon-decompiler.jar -jar soub.jar -o out
```

API Dekompilátoru

Dále je uveden způsob použití balíku Procyon v kódu, kde je totiž potřeba API dekompilátoru, které však obsahuje pouze dvě metody.

```
public static void decompile(
    final String internalName,
    final ITextOutput output
);

public static void decompile(
    final String internalName,
    final ITextOutput output
    final DecompilerSettings settings
);
```

Prvním parametrem funkce `decompile` je proměnná typu `String`, která požaduje v textovém tvaru identifikaci souboru pro dekompilaci. Identifikace může být definována buď adresou v souborovém systému, nebo adresou v Java notaci k standardní knihovně (např. `java.lang.String`). Druhý parametr je výstup metody v datovém typu `ITextOutput`, který je definován v rámci balíku Procyon. Poslední parametr je volitelný a je jím možno nastavit některé parametry dekompilace. V následujícím příkladu lze vidět jednoduché použití této funkce s výstupem na terminál.

```
final DecompilerSettings settings = DecompilerSettings.javaDefaults();

try (final FileOutputStream stream = new FileOutputStream("path/to/file");
     final OutputStreamWriter writer = new OutputStreamWriter(stream)) {

    Decompiler.decompile(
        "java/lang/String",
        new PlainTextOutput(writer),
        settings
    );
}
catch (final IOException e) {
    // handle error
}
```

2.3.2 BCEL

Zkratka BCEL značí slovní spojení Byte Code Engeneering Library [1], tedy se jedná o knihovnu pro práci s bajtkódem. Tato knihovna je vyvíjena společností The Apache Software Foundation a taktéž spadá pod licenci Apache 2.0.

Balík BCEL není určený pro plnou dekompilaci kódu. Jeho účel je majoritně zaměřený na modifikaci již zkompilovaného kódu (bajtkódu). Při zpracování však dává k dispozici tyto informace:

- název třídy,
- název souboru,
- název balíku,
- název rodičovské třídy,
- příznaky třídy,
- názvy implementovaných rozhraní,
- seznam atributů třídy (název, datový typ),
- seznam metod třídy (název, datový typ návratové hodnoty, datové typy parametrů).

Nevýhodou tohoto balíku je omezené množství poskytovaných informací a možnost zpracování pouze souborů `.class`, tedy nikoliv `.jar`. Na druhou stranu je doba zpracování velice krátká (jednotky sekund) na rozdíl od dekompilace pomocí balíku Procyon.

Kapitola 3

Návrh dotazovacího jazyka

S využitím knihovny BCEL bude vytvořen abstraktní syntaktický strom s dostupnými informacemi. Pokud bude požadavek na informace, které knihovna BCEL nebude poskytovat, provede se sestavení AST pomocí nástroje Procyon. Detekce požadovaných informací se provede během načítání dotazu, aby nedošlo k situaci, kdy by byl prvně vytvořen AST knihovnou BCEL a následně nástrojem Procyon. Tímto by mělo dojít ke zefektivnění interpretace dotazu. Dále se tento získaný AST uloží do grafové databáze.

Dle porovnání grafových databází (tabulka 2.1) bude v aplikaci využita knihovna pro grafové databáze Neo4J. Tato knihovna byla vybrána zejména díky své rychlosti spouštění, protože s každým dotazem se bude aplikace spouštět znovu společně s databázovým serverem. Třídy a rozhraní potom budou znázorněny jako vrcholy grafu. Vlastnosti tohoto vrcholu budou zastupovat informace o třídě jako např. modifikátory, přístupová práva apod. Hrany mezi vrcholy budou popisovat vztahy mezi třídami a rozhraními, tedy specifikaci, realizaci a asociaci. Druh vztahu bude specifikován typem vztahu. Metody a atributy tříd budou také znázorněny jako vrcholy grafu a s třídami budou mít vytvořen vztah s patřičným typem.

Dále se nad vzniklou databází provede dotaz, který bude definován pomocí speciálního dotazovacího jazyka. Tento jazyk je popsán v následující podkapitole.

3.1 Návrh dotazovacího jazyka nad Java AST

Aby byl dotazovací jazyk nad Java AST více intuitivní, bude jeho syntaxe odvozena od jiného známého a již používaného jazyka. Za tento inspirativní jazyk byl vybrán jazyk XPath. Více se o jazyku XPath lze dočíst na internetových stránkách [6, 27].

Uložení dat v databázi

Na základě porovnání databází z článku na portálu db-engines [2] bude k uložení dat použita databáze Neo4J. Vzhledem k opětovné použitelnosti nebude použito Java API poskytovaného knihovnou Neo4J, ale obecnější knihovnou Blueprints API, která dokáže ovládat i databázi vytvořenou pomocí Neo4J. Do budoucna tak bude možné jednoduše nahradit Neo4J jinou efektivnější databází.

Syntaxe a sémantika dotazu

Základní struktura dotazu se skládá z jednotlivých kroků. Podobně jako v jazyce XPath lze jednotlivé kroky dotazu vkládat mezi lomítka („/“). Není zde však kořenový uzel (vrchol), takže adresování musí být vždy relativní tzn. nikdy nebude dotaz začínat symbolem „/“.

```
krok/krok/...
```

Každý krok musí obsahovat výraz pro popis uzlu (nodeexpr). Nepovinnou částí kroku je filtr (filter).

```
nodeexpr[filter]
```

Obsahem popisu uzlu může být konkrétní jméno hledaného elementu nebo typ uzlu (třída, rozhraní apod.). Výsledkem takovýchto jednoduchých dotazů by potom byla množina všech uzlů s odpovídajícím jménem nebo množina uzlů se zadaným typem. Před jméno elementu lze přidat symbol „!“, který značí, že je požadována množina prvků, jehož jméno bylo zadáno, a všech jeho potomků. Jestliže se nejedná o první krok, může být obsahem také název vztahu. Potom by byla výsledkem množina uzlů, které jsou v patřičném vztahu s uzly z předchozího kroku. Jestliže bude jméno uzlu nebo typ uzlu zadán jako krok, který není na prvním místě, bude tento krok brán jako filtr. Například pokud bude v prvním kroku vyhledána množina tříd a ve druhém bude název elementu, potom se množina všech tříd zredukuje na množinu tříd, které mají definovaný název. V tomto případě by opačné pořadí kroků vedlo k ekvivalentnímu dotazu.

```
class/jmeno.tridy
jmeno.tridy/class
```

Filtr kroku je nepovinná část. Pokud je však zadán, je výsledkem taková množina elementů, které odpovídají filtru. Struktura filtru je podobná struktuře dotazu, tedy se skládá z kroků, popisu uzlu a dalších filtrů. Navíc je zde možné zadat podmínku na konkrétní vlastnost uzlu.

```
@property="value"
```

Výsledkem filtru je redukovaná množina ze vstupu. Je-li tedy filtrem podmínka vlastnosti uzlu, výsledkem je množina takových uzlů, které mají pro danou vlastnost nastavenou odpovídající hodnotu. Je-li filtr zapsán stejnou strukturou jako dotaz, potom je výsledkem taková množina uzlů, které po provedení filtru neměly jako výsledek prázdnou množinu.

Příkladem filtru s podmínkou vlastnosti uzlu může být následující dotaz. Zde je získána množina všech tříd, kterou filtr zredukuje na ty třídy, které jsou statické.

```
class[@static="true"]
```

Příklad filtru se strukturou dotazu může vypadat následovně. Výsledkem jsou tak všechny třídy, které obsahují alespoň jednu metodu.

```
class[have_method]
```

V dotazovacím jazyce je možné používat závorky. Jejich funkce je stejná jako v jiných jazycích.

Dále je možné v dotazu použít množinových operátorů `intersection` a `union`. Tyto operátory lze použít na různé kroky dotazu, na samostatné dotazy nebo v krocích, které jsou uvnitř filtru.

```
class union interface
class intersection inner class
class/(have_method union have_member)
```

Ve filtru lze použít navíc pravdivostní operace `and`, `or` a `not`. Je zde tedy nutno rozlišovat, zda se jedná o množinovou operaci nebo pravdivostní operaci.

```
class[have_method or have_member]
class[@access="public" or @access="protected"]
class[have_method and @access="public"]
class[extends/(have_member union have_method)]
```

Zkracování zápisu je možné podobně jako v jazyce XPath, avšak zkratky budou implementovány pro často používané konstrukce relevantní tomuto dotazovacímu jazyku. Výsledný seznam všech zkratek v jazyce ASTquery je vypsán do tabulky v příloze E.

V rámci této práce byl vytvořen seznam 30 komplexních dotazů, který ukazuje různé možnosti dotazovacího jazyka. Proto v něm není tolik kladen důraz na smysluplnost dotazu, nýbrž na různorodost. Každému dotazu předchází popis, co se v daném případě vyhledává. V některých případech je vypsáno několik ekvivalentních dotazů s rozdílným zápisem. Tento rozdílný zápis ekvivalentních dotazů může vést k rozdílnému postupu vyhodnocení a tedy rozdílným časům zpracování výsledku. Což může být zajímavým podnětem k experimentům. Zmiňovaný seznam se lze nalézt v příloze F.

Kapitola 4

Implementace dotazovacího jazyka

Při tvorbě gramatiky byl využit nástroj ANTLR. Díky tomuto nástroji lze lehce vygenerovat překladač jazyka. Práce s ANTLR je blíže popsána v následující podkapitole. Dále je popsána syntaxe a sémantika jazyka pomocí speciálního popisu jazyka, který se používá na vstup nástroje ANTLR. Jsou tak popsány především nejvýznamnější části jazyka. Závěrem je podkapitola věnována použití zkratek, které lze v Java AST dotazovacím jazyce používat.

4.1 Knihovna ANTLR

Pro pomoc při vytváření překladače dotazovacího jazyka byl vybrán nástroj ANTLR. Jedná se o generátor parseru pro čtení, zpracování, provádění nebo překládání strukturovaného textu či binárních souborů [8, 32]. Lze tak jednoduše získat abstraktní syntaktický strom dotazu.

Prerekvizity

Za prvé je potřeba mít vytvořený soubor s gramatikou, která bude mít koncovku `.g` nebo `.g4`. V tomto případě to bude soubor `ASTquery.g`. Pro jednoduchost ukázky může mít tento soubor následující obsah (převzato z [33]). Finální podoba souboru je v příloze D, jejíž obsah je popsán v následující kapitole.

```
grammar ASTquery;
r  : 'hello' ID ;// match keyword hello followed by an identifier
ID : [a-z]+ ;    // match lower-case identifiers
WS : [ \t\r\n]+ -> skip; // skip spaces, tabs, newlines
```

Další nezbytnou součástí je samozřejmě program ANTLR, který je možné stáhnout na adrese http://www.ANTLR.org/download.html (Complete ANTLR 4.5 Java binaries jar).

Vytvoření potřebných tříd z gramatiky

Jestliže máme soubor s gramatikou (`ASTquery.g`) a ANTLR (`ANTLR-4.5-complete.jar`) v jedné složce, můžeme spustit následující příkaz pro vygenerování tříd zpracovávající gramatiku (parser).

```
java -classpath ANTLR-4.5-complete.jar org.ANTLR.v4.Tool ASTquery.g
```

Tímto příkazem budou ve stejné složce vygenerovány soubory:

- ASTqueryParser.java,
- ASTqueryLexer.java,
- ASTquery.tokens,
- ASTqueryBaseListener.java,
- ASTqueryLexer.tokens,
- ASTqueryListener.java.

Testování gramatiky

Kompilace zdrojových souborů parseru a lexikálního analyzátoru se provádí s připojením potřebné knihovny ANTLR.

```
javac -cp ANTLR-4.5-complete.jar *.java
```

K následnému spuštění je potřeba navíc přiložit soubor obsahující funkci `main`, která s využitím tzv. „class loaderu“ načte zkompilované třídy zpracovávající jazyk. Dále je potřeba určit jméno gramatiky, počáteční uzel a způsob výpisu výsledného AST nebo požádat o výpis dalších informací.

```
java -cp ANTLR-4.5-complete.jar:. org.ANTLR.v4.runtime.misc.TestRig
ASTquery -r -tree
```

Po tomto příkazu bude terminál očekávat zápis v zadané gramatice. Ukončení vstupu lze provést stiskem Ctrl+D a na požadovaném výstupu bude výsledný syntaktický strom.

Spuštěním bez parametrů získáme nápovědu testovacího programu, jež ukazuje možné výstupy programu (tokeny, abstraktní syntaktický strom, AST zobrazený pomocí GUI a další).

```
$ java -cp ANTLR-4.5-complete.jar:. org.ANTLR.v4.runtime.misc.TestRig
java org.ANTLR.v4.runtime.misc.TestRig GrammarName startRuleName
  [-tokens] [-tree] [-gui] [-ps file.ps] [-encoding encodingname]
  [-trace] [-diagnostics] [-SLL]
  [input-filename(s)]
Use startRuleName='tokens' if GrammarName is a lexer grammar.
Omitting input-filename makes rig read from stdin.
```

V tomto projektu však nebude „class loader“ používán. Vygenerovaným zdrojovým souborům zpracovávajícím jazyk je připsána deklarace balíku (`package grammar;`). Ve třídách, kde jsou tyto třídy používány, jsou importovány klasickým způsobem. Tímto odpadá potřeba načítání tříd pomocí ANTLR.

4.2 Syntaxe a sémantika dotazovacího jazyka

Stručný popis v návrhu dotazovacího jazyka (kapitola 3.1) byl dodržen ve všech bodech. Podrobnější popis lze získat studováním obsahu souboru s gramatikou (v tomto případě se jedná o soubor `ASTquery.g`) pro nástroj ANTLR. Níže jsou rozebírány nejdůležitější části gramatiky. Kompletní obsah souboru s gramatikou je uveden v příloze D.

Počáteční pravidlo gramatiky je nazváno `query`. Obsah tohoto pravidla je stejný jako pravidla `query2` bez neterminálu `relationship`. Z tohoto pravidla lze přejít do neterminálu `node_type`, což znamená, že dotaz může obsahovat název typu uzlu (`class`, `method` a další). Dále je možné zadat přímo název hledaného elementu (`ID`). Toto si interpret vysvětlí, jako jméno uzlu jehož typ je třída, vnitřní třída, rozhraní, datový typ, generický typ, pole nebo výčet. Nepovinnou součástí je symbol vykřičníku, který by po zadání značil, že je požadavkem nejen ten uzel, jehož název byl zadán, ale také jeho potomci. Další pravidlo obsahující neterminál `slash` značí vlastnost jazyka, kde se jednotlivé kroky dotazu oddělují symbolem lomítko „/“. V pravidle `query` je tento zápis mírně rozdílný než v `query2` a to takto: `query slash query2`. Z tohoto je následně zřejmé, jak lze přejít z pravidla `query` do pravidla `query2`. Je tímto zamezeno možnosti, aby se neterminál `relationship` mohl vyskytnout v prvním kroku. Další pravidlo `query2 '[' expr ']'` značí zápis filtru a ukazuje možný přechod do pravidla `expr`, který je popsán níže. Dále už jsou jen pravidla pro množinové operace sjednocení a průniku a na závěr je pravidlo pro závorky.

Pravidlo `query2` má navíc, jak již bylo řečeno, k dispozici neterminál `relationship`, díky kterému je možné zadat vztah. Pokud je takto zadán vztah, je výsledkem množina elementů, které mají zadaný vztah s prvky, jež byly ve výsledné množině po provedení předchozího kroku.

```
query2
  : node_type
  | ('!')? ID
  | query2 slash query2
  | relationship
  | query2 '[' expr ']'
  | query2 SPACES union SPACES query2
  | query2 SPACES intersection SPACES query2
  | '(' query2 ')'
  ;
```

Rozdílů mezi pravidly `expr` a `expr2` je již více, než mezi `query` a `query2`. Možnost zadání názvu typu uzlu, konkrétního jména uzlu s případným vykřičníkem, pravidla pro oddělení kroků (`slash`), pravidla pro vložení filtru a pravidla pro závorky zůstalo nezměněné. Pravidlo `expr2` dále umožňuje operace pro sjednocení a průnik, jako tomu bylo u pravidel `query` a `query2`.

```
expr2
  : node_type
  | '!' ID
  | ID
  | expr2 slash expr2
  | relationship
  | expr2 '[' expr2 ']'
```

```
  | expr2 SPACES union SPACES expr2
  | expr2 SPACES intersection SPACES expr2
  | '(' expr2 ')'
  ;
```

U obou pravidel je možné zadat název vztahu. V případě `expr` nebude tak brána vstupní množina z předchozího kroku, ale z množiny vstupující do filtru. Dále v případě tohoto pravidla jsou poskytována pravidla pro pravdivostní operace negace, `and` a `or`. Poslední přepisovací pravidlo z neterminálu `expr` značí podmínku na vlastnost uzlu.

```
expr
  : node_type
  | '!' ID
  | ID
  | expr slash expr2
  | relationship
  | expr '[' expr ']'
  | not SPACES expr
  | expr SPACES and SPACES expr
  | expr SPACES or SPACES expr
  | '@' ID '=' STRING
  | '(' expr ')'
  ;
```

Neterminál `node_type` umožňuje zadání pouze vymezeného výčtu prvků, který je definovaný v rámci gramatiky. Jedná se o všechny typy uzlů vytvářených v grafové databázi. Přepisovací pravidlo neterminálu `ID` značí, že se zde může vyskytnout libovolný řetězec, který podléhá podmínkám identifikátorů jazyka Java. Nesmí tedy být obsahem symbol dolaru (`$`), lomítka (`/`) a podobně. Navíc byla přidána možnost zadání symbolu tečka („.“), který sice nemůže v identifikátorech být obsažen, ale je obsažen v celých jménech identifikátorů (např. `java.lang.String`). Neterminál `STRING` umožňuje zadání libovolného řetězce, který musí být ohraničen uvozovkami. Zde je tedy možno zadat řetězce i s nepovolenými znaky v identifikátorech. Závěrem je vhodné podotknout, že vzhledem tomu, že jazyk rozlišuje malá a velká písmena, byly textové operátory definovány tak, aby nebyl brán zřetel na velikost znaků v klíčových slovech. Jedná se o operátory `not`, `and`, `or`, `union` a `intersection`.

4.3 Zkratky

Veškeré možné zkratky, které lze použít v dotazovacím jazyce nad Java AST jsou vypsány v příloze E.

Mezi nejvýznamnější patří následující zkrácení, což lze samozřejmě provést pro jakýkoliv typ vrcholu.

```
class[@name="jmeno.elementu"]
class[jmeno.elementu]
```

Dalším významným zkrácením je zápis pro získání množiny elementů a jeho potomků, kde tento kořenový element definujeme názvem.

```
jmeno.elem/>e union jmeno.elem
!jmeno.elem
```

Všechny vztahy mezi vrcholy mají definovaný určitý název (např. `extends`), který je používán i v dotazovacím jazyce. Pro opačný směr vztahu se v grafu používá konstrukce pro otočení směru „šipky“. V dotazovacím jazyce jsou pro opačné směry vztahů definovány speciální názvy (např: `extended_by`). Zjednodušení spočívá v tom, že není nutno si pamatovat názvy pro oba směry vztahu, ale stačí zadat obecný název a na začátku určit symboly $>$ a $<$ pro určení směru vztahu. Potom jsou následující zápisy ekvivalentní.

```
class/extends
class/>extends
```

```
class/extended_by
class/<extends
```

Dále pro vybrané (nejvíce používané) vztahy, lze zapsat název vztahu pouze prvním písmenem.

```
class/extends
class/>e
```

Poslední jazyková konstrukce, která by se dala také označit jako zkratka, je přímé napsání názvu vyhledávaného elementu. Tedy například zápis `org.package.class` vrátí množinu všech tříd, vnitřních tříd, datových typů (dočasných vrcholů tříd), výčtů, generických typů, polí a rozhraní, které mají shodný název. Je zjevné, že nezkrácený zápis by v tomto případě byl opravdu dlouhý.

Kapitola 5

Uložení abstraktního syntaktického stromu do grafu

V této kapitole je podrobně probrán způsob uložení abstraktního syntaktického stromu do grafu. Jelikož je pro práci s databází Neo4J používáno Blueprints API, je možnost využít další podpůrné knihovny z dílny Blueprints. Knihovna Frames umožňující jednodušší a čitelnější ukládání dat do databáze je přiblížena následující podkapitolou. Dále jsou řešeny jednotlivé elementy jazyka Java, které jsou určeny k ukládání do grafové databáze. U každého elementu je studována syntaxe a sémantika zápisu, přičemž je diskutováno řešení uložení do databáze. Dále je vždy shrnutí pro každý element, jež vypisuje seznam všech ukládaných vlastností a možností navázání vztahů. Následuje popis řešení několika význačných problémů při ukládání dat do databáze. Na závěr je shrnutí všech vrcholů do přehledné tabulky.

5.1 Knihovna Frames

Knihovna Frames umožňuje pohlížet na prvky grafové databáze jako na objekty [5]. V jazyce Java se tak vytvoří rozhraní popisující vrcholy a hrany v grafu. Potom lze jednoduše a přehledně ukládat informace do databáze nebo z databáze číst.

Díky této knihovně jsou vytvořeny rozhraní pro všechny typy vrcholů. Rozhraní pro popis hran není potřeba vytvářet. Vzhledem k podobnostem typů vrcholů a k potřebě s některými pracovat stejně, je využito dědění rozhraní a tím sdíleny vlastnosti a chování mnoha vrcholů.

Po prostudování konstrukcí jazyka Java je zjevné, že pro všechny elementy je potřeba ukládat jejich název a vztah prvku s jeho anotací (skoro všechny elementy lze anotovat). Z tohoto důvodu byl vybrán příklad na vytvoření typu vrcholu, který bude předkem všem vrcholům.

