

Západočeská univerzita v Plzni  
Fakulta aplikovaných věd  
Katedra informatiky a výpočetní techniky

## **Diplomová práce**

# **Analýza popisů sémantického kontraktu v Java technologiích**

Místo této strany bude  
zadání práce.

# Prohlášení

Prohlašuji, že jsem diplomovou práci vypracoval samostatně a výhradně s použitím citovaných pramenů.

V Plzni dne 22. května 2018

Václav Mareš

# Poděkování

Chtěl bych poděkovat vedoucímu mé diplomové práce Doc. Ing. Přemyslu Bradovi, MSc., Ph.D. za cenné rady a připomínky, které mi pomohly tuto práci dokončit.

## **Abstract**

This master thesis deals with analysis of descriptions of semantic contracts in Java technologies. Main purpose of this diploma is creation of a tool which enables extraction of chosen constructions of design by contract which is part of semantic contracts. To be able to create the tool it is firstly necessary to design model which enables to store representations of various contracts. First part of this thesis is dedicated to theoretical introduction to contracts especially design by contract and then to analysis of programming language Java from the point of grammar and tokenization. Second part contains information about implementation of the tool including the design of model and results of this work.

## **Abstrakt**

Tato diplomová práce se zabývá analýzou popisů sémantického kontraktu v Java technologiích. Hlavní náplní práce je tvorba nástroje, který umožní extrakci vybraných konstrukcí design by contract, které se řadí do kategorie sémantických kontraktů. Aby bylo možné daný nástroj vytvořit je nejprve nutné navrhnout model, který umožní zachytit reprezentaci různých kontraktů. První část práce je věnována teoretickému úvodu do problematiky kontraktů, zejména pak design by contract a následně rozboru jazyka Java z hlediska gramatiky a tokenizace. Druhá část pak obsahuje informace o implementaci daného nástroje, společně s návrhem modelu a dosaženými výsledky.

# Obsah

<b>1</b>	<b>Úvod</b>	<b>1</b>
<b>2</b>	<b>Zajištění kvality software</b>	<b>2</b>
2.1	Kvalitní software . . . . .	2
2.1.1	Vlastnosti určující kvalitu software . . . . .	2
2.2	Aspekty ovlivňující kvalitu software . . . . .	3
2.2.1	Metodika řízení softwarového projektu . . . . .	3
2.2.2	Analýza požadavků . . . . .	4
2.2.3	Návrh systému . . . . .	4
2.2.4	Vývoj . . . . .	4
2.2.5	Testování . . . . .	4
2.3	Preventivní techniky zajištění kvality . . . . .	5
2.3.1	Defenzivní programování . . . . .	5
2.3.2	Doporučení pro psaní bezpečného kódu . . . . .	5
2.3.3	Kontrakty . . . . .	6
<b>3</b>	<b>Popis kontraktů softwarových rozhraní</b>	<b>7</b>
3.1	Koncept kontraktů softwarových modulů . . . . .	7
3.1.1	Úrovně kontraktů . . . . .	7
3.1.2	Vliv na kvalitu kódu a software . . . . .	9
3.2	Design by contract . . . . .	9
3.3	Rozdělení sémantických kontraktů . . . . .	10
3.4	Technologie pro popis sémantických kontraktů v jazyce Java	11
3.4.1	Guava Preconditions . . . . .	11
3.4.2	JSR305 . . . . .	12
3.4.3	Cofaja . . . . .	13
3.4.4	valid4j . . . . .	13
3.4.5	jContractor . . . . .	14
3.5	Popis sémantických kontraktů v jiných programovacích jazycích	15
3.5.1	Code Contracts v .NET . . . . .	15
3.5.2	PhpDeal v PHP . . . . .	15
3.5.3	Boost.Contract v C++ . . . . .	16
3.5.4	Jazyk Eiffel . . . . .	16
3.6	Souhrn technologií pro popis kontraktů . . . . .	17

<b>4</b>	<b>Analýza kódu jazyka Java</b>	<b>19</b>
4.1	O jazyce Java . . . . .	19
4.1.1	Kompilace jazyka Java . . . . .	19
4.2	Gramatika . . . . .	20
4.3	Rozbor kódu . . . . .	20
4.3.1	Lexikální analýza . . . . .	21
4.3.2	Syntaktická analýza . . . . .	21
4.3.3	Nástroje . . . . .	21
4.4	Rozbor přeložených souborů . . . . .	22
4.4.1	Dekompilace . . . . .	22
<b>5</b>	<b>Datový model</b>	<b>25</b>
5.1	Volba DbC konstrukcí . . . . .	25
5.2	Společné znaky reprezentací kontraktů . . . . .	25
5.3	Model pro extrakci kontraktů . . . . .	26
5.3.1	Třída <code>JavaFile</code> . . . . .	27
5.3.2	Třída <code>JavaClass</code> . . . . .	27
5.3.3	Třída <code>JavaMethod</code> . . . . .	28
5.3.4	Třída <code>Contract</code> . . . . .	28
5.4	Model pro porovnávání kontraktů . . . . .	29
5.4.1	Třída <code>JavaFolderCompareReport</code> . . . . .	31
5.4.2	Třída <code>JavaFileCompareReport</code> . . . . .	32
5.4.3	Třída <code>ContractCompareReport</code> . . . . .	33
5.5	Externí reprezentace modelu . . . . .	33
5.5.1	Specifikace formátu . . . . .	34
<b>6</b>	<b>Nástroj pro analýzu kontraktů</b>	<b>36</b>
6.1	Knihovna . . . . .	36
6.1.1	Použité technologie . . . . .	36
6.1.2	Návrh nástroje . . . . .	38
6.1.3	Dekompilace Bytecode . . . . .	38
6.1.4	Parsování Java souborů . . . . .	38
6.1.5	Extrakce kontraktů . . . . .	39
6.1.6	Porovnávání kontraktů . . . . .	40
6.1.7	Popis API . . . . .	40
6.1.8	Přidání parseru pro nový typ kontraktu . . . . .	40
6.1.9	Testování . . . . .	41
6.2	Uživatelská aplikace . . . . .	41
6.2.1	Použité technologie . . . . .	41
6.2.2	Ovládání aplikace . . . . .	42

6.2.3	Možnosti a limitace aplikace . . . . .	42
6.3	Optimalizace . . . . .	42
6.3.1	Analýza a refaktoring kódu . . . . .	42
6.3.2	Zjdenodušení modelu . . . . .	42
<b>7</b>	<b>Testování</b>	<b>43</b>
7.1	Jednotkové testy . . . . .	43
7.2	Funkční testování . . . . .	43
7.2.1	Testovací případy . . . . .	43
7.3	Testovací data . . . . .	45
7.4	Výsledky testů . . . . .	45
<b>8</b>	<b>Zhodnocení výsledků</b>	<b>46</b>
8.1	Úspěšnost detekce kontraktů . . . . .	46
8.2	Úspěšnost porovnání kontraktů . . . . .	46
8.3	Limitace . . . . .	46
8.4	Prostor pro zlepšení . . . . .	46
<b>9</b>	<b>Závěr</b>	<b>47</b>
	<b>Literatura</b>	<b>49</b>
<b>A</b>	<b>Uživatelská příručka</b>	<b>53</b>
<b>B</b>	<b>Obsah CD</b>	<b>54</b>



# 1 Úvod

S rozvojem objektově orientovaného programování se rozmohl trend dělení software do nezávislých komponent, které jsou snadno nahraditelné a lze je vyvíjet takřka nezávisle. Kromě mnoha nepopíratelných výhod této metody jsou zde samozřejmě také potenciální rizika. Jedním z možných rizik může být špatná komunikace těchto samostatných součástí. Sémantické kontrakty jsou jednou z možností, jak snížit chybovost rozhraní komponent a zvýšit jejich přehlednost. Z tohoto důvodu má smysl se těmito konstrukcemi zabývat a analyzovat jejich použití.

Jedním z cílů této diplomové práce je seznámit se s konceptem kontraktu softwarových modulů, zejména pak přístupem Design by Contract (DbC) a prostudovat způsoby popisu DbC kontraktu v Java technologiích. Primárním cílem je návrh a implementace nástroje pro extrakci, případně porovnání, konstrukcí DbC ze zdrojových, respektive přeložených, souborů jazyka Java. Součástí je také analýza a návrh modelu, který bude schopen takto získaná data reprezentovat. Závěrem práce bude ověření správnosti získaných výsledků a jejich souhrn.

Data získaná díky tomuto nástroji budou použita při analýze konstrukcí kontraktů. To může pomoci v otázkách, zda se vyplatí kontrakty používat, jaké druhy jsou oblíbené, jaký dopad má jejich použití na projekt atd.

Po přečtení této práce by měl čtenář získat základní informace o tom, co to jsou kontrakty, jakým způsobem se rozdělují a jaký mají vliv na kvalitu software. Podrobněji by se měl dozvědět o design by contract a různých způsobech jeho reprezentace. Čtenář také bude uveden do problematiky rozboru zdrojových i přeložených souborů jazyka Java a zejména pak s možnostmi extrakce kontraktů z těchto dat. V druhé části práci získá čtenář informace o implementaci daného nástroje a jakým způsobem jsou v něm reprezentována data. Závěrem se dozví podrobnosti o testování a dosažených výsledcích.

## 2 Zajištění kvality software

Jedním z obsáhlých odvětví softwarového inženýrství je zajištění kvality software. Mnoho institucí se touto problematikou zabývá a má velký význam jak pro komerční společnosti, tak pro výzkumné skupiny. V Těto kapitole bude tato problematika stručně nastíněna a budou zde uvedeny různé možnosti zajištění kvality software. Obsah je čerpán zejména z článku Software Development Process and Software Quality Assurance [1].

### 2.1 Kvalitní software

Aby bylo možné se bavit o možnostech zajištění kvality software, je třeba nejprve specifikovat, jaké vlastnosti určují, zda je daný software kvalitní. Klíčovou vlastností je samozřejmě správná funkčnost daného software neboli splnění funkčních požadavků. Mimo to je však na software kladena řada mimo-funkčních požadavků, jako je např. udržitelnost, stabilita, znovupoužitelnost atd. Důležitost dílčích vlastností je u každého projektu jiná a znalost jejich priority by měla být součástí správné analýzy.

#### 2.1.1 Vlastnosti určující kvalitu software

Zde je seznam některých atributů, které určují kvalitu software:

##### **Funkčnost**

Je logické, že software musí splňovat požadovanou funkčnost, jinak by nebyl k prospěchu. V závislosti na typu projektu ale může být vhodné udělat kompromis za účelem zvýhodnění jiných vlastností.

##### **Udržitelnost**

Určuje jak obtížné je provést změny na daném software. Tyto změny mohou být za účelem oprav, přizpůsobení se novým požadavkům, přidání nové funkčnosti atp. Obecně je snahou, aby tyto změny bylo možné provádět s využitím co nejmenšího množství zdrojů.

### **Spolehlivost**

Spolehlivý systém by měl odolávat vnějším vlivům, jako jsou například výpadky či útoky a neměl způsobit škodu při selhání. V důsledku by pak měl být software co nejvíce dostupný.

### **Efektivita**

Software by měl pracovat co nejefektivněji, tedy s co nejmenším využitím zdrojů. Často nás zajímá rychlost a nízké nároky na hardware.

### **Použitelnost**

Kvalitní software by měl umožňovat snadné použití, což typicky bývá spjato s přátelským uživatelským rozhraním, ale může být ovlivněno i náročností instalace či spuštění.

### **Znovupoužitelnost**

Při vývoji by se také mělo myslet na možnost znovu-použití již vytvořených komponent. Jednou vytvořené části se tak dají využít pro jiný projekt či jinou část aplikace, což omezuje duplicitu kódu a v důsledku šetří zdroje.

### **Testovatelnost**

Dobrý software je možné kvalitně otestovat a je známá množina testovacích případů. Díky tomu lze lépe předcházet chybám.

Všechny tyto atributy určují kvalitu software a v závislosti na typu projektu by mělo být cílem každého týmu, dosáhnout co nejlepších výsledků v daných oblastech.

## **2.2 Aspekty ovlivňující kvalitu software**

Po uvedení klíčových vlastností definující kvalitní systém je na místě prozkoumat možnosti zajištění těchto vlastností. Aspektů, které tyto vlastnosti ovlivňují je celá řada a zde je seznam některých z nich:

### **2.2.1 Metodika řízení softwarového projektu**

Volba vhodné metodiky řízení projektu je velmi důležitá, protože ovlivní celý průběh vývoje. Tato volba je závislá na více faktorech jako je povaha a

rozsah projektu, velikost a zkušenosti týmu, který bude na projektu pracovat atd. V dnešní době se obecně dává přednost agilním metodikám jako je např. SCRUM, což platí zejména pro větší projekty.

