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

Diplomová práce

Porovnání nástrojů ověřujících zpětnou kompatibilitu Java knihoven

Plzeň, 2016 Bc. Rudolf Augusta

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

V Plzni dne 12. 5. 2016

Bc. Rudolf Augusta

Abstrakt

Porovnání nástrojů ověřujících zpětnou kompatibilitu Java knihoven

Na katedře informatiky a výpočetní techniky Západočeské univerzity v Plzni je vyvíjen softwarový nástroj Java Compatibility Checker (JaCC) na kontrolu (testovaní) zpětné kompatibility softwaru. Nicméně samotný nástroj je nutné také otestovat, zda dovede nalézt změny provedené v daném softwaru. Testovací data k tomuto účelu však nejsou dostupná.

Cílem této práce je tedy vytvořit rozsáhlou sadu testovacích dat simulující inkrementální vývoj s nejrůznějšími změnami v knihovně porušující i zachovávající zpětnou kompatibilitu, neboť je důležité, aby nástroj odhalil porušení, ale zároveň neoznačil zachovávající změnu. Dalším bodem práce je nalézt nástroje na kontrolu zpětné kompatibility a porovnat je na základě testování za použití vytvořených dat a v této práci definovaných mimofunkčních charakteristik.

Abstract

The comparison of tools checking backward compatibility of Java libraries

In the Department of Computer Science and Engineering of University of West Bohemia in Pilsen there is being developed a software tool for checking of backward compatibility called Java Compatibility Checker (JaCC). This tool needs to be tested if it can find incompatible changes in a library successfully. Unfortunately data for this kind of testing does not exist.

The goal of this thesis is creation of an extensive set of testing data simulating incremental software development with many different changes in a library either breaking or keeping backward compatibility. It is important for a tool to discover broken compatibility but to not report compatible change. The next part of the thesis is the analysis of tools checking backward compatibility and compare them based on test results using prepared testing data and other non-functional criteria.

Obsah

1 Úvod

V životním cyklu softwaru dochází k jeho změnám, ať už ve fázi vývoje či údržby. Každý projekt začíná několika třídami pokrývajícími základní funkcionalitu a postupem času se rozrůstá o další, upravují se stávající nebo odstraňují, pokud jsou nahrazeny jinými či již nejsou nadále potřeba. Stejně je tomu tak i s Java knihovnami.

S každou změnou zde existuje riziko nežádoucích závad ve formě chyb, nekompatibilit či změny chování, které mohou a často mají pro klienta dané knihovny závažné důsledky. V nejzávažnějších případech dochází k finančním ztrátám či ohrožení na životech (např. řízení dopravy či letového provozu). Naštěstí k odhalení i takovýchto neduhů existují nástroje pro testování na různých úrovních - jednotkové, komponentní, integrační, systémové, výkonnostní, zátěžové či testování kompatibility. Je zodpovědností autorů daného softwaru, aby pečlivě otestovali své dílo, nicméně i přes velkou škálu nástrojů je v této moderní době ještě mnoho softwaru, kde toto chybí.

Samotné nástroje používané k testování je také nutné podrobit zkouškám, zda dovedou správně identifikovat a odzkoušet scénáře týkající se dané úrovně, pro kterou jsou vyvinuty. Je tedy nutné vytvořit vhodná testovací data, případy, scénáře, aby se zjistilo, zda daný nástroj dovede otestovat vše, co se od něho očekává. Takováto data pro testování nástrojů na kontrolu zpětné kompatibility nejsou však nikde dostupná.

Předmětem této práce je tedy vytvořit rozsáhlou sadu testovacích dat simulující inkrementální vývoj s nejrůznějšími změnami v knihovně porušující i zachovávající zpětnou kompatibilitu, neboť je důležité, aby nástroje odhalili porušující, ale zároveň neoznačili zachovávající změnu. Dále se tato práce zabývá samotným vyhodnocením úspěšnosti a efektivity nástroje Java Compatibility Checker (JaCC) vyvíjeného na katedře informatiky na Západočeské univerzitě v Plzni a dostupných konkurenčních nástrojů v odhalení změn při testovaní nad vytvořenými daty. Zpětné kompatibilita je testována v jazyce Java na úrovni zdrojového kódu (*Application Programming Interface*, zkr. *API*) [8] a binárních modulů (*Application Binary Interface*, zkr. *ABI*) [8].

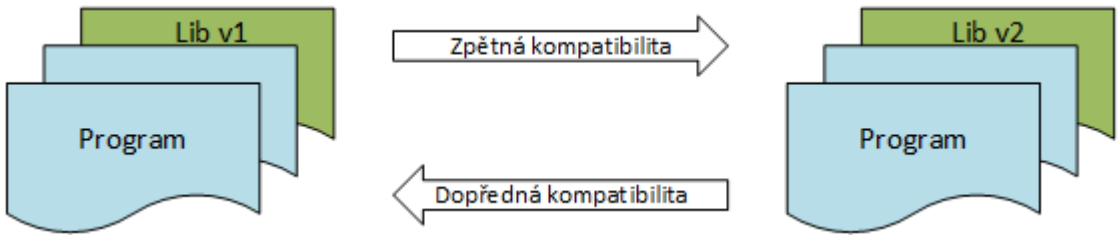
2 Kompatibilita

Pro slovo kompatibilita je ve slovníku cizích slov tato definice: vzájemná slučitelnost, snášenlivost, schopnost volného křížení, spojitelnost. Jinými slovy, pokud jsou dvě jednotky kompatibilní, znamená to, že jsou slučitelné či spojitelné. Rozlišuje se mnoho druhů kompatibility, např. mechanická (díly automobilu), biologická (buňky organismu), elektrická (přístroje a el. soustava), hardwarová nebo softwarová. Dále v textu se uvažuje pouze softwarová kompatibilita.

V informatice lze říci, že kompatibilita je schopnost dat, programů, softwaru a dalšího vybavení (hardware) vzájemně spolupracovat. To umožňuje spojení individuálních součástek (částí) do jednoho systému. Tato práce se zaobírá zvláštním druhem kompatibility, které se říká zpětná. Práce se omezuje na zpětnou kompatibilitu Java knihoven a nástrojů její analýzy.

Každý druh kompatibility má pozitivní a negativní stránku věci. Jinými slovy program, který fungoval, musí fungovat i nadále (pozitivní), program, který nefungoval, nesmí fungovat (negativní). Například mějme metodu, která provádí výpočet pouze s kladnými čísly (pozitivní), ale pro záporná vyhazuje výjimku (negativní).

Zpětná kompatibilita



Obr. 1 - Kompatibilita

Zpětná kompatibilita je speciální druh kompatibility, který je žádaný jak od softwaru tak i hardwaru. Ve své podstatě jde o schopnost zaměnitelnosti.

