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

Diplomová práce

Porovnání nástrojů ověřujících kompozici modulárních Java aplikací

Plzeň 2016

Michal Bratner

Prohlášení

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

V Plzni dne 10. května 2016

Michal Bratner

Poděkování

Chtěl bych poděkovat Ing. Kamilu Ježkovi, Ph.D. za ochotu, užitečné rady, trpělivost a čas věnovaný při konzultacích, které mi pomohly k lepšímu vypracování této práce.

Abstract

In real software projects is common to use third-party libraries. It brings challenges with incompatibilities between two libraries. There are tools for verifying the composition of modular Java applications to detect these problems. One of these tools is JaCC, which is developed at the University of West Bohemia. The problem by developing the tool is the lack of test data for this tool. The objective of this master's thesis is to create a set of test data to simulate possible compatible and incompatible changes in Java applications. Created data are used for testing existing tools for verifying the composition of modular Java applications. The tools are then compared based on tests results and other non functional characteristics.

Abstrakt

V reálných softwarových projektech se běžně pro různou funkcionalitu využívají knihovny třetích stran. To s sebou však také přináší problémy spojené s nekompatibilitami mezi jednotlivými knihovnami. Pro odhalování těchto problémů existují nástroje ověřující kompozici modulárních Java aplikací. Jedním takovým je nástroj JaCC vyvíjený na Západočeské univerzitě. Problémem při vývoji takového nástroje je však nedostatek testovacích dat. Tato práce řeší daný problém vytvořením sady testovacích dat, které pokrývají co největší množství možných kompatibilních i nekompatibilních změn při vývoji aplikací v jazyce Java. Dále v práci jsou pomocí vytvořených dat otestovány existující nástroje odhalující nekompatibilitu Java aplikací a jsou porovnány s nástrojem JaCC na základě dosažených výsledků na testovacích datech a dalších mimofunkčních charakteristik.

Obsah

1 Úvod **1**

2 Kompatibilita **2**
2.1 Zdrojová kompatibilita 3
2.2 Binární kompatibilita 5
2.3 Způsob testování 6

3 Testované nástroje **8**
3.1 jdeps 8
3.2 Animal Sniffer 9
3.3 JAPICC 10
3.4 JaCC 11
3.5 Možnosti spuštění 12
3.6 Mimofunkční charakteristiky 13

4 Testované oblasti **15**
4.1 Změna datového typu 15
4.2 Přístupové modifikátory 17
4.3 Nepřístupové modifikátory 18
4.4 Výjimky 20
4.5 Dědičnost 21
4.6 Generičnost 23
4.7 Třídy 25
4.8 Rozhraní 26
4.9 Pojmenování testů 26

5 Seznam testů **28**
5.1 Přehled testů 28
5.2 Shrnutí testů 40

6 Zhodnocení nástrojů **41**
6.1 Výsledky testů 41
6.1.1 Shrnutí výsledků 51
6.2 Mimofunkční charakteristiky 54
6.2.1 Snadnost použití 54
6.2.2 Integrovatelnost 55
6.2.3 Struktura výstupu 56
6.2.4 Formát výstupu 58
6.2.5 Licenční politika 59
6.2.6 Aktuálnost 60
6.3 Výsledné zhodnocení 61

7 Závěr **63**

A Příloha **67**
A.1 Uživatelská příručka 67

1 Úvod

Málokterý rozšířenější a využívnější projekt ve světě informačních technologií je vytvořen bez použití knihoven třetí strany, které plní jisté dílčí úkoly v projektu. Jedná se o běžný postup, který však spolu přináší také problémy spojené s nekompatibilitami mezi jednotlivými knihovnami. Problémy, které mohou vznikat nekompatibilními změnami uvnitř knihoven, navíc nabírají na složitosti s narůstajícím rozsahem projektu a s tím narůstajícím počtem využitých knihoven. Hledání problémů se tím prodlužuje a výrazné komplikace v odhalení příčiny může následně nastat také v situaci, kdy implementujeme knihovnu A, která implementuje knihovnu B. V případě, že knihovna B vydá novou verzi, která je nekompatibilní s předchozí, může tento problém zasáhnout i naši aplikaci. Takový problém již může být velmi komplikované odhalovat.

K odhalení těchto problémů slouží nástroje odhalující nekompatibilní změny z pohledu kompozice Java aplikace. Jedním takovým nástrojem je také nástroj JaCC vyvíjený na katedře informatiky a výpočetní techniky Západočeské univerzity v Plzni. Největším problémem při vývoji je však nedostatek testovacích dat, které by pomáhaly při vývoji tohoto nástroje.

Cílem této práce je tedy vytvořit testovací data, která budou představovat kompatibilní i nekompatibilní změny z hlediska kompozice aplikace složené z JAR souborů. Zde vytvořená data budou řešit problém s nedostatkem testů pro vývoj univerzitního nástroje. V rámci této práce bude také následně vyvíjený nástroj otestován na připravených datech a spolu s ním budou otestovány také další exisující nástroje, které dokáží odhalovat nekompatibilní změny z pohledu kompozice aplikace a jsou porovnány s nástrojem katerdy.

2 Kompatibilita

Hlavním cílem práce je otestovat nástroje, které odhalují nekompatibility z hlediska kompozice Java aplikací. Než se k tomu ale dostaneme, definujme si nejdříve několik základních pojmů a způsob testování, kterým jsou nekompatibility v této práci hledány.

Jako první si definujme samotný pojem kompatibilita a to, jak se k němu přistupuje v jazyce Java. Jak je uvedeno v [1], v programech napsaných v jazyce Java rozlišujeme tři základní druhy kompatibility:

- Zdrojová kompatibilita
- Binární kompatibilita
- Behaviorální kompatibilita

Zdrojovou a binární kompatibilitu si popíšeme dále, behaviorální kompatibilita v této práci nebude řešena, protože chování programu nedokážeme určit pomocí statické analýzy programu.

Základním problémem kompatibility je obtížnost nalezení a úpravy každého software a systému, který je změnou ovlivněn. Pokud by bylo možné dohledat a současně upravit všechny klienty, kteří danou knihovnu používají, pak by byl vývoj snažší. Bohužel toto je většinou nereálné, takže při vývoji knihoven je nutno dbát na kompatibilitu nových verzí se staršími, což velmi omezuje vývoj.

Uvažujme program využívající určitou sadu knihoven. Z hlediska kompozice aplikace nás zajímá, zda změna ve využívané sadě knihoven nezpůsobí problém v našem programu. Problém může vzniknout například z důvodu, že každá knihovna má svůj odlišný životní cyklu a může dojít například ke kolizím v názvech využívaných prvků. Další možnou příčinou problémů je samozřejmě také vydání nové verze knihovny, ve které může dojít k nekompatibilní změně.

Co se týče závažnosti jednotlivých typů kompatibility, zdrojová nekompatibilita je nejméně závažná, protože většinou má přímočaré řešení. Behaviorální nekompatibilita může mít řadu dopadů, zatímco binární nekompatibilita je nejhorší, protože ta nedovoluje slinkování JAR souborů.

Existuje velká skupina možných změn ve třídách a rozhraních, které mohou mít vliv na kompatibilitu. Jedním z úkolů této práce je vytvořit data, která budou takovéto změny simulovat, takže na konkrétní podobu nekompatibilních změn se podíváme později v této práci.

2.1 Zdrojová kompatibilita

Zjednodušeně bychom mohli zdrojovou kompatibilitu vysvětlit tak, že pokud lze klientský program P přeložit se starou verzí knihovny L a následně i s novou verzí této knihovny, pak můžeme prohlásit, že změny provedené v knihovně jsou zdrojově kompatibilní vůči klientskému programu P. V obecném měřítku je však obtížné o zdrojové kompatibilitě uvažovat takto, protože je prakticky nemožné znát všechny potenciálně existující klientské programy a tím pádem ani dopady změn na jednotlivé programy. Z tohoto důvodu je tudíž v podstatě nemožné dosáhnout plné kompatibility se všemi existujícími programy.

Nicméně, jak uvádí [1], toto je pouze základní definice a nezahrnuje celý prostor zájmu. Úkolem překladače v jazyce Java je také mapování abstraktnějších jmen na konkrétnější, přesněji mapování jednoduchých a kvalifikovaných jmen ze zdrojového kódu na binární jména v class souborech. Překladač řeší nejen to, zda je či není možné mapování provést, ale také to, zda výsledné class soubory jsou či nejsou vyhovující. Výsledná binární jména slouží jako jednoznačné identifikátory pro referencovatelnost daných prvků. Na základě tohoto rozšířeného pohledu můžeme definovat více úrovní zdrojové kompatibility:

- Lze kód stále přeložit?
- Pokud ano, jsou všechna jména přiřazena ke správným binárním jménům v class souboru?
- Pokud jde kód kompilovat, ale ne všechna jména jsou správně přiřazena k binárním jménům, je výsledný class soubor behaviorálně ekvivalentní?

Zda je či není program validní může ovlivnit také změna v konstrukci samotného jazyka. Dříve nevalidní program se může stát validním, jako když

byla přidána generika. Naopak nevalidním se může program stát například přidáním klíčová slova (např. assert, enum).

Uveďme si nyní příklad, kdy máme knihovnu s jednoduchou abstraktní třídou Foo a abstraktní metodou foo:

```
public abstract class Foo {
   public abstract void foo();
}
```

Dále máme klienta, který z knihovny dědí právě tuto třídu Foo. Máme tedy klienta s kódem:

```
import lib.Foo;

public class Main extends Foo {

   @Override
   public void foo() {
      System.out.println("foo");
   }

   public static void main(String[] args) {
      Foo foo = new Main();
      foo.foo();
   }
}
```

Pokud nyní přijde nová verze knihovny, ve které bude do třídy Foo přidána nová abstraktní metoda, bude porušena zdrojová kompatibilita s naším klientem, protože bude vyžadována implementace nově přidané abstraktní metody. Binární kompatibilitu to však neovlivní, protože Java linker nekontroluje chybějící implementace metod, takže slinkování již existujících binárních souborů proběhne bez problémů. Zde tedy vidíme ukázku toho, že zdrojová a binární kompatibilita nejsou totožné.

2.2 Binární kompatibilita

Budeme-li parafrázovat [1], můžeme binární kompatibilitu popsat jako zachování schopnosti slinkovat knihovny s klientskou aplikací. Pokud máme tedy vygenerované binární soubory klienta, které bylo možné přeložit a slinkovat s binárními soubory staré verze knihovny, pak změny provedené v knihovně jsou binárně kompatibilní, pokud lze již vygenerované binární soubory klienta slinkovat také s binárními soubory nové verze knihovny.

Jak je popsáno v [2], binární soubory v jazyce Java jsou kompilovány tak, aby byl zajištěn spolehlivý přístup k dostupným členům a konstruktorům jiných tříd a rozhraní. Aby byla zajištěna binární kompatibilita, musí třída nebo rozhraní zacházet se svými členy a konstruktory tak, jak uvádí jejich kontrakt s uživatelem.

Programovací jazyk Java je navržen tak, aby se zabránilo dodatkům ke kontraktům, či náhodným kolizím jmen, což by vedlo k porušení binární kompatibility. Jako konkrétní příklad uvedeme přidání většího počtu přetížených metod, které nerozbije binární kompatibilitu s již existujícími binárními soubory.

Binární kompatibilita není totéž co zdrojová nebo behaviorální. Je možné vytvořit sadu vzájemně kompatibilních binárních souborů, které jsou však vygenerovány ze zdrojových kódů, které by společně nebylo možné přeložit. Jako příklad uveďme situaci, kdy tělo metody, vykonávající smysluplnou činnost, je nahrazeno vyhozením výjimky. Ačkoliv se tímto razantně mění chování programu a pokud klient nezachycuje vyhazovanou výjimku, tak tato změna není ani zdrojově kompatibilní. Přesto to není otázka binárni kompatibility, pokud by binární soubory klienta a této třídy šly stále slinkovat.

Uveďme si tedy také příklad situace, kdy dojde k problému s binární kompatibilitou. Mějme knihovnu s jednoduchou třídou Foo a metodou foo:

```
public class Foo {
   public int foo() {
      return 42;
   }
}
```

Následně mějme také klientskou aplikaci, která využije tuto třídu Foo:

```
import lib.Foo;

public class Main {

   public static void main(String[] args) {
      Foo foo = new Foo();
      int tmp = foo.foo();
      System.out.println(tmp);
   }
}
```

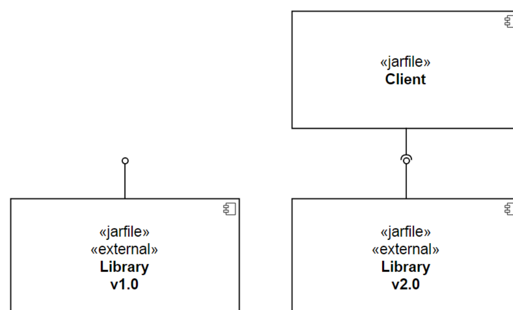
Pokud nyní přijde nová verze knihovny, která ve třídě Foo přidá k metodě foo modifikátor static, dojde k binární nekompatibilitě s dřívě přeloženým kódem klienta. Zdrojová kompatibilita však narušena nebude, protože z hlediska zdrojového kódu je jedno, zda je metoda statická, nebo není. Rozdíl, který způsobuje binární nekompatibilitu, je totiž ve využití jiných instrukcí bytekódu pro volání statické a nestatické metody. Vidíme zde tedy další příklad toho, že zdrojová kompatibilita není totéž co binární.

2.3 Způsob testování

Jelikož tématem práce je testováni z pohledu kompozice Java aplikací, musí tomu pochopitelně odpovídat také struktura připravovaných testů. Není potřeba vymýšlet zbytečně složité konstrukce, protože pro naše účely postačí jednoduchá kompozice dvou JAR souborů zobrazená na obrázku 2.1.

Jde o jednoduchou kompozici, kdy klientská aplikace využívá jednu knihovnu. Každý test má v klientovi i v knihovně vlastní balíček, aby byla zajištěna jednoduchost testů. V klientovi každý balíček obsahuje hlavní třídu, ve které je pouze využita testovaná funkcionalita z knihovny tak, aby byla prověřena kompatibilita změny.

Pro testy jsou následně připraveny dvě verze knihovny, které mají stejně dělenou strukturu jako klientská aplikace. První verze knihovny je plně kompatibilní s klientem, zatímco druhá verze knihovny, reprezentující vývoj dané knihovny, obsahuje změny, které již kompatibilní s klientem být nemusí.



Obrázek 2.1: Testovací kompozice JAR souborů klienta a knihovny

V zájmu této práce budou testovány problémy se zdrojovou a binární kompatibilitou. Zjišťování těchto nekompatibilit s druhou verzí knihovny je úkolem nástrojů, které jsou v této práci testovány.

Nástroje tedy musejí nabízet možnost využití, při které se na vstup umístí JAR soubory klienta a knihovny. Následnou analýzou těchto získaných souborů pak musejí umět odhalovat nekompatibilní změny vůči klientské aplikaci.

3 Testované nástroje

Testovací data, která byla vytvořena v rámci této práce, jsou použita pro otestování a porovnání nástrojů pro ověření kompozice Java programů. Jednotlivé nalezené nástroje pro tuto činnost jsou popsány v této kapitole. Mezi nimi je také nástroj JaCC vyvýjený na Západočeské univerzitě v Plzni.

