Univerzita Komenského v Bratislave

Fakulta matematiky, fyziky a informatiky

Statická analýza Java kódu

Bakalárska práca

2012

Peter Laca

Univerzita Komenského v Bratislave

Fakulta matematiky, fyziky a informatiky

Statická analýza Java kódu

Bakalárska práca

Študijný program: Informatika
Študijný odbor: 2508 Informatika
Školiace pracovisko: Katedra Informatiky
Vedúci: RNDr. Peter Borovanský, PhD.

Bratislava, 2012

Peter Laca

Univerzita Komenského v Bratislave
Fakulta matematiky, fyziky a informatiky

ZADANIE ZÁVEREČNEJ PRÁCE

Meno a priezvisko študenta: Peter Laca
Študijný program: informatika (Jednoodborové štúdium, bakalársky I. st., denná forma)
Študijný odbor: 9.2.1. informatika
Typ záverečnej práce: bakalárska
Jazyk záverečnej práce: slovenský

Názov: Statická analýza Java kódu

Cieľ: Cieľom práce je vytvoriť nástroj na statickú analýzu JAVA kódu za učelom detekcie závislostí medzi triedami projektu. Výsledný analyzátor bude implementovaný ako plugin do existujúceho vývojového prostredia.

Vedúci: RNDr. Peter Borovanský, PhD.
Katedra: FMFI.KAI - Katedra aplikovanej informatiky

Dátum zadania: 13.10.2011

Dátum schválenia: 24.10.2011

doc. RNDr. Daniel Olejár, PhD.
garant študijného programu

.....
študent

.....
vedúci práce

Ďakujem vedúcemu RNDr. Petrovi Borovanskému, PhD. za jeho odbornú pomoc, ako aj svojej rodine a priateľke za podporu.

Abstrakt

V súčasnej dobe vzniká stále viac a viac rámcov (frameworkov) a knižníc pre zjednodušenie práce. Tieto komponenty sú často veľmi rozsiahle. Programy, ktoré ich používajú, nepotrebujú väčšinou všetky jeho časti. To spôsobuje, že výsledný program je príliš veľký, alebo obsahuje veľa zbytočných súborov.

Cieľom tejto práce je vytvorenie zásuvného modulu (plug-inu) do vývojového prostredia Eclipse, ktorý pre Java projekt nájde všetky nepotrebné súbory z rámcov a knižníc, ako aj zo samotného projektu. Vzhľadom k tomu, že počas vývoja softvéru je ťažké predpovedať, či sa bude daný súbor používať, alebo nie, je modul použiteľný ako posledný krok pred nasadením softvéru do produkčného prostredia.

Kľúčové slová: statická analýza, závislosti v java projekte, eclipse zásuvný modul

Abstract

Nowadays, there are a lot of frameworks for simplifying our work. These frameworks mostly consist of large set of files and software using these frameworks usually doesn't need all the parts, which leads to large volume of the result software, or it contains a lot of unnecessary files.

The aim of this work is creating of plug-in for platform Eclipse, that finds all unnecessary files from included frameworks and libraries, or project itself for given Java project. Since it is hard to predict whether file will be used or not, the plug-in is recommended to use as the last step before installing the software in production environment.

Keywords: static analysis, dependencies in java code, eclipse plug-in

Zoznam použitých skratiek

API Application programming interface

AST Abstract syntax tree

IDE Integrated development environment

JADEAN Java Dependency Analyzer

JAR Java Archive

JVM Java Virtual Machine

RCP Rich Client Platform

SWT Standard widget toolkit

Zoznam obrázkov

Obsah

Úvod

Statická analýza kódu je analýza kódu bez nutnosti jeho spustenia. Nástroje statickej analýzy sa používajú pre optimalizáciu kódu, zlepšenie čitateľnosti, štandardizáciu, atď. Jednou z oblastí statickej analýzy je aj zisťovaníe závislostí v projekte, čím sa zaoberá táto práca.

V prvej kapitole je zhrnutý prehľad danej problematiky. Kapitola začína motiváciou a praktickým aplikovaním výsledkov práce. Nasleduje stručný popis platformy Eclipse a základné informácie o statickej analýze kódu. Kapitola je zakončená prehľadom existujúcich riešení.

Druhá kapitola sa zaoberá návrhom výsledného riešenia, ktoré je implementované ako zásuvný modul do vývojového prostredia Eclipse. Na začiatku je zadefinovaná problematika, ktorej sa práca venuje, po čom nasleduje objektový návrh, v ktorom sa teoretické definície transformujú do objektového návrhu a abstraktných štruktúr. Tieto abstraktné štruktúry sú navrhnuté pre akýkoľvek typ projektu. Ďalej sú v kapitole popísané spôsoby hľadania závislostí v Java projektoch. Pri hľadaní týchto spôsobov vzniká problém s reflexívnym modelom, na ktorý je zameraná ďalšia časť. Koniec kapitoly je venovaný návrhu integrácie do vývojového prostredia Eclipse a testovacím scenárom.

Tretia kapitola využíva poznatky z predošlej kapitoly na implementáciu návrhu v jazyku Java. Prvá časť sa zaoberá implementáciou objektového návrhu a odvodením abstraktných tried do konkrétnych tried pracujúcich s projektami typu Java. Neoddeliteľnou súčasťou zisťovania závislostí v projekte je aj spôsob ich hľadania, čím sa zaoberá ďalšia časť, ktorá popisuje implementačné detaily pre hľadanie závislostí v rôznych typoch zdrojov. Po nájdení nezávislých zdrojov projektu je možné tieto

zdroje vymazať, čím sa zaboerá nasledujúca časť. Záverečné časti zobrazujú integráciu do vývojového prostredia Eclipse a popis užívateľského prostredia a funkcionality s ukážkami výslednej aplikácie.

Kapitola 1

Prehľad

1.1 Motivácia

Rozvoj informačných technológií spôsobuje stále väčší dopyt po informačných systémoch a softvéri. Vývoj softvéru je podporovaný rôznymi programovacími jazykmi, ktoré majú rôzne koncepty. Tiež aj rámcami, ktoré majú uľahčiť programátorom prácu poskytnutím istej funkcionality, ktorú implementovala a otestovala tretia strana - vývojár rámca (framework developer). Rámce poskytujú široké spektrum funkcionality, od jednoduchých podporných knižníc, po robustné skupiny knižníc. Použitím robustného rámca sa často stáva, že softvér vyžaduje len istú časť jeho funkcionality. Ten však musí byť celý zahrnutý do finálneho softvéru, čo sa prejavuje aj na výslednej veľkosti.

Pre ilustráciu poslúži nasledovný program (praktická situácia):

Zadanie: Na vstupe je tar archív, ktorý obsahuje textové súbory. Textové súbory obsahujú zoznam geografických súradníc (zemepisná dĺžka a zemepisná šírka) oddelených bodkočiarkou. Na adrese localhost:5554 počúva telnet server, ktorému treba poslať všetky súradnice pomocou príkazu geo fix zem_sirka zem_vyska.

Analýza: Pre čítanie tar súboru je vhodný napr. balík `org.apache.commons.compress`. Pre simuláciu telnet klienta je vhodný napr. balík `org.apache.commons.net`.

Pozorovanie: Obidva balíky obsahujú viacero nezávislých komponentov (`org.apache-`

`.commons.compress` obsahuje aj podporu pre iné typy archívov, `org.apache.commons-.net` obsahuje aj podporu iných klientov). Vo výslednej aplikácii sú tieto komponenty nepotrebné, a teda zbytočne zaberajú miesto vo výslednej aplikácii. Výsledná veľkosť oboch knižníc je 501 kB, no potrebných z nich je 74 kB, čo je zhruba **15%** pôvodnej veľkosti.

Ako ukazuje pozorovanie, pribalené balíky obsahujú mnohokrát ďalšie nepotrebné komponenty, ktoré vo výslednom programe zaberajú zbytočne miesto. Cieľom tejto práce je nájsť nepotrebné komponenty na úrovni tried, identifikovať ich a odstrániť.

1.2 Eclipse

Informácie v tejto časti pochádzajú z oficiálneho zdroja [IBM05].

Eclipse je platforma bežiaca na viacerých operačných systémoch ponúkajúca prostriedky pre vývoj aplikácií, zásuvných modulov do už existujúcich aplikácií, či integrované grafické prostredie (IDE - Integrated Development Environment) pre vývoj aplikácií v rôznych jazykoch (Java, C, C++, PHP, ...).

Samotný Eclipse neponúka rozsiahlu funkcionalitu, jeho široké spektrum funkcionality je zabezpečené modelom rozšírovania funkčnosti aplikácie prostredníctvom zásuvných modulov. Zásuvný modul (plug-in) je súbor funkcií a dát, ktoré rozširujú funkcionalitu pôvodného systému.