### **2.2.2 Analýza požadavků**

Analýza a sběr požadavků jsou jedny z prvních činností, které je třeba při tvorbě software provést. Jedná se o důležitý krok, jehož chyby se mohou posléze projevit v celém projektu a typicky mohou vést k vyšším nárokům na zdroje, což se může negativně odrazit na výsledné kvalitě software. Je třeba nalézt všechny aktéry a rozpoznat všechny případy užití. Na základě toho zpracovat funkční i mimo-funkční požadavky, které zákazník očekává a zároveň budou v kompetenci vývojářů. Důležité je také správně stanovit rozsah projektu a určit si hranice.

### **2.2.3 Návrh systému**

Na základě zpracovaných požadavků by měla být provedena analýza, která povede ke tvorbě několika kandidátních architektur, ze kterých by se nakonec měla zvolit architektura, která bude ve výsledku použita. Posléze může začít návrh systému na úrovni komponenty později tříd atd. V tomto kroku je důležité dbát na všechny funkční i mimo-funkční požadavky a vytvořit dostatečně robustní návrh, který se dokáže vyrovnat s menšími změnami.

### **2.2.4 Vývoj**

Během vývoje je vhodné, aby vývojáři dbali na stanovené zásady programování v dané skupině. Cílem je, aby byl kód přehledný i pro ostatní členy týmu a aby byly snazší další potenciální úpravy. S tím souvisí komentování kódu a programování proti rozhraní, což značně zvyšuje znovupoužitelnost. Pro další zvýšení přehlednosti, vyhnutí se potenciálním chybám a zajištění splnění požadavků je také možné využít kontraktů softwarových rozhraní (viz níže).

### **2.2.5 Testování**

Testování je z hlediska kvality důležitým aspektem celého projektu, protože může odhalit řadu chyb, které ji značně snižují. Může se jím předejít pádům systému, chybám ve funkčnosti, problémům s výkonem atd. Pro testování je třeba správná analýza testovacích případů a hraničních hodnot, aby bylo docíleno vysokého pokrytí.

## 2.3 Preventivní techniky zajištění kvality

### TODO:

Pro zajištění vysoké kvality software byla vytvořena řada preventivních metod, které umožňují předcházet chybám, v důsledku čehož vzniká kvalitnější software. V této části budou představeny některé z těchto technik.

### 2.3.1 Defenzivní programování

Defenzivní programování [2] je technika, ve které předpokládáme, že náš program obsahuje chyby a očekáváme nejhorší možný vstup. Myšlenkou tedy je, snažit se odhalit všechny potenciální problémy (jakkoliv drobné) a pokusit se jím předejít. Ve výsledku to pak znamená větší množství ověřování a kontrol než je zvyklé při běžném programování. Snahou programátora by také mělo být napsat co nejprehlednější kód, který umožňuje znovupoužitelnost a snižuje šanci chyb.

V rámci této techniky je důležité upravit své programovací návyky, které mohou vést k chybovosti kódu. Častou chybou může být např. ignorování návratového typu různých metod, které zpravují o její úspěšnosti. Problémy také mohou způsobovat neinicializované proměnné, u kterých nemáme žádnou informaci o tom, jakou obsahují hodnotu. Obecně je také potřeba vždy kontrolovat uživatelské vstupy a důkladně je ověřovat. Tím se můžeme vyhnout např. přístupu do jiné části paměti či práci s nulovými objekty. Pro zvýšení účinnosti kontrol můžeme také použít různé specializované techniky, které umožní lepší kontrolu našeho programu. Příkladem mohou být doporučení pro psaní bezpečného kódu či kontrakty (viz níže).

Defenzivní programování je dobrý způsob, jak zvýšit přehlednost a snížit chybovost kódu, nicméně v extrémních případech můžeme docílit opačného efektu. Důvodem může být např. opakované kontrolování stejných hodnot, předcházení situacím, které nemohou nikdy nastat atd. Tuto techniku je tedy třeba brát s mírou.

### 2.3.2 Doporučení pro psaní bezpečného kódu

např. MISRA

### **2.3.3 Kontrakty**

Jednou z možností, jak zajistit kvalitu software, je také použití kontraktů. Ty reprezentují dohodu o použití dané komponenty, metody, rozhraní mezi vývojářem a uživatelem. Kladou určitá omezení na daný objekt, což snižuje šanci jeho špatného použití. Podrobnosti je možné nalézt v další kapitole, která je celá věnována kontraktům.

# 3 Popis kontraktů softwarových rozhraní

## 3.1 Koncept kontraktů softwarových modulů

Abychom v softwarovém inženýrství zajistili znovupoužitelnost a bezchybnost nezávislých komponent, je třeba specifikovat, jakým způsobem se mají používat a jak s nimi komunikovat. Jedná se o kontrakt mezi tím, kdo komponentu implementoval (dodavatel, vývojář) a tím, kdo ji používá (klient, uživatel). Vývojář zaručuje, že modul bude fungovat dle specifikace, za předpokladu, že bude používán správně. Text této kapitoly čerpá primárně z těchto zdrojů: [3][4][5][6].

V této kapitole bude čtenář kromě konceptu kontraktů také seznámen s vlivem použití na kvalitu kódu a bude zde rozebrán koncept design by contract. Následovat bude rozdělení kontraktů design by contract a příklady nástrojů pro práci s kontrakty pro jazyk Java, ale i jiné technologie.

### 3.1.1 Úrovně kontraktů

Kontrakty je možné dělit do čtyř úrovní, dle toho, jak jsou otevřené diskuzi, kde první úroveň je neměnná a čtvrtá je dynamická a otevřená změnám:

- 1. úroveň - syntaktické
- 2. úroveň - sémantické
- 3. úroveň - interakční
- 4. úroveň - mimo-funkční

#### Syntaktické kontrakty

Základní vrstvou kontraktů jsou kontrakty syntaktické. Jejich znění je neměnné a jedná se o nutnou podmínku pro dodržení dohody mezi vývojářem a uživatelem. Specifikují operace, které může daná komponenta provádět, vstupní a výstupní parametry komponenty a výjimky, které během daných operací mohou nastat. Můžeme tedy říci, že pokrývají signatury a definice rozhraní použitých konstrukcí.

## Sémantické kontrakty

Úroveň sémantických kontraktů specifikuje chování definovaných operací, což umožňuje zabránit jejich chybnému použití a také zvyšuje přehlednost a transparentnost daného rozhraní. Vytyčuje hraniční hodnoty za použití operací *assert*<sup>1</sup> (dále aserce), respektive pomocí definic *pre-conditions* (dále vstupní podmínky), *post-conditions* (dále výstupní podmínky) a *class invariants* (dále neměnné podmínky). Vstupní podmínky kladou požadavky na vstupní argumenty, kontrolují se tedy na začátku operace. Výstupní podmínky specifikují omezení pro výstup operace a jsou tedy vyhodnoceny po dokončení operace. Neměnné podmínky kladou požadavky na vstup i výstup každé veřejné operace v dané třídě. Trojice těchto podmínek využívá aserce a je součástí konceptu design by contract, kterému je věnována část práce níže. Zde je příklad v pseudokódu znázorňující princip sémantických kontraktů:

```
method number example(number x){
    // Vstupní podmínka na parametr x
    require(x > 0, "x has to be a positive number")

    ...

    // Výstupní podmínka na vrácení proměnné x
    ensure(x < 100, "x has to be lesser than 100")

    return x
}
```

V příkladu je znázorněna metoda se vstupem v podobě čísla *x*. Je zde vstupní podmínka, která říká, že *x* musí být větší než 0. Pokud bude tato podmínka porušena, nastane výjimka a vypíše se zpráva "*x has to be a positive number*". Obdobným způsobem funguje i výstupní podmínka, která je vyhodnocena před návratem z metody.

## Interakční kontrakty

Definují chování operací komponenty na úrovni synchronizace. Předchozí úrovně kontraktů považují jednotlivé operace za atomické, což samozřejmě

---

<sup>1</sup>Operace porovnání, která porovná reálnou hodnotu s hodnotou očekávanou. Pokud se tyto hodnoty neshodují nastane výjimka. Tato operace bývá často spojována s testováním.



nemusí být vždy pravda. Tato vrstva specifikuje paralelismy komponenty a s tím spjaté synchronizační prostředky.

### **Mimo-funkční kontrakty**

Tyto kontrakty určují mimo-funkční požadavky na danou komponentu. Typicky se jedná o různé vlastnosti, které zlepšují kvalitu dané služby. Může to být např. doba odezvy, přesnost výsledku apod. (viz Kapitola 2. Zajištění kvality software).

### **3.1.2 Vliv na kvalitu kódu a software**

Použití kontraktů v kódu přináší mnoho výhod, které mohou zvýšit kvalitu vývoje, respektive pak výsledného softwaru. Často vynucují správné chování při statické nebo dynamické kontrole a zajišťují tak správnost toku dat. Poskytují dodatečné informace při popisu rozhraní a pomáhají tak v lepší orientaci v projektu. Při použití kontraktů tak vývojář ví, jaké nároky může mít na danou operaci, a co se na oplátku očekává, že dodrží. Použití kontraktů může také pomoci při debuggingu, či při analýze vstupů a výstupů.

Z využití kontraktů však mohou také plynout určité nevýhody. Jednou z nich je chybné použití kontraktů důsledkem špatné analýzy, které může vést k různým problémům. Kontrakt může být příliš omezující a bránit tak plnému využití funkce, či naopak může být příliš volný a dovolovat nevalidní hodnoty. V závislosti na typu daného kontraktu může také dojít ke zvýšení režie a tedy zpomalení vykonávaného kódu, což by mohl být problém zejména u časově kritických operací. Obecně ale platí, že při zodpovědném používání, mohou být kontrakty velice prospěšné a přispět ke zlepšení kvality vyvíjeného software.

## **3.2 Design by contract**

Pojem design by contract zavedl francouzský profesor Bertrand Meyer [7][8]. První větší zmínka je uveden v publikaci *Design by Contract, Technical Report* v roce 1987. B. Meyer v průběhu let působil na řadě univerzit jako např. v Politecnico di Milano či ETH Zurich a je autorem mnoha publikací a knih. Mimo design by contract, byl jeho významným příspěvkem do oblasti softwarového inženýrství programovací jazyk Eiffel, který je s DbC úzce spjat.

Hlavním cílem design by contract je zvýšení spolehlivosti a správnosti u rozsáhlých softwarových projektů. Principem DbC je zajištění formální dohody mezi vývojářem a uživatelem určitého softwarového modulu. Jak bylo zmíněno výše, DbC je spjato se třemi typy podmínek (vstupní podmínky, výstupní podmínky a neměnné podmínky). O neměnných podmínkách je také možné říci, že se jedná o vstupní a zároveň výstupní podmínky vše veřejných metod v dané třídě. Není nutné aby tyto podmínky platily v průběhu jednotlivých operací.

Podmínky jsou definovány pomocí konstrukcí v kódu programu. V závislosti na typu daného kontraktu, mohou poskytovat statickou kontrolu a/nebo jsou ověřovány při běhu. V případě, že byla některá z nich porušena, je vyvolána výjimka. Tímto chováním je zajištěno, že kontrakt bude dodržen. I přesto, že sémantické kontrakty mohou působit dojmem, že slouží jako náhrada testů, nejedná se o zaměnitelné funkce a naopak by se měly navzájem doplňovat.

### 3.3 Rozdělení sémantických kontraktů

Kontrakty můžeme rozdělit do několika kategorií dle způsobu jejich použití:

- Podmíněné výjimky za běhu (Conditional Runtime Exceptions - CRE)
- API (využití metod knihovny)
- Assert (použití příkazu `assert`)
- Anotace (specifikace kontraktů pomocí anotací)
- Ostatní

**CRE** Nejběžnějším způsobem pro specifikaci kontraktů jsou podmíněné výjimky, které jsou vyvolány za běhu při porušení kontraktu (podmínky). K dispozici jsou různé typy výjimek, které je možné použít, mezi ně patří např. `IllegalStateException`, `IllegalArgumentException`, `NullPointerException`, `IndexOutOfBoundsException` či `UnsupportedOperationException`. Tyto nebo analogické výjimky jsou součástí většiny dnes běžných programovacích jazyků a prostředí, což je jeden z faktorů, proč je tento způsob tak četný.