3.1 jdeps

Nástroj jdeps slouží pro statickou analýzu programu. Dle [3] dokáže vyhledávat vazby mezi knihovnami a nalézt chybějící závislosti. To dokáže provádět na úrovni balíčků, nebo tříd.

Jedná se o nástroj spustitelný z příkazové řádky, který je distribuován společně s JDK, a sice od verze JDK 8. Na adrese [4] je k dispozici seznam přepínačů včetně jejich významu. Je zde také několik ukázek použití včetně výstupů nástroje.

Vstup nástroje může mít tyto podoby:

- cesta ke class souboru
- JAR archiv
- adresář
- plně kvalifikované jméno třídy pro analýzu všech class souborů

Výstup programu je umístěn na standardní výstup, tedy do konzole. Lze ale také nechat vygenerovat výstup do DOT souboru.

Vzhledem k tomu, že tento nástroj dokáže hledat závislost jen na úrovni tříd, dá se očekávat, že na plné sadě připravených testovacích dat bude nejslabším z testovaných nástrojů.

3.2 Animal Sniffer

Animal Sniffer, jak je uvedeno na stránce [5], poskytuje několik nástrojů, které pomáhají ověřit kompatibilitu programů.

Prvním je nástroj spouštěný z příkazové řádky, který zobrazí čísla verzí jednotlivých tříd, což pomůže při hledání problematického JAR souboru v případě, že se potýkáme s touto chybou: `UnsupportedClassVersionError`. Při spuštění nástroje z příkazové řádky je mu možno na vstup umístit libovolný počet

- Class souborů
- JAR archivů
- adresářů

Jsou-li na vstupu adresáře, jsou rekurzivně prohledány pro nalezení class souborů a JAR archivů. Jsou-li na vstupu JAR archivy, jsou prohledány class soubory uvnitř jednotlivých archivů.

Vzhledem k naší práci nás ale zajímají další funkcionality, které Animal Sniffer nabízí:

- Sada Ant tasků
- Pravidla pro využití v maven-enforcer-plugin
- Maven plugin

Sada Ant tasků umožňuje ověření našich tříd vůči signatuře API. Další task zase umožňuje signaturu API vygenerovat z JDK, nebo z předaných JAR archivů, class souborů, nebo ze souborů se signaturou jiných API. Možná je také kombinace zmíněných vstupů. Vůči vygenerované signatuře API je pak ověřena klientská aplikace.

Pravidla pro využití v maven-enforcer-plugin umožňují pouze ověření, že naše třídy jsou v souladu s dříve vygenerovanou signaturou API.

Maven plugin umožňuje obdobně jako Ant tasky nejen kontrolu našich tříd vůči signatuře API, ale umožňuje nám také tuto signaturu vygenerovat. Signaturu můžeme opět generovat z JDK, JAR archivů, class souborů, nebo ze souborů se signaturou jiných API. Možná je opět také kombinace zmíněných vstupů.

Pokud nám však ani jedna tato varianta nevyhovuje, můžeme využít zdrojový JAR archiv nástroje s dvěma definovanými vstupními body do jeho rozhraní:

- `SignatureChecker` pro testování tříd vůči specifické signatuře,
- `SignatureBuilder` pro vygenerování signatury ze skupiny tříd.

Jelikož vývoj testů není proveden v Maven projektu, je k otestování tohoto nástroje využita dostupná sada Ant tasků. Vzhledem k tomu, že tento nástroj je stále udržován a poslední verze před sepsáním této práce byla vydána v únoru 2016 (verze 1.15), mohli bychom od tohoto nástroje očekávat kvalitní výsledky.

3.3 JAPICC

Java API Compliance Checker je open-source nástroj vytvořený jako skript v jazyce Perl. Jedná se o nástroj spouštěný z příkazové řádky, který slouží pro kontrolu binární i zdrojové zpětné kompatibility API knihoven psaných v jazyce Java. Nicméně nástroj umožňuje také možnost kontrolovat provedené změny mezi dvěma verzemi knihoven vůči klientské aplikaci, takže kvůli malému počtu nalezených nástrojů jej v této práci využijeme také, ačkoli jeho srovnání s ostatními nástroji nebude úplně objektivní.

Pokud si tedy přiblížíme náš způsob použití, je potřeba na vstup vložit klientskou aplikaci, starou verzi knihovny a novou verzi knihovny. Nástroj následně vyhodnotí pouze změny uvnitř knihovny, které mohou ovlivnit danou klientskou aplikaci. Pokud je knihovna tvořena více než jedním JAR archivem, je možné vytvořit XML soubor podle návodu na adrese [6], do kterého můžeme uvést všechny potřebné JAR archivy.

Výstup nástroje je vygenerován do HTML souboru, kde jsou odděleny části pro zdrojovou a pro binární kompatibilitu. Nalezené problémy jsou roz-

děleny do několika skupin podle toho, o jaký problém se jedná. Navíc každá chyba má ještě přidělenu závažnost. Závažnost má tři stupně (Low, Medium, High). Problémy s nízkou závažností mohou být označeny pouze jako warning v případě, že není zapnut striktní mód.

Vzhledem k seznamu kontrolovaných problémů, který je také uveden na adrese [6], lze předpokládat, že tento nástroj bude mít špatné výsledky především v oblasti generičnosti.

3.4 JaCC

Nástroj JaCC je vyvíjen na katedře informatiky a výpočetní techniky Západočeské univerzity v Plzni.

Jak uvádí [7], jedná se v podstatě o sadu nástrojů vyvinutých pro ověřování kompozice modulárních Java aplikací. Nástroje obsahují společném jádro, které je nazýváno Java Compatibility Checker (JaCC) a sadu klientských nástrojů na tomto jádře. Jeho hlavní funkcí je reverzní inženýrství, díky čemuž může testovat existující binární soubory.

Nástroj umožňuje nejen testování zpětné kompatibility knihoven, ale také testování kompozice klient - knihovna. Na vstup jsou předávány JAR archivy, ze kterých si nástroj následně rozbalí class soubory a ty pomocí zmíněného reverzního inženýrství analyzuje. Nalezené problémy jsou následně vypsány do textového řetězce.

Integrace nástroje je možná několika různými způsoby. Jelikož je nástroj stále vyvíjen, je možné, že další možnosti budou časem přibývat, nicméně momentálně jsou k dispozici tyto možnosti:

- Maven plugin
- Eclipse plugin
- CoCAEx

Hlavním úkolem Maven pluginu je shromáždit veškeré knihovny, které jsou spojeny s vyvíjeným projektem. Shromážděné knihovny jsou následně zvalidovány nástrojem JaCC, který pak do konzole vypíše zprávu s výsledky

a pokud nalezne problém, způsobí chybu. Základní funkcí pluginu je vyhledávání chybějících nebo nekompatibilních prvků (třídy, metody, proměnné,...). Vzhledem k tomu, že v Maven projektech je střet verzí knihoven poměrně častý jev, plugin může navíc detekovat situace, kdy projekt může být opraven jednoduchým vyloučením jedné z knihoven.

Další možností je integrovat nástroj JaCC přímo do Ecplise IDE, což vývojáři umožní transparentně používat verifikaci přímo ze svého IDE. Plugin funguje podobně jako Maven plugin popsaný výše. Jediným rozdílem je, že výpis je zabudován přímo do uživatelského rozhraní v Eclipse.

Poslední zmíněnou možností je využití s CoCAEx, což je vizualizační nástroj vytvořený také katedrou informatiky a výpočetní techniky Západočeské univerzity v Plzni. Ten umožňuje zobrazovat výsledky graficky pomocí webové aplikace. Jednotlivé JAR archivy jsou znázorňovány jako vzájemně propojované uzly. Propojení jednotlivých archivů s sebou pak nese také případné informace o nekompatibilitách tohoto spojení.

3.5 Možnosti spuštění

Pro lepší přehlednost ještě na závěr teoretického přehledu uvedeme shrnutí toho, jak lze který nástroj spouštět. Pomůže nám to v porovnávání schopností jednotlivých nástrojů. Tabulka 3.1 zobrazuje základní přehled možností spuštění.

Tabulka 3.1: Možnosti spuštění nástrojů

Způsob použití	jdeps	Animal Sniffer	JAPICC	JaCC
CLI	✓		✓	
Maven		✓		✓
Ant		✓		
Eclipse				✓
Jiné		✓		✓

Nástroj Animal Sniffer sice má možnost spuštění z příkazové řádky, ale pouze pro zjištění čísel verzí jednotlivých tříd. Umožňuje však navíc ještě pravidlo pro použití v maven-enforcer-plugin. Nástroj JaCC umožňuje použití pomocí nástroje CoCAEx pro grafický výstup.

3.6 Mimofunkční charakteristiky

Ačkoliv naším hlavním měřítkem kvality jednotlivých výše popsaných testovaných nástrojů bude pochopitelně úspěšnost v hledání nekompatibilních změn knihovny, definujeme také několik mimofunkčních charakteristik. S jejich pomocí budeme moci zhodnotit kvalitu nástrojů z jiných úhlů pohledu a také v případě shodných výsledků při hledání chyb budeme mít více měřítek pro určení kvality nástrojů.

Zjištěné výsledky těchto charakteristik budou prezentovány později v této práci společně s úspěšností nástrojů při hledání nekompatibilit. K definování těchto konkrétních charakteristik jsem dospěl při testování nástrojů, kdy jsem se setkal s různou úrovní složitosti z hlediska samotného spuštění nástrojů. Po spuštění nástrojů a prohlédnutí si jejich výpisů, jsem dospěl k dalším charakteristikám, které se týkají struktury a přehlednosti výpisu a také možných formátů pro export dat. Dalšími faktory, které mohou negativně ovlivnit použitelnost nástrojů je licenční politika, se kterou je nástroj distibuován, a také jeho aktuálnost, která může napovědět o potenciálu jeho schopností.

Snadnost použití
Jednou z mimofunknčních charakteristik je samozřejmě snadnost použití. Lze nástroj využít jednoduchým spuštěním z příkazové řádky, případně přidáním modulu například pomocí Maven, či v Ant scriptu?

Integrovatelnost
Umožňuje nástroj integrovatelnost do vývojového cyklu? Pokud totiž používáme například Ant script, či máme Maven projekt, může pro nás být výhodné, když nástroj umožňí integraci pomocí odpovídajících postupů.

Přehlednost výstupu
Velmi důležitým faktorem je pochopitelně výstup programu. To je to, co nás zajímá a proč daný nástroj vůbec používáme. Můžeme tedy z jeho výstupu snadno vyčíst výsledky? Jak podrobný daný výstup je? Vidíme pouze přibližnou lokaci chyby, například pouze třídu, či dokonce jen balík, ve kterém k chybě došlo, nebo naopak vidíme i konkrétní označení prvku, který problém zavinil? A vidíme také jaký problém vznikl?

Formáty výstupu
Ne vždy se nám hodí stejný formát výstupních dat. Umožňuje nástroj volbu formátu dat, nebo máme pouze jednu konkrétní možnost? Je výpis možný pouze do konzole, nebo můžeme nechat vygenerovat výstupní soubor? A pokud ano, jaké formáty jsou nám nabízeny?

Licenční politika
Důležitou vlastností je také licenční politika. Pokud by totiž nástroj byl distribuován pod licencí omezující jeho využití, omezovalo by to také samozřejmě jeho širší použitelnost a byl by znevýhodněn oproti jiným dostupným nástrojům s volnější licencí.

Aktuálnost
Posledním faktorem, na který se podíváme, je aktuálnost nástroje. Jelikož se vyvíjí také samotný jazyk Java, je tento faktor poměrně důležitý z hlediska schopnosti nástroje odhalovat problémy s kompatibilitou. Nevyvíjený nástroj totiž pochopitelně bude mít minimální šanci odhalit problémy v nových konstrukcích jazyka, které se objevily až po poslední aktualizaci nástroje.

4 Testované oblasti

Aby bylo možné otestovat samotné nástroje, bylo nejdříve potřeba vytvořit testovací data, s jejichž pomocí by se dalo ověřit, jak moc kvalitní dané nástroje jsou. Pod pojmem kvalitní si v tomto případě můžeme představit množství odhalených problémů z připravených testů pokrývajících změny v různých oblastech jazyka Java. Pro co nejlepší otestování nástrojů je tedy zapotřebí testovacích dat, které by pokryly co nejvíce možných situací, ve kterých může dojít k problému s kompatibilitou.

Z důvodu co největšího pokrytí možných změn byly pro tvorbu testů zvoleny oblasti nejčastějších potenciálních změn při vývoji Java aplikací. Každá oblast má vytvořenu sadu testovacích dat, která odpovídá možným změnám popsaným dále v této kapitole. Jelikož bylo vytvořeno více než 250 testů, bylo také zavedeno takové pojmenování testů, které vede k jednoznačné identifikaci změny, která je daným testem simulována. Vytvořený princip pojmenování testů je popsán na konci této kapitoly. Veškeré testy jsou připraveny ve struktuře odpovídající popisu v kapitole 2.3.

4.1 Změna datového typu

První takovou skupinou je změna datového typu. Jak uvádí [11] a [12], změny typu lze provádět na základě následujících principů. V závorkách jsou uvedeny výrazy použité v názvech testů:

- Boxing (Box)
- Unboxing (Unbox)
- Narrowing (Narrow)
- Widening (Widen)
- Generalisation (Gen)
- Specialization (Spe)
- Mutation (Mut)

`Boxing` je změna, kdy primitivní datový typ je nahrazen typem odpovídající obalující třídy. Například tedy nahrazení primitivního typu `int` obalující třídou `Integer`. `Unboxing` je opačná změna, tedy v našem příkladu změna z `Integer` na `int`.

`Widening` je typ změny, při které je číselný typ nahrazen jiným číselným typem s větším rozsahem. Příkladem může být záměna typu `int` za typ `long`. `Narrowing` je následně opačná změna.

`Generalisation` je změna, kdy je datový typ nahrazen nadřazeným typem v hierarchii typů. Jako příklad takové změny může být nahrazení typu `Integer` nadřazeným typem `Number`. `Specialization` je opačná změna.

Změna `Mutation` představuje záměnu typů, které nejsou v hierarchii typů pod sebou, tudíž jsou nekompatibilní. Jako příklad můžeme uvést změnu z typu `String` na typ `Integer`.

Pomocí výše popsaných principů lze měnit datové typy třídních proměnných, parametrů metod a konstruktorů, návratového typu metody a také meze v generice, o těch se ale zmíníme až později. V případě návratového typu metody navíc ještě testujeme změnu libovolného datového typu na `void` a naopak. Všechny tyto změny jsou testovány v prostředí třídy a rozhraní. Kombinace těchto změn nám vytváří první část testovacích dat.

Z pohledu kompatibility jsou změny datových typů binárně nekompatibilní změny, protože nahrazují dříve existující singatury novou. Zdrojová kompatibilita pak záleží na konkrétních datových typech a především také na tom, zda jde o změnu z hlediska zápisu, či čtení. Tento rozdíl můžeme vidět například mezi změnami v parametrech metody a návratových typů metody. Trochu odlišná je situace v případě konstant, které jsou z hlediska kompatibility binárně kompatibilní pouze v případě, že původní konstanta byla primitivního datového typu, nebo typu String. To je způsobeno tím, že konstanty s primitivním datovým typem, nebo typem String, jsou během překladu vloženy do kódu klienta a pro jejich změnu je nutný opětovný překlad kódu klienta.

