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

Diplomová práce

Analýza popisů sémantického kontraktu v Java technologiích

Plzeň 2018 Václav Mareš

Místo této strany bude zadání práce.

Prohlášení

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

V Plzni dne 26. června 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 thesis is creation of a tool which enables extraction of chosen constructs 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 the implementation of the tool including the design of the 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

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 nepopiratelných výhod této metodiky, 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 při používání 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 práce 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 odvětí 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 Této 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 by 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 využití 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

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 nejpřehlednější 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 kódu

Jednou z preventivních technik pro zajištění kvality jsou různá doporučení pro psaní kódu. Ta mohou být určena standardem, neoficiálními konvencemi či firemní politikou. Kód psaný na základě těchto doporučení bývá zpravidla více čitelný pro ostatní členy týmu, efektivnější a bezpečnější.

Příkladem kódovacího standardu může být MISRA [3]. Jedná se o standard, který byl původně navržen pro zvýšení bezpečnosti a spolehlivosti vestavěných systémů v automobilech napsaných v jazyce C. Postupem času se rozšířil také do jazyka C++ a dostal se i mimo oblast automobilového průmyslu.

Analyzátory kódu

V rámci kontroly kódu je možné použít analyzátory kódu. Tyto nástroje kontrolují napsaný kód buď staticky při překladu či použitím daného nástroje. Tím vývojář získá rychlou odezvu k právě napsanému kódu a může jej ihned opravit. Nějaké analyzátory bývají zpravidla integrovány ve vývojovém prostředí a mnoho prostředí také umožňuje přidání jiných validátorů. Ověření probíhá na základě definované sady pravidel, která často bývá vytvořena na základě standardu či obecných *best practises* (osvědčené postupy).

2.3.3 Kontrakty

Jednou z možností, jak zajistit kvalitu software, je také použití kontraktů. Ty reprezentují dohodu o použití dané komponenty, či metody, 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: [4][5][6][7].

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*[1] (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

[1]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 [8][9]. 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 specifikaci 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 [10] 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 Preconditons 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`).

Tento nástroj je hojně používán, což lze vyvodit z textu Contracts in Wild [4], ale i z množství zmínek a návodů na internetu (např. [11]). Jedná se také o živý projekt, který je stále aktualizován.

3.4.2 JSR305

JSR305 [12] 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 neposky-

tuje 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 naloží. 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*[2], stále je hojně využíváno v řadě projektů a má tak smysl jej zkoumat.

3.4.3 Cofoja

Contracts for Java [13], 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);
```

Nejnovější verzí tohoto nástroje je 1.3, která byla vydána v únoru 2016. Poslední aktivita proběhla v srpnu 2017, je tak možné se domnívat, že se

[2] *Dormant* značí, že práce na tomto JSR projektu byla pozastavena. Může to být na základě hlasování komise, či protože dané JSR dosáhlo konec své životnosti.

v současné době jedná o definitivní verzi.

3.4.4 valid4j

Valid4j [14] 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 neposkytuje podporu pro neměnné podmínky za pomocí 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());
}
```

Poslední verzí tohoto nástroje je 0.5.0 z prosince 2015. Krátce poté se také zastavila aktivita v repozitáři GitHub, můžeme tak předpokládat, že projekt byl prozatím ukončen.

3.4.5 jContractor

Posledním příkladem je jContractor [15]. 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*[3] (dále bajtkód) a zajišťují tak kontrolu pouze při běhu programu. Zde je krátká ukázka kódu demonstrující použití nástroje jContractor:

[3]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, v kapitole 4 - Analýza kódu jazyka Java

```
// 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;
}
```

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 [16][17], 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. Pokud je kontrakt porušen, je vyhozena výjimka informující o dané chybě. Dle potřeby je možné tuto výjimku specifikovat a definovat tak vlastní chování. 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

Jedním z nástrojů, které umožňují reprezentaci kontraktů pro jazyk PHP je aplikační rámec PhpDeal [18]. Funguje na principu klíčových slov v dokumentačních komentářích, ty jako argument obsahují řetězec s PHP kódem, pomocí kterého je možné konstruovat podmínky. Při definici vstupních a výstupních podmínek jsou konstrukce uvedeny u dané funkce, neměnné podmínky jsou pak definovány v rámci třídy. Kontrakty 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. Podmínky jsou vyhodnocovány při běhu programu v závislosti na definici nástroje. Pokud je některá z podmínek porušena, je vyhozena výjimka. Je doporučeno vypnout vyhodnocování pro produkční verzi. 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)
{ ... }
```

PhpDeal není příliš rozšířený nástroj, ale je snadný na použití. Poslední aktualizace proběhla na konci roku 2017.

3.5.3 Boost.Contract v C++

Knihovna Boost.Contract [19] 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 [20], 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`. Jako u většiny nástrojů, pokud nejsou kontrakty dodrženy, je vyvolána výjimka. 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.

Co se základní funkcionality týče, až na výjimky nástroje poskytují srovnatelné možnosti. Typově se pak obvykle jedná o nástroje, které zprostředkovávají 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.

Název	Jazyk	Typ	Dostupné podmínky		
			Vstup.	Výstup.	Neměn.
Guava	Java	API	ANO	NE	NE
JSR305	Java	Anotace	ANO	ANO	ANO
Cofoja	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.
****** Anotace jsou následovány funkčním PHP kódem

Tabulka 3.1: Souhrn nástrojů pro práci s kontrakty

4 Analýza kódu jazyka Java

4.1 O jazyce Java

Java [21] 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 [22].

Tato kapitola bude čtenáře informovat o základních vlastnostech jazyka Java. 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 vykonáván 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 aplikace, 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 třídy 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.

4.2 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 nichž můžeme ve výsledku získat reprezentaci ve stromové podobě. Toho je běžně využíváno při tvorbě překladače a interpretu [23][25][26].

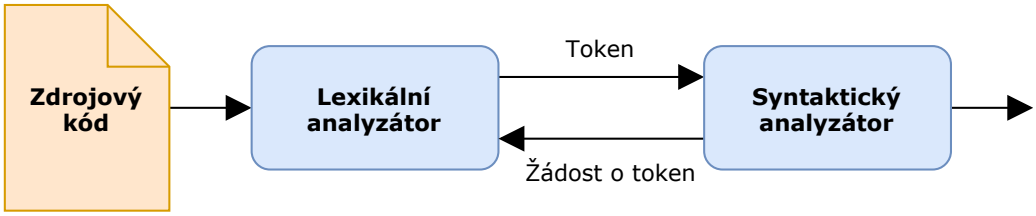
4.2.1 Lexikální analýza

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 do tzv. lexémů. Lexém je posloupnost znaků, která tvoří ucelenou část kódu (např. číslo, název proměnné atd.). Lze je rozdělit do různých kategorií: identifikátory, klíčová slova (rezervované identifikátory), čísla, jednoznakové omezovače atd.

Zpracování začíná prvním dosud nezpracovaným znakem ze vstupu. Končí ve chvíli, kdy se nacházíme v koncovém stavu a již není možné přidat další znak. Tím je daná posloupnost znaků uzavřena a tvoří tak jeden lexém. Např.

při zpracování kódu `var+4` začneme zpracovávat písmenem `v` a postupujeme dál přes `a` a `r`. Písmenem může začít pouze identifikátor nebo klíčové slovo. Ve chvíli, kdy dojdeme k `+`, víme, že je tento lexém kompletní, protože znak `+` nemůže být součástí identifikátoru resp. klíčového slova. Rozlišení, zda se jedná o identifikátor nebo klíčové slovo je řešeno pomocí tabulky klíčových slov.

Toto zpracování se řídí definovanou gramatikou jazyka. Protože se jedná o gramatiku typu 3, je možné jí nahradit regulárními výrazy. Pokud by analyzátor narazil na neplatný symbol v dané posloupnosti, je vyhozena chyba. Lexikální analýza není schopna odhalit chybu, jako např. že není platná posloupnost `* *`, protože se jedná o dva platné lexémy. Ve chvíli, kdyby by se jednalo o text `**`, vznikne chyba, protože to není platný tvar z hlediska regulárního výrazu. Seznam lexémů je dále předáván ke zpracování syntaktické analýze.



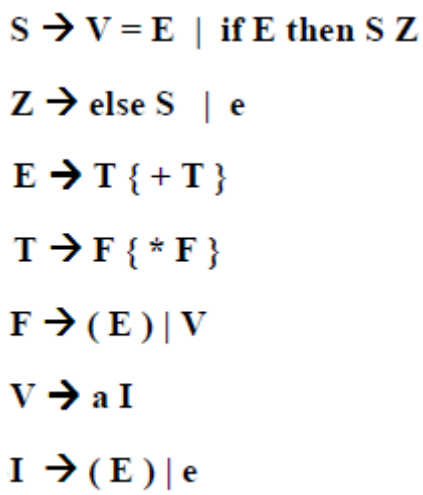
Obrázek 4.1: Princip lexikální analýzy

4.2.2 Syntaktická analýza

Ve chvíli, kdy je text převeden do lexémů pomocí lexikální analýzy, je možné provést syntaktickou analýzu, někdy nazývanou parsování. Lexikální analýza není schopna kontrolovat syntaxi kvůli limitacím regulárních výrazů. Syntaktická analýza však pracuje na úrovni bezkontextové gramatiky, která je té regulární nadřazená, proto to umožňuje. Na obrázku 4.2 je možné vidět ukázku této gramatiky. Velká písmena (`S`, `Z`, `E`, ..) představují neterminální symboly, jejichž výčet je na levé straně obrázku. Za šipkou pak následuje, na co je možné dané neterminální symbol přepsat. To často mohou být další neterminální symboly a celý výraz se takto rozrůstá. Svislými čarami (`|`) jsou odděleny možnosti, na které se může symbol přepsat. Symbol `F` tedy může být rozepsán na `(E)` nebo na `V`.

Ostatní řetězce (zde `if`, `then`, `{`, ...) jsou potom terminální výrazy, které jsou konečné a nejde je dále rozepsat. Symbol malé `e` je speciální a před-

stavuje prázdný znak, který ukončuje danou posloupnost. Pomocí symbolů + a * je možné vyjádřit opakování daného výrazu. Tímto způsobem je tedy možné vyjádřit gramatiku programovacího jazyka. Pokud vstupní lexémy neodpovídají této gramatice, vzniká syntaktická chyba.



Obrázek 4.2: Ukázka bezkontextové gramatiky

Jedním ze způsobů, jak zpracovávat tuto gramatiku je rekurzivní sestup. Každý z neterminálních symbolů je prezentován procedurou, která se vykoná, ve chvíli, kdy se jedná o daný symbol. Takto se postupně rekurzivně volají procedury dokud není zpracován celý vstup. V proceduře s terminálním souborem pak můžeme vykonat určitou akci, která koresponduje s daným výrazem, např. přidat do seznamu další strojovou instrukci. Pro rozhodování je použita technika, kdy se nahlíží na další lexémy na vstupu a podle toho se určí o jaký výraz se jedná. Gramatika musí být vždy jednoznačná.

4.2.3 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 [27] 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 ANTLR, BYacc/J či CUP.

ANTLR ANTLR [28] 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 [29] 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 [30] 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.

4.3 Gramatika jazyka Java

Vzhledem k tomu, že Java je plnohodnotný programovací jazyk s řadou funkcí, je jeho specifikace gramatiky [31] velice rozsáhlá. V této části budou pouze vyzdviženy některé informace, které mohou být důležité pro problematiku extrakce kontraktů. Tyto informace budou vycházet ze specifikace Java verze 1.8, nicméně by měly být shodné i s ostatními verzemi současné doby.

Java je definována dvěma bezkontextovými gramatikami. Jedna z nich je lexikální a ta specifikuje, jak vypadají jednotlivé lexémy. Určuje, co může obsahovat identifikátor, jaká jsou klíčová slova, omezovače atd. Této části zde nebude věnována velká pozornost. Naopak druhou gramatikou, která je z hlediska extrakce kontraktů důležitější, je syntaktická gramatika.

4.3.1 Syntaxe

Syntaxe určuje, jaké posloupnosti lexémů jsou povolené pro správně fungující program. Protože se stále jedná o gramatiku, bude se při popisu vycházet z toho, že se daný úsek vždy skládá z neterminálů, které se přepisují na terminály a neterminály. Terminály (např. `class`) jsou konečné prvky, které není dále možné rozvíjet, zatímco neterminály (např. `ClassModifier`) zastupují jiné neterminály a terminály. Neterminál uzavřený v hranatých závorkách (`[]`) se může vyskytnou nejvýše jedenkrát. Složené závorky (`{}`) pak určují,

že se daný neterminál může vyskytnout nejvýše n-krát. Pravá strana rozdělená do více řádek je pouze příliš dlouhá než aby se na daný řádek vešla. Nejedná se alternativní možnost přepisu. Ty jsou vždy rozděleny symbolem svislé čáry (|).

Kompilační jednotka

Výchozím bodem je tzv. *compilation unit*, nebo-li kompilační jednotka. Z hlediska gramatiky se jedná o nejobecnější neterminál, jinak řečeno počáteční symbol, ze kterého vycházejí všechny ostatní neterminály. V praxi se jedná o zdrojový soubor jazyka Java.

```
CompilationUnit →
  [PackageDeclaration] {ImportDeclaration} {TypeDeclaration}
```

Zde je vidět, že se kompilační jednotka skládá z deklarace balíčku, deklarace importů a deklarace typu. Deklarace balíčku je jedna volitelná položka, kterou lze použít ke specifikaci toho, v jakém balíčku se tento soubor nachází. Neterminál `ImportDeclaration` představuje importy, kterých může být libovolné množství. Symbol `TypeDeclaration` pak představuje deklaraci rozhraní nebo deklaraci třídy. Samotná deklarace třídy se pak ještě dělí na deklaraci výčtového typu a deklaraci běžné třídy.

Deklarace třídy

Třídy představují základní stavební jednotku objektového jazyka Java. Z hlediska syntaxe zde bude popsána deklarace běžné třídy, nicméně deklarace rozhraní a výčtového typu jsou v mnoha ohledech podobné.

```
NormalClassDeclaration →
    {ClassModifier} class Identifier [TypeParameters]
    [Superclass] [Superinterfaces] ClassBody
```

Neterminál `ClassModifier` zde představuje modifikátor třídy. Mohou zde být definovány anotace (viz níže) a také určení, zda je daná třída veřejná či privátní, zda je abstraktní, statická atd. Symbol `Identifier` reprezentuje identifikátor nebo-li pojmenování dané třídy. To musí být unikátní jinak nastává kompilační chyba. `TypeParameters` představuje typové určení ve špičatých závorkách (např. `List<String>`). V případě `Superclass` se pak jedná o dědičnost (tedy `extendes NAZEV`) a `Superinterfaces` zastupuje implementace rozhraní (tedy `implements NAZEV`). Položka `ClassBody`

představuje celé tělo třídy, jsou tu tedy atributy, konstruktory, metody atd.

Deklarace metody

Metoda se z hlediska gramatiky skládá ze tří složek. Nejprve jsou zde modifikátory metody `MethodModifier`. Jedná se o stejné položky jako v případě třídy, tedy anotace a definice `private/protected/public`, `static`, `final` atd. Neterminál `MethodHeader` představuje hlavičku metody, ta je podrobněji popsána dále. Poslední částí je `MethodBody`, tedy tělo metody. To se skládá z bloku kódu, který je z hlediska syntaxe velice komplexní. Jsou zde podmínky, cykly, volání metod, definice proměnných atd.

```
MethodDeclaration →
    {MethodModifier} MethodHeader MethodBody

MethodHeader →
    Result MethodDeclarator [Throws]

MethodDeclarator →
    Identifier ( [FormalParameterList] ) [Dims]
```

Hlavička metody tvoří hlavní část její signatury. Symbol `Result` představuje návratový typ metody. `Throws` umožňuje propagaci výjimek pomocí klíčového slova `throws`. Neterminál `MethodDeclarator` se pak dále člení. `Identifier` opět představuje identifikátor, tedy jméno dané metody. V závorce jsou pak uvedeny jednotlivé parametry. Ty se typicky skládají z definice typu a identifikátoru. Volitelné jsou pak opět různé modifikátory včetně anotací.

Deklarace anotace

Běžné anotace jsou definovány pouze symbolem `@` následovaným názvem anotace. Poté je možné volitelně do závorek uvést další parametry. Symbol `TypeName` reprezentuje název, ten musí představovat validní anotaci, jinak nastane chyba. Za názvem s v závorce mohou nacházet další parametry. Neterminál `ElementValue` představuje jednu položku hodnoty anotace. Může se jednat o libovolný objekt, pole, další anotaci, ternární výraz atd. `ElementValuePairList` pak představuje seznam přiřazení typu: `identifikátor = hodnota`.