**API** Další možností implementace kontraktů je využití specializovaného API, které poskytuje metody pro práci s kontrakty. Typicky se jedná o rozšířenou práci s výjimkami, se kterou se navenek pracuje jako se statickými metodami. Tato API zpravidla poskytují širší možnosti a umožňují tak sofistikovanější práci s kontrakty.

**Assert** Použitím klíčového slova `assert` je také možné kontrakty vytvářet. Stejně jako v případě výjimek, i zde se jedná o standardní součást jazyka. Aserce je tvrzení o stavu programu, které vyvolá výjimku není-li dodrženo. Aserce je typicky spojována s tvorbou testů, ale je možné ji použít i pro definici kontraktů.

**Anotace** Dalším způsobem je využití anotací, pomocí kterých je také možné specifikovat kontrakty. Anotace je možné uvádět před specifikací tříd či metod nebo například i před parametry, v závislosti na dané anotaci. Některé anotace pro specifikaci kontraktů poskytují také standardní knihovny Java, nicméně pro pokročilejší funkce je třeba využít externí zdroje.

**Ostatní** Existují také různé specifické způsoby definice kontraktů, které nepatří do žádné z těchto čtyř kategorií, nicméně nejsou příliš časté. Příkladem této kategorie může být `jContractor` (viz níže).

## 3.4 Technologie pro popis sémantických kontraktů v jazyce Java

V Java existuje celá řada nástrojů, které umožňují práci s kontrakty. Liší se v nabízených možnostech a ve způsobech, jak se s nimi pracuje. Některé z nich se již dále nevyvíjejí nicméně jsou stále používány. V tomto oddíle podrobněji rozebereme některé z nich.

### 3.4.1 Guava Preconditions

Knihovna Guava [9] od Google poskytuje řadu nových funkcí jako například různé kolekce, primitiva, práce se souběžnými programy atd. Z hlediska kontraktů je pro nás však zajímavá pouze třída `Preconditions`, která poskytuje metody pro validaci různých stavů. Je zde řada metod, které typicky začínají klíčovým slovem `check*` (např. `checkArgument`, `checkState`, `checkNotNull` atd.). Tyto metody jsou použity běžně v kódu programu a poskytují kontrolu pro vstupní argumenty, jedná se tedy pouze o definice vstupních podmínek,

jak již název napovídá. Při porušení takovéto podmínky je pak vyvolána výjimka při běhu programu. Volitelným parametrem každé metody je také zpráva, která má být při porušení kontraktu zobrazena. Tuto zprávu je pak možné parametrizovat dalšími argumenty.

Guava Preconditions poskytuje dobré prostředí pro práci s kontrakty, avšak její nevýhodou je, že je omezena pouze na vstupní podmínky. Zde je vidět příklad použití Preconditions v kódu:

```
public void guavaPreconditionsExample(Object x){
    String message = "x cannot be null.";
    Preconditions.checkNotNull(x, message);
}
```

V tomto příkladu je vstupní parametr `Object x` omezen a vstupem nemůže být hodnota `null`, pokud se tak stane, je vyhozena standardní výjimka `java.lang.NullPointerException`. Typ výjimky, který knihovna vrací je závislý na dané metodě (např. v případě metody `checkArgument(boolean)` se jedná výjimku `IllegalArgumentException`).

### 3.4.2 JSR305

JSR305 [10] umožňuje specifikaci kontraktů pomocí anotací. Na rozdíl od Guava Preconditions umožňuje také specifikaci výstupních i neměnných podmínek. Obecně platí, že anotace, které jsou uvedeny před argumenty metod např: `@NonNull Object x` specifikují vstupní podmínku pro parametr dané metody. Pokud je anotace uvedena pro celou metodu, jedná se o výstupní podmínku a kontrakt se tak váže na výstupní hodnotu metody. V případě, že je anotace vázaná na třídu, jedná se o neměnnou podmínku. Některé anotace je možné použít jako libovolný druh, nicméně mnoho z nich je specializována pro jeden či dva typy podmínek.

Pokud je kontrakt porušen, je opět vyhozena výjimka, v tomto případě `IllegalArgumentException`. Na rozdíl od Guava, zde není nativní možnost pro zadání vlastní chybové zprávy v případě porušení kontraktu, ale výchozí chyba je poměrně samovysvětlující. JSR305 nicméně neposkytuje pouze striktní podmínky, které při porušení skončí chybou, ale umožňuje také anotace, které slouží jako informace pro vývojáře. Např. anotace `@CheckForNull` upozorňuje, že daný objekt může nabýt hodnoty `null`, ale nevynucuje žádné chování a je pouze na vývojáři, jak s touto informací na-

loží. Příklad zobrazující všechny tři typy kontraktů je vidět zde:

```
@ParametersAreNullableByDefault
public class JSR3053ExampleClass {

    @CheckReturnValue
    public Object JSR305Example(@Nonnull Object x){
        // other code
    }
}
```

JSR značí Java Specification Requests, tedy specifikační požadavky pro Java. Jedná se o popisy finálních specifikací pro jazyk Java. Jednotlivé JSR se postupně schvalují a zhodnocují a jejich průběžný stav je možné sledovat. JSR305 rozšiřuje standardní knihovnu `javax.annotations`. I přesto, že JSR305 je ve stavu *dormant*<sup>2</sup>, stále je hojně využíváno v řadě projektů a má tak smysl jej zkoumat.

### 3.4.3 Cofoja

Contracts for Java [11], či zkráceně Cofoja, je aplikační rámec, který mimo jiné umožňuje práci s kontrakty. Kontrakty jsou definovány na úrovni anotace a slouží pouze ke kontrole za běhu programu, neposkytují tedy statickou kontrolu. Cofoja umožňuje použití všech tří typů podmínek a zajišťuje to pomocí klíčových slov `@Requires` pro vstupní podmínky, `@Ensures` pro výstupní podmínky a `@Invariant` pro neměnné podmínky. Praktické využití v kódu pak může vypadat takto:

```
@Requires("x >= 0")
@Ensures("result >= 0")
static double sqrt(double x);
```

### 3.4.4 valid4j

Valid4j [12] je jednoduchý nástroj, který poskytuje metody pro práci s kontrakty za pomoci aserce. Podobně jako jiné nástroje využívá klíčových slov `require` pro vstupní a `ensure` pro výstupní podmínky. Tento nástroj nepo-

---

<sup>2</sup>*Dormant* značí, že práce na tomto JSR projektu byla pozastavena. Může to být na základě hlasování komise, či protože dané JSR dosáhlo konce své životnosti.

skytuje podporu pro neměnné podmínky za pomoci specializovaných metod ale tohoto chování je možné dosáhnout použitím zmíněných metod. Zde je část kódu implementující kontrakty nástrojem `valid4j`.

```
public Stuff method(Object param) {
    require(param, notNullValue());
    require(getState(), equalTo(GOOD));
    ...
    ensure(getState(), equalTo(GREAT));
    return ensure(r, notNullValue());
}
```

### 3.4.5 jContractor

Posledním příkladem je `jContractor` [13]. Z hlediska definice kontraktů se jedná se o unikátní nástroj, který nepatří do žádné ze zmíněných skupin. Kontrakty jsou zde definovány tvorbou metod s danou jmennou konvencí. To znamená, že pokud chceme vytvořit vstupní podmínky pro metodu `push`, musíme definovat metodu `push_Precondition`. Obdobně to platí i pro výstupní podmínky s příponou `_Postcondition` a neměnné podmínky s příponou `_Invariant`. Jedná se o metody, jejichž návratovým typem je `boolean`, který určuje, zda byla podmínka splněna, či nikoliv. Podmínky jsou pak do kódu přidány při generování *bytecode*<sup>3</sup> (dále bajtkód) a zajišťují tak kontrolu pouze při běhu programu. Zde je krátká ukázka kódu demonstrující použití nástroje `jContractor`:

```
// metoda push s vlastním kódem
public void push (Object o) {
    ...
}

// metoda definující vstupní podmínku pro push
protected boolean push_Precondition (Object o) {
    return o != null;
}
```

---

<sup>3</sup>Programy psané v jazyce Java nejsou překládány přímo do strojového kódu, jak je typické, ale jsou překládány do tzv. bajtkódu, což umožňuje platformní nezávislost. Více informací je popsáno níže, ve 4. kapitole Tokenizace jazyka Java

## 3.5 Popis sémantických kontraktů v jiných programovacích jazycích

Použití kontraktů samozřejmě není omezeno pouze na Java, ale je rozšířeno do mnoha jiných jazyků. Mimo použití běžně dostupných prostředků, jako jsou například výjimky či aserce, které jsou k dispozici téměř v každém jazyce, existují také specializované nástroje, které umožňují rozšířenou práci s kontrakty. Následuje krátký výčet některých z nich.

### 3.5.1 Code Contracts v .NET

Prvním příkladem může být projekt Code Contracts [14][15], který byl vyvinut společností Microsoft a umožňuje použití kontraktů v .NET jazycích. Jedná se o open-source knihovnu, která formou API poskytuje funkce pro specifikaci kontraktů. Funguje na jednoduchém princip volání funkcí podobně jako Guava Preconditions, nicméně umožňuje použití všech tří typů podmínek. Základními funkcemi jsou `Contract.Requires()` pro definici vstupních podmínek, `Contract.Ensures()` pro zajištění správného výstupu a `Contract.Invariant()` pro reprezentaci neměnných podmínek. Mimo těchto základních funkcí poskytuje také různé specifické operace, se kterými je možné vytvářet komplexnější kontrakty. Umožňuje také např. specifikaci vyhozené výjimky. V následujícím příkladu je vidět použití základních konstrukcí:

```
Contract.Requires( x != null );  
Contract.Ensures( this.F > 0 );  
Contract.Invariant(this.y >= 0);
```

### 3.5.2 PhpDeal v PHP

**TODO: jak je vidět z příkladu, není to pravda, používá klíčová slova v dokumentačních komentářích - jsou tyto kontrakty nějak kontrolovány při běhu?**

Jedním z nástrojů, které umožňují reprezentaci kontraktů pro jazyk PHP je aplikační rámec PhpDeal [16]. Podobně jako jiné nástroje, pracuje na principu **anotací**. Při definici vstupních a výstupních podmínek jsou anotace uvedeny u dané funkce, neměnné podmínky jsou pak definovány jako anotace třídy. Anotace jsou definovány klíčovým slovem `@Contract`, kde za zpětným lomítkem následuje `Verify` pro vstupní podmínku, `Ensure` pro

výstupní podmínku a **Invariant** pro neměnnou podmínku. V závorce pak následuje řetězec s podmínkou, která je ověřována. Nástroj také umožňuje integraci do IDE pro zvýraznění syntaxe. Použití je vidět na následujícím příkladu:

```
/** @Contract\Verify("$amount>0 && is_numeric($amount)")
 * @Contract\Ensure("$this->bal == $_old->bal+$amount")
 */
public function deposit($amount)
{ ... }
```

### 3.5.3 Boost.Contract v C++

Knihovna Boost.Contract [17] poskytuje podporu pro design by contract v jazyce C++. Pracuje na principu API a poskytuje tak funkce pro realizaci podmínek. Funkce se nazývají **precondition** a **postcondition**. Pro realizaci neměnných proměnných slouží definice funkce jménem **invariant**, která pak zajišťuje kontrolu všech vstupů a výstupů. Pro aserci je používána funkce **BOOST\_CONTRACT\_ASSERT**.

### 3.5.4 Jazyk Eiffel

Některé programovací jazyky konstrukce DbC podporují nativně a není tak třeba používat žádné dodatečné nástroje. Jedním z předních zástupců této kategorie je jazyk Eiffel [18], který byl vyvíjen s cílem zajistit co největší spolehlivost v rámci programovacího jazyka. Jak již bylo zmíněno výše, byl navržen Bertrandem Meyerem, tedy stejným člověkem, který představil design by contract. Jazyk poskytuje mnoho zajímavých konstrukcí, zde bude však nastíněno použití konstrukcí DbC.

Definice je zajištěna klíčovými slovy **require** pro vstupní podmínky, **ensure** pro výstupní podmínky a **invariant** pro neměnné podmínky. Vzhledem k povaze jazyka jsou vstupní a výstupní podmínky začleněny přímo do těla funkce, které jsou zde však nazývány **feature**. Neměnné podmínky jsou pak samostatně definovány v bloku **invariant**. Eiffel poskytuje různé rozšířené možnosti pro práci s kontrakty, základní principy by měly být patrné z následujícího příkladu:



```

feature
  deposit (sum: INTEGER)
    require
      non_negative: sum >= 0
    do
      ...
    ensure
      updated: bal = old bal + sum

```

### 3.6 Souhrn technologií pro popis kontraktů

Výčet nástrojů výše je pouze omezený výběr příkladů, nikoliv kompletní seznam. To poukazuje na skutečnost, že nástrojů, které poskytují možnost práce se sémantickými kontrakty, existuje mnoho. Často se jedná o knihovny, které nabízejí širší možnosti použití než samotné kontrakty. V tabulce 3.1 je vidět souhrn uvedených nástrojů.