Na závěr si ještě v tabulce 4.1 shrňme jednotlivé změny datových typů a oblasti, ve kterých se dají použít. Vidíme, že v mezích generiky jsou poměrně malé možnosti, které jsou způsobeny tím, že jako meze se nesmí využívat primitivní datové typy.

Tabulka 4.1: Možné změny datových typů v prvcích programu

Typ změny	Proměnná	Parametr metody	Návratová hodnota	Generika
Boxing	✓	✓	✓	
Unboxing	✓	✓	✓	
Widening	✓	✓	✓	
Narrowing	✓	✓	✓	
Generalisation	✓	✓	✓	✓
Specialization	✓	✓	✓	✓
Mutation	✓	✓	✓	✓
Změna na void			✓	
Změna z void			✓	

4.2 Přístupové modifikátory

Další samostatnou kapitolou jsou přístupové modifikátory. Pro popis jednotlivých přístupových úrovní můžeme využít online tutorial společnosti Oracle dostupný na [8].

Pokud budeme pomocí přístupových modifikátorů zmenšovat úroveň viditelnosti prvků, může se prvek z pohledu klienta stát neviditelným, což pochopitelně způsobí také problémy s kompatibilitou. Překvapivě však mohou nastat také situace, kdy problém způsobí zvýšení úrovně viditelnosti prvku. Takový problém může nastat z hlediska dědičnosti, kdy například zvýšíme viditelnost metody, která však v klientovi zůstane přepsána s původním modifikátorem. Tato změna je sice binárně kompatibilní, zdrojově však již ne.

V testech jsou obsaženy změny na zmenšení i zvětšení úrovně viditelnosti. Dále jsou v datech také testy, kdy je přístupový modifikátor smazán, nebo naopak přidán. Přidání modifikátoru je však testováno jen v případech, kdy je možné mít klienta využívajícího prvek i v případě absence modifikátoru.

Ne každý prvek programu však může využívat všechny modifikátory. Jejich využitelnost nám popisuje tabulka 4.2.

Použitelnost modifikátorů u proměnné a metody však závisí také na tom, zda jsou umístěné ve třídě, nebo v rozhraní. Konstanty a metody v rozhraní musí být pouze `public`, případně bez modifikátoru, kdy jsou však stejně stále vedeny jako by měly modifikátor `public`. Použití modifikátorů `private`

Tabulka 4.2: Použitelnost modifikátorů pro jednotlivé prvky programu

Modifikátor	třída	rozhraní	proměnná	metoda	konstruktor
public	✓	✓	✓	✓	✓
protected			✓	✓	✓
bez modifikátoru	✓	✓	✓	✓	✓
private			✓	✓	✓

a `protected` je zakázáno. Ve třídě naopak můžeme pro prvky využívat jakýkoliv zmíněný modifikátor. Výjimkou je abstraktní metoda, která povoluje pouze viditelné modifikátory, tedy `public` nebo `protected`. Případně může být abstraktní metoda také bez modifikátoru. Toto dává smysl z hlediska významu abstraktních tříd, protože abstraktní metodu je nutné při dědění implementovat a metoda s modifikátorem `private` by pro potomka byla neviditelná.

V rámci této skupiny jsou testovány následující změny modifikátorů:

- public na protected
- public na private
- protected na public
- protected na private
- odstranění a přidání modifikátoru

Změny modifikátoru z `private` na ostatní nemá smysl testovat, protože prvky s tímto modifikátorem nejsou z pohledu klienta dostupné, takže jejich případné zpřístupnění neovlivní kompatibilitu s klientem zkompilovaným vůči předchozí verzi knihovny.

4.3 Nepřístupové modifikátory

V jazyce Java existují také tzv. nepřístupové modifikátory, které nemění viditelnost jednotlivých prvků programu, ale přidávají jim jiné vlastnosti.

Podle [9] jsou jako nepřístupové nazývány tyto modifikátory:

- static
- final
- abstract
- synchronized
- volatile

Tyto modifikátory mají odlišné možnosti použití u jednotlivých prvků programu. Vytvořená data pro tuto část obsahují testy na přidání a odebrání modifikátorů u prvků, u který je lze využít. Použitelnost jednotlivých modifikátorů je v tabulce 4.3. Součástí dat jsou také změny mezi těmi modifikátory, které lze pro daný prvek programu využít. Rozdíl v použitelnosti jednotlivých modifikátorů je také závislý na tom, zda jej chceme využít ve třídě nebo v rozhraní.

Tabulka 4.3: Použitelnost nepřístupových modifikátorů

Modifikátor	Třída	Rozhraní	Metoda	Proměnná
static			✓	✓
final	✓		✓	✓
abstract	✓	✓	✓	
synchronized			✓	
volatile				✓

Z hlediska kompatibility je mezi třídou a rozhraním pro nepřístupové modifikátory také zásadní rozdíl, protože v rozhraní jsou téměř všechny změny kompatibilní především z toho důvodu, že konstanty mají defaultně nastavené všechny povolené modifikátory, což jsou `public`, `static` a `final`. Jejich přidávání či odebírání tedy nemá na kompatibilitu vliv. Jediný rozdíl je z pohledu metod, kdy změny mezi abstraktní a statickou metodou kompatibilní nejsou.

Třídy jsou v tomhle směru odlišné právě tím, že umožňují využití jakýkoliv ze zmíněných modifikátorů a s tím jsou spojeny další nekompatibilní možnosti změn, jako třeba manipulace s modifikátory `static` a `final`.

4.4 Výjimky

U výjimek jsou možné testovat následující situace:

- Přidání a odebrání
- Přidání a odebrání druhé výjimky
- Zpřesnění a zobecnění
- Změna výjimky
- Přidání a odebrání ošetření

`Přidání` výjimky je dle názvu vcelku jednoznačné. Jde o situaci, kdy přibude možnost vyvolání výjimky, která v předchozí verzi nebyla, tudíž na ni klient nemusí být připraven. `Odebrání` výjimky je opačná situace, kdy je možnost některé výjimky odstraněna. V testovacích datech jsou také situace, kdy je přidána, či naopak odebrána, také druhá výjimka.

`Zpřesnění` výjimky je situace podobná specializaci datového typu. Příkladem je nahrazení výjimky `IOException` výjimkou `FileNotFoundException`. `Zobecnění` je opět opačná situace.

`Změna` výjimky představuje totéž co změna `Mutation` u datových typů. Jde o změnu, kdy je výjimka změněna na jinou, která není v hierarchii výjimek nadřazená, či podřazená, původní výjimce.

`Přidání ošetření` výjimky popisuje situaci, kdy v nové verzi knihovny již není výjimka pouze vyhozena, ale je zachycena a ošetřena. `Odebrání ošetření` výjimky je tedy logicky opět opačná situace, a sice tedy situace, kdy je ošetření výjimky odstranění.

Z pohledu kompatibility jsou změny ve výjimkách binárně kompatibilní, nicméně zdrojově kompatibilní je pouze zpřesnění výjimky, kdy klient je na odchycení výjimky připraven i po změně, protože s předchozí verzí knihovny musel odchytávat nadřazenou výjimku.

4.5 Dědičnost

Do této části jsou zahrnuty testy, které ovlivňují klienta z pohledu dědičnosti. Prvními takovými testy jsou testy na přidání a odebrání abstraktní metody. S abstraktní metodou je spojená povinnost její implementace, takže jejich přidávání, či odebírání, ovlivní klienta, který danou abstraktní třídu využívá. Obdobnou situaci také popisují testy přidání a odebrání metody z rozhraní. Přidávání metod je v tomto ohledu binárně kompatibilní, protože klient je dříve využívat nemohl a Java linker neověřuje chybějící implementace metod. Zdrojově však pochopitelně takové změny kompatibilní nejsou.

Dále je zde test také na nejednoznačnost metody. Ten popisuje případ, kdy po úpravě předpisů metod dojde k situaci, že by při určitém volání metody bylo možné zavolat více metod. Data také obsahují jeden podobný test, který popisuje situaci, kdy dojde ke konfliktu defaultní metody s abstraktní metodou z jiného rozhraní, která však má stejný předpis z různých rozhraní. Ačkoliv to nemusí být na první pohled zřejmé, tyto změny jsou binárně kompatibilní, zdrojově však již ne.

Změna abstraktní metody na static metodu testuje situaci, kdy klient implementuje abstraktní metodu z předchozí verze knihovny, ale v nové verzi knihovny je tato metoda implementována s tímto modifikátorem, nebo naopak překrytá statická metoda již v nové verzi není statická.

Další situace, které mohou nastat, jsou změny v sadě implementovaných rozhraní, či odděděných tříd. Změnami jsou myšleny přidávání a odebírání děděných tříd a rozhraní. Pokud klient oddědí z knihovny třídu, kde k takové změně dojde, může nastat situace, kdy například chybí některá z volaných metod, nebo naopak přibude povinnost implementovat metody nového rozhraní. Zatímco přidání je binárně kompatibilní z již zmíněného důvodu, že linker neřeší chybějící implementace metod, odstranění je nekompatibilní změnou. Zdrojová kompatibilita přidávání závisí na tom, zda tím pro klienta plyne implementovat novou metodu. Podobným testem je také test na záměnu děděné třídy za implementování rozhraní s obdobným významem. Tato záměna je ilustrována na třídě `Thread` a rozhraní `Runnable`. Ten již je nekompatibilní.

Ovlivnit klienta lze také změnou přístupových a nepřístupových modifikátorů metod nadřazené třídy. Pokud totiž přepisujeme zděděnou metodu, nesmíme snižovat jeji viditelnost. Stejně tak nesmíme z nestatické metody udělat statickou či naopak a finální metoda nemůže být přepsána. V datech

jsou tedy testy, které simulují rozšíření a zúžení viditelnosti metody. Tyto testy jsou pouze binárně kompatibilní.

Ovlivnit klienta může také posun děděné třídy v hierarchii tříd. Takovým posunem totiž může dojít ke změně sady poskytovaných metod a parametrů, což vyvolá problém, pokud klient využívá funkcionalitu, která tímto posunem zmizí.

Zásadní vliv na funkci klienta má taky situace, kdy je děděná třída změněna na rozhraní, či naopak implementované rozhraní změněno na třídu. Takové změny pochopitelně nejsou ani kompatibilní.

Poslední testovanou oblastí je konstruktor nadřazené třídy. Pokud totiž odstraníme nadřazený konstruktor, který klientská aplikace volá, pak taková změna pochopitelně nemůže být kompatibilní.

Na závěr si tedy ještě uveďme stručný přehled testovaných oblastí:

- Přidání/odebrání abstraktní metody
- Přidání/odebrání implementovaného rozhraní
- Přidání/odebrání děděné třídy
- Nejednoznačná metoda
- Konflikt defaultní metody rozhraní
- Posun třídy v hierarchickém stromu
- Zvýšení/snížení viditelnosti přepsané metody
- Přidání/odebrání modifikátoru static přepsané metody
- Odstranění nadřazeného konstruktoru
- Změna děděné třídy na implementované rozhraní
- Změna děděné třídy na rozhraní a naopak

4.6 Generičnost

Možná největší skupinou dat, která ještě nebyla zmíněna, je generičnost. Generičnost je v Javě podporována od verze 1.5 a dle [10] existuje poměrně velká skupina možných případů, které lze s pomocí generiky testovat. Základní rozdělení do skupin je následující:

- Přidání/odebrání generické deklarace
- Parametrické typy, vícenásobné parametrické typy
- Meze, vícenásobné meze
- Wildcards (upper bounded, unbounded, lower bounded)

Přidání a odebrání generické deklarace je testováno u všech skupin z uvedeného základního rozdělení. Jde o případy, kdy předchozí verze generickou deklaraci neobsahovaly, či naopak byla generika v nové verzi odstraněna.

Parametrické typy představují situaci, kdy nechceme (nebo nepotřebujeme) znát přesný datový typ. Můžeme díky tomu vytvářet třídy, nebo metody, které dokáží pracovat s více datovými typy, takže nemusíme implementovat prakticky totožné kopie pro jednotlivé datové typy. Parametrické typy můžeme použít v deklaraci třídy, rozhraní, jako typy parametrů metod a konstruktoru a také jako návratový typ metody. Můžeme je také využívat pro proměnné. Parametrický typ je převeden na konkrétní datový typ voláním dané třídy, či metody. Tento datový typ je pak použit v celém těle zavolané třídy, případně metody.

Pomocí mezí lze omezovat rozsah platných datových typů u parametrického typu, nebo u wildcards. Platným typem pak může být také jakákoliv podtyp typu uvedeného jako meze. Meze mohou být také vícenásobné. Datový typ pak musí být podtyp všech uvedených mezí.

`Wildcards`, označované jako `<?>`, reprezentují neznámý datový typ. `Upper bounded wildcards` můžeme použít k uvolnění restrikcí na datový typ proměnné. Zadáním hranice můžeme určit, které datové typy jsou platné. Platným typem je typ uvedený jako mez a všechny jeho podtypy. Například pokud chceme pracovat s libovolným číselným typem, můžeme napsat `<? extends Number>`.

`Unbounded wildcards` můžeme využít, když máme kód, který chceme využívat s libovolným datovým typem a jeho funkčnost na zadaném typu nezávisí. Příkladem jsou implementované seznamy.
`Lower bounded widcards`, jak již můžeme intuitivně tušit, má opačný princip než `upper bounded wildcards`. Zde tedy nastavujeme spodní mez možných datových typů. Platné následně jsou tedy typy uvedené jako mez a všechny nadtypy. Příklad zápisu takového `lower bounded widcard` je `<? super Integer>`.

Mezi parametrickým typem a wildcards je několik rozdílů. Jedním z rozdílů je například možnost využití mezí pro ohraničení platných datových typů. Možnosti využití jednotlivých variant mezí popisuje tabulka 4.4

Tabulka 4.4: Použitelnost mezí v generických konstrukcích

Meze	Parametrický typ	Wildcards
bez meze	✓	✓
horní mez	✓	✓
dolní mez		✓
vícenásobné meze	✓	

K této kategorii jsou připraveny testy na přidávání a odebírání generické deklarace, parametrických typů, mezí a wildcards. U parametrických typů a mezí jsou připraveny také testy pro vícenásobný výskyt. U testů obsahujících meze jsou také připraveny testy na změny datových typů mezí. Možné změny v mezích jsou `Generalisation`, `Specialization` a `Mutation`. Tyto změny jsou popsány v sekci 4.1. Připraveny jsou také testy na prohození mezí. Připravenost testů pro jednotlivé typy konstrukcí uvádí tabulka 4.5. Oblast testů je totožná pro třídu i pro rozhraní.

Tabulka 4.5: Použití generických konstrukcích

Místo	Parametrický typ	Wildcards
Třída	✓	
Rozhraní	✓	
Parametr metody	✓	✓
Návratový typ	✓	✓

Z hlediska kompatibility je mezi parametrickými typy a wildcards další rozdíl. Z pohledu wildcards jsou totiž veškeré změny binárně kompatibilní a zdrojově kompatibilní jsou pouze některé změny datových typů. Konkrétní změny závisí na použitých mezích. Naopak z hlediska parametrických typů

jsou v metodách binárně kompatibilní pouze operace s vícenásobnými mezemi. Naopak z hlediska tvorby parametrických tříd a rozhraní jsou veškeré změny, včetně přidání a odebrání generiky, binárně kompatibilní.

