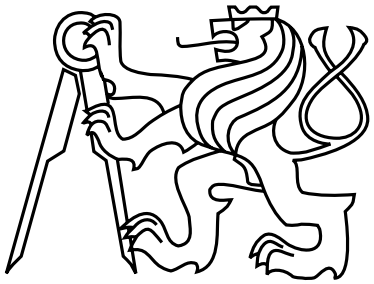
České vysoké učení technické v Praze
Fakulta elektrotechnická
Katedra počítačové grafiky a interakce



Diplomová práce

Validace principů objektového návrhu v kódu

Bc. Martin Vejmelka

Vedoucí práce: Ing. Ondřej Macek

Studijní program: Otevřená informatika, Navazující magisterský

Obor: Softwarové inženýrství

13. května 2011

Poděkování

Na tomto místě bych rád poděkoval Ing. Ondřeji Mackovi za vedení této práce a značnou podporu při její realizaci. Současně děkuji všem svým blízkým, v nichž jsem měl oporu během studia a tvorby této práce.

Prohlášení

Prohlašuji, že jsem práci vypracoval samostatně a použil jsem pouze podklady uvedené v přiloženém seznamu.
Nemám závažný důvod proti užití tohoto školního díla ve smyslu §60 Zákona č. 121/2000 Sb., o právu autorském, o právech souvisejících s právem autorským a o změně některých zákonů (autorský zákon).

V Praze dne 13. 5. 2011 .....

Abstract

This works deals with checking correctness of object-oriented software design using source code. New language for design principles description is proposed along with a tool which would allow us to check specified rules. Results of this work include theoretical basement for design principles formalization and extensible evaluation system, which is able to check validity of the rules specified in proposed formalization. Functionality of the tool is demonstrated on sample input projects.

Abstrakt

Práce se zabývá kontrolou správného objektového návrhu software na základě analýzy zdrojových kódů. Je navržen jazyk pro popis návrhových principů a nástroj, který umožňuje jejich ověřování. Výsledkem práce je teoretický základ pro formalizaci návrhových pravidel a rozšiřitelný vyhodnocovací systém, který umožňuje platnost pravidel realizovaných v této formalizaci ověřit. Činnost výsledného nástroje je demonstrována na vzorových příkladech.

Obsah

1 Úvod **1**
- 1.1 Motivace projektu, záměr práce . . . 1
- 1.2 Úvod do řešené problematiky . . . 2
- 1.3 Struktura práce . . . 3

2 Specifikace cílů práce **5**
- 2.1 Požadavky na analýzu základních návrhových principů . . . 5
- 2.2 Požadavky na formalizaci pravidel . . . 5
- 2.3 Požadavky na systém pro vyhodnocování pravidel . . . 6
  - 2.3.1 Funkční požadavky na výsledný systém . . . 6
  - 2.3.2 Nefunkční požadavky na výsledný systém . . . 7
- 2.4 Rešerše existujících řešení . . . 7
  - 2.4.1 CheckStyle . . . 7
  - 2.4.2 DP-Miner . . . 7
  - 2.4.3 FindBugs . . . 8
  - 2.4.4 JDepend . . . 8
  - 2.4.5 Macker . . . 8
  - 2.4.6 PMD . . . 8
  - 2.4.7 QJ-Pro . . . 9
  - 2.4.8 Soot . . . 9
  - 2.4.9 Squale . . . 9

3 Analýza **11**
- 3.1 Problematika návrhu software . . . 11
  - 3.1.1 Koncepty používané při návrhu software . . . 11
  - 3.1.2 Aspekty návrhu software . . . 12
  - 3.1.3 Objektově orientovaný návrh software . . . 14
- 3.2 Analýza principů objektového návrhu . . . 14
  - 3.2.1 Analyzované principy . . . 15
    - 3.2.1.1 Low coupling/dependency (nízká závislost/vazba) . . . 15
    - 3.2.1.2 High cohesion (vysoká koheze/soudržnost) . . . 17
    - 3.2.1.3 Law of Demeter . . . 18
  - 3.2.2 Ukázky kódu porušujícího některá z pravidel . . . 19
    - 3.2.2.1 Porušení principu law of Demeter . . . 20
    - 3.2.2.2 Ukázka kódu s příliš vysokou úrovní závislosti . . . 21

3.2.2.3 Ukázka kódu s nízkou úrovní koheze . . . 22
3.3 Analýza vstupních projektů realizovaných v jazyce Java . . . 23
3.3.1 Statický model programu v Javě . . . 23
3.3.1.1 Struktura softwarového projektu v Javě . . . 23
3.3.1.2 Syntaktické elementy programovacího jazyka Java . . . 24
3.3.2 Zpracovávání zdrojových kódů v jazyce Java . . . 25
3.3.2.1 Vlastní kód pro zpracovávání zdrojových kódů . . . 26
3.3.2.2 Použití compiler-compiler systému . . . 26
3.3.2.3 Použití vhodné knihovny nebo existujícího programu . . . 27
3.3.2.4 Použití prostředků poskytovaných platformou Java . . . 27
3.3.2.5 Použití prostředků současných vývojových IDE . . . 28

4 Návrh **29**
4.1 Návrh způsobu specifikace pravidel . . . 29
4.1.1 Formalizace modelu programu pomocí grafu . . . 29
4.1.2 Formalizace pravidel . . . 33
4.1.3 Analytický přístup k návrhovým principům . . . 35
4.2 Návrh architektury systému . . . 36
4.2.1 Globální struktura systému . . . 36
4.2.2 Doménové objekty . . . 37
4.2.2.1 Třída ValidationModel . . . 39
4.2.2.2 Třída ValidationReport . . . 39
4.2.3 Jádro systému . . . 40
4.2.3.1 Komponenta GraphModelGenerator . . . 41
4.2.3.2 Komponenta ValidationModelGenerator . . . 42
4.2.3.3 Komponenta ValidationTask . . . 43
4.2.3.4 Komponenta ArchVal (fasáda) . . . 43
4.2.3.5 Rozhraní registrů poskytovatelů implementací . . . 44
4.2.4 Rozhraní rozšíření systému . . . 45
4.2.4.1 Rozhraní GraphGeneratorIface . . . 45
4.2.4.2 Rozhraní OperatorIface . . . 45
4.2.4.3 Rozhraní AnalysisIface . . . 46
4.2.5 Základní průběh validačního procesu . . . 46
4.3 Návrh vstupního rozhraní (zadávání pravidel) . . . 46
4.4 Návrh výstupního rozhraní (provádění validace) . . . 47
4.5 Návrh způsobu integrace do různých typů prostředí . . . 47
4.6 Návrh technologií pro implementaci . . . 48

5 Implementace **49**
5.1 Specifikace pravidel požadovaných zadáním práce . . . 49
5.1.1 Law of Demeter . . . 49
5.1.2 Low coupling, high cohesion . . . 51
5.2 Implementace jádra systému . . . 52
5.2.1 Implementace formátu AVD . . . 52
5.2.2 Implementace sestavování validačního modelu . . . 52
5.2.3 Implementace základních operátorů . . . 53

5.3 Implementace modulů rozšíření . . . 54
5.3.1 Modul av-graphgen-demeter (generátor grafu) . . . 54
5.3.2 Modul av-operators-demeter (operátory pro LoD) . . . 54
5.4 Integrace do NetBeans IDE . . . 55
5.4.1 Použití jádra systému ArchVal . . . 55
5.4.2 Přidání uživatelské akce do nabídky prostředí . . . 56
5.4.3 Výstupní rozhraní . . . 56
5.4.4 Implementace registrů poskytovatelů služeb . . . 56
5.4.5 Podpora editace AVD souborů . . . 56

6 Testování **59**
6.1 Testování jednotek . . . 59
6.2 Ověřování funkčnosti na příkladech . . . 59
6.2.1 Příklady porušení principu LoD . . . 60
6.2.2 Falešně pozitivní případy . . . 60
6.2.3 Speciální případy . . . 60

7 Závěr **61**

A Seznam použitých zkratek **67**

B Instalační a uživatelská příručka **69**
B.1 Instalace programu . . . 69
B.2 Používání programu . . . 69
B.3 Gramatika jazyka pro specifikaci pravidel . . . 70

C Obsah přiloženého CD **77**

Seznam obrázků

1.1 Rozšíření gramatiky jazyka o množinu pravidel. 2

2.1 Struktura systému. 6

3.1 Znázornění vztahů mezi analyzovanými návrhovými principy. 15
3.2 Formy návrhového principu *LoD*. 19
3.3 Struktura základních syntaktických elementů programovacího jazyka Java. . . 25
3.4 Rozklad elementu TypeDeclaration. 25

4.1 Příklad formalizace hierarchie tříd pomocí grafu. 30
4.2 Body rozšíření systému. 37
4.3 Dekompozice systému na jádro a rozšíření. 38
4.4 Komponenty jádra systému *ArchVal*. 41
4.5 Znázornění vstupů a výstupů komponenty typu *GraphModelGenerator*. 42
4.6 Znázornění vstupů a výstupů komponenty *ValidationModelGenerator*. 42

5.1 Graf „typu demeter“ použitý pro validaci principu *LoD*. 51

Seznam tabulek

4.1 Tabulka doménových objektů. 38
4.2 Tabulka komponent jádra systému. 40

5.1 Tabulka integračních komponent systému. 55

C.1 Seznam souborů na CD. 77

Seznam výpisů kódu

5.1 Implementace univerzálního kvantifikátoru ∀. 53
5.2 Implementace existenčního kvantifikátoru ∃. 53
B.1 Gramatika jazyka pro specifikaci pravidel (AVD soubory). 70

Kapitola 1

Úvod

1.1 Motivace projektu, záměr práce

Při realizaci softwarových projektů zpravidla postupujeme od *sběru požadavků*, přes *analýzu* a *návrh* až k *fyzické realizaci* programového produktu ve zdrojovém kódu vhodného programovacího jazyka [40]. Vytvořením reprezentace systému ve zdrojovém kódu však životní cyklus softwarového díla nekončí. Vedle *testování*, *integrace*, *akceptace* a *dodávky* je nutné ještě uvažovat o další fázi – *údržbě*. Pro tyto fáze, které jsou neméně důležité, leč mnohdy opomíjené, je kritická především *architektura systému*.

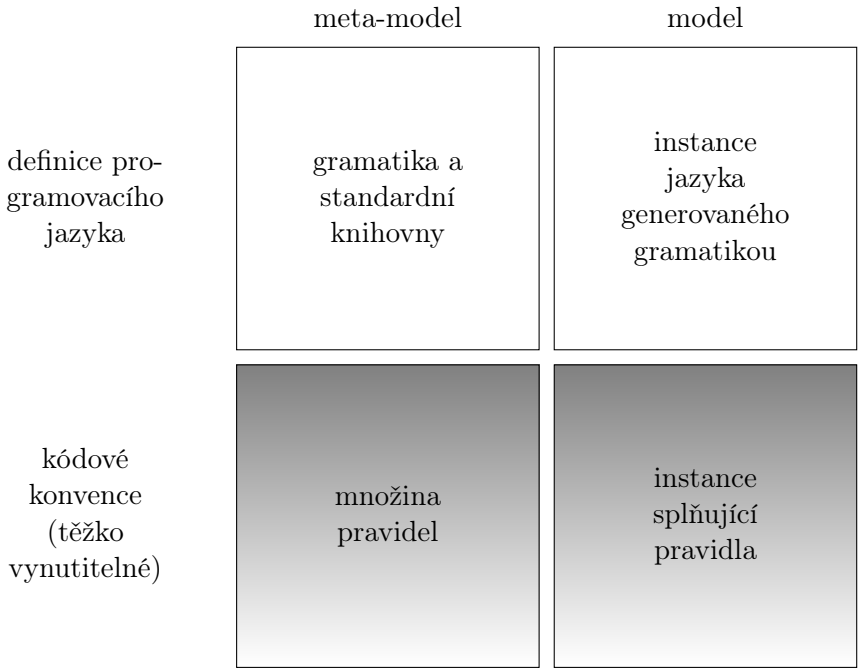
Architektura existujícího systému je většinou definována poměrně vágně. Při dalších úpravách se tak můžeme spolehnout na malou podmnožinu pravidel, kterou u daného systému předpokládáme a která často ani nemusí platit. Vždy se však můžeme spolehnout na to, že pokud zdrojový kód programového produktu projde bez chyb překladačem nebo interpretrem programovacího/skriptovacího jazyka, v němž je napsán, jedná se o kód podléhající pravidlům tohoto jazyka.

Syntaxe jazyka je definována gramatikou, sémantika potom dává jednotlivým konstruktům jazyka jejich význam (např.: `for` bude znamenat opakované provádění kódu mezi počáteční `{` a koncovou `}` složenou závorkou následujícího bloku).

Kromě struktury, kterou vynucuje překladač jazyka bývá často zavedena množina kódových konvencí. Vedle konvencí pro formátování zdrojového kódu (které nemají z hlediska analýzy architektury systému význam) můžeme zmínit konvence, které zakazují aby měla metoda více než nějaký daný počet příkazů/parametrů, konvence, které nařizují, aby se v kódu nevyskytovaly neprovolávané („mrtvé“) části kódu, a další.

V této práci se pokusíme zmíněné konvence zachytit jako pravidla, která lze vhodným způsobem ověřit. Znázornění tohoto konceptu je na obrázku 1.1. Množina pravidel zpřesní meta-model tvořený gramatikou jazyka – omezí množiny instancí daného jazyka na podmnožinu, která splňuje požadovaná pravidla.

Jistou analogii můžeme vidět ve značkovacím jazyce XML v pojmech *well-formedness* versus *validity*. Gramatika udává, že se v textu mohou vyskytovat elementy a atributy, že se nesmí „křížit“ jednotlivé tagy atd. Další dodatečná informace (DTD, XSchema, RELAX NG) potom zpřesní, jaké tagy, v jakém pořadí a počtu se smí ve výsledné instanci XML dokumentu vyskytovat.



Obrázek 1.1: Rozšíření gramatiky jazyka o množinu pravidel.

1.2 Úvod do řešené problematiky

Architektura softwarového systému [41] je dána množinou pravidel a tvrzení o uspořádání systému, která platí pro daný softwarový systém. Architekturu zpravidla uvažujeme z různých pohledů (views) [36], které popisují systémové komponenty a vztahy mezi nimi vždy z úhlu některé ze zainteresovaných stran (stakeholders). Např. *logical view* se zabývá architekturou z pohledu funkcionality, kterou systém poskytuje *koncovým uživatelům*, *physical view* poskytuje pohled pro *systémového inženýra* – uspořádání komponent v systému a propojení mezi nimi.

Pro systémového návrháře bude nejdůležitější právě architektura na fyzické úrovni, se kterou bude muset dále pracovat v případě dodatečných zásahů do systému. Jedná se o vztahy mezi konkrétními komponentami realizované v existujícím systému. Ačkoliv sem spadají všechny komponenty systému (tedy i hardwarové), budeme se v této práci zabývat zejména jeho softwarovými součástmi – funkčními bloky a moduly.

Základní prostředky pro dělení softwarového díla na vhodné funkční moduly a bloky poskytuje již většina moderních programovacích jazyků. Různé programovací jazyky podporují různá paradigmata a následně i strukturování zdrojových kódů programu [39]. Ve funkcionálním programovacím jazyce budeme strukturovat dílo do zanořených funkcí, u objektově orientovaného programování budou základními moduly třídy a balíčky. V textu práce se zaměříme právě na objektově orientované programovací jazyky.

Prostředky jazyka, které podporují správný návrh software rozšíříme navíc o množinu pravidel tak, jak bylo uvedeno v sekci 1.1. Množina těchto pravidel může být formulována neformálně ve formě požadavků na to, co by měla splňovat výsledná architektura. Uveďme

příklady takových formulací:

- „v běžném kódu by se *neměla vyskytovat* (nestatická) pole s modifikátorem `public`, pouze `private`, k nimž se přistupuje pomocí *getterů* a *setterů*“,
- „třídy z balíčku A *nesmí záviset* na jiných nesystémových *třídách*, ale nejvýše na *rozhraních* balíčku B“ (programování proti rozhraní namísto proti konkrétní implementaci),
- „třídy v tomto balíčku smí mít *maximálně pět metod*, z nichž každá smí mít *nejvýše tři parametry*“ (může být konvence v nějaké firmě).

Pro podobné neformální specifikace se pokusíme stanovit formalismus pomocí kterého bude možné uvedená pravidla zapsat a následně vyhodnotit. V optimálním případě by bylo možné na základě výstupu z validačního nástroje buď navrhnout nebo (je-li to možné) dokonce opravit návrh tak, aby pravidla splňoval.

1.3 Struktura práce

Práce je strukturována na základě běžného životního cyklu softwarového projektu. *Úvodní kapitola* zahrnuje motivaci a deklaraci záměru práce a poskytuje úvod do řešené problematiky. V kapitole druhé (*specifikace cílů práce*) jsou specifikovány konkrétní požadavky na výsledky práce. Fáze *analýzy* (kapitola 3) postupně provádí rešerši problematiky správného návrhu software, zabývá se analýzou projektů, které budou vstupem pro vyvíjený nástroj a poskytuje přehled o existujících možnostech zpracování zdrojového kódu v jazyce Java. *Návrh* řešení je poskytován v kapitole 4. Zde se zabýváme jednak návrhem formalizace pravidel pomocí vhodné matematické reprezentace a dále návrhem samotného nástroje pro ověřování těchto pravidel. Popis implementace návrhu je poskytován v kapitole 5, kde se zaměřujeme na podstatné části implementace. Je zde provedena formalizace pravidla *Law of Demeter* pomocí navrženého formalismu. Taktéž jsou na tomto místě popsány zajímavé implementační detaily vyvíjeného systému. Kapitola *testování* popisuje prováděné testování systému. *Závěr* shrnuje obsah a výsledky práce.

V přílohách nalezne čtenář seznam zkratek použitých v textu této práce, instalační a uživatelskou příručku realizovaného systému a popis obsahu přiloženého CD, které je nedílnou součástí této práce.

Kapitola 2

Specifikace cílů práce

Účelem této sekce je stanovení přesného popisu řešené problematiky a cílů práce. Tyto cíle popíšeme formou požadavků na výsledky práce. Při specifikaci požadavků budeme vycházet ze zadání. Na základě zadání můžeme specifikovat tři oblasti požadavků.

V první oblasti se budeme zabývat požadavky na rešeršní části práce – analýzu *principů* používaných při *objektově orientovaném návrhu a implementaci*. Část druhá bude obsahovat požadavky na *formalizaci pravidel*, která umožní popsat principy analyzované v části první. Třetí část poskytne rozbor *požadavků na systém*, který by měl demonstrovat vyhodnocování/validaci definovaných pravidel na existujících zdrojových kódech.

Na konci kapitoly rozebereme v rámci jedné sekce existující podobná řešení/nástroje, jejich výhody a nevýhody.

2.1 Požadavky na analýzu základních návrhových principů

Zadání práce specifikuje konkrétně tři návrhové principy – *low coupling*, *high cohesion* a *Law of Demeter*. Tyto principy jsou si v mnohém podobné. Všechny se zabývají zejména závislostmi mezi částmi zdrojových kódů a jejich propojeností.

V části analýzy je nutné projít postupně tyto návrhové principy a určit jejich vlastnosti a možnosti ověřování. To zahrnuje zejména provedení rešerší a specifikaci konkrétních tvrzení, která bude následně možné převést na pravidla. Může se ukázat, že ne všechny principy jsou exaktně definované, ověřitelné a vynutitelné. V takových případech je možné uvažovat spíše statistický přístup oproti ověřování pravidel. Výsledkem by potom nebyl výsledek splňuje/nesplňuje, ale např. množina statistických atributů (features), na níž by bylo možné provádět další druhy analýzy (klasifikaci, pattern matching, ...).

Výsledkem analýzy principů by mělo být především vymezení oblasti, kterou se budeme zabývat v další části (*formalizace pravidel*).

2.2 Požadavky na formalizaci pravidel

Formalizace pravidel bude mít dvě hlavní součásti. V obecné části bude definován model nad nímž budeme stavět pravidla a formát/způsob zápisu pravidel. Konkrétní část potom

poskytne specifikaci pravidel *Law of Demeter*, *low coupling* a *high cohesion* v navrženém formalismu (bude-li to možné).

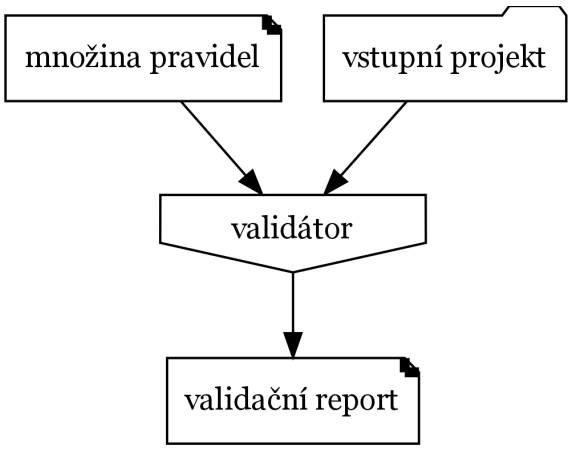
Definice modelu bude zahrnovat určení vhodné reprezentace problémové domény. Je třeba provést namapování analyzovaných objektů (elementy zdrojového kódu) na zvolenou reprezentaci domény (model) a poskytnout jazyk, pomocí něhož bude možné specifikovat pravidla, která chceme ověřit.

Po vytvoření vhodného modelu je nutné tento model následně převést do počítačové reprezentace a poskytnout vhodnou serializaci navrženého jazyka[1], abychom mohli pravidla vyhodnotit pomocí systému pro vyhodnocování pravidel.