Název	Jazyk	Typ	Dostupné podmínky		
			Vstup.	Výstup.	Neměn.
Guava Preconditions	Java	API	ANO	NE	NE
JSR305	Java	Anotace	ANO	ANO	ANO
Cofaja	Java	Anotace	ANO	ANO	ANO
valid4j	Java	API	ANO	ANO	NE*
jContractor	Java	Speciální	ANO	ANO	ANO
Code Contracts	.NET	API	ANO	ANO	ANO
PhpDeal	PHP	Anotace	ANO	ANO	ANO
Boost.Contract	C++	API	ANO	ANO	ANO

\* Neměnné podmínky je možné vytvořit pomocí vstupních a výstupních podmínek.

Tabulka 3.1: Souhrn nástrojů pro práci s kontrakty

Co se základní funkcionality týče, až na výjimky nástroje poskytují srovnatelné možnosti. Typově se pak obvykle jedná o nástroje, které zprostředkávají metody pomocí API či o anotace. Obecně se dá říci, že volba nástroje záleží spíše na preferencích skupiny, která bude daný nástroj používat. Je možné se rozhodovat podle toho, jaký typ reprezentace daný nástroj používá, či jak je technologie známá a oblíbená. Z tohoto důvodu má smysl zabývat se problematikou, jak zajistit unifikaci specifikací kontraktů z různých nástrojů a poskytnout tak jednotnou reprezentaci, která umožní další analýzu.

TODO: bylo by vhodné doplnit info o tom, jak živé jsou jednotlivé projekty a jak hodně se používají (jde vyčíst z článku Dietrich a kol.) co není v Dietrich napsat neuvedeno nebo zkusit nalézt a napsat nějak obecně používá se málo/hodně...

# 4 Analýza kódu jazyka Java

## 4.1 O jazyce Java

Java [19] je objektově orientovaný programovací jazyk, který byl vytvořen více než před 20 lety. Java byla inspirována řadou jazyků, jako je např. Eiffel, SmallTalk a Objective C. Snahou bylo vytvořit objektově orientovaný jazyk, který bude jednoduchý na pochopení, aniž by bylo třeba dlouhého tréninku. Jedním ze značných usnadnění, např. oproti jazyku C/C++, je Garbage Collector. Ten umožňuje automatické uvolňování paměti, což předchází častým chybám spojených s jejím manuálním uvolňováním. Cílem také bylo, aby byla Java robustní a zabezpečená a jednalo se tak o spolehlivý jazyk. Java byla vytvářena za účelem architektonické a platformní nezávislosti a bylo ji tak možné použít takřka všude. Této přenositelnosti bylo zajištěno především tím, že se jedná o interpretovaný jazyk (viz níže). Při vývoji bylo také samozřejmě cíleno na zajištění co největší výkonnosti s čímž souvisí i umožnění tvorby vícevláknových aplikací.

Java má řadu výhod, ale také nevýhod a stejně jako u každé jiné technologie je třeba zvážit, zda se jedná o vhodnou volbu pro náš projekt. Jazyk Java je dlouhodobě jedním z nejpoužívanějších programovacích jazyků a stále se vyvíjí [20].

Tato kapitola bude čtenáře informovat o základních vlastnostech jazyka Java, především pak o jeho překladu do bajtkódu. Jejím hlavním cílem je představení gramatiky tohoto jazyka a úvod do problematiky tokenizace, která v důsledku umožní extrakci konstrukcí DbC. Dalším tématem budou také možnosti dekompilace bajtkódu a rozbor toho, jak se výsledek liší oproti zdrojovému kódu.

### 4.1.1 Kompilace jazyka Java

Jazyk Java je tzv. interpretovaný jazyk, což znamená, že programovací kód není překládán přímo do strojového kódu daného zařízení, ale je přeložen do bajtkódu. Bajtkód je speciální vysoce-úrovňový kód, který je platformově nezávislý. V rámci kompilace Java to znamená, že zdrojové soubory `.java` jsou překompilovány do souborů `.class`. Výsledný Java program je pak distribuován ve formátu `.jar` nebo `.war`, což jsou prakticky archivy obsahující

přeložené soubory. Na cílovém zařízení je pak program spouštěn pomocí *Java Virtual Machine* (Virtuální stroj jazyka Java, dále JVM).

## Java Virtual Machine (JVM)

JVM je softwarová abstrakce pro obecnou hardwarovou platformu. Slouží k tomu, aby bylo možné spustit program napsán v Java na různých zařízeních. Program je pak virtuálně spuštěn na JVM místo přímo na daném zařízení. Vzhledem k tomu, že se jedná o virtualizaci, je logické, že z hlediska výkonu a nároků na paměť není možné dosáhnout srovnatelných výsledků s programem, který je na danou platformu plně zoptimalizován. V tomto případě se jedná o daň za multiplatformnost jazyka Java.

Když spustíme Java aplikaci, nejprve se spustí JVM na daném zařízení. Ten načte hlavní třídu s metodou `main` spolu s jinými klíčovými částmi jako je např. `java.lang.Object`. K načtení těchto tříd se využívá tzv. *Class Loader*. Poté, co jsou načteny klíčové části již JVM načítá třídy pouze na vyžádání, což se děje ve chvíli, kdy je daná třída v programu potřeba. Tímto způsobem obsahuje JVM jen ten přeložený kód, který je v danou chvíli potřeba, což snižuje nároky na paměť, ale snižuje rychlost programu. Tato technika se nazývá *Just-in-time kompilace* [22][23].

**TODO: JIT je něco jiného**

## 4.2 Gramatika

**TODO: reprezentace některých konstrukcí, vč. anotací - do jakých základních konstrukcí jsou převáděny, jak se poznají v bytecode; retention policy u anotací.**

## 4.3 Rozbor kódu

Abychom mohli zpracovávat zdrojový kód jazyka Java, je vhodné jej nejprve převést na snadněji zpracovatelnou formu než surový text. K tomu nám může pomoci lexikální a syntaktická analýza, pomocí níž můžeme ve výsledku získat reprezentaci ve stromové podobě [21].

### 4.3.1 Lexikální analýza

**TODO: toto je hodně volný a nepřesný popis lex a syn analýzy - nutno zpřesnit, viz aspoň FJP!**

Aby bylo možné provést rozbor kódu, který je prezentován v textové podobě, je třeba nejprve provést lexikální analýzu. Ta spočívá v tom, že je proud znaků zpracováván a rozdělován dle bílých znaků<sup>1</sup> nebo operačních, respektive speciálních, symbolů (=, :, &, ...) na tzv. lexémy, které reprezentují **dílčí části kódu**. Na základě sekvenční znaků analyzátor pomocí regulárních výrazů určí, zda se jedná o identifikátor, číslo, klíčové slovo, řetězec atd. Pokud analyzátor narazí na nesrovnalost, např. v Java by začal lexém číslem a pokračoval písmeny, což by představovalo ilegální pojmenování identifikátoru, analýza vyhodí chybu. Tímto vznikne seznam lexémů, který je předán dále syntaktické analýze.

### 4.3.2 Syntaktická analýza

Ze vstupního proudu lexému vytvořeného pomocí lexikální analýzy můžeme nyní za pomoci syntaktické analýzy vytvořit derivační strom. Díky této reprezentaci **můžeme již dané lexémy rozdělit do různých konstrukcí jazyka**, jako jsou třídy, metody, podmínky atd. Po tomto kroku jsme již tedy schopni zpracovávat zdrojový kód na takové úrovni, která nám umožní procházet anotace jednotlivých konstrukcí, či příkazy těla metod. Na základě toho jsme tedy schopni provádět analýzu, která umožní nalezení kontraktů ve zdrojovém kódu.

### 4.3.3 Nástroje

I přesto, že by bylo možné provádět zmíněné analýzy pomocí vlastního kódu, je také možné použít některý z dostupných nástrojů. Tyto nástroje umožňují parsování zdrojového kódu za použití samostatné aplikace nebo se může jednat o knihovnu, jejíž API můžeme použít přímo v našem kódu.

**JFLex** JFLex [24] je lexikální analyzátor pro jazyk Java. Protože lexikální analýza sama o sobě nestačí k analýze kódu vhodné pro detekci kontraktů, je třeba tento nástroj použít s jiným, který umožní syntaktickou analýzu. Může se jednat např. o CUP, BYacc/J či ANTLR (viz níže).

---

<sup>1</sup>Bílými znaky jsou myšleny netisknutelné znaky jako jsou například mezery, tabulátory, nové řádky atd.

**CUP** TODO

**BYacc/J** TODO

**ANTLR** ANTLR [25] je široce používaný nástroj, který umožňuje na základě zadané gramatiky vytvořit a procházet strom daného zdrojového kódu. ANTLR je primárně určen ke zpracování vlastní gramatiky např. při tvorbě vlastního překladače, nicméně je možné také použít gramatiku jazyka Java pro rozbor jejího zdrojového kódu.

**JavaParser** JavaParser [26] je jedním z dostupných nástrojů, který umožňuje analýzu, parsování ale i konstruování kódu jazyka Java. Funguje jako knihovna, kterou můžeme použít v našem kódu. Jedná se o vyspělou technologii, která je použita v řadě projektů.

**Roaster** Roaster [27] je nástroj podobný JavaParser. Také umožňuje nejen parsování, ale i editaci zdrojových kódů jazyka Java. Mimo dostupného API umožňuje také práci s konzolí pro jednoduché použití.

## 4.4 Rozbor přeložených souborů

Při analýze kontraktů v jiných projektech není vždy možné získat zdrojové soubory. Z toho důvodu je užitečné umět analyzovat nejen soubory `.java`, ale také přeložené soubory `.class`.

### 4.4.1 Dekompilace

Aby bylo možné soubory analyzovat je nutné je nejprve dekompileovat. Dekompilace je vlastně opačný proces kompilace a je třeba nahradit prvky, které byly přidány do bajtkódu těmi, které zde byly původně. **TODO: Zlepšit formulaci, více podrobností**

#### Nástroje

Nástrojů pro dekompilaci souborů v jazyce Java je celá řada. Některé je nutné pouštět samostatně, ty pak mohou být vytvořeny i v jiných programovacích jazycích. Vývojová prostředí jako např. Eclipse či IDEA IntelliJ umožňují instalaci balíčků, které podporují dekompilaci souborů. Některé z

nástrojů je také možné použít přímo v kódu formou knihovny. Následuje seznam některých dostupných nástrojů, které jsou stále aktualizovány a umožňují dekompilaci moderních prvků jazyka Java [28][29][30].

**TODO: chybí podrobnější informace, do jaké reprezentace je bytecode daným nástrojem převeden a jestli/jak se s ní dá programově pracovat (tj. doménové a funkční API);, vč. příkladů; dále jaká jsou omezení či jiné pozoruhodné vlastnosti nástrojů**

**JD Project** JD Project (Java Decompiler) [31] je jedním z nejčastěji referovaných nástrojů v rámci Java dekompilace. JD disponuje samostatnou aplikací s grafickým uživatelským rozhraním, pomocí kterého je možné soubory dekompileovat a ihned vidět výsledek. JD také poskytuje doplněk do Eclipse a IDEA IntelliJ. Nejedná se o open-source nástroj, ale je dostupný zdroj pro GUI aplikaci.

**Procyon** Procyon [32] je open-source nástroj, který také umožňuje dekompilaci. Nemá samostatné GUI jako JD, nicméně je možné reprezentaci zobrazit použitím externích nástrojů. Disponuje také API, které je možné použít přímo v kódu.

**CFR** CFR [33] je dalším z nástrojů, který umožňuje dekompilaci i konstrukcí Java 1.9. Nástroj je možné použít pouze v rámci příkazové řádky.

Nástrojů pro dekompilaci jazyka Java je velké množství a jejich výběr není snadný zejména proto, že se neustále vyvíjejí na základě nových verzí Java. Při výběru je samozřejmě také důležité, jakým způsobem je daný nástroj možné používat, jaké konstrukce dokáže dekompileovat a roli může hrát také rychlost.