V rámci softwaru je knihovna zpětně kompatibilní, pokud může nahradit svojí starší verzi. Nová verze knihovny musí tedy být schopná spolupracovat s programem, ve kterém byla dříve používaná její starší verze, viz Obr. 1. Pokud by při vývoji knihovny nebyl tento druh kompatibility dodržen, následky zmíněné inovace verzí mohou být fatální. Je poté možné, že program nebude možné spustit z důvodu různých typů chyb.

Jednoduše řečeno zpětná kompatibilita je důležitá, protože nám umožňuje začít používat novou verzi knihovny, aniž bychom museli zasahovat do našeho programu. Tato vlastnost má spoustu výhod. Např. je možné velmi jednoduše provést vylepšení či opravu (*patch*) v nové verzi knihovny bez nutnosti zásahu do klientského kódu.

Abychom mohli říci, že knihovna je zpětně kompatibilní, nesmí porušit ani jednu z následující tři typů kompatibilit: *zdrojovou, binární a behaviorální*.

Mějme tedy program P a knihovnu L, respektive dvě verze této knihovny L_1 a L_2, které se různými způsoby liší [3].

Zdrojová kompatibilita

Dvě verze téže knihovny L_1 a L_2 jsou zdrojově kompatibilní s programem P, pokud program přeložený se starší verzí knihovny L_1 lze přeložit také s verzí novou L_2 bez nutnosti zásahu do zdrojového kódu programu. V platformě Java toto podléhá Java překladači (*Java compiler*), který bezchybně přeloží program P i s novou verzí knihovny L_2. V případě porušení zdrojové kompatibility není schopný daný kód přeložit a hlásí chyby ve zdrojovém kódu programu P.

Nicméně takovýto test nezaručuje zdrojovou kompatibilitu se všemi existujícími programy, používajícími danou knihovnu L. Zdrojová kompatibilita se totiž také týká mapování zdrojového kódu na přeložené soubory a při definování importů je možné použít tzv. *wild card* (“`import balík.*`”). Dejme tomu, že v knihovně L_1 bude balík *bar a* balík *foo* obsahující třídu *Boo*. V knihovně L_2 bude do balíku *bar* také přidána třída *Boo*. Následující příklad je poté zdrojově nekompatibilní [3].

```
import foo.*;
import bar.*;

public class HelloBoo {
    public static void main(String... args) {
    Object o = new Boo();
    System.out.println("Hello " + o.toString());
    }
}
```

Zdrojová kompatibilita může být také ovlivněna změnou v programovacím jazyce. Například některé z používaných slov se může stát klíčovým slovem daného jazyka, jako tomu bylo se slovy `strictfp`, `assert` a `enum`.

Nastanou-li problémy s nekompatibilitami, pokud lze hodnotit jednotlivé druhy podle závažnosti, dá se říci, že problémy se zdrojovou kompatibilitou jsou nejméně závažné, neboť se jedná o chyby v době překladu (*compile-time*), na které upozorní překladač a tudíž daný program *P* nelze přeložit ani spustit. K odhalení tohoto druhu nekompatibility existují i další nástroje.

Binární kompatibilita

Binární kompatibilita je velmi přesně definovaná přímo ve specifikaci jazyka Java (*Java Language Specification*) [2] nebo se jí věnuje celý článek na Eclipse wiki [1].

Dvě verze téže knihovny L_1 a L_2 jsou binárně kompatibilní s programem *P*, pokud program slinkovaný se starší verzí knihovny L_1 lze slinkovat také s verzí novou L_2. V platformě Java toto podléhá ClassLoaderu provádějícímu dynamické linkovaní, který bezchybně slinkuje program *P* i s novou verzí L_2 bez nutnosti překladu programu *P*. V případě porušení binární kompatibility se chyba projeví až při běhu programu *P* vyhozením výjimky linkeru, neboť Java používá dynamické linkování [6].

Problémy s binární nekompatibilitou jsou závažnější. Pokud je uvažována platforma Java, program vyhodí výjimku až za běhu (*run-time*), což může vést k závažným problémům, nicméně snadno identifikovatelným. Nástroje odhalující zdrojovou nekompatibilitu zpravidla odhalují i binární nekompatibilitu, tudíž k odhalení i tohoto druhu nekompatibility také existují nástroje.

Behaviorální kompatibilita

Tento druh kompatibility znamená, že program *P* by pro stejný vstup měl provádět operace přesně stejné nebo ekvivalentní a stále se stejným výsledkem nezávazně na verzi knihovny, kterou používá. Jinými slovy by program měl pro stejný vstup mít stále stejný výstup. Základní vlastností tohoto druhu kompatibility by mělo být spjatost se specifikací, co by daný program měl provádět.

Nejzávažnější problémy však nastávají s behaviorální nekompatibilitou, neboť může dojít k velmi závažným problémům, které jsou navíc těžko identifikovatelné. Speciální nástroje k odhalení tohoto druhu nekompatibility neexistují. Jelikož už se jedná o běh samotného programu, lze tento druh nekompatibility odhalit pouze podrobným testování systému na několika úrovních (jednotkové, komponentní, integrační, systémové). Proto z praktického hlediska není možné takový nástroj vytvořit, neboť existuje nepřeberné množství softwaru, zpracovávající ještě větší počet vstupů s různými výstupy, vytvářející tak prakticky nekonečné množství testů.

3 Změny v Java programu

Životní cyklus softwarového vývoje probíhá v několika fázích lišících se podle zvoleného modelu [7]. Zjednodušeně řečeno software prochází fázemi plánování, vytváření, nasazení a údržba. V moderním pojetí tohoto procesu prochází software těmito fázemi opakovaně neboli iterativní model. V počátku je software tvořen nejzákladnější funkcionalitou a s každou iterací jsou přidávány další, což v mnoha případech znamená i změny ve stávajícím zdrojovém kódu. Životní cyklus softwarového vývoje je velmi dynamický proces, kdy dochází k neustálým změnám, ať už z důvodu změny požadavků a tím i funkcionality, změny architektury, pomocných knihoven, v programovacím jazyku či tzv. zrefaktorování zdrojového kódu. Důvodů je mnoho.