2.3 Požadavky na systém pro vyhodnocování pravidel

Třetí část požadovaná zadáním zahrnuje vytvoření nástroje pro ověřování pravidel (*validátor*). Vstupem pro tento nástroj bude *množina pravidel* a vhodně předzpracované zdrojové kódy analyzovaného projektu (*vstupní projekt*). Nástroj může být navržen obecněji, budeme-li na vstupu předpokládat vhodnou vnitřní reprezentaci zdrojových kódů. Doprogramováním vstupních modulů[2] bude možné poskytnout podporu i pro další jazyky.

Celková struktura výsledného systému je znázorněna na obrázku 2.1.



Obrázek 2.1: Struktura systému.

Výstupem systému by měl být *validační report*, který vypíše, která pravidla ze vstupního souboru byla porušena. Formát vstupních pravidel a výstupního reportu bude definován dále.

2.3.1 Funkční požadavky na výsledný systém

Na základě obecných požadavků můžeme specifikovat základní funkční požadavky. Požadujeme, aby systém uměl provádět následující funkčnosti:

[1] Jazyk může obsahovat různé matematické symboly ($\forall$, $\exists$, atd.), které je obtížné zadávat do počítače přímo. Proto je vhodné poskytnout alternativní reprezentaci jazyka zahrnující pouze omezené množství znaků.

[2] Vstupní moduly musí provést zparsování zdrojových kódu, vygenerování AST a rozlišení jmen a typů (name/type resolution).

- načíst vhodně předzpracovaný projekt realizovaný v jazyce Java a vytvořit vnitřní model pro další analýzu,
- načíst ze vstupního souboru množinu pravidel v definovaném formátu,
- provést validaci načteného modelu pomocí množiny pravidel,
- vypsat report obsahující informace o splnění resp. porušení pravidel.

2.3.2 Nefunkční požadavky na výsledný systém

Z nefunkčních požadavků je zadáním práce dán požadavek na *rozšiřitelnost*. Systém by mělo být možné rozšířit o další modely a možnosti specifikace nových/složitějších pravidel nad modely.

2.4 Rešerše existujících řešení

V této sekci se postupně podíváme na různé nástroje, které si kladou za cíl ověřování kvality kódu na různých úrovních. Každý z těchto nástrojů pojímá kontrolu kvality zdrojového kódu poněkud jiným způsobem. Některé se zabývají „pouhou“ analýzou na úrovni jedné kompilační jednotky[3], jiné provádějí strukturální analýzu na úrovni vztahů mezi jednotlivými třídami.

2.4.1 CheckStyle

CheckStyle je vývojový nástroj, který si klade za cíl pomoci programátorům psát Java kód, který vyhovuje konkrétním kódovým konvencím [2]. Pracuje na úrovni jednoduchého zpracovávání `*.java` souborů a kontroluje např. přítomnost `javadoc` komentářů nad třídami, atributy a metodami, jmenné konvence, maximální povolené počty parametrů, délky řádků, duplicitní kód ad. [32]. Zvládne i jednodušší testy na zjištění složitosti kódu (podle klíčových slov `for`, `while`, ...). Kódové konvence jsou konfigurovatelné pomocí XML souborů, kde lze vybírat, která předdefinovaná pravidla se budou v kódu kontrolovat (např. pravidlo „AvoidStarImport“).

2.4.2 DP-Miner

Autoři článku [23] a nástroje *DP-Miner* [3] se zabývají výzkumem v oblasti návrhových vzorů. Ukazuje se, že návrhové vzory jsou velmi důležité a extenzivně využívané ve fázi návrhu software. Při realizaci v konkrétním zdrojovém kódu se však informace o návrhových vzorech zpravidla ztrácí. Cílem nástroje *DP-Miner* je objevování návrhových vzorů v existujících projektech, k nimž neexistuje vývojová dokumentace. Ve článku [23] jsou demonstrovány schopnosti odhalit návrhové vzory na reálných projektech jako např. *JUnit*, *JEdit*, *Java.AWT* a dalších. Pro odhalování vzorů používají autoři matici vztahů mezi třídami analyzovaného systému.

[3] Kompilační jednotka je u jazyka Java představována jedním `*.java` souborem.

2.4.3 FindBugs

Nástroj *FindBugs* [4] vyhledává potenciální zdroje chyb v Java programech. K vyhledávání chyb využívá chybových vzorů. Chybové zdroje jsou idiomy, které se ve zdrojových kódech často vyskytují chybně použity. Většinou se jedná o obtížněji zvládnutelné vlastnosti jazyka, nepochopení rozhraní některého API apod. (ypickým příkladem může být porovnávání objektů `String` pomocí operátoru `==`). Pro odhalování chyb je používána statická analýza. *FindBugs* pracuje nad byte kódem (analyzuje zkompilované třídní `*.class` soubory) – je tedy možné analyzovat i programy, k nimž nemáme k dispozici zdrojové kódy. Analýza je mnohdy nepřesná (ne všechny vzory chyb jsou vždy opravdu chybami – může se jednat o záměrné použití daného idiomu). Nástroj v takovém případě chybně vypisuje varování, která neznamenají chybu (falešně pozitivní případy).

2.4.4 JDepend

Podobně jako předchozí nástroj pracuje i *JDepend* [5] nad zkompilovanými `*.class` soubory jazyka Java. Funguje však na jiném principu a na daleko vyšší úrovni. Zabývá se analýzou závislostí mezi třídami a balíčky. Systém poskytuje výpis základních metrik, jako je počet tříd a rozhraní v balíčcích, počet závislých balíčků (to je zde označováno jako „zodpovědnost“ – čím více tříd na balíčku závisí, tím větší je jeho zodpovědnost). Vesměs se jedná o metriky založené na počtech elementů v projektu. Za zmínku stojí odhalování cyklů v závislostech balíčků.

2.4.5 Macker

Velmi zajímavým nástrojem je *Macker* [6]. Pracuje na strukturální úrovni během kompilace a umožňuje kontrolovat nejrůznější strukturální pravidla. Stejně jako předchozí dva nástroje operuje nad zkompilovanými `*.class` soubory. Mezi zajímavá pravidla, která je schopen vynutit patří:

- „třídy z UI vrstvy nesmí přímo přistupovat k objektům v datové vrstvě nebo používat třídy v balíčku `java.sql`“,
- „externí systémy nesmí přistupovat k interním implementačním třídám (které mají sufix 'Impl')“,
- „jeden funkční modul smí přistupovat ke druhému pouze přes jeho API“.

Z uvedených pravidel je patrné, že *Macker* operuje na globální strukturální úrovni a je schopen vynutit dodatečnou kontrolu přístupu (vedle standardních prostředků jazyka Java, kterými jsou modifikátory přístupu).

2.4.6 PMD

Velmi rozšířený nástroj *PMD* [7] prochází zdrojové kódy jazyka Java a vyhledává potenciální zdroje problémů (srov. s nástrojem *FindBugs*). Příkladem vyhledávaných vzorů jsou:

- možné chyby – prázdné příkazy `try`, `catch`, `finally`, `switch`,
- mrtvý kód – nepoužité lokální proměnné, parametry a privátní metody[4],
- neoptimální kód – neefektivní využívání tříd `String`/`StringBuffer`,
- příliš komplikované výrazy – zbytečné `if` příkazy, cykly `for`, které mohou být realizovány pomocí `while`.

Výhodou *PMD* je integrovanost s velkým množstvím nástrojů (JDeveloper, Eclipse, JEdit, JBuilder, ItelliJ IDEA, Maven, Emacs).

2.4.7 QJ-Pro

Dalším pokročilým nástrojem je *QJ-Pro* [8]. Pracuje nad zdrojovými kódy a poskytuje velké množství pravidel, která lze vyhodnocovat. Kromě vynucení běžných kódových konvencí podporuje navíc analýzu nejrůznějších metrik (počet metod na třídu, poměr privátních a veřejných polí). Existuje integrace do vývojových prostředí – např. Eclipse IDE a Borland JBuilder.

2.4.8 Soot

Poněkud odlišným nástrojem je *Soot* [9]. Jedná se o framework pro optimalizaci Java kódu. Na rozdíl od předchozích zmiňovaných nástrojů, které jsou určeny spíše běžným programátorům, se jedná výzkumný projekt a platformu, na které je možné vyvíjet nástroje pro optimalizaci a analýzu. Nástroj pracuje nad Java byte kódem. Existuje integrace do Eclipse IDE.

2.4.9 Squale

Squale[5] [10] představuje zcela jiný pohled na práci se zdrojovými kódy. Nejedná se o běžný analyzátor kódu (jako téměř všechny předchozí případy) ale spíše o monitorovací nástroj, který umožní sledovat přehledně metriky zdrojových kódů. Případy užití tohoto nástroje tedy spadají spíše do oblasti managementu (vyhodnocování výkonnosti týmu, kvality jejich kódu).

[4] To je dnes běžné téměř pro všechna nejčastěji používaná IDE pro vývoj v jazyce Java.
[5] Uvedený název je odvozen z úplného názvu „Software QUALity Enhancement“.

Kapitola 3

Analýza

Kapitola *analýza* se zaměřuje zejména na rešerše v oblasti problémové domény a ke zjištění informací potřebných k realizaci práce. Je rozdělena celkem na tři základní části.

V části první se zabýváme obecnou problematikou návrhu software, často používanými koncepty, aspekty, které je třeba při návrhu zohlednit a objektově orientovaným návrhem.

Druhá část se zabývá konkrétními principy objektového návrhu, konkrétně principy *low coupling*, *high cohesion* a *Law of Demeter*. Kromě rešerše týkající se těchto principů jsou zde demonstrovány jednoduché ukázky jejich porušení.

Poslední část se zabývá analýzou Java projektů, které budou vstupem pro validaci a analýzu. Je zde rozebráno, z jakých souborů se projekt realizovaný v jazyce Java skládá a jaká je základní struktura zdrojových `*.java` souborů. Dále tato část obsahuje rešerši existujících nástrojů pro předzpracování zdrojových kódů do podoby vhodné pro další práci.

3.1 Problematika návrhu software

Při návrhu software se uplatňují nejrůznější přístupy, vzory, pravidla. Cílem této sekce je podat rychlý přehled o používaných návrhových konceptech, aspektech návrhu software a objektově orientovaném návrhu. U některých aspektů uvažujeme možnost jejich podpory pomocí vhodného nástroje. Mnoho z uvedených pojmů a postupů je dále použito při návrhu nástroje, jehož vytvoření je jedním z cílů této práce.

3.1.1 Koncepty používané při návrhu software

Existující koncepty návrhu software poskytují vývojářům základ, z něhož lze odvodit a aplikovat další sofistikované metody [42]. Většinou se jedná o různé pohledy na způsob dekompozice systému na části realizovatelné v konkrétním programovacím jazyce. Postupem času se vyvinula celá množina konceptů návrhu software [20]. Uveďme alespoň některé:

Abstrakce (Abstraction) Abstrakce je proces nebo výsledek zobecnění a redukce informačního obsahu konceptu nebo pozorovatelného jevu. Typicky ponecháváme pouze informaci, která je relevantní pro konkrétní účel nebo záměr.

Postupné zjemňování (Refinement) Postupné zjemňování/zpřesňování představuje proces tvorby[1]. Je vytvářena hierarchie dekompozicí makroskopických příkazů funkce po krocích, dokud nejsou dosaženy elementární příkazy programovacího jazyka. V každém kroku je dekomponována jedna nebo více instrukcí daného programu do detailnějších instrukcí. Abstrakce a zjemňování jsou vzájemně komplementární oblasti.

Modularita (Modularity) Systém je rozdělen do samostatných komponent nazvaných moduly (popsáno dále v podsekci 3.1.2).

Architektura (Software Architecture) Tento pojem odkazuje na celkovou strukturu softwarového systému a způsoby, kterými tato struktura zajišťuje konceptuální ucelenost systému.

Hierarchie řízení (Control Hierarchy) Struktura programu, která reprezentuje organizaci programových komponent a určuje hierarchii řízení. Tato struktura však nedefinuje přesné pořadí operací.

Strukturální dělení (Structural Partitioning) Struktura programu může být definována jak v horizontálním, tak vertikálním smyslu. Horizontální dělení představuje samostatné větve hierarchie modulů pro hlavní funkčnosti programu. Vertikální dělení slouží k distribuci řízení a průběhu zpracování do jednotlivých úrovní (vysokoúrovňové rutiny a nízkoúrovňové operace).

Struktura dat (Data Structure) Logická reprezentace vztahů mezi jednotlivými datovými elementy.

Softwarová procedura (Software Procedure) Koncept *softwarové procedury* se zaměřuje na průběh zpracovávání každého jednotlivého modulu. Na rozdíl od konceptu *hierarchie řízení*, který se zabývá pouze vztahy (která komponenta je řízena kterou komponentou), se zaměřuje na návrh přesné posloupnosti kroků, které modul provádí.

Zapouzdření (Information Hiding) Moduly by měly být navrhovány tak, aby informace v nich obsažená byla nedostupná modulům, které tuto informaci nepotřebují.

3.1.2 Aspekty návrhu software

Při návrhu software musíme vzít v potaz velké množství aspektů [42]. Důležitost přikládaná jednotlivým aspektům je odvozena od účelu, za nímž je software realizován. Pro některé systémy může být klíčová rozšiřitelnost, pro jiné stabilita. Mezi základní aspekty patří:

- *kompatibilita (compatibility)*,

[1] Tento koncept byl použit i pro tvorbu této práce. Nejprve byla provedena dekompozice tématu na základní bloky (specifikace, analýza, návrh, ...) a poté byly „rekurzivním sestupem“ realizované další sekce, podsekce a nakonec i elementární součásti práce (odstavce a věty).

- *rozšiřitelnost (extensibility)*,
- *zotavení se z chyb (fault-tolerance/graceful degradation)* [35],
- *udržovatelnost (maintainability)*,
- *modularita (modularity)*,
- *balení (packaging)*,
- *spolehlivost (reliability)*,
- *znovupoužitelnost (reusability)*,
- *stabilita (robustness)*,
- *bezpečnost (security)*,
- *použitelnost (usability)*.

Zatímco některé aspekty jsou pro tuto práci zcela nepodstatné (dobrým příkladem je *balení*, které se zabývá způsobem dodávky software a podpůrných materiálů), jiné je možné podpořit pomocí vhodného nástroje. Pojďme se nyní podívat na aspekty, jejichž podpora možná je:

Rozšiřitelnost Tato vlastnost znamená praktický důsledek, kdy nové funkčnosti mohou být do existujícího systému přidány bez majoritních změn ve *výchozí architektuře*. Pokud by bylo možné určit pravidla, která určují základní architekturu a která je možné vynutit, bylo by možné při rozšiřování systému tato pravidla opět aplikovat, což by vedlo k dodržení výchozí architektury.

Udržovatelnost Klíčovou charakteristikou *udržovatelnosti* je možnost aktualizovat stav systému v rozumném čase. Příkladem mohou být aktualizace virové báze antivirových programů. Jiným příkladem může být instalace bezpečnostních aktualizací. Tuto vlastnost nelze podpořit zcela přímo, je však důsledkem ostatních vlastností (zejména *rozšiřitelnost* a *modularita*).

Modularita *Modularita* znamená, že výsledný systém sestává z dobře definovaných a nezávislých komponent. To vede k lepší *udržovatelnosti*. Komponenty mohou být v takovém případě vyvíjeny a testovány izolovaně předtím než jsou integrovány do výsledného požadovaného softwarového systému. Navíc získáme dobrou možnost dělby práce v rámci vývojového týmu pracujícího na softwarovém projektu.

Znovupoužitelnost Je důležité, aby bylo možné přidávat nové vlastnosti (features) systému a provádět modifikace s pouze omezenou nebo žádnou modifikací existujících komponent (možnost použít komponenty znovu pro jiné případy užití).

3.1.3 Objektově orientovaný návrh software

Základním vstupem pro objektově orientovaný návrh bývá (mimo jiné) *konceptuální model*, případně seznam *případů užití* systému. To jsou výsledky předešlé fáze *analýzy*. Na základě těchto informací je potom zpravidla sestavován návrh systému, který pracuje nad dříve definovanými doménovými objekty.

Klíčovými pojmy při objektově orientovaném návrhu jsou [38]:

Objekt/třída Objekt/třída představují těsné propojení mezi datovými strukturami a metodami/funkcemi, které nad těmito daty pracují. Každý objekt by měl sloužit samostatné funkci. Je definován svými atributy/vlastnostmi, tím co je a tím co může dělat.

Zapouzdření Zapouzdření je schopnost chránit některé součásti objektu před externími entitami. To je zpravidla realizováno klíčovými slovy implementujícího programovacího jazyka, která umožňují deklarovat proměnnou jako privátní nebo chráněnou uvnitř vlastnické třídy.

Dědičnost Dědičnost umožňuje nové třídě rozšířit nebo i přepsat funkcionalitu jiné třídy. Výsledná podtřída má kompletní část, která je derivována z nadtřídy a navíc své vlastní funkce a svá data.

Rozhraní Rozhraní představuje možnost odložení implementace metod. Je možné definovat signatury funkcí nebo metod, aniž by bylo nutné tyto metody implementovat.

Polymorfismus Polymorfismus nám umožňuje nahradit objekt jeho podobjektem (instancí podtřídy). Proměnná tedy může obsahovat buď objekt nebo kterýkoliv jeho podobjekt.

Při vlastním návrhu systému se používají principy objektového návrhu (popsáno dále v sekci 3.2) a návrhové vzory. Realizace objektových systému se provádí často za pomocí vhodných frameworků a knihoven.

3.2 Analýza principů objektového návrhu

Principy objektového návrhu reprezentují množinu pokynů, které nám pomáhají vyhnout se špatnému návrhu [12]. Špatný návrh vykazuje zejména následující vlastnosti [29]:

- *rigidity* – je obtížné provést změnu v systému protože každá změna ovlivňuje příliš mnoho dalších částí systému,
- *fragility* – pokud je provedena změna, mohou přestat fungovat části systému u nichž se to nečeká,
- *immobility* – je obtížné znovupoužití části systému v jiné aplikaci, protože tato část nemůže být „vymotána“ z aktuální aplikace.

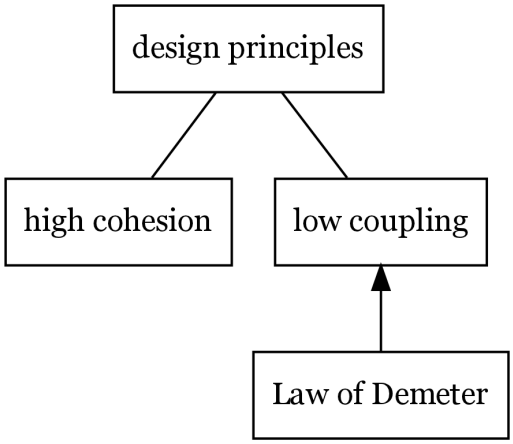
Autor výše zmíněného článku [29] cituje zajímavé návrhové principy. Například *Open Closed Principle*, který vyžaduje, aby se existující třídy a řešení nemodifikovaly, ale rozšiřovaly. Dalším principem je *Liskov's substitution principle*, který vymezuje pravidla pro dědění tak, aby instance podtřídy mohla bezpečně vystupovat na místě své nadtřídy[2].

Důležité je srovnání *návrhových principů* s *návrhovými vzory*. Toto srovnání podává článek [31]. Zatímco aplikace návrhového vzoru představuje znovupoužití „úspěšného" návrhu, princip představuje spíše obecně platné pravidlo, jehož dodržováním během návrhu bychom měli získat návrh přijatelné kvality. V případě přesného stanovení potřebných principů by bylo možné provádět automatizovaný systematický návrh (což není, vzhledem k mnoha aspektům popisovaným v článku, dobře možné).

3.2.1 Analyzované principy

Na základě požadavků specifikovaných sekci 2.1 se budeme postupně zabývat principy *low coupling*, *high cohesion* a *Law of Demeter*. Jedná se o principy, které se týkají strukturální kvality kódu.

Ukázkové návrhové principy a vztahy mezi nimi jsou znázorněny na obrázku 3.1. Poznamenejme, že *Law of Demeter* (dále budeme používat zkratku *LoD*) je konkretizací požadavků na *nízkou závislost* (*low coupling*) mezi moduly.



Obrázek 3.1: Znázornění vztahů mezi analyzovanými návrhovými principy.

3.2.1.1 Low coupling/dependency (nízká závislost/vazba)

Závislost/vazba (coupling/dependency) mezi moduly softwarového projektu určuje, do jaké míry se jeden modul spoléhá na každý z ostatních modulů [34]. Zatímco některé moduly spolu vůbec nekomunikují (nemají žádnou závislost), jiné se spoléhají nejen na rozhraní ostatních modulů, ale mnohdy i na jejich vnitřní fungování, způsob reprezentace dat, časování atd.

[2] Ve článku je rozebráno, jaké operace bychom neměli provádět v děděné třídě – je totiž možné změnit chování oddělené třídy tak, že již nemůže bezpečně vystupovat na místě svého předka.

Důsledkem vyšší závislosti je potom nutnost rozsáhlých úprav v mnoha modulech i při úpravě naprostých drobností v jednom konkrétním modulu.

Proto je důležitou návrhovou zásadou snaha snížit závislost modulů na minimum. Je zřejmé, že závislost mezi moduly vždy existuje (jinak by neměl modulární návrh smysl). Proto nelze striktně říci, do jaké míry smí/nesmí být moduly na sobě závislé.

V [34] a [25] (podrobnější a formálnější definice) je podáván přehled úrovní závislosti jednoho modulu na druhém. Následující zjednodušený seznam úrovní závislostí je seřazen od nejvyšší formy závislosti po nejnižší:

Content coupling (nejvyšší forma závislosti) Jeden modul modifikuje jiný modul nebo se spoléhá na vnitřní fungování jiného modulu (např. přístup k lokálním datům jiného modulu). V důsledku platí, že změní-li se způsob, kterým tento druhý modul produkuje data (umístění, typ, časování), povede to zcela jistě ke změnám v závislém modulu.