4.7 Třídy

Kromě již dříve zmíněných testů jsou součástí dat také testy, které simulují změny v samotných třídách, ale neodpovídají předchozím skupinám. Prvním takovým testem je test na odstranění třídy z nové verze knihovny.

Další testy jsou připraveny pro simulaci přidávání a odebírání různých prvků třídy, konkrétně proměnné, konstruktoru, metody a parametrů konstruktoru a metody. U parametrů konstruktoru a metody jsou navíc také připraveny testy na jejich prohození, což může mít za následky různé problémy. Pokud jsou prohozeny parametry se stejným datovým typem, pak pravděpodobně dojde pouze k behaviorální nekompatibilitě, ale v případě záměny parametrů s různými datovými typy již program nepůjde ani přeložit.

Kompatibilní změny jsou však pouze přidání konstruktoru, metody a proměnné. Všechny ostatní změny jsou jak zdrojově, tak i binárně nekompatibilní.

Stručně řečeno sem patří testy na tyto změny:

- Přidání/odebrání konstruktoru
- Přidání/odebrání metody
- Změna pořadí a počtu parametrů konstruktoru
- Změna pořadí a počtu parametrů metody
- Přidání/odebrání proměnné
- Smazání třídy

4.8 Rozhraní

Zde zmíněné testy jsou prakticky totožné s těmi, které jsou zmíněny v sekci 4.7, pouze jsou vytvořen z pohledu rozhraní. Patří sem tedy testy na odstranění rozhraní z nové verze knihovny. Pokud tedy některý klient toto rozhraní implementoval, tato změna rozbije jeho kód.

Také další skupinka testů je totožná se sekcí 4.7. Tentokrát z pohledu rozhraní jsou připraveny testy také pro přidání a odebrání konstanty, metody a parametrů metody. U parametrů metody jsou opět také připraveny testy na jejich prohození. Stejně jako v sekci 4.7 je prozohení simulováno s parametry různých datových typů.

Oproti testů v třídě je tu však navíc test na změnu, kdy je v rozhraní hlavička metody z předchozí verze nahrazena implementací této metody jako `default` metody a naopak.

V rámci rozhraní jsou kompatibilní nejen přidání konstanty, ale také změna abstraktní metody na defaultní metodu. Navíc binárně kompatibilní jsou také testy na odstranění konstanty a přidání nové abstraktní metody.

V této sekci jsou tedy testy na tyto změny:

- Přidání/odebrání metody
- Změna pořadí a počtu parametrů metody
- Změna abstract metody na default a naopak
- Přidání/odebrání konstanty
- Smazání rozhraní

4.9 Pojmenování testů

Struktura názvů jednotlivých testů byla navržena s cílem jednoznačné identifikace testované změny. V názvu je tedy postupně popsáno umístění testované změny od obecnějšího umístění k přesnějšímu, například od třídy k metodě, a následně název obsahuje provedenou změnu.

Konkrétní struktura vypadá takto:

<umístění><prvek><skupina_změn><prováděná_změna>

Jako `umístění` je označeno, zda je daný test implementován jako změna ve třídě, nebo v rozhraní. U některých skupin testů (dědičnost a výjimky) je tato část vynechána, protože jsou implementovány pouze jednou.

Část nazvaná `prvek` popisuje konkrétní umístění změny, tedy například proměnná, parametr metody atd.

`Skupina_změn` popisuje, které ze skupin, zmíněných dříve v této kapitole, daný test odpovídá. Příkladem může být třeba změna datového typu

`Prováděná_změna` pak již popisuje konkrétní změnu, která je testována.

Jako příklad uvedeme test, ve kterém ve třídě u metody testujeme změnu datového typu parametru. Konkrétně změnu typu `boxing`. Takový test má tedy název

classMethodParamTypeBox

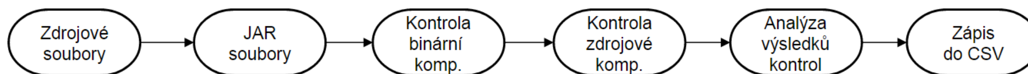
Začátek názvu, tedy `class`, určuje, že jde o test změny ve třídě, `MethodParam` označuje, že se testuje změna u parametru metody, `Type` určuje, že se testuje změna datového typu a `Box` pak určuje, že jde o změnu typu `boxing`.

5 Seznam testů

V této kapitole je uveden kompletní seznam připravených testů, které slouží k ověření funkčnosti jednotlivých nástrojů na ověření kompozice modulárních Java aplikací. Jelikož názvy testů jsou tvořeny tak, aby z nich bylo na první pohled patrné, jakou změnu daný test simuluje, omezíme se v této kapitole pouze na tabulky uvádějící, které testy obsahují zdrojově nebo binárně kompatibilní změny. Jelikož je však testů příliš na to, aby byly všechny uvedeny v jedné tabulce, budou rozděleny do více tabulek tak, aby odpovídaly jednotlivým skupinám změn popsaných v kapitole 4. Značka v tabulce značí odpovídající kompatibilitu testu. Sloupec `src` zastupuje zdrojovu kompatibilitu a sloupec `bin` zastupuje binární kompatibilitu.

5.1 Přehled testů

Data pro následující tabulky byla vygenerována postupem uvedeným na obrázku 5.1. Pomocí Ant skriptu se nejprve přeloží zdrojové soubory klienta vůči knihovně ve verzi 1, která je plně kompatibilní. Následně se pokusí slinkovat a spustit přeložené soubory klienta s přeloženými soubory knihovny ve verzi 2, čímž ověřuje binární kompatibilitu. Následně se pokusí zdrojové soubory klienta přeložit vůči knihovně ve verzi 2, čímž ověřuje zdrojovou kompatibilitu. Záznamy výsledků kontroly jednotlivých kompatibilit uloží do souborům které jsou následně vyhodnoceny jednoduchou aplikací, která ve výpisech hledala hlášení o chybě. Pokud byla chyba nalezena, byl daný test označen jako binárně či zdrojově nekompatibilní. Výsledky kompatibility následně aplikace ukládá do CSV souboru. Tento postup je opakován pro každý připravený test, čímž vznikne kompletní tabulka kompatibilit jednotlivých testů. Pro spuštění tohoto postupu pro všechny testy je navíc připraven také skript, který zajišťuje postupné spouštění pro každý test.



Obrázek 5.1: Postup ověření kompatibility

První skupinou připravených testů byly pro změny datového typu. Tabulka 5.1 obsahuje testy ve třídách pro parametry konstruktoru a proměnné.

Tabulka 5.1: Změny datových typů v konstruktoru a proměnných

Název testu	src	bin
classConstructorParamTypeBox	✓	
classConstructorParamTypeGen	✓	
classConstructorParamTypeMut		
classConstructorParamTypeNarrow		
classConstructorParamTypeSpe		
classConstructorParamTypeUnbox	✓	
classConstructorParamTypeWiden	✓	
classFieldTypeBox	✓	
classFieldTypeGen		
classFieldTypeMut		
classFieldTypeNarrow	✓	
classFieldTypeSpe	✓	
classFieldTypeUnbox	✓	
classFieldTypeWiden		

Další tabulka obsahuje změny datových typů v metodách třídy.

Tabulka 5.2: Změny datových typů v metodách třídy

Název testu	src	bin
classMethodParamTypeBox	✓	
classMethodParamTypeGen	✓	
classMethodParamTypeMut		
classMethodParamTypeNarrow		
classMethodParamTypeSpe		
classMethodParamTypeUnbox	✓	
classMethodParamTypeWiden	✓	
classMethodReturnTypeBox	✓	
classMethodReturnTypeGen		
classMethodReturnTypeChangeFromVoid	✓	
classMethodReturnTypeChangeToVoid		
classMethodReturnTypeMut		
classMethodReturnTypeNarrow	✓	
classMethodReturnTypeSpe	✓	
classMethodReturnTypeUnbox	✓	
classMethodReturnTypeWiden		

Nyní ještě zbývají testy v rozhraní. Začněme tabulkou se změnami datových typů konstant. Vidíme, že změny odpovídající situacím, kdy jsou konstanty vloženy do kódu klienta, jsou binárně kompatibilní.

Tabulka 5.3: Změny datových typů konstant v rozhraní

Název testu	src	bin
interfaceFieldTypeBox	✓	✓
interfaceFieldTypeGen		
interfaceFieldTypeMut		✓
interfaceFieldTypeNarrow	✓	✓
interfaceFieldTypeSpe	✓	
interfaceFieldTypeUnbox	✓	
interfaceFieldTypeWiden		✓

Jako poslední v této sekci již zbývají pouze změny datových typů v metodách rozhraní.

Tabulka 5.4: Změny datových typů v metodách rozhraní

Název testu	src	bin
interfaceMethodParamTypeBox	✓	
interfaceMethodParamTypeGen	✓	
interfaceMethodParamTypeMut		
interfaceMethodParamTypeNarrow		
interfaceMethodParamTypeSpe		
interfaceMethodParamTypeUnbox	✓	
interfaceMethodParamTypeWiden	✓	
interfaceMethodReturnTypeBox	✓	
interfaceMethodReturnTypeGen		
interfaceMethodReturnTypeChangeFromVoid	✓	
interfaceMethodReturnTypeChangeToVoid		
interfaceMethodReturnTypeMut		
interfaceMethodReturnTypeNarrow	✓	
interfaceMethodReturnTypeSpe	✓	
interfaceMethodReturnTypeUnbox	✓	
interfaceMethodReturnTypeWiden		

Jak nám ukazují tabulky, z pohledu kompatibility změn není velký rozdíl, zda se jedná o změnu ve třídě, nebo v rozhraní. Jediné rozdíly jsou v kompatibilitách konstant rozhraní, kde jsou na rozdíl od proměnných ve třídě

měkteré změny binárně kompatibilní. Kompatibility metod jsou totožné ve třídě i v rozhraní.

V metodách je pak také vidět rozdíl v tom, zda je změna provedena v parametru metody, nebo v návratovém typu. Je také patrné, že testy proměnných mají stejné chování jako testy návratových typů, protože testy jsou připraveny na čtení z proměnných.

Další sekcí testů jsou přístupové modifikátory. Možnosti přístupových modifikátorů jsou v rozhraní velmi omezeny, povolen je totiž pouze modifikátor `public`. Proto je možné testovat pouze přidání a odebrání tohoto modifikátoru, což však ve výsledku nemá žádný efekt, protože metody i proměnné jsou v rozhraní defaultně `public`.

Tabulka 5.5: Testy přístupových modifikátorů

Název testu	src	bin
abstractClassAccessDelModifier		
classAccessDelModifier		
classConstructorAccessDelModifier		
classFieldAccessDelModifier		
classFieldAccessChangeProtectedToPrivate		
classFieldAccessChangeProtectedToPublic	✓	✓
classFieldAccessChangePublicToPrivate		
classFieldAccessChangePublicToProtected		
classMethodAccessDelModifier		
classMethodAccessChangeProtectedToPrivate		
classMethodAccessChangeProtectedToPublic	✓	✓
classMethodAccessChangePublicToPrivate		
classMethodAccessChangePublicToProtected		
interfaceAccessDelModifier		
interfaceFieldAccessAddModifier	✓	✓
interfaceFieldAccessDelModifier	✓	✓
interfaceMethodAccessAddModifier	✓	✓
interfaceMethodAccessDelModifier	✓	✓

Změny jednotlivých modifikátorů lze tedy testovat pouze ve třídě. Testy, kde původní modifikátor je `protected`, jsou navrženy s odděděnou odpovídající třídou knihovny, aby byl možný přístup k těmto prvkům programu. Naopak testy, kde původní modifikátor je `public`, jsou bez oddědění dané třídy, proto není kompatibilní změnou ani přechod na modifikátor `protected`.

O něco větší skupinou jsou nepřístupové modifikátory, protože jejich využití je možné také v rozhraní.

V první tabulce si tedy uvedeme pouze testy prováděné ve třídě.

Tabulka 5.6: Změny nepřístupových modifikátorů ve třídě

Název testu	src	bin
abstractClassModifierDelAbstract	✓	✓
abstractClassModifierChangeAbstractToFinal		
classFieldModifierAddFinal		
classFieldModifierAddStatic	✓	
classFieldModifierAddVolatile	✓	✓
classFieldModifierDelFinal	✓	✓
classFieldModifierDelStatic		
classFieldModifierDelVolatile	✓	✓
classMethodModifierAddAbstract		
classMethodModifierAddFinal		
classMethodModifierAddStatic	✓	
classMethodModifierAddSynchronized	✓	✓
classMethodModifierDelAbstract	✓	✓
classMethodModifierDelFinal	✓	✓
classMethodModifierDelStatic		
classMethodModifierDelSynchronized	✓	✓
classMethodModifierChangeAbstractToFinal		
classMethodModifierChangeAbstractToStatic		
classMethodModifierChangeFinalToAbstract		
classMethodModifierChangeFinalToStatic	✓	
classMethodModifierChangeStaticToAbstract		
classMethodModifierChangeStaticToFinal		
classModifierAddAbstract		
classModifierAddFinal		
classModifierDelFinal	✓	✓
classModifierChangeFinalToAbstract		

Druhou skupinu tvoří testy implementované v rozhraní. Jelikož však v rozhraní nejsou povoleny všechny modifikátory použitelné ve třídě, je skupina testů pro rozhraní výrazně menší. Výsledky jsou uvedeny v tabulce 5.7.

Tabulka 5.7: Změny nepřístupových modifikátorů v rozhraní

Název testu	src	bin
interfaceFieldModifierAddFinal	✓	✓
interfaceFieldModifierAddPublic	✓	✓
interfaceFieldModifierAddStatic	✓	✓
interfaceFieldModifierDelFinal	✓	✓
interfaceFieldModifierDelPublic	✓	✓
interfaceFieldModifierDelStatic	✓	✓
interfaceMethodModifierAddAbstract	✓	✓
interfaceMethodModifierAddStatic		
interfaceMethodModifierDelAbstract	✓	✓
interfaceMethodModifierDelStatic		
interfaceMethodModifierChangeAbstractToStatic		
interfaceMethodModifierChangeStaticToAbstract		

Další skupinou testů jsou testy výjimek, kde tabulka ukazuje, že připravené testy jsou binárně kompatibilní, ale zdrojově kompatibilní je pouze jeden.

Tabulka 5.8: Testy na výjimky

Název testu	src	bin
exceptionAdd		✓
exceptionDel		✓
exceptionGen		✓
exceptionHandleAdd		✓
exceptionHandleDel		✓
exceptionMut		✓
exceptionSecondAdd		✓
exceptionSecondDel		✓
exceptionSpe	✓	✓

Následuje skupina testů na dědičnost, ke kteréi jsou kromě takto označených testů připojeny také testy abstraktní třídy, které svým smyslem odpovídají testům dědičnosti.

