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 8. května 2018

Václav Mareš

Poděkování

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

Abstract

The text of the abstract (in English). It contains the English translation of the thesis title and a short description of the thesis.

Abstrakt

Text abstraktu (česky). Obsahuje krátkou anotaci (cca 10 řádek) v češtině. Budete ji potřebovat i při vyplňování údajů o bakalářské práci ve STAGu. Český i anglický abstrakt by měly být na stejné stránce a měly by si obsahem co možná nejvíce odpovídat (samozřejmě není možný doslovný překlad!).

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 těchto integrací a zvýšit jejich přehlednost. Z tohoto důvodu má smysl se těmito konstrukcemi zabývat a analyzovat jejich použití.

Cílem 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. Hlavním cílem práce 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 takto získaná data reprezentovat. Závěrem práce by mělo být ověření správnosti získaných výsledků a jejich souhrn.

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

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 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 vývojářského týmu dosáhnout co nejlepších výsledků v daných oblastech.

2.2 Zajištění kvality

P o 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. Ty jsou podrobněji rozepsány v následující kapitole.

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.

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: [2][3][4][5].

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 - interaktivní
- 4. úroveň - mimo-funkční

3.1.1 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.

3.1.2 Sémantické kontrakty

3.1.3 Interaktivní kontrakty

3.1.4 Mimo-funkční kontrakty

...

V této kapitole bude čtenář kromě konceptu kontraktů také seznámen s vlivem použití na kvalitu kódu. Podrobně bude rozebrána...

...

...Kontrakty mohou působit dojmem, že slouží jako náhrada testů, nicméně se nejedná o zaměnitelné funkce a naopak by se měly navzájem doplňovat.

3.2 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.

3.3 Design by contract

Pojem design by contract zavedl francouzský profesor Bertrand Meyer. První větší zmínka je uveden v publikaci *Design by Contract, Technical Report* v roce 1987. Profesor 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 [6][**?**][7].

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. Pomocí design by contract může vývojář specifikovat očekávané hodnoty. Za předpokladu, že vývojář odvede kvalitní práci a uživatel tento kontrakt dodrží, mělo by to zamezit množství chyb spojených s integrací. Design by contract zajišťuje správnost pomocí *assertions*, kterých jsou celkem tři typy. Prvním z nich jsou tzv. pre-conditions, které zajišťují správné hodnoty vstupních parametrů, tedy hodnoty před tím, než je provedena určitá operace. Jejich opakem jsou pak post-conditions, které naopak zajišťují správnost výstupu, tedy hodnot po provedení dané operace. Třetím typem jsou tzv class invariants, nebo-li neměnné proměnné, které musejí platit po celou dobu. Je možné říci, že class invariant je pre-condition a post-condition pro všechny veřejné metody v dané třídě.

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.

3.4 Rozdělení kontraktů

Kontrakty můžeme rozdělit do několika kategorií dle způsobu jejich použití:

- Podmíněné výjimky za běhu (Conditional Runtime Exceptions - CRE)
- API
- Assert
- Anotace
- 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. Všechny tyto výjimky jsou součástí standardní Javy, 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. Assert je typicky spojováno s tvorbou testů, ale je možné jej 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 poskytuje 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.

3.5 Způsoby popisu kontraktů v Java technologiích

V Java exstuje 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 jsou stále používány. Zde je výčet některých z těchto nástrojů:

3.5.1 Guava Preconditions

Knihovna Guava od Google poskytuje řadu nových funkcí jako například různé kolekce, primitiva, práce se souběžnými programy atd. Z hlediska kon-

traktů je pro nás však zajímává 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 pre-conditions, jak již název napovídá. Při porušení takovéto podmínky je pak vyvolána podmínky při běhu programu. Volitelnou částí 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 pre-conditions. 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, nastane standardní výjimka `java.lang.NullPointerException`. Typ výjimky, který knihovna vrací je závislý na dané metodě (např. pro metodu `checkArgument(boolean)` je vrácena výjimka `IllegalArgumentException`).

3.5.2 JSR305

JSR305 umožňuje specifikaci kontraktů pomocí anotací. Na rozdíl od Guava Preconditions umožňuje také použití post-conditions i class invariants. Obecně platí, že anotace, které jsou uvedeny před argumenty metod např: `@Nonnull Object x` specifikují pre-conditions pro parametr dané metody. Pokud je anotace uvedena pro celou metodu, jedná se o post-condition a kontrakt se tak váže na výstupní hodnotu metody. V případě, že je anotace vázaná na třídu, jedná se o class invariant. Některé anotace je možné použít jako libovolný druh, nicméně mnoho z nich je specializována pro jeden či dva typy podmínek.