Jadro Eclipse tvorí systém, tzv. Platform Runtime, ktorý sa stará o dynamické zisťovanie, zavádzanie a spúšťanie zásuvných modulov. Tento systém je založený na OSGi rámci (OSGi framework), ktorý ponúka funkcionalitu pre tvorbu aplikácií s dynamickými modulmi, alebo komponentami. Ďalší dôležitý pojem, týkajúci sa jadra Eclipse platformy, je Equinox, čo je implementácia OSGi rámca rozšírená o ďalšiu funkcionalitu.

Do jadra sa zásuvné moduly pridávajú prostredníctvom tzv. rozšírujúcich bodov (extension points), čo sú miesta v systéme, kam sa pridáva nová funkcionalita zásuvných modulov. Túto funkcionalitu zabezpečuje práve Equinox, ktorý poskytuje register

rozšírení (extension registry), kde sa určujú vzťahy medzi jednotlivými zásuvnými modulmi.

Platforma Eclipse prichádza aj s návrhom užívateľského rozhrania, ktorého základnými stavebnými kameňmi sú perspektívy (perspectives), zobrazenia (views), editory (editors) a akcie a príkazy (actions and commands). Perspektíva je zoskupenie a umiestnenie zobrazení a editorov, zobrazenie je grafická komponenta zobrazujúca istý druh údajov (stromovú štruktúru, tabuľku, text, ...), editor je rovnako grafická komponenta poskytujúca funkcionalitu pre prehliadanie a editovanie zdrojov, akcie a príkazy vyjadrujú činnosti, ktoré sa vykonajú po vyvolaní nejakej udalosti (napr. kliknutie na položku v menu). Grafické komponenty sú založené na technológiach SWT, alebo JFace.

Pre prácu s abstraktnými zdrojmi poskytuje Eclipse tzv. manažment zdrojov (resource management), čo je API pre prácu s projektami, adresármi a súbormi vytvorenými aplikáciou uloženými v súborovom systéme.

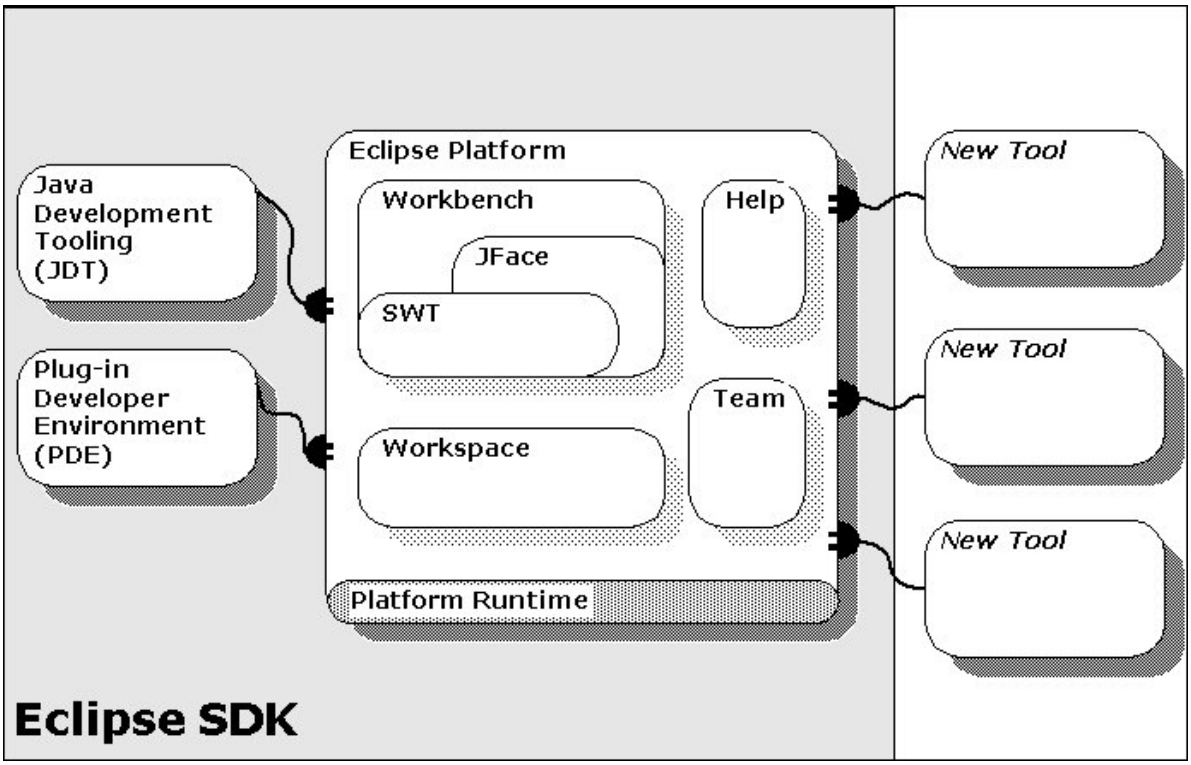
Obrázok 1.1 ukazuje architektúru Eclipse platformy.

1.3 Statická analýza

Statická analýza kódu je analýza kódu bez nutnosti jeho spustenia. Teda analýza nad zdrojovými súbormi, alebo skompilovaným programom. Cieľom statickej analýzy je zisťovať dodržiavanie pravidiel, udržovanie čistoty kódu, pomoc pri odhaľovaní možných chýb programu (výnimky bez ošetrenia), optimalizácia (premenná, ktorá nie je nikde použitá), ale aj zisťovanie rôznych metrík, týkajúcich sa rozšíriteľnosti, udržiavateľnosti, či znovupoužiteľnosti kódu.

1.4 Existujúce riešenia

V súčasnej dobe existuje viacero nástrojov statickej analýzy so zámerom zlepšiť kvalitu kódu, optimalizovať kód a pod.



Obr. 1.1: Eclipse platforma

1.4.1 JDepend

JDepend ([jde12]) prechádza class súbory projektov a na základe nich generuje rôzne metriky pre každý balík (kvality kódu, znovupoužiteľnosti, rozšíriteľnosti), no odhaľuje aj cykly v závislostiach. Neposkytuje však možnosť získania nezávislých tried s voľbou ich zmazania. JDepend je implementovaný aj ako zásuvný modul do vývojového prostredia Eclipse.

1.4.2 UCDetector

UCDetector ([ucd12]) je taktiež zásuvný modul do prostredia Eclipse, ktorý umožňuje hľadanie nepoužitých častí kódu na úrovni tried, metód alebo členských premenných tried. Nepoužitý kód však nehľadá v referencovaných knižniciach.

1.4.3 Sonar

Sonar ([son12]) je pomerne robustný systém pre kontrolu kvality kódu. Pre používanie ho však treba inštalovať osobitne a integrovať do prostredia Eclipse prostredníctvom zásuvného modulu. Podporuje aj analýzu závislostí, no vyžaduje sa pomerne zložitá konfigurácia.

Kapitola 2

Návrh

Cieľom návrhu je umožniť rozšíriteľnosť riešenia pre rôzne typy projektov (PHP, C++, ...). Návrh teda musí počítať s abstraktnými štruktúrami, ktoré by sa mali ľahko implementovať pre konkrétne typy projektov. Rovnako by malo byť možné ľahko zmeniť stratégiu získavania závislých tried v projekte. V nasledujúcich kapitolách je popísaný všeobecný objektový návrh a analýza hľadania závislostí v Java projekte, ktorá je iná pre triedy projektu a pre referencované knižnice. Odhaľovaním závislostí vzniká problém s reflexívnym modelom jazyka Java. Tejto problematike sa venuje kapitola 2.4. Po vybudovaní modelu pre odhaľovanie závislostí nasleduje integrácia do vývojového prostredia Eclipse a posledná kapitola sa zaoberá návrhom testovacích scenárov.

2.1 Definície pojmov

Táto kapitola sa zaoberá problematikou hľadania závislostí v projekte. Na začiatok je však potrebné zadefinovať, čo je to projekt, zdroj a vstupný bod aplikácie. Ďalej sú definované pojmy ako závislosť medzi zdrojmi, závislý a nezávislý zdroj.

Definícia 1 (Zdroj)**.** Zdroj (Resource) je v kontexte tejto práce základná jednotka, ktorú využíva aplikácia. V prípade jazyka Java sú to triedy, no vo všeobecnosti sa môže jednať aj o obrázky, textové alebo binárne súbory alebo súbory, ktoré obsahujú zdrojový kód v niektorom jazyku (Java, PHP, XML, HTML, ...).

Definícia 2 (Projekt)**.** Projekt (Project) je množina zdrojov, ktorá vytvára aplikáciu.

V závislosti od programovacieho jazyka a použitých technológií môže mať projekt viacero typov (Java projekt, C++ projekt, PHP projekt, ale aj PHP Zend projekt).

Definícia 3 (Vstupný bod aplikácie)**.** Vstupný bod aplikácie (Entry Point) je zdroj, ktorý je zodpovedný za spustenie výslednej aplikácie (v Java napr. trieda s metódou `public static void main(String[] args)`, v PHP napr. súbor `index.php`).