```
Annotation →
    @ TypeName |
    @ TypeName ( ElementValue ) |
    @ TypeName ( [ElementValuePairList] )
```

4.4 Retenční politika anotací

Retention Policy [32], nebo-li retenční politika, určuje, v jaké chvíli mají být anotace odebrány z programu. V Java existují tři typy retenčních politik, jsou uchovány ve výčtovém typu `java.lang.annotation.RetentionPolicy`:

- `SOURCE`
- `CLASS`
- `RUNTIME`

Anotace s retenční politikou `SOURCE` bude uchována pouze ve zdrojovém kódu a během kompilace bude zahozena. Politika `CLASS` říká, že anotace zůstane zachována během kompilace, ale je zahozena při běhu programu. Retenční politika `RUNTIME` pak zajišťuje, že bude anotace dostupná i při běhu programu. Výchozí hodnotou je `CLASS`. To je možné změnit při definici dané anotace využitím jiné anotace `@Retention`. To je možné vidět na příkladu níže.

```
@Retention(RetentionPolicy.RUNTIME)
public @interface MySampleAnnotation {
  ...
}
```

Pokud nemají anotace retenční politiku `SOURCE`, jsou také zaneseny do bajtkódu. Jedná se tedy o podmínkou proto, abychom mohli extrahovat kontrakty definované anotacemi z přeložených souborů. Např. anotace reprezentací JSR305 mají retenční politiku `RUNTIME` a jsou tedy uchovány v bajtkódu.

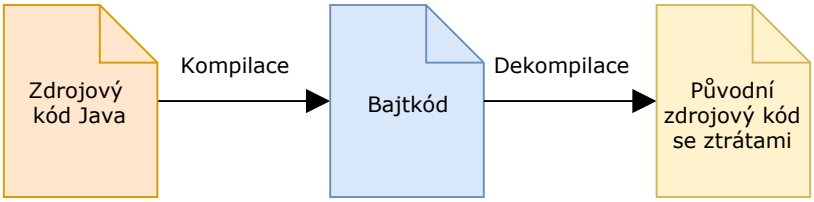
4.5 Rozbor přeložených souborů

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`.

4.5.1 Dekompilace

Aby bylo možné soubory analyzovat, musíme je nejprve dekompilovat. Dekompilace je vlastně opačný proces kompilace a jejím cílem je tedy získat původní zdrojový kód z cílové podoby. Tou bývají obvykle strojové instrukce, v případě Java se jedná o bajtkód. Výhodou bajtkódu je, že obsahuje vysoké množství různých podrobností, které umožňují téměř bezztrátovou rekonstrukci zdrojového kódu. Jednoduchý princip je vidět na obrázku 4.3.



Obrázek 4.3: Dekompilace

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 [33][34][35].

JD Project JD Project (Java Decompiler) [36] je jedním z nejčastěji referovaných nástrojů v rámci Java dekompilace. JD disponuje samostatnou aplikací s grafickým uživatelským rozhraním, pomocí kterého je možné soubory dekompilovat 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. Nástroj bohužel nedisponuje API, aby bylo možné jej použít přímo v kódu.

Procyon Procyon [37] 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. Stačí zavolat metodu `decompile()`, kam vstupuje jméno

souboru, který chceme dekompilovat a také kam má být směrován výstup. Výsledkem je pak zdrojový kód. Nejedná se samozřejmě přesnou kopii, ale jsou zde jisté limitace (viz níže).

CFR CFR [38] je dalším z nástrojů, který umožňuje dekompilaci i konstrukcí Java 1.9. Nástroj je možné použít pouze v rámci příkazové řádky, kdy je na výstup směrován dekompilovaný kód. Součástí tedy není API, které by umožnilo použití v kódu.

Nástrojů pro dekompilaci jazyka Java je velké množství a jejich výběr není snadný zejména poroto, 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 dekompilovat 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 nezanášejí, 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 atributech 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, tedy se jedná o knihovnu, která zprostředkovává API. Avšak Guava Preconditions umožňuje tvorbu pouze vstupních podmínek.

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 [4]. Principy obou těchto nástrojů byly již popsány výše (viz kapitola 3 - 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 kon-

trakt 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 ukládat také informace o třídách a metodách v daném souboru.

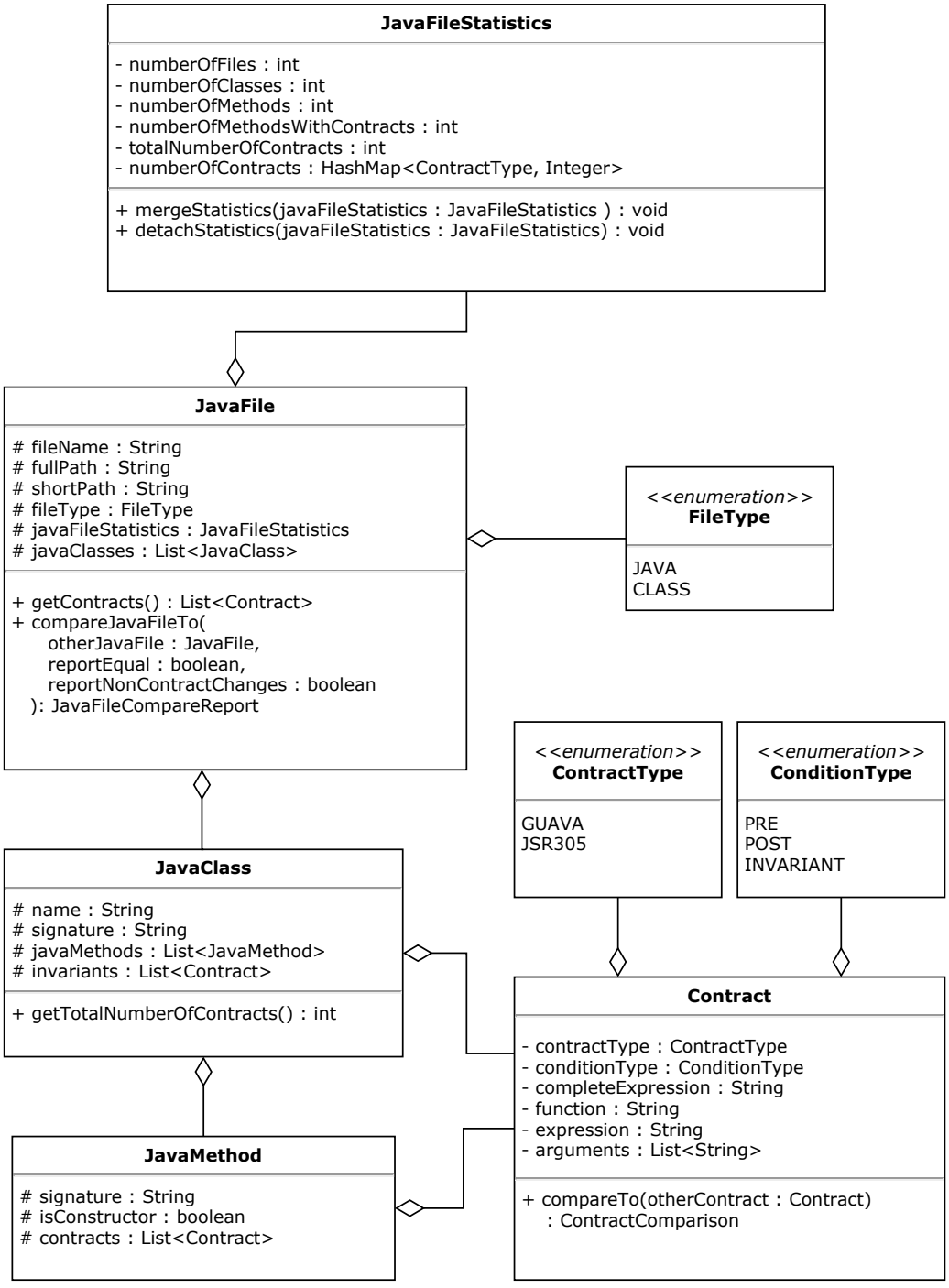
Rozhodl jsem se tedy vytvořit strukturu podobnou stromu, jejíž kořenem je samotný zdrojový soubor (třída `JavaFile`). Tento soubor obsahuje různé podrobnosti o tomto souboru jako je jeho jméno a cesta, typ a statistiky o jeho obsahu. Také obsahuje seznam všech tříd (třída `JavaClass`) obsažených v tomto souboru.

Každá jednotlivá třída pak obsahuje jméno, svou hlavičku, seznam metod (třída `JavaMethod`) a také seznam všech kontraktů týkajících se této třídy, tedy neměnných podmínek. Metoda pak nese informaci o své signatuře a také seznam všech kontraktů (třída `Contract`) dané metody. Samotný kontrakt pak obsahuje informaci o tom, o jaký typ kontraktu a podmínky se jedná, kompletní výraz a také jeho dílčí části (viz podrobnosti níže).

V projektu je model reprezentován v rámci knihovny nástroje. Je zde balíček `model`, kde se nacházejí všechny zmíněné třídy.

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 porov-

návání kontraktů, který je na obrázku 5.2.



Obrázek 5.1: UML diagram datového modelu pro extrakci kontraktů

5.3.1 Podrobnosti modelu

Následuje rozbor některých prvků modelu, které nemusejí být na první pohled samovysvětlující či je vhodné je vyzdvihnout kvůli své důležitosti.

Atribut `shortPath`

Atribut `shortPath` ve třídě `JavaFile` slouží k zobrazení zkrácené cesty souboru, což zlepšuje přehlednost uživatelské aplikace. Pokud jsou přidány soubory ze stejné složky, které jsou hluboko v hierarchii souborového systému, název souboru by mohl být příliš dlouhý, aniž by poskytoval užitečnou informaci.

Např. místo toho, aby byl soubor reprezentován dlouhou absolutní cestou `C:/test/guava-10.0/com/google/common/base/Objects.java`, zobrazí se pouze `base/Objects.java`. Jiný soubor pak může být reprezentován např. pomocí `util/concurrent/Atomics.java`. Zkrácená cesta se průběžně mění na základě přítomných souborů, aby bylo vždy možné je jednoznačně rozeznat. V případě potřeby, či aktualizace této cesty, je k dispozici absolutní cesta v atributu `fullPath`.

Třída `JavaFileStatistics`

Atribut `totalNumberOfContracts` obsahuje informaci o celkovém počtu kontraktů nehledě na typ reprezentace. Naopak hash mapa `numberOfContracts` poskytuje celkový počet kontraktů pro daný typ reprezentace.

Metody `mergeStatistics()` a `detachStatistics()` slouží ke sloučení respektive odloučení dané statistiky od aktivní statistiky. Tyto metody jsou využity při kalkulaci globálních statistik, kde je nutné provést aktualizaci vždy po přidání resp. odebrání souborů.

Třída `Contract`

Klíčovou částí celého modelu je jednoznačně třída `Contract`. Instance této třídy představují jednotlivé kontrakty souboru. Kontraktem je zde myšlena jedna vstupní, výstupní či neměnná podmínka doprovázena dalšími atributy. Následuje krátký popis a vysvětlení jednotlivých atributů této třídy.

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

`ConditionType conditionType` Opět výčtový 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 samotné zprávy zde také často bývají proměnné použité v šabloně zprávy.

Další třídy

Během zpracování byly také používány následující třídy: `ExtendedJavaFile`, `ExtendedJavaClass` a `ExtendedJavaMethod`. Ty obsahují dodatečné informace vůči svým protějškům ve výše zmíněném modelu. Navíc obsahují informace o anotacích, vstupních parametrech a jednotlivých částech 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 cílem nástroje nebyla pouze extrakce kontraktů, ale také jejich případné porovnání, bylo třeba vytvořit datový model, který umožní zachytit i tyto informace. Model jsem navrhoval s vědomím, že porovnání bude mít největší hodnotu ve chvíli, kdy bude možné porovnat kontrakty dvou projektů, tedy dvou složek. Samozřejmě nemá velký smysl

porovnávat kontrakty dvou naprosto rozdílných projektů. Myšlenkou je porovnání dvou různých verzí téhož projektu. Pak je možné pozorovat, zda kontrakty přibývají či ubývají s rostoucím množstvím kódu a obecně různé trendy, které mohou poskytnout zajímavé informace o této problematice.

S porovnáním dvou složek však také přibývá problém párování jednotlivých souborů, tříd, metod a kontraktů se svými protějšky v druhé složce. Na základě těchto informací jsem se tedy rozhodl, že hlavním objektem pro porovnání bude zpráva o rozdílech těchto dvou složek (projektů). Tato zpráva bude obsahovat různé podrobnosti o tom, zda byly složky shodné, konkrétní případy, kde se lišily a také seznam souborů, ke kterým nebyl nalezen adekvátní pár v druhé složce. Tím vzniká tedy seznam přidaných respektive odebraných souborů podobně jako u porovnávání dvou verzí při práci s VCS[1]. Stejně jako v případě extrakce, i zde je souhrn statistik, který poskytuje dodatečné informace o tomto srovnání.

Porovnání jednotlivých souborů jsou pak zachycena v objektu, který má 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. Model by měl být patrný z z obrázku 5.2, kde je znázorněn graficky formou UML diagramu. Následuje také popis některých částí modelu, které nemusejí být na první pohled jasné, či se jedná o důležité prvky.

V projektu je pak tento model opět součástí knihovny a nachází se v balíčku pro porovnávání (`comparator.comparatormodel`).

5.4.1 Podrobnosti modelu

Atributy `apiEqual` a `contractEqual`

Tyto atributy určují, zda dvě složky či soubory jsou shodné z hlediska API respektive kontraktů. Atribut `apiEqual` tedy značí, zda jsou dané objekty shodné z hlediska definice tříd a jejich metod. Pokud například jeden soubor obsahuje metodu, kterou ten druhý nemá, či ji obsahuje, ale má pozměněnou signaturu, hodnotou `apiEqual` by bylo `false`.

Atribut `contractEqual` značí, zda jsou dané objekty shodné z hlediska kontraktů. Tedy jestli jich obsahují stejný počet a zda jsou jednotlivé kon-

[1]VSC, neboli Version Control System je druh nástroje, který umožňuje týmu spravovat kód v průběhu času.

trakty shodné.

Třída `ApiChange`

Pomocí instancí této třídy je možné vyjádřit změny v API. Atribut `apiType` určuje, zda se změna týká třídy či metody. `apiState` pak určuje, zda byl daný element přidán, odebrán či tzv. `FOUND_PAIR`. V tom případě se nejedná o změnu, ale naopak byl nalezen protějšek daného objektu. Pomocí instancí této třídy je pak vyjádřeno, které metody a třídy chybějí či se naopak nově objevily.

Třída `ContractComparison`

Tento výčtový typ představuje výsledek po porovnání dvou kontraktů. Může obsahovat tyto hodnoty:

`EQUAL` Tento výsledek značí, že jsou oba kontrakty naprosto shodné.

`MINOR_CHANGE` Tento stav značí, že kontrakty jsou shodné, ale liší se v ostatních argumentech. Typicky se jedná o změnu zprávy o chybě či různá upřesnění. Z tohoto důvodu se jedná o *minor change* (menší změnu).

`SPECIALIZED` Tímto tvarem je možné vyjádřit, že daný kontrakt je shodný jako ten druhý, nicméně má přísnější podmínku, je tedy více specializovaný. Triviálním příkladem může být situace, kdy první kontrakt má podmínku `x >= 0` a ten druhý podmínku `x > 0`. Zde je vidět, že druhá podmínka je přísnější. V současném stavu nástroje není tato hodnota používána, byla však připravena pro budoucí rozšíření.

`GENERALIZED` Jedná se o stejnou hodnotu jako `SPECIALIZED` s tím rozdílem, že je kontrakt naopak více obecný a má tedy mírnější podmínku. Stejně ani tento stav není momentálně využit.

`DIFFERENT_EXPRESSION` Tímto stavem je vyjádřeno, že jsou dané kontrakty typově shodné, liší se však v atributu `expression`. Obvykle tedy platí, že se liší v podmínce, či v objektu, kterého se daný kontrakt týká. V současném stavu tento stav zahrnuje také stavy `SPECIALIZED` i `GENERALIZED`. Do budoucna by však stav `DIFFERENT_EXPRESSION` měl být přípustný pouze pokud ani z dvou zmíněných stavů není možné prokázat (např. podmínku `y == 1` nelze porovnávat s podmínkou `x > 0`).

`DIFFERENT` Tento výsledek říká, že se oba kontrakty liší v některém ze základních atributů, tedy typ kontraktu či podmínky. Při porovnávání dvou souborů či složek by nemělo být možné tohoto stavu docílit, protože není možné nalézt pár k danému kontraktu, jestliže se takto výrazně liší. Důvodem tohoto stavu je zajištění úplnosti v případě porovnávání dvou kontraktů bez známého kontextu.

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 ukládaní objektů typu Java. JSON je tak vhodný pro zpracovaní 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

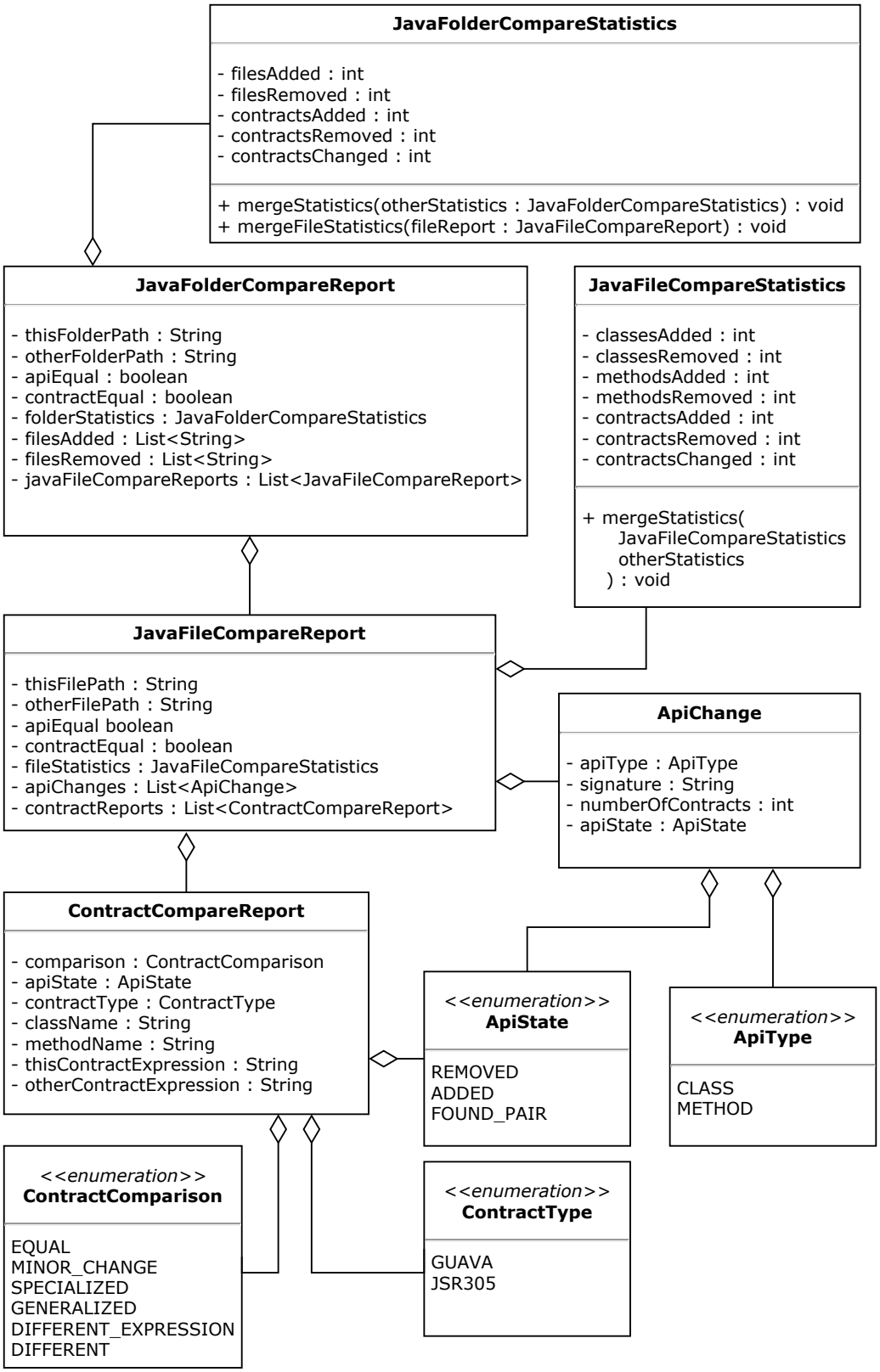
Formát JSON [39] je velice jednoduchý a stručný. Objekty reprezentované v JSON jsou uzavřeny ve složených závorkách (`{}`), jelikož celý soubor vždy představuje jeden komplexní objekt, daný soubor vždy začíná i končí složenou závorkou. Názvy dílčích atributů jsou pak uzavřeny v uvozovkách a za dvojtečkou následuje jejich hodnota. Více atributů je pak odděleno čárkami.

Daný atribut může mít přirozeně různé typy hodnot. Řetězce jsou standardně uvnitř uvozovek, čísla a výrazy `true/false` jsou pak bez uvozovek. Hodnotou však může být i další objekt či pole. Pole je uzavřeno v hranatých závorkách (`[]`) a položky jsou odděleny čárkami. Pokud je pole prázdné, jsou zde pouze obě závorky bez obsahu.

Formát exportovaných souborů přímo vychází z výše zmíněných modelů. Daný soubor obsahuje tedy všechny objekty uvedené v modelu a je pouze

převeden do formátu JSON. V následujícím příkladu je ukázka formátu souboru, který je vytvořen po extrakci kontraktů. Příklad je úmyslně zkrácen, v atributu `javaMethods` by standardně byly jednotlivé metody a v každé z daných metod pak případné kontrakty. Také by zde byly další atributy jako jsou statistiky, typ souboru atd.