Ke změnám dochází na mnoha různých úrovních. Jednou takovou úrovní jsou datové typy, kde je možné zaměnit jeden datový typ za jiný. Tuto změnu je možné provádět například v rámci hierarchie dědičnosti. Další úrovní jsou modifikátory přístupu, kde lze zvyšovat resp. snižovat přístup daných Java elementů (tříd, rozhraní, metod, kontruktorů, proměnných). Je toho dosaženo změnou jednoho modifikátoru na druhý, popřípadě odstraněním modifikátoru. U ostatních modifikátorů se jedná o přidání nebo odstranění daného modifikátoru, kde každý z modifikátorů upravuje jinou vlastnost elementu. Nicméně tentýž modifikátor použitý u různých elementů upravuje tu samou vlastnost. Jednotlivé členy tříd či rozhraní je také možné přidávat nebo odebírat. To samé se týká i parametrů u metod a konstruktorů. V kategorii dědičnosti lze provádět změny v rámci hierarchie tříd. Příkladem může být přesun metody z potomka do rodiče. Generika jsou kapitola sama pro sebe. Jedná se o přidání nebo odebrání generického datového typu, který je možné omezit shora. Wildcards jsou speciálním typem generik a je možné je navíc omezit i zdola. Výjimky lze přidávat, odebírat nebo kombinovat.

Ve své podstatě se vždy jedná o typ změny přidávání nebo odstraňování. Ať už se jedná třeba o členy tříd (metody, konstruktory, proměnná) nebo třídy a rozhraní samotné. Například při změně datového typu formálního parametru metody z jednoho

na druhý, jde stále o ten samý typ změny. Tato změna vytváří novou metodu a má stejný efekt jako smazání staré metody, kde v hlavičce byl uveden starý datový typ, a přidání nové metody s novým typem.

Dále je tyto změny možné provést na určitých elementech Javy. Například odstranit metodu můžeme z třídy nebo z rozhraní. Změnu datového typu je možné provést v proměnné, formálním parametru metody nebo v jejím návratovém typu. Kde je danou změnu možné provést je uvedeno v tabulkách v kapitole 4 (hlavičková zelená řádka).

Programové a binární rozhraní aplikace

Programové rozhraní aplikace (*Application Programming Interface*, zkr. *API*) [8] je rozhraní na úrovni zdrojového kódu, které software poskytuje, aby ho ostatní softwary (programy) mohli používat. API lze nalézt u aplikací, knihoven nebo operačního systému. Pokud programátor používá některou z Java knihoven, pracuje s ní tedy přes její API. Jedná se o sadu popisující třídy, metody a konstanty, které jsou v knihovně přístupné programátorovi, jenž nemusí znát, jakým způsobem jsou dané elementy implementovány.

Binární rozhraní aplikace (*Application Binary Interface*, zkr. *ABI*) [8] naopak definuje nízkoúrovňové rozhraní na úrovni binárních modulů. Definuje, jakým způsobem se spolu spolupracují jednotlivé aplikace, aplikace s jádrem operačního systému nebo aplikace s knihovnou.

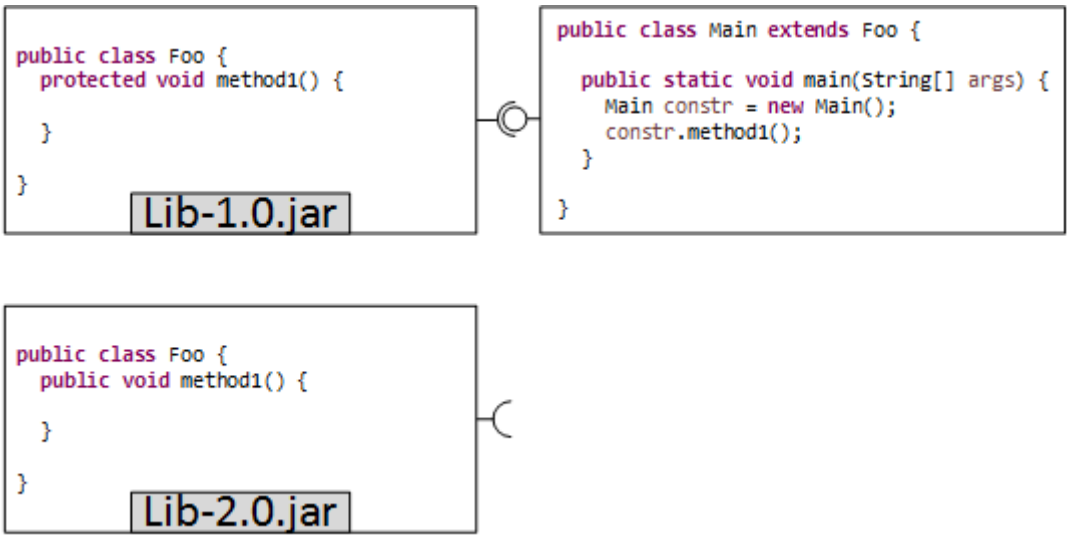
Zpětná kompatibilita Java knihovny

Zpětná kompatibilita u Java knihoven je velmi důležitá vlastnost, která umožňuje používat novější verze dané knihovny bez potřeby změn v klientském kódu. Na druhou stranu tato vlastnost může ve vývoji knihovny způsobit vcelku zásadní omezení. Pokud by existoval způsob, jak dohledat všechny klienty dané knihovny a upravit je se změnou v knihovně, celá oblast zpětné kompatibility by nebyla zapotřebí. Toto je možné pouze pro privátní knihovnu jednoho nebo několika málo projektů, které je možné takovýmto způsobem upravit. Nicméně v ostatních případech je toto nemožné. Autor knihovny musí vycházet z předpokladu, že každou dostupnou funkcionalitu v jeho knihovně využila alespoň jedna klientská aplikace, tudíž při každé změně rozhraní se musí zabývat otázkou zpětné kompatibility.

Jak již bylo zmíněno zpětná kompatibilita je užitečná vlastnost, nicméně ne vždy je její dodržení možné či praktické kvůli způsobeným omezením. Důkazem toho je samotný prostředek jazyka Java, a to anotace `@Deprecated`, která je právě použita pro dočasné zachování zpětné kompatibility, nicméně daná funkcionalita byla nahrazena novou, a to jiným způsobem, že prakticky vzniklo nové rozhraní.