Každý typ projektu obsahuje zdroje, ktoré sú jeho súčasťou - môžu byť na sebe závislé, alebo nezávislé. Ak je daná definícia závislosti zdrojov pre daný projekt a zoznam jeho zdrojov, dá sa zostrojiť graf závislosti zdrojov. Definícia závislosti je rôzna pre rôzne druhy projektov.

Definícia 4 (Závislosť zdrojov)**.** Nech Z je množina zdrojov projektu a $\circ$ je relácia medzi zdrojmi projektu, pre ktorú platí: $\forall x, y \in Z : x \circ y \iff x$ je závislý zdroj na y (inak povedané, zdroj x využíva zdroj y).

Poznámka 1. *Projekt sa teda dá charakterizovať usporiadanou dvojicou $P = (Z, \circ)$, kde Z je množina zdrojov projektu a $\circ$ je relácia popisujúca závislosť medzi zdrojmi.*

Definícia 5 (Graf závislostí zdrojov)**.** Nech $P = (Z, \circ)$ je projekt. Potom $G = (Z, E)$, kde $E \subseteq Z \times Z$, pričom $\forall x, y \in Z : (x, y) \in E \iff x \circ y$, je graf závislosti zdrojov projektu.

Prehľadávaním grafu závislostí zdrojov do šírky zo vstupného bodu aplikácie vznikne z navštívených vrcholov množina závislých zdrojov. Množina nezávislých zdrojov je potom množinový rozdiel množiny všetkých zdrojov a závislých zdrojov. Formálnejšie:

Definícia 6 (Závislé zdroje)**.** Nech $P = (Z, \circ)$ je projekt, $G = (Z, E)$ je graf závislostí daného projektu a $v \in Z$ je vstupný bod aplikácie. Nech $U \subseteq Z$ je množina zdrojov, ktorá vznikne prehľadaním grafu G do šírky zo zdroja v. Množina U sa potom nazýva **množina závislých zdrojov projektu**. Každý zdroj $x \in U$ sa nazýva **závislý zdroj**.

Definícia 7 (Nezávislé zdroje)**.** Nech $P = (Z, \circ)$ je projekt a U je množina závislých zdrojov projektu P. Potom množina $N = Z - U$ sa nazýva **množina nezávislých zdrojov projektu**. Každý zdroj $x \in N$ sa nazýva **nezávislý zdroj**.

2.2 Objektový návrh

Podľa definícií z predošlej časti vznikajú dva abstraktné prvky: **Projekt** a **Zdroj**.

Pre každý typ projektu platí:

- má svoje meno
- obsahuje zdroje
- vie získať množinu závislých a nezávislých zdrojov
- vie vymazať zadanú množinu zdrojov

Pre zdroj platí:

- má svoje meno
- vie získať množinu závislých zdrojov

Tieto dva druhy poslúžia ako základ pre vytvorenie dvoch abstraktných tried - `Project` (predstavuje projekt) a `Resource` (predstavuje zdroj).

Táto práca sa zaoberá projektom typu Java. Ako prvý krok je potrebné odvodiť triedy `Project` a `Resource` pre túto implementáciu. Následne sa definuje relácia $\circ$, z čoho vyplýva aj potreba nájdenia spôsobu pre hľadanie závislých zdrojov. Bežný Java projekt v prostredí Eclipse je charakterizovaný nasledujúcimi vlastnosťami:

- má svoj názov
- má zadefinované adresáre, kde sú uložené zdrojové kódy
- má zadefinovaný adresár, kam sa zdrojové kódy kompilujú
- má určený zoznam referencovaných knižníc

Tieto vlastnosti vytvárajú atribúty a operácie triedy `JavaProject`, odvodenej od abstraktnej triedy `Project`.

Rovnako sa pre bežný Java projekt rozlišujú tieto druhy zdrojov:

- trieda projektu
- balík projektu
- referencovaná trieda
- referencovaný balík
- systémová trieda
- systémový balík

Trieda projektu je trieda, ktorá je súčasťou zdrojového kódu projektu. Balík projektu je balík tried projektu. Referencovaná trieda je trieda obsiahnutá v nejakej referencovanej knižnici. Referencovaný balík je balík referencovaných tried. Systémová trieda je trieda JVM (Java virtual machine) - napr. `java.lang.String`, `java.io.IOException`, resp. trieda, ktorá sa nachádza v niektorom classpath zázname. Systémový balík je balík systémových tried - napr. `java.net.*`. Posledné dva typy zdrojov však nevyžadujú hľadanie závislých zdrojov z dôvodu, že sú to systémové triedy, ktoré sú zapúzdrené vnútri JVM.

Všetký zdroje sú odvodené od abstraktnej triedy `Resource`, keďže sú špecifické pre Java projekt. Pre každý typ zdroja je rôzny spôsob získania množiny závislých zdrojov, ktorý je popísaný v nasledujúcej kapitole.

2.3 Hľadanie závislostí v Java projekte

Hľadanie závislostí v Java projekte vyžaduje identifikáciu troch kľúčových prvkov: charakteristiky projektu, zdrojov a relácie opisujúcej závislosť zdrojov. Charakteristika projektu a identifikácia zdrojov je popísaná v predchádzajúcej kapitole. Táto kapitola sa zaoberá hľadaním relácie opisujúcej závislosť zdrojov. Jednotlivé podkapitoly sa venujú spôsobom hľadania závislostí pre jednotlivé druhy zdrojov, nakoľko sú tieto zdroje odlišné, a teda odlišné sú aj spôsoby.

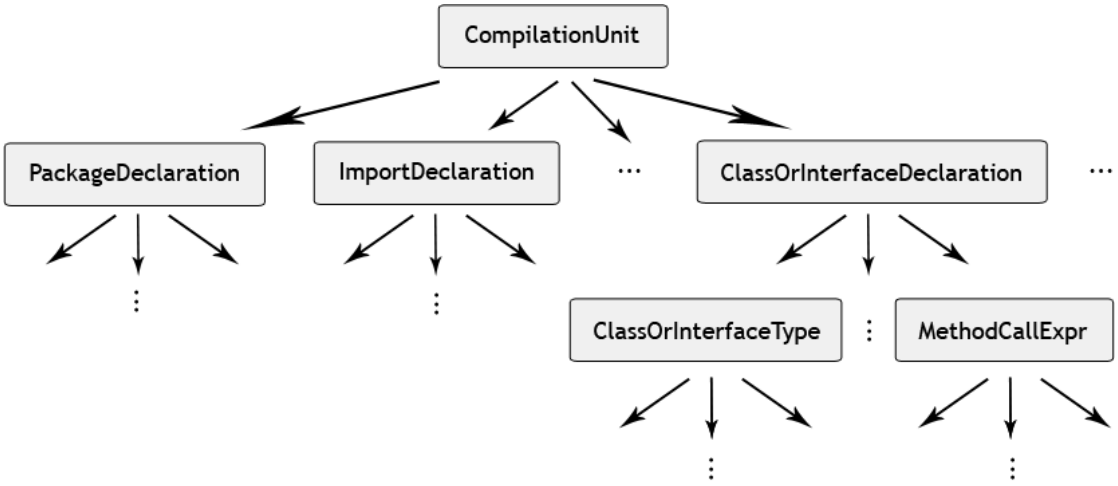
2.3.1 Triedy a balíky projektu

Prvý typ zdroja je trieda projektu, teda trieda, ktorá je súčasťou zdrojových kódov projektu. Na úvod sú vysvetlené niektoré pojmy z oblasti statickej analýzy potrebné pre pochopenie problematiky. Pre hľadanie závislých zdrojov projektu poslúži syntaktická analýza, ktorú stručne opisuje časť Syntaktická analýza. Neoddeliteľnou súčasťou syntaktickej analýzy je popísanie syntaxe zdrojového kódu v jazyku Java, ktorý popisuje časť Syntax Java súboru. Po pochopení syntaxe Java súboru je potrebné nájsť závislé zdroje, teda triedy a balíky, ktoré sú potrebné v danom java súbore. Tento proces je založený na prehľadávaní syntaktického stromu a ten objasňuje časť Prehľadávanie syntaktického stromu.

Syntaktická analýza

Syntaktická analýza poslúži pri hľadaní závislých zdrojov triedy, ku ktorej je k dispozícii zdrojový kód. Zdrojový kód treba prečítať a hľadať v ňom výskyt názvov iných tried. Triviálne riešenie čítania súboru ako textu sa ukazuje ako nesprávne, pretože názov triedy môže byť ukrytý napr. v komentári, čo ale nevyjadruje závislosť. Syntaktická analýza na základe definovanej gramatiky skonštruuje syntaktický strom a v prípade nesprávnej syntaxe vyhlási chybu. Nekontroluje sa však sémantika zdrojového kódu, teda je potrebný predpoklad, že zdrojový kód je skompilovateľný.