Pokud je kontrakt porušen, je opět vyhozena výjimka, v tomto případě `IllegalArgumentException`. Na rozdíl od Guava, zde není nativní možnost pro zadání vlastní chybové zprávy v případě porušení kontraktu, ale výchozí chyba je poměrně samovysvětlující. JSR305 nicméně neposkytuje pouze striktní podmínky, které při porušení skončí chybou, ale umožňuje také anotace, které slouží jako informace pro vývojáře. Např. anotace

`@CheckForNull` upozorňuje, že daný objekt může nabýt hodnoty `null`, ale nevynucuje žádné chování a je pouze na vývojáři, jak s touto informací naloží. Příklad zobrazující všechny tři typy kontraktů je vidět zde:

```
@ParametersAreNullableByDefault
public class JSR3053ExampleClass {

   @CheckReturnValue
   public Obejct 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*[1], stále je hojně využíváno v řadě projektech a má tak smysl jej zkoumat. [8]

3.5.3 Cofoja

Contracts for Java, 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 pre-conditions, `@Ensures` pro post-conditions a `@Invariant` pro class invariants [9]. Praktické využití v kódu pak může vypadat takto:

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

[1] *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.

3.5.4 valid4j

Valid4j je jednoduchý nástroj, který poskytuje metody pro práci s kontrakty za pomoci aserce. Podobně jako jiné nástroje využívá klíčových slov `require` pro pre-conditions a `ensure` pro post-conditions. Tento nástroj neposkytuje podporu pro class invariants za pomocí specializovaných metod ale tohoto chování je možné dosáhnout použitím zmíněných metod [10]. Zde je část kódu implementující kontrakty nástrojem valid4j.

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

3.6 Kontrakty v jiných technologiích

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.

3.6.1 Code Contracts v .NET

Prvním příkladem může být projekt Code Contracts, 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 kontraktů (pre-conditions, post-conditions a class invariants). Základními funkcemi jsou `Contract.Requires()` pro vynucení správného vstupu, tedy preconditions, `Contract.Ensures()` pro zajištění správného výstupu, tedy postcondition a `Contract.Invariant()` pro reprezentaci class invariant. Mimo těchto základních funkcí poskytuje také různé specifické operace, se kterými je možné vytvářet komplexnější kontrakty. Umožňuje také např. specifikaci vyhozené výjimky [11][12]. V následujícím příkladu je vidět použití základ-

ních konstrukcí:

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

4 Reprezentace gramatiky a jazyky

4.1 Jazyk Java a jeho gramatika

4.1.1 Možnosti parsování

4.2 Bytcode

4.2.1 Možnosti parsování

4.2.2 Limitace dekompilace

5 Datový model

5.1 Rozbor vybraných DbC konstrukcí

Po analýze dostupných materiálů jsem se rozhodl zvolit pro implementaci konstrukce Guava Preconditions a JSR305. Důvodem byla především jejich rozdílná reprezentace, kdy Guava Preconditions je realizováno pomocí volání metod uvnitř těl metod a umožňuje vytvářet pre-conditions. Na druhé straně JSR305 je tvořené anotacemi v záhlaví tříd, metod a také jako součást parametrů metod. Umožňuje tvoření všech tří typů kontraktů (pre-conditions, post-conditions, class invariants). Kromě této diverzity se také jedná o jedny z četných konstrukcí používaných v projektech [2].

5.1.1 Guava Preconditions

Jak již bylo zmíněno, Guava umožňuje tvorbu kontraktů pomocí pre-conditions voláním metod.

5.1.2 JSR305

5.2 Společné znaky

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

5.3 Vytvořený model

5.3.1 Popis

Výsledný model jsem vytvořil na základě analýzy konstrukcí kontraktů s ohledem na následný export do dat, které bude možné dále zpracovávat. Aby byl zachován kontext kontraktů, usoudil jsem, že bude třeba zachovávat také informace o třídách a metodách v daném souboru. Rozhodl jsem se tedy vytvořit strukturu podobnou stromu, jejíž kořenem je samotný zdrojový soubor. Tento soubor obsahuje různé podrobnosti o tomto souboru jako je jeho jméno a cesta, typ a statistiky o jeho obsahu. Také obsahuje seznam

všech tříd obsažených v tomto souboru. Každá jednotlivá třída pak obsahuje jméno, svou hlavičku, seznam metod a také seznam všech kontraktů týkajících se této třídy, tedy class invariants. Metoda pak nese informaci o své signatuře a také seznam všech kontraktů této metody.

Samotný kontrakt se pak skládá z těchto informací:

`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-condtion, post-condition a class invariant.

`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 funkce o jaký se jedná. V případě Guava se jedná o název metody, v případě JSR305 o název anotace. Obecně se jedná o hlavní označení určující daný typ kontraktu.

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

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