```
public interface IObject {
    @Property("name")
    public void setName(String name);
    @Property("name")
    public String getName();

    @Adjacency(label="have_annotation", direction=Direction.OUT)
    public Iterable<IObject> getAnot();
```

```
    @Adjacency(label="have_annotation", direction=Direction.OUT)
    public void addAnot(final IObject a);
}
```

V kódu je ukázáno, že jména vlastností se vkládají jako element anotace `Property` před metody, které s danou vlastností pracují. Zde jsou deklarovány metody pro nastavení hodnoty této vlastnosti a pro čtení této vlastnosti. Obdobně se pracuje se vztahy objektů, kde se název vztahu vkládá jako element anotace `Adjacency`. Dalším elementem této anotace je orientace vztahu. Tedy vztah s názvem `have_annotation`, který vždy vychází z daného vrcholu, má deklarované opět dvě metody. První metoda vrací seznam všech vrcholů, které mají s daným vrcholem zmiňovaný vztah. Druhá metoda umožňuje přidat vrcholu požadovaný vztah.

Při vytváření nového vrcholu, který má být definovaného typu, se zadá reference na rozhraní jako parametr funkce `addVertex`.

```
IObject objct = (IObject)framedGraph.addVertex(1, IObject.class);
```

Práce s vrcholem se dále provádí pomocí deklarovaných metod v rozhraní.

```
objct.setName("novy objekt");

IObject anotace = (IObject)framedGraph.addVertex(1, IObject.class);
objct.addAnot(anotace);
```

5.2 Třída

Podle popisu třídy v dokumentaci Java 7 [26] definuje deklarace třídy nový referenční typ a popisuje, jak je implementována.

Existují třídy tzv. „top level“ a naopak vnořené třídy. Vnořené třídy jsou deklarované uvnitř jiné třídy nebo rozhraní. Mezi vnořené třídy se řadí členské třídy, lokální třídy a anonymní třídy. V grafu disponují vnitřní třídy speciálním vrcholem, kde anonymní třídy jsou odlišeny jiným druhem vrcholu.

Třída může obsahovat atributy, metody, instance, statickou inicializaci a konstruktory. V grafu mají třídy vztah s atributy (`have_attribute`) a metodami (`have_method`). S konstruktory je zacházeno jako s metodami s tím rozdílem, že vrcholy jsou odlišeny typem.

Popis deklarace třídy je následující.

```
ClassDeclaration:
    NormalClassDeclaration
    EnumDeclaration
NormalClassDeclaration:
    ClassModifiers class Identifier TypeParameters
                                          Super Interfaces ClassBody
```

Z první části je zjevné, že existují dva druhy tříd. První jsou normální třídy a druhou je výčet (`enum`). Mezi normální třídou a výčtem je mnoho rozdílného, z hlediska grafu však ne, proto jsou tyto dvě entity rozlišeny pouze typem vrcholu a dále je s nimi zacházeno stejně.

Při deklaraci třídy jsou možné následující modifikátory:

- `Annotation`,
- `public`,
- `protected`,
- `private`,
- `abstract`,
- `static`,
- `final`,
- `strictfp`.

Modifikátor `Annotation` značí anotaci třídy. Tedy, že v tomto místě bude anotace, která se vztahuje na následující třídu. Tento vztah je v grafu symbolizován hranou mezi třídou a vrcholem anotace s názvem `have_annotation`. Přístupové modifikátory `public`, `protected` a `private` jsou zachyceny v grafu vlastností `access`. `Abstract`, `static` a `final` mají v grafu speciální položku mezi vlastnostmi vrcholu. `Strictfp`, který striktně vynucuje plovoucí řadovou čárku u výrazů [26], je do grafu ukládán pod stejným názvem.

S třídami, které deklarují generický typ, je zacházeno jako s ostatními třídami. Jméno třídy je použito bez typového parametru. Java kompilátor se již postará o to, aby byl typový parametr používán správně. Proto není potřeba toto v grafu ošetřovat.

Třída může dědit od rodiče, kterého může mít pouze jednoho. V grafu není na počet předků brán zřetel (o to se stará kompilátor). Vrchol zastupující třídu tak má možnost mít vztah s jinou třídou (`extends`), což je charakterizováno děděním. Dále může třída implementovat rozhraní. Je možné implementovat více rozhraní, proto je v grafu možnost navázat vztah na více vrcholů typu rozhraní.

Protože z hlediska vytváření grafu není mezi vnitřní třídou, anonymní třídou a jinou třídou rozdíl, je s nimi zacházeno stejně s tím rozdílem, že mají rozlišen typ vrcholu.

Neboť není potřeba podrobně mapovat strukturu java balíků (package), je tato informace uložena v každé třídě pouze jako vlastnost.

Souhrn

Vlastnosti:

- `name` (String),
- `static` (boolean),
- `abstract` (boolean),
- `final` (boolean),
- `anonym` (boolean),
- `inner` (boolean),
- `access` (public/private/protected),
- `package` (String),

- `strictfp` (boolean).

Vztahy:

- `extends` (Třída),
- `uses` (Třída),
- `implements` (Rozhraní),
- `have_inner` (Vnitřní třída),
- `have_anonym` (Anonymní třída),
- `have_method` (Metoda),
- `have_attribute` (Atribut),
- `have_annotation` (Anotace).

Stejný popis vrcholu má mít vnitřní třída, anonymní třída a výčet.

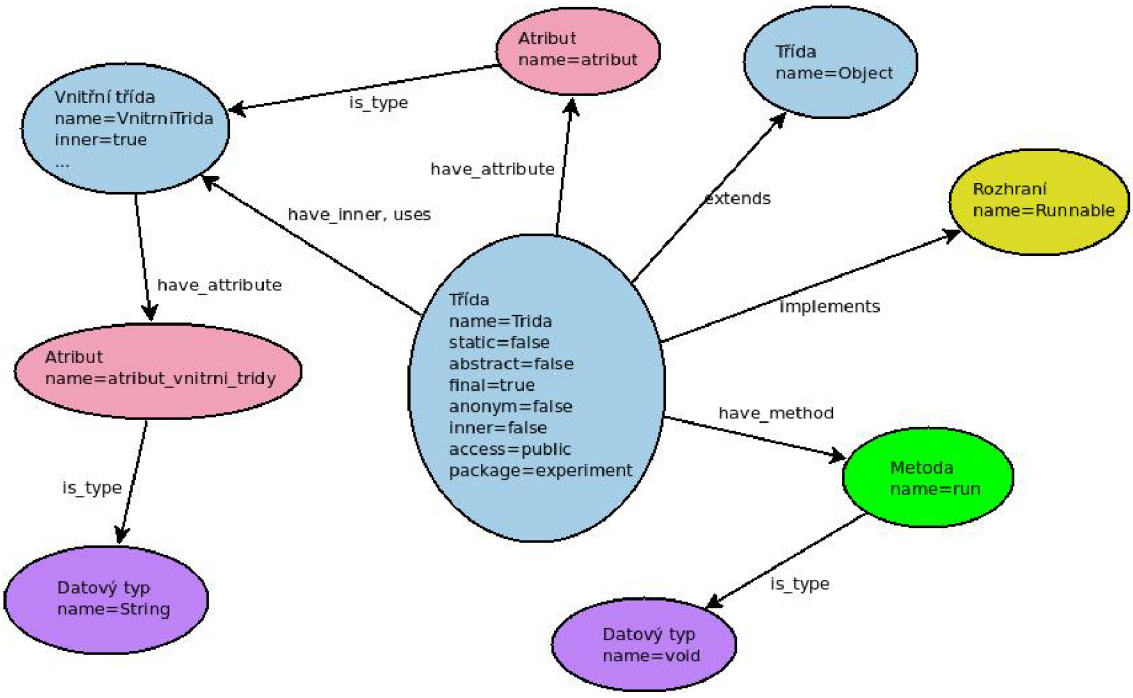
Pro příklad je vytvořen následující kód. Třída s názvem `Trida` je označena modifikátory `public` a `final`. Dále tato třída dědí od třídy `Object` a implementuje rozhraní s názvem `Runnable`. Toto rozhraní vyžaduje implementaci metody `run`. Třída dále obsahuje jeden atribut, který je navíc typu, jež je deklarován uvnitř třídy. Obsažena je tedy také vnitřní třída s názvem `VnitrniTrida`, která disponuje jedním atributem. Na obrázku 5.1 je zobrazen odpovídající graf ke zmiňovanému kódu.

```
package experiment;

public final class Trida extends Object implements Runnable{
    class VnitrniTrida{
        public String atribut_vnitrni_tridy;
    }

    private VnitrniTrida atribut = new VnitrniTrida();
    public void run() {
        // ..
    }
}
```

5.3 Rozhraní

Dokumentace Java definuje rozhraní jako nový referenční typ, který může obsahovat třídy, rozhraní, konstanty a abstraktní metody. Rozhraní neobsahuje implementaci metod, avšak implementující třídy musí tyto metody implementovat [26].

```
    InterfaceDeclaration:
        NormalInterfaceDeclaration
        AnnotationTypeDeclaration
    NormalInterfaceDeclaration:
        InterfaceModifiers interface Identifier
        TypeParameters ExtendsInterfaces InterfaceBody
```



Obrázek 5.1: Příklad grafu zaznamenávajícího třídu a jeho okolí.

Jsou k dispozici 2 druhy deklarací rozhraní. Buď lze deklarovat klasické rozhraní, nebo anotační typ. Deklaraci anotačních typů se věnuje kapitola 5.4.

Hlavička klasického rozhraní obsahuje seznam modifikátorů, klíčové slovo pro identifikaci deklarace rozhraní a identifikátor označující konkrétní rozhraní. Dále je možné přiložit typové parametry, pokud je deklarováno generické rozhraní. Následně je možnost vypsat seznam rozhraní, od kterých je potřeba dědit. Nakonec je nutno napsat tělo rozhraní.

Možné modifikátory při deklaraci rozhraní jsou:

- `Annotations`,
- `public`,
- `protected`,
- `private`,
- `abstract`,
- `static`,
- `strictfp`.

Rozhraní má také možnost být anotováno. Toto značí modifikátor `Annotations` a v grafu je anotace rozhraní interpretována vztahem s názvem `have_annotation`, který je mezi vrcholem rozhraní a anotací. Modifikátory přístupu jsou také zaznamenány do vlastnosti s názvem `access` a ostatní modifikátory mají vlastní položku se stejnojmenným názvem a pravdivostní hodnotou.

Dědění od jiných rozhraní je značeno vztahem s názvem `extends`, který bude mezi vrcholem rozhraní, které dědí, a rozhraní, od kterého je děděno. Na rozdíl od tříd je možné dědit od více rozhraní.

Jak již bylo řečeno, v těle deklarace rozhraní mohou být:

- deklarace konstant,
- deklarace abstraktních metod,
- deklarace tříd,
- deklarace dalších rozhraní.

Pokud je porovnán možný obsah třídy a rozhraní z hlediska uložení do grafu, lze najít podobnosti. Například deklarace tříd a rozhraní může být obsaženo jak ve třídě, tak v rozhraní. V grafu jsou tak tyto entity uloženy jako vrcholy s patřičným typem. Je navázán vztah mezi rozhraním a touto členskou entitou s názvem `have_inner`. Dále je ve třídě umožněno deklarovat metody, ale v rozhraní je možné deklarovat pouze abstraktní metody. Abstraktní metody jsou však podmnožina klasických metod, takže je možné s nimi pracovat a ukládat je stejným způsobem. Čili pro metodu je vytvořen zvláštní vrchol s typem `method` a vytvoří se vztah mezi rozhraním a touto metodou. Podobná situace nastává u konstant v u rozhraní a atributů ve třídě. Pro každou konstantu v rozhraní lze vytvořit vrchol s typem `attribute`. Vztahem se vytvoří relace mezi rozhraním a tímto atributem zastupujícím konstantu.

Souhrn

Vlastnosti:

- `name` (String),
- `static` (boolean),
- `abstract` (boolean),
- `inner` (boolean),
- `access` (public/private/protected),
- `package` (String),
- `strictfp` (boolean).

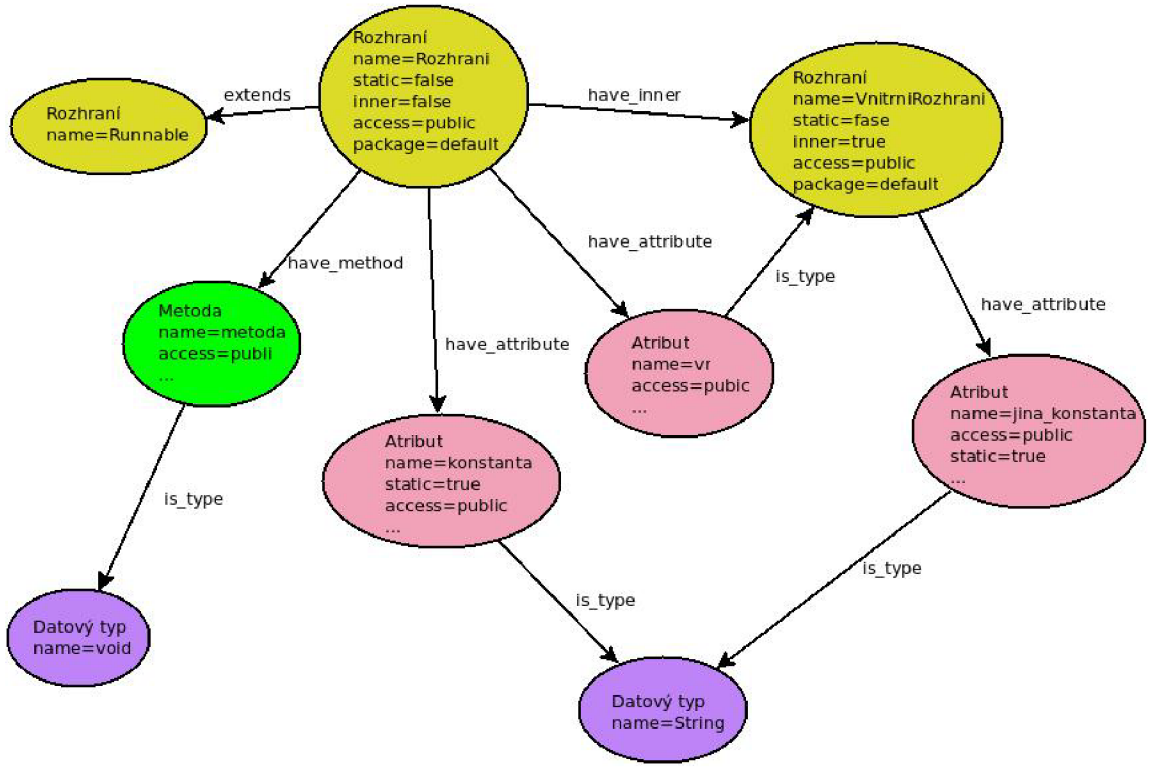
Vztahy:

- `extends` (Rozhraní),
- `uses` (Třída, rozhraní nebo výčet),
- `have_inner` (Vnitřní třída, rozhraní nebo výčet),
- `have_anonym` (Anonymní třída),
- `have_method` (Metoda),

- `have_attribute` (Atribut).

Jako jednoduchý příklad je vytvořeno rozhraní s jednou abstraktní metodou a dvěma konstantami, z nichž jedna je instancí vnitřního rozhraní. V grafu níže je znázorněn výsledek převedení kódu do databáze (viz obrázek 5.2).

```
public interface Rozhrani extends Runnable{

 interface VnitrniRozhrani{
  public static String jina_konstanta = "hodnota";
 };

 public static String konstanta = "hodnota";
 public VnitrniRozhrani vr = new VnitrniRozhrani() {};
 public void metoda();
}
```



Obrázek 5.2: Příklad grafu zaznamenávajícího rozhraní a jeho okolí.

5.4 Anotační typ

V jazyce Java je možné deklarovat speciální druh rozhraní zvaný anotační typ. Tento druh rozhraní se odlišuje od klasické anotace symbolem `@`, který předchází klíčovému slovu `interface` [26], jak je vidět na následujícím popisu gramatiky rozhraní.

```
AnnotationTypeDeclaration:
    InterfaceModifiers @ interface Identifier AnnotationTypeBody
AnnotationTypeBody:
    { AnnotationTypeElementDeclarations }
AnnotationTypeElementDeclarations:
    AnnotationTypeElementDeclaration
    AnnotationTypeElementDeclarations AnnotationTypeElementDeclaration
```

Dle popisu lze deklaraci anotačního typu začít seznamem modifikátorů, následuje název anotačního typu a na závěr je nutno doplnit tělo deklarace anotačního typu. Při vytváření anotačního typu není možné implementovat žádné rozhraní ani dědit od jiného anotačního typu nebo čehokoliv jiného. Každý anotační typ je automaticky potomkem třídy „java.lang.annotation.Annotation“. I tento vztah dědění je do grafu zaznamenán klasickým způsobem, jaký je u rozhraní.

Modifikátory anotačního typu jsou stejné jako u rozhraní a proto budou ukládány stejným způsobem. Gramatika těla anotačního typu je v dokumentaci Java popsána následovně.

```
AnnotationTypeElementDeclaration:
    AbstractMethodModifiers Type Identifier ( ) Dims DefaultValue ;
    ConstantDeclaration
    ClassDeclaration
    InterfaceDeclaration
    EnumDeclaration
    AnnotationTypeDeclaration
    ;
DefaultValue:
    default ElementValue
```

Zde jsou opět jisté podobnosti s deklarací rozhraní resp. třídy. Deklarace konstant, vnitřních tříd a vnitřní rozhraní je stejná jako v deklaraci rozhraní. Rozdíl jde vidět u deklarace metod. V deklaraci anotačního typu nelze deklarovat generickou metodu a dále zde lze nastavit defaultní hodnotu metody. Protože defaultní hodnotu metody není zapotřebí znát, je tato informace v grafu ignorována. S takovýmto zjednodušením lze na deklaraci abstraktních metod v anotačním typu pohlížet stejně jako na deklaraci metod v klasickém rozhraní. Tělo deklarace anotačního typu navíc umožňuje deklaraci výčtů (`enum`) a dalších vnitřních anotačních typů. S deklarovanými vnitřními výčty a anotačními typy je pracováno jako s klasickými vnitřními třídami. Dále je vytvořen vztah s názvem `have_inner` mezi vrcholem anotačního typu a relevantním vrcholem (vrcholem výčtu nebo vnitřním anotačním typem).

Souhrn

Vlastnosti:

- `name` (String),
- `static` (boolean),
- `abstract` (boolean),
- `inner` (boolean),

- `access` (public/private/protected),
- `package` (String),
- `strictfp` (boolean).

Vztahy:

- `extends` („java.lang.annotation.Annotation“),
- `uses` (Třída, výčet nebo rozhraní),
- `have_inner` (Vnitřní třída, výčet nebo rozhraní),
- `have_anonym` (Anonymní třída),
- `have_method` (Metoda),
- `have_attribute` (Atribut).

Jednoduchým příkladem může být následující kód a jeho odpovídající graf (obrázek 5.3). V příkladu je deklarováno vnitřní rozhraní s atributem. Dále jeden atribut s datovým typem `String` a jedna metoda s návratovým typem `int` a defaultní hodnotou. Lze použít pouze primitivních datových typů, čili datový typ `void` není povolen.

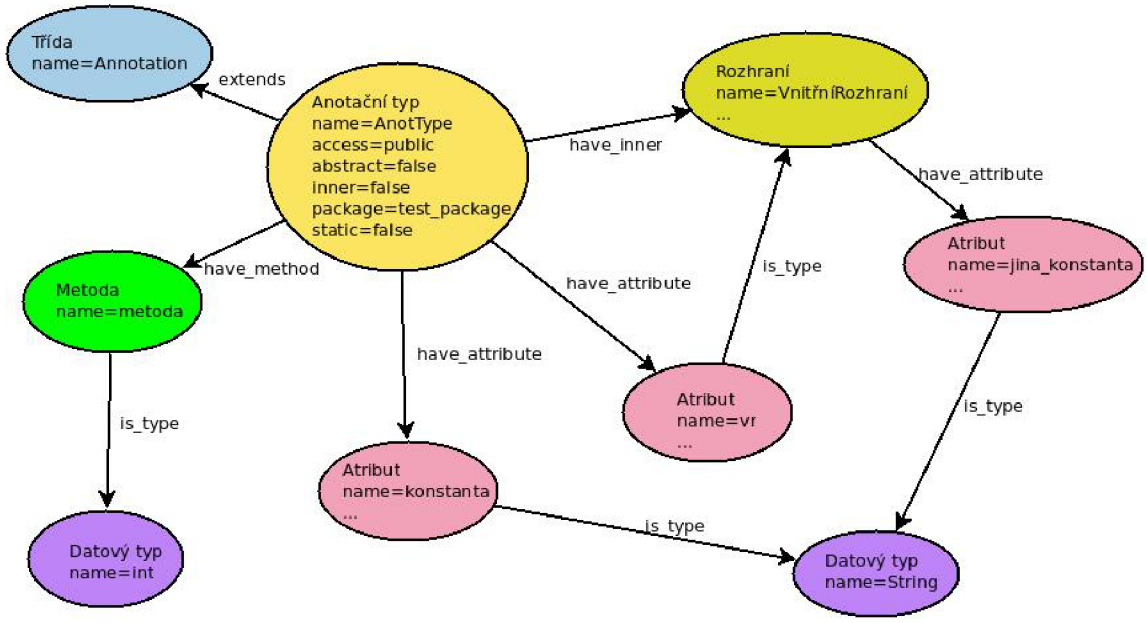
```
package test_package;
public @interface AnotType {
 interface VnitrniRozhrani{
  public static String jina_konstanta = "hodnota";
 };
 public VnitrniRozhrani vr = new VnitrniRozhrani() {};
 public String konstanta = "hodnota";
 public int metoda () default 1;
}
```

5.5 Anotace

Anotace je modifikátor, který obsahuje název anotačního typu. Dále může obsahovat nula nebo více párů, jež se skládají z názvu elementu a jeho hodnoty. Tento modifikátor je možné použít při deklaraci balíku, třídy, výčtu, rozhraní, anotačního typu, atributu, metody, parametru, konstruktoru a lokálních proměnných [26].

Gramatika anotace je v dokumentaci Java popsána následovně.