## Rozdíly oproti původním souborům

Jak již bylo zmíněno, velice závisí na použitém nástroji pro dekompilaci. Jestliže daný nástroj nepodporuje určité konstrukce, pak je logické že jejich dekompilace nebude možná a není tak možné získat původní zdrojový soubor. Pokud nástroj umožňuje dekompilaci všech použitých konstrukcí, výsledný kód stále nebude zcela shodný. Přeložený kód obsahuje plné cesty k objektům, doplňuje neuvedená přetypování atd. Tyto informace často ve zdrojových kódech nebývají uvedené, protože jsou redundantní, ale při rekonstrukci přeloženého souboru není možné určit, kde tyto informace byly a

kde ne. Přirozeně také není možné získat data, která se do bajtkódu neza-  
nášejí, jako jsou např. komentáře.



## 5 Datový model

Tato kapitola se zabývá podrobnostmi o datovém modelu. Pojednává o volbě reprezentace kontraktů, které byly použity pro tento projekt a obecně o společných attributech kontraktů. Jsou zde uvedeny podrobnosti o datovém modelu určeném pro reprezentaci kontraktů, ale také pro výsledky jejich porovnání. Oba tyto modely jsou zde popsány slovně, ale i pomocí UML diagramu. Závěr tvoří informace o externí reprezentaci těchto dat.

### 5.1 Volba DbC konstrukcí

Po analýze dostupných materiálů jsem se rozhodl zvolit pro implementaci konstrukce Guava Preconditions a JSR305. Důvodem byla především jejich rozdílná reprezentace, kdy Guava Preconditions je realizováno pomocí volání metod uvnitř těl metod a umožňuje vytvářet vstupní podmínky. Na druhé straně JSR305 je tvořené anotacemi v záhlaví tříd, metod a také jako součást parametrů metod. Umožňuje tvoření všech tří typů kontraktů (vstupní, výstupní i neměnné podmínky). Kromě této diverzity se také jedná o jedny z častých konstrukcí používaných v projektech viz [3]. Principy obou těchto nástrojů byly již popsány výše (viz 3. kapitola Popis kontraktů softwarových rozhraní), z čehož se bude vycházet.

### 5.2 Společné znaky reprezentací kontraktů

Při rozboru jednotlivých nástrojů pro reprezentaci design by contract zjistíme, že sdílejí mnoho podobných aspektů, které jsou klíčové pro vytvoření obecného modelu, který je schopen zachytit libovolnou konstrukci kontraktu.

Do modelu je třeba nejprve zanést, o jaký nástroj pro práci s kontrakty se jedná (Guava, JSR305, atd.). Každý kontrakt je také vždy definován jedním ze tří typů podmínek (vstupní, výstupní a neměnná), to je také velmi důležitá informace, kterou je třeba do modelu zaznamenat. Kontrakty jsou také typicky definovány funkcí, která určuje, co kontrakt ověřuje. Může se jednat o kontrolu argumentu, zda objekt není `null` atp. V případě, že kontrakt tuto informaci neobsahuje, může tato položka zůstat prázdná. Další důležitou informací je hodnota daného kontraktu, která představuje výraz, který je vyhodnocován. Některé kontrakty tuto hodnotu nevyžadují (např.

@NonNull u JSR305), ale ve většině případů jde o podmínku (např.  $x > 0$ ). Některé druhy reprezentací kontraktů však umožňují i další argumenty. Typicky se jedná o informace o typu výjimky, která je vyhozena, či o její zprávě. Může se však také jednat o další upřesňující data. Obvykle tyto informace nemají z hlediska důležitosti vysokou prioritu, přesto by ale mělo být možné je do modelu zahrnout.

## 5.3 Model pro extrakci kontraktů

Aby bylo možné kontrakty extrahovat ze zdrojových souborů, je také třeba mít k dispozici datový model, do kterého bude možné tato data uložit. Tento model jsem vytvořil na základě analýzy konstrukcí kontraktů s ohledem na následný export do formátu dat, který bude možné dále zpracovávat. Aby byl zachován kontext kontraktů, usoudil jsem, že bude třeba zachovávat také informace o třídách a metodách v daném souboru. Rozhodl jsem se tedy vytvořit strukturu podobnou stromu, jejíž kořenem je samotný zdrojový soubor. Tento soubor obsahuje různé podrobnosti o tomto souboru jako je jeho jméno a cesta, typ a statistiky o jeho obsahu. Také obsahuje seznam všech tříd obsažených v tomto souboru.

Každá jednotlivá třída pak obsahuje jméno, svou hlavičku, seznam metod a také seznam všech kontraktů týkajících se této třídy, tedy neměnných podmínek. Metoda pak nese informaci o své signatuře a také seznam všech kontraktů této metody. Samotný kontrakt pak obsahuje informaci o tom, o jaký typ kontraktu a podmínky se jedná, kompletní výraz a také jeho dílčí části. Následuje podrobnější rozbor tohoto modelu rozdělením na jednotlivé entity.

Na obrázku 5.1 je vidět grafické znázornění datového modelu formou UML diagramu. Nejsou zde záměrně zobrazeny objekty `ContractComparison` a `JavaFileCompareReport`. Těm je věnován prostor v diagramu pro porovnávání kontraktů, který je na obrázku 5.2.

**TODO: hlavní je model-diagram, popis jednotlivých metod je spíš nežádoucí (zbytečné podrobnosti, umělé nafukování textu práce) - omezit jen na klíčové příp. nesamovysvětlující metody/atributy.**

### 5.3.1 Třída `JavaFile`

Instance této třídy představují samotné soubory. Zde je jeho kompletní struktura:

**`String fileName`**

Jméno daného souboru bez cesty a bez koncovky.

**`String fullPath`**

Tento atribut obsahuje absolutní cestu k analyzovanému souboru.

**`String shortPath`**

Tato zkrácená cesta je použita při zpracování více souborů. Je pak uchována pouze rozdílná část těchto cest. To umožňuje přehlednější zobrazení v uživatelské aplikaci. Příkladem mohou být soubory, které mají tyto absolutní cesty: `C:/files/test/base/alpha.java` a `C:/files/test/common/beta.java`. Zkrácené cesty pak vypadají takto: `base/alpha.java` a `common/beta.java`.

**`FileType fileType`**

Atribut `fileType` je výčtový typ, který představuje typ daného souboru. Může se jednat o `JAVA` pro soubory `.java` a `CLASS` pro soubory `.class`.

**`JavaFileStatistics javaFileStatistics`**

Tento objekt obsahuje statistické údaje o daném souboru. Je zde uložen počet tříd a metod. Také zde je počet metod s kontrakty a také celkový i dílčí počet všech kontraktů.

**`List<JavaClass> javaClasses`**

Zde, v seznamu objektů `JavaClass` (viz níže), jsou uloženy jednotlivé třídy, které jsou v daném souboru obsaženy.

### 5.3.2 Třída `JavaClass`

Instance `JavaClass` představují jednotlivé třídy v daném souboru.

**`String name`**

Obsahuje jméno dané třídy.

**String signature**

Představuje signaturu dané třídy a mimo názvu obsahuje také ostatní podrobnosti.

**List<JavaMethod> javaMethods**

Seznam objektů `JavaMethod` (viz níže), které reprezentují metody této třídy.

**List<Contract> invariants**

Seznam objektů typu `Contract` (viz níže), které představují neměnné podmínky této třídy.

### 5.3.3 Třída `JavaMethod`

Objekty typu `JavaMethod` reprezentují metody v nadřazené třídě.

**String signature**

Signatura této metody, která mimo jiné obsahuje návratový typ, název a vstupní parametry.

**boolean isConstructor**

Určuje, zda daná metoda je konstruktor, či se jedná o běžnou metodu.

**List<Contract> contracts**

Seznam objektů typu `Contract` (viz níže), které představují dílčí kontrakty dané metody. Jsou definovány vstupními i výstupními podmínkami.

### 5.3.4 Třída `Contract`

**ContractType contractType**

Jedná se o výčetový typ, který určuje o jaký druh kontraktu se jedná. Při současném stavu knihovny to mohou být hodnoty Guava či JSR305.

**ConditionType conditionType**

Opět výčetový typ který určuje typ kontraktu dle jeho podmínky. Rozlišují se tři druhy: `PRE` pro vstupní podmínku, `POST` pro výstupní podmínku a `INVARIANT` pro neměnnou podmínku.

### **String completeExpression**

Reprezentuje kompletní výraz celého kontraktu. I přesto, že celý výraz je možné vytvořit z jeho dílčích částí, je zde uveden pro rychlý přehled. Může také posloužit jako kontrola parsování či pro rychlé porovnání.

### **String function**

Tento řetězec určuje funkci o jakou se jedná v rámci daného kontraktu. V případě Guava se jedná o název metody, v případě JSR305 o název anotace. Obecně se jedná o hlavní označení určující daný kontrakt.

### **String expression**

Obsahuje první parametr dané funkce. Důvodem, proč oddělit první parametr od ostatních, bylo, že kontrakty mají často pouze jeden parametr a pokud jich mají více, ostatní často nejsou tolik relevantní. Pro zvýšení přehlednosti byl tedy tento parametr uveden samostatně.

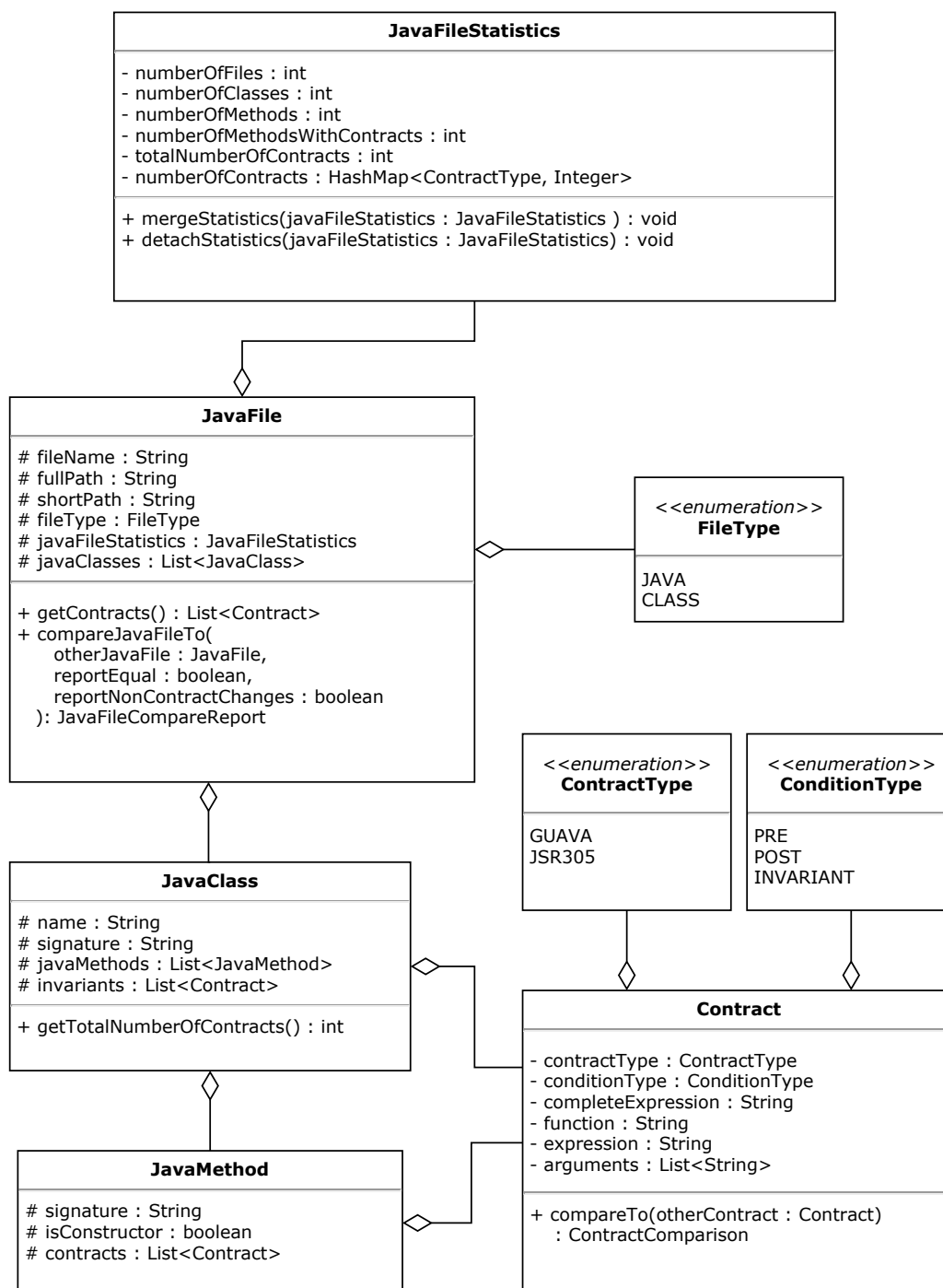
### **List<String> arguments**

Seznam ostatních argumentů daného kontraktu. Ostatní atributy až na výjimky slouží pouze k uvedení chybové zprávy, která se má zobrazit při porušení kontraktu. Mimo zprávy zde také bývají proměnné použité ve zprávě.