V súčasnosti existuje viacero parserov Java súborov, pre túto prácu poslúži projekt `javaparser` ([jav12]), ktorý poskytuje abstraktný syntaktický strom (AST) a podporuje tzv. visitor pattern nad týmto stromom. Tento parser však zatiaľ pracuje nad



Obr. 2.1: Ukážka abstraktného syntaktického stromu jazyka Java

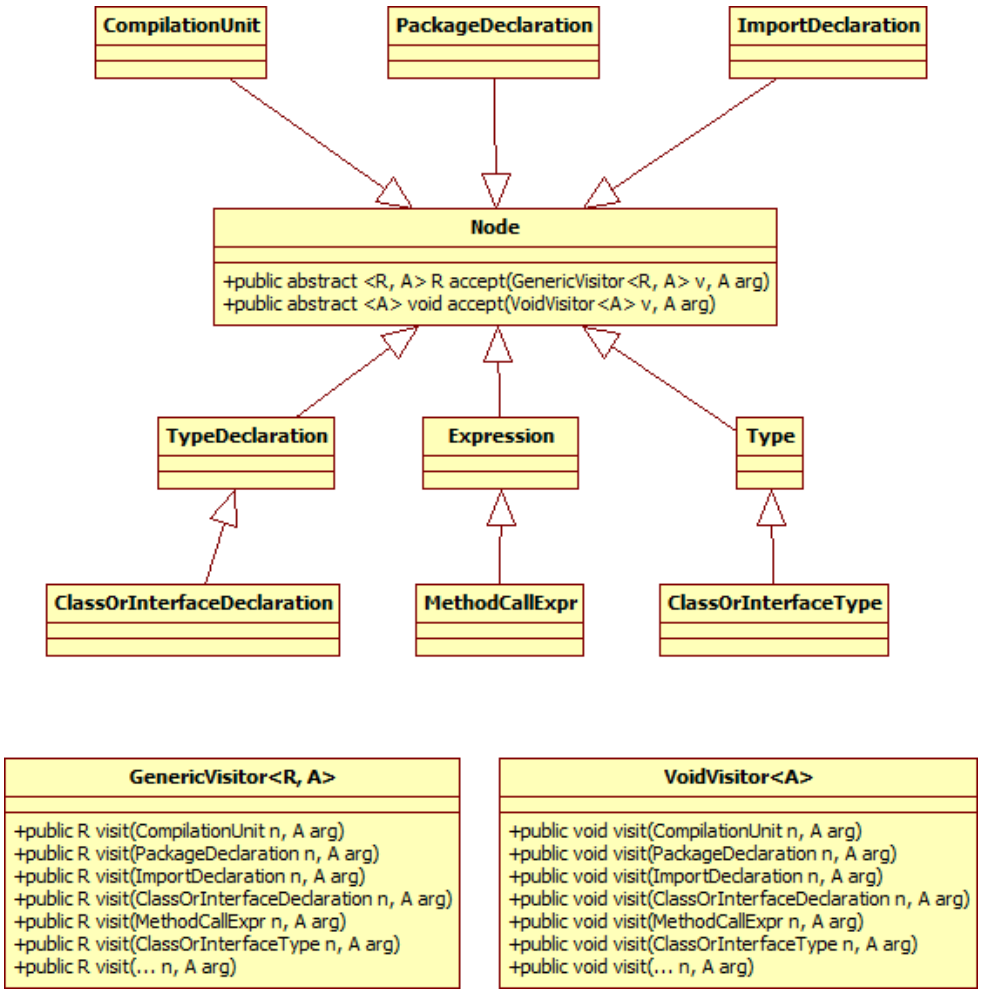
verziou Java 1.5, teda nepodporuje vyššie verzie a ich nové vlastnosti a možnosti. Pre väčšinu Java projektov je ale parser verzie 1.5 postačujúci.

Syntaktický strom

Štruktúra syntaktického stromu úzko súvisí s gramatikou jazyka, ktorý ho generuje. Pre jazyk Java 1.5 stručne ukazuje štruktúru takéhoto stromu obrázok **2.1**. Projekt javaparser podporuje visitor pattern nad týmto stromom, čo je návrhový vzor, ktorý umožňuje pridávať novú funkčnosť bez nutnosti zmeny existujúcich dátových štruktúr a takisto aj prechádzať dátovú štruktúru. Obrázok **2.2** ukazuje vzťah triedy Visitor a syntaktického stromu.
Pre potreby tejto práce sú dôležité 3 prvky tohoto stromu: `PackageDeclaration`, `ImportDeclaration`, `ClassOrInterfaceType`, ktoré sa dajú získať pomocou špeciálneho visitoru naviazaného na syntaktický strom.

2.3.2 Referencované triedy a balíky

Referencované triedy a balíky sú obsiahnuté v niektorej z referencovaných knižníc projektu. Pri tomto type zdrojov však už syntaktická analýza nestačí z dôvodu, že zdroje sú už skompilované class súbory, resp. sú zbalené do jar súboru. Z tohto dôvodu je potrebné vedieť prečítať jar súbor a nájsť, podobne ako u triedy projektu, závislé



Obr. 2.2: Vzťah visitora a syntaktického stromu

triedy v class súbore, nie však pomocou syntaktickej analýzy. Ako správny spôsob sa ukazuje čítanie class súboru ako binárneho súboru, nakoľko z oficiálneho popisu class súboru ([Sun12]) sú všetký názvy tried v triede uložené na jednom pomerne ľahko čitateľnom mieste.

Štruktúra JAR súboru

JAR súbor ako knižnica obsahuje skompilované class súbory usporiadané do adresárov, ktorých umiestnenie definuje balík, ktorého sú súčasťou. Napríklad súbor `Trieda.class` umiestnený v `a/b/c` je súčasťou balíka `a.b.c` a jeho plne kvalifikované

meno (fully qualified name - FQN) je `a.b.c.Trieda`.

JAR súbor môže obsahovať navyše aj popisný súbor `MANIFEST.MF`, ktorý obsahuje dvojice atribút a hodnota. Ak je jeden z atribútov `Class-Path`, jeho hodnota je zoznam ďalších knižníc potrebných pre daný JAR súbor.

V programovacom jazyku Java existuje štandardná podpora čítania JAR súboru, balík `java.util.jar.*`.

Popis CLASS súboru

Class súbor obsahuje Java bytecode pre Java virtual machine (JVM). Tento súbor má formát binárneho súboru a má zadefinovanú svoju štruktúru. Existuje oficiálna dokumentácia k formátu class súboru ([Sun12]), ktorá je však pre účely tejto práce príliš rozsiahla. Pre potreby tejto práce stačí začiatok class súboru.

Celý class súbor sa dá popísať nasledovnou štruktúrou, ktorá zobrazuje len podstatný začiatok:

Listing 2.1: Štruktúra class súboru

```
ClassFile {
    u4              magic;
    u2              minor_version;
    u2              major_version;
    u2              constant_pool_count;
    cp_info         constant_pool[constant_pool_count-1];
    ...
}
```

`u4`, `u2` predstavujú 4, 2 neznamienkové bajty, `cp_info`, `field_info`, `method_info`, `attribute_info` sú ďalšie štruktúry, z ktorých je podstatná len `cp_info`.

Magická konštanta `magic` špecifikuje class súbor, jej hodnota je vždy `0xCAFEBABE`. Identifikátory `minor_version` a `major_version` predstavujú verziu class súboru.

Položka `constant_pool_count` obsahuje veľkosť tabuľky `constant_pool`, ktorá je tvorená položkami typu `cp_info`. Pre získanie zoznamu závislých zdrojov je potrebné získať názvy všetkých závislých tried, ktoré sú uložené v tejto tabuľke.

Tabuľka 2.1: Typy položiek cp_info

tag	typ cp_info
7	CONSTANT_Class
1	CONSTANT_Utf8

`cp_info` je štruktúra, ktorá je popísaná nasledovne:

Listing 2.2: Štruktúra CP_INFO

```
cp_info {
    u1 tag;
    u1 info[];
}
```

`cp_info` obsahuje jeden neznamienkový bajt určujúci typ položky a dodatočné informácie, ktoré sú iné pre každý typ. Týchto typov je celkom 14, nasledujúca tabuľka zobrazuje relevantné typy položiek `cp_info`.