```
Annotations:
    Annotation
    Annotations Annotation

Annotation:
    NormalAnnotation
    MarkerAnnotation
    SingleElementAnnotation
```



Obrázek 5.3: Příklad grafu zaznamenávajícího anotační typ a jeho okolí.

Z popisu gramatiky anotace je zřejmé, že každý element může být anotován i více anotacemi. Dále gramatika popisuje tři druhy anotací: klasickou anotaci, indikátorovou anotaci a anotaci s jedním elementem.

Pro interpretaci anotace v grafu je nutno zavést dva nové vrcholy. Prvním vrcholem je vrchol anotace, který neobsahuje žádnou vlastnost. Naproti tomu je nutno pro tento vrchol vytvořit vztah vedoucí k odpovídajícímu anotačnímu typu. Vzhledem tomu, že se jedná o typ, je tento vztah nazván stejně jako vztah určující datový typ a to `main_type`. Dále je možnost vytvořit nula nebo více vztahů s vrcholem zachycujícím hodnotu elementu anotace. Vrchol hodnoty má jedinou vlastnost. Tato vlastnost je nazvána `value` a je zaznamenána nově nastavenou hodnotou elementu. S vrcholem elementu, kterému je přiřazena hodnota, je tento vrchol spjat vztahem s názvem `is_value_of`. Následuje shrnutí popsaných vrcholů.

Shrnutí anotace

Vlastnosti:

- `nic`.

Vztahy:

- `is_type` (Anotační typ),
- `have_element` (Hodnota).

Shrnutí hodnoty

Vlastnosti:

- `value` (String).

Vztahy:

- `is_value_of` (Metoda).

Nyní jsou probrány 3 možné druhy anotování a je pro každý druh vytvořen demonstrační příklad s předvedením uložení v grafové databázi.

Klasická anotace (`NormalAnnotation`)

Klasická anotace má v dokumentaci nejobsáhlejší popis gramatiky oproti ostatním anotacím. Na následujícím popisu je ukázána pouze část.

```
NormalAnnotation:
    @ TypeName ( ElementValuePairs )
ElementValuePairs:
    ElementValuePair
    ElementValuePairs , ElementValuePair
ElementValuePair:
    Identifier = ElementValue
ElementValue:
    ConditionalExpression
    Annotation
    ElementValueArrayInitializer
```

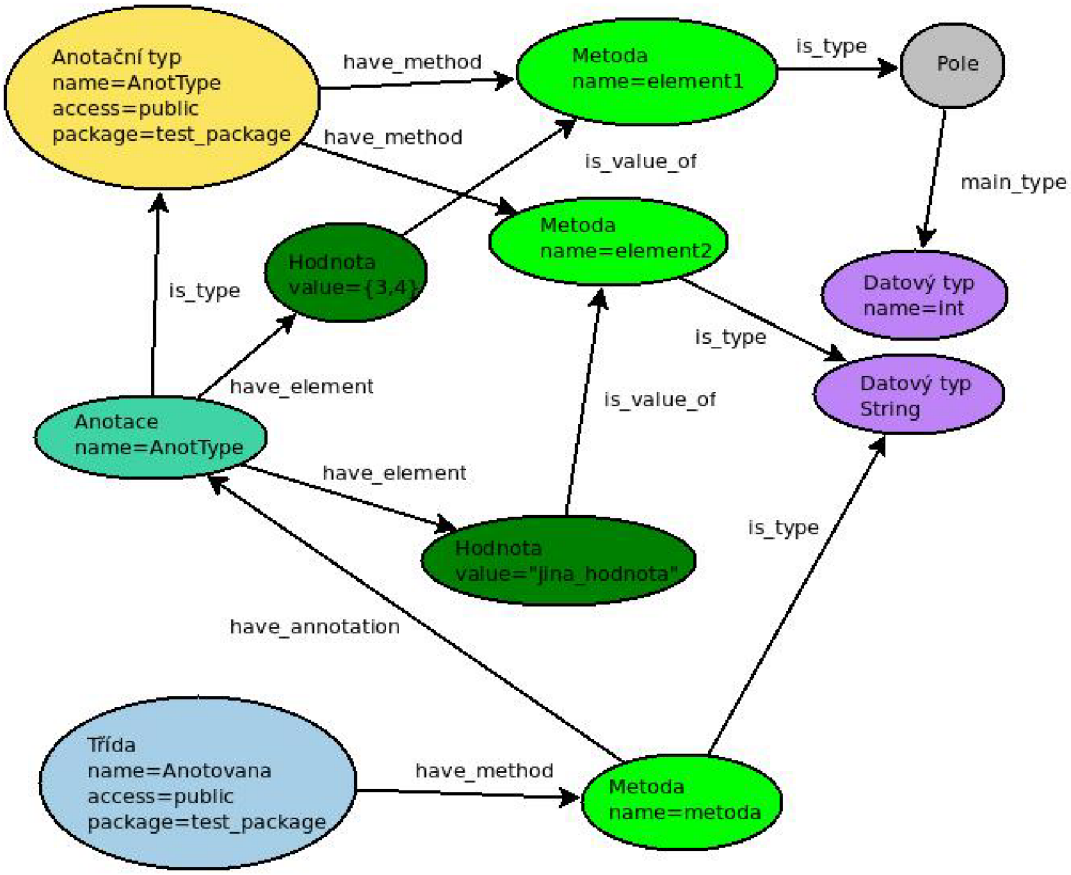
U tohoto druhu anotace je tedy možné vypisovat páry element a hodnota, kde hodnota může obsahovat podmíněný výraz, anotaci, hodnotu či pole hodnot. Tyto páry se vepisují do kulatých závorek a je možné jich zadat nula a více.

Jako příklad je prvně nadeklarován anotační typ `AnotType`, který je v následujících příkladech anotací využíván.

```
package test_package;
public @interface AnotType {
 int[] element1 () default {1,2};
 String element2() default "hodnota";
}
```

Následuje příklad klasické anotace metody a odpovídající graf (obrázek 5.4).

```
package test_package;
public class Anotovana {
 @AnotType(element2="jina_hodnota", element1={3,4}) String metoda(){
  return "nic";
 }
}
```



Obrázek 5.4: Příklad grafu zaznamenávajícího klasickou anotaci metody.

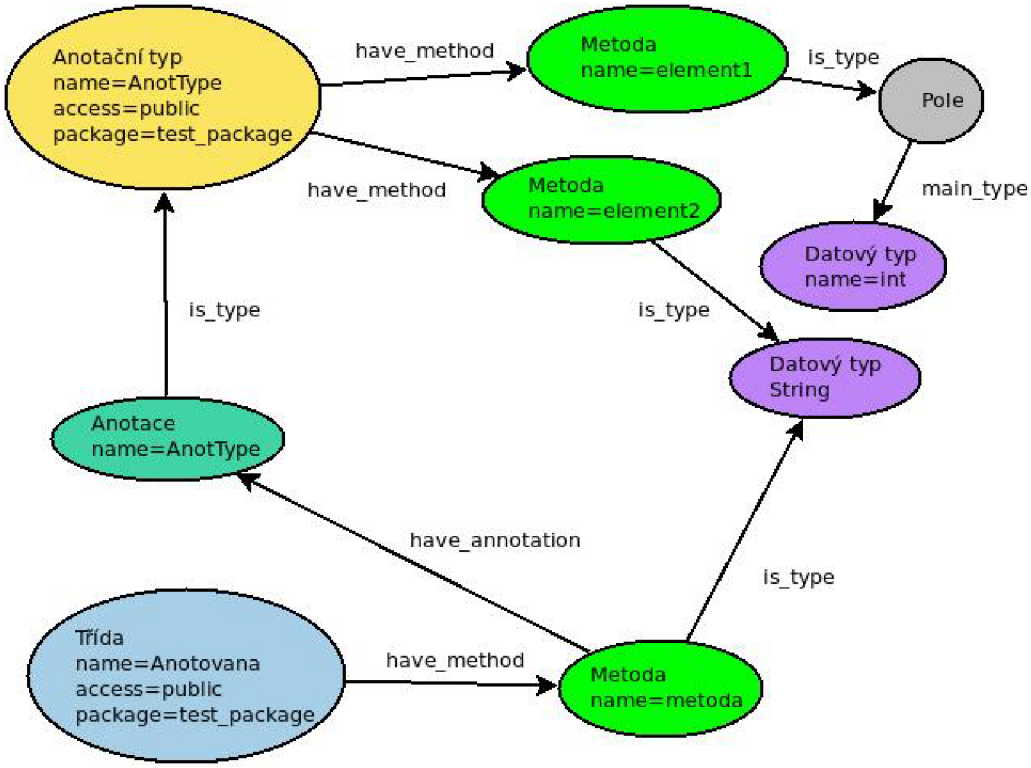
Indikátorová anotace (`MarkerAnnotation`)

Dalším druhem anotace je indikátorová anotace, která je pro změnu nejjednodušší. Jediné, co se při této anotaci uvádí, je název anotačního typu. Programátor se tak spoléhá na defaultní hodnoty uvedené při deklaraci anotačního typu. V tomto případě příklad kódu a odpovídajícího grafu (obrázek 5.5) vypadá následovně.

```
package test_package;
public class Anotovana {
 @AnotType String metoda(){
  return "nic";
 }
}
```

Anotace s jedním elementem (`SingleElementAnnotation`)

Posledním způsobem anotace je zjednodušení klasické anotace. Pokud tedy máme anotační typ s právě jedním elementem, jehož název je `value`, potom při anotaci není potřeba název tohoto elementu uvádět.



Obrázek 5.5: Příklad grafu zaznamenávajícího indikátorovou anotaci metody.

Pro příklad je nutné si prvně vytvořit demonstrativní anotační typ, který obsahuje pouze jeden element. Dále je ukázáno použití tohoto anotačního typu při anotaci a odpovídající graf (obrázek 5.6).

Anotační typ `AnotType` je deklarován takto.

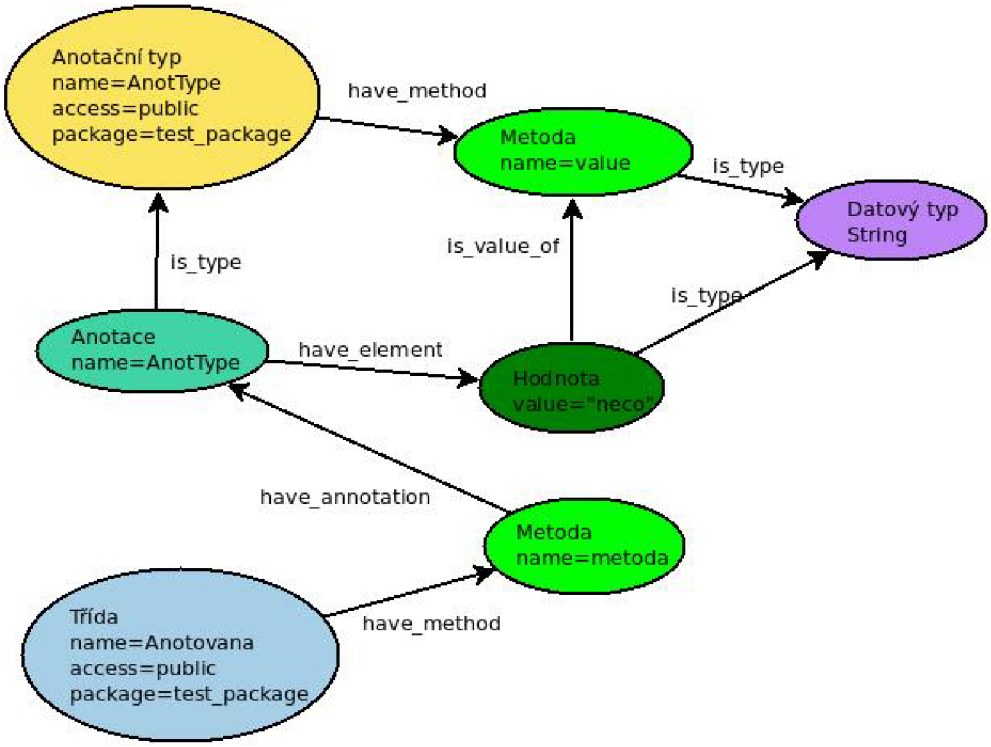
```
package test_package;
public @interface AnotType {
 String value() default "hodnota";
}
```

Použití anotačního typu je ukázáno na následujícím příkladu.

```
package test_package;
public class Anotovana {
 @AnotType("neco") String metoda(){
  return "nic";
 }
}
```

5.6 Metoda

Dle specifikace jazyka Java metody deklarují spustitelný kód, který můžeme volat. Metody mohou být parametrizovány [26].



Obrázek 5.6: Příklad grafu zaznamenávajícího anotaci metody s jedním elementem.

Popis deklarace metody je následující.

```
MethodDeclaration:
    MethodHeader MethodBody

MethodHeader:
    MethodModifiers TypeParameters Result MethodDeclarator Throws

MethodDeclarator:
    Identifier ( FormalParameterList )
```

Z popisu je zjevné, že metoda se skládá z hlavičky a těla metody. Hlavička může obsahovat seznam modifikátorů, parametrický typ a referenci na třídu zajišťující zachytávání výjimek. Nutný obsah hlavičky metody je návratový typ a název s případným seznamem přijímaných parametrů.

Jako modifikátor metody lze použít:

- `Annotation`,
- `public`,
- `private`,
- `protected`,
- `abstract`,

- `static`,
- `final`,
- `synchronized`,
- `native`,
- `strictfp`.

První modifikátor značí anotaci metody. Nejedná se o řetězec „Annotation“, nicméně může před metodou být anotace, která se k této metodě vztahuje. Tato anotace je v grafu interpretována vztahem metody s vrcholem anotace s názvem `have_annotation`. Modifikátory `public`, `private` a `protected` jsou grafem zachyceny pomocí vlastnosti vrcholu metody s názvem `access`. Dále má vrchol grafu s typem `method` vlastnosti `abstract`, `static`, `final`, `synchronized`, `strictfp` a `native` značící stejnojmenné modifikátory metody.

Generickým metodám lze parametrizovat datový typ parametru, návratové hodnoty anebo třídy zajišťující zpracování výjimek. Typové parametry se při deklaraci metody zadávají do špičatých závorek např. `<T>`. Udávají tak název typového parametru v metodě. V extrémním případě může deklarace metody vypadat následovně.

```
public <T extends Exception> T method(T prom) throws T{
 return prom;
}
```

V grafu je každý generický element (typ návratové hodnoty, datový typ parametru, třída pro zpracování výjimek) navázán vztahem určujícím datový typ na speciální uzel značící typový parametr.

Hlavička metody může obsahovat `throws` klauzuli pro deklaraci tříd zajišťujících zpracovávání výjimek. Každá metoda může takto deklarovat žádnou nebo více tříd. Tato vlastnost je v grafu zaznamenána vztahem `throws` mezi současnou třídou a referovanou třídou.

Každá metoda má svůj návratový datový typ. Ten může být buď typu `void` nebo jiného typu. Tato vlastnost je zachycena vztahem s názvem `is_type` mezi metodou a referovaným typem (resp. třídou). Dále má vrchol metody vztah `have_parameter` se všemi svými parametry. Popis vrcholu parametru je níže.

Bylo popsáno ukládání hlavičky metody do grafu. Nyní je potřeba extrahovat některé informace z těla metody. Tělo metody obsahuje blok kódu. Z tohoto bloku kódu jsou zejména zajímavé metody volané z těla, metody volané příkazem `throw`, třídy využívané v klauzuli `catch` a používané datové typy. Aby byly zachyceny všechny používané datové typy, jsou označeny datové typy proměnných v těle metody, parametry a návratové typy jako „používané“ a je vytvořen vztah mezi metodou a referovanou třídou s názvem `uses`. Jestliže se v těle metody najde volání jiné metody, je vytvořen vztah mezi metodou, odkud je volána metoda, a volanou metodou s názvem `call`. Dále pokud metoda používá jinou třídu (datový typ) anebo volá metodu nějaké třídy, je vytvořen vztah `uses` mezi třídou, ve které je metoda deklarována, a třídou, která je používaná nebo je volána některá její metoda. Jestliže se nalezne klauzule `throw` v bloku kódu, je vytvořen vztah s názvem `call_throw` s vrcholem metody, jež odpovídá referenci. Stejným způsobem je navázán vztah `uses` mezi analyzovanou třídou a třídou deklarující volanou metodu. Poslední požadované informace z těla kódu jsou datové typy zachytávaných výjimek klauzulí `catch`. Pokud se tedy na nějaký takový datový typ narazí, je vytvořen vztah s názvem `catch` s patřičnou třídou.

Souhrn

Vlastnosti:

- `name` (String),
- `static` (boolean),
- `final` (boolean),
- `access` (public/private/protected),
- `abstract` (boolean),
- `synchronized` (boolean),
- `native` (boolean),
- `strictfp` (boolean).

Vztahy:

- `is_type` (Třída),
- `have_parameters` (Parametr),
- `call` (Metoda),
- `uses` (Třída),
- `throws` (Třída),
- `have_annotation` (Anotace),
- `call_throw` (Metoda),
- `catch` (Třída).

Totožný popis uložení do grafu platí také pro konstruktory metody. Jediný rozdíl v tomto případě je v typu vrcholu.

Parametry

Deklarace metody obsahuje také deklarace parametrů metody. Parametr může následovat po anotaci. Což je v grafu znázorněno vztahem parametru s vrcholem anotace s názvem `have_annotation`. Dále mohou být určeny modifikátory, které však umožňují pouze řetězec `final`. Tento modifikátor je tedy zařazen mezi možné vlastnosti vrcholu v grafu. Nutnou součástí je určení datového typu parametru a název parametru. V grafu je název parametru zařazen mezi vlastnosti vrcholu. Typ parametru je interpretován hranou mezi vrcholem parametru a referovaným typem.

V jazyce Java je možné při deklaraci metody nastavit poslední parametr jako proměnlivý („variable arity parameter“). Toto umožňuje vložit do parametru libovolný počet parametrů deklarovaného typu. Popis této konstrukce je následující.

```
VariableModifiersopt Type... VariableDeclaratorId
```

Je jej tedy možné rozeznat pomocí třech teček mezi určením typu a názvem parametru. Tato konstrukce se však může vyskytovat pouze jako poslední parametr. Proměnlivé parametry jsou v grafu zaznamenány jako pole odpovídajícího datového typu. Tedy pokud je parametr deklarován podle následující ukázky, má metoda jeden parametr s názvem `prom`. Tento parametr následně disponuje vztahem s vrcholem typu `field`, jehož hlavní typ (`main_type`) je propojen s vrcholem, jehož název je `String`.