Během zpracování jsou používány následující třídy: `ExtendedJavaFile`, `ExtendedJavaClass` a `ExtendedJavaMethod`. Ty obsahují dodatečné informace vůči výše zmíněným objektům. Jedná se o anotace, vstupní parametry a jednotlivé části těl metod. Po zpracování jsou pak tyto objekty redukovány na ty výše zmíněné, které jsou připraveny pro externí reprezentaci.

## **5.4 Model pro porovnávání kontraktů**

Vzhledem k tomu, že nástroj umožňuje kromě extrakce kontraktů také jejich porovnání, je třeba vytvořit datový model, který umožní zachytit i tyto informace. Na základě toho, že porovnání bude mít největší hodnotu ve chvíli, kdy bude možné porovnat dva projekty, tedy dvě složky s mnoha zdrojovými, respektive přeloženými, soubory jazyka Java, usoudil jsem, že bude jeden hlavní objekt, který bude reprezentovat porovnání dvou složek. Ten bude obsahovat různé podrobnosti o tom, zda byly složky shodné, konkrétní



Obrázek 5.1: UML diagram datového modelu pro extrakci kontraktů

případy, kde se lišily a také seznam souborů, ke kterým nebyl nalezen adekvátní pár v druhé složce. Jedná se tedy o přidané a odebrané soubory. Stejně jako v případě extrakce, i zde by měl být souhrn statistik, který poskytne dodatečné informace o tomto srovnání.

**TODO: je třeba vysvětlení typů rozdílů, algoritmu jak se určí**

Porovnání jednotlivých souborů budou pak zachycena v objektu, který bude mít podobné atributy jako ten pro složku. Tedy v čem se dané soubory lišili z hlediska tříd, metod a zejména kontraktů. I zde je k dispozici souhrn statistik. Celý model by měl být patrný z následujícího popisu, případně pak z obrázku 5.2, kde je tento model znázorněn graficky formou UML diagramu.

### 5.4.1 Třída `JavaFolderCompareReport`

Instance třídy `JavaFolderCompareReport` představují zprávy o porovnání dvou složek.

**`String thisFolderPath`**

Cesta k první složce, která je porovnávána.

**`String otherFolderPath`**

Cesta k druhé složce, která je porovnávána.

**`boolean apiEqual`**

Obsahuje informaci o tom, zda jsou složky shodné na úrovni API. K tomu se vztahuje, zda jsou shodné soubory, třídy a metody na úrovni definic (nikoliv jejich těl).

**`boolean contractEqual`**

Říká zda jsou složky shodné na úrovni kontraktů. Jedná se o to, zda jsou nějaké kontrakty přidány či odebrány nebo jakkoliv změněny.

**`JavaFolderCompareStatistics javaFolderCompareStatistics`**

Tento objekt obsahuje statistické údaje k danému porovnání. Je zde počet přidanych a odebranych souborů a také počet přidanych/odebranych/změněných kontraktů.

**`List<String> filesAdded`**

Seznam cest souborů, které byly přidány.

**List<String> filesRemoved**

Seznam cest souborů, které byly odebrány.

**List<JavaFileCompareReport> javaFileCompareReports**

Jedná se o seznam porovnávacích zpráv jednotlivých souborů (viz níže).

### 5.4.2 Třída JavaFileCompareReport

Tyto objekty reprezentují zprávy o porovnání dvou souborů.

**String thisFilePath**

Cesta k prvnímu souboru, který je porovnáván.

**String otherFilePath**

Cesta k druhému souboru, který je porovnáván.

**boolean apiEqual**

Obsahuje informaci o tom, zda jsou soubory shodné na úrovni API. K tomu se vztahuje, zda jsou stejné deklarace tříd a metod.

**boolean contractEqual**

Říká zda jsou složky shodné na úrovni kontraktů. Jedná se o to, zda jsou nějaké kontrakty přidány či odebrány nebo jakkoliv změněny.

**JavaFileCompareStatistics javaFileCompareStatistics**

Statistiky porovnání daných dvou souborů. Obsahuje informace o počtu přidáných a odebraných tříd, metod a kontraktů.

**List<ApiChange> apiChanges**

Seznam všech změn API, které byly mezi soubory detekovány. Obsahují informace o tom jaká třída či metoda byla přidána/odebrána a kolik kontraktů to ovlivnilo.

**List<ContractCompareReport> contractCompareReports**

Seznam všech zpráv o porovnání jednotlivých kontraktů (viz níže).



### 5.4.3 Třída `ContractCompareReport`

Jednotlivé objekty `ContractCompareReport` představují zprávy o porovnání dvou kontraktů.

#### **`ContractComparison contractComparison`**

Tento výčtový typ určuje, jestli byly kontrakty shodné, rozdílné či se jednalo o menší změny.

#### **`ApiState apiState`**

Určuje zda byl nalezen párový kontrakt v druhém souboru, či jestli byl přidán nebo naopak odebrán.

#### **`ContractType contractType`**

O jaký typ kontraktu se jedná (Guava, JSR305, atd.).

#### **`String className`**

Označení třídy, ve které se oba kontrakty nacházejí.

#### **`String methodName`**

Označení metody, ve které se oba kontrakty nacházejí.

#### **`String thisContractExpression`**

Kompletní výraz, který reprezentuje první porovnávaný kontrakt.

#### **`String otherContractExpression`**

Kompletní výraz, který reprezentuje druhý porovnávaný kontrakt.

## 5.5 Externí reprezentace modelu

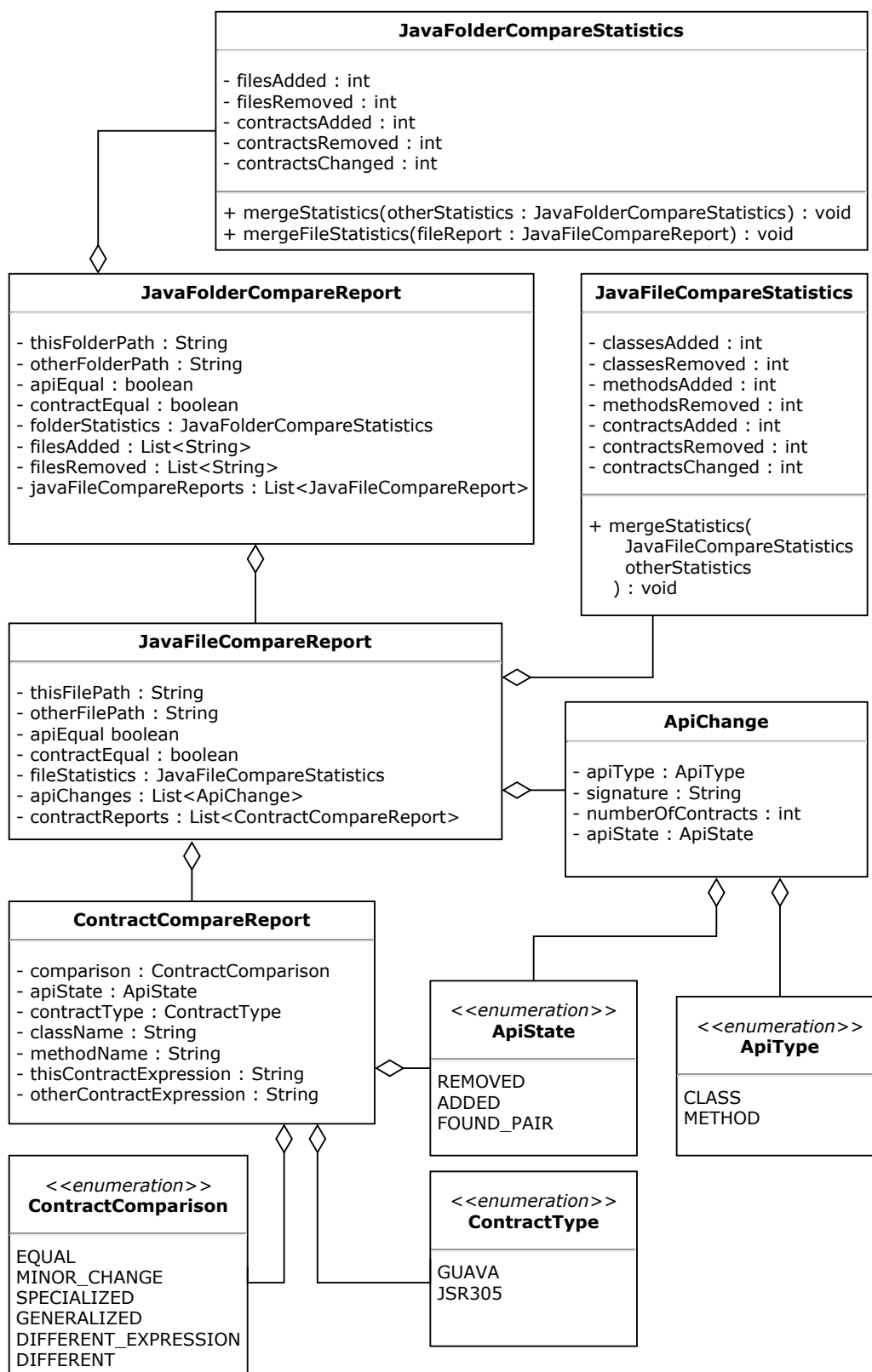
Pro externí reprezentaci modelu jsem zvolil použití formátu JSON. Tento formát je široce používaný zápis dat a umožňuje relativně snadné ukládání objektů typu Java. JSON je tak vhodný pro zpracování strojem, ale je dobře čitelný i pro lidské oko (v případě, že byl zformátován). Díky těmto kvalitám je JSON vhodným formátem pro zobrazení, archivaci i další zpracování

extrahovaných dat.

Alternativou bylo použití formátu XML. Tento formát má v podstatě stejné přednosti jako JSON, s tím rozdílem, že obvykle není potřeba dalšího formátování proto, aby byl čitelný pro člověka, nicméně je oproti formátu JSON více opsaný. I přesto, že XML by byla také validní možnost pro reprezentaci dat, po dohodě s vedoucím práce jsme se rozhodli pro použití JSON. Důvodem byla zejména jeho stručnost, ale také lepší vlastnosti pro předávání mezi jinými nástroji.

### **5.5.1 Specifikace formátu**

**TODO: Doplňit**



Obrázek 5.2: UML diagram datového modelu pro porovnání kontraktů

## 6 Nástroj pro analýzu kontraktů

Cílem práce z hlediska implementace bylo vytvořit nástroj, který by umožňoval získání dat podle výše navrženého modelu ze zdrojové či přeložené formy Java programu. Výsledná aplikace by pak měla umožnit vytvoření externí reprezentace dat a případně také porovnání DbC konstrukcí. Nástroj by měl být schopen zpracovat alespoň dva způsoby popisu DbC konstrukcí a měl by dovolovat snadné rozšíření pro další způsoby. S využitím tohoto nástroje by pak měla být vytvořena jednoduchá uživatelská aplikace, která by sloužila k načtení a zobrazení dat modelu.

Smyslem aplikace je umožnit detekci kontraktů ve zdrojových, respektive přeložených, souborech jazyka Java. Nalezené kontrakty je pak možné analyzovat a zkoumat způsob a četnost použití jednotlivých typů v různých projektech. Aby bylo možno získaná data dále analyzovat je zde také export do formátu JSON. Nástroj umožňuje porovnání dvou adresářů s soubory, což může užitečné zejména při porovnání dvou projektů jiné verze my chceme zjistit, zda se nezměnilo rozhraní či jejich omezení v rámci kontraktů oproti předchozí verzi.

**TODO: celková "architektura" nástroje - tj. knihovna + gui, dále struktura projektu a potom v textu důležité třídy v implementaci (+možná jejich uml model)**

### 6.1 Knihovna

Pro realizace nástroje jsem se rozhodl implementovat knihovnu, která poskytuje metody potřebné pro extrakci, porovnání a export kontraktů. Její součástí je také samozřejmě model použitý pro jejich reprezentaci.