Tabulka 5.9: Testy na dědičnost

Název testu	src	bin
abstractClassAddInterface		✓
abstractClassDelInterface		
abstractClassMethodAdd		✓
abstractClassMethodDel		
inheritanceAmbigousMethod		✓
inheritanceDefaultMethodConflict		✓
inheritanceHiddenMethodDelStatic		✓
inheritanceHierarchyClassMoveDown	✓	✓
inheritanceHierarchyClassMoveUp		
inheritanceChangeExtendsToImplements		
inheritanceChangeClassToInterface		
inheritanceChangeInterfaceToClass		
inheritanceImplementedSetAdd	✓	✓
inheritanceImplementedSetDel		
inheritanceMethodAccessChangeProtectedToPublic		✓
inheritanceMethodAccessChangePublicToProtected		✓
inheritanceOverridedMethodAddStatic		
inheritanceRemoveSuperConstructor		
inheritanceSuperclassSetAdd	✓	✓
inheritanceSuperclassSetDel		

Největší skupinou testů jsou generiky, které tvoří přibližně třetinu všech testů. To je způsobeno velkým množstvím možných změn z hlediska mezí a wildcards, které se v generice využívají. Z tohoto důvodu rozčleníme seznam testů do více tabulek, aby byla zachována přehlednost.

Nejdříve se však zaměříme na testy parametrických typů. Ty rozdělíme do tabulek dle toho, zda jsou připraveny pro třídu, nebo rozhraní. V první tabulce jsou uvedeny testy s parametrickými typy ve třídě.

Tabulka 5.10: Testy parametrických typů ve třídě

Název testu	src	bin
classConstructorGenericsTypeParamAdd	✓	
classConstructorGenericsTypeParamAddSecond	✓	✓
classConstructorGenericsTypeParamBoundAdd		
classConstructorGenericsTypeParamBoundAddSecond		✓
classConstructorGenericsTypeParamBoundDel	✓	
classConstructorGenericsTypeParamBoundDelSecond	✓	✓
classConstructorGenericsTypeParamBoundsSwap		
classConstructorGenericsTypeParamDel		
classConstructorGenericsTypeParamDelSecond	✓	✓
classGenericsTypeParamAdd	✓	✓
classGenericsTypeParamAddBound		✓
classGenericsTypeParamAddSecond		✓
classGenericsTypeParamAddSecondBound		✓
classGenericsTypeParamBoundGen	✓	✓
classGenericsTypeParamBoundMut		✓
classGenericsTypeParamBoundSpe		✓
classGenericsTypeParamBoundsSwap		✓
classGenericsTypeParamDel		✓
classGenericsTypeParamDelBound	✓	✓
classGenericsTypeParamDelSecond		✓
classGenericsTypeParamDelSecondBound	✓	✓
classMethodGenericsReturnTypeParamAdd	✓	
classMethodGenericsReturnTypeParamDel		
classMethodGenericsTypeParamAdd	✓	
classMethodGenericsTypeParamAddSecond	✓	✓
classMethodGenericsTypeParamBoundAdd		
classMethodGenericsTypeParamBoundAddSecond		✓
classMethodGenericsTypeParamBoundDel	✓	
classMethodGenericsTypeParamBoundDelSecond	✓	✓
classMethodGenericsTypeParamBoundsSwap		
classMethodGenericsTypeParamDel		
classMethodGenericsTypeParamDelSecond	✓	✓

Další tabulkou jsou testy parametrických typů v rozhraní.

Tabulka 5.11: Testy parametrických typů v rozhraní

Název testu	src	bin
interfaceGenericsTypeParamAdd	✓	✓
interfaceGenericsTypeParamAddBound		✓
interfaceGenericsTypeParamAddSecond		✓
interfaceGenericsTypeParamBoundAddSecond		✓
interfaceGenericsTypeParamBoundDelSecond	✓	✓
interfaceGenericsTypeParamBoundGen	✓	✓
interfaceGenericsTypeParamBoundMut		✓
interfaceGenericsTypeParamBoundSpe		✓
interfaceGenericsTypeParamBoundsSwap		✓
interfaceGenericsTypeParamDel		✓
interfaceGenericsTypeParamDelBound	✓	✓
interfaceGenericsTypeParamDelSecond		✓
interfaceMethodGenericsReturnTypeParamAdd	✓	
interfaceMethodGenericsReturnTypeParamDel	✓	
interfaceMethodGenericsTypeParamAdd	✓	
interfaceMethodGenericsTypeParamAddSecond	✓	✓
interfaceMethodGenericsTypeParamBoundAdd		
interfaceMethodGenericsTypeParamBoundAddSecond		✓
interfaceMethodGenericsTypeParamBoundDel	✓	
interfaceMethodGenericsTypeParamBoundDelSecond	✓	✓
interfaceMethodGenericsTypeParamBoundsSwap		
interfaceMethodGenericsTypeParamDel		
interfaceMethodGenericsTypeParamDelSecond	✓	✓

Vidíme, že změny parametrických typů rozhraní i tříd jsou binárně kompatibilní. Jediné potíže z pohledu binární kompatibility tedy přicházejí v případě změn v metodách, kde jsou výsledky shodné pro metody ve třídě i v rozhraní. Z hlediska kompatibilit také není rozdíl mezi konstruktorem a běžnou metodou.

Další velkou částí generik jsou wildcards, které navíc mohou mít dolní i horní meze. Pro větší přehlednost wildcards rozdělíme do tří tabulek. Jednu pro obecné změny jako přidání a odebrání wildcards, další pro wildcards s dolní mezí a poslední tabulku pro wildcards s horní mezí.

V této tabulce jsou obecné testy s wildcards.

Tabulka 5.12: Testy obecných změn s wildcards

Název testu	src	bin
classConstructorGenericsParamWildcardsAdd	✓	✓
classConstructorGenericsParamWildcardsBoundSwapLowerToUpper		✓
classConstructorGenericsParamWildcardsBoundSwapUpperToLower		✓
classConstructorGenericsParamWildcardsDel		✓
classMethodGenericsParamWildcardsAdd	✓	✓
classMethodGenericsParamWildcardsBoundSwapLowerToUpper		✓
classMethodGenericsParamWildcardsBoundSwapUpperToLower		✓
classMethodGenericsParamWildcardsDel		✓
interfaceMethodGenericsParamWildcardsAdd	✓	✓
interfaceMethodGenericsParamWildcardsBoundSwapLowerToUpper		✓
interfaceMethodGenericsParamWildcardsBoundSwapUpperToLower		✓
interfaceMethodGenericsParamWildcardsDel		✓

V další tabulce jsou testy se spodní mezí wildcards.

Tabulka 5.13: Testy změn se spodní mezí wildcards

Název testu	src	bin
classConstructorGenericsParamWildcardsLowerBoundAdd		✓
classConstructorGenericsParamWildcardsLowerBoundDel	✓	✓
classConstructorGenericsParamWildcardsLowerBoundGen		✓
classConstructorGenericsParamWildcardsLowerBoundMut		✓
classConstructorGenericsParamWildcardsLowerBoundSpe	✓	✓
classConstructorGenericsParamWildcardsLowerBoundSwap		✓
classMethodGenericsParamWildcardsLowerBoundAdd		✓
classMethodGenericsParamWildcardsLowerBoundDel	✓	✓
classMethodGenericsParamWildcardsLowerBoundGen		✓
classMethodGenericsParamWildcardsLowerBoundMut		✓
classMethodGenericsParamWildcardsLowerBoundSpe	✓	✓
classMethodGenericsParamWildcardsLowerBoundSwap		✓
interfaceMethodGenericsParamWildcardsLowerBoundAdd		✓
interfaceMethodGenericsParamWildcardsLowerBoundDel	✓	✓
interfaceMethodGenericsParamWildcardsLowerBoundGen		✓
interfaceMethodGenericsParamWildcardsLowerBoundMut		✓
interfaceMethodGenericsParamWildcardsLowerBoundSpe	✓	✓
interfaceMethodGenericsParamWildcardsLowerBoundSwap		✓

V poslední tabulce jsou testy s horní mezí wildcards. Testy jsou totožné s těmi s dolní mezí, pouze je dolní mez zaměněna za horní. Rozdíl je však ve výsledcích při určitých změnách datového typu mezí.

Tabulka 5.14: Testy změn se spodní mezí wildcards

Název testu	src	bin
classConstructorGenericsParamWildcardsUpperBoundAdd		✓
classConstructorGenericsParamWildcardsUpperBoundDel	✓	✓
classConstructorGenericsParamWildcardsUpperBoundGen	✓	✓
classConstructorGenericsParamWildcardsUpperBoundMut		✓
classConstructorGenericsParamWildcardsUpperBoundSpe		✓
classConstructorGenericsParamWildcardsUpperBoundSwap		✓
classMethodGenericsParamWildcardsUpperBoundAdd		✓
classMethodGenericsParamWildcardsUpperBoundDel	✓	✓
classMethodGenericsParamWildcardsUpperBoundGen	✓	✓
classMethodGenericsParamWildcardsUpperBoundMut		✓
classMethodGenericsParamWildcardsUpperBoundSpe		✓
classMethodGenericsParamWildcardsUpperBoundSwap		✓
interfaceMethodGenericsParamWildcardsUpperBoundAdd		✓
interfaceMethodGenericsParamWildcardsUpperBoundDel	✓	✓
interfaceMethodGenericsParamWildcardsUpperBoundGen	✓	✓
interfaceMethodGenericsParamWildcardsUpperBoundMut		✓
interfaceMethodGenericsParamWildcardsUpperBoundSpe		✓
interfaceMethodGenericsParamWildcardsUpperBoundSwap		✓

Z tabulek je zřejmé, že změny ohledně wildcards nejsou samy o sobě binárně nekompatibilní. Také můžeme vidět, že změny v rozhraní jsou totožné se změnami ve třídách, protože testy v rozhraní jsou strukturovány s anonymní implementací uvnitř definice rozhraní.

Principy změn datových typů odpovídají změnám popsaným v kapitole 4.1. Ne všechny jsou však použitelné, protože jako meze nelze použít primitivní datové typy, tím se možné změny limitují na `Generalisation`, `Mutation` a `Specialization`. Spolu s tím je testováno také prohození mezí.

Na závěr jsou zde ještě shrnutí zbylých testů tříd a rozhraní, které neodpovídají předchozím sekcím. Nejdříve si uvedeme testy tříd.

Tabulka 5.15: Testy tříd

Název testu	src	bin
classConstructorAdd	✓	✓
classConstructorDel		
classConstructorParamAdd		
classConstructorParamDel		
classConstructorParamSwap		
classDel		
classFieldAdd	✓	✓
classFieldDel		
classChangeToInterface		
classMethodAdd	✓	✓
classMethodDel		
classMethodParamAdd		
classMethodParamDel		
classMethodParamSwap		

V poslední tabulce jsou uvedeny zbylé testy rozhraní.

Tabulka 5.16: Testy rozhraní

Název testu	src	bin
interfaceDel		
interfaceFieldAdd	✓	✓
interfaceFieldDel		
interfaceChangeDefaultToPrototype		
interfaceChangePrototypeToDefault	✓	✓
interfaceMethodAdd		✓
interfaceMethodDel		
interfaceMethodParamAdd		
interfaceMethodParamDel		
interfaceMethodParamSwap		

5.2 Shrnutí testů

Nyní zde ještě uvedeme závěrečné shrnutí, ve kterém shrneme počet testů v jednotlivých sekcích a také poměry zdrojově a binárně nekompatibilních testů.

Tabulka 5.17: Souhrn testů

Skupina testů	Počet testů	Zdrojově nekompatibilní	Binárně nekompatibilní
Změna datového typu	53	23	49
Přístupové modifikátory	18	12	12
Nepřístupové modifikátory	38	18	21
Výjimky	9	8	0
Dědičnost	20	17	10
Generičnost	103	62	19
Třídy	14	11	11
Rozhraní	10	8	7
Celkem	265	159	129

Z tabulky je zřejmé, že ne všechny testy simulují nekompatibilní změny. Testy také nejsou striktně pouze zdrojově či binárně nekompatibilní, ale některé porušují oba druhy kompatibility. V následující tabulce lze ještě vidět srovnání počtů připravených testů ve třídách a v rozhraních.

Tabulka 5.18: Počet testů ve třídách a rozhraních

Skupina testů	Počet testů	Třída	Rozhraní
Změna datového typu	53	30	23
Přístupové modifikátory	18	13	5
Nepřístupové modifikátory	38	26	12
Výjimky	9	9	0
Dědičnost	20	18	2
Generičnost	103	64	39
Třídy	14	14	0
Rozhraní	10	0	10
Celkem	265	174	91

6 Zhodnocení nástrojů

V této kapitole se dostáváme k samotnému testování nástrojů. Nejprve nástroje projdou testováním na připravených datech. Na to následně budou navázány také mimofunkční charakteristiky definované v sekci 3.6.

6.1 Výsledky testů

Začneme nejprve testováním té nejdůležitější vlastnosti nástrojů, a sice schopnosti odhalovat nekompatibilní změny v aplikacích. Z tabulek v této kapitole budou vynechány testy, které jsou zdrojově i binárně kompatibilní. Pro otestování nástroje Animal Sniffer byl podle návodu na stránce [5] vytvořen Ant skript a pro nástroj JaCC byl vytvořen jednoduchý klient, který spustil nástroj se zadanými knihovnami a výstup vypsal do konzole. Byl také připraven skript, který nejprve spustí překlad zdrojových souborů a vygenerování JAR souborů pro klienta i obě verze knihoven. Následně postupně spouští jednotlivé testované nástroje a jejich výstupy přesměruje do textových souborů. Informace o nalezených problémech pro následující tabulky pak již byly získávány ručním průchodem vygenerovaných souborů. Značka v tabulce znamená, že daný nástroj dokázal odhalit nekompatibilní změnu simulovanou odpovídajícím testem.

Začneme opět se změnami datových typů. Konkrétně nejdříve se změnami datových typů konstant v rozhraních.

Tabulka 6.1: Nalezení problémů změn datových typů konstant v rozhraní

Název testu	JaCC	JAPICC	jdeps	AS
interfaceFieldTypeGen	✓			✓
interfaceFieldTypeMut				
interfaceFieldTypeSpe	✓			✓
interfaceFieldTypeUnbox	✓			✓
interfaceFieldTypeWiden				

Nástroje JaCC a Animal Sniffer tedy dokázaly v této sekci odhalit změny, u kterých nedošlo ke vložení konstanty do kódu klienta v době překladu. Zbylé dva nástroje nedokázaly odhalit nic.

Červeně zvýrazněné testy v tabulce 6.1 však simulují změnu, při které došlo při překladu ke vložení konstanty do kódu klienta. Nedetekování těchto testů je tedy chybou pouze z hlediska zpětné kompatibility, takže chybné tyto testy jsou pouze pro nástroj JAPICC. Z tohoto důvodu nebudou tyto testy v závěrečném vyhodnocení započteny, aby bylo možné spravedlivě nástroje porovnat.

Podívejme se nyní dále na jejich výsledky také v testech pro třídy.

Tabulka 6.2: Nalezení problémů změn datových typů v metodách třídy

Název testu	JaCC	JAPICC	jdeps	AS
classMethodParamTypeBox	✓	✓		✓
classMethodParamTypeGen	✓	✓		✓
classMethodParamTypeMut	✓	✓		✓
classMethodParamTypeNarrow	✓	✓		✓
classMethodParamTypeSpe	✓	✓		✓
classMethodParamTypeUnbox	✓	✓		✓
classMethodParamTypeWiden	✓	✓		✓
classMethodReturnTypeBox	✓	✓		✓
classMethodReturnTypeGen	✓	✓		✓
classMethodReturnTypeChangeFromVoid	✓	✓		✓
classMethodReturnTypeChangeToVoid	✓	✓		✓
classMethodReturnTypeMut	✓	✓		✓
classMethodReturnTypeNarrow	✓	✓		✓
classMethodReturnTypeSpe	✓	✓		✓
classMethodReturnTypeUnbox	✓	✓		✓
classMethodReturnTypeWiden	✓	✓		✓

V tabulkách 6.2 a 6.3 můžeme vidět, že ve třídách byly tři nástroje v této oblasti změn stoprocentní. Z pohledu tříd tedy pro nástroje není problém detekovat změny datových typů.