Změny zdrojově i binárně kompatibilní

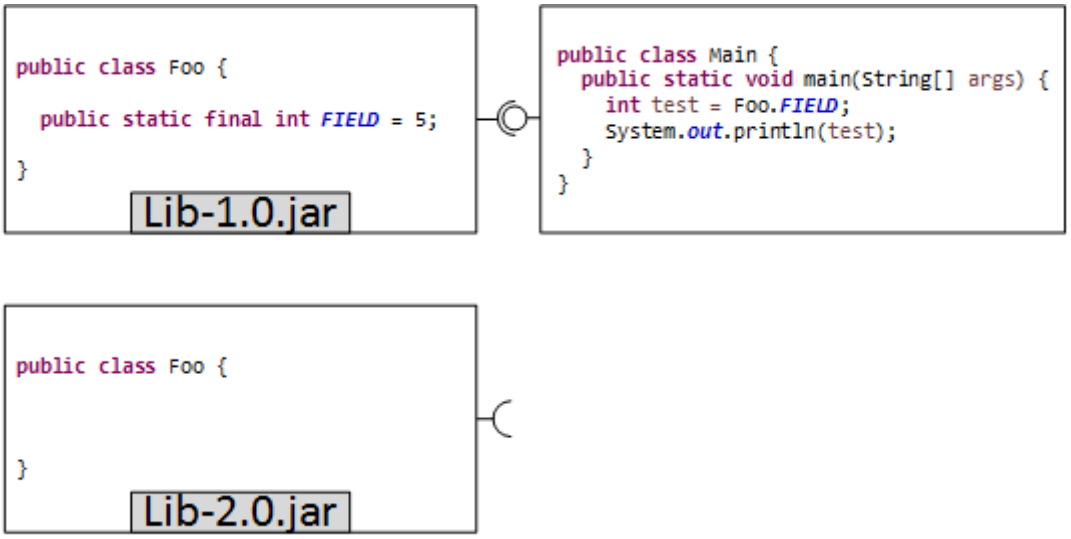
Z pohledu zdrojové a binární kompatibility je nejpříznivějším typem změny rozhraní taková, která neporušuje ani jednu z nich, neboli změna je zdrojově i binárně kompatibilní, tudíž nezpůsobuje žádné problémy klientské aplikaci. Příkladem může být zvýšení viditelnosti jakéhokoliv z Java elementů, ať už samotné třídy nebo některého z jejich členů, viz Obr. 2.



Obr. 2 - Příklad kompatibilní změny

Změny zdrojově nekompatibilní

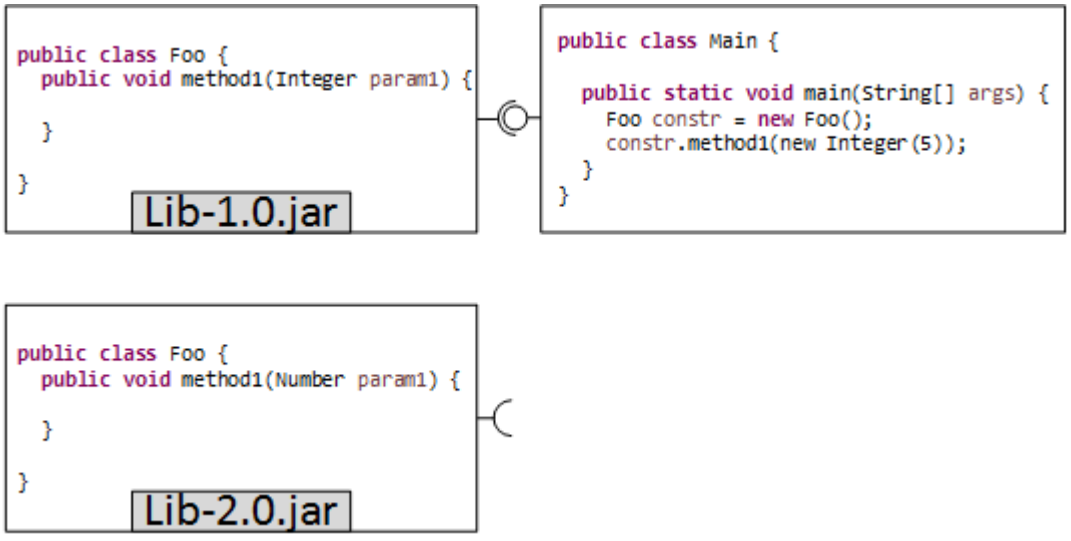
Dalším typem změny je zdrojově nekompatibilní avšak binárně kompatibilní. Příkladem může být odstranění konstanty v nové verzi knihovny, viz Obr. 3. Tato změna je binárně kompatibilní, protože byl proveden inlining (veškerý výskyt konstanty `FIELD`, byl nahrazen konkrétním číslem "5"). Změna však není zdrojově kompatibilní, protože překladač nemůže nalézt konstantu `FIELD`.



Obr. 3 - Příklad zdrojově nekompatibilní změny

Změny binárně nekompatibilní

Opačný případ k předešlému je změnu binárně nekompatibilní avšak kompatibilní zdrojově. Zde může být příkladem změna datového typu u formálního parametru metody, viz Obr. 4. Přesněji se jedná o takzvanou generalizaci datového typu, kdy datový typ `Integer` ve staré verzi knihovny nahradíme datovým typem `Number`.



Obr. 4 - Příklad binárně nekompatibilní změny

Tato změna datového typu je zdrojově kompatibilní, protože datový typ `Number` je rodičovská třída třídy `Integer`. Nicméně změna je binárně nekompatibilní.

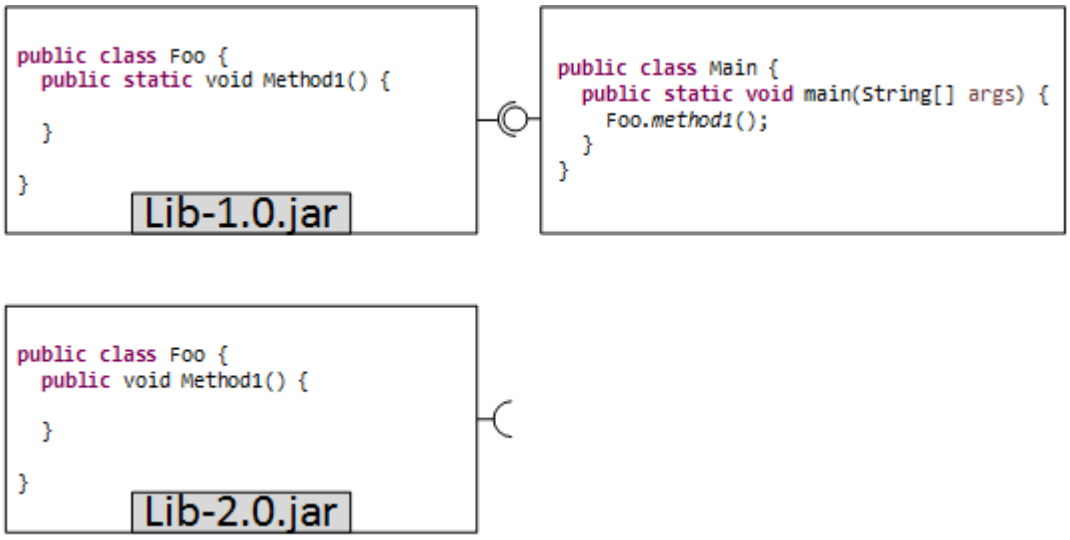
Při spuštění programu s novou verzí knihovny bez provedení opětovného překladu, se vyskytne následující chyba:

```
java.lang.NoSuchMethodError:
testing_lib.dataTypeClazzMethodParamGeneralization.Foo.method1(Ljava/l
ang/Integer;)V
```

Z této chyby je vidět, že ClassLoader při linkování hledá metodu, která má za parametr `Integer` a ne jeho rodičovskou třídu. Aby mohla být tedy použita nová verze knihovny, je nutné program znova přeložit.