Common coupling Dva moduly sdílí stejná globální data (např. globální proměnnou), změna sdíleného globálního zdroje implikuje změny všech modulů, které je používají.

External coupling Dva moduly sdílí externě definovaný (standardizovaný) datový formát, komunikační protokol nebo rozhraní zařízení.

Control coupling Jeden modul kontroluje tok druhého tím, že mu posílá informaci o tom, co má konat (např. předání „to-do“ příznaku).

Stamp data coupling Jeden modul předává druhému modulu složenou datovou strukturu jako parametr. Ten ji používá pouze pro výpočty (nikoliv pro rozhodování řízení toku programu).

Scalar data coupling Moduly sdílí data pomocí parametrů. Každý parametr je elementární datový typ a jedná se o jediná data, která jsou sdílená (např. předávání celočíselné hodnoty funkci, která spočítá jeho druhou mocninu). Modul, kterému jsou předávány parametry, jich používá pouze k výpočtu hodnoty a nikoliv pro rozhodování řízení toku programu.

Message coupling (nejnižší forma závislosti) Moduly jsou komunikují pouze pomocí posílání zpráv (message passing). Jedná se o nejnižší úroveň závislosti. Moduly o sobě navzájem nemusí mít žádnou znalost. Moduly nepoužívají vzájemně žádné předávání parametrů, nemají žádné sdílené reference na objekty nebo globální data.

Independent coupling/No coupling Mezi moduly není žádná závislost. Moduly spolu vůbec nekomunikují a nejsou zde žádné sdílené reference na proměnné nebo reference na externí data sdílená mezi moduly.

Subclass coupling Pro objektově orientovaný návrh můžeme navíc uvažovat závislost typu *subclass coupling*, která popisuje vztah mezi třídou a její rodičovskou třídou. Dětská třída má závislost na rodičovské, rodičovská však žádnou závislost vzhledem k dětské třídě nemá.

Cílem návrhu je snižovat co nejvíce míru závislosti modulů. Protože se ale jedná o kvantitativní záležitost, nejsme schopni určit zcela přesná pravidla, která musí platit nebo která lze vynutit. U některých typů programů může být akceptovatelný i *content coupling* z výkonnostních důvodů[3], naopak u jiných systémů může být nízká vazba (*message coupling*) dána již návrhem (např. CORBA, web services, atd.).

3.2.1.2 High cohesion (vysoká koheze/soudržnost)

Pojmem *koheze* je označována míra, jak silně související/příbuzná je funkcionalita vyjádřená modulem programu [33]. Existují různé přístupy k měření této metriky, od kvalitativních, vyjadřujících se pomocí slovních ohodnocení, po kvantitativní, vyjadřující míru soudržnosti kódu pomocí čísel.

Moduly s vysokou kohezí jsou zpravidla preferovány, protože jejich vlastnosti implikují „dobré“ vlastnosti návrhu včetně stability, spolehlivosti, znovupoužitelnosti a srozumitelnosti, zatímco moduly s nízkou kohezí je obtížné udržovat a testovat.

Ke snížení koheze dochází když:

- funkcionalita vyjádřená třídou, ke které přistupujeme pomocí metod, nemá mnoho společného,
- metody provádějí velké množství rozdílných aktivit často nad rozsáhlými nebo nesouvisejícími daty.

Nevýhodou nízké koheze je:

- obtížnější porozumění modulům,
- obtížnější údržba systému – logické změny v řešené doméně ovlivňují více modulů a také proto, že změny v jednom modulu vyžadují změny v modulu jiném,
- obtížnější znovupoužití modulu – většina aplikací nepotřebuje náhodnou množinu operací, kterou modul poskytuje.

Kohezi můžeme uvažovat na různých úrovních. Jako úroveň můžeme zvolit např. *metodu*, *třídu*, *balíček*, *modul* nebo *projekt*. Program může mít vysokou kohezi na úrovni tříd (metody jsou dobře seskupené ve třídách podle funkcionality a pracují nad shodnými daty), zatímco na úrovni balíčků může vykazovat nižší kohezi (třídy v rámci balíčku spolu logicky nesouvisí).

Problematikou soudržnosti programových modulů se zabývá článek [27], který představuje dvě různé metriky používané k jejímu měření.

[3] Nízká vazba implikuje téměř vždy snížení výkonu z důvodu nutnosti dalších mechanismů, které zprostředkovávají komunikaci mezi moduly (např. *message passing* mechanismus).

Článek [26] uvádí dělení koheze podle tzv. SMC[4] Cohesion. Míra koheze je vyjádřena na základě typu asociace mezi procesními elementy (seřazeno od nejnižší úrovně koheze po nejvyšší):

Coincidental association Neexistuje souvislost mezi elementy provádějícími zpracování.

Logical association Oba elementy provádějící zpracování patří do stejné logické třídy příbuzných funkcí.

Temporal association Každý výskyt obou elementů provádějících zpracování je v tom samém omezeném časovém období při provádění programu.

Procedural association Oba elementy provádějící zpracování jsou elementy stejné procedurální jednotky, která je iterativním nebo rozhodovacím procesem.

Communicational association Oba elementy provádějící zpracování pracují nad stejnou množinou vstupních dat a/nebo produkují stejná výstupní data.

Sequential association Výstupní data jednoho elementu jsou vstupními daty pro druhý element.

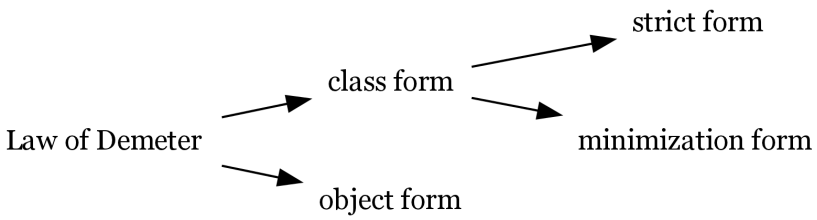
Functional association Oba elementy jsou nezbytné pro provedení jedné funkce/operace.

3.2.1.3 Law of Demeter

Jak bylo zmíněno v sekci 3.2.1, lze na *LoD* pohlížet jako na speciální případ principu pro zachování nízké vazby. *LoD* je také někdy nazýván jako princip nejmenší znalosti (Principle of Least Knowledge) [37]. Určuje, že každá jednotka by měla mít pouze omezenou znalost ostatních jednotek. Měla by komunikovat pouze s omezeným počtem jednotek úzce souvisejících s logikou této jednotky. Zde se nám částečně prolínají principy *low coupling* a *high cohesion*. Zatímco princip *low coupling* požaduje, aby modul nezávisel na příliš detailních znalostech o implementaci druhého modulu, *high cohesion* vyžaduje, aby úzce souvisela logika modulu s logikou závislých modulů (aby se nevolaly vzdálené logicky nesouvisející části kódu).

Princip *Law of Demeter* je velmi dobře popsán v [28]. Článek pojednává o třídě jako o „dodavateli funkcionality“. *LoD* potom specifikuje, kteří dodavatelé funkcionality jsou pro konkrétní třídu preferovaní (tzv. „preferred suppliers“). To nám vymezuje množinu tříd/objektů, na něž se smí vyskytnout reference v rámci definované třídy. V článku je rozebráno více forem principu, které jsou vhodné pro různé způsoby aplikace. Klasifikace těchto forem je znázorněna na obrázku 3.2.

[4] Podle původních autorů Stevens, Myers a Constantine.



Obrázek 3.2: Formy návrhového principu *LoD*.

Kromě striktní formy, která vyžaduje, aby všechny třídy dodržovaly pravidla daná principem *LoD*, existuje ještě minimalizační verze, která se snaží minimalizovat počet porušení pravidla pod přijatelnou mez[5].

Pro statickou analýzu je vhodné použití *class formy* pravidla *LoD*. *Class forma* a *objektová forma* se lehce liší v tom, které třídy jsou preferovanými dodavateli funkcionality. Pro praktické použití (udržovatelnost kódu, modularita, atd.) to však nemá až takový význam.

Uveďme nyní *striktní třídní formu* pravidla *Law of Demeter*:

Návrhový princip. *Ve všech metodách M třídy C je povoleno používat pouze členy (metody a data) následujících tříd a jejich nadtříd [28]:*

- *C,*
- *třídy členských datových polí třídy C,*
- *třídy argumentů M,*
- *třídy, jejichž konstruktor je volán v M,*
- *třídy globálních proměnných použitých v M.*

Tento princip budeme implementovat jako ukázkové pravidlo vyvíjeného systému. Díky tomu, že je definováno na třídách a nikoliv na objektech je možné jej použít přímo nad vhodně předzpracovanými zdrojovými kódy v programovacím jazyce Java.

3.2.2 Ukázky kódu porušujícího některá z pravidel

Abychom demonstrovali, jak vypadá kód, který není vyhovující z hlediska návrhu, pojďme se podívat na ukázky porušení návrhových principů. U principu *LoD* můžeme uvažovat konkrétní porušení. Pro *low coupling* a *high cohesion* se můžeme podívat na rozdíly mezi *úrovněmi vazby* a *koheze*.

[5] Například umožní porušení principu *LoD* v privátním API, ale bude vyžadovat dodržení ve veřejně přístupných API.

3.2.2.1 Porušení principu law of Demeter

Podívejme se na nejtypičtější ukázku porušení *LoD*, která je uváděna v [21]. Zde autor demonstruje porušení principu *LoD* a současně modeluje reálnou situaci. Je patrné, že dodržení *LoD* vede k přirozenějšímu mapování reálných objektů na objekty v programovacím jazyce.

První částí ukázky je jednoduchá datová třída modelující zákazníka:

```
public class Customer {

    private String firstName;
    private String lastName;
    private Wallet myWallet;

    public String getFirstName(){
        return firstName;
    }

    public String getLastName(){
        return lastName;
    }

    public Wallet getWallet(){
        return myWallet;
    }
}
```

Třída `Wallet` má následující definici:

```
public class Wallet {

    private float value;

    public float getTotalMoney() {
        return value;
    }

    public void setTotalMoney(float newValue) {
        value = newValue;
    }

    public void addMoney(float deposit) {
        value += deposit;
    }

    public void subtractMoney(float debit) {
        value -= debit;
    }
}
```

Kód pro zaplacení prodavači novin potom vypadá takto:

```
// code from some method inside the Paperboy class
payment = 2.0; // I want my two dollars!

Wallet theWallet = myCustomer.getWallet();
if (theWallet.getTotalMoney() > payment) {
    theWallet.subtractMoney(payment);
} else {
    // come back later and get my money
}
```

Autor výše uvedené ukázky dále rozebírá, že je chybné předpokládat, že si od nás vezme pouliční prodavač novin naši peněženku, vezme si z ní patřičný obnos a peněženku nám opět vrátí. Celá situace je vyřešena přidáním nové třídy reprezentující platbu a speciální metodou `getPayment` třídy `Customer`. Zákazník se tak o svou peněženku stará sám a prodavači pouze předá požadovaný obnos.

Ukázka nevyhoví definované class verzi principu *LoD*. Kód který se stará o získání platby podle všeho dostává referenci na zákazníka. Na základě této reference získá referenci na objekt třídy `Wallet`. A právě objekt třídy `Wallet` nespadá ani do jedné z kategorií povolených dodavatelů funkcionality (není třídou třídní proměnné, ani třídou předávanou v parametrech, není ve třídě konstruován a není ani globální proměnnou).

3.2.2.2 Ukázka kódu s příliš vysokou úrovní závislosti

Nejhorší variantou závislosti je závislost na konkrétním obsahu jiného modulu. Typickým příkladem může být přímý přístup k proměnným konkrétní třídy.

```
class A {
    public long pieceOfData;

    long getNumber() {
      return pieceOfData;
    }
}

class B {

    long computeSquare(A a) {
        return a.pieceOfData * a.pieceOfData;
    }

}
```

Tento kus kódu je špatný jednak z hlediska zapouzdření (proměnná `pieceOfData` má být definována jako `protected` nebo `private`) a dále také z důvodu přílišné závislosti třídy **B** na třídě **A**. V případě, že by se nám nelíbil název nebo způsob uložení proměnné `pieceOfData`,

její změnou dojde k nutnosti úpravy ve třídě **B**. Jedná se o nejvyšší formu závislosti – *content coupling*.

Podíváme-li se na jinou úroveň závislostí, můžeme se setkat s kódem podobným následujícímu:

```
class A {

    int do(int what) {
        if (what == 1) {
            // do some action
            return 0;
        } else {
            // do another action
            return 1;
        }
    }
}

class B {

    void launchAction(A a) {
        int b = a.do(1);
    }
}
```

To je typická ukázka *control coupling*. Třída **B** využívá služeb třídy **A**. Metoda `do` je řízena vstupní proměnnou předávanou z metody třídy **B**. To má tu nevýhodu, že v případě změny způsobu, jakým pracuje metoda `do` (např. přidání dalšího stavu řídící proměnné) může být nutné provést reimplementaci metody třídy **B**, která nepočítá s jinou návratovou hodnotou nebo jiným průběhem zpracování.

3.2.2.3 Ukázka kódu s nízkou úrovní koheze

Nejnižší úrovní koheze je *coincidental cohesion*. Typicky se jedná o různé balíky funkcionality označované jako `Utils`, `Helper` a podobně. Funkce v takovém balíčku většinou nemají téměř žádnou příbuznost. Často se též jedná o velké bloky funkcionality, které provádějí celou řadu nesouvisejících věcí, ale zůstávají v jednom velkém bloku jednoduše proto, že se jich programátoři „bojí dotknout“.

Příkladem úrovně *logical cohesion* může být následující třída:

```
class ErrorPublisher {

    void printErrorToTerminal(String message) {
        // implementation
    }

    void printErrorToFile(String message, File file) {
```

```
        // implementation
    }

    void logErrorToSyslog(String message) {
        // implementation
    }
}
```

Funkčnosti jsou seskupeny do jedné třídy jen z toho důvodu, že všechny někam vypisují chybu. Každá implementace však bude zcela odlišná (metody nesdílejí žádný stav).

3.3 Analýza vstupních projektů realizovaných v jazyce Java

V rámci této sekce se podíváme nejprve na to, z čeho se skládá softwarový systém realizovaný v programovacím jazyce Java. Ve druhé části je provedena rešerše existujících nástrojů pro zpracování zdrojových kódů programovacího jazyka Java do podoby vhodné pro vygenerování potřebné vnitřní struktury, která bude předmětem validace.

3.3.1 Statický model programu v Javě

Vstupem pro validační systém budou zdrojové kódy existujícího Java projektu ve vhodné podobě. Jedná se vlastně o fyzický model programu. V případě Java projektu sestává tento model z nějaké množiny souborů, které obsahují data různých typů. První podsekce této sekce se zabývá typy souborů, které se mohou vyskytnout v projektu, ve druhé podsekci je potom rozebírán obsah a formát souborů programovacího jazyka Java.

Zastavme se ještě nad pojmem *fyzický model* programu. Tento model je zpravidla výsledkem vhodné transformace (ať už automatizované nebo ruční) nějakého předešlého modelu (většinou UML model, textová specifikace atd.). S tímto modelem můžeme dále pracovat a provádět nad ním transformace. To je ostatně cílem různých nástrojů pro refaktoring a automatizované zvyšování kvality kódu. Konkrétně refaktoring je příkladem *endogenní horizontální transformace* (zdrojový i cílový meta-model je shodný a úroveň abstrakce zůstává nezměněna) [30].

3.3.1.1 Struktura softwarového projektu v Javě

Existuje velké množství způsobů, kterými jsou organizovány softwarové projekty. Téměř každé Java IDE má navíc svůj vlastní formát, ve kterém ukládá metadata o projektu. Přesto všechny tyto projekty zahrnují stejnou podmnožinu konkrétních zdrojových souborů, z nichž se skládá vlastní softwarový projekt. Podívejme se, jaké soubory můžeme nalézt v běžném Java projektu:

- soubory `*.java` – soubory se zdrojovými kódy v jazyce Java, tyto soubory představují z hlediska gramatiky jazyka Java kořenový element *CompilationUnit* (o něm bude dále pojednáno v podsekci 3.3.1.2),

- binární součásti projektu – soubory `*.jar` a `*.class`, často se jedná o různé knihovny, zkompilované `*.java` soubory atd.,
- build scripty a konfigurační soubory sestavovacích nástrojů (`build.xml`, `pom.xml`),
- soubory zdrojů (resources, resource bundles) – read-only data využívaná programem (ikony, lokalizační řetězce, atd.),
- dokumentace (`javadoc`, uživatelská příručka),
- soubory gramatik pro parsery a *compiler-compiler* systémy (např. pro `lex`, `flex`, `javacc`, `antlr`),
- šablony (typické třeba pro webové projekty, `*.xhtml` a jiné přípony),
- konfigurační soubory (zpravidla `*.xml` soubory, často odkazují konkrétní třídy programovacího jazyka Java),
- jiné soubory.

Pro potřeby této práce jsou podstatné v zásadě pouze soubory obsahující zdrojový kód – soubory `*.java`. Při analýze nebudeme program spouštět nebo uvažovat jeho dynamické chování (běh). Bude se jednat o analýzu statického stavu programu (analýza struktury). Práce bude probíhat nad definicemi tříd, nikoliv nad jejich instancemi v paměti JVM. Nejsou tedy uvažovány všechny možné běhové instance programu (všechny možné stavy objektů v paměti virtuálního stroje).

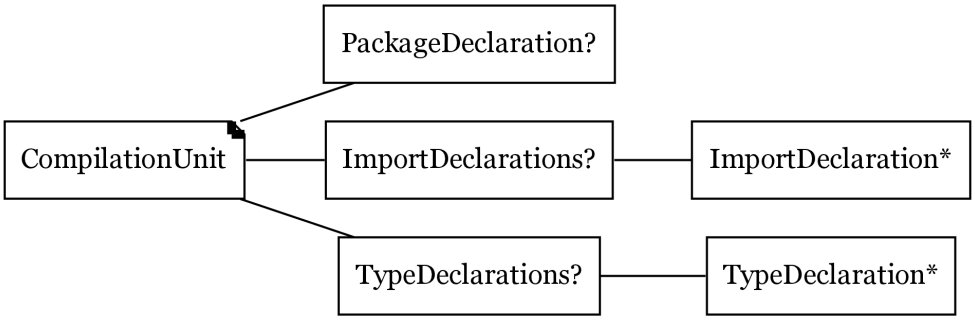
Projekt může záviset na velkém množství dalších tříd, které nejsou součástí zdrojových kódů projektu. Tyto třídy zpracovávat nebudeme. Jedná se například o:

- knihovny třetích stran,
- standardní knihovnu jazyka Java (i.e. Java 2 Platform SE 5.0 API pro Javu verze 5),
- podprojekty a části projektu, které analyzovat nechceme, nepotřebujeme nebo z nějakého důvodu nemůžeme (např. projekty, které nejsou pod naší kontrolou a nemůžeme u nich návrh opravit, atd).

Některé z těchto tříd budou povolenými závislostmi pro všechny třídy projektu (např. všechny třídy projektu mohou záviset na standardních knihovnách jazyka Java), jiné budou naopak povolené jen pro určité oblasti projektu (budeme chtít, aby rozhraní pro přístup k databázi využívaly pouze objekty DAO vrstvy).

3.3.1.2 Syntaktické elementy programovacího jazyka Java

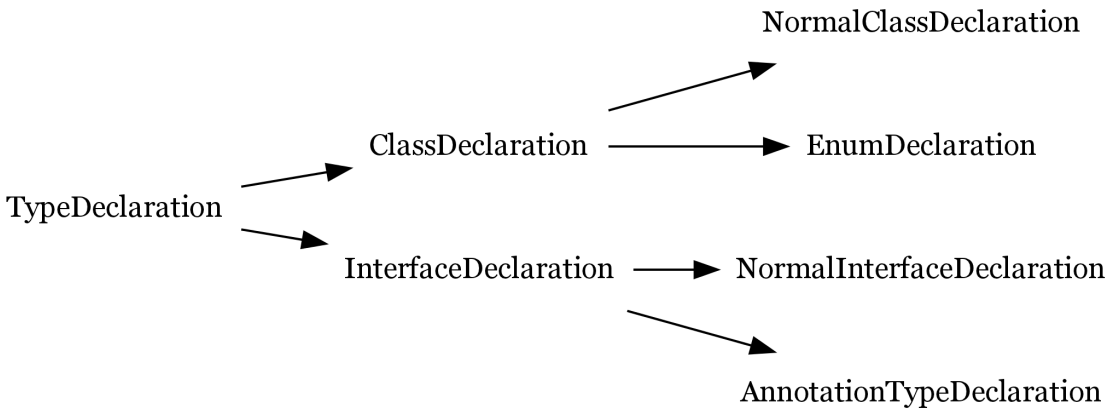
Jak již bylo zmíněno v předchozí sekci, základním kořenovým elementem gramatiky programovacího jazyka Java je *CompilationUnit*. Kompletní popis všech elementů jazyka poskytuje specifikace jazyka Java [24].



Obrázek 3.3: Struktura základních syntaktických elementů programovacího jazyka Java.

Element *CompilationUnit* obsahuje volitelně *deklaraci balíčku*, *příkazy importu* prvků z jiných jmenných prostorů a *deklarace datových typů* (obrázek 3.3). Ty jsou pro prováděnou validaci a analýzu nejdůležitější.

Každá deklarace datového typuje představována neterminálním symbolem *TypeDeclaration* gramatiky jazyka Java. Tento neterminální symbol se dále přepisuje na symboly uvedené na obrázku 3.4. Výsledný deklarovaný datový typ může být jeden z následujících elementů: *deklarace třídy*, *deklarace výčtového typu*, *deklarace rozhraní* a *deklarace anotace*.



Obrázek 3.4: Rozklad elementu TypeDeclaration.