Tabulka 6.3: Nalezení problémů změn datových typů ve třídách

Název testu	JaCC	JAPICC	jdeps	AS
classConstructorParamTypeBox	✓	✓		✓
classConstructorParamTypeGen	✓	✓		✓
classConstructorParamTypeMut	✓	✓		✓
classConstructorParamTypeNarrow	✓	✓		✓
classConstructorParamTypeSpe	✓	✓		✓
classConstructorParamTypeUnbox	✓	✓		✓
classConstructorParamTypeWiden	✓	✓		✓
classFieldTypeBox	✓	✓		✓
classFieldTypeGen	✓	✓		✓
classFieldTypeMut	✓	✓		✓
classFieldTypeNarrow	✓	✓		✓
classFieldTypeSpe	✓	✓		✓
classFieldTypeUnbox	✓	✓		✓
classFieldTypeWiden	✓	✓		✓

Nyní se ještě podívejme, jak úspěšné budou ve zbývajících testech pro rozhraní.

Tabulka 6.4: Nalezení problémů změn datových typů v metodách rozhraní

Název testu	JaCC	JAPICC	jdeps	AS
interfaceMethodParamTypeBox	✓	✓		✓
interfaceMethodParamTypeGen	✓	✓		✓
interfaceMethodParamTypeMut	✓	✓		✓
interfaceMethodParamTypeNarrow	✓	✓		✓
interfaceMethodParamTypeSpe	✓	✓		✓
interfaceMethodParamTypeUnbox	✓	✓		✓
interfaceMethodParamTypeWiden	✓	✓		✓
interfaceMethodReturnTypeBox	✓	✓		✓
interfaceMethodReturnTypeGen	✓	✓		✓
interfaceMethodReturnTypeChangeFromVoid	✓	✓		✓
interfaceMethodReturnTypeChangeToVoid	✓	✓		✓
interfaceMethodReturnTypeMut	✓	✓		✓
interfaceMethodReturnTypeNarrow	✓	✓		✓
interfaceMethodReturnTypeSpe	✓	✓		✓
interfaceMethodReturnTypeUnbox	✓	✓		✓
interfaceMethodReturnTypeWiden	✓	✓		✓

Jak vidíme z předchozích tabulek, změny datových typů v rámci metod nedělaly nástrojům problémy ve třídách ani v rozhraních. Jediné problémy tedy zatím nastaly se změnami u konstant v rozhraních, kde pouze nástroje JaCC a Animal Sniffer dokázaly detekovat alespoň binárně nekompatibilní změny.

Další sekcí testů jsou testy na přístupové modifikátory, které jsou v této kapitole výrazně zredukovány, protože obsahují řadu zdrojově i binárně kompatibilních testů. V tabulce 6.5 tedy vidíme výsledky testů této sekce. Jednoznačně nejlepším nástrojem je zde nástroj JAPICC. Ostatní nástroje nejsou v této oblasti příliš spolehlivé. V této skupině testů je pouze jeden test pro rozhraní, protože v rozhraní není možné využívat prakticky žádné přístupové modifikátory.

Tabulka 6.5: Nalezení problémů přístupových modifikátorů

Název testu	JaCC	JAPICC	jdeps	AS
abstractClassAccessDelModifier		✓		
classAccessDelModifier		✓		
classConstructorAccessDelModifier		✓		
classFieldAccessDelModifier		✓		
classFieldAccessChangeProtectedToPrivate		✓		
classFieldAccessChangePublicToPrivate	✓	✓		
classFieldAccessChangePublicToProtected		✓		
classMethodAccessDelModifier		✓		
classMethodAccessChangeProtectedToPrivate		✓		
classMethodAccessChangePublicToPrivate	✓	✓		
classMethodAccessChangePublicToProtected		✓		
interfaceAccessDelModifier				

Dále tu máme testy nepřístupových modifikátorů. Zde opět máme velmi omezené zastoupení testů pro rozhraní, protože i v oblasti nepřístupových modifikátorů platí pro rozhraní jistá omezení. Především však byla velká část těchto testů zdrojově i binárně kompatibilní. Z tabulky 6.6 je evidentní, že největší potíže s nepřístupovými modifikátory má nástroj Animal Sniffer. Naopak nejlépe se zatím jeví nástroj JAPICC.

Tabulka 6.6: Nalezení problémů změn nepřístupových modifikátorů

Název testu	JaCC	JAPICC	jdeps	AS
abstractClassModifierChangeAbstractToFinal		✓		
classFieldModifierAddFinal		✓		
classFieldModifierAddStatic	✓	✓		
classFieldModifierDelStatic		✓		
classMethodModifierAddAbstract	✓	✓		
classMethodModifierAddFinal	✓	✓		
classMethodModifierAddStatic	✓	✓		
classMethodModifierDelStatic	✓	✓		
classMethodModifierChangeAbstractToFinal	✓	✓		
classMethodModifierChangeAbstractToStatic	✓	✓		
classMethodModifierChangeFinalToAbstract	✓	✓		
classMethodModifierChangeFinalToStatic	✓	✓		
classMethodModifierChangeStaticToAbstract	✓	✓		
classMethodModifierChangeStaticToFinal	✓	✓		
classModifierAddAbstrac		✓		
classModifierAddFinal		✓		
classModifierChangeFinalToAbstract		✓		
interfaceMethodModifierAddStatic	✓	✓		
interfaceMethodModifierDelStatic	✓	✓		
interfaceMethodModifierChangeAbstractToStatic	✓	✓		
interfaceMethodModifierChangeStaticToAbstract	✓	✓		

Nyní následuje skupinka testů na výjimky, kde možná trochu překvapivě dokázal připravené testy odhalit jediný nástroj, a sice nástroj JAPICC. Ostatní nástroje evidentně nejsou schopny zachytit změny ve výjimkách.

Tabulka 6.7: Nalezení problémů změn ve výjimkách

Název testu	JaCC	JAPICC	jdeps	AS
exceptionAdd		✓		
exceptionDel		✓		
exceptionGen		✓		
exceptionHandleAdd		✓		
exceptionHandleDel		✓		
exceptionMut		✓		
exceptionSecondAdd		✓		
exceptionSecondDel		✓		

Následují testy na dědičnost, ve kterých opět dokázal některé problémy odhalit také nástroj Animal Sniffer. Stále však na nástroje JaCC a JAPICC evidentně nestačí. Nicméně v kategorii dědičnosti můžeme vypozorovat oproti předchozím skupinám horší výsledky také u tří prozatím nejlepších nástrojů.

Tabulka 6.8: Nalezení problémů s dědičností

Název testu	JaCC	JAPICC	jdeps	AS
abstractClassAddInterface	✓	✓		
abstractClassDelInterface	✓	✓		✓
abstractClassMethodAdd	✓	✓		
abstractClassMethodDel	✓	✓		✓
inheritanceAmbigousMethod				
inheritanceDefaultMethodConflict				
inheritanceHiddenMethodDelStatic	✓	✓		
inheritanceHierarchyClassMoveUp		✓		
inheritanceChangeExtendsToImplements		✓		
inheritanceChangeClassToInterface	✓	✓		✓
inheritanceChangeInterfaceToClass				
inheritanceImplementedSetDel	✓	✓		✓
inheritanceMethodAccessChangeProtectedToPublic	✓			
inheritanceMethodAccessChangePublicToProtected		✓		
inheritanceOverridedMethodAddStatic	✓	✓		
inheritanceRemoveSuperConstructor	✓	✓		✓
inheritanceSuperclassSetDel	✓	✓		✓

Nyní přichází opět na řadu nejpočetnější skupina testů, a sice generika. V této kapitole začneme testy na wildcards. Z důvodu velmi dlouhých názvů testů jsou zde názvy nástrojů nahrazeny zkratkami. Pořadí nástrojů zůstalo neměnné, takže zkratka A zastupuje nástroj JaCC, zkratka B odpovídá nástroji JAPICC, zkratka C je nástroj jdeps a zkratka D odpovídá nástroji Animal Sniffer.

Tabulka 6.9: Nalezení problémů obecných změn s wildcards

Název testu	A	B	C	D
classConstructorGenericsParamWildcardsBoundSwapLowerToUpper				
classConstructorGenericsParamWildcardsBoundSwapUpperToLower				
classConstructorGenericsParamWildcardsDel				
classMethodGenericsParamWildcardsBoundSwapLowerToUpper				
classMethodGenericsParamWildcardsBoundSwapUpperToLower				
classMethodGenericsParamWildcardsDel				
interfaceMethodGenericsParamWildcardsBoundSwapLowerToUpper				
interfaceMethodGenericsParamWildcardsBoundSwapUpperToLower				
interfaceMethodGenericsParamWildcardsDel				

Již po úvodní tabulce je zřejmé, že generiky budou nástrojům dělat největší potíže. Zatím žádný nástroj nedokáže pracovat s wildcards.

Tabulka 6.10: Nalezení problémů změn se spodní mezí wildcards

Název testu	A	B	C	D
classConstructorGenericsParamWildcardsLowerBoundAdd				
classConstructorGenericsParamWildcardsLowerBoundGen				
classConstructorGenericsParamWildcardsLowerBoundMut				
classConstructorGenericsParamWildcardsLowerBoundSwap				
classMethodGenericsParamWildcardsLowerBoundAdd				
classMethodGenericsParamWildcardsLowerBoundGen				
classMethodGenericsParamWildcardsLowerBoundMut				
classMethodGenericsParamWildcardsLowerBoundSwap				
interfaceMethodGenericsParamWildcardsLowerBoundAdd				
interfaceMethodGenericsParamWildcardsLowerBoundGen				
interfaceMethodGenericsParamWildcardsLowerBoundMut				
interfaceMethodGenericsParamWildcardsLowerBoundSwap				

Jak se ukázalo, žádný nástroj v připravených testech nedokázal detekovat problémy s wildcards. Potvrzuje se tedy skutečnost, že generiky jsou nejobtížnější oblastí pro vvšechny nástroje. A to i pro nástroj JAPICC kontrolující zpětnou kompatibilitu.

Tabulka 6.11: Nalezení problémů změn s horní mezí wildcards

Název testu	A	B	C	D
classConstructorGenericsParamWildcardsUpperBoundAdd				
classConstructorGenericsParamWildcardsUpperBoundMut				
classConstructorGenericsParamWildcardsUpperBoundSpe				
classConstructorGenericsParamWildcardsUpperBoundSwap				
classMethodGenericsParamWildcardsUpperBoundAdd				
classMethodGenericsParamWildcardsUpperBoundMut				
classMethodGenericsParamWildcardsUpperBoundSpe				
classMethodGenericsParamWildcardsUpperBoundSwap				
interfaceMethodGenericsParamWildcardsUpperBoundAdd				
interfaceMethodGenericsParamWildcardsUpperBoundMut				
interfaceMethodGenericsParamWildcardsUpperBoundSpe				
interfaceMethodGenericsParamWildcardsUpperBoundSwap				

Co se týče parametrických typů ve třídách, dokáží je nástroje JaCC, JAPICC a Animal Sniffer odhalovat totožně. Žádný z těchto nástrojů však nedokáže odhalit změny parametrických typů třídy ani rozhraní, dokáží to pouze v metodách. Podívejme se nejdříve na výsledky v rozhraní.

Tabulka 6.12: Nalezení problémů parametrických typů v rozhraní

Název testu	A	B	C	D
interfaceGenericsTypeParamAddBound				
interfaceGenericsTypeParamAddSecond				
interfaceGenericsTypeParamBoundAddSecond				
interfaceGenericsTypeParamBoundMut				
interfaceGenericsTypeParamBoundSpe				
interfaceGenericsTypeParamBoundsSwap				
interfaceGenericsTypeParamDel				
interfaceGenericsTypeParamDelSecond				
interfaceMethodGenericsReturnTypeParamAdd	✓	✓		✓
interfaceMethodGenericsReturnTypeParamDel	✓	✓		✓
interfaceMethodGenericsTypeParamAdd	✓	✓		✓
interfaceMethodGenericsTypeParamBoundAdd	✓	✓		✓
interfaceMethodGenericsTypeParamBoundAddSecond				
interfaceMethodGenericsTypeParamBoundDel	✓	✓		✓
interfaceMethodGenericsTypeParamBoundsSwap	✓	✓		✓
interfaceMethodGenericsTypeParamDel	✓	✓		✓

Nyní se ještě podíváme na výsledky v rámci tříd.

Tabulka 6.13: Nalezení problémů parametrických typů ve třídě

Název testu	A	B	C	D
classConstructorGenericsTypeParamAdd	✓	✓		✓
classConstructorGenericsTypeParamBoundAdd	✓	✓		✓
classConstructorGenericsTypeParamBoundAddSecond				
classConstructorGenericsTypeParamBoundDel	✓	✓		✓
classConstructorGenericsTypeParamBoundsSwap	✓	✓		✓
classConstructorGenericsTypeParamDel	✓	✓		✓
classGenericsTypeParamAddBound				
classGenericsTypeParamAddSecond				
classGenericsTypeParamAddSecondBound				
classGenericsTypeParamBoundMut				
classGenericsTypeParamBoundSpe				
classGenericsTypeParamBoundsSwap				
classGenericsTypeParamDel				
classGenericsTypeParamDelSecond				
classMethodGenericsReturnTypeParamAdd	✓	✓		✓
classMethodGenericsReturnTypeParamDel	✓	✓		✓
classMethodGenericsTypeParamAdd	✓	✓		✓
classMethodGenericsTypeParamBoundAdd	✓	✓		✓
classMethodGenericsTypeParamBoundAddSecond				
classMethodGenericsTypeParamBoundDel	✓	✓		✓
classMethodGenericsTypeParamBoundsSwap	✓	✓		✓
classMethodGenericsTypeParamDel	✓	✓		✓

Předchozí tabulky nám evidentně prokazují, že v rámci parametrických typů jsou skutečně největším problémem parametrické třídy a rozhraní, zatímco s metodami si nástroje vcelku dokáží poradit.

Na závěr tu již opět máme pouze testy tříd a rozhraní, které neodpovídají předchozím sekcím. Začněme tedy opět s třídami.

Tabulka 6.14: Nalezení problémů v testech tříd

Název testu	JaCC	JAPICC	jdeps	AS
classConstructorDel	✓	✓		✓
classConstructorParamAdd	✓	✓		✓
classConstructorParamDel	✓	✓		✓
classConstructorParamSwap	✓	✓		✓
classDel	✓	✓	✓	✓
classFieldDel	✓	✓		✓
classChangeToInterface	✓	✓		✓
classMethodDel	✓	✓		✓
classMethodParamAdd	✓	✓		✓
classMethodParamDel	✓	✓		✓
classMethodParamSwap	✓	✓		✓

A ještě na úplný závěr posledních několik testů z rozhraní.