Změny zdrojově i binárně nekompatibilní

Poslední možností je kombinace obou nekompatibilit, a to změna zdrojově i binárně nekompatibilní. Příkladem zde může být změna metody ze statické na nestatickou, viz Obr. 5.



Obr. 5 - Příklad zdrojově i binárně nekompatibilní změny

Dalším příkladem změny porušující oba typy kompatibility může být odstranění metody ze třídy.

4 Testovací data

Jak již bylo zmíněno v úvodu práce, k testování nástrojů je nutné vytvořit sadu dat simulující inkrementální vývoj. Aby data byla kompletní, je nutné do nich zavést jistou systematičnost, a to například následujícím způsobem. Javu rozdělíme na určité kategorie, čemuž odpovídají následující podkapitoly. Každá změna (první modrý sloupce tabulky níže) v každé kategorii může nastat pro určitý element v kódu (hlavičková zelená řádka tabulky). Celkem je tedy realizováno kolem 250 různých změn v osmi kategoriích.

Vyhodnocení jednotlivých kombinací z hlediska zdrojové a binární kompatibility je provedeno k vyhodnocení, zda nástroje daný test vyhodnotili dobře a je uvedeno v Příloze A. Je provedeno na základě Ant skriptu, který nejprve vytvoří Jar soubory připraveného klienta, starou (před změnou) a novou (po změně) verzi knihovny. Vytvořený klient vždy používá element, na kterém je změna provedena.

Ověření, zda je změna binárně kompatibilní, se provede testem slinkovaní nejprve se starou knihovnou, což by mělo proběhnout vždy bez problému, protože klient je psán pro tuto verzi knihovny. Následně se již však testuje slinkování s novou knihovnou. V případě nekompatibility bude ve výstupu konzole vypsaná chyba.

Dále se testuje zdrojová kompatibilita podobným způsobem, jen s tím rozdílem, že se jedná o přeložení klienta s různými verzemi knihovny. Pokud nastane chyba při překládání klienta s novou verzí knihovny, je změna zdrojově nekompatibilní.

Testování proběhlo ručně měněním vlastnosti (`property`) *package* a spouštěním balíku po balíku.

Termínem proměnná (*field*) se nadále v textu rozumí proměnná na úrovni třídy.

4.1 Datové typy

Změna datového typu [19] je první kategorií a jedna ze základních změn, která může při vývoji nastat. Důvody mohou být velmi různé v závislosti na provedené změně. Příkladem může být úplná výměna datového typu, neboť předešlý již není dostačující, nebo záměna z primitivního datového typu na objektový z důvodu reprezentace prázdné hodnoty (`null`), či generalizací použitím rodičovského rozhraní či třídy, aby daná metoda byla funkční i pro další podtypy. Jde tedy o změny, které je možné provádět jak ve třídě, tak většinu z nich i v rozhraní, viz Tabulka 1.

Co se týče kompatibility, změny datového typu jsou téměř všechny binárně nekompatibilní, protože ClassLoader při linkování hledá metodu se starou hlavičkou metody (se starým datovým typem), která je uložená v byte kódu programu. V byte kódu nové knihovny takovouto metodu nemůže však nalézt. U konstant je poté proveden tzv. inlining.

Při pohledu na změny provedené v třídě jsou některé zdrojově nekompatibilní. Překladač při překladu s novou verzí provede kontrolu datových typů a skončí chybou. Je vidět, že změny týkající se konstruktoru vycházejí stejně jako změny týkající se metody, protože konstruktor je speciálním typem metody. Stejného jevu si můžeme všimnout v rámci proměnných a návratových typů. Klienti pro testování jsou v tomto případě velice podobní, získaná hodnota se ukládá do proměnné určitého datového typu.

V rozhraní jsou však všechny změny naopak zdrojově nekompatibilní, protože metoda s novým typem se tváří jako nová metoda. To vede k tomu, že překladač skončí chybou, protože tato nová metoda není implementovaná.

	proměnná	parametr *metody* / *konstruktoru*	návratový typ metody
mutace	●	●	●
generalizace	●	●	●
specializace	●	●	●
boxing	●	●	●
unboxing	●	●	●
widening	●	●	●
narrowing	●	●	●

Tabulka 1 - Datové typy

- **Mutace** – změna z jednoho datového typu na druhý, která nezapadá svojí povahou do žádné z následujících kategorií (např. `Integer` → `String`),
- **Generalizace** – změna datového typu v rámci hierarchie dědičnosti směrem nahoru (např. `Integer` → `Number`),
- **Specializace** – změna datového typu v rámci hierarchie dědičnosti směrem dolu (např. `Number` → `Integer`),
- **Boxing (zabalení)** – změna datového typu z primitivního na jeho obalující třídu (např. `int` → `Integer`),
- **Unboxing (rozbalení)** – změna datového typu z obalující třídy na jeho primitivní datový typ (např. `int` → `Integer`),
- **Widening (rozšíření)** – změna primitivního datového typu na primitivní datový typ s širším rozsahem (např. `int` → `double`),
- **Narrowing (zúžení)** – změna primitivního datového typu na primitivní datový typ s užším rozsahem (např. `double` → `int`).

4.2 Přístupové modifikátory

Další kategorií jsou přístupové modifikátory. Ačkoliv jsou přístupové modifikátory čtyři (`public`, `protected`, bez modifikátoru a `private` – seřazeny od modifikátoru s nejvyšší viditelností po nejnižší), mohou nastat pouze dva druhy změny. První změnou je *snížení viditelnosti* elementu, tedy přechod z modifikátoru s vyšší viditelností na modifikátor s nižší (např. `public` → `protected`). Druhá změna *zvýšení viditelnosti* je opakem první změny, viz Tabulka 2. Důvodem ke zvýšení viditelnosti může být potřeba použití metody i mimo její balík (bez modifikátoru na `public`), naopak ke snížení viditelnosti omezení přístupu k metodě kvůli lepšímu zapouzdření.