```
metoda(String...prom){}
```

V grafu je parametr interpretován jako zvláštní vrchol. Ke zmiňovaným vlastnostem vrcholu se dále ukládá pozice parametru mezi ostatními parametry, protože při porovnávání reference metod je důležitý počet parametrů a posloupnost datových typů. Tato pozice se počítá od hodnoty 0.

Souhrn pro parametr

Vlastnosti:

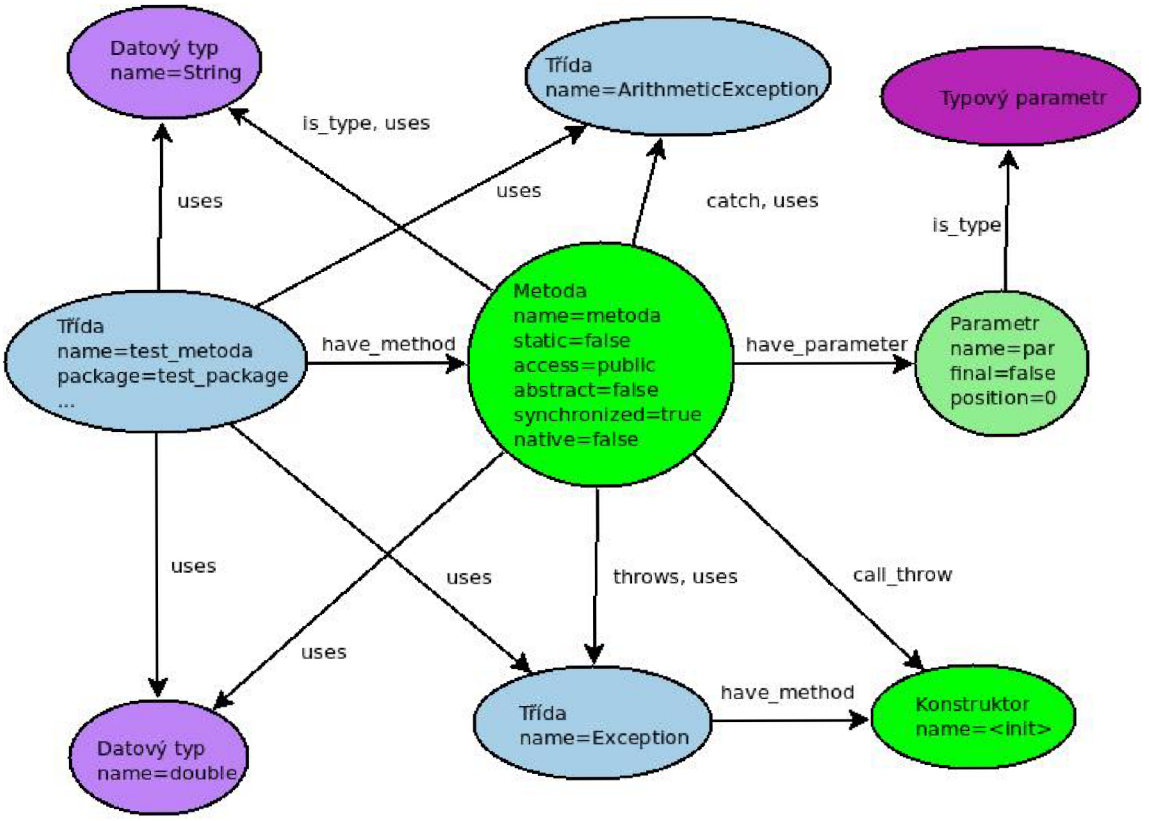
- `name` (String),
- `final` (boolean),
- `position` (int).

Vztahy:

- `is_type` (Třída),
- `have_annotation` (Anotace).

Jako komplexní příklad uložení metody a jejích parametrů do grafu nám poslouží následující kód.

```
package test_package;
import java.io.IOException;

public class test_metoda <T> {
 public synchronized strictfp String metoda(T par) throws Exception{
  double d;
  if (par instanceof String){
   //
  }else{
   throw new Exception();
  }

  try{
   d = 3/0;
  }catch(Exception e){
   // ...
  }
  return "";
 }
}
```



Obrázek 5.7: Příklad grafu zaznamenávajícího metodu a její okolí.

5.7 Atribut třídy

Gramatika jazyka Java specifikuje deklaraci atributů třídy následujícím popisem.

```
FieldDeclaration:
    FieldModifiersopt Type VariableDeclarators ;
```

Akceptovatelné modifikátory pro atributy jsou:

- `Annotation`,
- `public`,
- `protected`,
- `private`,
- `static`,
- `final`,
- `transient`,
- `volatile`.

`Annotation` značí anotaci atributu a je v grafu zaznačena vztahem `have_annotation` podobně jako je tomu u metod nebo tříd. Přístupové modifikátory `public`, `protected` a `private` jsou opět zaznamenány ve vlastnosti `access`. Všechny ostatní modifikátory, tedy `static`, `final`, `transient` a `volatile`, jsou v grafu interpretovány vlastní vlastností se stejnojmenným názvem a pravdivostní hodnotou.

Další část deklarace `Type` specifikuje datový typ atributu. Podobně, jako je tomu u návratových hodnot metod a datového typu parametrů metod, je tato vlastnost charakterizována vztahem s názvem `is_type`, který je mezi vrcholem atributu a referovaným typem. Jestliže je datový typ atributu generický, tedy že datový typ atributu bude znám až po instanciaci, je vztah značící datový typ navázán na speciální vrchol pro typové parametry.

Poslední částí je určení názvu atributu, který může obsahovat buď pouze název atributu, název s označením atributu jako pole (označí se hranatými závorkami) nebo inicializaci atributu. Název se ukládá do vlastnosti `name`. Jestliže se však jedná o pole, není vrchol atributu spojen čistě s vrcholem datového typu ve vztahu `is_type`. V tomto případě se mezi vrcholy typu a atributu nachází další vrchol typu `field`, který značí, že se jedná o pole. V tomto vrcholu se uchovává pouze jediná věc – jakého datového typu je. Tento vztah je pojmenován `main_type`.

Shrnutí pole

Vlastnosti:

- `nic`.

Vztahy:

- `main_type` (Třída).

Shrnutí atributu

Vlastnosti:

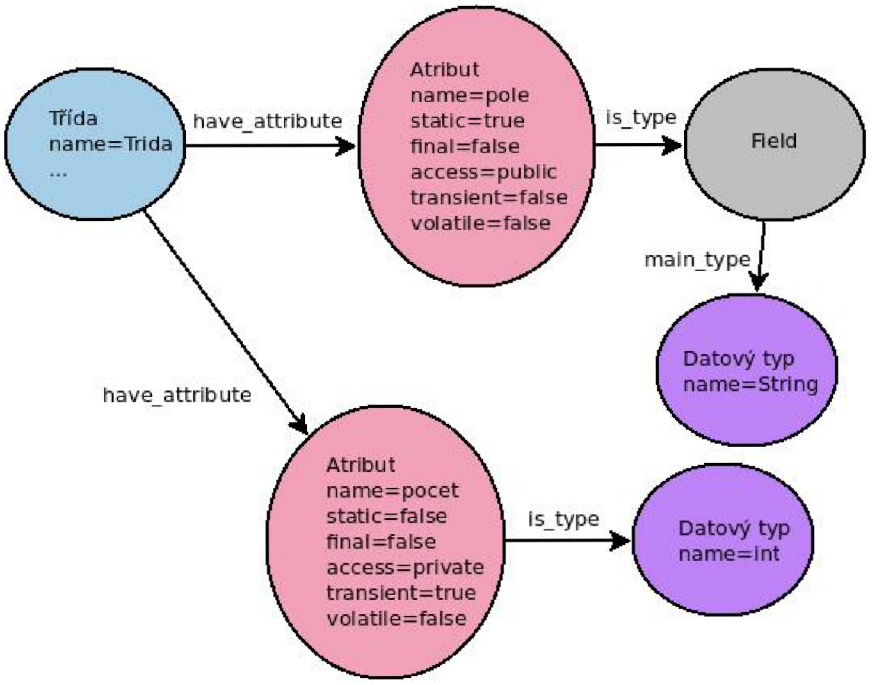
- `name` (String),
- `static` (boolean),
- `final` (boolean),
- `access` (public/private/protected),
- `transient` (boolean),
- `volatile` (boolean).

Vztahy:

- `have_annotation` (Anotace),
- `is_type` (Třída).

Dále je ukázán jednoduchý příklad uložení atributů třídy do grafu. Následuje zdrojový kód v jazyce Java, který deklaruje několik atributů a poté je vyobrazen relevantní graf (obrázek 5.8).

```
public class Trida{
 public static String pole[];
 private transient int pocet;
}
```



Obrázek 5.8: Příklad grafu zaznamenávajícího atribut a jeho okolí.

5.8 Generické typy

Je potřeba aby se generický typ choval jako obyčejný typ nebo třída kvůli dotazování. Přesto musí poskytovat navíc informace o hlavním typu (např. `List`) a generických parametrech.

Využitý obsah ve vrcholu reprezentujícím generický typ

Vlastnosti:

- `nic`.

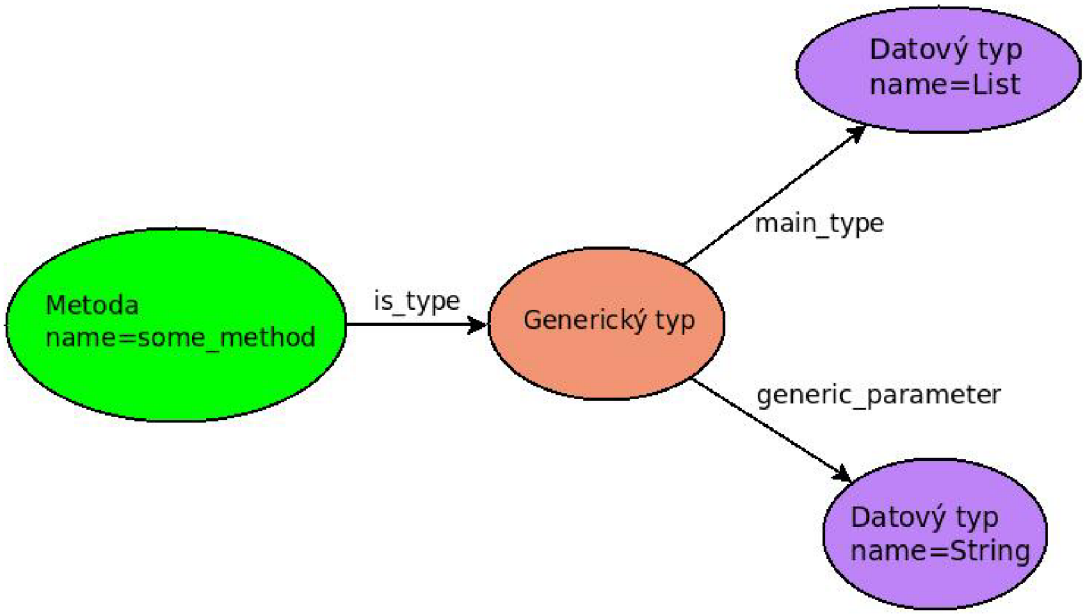
Vztahy:

- `main_type` (Třída),
- `generic_parameter` (Třída).

Protože použití generických datových typů umožňuje spoustu možností, byly vytvořeny tři příklady pokrývající hlavní větve problému.

1. Příklad uložení následujícího kódu do grafu

```
public List<String>  some_method(){
   ...
}
```



Obrázek 5.9: Příklad grafu zaznamenávajícího generický typ a jeho okolí.

Graf (obrázek 5.9) tak obsahuje vrchol zastupující metodu se jménem `some_method`. Tato metoda má návratovou hodnotu `List<String>`, což je generický typ. Návratová hodnota je popsána vztahem `is_type` s vrcholem „Generický typ“, který je dále popsán vztahy `main_type` a `generic_parameter`.

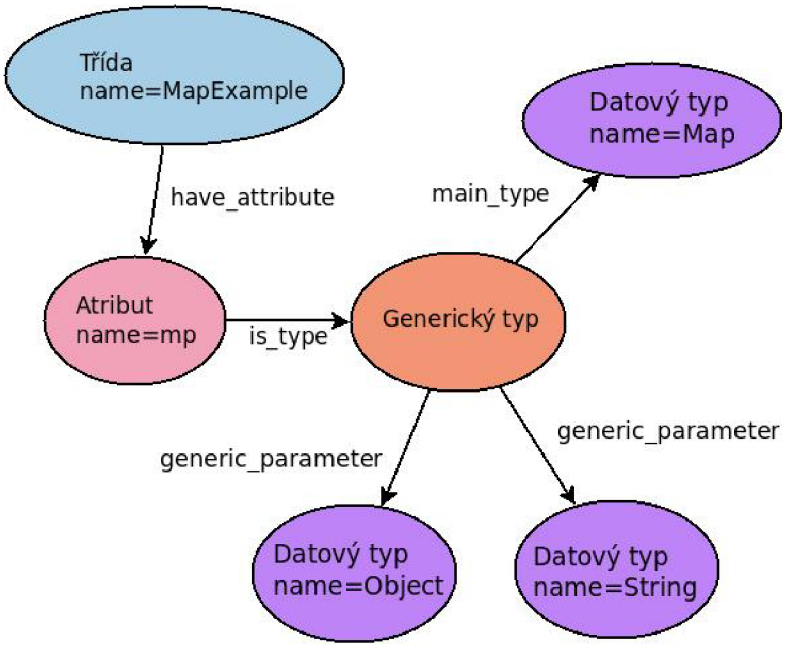
2. Příklad uložení následujícího kódu do grafu

```
public class MapExample {
        Map<Object,String> mp;
}
```

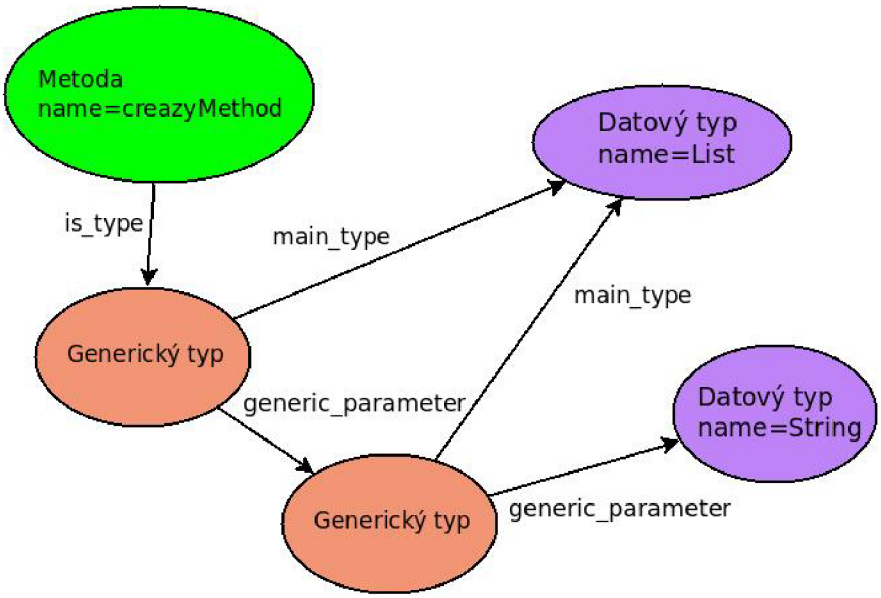
Na tomto příkladu je demonstrováno uložení do grafové databáze v případě, že se v kódu nachází větší počet generických parametrů. Jediný rozdíl je v tom, že uzel `Generický typ` má další vztah `generic_parameter`. Pořadí generických parametrů nás v tomto případě příliš nezajímá, proto se tato informace neukládá. Výsledek je zobrazen na obrázku 5.10.

3. Příklad uložení následujícího kódu do grafu

```
public List<List<String>> creazyMethod(){
  ...
}
```



Obrázek 5.10: Příklad grafu zaznamenávajícího generický typ a jeho okolí.



Obrázek 5.11: Příklad grafu zaznamenávajícího generický typ a jeho okolí.

V případě, že bude typovým parametrem opět generický typ, je potřeba mít možnost místo konkrétního typu navázat na vztah `generic_parameter` další uzel „Generický typ“. Uložení se provádí podle obrázku 5.11.

5.9 Datový typ

Důvodem definování typu vrcholu „Datový typ“ a „Třída“ je oddělení typů, které jsou definovány v rámci analyzovaného kódu a neanalyzovaného kódu. Příkladem může být jednoduchý program „Hello world“, ve kterém je používána konstanta typu `String`.

```
public class Main {
 public static void main(String[] args) {
  String hw = "Hello world";
  System.out.print(hw);
 }
}
```

Analýzou čistě tohoto programu není k dispozici AST třídy `String`, proto je `String` do grafu zaznačen jako „Datový typ“. Tento typ vrcholu není v grafu popsán více než názvem a vztahy, kdo jej využívá. Případně, pokud je využívána metoda této nemapované třídy, vytvoří se ještě vztah s informací, že tento datový typ disponuje volanou metodou.

Problém může nastat při větším množství dekompilovaných souborů, kde je právě analyzována třída, která využívá jinou třídu, jež bude teprve dekompilována. Příkladem může být dekompilování souboru s kódem uvedeným výše jako prvního a následně dekompilace `java/lang/String`. Dekompilací prvního souboru tak vznikne datový typ `String` a později je při dekompilaci třídy `String` vytvořen vrchol `String` znovu. Proto je potřeba před každým vytvořením vrcholu popisujícího libovolnou třídu (resp. rozhraní, výčet prvků, generický typ nebo datový typ) vyhledat v databázi, zda již neexistuje vrchol „Datový typ“ s totožným názvem. Jestliže je takový vrchol nalezen, je nutno veškeré vztahy překopírovat do nově vytvářeného vrcholu. Tímto bude zachována informace o použití této třídy jinými třídami či metodami.

Překopírováním vztahů může nastat situace, kdy první třída nejenže `String` využívá, ale dokonce používá některou z jeho metod. Potom by se vytvořil vrchol „Datový typ“ se vztahem `have_method`. Tato metoda však bude popsána neúplným množstvím informací, protože doposud více informací o dané metodě nebylo k dispozici. Při dekompilaci `java/lang/String` se následně překopírují všechny vztahy včetně vztahu s metodou, která se však záhy vytváří znovu. Zde je tedy potřeba také vyhledávat existenci již vytvářené metody. Java však umožňuje přetěžování funkcí a tedy není možné vyhledávat pouze na základě názvu funkce. Klíčovou informací je tedy také počet a datový typ parametrů. Při porovnávání generických datových typů stačí porovnat hlavní typ a generické parametry ignorovat, neboť v tomto java kompilátory nedělají rozdíl. Hledáním shody reference a deklarace metody se zabývá následující kapitola.

5.10 Shoda reference a deklarace metody

Pokud je vytvořen vrchol metody podle deklarace a později při analýze se narazí na volání metody, je potřeba definovat způsob rozpoznání shody této reference metody s deklarací. V opačném případě by měl tento postup fungovat taktéž. Tedy pokud je prvně zpracováno volání metody, která ještě nebyla v grafu vytvořena, vytvoří se dočasný uzel pro tuto metodu. Dále, při zpracovávání AST této metody, je potřeba nalézt dočasný uzel a nahradit ho.

V následujícím textu jsou často používané pojmy, které je nutno pro srozumitelnost specifikovat.