`String file, className, methodName` Kontrakt také obsahuje jméno rodičovského souboru, jméno třídy a metody. I přesto, že je tyto atributy možné získat od rodičovských objektů, jsou zde z důvodů přehlednosti exportu a zejména pak kvůli porovnávání, kde urychlují celý proces.

5.3.2 Diagram

Obrázek

5.3.3 Reprezentace modelu

Po dohodě s vedoucím práce jsem se rozhodl pro externí reprezentaci modelu použít formát JSON. Vzhledem k tomu, že JSON je široce používaný zápis dat a umožňuje relativně snadné ukládaní objektů typu Java a také umožňuje další zpracování. JSON je tak vhodný pro zpracovaní strojem, ale je dobře čitelný i pro lidské oko (v případě, že byl zformátován).

6 Nástroj pro analýzu kontraktů

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

Smyslem aplikace je umožnit detekci kontraktů ve zdrojových, respektive přeložených, souborech jazyka Java. Nalezené kontrakty je pak možné analyzovat a zkoumat způsob a četnost použití jednotlivých typů v různých projektech. Do budoucna by tento nástroj mohl pomoci při analýze změny požadavků na rozhraní

6.1 Knihovna

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

6.1.1 Použité technologie

Knihovna byla implementována v jazyce Java verze 1.8 ve vývojovém prostředí IDEA IntelliJ Ultimate 2017.3.3. Pro zajištění snadného získání závislostí a následného zjednodušení použití knihovny byla pro vytvoření projektu použita technologie Apache Maven.

Externí knihovny

Při vývoji knihovny pro analýzu kontraktů byly použity následující knihovny třetích stran:

Apache Log4j Tato knihovna umožňuje pokročilé možnosti pro logování. Byla použita zejména pro zaznamenávání chyb a různých informačních zá-

znamů [13].

Procyon Knihovna Procyon byla použita pro dekompilaci přeložených Java souborů `*.class` [14].

JavaParser JavaParser byl použit tokenizaci zdrojových souborů [15].

Google Gson Tato knihovna poskytuje prostředky pro uložení objektů jazyka Java do reprezentace pomocí formátu JSON [16].

jUnit 5 Poskytuje možnosti testování pomocí jednotkových testů [17].

6.1.2 Dekompilace Bytcode

Pro dekompilaci Java `*.class` souborů byla použita knihovna Procyon. Ta umožňuje použití metody `decompile()`, která přečte vstupní soubor s přeloženým kódem a do jiného souboru uloží jeho dekompilovanou verzi. Tento dočasný soubor s dekompilovaným kódem je poté předán pro zpracování třídě `JavaFileParser`, která jej zpracuje stejně jako běžný zdrojový soubor. V knihovně dekompilaci obstarává metoda `decompileClassFile()`, která se nachází v třídě `io.IOServices`.

6.1.3 Parsování Java souborů

Pro zpracování zdrojových souborů jazyku Java byla použita knihovna JavaParser. Ta poskytuje metodu `parse()`, která vytvoří komplexní strukturu daného zdrojového souboru. V prvním kroku se tato struktura projde a vyhledá všechny třídy (*class*) a také rozhraní *interface* a výčtové typy *enum*. Pro účely modelu jsou si tyto tři prvky rovny. Každý nalezený prvek je následně uložen do modelu. V případě třídy a rozhraní je se struktura prochází dále a do modelu jsou uloženy všechny konstruktory, které se z hlediska modelu považují za metody (viz níže). Následně jsou uloženy všechny anotace dané „třídy“.

Po této přípravě je využita třída `MethodVisitor`, která dědí od třídy `VoidVisitorAdapter` a umožňuje procházet všechny metody v daném souboru. V metodě pak máme k dispozici objekt typu `MethodDeclaration`, který obsahuje všechny potřebné údaje a také rodičovský `ExtendedJavaFile`. Pro každou metodu je nalezena její rodičovská třída. Hledá se nejvyšší rodič a tudíž vnořené metody nemají jako rodiče vyšší metodu ale nejvyšší dostupnou třídu. Pro danou metodu jsou následně uloženy všechny anotace a

i její parametry. Následně je uloženo celé tělo metody jako seznam objektů typu `Node`, které umožňují další zpracování. Z těchto získaných dat je vytvořena instance objektu `ExtendedJavaMethod`, která je následně uložena do své rodičovské `ExtendedJavaClass`.

6.1.4 Extrakce kontraktů

Obecně

Poté, co je ze Java souboru vytvořen objekt typu `ExtendedJavaFile`, je možné začít extrahovat kontrakty. Během získávání kontraktů se tato struktura prochází a postupně se k jednotlivým třídám a metodám přidávají kontrakty. Poté, co jsou všechny extrakce dokončeny, je za 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 Precondtions

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

JSR305

Na rozdíl od Guava Preconditions mohou být kontrakty typu JSR305 obsaženy v anotacích tříd a metod a také v jejich parametrech. Zde je tedy nutné procházet tyto bloky a naopak těla metod je možné zanedbat. Postupně se procházejí jednotlivé anotace tříd i metod. Jakmile je daná anotace výrazem JSR305, je uložena jako kontrakt. Tvar anotace představuje `function` a stejně jako v případě Guava, první parametr je uložen jako `expression` a ostatní jsou uloženy do seznamu `arguments`. Tyto anotace však obvykle parametr nemají. Takto nalezené kontrakty v anotacích třídy jsou označeny