Datové typy budou výchozími body při sestavování vnitřní reprezentace pro provádění validace a analýzy. Jejich další pod-elementy jsou tvořeny členskými proměnnými, konstruktory, metodami, abstraktními metodami (signatury metod v rozhraních) a vnořenými datovými typy. Dále se elementy dělí až na úroveň jednotlivých příkazů, konstrukcí objektů, výrazů atd.

3.3.2 Zpracovávání zdrojových kódů v jazyce Java

Protože navrhované ověřované principy přesahují rámec jedné kompilační jednotky, je nutné provést vhodné předzpracování zdrojových kódů tak, aby bylo možné analyzovat vztahy částí

kódu i mezi jednotlivými kompilačními jednotkami.

V principu to znamená provést podobnou množinu operací, jakou provádí první kompilátor jazyka Java při kompilaci [11]. Ten nejprve načte všechny `*.java` soubory a namapuje sekvence tokenů na uzly AST stromu. Následně vloží všechny nalezené symboly do tabulky symbolů (jména datových typů, proměnných). Provede zpracování anotací nalezených v kompilačních jednotkách. Ve fázi *attribute* jsou provedeny operace jako rozlišení jmen, kontrola datových typů a zjednodušení konstantních výrazů. Následuje kontrola datových toků v programu (pracuje na stromu získaném v předchozích krocích), která má za cíl určit např. nedostupné části zdrojového kódu (po příkazu `return` apod.). Předposlední fáze – *desugar* – přepíše AST a zjednoduší některé výrazy používané jako „syntaktický cukr“ na jednodušší výrazy. Na závěr provede kompilátor vlastní vygenerování zdrojových kódů nebo class souborů.

Pro provedení analýzy nám postačí zdrojové kódy zpracované do fáze *attribute*. Jedná se o AST s rozlišenými jmény datových typů a proměnných. Nad těmito stromy můžeme dále pracovat.

Podívejme se nyní, jakými prostředky je možné tuto reprezentaci získat:

3.3.2.1 Vlastní kód pro zpracovávání zdrojových kódů

Samozřejmě je možné napsat pro zpracování zdrojových kódů vlastní nástroj. Zahrnovalo by to vytvoření lexikálního analyzátoru (tokenizer), syntaktického parseru (zásobníkový automat) a dalších mechanismů pro vybudování AST, rozlišení jmen, navázání proměnných a realizaci velkého množství dalších operací.

Jedná se však o velmi náročný a rozsáhlý systém, jehož rozsah by překračoval nejen rámec této práce, ale i možnosti jednotlivce realizovat takovou práci v rozumném čase.

3.3.2.2 Použití compiler-compiler systému

Pro zpracování zdrojových kódů a různě formátovaného textu se často používá tzv *compiler-compiler* systémů. Tyto systémy vygenerují jak lexikální analyzátor, tak syntaktický parser na základě vhodného popisu. Zpravidla se jedná o popis ve formě BNF nebo EBNF gramatiky. Často je možné na základě specifikovaných pravidel vygenerovat i AST vhodný pro další zpracovávání.

Nejčastěji používanými nástroji jsou:

JavaCC *JavaCC* [16] je systém, pro který existuje velké množství gramatik. Mimo jiné také gramatika pro jazyk Java 1.5. Součástí distribuce tohoto nástroje je i program *JJTree*, který je schopen na základě rozšířeného textového popisu gramatiky vygenerovat AST.

ANTLR *ANTLR* [13] je velmi pokročilý *compiler-compiler* systém. Disponuje kompletním vývojovým prostředím, kterému nechybí vlastnosti jako zvýrazňování syntaxe, doplňování výrazů a podpora lazení (umožňuje krokovat generování syntaktického i AST stromu).

Pro většinu *compiler-compiler* nástrojů existuje velké množství vstupních gramatik pro různé jazyky. Většinou nalezneme i podporu pro jazyk Java verze 1.5[6].

Při použití některého z výše uvedených nástrojů bychom se dostali na úroveň vygenerovaného AST stromu. Pro potřeby analýzy je však nutné provést ještě rozlišení jmen. To by bylo nutné provést ručně. Jedná se stále o dost náročnou operaci.

3.3.2.3 Použití vhodné knihovny nebo existujícího programu

Další možností jsou existující knihovny a nástroje pro zpracování kódů v jazyce Java.

Jako příklad můžeme uvést knihovnu *JavaParser* [17]. Jedná se o samostatný Java projekt, který je možné zaintegrovat prostřednictvím nástroje Maven do vlastního projektu. Při bližším prozkoumání zjistíme, že se jedná vlastně o aplikaci nástroje JavaCC (resp. JJTree) distribuovaného spolu s gramatikou jazyka Java 1.5.

Jiným nástrojem je *spoon* [19]. Je to volně dostupný nástroj, preprocesor kódu v jazyce Java. Poskytuje úplný a detailní meta-model jazyka Java, kde každý element může být jak čten tak i modifikován. Další významnou vlastností je jeho integrovanost do platformy Eclipse formou pluginu.

Zatímco nástroj *JavaParser* neprovádí rezoluci jmen, nástroj *spoon* vnitřně deleguje parsování a attribution na `javac` kompiler. Proto se v dalších sekcích zaměříme na možnost využití některého z nástrojů, které podobným způsobem poskytují zpracování až do požadované fáze.

3.3.2.4 Použití prostředků poskytovaných platformou Java

Jazyk Java od verze 6 poskytuje zajímavé možnosti integrace volání kompilátoru jazyka Java do kódu uživatelských programů [43]. Jedná se o následující aplikační rozhraní:

- *JSR 199 – Java Compiler API* [1]
  - specifikuje způsoby volání překladače jazyka Java pomocí API ze zdrojového kódu programu
  - balíček `javax.tools`
- *JSR 269 – Pluggable Annotation Processing API*
  - možnost přidání vlastního kódu pro zpracovávání anotací (a zčásti i kódu) do instance překladače
  - balíček `javax.annotation.processing` – zpracovávání anotací
  - balíček `javax.lang.model` – třídy poskytující model pro syntaktické elementy jazyka Java
- *Compiler Tree API* [14]
  - nestandardní rozšíření Java JDK (Sun verze)

[6] Což by pro naše účely postačovalo, protože Java 6 se od verze 5 neliší v gramatice jazyka, ale v poskytovaných rozhraních standardní knihovny.

- balíček `com.sun.source.tree` – poskytuje rozhraní pro reprezentaci zdrojového kódu jako AST
- balíček `com.sun.source.util` – poskytuje rozhraní pro operace nad AST

Pomocí těchto nástrojů je možné získat poměrně snadno přístup k průběhu zpracování zdrojového kódu pomocí `javac` kompilátoru. Přístup však je na úrovni zpracovávání anotací (viz výše) a probíhá po částech (rounds). Nemáme tak k dispozici plný AST strom, nad kterým by byla již provedena fáze *attribute* [11].

3.3.2.5 Použití prostředků současných vývojových IDE

Většina majoritních integrovaných prostředí pro vývoj v jazyce Java (IntelliJ Idea, Eclipse, NetBeans) provádí rozsáhlou kontrolu zdrojových kódů přímo během psaní. Programátor je upozorňován na to, že některá metoda nebo třída není definována nebo je definována s jinými parametry. A to v případě, že daná třída se nachází ve zcela jiné kompilační jednotce.

To je ale přesně to, co potřebujeme pro naši práci. Proto byla provedena rešerše v této oblasti. Byly nalezeny dva základní nástroje, které je možné použít. Jedná se o platformu Eclipse a platformu NetBeans.

Platforma Eclipse poskytuje rozhraní, které reprezentuje Java projekt jako DOM prostřednictvím balíčku `org.eclipse.jdt.core.dom` [15]. Poskytovaný AST je možné procházet i modifikovat.

Podobnou podporou poskytuje i platforma NetBeans. Ta vnitřně používá instanci `javac` kompilátoru. Přístup k AST v libovolné fázi (před resolvováním jmen i po této fázi) poskytuje prostřednictvím svého *Java Source API* [18].

Výhodou použití některé z existujících platforem je navíc možnost přímé integrace výsledného nástroje do vývojového prostředí. Nevýhodou je potom svázanost s tímto prostředím. To je však možné zčásti vyřešit vhodným návrhem jádra systému, které může být realizováno tak, aby bylo na platformě zcela nezávislé a závislost na platformě byla přidávána až integrací do konkrétního prostředí.

Kapitola 4

Návrh

V kapitole *návrh* postupně projdeme jednotlivé součásti výsledného systému. Nejprve se budeme zabývat formalizací způsobu specifikace pravidel správného návrhu software.

Část druhá je věnována rozboru navrhované architektury systému. Systém je postupně dekomponován shora dolů. Hlavní funkční bloky jsou postupně dekomponovány na komponenty, jejichž realizace je triviální. Je navrhováno rozdělení systému na jádro (core) a moduly (extensions), které budou poskytovat konkrétní funkcionalitu.

Další části se věnují návrhu vstupního rozhraní, výstupního rozhraní a způsobu integrace jádra do různých prostředí.

Poslední část je věnována návrhu konkrétních technologií, v nichž bude výsledný systém realizován.

4.1 Návrh způsobu specifikace pravidel

Abychom mohli specifikovat pravidla, musíme nejprve definovat strukturu, nad kterou budou tato pravidla platit. V tomto textu formalizujeme model programu pomocí teorie grafů. Nad vytvořeným modelem potom definujeme způsob konstrukce pravidel, která musí vstupní projekt splňovat.

4.1.1 Formalizace modelu programu pomocí grafu

Analyzovaný softwarový projekt abstrahujeme jako orientovaný multigraf rozšířený o zobrazení množiny uzlů do množiny typů[1] a zobrazení hran do množiny jejich klasifikátorů (označení typu vztahu mezi uzly). Dále přidáme ke každému vrcholu zobrazení, které mu přiřadí jméno (řetězec). Získáme tak následující definici pojmu *grafový model programu*:

Definice. ***Grafovým modelem projektu*** *nazveme strukturu*

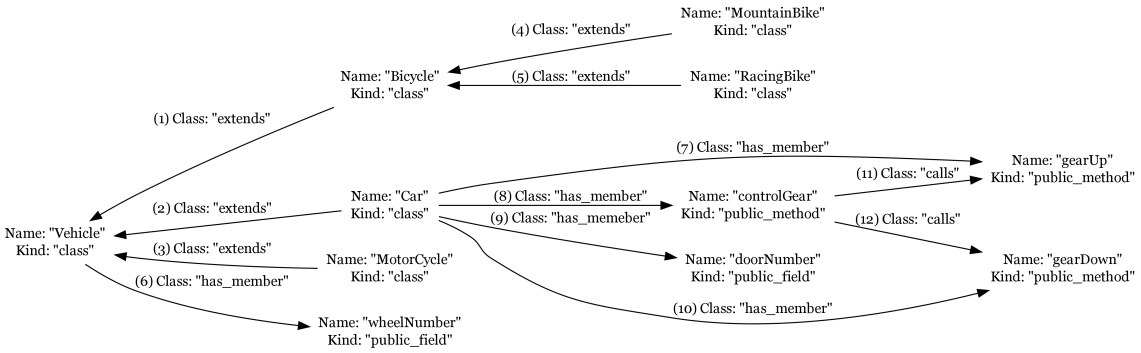
$$G = \langle V, E, \rho, K, C, N, Kind, Classifier, Name \rangle$$

v níž platí:

[1] Pod pojmem typ zde rozumíme jakékoliv označení, které specifikuje o jaký objekt se jedná, nikoliv datový typ. Může se jednat o vrchol typu třída, metoda, příkaz nebo třeba celý zdrojový soubor, budeme-li analyzovat vztahy mezi kompilačními jednotkami.

- *V je množina elementů (v našem případě části kódu),*
- *E je množina hran (v našem případě vztahy mezi částmi kódu – např. volání funkce, dědičnost),*
- *$V \cap E = \emptyset$,*
- *$\rho : E \mapsto V \times V$ je zobrazení množiny hran do množiny uspořádaných dvojic vrcholů (incidence),*
- *K je libovolná množina označení typů vrcholů,*
- *C je množina klasifikátorů hran,*
- *N je množina jmen (řetězců),*
- *$Kind : V \mapsto K$ je zobrazení, které přiřadí každému vrcholu jeho typ,*
- *$Classifier : E \mapsto C$ je zobrazení, které přiřadí každé hraně její klasifikátor (zda se jedná o vyvolání metody, dědičnost, apod.),*
- *$Name : V \mapsto N$ je zobrazení, které přiřadí vrcholu jeho jméno (např. jméno třídy, jméno metody).*

Ukázka formalizace zdrojového kódu pomocí grafu je na obrázku 4.1. V uvedeném příkladě můžeme strukturu G namapovat následujícím způsobem:



Obrázek 4.1: Příklad formalizace hierarchie tříd pomocí grafu.

Množinu vrcholů můžeme ztotožnit s množinou názvů elementů (v našem případě třídy, pole, metody)[2].

$$
\begin{aligned}
V \ = \{ & \\
& \text{“}Vehicle\text{”}, \text{“}Car\text{”}, \text{“}MotorCycle\text{”}, \text{“}Bicycle\text{”}, \text{“}MountainBike\text{”}, \text{“}RacingBike\text{”}, \\
& \text{“}wheelNumber\text{”}, \text{“}doorNumber\text{”}, \\
& \text{“}controlGear\text{”}, \text{“}gearUp\text{”}, \text{“}gearDown\text{”} \\
& \}
\end{aligned}
$$

[2] V konečném důsledku bude tato množina představována konkrétními elementy tak, jak se nalézají ve zdrojovém kódu.

Abychom mohli pracovat s množinou hran, bylo provedeno jejich očíslování. Konkrétní hranu zde tedy identifikujeme číslem. V počítačové reprezentaci se může jednat o konkrétní objekt (resp. referenci na něj) uložený v poli. Zvolený identifikátor hrany nemá vliv na prováděnou analýzu, musí pouze zajistit jednoznačnou identifikaci hrany. Množinu hran reprezentujeme prostým výčtem čísel[3]:

$$E = \{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12\}$$

Zobrazení ρ, které představuje přiřazení jednotlivých hran dvojicím uzlů, můžeme formálně zapsat jako následující množinu[4]:

$$\begin{aligned}\rho = \{ & \\ & (1, (\text{“}Bicycle\text{”}, \text{“}Vehicle\text{”})), \\ & (2, (\text{“}Car\text{”}, \text{“}Vehicle\text{”})), \\ & (3, (\text{“}MotorCycle\text{”}, \text{“}Vehicle\text{”})), \\ & (4, (\text{“}MountainBike\text{”}, \text{“}Bicycle\text{”})), \\ & (5, (\text{“}RacingBike\text{”}, \text{“}Bicycle\text{”})), \\ & (6, (\text{“}Vehicle\text{”}, \text{“}wheelNumber\text{”})), \\ & (7, (\text{“}Car\text{”}, \text{“}gearUp\text{”})), \\ & (8, (\text{“}Car\text{”}, \text{“}controlGear\text{”})), \\ & (9, (\text{“}Car\text{”}, \text{“}doorNumber\text{”})), \\ & (10, (\text{“}Car\text{”}, \text{“}gearDown\text{”})), \\ & (11, (\text{“}controlGear\text{”}, \text{“}gearUp\text{”})), \\ & (12, (\text{“}controlGear\text{”}, \text{“}gearDown\text{”})) \\ & \}\end{aligned}$$

Zbývají nám množina typů K,

$$K = \{\text{“}class\text{”}, \text{“}public\_field\text{”}, \text{“}public\_method\text{”}\}$$

množina klasifikátorů hran C,

$$C = \{\text{“}extends\text{”}, \text{“}has\_member\text{”}, \text{“}calls\text{”}\}$$

[3] V programu bude potom hrana reprezentována konkrétním objektem případně rozšířeným o vhodný jedinečný identifikátor.

[4] Zobrazení je binární relace. Proto jej můžeme reprezentovat jako množinu uspořádaných dvojic.

přiřazení typů uzlům,

$$
\begin{aligned}
Kind = \{ & \\
& (\text{“}Bicycle\text{”}, \text{“}class\text{”}), \\
& (\text{“}Car\text{”}, \text{“}class\text{”}), \\
& (\text{“}MotorCycle\text{”}, \text{“}class\text{”}), \\
& (\text{“}MountainBike\text{”}, \text{“}class\text{”}), \\
& (\text{“}RacingBike\text{”}, \text{“}class\text{”}), \\
& (\text{“}Vehicle\text{”}, \text{“}class\text{”}), \\
& (\text{“}wheelNumber\text{”}, \text{“}public\_field\text{”}), \\
& (\text{“}doorNumber\text{”}, \text{“}public\_field\text{”}), \\
& (\text{“}controlGear\text{”}, \text{“}public\_method\text{”}), \\
& (\text{“}gearUp\text{”}, \text{“}public\_method\text{”}), \\
& (\text{“}gearDown\text{”}, \text{“}public\_method\text{”}) \\
& \}
\end{aligned}
$$

a přiřazení klasifikátorů hranám:

$$
\begin{aligned}
Classifier = \{ & \\
& (1, \text{“}extends\text{”}), \\
& (2, \text{“}extends\text{”}), \\
& (3, \text{“}extends\text{”}), \\
& (4, \text{“}extends\text{”}), \\
& (5, \text{“}extends\text{”}), \\
& (6, \text{“}has\_member\text{”}), \\
& (7, \text{“}has\_member\text{”}), \\
& (8, \text{“}has\_member\text{”}), \\
& (9, \text{“}has\_member\text{”}), \\
& (10, \text{“}has\_member\text{”}), \\
& (11, \text{“}calls\text{”}), \\
& (12, \text{“}calls\text{”}), \\
& \}
\end{aligned}
$$

Uvedené množiny můžeme poměrně snadno reprezentovat v programovacím jazyce Java jako objekty. V dalším návrhu bude zavedena třída *Vertex*, hrana *Edge* a další objekty pro reprezentaci výše definovaných pojmů.

Vrcholy takto vytvořeného grafu nemusí být pouze třídy, můžeme použít libovolné syntaktické elementy z kódu, který analyzujeme. Vždy záleží na úrovni prováděné analýzy, co zvolíme jako vrcholy grafu, jaké hrany mezi nimi zvolíme a jaká označení a typy budou mít. Nástroj, který budeme navrhovat bude počítat s libovolným takto vybudovaným grafem. Ke grafům konkrétního typu přiřadíme identifikátory a při specifikaci pravidel vždy uvedeme, nad kterým typem grafu má definované pravidlo platit.

Grafový model projektu je definován velmi obecně, protože nespecifikuje, jaké klasifikátory hran se mohou vyskytnout mezi kterými typy vrcholů. Proto budeme uvažovat pojem *typ grafového modelu projektu*. Ten bude udávat, jaké klasifikátory mohou mít hrany, které spojují vrcholy konkrétních typů. Pojem zde nebudeme definovat, pouze naznačme, jak bychom postupovali. Je nutné přidat dodatečné zobrazení, které přiřadí každé dvojici vrcholů dvojici jejich typů. Potom je možné tyto dvojice zobrazit do množiny klasifikátorů. Toto zobrazení nám potom vymezí, jaké typy hran se mohou vyskytnout mezi vrcholy konkrétních typů.

4.1.2 Formalizace pravidel

Nyní, když máme definován model, můžeme popsat jazyk pravidel, která mají pro daný model platit. Pokusíme se pravidla zavést co nejformálněji. Budeme uvažovat základní matematické operace (kvantifikace, funkce, ...).

Pravidla budeme definovat nad množinami V a E ve výše zmíněné definici *grafového modelu projektu*. Můžeme tak hovořit o vrcholových a hranových pravidlech. O jaké pravidlo se bude jednat a nad kterými vrcholy nebo hranami bude platit vymezíme pomocí kvantifikace. Např.

$$\exists v \in V$$

určí, že pravidlo musí platit alespoň pro jeden vrchol. Elementy, nad nimiž budeme chtít specifikovat pravidlo můžeme navíc vymezit pomocí podmínky, která má platit pro všechny vybírané prvky. To může být třeba požadavek, aby všechny vybírané prvky měly konkrétní hodnotu zobrazení $Kind$:

$$\exists v \in V : HasKind(v, \text{“}function\_name\text{”})$$

Element typu *HasKind* označíme pro další použití v textu jako *predikát*. Bude se jednat o funkci, jejíž návratová hodnota má logický výsledek *platí/neplatí*. Dále můžeme nad vybranými elementy specifikovat vhodné pravidlo. To budeme provádět pomocí dalších operátorů. Uveďme příklad:

$$\forall v \in V : HasKind(v, \text{“}function\_name\text{”})$$

$$CountLessThan(OutgoingEdges(v, \text{“}argument\text{”}), 4)$$

V uvedeném případě bychom chtěli vyjádřit požadavek, aby počet argumentů každé (všimněme si změněného kvantifikátoru) funkce byl menší než čtyři. Sémantika výrazu ovšem závisí na tom, jak zadefinujeme použité operátory (resp. funkce) a současně na tom, jaký význam má hrana s klasifikátorem “*argument*” (které elementy spojuje nebo může spojovat). Je zřejmé, že základní specifikace pravidla by měla mít následující součásti:

- označení typu grafu, nad nímž má pravidlo platit,

- jméno pravidla (to sice není podstatné pro definici pravidla, v dalším textu však umožníme definici složených pravidel, pro která bude zavedení identifikace pravidel nezbytné),
- vymezení množiny, nad kterou má pravidlo platit (kvantifikace, podmínka výběru prvků),
- logický výraz skládající se z predikátů.