```
<T> // typovy parametr
<T> void metoda(T) // metoda s typovym parametrem
void metoda(List<String>) // metoda s generickym parametrem
List<T> a // generický datový typ s typovym parametrem
void metoda(String...a) // metoda s proměnlivým parametrem
```

Podle popisu jazyka Java [26] se propojení volání (resp. reference) metody a deklarace metody provádí ve třech fázích. Pokud propojení neuspěje v jedné fázi, pokračuje se další. Nepovede-li se propojení ve třetí fázi, je nahlášena chyba při překladu. V první řadě se však detekuje, zda je metoda potenciálně propojitelná. Tedy vyhledá se množina aplikovatelných deklarací metod. Tyto potenciální metody musí splňovat všechny následující kritéria.

- Identické názvy reference a deklarace metody.
- Blok kódu volající metodu musí mít patřičný přístup k deklaraci metody.
- Jestliže má metoda `n` parametrů se speciálním parametrem určujícím proměnlivý počet parametrů („variable arity method“), musí mít reference na metodu více nebo rovno `n-1` parametrů.
- Jestliže nemá metoda tento speciální parametr a má `n` parametrů, potom musí mít reference metody právě `n` parametrů.
- Jestliže je v referenci metody použit explicitní typový argument a metoda je generická, potom musí být počet typových parametrů roven počtu typových parametrů metody.

První fáze

V první fázi propojování reference metody s deklarací se vychází z množiny potenciálně aplikovatelných metod. Zde se postup větví podle toho, zda je metoda generická či ne. Jestliže se jedná o generickou metodu, je nutno kontrolovat, zda odpovídají datové typy parametru, které nemají generický datový typ. Dále, zda lze generický typ navázat na datový typ z volání metody. Jestliže se naopak nejedná o generickou metodu, je propojení možné, pokud jsou parametry referované metody stejného datového typu, jako deklarované metody nebo podtypu. Další možností je, že jsou datové typy „konvertovatelné“, tedy že je možné provést typovou konverzi na datový typ v parametru metody.

Jestliže nejsou podmínky první fáze splněny s žádnou deklarací, pokračuje se druhou fází propojení reference metody s deklarací.

Druhá fáze

Tato fáze se na rozdíl od první pokouší o explicitní konverzi parametrů, pokud nejsou generické. V případě parametrů s generickým datovým typem je postup stejný.

Třetí fáze

Poslední fáze spočívá v propojování metod, které mají proměnlivý počet parametrů. Detekce propojení je založena na předchozích fázích. Navíc je mezi podmínkami možnost výskytu tohoto parametru umožňující proměnlivý počet parametrů.

V případě parametrů s negenerickým datovým typem je potřeba při vytváření dočasných vrcholů metod v grafu vytvořit tento uzel s již konvertovanými typy. Tuto možnost

poskytuje knihovna Procyon, díky které je při volání metody možné získat výsledný datový typ konverze typu parametru. Pokud je později zpracovávána deklarace metody, může již porovnání probíhat pomocí uložených datových typů. Pokud probíhá vytváření vrcholů v opačném pořadí, opět je potřeba srovnávat parametry vrcholu deklarované metody s konečným datovým typem parametrů referované metody.

Protože je možné za typové parametry dosadit jakýkoliv typ, je při hledání shody metod porovnání libovolného typu s typovým parametrem vždy označováno jako shodné. Detekce špatného přiřazení typu do parametru se neprovádí. Předpokládá se, že kompilátor provedl bezchybný převod do bajtkódu.

Knihovna Procyon pracuje s proměnlivým parametrem jako s polem. V případě volání metody s tímto proměnlivým parametrem dokáže rovnou tvrdit, že je poslední parametr pole. Proto je práce s voláním metody stejná, jakoby se na poslední pozici vyskytovalo pole odpovídajícího datového typu. Tedy práce s touto metodou se nebude nijak lišit oproti práci s jinými metodami.

Shrnutí propojení reference a deklarace metod

Propojení volané metody s deklarací metody bude probíhat na základě následujících bodů:

- shodné jméno metody,
- shodný počet parametrů,
- shodný datový typ všech parametrů,
  - typový parametr bude označen jako shodný s libovolným typem,
  - generické parametry se shodným hlavním typem (např. List, Map), ale rozdílnými typovými parametry budou taktéž označeny jako shodné.

5.11 Shrnutí

V následujících dvou tabulkách (tabulka 5.1 a 5.2) jsou vypsány všechny používané typy vrcholů v grafové databázi a k nim je zaznamenáno, které vlastnosti a vztahy se pro daný vrchol ukládají. Symbol „A“ znamená, že se položka používá, „N“, že se položka nepoužívá.

	objecttype	name	static	abstract	access	package	final	anonym	inner	strictfp	transient	volatile	value	synchronized	native	position
Vrchol (Object)	A	A	N	N	N	N	N	N	N	N	N	N	N	N	N	N
Třída (Class)	A	A	A	A	A	A	A	A	A	A	N	N	N	N	N	N
Vnitřní třída (IC_inner)	A	A	A	A	A	A	A	A	A	A	N	N	N	N	N	N
Anonymní třída (IC_anonym)	A	A	A	A	A	A	A	A	A	A	N	N	N	N	N	N
Výčet (Enum)	A	A	A	A	A	A	A	A	A	A	N	N	N	N	N	N
Datový typ (Datt)	A	A	N	N	N	N	N	N	N	N	N	N	N	N	N	N
Pole (Field)	A	A	N	N	N	N	N	N	N	N	N	N	N	N	N	N
Generický typ (Gent)	A	A	N	N	N	N	N	N	N	N	N	N	N	N	N	N
Rozhraní (If)	A	A	A	A	A	A	A	A	A	A	N	N	N	N	N	N
Anotační typ (Annot_type)	A	A	A	A	A	A	A	A	A	A	N	N	N	N	N	N
Anotace (Annot)	A	A	N	N	N	N	N	N	N	N	N	N	N	N	N	N
Element antoace (Element)	A	A	N	N	N	N	N	N	N	N	N	N	A	N	N	N
Metoda (Meth)	A	A	A	A	A	N	A	N	A	N	N	N	N	A	A	N
Konstruktor (Meth_con)	A	A	A	A	A	N	A	N	A	N	N	N	N	A	A	N
Parametr metody (Par)	A	A	N	N	N	N	A	N	N	N	N	N	N	N	N	A
Atribut (Attr)	A	A	A	N	A	N	A	N	N	N	A	A	N	N	N	N
Typový parametr (Type_param)	A	A	N	N	N	N	N	N	N	N	N	N	N	N	N	N

Tabulka 5.1: Shrnutí vlastností všech vrcholů v grafové databázi.

	have_annotation	extends	uses	implements	have_inner	have_anonym	have_method	have_attribute	main_type	generic_parameter	have_element	is_type	is_value_of	have_parameter	call	throws	call_throws	catch
Vrchol (Object)	A	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N
Třída (Class)	A	A	A	A	A	A	A	A	N	N	N	N	N	N	N	N	N	N
Vnitřní třída (IC_inner)	A	A	A	A	A	A	A	A	N	N	N	N	N	N	N	N	N	N
Anonymní třída (IC_anonym)	A	A	A	A	A	A	A	A	N	N	N	N	N	N	N	N	N	N
Výčet (Enum)	A	A	A	N	A	A	A	A	N	N	N	N	N	N	N	N	N	N
Datový typ (Datt)	A	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N
Pole (Field)	A	N	N	N	N	N	N	N	A	N	N	N	N	N	N	N	N	N
Generický typ (Gent)	A	N	N	N	N	N	N	N	A	A	N	N	N	N	N	N	N	N
Rozhraní (If)	A	A	A	N	A	A	A	A	N	N	N	N	N	N	N	N	N	N
Anotační typ (Annot_type)	A	A	A	N	A	A	A	A	N	N	N	N	N	N	N	N	N	N
Anotace (Annot)	A	N	N	N	N	N	N	N	N	N	A	A	N	N	N	N	N	N
Element anotace (Element)	A	N	N	N	N	N	N	N	N	N	N	N	A	N	N	N	N	N
Metoda (Meth)	A	N	A	N	N	N	N	N	N	N	N	A	N	A	A	A	A	A
Konstruktor (Meth_con)	A	N	A	N	N	N	N	N	N	N	N	A	N	A	A	A	A	A
Parametr metody (Par)	A	N	N	N	N	N	N	N	N	N	N	A	N	N	N	N	N	N
Atribut (Attr)	A	N	N	N	N	N	N	N	N	N	N	A	N	N	N	N	N	N
Typový parametr (Type_param)	A	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N

Tabulka 5.2: Shrnutí možných vztahů všech vrcholů v grafové databázi.

Kapitola 6

Implementace aplikace

Cílem této práce je vytvořit především knihovnu pro umožnění dotazování nad abstraktním syntaktickým stromem. Je důležité, aby API této knihovny bylo jednoduché a intuitivní, proto umožňuje třída `ASTquery` šest nepostradatelných metod. Kompletní API je popsáno v příloze C.

Pro tuto knihovnu byla vytvořena třída, která demonstrativně využívá `ASTquery` API. Výsledná aplikace je určena pro použití v příkazové řádce. Manuál k použití aplikace je sepsán v příloze B.

Aplikace je implementována v jazyce Java. Při vývoji byl využit repozitář na stránkách `gitlab.com`. Aplikace je vytvářena jako open source projekt a její zdrojové kódy jsou dostupné na adrese https://gitlab.com/greg007/java-AST-query-language.

6.1 Použité nástroje

Jak již bylo dříve probíráno, pro vytvoření lexikálního analyzátoru a parseru je použita knihovna ANTLR. Gramatika pro tuto knihovnu je uvedena v příloze D. Třídy vygenerované pomocí ANTLR jsou přímo importované do projektu. Taktéž knihovna ANTLR je obsažena ve výsledné aplikaci, takže není zapotřebí mít již tuto knihovnu předinstalovanou.

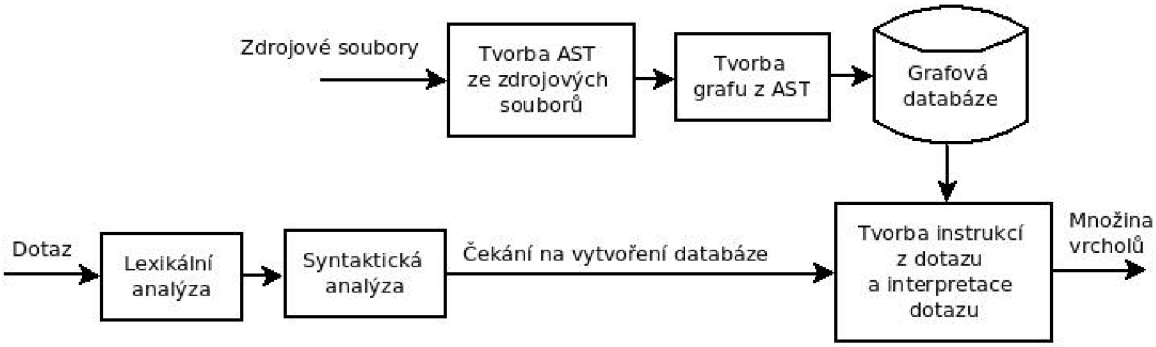
Pro dekompilaci a tvorbu abstraktního syntaktického stromu vstupních souborů je využit nástroj Procyon. Tímto nástrojem je získán AST každého vstupního souboru. Přestože byla tato knihovna vyhodnocena jako nejkvalitnější ze zdarma dostupných knihoven pro dekompilaci Java kódu, není problém narazit při dekompilaci na chyby. Z toho důvodu je při práci s nástrojem Procyon zvýšena pozornost na selhávání dekompilace a v případě problému se vypíše hlášená chyba. Proto selhání Procyonu nenaruší běh aplikace, ale je vypsáno upozornění na chybnou dekompilaci konkrétního zdrojového souboru.

Vzhledem k rešerši o grafových databázích byla vybrána knihovna Neo4J. Výhodou této databáze byla mimo jiné rychlost spouštění a odpojování, což je v případě pouhého dotazování (bez dekompilace) velice znatelné i v případě rozsáhlého obsahu databáze. Přestože se využívá knihovny Neo4J pro práci s databází, za účelem větší škálovatelnosti je práce s databází zastřešena knihovnou Blueprints. Tento nástroj umožňuje připojení a práci s různými databázemi.

Díky použití knihovny Blueprints se naskytla možnost využít nástroje Frames, který umožňuje pohled na databázové objekty jako na objekty objektově orientovaného jazyka. Toho je využito pro oddělení a definici typů vrcholů, jejich vlastností (ukládaných informací) a určení výčtu vztahů, které je možné navázat.

6.2 Struktura aplikace

Struktura aplikace a postup vyhodnocování je znázorněn na obrázku 6.1.



Obrázek 6.1: Schéma posloupnosti zpracování dotazu a zdrojových souborů.

Na obrázku je posloupnost operací znázorněna zleva doprava. Dále je tak z obrázku vidět, že systém obsahuje dvě větve programu, které se propojují až v posledním kroku.

Subsystém v horní části schématu má na starosti naplnění databáze ze vstupních souborů. Prvním krokem je použití nástroje Procyon a tedy vytvoření abstraktního syntaktického stromu ze vstupních souborů. Výstupem první části je tedy AST, který se dále převádí za pomoci nástroje Frames do objektů odpovídajících patřičným typům vrcholů. S využitím Frames, který je propojený s Blueprints API, se postupně zpracovávané AST jednotlivých vstupních souborů ukládají do grafové databáze, která je konečným výstupem tohoto subsystému. Dekompilace vstupních souborů a plnění grafové databáze se nemusí provádět při každém běhu programu. V případě, že nejsou vstupní soubory zadány, je předpokládáno, že je databáze již vytvořena a přejde se rovnou ke kroku zpracování dotazu.

Druhý subsystém, který je vyobrazen ve spodní části schématu, se provádí při každém běhu aplikace. Vstupem je dotaz, který podléhá pravidlům syntaxe a sémantiky definované souborem s gramatikou (příloha D) pro nástroj ANTLR. Prvními kroky zpracování dotazu je lexikální a syntaktická analýza prováděná pomocí tříd vygenerovaných nástrojem ANTLR. Protože je program vykonáván sekvenčně, po lexikální a syntaktické analýze dostává slovo subsystém pro plnění databáze (pokud je vyžadován). Tato posloupnost procesů je zde vhodná především proto, že v případě chybného zápisu dotazu je informace o chybě známa prakticky okamžitě a není potřeba čekat na převedení vstupních souborů na graf (pro 10 000 tříd je doba zpracování v jednotkách hodin na průměrném stroji). Závěrem přichází klíčový proces, v rámci kterého je z AST dotazu vytvořena fronta instrukcí. Následně jsou tyto instrukce provedeny na definované nebo defaultní databázi. Výstupem tohoto subsystému je množina vrcholů z grafové databáze, které odpovídají zadanému dotazu.

Demonstrativní aplikace dále z výstupní množiny vrcholů vypisuje číslo uzlu v grafové databázi, typ uzlu (`class`, `interface` apod.) a název uzlu. Každému prvku výstupní množiny tak odpovídá jeden řádek ve výpisu předváděcí aplikace.

Optimalizace

Pro zrychlení subsystému, který plní grafovou databázi, bylo původně zamýšleno využití knihovny BCEL. Tato knihovna dokáže velice rychle analyzovat vstupní soubory a dávat výsledky. Problém však je v nedostatečnosti na požadované množství informací ze vstupních

souborů. Proto měla tato knihovna nastavit pouze hrubé informace a v případě potřeby (na základě vstupního dotazu) by se provedlo doplnění pomocí knihovny Procyon. Knihovna Procyon však běží na několika úrovních abstrakce dekompilace, kde poslední úrovní je zdrojový kód. Fáze před poslední úrovní je převádění AST na tento zdrojový text. Tato fáze je časově nejnáročnější z celého procesu dekompilace. Během experimentování s touto knihovnou bylo zjištěno, že před krokem k AST je fáze s elementy AST uloženými ve strukturovaných objektech. V podstatě se jedná také o AST, avšak jsou uzlům stromu přiděleny objekty zastupující odpovídající element jazyka. Experimenty s využitím knihovny BCEL ukázaly, že získávání informací z této fáze je časově srovnatelné s použitím knihovny Procyon. Proto nebyla knihovna BCEL v aplikaci využita.

Jako další optimalizace byly zavedeny zkratky v dotazovacím jazyce. Pro nejčastěji používané konstrukce jazyka (hledání třídy podle jména, hledání množiny potomků jisté třídy) jsou vytvořeny speciální konstrukce (přímé zadání jména třídy, jméno třídy začínající symbolem vykřičníku). Dále pro všechny vztahy mezi elementy (např. `have_method`) jsou zavedeny zkrácené názvy (`>method`), kde symbol šipky (`>` nebo `<`) ukazuje směr vztahu. Tímto je sníženo množství klíčových slov k zapamatování pro programátora. Dále je možné vybrané vztahy zapisovat zadáním pouze prvního písmene vztahu (`>m`). Zavedením těchto zkratek rapidně klesla délka zápisu dotazu.

Kapitola 7

Experimenty a vyhodnocení

V rámci experimentů byla zjišťována složitost algoritmů vytvoření grafu a zpracování dotazu. Bylo vybráno 10 různých aplikací nebo knihoven dostupných na internetu, které byly vytvářeny v jazyce Java. Při výběru těchto projektů byl kladen důraz na licenci, aby dekompilací kódu nebyly porušeny licenční podmínky. Všechny vybrané aplikace a knihovny jsou proto open source. Jednou z aplikací, na kterých byly prováděny experimenty, je i aplikace vytvořená v rámci této práce.

V prvním experimentu bylo 10 vybraných projektů a aplikací použito jako vstupní soubor. Každým projektem tak byla naplněna jedna databáze a byl měřen čas zpracování vstupního souboru a velikost výsledné databáze. Ve druhém experimentu byl použito pět databází z prvního experimentu. Na těchto vybraných databázích byla provedena sada dotazů (k nalezení v příloze F), které byly vytvořeny jako seznam příkladů dotazovacího jazyka. Protože je každý projekt jiný a v dotazech se často dotazuje na konkrétní názvy elementů jazyka, bylo potřeba sadu dotazů upravit pro každou databázi (projekt) tak, aby byl výsledkem alespoň nějaký záznam. Závěrem je zaměřena pozornost zvláště na dotazy, jež mají ekvivalentní výstup, a je sledována jejich časová složitost.

K měření času je použit nástroj „time“ dostupný na Linuxových systémech. Je tak zaznamenáván reálný čas (hodnota u popisku `real`). Tento čas značí skutečný uběhlý čas využitý procesem.

Experimenty jsou prováděny na stroji s operačním systémem Ubuntu 14.04, 3,8 GiB pamětí RAM a se čtyřjádrovým procesorem Intel Core i3-4005U s frekvencí jádra 1.70 GHz.

7.1 Složitost vytvoření grafu

Tímto experimentem je sledována časová složitost algoritmu převedení vstupního souboru do grafové databáze (včetně dekompilace). Pro tento účel bylo vybráno 10 aplikací a knihoven vytvořených v jazyce Java. Jedním ze vstupních programů je také aplikace vytvořená v rámci této práce. Při výběru vstupních souborů (projektů a knihoven vytvořených v jazyce Java) byl kladen důraz na licenci, protože není přijatelné dekompilací porušovat licenční smlouvu. Proto jsou všechny vybrané projekty open source. Dále by bylo vhodné vybrat takové projekty, které naplní databázi v rozličné kapacitě. Toto je však problém, neboť nelze předem odhadnout množství elementů databáze. Mírou odhadu výsledné velikosti databáze tak byla považována velikost vstupního souboru. Avšak vstupní soubor (např. soubor s koncovkou jar) může obsahovat multimediální data nebo jiné zdroje, proto může být tato míra velice klamná.

Protože aplikace neumožňuje spuštění bez zadání dotazu, pro účel experimentování je zadán dotaz hledající neexistující třídu podle názvu. Tento dotaz byl vybrán, protože je jeho zpracování velice rychlé na databázi s libovolným množstvím uzlů. Proto je čas zpracování dotazu zanedbatelný. Měření času je provedeno příkazem `time`. Příkaz spuštění aplikace pro první experiment tedy vypadá následovně.