Na rozdíl od datových typů, zde změna poruší vždy oba typy kompatibility nebo ani jeden z nich. Důvodem porušení je pokus ClassLoaderu při linkování použít element, který má již sníženou viditelnost, tudíž není pro program dostupný. Překladač také skončí chybou, protože element není již nadále dostupný. Konkrétně změny *snížení viditelnosti* jsou tedy nekompatibilní, změny *zvýšení viditelnosti* jsou vždy kompatibilní.

	proměnná	metoda / konstruktor	třída
snížení viditelnosti	●	●	●
zvýšení viditelnosti	●	●	●
	rozhraní	zahnízděná třída	zahnízděné rozhraní
snížení viditelnosti	●	●	●
zvýšení viditelnosti	●	●	●

Tabulka 2 - Přístupové modifikátory

4.3 Nepřístupové modifikátory

Nepřístupové modifikátory, viz Tabulka 3, oproti přístupovým nastavují další vlastnosti elementů jako chování ve vícevláknovém prostředí, možnosti dědičnosti či pozměnitelnosti. K takovýmto změnám v softwaru může být opět nepřeberné množství důvodu, příkladem může být definování třídy `final`, aby se zabránilo její nežádoucímu dědění a tím zajištění jistého omezení ze strany knihovny.

	proměnná	metoda	třída
`static`	●	●	●
`final`	●	●	●
`abstract`		●	●
`synchronized`		●	
`native`		●	
`strictfp`		●	●
`transient`	●		
`volatile`	●		

Tabulka 3 - Nepřístupové modifikátory

- `static` – Statické elementy jsou svázané s třídou, tudíž za běhu vždy existuje pouze jedna instance.
- `final` – Takto označené elementy nelze měnit. Třída nemůže být děděna, metoda přepsána a proměnná nemůže změnit hodnotu.
- `abstract` – Metoda bez těla, která musí být přepsaná děděnou třídou.
- `native` - Lze použít pouze u metody. Nativní metody představují metody, které budou implementované v jiném programovacím jazyce. Metody jsou deklarované bez těla a nemohou být abstraktní. Před zrychlením Javy

se používali k urychlení průchodnosti kritických sekcí, v dnešní době už to není tak časté. Nyní se používají, pokud je zapotřebí z Javy zavolat knihovnu napsanou v jiném programovacím jazyce. Je možné je použít, pokud potřebujeme přístup k systémovým nebo hardwarovým prostředkům, které jsou dostupné, pouze z jiného jazyka (typicky C).

- `strictfp` – Lze použít u metody nebo pro celou třídu. Při práci s čísly s plovoucí desetinnou čárkou zajistí vždy stejnou přesnost výsledku nezávisle na platformě.
- `transient` – Pokud chceme například poslat proměnnou přes síť, je nutné ji serializovat (převést na pole bytů). Proměnná s modifikátorem `transient` nemůže být serializovaná.
- `volatile` – Při práci s vlákny mohou nastat problémy při přístupu ke sdílené proměnné. Pokud jedno vlákno mění její hodnotu, může druhé vlákno používat stále starou hodnotu, kterou má uchovanou v *cache* paměti. Pokud proměnná bude mít modifikátor `volatile`, vlákna používají při práci s ní hlavní paměť a mají vždy aktuální hodnotu.

Zatímco modifikátory `static`, `final` a `abstract` mají tedy vcelku zásadní vliv na zdrojovou či binární kompatibilitu, zbytek nemá vliv žádný. Naopak mají vliv na behaviorální kompatibilitu.

Modifikátor `static` porušuje zdrojovou kompatibilitu při odstranění od elementu, protože statický přístup k nestatickému objektu není možný. Binární kompatibilitu porušuje poté, jak při odstranění, tak při přidání. Element se v případě s modifikátorem `static` do byte kódu přeloží s jiným voláním než bez něho. Modifikátory `abstract` a `final` porušují oba typy kompatibility, nicméně pouze v případu kdy je daný modifikátor přidán. ClassLoader při linkování provádí kontrolu, zda jsou `abstract` elementy implementovány a zda `final` elementy nejsou děděny. Stejnou kontrolu provádí i překladač.

Dále je možné vytvořit speciální případ přidáním modifikátoru `final` k tzv. effectively final třídě (třída s privátním konstruktorem). Přidání tohoto modifikátoru ke třídě je samo o sobě změna nekompatibilní. Nicméně, za předpokladu, že třída má privátní konstruktor není se třeba obávat nekompatibilit s kterýmkoli klientem.

4.4 Členové

Jedná se o změny typu přidání nebo odstranění jednotlivých členů (proměnné, metody, konstruktory) nebo změny uvnitř členů (parametrů metod a konstruktorů). Dále je zde přidání nebo odstranění vnitřních tříd a rozhraní, viz Tabulka 4. Opět je zde celá řada důvodu pro změny tohoto typu, příkladem může být přidání proměnné do třídy, která modeluje dodatečnou vlastnost modelovaného doménového objektu.

Pokud se jedná o případy odstranění některého člena třídy nebo snad vnitřní třídy či vnitřního rozhraní, jsou to změny nekompatibilní. Stejně tak je tomu i u přidání nebo odstranění parametrů konstruktorů a metod. ClassLoader při linkování hledá daný element, ale nemůže ho v byte kódu nové knihovny nalézt. Přidání člena, vnitřní třídy nebo vnitřního rozhraní je poté změnou kompatibilní.

Za předpokladu, že se změna týká metody, která je navíc abstraktní, nezáleží poté, zda ji přidáváme či odebíráme. Tato změna ať ve třídě nebo v rozhraní je binárně kompatibilní, nicméně zdrojově nekompatibilní. Překladač kontroluje, zda je metoda implementována.

	třída	rozhraní
přidání proměnné	●	
odstranění proměnné	●	
přidání konstanty	●	●
odstranění konstanty	●	●
přidání metody	●	●
odstranění metody	●	●
přidání parametru metody	●	●
odstranění parametru metody	●	●
přidání konstruktoru	●	
odstranění konstruktoru	●	
přidání parametru konstruktoru	●	
odstranění parametru konstruktoru	●	
přidání vnitřní třídy	●	
odstranění vnitřní třídy	●	
přidání vnitřního rozhraní	●	●
odstranění vnitřního rozhraní	●	●

Tabulka 4 - Členové

4.5 Dědičnost

Jedná se o změny v hierarchické struktuře dědičnosti, viz Tabulka 5, ať už přidáním či odebráním rodičovské třídy nebo rozhraní či přesuny členů v různých směrech ve struktuře. Příkladem důvodu této změny může být přesun metody do rodičovské třídy, která je společná několik dědícím třídám, aby tato metoda byla dostupná i dalším dědícím třídám.