Typ `CONSTANT_Class` reprezentuje triedu alebo rozhranie. Štruktúra je nasledovná:

Listing 2.3: Štruktúra CONSTANT_Class

```
CONSTANT_Class {
    u1 tag;
    u2 name_index;
}
```

kde tag obsahuje príslušný identifikátor typu, teda číslo 7 z predchádzajúcej tabuľky, a `name_index` je správny index do tabuľky `constant_pool`, kde sa nachádza položka typu `CONSTANT_Utf8`. Názov triedy alebo rozhrania je uložený vo forme plne kvalifikovaného mena (fully qualified name) spĺňajúceho nasledovné podmienky:

- ak sa jedná o primitívny dátový typ (t.j. `byte`, `char`, `double`, `float`, `int`, `long`, `short`, `boolean`), nahradí sa názov veľkým prvým písmenom názvu typu (napr. `int` - **I**, `boolean` - **B**)
- ak sa jedná o triedu s názvom `Trieda`, doplní sa pred názov znak **L** a na koniec znak **;** (napr. `Thread` - **LThread;**)

- ak sa jedná o pole s dimenziou n, doplní sa pred upravený názov prvým, alebo druhým krokom n-krát znak [(napr. `int[]` - [**I**, `Object[][][]` - [[[**LObject;**)

Typ `CONSTANT_Utf8` má nasledujúcu štruktúru:

Listing 2.4: Štruktúra CONSTANT_Utf8

```
CONSTANT_Utf8_info {
    u1 tag;
    u2 length;
    u1 bytes[length];
}
```

kde `tag` obsahuje príslušný identifikátor typu, teda číslo 1 z tabuľky typov, `length` je dĺžka reťazca a v poli `bytes` sú uložené znaky reťazca.
Postupným čítaním tabuľky `constant_pool` sa dá teda získať zoznam názvov všetkých použitých tried pre referencované balíky a triedy.

2.4 Reflexívny model

Reflexívny model jazyka Java poskytuje programátorom možnosť monitorovať, alebo meniť správanie aplikácií bežiacich na JVM za behu ([Ora12]). Funkcionalitu zabezpečuje **Reflection API**, ktorého základnou triedou je trieda `Class`. Trieda `Class` poskytuje metódy pre zisťovanie informácií o objekte za behu aplikácie. Vie získať zoznam metód, alebo volať metódy nad daným objektom, ako aj čítať a meniť jeho členské premenné, či vytvoriť jeho inštanciu.

Pri vytváraní inštancie nejakej triedy pomocou Reflection API je možné ju vytvoriť na základe zadaného reťazca, ktorý obsahuje názov triedy. Z tohoto dôvodu je ťažké predpovedať, aké triedy budú volané z triedy, ktorá používa funkcionalitu Reflection API. Preto sa nedá presne predpovedať, či v projekte existujú zdroje, ktoré sú nepotrebné.

Nájdenie použitia reflexívneho modelu je nájdenie volania metódy `Class.forName()`, alebo metód vracajúcich inštanciu triedy `Class`, teda `getSuperclass()`, `getClasses()`, `getDeclaredClass()`, `getDeclaringClass()` a `getEnclosingClass()` triedy `Class`. Vtedy je potrebné zaznamenať jeho používanie, a teda upozorniť na možné následky

pri odstránení zdrojov identifikovaných ako nezávislé. Jedným z možných spôsobov, ako zistiť názvy tried, ktoré sa používajú reflexívným modelom, je zistiť hodnotu parametra `Class.forName()`, alebo názov triedy vrátenej vyššie spomínanými metódami.

Ak je parameter tejto metódy reťazec, je riešenie triviálne - v reťazci je uložený názov závislej triedy. Problém nastáva, ak je parametrom tejto metódy premenná, alebo výsledok volania metódy. Vtedy je to nerozhodnuteľný problém a riešenia sú len aproximatívne.

2.5 Integrácia do Eclipse IDE

Ako vhodný komponent Eclipse platformy pre integráciu tohoto zásuvného modulu sa ukazuje **Wizard**, čo je sprievodca, ktorý obsahuje obrazovky, medzi ktorými sa dá prepínať prostredníctvom tlačidiel Next a Previous. Sprievodca sa zobrazí a inicializuje pre každý projekt zvlášť, teda sa musí spúšťať závisle od projektu. To sa dosiahne zaregistrovaním akcie do kontextového menu v zozname projektov. Po vyvolaní kontextového menu pre projekt sa zobrazí možnosť spustiť zásuvný modul. To vtedy, ak existuje podpora daného typu projektu. Následne, po vyvolaní akcie, sa zobrazí sprievodca s obsahom, ktorý je závislý od typu projektu.

Java projekt zobrazí v sprievodcovi nastavenia pre zdrojové adresáre, adresár pre skompilované súbory, referencované knižnice, výber vstupného bodu do aplikácie (napr. trieda, ktorá obsahuje metódu `main`), spustenie analýzy závislostí a na záver zoznam závislých a nezávislých zdrojov, s možnosťou zmazať nezávislé zdroje. Ak sa používa reflexívny model, zobrazí sa taktiež varovanie o možnom riziku.

2.6 Testovacie scenáre

Testovacie scenáre by mali preveriť funkčnosť hľadania závislých zdrojov pre jednotlivé zdroje, ako aj nájdenie výslednej množiny závislých a nezávislých zdrojov. Testy implementácie pre Java projekt musia zohľadniť aj projekty, ktoré využívajú reflexívny model, no aj tie, ktoré ho nepoužívajú.

Pre triedy a balíky projektu je potrebné otestovať prechádzanie syntaktického stromu a následne získanie zoznamu závislých zdrojov.

Pre referencované triedy a balíky je potrebné otestovať čítanie class súboru, získavanie názvu triedy zo špeciálneho formátu používaného v class súbore a rovnako aj získanie výsledného zoznamu závislých zdrojov.

Kapitola 3

Implementácia

V predošlej kapitole je popísaný návrh funkcionality zásuvného modulu, ako aj jeho integrácia do vývojového prostredia Eclipse. Táto kapitola sa zaoberá implementáciou tohoto návrhu, rovnako aj jeho implementačnými detailami a použitými technológiami. Prvá podkapitola opisuje implementáciu objektového návrhu v jazyku Java, druhá kapitola popisuje implementáciu algoritmov hľadania závislostí, tretia spôsoby vymazávania zdrojov, štvrtá kapitola sa venuje integrácii funkčnosti do vývojového prostredia Eclipse a posledná kapitola ukazuje výsledný produkt ako zásuvný modul do vývojového prostredia, jeho užívateľské rozhranie a funkcionalitu.

3.1 Implementácia objektového návrhu

Objektový návrh bol načrtnutý v **Kapitole 2: Návrh** a táto kapitola sa zaoberá jeho implementáciou. Podkapitola Trieda `Project` zobrazuje implementáciu konkrétnej triedy `JavaProject` odvodenej od abstraktnej triedy Project a podkapitola `Trieda Resource` implementáciu konkrétnych tried odvodených od abstraktnej triedy.

3.1.1 Štruktúra

Zásuvný modul sa skladá z dvoch hlavných balíkov: **dean** a **ui**. Balík dean (Dependency Analyzer) obsahuje logiku (algoritmy pre hľadanie závislostí a príslušné dátové štruktúry) a balík ui obsahuje zobrazovaciu časť (teda implementáciu do Eclipse prostredia). Kvôli prehľadnosti a testovaniu je balík dean úplne oddelený od akéhokoľvek

zobrazovania.

Abstraktné triedy a funkčnosť sú obsiahnuté v balíku dean. Ten ďalej obsahuje výnimky pre základné situácie, ktoré môžu nastať pri behu (`ResourceInvalidException` - nesprávny typ zdroja, `ResourceIOException` - chyba pri čítaní zdroja, `ResourceNotFoundException` - zdroj neexistuje, `ResourceUnsupportedException` - nepodporovaný typ zdroja).

Balík dean ďalej obsahuje 2 základné abstraktné triedy: `Project` a `Resource`.

3.1.2 Trieda Project

Trieda `Project` popisuje základné vlastnosti projektov nasledovnými abstraktnými metódami:

Listing 3.1: Trieda Project

```
public abstract class Project {
   private String _projectName;

   public Project(String projectName) { ... }
   public void setProjectName(String projectName) { ... }
   public String getProjectName() { ... }

   public abstract Graph<Resource, DefaultEdge> getDependencyGraph(
      String entryPoint);
   public abstract Collection<Resource> getUsedResources(String
      entryPoint);
   public abstract Collection<Resource> getAllResources();
   public abstract Collection<Resource> getUnusedResources(String
      entryPoint);
   public abstract void removeResources(Collection<Resource>
      resources);
}
```

V zdrojovom kóde sa nachádzajú neznáme typy - `Graph` a `DefaultEdge`. Obidva typy sú súčasťou balíka `jgrapht`, trieda `Graph` predstavuje graf ako matematický objekt a trieda `DefaultEdge` predstavuje rovnako hranu grafu. Viac informácií o balíku `jgrapht` sa nachádza v [Bar12].

3.1.3 Trieda Resource

Trieda `Resource` predstavuje abstraktný návrh zdroja projektu:

Listing 3.2: Trieda Resource

```
public abstract class Resource {
   public abstract String getName();
   public abstract Collection<Resource> getReferrencedResources();
   public abstract void delete();
}
```

3.1.4 Balík `dean.java.*`

Balík `dean.java.*` obsahuje implementáciu tried popísaných vyššie pre projekt typu Java.