Výraz uvedený ve druhé části pravidla (v našem případě $CountLessThan(...)$) může být jakýkoliv logický výraz, jehož elementy jsou predikáty platící nad nějakými objekty. Mezi predikáty budeme uvažovat *úplný systém logických spojek*. Ten nám pokryje množina logických spojek $\{\wedge, \vee, \neg\}$[5]. Logické spojky budou mít standardní prioritu vyhodnocování. Navíc umožníme vynutit jinou prioritu pomocí závorek. Výsledkem výrazu bude logická hodnota, která nám bude udávat splnění resp. nesplnění požadovaného pravidla. Celou konstrukci (vymezení typu grafu, pojmenování pravidla, kvantifikace a logický výraz) označíme pojmem *atomické pravidlo*.

Abychom umožnili větší flexibilitu při definování pravidel. Můžeme atomická pravidla dále použít pro definici pravidel *složených*. Jediné, co k tomu potřebujeme, je úplný systém logických spojek tak, jak byl zmíněn výše. Složené pravidlo potom bude vypadat takto:

$$P1 \wedge A1 \vee (A2 \vee \neg P3)$$

kde všechna použitá pravidla musí být před použitím definována. Pravidla, znichž se bude složené pravidlo skládat, mohou být jak atomická tak složená. Díky tomu můžeme vytvořit na základě dříve definovaných primitiv složitější požadavky na platnost pravidel.

V uvedených výrazech používáme různé funkce/operátory. Typy těchto funkcí omezíme na následující:

- *predikát (predicate)* – operátor, který na základě vstupních parametrů a poskytované rozhodovací funkce vrací booleovskou hodnotu (true/false),
- *vrcholový selektor (vertex selector)* – vrací množinu vrcholů na základě vstupních parametrů a poskytované výběrové funkce,
- *hranový selector (edge selector)* – vrací množinu hran na základě vstupních parametrů a poskytované výběrové funkce.

Pro výše zmíněné operátory umožníme libovolný počet parametrů. Pro potřeby této práce však vymezíme typy elementů, které mohou být parametry. Budeme pracovat s následujícími typy:

- vrchol,
- hrana,

[5] Jeden z operátorů $\wedge$ a $\vee$ je ve zmíněném systému logických spojek nadbytečný, ale pro pohodlnost použití jej ponecháme.

- množina vrcholů,
- množina hran,
- celočíselná hodnota,
- řetězec (pro reprezentaci klasifikátorů a typů),
- logická hodnota `true`/`false`.

Ve výše uvedeném příkladu (*LessThan*(...)) má predikát *LessThan* dva parametry, kde první je typu *množina hran* a druhý typu *celé číslo*. Na místě prvního operátoru vystupuje operátor typu *hranový selektor*, jehož návratovou hodnotou je *množina hran*. Parametry tohoto selektoru jsou potom element typu *vrchol* a *řetězec*.

Protože výše zmíněná formalizace pravidel není zcela vhodná pro zadávání do počítače, poskytneme vhodnou formu serializace této matematické notace do vhodného formátu. Jedná se o jednoduché mapování, které nezmění podstatu navrženého formalismu. Formálně je způsob zadávání pravidel popsán dále pomocí gramatiky specifikované v příloze B.3. Gramatika je popsána ve vstupním formátu nástroje ANTLR (zmiňovaného v sekci 3.3.2). Protože se však jedná o pouze drobně modifikovanou formu EBNF, používáme ji současně jako referenční příručku pro zápis nových pravidel. Jazyk pro definici pravidel, který tímto způsobem vznikl budeme označovat zkratkou AVD (**A**rch**V**al[6] **D**efinitions) a soubory pravidel generované zmíněnou gramatikou budeme dále nazývat jako AVD soubory.

4.1.3 Analytický přístup k návrhovým principům

Ne všechny aspekty návrhu je možné zachytit pomocí konkrétně vymezitelných požadavků. Mnohdy můžeme kvalitu návrhu posuzovat pouze kvalitativně a statisticky (dobrý návrh, lepší, moc vazeb, málo vazeb, atd.). Pro některé návrhové principy (např. low coupling[7], high cohesion) může být vhodnější poskytnout statistický pohled (testování statistických hypotéz, použití vhodného klasifikátoru).

Kromě exaktního zadávání a vyhodnocování pravidel, která mají platit nad konkrétním grafem tak můžeme uvažovat ještě nejrůznější druhy analýzy. Můžeme použít např.

- poznatky z teorie grafů,
- vyhledávání vzorů,
- vyhodnocování statistických pravidel (meze počtů elementů v grafech, atd.), generování dat pro vykreslení histogramů,
- použití vytěžování dat (klasifikace modelů na základě vhodné trénovací množiny, shlukování).

[6] Označení *ArchVal* je použité kodové jméno projektu, které je zavedeno dále.
[7] *LoD* je již konkretizací principu *low coupling*, u které lze uvažovat hodnoty *splněno*/*nesplněno*.

Zajímavou možností je právě využití klasifikátorů. Každý graf má velké množství zajímavých statistických znaků (features), které je možné využít. Můžeme vytvořit klasifikátor na základě existujících vzorků „dobrých“ a „špatných“ architektonických návrhů (přípravou vhodných trénovacích a testovacích příkladů lze ovlivňovat chování klasifikátoru pro různé účely použití) a ten následně použít pro ověření návrhu zcela neznámých projektů.

4.2 Návrh architektury systému

Při návrhu architektury systému pro vyhodnocování pravidel v navrženém formalismu postupujeme metodou shora dolů. Nejprve budeme systém uvažovat jako jeden velký celek s globální zodpovědností (funkcionalitou). Tento velký blok následně dekomponujeme na jednotlivé komponenty, kterým přidělíme jasně definované zodpovědnosti.

Pro výsledný systém budeme používat kódové jméno *ArchVal* (**Arch**itecture **Val**idator), zkracované často v názvech modulů jako `av`.

Globální pohled na systém byl zaveden již v části 2.3 na obrázku 2.1. Toto znázornění je velmi obecné. Vidíme však, že systém má dva hlavní vstupy[8] a jeden výstup. Vstupy a výstupy představují podstatnou část doménových dat nad nimiž budeme dále pracovat. V části 4.2.2 popíšeme jednotlivé datové objekty, které postupně „protékají“ celým systémem. Ty budou popsány a dekomponovány jako první, protože jejich popis je nezbytný pro definici ostatních komponent systému.

Ve druhé fázi budeme dekomponovat jádro systému. Zodpovědnost jádra představuje zodpovědnost celého systému (zodpovědností celého systému je „provést validaci projektu na základě množiny pravidel“). Tuto zodpovědnost dále rozdrobíme na základní systémové komponenty (jádro) a množinu rozšíření, která bude možné postupně přidávat.

Definujeme veřejná rozhraní (SPI) [44], která budou implementována poskytovateli služeb. Každému z těchto rozhraní bude věnována jedna podsekce.

4.2.1 Globální struktura systému

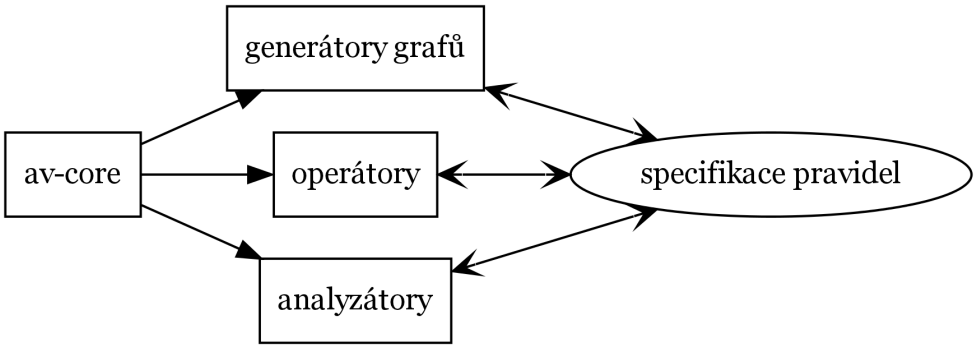
Prvním stupněm dekompozice systému je jeho rozdělení na části, které budou *neměnné* a na *rozšíření*, která bude možné přidávat a odebírat a která tak umožní běh systému v různých konfiguracích.

Základním kamenem celého systému bude *jádro*, které bude integrovat dohromady všechna rozšíření. Jádro bude obsahovat pouze základní logiku umožňující vyhodnocování pravidel a provádění analýz poskytovaných prostřednictvím komponent rozšíření.

Znázornění základních bodů rozšíření jádra je na obrázku 4.2. Element *specifikace pravidel* představuje *soubor definic* obsahující pravidla ve formátu definovaném v sekci 4.1.2 a další definice určující výsledný průběh validace.

Vzhledem k formalismu navrženému v sekci 4.1.1 je nutné nejprve vhodným způsobem zpracovat vstup. K tomu budou sloužit *generátory grafů*. Jejich zodpovědností bude vygenerování určitého typu grafu z AST analyzovaného programovacího jazyka.

[8] Jako vstupy/výstupy neuvažujeme další běžné součásti systému jako např. načítání konfiguračních souborů nebo logování.



Obrázek 4.2: Body rozšíření systému.

Rozšíření typu *operátor* představuje jeden konkrétní operátor (selektor nebo predikát), který je možné použít v rámci *souboru definic*. Každý operátor bude deklarovat své *jméno*, *počet* a *typy operandů* a *návratový typ*.

Protože ne všechny kontroly zdrojového kódu lze zapsat jako pravidla s výsledkem `true` nebo `false`, bude možné do systému vložit vlastní *analýzu*, jejímž vstupem bude graf konkrétního typu. Nad tímto grafem bude analýza vyhodnocena a výstup ve vhodném vnitřním formátu bude vrácen jako výsledek.

Na základě provedené dekompozice definujeme seznam rozhraní, která budou muset být implementována poskytovateli služeb (service providers):

- *GraphGeneratorIface* – rozhraní poskytovatele generátoru grafu,
- *OperatorIface* – rozhraní poskytovatele operátorů (selektorů a predikátů),
- *AnalysisIface* – rozhraní poskytovatele analýzy (analytického modulu).

Každé z těchto rozšíření může být implementování více různými poskytovateli. Pouze je nutné zajistit, aby nekolidovaly názvy poskytovaných služeb (jména operátorů, typů generátorů grafů a názvy analýz).

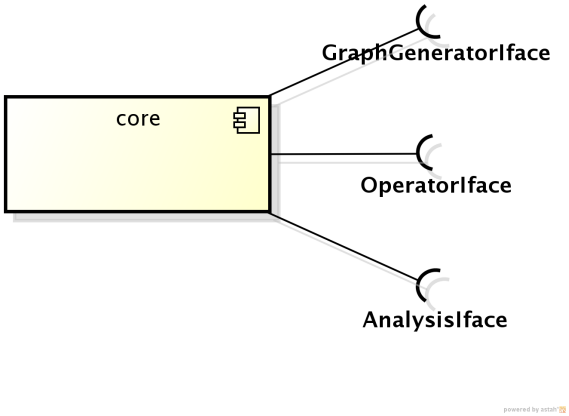
Celá dekompozice provedená v tomto stupni je zachycena v diagramu komponent na obrázku 4.3. Modul jádra nazveme `av-core`. Jeho dekompozice bude provedena v dalších sekcích. Ještě předtím se ale podíváme na důležité doménové objekty, které budou reprezentovat data předávaná mezi jednotlivými komponentami v rámci systému.

4.2.2 Doménové objekty

Přehled doménových objektů podává tabulka 4.1. Protože jsou tyto objekty součástí jádra a změna jejich rozhraní by nutně vedla k úpravě ostatních komponent, navrhujeme tyto datové typy jako konkrétní třídy[9].

Graph, *Vertex* a *Edge* jsou elementárními datovými třídami představujícími vygenerovaný graf jeho vrcholy a hrany. V této práci navrhujeme tyto elementy jako třídy. Je možné

[9] U ostatních komponent předpokládáme, že budou implementovány třetími stranami. Je třeba pro ně třeba přesně definovat veřejná rozhraní SPI.



Obrázek 4.3: Dekompozice systému na jádro a rozšíření.

Tabulka 4.1: Tabulka doménových objektů.

Název	**Typ**	**Zodpovědnost**
Graph	class	graf reprezentující analyzované elementy
Vertex	class	vrchol grafu
Edge	class	hrana grafu
GraphModel	class	množina grafů různých typů
ValidationModel	class	model validace načtený ze vstupního *souboru definic*
ValidationReport	class	objekt reprezentující výstup validace

tyto elementy zadefinovat též jako rozhraní. Tím bychom získali možnost analyzovat zcela libovolné elementy (např. existující objektovou hierarchii). Důležitou vlastností třídy *Graph* bude deklarace jejího typu, který si musí nést s sebou. Jednotlivé instance grafů různých typů seskupíme do jedné entity *GraphModel*, kterou budeme dále v systému předávat jako celek. To bude množina[10] grafů různých typů.

Třída *ValidationModel* představuje klíčovou komponentu nesoucí veškeré informace potřebné pro provedení validačního procesu. Bude se jednat o strom pravidel vygenerovaný na základě souboru specifikace pravidel, který bude mít schopnost se samostatně vyhodnotit na základě předané vstupní instance typu *GraphModel*. Výsledkem vyhodnocení bude potom instance třídy *ValidationReport*, která bude poskytovat informace o průběhu zpracování.

Navrhované třídy *ValidationModel* a *GraphModel* je možné vygenerovat a používat opakovaně (cache). To se může hodit v případech, kdy chceme tentýž projekt (pro nějž jsme již jednou generovali graf) validovat pomocí jiné/upravené množiny pravidel nebo naopak pokud chceme použít jednu množinu pravidel pro validaci více projektů. Bylo by zbytečné opakovaně provádět náročné operace generování grafu nebo parsování souboru pravidel a navazování operátorů.

[10] Každý typ grafu může být zahrnut právě jednou.

Některé z těchto tříd rozebereme nyní detailněji:

4.2.2.1 Třída ValidationModel

Validační model (třída *ValidationModel*) sestává z atomických pravidel, složených pravidel a seznamu požadovaných poskytovatelů analýzy. Pravidla je nutné načíst z existujícího souboru specifikací a sestavit z nich strom vhodný pro vyhodnocení. Přesná definice vstupního souboru je popsána v příloze B.3. Zahrnuje specifikaci atomických pravidel, složených pravidel a příkazů, které umožní vybrat pouze konkrétní podmnožinu pravidel, která se mají ověřit a příkazů, které umožní provedení vybraných analýz.

To vše musí model vhodným způsobem reflektovat. Na základě vstupních elementů vygenerujeme AST (nejspíš pomocí vhodného nástroje), který následně převedeme na strom, který se bude skládat z jednotlivých operátorů a vstupních parametrů, který bude možné vyhodnotit nad existující instancí třídy *GraphModel*.

Třídu *ValidationModel* dekomponujeme dále na třídy *AtomicRule* a *CompoundRule* (které budou mít společného předka *Rule*, abychom mohli k těmto pravidlům při vyhodnocování přistupovat stejně. Tyto třídy budou dále obsahovat již konkrétní syntaktický strom uzlů, které budou mít metodu *evaluate()*, která vždy zajistí vyhodnocení příslušného uzlu a návrat výsledné hodnoty. Díky tomu bude vyhodnocení celého stromu pravidel (z hlediska uživatele modelu) triviální – provedeme zavolání metody *validate()* na kořenovém elementu komponenty *ValidationModel*. Ta následně zavolá *evaluate()* metody na svých potomcích a takto to bude probíhat až k listovým elementům.

Pro vybudování zmíněného stromového modelu vytvoříme samostatnou komponentu *ValidationModelGenerator*, jejíž rozhraní bude definováno dále.

4.2.2.2 Třída ValidationReport

Výstupem validačního procesu bude objekt třídy *ValidationReport*. Ačkoliv matematická pravidla popsaná výše mají jako výstup pouze logickou hodnotu, je vhodné, aby bylo možné nějakým způsobem zpětně zjistit, v jaké fázi vyhodnocování došlo k nesplnění pravidla. Proto poskytneme právě datovou třídu *ValidationReport*. Ta ponese nejen elementární výsledky pravidel, ale také mezivýsledky vyhodnocení jednotlivých uzlů syntaktického stromu popsaného dříve.

Strom výsledků dekomponujeme podobným způsobem, jakým jsme dekomponovali třídu *ValidationModel*. Poskytneme třídy *AtomicRuleResult* a *CompoundRuleResult*. Tyto třídy opět podědíme od společného předka, abychom je mohli uchovávat v rámci jednoho seznamu výsledků.

Kromě toho přidáme ještě třídu *AnalysisResult*, která ponese informaci o konkrétní prováděné analýze. Pro třídu *AnalysisResult* strom generovat nebudeme, protože neznáme přesný způsob práce konkrétního analytického modulu. Třída *AnalysisResult* bude obsahovat seznam tvrzení o množinách vrcholů nebo hran. Každý záznam bude obsahovat vždy množinu vrcholů nebo hran a k ní řetězec, který bude reprezentovat výrok, který o příslušné množině platí. Způsob provádění analýz a zpracovávání výsledků je oblast, kterou je možné řešit samostatně v rámci dalších návazností této práce.

Tabulka 4.2: Tabulka komponent jádra systému.

Název	**Typ**	**Zodpovědnost**
GraphModelGenerator	class	provede vygenerování instance třídy *GraphModel* obsahující množinu grafů požadovaných typů
GraphModelGeneratorIface	interface	rozhraní předchozí komponenty
ValidationModelGenerator	class	generátor, který na základě validační specifikace vygeneruje instanci třídy *ValidationModel*
ValidationModelGeneratorIface	interface	rozhraní předchozí komponenty
ValidationTask	class	komponenta zabalující hlavní proces validace
ValidationTaskIface	interface	rozhraní předchozí komponenty
ArchVal	class	fasáda *ArchVal* systému – bude poskytovat instance výše specifikovaných komponent
GraphGeneratorsRegisterIface	interface	rozhraní registru existujících poskytovatelů generátorů grafu
OperatorsRegisterIface	interface	rozhraní registru existujících poskytovatelů operátorů
AnalysesRegisterIface	interface	rozhraní registru existujících poskytovatelů komponent analýzy

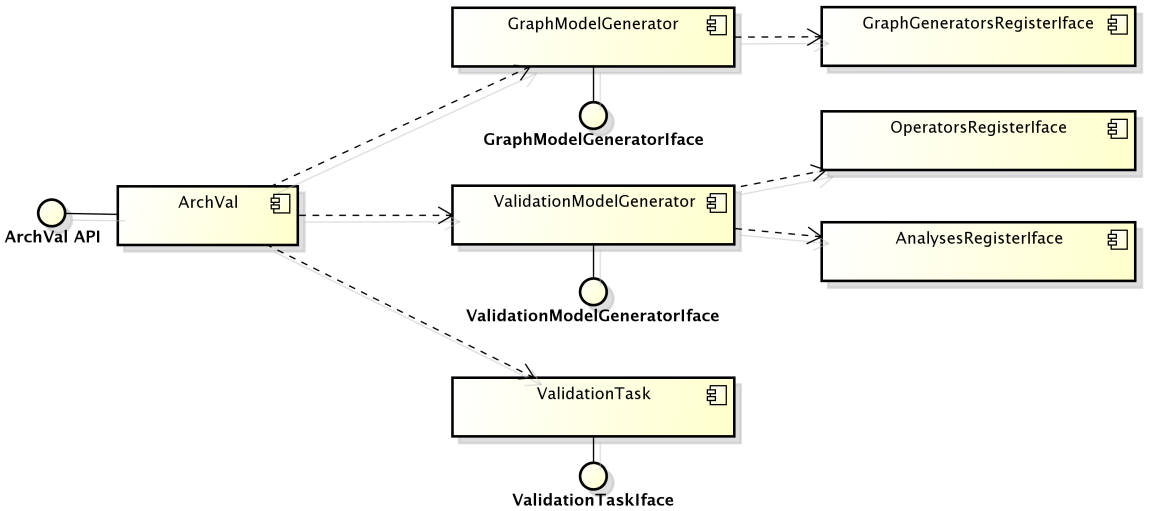
Generování objektů třídy *ValidationReport* bude součástí průběhu zpracování validace. Metoda *validate()* třídy *ValidationModel* provede vytvoření objektu *ValidationReport* a příslušných podobjektů (*AtomicRuleResult*, atd.). Tyto objekty následně předá prostřednictvím *evaluate()* metod do dalších vyhodnocovacích uzlů. Ty provedou totéž o úroveň níže. Každý uzel po provedení příslušné operace uloží do předaného objektu svou návratovou hodnotu. Díky tomu získáme strom obsahující data umožňující rekonstruovat průběh zpracování.

4.2.3 Jádro systému

Jádro systému samotné nebude poskytovat žádná konkrétní pravidla pro validaci ani žádné analýzy. Bude se jednat o platformu, která umožní přidávání nových komponent, které toto budou provádět na základě textového popisu pravidel. Součástí jádra budou komponenty nezbytné pro zpracování vstupního souboru definic do podoby vhodné k provedení (vytvoření modelů, navázání poskytovaných operátorů). Přehled navrhovaných komponent a rozhraní jádra uvádíme v tabulce v tabulce 4.2.

Vztahy mezi jednotlivými komponentami jádra systému jsou znázorněny na obrázku 4.4. Komponenta *ArchVal* bude zastřešovat přístup k funkcionalitám jádra. Díky tomu budeme moci komponenty *GraphModelGenerator*, *ValidationModelGenerator* a *ValidationTask* skrýt za vhodná rozhraní a v případě potřeby je reimplementovat. Všimněme si komponent *GraphGeneratorsRegisterIface*, *OperatorsRegisterIface* a *AnalysesRegisterIface*. Ty představují roz-

hraní samostatných komponent, které nám budou schopny poskytnout poskytovatele implementace existujících rozšíření (generátory grafů, operátory a analýzy). Díky této abstrakci bude možné jádro systému zakomponovat do jakékoliv platformy bez nutnosti uvažovat mechanismy pro registraci poskytovatelů služeb[11].