#### 6.1.1 Použité technologie

##### Programovací jazyk

Pro realizaci knihovny jsem použil jazyk Java verze 1.8. Jedním z hlavních důvodů bylo, že nástroj zkoumá reprezentace kontraktů v jazyce Java, díky

tomu je možné dané konstrukce snadno testovat a zkoušet přímo v tomto projektu. Vedoucí práce také upřednostňoval použití jazyka Java z důvodů případného propojení s jinými nástroji, které byly vyvinuty pro práci s kontrakty v rámci univerzity a jsou také realizovány v Java. Osobně mám s jazykem Java pravděpodobně největší zkušenosti, což byl další z důvodů, proč tento jazyk použít. Z těchto důvodů byla volba jazyka poměrně jednoznačná.

I přesto, že v průběhu vývoje projektu vyšla verze Java 1.9 a posléze i 1.10, rozhodl jsem se po domluvě s vedoucím práce ponechat stabilní verzi 1.8 a nepřecházet v průběhu vývoje na novější verzi jazyka.

### **Vývojové prostředí**

Nástroj byl realizován ve vývojovém prostředí IDEA IntelliJ Ultimate 2017.3.3.

### **Práce se závislostmi**

Pro zajištění závislostí jako jsou knihovny třetích stran, ale také pro snadnou distribuci knihovny jsem zvolil technologii Apache Maven [34]. Jedná se o široce používaný nástroj pro získávání závislostí a tvorbu projektů.

### **Logování**

Pro logování, tedy zobrazení a uložení chybových či informačních zpráv, byla použita knihovna Apache Log4j [35]. Tato knihovna umožňuje pokročilé možnosti logování, které je možné dobře nastavit pomocí konfiguračních souborů. Opět se jedná o široce používanou technologii.

### **Tokenizace zdrojových souborů jazyka Java**

Pro implementaci tokenizace souborů jsem se rozhodl použít knihovnu JavaParser [26]. Učinil jsem tak na základě mého průzkumu (viz 4. kapitola Tokenizace jazyka Java). Knihovna poskytuje komplexní reprezentaci daného zdrojového souboru a je tak možné jej dále analyzovat a zpracovávat. Knihovnu je možné snadno použít přímo v projektu díky zprostředkovanému API.

### **Dekompilace přeložených souborů jazyka Java**

Jako dekompilátor přeložených souborů jazyka Java (`.class`) jsem použil knihovnu Procyon [32]. Jedná se o nástroj, který umožňuje snadnou dekompilaci souborů a to včetně moderních konstrukcí jazyka Java. Hlavním

důvodem volby této knihovny byla možnost použití dekompilace v rámci kódu za pomoci API.

## Práce s formátem JSON

Pro ukládání reprezentací do formátu JSON byla použita knihovna Gson [36]. Umožňuje intuitivní převod objektu typu Java<sup>1</sup> do formátu JSON za pomoci API. Mimo jiné také umožňuje formátování *Pretty Print*, které je lépe čitelné pro člověka.

## Testování

Pro tvorbu jednotkových testů byla použita technologie JUnit 5 [37].

### 6.1.2 Návrh nástroje

Celý nástroj je rozdělen do několika modulů .... viz obrázek

### 6.1.3 Dekompilace Bytecode

Jak již bylo zmíněno výše, pro dekompilaci Java `.class` souborů byla použita knihovna Procyon. Ta poskytuje metodu `void decompile(String internalName, ITextOutput output)`, která přečte vstupní soubor s přeloženým kódem a do jiného souboru uloží jeho dekompilovanou verzi. V mém nástroji dekompilaci obstarává obalovací metoda `boolean decompileClassFile(String filename)`, která se nachází v třídě `io.IOServices`. Ta vytvoří dočasný soubor dle konfigurace a vrátí, zda byla operace úspěšná. Z hlediska pracovního postupu dekompilaci vyvolává třída `JavaFileParser` v metodě `ExtendedJavaFile parseFile(File file)` v případě, že se vstupní soubor má koncovku `.class`. Pokud je dekompilace bez chyb, je daný, dočasně vytvořený, soubor zpracován stejným způsobem, jako by se jednalo o zdrojový soubor. Po zpracování je dočasný soubor smazán.

### 6.1.4 Parsování Java souborů

Pro zpracování zdrojových souborů jazyku Java byla použita knihovna `JavaParser`. Ta poskytuje metodu `parse()`, která vytvoří komplexní strukturu daného zdrojového souboru. V prvním kroku se tato struktura projde a vyhledá všechny třídy (*class*) a také rozhraní *interface* a výčtové typy *enum*.

---

<sup>1</sup>Mohou použity téměř libovolné objekty, avšak nesmějí být cyklické (Objekt nesmí ve své hierarchii atributů opět obsahovat tentýž objekt)

Pro účely modelu jsou si tyto tři prvky rovny. Každý nalezený prvek je následně uložen do modelu. V případě třídy a rozhraní je se struktura prochází dále a do modelu jsou uloženy všechny konstruktory, které se z hlediska modelu považují za metody (viz níže). Následně jsou uloženy všechny anotace dané „třídy“.

Po této přípravě je využita třída `MethodVisitor`, která dědí od třídy `VoidVisitorAdapter` a umožňuje procházet všechny metody v daném souboru. V metodě pak máme k dispozici objekt typu `MethodDeclaration`, který obsahuje všechny potřebné údaje a také rodičovský `ExtendedJavaFile`. Pro každou metodu je nalezena její rodičovská třída. Hledá se nejvyšší rodič a tudíž vnořené metody nemají jako rodiče vyšší metodu ale nejvyšší dostupnou třídu. Pro danou metodu jsou následně uloženy všechny anotace a i její parametry. Následně je uloženo celé tělo metody jako seznam objektů typu `Node`, které umožňují další zpracování. Z těchto získaných dat je vytvořena instance objektu `ExtendedJavaMethod`, která je následně uložena do své rodičovské `ExtendedJavaClass`.

### 6.1.5 Extrakce kontraktů

#### Obecně

Poté, co je ze Java souboru vytvořen objekt typu `ExtendedJavaFile`, je možné začít extrahovat kontrakty. Během získávání kontraktů se tato struktura prochází a postupně se k jednotlivým třídám a metodám přidávají kontrakty. Poté, co jsou všechny extrakce dokončeny, je za pomoci třídy `Simplifier` objekt převeden na typ `JavaFile`, který obsahuje pouze relevantní informace a je připraven pro export. Během získávání kontraktů se také postupně aktualizují statistické údaje o počtu kontraktů a o počtu metod, které kontrakty obsahují.

#### Guava Preconditions

Vzhledem k tomu, že všechny kontrakty tohoto typu jsou realizovány pomocí volání metod ze třídy `Preconditions`, zaměřuje se extrakce pouze na těla metod a tříd či ostatních částí metod si algoritmus nevšímá. Postupně se procházejí jednotlivé části metody (objekty `Node`) a ve chvíli kdy se narazí na `Node`, který je typu `MethodCallExpr`, tedy jedná se o volání metody, zjišťuje se, zda se jedná o volání některé z metod třídy `Preconditions`, pokud ano, je tento výraz dále zpracováván. Název Guava metody je uložen do kontraktu jako atribut `function`. První parametr metody, zpravidla ten klíčový, je

uložen jako atribut **expression**. Ostatní parametry obvykle souvisejí pouze s tvarem chybové zprávy, ty jsou uloženy do seznamu **arguments**.

## JSR305

Na rozdíl od Guava Preconditions mohou být kontrakty typu JSR305 obsaženy v anotacích tříd a metod a také v jejich parametrech. Zde je tedy nutné procházet tyto bloky a naopak těla metod je možné zanedbat. Postupně se procházejí jednotlivé anotace tříd i metod. Jakmile je daná anotace výrazem JSR305, je uložena jako kontrakt. Tvar anotace představuje **function** a stejně jako v případě Guava, první parametr je uložen jako **expression** a ostatní jsou uloženy do seznamu **arguments**. Tyto anotace však obvykle parametr nemají. Takto nalezené kontrakty v anotacích třídy jsou označeny za neměnné proměnné (class invariants) a v anotacích metod se pak jedná o post-conditions, vztahují se k výstupu metody. Zbývají parametry metod, u kterých se opět zkoumají anotace stejným způsobem. Tyto anotace však vždy mívají alespoň jeden atribut a tím je tvar samotného parametru.

**TODO: Uvést co vše rozpoznávám**

### 6.1.6 Porovnávání kontraktů

### 6.1.7 Popis API

### 6.1.8 Přidání parseru pro nový typ kontraktu

Při vytváření knihovny i aplikace byl kladen důraz na abstrakci od použitých typů kontraktů, aby bylo možné snadno přidat parser pro nový typ kontraktu. Grafickou aplikaci není třeba nijak měnit, ale je třeba provést několik kroků v rámci knihovny. Pro zprovoznění nového typu kontraktu jsou potřeba tyto kroky:

#### Přidání položky do **ContractType**

Nejprve je třeba přidat položku do výčtového typu **ContractType**. Název by měl být vhodně zvolen, protože je zobrazen v exportovaných datech, ale i v grafické aplikaci.

**TODO: tady by se vhodně dalo odkázat na uml model podstatných tříd/rozhraní nástroje (viz pozn. výše)**



## Vytvoření nového analyzátoru

Následně je třeba vytvořit funkční část daného parseru. Je tedy nutné vytvořit třídu, která bude implementovat rozhraní `ContractParser`. Toto rozhraní požaduje implementaci pouze jedné metody a tou je `ExtendedJavaFile retrieveContracts(ExtendedJavaFile extendedJavaFile)`. Aby byly zachovány jmenné konvence současné knihovny, měla by se tato třída jmenovat `TypXParser`, kde `TypX` reprezentuje název nového typu kontraktu. Tato třída by se měla nacházet v balíčku se stejným jménem (ale s malými písmeny) a ten by se měl nacházet v balíčku `cz.zcu.kiv.contractparser.parser`. Tvar samotné metody již závisí na principech daného kontraktu. Obecně platí, že by se měly kontrakty detekovat a vytvořit na základě dat ze vstupního objektu typu `ExtendedJavaFile` a ve stejném objektu je také vrátit. Pro lepší představu doporučuji prozkoumat již implementované analyzátory pro JSR305 a Guava Preconditions.

## Doplnění továrny `ParserFactory`

Dalším krokem je doplnění továrny `ParserFactory`. Zde je pouze třeba přidat nový `case` do konstrukce `switch`. Tento blok by měl vracet instanci nového parseru v případě že vstoupí tento typ v objektu `ContractType`.

### 6.1.9 Testování

Pro bezchybnou funkci daného analyzátoru je vhodné vytvoření testů. Testovací data pro současné testy jsou umístěny v `resources/testFiles`, kde jsou pak dále děleny do složek. Pro přehlednější zobrazení ve vývojovém prostředí doporučuji v testovacích datech použít referenční jména tříd, ne pouze souborů. Důvodem je to, že IDE soubory typu `.java` považuje za součást projektu a může tak zobraz pouze název třídy, místo názvu daného souboru.

## 6.2 Uživatelská aplikace

### 6.2.1 Použité technologie

Aplikace byla, stejně jako knihovna, implementována v jazyce Java verze 1.8 ve vývojovém prostředí IDEA IntelliJ Ultimate 2017.3.3 s využitím Apache Maven. Grafické uživatelské rozhraní bylo vytvořeno využitím platformy JavaFX.

## Externí knihovny

Mimo následujících knihoven byly opět využity externí knihovny Apache Log4j a Google Gson.

**ControlsFX** Tato knihovna rozšiřuje JavaFX a umožňuje použití dalších funkcí a objektů zejména pak `CheckListView`, což je použito pro zobrazení seznamu souborů [38].

**FontAwesomeFX** Knihovna FontAwesomeFX slouží opět k rozšíření JavaFX. Tuto knihovnu jsem použil pro rozšíření možností zobrazení ikon [39].

### 6.2.2 Ovládání aplikace

Pro zlepšení práce s aplikací, byla rozdělena na dvě části. Aplikaci je možné spustit bez parametrů jako grafickou aplikaci, případně je možné s použitím parametrů aplikaci obsluhovat pomocí konzole.

#### Grafická část

#### Konzolová část

### 6.2.3 Možnosti a limitace aplikace

## 6.3 Optimalizace

obecně, snažil jsem se zajistit co nejlepší...