Trieda `Resource` je implementovaná ako abstraktná trieda `JavaResource`, ktorá k pôvodnej funkčnosti pridáva kontext zdroja (trieda `JavaResourceContext`). Kontext zdroja obsahuje názov balíka, v ktorom sa zdroj nachádza, a zoznam importovaných tried a balíkov v danom zdroji. O vytváranie kontextu zdroja sa stará trieda `JavaResourceContextFactory`, ktorá prijíma ako vstupný parameter objekt typu `JavaProject` a názov triedy, na základe vstupných parametrov spustí syntaktickú analýzu zdrojového kódu triedy. Následne prejde syntaktický strom, kde nájde balík, v ktorom sa trieda nachádza, a tiež importované triedy a balíky. `JavaResourceContextFactory` vracia objekt typu `JavaResourceContext`.

Od triedy `JavaResource` sú odvodené ďalšie triedy, pre každý typ zdroja zvlášť:

- `JavaResourceProjectResource` ako abstraktná trieda pre triedy a balíky projektu
- `JavaResourceProjectClass` pre triedy projektu
- `JavaResourceProjectPackage` pre balíky projektu
- `JavaResourceReferencedClass` pre referencované triedy
- `JavaResourceReferencedPackage` pre referencované balíky

- `JavaResourceClasspathClass` pre systémové triedy
- `JavaResourceClasspathPackage` pre systémové balíky

Implementácia týchto tried je podrobnejšie popísaná v ďalšej kapitole, keďže podstatnú časť ich implementácie tvorí hľadanie závislých zdrojov.

Trieda `Project` je implementovaná ako trieda `JavaProject` a rozširuje ju o ďalšiu funkčnosť a vlastnosti:

- zoznam zdrojových adresárov
- adresár pre skompilované zdrojové súbory
- zoznam referencovaných balíkov
- použitie reflexívneho modelu v projekte
- zistenie, akého typu je zadaný zdroj
- zistenie potenciálnych vstupných bodov do aplikácie

3.2 Hľadanie závislostí

Kapitola 2: Návrh popisuje hľadanie závislostí v projekte, tu sa venujeme implementovaniu predstaveného riešenia a jeho implementačným detailom v jazyku Java. Prvá časť sa zaoberá hľadaním závislostí pre triedy a balíky projektu, teda implementáciou syntaktickej analýzy a prehľadávania abstraktného syntaktického stromu, na čo bola použitá knižnica javaparser ([jav12]). Druhá časť sa zaoberá implementáciou hľadania závislostí v referencovaných triedach a balíkoch, teda spracovaním jar a class súborov.

3.2.1 Triedy a balíky projektu

Triedy a balíky projektu sú súčasťou projektu, a teda k nim existujú aj zdrojové kódy. Balík projektu je súbor tried projektu, a teda sa stačí venovať triedam projektu. Ako bolo načrtnuté v **kapitole 2.3.1: Triedy a balíky projektu**, čítanie zdrojového

kódu jazyku Java je vhodné realizovať prostredníctvom prechádzania syntaktického stromu.

Balík `japa.parser.*` poskytuje dátovú štruktúru abstraktný syntaktický strom (AST - koreň stromu je trieda `japa.parser.ast.CompilationUnit`) s podporou Visitor patternu ([Eri95]) a metódu `japa.parser.JavaParser.parse()`, ktorá načíta zdrojový kód v jazyku Java verzie najviac 1.5 a vráti objekt typu `CompilationUnit`.

Visitor syntaktického stromu implementuje rozhranie `japa.parser.ast.visitor.GenericVisitor`, alebo `japa.parser.ast.visitor.VoidVisitor`, ktoré sa líšia v návratovej hodnote metódy `visit()`.

Pre získanie závislých zdrojov triedy projektu je vytvorený visitor, ktorý prejde celý strom, a keď narazí na typ `ClassOrInterfaceType` alebo `ImportDeclaration`, zaznamená si názov. Počas prechádzania stromu si vytvára aj kontext daného zdroja (popísaný v **3.1.4 Balík** `dean.java.*`). Po prejdení celého stromu je k dispozícii zoznam závislých zdrojov, ktorý sa ešte prečistí od duplikátov a systémových tried.

Nasledujúci zdrojový kód zobrazuje ukážku zdrojového kódu visitora:

Listing 3.3: Trieda Resource

```
class GetReferencedClassesVisitor<A> implements VoidVisitor<A> {
   ...
   public void visit(ImportDeclaration n, A arg) {
      String add = "";
      if (n.toString().trim().endsWith(".*;")) {
         this._importedPackages.add(n.getName().toString());
         add = ".*";
      }
      else {
         this._importedClasses.add(n.getName().toString());
      }
      this._classes.add(n.getName().toString() + add);
      n.getName().accept(this, arg);
   }
   ...
}
```

3.2.2 Referencované triedy a balíky

Čítanie jar / class súborov je popísané v **2.3.2 Referencované triedy a balíky**. Referencované triedy sú súčasťou balíka nejakej referencovanej knižnice, preto sa táto kapitola bude venovať čítaniu class súboru z jar súboru.

Čítanie class súboru z jar súboru prebieha v dvoch krokoch:

1. prečítanie jar súboru a získanie vstupného prúdu pre čítanie class súboru
2. čítanie class súboru

Pre čítanie jar súboru existuje systémový balík `java.util.jar.*`, ktorý poskytuje triedu `JarFile`. Trieda `JarFile` vracia metódou `entries()`, ktorá vracia zoznam položiek jar súboru, ktoré sú typu `JarEntry`. Trieda `JarEntry` obsahuje metódu `getInputStream()`, ktorá vracia objekt typu `InputStream`, teda prúd pre čítanie danej položky jar súboru.

Na základe špecifikácie class súboru bol vytvorený objektový návrh štruktúrou aj hierarchiou zodpovedajúci class súboru pre prehľadnosť a jednoduchosť čítania. Podľa popisu štruktúry class súboru je metódou `parse()` triedy `ClassResourceParser` prečítaný class súbor a po narazení na názov triedy, alebo balíka je zaznamenaný do zoznamu závislých zdrojov. Spolu s objavovaním závislých zdrojov sa zisťuje aj použitie reflexívnych volaní. Zoznam je následne očistený od duplikátov a systémových zdrojov a názvy tried sú príslušne upravené do štandardného formátu.

Reflexívne volania sa v class súbore odhaľujú čítaním položiek tabuľky `constant_pool`, ktoré sú typu `CONSTANT_Methodref`. Táto štruktúra obsahuje názov volanej metódy.

3.3 Vymazanie zdrojov

Po nájdení nezávislých zdrojov projektu je potrebné zdroje vymazať. Pre vymazanie tried projektu a referencovaných tried sa používa odlišný prístup.

Vymazanie tried projektu je realizované zmazaním súboru obsahujúceho zdrojový kód triedy.

Vymazanie referencovaných tried je vymazanie položky jar súboru, čo je implementované nasledovným algoritmom:

1. vytvor dočasný jar súbor
2. prejdi všetky položky pôvodného jar súboru a tie, ktoré sa nemajú zmazať, pridaj do dočasného súboru
3. očisti dočasný súbor od prázdnych adresárov
4. premenuj dočasný súbor na pôvodný

3.4 Vytvorenie zásuvného modulu

Integrácia do vývojového prostredia je popísana v kapitole 2 Návrh. V tejto časti sa implementuje návrh a popisujú sa implementačné detaily Eclipse RCP rámca (Eclipse RCP framework).

Rozšírenia do Eclipse prostredia sú realizované prostredníctvom tzv. bodov rozšírenia (extension points), ktoré sa definuju v manifeste zásuvného modulu, `plugin.xml`. Obsahujú prvky ako perspektívy, zobrazenia, akcie a príkazy a pod. Podstatné rozšírenia pre tento zásuvný modul sú príkazy, ktoré rozšíria kontextové menu v zozname projektov a sprievodca, ktorý tvorí hlavnú časť zásuvného modulu.

Pre spustenie zásuvného modulu je vytvorený príkaz, ktorý sa pridáva do bodu rozšírenia `org.eclipse.ui.commands`. Príkaz je zobrazený ako položka v kontextovom menu zoznamu projektov s názvom Dependency Analyzer, teda rozšírenie `org.eclipse.ui.menus`.

Príkaz obsluhuje trieda `BaseDependencyAnalyzerHandler`, ktorá po vyvolaní príkazu zistí typ projektu, a na základe toho pomocou triedy `DependencyAnalyzerHandlers-Factory` vytvorí implementáciu rozhrania DependencyAnalyzerHandler, obsahujúceho jednu metódu `public Object handle(Object projectElement)`, ktorá obsluhuje príkaz. Vstupný parameter funkcie je element stromovej štruktúry projektu. Pri Java projektoch je to typ `IJavaProject`, alebo `IJavaElement`. Pre projekty typu Java sa

vytvorí inštancia triedy `JavaDependencyAnalyzerHandler`, ktorá inicializuje sprievodcu. Vstupné dáta pre sprievodcu sú získané v obsluhe príkazu a sú uložené v objekte typu `WizardInput`, ktorý je následne odovzdaný sprievodcovi.