Zde jsou některé změny, jak binárně, tak zdrojově nekompatibilní. U zredukování množiny rodičovských tříd či rozhraní je důvodem u ClassLoaderu, že při linkování hledá element, který již v novém byte kódu neexistuje. To samé je i v případě, že třída či rozhraní přestane dědit.

	třída	rozhraní
přidání dědičnosti (začne dědit)	●	●
odstranění dědičnosti (přestane dědit)	●	●
rozšíření množiny rodičovských rozhraní		●
zredukování množiny rodičovských rozhraní		●
rozšíření množiny rodičovských tříd	●	
zredukování množiny rodičovských tříd	●	
metoda přesunuta do rodičovské třídy / rozhraní	●	●
metoda přesunuta z rodičovské třídy / rozhraní	●	●
přepsání (`override`) zděděné metody	●	
odstranění (`override`) zděděné metody	●	

Tabulka 5 - Dědičnost

Mějme třídy A, B, C a rozhraní IA, IB, IC.

- **Přidání dědičnosti** – třída A nejprve nemá za rodičovskou třídu žádnou z uvedených. V nové verzi knihovny začne třída A dědit třídu B. Opakem této změny je **odstranění dědičnosti**. Třída A dědí třídu B a v nové verzi ji přestane dědit.
- **Rozšíření množiny rodičovských rozhraní** – mějme rozhraní IA, který dědí rozhraní IC. Mluvíme-li o **rozšíření množiny rodičovských rozhraní,** jedná se o vložení rozhraní IB mezi tyto dvě rozhraní. Poté budeme mít tedy IA dědící IB a IB dědící IC.

- **Zredukování množiny rodičovských rozhraní** – pokud z hierarchie dědičnosti (IA dědí IB a IB dědí IC) odstraníme rozhraní IB.
- **Rozšíření množiny rodičovských tříd** – stejný případ jako při **rozšíření množiny rodičovských rozhraní**, pouze se jedná o třídy místo rozhraní.
- **Zredukování množiny rodičovských tříd** - stejný případ jako při **zredukování množiny rodičovských rozhraní**, pouze se jedná o třídy místo rozhraní.
- **Metoda přesunuta do rodičovské třídy/rozhraní** – mějme metodu `metoda1()` ve třídě A, která dědí od třídy B. V nové verzi knihovny je tato metoda `metoda1()` přesunuta do třídy B (rodičovské). Pokud metodu `metoda1()` přesuneme z rodičovské třídy do potomka, jedná se o změnu m**etoda přesunuta z rodičovské třídy/rozhraní.**
- **Přepsání (override) zděděné metody** – potomek v nové verzi přepíše (i s anotací @Override) metodu `metoda1()` zděděnou ze svého rodiče.
- **Odstranění (override) zděděné metody** – odstranění přepsané zděděné metody z potomka (v rodičovi zůstane).

4.6 Generika

Tato kategorie byla do Javy přidána ve verzi 1.5. Základní změnou v této kategorii je vytvoření generického typu nebo generické metody či konstruktoru nebo přidáním typového parametru k původním elementům Javy. Typové parametry je dále možné omezit v rámci možných použitých datových typů, což zpřístupňuje další možnosti testování, viz Tabulka 6.

Dále je možné přidat více typových parametrů k elementům Javy nebo přidat více omezení. V případě vícenásobného omezení je možné použít pouze jednu třídu, která omezí typový parametr. Ostatní omezení mohou být jedině typu rozhraní.

Také je možné provést opak těchto změn (tj. odstranění typového parametru od Java elementům, odstranění omezení typových parametrů nebo pouze zredukování počtů jak typových parametrů tak omezení).

Všechny níže uvedené změny jsou binárně kompatibilní. Je to důsledkem vymazání generických typů v době překladu (*type erasure*) a jejich nahrazením v byte kódu typem `Object` nebo ohraničujícím typem, přetypováním či přemostící pomocnou metodou (*bridge methods*) ve složitějších případech [9]. Avšak hrozí zde zdrojová nekompatibilita, pokud například v nové verzi knihovny odstraníme generický typ.

	metoda / konstruktor	třída	rozhraní
přidání gen. typu	●	●	●
odstranění gen. typu	●	●	●
přidání druhého gen. typu	●	●	●
odstranění jednoho z dvou gen. typu	●	●	●
změna pořadí gen. typu	●	●	●
přidání ohraničení (`extends`)	●	●	●
odstranění ohraničení (`extends`)	●	●	●
přidání druhého ohraničení	●	●	●
odstranění jednoho z ohraničení	●	●	●
změna pořadí ohraničení	●	●	●

Tabulka 6 - Generika

Wildcards

Součástí Generik jsou tzv. wildcards, které se v kódu značí jako "?".

	proměnná	parametr *metody / konstruktoru*
nahrazení konkrétního datového typu	●	●
nahrazení konkrétním datovým typem	●	●
přidání horní ohraničení	●	●
odstranění horní ohraničení	●	●
mutace horního ohraničení	●	●
generalizace horního ohraničení	●	●
specializace horního ohraničení	●	●
změna horního ohraničení na dolní	●	●
přidání dolní ohraničení	●	●
odstranění dolního ohraničení	●	●
mutace dolního ohraničení	●	●
generalizace dolního ohraničení	●	●
specializace dolního ohraničení	●	●
změna dolního ohraničení na horní	●	●

Tabulka 7 - Wildcards

Změnou zde může být, nahrazení datového typu znakem wildcards popřípadě nahrazení znaku wildcards datovým typem. Wildcards je také možné omezit shora a navíc oproti generikám i zdola, což umožňuje dělat další změny, viz Tabulka 7.

Zde se také používá *type erasure*, změny zde provedené jsou tedy též binárně kompatibilní, avšak stejně jako je tomu u základních generik může nastat zdrojová nekompatibilita. Například přidáme-li některé z omezení. Některý z klientů používající knihovnu mohl volat element s datovým typem, který toto nové omezení nesplňuje. Překladač poté skončí chybou při typové kontrole jako u generik.

4.7 Výjimky

Pokud v metodě dojde k výjimečnému stavu, z kterého se neumí zotavit nebo má být zotavení zařízeno volajícím kódem, vyhazuje výjimku. V Javě existují dva typy výjimek a to kontrolované (*checked*) a nekontrolované (*unchecked*). Rozdíl mezi nimi je podstatný. Kontrolované výjimky jsou odděděny od obecné rodičovské třídy `Exception`, kdežto nekontrolované `RuntimeException`. Rozdíly jsou i v chování. Pokud metoda vyhazuje kontrolovanou výjimku, překladač donutí klientský kód tuto výjimku ošetřit, kdežto pro nekontrolovanou výjimku toto neplatí.

Ošetření je možno dosáhnout jedním ze dvou způsobů, a to ošetřením v klientském kódu pomocí `try-catch` bloku nebo propagací do vyšší úrovně pomocí `throws` klauzule v signatuře metody.

Z výše uvedeného tudíž vyplývá, že jsou zde i podstatné rozdíly mezi kontrolovanými a nekontrolovanými výjimku z pohledu zdrojové a binární kompatibility.