```
$ time ./ASTquery.jar -query "trida.ktera.neexistuje"
-source <vybrany projekt> -database-directory "graph<x>.db"
```

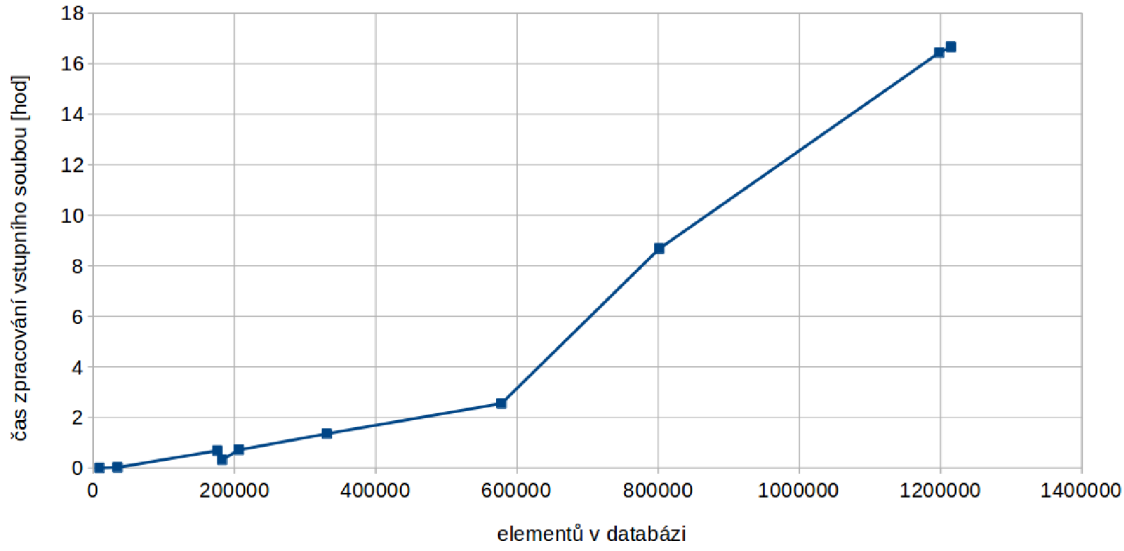
V tomto příkazu se řetězec `<vybrany projekt>` nahradí adresou k právě testovanému projektu a řetězec `<x>` se nahradí číslem pořadí projektu v tabulce. Každý projekt je analyzován aplikací celkem třikrát. Výsledný čas zapsaný v tabulce je průměrem časů z těchto tří spuštění.

Projekt	Velikost	Počet souborů	Počet uzlů	Počet vztahů	Čas
ASTquery.jar	12,4 MB	8403	200844	1013818	16h 39m 30s
drjava-stable-20140826-r5761/ [3]	39,9 MB	6068	191396	1006788	16h 26m 03s
jboss-fsw-installer-6.0.0.GA-redhat-4 [4]	412,1 MB	4591	135096	666667	8h 40m 52s
elan/ [7]	6,4 MB	2215	81045	497141	2h 33m 19s
neo4j-cypher-1.9.9.jar [23]	4,5 MB	3277	56057	274846	1h 21m 40s
procyon-decompiler-0.5.28.jar [35]	1,8 MB	990	32530	174013	43m 27s
ANTLR-4.5-complete.jar [8]	1,5 MB	937	28699	147308	41m 23s
neo4j-kernel.jar [23]	1,6 MB	1252	31052	151679	19m 43s
blueprints-core-2.7.0.jar [9]	274,2 kB	217	6185	28589	1m 39s
frames-2.7.0.jar [5]	78,2 kB	72	1843	7306	27s

Tabulka 7.1: Tabulka výsledků ze zpracování vstupních souborů (resp. adresářů) do grafu.

Hlavní náplní tohoto experimentu je sledování doby zpracování vstupního souboru (aplikace či knihovny, která je vytvořena v jazyce Java) v závislosti na množství vytvořených elementů databáze. Tabulka 7.1 zachycuje data získaná při měření času zpracování vstupu a graf 7.1 vizualizuje naměřená data. Na tomto grafu osa `x` značí hodnotu součtu množství vrcholů a vlastností výsledné databáze. Osa `y` pak udává čas strávený zpracováním vstupu v hodinách.

Z tabulky 7.1 lze vyčíst, že nejdelší dobu zpracování (16 hodin, 39 minut a 30 sekund) měl vstupní soubor `ASTquery.jar`. Tento soubor má velikost 12,4 MB a vytvořil tak přes 200 tisíc vrcholů a přes milion vztahů. Nejrychleji zpracovaným souborem byl `frames-2.7.0.jar`, který má velikost 72 kB. Tento vstupní soubor byl zpracován za pouhých 27 sekund a bylo vytvořeno 1843 vrcholů a 7306 vztahů. Vstupní soubor, na kterém je ukázána neúměrnost velikosti vstupního souboru a množství elementů v databázi, je knihovna JBoss společnosti Red Hat (`jboss-fsw-installer-6.0.0.GA-redhat-4`). Tato knihovna byla zpracována za 8 hodin 40 minut a 52 sekund, přestože byl vstupní soubor velikosti 412,1 MB. Při tom bylo vytvořeno přes 135 tisíc vrcholů a 666667 vztahů.



Obrázek 7.1: Závislost doby zpracování vstupního souboru na velikosti databáze.

Z grafu 7.1 je vidět zvyšování nárůstu doby při každém zvýšení velikosti vstupního souboru. Lze tedy odhadovat, že složitost je linearitmická.

7.2 Složitost zpracování dotazů

V rámci tohoto experimentu je sledována závislost doby zpracování dotazu a vypsání výsledku. Hodnota těchto metrik byla pozorována v závislosti na množství vrcholů uložených v grafové databázi. Jako testovací dotazy byly použity příklady ze seznamu v příloze F. V tomto experimentu jsou dotazy zastoupeny jejich pořadovým číslem ze zmiňovaného seznamu. Každý dotaz byl vyhodnocován na vybraných pěti databázích vytvořených v prvním experimentu. Pro každou databázi je tak většina dotazů upravena, aby vyhledávaly existující elementy (jedná se zejména o konkrétní názvy nebo obsahy vlastností vrcholů). Zde je každý dotaz proveden nad každou databází právě třikrát. Výsledný čas, který je zapsaný v tabulce 7.2, je průměrem těchto hodnot. Měření času bylo opět provedeno pomocí příkazu `time`, kde byla odečtena hodnota `real`.

Obrázek 7.2 ukazuje průměrný čas dotazování nad jednotlivými databázemi. Z tohoto grafu lze odhadovat liearitmickou složitost.

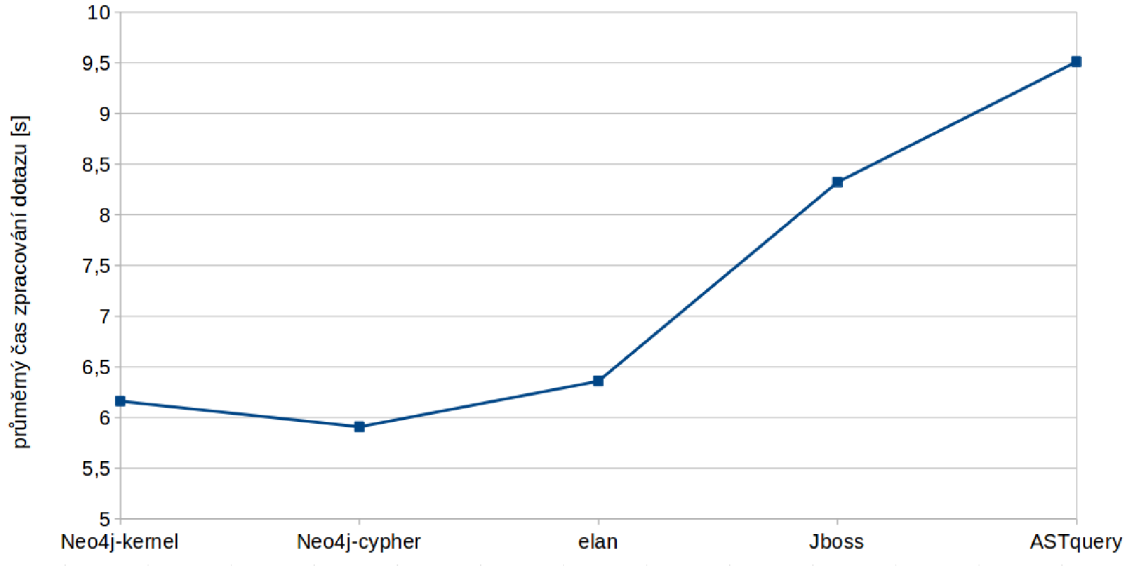
Graf 7.3, který zobrazuje časy jednotlivých dotazů strávené nad všemi databázemi, se může zdát chaotický. Názorně však ukazuje, že všechny dotazy měly čas vykonání větší než cca 5 sekund. To může být způsobeno konstantní dobou startování grafové databáze Neo4J. Pokud by se tento čas odečetl, potom by u nejmenší databáze skončila většina dotazů do jedné sekundy a u databáze střední velikosti do dvou sekund. Nejkratší čas zpracování dotazu byl 4,712 sekund. Jednalo se o dotaz číslo 8 nad databází vytvořenou z projektu „neo4j-cypher.jar“. Dotaz s největší dobou zpracování je příklad číslo 7 nad databází vytvořenou z `JBoss` s časem 14,422 sekundy. Tento dotaz má za výsledek největší množství výsledných záznamů.

	ASTquery.jar		elan/		neo4j-kernel.jar		JBoss		neo4j-cypher.jar	
č.d.	Výs.	Čas	Výs.	Čas	Výs.	Čas	Výs.	Čas	Výs.	Čas
1	2183	12,31	629	7,33	195	6,34	3074	11,605	677	5,778
2	3	7,93	1	5,56	12	6,07	1	6,963	2	5,145
3	1	8,1	2	5,44	1	5,64	1	6,488	1	5,348
4	2	8,07	1	5,85	3	6,01	3	7,458	3	6,031
5	2010	13,64	507	7,81	309	7,58	1636	11,33	3	7,499
6	3131	13,5	121	7,1	674	6,86	2833	10,902	601	7,009
7	5988	14,05	13	6,74	226	5,95	9435	14,422	145	5,503
8	44	6,68	17	5,17	12	5,25	34	6,305	10	4,712
9	2	6,5	0	5,54	1	5,24	1	6,64	1	4,857
10	60	6,61	0	5,14	7	5,13	42	8,051	0	4,909
11	1	8,58	0	5,83	1	5,71	2	8,176	1	6,208
12	2	9,89	0	6,33	0	6,27	192	8,801	0	5,85
13	50	6,34	0	4,93	116	5,34	43	5,602	403	5,176
14	1808	11,49	243	6,67	309	6,61	1335	11,094	255	6,462
15	2747	12,38	452	7,42	456	6,29	2447	10,371	426	7,293
16	1	12,59	1	8,04	1	7,67	1	9,627	1	7,803
17	106	11,32	32	7,2	7	8,59	36	9,72	2	6,425
18	106	11,88	32	8,16	7	7,63	36	10,945	2	7,076
19	128	7,66	37	6,18	7	7,04	40	6,943	2	5,435
20	3	8,58	1	6,3	6	5,54	3	7,026	1	5,262
21	3	8,66	1	6,72	6	5,66	3	6,836	1	5,255
22	3	7,44	1	5,65	6	5,49	3	6,317	1	5,163
23	3	7,6	1	6,44	6	5,7	3	6,782	1	5,542
24	1	7,08	0	5,36	9	5,65	1	5,924	2	5,447
25	1	6,54	0	5,04	9	5,34	1	5,864	2	5,957
26	1	8,72	2	6,03	1	5,58	6	6,615	1	5,091
27	1	13,93	2	8,09	1	7,51	6	11,555	1	8,541
28	0	9,09	0	6,21	0	5,66	0	7,169	0	5,668
29	142	9,35	23	6,96	25	5,92	111	7,396	7	5,163
30	194	8,78	26	5,5	62	5,55	125	6,708	23	5,636

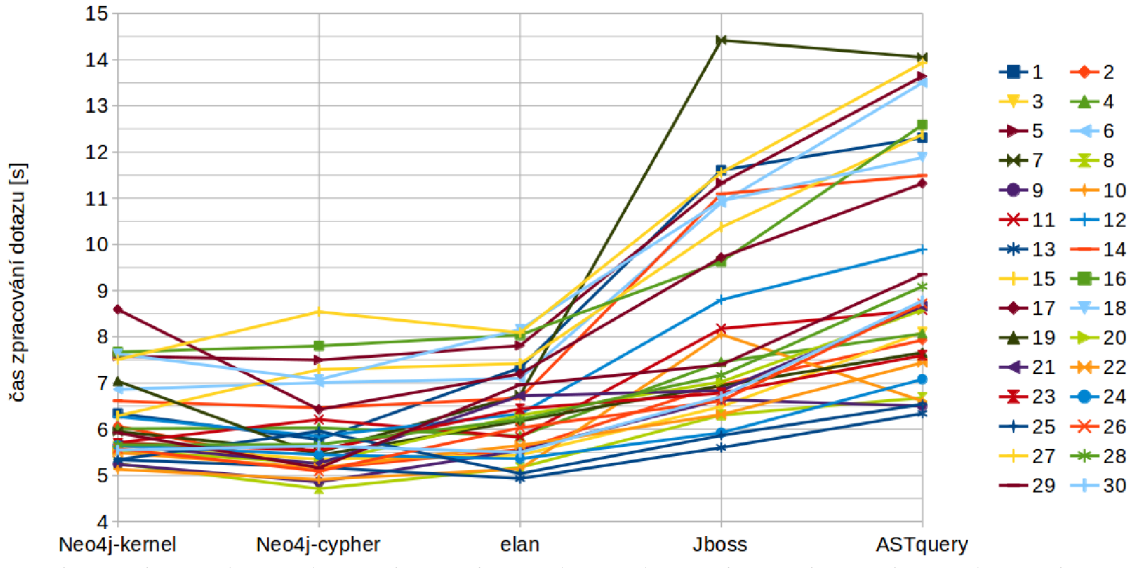
Tabulka 7.2: Tabulka výsledků ze zpracování dotazů.

V sadě příkladů jsou skupiny takových dotazů, které vyhledávají ekvivalentní výsledky, přestože mají rozdílný zápis. Těmito dotazy je sledována efektivita dotazování podle posloupnosti kroků dotazu. V mnoha případech je dotazováno způsobem „od toho, co je známo“. Což znamená, že se hned z počátku omezí množina mezivýsledků na co nejmenší mohutnost. Další kroky dotazování jsou potom mnohem rychlejší.

Přestože by měly být dotazy ekvivalentní, počet výsledků může být rozdílný. To je zapříčiněno tím, že dotazy mohou vracet duplicitní výsledky. Například pokud je prvně dotazováno na všechna pole a potom na jejich datové typy. Potom bude výsledných záznamů stejně jako počet použitých polí v programu a každý záznam bude určovat datový typ odpovídajícího pole. Je možné se zeptat v opačném pořadí. Tedy v první řadě na všechny datové typy a až posléze na to, zda jsou typem nějakého pole. V tomto případě zůstanou

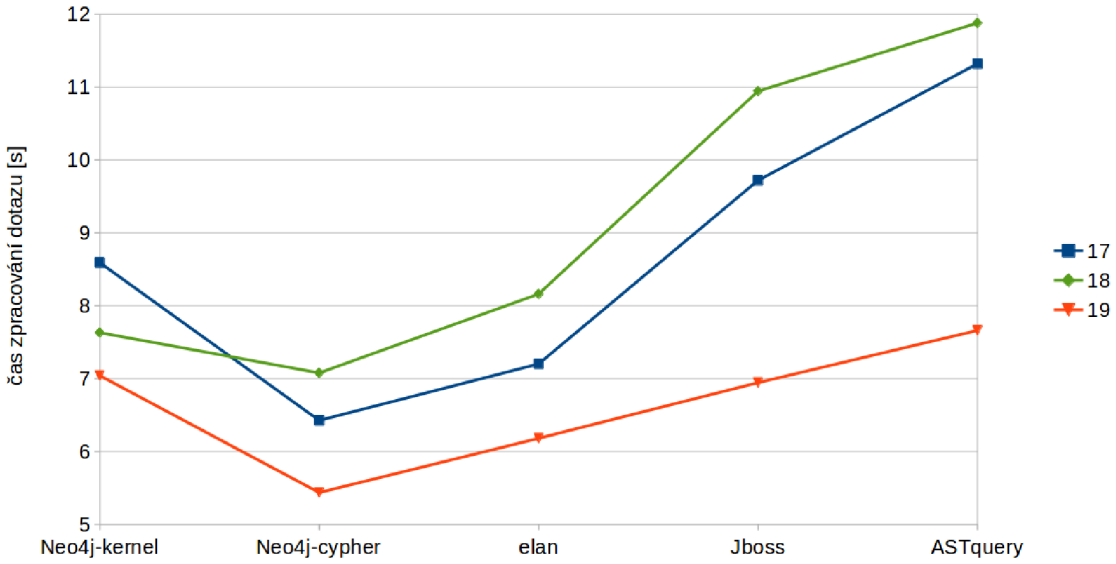


Obrázek 7.2: Průměrný čas strávený dotazy nad jednotlivými databázemi.



Obrázek 7.3: Čas strávený jednotlivými dotazy nad všemi databázemi. Dotazy jsou zastoupeny jejich pořadovým číslem ze seznamu z přílohy F. Grafem je demonstrována především spodní a horní hranice času stráveného zpracováním dotazu.

jenom takové typy, které jsou typy nějakého pole. Výsledkem druhého dotazu bude tak pravděpodobně méně záznamů. Rychlost vypracování dotazů však může být v těchto případech opačná, protože ve druhém případě se pracuje z počátku s mnohem větší množinou mezivýsledků nežli v prvním případě. Důležité je, jak se k dané množině datových typů dotaz dopracuje. Následující experimenty prověřují tyto případy dotazování a zobrazují grafy rychlostí zpracování dotazů nad databázemi vytvořených z různých vstupních souborů.



Obrázek 7.4: Srovnání ekvivalentních dotazů se rozdílným zápisem. Srovnání dotazů č. 17, 18 a 19.

Na obrázku 7.4 je srovnání rychlosti zpracování dotazů číslo 17, 18 a 19. Vybrané dotazy vrací všechny metody, jejichž alespoň jeden parametr je typu pole celých čísel.

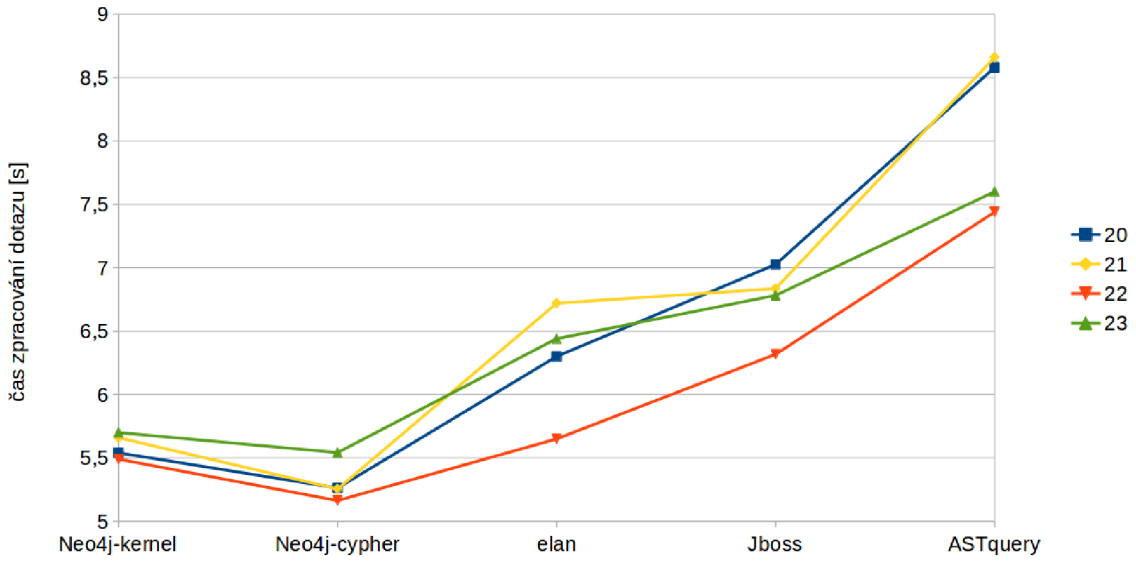
```
17. method[have_parameter/is_type/field/main_type[@name="int"]]
18. method[have_parameter[is_type[field[main_type[@name="int"]]]]]
19. int/main_typed_by/field/typed_by/is_parameter/method
```

V prvním případě je dotaz pokládán tak, že na počátku je vymezen typ požadovaného vrcholu. Filtr následně obsahuje omezení zapsané v krocích. Tento postup se ukazuje jako zcela neefektivní. Důvodem tak může být provádění celé podmínky filtru na všechny prvky množiny všech metod. Druhý dotaz obsahuje zanořené filtry. Vysoce mohutná množina metod je tak postupně redukována. Jako nejrychlejší dotaz ve všech případech se projevil dotaz číslo 19. V tomto případě je na počátku dotazu množina všech datových typů s názvem `int`, která se však v databázi vyskytuje pouze jednou. Dále je množina mezivýsledků postupně expandována.

První dva dotazy ve všech případech vrátily stejně mohutnou množinu výsledků. Třetí případ vrací mohutnější množinu. Ve výsledku byly obsaženy elementy, které jsou duplicitní.

Graf 7.5 znázorňuje výsledky doby zpracování dotazů číslo 20, 21, 22 a 23 na pěti databázích vytvořených z vybraných vstupních souborů.