`WizardInput` obsahuje inštanciu triedy `JavaProject`, cestu k projektu, vstupný bod a zoznam poslucháčov na zmenu dát. Ak dôjde k zmene niektorej premennej, dôjde k vyvolaniu metódy `public void inputChanged()` u každého poslucháča, ktorý implementuje rozhranie `WizardInputChangedListener`.

Sprievodca je rozšírením `org.eclipse.ui.newWizards` a je implementovaný ako odvodená trieda od triedy `Wizard`. Obsahuje tri stránky, ktoré sú realizované implementovaním rozhrania `IWizardPage`.

3.5 Popis užívateľského prostredia a funkcionality

Výsledný produkt tejto práce je zásuvný modul do užívateľského prostredia Eclipse IDE pre Java projekty nazvaný JADEAN (Java Dependency Analyzer). Spúšťa sa kliknutím na Dependency analyzer v kontextovom menu niektorého projektu zo zoznamu projektov. Po kliknutí sa zobrazí sprievodca, ktorý na dvoch obrazovkách prevedie užívateľa od získania, resp. potvrdenia vstupných údajov cez analýzu závislostí až k výslednému zoznamu závislých a nezávislých zdrojov.

Prvá stránka sprievodcu obsahuje

- editovateľné textové pole typu `org.eclipse.swt.widgets.Text` zobrazujúce názov projektu
- zoznam zdrojových adresárov implementovaný ako tabuľka typu `org.eclipse.swt.widgets.Table` zobrazovaná komponentou `org.eclipse.swt.widgets.TableViewer` spolu s tlačidlami pre pridanie a odstránenie niektorej položky zo zoznamu, ktoré sú typu `org.eclipse.swt.widgets.Button`
- editovateľné textové pole zobrazujúce adresár so skompilovanými zdrojovými súbormi projektu

- zoznam referencovaných knižníc implementovaný rovnako ako zoznam zdrojových adresárov s rovnakou funkcionalitou (pridať, odstrániť)

Stránka je hneď pripravená na pokračovanie ďalšou stránkou, po kliknutí na tlačidlo Next.

Druhá obrazovka prijíme ako vstup od používateľa vstupný bod do aplikácie, čo sa dá realizovať vybratím triedy zo zoznamu nájdených tried (obsahujúcich main metódu), alebo manuálnym vyhľadaním súboru obsahujúceho zdrojový kódu vstupného bodu. Ak užívateľ vyberie možnosť vybrať vstupný bod z nájdeného zoznamu, možnosť vybrať vstupný bod manuálne bude neaktívna a naopak. Prepínanie medzi jednotlivými možnosťami je realizované prvkom grafického rozhrania typu Button (`org.eclipse.swt.widgets.Button`), ktorý má správanie ako prepínacie tlačidlo (RadioButton), čo sa dosiahne volaním konštruktora s nasledovnými parametrami:

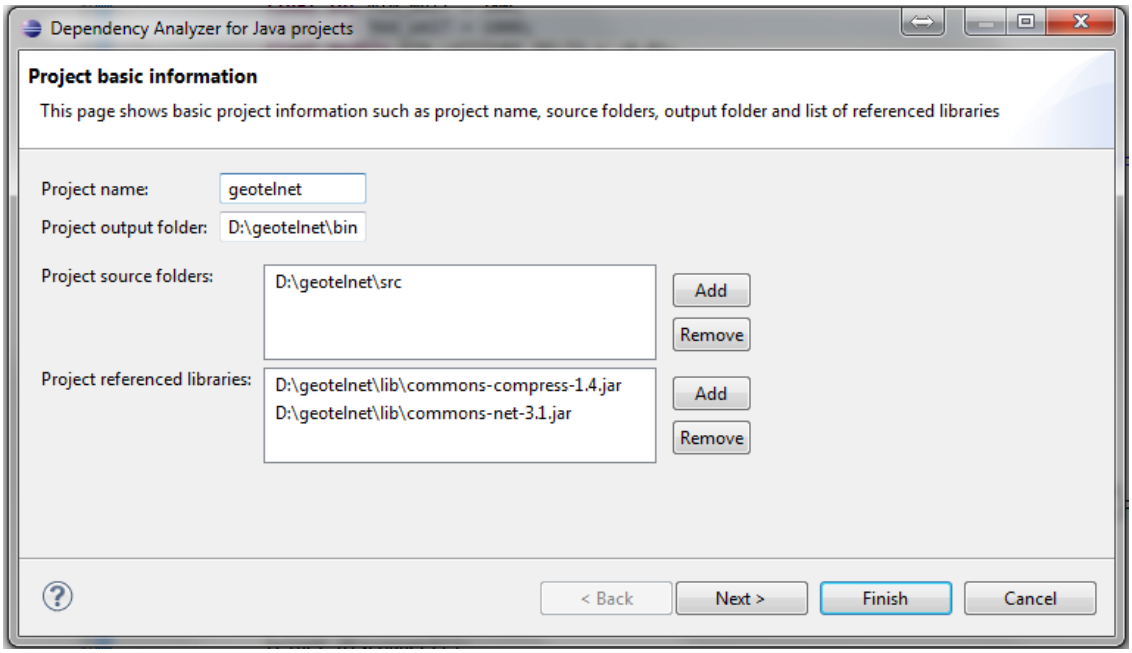
Listing 3.4: Vytvorenie Radio Button-u

```
Button radioButton = new Button(parentComposite, SWT.RADIO);
```

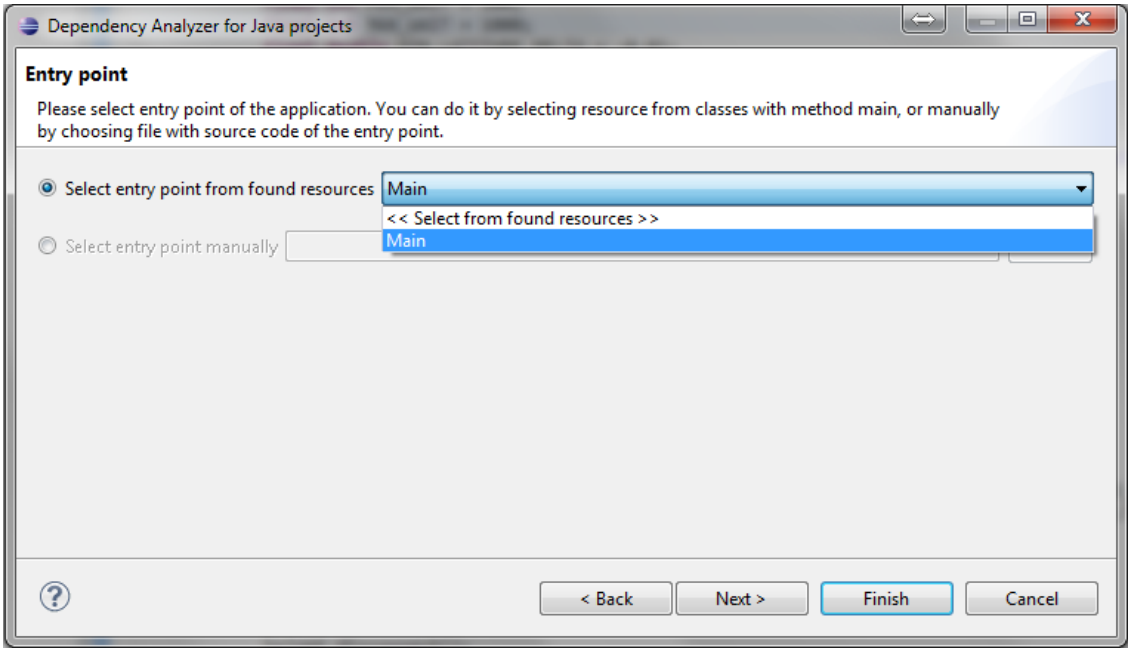
Možnosť vybrať vstupný bod zo zoznamu je realizovaný výberovým zoznamom - Combo typu `org.eclipse.swt.widgets.Combo`. Druhá možnosť je implementovaná ako textové pole s tlačidlom, po ktorého stlačení dôjde k vyvolaniu štandardného dialógu pre otvorenie súboru typu `org.eclipse.swt.widgets.FileDialog`. Ten je nastavený tak, aby zobrazoval len java súbory z adresára, ktorého cesta je uložená v objekte typu `WizardInput`.

Po prejdení na tretiu obrazovku začne analýza závislostí. Táto akcia môže chvíľu trvať, čo závisí od veľkosti projektu. Následne je zobrazená stránka, na ktorej sa nachádzajú dve tabuľky rozdelené do separátnych záložiek (tabov), čo je realizované prostredníctvom komponenty TabFolder (`org.eclipse.swt.widgets.TabFolder`). Ak sa pri analýze objavilo použitie reflexívneho modelu, je o tom užívateľ upozornený v hornej časti stránky.