Obrázek 4.4: Komponenty jádra systému *ArchVal*.

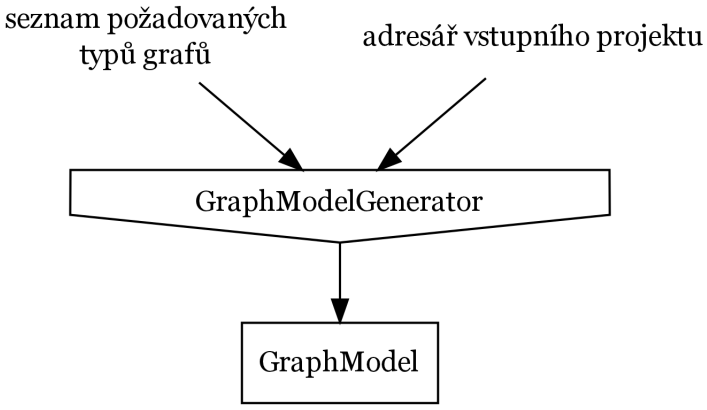
4.2.3.1 Komponenta GraphModelGenerator

Komponenta *GraphModelGenerator* bude mít na zodpovědnosti vygenerování všech grafů potřebných pro provedení analýzy. Abychom mohli časem komponentu reimplementovat, skryjeme její implementaci za rozhraní *GraphModelGeneratorIface*. To bude vystupovat v kontaktu s „vnějšími“ entitami jádra (u klientského uživatele knihovny jádra *ArchVal*). Jak je patrné již z obrázku 4.4, tato komponenta má závislost na rozhraní *GraphGeneratorsRegisterIface*. Samotná komponenta *GraphModelGenerator* vlastně žádné grafy generovat přímo nebude. Bude k tomuto úkolu využívat některé z dostupných generátorů grafů dostupných prostřednictvím registru.

Vstupy a výstupy komponenty jsou znázorněny na obrázku 4.5. U vstupů se jedná o adresář obsahující vstupní Java soubory analyzovaného projektu a seznam identifikátorů typů grafů, které chceme vygenerovat. Výstup je potom představován instancí třídy *GraphModel*. Je možné, že komponenta *GraphModelGenerator* nebude mít k dispozici potřebné generátory grafů. V tom případě vyhodí výjimku.

Rozhraní Rozhraní této komponenty bude představováno jedinou metodou:

[11] Jak uvidíme dále v části *Implementace*, registrace poskytovatelů je vždy nějakým způsobem nutná a různé platformy ji podporují různým způsobem. Zmiňme *Lookup* na platformě NetBeans nebo nově přidanou vlastnost jazyka Java verze 6 *ServiceLoader*.



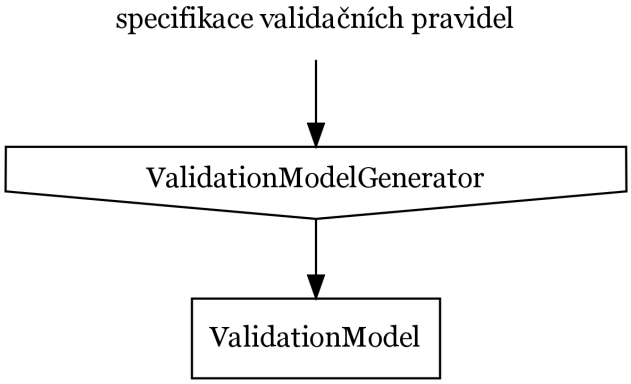
Obrázek 4.5: Znázornění vstupů a výstupů komponenty typu *GraphModelGenerator*.

- *generateModel(Set<String> requiredGraphTypes, File projectDirectory) : GraphModel*

Poznamenejme, že v rozhraních budeme uvádět datové typy jazyka Java, což není z hlediska návrhu zcela správné[12], nicméně pro potřeby této práce to bude postačující.

4.2.3.2 Komponenta ValidationModelGenerator

Druhou významnou vstupní komponentou je vedle generátoru modelu grafu komponenta *ValidationModelGenerator*, která bude sloužit k vygenerování instance dříve popsané doménové třídy *ValidationModel*. Tato komponenta má na zodpovědnosti zpracování vstupního souboru, který jí bude ve vhodné podobě předán a vytvoření stromové struktury třídy *ValidationModel*. Podobně jako předchozí komponentu i tuto zpřístupníme jako rozhraní, v tomto případě rozhraní *ValidationModelGeneratorIface*.



Obrázek 4.6: Znázornění vstupů a výstupů komponenty *ValidationModelGenerator*.

[12] Na úrovni návrhu bychom se měli snažit být maximálně nezávislí na platformě. Jistá závislost je však nutná vždy.

Znázornění vstupů a výstupů komponenty *ValidationModelGenerator* je na obrázku 4.6. Vstupem je množina pravidel zapsaná ve formátu AVD. Ta může být zadána buď jako vstupní *stream* nebo řetězec[13]. Výstupem kpomonenty je instance třídy *ValidationModel*.

Rozhraní Rozhraní komponenty je představováno metodami:

- *constructValidationModel(InputStream is) : ValidationModelIface*
- *constructValidationModel(String string) : ValidationModelIface*

4.2.3.3 Komponenta ValidationTask

Centrální komponentou umožňující spuštění validačního procesu bude *ValidationTask* (implementace skrytá za *ValidationTaskIface*). Tato komponenta bude mít na starosti provedení validačního procesu pomocí komponent *ValidationModel* a *GraphModel*. Implementace bude poměrně triviální, vzhledem k faktu, že veškerá logika potřebná k provedení validace bude zapouzdřena již v komponentě *ValidationModel*. Komponenta *ValidationTask* tak bude mít na starosti zajištění bezpečného spouštění procesu validace a to jak synchronním způsobem tak asynchronně (v samostatném vlákně).

Rozhraní Rozhraní komponenty *ValidationTask* bude představováno následujícími metodami:

- *runSynchronous() : void* – synchronní spuštění validace
- *runAsynchronous() : void* – běh validace v samostatném vlákně
- *registerValidationCompletedListener(ValidationCompletedListener validationCompletedListener) : void* – registrace listeneru, který bude notifikován o doběhnutí vlákna validace v případě jejího asynchronního spuštění
- *getReport() : ValidationReport* – vrátí instanci třídy *ValidationReport*

Výsledný objekt *ValidationReport* si volající může zpracovat libovolným způsobem. Například je možné ji reprezentovat ve vhodné swing komponentě (JTree).

4.2.3.4 Komponenta ArchVal (fasáda)

Předchozí komponenty zapouzdříme pomocí návrhového vzoru fasáda, za nímž skryjeme detaily implementace. Poskytneme tak službu pro získání hotových komponent pro generování modelů. Navíc můžeme tímto způsobem zajistit, aby pro každou instanci třídy *ArchVal* byly k dispozici třídy *ValidationModelGenerator* a *GraphModelGenerator* pouze v jedné instanci.

[13] Tato možnost zde byla přidána s ohledem na možnost přímé integrace, kdy by uživatel mohl zadávat pravidla do nějakého textového vstupu aplikace a tato pravidla by byla rovnou použita pro validaci projektu.

Rozhraní Rozhraní obsahuje následující metody:

- *ArchVal(GraphGeneratorsRegisterIface graphGeneratorsRegister, OperatorsRegisterIface operatorsRegister, AnalysesRegisterIface analysesRegister) : ArchVal* – konstruktor třídy
- *GraphModelGeneratorIface getGraphModelGenerator()*
- *ValidationModelGeneratorIface getValidationModelGenerator()*
- *ValidationTaskIface createValidationTask(GraphModel graphModel, ValidationModelIface validationModel) : GraphModel*

Metoda *createValidationTask()* představuje tovární metodu, která vrátí pokaždé novou instanci třídy *ValidationTask*[14].

4.2.3.5 Rozhraní registrů poskytovatelů implementací

Jak bylo zmíněno již v sekci 4.2.1, rozšíření budou získávána pomocí rozhraní *GraphGeneratorsRegisterIface*, *OperatorsRegisterIface* a *AnalysesRegisterIface*. Zde uveďme operace, které požadujeme od těchto rozhraní:

Rozhraní GraphGeneratorsRegisterIface Od registru generátorů grafů budeme požadovat, aby byl schopen vrátit seznam všech dostupných typů generátorů a dále konkrétní generátor na základě typu.

- *getAvaliableGeneratorTypes() : Set<String>*
- *getGraphGeneratorByType(String type) : GraphGeneratorIface*

Rozhraní OperatorsRegisterIface Zde požadujeme pouze zisk existujícího operátoru na základě jména. Pokud by bylo časem nutné podpořit editaci AVD souborů, zcela jistě by bylo potřeba získat i úplný seznam operátorů, případně implementovat vyhledávání i podle jiných kritérií (např. návratová hodnota).

- *getOperatorByName(String name) : OperatorIface*

Rozhraní AnalysesRegisterIface Zpravidla budeme potřebovat pouze metodu na zisk existujícího analytického modulu na základě jeho názvu. Přesto poskytujeme i metodu, která umožní získat seznam všech existujících analytických modulů.

- *getAnalysesList() : List<AnalysisIface>*
- *getAnalysisByName(String name) : AnalysisIface*

[14] Vzhledem k možnosti asynchronního provolávání metod třídy *ValidationTask* je požadováno, aby každá instance *ValidationTask* byla použita nejvýše jednou.

4.2.4 Rozhraní rozšíření systému

Nyní se podívejme na vlastní rozhraní, která musí být implementovaná poskytovali služeb poskytovaných prostřednictvím výše zmíněných registrů. Jedná se o rozhraní *GraphGeneratorIface*, *OperatorIface* a *AnalysisIface*.

4.2.4.1 Rozhraní GraphGeneratorIface

Úkolem implementátora generátoru grafu je zaimplementovat rozhraní *GraphGeneratorIface*. Toto rozhraní bude definovat následující operace:

- *getGraphType() : String* – typ grafu, který produkuje tento generátor grafu
- *getGraph(File projectDirectory) : Graph* – metoda, která provede vlastní vygenerování grafu

Vstupem pro metodu pro generování grafu je specifikace adresáře, který obsahuje projekt, z něhož má být vygenerován grafový model. Každý generátor musí vracet identifikátor typu grafu, který generuje. Systém odmítne spustit validaci, pokud bude obsahovat více poskytovatelů generátoru grafu, kteří poskytují generování grafu stejného typu.

4.2.4.2 Rozhraní OperatorIface

Soubory specifikací pravidel mohou obsahovat volání predikátů a selektorů. Pro každé takové volání musí v systému existovat odpovídající implementace operátoru. Všechny tyto operátory musí implementovat rozhraní *OperatorIface*. Rozhraní obsahuje metody, které jsou nezbytné pro vykonání operátoru. Jedná se o jméno operátoru (odpovídá jménu uvedenému v AVD souboru), počet a typ operandů a jeho návratovou hodnotu.

- *getName() : String* – název operátoru
- *getOperandsCount() : int* – počet operandů
- *getOperandType(int index) : DataType* – typy operátoru na pozici udávané indexem
- *getReturnType() : DataType* – návratový typ
- *execute(Graph graph, List<Object> operands) : Object* – metoda umožňující provedení operátoru se specifikovanými parametry

Datové typy používané u operátorů odpovídají datovým typům zmiňovaným dříve v podsekci 4.1.2 (množina vrcholů, množina hran, atd.). V jazyce java můžeme identifikátory těchto typů realizovat jako výčtový datový typ `enum`.

4.2.4.3 Rozhraní AnalysisIface

Rozhraní *AnalysisIface* reprezentuje jednu konkrétní analýzu, kterou je možné provést nad grafem konkrétního typu. To, nad jakým grafem lze analýzu spustit deklaruje analýza prostřednictvím metody *getRequiredGraphType() : String*. Podobně jako ostatní komponenty rozšíření musí i analýza specifikovat své jméno.

- *getAnalysisName() : String*
- *getRequiredGraphType() : String*
- *evaluate(Graph graph) : AnalysisResult*

Výsledkem provedené analýzy je instance třídy *AnalysisResult* zmiňovaná dříve v podsekci 4.2.2.2.

4.2.5 Základní průběh validačního procesu

Hlavním vstupním bodem pro provedení validačního procesu je komponenta *ValidationTask* popsaná dříve v podsekci 4.2.3.3.

Systém nejprve vygeneruje validační model pomocí komponenty *ValidationModelGenerator*. Na základě vygenerovaného modelu je potom možné získat seznam požadovaných typů grafů, které jsou nezbytné pro provedení validace a analýzy.

Seznam požadovaných typů grafů je následně předán komponentě *GraphModelGenerator*, která vygeneruje instanci třídy *GraphModel* (resp. *GraphModelIface*).

ValidationTask následně předá získanou instanci *GraphModelIface* validačnímu modelu (pomocí metody *validate()*).

Po provedení validace (ať už synchronním nebo asynchronním[15] způsobem) lze získat výsledky validace pomocí metody *getReport()* třídy *ValidationTask*. Výslednou instanci třídy *ValidationReport* je možné vhodným způsobem prezentovat uživateli nebo jinak využít.

4.3 Návrh vstupního rozhraní (zadávání pravidel)

V sekci 4.1.2 byl stanoven způsob specifikace validačních pravidel. To, jakým způsobem budeme pravidla do systému zadávat, závisí do značné míry na případech užití realizovaného nástroje. Jednou možností je přímá specifikace pravidel prostřednictvím nějakého textového pole v grafickém uživatelském prostředí, jinou možností je zadání cesty k existujícímu AVD souboru.

Pro navrhovaný systém použijeme kombinovanou možnost. Umožníme přímou editaci editaci AVD souboru v prostředí nástroje a bude-li to možné, poskytneme další formy podpory editace (zvýrazňování syntaxe). Při vlastním procesu validace však budeme načítat již pravidla z vybraného existujícího souboru.

[15] V případě asynchronního způsobu je nutné si zaregistrovat listener, který bude notifikován o doběhnutí vlákna validace.

4.4 Návrh výstupního rozhraní (provádění validace)

Vzhledem k navržené architektuře je definice způsobu výstupu ponechána jako oblast, kterou lze řešit poměrně nezávisle. Po provedení validační úlohy nám třída *ValidationTask* poskytne instanci třídy *ValiationReport*. Tento výstup je následně možné zpracovat libovolným způsobem.

Uveďme možnosti dalšího zpracování výstupního reportu:

- export v některém existujícím standardním formátu – html, pdf, tex, ad.,
- export prostého textového výstupu ve vlastním formátu,
- textový výstup v okně aplikace,
- reprezentace výstupu vhodnou swing komponentou (např. JTree komponenta).

Pro potřeby této práce postačí jednoduchý textový výstup, který pro realizovaná pravidla vypíše pouze zda pro daný vstup platí či nikoliv. Zpětné procházení stromu a vyhledávání příčin porušení konkrétního pravidla je ponecháno pro budoucí zpracování[16].

4.5 Návrh způsobu integrace do různých typů prostředí

Aplikační rozhraní systému *ArchVal* je od začátku navrhováno tak, aby bylo možné jej zaintegrovat do libovolné aplikace. Díky nezávislosti na konkrétním způsobu poskytování implementátorů služeb je možné integrovat výsledný vyhodnocovací systém například do:

CLI programů V případě integrace prostředí *ArchVal* do samostatných skriptů a programů pro příkazovou řádku by bylo nutné zaimplementovat nějaký způsob registrace poskytovatelů služeb. To ale mohou být prosté textové soubory s výčtem tříd, které se mají načíst. Případně mohou být seznamy operátorů definovány přímo v konkrétní implementaci registru operátoru, apod. U řádkových rozhraní je komplikovaná podpora editace AVD souborů.

GUI prostředí Systém může být integrován do existujících prostředí IDE. V této práci navrhujeme a realizujeme integraci systému do prostředí NetBeans IDE. Současně využíváme výhod existujícího parseru zdrojových kódů jazyka Java. Podobně bychom mohli využít platformu *Eclipse* nebo jiné prostředí. Na platformě NetBeans je možné implementovat registry poskytovatelů služeb velmi pohodlně pomocí tzv. *Lookup API*, které je schopné poskytnout existující implementace konkrétního rozhraní[17].

[16] Model byl záměrně navržen tak, aby toto podporoval – výsledkem validace bude strom výsledků pro jednotlivé uzly pravidla (jak bylo popsáno dříve v podsekci 4.2.2.2), který stačí vhodným způsobem projít.

[17] Stačí zaregistrovat poskytovatele služby v souboru *META-INF/services*.

4.6 Návrh technologií pro implementaci

Pro implementaci systému použijeme jazyk *Java 1.6* a platformu *NetBeans* verze *6.9.1*[18]. Pro zpracovávání zdrojových kódů využijeme prostředků poskytovaných touto platformou (*Java Source API*).

Abychom mohli předpokládat pevnou strukturu vstupních projektů, omezíme možné vstupní projekty na standardní *Maven Java projekty*.

[18] V době realizace práce majoritní verze.

Kapitola 5

Implementace

Implementační část práce se skládala z několika součástí. Za nejvýznamnější součást lze považovat formální specifikaci pravidla *Law of Demeter*, která je uváděna jako první v rámci této kapitoly. V dalších částech kapitoly se potom zabýváme vybranými poznámkami z implementace jádra systému *ArchVal*, který má sloužit jako platforma pro provádění matematických formulací návrhových pravidel a implementaci rozšíření tohoto systému. V poslední části je rozebírán způsob integrace systému do platformy NetBeans.

5.1 Specifikace pravidel požadovaných zadáním práce

Nejprve definujeme vhodné operátory, které nám umožní vybrat množiny vrcholů tak, abychom mohli specifikovat *LoD*. Vybrané množiny ověříme pomocí predikátu, který určí, zda jsou tyto množiny v pořádku, či zda porušují *LoD* princip.

Definice. *Mějme grafový model projektu* $G = \langle V, E, \rho, K, C, N, Kind, Classifier, Name \rangle$. *Definujme selektor vrcholů* $L_0(G, v', c')$, $v' \in V$, $c' \in C$ *jako množinu vrcholů* $v \in V$, *které jsou přímými následníky vrcholu* v', *do kterých se dostaneme po hraně* $e \in E$, *pro kterou platí* $Classifier(e) = c'$.

Definice. *Mějme grafový model projektu* $G = \langle V, E, \rho, K, C, N, Kind, Classifier, Name \rangle$. *Definujme selektor vrcholů* $L_1(G, v', c_1, c_2)$, $v' \in V$, $c_1 \in C$, $c_2 \in C$ *jako množinu vrcholů* $v \in V$, *do nichž se dostaneme z vrcholu* v' *po orientované cestě délky 2, pro jejíž první hranu* e_1 *platí* $Classifier(e_1) = c_1$ *a pro druhou hranu* e_2 *platí* $Classifier(e_2) = c_2$.

5.1.1 Law of Demeter

Pro definici princip *LoD* zavedeme speciální typ grafového modelu. Ačkoliv jsme pojem *typ grafového modelu* přesně nedefinovali a zavedli jej pouze intuitivně, pokusíme se nyní vymezit pojem *demeter graph model*, který zde budeme dále používat, přesněji.

U grafového modelu typu *Demeter graph type* povolíme libovolná označení vrcholů (konkrétně pro potřeby *LoD* se bude jednat o plně kvalifikované názvy tříd a metod). Množinu typů vrcholů omezíme na následující množinu:

$$K = \{\text{“}class\text{”}, \text{“}method\text{”}\}$$

Význam těchto typů je zřejmý. Typ *class* použijeme pro vrcholy, které budou odpovídat deklarovaným třídám v existujícím projektu a typ *method* bude označovat jejich metody.

Taktéž množina klasifikátorů hran nebude libovolná, ale bude přesně daná:

$$C = \{\text{“}field\text{”}, \text{“}self\text{”}, \text{“}arg\text{”}, \text{“}constr\text{”}, \text{“}use\text{”}\}$$

Podívejme se na význam těchto označení:

- “*field*” – hrana vede do třídy, která reprezentuje třídní proměnnou nebo do její nadtřídy,
- “*self*” – hrana vede do třídy, která je vlastníkem této metody nebo do její nadtřídy,
- “*arg*” – hrana vede do třídy argumentu metody nebo do její nadtřídy,
- “*constr*” – hrana vede do třídy použité pro konstrukci nového objektu v těle metody nebo do její nadtřídy,
- “*use*” – hrana vede do třídy, jejíž použití (vyvolání metody nebo přístup k poli) se vyskytuje v těle metody.