Tabulka 6.15: Nalezení problémů v testech rozhraní

Název testu	JaCC	JAPICC	jdeps	AS
interfaceDel	✓		✓	
interfaceFieldDel	✓			✓
interfaceChangeDefaultToPrototype	✓			
interfaceMethodAdd	✓	✓		
interfaceMethodDel	✓	✓		✓
interfaceMethodParamAdd	✓	✓		✓
interfaceMethodParamDel	✓	✓		✓
interfaceMethodParamSwap	✓	✓		✓

V rámci dodatečných testů na rozhraní můžeme vidět, že nejlépe je na tom v této oblasti nástroj JaCC. Vzhledem k předchozím výsledkům má v této oblasti nástroj JAPICC až překvapivé nedostatky v odhalování změn.

6.1.1 Shrnutí výsledků

Po průchodu výsledků jednotlivých nástrojů pro všechny testy již zbývá provést pouze závěrečný souhrn. Nejprve si uveďme celkový souhrn výsledků pro všechny testy. Můžeme vidět, že nástroj JAPICC dopadl nejlépe. Nástroj JaCC odhalil o 19 testů méně. Dle testů by však tento rozdíl neměl být způsoben jiným zaměřením nástrojů. Nástroj Animal Sniffer je až nečekaně daleko za první dvojicí a poslední je podle očekávání nástroj jdeps, který řeší problémy pouze na úrovni tříd. Jak již bylo zmíněno, ve výsledcích nejsou započteny zvýrazněné řádky tabulky 6.1.

Tabulka 6.16: Počty nalezených problémů

Skupina testů	Počet testů	JaCC	JAPICC	jdeps	AS
Změna datového typu	49	49	46	0	49
Přístupové modifikátory	12	2	11	0	0
Nepřístupové modifikátory	21	15	21	0	0
Výjimky	8	0	8	0	0
Dědičnost	17	11	13	0	6
Generičnost	71	19	19	0	19
Třídy	11	11	11	1	11
Rozhraní	8	8	5	1	5
Celkem	197	115	134	2	90

Pokud se na výsledky podíváme napříč jednotlivými skupinami testů, můžeme vidět, že nejsnažší jsou pro nástroje změny datového typu. Naopak jednoznačně nejtěžší jsou evidentně testy z oblasti generiky, kde sice byly výsledky nástrojů totožné, ale většina testů zůstala neodhalena. Překvapivé výsledky nastaly také v rámci testů na výjimky, kde jediný nástroj, který je zřejmě na tyto změny připraven, je nástroj JAPICC.

Nyní se ještě podíváme na výsledky z několika různých dílčích pohledů, které nám mohou poskytnou zajímavý náhled na rozdělení testovacích dat dle typů kompatibility, nebo dle dělení na testy ve třídách a v rozhraní.

Nejprve se podívejme odděleně na schopnosti nástrojů odhalovat zdrojové a binární nekompatibility.

Tabulka 6.17: Počty nalezených problémů zdrojové kompatibility

Skupina testů	Počet testů	JaCC	JAPICC	jdeps	AS
Změna datového typu	21	21	20	0	21
Přístupové modifikátory	12	2	11	0	0
Nepřístupové modifikátory	18	12	18	0	0
Výjimky	8	0	8	0	0
Dědičnost	17	11	13	0	6
Generičnost	62	10	10	0	10
Třídy	11	11	11	1	11
Rozhraní	8	8	5	1	5
Celkem	157	75	96	2	53

Tabulka 6.18: Počty nalezených problémů binární kompatibility

Skupina testů	Počet testů	JaCC	JAPICC	jdeps	AS
Změna datového typu	49	49	46	0	49
Přístupové modifikátory	12	2	11	0	0
Nepřístupové modifikátory	21	15	21	0	0
Výjimky	0	0	0	0	0
Dědičnost	10	7	9	0	6
Generičnost	19	19	19	0	19
Třídy	11	11	11	1	11
Rozhraní	7	7	4	1	5
Celkem	129	110	121	2	90

Obecně úspěšnější tedy všechny nástroje byly při hledání binárních nekompatibilit. V obou tabulkách jsou však započteny také testy, které jsou zdrojově i binárně nekompatibilní a ne každý nástroj poskytuje možnost ověřit, zda daný problém odhalil jako zdrojově či binárně nekompatibilní. Při testech se také ukázalo, že ačkoliv to autoři nikde nezmiňují, tak nástroj Animal Sniffer dokáže odhalovat pouze problémy s binární kompatibilitou. Právě u něj také všechny záznamy v tabulce pro zdrojovou kompatibilitu odpovídají testům, které jsou zároveň také binárně nekompatibilní.

Další statistikou, na kterou se ještě podíváme, je porovnání schopností nástrojů hledat nekompatibility ve třídách a v rozhraních.

Tabulka 6.19: Počty nalezených problémů ve třídách

Skupina testů	Počet testů	JaCC	JAPICC	jdeps	AS
Změna datového typu	30	30	30	0	30
Přístupové modifikátory	11	2	11	0	0
Nepřístupové modifikátory	17	11	17	0	0
Výjimky	8	0	8	0	0
Dědičnost	15	11	13	0	6
Generičnost	44	12	12	0	12
Třídy	11	11	11	1	11
Rozhraní	0	0	0	0	0
Celkem	136	77	102	1	59

Tabulka 6.20: Počty nalezených problémů v rozhraních

Skupina testů	Počet testů	JaCC	JAPICC	jdeps	AS
Změna datového typu	19	19	16	0	19
Přístupové modifikátory	1	0	0	0	0
Nepřístupové modifikátory	4	4	4	0	0
Výjimky	0	0	0	0	0
Dědičnost	2	0	0	0	0
Generičnost	27	7	7	0	7
Třídy	0	0	0	0	0
Rozhraní	8	8	5	1	5
Celkem	61	38	32	1	31

V těchto statistikách můžeme vidět, že v oblasti testů pro rozhraní byl nejlepší nástroj JaCC.

Když shrneme naměřené výsledky, dojdeme k závěru, že nejkvalitnějším z porovnávaných nástrojů je z hlediska testovacích dat nástroj JAPICC. Vzhledem k výše zmíněné korekci dat je tento výsledek porovnatelný s ostatními nástroji i přes odlišný způsob přístupu k testování dat. Jen těsně za ním skončil nástroj JaCC, který lze tedy označit za nejlepší nástroj z pohledu plnohodnotného testování kompozice programu. Byl také jednoznačně nejlepší v oblasti generik, z čehož také vyplynula již zmíněná nejvyšší úspěšnost nástroje JaCC v oblasti rozhraní. Nástroj Animal Sniffer do jisté míry trpěl svým omezením pouze na binární kompatibilitu, nicméně i z pohledu binární kompatibility měl horší výsledky než nástroje JAPICC a JaCC.

6.2 Mimofunkční charakteristiky

Nyní, když máme dokončené porovnání nástrojů v nejdůležitější oblasti, a sice v odhalování nekompatibilních změn, se zaměříme také na mimofunkční charakteristiky, které jsou definovány v sekci 3.6.

6.2.1 Snadnost použití

První definovanou mimofunkční charakteristikou byla snadnost použití. V této práci je hodnoceno použití mimo vývojový projekt. Uvažujeme tedy situaci, kdy máme k dispozici již sestavenou klientskou aplikaci a verzi knihovny. Tyto přeložené soubory použijeme jako vstup nástrojů. Uvažujeme náročnost před prvním spuštěním, protože po přípravě skriptů, či jiných variant spuštění, může již následující používání být mnohem snažší.

Prvním nástrojem je jdeps, který je distribuován společně s Javou od verze JDK 8. Jelikož se jedná o nástroj spustitelný z příkazové řádky, lze jej snadno spustit zadáním jednoho příkazu do příkazové řádky. Stačí pouze nastavit podrobnost výstupu, případně zadat požadavek na generování výsledku do souboru a samozřejmě vstupní soubory, které se mají zkontrolovat. Tyto soubory mohou mít různé formáty, jednotlivé možnosti jsou uvedeny v sekci 3.1.

Druhým nástrojem je Animal Sniffer. Ten již možnost spuštění z příkazové řádky nemá, takže pokud jej chceme využít samostatně mimo náš projekt, je nejprve nutná příprava například Ant skriptu, s jehož pomocí si nástroj ze vstupních dat vygeneruje signaturu API, kterou následně zkontroluje vůči zadanému klientovi. Vstupní data opět mohou mít více možností zadání, konkrétně jsou vypsána v sekci 3.2. Vzhledem k nutnosti přípravy skriptu pro spuštění nástroje je první použití nástroje mnohem zdlouhavější než u nástrojů s podporou spuštění z příkazové řádky.

Dalším nástrojem je JAPICC. Tento nástroj opět podporuje spuštění z příkazové řádky, díky čemuž je jeho použití rychlejší než například u předchozího nástroje. Lze jej tedy také snadno spustit pomocí jediného příkazu, ve kterém stačí zadat starou verzi knihovny, novou verzi knihovny a klienta. Vstupní data musí být JAR soubory, ale je umožněno zadání většího počtu souborů, které pak je nutno popsat XML souborem s definovanou strukturou.

Posledním nástrojem je JaCC. Tento nástroj není strukturovaný pro samostatné použití mimo vývojový projekt, takže nemá podporu spuštění z příkazové řádky. Pro potřeby této práce bylo tedy zapotřebí vytvořit jednoduchého klienta, který zajistí spuštění nástroje se zadanými soubory. Tento klient je skutečně jednoduchý a můžeme říci, že jeho příprava je srovnatelně zdlouhavá, možná dokonce rychlejší než příprava skriptu pro nástroj Animal Sniffer. Oproti nástrojům spustitelným z příkazové řádky je však pochopitelně příprava stále pracnější.

Zhodnocení této charakteristiky je celkem jednoznačné, protože nástroje spustitelné z příkazové řádky jsou pochopitelně mnohem snažší na použití. Z tohoto důvodu jsou tedy nejsnáze použitelné nástroje jdeps a JAPICC. Za tyto nástroje lze zařadit nástroj JaCC společně s nástrojem Animal Sniffer.

6.2.2 Integrovatelnost

Často však můžeme také od nástroje požadovat možnost jeho integrování do našeho již existujícího projektu. Do jisté míry nám tyto možnosti popisuje již tabulka 3.1.

Z tabulky vidíme, že nástroje jdeps ani JAPICC nepodporují žádnou možnost integrace do vývojového cyklu, protože jejich jedinou možností spuštění je příkazová řádka. Implementaci do Maven pluginu umožňují nástroje Animal Sniffer a JaCC. Nástroj Animal Sniffer navíc umožňuje také implementaci do Ant skriptu. Naopak nástroj JaCC umožňuje implementaci přímo do Eclipse IDE.

Vzhledem ke zmíněným možnostem je nejlépe integrovatelným nástrojem Animal Sniffer, který umožňuje integraci jak do Maven projektu, tak do Ant skriptu. Nástroj JaCC sice umožňuje také integraci do Eclipse IDE, ale tím je uživatel limitován pouze na toto vývojové prostředí. Naopak nástroje jdeps a JAPICC jsou z hlediska integrovatelnosti do vývojového cyklu nepoužitelné.

6.2.3 Struktura výstupu

Nyní se již zaměříme na výstupy jednotlivých nástrojů. Každý nástroj má pochopitelně jinak strukturované výstupy a jinak podrobné informace o nalezených chybách.

Jako první si opět přiblížíme nástroj jdeps. Tento nástroj pracuje pouze na úrovni tříd, což je na první pohled zřejmé také z formátu výstupních dat nástroje. Uveďme si ukázku výstupu tohoto nástroje:

```
classConstructorParamTypeWiden (main.jar)
   -> java.lang
   -> lib.classConstructorParamTypeWiden       lib-2.0.jar
classDel (main.jar)
   -> java.lang
   -> lib.classDel                             lib-2.0.jar
   -> lib.classDel                             not found
```

Výstup je jasně strukturovaný a přehledný. Výsledky jsou odděleny pro každý balík a jelikož jsou řešeny jen vazby na úrovni tříd, z hlediska podrobností je pouze vidět, která třída má vazbu na kterou knihovnu a v případě chyby je jen napsána hláška, že vazba nebyla nalezena.

Dalším nástrojem je Animal Sniffer, který již má mnohem větší schopnost odhalovat nekompatibilní změny v aplikacích. Uveďme si tedy také ukázku jeho výstupu:

```
[as:check-signature] /home/michal/pokusy/final/Bratner_DIP/scripts/
diplomka/main.jar:classFieldTypeUnbox/Main.class:
Undefined reference: Integer lib.classFieldTypeUnbox.Foo.foo

[as:check-signature] /home/michal/pokusy/final/Bratner_DIP/scripts/
diplomka/main.jar:classFieldTypeWiden/Main.class:
Undefined reference: int lib.classFieldTypeWiden.Foo.foo
```

Záznam každého nalezeného problému začíná výrazem `[as:check-signature]`, což velmi zlepšuje přehlednost výpisu. Každý záznam také uvádí nejen třídu včetně jejího umístění, kde k problému došlo, ale také konkrétní prvek, který

problém způsobil. Žádné další informace však nástroj neposkytuje, takže příčinu problému již musí uživatel hledat sám.

Dalším nástrojem je JAPICC, který jediný z testovaných nástrojů generuje svůj výstup do HTML souboru. Pro tento nástroj si tedy na obrázku 6.1 uvedeme ukázku vzhledu vygenerovaného HTML souboru.

Removed Methods (78)

lib-1.0.jar, Boo.class
package lib.classDel
Boo.Boo ( )

lib-1.0.jar, Foo.class
package lib.abstractClassAccessDelModifier
Foo.Foo ( )
package lib.abstractClassMethodDel
Foo.moo ( ) [abstract] : void
package lib.classAccessDelModifier
Foo.Foo ( )
package lib.classChangeToInterface
Foo.Foo ( )
package lib.classConstructorAccessDelModifier
Foo.Foo (int *p1*)

Obrázek 6.1: Vzhled výpisu nástroje JAPICC

Tento nástroj má ve výstupu oddělené stránky pro binární a zdrojovou kompatibilitu. Každá stránka je následně ještě dělena do jednotlivých sekcí jako jsou přidaná metoda, odebraná metoda, problémy s datovými typy a další, kompletní výpis můžeme najít na stránce [6]. Takové rozdělení sice na první pohled může vypadat poměrně přehledně, ale ve skutečnosti to tak příliš neplatí. Zařazení problému do některé ze skupin nelze brát jako směrodatné pro identifikování problému. Většina nalezených problémů je totiž bez bližšího popisu a občas zvláštně zařazena. Příkladem může být test na odstranění abstraktní metody, který byl vyhodnocen jako problémový z hlediska datových typů.

Posledním nástrojem je JaCC, jehož výstup se dá označit jako nejlepší. Uveďme si opět příklad:

```
Class: lib.classFieldTypeWiden.Foo (Classification: C2)
../diplomka/lib-2.0.jar
 Field: foo (Classification: F2)
(imported by: classFieldTypeWiden.Main, main.jar)
  Type: int x long (Specialised) (Classification: F5)
```

```
Class Inherited: inheritanceHiddenMethodDelStatic.Main --|>
lib.inheritanceHiddenMethodDelStatic.Foo
../diplomka/main.jar --|> ../diplomka/lib-2.0.jar
 Method: foo (Classification: M2)
  Modifier: public static x public (Not found)
  Return type: void
```

Každý záznam o nalezeném problému obsahuje třídu klienta a knihovny, mezi kterými došlo k problému. Dále můžeme vidět konkrétní prvek programu, který problém způsobil a také popis nalezeného problému. Takto podrobný popis neposkytuje žádný jiný testovaný nástroj.