```
{
  "fileName": "example.java",
  "fullPath": "C:/test/example.java",
  ...
  "javaClasses": [
    {
      "name": "Example"
      "signature": "public class Example"
      "javaMethods": [
        ...
      ],
      "invariants": []
    }
  ]
}
```



Obrázek 5.2: UML diagram datového modelu pro porovnání kontraktů

6 Nástroj pro analýzu kontraktů

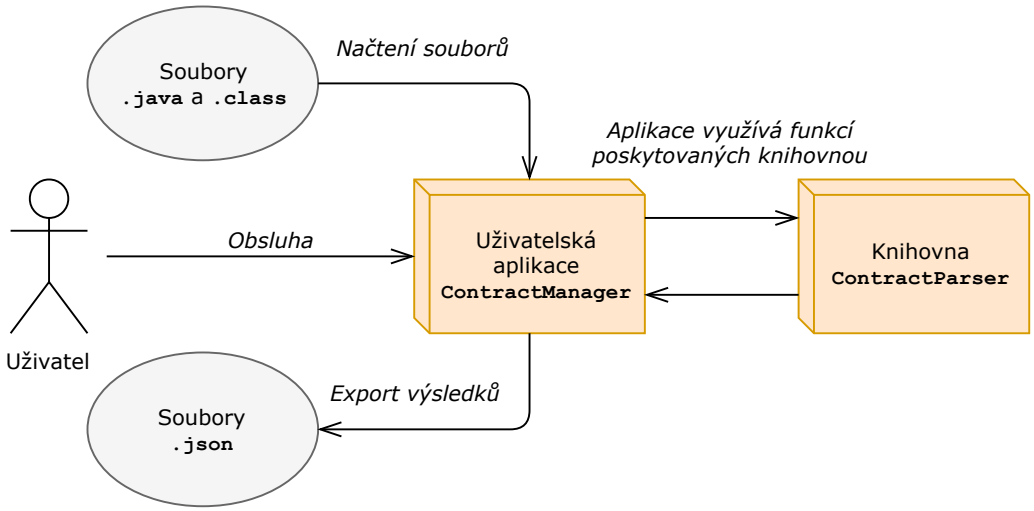
Cílem práce z hlediska implementace bylo vytvořit nástroj, který by umožňoval získání informací o kontraktech podle výše navrženého modelu ze zdrojové či přeložené formy Java programu. Výsledná aplikace by pak měla umožnit vytvoření externí reprezentace dat a případně také porovnání DbC konstrukcí. Nástroj 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 se soubory, což může být užitečné zejména při porovnání různých verzí projektu. Můžeme tak zjistit, zda se nezměnilo rozhraní či zda se zpřísnily podmínky kontraktů oproti předchozí verzi.

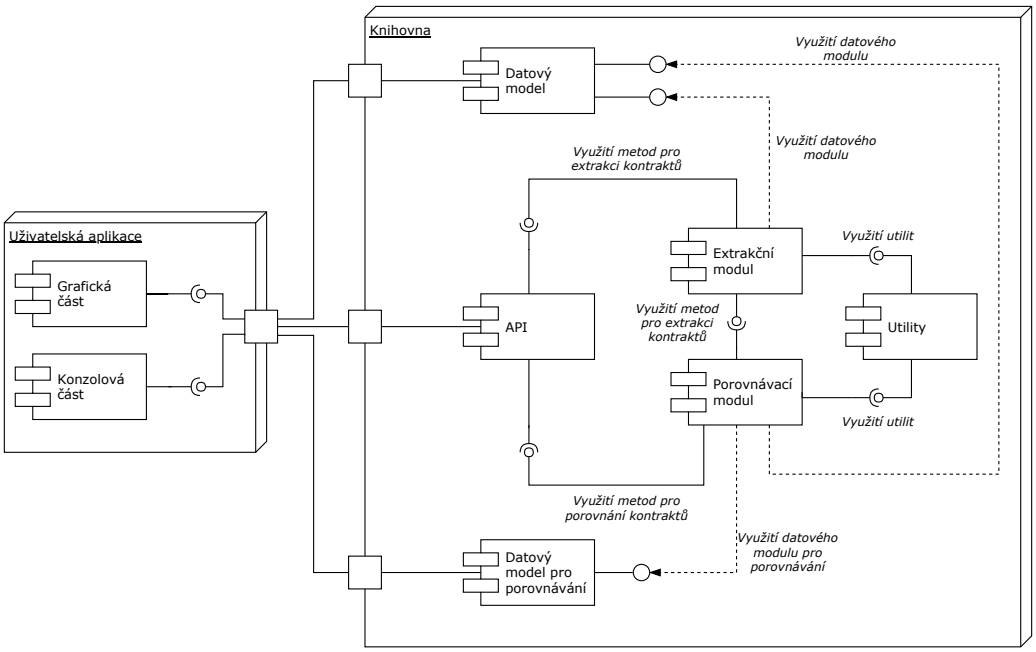
Celý nástroj je rozdělen do dvou velkých částí. První z nich je knihovna, která poskytuje všechny potřebné metody pro práci se soubory jazyka Java, jejich analýzu a zpracování, extrakci kontraktů a jejich následné porovnání a export. Druhou částí je uživatelská aplikace, která využívá metod této knihovny a umožňuje její pohodlnou obsluhu. Tento vztah je znázorněn na stručném obrázku 6.1. Na obrázku 6.2 je pak vidět komponentový diagram, který obsahuje více podrobností.

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 model použitý pro jejich reprezentaci.



Obrázek 6.1: Stručná architektura nástroje



Obrázek 6.2: Komponentový diagram nástroje

6.1.1 Použité technologie

Programovací jazyk

Pro tvorbu 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. Díky těmto okolnostem 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 [40]. 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 [41]. 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 [29]. Učinil jsem tak na základě mého průzkumu (viz kapitola 4 - Analýza kódu 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 [37]. 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 pomocí API. Více informací je opět k dispozici ve 4. kapitole.

Práce s formátem JSON

Pro ukládání reprezentací do formátu JSON byla použita knihovna Gson [42]. Umožňuje intuitivní převod objektu typu Java[1] do formátu JSON za pomocí API. Mimo jiné také umožňuje formátování *Pretty Print*, které je lépe čitelné pro člověka.

Testování

Pro tvorbu jednotkových testů byla použita technologie jUnit 5 [43].

6.1.2 Návrh nástroje

Celý nástroj je rozdělen do několika modulů, viz obrázek 6.2. Oba datové modely byly popsaný výše v 5. kapitole. Parsovací modul je klíčovou součástí celého nástroje. Umožňuje zpracování kódu jazyka Java a jeho analýzu. Poskytuje prostředky pro uležení tohoto analyzovaného kódu do navržené struktury a následnou extrakci kontraktů. Porovnávací modul umožňuje porovnávání složek a souborů z hlediska kontraktů a API. Utility poskytují metody pro práci se zdroji a soubory. API je navržena pro snadné vnější použití celé knihovny. Jednotlivé moduly jsou popsány níže.

6.1.3 Parsovací modul

Tento modul zajišťuje zpracování zdrojových souborů jazyka Java a následné uložení do reprezentace `ExtendedJavaFile`. To má na starosti třída `JavaFileParser`, která také využívá třídy `MethodVisitor`. Pokud je vstupní soubor typu `.class` je nejprve dekompilován. Dekompilaci, stejně tak jako ostatní práci se soubory, zajišťuje třída `IOServices` (viz níže).

Třída `ExtendedJavaFile` je součástí modelu pro parsování, který dědí od svých rodičovských tříd ze základního datového modelu výše (zde konkrétně je děděno od třídy `JavaFile`). Dále je zde třída `ExtendedJavaClass` a `ExtendedJavaMethod`. Tento rozšířený model ukládá dodatečné informace jako je tělo metod, anotace atd. Z těchto informací jsou následně získávány

[1] Mohou být použity téměř libovolné objekty, avšak nesmějí být cyklické (Objekt nesmí ve své hierarchii atributů opět obsahovat tentýž objekt)

kontrakty.

Vytvořená struktura je následně procházena a jsou z ní extrahovány kontrakty. To zajišťují třídy `GuavaParser` a `JSR305Parser` (každá svůj cílený typ kontraktu). Tyto třídy implementují rozhraní `ContractParser` a továrna `ParserFactory` umožňuje vytváření jejich instancí. Tímto způsobem se instance objektu `ExtendedJavaFile` postupně rozšiřuje.

Poté co byly extrahovány všechny kontrakty, podrobnosti o metodách a třídách jsou již redundantní a z toho důvodu je objekt `ExtendedJavaFile` zredukován na `JavaFile`. Totéž platí pro zbytek modelu. K této redukci slouží třída `Simplifier`.

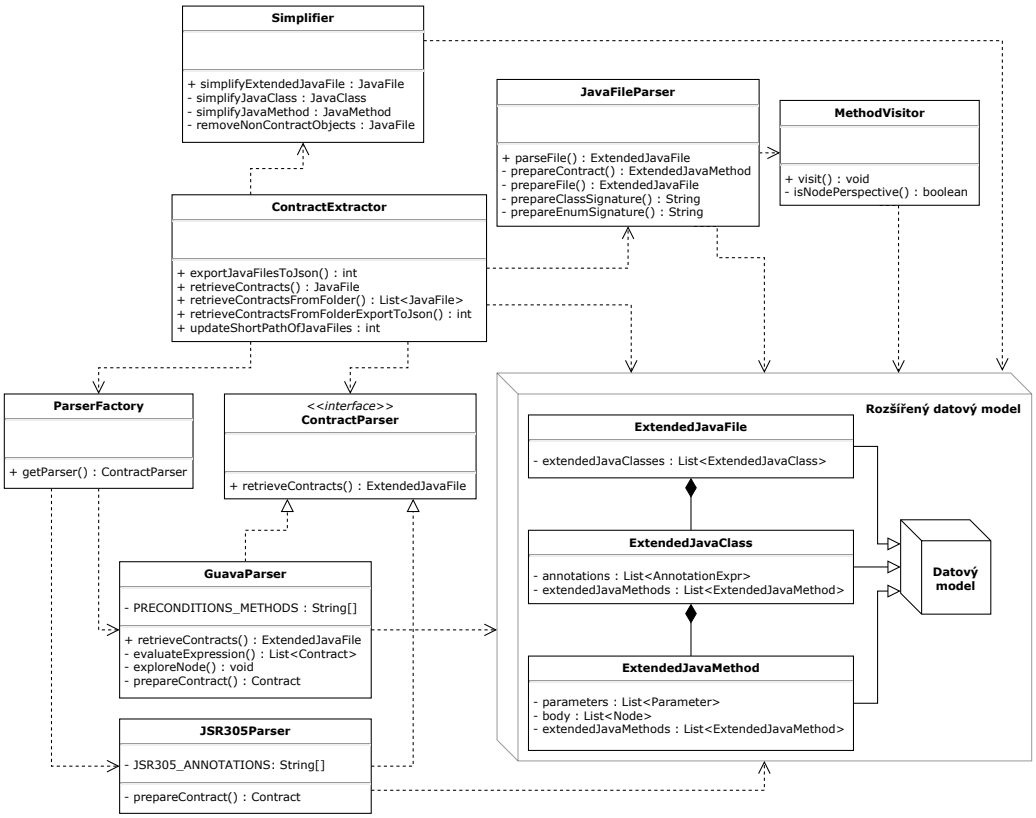
Poslední částí tohoto modulu je třída `ContractExtractor`. Ta slouží jako výchozí bod pro komunikaci s ostatními částmi nástroje a spojuje dílčí třídy a jejich funkce dohromady. V podstatě se jedná o API pro vnitřní užití knihovny.

Graficky je možné vidět tento modul v podobě UML diagramu, který je zobrazen na obrázku 6.3. Jedná se o zjednodušený UML diagram tříd, kde jsou jednotlivé metody omezeny pouze na svůj název a návratovou hodnotu. Důvodem je to, že metody mají mnoho vstupních parametrů, což by činilo diagram nepřehledný. Podrobnosti o konkrétních třídách je pak možné získat z JavaDoc dokumentace (na CD ve složce `javaDoc`). Rozšíření model je také zjednodušen a jsou zde zobrazeny pouze atributy, které nemají rodičovské třídy zobrazené v modelu výše. K datovému modelu je také přistupováno jako k celku, aby se zmenšilo množství závislostních vazeb (mnoho tříd pracuje s většinou částí modelu). V projektu je pak modul reprezentován balíčkem `parser`

Základní pracovní postup pro extrakci kontraktů je možné vidět na obrázku 6.4. Jedná se o jednoduchý diagram, který slouží jako grafický podklad k popisu jednotlivých činností a jejich návazností. Diagram slouží pouze pro rychlou orientaci a přehled v procesu, proto tu nejsou znázorněny nestandardní aktivity jako zpracování chyb atd.

Parsování Java souborů

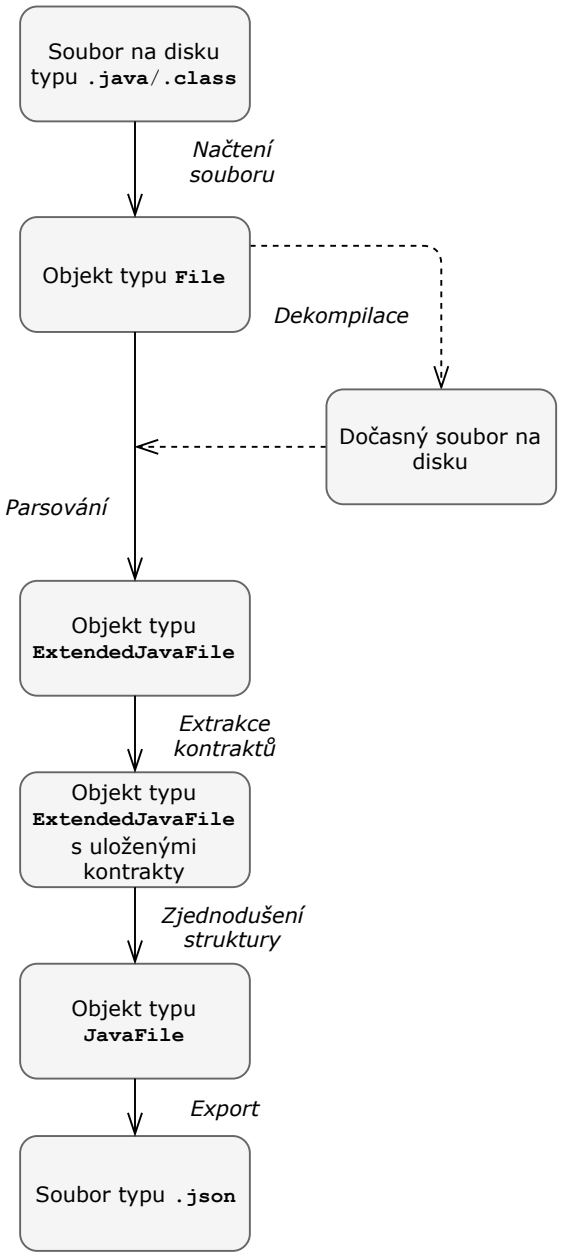
Pro zpracování zdrojových souborů jazyku Java byla použita knihovna JavaParser. Ta poskytuje metodu `parse()`, která vytvoří komplexní strukturu



Obrázek 6.3: UML diagram parsovacího modulu

daného zdrojového souboru. V prvním kroku se tato struktura projde a vyhledá všechny třídy (*class*) a také rozhraní *interface* a výčtové typy *enum*. Pro účely modelu jsou si tyto tři prvky rovny. Každý nalezený prvek je následně uložen do modelu. V případě třídy a rozhraní 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. Vstupuje také rodičovský objekt `ExtendedJavaFile`, do kterého jsou získané údaje uloženy. 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



Obrázek 6.4: Základní pracovní postup

objektu `ExtendedJavaMethod`, která je následně uložena do své rodičovské `ExtendedJavaClass` (ta je již součástí vstupního `ExtendedJavaFile`).

Extrakce kontraktů

Obecně Poté, co je z Java souboru vytvořen objekt `ExtendedJavaFile`, je možné začít extrahovat kontrakty. Během získávání kontraktů se tato struktura prochází a postupně se k jednotlivým třídám a metodám přidá-

vají kontrakty. Poté, co jsou všechny extrakce dokončeny, je za pomocí 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 obsahuje název některé z metod třídy `Preconditions`, je tento výraz dále zpracováván. Název Guava metody je uložen do kontraktu jako atribut `function`. První parametr metody, ten klíčový, je uložen jako atribut `expression`. Ostatní parametry, obvykle souvisejí pouze s tvarem chybové zprávy, jsou uloženy do seznamu `arguments`.

Nástroj umožňuje rozpoznání těchto metod: `checkArgument`, `checkState`, `checkNotNull`, `checkElementIndex`, `badElementIndex`, `checkPositionIndex`, `badPositionIndex`, `checkPositionIndexes`, `badPositionIndexes`.

Vysvětlení, použití a jiné podrobnosti jednotlivých metod je možné zjistit v dokumentace knihovny. I přesto, že projekt v současné podobě umožňuje rozpoznat všechny metody Guava Preconditions, časem mohou přibýt jiné konstrukce, které bude třeba do nástroje doplnit. Vzhledem k tomu, že Guava je stále živý projekt, který se vyvíjí, je třeba kontrolovat nové verze, zda nepřidávají nové metody pro reprezentaci kontraktů. Názvy metod jsou definovány v konstantě `PRECONDITIONS_METHODS`, což je pole řetězců.

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 často parametr nemají. Takto nalezené kontrakty v anotacích třídy jsou označeny za neměnné proměnné a v anotacích metod se pak jedná o výstupní podmínky. 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, takto se extrahují vstupní podmínky.

Nástroj umí rozpoznávat následující anotace: `CheckForNull`, `CheckForSigned`, `CheckReturnValue`, `Detainted`, `MatchesPattern`, `Nonnegative`, `Nonnull`, `Nullable`, `OverridingMethodsMustInvokeSuper`, `ParametersAreNonnullByDefault`, `ParametersAreNullableByDefault`, `PropertyKey`, `RegEx`, `Signed`, `Syntax`, `Tainted`, `Untainted`, `WillClose`, `WillCloseWhenClosed`, `WillNotClose`.

Jedná se o všechny anotace, které má knihovna k dispozici v poslední verzi. Jejich vysvětlení, použití a jiné podrobnosti je možné zjistit v dokumentace knihovny. Vzhledem k tomu, že nástroj je již delší dobu beze změn, není pravděpodobné, že se v blízké době tento seznam bude měnit.

6.1.4 Modul utilit