### 6.3.1 Analýza a refaktoring kódu

- ručně, nástroje IDE, -> snížení cyklomatickosti, zpřehlednění kódu

### 6.3.2 Zjedenodušení modelu

- vyhnutí se použití rozsáhlých objektů knihovny - pozitivní dopad na paměťovou náročnost, zpřehlednění modelu

- při batch soubory průběžně ukládat, aby nezatěžovalo paměť - přeparsování souborů se nevyplatí - stačí udělat vše a pak jen filtrovat - rozebrat nároky na paměť v aplikaci

## 7 Testování

S cílem zajistit co největší spolehlivost a tedy i kvalitu tohoto nástroje, byla knihovna i uživatelská aplikace řádně otestována. Nástroj byl otestován pomocí automatizovaných jednotkových testů, ale také manuálním testováním.

### 7.1 Jednotkové testy

V rámci knihovny byla vytvořena řada jednotkových testů. Tyto testy ověřují různé testovací případy a byly použity pro zajištění spolehlivosti, ale také jako kontrola během vývoje. Testy jsou zaměřeny zejména na extrakci kontraktů, kde se snaží analyzovat kontrakty z uměle vytvořených testovacích dat (viz níže). Jsou zde také testy pro ověření správnosti porovnávacího nástroje a také modulu pro tokenizaci zdrojových souborů. Celkem bylo implementováno 55 jednotkových testů různé complexity.

### 7.2 Funkční testování

Vzhledem k tomu, že uživatelská aplikace je plně závislá na vytvořené knihovně, je možné tuto knihovnu testovat zároveň s uživatelskou aplikací. Ta byla testována pomocí manuálního testování tím, že byly ověřovány různé testovací případy. Následuje seznam těchto případů, pro které byla aplikace úspěšně otestována.

#### 7.2.1 Testovací případy

Zde zjednodušený popis testovacích případů, pro které byla uživatelská aplikace testována. Jednotlivá tlačítka a operace zmíněné v rámci popisu testovacích případů jsou podrobně popsány v příloze A Uživatelská příručka.

#### **Spuštění aplikace**

Po spuštění aplikace bez parametrů v pořádku naběhne grafická část aplikace. Ikony tlačítek i celého okna aplikace by měly být zobrazeny.

#### **Přidání souborů - přidání prvních souborů**

Po stisknutí tlačítka *Přidat soubory* naběhne okno pro výběr souborů. Zde je možné vybrat více souborů a je možné vybrat pouze soubory `.java` nebo

**.class.** Po vybrání souborů naběhne načítací okno zobrazující průběh extrakce kontraktů (v případě malého množství souborů a/nebo vysokého výkonu se okno zobrazí jen velmi krátce).

Po přidání souborů se vybrané soubory objeví v levém seznamu souborů (pokud není vybrán filtr *Show Non Contract Objects*, soubory bez kontraktů se nezobrazí). Na základě toho se aktualizuje popisec pod seznamem (*Files selected*, který by měl nyní jako druhé číslo ukazovat nový počet souborů. Dále se aktualizují údaje v pravé části *Global Statistics*. Počet souborů i ostatní informace odpovídá přidaným datům přidaných souborů. Také se zpřístupní tlačítka *Select All*, respektive *Deselect All*. Do konzole je také vypsáno, kolik souborů se přidalo (celkové číslo zahrnuje i soubory skryté filtrem).

Jestliže je okno je zavřeno tlačítkem *Storno* či křížkem, okno se schová a aplikace pokračuje v běhu bez jakékoliv chyby.

### **Přidání souborů - přidání dalších souborů**

Každé další přidání souborů funguje stejným způsobem jako to první. Pokud se ale pokusíme přidat soubory, které již v seznamu jsou, opětovně přidány nejsou a také nebudou zahrnuty do počtu přidaných souborů ve výpisu do konzole.

### **Přidány složky**

Po stisknutí tlačítka *Přidat složku* se opět zobrazí okno pro výběr souborů. Nyní je však možné vybrat pouze složky a žádné soubory nejsou zobrazeny. Po výběru složky probíhá akce identicky jako při přidáním jednotlivých souborů (to samé platí pro opětovné přidání).

Opět platí, že pokud je okno zavřeno tlačítkem *Storno* či křížkem, schová se a aplikace pokračuje v běhu bez jakékoliv chyby.

### **Vybrání souboru**

Ve chvíli, kdy jsou nějaké soubory v levém seznamu, je možné kliknout na název souboru (ne na zaškrťovací pole vedle), to soubor zvýrazní a také aktualizuje oblast *File Details* v pravé dolní části. Zde by se měl zobrazit název daného souboru s kompletní cestou. Současně by se měly aktualizovat počty kontraktů dle typu. Také se zpřístupní tlačítka *Show Details*.

## Označení souborů

Soubory je možné označit pomocí zaškrťovacích polí nalevo od názvů souborů či pomocí tlačítka *Select All/Deselect All*. Jestliže nejsou označeny všechny soubory, tlačítko by mělo mít nápis *Select All* a jeho stisknutí by mělo všechny soubory označit. Naopak pokud jsou všechny soubory označeny, měl by zde být nápis *Deselect All*, což by mělo zrušit označení všech položek.

Ať je použito tlačítko, či zaškrťovací políčka, měl by vždy se aktualizovat popisek pod seznamem oznamující počet označených souborů. Mimo to by se také měly vždy zpřístupnit tlačítka pro export a mazání souborů v horním panelu. Naopak pokud není žádný soubor označen, tlačítka by se měla opět zablokovat.

## Mazání souborů

Tlačítko pro mazání souborů by mělo být přístupné pouze ve chvíli, kdy je označen alespoň jeden soubor. Po jeho stisku by měly ihned zmizet označené soubory ze seznamu a stejným způsobem jako při přidávání souborů by se měly aktualizovat všechny patřičné elementy rozhraní. Do konzole by se také měla napsat informace o počtu smazaných souborů. Pokud byly smazány všechny viditelné soubory, ale některé soubory zůstávají skryty, uživatele o tom informuje popisek zobrazený v seznamu. V opačném případě je zde napsáno, že list je prázdný a je možné soubory přidat pomocí tlačítek.

## Export souborů

Stejně jako mazání souborů, je i export dostupný pouze v případě, pokud je označen alespoň jeden soubor.

## 7.3 Testovací data

- popis testovacích dat (syntetická, skutečná - výsledky testů)

## 7.4 Výsledky testů

## 8 Zhodnocení výsledků

TODO: Doplnit

### 8.1 Úspěšnost detekce kontraktů

### 8.2 Úspěšnost porovnání kontraktů

### 8.3 Limitace

### 8.4 Prostor pro zlepšení

- co by šlo zlepšit doplnit - lepší parsování kontrakt expression - zhruba jak

## 9 Závěr

TODO: Doplnit

# Přehled zkratek a použitých výrazů

**TODO: Doplnit**

**DbC** Design By Contract



# Literatura

- [1] Dr. Ulbert Zsolt: *Software Development Process and Software Quality Assurance*. University of Pannonia 2014
- [2] *Defensive Programming* [online]. [cit. 2018-05-22]. <<http://www.drdoobbs.com/defensive-programming/184401915>>
- [3] Jens Dietrich, David J. Pearce, Kamil Jezek, and Premek Brada: *Contracts in the Wild: A Study of Java Programs*. In LIPIcs-Leibniz International Proceedings in Informatics (Vol. 74), ECOOP 2017. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik. 2017
- [4] Bertrand Meyer, “Applying ‘design by contract’,” *Computer*, vol. 25, no. 10, pp. 40–51, Oct 1992
- [5] Bertrand Meyer, *Object-oriented software construction*, Prentice-Hall international series in computer science, 1988
- [6] Antoine Beugnard, Jean-Marc Jézéquel, Noël Plouzeau, and Damien Watkins. Making Components Contract Aware. *Computer*, 32(7):38–45, 1999
- [7] *Bertrand Meyer’s technology + blog* [online]. [cit. 2018-04-11]. <[bertrandmeyer.com/bio/](http://bertrandmeyer.com/bio/)>
- [8] *EiffelStudio* [online]. [cit. 2018-04-13]. <[dev.eiffel.com](http://dev.eiffel.com)>
- [9] *Guava Preconditions* [online]. [cit. 2018-04-21]. <<https://github.com/google/guava>>
- [10] *JSR305* [online]. [cit. 2018-04-21]. <<https://jcp.org/en/jsr/detail?id=305>>
- [11] *Cofoja* [online]. [cit. 2018-04-21]. <<https://github.com/nhatminhle/cofoja>>
- [12] *valid4j* [online]. [cit. 2018-04-21]. <<http://www.valid4j.org>>
- [13] *jContractor* [online]. [cit. 2018-04-22]. <<http://jcontractor.sourceforge.net>>
- [14] *Code Contracts* [online]. [cit. 2018-04-22]. <<https://www.microsoft.com/en-us/research/project/code-contracts/>>

- [15] *Dokumentace Code Contracts* [online]. [cit. 2018-04-22]. <<https://docs.microsoft.com/en-us/dotnet/framework/debug-trace-profile/code-contracts>>
- [16] *PhpDeal* [online]. [cit. 2018-04-22]. <<https://github.com/php-deal/framework>>
- [17] *Boost.Contract* [online]. [cit. 2018-04-22]. <<https://www.boost.org/doc/libs/master/libs/contract>>
- [18] *Eiffel* [online]. [cit. 2018-04-22]. <<https://www.eiffel.org>>
- [19] *The Java Language Environment* [online]. [cit. 2018-05-01]. <<http://www.oracle.com/technetwork/java/langenv-140151.html>>
- [20] *TIOBE Trend of programming languages* [online]. [cit. 2018-05-01]. <<https://www.tiobe.com/tiobe-index>>
- [21] *Compiler Design Tutorial* [online]. [cit. 2018-05-01]. <[https://www.tutorialspoint.com/compiler\\_design](https://www.tutorialspoint.com/compiler_design)>
- [22] *Just-In-Time Compilers* [online]. [cit. 2018-05-01]. <<http://www.webbasedprogramming.com/Java-Unleashed-Second-Edition/ch44.htm>>
- [23] *Internals of Java Class Loading* [online]. [cit. 2018-05-01]. <<http://www.onjava.com/pub/a/onjava/2005/01/26/classloading.html>>
- [24] *JFlex* [online]. [cit. 2018-05-01]. <<http://jflex.de/>>
- [25] *ANTLR* [online]. [cit. 2018-05-01]. <<http://www.antlr.org>>
- [26] *JavaParser* [online]. [cit. 2018-05-01]. <<http://javaparser.org>>
- [27] *Roaster* [online]. [cit. 2018-05-01]. <<https://github.com/forgel/roaster>>
- [28] *Top 8 Java Decompilers* [online]. [cit. 2018-05-02]. <<http://www.crunchytricks.com/2016/07/best-offline-java-decompilers.html>>
- [29] *Decompilers Online* [online]. [cit. 2018-05-02]. <<http://www.javadecompilers.com>>

- [30] *A Quick Look at Java Decompilers* [online]. [cit. 2018-05-02]. <<http://blog.macuyiko.com/post/2015/a-quick-look-at-java-decompilers.html>>
- [31] *JD Project* [online]. [cit. 2018-05-02]. <<http://jd.benow.ca>>
- [32] *Procyon* [online]. [cit. 2018-05-02]. <[bitbucket.org/mstrobels/procyon](http://bitbucket.org/mstrobels/procyon)>
- [33] *CFR* [online]. [cit. 2018-05-02]. <<http://www.benf.org/other/cfr>>
- [34] *Apache Maven* [online]. [cit. 2018-05-05]. <<https://maven.apache.org>>
- [35] *Apache Log4j* [online]. [cit. 2018-04-11]. <[logging.apache.org/log4j/2.x/download.html](http://logging.apache.org/log4j/2.x/download.html)>
- [36] *Gson* [online]. [cit. 2018-04-11]. <[github.com/google/gson](https://github.com/google/gson)>
- [37] *jUnit* [online]. [cit. 2018-04-11]. <[junit.org/junit5/](http://junit.org/junit5/)>
- [38] *ControlsFX* [online]. [cit. 2018-04-12]. <[fxexperience.com/controlsfx/](http://fxexperience.com/controlsfx/)>
- [39] *FontAwesomeFX* [online]. [cit. 2018-04-12]. <[bitbucket.org/Jerady/fontawesomefx](http://bitbucket.org/Jerady/fontawesomefx)>

# Seznam příloh

- A Uživatelská příručka
- B Obsah CD

# A Uživatelská příručka

TODO: Doplnit

## B Obsah CD

TODO: Doplnit