Z tohoto důvodu je také nástroj JaCC určen jako jednoznačně nejlepší nástroj z hlediska struktury a podrobnosti výpisu nalezených problémů. Následuje nástroj JAPICC, který má sice rozdělení chyb na základní skupiny, ale občas se zvláštním dělení. Poté nástroje jdeps a Animal Sniffer, které pouze oznamují nenalezené reference.

6.2.4 Formát výstupu

Kromě struktury výstupu se také podívejme na možnosti formátů výstupu jednotlivých nástrojů. Nejlepší způsob srovnání nám v tomto případě poskytne tabulka.

Tabulka 6.21: Formáty výstupů jednotlivých nástrojů

Formát výstupu	jdeps	Animal Sniffer	JAPICC	JaCC
CLI	✓	✓		
String				✓
HTML			✓	
DOT	✓			

Nástroj jdeps kromě standardního výstupu do konzole umožňuje také výstup do vlastního formátu DOT. Rozdíl je však pouze ve struktuře, obsah dat je totožný s výstupem do konzole. Nástroj Animal Sniffer vypisuje své výsledky do konzole, přičemž nalezené chyby jsou v chybovém kanálu. Nástroj JAPICC zase poskytuje pouze výpis výsledků do HTML souboru. Nejuniverzálnější způsob však poskytuje nástroj JaCC, který svůj výstup směřuje v programu do textového řetězce. To programátorovi dává možnost naložit

s výstupem dle vlastního uvážení. Může jej jednoduše vypsat do konzole, či do souboru, nebo se může rozhodnout výstup parsovat a dále zpracovávat.

Z důvodu zmíněné flexibility je z hlediska formátů výstupu jako nejlepší hodnocen nástroj JaCC, který dává uživateli největší volnost v ovlivnění výsledného formátu dat. Následuje nástroj jdeps, který kromě konzole podporuje také vlastní formát souboru. Nástroje JAPICC a Animal Sniffer se dají považovat za rovnocenné.

6.2.5 Licenční politika

Po charaketristikách souvisejících s funkcionalitou nástrojů se také podíváme na charakteriky spojené s jejich distribucí. První takovou charakteristikou je licenční politika, která nám určuje možnost legálního využití nástroje pro naše potřeby.

Nástroj jdeps, který je distribuován s jazykem Java od společnosti Oracle, je distibuován se stejnou licencí jako samotný jazyk Java, tedy Oracle Binary Code License. Ta nám sice nedovoluje upravovat kód nástroje, ani jeho další distribuci, ale umožňuje jeho bezplatné používání. Z hlediska našich potřeb pro kontrolu aplikací je tedy tento nástroj volně dostupný.

Dalším nástrojem je Animal Sniffer, který je distribuován pod MIT licencí, která nám umožňuje nejen bezplatné používání, ale dovoluje také úpravy nástroje a jeho následnou redistribuci s podmínkou, že k nové distribuci budou opět přiloženy licenční podmínky zmíněné u oficiální verze nástroje.

Nástroj JAPICC je distribuován jako volný software, který lze bezplatně používat, modifikovat a opětovně distibuovat pod licencí GNU GPL nebo LGPL.

Nástroj JaCC spadá pod licenci Západočeské univerzity a lze vyjednat jeho bezplatné použití.

Ve výsledku jsou tedy všechny nástroje distribuovány s licencí, která umožňuje jeho bezplatné používání. Licence nástroje jdeps sice nepovoluje jeho úpravy, ale pro tuto práci je relevantní pouze možnost bezplatného používání, takže to není problém.

6.2.6 Aktuálnost

Další poměrně důležitou charakteristikou je aktuálnost nástrojů. Pokud je nástroj aktivně vyvíjen, pak je mnohem větší pravděpodobnost, že dokáže detekovat chyby v nových konstrukcích jazyka a také má větší potenciální využití do budoucna.

Jelikož je nástroj jdeps vyvíjen a distribuován přímo s jazykem Java, můžeme jej v době sepsání této práce bez problémů označit za aktuální. Navíc je tento nástroj orientován pouze na závislosti na úrovni tříd, takže jej neovlivní většina možných změn jazyka.

Nástroj Animal Sniffer má dle stránky [5] jako termín poslední verze uveden únor 2016. I tento nástroj tedy můžeme označit jako aktuální vůču verzi jazyka Java dostupné v době sepsání této práce.

Ještě novější verzi má však nástroj JAPICC, jelikož poslední změna v úložišti, ze kterého je možné jej stáhnout, proběhla v dubnu 2016. Z dostupných informací o změnách repozitáře můžeme usoudit, že nástroj je udržován a průběžně aktualizován.

Nástroj JaCC je stále aktivně vyvíjen a v době tvorby této práce můžeme datovat poslední změnu na duben 2016. Lze jej tedy také snadno označit za aktuální.

O všech nástrojích tedy v době sepsání této práce můžeme prohlásit, že jsou aktuální a z tohoto pohledu by neměl být problém s novými konstrukcemi jazyka, které by nástroje nemusely znát. Jelikož některé nástroje jsou aktualizovány častěji než jiné, uveďme si přehled verzí nástrojů, které byly využity v této práci.

- jdeps distribuovaný s verzí Java 8u66
- Animal Sniffer v1.15
- JAPICC v1.4.3
- JaCC v1.0.9

6.3 Výsledné zhodnocení

Nyní již máme nástroje zhodnocené z pohledu všech sledovaných charakteristik. Shrňme si tedy výsledky dosažené v jednotlivých sledovaných charakteristikách. V tabulkách 6.23 a 6.25 je uvedeno pořadí, ve kterém se v jednotlivých kategoriích nástroje umístily. U kategorie odhalených problémů jsou v závorce uvedeny počty odhalených problémů v testovacích datech.

Pořadí současně také slouží jako body pro hodnocení nástrojů. Celkové skóre, uvedené v posledních řádcích tabulek6.23 a 6.25, je výsledkem součtu bodů ze všech kategorií. Pro zohlednění důležitosti jednotlivých kategorií jsou nejdříve body z každé kategorie přenásobeny koeficientem důležitosti dané kategorie. Jelikož můžeme požadovat různé využití nástrojů, uvedeme si hodnocení dvou různých pohledů. Koeficienty pro jednotlivé pohledy jsou uvedeny v tabulkách 6.22 a 6.24, konkrétní sada koeficientů závisí na daném pohledu na nástroje. Pro výsledné skóre pak platí, že menší hodnota je lepší.

V těchto pohledech nejsou uvažovány charakteristiky licenční politiky a aktuálnosti nástrojů, protože nástroje z těchto pohledů můžeme označit za prakticky totožné. Z toho důvodu si můžeme dovolit tyto výsledky zanedbat bez vlivu na výsledné hodnocení.

První pohled uvažuje nástroje v situaci, kdy je chceme využít nezávisle na našem vývojovém cyklu. Z toho důvodu je pro tento pohled vynechána kategorie integrovatelnosti, protože je v tomto případě zbytečná.

Tabulka 6.22: Koeficienty důležitosti kategorií v prvním pohledu

Sledovaná charakteristika	Koeficient
Odhalení problémů	0.5
Snadnost použití	0.15
Struktura výstupu	0.25
Formát výstupu	0.1

S ohledem na celkové skóre uvedené v tabulce 6.23 můžeme vyvodit závěr, že podle sledovaných charakteristik je nejlepším nástrojem pro využití nezávisle na našem vývojovém cyklu nástroj JAPICC před nástrojem JaCC. Nástroj Animal Sniffer je až nečekaně pozadu a nástroj jdeps je podle očekávání nejhorším nástrojem.

Tabulka 6.23: Shrnutí výsledků jednotlivých nástrojův prvním pohledu

Sledovaná charakteristika	jdeps	Animal Sniffer	JAPICC	JaCC
Odhalení problémů	4 (2)	3 (90)	1 (134)	2 (115)
Snadnost použití	1	3	1	3
Struktura výstupu	3	3	2	1
Formát výstupu	2	3	3	1
Skóre	3.1	3	1.45	1.8

Druhý pohled budeme uvažovat jako situaci, kdy hledáme nástroj pro testování v rámci našeho vývojového cyklu. Zde tedy naopak kategorii integrovatelnost započteme a nebudeme uvažovat kategorii snadnosti použití, protože ta hodnotí nástorje z pohledu využití mimo vývojový cyklus.

Tabulka 6.24: Koeficienty důležitosti kategorií v druhém pohledu

Sledovaná charakteristika	Koeficient
Odhalení problémů	0.5
Integrovatelnost	0.15
Struktura výstupu	0.25
Formát výstupu	0.1

Tabulka 6.25: Shrnutí výsledků jednotlivých nástrojů v druhém pohledu

Sledovaná charakteristika	jdeps	Animal Sniffer	JAPICC	JaCC
Odhalení problémů	4 (2)	3 (90)	1 (134)	2 (115)
Integrovatelnost	3	1	3	2
Struktura výstupu	3	3	2	1
Formát výstupu	2	3	3	1
Skóre	3.4	2.7	1.75	1.65

Z pohledu nástrojů umožňujících integraci do vývojového cyklu je lepší nástroj JaCC, který oproti nástroji JAPICC podporuje integraci. Nástroj JAPICC sice zůstává na druhém místě, ale pro jeho použití by byl nutný například skript, který by současně s jinou aplikací daného vývojového cyklu spouštěl také tento nástroj. Pokud bychom k nástroji JaCC hledali náhradu přímo integrovatelnou do vývojového cyklu, museli bychom z testovaných nástrojů zvolit nástroj Animal Sniffer.

7 Závěr

V rámci této práce byla vytvořena sada testovacích dat simulujících změny popsané v kapitole 4. Následně byla tato data použita k otestování nástrojů ověřujících kompozici modulárních Java aplikací. Jak se v průběhu práce ukázalo, existuje řada nástrojů, které dokáží ověřovat zpětnou kompatibilitu, ale nástrojů ověřujících kompozici programů není mnoho.

Otestováním nalezených nástrojů jsme dosáhli výsledku, kdy podle očekávání byly nejsilnější nástroje JAPICC a JaCC.Naopak nástroj Animal Sniffer nenaplnil očekávání a oproti prvním nástrojům byl znatelně slabší. Nástroj jdeps byl podle očekávání nejslabší, protože dokáže ověřovat kompozici pouze na úrovni tříd. Nástroj JAPICC byl nejlepší ve schopnosti odhalování problémů s kompatibilitou, ale není použitelný, pokud potřebujeme nástroj implementovat do vývojového cyklu. V takovém případě je nejlepší volbou nástroj JaCC.

Při hledání nástrojů, které by byly schopné testovat kompozici Java aplikací, jsem narazil také na nástroj joops. Jeho dostupnou verzi se mi však nepodařilo spustit, což je důvod, proč není v práci zmíněn a nejsou uvedeny ani jeho výsledky na testovacích datech.

Do budoucna by bylo možné připravenou sadu testovacích dat ještě dále rozšířit například o sekci anotací. Rozšíření dat by také bylo možné provést o testy, které by procházely stejné změny, ale používaly by jinou koncepci využití knihovny na straně klienta, což by vedlo k testům, které by mohly mít jiné vlastnosti z hlediska kompatibility.

Přehled zkratek

JAR - Java Archive

API - Application Programming Interface

JDK - Java Development Kit

jdeps - Java Dependency Analysis Tool

JaCC - Java Compatibility Checker

JAPICC - Java API Compliance Checker

XML - Extensible Markup Language

HTML - HyperText Markup Language

CSV - Comma-separated values

IDE - Integrated Development Environment

MIT - Massachusetts Institute of Technology

GNU GPL - GNU General Public License

GNU LGPL - GNU Lesser General Public License

Literatura

[1] *Kinds of Compatibility: Source, Binary, and Behavioral* [online]. 2008 [cit. 4.2016]. https://blogs.oracle.com/darcy/entry/kinds_of_compatibility

[2] *JLS, Chapter 13. Binary Compatibility* [online]. 2015 [cit. 4.2016]. http://docs.oracle.com/javase/specs/jls/se8/html/jls-13.html#jls-13.4.12

[3] *Java Dependency Analysis Tool* [online]. 2016 [cit. 4.2016]. https://wiki.openjdk.java.net/display/JDK8/Java+Dependency+Analysis+Tool

[4] *jdeps manual page* [online]. 2016 [cit. 4.2016]. https://docs.oracle.com/javase/8/docs/technotes/tools/unix/jdeps.html

[5] *Animal Sniffer* [online]. 2016 [cit. 4.2016]. http://www.mojohaus.org/animal-sniffer/

[6] *Java API Compliance Checker* [online]. 2016 [cit. 4.2016]. http://ispras.linuxbase.org/index.php/Java_API_Compliance_Checker

[7] JEZEK, K., HOLY, L., DANEK, J. *Preventing Composition Problems in Modular Java Applications* Plzeň, 2015.

[8] *Java tutorials - Controlling Access to Members of a Class* [online]. 2015 [cit. 3.2016]. https://docs.oracle.com/javase/tutorial/java/javaOO/accesscontrol.html

[9] *Java Non Access Modifiers* [online]. 2016 [cit. 3.2016]. http://www.tutorialspoint.com/java/java_nonaccess_modifiers.htm

[10] *Lesson: Generics (Updated)* [online]. 2015 [cit. 3.2016]. https://docs.oracle.com/javase/tutorial/java/generics/index.html

[11] *Kinds of Conversion* [online]. 2015 [cit. 5.2016]. http://docs.oracle.com/javase/specs/jls/se8/html/jls-5.html#jls-5.1

[12] JEZEK, K., BRADA, P., DANEK, J. *Understanding the Types of Software Compatibility and their Impact in Java* (nepublikováno) Plzeň, 2015.

A Příloha

A.1 Uživatelská příručka

Pro možnost opakování testů jsou připraveny skripty pro operační systém Linux. Nejdříve je ale nutné nainstalovat nástroj JAPICC. Odkaz na jeho stažení i návod k instalaci jsou dostupné na adrese [6], případně jsou jeho zdrojové soubory přiloženy na CD. Pro funkci nástroje jdeps je nutné mít nainstalovánu Javu ve verzi minimálně JDK 8. Zdrojové soubory pro nástroje Animal Sniffer a klient pro spuštění testů na nástroji JaCC jsou přiloženy se skripty.

V adresáři `scripts` jsou připraveny tři podadresáře. V adresáři `tests` je připraven Ant skript, který zajistí přeložení testů a vygenerování JAR souborů. Tento Ant skript stačí spustit bez jakýchkoliv parametrů. V adresáři `compatibility` je připraven shellový skript `compatibility.sh`, který pomocí přiloženého Ant skriptu zajistí ověření kompatibility jednotlivých testů. Výsledky jsou pomocí jednoduchého klienta, který je také přiložen, vygenerovány do CSV soubor s přehledem zdrojové a binární kompatibility jednotlivých testů. Generování spustíme jednoduchým příkazem

./compatibility.sh

V adresáři `tools` jsou všechny potřebné soubory a skripty pro spuštění jednotlivých nástrojů a také shellový skript `tools.sh`, který zajistí přeložení zdrojových souborů testů a následné spuštění jednotlivých nástrojů s vygenerovanými soubory. Výstupy nástrojů jsou přesměrovány do souborů. Skript opět spustíme jednoduchým příkazem

./tools.sh