Tento model obsahuje pouze dvě třídy, `IOServices` a `ResourceHandler`. `ResourceHandler` je třída, která poskytuje metody pro snadnou práci se zdroji jako jsou zobrazované chybové zprávy a *properties* (konfigurovatelné vlastnosti). `IOServices` pak poskytuje metody pro práci se soubory. Je tu tedy např. metoda, která vrátí seznam souborů v dané složce, získání koncovky souboru, kontrola, zda se jedná o soubor či složku atd. Jsou zde také prostředky pro export do formátu JSON a také metoda pro dekompilaci přeložených souborů. Modul je reprezentován balíčkem `utils`.

Dekompilace souborů `.class`

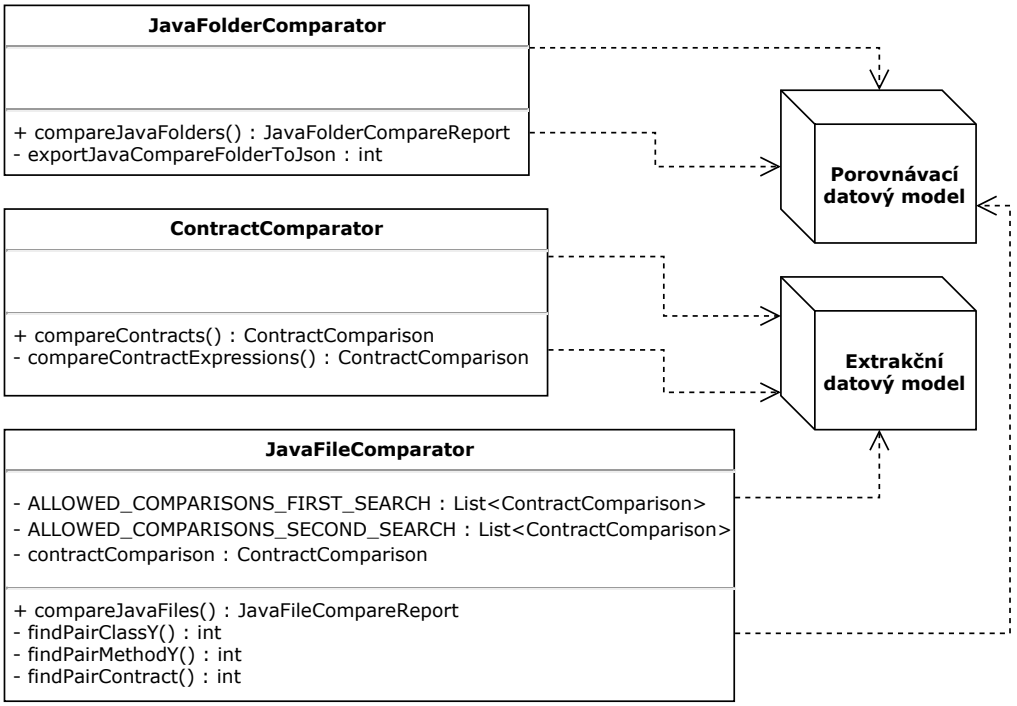
Jak již bylo zmíněno výše, pro dekompilaci Java `.class` souborů byla použita knihovna Procyon. Ta poskytuje metodu `void decompile(String internalName, ITextOutput output)`, která přečte vstupní soubor s přeloženým kódem a do jiného souboru uloží jeho dekompilovanou verzi. V mém nástroji dekompilaci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

6.1.5 Porovnávací modul

Modul pro porovnávání kontraktů se skládá ze tří tříd a modelu pro porovnávání. Struktura tohoto modulu je vidět na obrázku 6.5. Model byl představen výše (viz kapitola 5 - Datový model) a nebude zde proto opět detailně rozebírán. Dílčími třídami pak jsou `ContractComparator`, která specifikuje porovnávání samotných kontraktů, `JavaFileComparator` pro porovnávání souborů a nakonec `JavaFolderComparator`, který umožňuje porovnávání složek. Porovnávací modul je v projektu uložen v balíčku `comparator`.

Porovnávací metoda `compareContracts()` z `ContractComparator` je využita v modelové třídě `Contract`, kde je volána v metodě `compareTo()`. To samé platí pro metodu `compareJavaFiles()` ze třídy `JavaFileComparator`, která je volána v `JavaFile` v metodě `compareJavaFileTo()`. Díky tomu je možné kontrakty a soubory snadno porovnávat pomocí konstrukce jako např. `contractX.compareTo(contractY)`.

Na obrázku 6.2 je vidět UML diagram pro tento modul. Stejně jako diagram pro parsování, je i tento zjednodušení. Níže je pak stručný popis jednotlivých porovnání.



Obrázek 6.5: UML diagram porovnávacího modulu

Porovnávání kontraktů

Při porovnávání kontraktů se nejprve zkontroluje, zda je shodný typ reprezentace kontraktu, typ podmínky a také funkce. Jestliže je nějaká z těchto položek negativní, kontrakty jsou považovány za rozdílné (`DIFFERENT` dle výčtového typu `ContractComparison`). Pokud jsou shodné, je porovnán výraz obou kontraktů. Jestliže je výraz rozdílný, je vrácena hodnota `DIFFERENT_EXPRESSION`, pokud ne, zkontrolují se ostatní argumenty. Pokud je některý z nich rozdílný, či je jich rozdílný počet, je navrácena hodnota `MINOR_CHANGE`). Pokud byly všechny testované položky v pořádku, je vráceno `EQUAL`.

Podrobnosti o modelu a jednotlivých hodnotách porovnání byly zmíněny výše v 5.kapitole. Je zde velký prostor pro zlepšení v oblasti porovnávání výrazů, to je více rozebráno níže v kapitole 8 - Zhodnocení výsledků.

Porovnávání souborů

Porovnávání kontraktů v souborech není triviální z důvodu párování jednotlivých kontraktů. Pokud je změněno pořadí kontraktů a změní se jejich definice, je obtížné přiřadit daný kontrakt ke svému protějšku v druhém souboru.

Algoritmus nejprve prochází všechny třídy daného souboru. Pokud se podaří najít shodnou třídu, pokračuje se dále. V opačném případě je přidána do seznamu změn API odebraná třída. Jestliže ale třída byla nalezena, prochází se kontrakty dané třídy (tedy neměnné podmínky).

Párování jednotlivých kontraktů funguje následujícím způsobem. Aktuální kontrakt se hledá v seznamu kontraktů protější třídy, tím že se jednotlivé kontrakty porovnávají. Pokud je výsledkem `EQUAL` nebo `MINOR_CHANGE`, kontrakt je považován za nalezený. V tom případě se přidá záznam o jejich porovnání dle získané hodnoty a kontrakt je vymazán ze seznamu kontraktů protějšku, aby jej nebylo možné znova spárovat. Pokud kontrakt nebyl nalezen, je přidán do seznamu nenalezených kontraktů a hledá se další kontrakt.

Jakmile byly vyhledány všechny kontrakty, prochází se seznam nenalezených kontraktů. Opět se daný kontrakt snažíme vyhledat v seznamu protější třídy, ten je však již redukován. Nyní je shodu považován jakýkoliv výsledek kromě `DIFFERENT`. Pokud je protějšek nalezen, je považováno, že se kontrakt změnil a je přidán do seznamu porovnání. Pokud kontrakt nebyl nalezen, je

předpokládáno, že byl odstraněn a je přidán záznam o odebrání kontraktu.

Toto párování kontraktů je heuristika, při které se může stát, že by se kontrakt spároval s jiným, protože by bylo změněno pořadí kontraktů a zároveň by byly naprosto shodné až na argumenty (tím by se vrátila odpověď `MINOR_CHANGE`). Tento scénář je však velmi nepravděpodobný, protože kontrakty, které se liší pouze v ostatních argumentech by byly velice nezvyklé. I v případě, že by tato situace nastala, bylo by možné tuto záměnu poznat ze zprávy, kde by byl dvakrát registrován rozdíl v kontraktech. Vzhledem k tomu, že tyto situace, které mohou nastat jsou vysoce nepravděpodobné, a i když nastanou, je možné je dohledat ve změnách, rozhodl jsem se tuto heuristiku použít. Díky tomu je možné dosáhnout mnohem lepších výsledků, protože není tolik kontraktů považovaných za nenalezené.

Po porovnání neměnných podmínek se prochází jednotlivé metody této třídy. Zde pak probíhá stejný proces jako u párování tříd. Poté se procházejí jednotlivé kontrakty metody. Zde opět funguje stejný princip jako v případě neměnných podmínek.

Následně se zkontroluje, zda některé kontrakty nezbyly v seznamu, pokud ano, jsou registrovány jako nově přidané kontrakty. Metoda je následně vymazána ze seznamu ze stejného důvodu a stejným způsobem jsou ohlášeny všechny metody, které zbyly jako nově přidané. Totéž platí pro třídy a neměnné podmínky. Přidáním jednotlivých záznamů o přidaných/odebraných metodách, třídách a kontraktech (v případě kontraktů také o změněných), jsou vytvořeny seznamy, které pak tvoří závěrečnou zprávu porovnání obou souborů `JavaFileCompareReport` (viz model výše).

Porovnávání složek

Porovnávání složek funguje podobným způsobem jako porovnávání souborů. Respektive stejně jako byly párovány jednotlivé třídy, metody a kontrakty, jsou zde párovány soubory. Je zde tedy také zaznamenáno, které soubory byly přidány a odebrány. Součástí je také seznam jednotlivých zpráv o porovnání souborů. Tím se tvoří `JavaFolderCompareReport`, který je výsledkem tohoto porovnání.

Na rozdíl od předchozího případu je zde nutné nejprve upravit cesty jednotlivých složek, respektive souborů, tak, aby jejich relativní cesta byla shodná.

6.1.6 API Modul

Knihovna obsahuje API pro snazší přístup zvnějška. Skládá se z několika částí. Je zde třída `ApiFactory`, která umožňuje instancování jednotlivých API, jedná se o návrhový vzor továrny. V nástroji jsou definovány celkem čtyři různé typy API, každý z nich implementuje specifické rozhraní, pomocí kterého je možné využít továrny. V nástroji je pouze jedna implementace pro každý typ, továrna a rozhraní jsou zde tedy pouze pro přehlednost a možnost snadného rozšíření.

Zde je seznam všech čtyř typů API podle názvů jejich rozhraní. Implementační třídy pak mají stejný název, ale s přidanou předponou `Default`:

- `ContractExtractorApi`
- `BatchContractExtractorApi`
- `ContractComparatorApi`
- `BatchContractComparatorApi`

Následuje stručný výčet toho, co jednotlivá API umožňují. Pro přesný popis metod, jejich parametrů a návratového typu je možné se podívat do JavaDoc projektu.

`ContractExtractorApi`

Zde jsou zejména metody pro získání kontraktů. Metoda `retrieveContracts` umožňuje extrakci kontraktů ze vstupního souboru. Je možné definovat, které typy kontraktů se mají analyzovat a zda mají být vráceny i objekty bez kontraktů. Vrácena je instance `JavaFile`, která obsahuje extrahované kontrakty. Další metodou je `retrieveContractsFromFolder`, která poskytuje stejnou funkcionalitu s tím rozdílem, že vstupem je celá složka nikoliv soubor a je vrácen seznam `JavaFile`.

Následně je zde `exportJavaFilesToJson`. Tato metoda slouží k exportu vstupního seznamu `JavaFile` do JSON. Je třeba nastavit výstupní složku a také, zda má být formát v minimalistické formě. Poslední metodou pak je `updateShortPathOfJavaFiles`, která aktualizuje atribut `shortPath` všech `JavaFile` v seznamu.

`BatchContractExtractorApi`

Toto API poskytuje podobnou funkcionalitu jako předchozí. Obsahuje pouze jednu metodu a to `retrieveContractsFromFolderExportToJson`. Jedná se v podstatně o kombinaci předchozích metod. Umožňuje načtení kontraktů z dané složky a jejich následný export do zadaného adresáře, součástí je také shodné nastavení. Tato metoda nejprve načte jeden soubor ten zpracuje, exportuje a pak pokračuje dalším, díky tomu je méně náročná na paměť. Toto API je využito pro konzolovou část uživatelské aplikace.

`ContractComparatorApi`

Poskytuje metodu `compareJavaFolders`, která porovná dva zadané adresáře na úrovni kontraktů ale i jejich API. Následně vytvoří report typu `JavaFolderCompareReport` informující o tomto srovnání. Je možné zvolit, zda se mají reportovat shodné objekty a také jestli se mají reportovat změny API, které přímo neovlivňují kontrakty. Druhou metodou je pak `exportJavaFolderCompareReportToJson`, což umožňuje vytvořený report exportovat do JSON.

`BatchContractComparatorApi`

Toto je poslední dostupné API. Opět poskytuje obdobné chování jako předchozí API, s tím rozdílem, že je určeno pro dávkové zpracování a kombinuje obě metody do jedné(`compareJavaFoldersAndExportToJson`). Tato metoda porovná obě složky a výslednou zprávu následně exportuje do JSON. Stejně jako pro předchozí API tohoto typu i zde platí, že je použito v konzolové části aplikace.

Použití API

Použití API je velmi snadné, jak je vidět na následujícím příkladu:

```
// Získání instance API pomocí továrny
ApiFactory f = new ApiFactory();
ContractExtractorApi api = f.getContractExtractorApi();

// Nyní je možné využít metody knihovny
api.retrieveContracts(...);
```

6.1.7 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 rozpoznávání nové reprezentace kontraktu jsou potřeba tyto kroky:

1. Přidání položky do `ContractType`
2. Vytvoření nového analyzátoru
3. Doplnění továrny `ParserFactory`
4. Testování

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. Kontext je dobře vidět na diagramu datového modelu (obrázek 5.1 v předchozí kapitole).

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.

Podrobnosti mohou být patrné z diagramu 6.3, kde jsou zobrazeny všechny zmíněné komponenty. Nový analyzátor by pak byl na úrovni `GuavaParser` či `JSR305Parser`.

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`.

Testování

Pro bezchybnou funkci daného analyzátoru je možné vytvořit jednotkové testy. Testovací data jsou umístěna v `resources/testFiles`, kde jsou pak dále děleny do složek.

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, které byly popsány výše.

ControlsFX Tato knihovna rozšiřuje JavaFX a umožňuje použití dalších funkcí a objektů zejména pak `CheckListView`, což je použito pro zobrazení seznamu souborů [44].

FontAwesomeFX Knihovna FontAwesomeFX slouží opět k rozšíření JavaFX. Tuto knihovnu jsem použil pro rozšíření možností zobrazení ikon [45].

6.2.2 Design grafické části aplikace

Grafická část aplikace byla rozdělena na dvě části dle její funkčnosti. První z nich je extrakční, ta umožňuje extrakci kontraktů ze souborů a jejich následný export. Také je možné prohlížet si podrobnosti o jednotlivých souborech. Druhá část slouží k porovnávání kontraktů. Mezi těmito částmi je možné přepínat pomocí záložky.

Každá část disponuje panelem nástrojů, pomocí kterého je možné přidávat, mazat a exportovat označené soubory. Dále je zde panel filtrů, kde lze nastavit různé podrobnosti ohledně zobrazených dat. Obě části také mají seznam souborů, kde jsou zobrazeny všechny soubory, které byly přidány (může být ovlivněno filtry). Jednotlivé soubory pak mohou být označeny pro výše zmíněné mazání a export. Součástí obou částí jsou také globální statistiky, které zobrazují různé podrobnosti a shrnutí vybraných souborů. Vidět je také detail právě vybraného souboru. Zde je možné zobrazit si podrobnosti a otevřít tak nové okno aplikace, kde jsou detailní informace o daném souboru, včetně stromové reprezentace daného souboru.

Tato sekce představuje pouze krátké shrnutí toho, jak je aplikace členěna a co umožňuje pro pochopení kontextu. Pro podrobnější popis je možné nahlédnout do přílohy A Uživatelská příručka, kde jsou podrobně vysvětleny všechny funkce za pomoci obrázků aplikace.

6.2.3 Struktura aplikace

Uživatelská aplikace je rozdělena do několika částí dle balíčků v projektu. Prvním z těchto balíčků je `application`, který má na starosti uchování dat aplikace, jako jsou data seznamů souborů, statistiky, aktuálně vybraný soubor atd. Jsou zde také nastavovány různé funkce pro dané objekty, které jsou vykonávány na základě vnějších podnětů. Příkladem může být aktualizace dat, přidávání souborů do seznamu atd.

Další částí je `controller`, kde se nachází ovládací prvek, jež definuje akce po stisknutí tlačítek . Zde jsou typicky volány akce definované v části `application`.

Následuje část `utils`, která poskytuje různé funkce jako je práce se zdroji, práce se soubory, ale také je zde definována konzolová část aplikace.

Mimo tyto balíčky je zde také třída `ContractManager`, která představuje hlavní třídu celé aplikace a definuje její spuštění. Kontroluje, zda se má aplikace spustit v grafickém režimu či jen vykonat daný příkaz. Je zde také definováno okno a scéna grafické části.

Následuje stručný popis jednotlivých balíčků a jejich důležitých tříd. Další podrobnosti je pak možné nalézt ve vygenerované dokumentaci JavaDoc.

Balíček `application`

Nachází se zde třída `Settings`, která uchovává nastavení filtrů aplikace. Potenciálně je tuto část možné rozšířit o další nastavení, které by mohlo měnit chování či vzhled aplikace. Hlavní třídou balíčku je pak `ApplicationData`. Ta obsahuje data obou záložek i zmíněné nastavení, jedná se o přístupový bod ke zbytku aplikace. Samotné záložky jsou pak definovány dvěma třídami `ExtractorApplicationTab` a `ComparatorApplicationTab`. Ty obě dělí od obecné třídy `ApplicationTab`. Každá z těchto tříd také obsahuje třídu `ExtractorFileList` respektive `ComparatorFileList`, které obsahují detaily o seznamech souborů včetně samotných souborů.

Balíček `controller`

Tento balíček je poměrně přímočarý a obsahuje pouze jednu třídu `Controller`. Zde jsou pak mapovány metody na jednotlivá stisknutí tlačítek.

Balíček `utils`

Třída `ConsoleApplication` představuje konzolovou část aplikace. Zde probíhá zpracování vstupních parametrů a jejich následné předání patřičným metodám. Jsou zde také ošetřeny chybné vstupy a nastavení chybových zpráv.

Dále je zde třída `FileHandler`, která má na starosti výběr souborů také jejich export.

Pomocí třídy `ResourceHandler` je pak možné přistupovat ke zdrojům, které obsahují nastavitelné vlastnosti `properties` a také lokalizované texty uživatelského rozhraní spolu s chybovými a informačními hláškami. Texty jsou k dispozici pouze v angličtině, ale díky přístupu pomocí zdrojů je možné snadno přidat další jazyky.

Samotná třída `Utils` pak obsahuje pomocné metody pro snazší vyhledávání prvků v rozhraní, centrování okna na obrazovku atd.

6.2.4 Ovládání aplikace

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

spustit s parametry, čímž se provede jednorázová akce pro dávkové zpracování. Podrobné informace o spouštění a používání aplikace jsou uvedeny v příloze A. Uživatelská příručka.

Grafická část

Standardním spuštěním aplikace bez parametrů se zobrazí grafická uživatelská část. Zde je možné extrahovat kontrakty, zobrazovat si je v kontextu hierarchie daného souboru, exportovat získaná data a také porovnávat složky za účelem zjištění rozdílů v API a kontraktech. Všechny tyto činnosti je možné obsluhovat pomocí jednoduchého a intuitivního uživatelského rozhraní.

Konzolová část

V případě, že chceme pouze provést jednorázovou akci extrakce či porovnávání kontraktů a následný export, je možné využít konzolové části aplikace, tedy spustit aplikaci s příslušnými parametry (viz Uživatelská příručka). Takto je možné snadno a rychle provádět dávkové operace. Tento způsob zpracování je také méně náročný na paměť zařízení.

6.2.5 Možnosti a limitace aplikace

Při vývoji uživatelské aplikace byl kladen důraz na snadné a intuitivní použití, které poskytne možnost jak vyzkoušet a aktivně využít vytvořenou knihovnu. Nástroj je určen pro výzkumné účely uzavřené skupiny, nikoliv pro použití širší veřejností. Z tohoto důvodu nebyl kladen důraz na její široké možnosti a uživateli může připadat, že postrádá prvky, které jsou typické pro komerční aplikace. Příkladem může být perzistence dat, široké možnosti filtrování a řazení, propojení s externími editory atd. Níže, v podkapitole Prostor pro zlepšení, je této problematice věnována větší část.

6.3 Optimalizace

I přesto, že efektivita nebyla prioritou tohoto projektu, v rámci možností jsem se snažil obě části nástroje zanalyzovat a následně optimalizovat. Díky tomu se mi podařilo zpřehlednit kód, odhalit řadu chyb, snížit nároky na paměť i zrychlit různé algoritmy.

6.3.1 Analýza a refaktoring kódu

V průběhu projektu jsem kód analyzoval kvůli potenciálním možnostem vylepšení a odstranění chyb. To vedlo k refektorování nepřehledných či špatně navržených úseků kódu a v důsledku toho jsem snížil cyklomatičnost některých algoritmů a zpřehlednil kód. Díky tomu jsem také odstranil řadu chybu jako jsou neinicializované proměnné, nekontrolované přetypování atd.

Tuto analýzu jsem prováděl ručně, ale také za pomoci nástrojů vývojového prostředí, které poskytují řadu způsobů, jak zkvalitnit kód.

6.3.2 Zjednodušení modelu

V rámci optimalizace jsem také zjednodušil datový model. Ten původní obsahoval komplexní objekty vytvořené knihovnou JavaParser, což vedlo k vyšším nárokům na paměť a snižovalo to celkovou přehlednost kódu, ale také exportovaných dat. V původním modelu jsem také uchovával informace jako těla metod a anotace, která již nejsou ve finálním modelu potřebná, což opět vedlo ke snížení přehlednosti a zvýšení paměťové náročnosti. Na základě toho vznikly pomocné třídy `extendedJavaFile/Class/Method`, které umožňují uchování důležitých dat pouze během analýzy.

6.3.3 Snížení nároků na paměť pro dávkové zpracování

Ve své původní verzi, konzolová část uživatelské aplikace využívala stejné API jako část grafická. Proto byly dávkové operace zpracovávány tak, že se nejprve všechny položky načetly do paměti a teprve poté byly exportovány. To samozřejmě bylo zbytečně náročné na paměť, protože v dávkovém zpracování není třeba průběžně uchovávat analyzované soubory. Z tohoto důvodu vznikla nová API a metody, které umožňují průběžné zpracování bez zbytečného zatížení paměti.

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 aspekty nástroje a byly použity pro zajištění spolehlivosti výsledného produktu, 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 modulu pro tokenizaci zdrojových souborů. Celkem bylo implementováno 56 jednotkových testů různé komplexity.

7.1.1 Ukázka jednotkového testu

Následuje ukázka jednotkového testu `testGuava()`, který se nachází ve třídě `ContractExtractorGuavaTest`. Tento test je použit v projektu a zde je zobrazen formou zjednodušeného Java kódu. Testovací soubor je pak zobrazen níže a jeho obsah je téměř shodný s reálným souborem. Jedná se o soubor `testFiles/Guava/TestGuava.java`. Ostatní testy byly realizovány obdobným způsobem.

Testovací soubor `TestGuava.java`

Zde je zobrazen obsah testovacího souboru. Jak je vidět jedná se o jednoduchý kód, který ověřuje, zda vstupní parametr `x` není `null` pomocí knihovny Guava Preconditions.

```
public class TestGuava {
  public void guavaTest(String x){
    checkNotNull(x);
  }
}
```

Zjednodušený kód testu

Zde je zjednodušený kód jednotkového testu, který ověřuje správnost extrakce kontraktu typu Guava ze souboru uvedeného výše.