za neměnné proměnné (class invariants) a v anotacích metod se pak jedná o post-conditions, vztahují se k výstupu metody. Zbývají parametry metod, u kterých se opět zkoumají anotace stejným způsobem. Tyto anotace však vždy mívají alespoň jeden atribut a tím je tvar samotného paramteru.

6.1.5 Porovnávání kontraktů

6.1.6 Popis API

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 zprovoznění nového typu kontraktu jsou potřeba tyto kroky:

Přidání položky do `ContractType`

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

Vytvoření nového analyzátoru

Následně je třeba vytvořit funkční část daného parseru. Je tedy nutné vytvořit třídu, která bude implementovat rozhraní `ContractParser`. Toto rozhraní požaduje pouze jednu metodu a tou je `ExtendedJavaFile retrieveContracts(ExtendedJ extendedJavaFile)`. Aby byly zachovány 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 tento balíček by se měl nacházet v balíčku `cz.zcu.kiv.contractparser.parser`. Tvar samotné metody již závisí na principech daného kontraktu. Obecně platí, že by se měly kontrakty detekovat a vytvořit na základě dat ze vstupního objektu typu `ExtendedJavaFile` a ve stejném objektu je také vrátit. Pro lepší představu doporučuji prozkoumat již implementované analyzátory pro JSR305 a Guava Preconditions.

Doplnění továrny `ParserFactory`

Dalším krokem je doplnění továrny `ParserFactory`. Zde je pouze třeba přidat nový `case` do konstrukce `switch`. Tento blok by měl vracet instanci

nového parseru v případě že vstoupí tento typ v objektu `ContractType`.

6.1.8 Doplnění metody `retrieveContracts()`

Posledním krokem je přidat podmínku do metody `retrieveContracts()` ve třídě `parser.ContractExtractor`. Jedná se o jednoduchý kód, který zajišťuje, aby se provedla extrakce daného typu kontraktu, pokud byl daný kontrakt ve vstupní mapě, či vstupní mapa s typy kontraktů byla `null`. Dokončením této části by již měl být daný typ kontraktu plně funkční.

6.1.9 Testování

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

6.2 Uživatelská aplikace

6.2.1 Použité technologie

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

Externí knihovny

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

ControlsFX Tato knihovna rozšiřuje JavaFX a umožňuje použití dalších funkcí a objektů zejména pak `CheckListView`, což je použito pro zobrazení seznamu souborů [18].

FontAwesomeFX Knihovna FontAwesomeFX slouží opět k rozšíření JavaFX. Tuto knihovnu jsem použil pro rozšíření možností zobrazení ikon [19].

6.2.2 Rozdělení aplikace

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

Grafická část

Konzolová část

6.2.3 Možnosti a limitace aplikace

6.3 Optimalizace

obecně, snažil jsem se zajistit co nejlepší...

6.3.1 Analýza a refaktoring kódu

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

6.3.2 Zjdenodušení modelu

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

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

7 Testování

7.1 Prostředky použité k testování

- Unit testy - integrační testy - malá aplikace nemusí se tolik řešit - funkční testy - testovat na úrovni GUI (ošetření vstupů) - PMD

7.2 Testovací data

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

7.3 Výsledky testů

8 Zhodnocení výsledků

8.1 Úspěšnost detekce kontraktů

8.2 Úspěšnost porovnání kontraktů

8.3 Limitace

8.4 Prostor pro zlepšení

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

9 Závěr

TODO

Přehled zkratek a použitých výrazů

DbC Design By Contract

Literatura

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

[2] 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

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

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

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

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

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

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

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

[10] *valid4j* [online]. [cit. 2018-04-21]. <http://www.valid4j.org

[11] *Code Contracts* [online]. [cit. 2018-04-21]. <https://www.microsoft.com/en-us/research/project/code-contracts/>

[12] *Dokumentace Code Contracts* [online]. [cit. 2018-04-21]. <https://docs.microsoft.com/en-us/dotnet/framework/debug-trace-profile/code-contracts>

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

[14] *Procyon* [online]. [cit. 2018-04-11]. <bitbucket.org/mstrobel/procyon>

[15] *JavaParser* [online]. [cit. 2018-04-11]. <github.com/javaparser/javaparser>

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

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

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

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

Seznam příloh

A Uživatelská příručka

B Obsah CD