```
20. class[extends/class[@name="jmeno.tridy"] or extends/class
    [@name="jmeno.tridy2"]]
21. class[extends[@name="jmeno.tridy" or @name="jmeno.tridy2"]]
22. (class[@name="jmeno.tridy"] union class[@name="jmeno.tridy2"])/
    extended_by
23. (jmeno.tridy union jmeno.tridy2)/extended_by
```



Obrázek 7.5: Srovnání ekvivalentních dotazů s rozdílným zápisem. Srovnání dotazů č. 20, 21, 22 a 23.

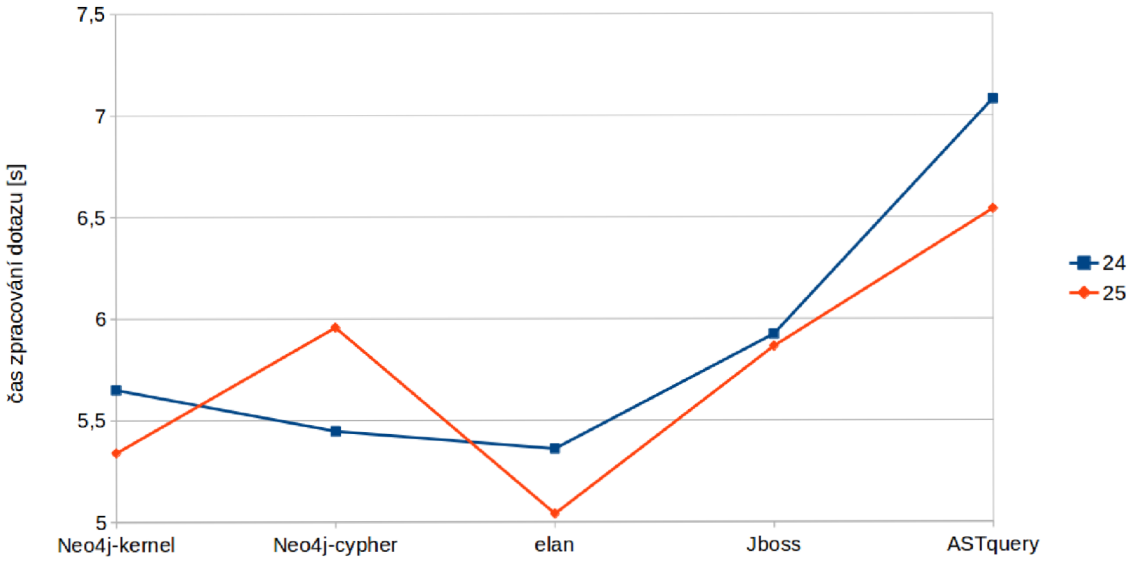
Výsledkem těchto dotazů jsou třídy, které dědí od třídy, jež má jméno `jmeno.tridy` nebo `jmeno.tridy2`. Tentokrát rozdíly v rychlosti dotazování nejsou tolik velké, jako v předchozím případě. Přesto se ve všech případech vyskytuje dotaz číslo 22 jako nejrychlejší. Přestože bylo očekáváno, že jazyková struktura zkracující vyhledávání elementu podle názvu bude efektivnější (dotaz číslo 23), nezkrácený zápis dosáhl lepších výsledků. Všechny dotazy vrátily pro každou databázi stejné množství výsledků.

Další srovnání rychlosti dotazování je prováděno na příkladech 24 a 25. Graf 7.6 ukazuje, že kromě jednoho případu byl dotaz číslo 25 rychlejší. Dotazy mají už poměrně komplexní strukturu. Příklady vrací třídy, jejichž metoda volá jinou metodu s anotací `@jmeno.anotace`, která obsahuje element `jmeno.elementu="cokoli"`. Tato metoda, jež je z jiné metody volána, musí být obsažena ve třídě, která implementuje rozhraní `jmeno.typu`.

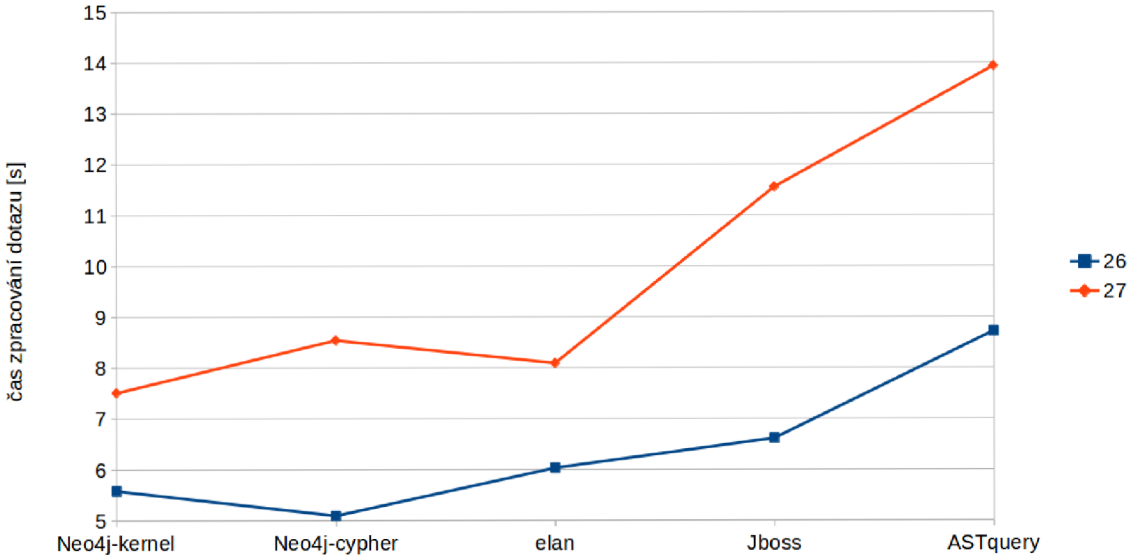
```
24. data type[@name="jmeno.typu"]/implemented_by/have_method[>annotated
    [@name="jmeno.anotace"]/have_element/is_value[@name="jmeno.elementu"]]/
    called_by/is_method
25. annotation[@name="jmeno.anotace"][have_element/is_value
    [@name="jmeno.elementu"]]/<annotated[is_method/implements
    [@name="jmeno.typu"]]/called_by/is_method
```

Oba dotazy tak postupují „od toho, co je známo“. Protože známe jména dvou elementů jazyka, každý dotaz tak zkouší brát jako „známo“ jeden z těchto elementů. Jelikož dávají lepší výsledky dotazy začínající s mezivýsledky s menší mohutností množiny, je v těchto dotazech klíčové, zda databáze obsahuje více anotací nebo datových typů. Toto může být příčinou opačného výsledku v jednom z případů, kdy druhý dotaz dopadl hůře. Dotazy pro každou databázi vrátily stejné množství výsledků.

Předposlední srovnání ekvivalentních dotazů s rozdílným zápisem je prováděno na příkladech číslo 26 a 27. Výsledek srovnání je vidět na grafu 7.7.



Obrázek 7.6: Srovnání ekvivalentních dotazů s rozdílným zápisem. Srovnání dotazů č. 24 a 25.

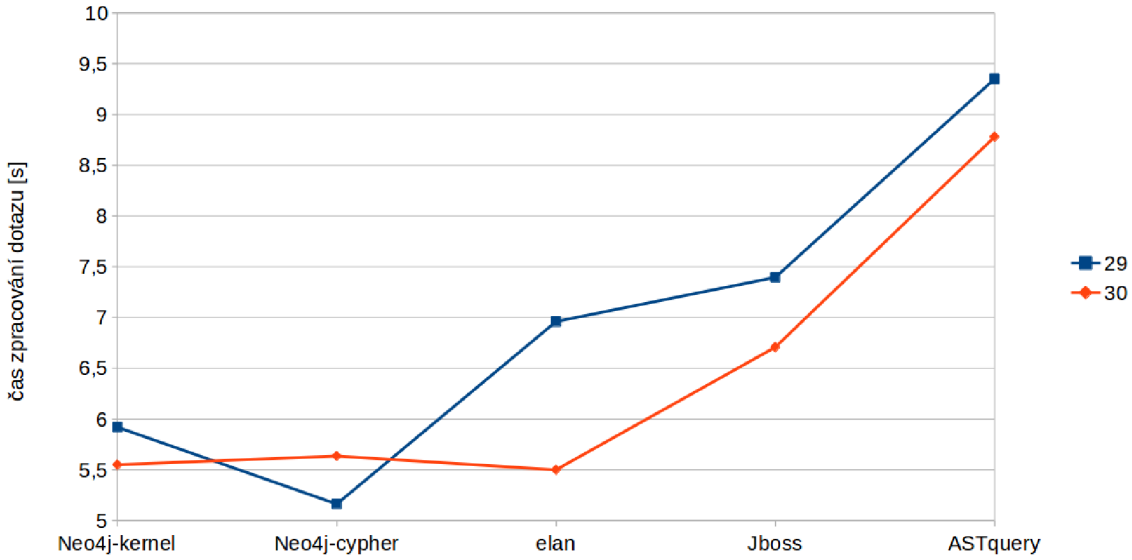


Obrázek 7.7: Srovnání ekvivalentních dotazů s rozdílným zápisem. Srovnání dotazů č. 26 a 27.

```
26. class[extended_by[@name="jmeno.tridy"]]/have_method
    [@name="jmeno.metody" AND have_parameter/is_type
    [@name="java.lang.String"]]/called_by/is_method
```

```
27. (method union constructor)[@name="jmeno.metody" AND have_parameter/
    is_type[@name="java.lang.String"] and is_method/extended_by
    [@name="jmeno.tridy"]]/called_by/is_method
```

Zmiňované dotazy jsou také velice komplexní struktury. Jejich výsledkem jsou všechny třídy, jejichž metody volají metodu se jménem `jmeno.metody` s alespoň jedním parametrem typu `String`. Tato metoda musí být deklarována ve třídě, od které dědí třída jménem `jmeno.tridy`. Rozdíly v čase zpracování dotazů jsou na všech databázích markantní. Důvodem je velice neefektivní dotazování v případě dotazu číslo 27. Tento dotaz totiž prvně získá množinu všech metod a konstruktorů, která má velkou mohutnost. Dále nad touto množinou provádí několik filtrů, které jsou spojeny operátorem `and`. S každým prvkem této množiny se provedou v podstatě 3 filtry. Naproti tomu první dotaz získá množinu všech tříd, od kterých dědí třída `jmeno.tridy`. Dle definice jazyka Java je zřejmé, že takováto třída může být pouze jedna. Potom je expanze této množiny velice rychlá. Pro každou databázi vrátily dotazy stejně mohutné množiny výsledků.



Obrázek 7.8: Srovnání ekvivalentních dotazů s rozdílným zápisem. Srovnání dotazů č. 29 a 30.

Objektem posledního srovnání jsou dotazy číslo 29 a 30. Výsledný graf lze nalézt na obrázku 7.8. Dotazy vypisují všechny třídy, které používají jako datový typ některého ze svých atributů vnitřní nebo anonymní třídu.

```
29. class[have_member/is_type[@anonym="true" or @inner="true"]]
30. (inner class union anonym class)/typed_by/<member/class
```

První dotaz začíná práci nad množinou všech tříd, zatímco druhý dotaz začíná s množinou vnitřních a anonymních tříd. Vnitřních tříd je zjevně ve většině experimentovaných případů méně, než „top level“ tříd. Výjimečný stav nastává u databáze vytvořené ze vstupního souboru `Neo4J-cypher.jar`, kde je zjevně vnitřních (včetně anonymních) tříd více.

Zajímavým úkazem těchto dvou dotazů je také značný rozdíl v mohutnosti výsledné množiny. První dotaz začíná množinou všech tříd, kde je zastoupena každá právě jednou, a dále ji postupně filtruje. Ve výsledku se tak nevyskytují duplicitní záznamy. Druhý dotaz se na konci dostává do situace, kdy požaduje všechny atributy, které jsou typované některou z vnitřních tříd. Pokud tedy existuje více atributů stejného typu (který je deklarovaný vnitřní třídou), bude i více záznamů ve výsledné množině, které budou duplicitní.

Bylo ukázáno, že dotazování má linearitmickou složitost vzhledem k množství elementů v databázi. Dále bylo rozebráno několik dotazů a jejich alternativní zápisy. Ekvivalentní dotazy byly srovnány z hlediska času zpracování. Z experimentů s ekvivalentními dotazy lze poznamenat, že efektivním zápisem dotazu lze dosáhnout až 40% zrychlení dotazování.

Kapitola 8

Závěr

Hlavním smyslem této práce byl návrh dotazovacího jazyka nad abstraktním syntaktickým stromem Java a implementace nástroje využívajícího tento jazyk. Cíl práce byl splněn.

Seznámil jsem se s nástroji pro analýzu bajtkódu a zdrojových textů jazyka Java. V kapitole 2.3 jsou vypsány dostupné dekompilátory a analyzátory jazyka Java. Vzhledem ke kvalitě, aktuálnosti a kompatibilitě byl blíže popsán nástroj Procyon, který byl využit při implementaci cílové aplikace. Analýza grafových databází je sepsána v kapitole 2.2. Podle této rešerše byla vybrána knihovna Neo4J, která byla použita v implementaci aplikace. Dle předchozí analýzy byl navržen dotazovací jazyk. Návrh jazyka je sepsán v kapitole 3. Dále byl vytvořen nástroj umožňující dotazování nad Java AST. Implementace tohoto nástroje je probírána především v kapitole 6. Tato kapitola líčí tvorbu aplikace na základě popisu implementace dotazovacího jazyka z kapitoly 4 a podrobné studie jazykových konstrukcí jazyka Java. Dále je tato kapitola založena na popisu uložení jednotlivých entit do grafové databáze z kapitoly 5. Dokumentace API výsledného nástroje lze nalézt v příloze C. V kapitole o návrhu dotazovacího jazyka jsou popsány jazykové konstrukce na několika demonstrativních příkladech. Pro testování a pro příkladový popis jazyka bylo navíc vytvořeno 30 komplexních dotazů (viz příloha F). Na těchto dotazech byl výsledný nástroj testován a byl vynesen graf časové složitosti zpracování zmiňovaných dotazů. Dále byl také vytvořen graf závislosti zpracování vstupních souborů na velikosti databáze. Výsledky experimentů a testování jsou zaznamenány v kapitole 7.

Experimenty ukázaly, že vytváření grafu ze vstupních souborů má linearitmickou složitost vzhledem k množství elementů v databázi. Například při zpracování projektu, který byl vytvořen v rámci této práce, který má 12,4 MB, vzniklo v databázi přes 200 tisíc vrcholů a přes 1 milion vztahů. Doba zpracování tohoto souboru byla více jak 16 hodin a 39 minut na průměrném počítači. Dále bylo ukázáno, že způsob dotazování může mít zásadní vliv na rychlost provedení dotazu. Nad zmiňovanou databází dosáhl nejdéle trvající dotaz 14,05 sekund a nejkratší dotaz byl proveden za 6,5 sekundy. V rámci experimentů byly také porovnávány ekvivalentní dotazy s rozdílným zápisem. Bylo zjištěno, že se vyplatí zapisovat dotazy tak, aby byla množina mezivýsledků redukována co nejdříve.

8.1 Možné pokračování práce

Z experimentů je zřejmé, že nejvíce času spotřebuje proces zpracování vstupních souborů a plnění databáze. Tento proces je vytvořen pro sekvenční zpracování a proto neefektivně využívá procesorů, jež disponují větším počtem jader. Proto by bylo vhodné zparalelizovat

proces zpracování souborů a plnění databáze, aby mohl program využít více jader a zrychlit tak násobně svůj průběh.

Dalším rozšířením mohou být nové jazykové konstrukce dotazovacího jazyka. Zejména by bylo vhodné umožnění zápisu dalšího výrazu místo konstanty v podmínce vlastnosti vrcholu. V současné chvíli je syntaxe této konstrukce `@ID=String`. Po rozšíření by tak měla být umožněna konstrukce `@ID=EXPR`, což by rozšířilo možnosti dotazování.

Poslední navrhovanou možností pokračování práce je aktualizace aplikace pro akceptování kódu nově vzniklé verze jazyka Java (Java 8). Toto pokračování je však závislé na používaných nástrojích (především Procyon), proto je tato revize kódu možná až po patřičné aktualizaci nezbytných knihoven.

Literatura

[1] BCEL. http://commons.apache.org/proper/commons-bcel/, 2014 [cit. 29.12.2014].

[2] DB-Engines Ranking of Graph DBMS. http://db-engines.com/en/ranking/graph+dbms, 2014 [cit. 29.12.2014].

[3] DrJava. http://www.drjava.org/, [cit. 12.5.2015].

[4] JBoss Developer. http://www.jboss.org/downloads/, [cit. 12.5.2015].

[5] Frames. https://github.com/tinkerpop/frames/wiki, [cit. 1.5.2015].

[6] XPath Tutorial. http://www.w3schools.com/xpath/, [cit. 2.1.2015].

[7] Elan. https://tla.mpi.nl/tools/tla-tools/elan/download/, [cit. 2.5.2015].

[8] ANTLR. http://www.antlr.org/, [cit. 28.4.2015].

[9] Blueprints. https://github.com/tinkerpop/blueprints/wiki, [cit. 28.4.2015].

[10] Candle. https://github.com/bradsdavis/candle-decompiler, [cit. 29.12.2014].

[11] CRF (Class File Reader). http://www.benf.org/other/cfr/, [cit. 29.12.2014].

[12] DJ. http://www.neshkov.com/dj.html, [cit. 29.12.2014].

[13] EDJC (Emilio's Java Decompiler). http://sourceforge.net/projects/ejdc/, [cit. 29.12.2014].

[14] FernFlower. http://forum.xda-developers.com/showthread.php?t=2029842, [cit. 29.12.2014].

[15] GraphBase. http://graphbase.net/, [cit. 29.12.2014].

[16] InfiniteGraph. http://infinitegraph.com/, [cit. 29.12.2014].

[17] InfoGrid. http://infogrid.org/trac/, [cit. 29.12.2014].

[18] JaD (JAva Decompiler). http://varaneckas.com/jad/, [cit. 29.12.2014].

[19] JBVD (Java Bytecode Viewer & Decompiler). http://jbdec.sourceforge.net/, [cit. 29.12.2014].

[20] JD. http://jd.benow.ca/, [cit. 29.12.2014].

[21] Krakatau. https://github.com/Storyyeller/Krakatau/, [cit. 29.12.2014].

[22] Mocha. http://www.brouhaha.com/~eric/software/mocha/, [cit. 29.12.2014].

[23] Neo4J. http://neo4j.com/, [cit. 29.12.2014].

[24] Titan. http://thinkaurelius.github.io/titan/, [cit. 29.12.2014].

[25] Bečvář, M.: *Modelování v grafové nerelační databázi*. Diplomová práce, Západočeská univerzita v Plzni, 2013.

[26] Gosling, J., Joy, B., Bracha, G., Buckley, A.: *The Java Language Specification: Java SE 7 Edition*. Oracle America, Inc, únor 2013, http://docs.oracle.com/javase/specs/jls/se7/jls7.pdf [cit. 27.4.2015].

[27] Kosek, J.: *XML pro každého*. Grada Publishing, 2000, iSBN 80-7169-860-1.

[28] LTD., O. T.: OrientDB. http://www.orientechnologies.com/, [cit. 29.12.2014].

[29] Neubauer, P.: Graph Databases, NOSQL and Neo4j. http://www.infoq.com/articles/graph-nosql-neo4j, 2010 [cit. 29.12.2014].

[30] Oracle: Oracle NoSQL Database. http://www.oracle.com/technetwork/products/nosqldb/overview/index.html, [cit. 29.12.2014].

[31] Oracle: Oracle Spatial and Graph. http://www.oracle.com/technetwork/database/options/spatialandgraph/overview/index.html, [cit. 29.12.2014].

[32] Parr, T.: *The Definitive ANTLR 4 Reference*. Pragmatic Bookshelf, 2013, iSBN 978-1934356999.

[33] Parr, T.: Getting Started with ANTLR v4. https://theantlrguy.atlassian.net/wiki/display/ANTLR4/Getting+Started+with+ANTLR+v4, [cit. 28.4.2015].

[34] Sqrrl Data, I.: Sqrrl Enterprise. http://sqrrl.com/, [cit. 29.12.2014].

[35] Strobel, M.: Procyon. https://bitbucket.org/mstrobel/procyon, [cit. 29.12.2014].

[36] Žižka, O.: Java Decompilers, 2014. https://developer.jboss.org/people/ozizka/blog/2014/05/06/java-decompilers-a-sad-situation-of, 2014 [cit. 29.12.2014].

Příloha A

Obsah CD

- `tex` (složka se zdrojovým textem dokumentu diplomové práce)
- `xbilek16.pdf` (dokument diplomové práce)
- `Java-AST-query-language`
  - `source` (složka obsahující zdrojové kódy výsledné aplikace, potřebné knihovny a pro úplnost soubor s gramatikou pro ANTLR)
  - `tests` (složka obsahující sadu testů a skript (`test.sh`) pro spuštění této sady testů)
  - `ASTquery.jar` (výsledná aplikace dotazovacího jazyka)

Příloha B

Manual

Aplikace je určena pro použití v terminálu. Při spuštění aplikace bez parametrů se vypíše do terminálu nápověda.