```
// Získání reprezentace kontraktu formou JavaFile
JavaFile javaFile = getTestJavaFile("TestGuava.java");
// Test očekává právě jeden kontrakt
assertEquals(1, javaFile.getContractsSize());

// Uložení jediného kontraktu do proměnné
Contract c = javaFile.getFirstContract();

// Ověření, zda daný kontrakt je typu Guava
assertEquals(ContractType.GUAVA, c.getContractType());
// Ověření zda se jedná o vstupní podmínku
assertEquals(ConditionType.PRE, c.getConditionType());
// Ověření, zda se shoduje celý výraz
assertEquals("checkNotNull(x);", c.getCompleteExpr());
// Ověření, zda se shoduje použitá funkce
assertEquals("checkNotNull", c.getFunction());
// Ověření, zda se shoduje logický výraz
assertEquals("x", c.getExpression());
// Ověření, zda nebyl nalezen žádný argument
assertEquals(0, contract.getArguments().size());
```

Jak je vidět, i přesto, že se jedná o jednoduchý kontrakt ve stručném souboru, test obsahuje mnoho asercí. Z tohoto důvodu jsou testy realizované v projektu parametrizované pomocí parametrů a jsou využity pomocné metody. Zde byl test pro lepší čitelnost realizován tímto způsobem bez pomocných metod.

7.2 Funkční testování

Vzhledem k tomu, že uživatelská aplikace je plně závislá na vytvořené knihovně, je možné knihovnu testovat zároveň s uživatelskou aplikací. Ta byla testována pomocí manuálního testování tím, že byly ověřovány různé testovací případy.

7.3 Testovací data

Vzhledem k povaze nástroje bylo třeba disponovat testovacími daty pro kontrolu funkčnosti. Pro tento účelem jsem využil skutečná data dodaná vedoucím práce, ale také jsem si některá sám vytvořil.

7.3.1 Skutečná data

Dodaná data obsahovala zdrojové soubory několika projektů různých verzí. Během testování jsem primárně využil samotného projektu Guava, který obsahuje velké množství kontraktů realizovaných pomocí JSR305 a Guava Preconditions. Knihovna se vyskytuje v mnoha verzích a obsahuje řadu souborů. Z tohoto důvodu byla vhodná nejen pro testování extrakce kontraktů, ale také pro ověření správnosti porovnávání. V mnohem menší míře byly také použity projekty Annotations, JSR305 a Reflections. Důvodem byl nízký počet kontraktů zkoumaného typu.

Skutečná data byla použita při manuálním testování nástroje. Díky své obsáhlosti mi nepřišla data vhodná pro automatizované testy. Mnoho jednotkových testů však bylo inspirováno použitím reálných reprezentací kontraktů nalezených v těchto datech.

7.3.2 Syntetická data

V reálných datech je obvyklé, že se využije jen část funkčnosti, kterou daná knihovna nabízí. Aby byl nástroj řádně otestován, uměle jsem vytvořil data, která pokryla různé možnosti použití kontraktů. To jak vytvořená data je dobře vidět v ukázce jednotkového testu výše. Zde se jedná o jednoduchý test vhodný pro demonstraci v dokumentu, jiná testovací data jsou více komplexní. Všechna tato data se nacházejí v projektu ve zdrojích ve složce `testFiles`. Zde jsou rozděleny do složek podle oblasti testování (`comparator` pro porovnávání kontraktů, `extractor` pro extrakci kontraktů a `parser` pro analýzu jazyka Java). Tato data jsou umístěna na CD ve složce `src/ContractParser/src/main/resources/testFiles`.

7.4 Výsledky testů

Ve finální verzi nástroje byly všechny testy úspěšně vykonány. Testy byly aktivně využívány v závěrečné fázi vývoje, kdy byly používány k zjištění, zda je stále zachována funkčnost nástroje po provedených změnách.

8 Zhodnocení výsledků

Hlavní cíle této práce byl úspěšně zapracovány. Ze vstupních souborů je možné extrahovat konstrukce DbC, které je možné dále exportovat do vnější reprezentace či porovnávat. Funkcionalita tohoto nástroje je pak zakomponována do uživatelské aplikace. Řešení má samozřejmě své silné i slabé stránky a zde bude proveden jejich rozbor.

8.1 Úspěšnost detekce kontraktů

Na základě provedených testů a analýzy se mi podařilo úspěšně detekovat všechny kontrakty typu JSR305 či Guava Preconditions. Tyto kontrakty byly úspěšně uloženy do navržené datové struktury připraveny pro další zpracování. Kontrakty je možné extrahovat ze zdrojových i přeložených souborů jazyka Java. Kód daného souboru však musí být validní, jinak není možné jej zpracovat analyzátorem.

8.2 Úspěšnost porovnání kontraktů

Dle provedených testů a zkoumání těchto aktivit si myslím, že jsem při porovnávání dosáhl dobrých výsledků. Je možné porovnávat dvě složky, kde jsou spárovány jednotlivé soubory, třídy, metody a kontrakty. Pro párování kontraktů byla použita heuristika, která s určitými limitacemi umožňuje spojení i kontraktů, které se změnily, či bylo změněno jejich pořadí. Výsledkem porovnávání je seznam změn v rámci kontraktů ale i API obou složek, resp. souborů. Tyto změny je možné dále analyzovat a využít je při zkoumání problematiky použití Design by Contract.

Protože oblast párování, nejen kontraktů ale i metod, tříd a souborů, je rozsáhlá a stejně tak to platí pro samotné porovnávání kontraktů, je tyto činnosti možné různě vylepšit. Díky tomu by bylo možné získat více užitečných a přesnějších informací (viz níže).

8.3 Prostor pro zlepšení

8.3.1 Rozpoznání dalších kontraktů

Nástroj momentálně umožňuje rozpoznání dvou typů kontraktů (JSR305, Guava Preconditions). To pro použití v praxi samozřejmě nemusí být dostačující a dalším logickým krokem ve vývoji by mělo být přidání rozpoznání dalších specifikací. Nástroj je na to připraven a z hlediska integrace by se tedy nemělo jednat o problém. Obtížnost samotného rozpoznání je pak závislá na povaze dané reprezentace. To samé platí pro párování metod a tříd, kdy změna v signatuře vede k tomu, že tyto objekty nejsou spárovány. Pro párování kontraktů byla vytvořena heuristika, která umožňuje do jisté míry spárovat i kontrakty, které se změnily, či se změnilo jejich pořadí. I přesto, že tato heuristika dosahuje dobrých výsledků, jistě je zde také prostor pro zlepšení.

Samotné rozhodnutí které další reprezentace přidat nemusí být triviální. Některé reprezentace může být velmi složité extrahovat a je tak třeba zvážit, zda jsou natolik používané, aby se tato práce vyplatila. K tomuto rozhodování může pomoci článek Contracts in Wild [4] i tento text, avšak jistě bude vhodné tyto poznatky rozšířit o další analýzu.

8.3.2 Porovnávání

Pro porovnávání dvou složek se soubory obsahujícími kontrakty je možné udělat pokročilejší párování souborů. Pokud je soubor pouze přejmenován či přesunut, je to kvalifikováno, jako že byl soubor odstraněn, respektive přidán, díky čemuž dané dva soubory nejsou porovnány. Pro částečné odstranění tohoto problému by bylo možné vytvořit heuristiku, která by porovnávala obsahy souborů, případně by bylo možné dát uživateli volbu v případě neprůkaznosti.

Prostorem pro zlepšení je také jednoznačně samotné porovnávání výrazů kontraktů. Momentálně se pouze zkoumá, zda jsou výrazy kontraktů shodné či nikoliv. Toto porovnání však zřejmě může být více komplexní. V případě, že se jedná o logickou podmínku, je možné zkoumat, zda je daný logický výraz shodný i přesto, že je zaměněno pořadí (např. výrazy `x > 0` a `0 < x` jsou z logického hlediska shodné, avšak nástroj je vyhodnotí jako rozdílné). Je také možné zaznamenat, zda je některý výraz zpřesňuje či naopak zobecňuje ten druhý. Tyto informace by pak mohly být užitečné při porovnávání

různých verzí téhož projektu a určovaní, zda se nezpřísnily kontrakty pro dané API.

Pro realizaci tohoto rozšířeného porovnávání by bylo třeba nejprve zjistit, zda daný výraz, obsahuje logický výraz či jiný útvar (může se např. jednat o definici parametru v případě JSR305). Pokud by se jednalo o logický výraz, bylo by jej třeba převést do formy, kde by bylo možné provést další analýzu (např. do stromové struktury). Následně tuto formu porovnat s výrazem svého protějška a vyvést z toho důsledky.

8.3.3 Efektivita

Aplikace poskytuje prostor pro zlepšení v rámci efektivity algoritmů. Extrakce porovnání kontraktů se v současné době provádí sekvenčně jak v rámci jednotlivých souborů, tak v rámci typů kontraktů. To má za důsledek delší dobu běhu algoritmu. Tuto činnost by mělo být možné zpracovávat paralelně, což by mohlo celý proces značně urychlit. Samozřejmě by však bylo nutné také přidat ošetření z hlediska konkurence procesů. Alternativně by bylo možné spojit dohromady extrakce jednotlivých typů kontraktů a snížit tak cyklomatičnost celého procesu. Toto zlepšení by však bylo na úkor přehlednosti, jelikož by se museli jednotlivé algoritmy prolínat. Větší efektivita, ale také větší ztráta přehlednosti by byla v případě většího množství rozpoznávaných reprezentací.

Stejně tak to je i v případě porovnávání kontraktů. Zde by však bylo nutné dobře vyřešit párování jednotlivých objektů, jelikož by zde mohlo docházet ke kolizi procesů.

8.3.4 Uživatelská aplikace

Vzhledem k tomu, že bylo cílem vytvořit pouze jednoduchou uživatelskou aplikaci, která bude primárně sloužit pro další výzkum této problematiky, je v této oblasti mnoho prostoru pro zlepšení. V současné podobně je aplikace vhodná pro zpracování menšího či středního množství souborů. Pro rozšířené použití, by bylo nasnadě zlepšit zobrazení souborů, které by bylo možné např. členit dle projektů, ukládat či jinak zachovávat současné projekty a obecně přidání funkcionalit tohoto typu.

S těmito rozšířeními pro zpracování souborů by také souviselo perzistence zpracovaných dat. Momentálně jsou zpracované soubory uchovávány

v paměti, což při malém množství souborů zrychluje práci, nicméně při větším množství souborů se jedná o paměťově náročnou činnost. Zpracované soubory by se tak mohli ukládat např. na pevný disk či do databáze. Díky tomu by také bylo možné se vyhnout opětovné extrakci kontraktů při dalším spuštění aplikace, což by mohlo zrychlit práci v případě, že by se stejnými soubory pracovalo více sezení.

Pro zlepšení celkové uživatelské zkušenosti by také bylo vhodné doplnit řadu dodatečných akcí, které jsou typické pro programy na správu dat, jak je např. řazení souborů, další možnosti filtrace, ale např. také umožnit použití klávesových zkratek.

V případě potřeby by také bylo možné zdokonalit konzolovou část aplikace, která by mohla poskytovat dodatečnou funkcionalitu či nastavení.

Myslím, že díky své silně technické a akademické povaze, není potřebné žádné vylepší vzhledu, které by mohlo být prospěšné např. u komerční aplikace.

9 Závěr

Cílem práce bylo se seznámit s konceptem kontraktu softwarových modulů, zejména pak přístupem Design by Contract (DbC) a prostudovat způsoby popisu DbC kontraktu v Java technologiích. Primárním cílem bylo navrhnout a implementovat nástroj pro extrakci, případně porovnání, konstrukcí DbC ze zdrojových, respektive přeložených, souborů jazyka Java. Zároveň měla být provedena analýza a návrh modelu, který bude schopen takto získaná data reprezentovat. Závěrem bylo za úkol ověřit správnost získaných výsledků a jejich souhrn.

V rámci teoretického úvodu bylo třeba čtenáře uvést do problematiky zajištění kvality software, informovat jej o dělení a použití kontraktů a také poskytnout náhled na jazyka Java v oblasti analýzy a dekompilace. Následně byl popsán způsob, jak byl navržen model pro reprezentaci dat a implementace samotného nástroje.

Nástroj byl úspěšně implementován a umožňuje extrakci konstrukcí kontraktů Guava Preconditions a JSR305. Součástí jsou také prostředky pro porovnání těchto konstrukcí a to nejen z hlediska samotných kontraktů, ale také v rámci souborů a celých adresářů. Nástroj lze ovládat pomocí vytvořené uživatelské aplikace, která umožňuje použití příkazů pro rychlé zpracování většího množství souborů či využití grafického rozhraní pro přehlednou práci. Funkčnost obou částí nástroje byla řádně otestována a výsledky zhodnoceny.

Všechny stanovené cíle byly úspěšně splněny. Práci je tedy možné použít pro analýzu konstrukcí DbC a dále jí snadno rozšiřovat dle potřeb výzkumu.

Přehled zkratek

API Application Programming Interface

CRE Conditional Runtime Exception

DbC Design By Contract

GUI Graphical User Interface

IDE Integrated Development Environment

JSON JavaScript Object Notation

JSR Java Specification Request

JVM Java Virtual Machine

MISRA Motor Industry Software Reliability Association

UML Unified Modeling Language

VCS Version Control System

XML Extensible Markup Language

Literatura

[1] Dr. Ulbert Zsolt:*Software Development Process and Software Quality Assurance*. University of Pannonia 2014

[2] *Defensive Programming* [online]. [cit. 2018-05-22]. <drdobbs.com/defensive-programming/184401915>

[3] *MISRA* [online]. [cit. 2018-06-03]. <misra.org.uk>

[4] 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

[5] Bertrand Meyer, "Applying 'design by contract'," Computer, vol. 25, no. 10, pp. 40–51, Oct 1992

[6] Bertrand Meyer, Object-oriented software construction, Prentice-Hall international series in computer science, 1988

[7] Antoine Beugnard, Jean-Marc Jézéquel, Noël Plouzeau, and Damien Watkins. Making Components Contract Aware. Computer, 32(7):38–45, 1999

[8] *Bertrand Meyer's technology + blog* [online]. [cit. 2018-04-11]. <bertrandmeyer.com/bio/>

[9] *EiffelStudio* [online]. [cit. 2018-04-13]. <dev.eiffel.com>

[10] *Guava Preconditions* [online]. [cit. 2018-04-21]. <github.com/google/guava>

[11] *tutorialspoint - Guava Preconditions* [online]. [cit. 2018-06-15]. <https://www.tutorialspoint.com/guava/guava_preconditions_class.htm>

[12] *JSR305* [online]. [cit. 2018-04-21]. <jcp.org/en/jsr/detail?id=305>

[13] *Cofoja* [online]. [cit. 2018-04-21]. <github.com/nhatminhle/cofoja

[14] *valid4j* [online]. [cit. 2018-04-21]. <valid4j.org

[15] *jContractor* [online]. [cit. 2018-04-22]. <jcontractor.sourceforge.net

[16] *Code Contracts* [online]. [cit. 2018-04-22]. <microsoft.com/en-us/research/project/code-contracts/>

[17] *Dokumentace Code Contracts* [online]. [cit. 2018-04-22]. <docs.microsoft.com/en-us/dotnet/framework/debug-trace-profile/code-contracts>

[18] *PhpDeal* [online]. [cit. 2018-04-22]. <github.com/php-deal/framework>

[19] *Boost.Contract* [online]. [cit. 2018-04-22]. <boost.org/doc/libs/master/libs/contract>

[20] *Eiffel* [online]. [cit. 2018-04-22]. <eiffel.org>

[21] *The Java Language Environment* [online]. [cit. 2018-05-01]. <oracle.com/technetwork/java/langenv-140151.html>

[22] *TIOBE Trend of programming languages* [online]. [cit. 2018-05-01]. <tiobe.com/tiobe-index>

[23] *Compiler Design Tutorial* [online]. [cit. 2018-05-01]. <tutorialspoint.com/compiler_design>

[24] *Internals of Java Class Loading* [online]. [cit. 2018-05-01]. <onjava.com/pub/a/onjava/2005/01/26/classloading.html>

[25] Torben Mogensen:*Basics Of Compiler Design*. University of Copenhagen 2000

[26] Prof. Ing. Karel Ježek CSc.:*Materiály pro předmět KIV/FJP na ZČU, 2018*

[27] *JFlex* [online]. [cit. 2018-05-01]. <jflex.de>