Je patrné, že tyto hrany zčásti odpovídají požadavkům *LoD*. Prozatím jsme uvedli množiny klasifikátorů bez vymezení, kde lze přesně tyto klasifikátory použít. Uvažujme následující množinu, která reprezentuje zobrazení množiny klasifikátorů do množiny dvojic typů vrcholů:

$$\begin{aligned} GT = \{ & \\ & (\text{“}field\text{”}(\text{“}class\text{”}, \text{“}class\text{”})), \\ & (\text{“}self\text{”}(\text{“}method\text{”}, \text{“}class\text{”})), \\ & (\text{“}arg\text{”}(\text{“}method\text{”}, \text{“}class\text{”})), \\ & (\text{“}constr\text{”}(\text{“}method\text{”}, \text{“}class\text{”})), \\ & (\text{“}use\text{”}(\text{“}method\text{”}, \text{“}class\text{”})) \\ & \} \end{aligned}$$

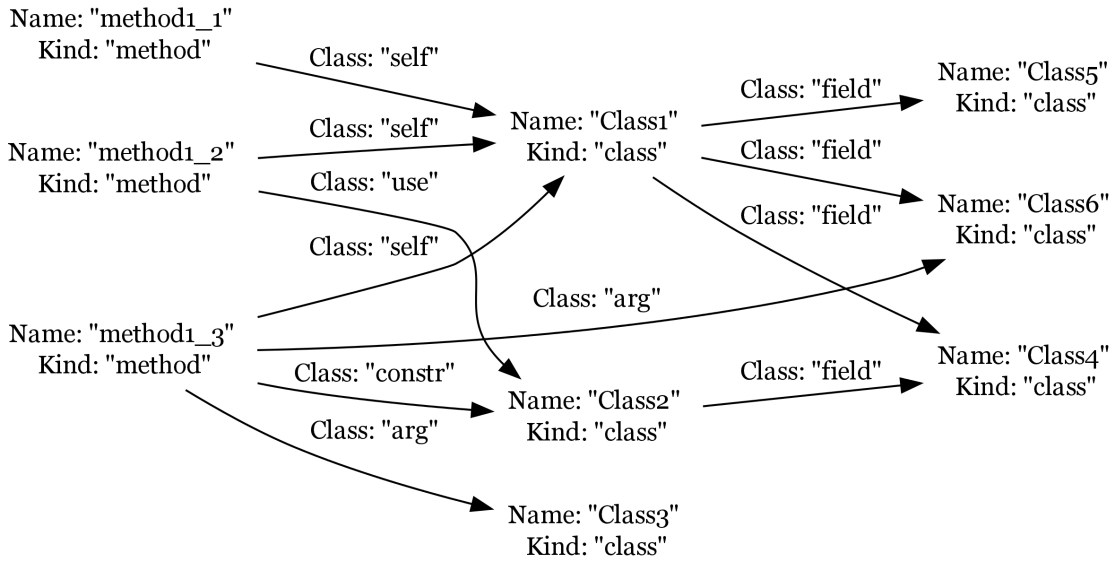
Pokud budeme uvažovat koncové vrcholy jednotlivých hran a jejich zobrazení do množiny typů, potom musí platit, že obraz jednotlivých komponent hrany (vrcholů) bude shodný s obrazem hrany provedeného pomocí zobrazení GT.

Pokud *grafový model projektu* splní všechny tyto podmínky, řekneme o něm, že je typu *demeter*.

Příklad takového grafu je na obrázku 5.1. V uvedeném obrázku princip *LoD* porušuje metoda `method1_2`, která používá třídu `Class2`, ačkoliv není ani jednou z dovolených tříd principu *LoD* (třída pole vlastnické třídy metody, argument metody, atd.).

Návrhový princip. *Mějme grafový model projektu* $G = \langle V, E, \rho, K, C, Kind, Classifier \rangle$ *typu demeter. Model* G *splňuje návrhový princip* LoD*, pokud pro něj platí:*

$$\forall v \in V : Kind(v) = \text{“}method\text{”}$$



Obrázek 5.1: Graf „typu demeter“ použitý pro validaci principu *LoD*.

$$L_0(G, v, \text{“}use\text{”}) \setminus (L_0(G, v, \text{“}self\text{”}) \cup L_1(G, v, \text{“}self\text{”}, \text{“}field\text{”}) \cup$$
$$L_0(G, v, \text{“}arg\text{”}) \cup L_0(G, v, \text{“}constr\text{”})) = \emptyset$$

Specifikované pravidlo vyjadřuje požadavek, aby množina vrcholů (představujících třídy) do níž se dostaneme pomocí hran s klasifikátorem “*use*” (všechna použití tříd uvnitř metody) byla podmnožinou sjednocení vrcholů, které pro danou metodu představují povolená použití tříd (vlastní třída metody, třída třídní proměnné, argument metody nebo konstruovaná třída). Je-li totiž množina “*use*” vrcholů podmnožinou sjednocení ostatních množin, množinový rozdíl bude roven prázdné množině. V opačném případě bude množinový rozdíl nenulový, což bude znamenat porušení principu *LoD*.

5.1.2 Low coupling, high cohesion

Zatímco princip *LoD* se nám podařilo přesně specifikovat pomocí množin. U principů *low coupling* a *high cohesion* bychom spíše uvažovali o realizaci vhodné analýzy. Tyto analýzy však nebyly cílem této práce.

Přesto byla realizována programová podpora, která umožní snadné přidání modulu, který může pracovat nad libovolným grafem poskytovaným některým z existujících poskytovatelů generátorů grafů a tyto nebo jiné podobné vlastnosti analyzovat. Většinou by se jednalo o nejrůznější statistické analýzy založené na počtech hran, seskupování nebo generování histogramů.

Konkrétní informace a rešerše týkající se těchto principů byly rozebrány v kapitole 3.

5.2 Implementace jádra systému

Jádro systému bylo implementováno v rámci samostatného modulu platformy NetBeans. Ačkoliv je takto svázáno s konkrétní platformou, zdrojové kódy neobsahují žádné závislosti na prvky této platformy a je tedy možné je znovu použít v jiných projektech.

V této sekci nebudeme procházet úplně všechny implementované komponenty systému. Jednak byly popsány již v části návrhu a kromě toho jsou jejich implementace k dispozici v rámci zdrojových kódů. Při implementaci byla snaha o maximální zdokumentovanost – většina klíčových tříd je řádně opatřena dokumentací `javadoc`.

Zde se podíváme na implementaci formátu pro zadávání pravidel (AVD specifikace). Dále na způsob sestavování vyhodnocovacího stromu a nakonec ukážeme implementaci zajímavých operátorů, které jsou základem systému pro vyhodnocování pravidel.

5.2.1 Implementace formátu AVD

Formát AVD byl vytvořen snahou o namapování dříve definovaných matematických výrazů do vhodné serializace. Pro zpracování existujícího souboru byl použit lexikální a syntaktický analyzátor vygenerovaný pomocí nástroje ANTLR. Tento nástroj umí kromě vytvoření čistého *parse tree* vytvořit i vhodný AST pomocí jednoduchých přepisovacích pravidel (konkrétní příklady pravidel si můžete prohlédnout v příloze B.3, kde je pro refrenci uvedena celá vstupní gramatika pro nástroj ANTLR).

V rámci AVD specifikace bylo nutné definovat několik nezávislých jmenných prostorů:

- názvy analýz,
- jména typů grafů,
- jména atomických a složených pravidel (jmenný prostor pro atomická i složená pravidla je sdílený),
- jména operátorů.

Proto je například možné použít stejný název pro typ grafu a pro atomické pravidlo, ale není možné použít shodný název pro dvě pravidla (ať už atomická nebo složená).

5.2.2 Implementace sestavování validačního modelu

Na základě AST, který byl získán pomocí syntaktického parseru vygenerovaného pomocí nástroje ANTLR byl postupně budován vyhodnocovací strom odpovídající zpracovávanému AVD souboru.

Zpracování AST probíhá tím způsobem, že je nejprve zkontrolována bezkoliznost použitých jmen (viz výše zmíněné jmenné prostory). Následně jsou procházeny jednotlivé listy představující pravidla a jsou postupně doplňovány názvy nalezených pravidel do tabulky symbolů. Pro každé pravidlo je nutné sestoupit do syntaktických stromů představujících např. logické výrazy a pro každý takto nalezených vrchol (např. operátor `OR`) je nutné vytvořit nový vrchol vyhodnocovacího stromu.

Tyto nově vytvářené vrcholy jsou realizované pro základní operátory jako samostatné třídy. Pro uživatelské operátory potom existují vrcholy, kterým je při konstrukci možné nastavit operátor, který mají provádět. Předtím, než jsou ale tyto vrcholy vytvořeny, je provedena kontrola, zda počty argumentů a návratová hodnota odpovídá očekávaným hodnotám (např. při zpracování predikátu kontrolujeme, zda množina operandů odpovídá množině deklarované operátorem nalezeným v registru operátorů.).

Při budování stromu je současně budována infrastruktura pro generování stromu výsledků. V každém přidávaném vrcholu (představován třídou) je tedy již přítomen kód, který při vyhodnocení uzlu (metoda `evaluate()`) naplní datovou strukturu předanou jako parametr.

5.2.3 Implementace základních operátorů

V této podsekci se podíváme na některé zajímavé operátory a jejich implementaci. Konkrétně na existenční ∃ a univerzální ∀ kvantifikátor.

Vyhodnocení univerzálního kvantifikátoru $\forall v \in V$ lze přepsat jako jednoduchý cyklus přes vrcholy vrácené zpřesňující operátorem, která vybírá nějakou podmnožinu z množiny vrcholů analyzovaného grafu. Získáme tak kód podobný výpisu 5.1. V tomto kódu, který je fragmentem metody, používáme symbolický název `condition` pro podmínku, kterou musí splňovat každý prvek v iterované kolekce vrcholů *vertices*.

Výpis kódu 5.1: Implementace univerzálního kvantifikátoru ∀.

```
for (Vertex v : vertices) {
    if (!condition(v, ...)) {
        return false;
    }
}
return true;
```

Analogicky budeme postupovat u existenčního kvantifikátoru ∃. Zde nám stačí nalézt alespoň jeden element, pro který vyhodnocovaná vlastnost platí. Výpis 5.2 představuje fragment metody. Pokud se podaří nalézt alespoň jeden prvek, který splňuje požadovanou podmínku, metoda vrátí hodnotu `true`. Vzhledem k použití příkazu `return` je zřejmé, že dochází k „línému“ vyhodnocování – vyhodnocení je ukončeno nalezením prvního vyhovujícího elementu (další se neprohledávají). Stejně jako u operátoru ∀ i zde název `condition` symbolizuje konkrétní podmínku, která má platit nad jedním prvkem množiny uzlů.

Výpis kódu 5.2: Implementace existenčního kvantifikátoru ∃.

```
for (Vertex v : vertices) {
    if (condition(v, ...)) {
        return true;
    }
}
return false;
```

Můžeme si všimnout, že se obě implementace liší pouze prohozením podmínek – zatímco v prvním případě (∀) musíme projít všechny prvky, abychom zjistili, zda všechny prvky splňují požadovanou vlastnost, ve druhém případě (∃) budeme všechny prvky procházet pouze v krajním případě, kdy vlastnost neplatí pro žádný z prvků.

Podobným způsobem jsou realizovány i další operátory. Například sjednocení množin využívá operací poskytovaných třídou `HashSet` programovacího jazyka Java.

5.3 Implementace modulů rozšíření

V rámci této práce byly realizovány celkem dva NetBeans moduly realizující rozšíření. Tyto moduly mohou současně sloužit jako návod k implementaci dalších rozšíření. Jedná se o poskytovatele generátoru grafu typu *demeter* a balíček operátorů, které jsou nezbytné k provedení pravidla *LoD*. Tyto operátory nicméně nejsou vázány pouze na princip *Law of Demeter*. Například operátor, který testuje, zda je množina vrcholů prázdná má zcela jistě obecné použití.

5.3.1 Modul av-graphgen-demeter (generátor grafu)

Modul pro generování grafu nad nímž je možné provést kontrolu principu *LoD*, je realizován díky možnosti využít existující infrastrukturu platformy NetBeans pro práci se zdrojovými kódy v jazyce Java a jejich syntaktickými stromy. Jak již bylo zmíněno dříve v rámci analýzy, platforma NetBeans poskytuje AST stromy, které mají již rozlišené symboly. Díky tomu můžeme stromy postupně procházet a dotazovat se na plně kvalifikovaná jména elementů představujících třídy, rozhraní atd.

Generátor grafu je realizován v rámci třídy *DemeterGraphGenerator* (implementuje rozhraní *GraphGeneratorIface*). Protože vstupním parametrem pro generátor grafu je „pouze“ adresář projektu, je nutné vylistovat všechny soubory, které mohou obsahovat zdrojový kód v jazyce Java (`*.java` soubory). Následně je pro každý takový soubor získána instance třídy *JavaSource* poskytované platformou NetBeans. Tato třída je vstupním bodem k funkcionalitám poskytovaným kompilátorem jazyka Java (třída JavaSource tato rozhraní zabaluje). Díky tomu je možné využít *Java Tree API* pro iteraci všech syntaktických elementů v rámci kompilační jednotky.

To je provedeno formou návrhového vzoru *Visitor*. Při generování demeter grafu nemusíme naštěstí přepisovat všechny metody třídy *TreePathVisitor*, ale zaměřujeme se pouze na elementy, které potřebujeme pro vygenerování grafu. Pro vygenerování potřebného grafu nám stačí získat následující informace: které třídy je metoda členem, zjištění všech nadtříd třídy, konstrukce nové proměnné (nebo pole proměnných), deklarace polí třídy, vyvolání metody a přístup k poli třídy. Ostatní syntaktické konstrukce nejsou pro tento druh generování grafu podstatné.

5.3.2 Modul av-operators-demeter (operátory pro LoD)

Balík operátorů poskytovaný modulem `av-operators-demeter` obsahuje následující operátory:

Tabulka 5.1: Tabulka integračních komponent systému.

Název	**Typ**	**Zodpovědnost**
ArchvalInstance	class (singleton)	zabaluje jednotnou instanci jádra *ArchVal*
ValidateMainProject	class	integrace s běhovou platformou, vstupní/aktivační bod
GraphGeneratorRegister	class	třída poskytující přístup k informacím o existujících poskytovatelích generátorů grafu
OperatorRegister	class	třída poskytující přístup k informacím o existujících poskytovatelích operátorů
AnalysesRegister	class	třída poskytující přístup k informacím o existujících poskytovatelích komponent analýzy

- `vertex_set_is_empty` – třída *EmptyVertexSetPredicate* – operátor, který vrátí `true` v případě, že je množina vrcholů předaná jako parametr prázdná,
- `level_zero_vertex_selector` – třída *LevelZeroVertexSelector* – implementace operátoru L_0 definovaného výše,
- `level_one_vertex_selector` – třída *LevelOneVertexSelector* – implementace operátoru L_1 definovaného výše,
- `kind` – třída *VertexKindPredicate* – predikát, který vrátí `true`, pokud má vrchol předaný jako první parametr typ shodný s řetězcem předaným jako druhý parametr.

Implementace operátorů je poměrně přímočará. Operátory L_1, L_2 na základě předaného vrcholu prochází graf a vyhledávají potřebné vrcholy dle definice. Operátor *Kind* pouze testuje, zda vrchol předaný jako parametr, je požadovaného typu.

5.4 Integrace do NetBeans IDE

Pro demonstraci jádra systému bylo nutné jej zaintegrovat do vhodného běhového prostředí. K tomu dobře posloužila platforma NetBeans. Při implementaci bylo hojně využíváno informací z [22]. Integrace byla provedena realizací vhodných integračních komponent v rámci modulu `av-platform-integration`. Jejich seznam je k dispozici v tabulce 5.1.

5.4.1 Použití jádra systému ArchVal

Vstupním bodem pro použití rozhraní jádra systému *ArchVal* v rámci integračního modulu je třída *ArchvalInstance*. Tato třída je implementována jako návrhový vzor *singleton*

a poskytuje jedinou instanci třídy *ArchVal* pro celý systém. Díky tomu je možné se dále dostat k důležitým komponentám jako *ValidationModelGenerator*, *GraphModelGenerator* a vygenerovat instanci *ValidationTask*, která umí provést výslednou validaci.

5.4.2 Přidání uživatelské akce do nabídky prostředí

Aby bylo možné vyvolat validaci, bylo nutné provést integraci akce do nabídky NetBeans IDE. Akce představovaná třídou *ValidateMainProject* byla realizována jako podmíněně povolená akce, která je k dispozici pouze pokud je vybrán soubor konkrétního typu. V tomto případě se jedná o soubor typu AVD. Celý případ užití funguje potom tím způsobem, že uživatel klepne pravým tlačítkem na AVD soubor a vybere položku *Validate main project*. Pokud není žádný projekt vybrán jako hlavní, v oznamovací oblasti se pouze objeví upozornění a není provedena žádná akce. V opačném případě máme k dispozici oba vstupy potřebné k provedení validace (pomocí *Project API* získáme informace o hlavním projektu a tedy i vstupním adresáři a cesta k AVD souboru je poskytována v rámci handleru kontextové akce AVD souboru).

5.4.3 Výstupní rozhraní

Pro zobrazení výstupu použijeme NetBeans *I/O API*, které nám umožňuje využít jednoduché textové okno, které se používá v NetBeans IDE například pro zobrazení výstupů z logu nebo informace o průběhu kompilace. Poskytované aplikační rozhraní nám umožňuje otevřít novou záložku, kde můžeme samostatně logovat výsledky validační akce. Nástroj v základní verzi implementuje pouze výstup ve tvaru *[název pravidla] OK* nebo *[název pravidla] Violation found.*

5.4.4 Implementace registrů poskytovatelů služeb

Díky systému *Lookup* poskytovanému platformou NetBeans je implementace registrů poskytovatelů služeb velmi pohodlná. Pro každé rozhraní, pro něž chceme využívat *lookup* stačí přidat seznam implementujících tříd do konkrétního souboru v adresáři *META-INF/services*. Následně pak můžeme využít jednoduché volání pro zisk všech instancí implementujících dané rozhraní v celém systému:

```
Lookup.getDefault().lookupAll(OperatorIface.class)
```

Zmíněný příklad slouží k získání všech existujících tříd poskytujících rozhraní operátorů. Získané reference si v implementovaných registračních třídách ponecháváme uložené v kolekcích, abychom nemuseli opakovaně vyhledávat instance v lookupu, zvláště když víme, že tam vzhledem k jejich povaze žádné nemohou přibýt.

5.4.5 Podpora editace AVD souborů

Pro zjednodušení zadávání AVD souboru byla realizována podpora zvýrazňování syntaxe. Ta je realizována v rámci samostatného modulu `av-avd-highlighter`, který využívá služeb

jádra (ANTLR lexer pro tokenizing AVD souborů). Zvýrazňování syntaxe bylo integrováno do platformy NetBeans pomocí několika jednoduchých adaptérů, které zapouzdřují lexer vygenerovaný pomocí nástroje ANTLR (který se používá již v modulu `av-core`).

Aby platforma NetBeans mohla soubory AVD rozpoznat a pracovat s nimi stejně jako s ostatními datovými uzly, je nutné zaregistrovat do platformy nový typ souboru. Pro potřeby této práce byl použit experimentální MIME typ `text/x-avd`.

Kapitola 6

Testování

Výsledný systém byl testován jednak během vývoje a následně i po jeho dokončení. Testování zahrnovalo jednak *testy jednotek*, které jsou popsány v první sekci této kapitoly a dále *ověřování činnosti nástroje na vzorových příkladech kódu*, jak bylo požadováno zadáním práce.

Pro ověření činnosti nástroje byly vytvořeny příklady různých typů. Zabýváme se zde příklady, které princip *LoD* porušují, dále příklady, pro něž systém vyhodnocuje chybně správné příklady jako nevalidní a speciálními případy.

6.1 Testování jednotek

Pro zajištění kvalitního zdrojového kódu byly použity testy jednotek. Fáze návrhu již bere jednotkové testování v potaz. Díky specifikaci rozhraní pro většinu komponent bylo možné poskytnout vlastní testovací implementace objektů (mock objekty). Například třída *MockOperatorsRegister* implementuje rozhraní *OperatorsRegisterIface* a poskytuje falešné instance operátorů, aby bylo možné otestovat, zda vytváření vyhodnocovacího stromu probíhá dle očekávání.

Vzhledem k časové náročnosti realizace testů jsou napsány pouze testy klíčových komponent systému. Jednotkové testování bylo velmi pohodlným nástrojem zejména v počátečních fázích projektu, kdy celý projekt nebyl spustitelný a nebylo jej možné otestovat ručně.

Kromě testování jednotek byla provedena rešerše možností testování na platformě NetBeans. Platforma NetBeans v posledních verzích podporuje nejen jednotkové testování ale i testy funkčnosti prostřednictvím tzv. operátorů. Tyto operátory jsou schopné realizovat uživatelské akce jako výběr položky z menu, založení projektu atd. Byl realizován pouze elementární test funkčnosti, který sloužil ke kontrole toho, jestli se aplikace správně spustí (podaří se načtení všech modulů).

6.2 Ověřování funkčnosti na příkladech

Abychom demonstrovali funkčnost vytvořeného systému, bylo provedeno testování na konkrétních příkladech. Tyto příklady jsou zahrnuty v podadresáři `samples` na CD, které je nedílnou součástí této práce. Podívejme se na jednotlivé případy, které byly ověřovány:

6.2.1 Příklady porušení principu LoD

Ukázka `method_return_access` demonstruje přístup k rozhraní třetí třídy na základě reference vrácené voláním metody na jiné třídě, k níž máme přístup. Podobně funguje i ukázka `member_field_access`.

Příklad `hidden_violation` ukazuje další porušení principu*LoD*. Současně je zde demonstrováno, jak by mělo zpracování probíhat správně.

Všechny tyto příklady vyhodnotí *ArchVal* správně jako `Violation found`.

6.2.2 Falešně pozitivní případy

Na základě testování se ukázalo, že systém pro některé příklady, které princip *LoD* neporušují vrací informaci o porušení tohoto principu. Při bližším zkoumání se ukázalo, že se jedná o výjimky, jejichž použití je běžné nebo zvláštní vlastnosti jazyka, s nímž nepočítá generátor grafu.

Příklad `string_literal_invalid` demonstruje falešně pozitivní detekci porušení principu *LoD* ačkoliv z hlediska *LoD* se jedná o případ, kdy konstruujeme nový objekt. Protože ale generátor grafu vyhledává volání konstruktorů, je konstrukce řetězce pomocí uvozovek vynechána. Pokud bychom nahradili tyto řetězce voláním konstruktoru pro objekt `String`, systém bude reagovat správně.

Dalším příkladem, který vyhodnotí systém jako porušení je běžný výpis řetězce na standardní výstup:

```
System.out.println();
```

Pro nasazení na reálných projektech je nutné tyto výjimky zohlednit a uměle rozšířit množiny hran v grafu tak, aby tyto případy byly ignorovány.