Prvá záložka zobrazuje zoznam všetkých zdrojov projektu, závislých aj nezávislých. To je implementované podobne ako zoznam zdrojových adresárov na prvej stránke sprievodcu s tým rozdielom, že závislé zdroje sú zobrazené tučným písmom (bold). Túto zobrazovaciu funkcionalitu zabezpečujú trieda AllResourcesLabelProvider, ktorá implementuje dve rozhrania: `ITableLabelProvider` a `ITableFontProvider`, ktoré



Obr. 3.1: Základné informácie o projekte

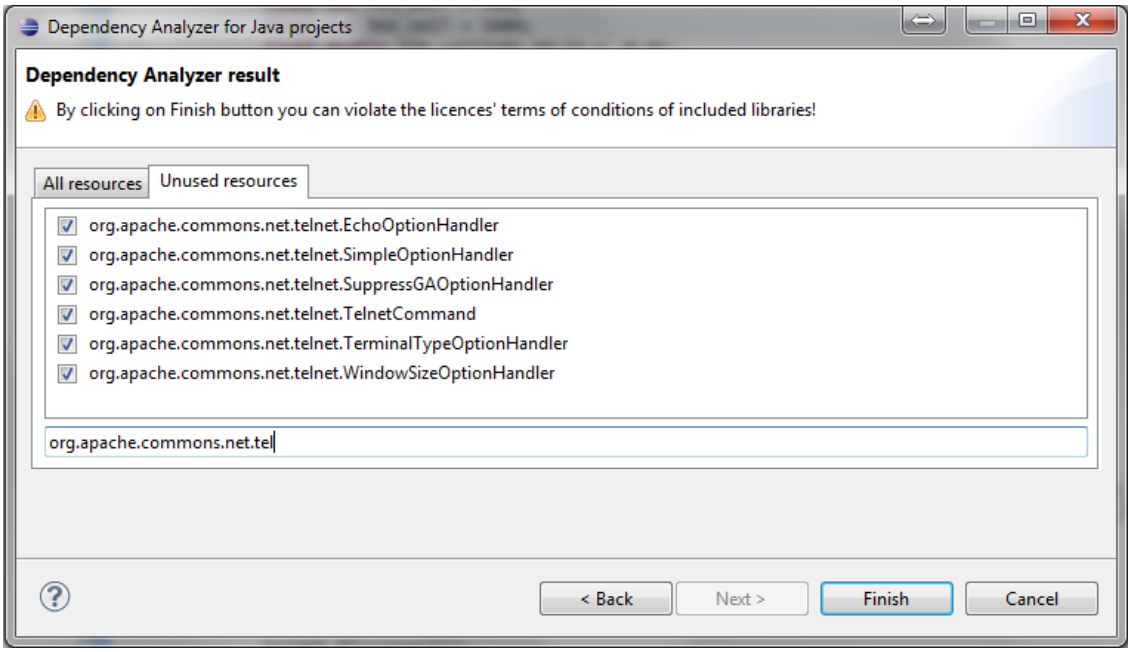


Obr. 3.2: Výber vstupného bodu

upravujú zobrazovanie elementov tabuľky.

Druhá záložka obsahuje zoznam nezávislých zdrojov projektu, ktoré zobrazuje v komponente `CheckboxTableViewer` z praktického hľadiska - užívateľ si môže ešte pred vymazaním odznačiť zdroje, ktoré nechce aby boli vymazané.

Pre orientáciu v zozname zdrojov sú zdroje abecedne vzostupne usporiadané podľa plne kvalifikovaného mena zdroja (fully qualified name). Je pridané filtrovacie textové pole, ktoré po zadaní textu obnoví zoznam zdrojov v danej záložke a zobrazí len tie zdroje, ktoré sa začínajú zadaným textom.
Ukončením sprievodcu sa spustí vymazávanie všetkých zaškrtnutých zdrojov zo zoznamu nepoužitých zdrojov.



Obr. 3.3: Zoznam nezávislých zdrojov

Záver

Práca **Statická analýza Java kódu** mala ako hlavný cieľ vytvoriť zásuvný modul do vývojového prostredia Eclipse, ktorý pre zadaný Java projekt zistí zoznam nezávislých zdrojov, teda takých, ktoré nie sú dosiahnuteľné zo vstupného bodu aplikácie. Vedľajším cieľom bolo vytvoriť architektúru pre zisťovanie nezávislých zdrojov projektu, ktorú je možné rozšíriť o ďalšie typy projektov.

Prvý cieľ bol splnený vytvorením zásuvného modulu do prostredia Eclipse s názvom **JADEAN** (Java Dependency Analyzer), ktorý vie zistiť zoznam nezávislých zdrojov Java projektu prehľadávaním zdrojov syntaktickou analýzou a prechádzaním syntaktického stromu a čítaním class súborov. Táto funkcionalita je však obmedzená použitím Reflection API jazyka Java, keďže je v prípade použitia tohoto modelu nerozhodnuteľné, či zdroj, ktorý bol vyššie spomínanými metódami označený ako nezávislý, bude v projekte závislý. Preto možné riešenia sú len aproximatívne. Možným riešením je vyhľadávať v zdrojoch volania metód Reflection API a na základe volaní vyextrahovať zdroj. Ďalšie možné riešenie však vyžaduje zmenu návrhu aplikácie (pridávať anotácie so zoznamom závislých zdrojov, alebo ich umiestniť na jedno miesto v aplikácii).

Druhý cieľ bol splnený vytvorením abstraktných tried, ktoré sú rozšíriteľné inými typmi projektu. Pre pridanie nového typu projektu stačí odvodiť triedy `Project`, `Resource` a `DependencyAnalyzerHandler` a zaregistrovať typ projektu v triede `DependencyAnalyzerHandlersFactory`.

Nakoľko bol splnený druhý cieľ, teda vytvorenie architektúry umožňujúcej ľahké rozšírenie o ďalšie typy projektov, ďalšími cieľmi tejto práce teda môžu byť doplnenia riešenia o podporu iných typov projektov, napríklad PHP projektov založených na

Zend rámci (Zend framework - [Zen12]), alebo projektov založených na budovacom systéme Maven ([The12]).

Rovnako, ďalším rozšírením tejto práce môže byť hľadanie spôsobov pre zisťovanie závislostí pri použití reflexívneho modelu. Popísali sme dve aproximatívne riešenia, ktoré sú pomerne triviálne. Pre rôzne typy projektov a s použitím rôznych typov knižníc môžu byť tieto spôsoby iné. Preto je to stále otvorená oblasť.

Vzhľadom na to, že jazyk Java sa neustále vyvýja, je potrebné aktualizovať aj balík implementujúci syntaktickú analýzu a čítanie class súborov. Ako vhodný kandidát pre syntaktickú analýzu zdrojových kódov jazyka Java verzie vyššej ako 1.5 sa ukazuje napríklad balík `org.eclipse.jdt.core.dom` ([Lar12]), ktorý má však pomerne rozsiahle závislosti v jadre Eclipse.

Literatúra

[Bar12] Barak Naveh and Contributors. JGraphT, 2012. http://jgrapht.org/.

[Eri95] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides. *Design Patterns, Elements of Reusable Object-Oriented Software 1st edition*. Addison-Wesley Pub Co, 1995.

[Eri08] Eric Clayberg and Dan Rubel. *Eclipse Plug-ins (3rd Edition)*. Addison-Wesley Professional, 2008.

[IBM05] IBM Corporation and others. Platform Plug-in Developer Guide, Feb 2005. http://www.eclipse.org/documentation/.

[jav12] javaparser - Java 1.5 Parser and AST, 2012. http://code.google.com/p/javaparser/.

[jde12] JDepend, 2012.

[Lar12] Lars Vogel. Eclipse JDT - Abstract Syntax Tree (AST) and the Java Model - Tutorial, Apr 2012. http://www.vogella.com/articles/EclipseJDT/article.html.

[Ora12] Oracle. Trail: The Reflection API, 2012. http://docs.oracle.com/javase/tutorial/reflect/index.html.

[son12] Sonar, 2012. http://www.sonarsource.org/.

[Sun12] Sun Microsystems. The class File Format, 2012. http://docs.oracle.com/javase/specs/jvms/se5.0/html/ClassFile.doc.html.

[The12] The Apache Software Foundation. Maven, 2012. http://maven.-apache.org/.

[ucd12] UCDetector, 2012. http://www.ucdetector.org/.

[Zen12] Zend Technologies Ltd. Zend Framework, 2012. http://framework.-zend.com/.

Zoznam príloh

Dole je uvedený zoznám príloh na pribalenom CD.

1. Zdrojový kód zásuvného modulu
2. Skompilované zdrojové súbory
3. Návod na inštaláciu
4. Ukážkový projekt