[28] *ANTLR* [online]. [cit. 2018-05-01]. <antlr.org>

[29] *JavaParser* [online]. [cit. 2018-05-01]. <javaparser.org>

[30] *Roaster* [online]. [cit. 2018-05-01]. <github.com/forge/roaster>

[31] *The Java Language Specification* [online]. [cit. 2018-06-20]. <https://docs.oracle.com/javase/specs/jls/se8/html/index.html>

[32] *java2novice - Retention policy in annotations* [online]. [cit. 2018-06-20]. <java2novice.com/java-annotations/retention-policy>

[33] *Top 8 Java Decompilers* [online]. [cit. 2018-05-02]. <crunchytricks.com/2016/07/best-offline-java-decompilers.html>

[34] *Decompilers Online* [online]. [cit. 2018-05-02]. <javadecompilers.com>

[35] *A Quick Look at Java Decompilers* [online]. [cit. 2018-05-02]. <blog.macuyiko.com/post/2015/a-quick-look-at-java-decompilers.html>

[36] *JD Project* [online]. [cit. 2018-05-02]. <jd.benow.ca>

[37] *Procyon* [online]. [cit. 2018-05-02]. <bitbucket.org/mstrobel/procyon>

[38] *CFR* [online]. [cit. 2018-05-02]. <benf.org/other/cfr>

[39] *W3Schools - JSON Sytnax* [online]. [cit. 2018-16-06]. <https://www.w3schools.com/js/js_json_syntax.asp>

[40] *Apache Maven* [online]. [cit. 2018-05-05]. <maven.apache.org>

[41] *Apache Log4j* [online]. [cit. 2018-04-11]. <logging.apache.org/log4j/2.x/download.html>

[42] *Gson* [online]. [cit. 2018-04-11]. <github.com/google/gson>

[43] *jUnit* [online]. [cit. 2018-04-11]. <junit.org/junit5/>

[44] *ControlsFX* [online]. [cit. 2018-04-12]. <fxexperience.com/controlsfx/>

[45] *FontAwesomeFX* [online]. [cit. 2018-04-12]. <bitbucket.org/Jerady/fontawesomefx>

Seznam příloh

A Uživatelská příručka

V této uživatelské příručce je vysvětleno jak nástroj nainstalovat, spustit a používat.

A.1 Instalace

Pro instalaci je třeba mít k dispozici Apache Maven [40], aby bylo možné aplikaci sestavit. Tento software je zdarma ke stažení. Po jeho instalaci jej můžeme využít pomocí příkazu `mvn` (pro snazší použití je vhodné přidat Maven do systémové proměnné).

Nejprve je třeba kompilovat knihovnu ContractParser. Ta se nachází na CD v adresáři `src/ContractParser`. Projekt sestavíme standardním příkazem `mvn install` (je vhodné použití `mvn clean install`, protože se zároveň odstraní předchozí verze). Následně je možné zkompilovat ContractManager, který se na CD nachází ve složce `src/ContractManager`. Zde je třeba použít tento příkaz: `mvn clean compile assembly:single`. Ten aplikaci sestaví a vytvoří z ní jeden spustitelný soubor `.jar`.

Případně je možné použít již zkompilovaný soubor `ContractManager.jar`, který se na disku nachází ve složce `bin`.

A.2 Spuštění

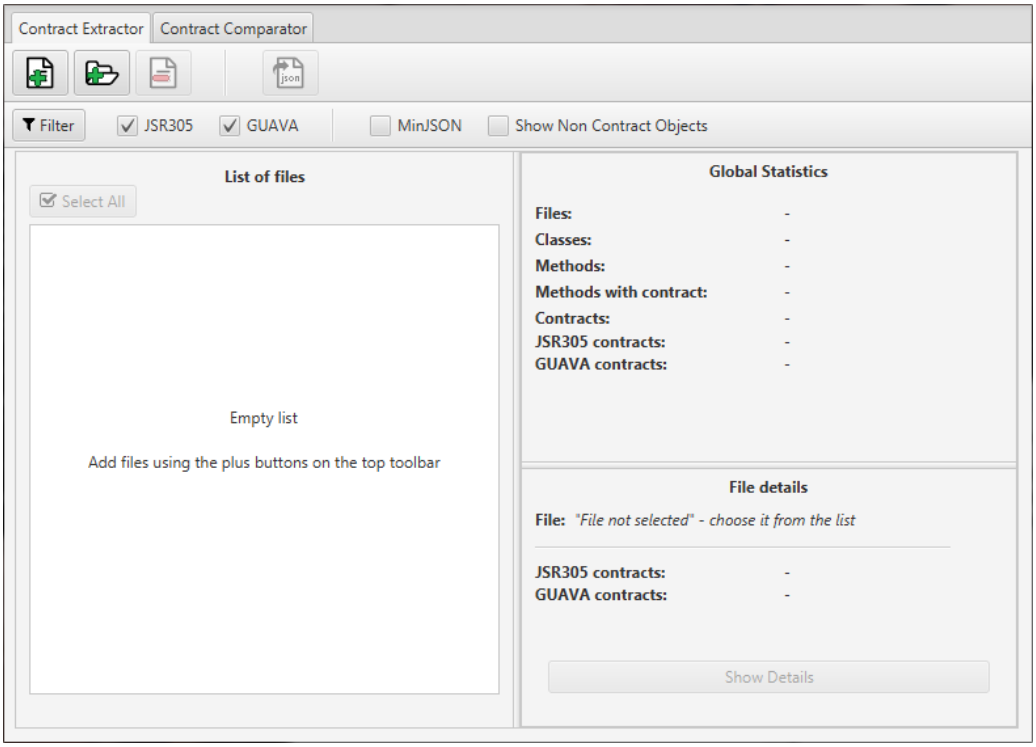
Aplikaci je možné používat ve dvou režimech. Prvním z nich je využití grafického uživatelského rozhraní. Aby se spustila tato varianta stačí spustit vytvořený soubor `.jar` nebo použít příkaz `java -jar ContractManager.jar`.

Druhou možností je spustit aplikaci s parametry, čímž je možné snadno zpracovat větší množství souborů. Více informací je popsáno níže.

A.3 Obsluha

A.3.1 Grafická část

Poté co je aplikace spuštěna, zobrazí se ve výchozím stavu, který je vidět na obrázku A.1. Ve vrchní části si můžeme povšimnout záložek, kde je označena **Contract Extractor**. Tato část aplikace slouží k extrakci kontraktů ze souborů. Druhá část, která slouží k porovnávání, bude popsána níže.

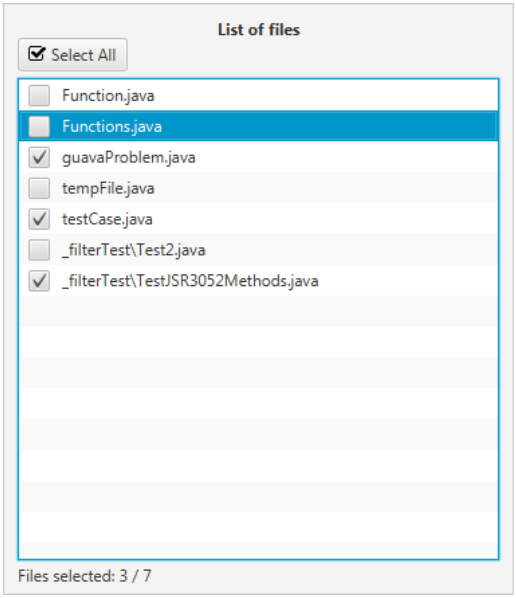


Obrázek A.1: Výchozí stav uživatelské aplikace

Extrakce kontraktů - základní zobrazení

Tlačítka pro přidání souborů Pod záložkami je panel nástrojů. Pomocí něj je možné soubory přidávat, odebírat či exportovat. První tlačítko (zelené plus na ikoně souboru) slouží k přidání individuálních souborů. Po jeho stisknutí se zobrazí okno pro výběr souborů. Je možné vybrat jeden či více souborů, ale pouze ty, které mají koncovku `.java` nebo `.class`. Pomocí druhého tlačítka (zelené plus na ikoně složky) je možné vybrat adresář a přidat tak všechny zdrojové či přeložené Java soubory z daného adresáře.

Přidání souborů Po přidání souborů jedním z těchto způsobů se vybrané položky zobrazí v seznamu v levé části okna, jak je vidět na obrázku A.2. Přidání souborů může nějakou dobu trvat, protože jsou v této fázi všechny soubory zároveň analyzovány a jsou z nich extrahovány kontrakty. Doba této operace závisí na počtu i komplexnosti souborů a také na výkonu zařízení. Pokud jsou přidány i soubory, které kontrakty neobsahují, v seznamu se nezobrazí. To se dá změnit nastavením filtrů (viz níže).



Obrázek A.2: Extrakce kontraktů - seznam aktuálních souborů

Odebírání a označování souborů Třetí tlačítko na panelu (červené mínus na ikoně souboru) slouží k odebírání souborů. Z obrázku A.2 je patrné, že každý souboru v seznamu má po své levé straně zaškrtávací pole. Po jeho vybrání je soubor označen a následně je možné jej pomocí tlačítka pro odebrání smazat se seznamu (fyzický soubor zůstane nedotčen). Ve spodní části oblasti se seznamem je zobrazen počet vybraných souborů z celkového počtu souborů v seznamu. Pomocí tlačítka **Select All** je také možné vybrat všechny soubory najednou (pokud již jsou všechny vybrány, tlačítko změní popisek na **Deselect All** a naopak zruší celý výběr).

Export souborů Posledním tlačítkem je export souborů. Po jeho stisknutí se zobrazí okno, kde se zvolí adresář, kam se mají exportovat všechny označené soubory. Výsledkem je soubor typu JSON pro každý z vybraných

souborů. Zároveň je vytvořen jeden soubor `_globalStatistics.json`, kde jsou uloženy globální statistiky o daných souborech. Podobu jednotlivých souborů ve formátu JSON je možné také vidět v detailech jednotlivých souborů (viz níže).

Globální statistiky Nahoře, v pravé části okna, je sekce **Global Statistics**, kde jsou zobrazeny globální statistiky pro všechny soubory v seznamu (jedná se o stejné informace, které jsou exportovány do speciálního souboru). Je zde informace o celkovém počtu souborů (**Files**), o počtu tříd (**Classes**) a počtu metod (**Methods**). Také je zde počet metod s kontrakty (**Methods with contracts**) a také počet samotných kontraktů (**Contracts**). Poslední informací je počet kontraktů pro jednotlivé typy reprezentací (např. **JSR305 contracts**). Tyto globální statistiky je možné vidět na obrázku A.3.

Obrázek A.3: Extrakce kontraktů - Globální statistiky

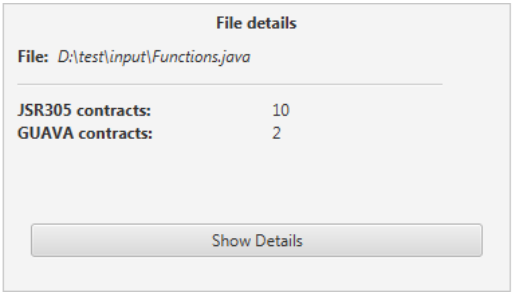
Sekce s detaily souboru Pod globálními statistikami se nachází panel s detaily o aktuálně vybraném souboru (viz obrázek A.4). Zde se nejedná o výběr pomocí zaškrtávací pole, ale označení souboru kliknutím na jeho název (nebo jinam do oblasti daného řádku). Označený soubor je vidět na obrázku A.2, kde je modře zvýrazněn (soubor `Functions.java`). Tyto detaily zobrazují absolutní cestu k danému souboru a také počet jednotlivých reprezentací kontraktů, které se v daném souboru nacházejí. Také je zde tlačítko **Show Details**, pomocí kterého je možné zobrazit okno s podrobnostmi o daném souboru (viz níže).

Filtry Poslední sekcí extrakční části jsou filtry, které jsou zobrazeny pod panelem nástrojů a jsou realizovány zaškrtávacími políčky (viz obrázek A.1). Pomocí těchto filtrů je možné změnit zobrazení dat v aplikaci a také ovlivnit

export. Nejprve jsou zde jednotlivé reprezentace kontraktů, které je možné libovolně zobrazovat či skrývat (ve výchozím stavu jsou všechny viditelné). Pak je zde možnost **Show Non Contract Objects**. Ta určuje, zda se mají zobrazovat i soubory, třídy či metody, které neobsahují žádný kontrakt.

Posledním filtrem je **MinJSON**. Pokud je tento filtr použit, veškerý export je ukládán v minimalizovaném formátu JSON. To může být užitečné např. při další práci s těmito soubory, aby se zmenšila jejich velikost. Ve výchozím stavu je tento přepínač vypnutý a všechny exporty jsou prováděny v tzv. *pretty print*, což znamená, že je JSON dobře čitelný pro člověka, protože je členěn dle hierarchie.

Aby nastavené filtry byly aktivovány, je třeba stisknout tlačítko **Filter**. Kromě filtru **MinJSON**, který ovlivní pouze export, ovlivňují filtry vše v aplikaci. Mění tedy seznam souborů, statistiky i detail daného souboru.

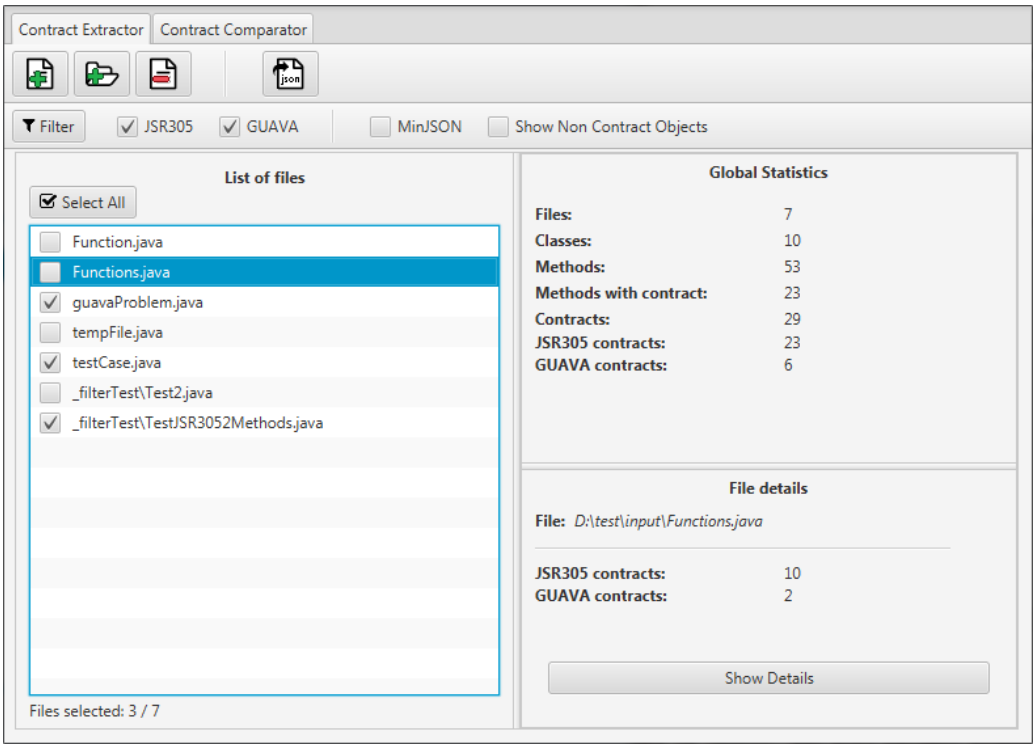


Obrázek A.4: Extrakce kontraktů - sekce detailu souboru

Aplikace po extrakci kontraktů Na obrázku A.5 je možné vidět, jak aplikace vypadá jako celek ve chvíli, kdy byly přidány soubory.

Extrakce kontraktů - detail souboru

Po stisku tlačítka **Show Details** se zobrazí nové okno s podrobnostmi o vybraném souboru (viz A.6). Je zde tedy název souboru a statistické údaje jako je počet tříd, metod a kontraktů. Cennou informací je zde strom zpracovaného souboru. Zde jsou vidět jednotlivé třídy a metody daného souboru v hierarchické podobě a uvnitř nich se nachází samotné kontrakty. Zde je možné zjistit podrobnosti o tom, o jaké kontrakty se jedná, jaký mají přesný tvar atd. Tento strom je zjednodušenou verzí JSON verze tohoto souboru. Ten je možné zobrazit přepnutím záložky z **Tree View** do **Raw Unfiltered JSON**. Jedná se o stejná data, která jsou uložena v případě využití exportu.



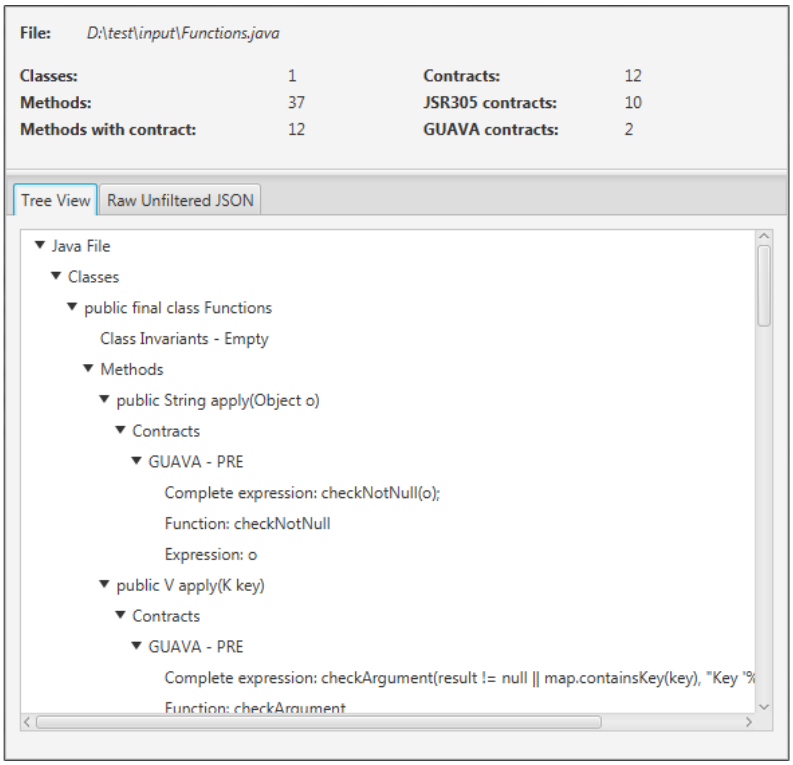
Obrázek A.5: Aplikace po extrakci kontraktů

Kromě zobrazení těchto informací zde není možné provádět žádnou aktivní činnost, okno je možné pouze zavřít standardním způsobem. Okno se také zavře v případě, že byla zavřena hlavní aplikace, nicméně je možné mít otevřené libovolné množství těchto oken s podrobnostmi, což je možné využít např. při jejich porovnávání.

Porovnávání kontraktů - základní zobrazení

Jak již bylo zmíněno výše, aby bylo možné porovnávat kontrakty je třeba přepnout záložku na vrcholu okna na **Contract Comparator**. Tím se změní rozložení aplikace, jak je vidět na obrázku A.7. Jediné co je mezi oběma částmi sdíleno, je nastavení společných filtrů. Nicméně obě části aplikace jsou si velmi podobné, a proto zde budou rozepsány pouze ty informace, které se liší oproti extrakční části.

Panel nástrojů Aplikace umožňuje pouze porovnání dvou složek. Tyto složky vybereme pomocí panelu nástrojů, který je oproti extrakční části změněn. První dvě tlačítka slouží tedy k výběru složek, po jejich stisknutí se zobrazí okno průzkumníka souborů, kde je možné vybrat pouze adresář. Tyto vybrané složky pak můžeme porovnat pomocí tlačítka **Compare**,

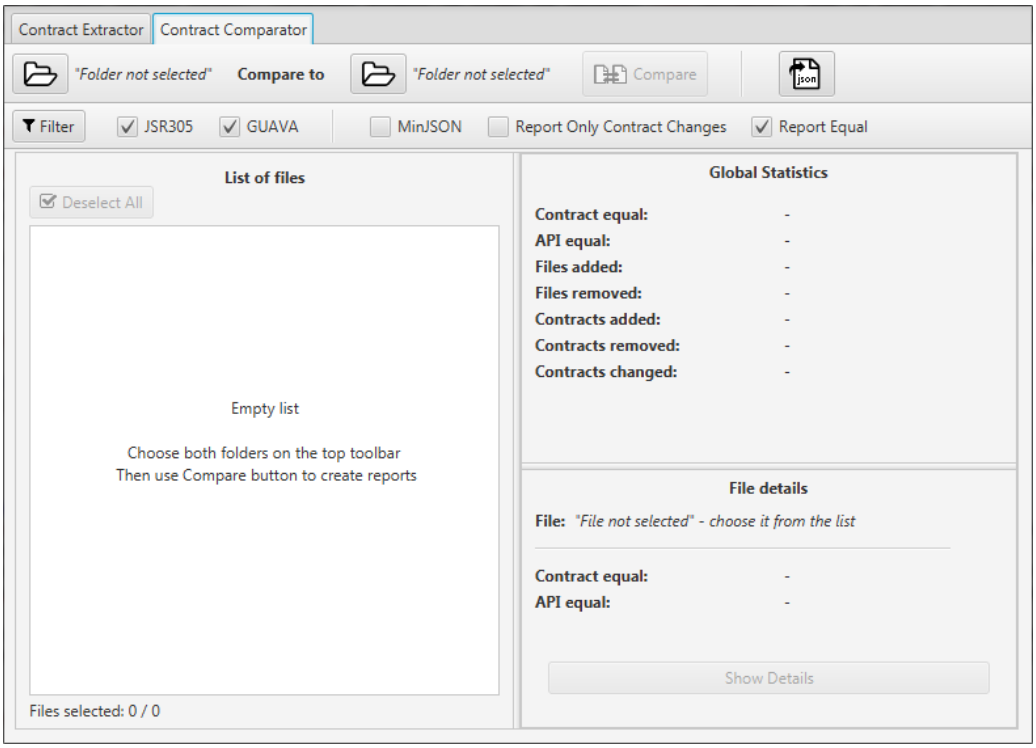


Obrázek A.6: Extrakce kontraktů - Okno s detaily souboru

které se po výběru složek zpřístupní. Tímto se podobně jako v předchozí části přidají soubory do seznamu v levé části. Tato akce opět zabere nějaký čas, během kterého je zobrazeno načítací okno. V případě, že bychom chtěli porovnat jiné složky, stačí zvolit novou složku pomocí jednoho z tlačítek a aplikace smaže všechny ostatní hodnoty. Na rozdíl od části pro extrakci zde není možné soubory mazat, ale vybrané soubory je možné extrahovat opět pomocí tlačítka na konci panelu.

Seznam souborů Seznam souborů vypadá stejně jako v extrakční části. Rozdílem však je, že zde jednotlivé položky představují zprávy o porovnání daných souborů. Každá tato zpráva pak obsahuje hierarchii tříd a metod, kde je pro každý kontrakt informace, zda byl přidán odebrán či změněn. Také jsou zde informace o přidaných a odebraných metodách a shrnutí zda je daný soubor se svým protějškem shodný v rámci API a kontraktů.

Globální statistiky Zbytek aplikace funguje stejným způsobem jako část pro extrahování, ale zobrazená data se liší. Globální statistiky zobrazují hodnoty z porovnání obou složek. Je zde informace o tom, zda jsou dané



Obrázek A.7: Výchozí stav porovnávací části aplikace

adresáře shodné na úrovni kontraktů (**Contract equal**) či na úrovni API (**API equal**). Je zde také informace o tom, kolik souborů bylo oproti předchozí složce přidáno respektive odebráno (**Files added / removed**). Tyto informace jsou zde také pro kontrakty. Přibývá však i informace o počtu změněných kontraktech (**Contracts added / removed / changed**).

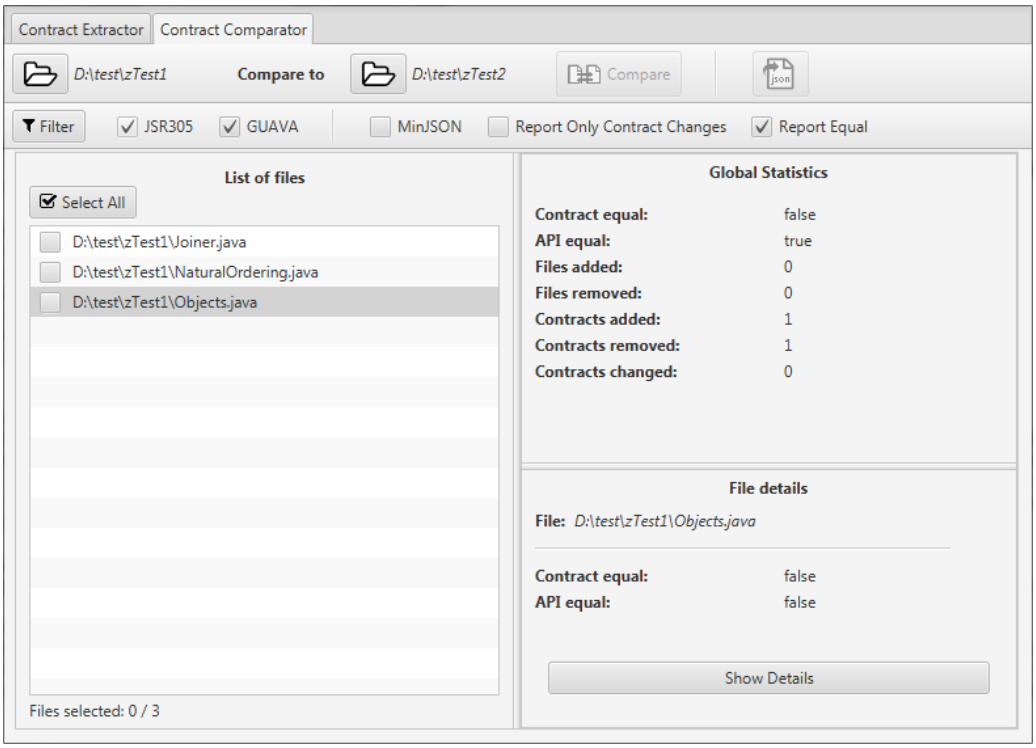
Detail souboru Sekce zobrazující detaily o souboru obsahují mimo jména souboru také informaci o tom, zda je tento soubor shodný v rámci API a kontraktů vůči svému protějšku. Jedná se tedy o informace, které jsou v globálních statistikách, zde se však týkají pouze vybraného souboru. Také je zde tlačítko **Show Details**, které zobrazí okno s detaily o daném porovnání.

Filtry I v této části jsou filtry umožňující skrýt různé reprezentace kontraktů a také je tu filtr pro minimalistický formát exportu. Navíc zde však přibyl filtr **Report Only Contract Changes**. Pokud je označen, nejsou zobrazeny žádné informace o souborech, třídách a metodách, které nijak neovlivňují kontrakty. To může velmi zjednodušit data, pokud nás zajímají

pouze informace kontraktech.

Posledním filtrem je **Report Equal**. Ve své výchozí pozici je zapnutý a určuje to, zda se mají zobrazovat i zprávy o tom, že jsou dané soubory, třídy, metody a kontrakty shodné. Vypnutím tohoto filtru můžeme snadno pouze změny a shodné objekty ignorovat.

Aplikace po porovnání složek Na obrázku A.8 je zachycen stav aplikace po porovnání dvou složek. Jak je vidět, tlačítko **Compare** není momentálně přístupné. Znovu se zpřístupní ve chvíli, kdy se vybere nová složka, důvodem je, že opětovné porovnání stejných složek by bylo zbytečné.



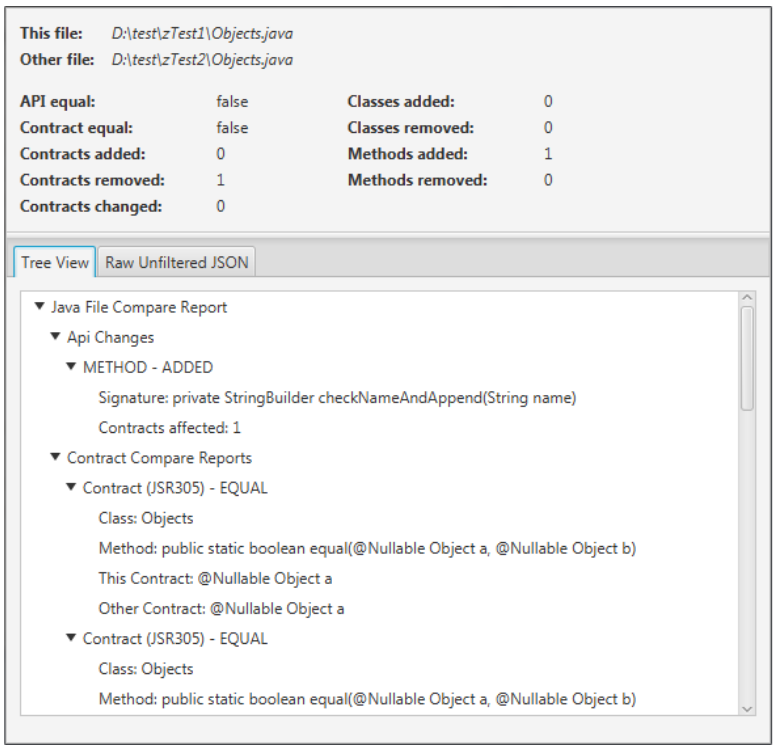
Obrázek A.8: Stav aplikace po porovnání vybraných složek

Porovnávání kontraktů - detail souboru

Po stisku tlačítka **Show Details** se opět otevře nové okno s podrobnostmi o dané položce (viz A.9). Stejně jako v případě extrakce, jsou zde také v horní části statistické informace a dole je poté stromové zobrazení. Co se statistických údajů týče, jsou zde informace o tom, zda je daný soubor shodný

v rámci kontraktů a API se svým protějškem. Také jsou zde informace o tom, kolik tříd, metod a kontraktů se přidalo nebo ubralo. V případě kontraktů pak také kolik se jich změnilo.

Stromové zobrazení je opět možné přepnout na JSON formát. Provede se to přepnutím záložky z **Tree View** na **Raw Unfiltered JSON**. Jedná se o stejný JSON, který je exportován pokud není vybraná možnost **MinJSON**.



Obrázek A.9: Porovnávání kontraktů - Okno s detaily souboru

A.3.2 Konzolová část

Pokud nepotřebujeme grafické rozhraní je možné použít konzolovou část aplikace. Pokud aplikaci spustíme se správnými parametry jednorázově provede extrakci kontraktů z dané složky či porovnání dvou složek. Výsledek je pak exportován ve formátu JSON do zvoleného adresáře. Dále je popsán způsob jakým je třeba zadat parametry.

Jako první parametr musí být vždy uveden `-e` pro extrakci kontraktů nebo `-c` pro porovnávání složek. Speciálním případem je pak parametr `-h`, který zobrazí nápovědu (je možné použít i `--help`).

Extrakce kontraktů

Příkaz pro extrahování kontraktů má tento tvar:

```
-e <input_folder> <output_folder> [-r] [-m]
```

Jak již bylo zmíněno, parametr `-e` značí, že se jedná o extrakci kontraktů. `<input_folder>` je povinný parametr, kde je třeba uvést složku, ze které budou kontrakty extrahovány. Dalším povinným parametrem je `<output_folder>`, kde je třeba uvést výstupní složku, kam se uloží exportovaná data.

Volitelný parametr `-r` určuje, zda mají být odebrány soubory, třídy či metody, které neobsahují žádné kontrakty. Export spuštěný s tímto parametrem bude tedy stručnější a bude obsahovat pouze objekty, které obsahují kontrakty.

Druhým a posledním volitelným parametrem je `-m`. Pomocí něj můžeme zajistit, že výstupní JSON bude ve své minimalistické podobě.

Zde je příklad spuštění aplikace tak, aby extrahovala všechny kontrakty ze složky `data/project` do `my/output` v minimalistickém formátu:

```
java -jar ContractMnager.jar -e data/project my/output -m
```

Porovnávání kontraktů

Příkaz pro porovnávání kontraktů má tento tvar:

```
-c <input_folder1> <input_folder2> <output_folder> [-q] [-o]
[-m]
```

Zde je třeba začít pomocí `-c`, což značí porovnávání. Následují tři povinné parametry, kde prvním (`<input_folder1>`) a druhým (`<input_folder2>`) jsou vstupní složky, jejichž obsah má být porovnán z hlediska API a zejména z hlediska rozdílnosti kontraktů. Třetím povinným parametrem je opět výstupní složka (`<output_folder>`), kam mají být exportovány výsledky.

Porovnání kontraktů má tři volitelné parametry, které korespondují s grafickou částí aplikace. Prvním z nich je `-q`. Ten určuje, zda mají být exportovány také objekty, které byly vyhodnoceny jako shodné. Ve výchozím stavu

jsou tedy tato data ignorována.

Dalším parametrem je `-o`. Pokud je tento přepínač vybrán, budou exportovány pouze změny, které se přímo týkají kontraktů. Tímto je možné zanedbat změny API, které se přímo kontraktů netýkají.

Posledním volitelným parametrem je opět `-m`, který i jako v předchozím případě umožňuje minimalistickou verzi formátu JSON.

B Obsah CD

Přiložené CD je členěno následujícím způsobem: Je zde složka `bin`, ve které se nachází spustitelný soubor `ContractManager.jar`, který slouží ke spuštění uživatelské aplikace. Dále je zde složka `document`, ve které se nachází tento dokument ve formátu pdf (`vmares_dp.pdf`) spolu se zdrojovými soubory, které se nachází ve složce `src`. Práce byla vytvořena pomocí sázecího software LaTeX.

Adresář `poster` obsahuje poster k diplomové práci ve formátu `.pub` a `pdf`. Poslední složkou je pak `src`, která obsahuje adresáře `ContractParser` a `ContractManager`. Tyto složky obsahují zdrojové kódy pro knihovnu a uživatelskou aplikace vytvořeného nástroje. Poslední složkou na CD je `javaDoc`, která obsahuje vygenerovanou dokumentaci kódu formou JavaDoc.

Na CD se také nachází soubor `readme.txt`, který také obsahuje popis adresářové struktury. Grafické znázornění adresářové struktury je možné vidět níže.

Adresářová struktura

```
|--- bin
|    |--- ContractManager.jar
|
|--- document
|    |--- src
|    |--- vmares_dp.pdf
|
|--- javadoc
|
|--- poster
|
|--- src
|    |--- ContractManager
|    |--- ContractParser
|
|--- readme.txt
```