6.2.3 Speciální případy

Kromě testování na běžných projektech bylo provedeno ještě testování na specifickém případu, kdy projekt obsahuje prázdné `*.java` soubory nebo soubory, které obsahují pouze deklarace `import`. Systém se v takových případech choval správně – žádné porušení principu *LoD* nebylo detekováno.

Kapitola 7

Závěr

Práce řeší problematiku kontroly správného objektového návrhu software na základě analýzy zdrojových kódů.

V práci navrhuji použití teorie grafů a jednoduchého matematického aparátu skládajícího se z kvantifikátorů, predikátů a selektorů pro zobecnění definice návrhových principů. Tento návrh je podpořen vytvořením nástroje, který umí takto specifikovaná pravidla vyhodnocovat.

Při návrhu systému je uvažována zejména jeho rozšiřitelnost. Je možné snadno přidávat nové operátory implementací jednoduchého rozhraní a registrací tohoto operátoru do systému. Taktéž je možné systém rozšířit o různé druhy analýzy prováděné nad grafovými reprezentacemi projektů. Systém byl záměrně navržen tak, aby bylo možné přidáním generátoru grafů tento nástroj využít nejen pro analýzu kódu v jazyce Java, ale v libovolném jazyce, pro nějž poskytneme příslušnou komponentu na jeho převedení do grafové reprezentace.

V rámci vyhodnocovacího systému bylo realizováno pravidlo *Law of Demeter*. Pravidlo je formulováno pomocí jednoduchého predikátu předpokládajícího prázdnou množinu a několika selektorů, které vybírají vhodné množiny vrcholů z grafu.

Výsledkem práce je jednak návrh přístupu ke strukturální analýze softwarových projektů a dále existující nástroj pro vyhodnocování matematických pravidel nad vhodnými grafovými reprezentacemi zdrojových kódů analyzovaných projektů.

Možná pokračování práce

Při realizaci systému bylo realizováno vyhodnocovací jádro, které vyžaduje ke své činnosti další operátory a rozšíření. Ukázková implementace detekce porušování principu *Law of Demter* je velmi striktní. Pro nasazení nástroje v praxi by bylo nutné rozšířit nástroj o povolené výjimky z pravidla *LoD*.

Dalším důležitým pokračováním je zlepšení výstupu nástroje. Během realizace vyhodnocovacího systému byly již ve fázi návrhu uvažovány možnosti lepšího výstupu. Systém během vyhodnocování generuje strom výsledků pro každý uzel vyhodnocovacího stromu. Vhodným zpracováním zmíněného stromu by bylo možné zobrazit konkrétní místa, kde došlo k porušení vyhodnocovaného principu.

Literatura

[1] *JDK 6 Java Compiler (javac)-related APIs & Developer Guides* [online]. [cit. 4. 5. 2011]. Dostupné z: <http://download.oracle.com/javase/6/docs/technotes/guides/javac/index.html>.

[2] *checkstyle - Checkstyle 5.3* [online]. 2011. [cit. 1. 5. 2011]. Dostupné z: <http://checkstyle.sourceforge.net/>.

[3] *DP-Miner, A Tool for Design Pattern Recovery* [online]. 2011. [cit. 1. 5. 2011]. Dostupné z: <http://www.utdallas.edu/~yxz045100/DesignPattern/DP_Miner/>.

[4] *FindBugs TM - Find Bugs in Java Programs* [online]. 2011. [cit. 1. 5. 2011]. Dostupné z: <http://findbugs.sourceforge.net/>.

[5] *JDepend* [online]. 2011. [cit. 1. 5. 2011]. Dostupné z: <http://www.clarkware.com/software/JDepend.html>.

[6] *Macker* [online]. 2011. [cit. 1. 5. 2011]. Dostupné z: <http://innig.net/macker/>.

[7] *PMD* [online]. 2011. [cit. 1. 5. 2011]. Dostupné z: <http://pmd.sourceforge.net/>.

[8] *Code Analyzer for Java* [online]. 2011. [cit. 1. 5. 2011]. Dostupné z: <http://qjpro.sourceforge.net>.

[9] *Soot: a Java Optimization Framework* [online]. 2011. [cit. 1. 5. 2011]. Dostupné z: <http://www.sable.mcgill.ca/soot/>.

[10] *Squale - Squale: the open source quality and qualimetry project!* [online]. 2011. [cit. 1. 5. 2011]. Dostupné z: <http://www.squale.org/>.

[11] *The Hacker's Guide to Javac* [online]. [cit. 4. 5. 2011]. Dostupné z: <http://scg.unibe.ch/archive/projects/Erni08b.pdf>.

[12] *Design Principles | Object Oriented Design* [online]. [cit. 3. 5. 2011]. Dostupné z: <http://www.oodesign.com/design-principles.html>.

[13] *ANTLR Parser Generator* [online]. [cit. 4. 5. 2011]. Dostupné z: <http://www.antlr.org/>.

[14] *Compiler Tree API* [online]. 2011. [cit. 1. 5. 2011]. Dostupné z: <http://download.oracle.com/javase/6/docs/jdk/api/javac/tree/index.html>.

[15] *Help - Eclipse SDKa* [online]. [cit. 4. 5. 2011]. Dostupné z: <http://help.eclipse.org/helios/index.jsp?topic=/org.eclipse.jdt.doc.isv/reference/api/org/eclipse/jdt/core/dom/package-summary.html>.

[16] *JavaCC Documentation Index* [online]. 2011. [cit. 1. 5. 2011]. Dostupné z: <http://www.cs.purdue.edu/homes/hosking/352/javaccdocs/docindex.html>.

[17] *javaparser - Java 1.5 Parser and AST - Google Project Hosting* [online]. 2011. [cit. 1. 5. 2011]. Dostupné z: <http://code.google.com/p/javaparser/>.

[18] *NetBeans API Index* [online]. 2011. [cit. 1. 5. 2011]. Dostupné z: <http://bits.netbeans.org/6.9.1/javadoc/>.

[19] *Spoon: Program Processing, Analysis, and Transformation in Java* [online]. 2011. [cit. 2. 5. 2011]. Dostupné z: <http://spoon.gforge.inria.fr/>.

[20] AYDOGAN, S. *Design concepts and principles* [online]. 2005. [cit. 2. 5. 2011]. Dostupné z: <http://cs.bilgi.edu.tr/pages/courses/year_3/comp_332/Archive/2004-2005/Article.pdf>.

[21] BOCK, D. *The Paperboy, The Wallet, and The Law Of Demeter* [online]. [cit. 4. 5. 2011]. Dostupné z: <http://www.ccs.neu.edu/research/demeter/demeter-method/LawOfDemeter/paper-boy/demeter.pdf>.

[22] BOCK, H. *Platforma NetBeans : podrobný průvodce programátora.* Brno : Computer Press, 2010. ISBN 978-80-251-3116-9.

[23] DONG, J. – ZHAO, Y. Experiments on Design Pattern Discovery. In *Predictor Models in Software Engineering, 2007. PROMISE'07: ICSE Workshops 2007. International Workshop on*, s. 12, may 2007. doi: 10.1109/PROMISE.2007.6.

[24] GOSLING, J. et al. *Java(TM) Language Specification, The (3rd Edition) (Java (Addison-Wesley))*. Addison-Wesley Professional, 2005. ISBN 0321246780.

[25] JIN, Z. – OFFUTT, A. J. Coupling-based criteria for integration testing. *Software Testing, Verification and Reliability*. 1998, 8, 3, s. 133–154. ISSN 1099-1689. doi: 10.1002/(SICI)1099-1689(1998090)8:3<133::AID-STVR162>3.0.CO;2-M. Dostupné z: <http://dx.doi.org/10.1002/(SICI)1099-1689(1998090)8:3<133::AID-STVR162>3.0.CO;2-M>.

[26] KANG, B. – BIEMAN, J. Using design cohesion to visualize, quantify, and restructure software. In *SEKE '96: THE 8TH INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING AND KNOWLEDGE ENGINEERING, PROCEEDINGS*, s. 222–229, 1996. ISBN 0-9641699-3-2, 8th International Conference on Software Engineering and Knowledge Engineering (SEKE 96), LAKE TAHOE, NV, JUN 10-12, 1996.

[27] KANG, B.-K. – BIEMAN, J. M. Design-Level Cohesion Measures: Derivation, Comparison, and Applications. In *Proceedings of the 20th Conference on Computer Software and Applications*, COMPSAC '96, s. 92–, Washington, DC, USA, 1996. IEEE Computer Society. Dostupné z: <http://portal.acm.org/citation.cfm?id=872750.873361>.

[28] LIEBERHERR, K. – HOLLAND, I. Assuring good style for object-oriented programs. *Software, IEEE*. September 1989, 6, 5, s. 38 – 48. ISSN 0740-7459. doi: 10.1109/52.35588.

[29] MARTIN, R. C. *Design Principles and Design Patterns* [online]. [cit. 3. 5. 2011]. Dostupné z: <http://www.objectmentor.com/resources/articles/Principles_and_Patterns.pdf>.

[30] MENS, T. – CZARNECKI, K. – GORP, P. V. A taxonomy of model transformation. In *Proc. Dagstuhl Seminar on "Language Engineering for Model-Driven Software Development". Internationales Begegnungs- und Forschungszentrum (IBFI), Schloss Dagstuhl.* Electronic, 2005.

[31] PESCIO, C. Principles versus patterns. *Computer*. September 1997, 30, 9, s. 130 –131. ISSN 0018-9162. doi: 10.1109/2.612257.

[32] Přispěvatelé Wikipedie. *Checkstyle - Wikipedia, the free encyclopedia* [online]. 2011. [cit. 1. 5. 2011]. Dostupné z: <http://en.wikipedia.org/wiki/Checkstyle>.

[33] Přispěvatelé Wikipedie. *Cohesion (computer science) - Wikipedia, the free encyclopedia* [online]. 2011. [cit. 1. 4. 2011]. Dostupné z: <http://en.wikipedia.org/wiki/Cohesion_(computer_science)>.

[34] Přispěvatelé Wikipedie. *Coupling (computer programming) - Wikipedia, the free encyclopedia* [online]. 2011. [cit. 25. 3. 2011]. Dostupné z: <http://en.wikipedia.org/wiki/Coupling_(computer_programming)>.

[35] Přispěvatelé Wikipedie. *Fault-tolerant system - Wikipedia, the free encyclopedia* [online]. 2011. [cit. 3. 5. 2011]. Dostupné z: <http://en.wikipedia.org/wiki/Fault-tolerance>.

[36] Přispěvatelé Wikipedie. *4+1 Architectural View Model - Wikipedia, the free encyclopedia* [online]. 2011. [cit. 11. 4. 2011]. Dostupné z: <http://en.wikipedia.org/wiki/4%2B1_Architectural_View_Model>.

[37] Přispěvatelé Wikipedie. *Law of Demeter - Wikipedia, the free encyclopedia* [online]. 2011. [cit. 6. 4. 2011]. Dostupné z: <http://en.wikipedia.org/wiki/Law_of_Demeter>.

[38] Přispěvatelé Wikipedie. *Object-oriented design - Wikipedia, the free encyclopedia* [online]. 2011. [cit. 3. 5. 2011]. Dostupné z: <http://en.wikipedia.org/wiki/Object-oriented_design#Object-oriented_concepts>.

[39] Přispěvatelé Wikipedie. *Programming paradigm - Wikipedia, the free encyclopedia* [online]. 2011. [cit. 11. 4. 2011]. Dostupné z: <http://en.wikipedia.org/wiki/Programming_paradigm>.

[40] Přispěvatelé Wikipedie. *Systems Development Life Cycle - Wikipedia, the free encyclopedia* [online]. 2011. [cit. 11. 4. 2011]. Dostupné z: <http://en.wikipedia.org/wiki/Systems_Development_Life_Cycle>.

[41] Přispěvatelé Wikipedie. *Software architecture - Wikipedia, the free encyclopedia* [online]. 2011. [cit. 11. 4. 2011]. Dostupné z: <http://en.wikipedia.org/wiki/Software_architecture>.

[42] Přispěvatelé Wikipedie. *Software design - Wikipedia, the free encyclopedia* [online]. 2011. [cit. 3. 5. 2011]. Dostupné z: <http://en.wikipedia.org/wiki/Software_design>.

[43] Richard S. *Source Code Analysis Using Java 6 APIs : Core Java Technologies Tech Tips* [online]. 2008. [cit. 27. 3. 2011]. Dostupné z: <http://today.java.net/pub/a/today/2008/04/10/source-code-analysis-using-java-6-compiler-apis.html>.

[44] THAKUR, T. *Service Provider Interface: Creating Extensible Java Applications* [online]. 2011. [cit. 2. 5. 2011]. Dostupné z: <http://www.developer.com/java/article.php/3848881/Service-Provider-Interface-Creating-Extensible-Java-Applications.htm>.

Příloha A

Seznam použitých zkratek

API Application Programming Interface

AST Abstract Syntax Tree

AVD ArchVal definition – zkratka používaná pro označení souboru specifikace pravidel

BNF Backus-Naur form – standardní forma zápisu bezkontextových gramatik

CLI Command line interface

CORBA Common Object Request Broker Architecture

DAO Data Access Object

DOM Document object model

DTD Document Type Definition

EBNF Extedned Backus-Naur form – novější způsob specifikace bezkontextových gramatik

GUI Graphical user interface

IDE Integrated development environment

JDK Java Development Kit

LoD Law of Demeter

PDF Portable Document Format

SPI Service Provider Interface

UML Unified Modeling Language

XML Extensible Markup Language

Příloha B

Instalační a uživatelská příručka

B.1 Instalace programu

K dispozici jsou dvě varianty instalace programu (v adresáři `dist` na přiloženém cd). Soubor `archval.zip` obsahuje spustitelnou variantu projektu. Pro platformu Linux je možné využít instalátor (soubor `achval-linux.sh`).

V případě instalace ze zip souboru stačí tento soubor rozbalit do vhodného umístění. Pro spuštění instalátoru může být nutné nastavit *executable bit* pomocí příkazu:

```
chmod u+x archval-linux.sh
```

U obou typů instalací je nutné provést konfiguraci systému v souboru `etc/archval.conf` v cílovém adresáři. Důvodem je přímá závislost projektu na souborech *Sun JDK API*. V souboru je nutné nastavit proměnnou `jdkhome` na úplnou cestu k JDK.

Program je možné spustit pomocí spouštěcích souborů, které jsou k dispozici v adresáři `bin/` (verze pro Linux a pro Windows).

B.2 Používání programu

Po spuštění programu je k dispozici základní verze NetBeans IDE. Systém podporuje jednak vytváření a editaci AVD souborů a dále provedení validace Maven Java projektů na základě existujícího AVD souboru.

Práce s AVD soubory je podobná práci s jinými typy souborů na platformě NetBeans. Nový AVD soubor vytvoříme pomocí posloupnosti akcí *File | New file... | Other | Empty AVD file*.

Editace souborů je stejná jako pro kterýkoliv jiný soubor projektu – stačí je otevřít buď poklepáním nebo pomocí pravého tlačítka myši. Soubor je otevřen v editoru, který podporuje zvýrazňování AVD syntaxe.

Máme-li k dispozici AVD soubor, můžeme jej použít k validaci projektu, který je vybrán jako hlavní projekt. Klepneme pravým tlačítkem na AVD soubor a vybereme položku *Validate main project*. Není-li vybrán žádný projekt jako hlavní, je zobrazena informace o absenci výběru ve status baru a není provedena žádná akce. V opačném případě je provedena validace projektu a výstup je zobrazen ve standardním výstupním okně NetBeans IDE.

B.3 Gramatika jazyka pro specifikaci pravidel

Výpis B.1 poskytuje k dispozici gramatiku jazyka AVD pro specifikaci pravidel. Tuto gramatiku je možné používat jako referenci při psaní AVD specifikací a také při realizaci rozšíření.

Výpis kódu B.1: Gramatika jazyka pro specifikaci pravidel (AVD soubory).

```
grammar ArchvalRulesetGrammar;

options {
    output=AST;
}

tokens {
    VALIDATION_UNIT;
    ATOMIC_RULES;
    COMPOUND_RULES;
    VALIDATE_COMMANDS;
    ANALYZE_COMMANDS;
    RULE_EXPRESSION;
}

@rulecatch {
}

@header {
    package cz.cvut.fel.archval.core.avd.parser;
}

@lexer::header {
    package cz.cvut.fel.archval.core.avd.parser;
}

validation_unit
        :
        atomic_rule*
        compound_rule*
        validate_command*
        analyze_command*
        EOF
        ->
        ^(VALIDATION_UNIT
        ^(ATOMIC_RULES atomic_rule*)
        ^(COMPOUND_RULES compound_rule*)
        ^(VALIDATE_COMMANDS validate_command*)
```

```
        ^(ANALYZE_COMMANDS analyze_command*))
        ;

atomic_rule :
        ATOMIC_RULE_KW rname=Name LPAREN grname=Name RPAREN
        LBRACE atomic_rule_spec RBRACE SEMICOLON
        ->
        ^($rname $grname atomic_rule_spec)
        ;

compound_rule :
        COMPOUND_RULE_KW Name
        LBRACE compound_rule_spec RBRACE SEMICOLON
        ->
        ^(Name compound_rule_spec)
        ;

validate_command :
        VALIDATE_KW LPAREN validate_command_params
        RPAREN SEMICOLON
        ->
        validate_command_params
        ;

analyze_command :
        ANALYZE_KW LPAREN analyze_command_params
        RPAREN SEMICOLON
        ->
        analyze_command_params
        ;

validate_command_params :
        Name (COMMA! Name)*
        ;

analyze_command_params :
        Name (COMMA! Name)*
        ;

atomic_rule_spec :
        set_spec_clause LBRACE orexpression RBRACE
        ->
        set_spec_clause ^(RULE_EXPRESSION orexpression)
        ;

set_spec_clause
```

```
        :
        quantifier_clause
        quantification_variable
        quantification_predicate?
        ->
        ^(quantifier_clause
        quantification_variable)
        quantification_predicate?
        ;

quantifier_clause
        :
        ALL^
        |
        EXISTS^
        ;

quantification_variable
        :
        Vertex IN VertexSet -> ^(Vertex ^(IN VertexSet))
        |
        Edge IN EdgeSet -> ^(Edge ^(IN EdgeSet))
        ;

quantification_predicate
        :
        (COLON Name LPAREN selector_params? RPAREN)
        ->
        ^(Name selector_params?)
        ;

orexpression
        :
        andexpression (OR^ andexpression)*
        ;

andexpression
        :
        notexpression (AND^ notexpression)*
        ;

notexpression
        :
        NOT^? atom
        ;
```

```
atom
        :
        condition
        |
        LPAREN orexpression RPAREN
        ->
        orexpression
        ;

condition
        :
        True
        |
        False
        |
        Name LPAREN predicate_params? RPAREN
        ->
        ^(Name predicate_params?)
        ;

predicate_params
        :
        predicate_param (COMMA predicate_param)*
        ->
        predicate_param predicate_param*
        ;

predicate_param
        :
        Number
        |
        Label
        |
        Vertex
        |
        Edge
        |
        set_expression
        ;

set_expression
        :
        set_atom ((INTERSECT^ | UNION^ | SETMINUS^) set_atom)*
        ;

set_atom
```

```
        :
        Name LPAREN selector_params? RPAREN
        ->
        ^(Name selector_params?)
        |
        LPAREN set_expression RPAREN
        ->
        set_expression
        ;

selector_params
        :
        selector_param (COMMA selector_param)*
        ->
        selector_param selector_param*
        ;

selector_param
        :
        Vertex
        |
        Edge
        |
        Number
        |
        Label
        ;

compound_rule_spec
        :
        candexpression (OR^ candexpression)*
        ;

candexpression
        :
        cnotexpression (AND^ cnotexpression)*
        ;

cnotexpression
        :
        NOT^? catom
        ;

catom
        :
        Name
```

```
        |
        LPAREN compound_rule_spec RPAREN
        ->
        compound_rule_spec
        ;

// operators
EXISTS  :       'EXISTS';
ALL     :       'ALL';
IN      :       'IN';

INTERSECT
        :       'INTERSECT';
UNION   :       'UNION';
SETMINUS:       'SETMINUS';

NOT     :       'NOT';
AND     :       'AND';
OR      :       'OR';

// keywords
ATOMIC_RULE_KW
        :       'atomic_rule';
COMPOUND_RULE_KW
        :       'compound_rule';
VALIDATE_KW
        :       'validate';
ANALYZE_KW
        :       'analyze';

// grouping operators
LBRACE  :       '{';
RBRACE  :       '}';
LPAREN  :       '(';
RPAREN  :       ')';
COMMA   :       ',';
SEMICOLON
        :       ';';
COLON   :       ':';

// literals
True    :       'true';
False   :       'false';
Vertex  :       'v';
Edge    :       'e';
VertexSet
```

```
        :       'V';
EdgeSet :       'E';
Name    :       ('a'..'z' | 'A'..'Z')
                ('a'..'z' | 'A'..'Z' | '0'..'9' | '_')*;
Number  :       ('0'..'9')+;
Label   :       '"' ('a'..'z' | 'A'..'Z' | '0'..'9'
                | '_' | '-')* '"';

// whitespace tokens (ignored)
WS : (' ' | '\t' | '\n' | '\r')+ { $channel=HIDDEN; };
```

Příloha C

Obsah přiloženého CD

Obsah přiloženého CD je popsán v tabulce C.1.

jméno	typ	popis
index.html	soubor	index stránka CD (relativní odkazy na další dokumenty v adresáři `./html`)
html	adresář	obsahuje soubory dokumentace projektu (javadoc pro *av-core*)
project	adresář	zdrojové kódy všech modulů projektu *ArchVal*
text	adresář	obsahuje text diplomové práce ve formátu *pdf*
dist	adresář	obsahuje distribuční verzi projektu *ArchVal*
samples	adresář	obsahuje příklady, na nichž byl nástroj *ArchVal* testován

Tabulka C.1: Seznam souborů na CD.