```
 -query <query> [-source <dirs>] [-database-directory <dir>] [-debug]
```

Jediné povinné parametry jsou `-query <query>`. Tyto parametry značí konkrétní dotaz, kde se řetězec `<query>` za požadovaný řetězec s dotazem nahradí.

Dalším možným párem parametrů jsou `-source <dirs>`, pomocí kterého lze určit soubory pro dekompilaci a převedení do grafu. Je možné takto zadat soubory s koncovkou jar, class nebo složky, ze kterých se použijí soubory s již zmiňovanými koncovkami. Dále lze zadat více zdrojů, přičemž se jednotlivé adresy oddělují středníky. Pokud nebudou tyto parametry zadané, nebude prováděna dekompilace a převedení na graf. Dotaz se provede nad existující databází.

Specifikovat adresu databáze je možné pomocí parametrů `-database-directory <dir>`. V případě, že nebudou tyto parametry zadány použije se defaultní adresa, která je nastavena na `/tmp/ASTquery/graph.db`.

Poslední parametr je `-debug`. Jestliže je tento parametr zadán, jsou kontrolní výpisy programu obsáhlejší.

Je nutné, aby za parametrem `-query` následoval dotaz, za parametrem `-source` seznam souborů (nebo složek) a za parametrem `-database-directory` adresa databáze. Na dalším pořadí parametrů nezáleží.

Příklady spuštění aplikace:

```
$ ASTquery.jar -query "class[>m]"

$ ASTquery.jar -query "class and interface" -source
  "/home/user/sources/;/dirs/file.class"

$ ASTquery.jar -query "method[>t[String]]"
  -database-directory "/tmp/graph.dp"

$ ASTquery.jar -debug -query "(class and interface)/>m"
  -source "/home/user/sources/" -database-directory "/tmp/graph.dp"
```

Příloha C

ASTquery API

Protože je cílem tohoto projektu vytvořit především knihovnu, která bude umožňovat dotazování nad Java AST, je následně popsáno API této knihovny. Hlavní třída nese název `ASTquery`. Metody poskytované touto třídou jsou dále vypsány.

Metoda	**Popis**
`ASTquery()`	Konstruktor třídy nastavuje defaultní hodnoty, které je možné následně měnit k tomu určenými metodami.
`String getAnswer(String, boolean)`	Tuto metodu je možné zavolat až po připojení k databázi. Jestliže nejsou zadány žádné soubory ke zpracování do grafu, ihned se zpracuje dotaz. V opačném případě se prvně spustí převod zdrojových souborů do grafu a zpracování dotazu započne poté.
`void setDBdirectory(String)`	Metoda umožňuje nastavit adresu grafové databáze. V parametru je požadovaná cesta.
`void setSource(String)`	Nastavení zdroje souborů ke zpracování do grafu. V parametru může být více zdrojů. Jednotlivé adresy se oddělují symbolem „;“.
`DBConnector connect()`	Metoda pro vytvoření připojení ke grafové databázi. Jestliže nebude nastavena cesta k této databázi, použije se defaultní cesta.
`void disconnect()`	Po ukončení práce s databází je vyžadováno odpojení.

Tabulka C.1: Kompletní přehled metod API ASTquery.

Příloha D

Gramatika dotazovacího jazyka pro ANTLR

```
grammar ASTquery;

and         : AND ;
or          : OR;
not         : NOT;
intersection: INTERSECTION;
union       : UNION;
AND         : [aA] [nN] [dD] ;
OR          : [oO] [rR];
NOT         : [nN] [oO] [tT];
INTERSECTION: [iI][nN][tT][eE][rR][sS][eE][cC][tT][iI][oO][nN];
UNION       : [uU][nN][iI][oO][nN];
slash: '/';

SPACES: SPACE+;
SPACE : (' ' | '\t' | '\r');

r_extend
  : 'extends'
  | '>extends'
  | '>e'
  ;

r_extended
  : 'extended_by'
  | '<extends'
  | '<e'
  ;

r_implements
  : 'implements'
  | '>implements'
```

```
  | '>i'
  ;

r_implemented_by
  : 'implemented_by'
  | '<implements'
  | '<i'
  ;

r_uses
  : 'uses'
  | '>uses'
  | '>u'
  ;

r_used_by
  : 'used_by'
  | '<uses'
  | '<u'
  ;

r_calls
  : 'calls'
  | '>calls'
  | '>c'
  ;

r_called_by
  : 'called_by'
  | '<calls'
  | '<c'
  ;

r_type
  : 'type_of'
  | '>type'
  | '>t'
  ;

r_typed_by
  : 'typed_by'
  | '<type'
  | '<t'
  ;

r_have_anonym
  : 'have_anonym'
  | '>anonym'
```

```
  ;

r_is_anonym
  : 'is_anonym'
  | '<anonym'
  ;

r_have_inner
  : 'have_inner'
  | '>inner'
  ;

r_is_inner
  : 'is_inner'
  | '<inner'
  ;

r_have_member
  : 'have_member'
  | '>member'
  ;

r_is_member
  : 'is_member'
  | '<member'
  ;

r_have_member_with_extends
  : '>>member'
  ;

r_is_member_with_extends
  : '<<member'
  ;

r_have_method
  : 'have_method'
  | '>method'
  | '>m'
  ;

r_is_method
  : 'is_method'
  | '<method'
  | '<m'
  ;

r_have_method_with_extends
```

```
  : '>>method'
  | '>>m'
  ;

r_is_method_with_extends
  : '<<method'
  | '<<m'
  ;

r_have_param
  : 'have_parameter'
  | '>parameter'
  | '>p'
  ;

r_is_param
  : 'is_parameter'
  | '<parameter'
  | '<p'
  ;

r_main_type
  : 'main_type'
  | '>main_type'
  ;

r_main_typed_by
  : 'main_typed_by'
  | '<main_type'
  ;

r_generic_param
  : 'have_generic_parameter'
  | '>generic_parameter'
  ;

r_generic_param_by
  : 'is_generic_parameter'
  | '<generic_parameter'
  ;

r_annotated
  : 'have_annotated'
  | '>annotated'
  | '>a'
  ;

r_annotated_by
```

```
  : 'is_annotated_by'
  | '<annotated'
  | '<a'
  ;

r_annotated_with_extends
  : '>>annotated'
  | '>>a'
  ;

r_annotated_by_with_extends
  : '<<annotated'
  | '<<a'
  ;

r_element
  : 'have_element'
  | '>element'
  ;

r_element_by
  : 'is_element_by'
  | '<element'
  ;

r_throws
  : 'throws'
  | '>throws'
  ;

r_throws_by
  : 'throws_by'
  | '<throws'
  ;

r_call_throws
  : 'call_throws'
  | '>call_throws'
  ;

r_call_throws_by
  : 'call_throws_by'
  | '<call_throws'
  ;

r_catch
  : 'catch'
  | '>catch'
```

```
  ;

r_catch_by
  : 'catch_by'
  | '<catch'
  ;

r_value
  : 'is_value'
  | '>value'
  ;

r_value_by
  : 'value_by'
  | '<value'
  ;


node_type
  : 'class'
  | 'interface'
  | 'annonym class'
  | 'inner class'
  | 'data type'
  | 'generic type'
  | 'type parameter'
  | 'field'
  | 'enum'
  | 'annotation'
  | 'annotation type'
  | 'member'
  | 'method'
  | 'constructor'
  | 'parameter'
  | 'element'
  ;

// not include relationship
query
  : node_type
  | ('!')? ID    // node_type (ID = name of node) and his child in
   extend relationship
  | query slash query2
  | query '[' expr ']'
  | query SPACES union SPACES query
  | query SPACES intersection SPACES query
  | '(' query ')'
  ;
```

```
query2
  : node_type
  | ('!')? ID    // node_type (ID = name of node) and his child in
  extend relationship
  | query2 slash query2
  | relationship
  | query2 '[' expr ']'
  | query2 SPACES union SPACES query2
  | query2 SPACES intersection SPACES query2
  | '(' query2 ')'
  ;

// not include relationship
expr
  : node_type
  | '!' ID    // node_type (ID = name of node) and his child in extend
  relationship example: class[!MyClass/have_method/method[getName]]
  | ID        // name of node expample: class[MyClass]
  | expr slash expr2
  | relationship
  | expr '[' expr ']'
  | not SPACES expr
  | expr SPACES and SPACES expr
  | expr SPACES or SPACES expr
  | '@' ID '=' STRING  // example class[@name="MyClass"]
  | '(' expr ')'
  ;

expr2
  : node_type
  | '!' ID    // node_type (ID = name of node) and his child in extend
  relationship example: class[!MyClass/have_method/method[getName]]
  | ID        // name of node expample: class[MyClass]
  | expr2 slash expr2
  | relationship
  | expr2 '[' expr2 ']'
  | expr2 SPACES union SPACES expr2
  | expr2 SPACES intersection SPACES expr2
  | '(' expr2 ')'
  ;


STRING: '"' CHAR* '"';

fragment
CHAR  : ~["] ;
```

```
ID : JavaLetter JavaLetterOrDigit*;

fragment
JavaLetter
  : [a-zA-Z$_] // these are the "Java letters" below 0xFF
  | // covers all characters above 0xFF which are not a surrogate
  ~[\u0000-\u00FF\uD800-\uDBFF]
  {Character.isJavaIdentifierStart(_input.LA(-1))}?
  | // covers UTF-16 surrogate pairs encodings for U+10000 to U+10FFFF
  [\uD800-\uDBFF] [\uDC00-\uDFFF]
  {Character.isJavaIdentifierStart(Character.toCodePoint((char)_input.LA(-2),
  (char)_input.LA(-1)))}?
  ;

fragment
JavaLetterOrDigit
  : [a-zA-Z0-9$_.] // these are the "Java letters or digits" below 0xFF
  | // covers all characters above 0xFF which are not a surrogate
  ~[\u0000-\u00FF\uD800-\uDBFF]
  {Character.isJavaIdentifierPart(_input.LA(-1))}?
  | // covers UTF-16 surrogate pairs encodings for U+10000 to U+10FFFF
  [\uD800-\uDBFF] [\uDC00-\uDFFF]
  {Character.isJavaIdentifierPart(Character.toCodePoint((char)_input.LA(-2),
  (char)_input.LA(-1)))}?
  ;

relationship
  : r_extend
  | r_extended
  | r_implements
  | r_implemented_by
  | r_uses
  | r_used_by
  | r_calls
  | r_called_by
  | r_type
  | r_typed_by
  | r_have_anonym
  | r_is_anonym
  | r_have_inner
  | r_is_inner
  | r_have_member
  | r_is_member
  | r_have_member_with_extends
  | r_is_member_with_extends
  | r_have_method
  | r_is_method
  | r_have_method_with_extends
```

```
  | r_is_method_with_extends
  | r_have_param
  | r_is_param
  | r_main_type
  | r_main_typed_by
  | r_generic_param
  | r_generic_param_by
  | r_annotated
  | r_annotated_by
  | r_annotated_with_extends
  | r_annotated_by_with_extends
  | r_element
  | r_element_by
  | r_throws
  | r_throws_by
  | r_call_throws
  | r_call_throws_by
  | r_catch
  | r_catch_by
  | r_value
  | r_value_by
  ;

WS : [\n]+ -> skip ; // skip newlines
```

Příloha E

Zkratky v dotazovacím jazyce nad Java AST

Klasický zápis	Zkrácený zápis	Poznámky
`class[@name="jmeno.elementu"]`	`class[jmeno.elementu]`	Stejně lze zkrátit zápis pro libovolný typ vrcholu.
`jmeno.elem/>e or jmeno.elem`	`!jmeno.elem`	
`class/extends`	`class/>extends` nebo `class/>e`	Lze použít pro všechny typy vrcholů, kde je tento vztah relevantní.
`class/extended`	`class/<extends` nebo `class/<e`	Lze použít pro všechny typy vrcholů, kde je tento vztah relevantní.
`class/implements`	`class/>implements` nebo `class/>i`	Lze použít pro všechny typy vrcholů, kde je tento vztah relevantní.
`class/implemented_by`	`class/<implements` nebo `class/<i`	Lze použít pro všechny typy vrcholů, kde je tento vztah relevantní.
`class/uses`	`class/>uses` nebo `class/>u`	Lze použít pro všechny typy vrcholů, kde je tento vztah relevantní.
`class/used_by`	`class/<uses` nebo `class/<u`	Lze použít pro všechny typy vrcholů, kde je tento vztah relevantní.

Tabulka E.1: Seznam možných zkratek v dotazovacím jazyce nad Java AST.

Klasický zápis	Zkrácený zápis	Poznámky
`method/calls`	`method/>call` nebo `method/>c`	Lze použít pro všechny typy vrcholů, kde je tento vztah relevantní.
`method/called_by`	`method/<call` nebo `method/<c`	Lze použít pro všechny typy vrcholů, kde je tento vztah relevantní.
`method/type`	`method/>type` nebo `method/>t`	Lze použít pro všechny typy vrcholů, kde je tento vztah relevantní.
`method/typed_by`	`method/<type` nebo `method/<t`	Lze použít pro všechny typy vrcholů, kde je tento vztah relevantní.
`class/have_anonym`	`class/>anonym`	Lze použít pro všechny typy vrcholů, kde je tento vztah relevantní.
`class/is_anonym`	`class/<anonym`	Lze použít pro všechny typy vrcholů, kde je tento vztah relevantní.
`class/have_inner`	`class/>inner`	Lze použít pro všechny typy vrcholů, kde je tento vztah relevantní.
`class/is_inner`	`class/<inner`	Lze použít pro všechny typy vrcholů, kde je tento vztah relevantní.
`class/have_member`	`class/>member`	Lze použít pro všechny typy vrcholů, kde je tento vztah relevantní.
`class/is_member`	`class/<member`	Lze použít pro všechny typy vrcholů, kde je tento vztah relevantní.
`class/have_method`	`class/>method` nebo `class/>m`	Lze použít pro všechny typy vrcholů, kde je tento vztah relevantní.
`class/is_method`	`class/<method` nebo `class/<m`	Lze použít pro všechny typy vrcholů, kde je tento vztah relevantní.
`method/have_parameter`	`method/>parameter` nebo `method/>p`	Lze použít pro všechny typy vrcholů, kde je tento vztah relevantní.
`method/is_parameter`	`method/<parameter` nebo `method/<p`	Lze použít pro všechny typy vrcholů, kde je tento vztah relevantní.

Tabulka E.2: Seznam možných zkratek v dotazovacím jazyce nad Java AST.

Klasický zápis	**Zkrácený zápis**	**Poznámky**
`field/main_type`	`field/>main_type`	Lze použít pro uzly pole a generického typu.
`field/main_typed_by`	`field/<main_type`	Lze použít pro uzly pole a generického typu.
generic type/have_generic_parameter	generic type/ >generic_parameter	Lze použít pouze pro uzly generického typu.
generic type/is_generic_parameter	generic type/ <generic_parameter	Lze použít pouze pro uzly generického typu.
`class/annotated`	`class/>annotated` nebo `class/>a`	Lze použít pro všechny typy vrcholů, kde je tento vztah relevantní.
`class/annotated_by`	`class/<annotated` nebo `class/<a`	Lze použít pro všechny typy vrcholů, kde je tento vztah relevantní.
`annotation/have_element`	`annotation/>element`	Lze použít pouze pro vrchol anotace.
`annotation/is_element_by`	`annotation/<element`	Lze použít pouze pro vrchol anotace.
`method/throws`	`method/>throws`	Lze použít pouze pro vrchol metoda.
`method/throws_by`	`method/<throws`	Lze použít pouze pro vrchol metoda.
`method/call_throws`	`method/>call_throws`	Lze použít pouze pro vrchol metoda.
`method/call_throws_by`	`method/<call_throws`	Lze použít pouze pro vrchol metoda.
`method/catch`	`method/>catch`	Lze použít pouze pro vrchol metoda.
`method/catch_by`	`method/<catch`	Lze použít pouze pro vrchol metoda.
`element/is_value`	`element/>value`	Lze použít pouze pro vrchol hodnota.
`element/value_by`	`element/<value`	Lze použít pouze pro vrchol hodnota.

Tabulka E.3: Seznam možných zkratek v dotazovacím jazyce nad Java AST.

Příloha F

Sada příkladů dotazovacího jazyka ASTquery

1. Výsledkem dotazu jsou všechny metody, které mají návratovou hodnotu datového typu `String`. Tento dotaz demonstruje zanoření filtru.

   ```
   class/have_method[is_type[java.lang.String]]
   ```

2. Na výstupu bude množina metod, z jejichž těla je volána metoda s názvem `jmeno.metody`. Demonstrace použití vztahů s opačnou orientací.

   ```
   method[@name="jmeno.metody"]/called_by
   ```

3. Vrátí průnik množiny všech veřejných tříd a množiny obsahující třídu s názvem `jmeno.tridy` a její potomky. Předvedení použití operátoru pro průnik a operátoru „!“.

   ```
   class[@access="public"] intersection !jmeno.tridy
   ```

4. Výsledkem je množina obsahující třídy, které mají název `jmeno.tridy`, anotační typy s názvem `jmeno.anotacniho.typu` a metody se jménem `jmeno.metody`. Zde je tedy demonstrováno, že ve výsledné množině můžou být vrcholy různých typů.

   ```
   jmeno.tridy union annotation type[@name="jmeno.anotacniho.typu"] union
   method[@name="jmeno.metody"]
   ```

5. Vypíše všechny metody, které vrací pole libovolného typu a všechny atributy, které jsou pole různého typu. Zde je předvedeno, že lze s uzly rozdílného typu pracovat stejně (pokud oba typy disponují stejným typem vztahu).

   ```
   (method union member)[is_type/field]
   ```

6. Vrátí všechny metody, v jejichž těle je volaná jiná metoda anebo je volaná metoda pomocí klíčového slova `throws`.

   ```
   method[call or call_throws]
   ```

7. Výstupem jsou všechny metody tříd, které se starají o výjimky zachycované ve všech metodách.

   ```
   method/((throws/have_method) union (catch/have_method))
   ```

8. Tento dotaz vrazí všechny metody třídy `jmeno.tridy` a `jmeno.tridy2`. Zde je předvedeno použití pravdivostní funkce `or`.

   ```
   class[@name="jmeno.tridy" or @name="jmeno.tridy2"]/have_method
   ```

9. Vypíše všechny anotace rozhraní se jménem `jmeno.rozhrani`. Demonstruje používání anotací.

   ```
   interface[@name="jmeno.rozhrani"]/have_annotated
   ```

10. Informuje o všech metodách a konstantách (atributech) všech anotačních typů.

    ```
    annotation type/(have_method union have_member)
    ```

11. Vrátí všechny hodnoty všech anotací metody s názvem `jmeno.metody`.

    ```
    method[@name="jmeno.metody"]/have_annotated/have_element
    ```

12. Výstupem je množina tříd, které mají anotaci v níž je nějaká hodnota nastavena na řetězec `have_method`.

    ```
    method[have_annotated/have_element[@value="have_method"]]
    ```

13. Nalezne všechny elementy, které jsou anotovány anotací s názvem `jmeno.anotace`.

    ```
    annotation type[@name="jmeno.anotace"]/<main_type/is_annotated_by
    ```

14. Dotaz vypíše všechny metody, které mají právě 3 parametry.

    ```
    method[have_parameter[@position="2"] and not have_parameter
    [@position="3"]]
    ```

15. Tento dotaz může vypadat jako ekvivalent k předchozímu, avšak není tomu tak. Výsledkem by byly všechny metody, které mají alespoň 3 parametry. Zanoření filtrů tak vlastně říká, že požadujeme takové metody, které disponují alespoň jedním parametrem, který je má pozici 2 (čísluje se od 0) a zároveň není na pozici 3.

    ```
    method[have_parameter[@position="2" and not @position="3"]]
    ```

16. Výsledkem jsou všechny metody s názvem `jmeno.metody`, které mají první parametr typu `String` a druhý parametr typu `int`. Žádný další parametr tato metody mít nesmí. Ve zkratce by se dalo říci, že dotaz hledá metodu `jmeno.metody(String,int)`.

    ```
    method[@name="jmeno.metody" and have_parameter[@position="0"]/is_type
    [@name="java.lang.String"] and have_parameter[@position="1"]/is_type
    [@name="int"] and not have_parameter[@position="2"]]
    ```

17. Vrátí všechny metody, jehož alespoň jeden parametr je typu pole celých čísel.

    ```
    method[have_parameter/is_type/field/main_type[@name="int"]]
    ```

18. Tento dotaz je ekvivalentní s předchozím. Rozdíl je v tom, že filtrace není strukturou dotazu, ale dalším zanořením filtru.

    ```
    method[have_parameter[is_type[field[main_type[@name="int"]]]]]
    ```

19. Opět ekvivalentní dotaz s předchozím. Tentokrát je dotaz psán v opačném pořadí od toho, co je známo.

    ```
    int/main_typed_by/field/typed_by/is_parameter/method
    ```

20. Výsledkem dotazu budou třídy, které dědí od třídy `jmeno.tridy` nebo `jmeno.tridy2`.

    ```
    class[extends/class[@name="jmeno.tridy"] or extends/class
    [@name="jmeno.tridy2"]]
    ```

21. Ekvivalentní dotaz předchozímu. Zde je zjednodušený zápis filtru.

    ```
    class[extends[@name="jmeno.tridy" or @name="jmeno.tridy2"]]
    ```

22. Opět ekvivalentní dotaz předchozím dvěma dotazům. Nyní je znovu zkoušen postup: od toho, co je známo.

    ```
    (class[@name="jmeno.tridy"] union class[@name="jmeno.tridy2"])/
    extended_by
    ```

23. Ještě jeden ekvivalentní dotaz, který využívá zkrácení zápisu pro vyhledávání konkrétní třídy.

    ```
    (jmeno.tridy union jmeno.tridy2)/extended_by
    ```

24. Vrací třídy, jejichž metoda volá jinou metodu s anotací `@jmeno.anotace` , která obsahuje element `jmeno.elementu="cokoli"`. Tato volaná metody musí být obsažena ve třídě, která implementuje rozhraní `jmeno.neceho`.

    ```
    data type[@name="jmeno.neceho"]/implemented_by/have_method[>annotated
    [@name="jmeno.anotace"]/have_element/is_value[@name="jmeno.elementu"]]/
    called_by/is_method
    ```

25. Ekvivalentní dotaz, jako předchozí avšak je zde snaha o rozdílný přístup. Zde jsou známy 2 informace (jméno rozhraní a anotace), proto je přístup „od toho, co je známo“ možný provést těmito dvěmi způsoby.

    ```
    annotation[@name="jmeno.anotace"][have_element/is_value
    [@name="jmeno.elementu"]]/<annotated[is_method/implements
    [@name="jmeno.neceho"]]/called_by/is_method
    ```

26. Vrátí všechny třídy, jejíchž metody volají metodu se jménem `jmeno.metody` s alespoň jedním parametrem typu `String` deklarovanou ve třídě, od které dědí třída jménem `jmeno.tridy`.

    ```
    class[extended_by[@name="jmeno.tridy"]]/have_method
    [@name="jmeno.metody" AND have_parameter/is_type
    [@name="java.lang.String"]]/called_by/is_method
    ```

27. Opět ekvivalentní dotaz, jako předchozí, tentokrát z jiného pohledu.

```
(method union constructor)[@name="jmeno.metody" AND have_parameter/
is_type[@name="java.lang.String"] and is_method/extended_by
[@name="jmeno.tridy"]]/called_by/is_method
```

28. Tento dotaz vrátí takové třídy, které obsahují pouze statické metody a atributy.

    ```
    class[have_method and not have_method[@static="false"] and have_member
    and not have_member[@static="false"]]
    ```

29. Vypíše všechny třídy, které používají jako datový typ některého ze svých atributů vnitřní nebo anonymní třídu.

    ```
    class[have_member/is_type[@anonym="true" or @inner="true"]]
    ```

30. Ekvivalentní dotaz předchozímu. Nyní s opačným přístupem.

    ```
    (inner class union anonym class)/typed_by/<member/class
    ```