Jednotlivé změny v kategorii výjimky jsou vypsány v tabulce Tabulka 8. Mutace zde znamená změna z checked na unchecked exception a naopak.

	metoda
přidání `catch` bloku	●
odstranění `catch` bloku	●
přidání `finally` bloku	●
odstranění `finally` bloku	●
přidání checked `throw`	●
odstranění checked `throw`	●
mutace checked `throw`	●
generalizace checked `throw`	●
specializace checked `throw`	●
přidání unchecked `throw`	●
odstranění unchecked `throw`	●
mutace unchecked `throw`	●
generalizace unchecked `throw`	●
specializace unchecked `throw`	●
`Try catch` blok na `throw`	●
`throw` na `try catch` blok	●

Tabulka 8 - Výjimky

4.8 Ostatní

Do této kategorie patří změny, které nezapadají do žádné z předchozích kategorií, jako je například změna definice třídy na rozhraní a naopak. Dále zde může být přidání či odstranění těchto dvou zmíněných elementů, viz Tabulka 9.

Odstranění třídy či rozhraní, nebo změna z interface na třídu či naopak, jsou změny binárně a zdrojově nekompatibilní. Důvod je zde stejný jako u odstranění členů, tedy změny v byte kódu nové knihovny.

	balík
třída na rozhraní	●
rozhraní na třídu	●
přidání třídy/ rozhraní	●
odstranění třídy/ rozhraní	●

Tabulka 9 - Ostatní změny

4.9 Pojmenování testů

Pro rychlou identifikaci jednotlivých testů bylo použito specifického pojmenování, které je použité v názvech balíků a také v názvech tříd, a mají následující strukturu:

<kategorie> <element> <změna>

Podkapitoly této kapitoly tvoří jednotlivé kategorie změn, čemuž odpovídá část *<kategorie>*.

Při pouhém pohledu na název balíku musí být jasné, jakého elementu Javy se daná změna týká. Zda jde o třídu, rozhraní nebo některý z jejich členů resp. parametr jejich členů. Tohoto je docíleno částí *<element>*. Definice elementu se provádí od obecné (třída či rozhraní) k nejvíce specifické (až parametr metody či konstruktoru).

Poslední část *<změna>* je poté stručný popis samotné testované změny.

Příkladem zde může být změna datového typu v parametru konstruktoru.

`dataTypeClazzConstructorParamMutation`

První část `dataType` tedy uvádí, že se jedná o změnu v kategorii datových typů. Následující část `ClazzConstructorParam` poté specifikuje Java element. Poslední část `Mutation` definuje změnu datového typu.

5 Metodika

Pro samotné testování a porovnání jednotlivých nástrojů na kontrolu zpětné kompatibility je nutné určit rámec, ve kterém samotné srovnání bude probíhat.

V následující části této kapitoly je dále definovány zmíněný rámec, jejímž základem jsou kritéria na porovnání daného softwaru. Při stanovení těchto kritérií je nutné vycházet ze samotného zadání práce, z něhož vychází, že hlavním kritériem je tedy míra, do jaké nástroje dokáží odhalit možné nekompatibility. Ale také je nutné si uvědomit, že s nástroji pracují uživatelé. Vedlejší kritéria tedy vychází z jejich běžné práce s nástroji.

Po stanovení srovnávacích kritérií je nutné následně určit vhodnou metodu srovnání, která se u každého kritéria může lišit.

5.1 Stanovení kritéria srovnání

Zde jsou podrobně vysvětlená jednotlivá kritéria srovnání.

Odhalení nekompatibilit

Odhalení nekompatibilit neboli míra, do jaké nástroje dokáží odhalit možné nekompatibility. Jedná se o primární kritérium pro srovnání nástrojů, neboť je to vlastnost, která byla důvodem vzniku testovaných nástrojů.

Jedná se o úspěšnost nástrojů, neboli kolik ze změn provedených v knihovně, dokáže nástroj nalézt správně. Nejde pouze o to, zda dovede nalézt nekompatibilní změny, ale také zda vyhodnotí správným způsobem kompatibilní změny.

Snadnost použití

Prvním mimofunkčním hlediskem k porovnání je snadnost použití. Tímto kritériem se rozumí, jakými způsoby je možné dané nástroje použít. Typickým může být použití nástroje přes příkazový řádek. V dnešní době se často využívá spuštění jako Maven plugin. Mimo tyto možnosti je možné některé nástroje použít také přes Ant task.

Výstupní formát

Dalším hlediskem k porovnání jsou možnosti volby výstupního formátu. Základní položkou v tomto kritériu je výstupní soubor obsahující text popisující jednotlivé testované změny. Dalšími možnými výstupy jsou soubory ve formátu XML nebo HTML

Přehlednost výstupu

Nedílnou součástí výstupního formátu je také jeho přehlednost. Uživatel by se měl ve výstupu jednoduše orientovat. Je nutné, aby z něho bylo možné snadno a rychle vyčíst veškeré nástrojem nalezené změny. Hlavním prvkem je určení závažnosti změny. Dále by určitě nemělo u jednotlivých změn chybět jejich umístění v rámci knihovny. Čím přesnější bude, tím lépe. Uživatel poté může rychle nalézt nekompatibilní změny. Ve výstupu by také neměl chybět alespoň stručný popis změny.

Integrovatelnost do vývojového cyklu

Nástroje jsou porovnávány i z hlediska integrovatelnosti do vývojového cyklu. V případě, že uživatel používá Maven projekt nebo Ant, je možnost nástroje ho integrovat například ve formě pluginu vítanou vlastností. Je tedy nutné zjistit, jaké nástroje toto umožňují.

Uživatelská dokumentace

Posledním kritériem je uživatelská dokumentace, která je nezbytností. Samozřejmě záleží na kvalitě dokumentace. Je vhodné, aby nástroje měli příklady použití pro každý typ použití (CLI, Maven, atd.) a příklady výstupů.

5.2 Způsob vyhodnocení kritérií

Zdrojové kódy simulující změny při vývoji Java knihovny jsou umístěny ve složkách *src* vytvořeného Maven projektu a jeho modulů, přeloží se příkazem `ant compile` ze složky *scripts* nebo použitím příkazu `mvn install` ze složky projektu. Poslední možností je rovnou použít test kompatibility `ant -Dpackage="název testovaného balíku"`.

Tím vzniknou dva JAR soubory ve složkách target v jednotlivých modulech projektu, představující původní verzi Java knihovny L_1 (*testing-lib-v1-0.0.1.jar*) a změněnou verzi L_2 (*testing-lib-v2-0.0.2.jar*), které slouží jako vstupy pro následné testování nástrojů.

Nástroje jsou rozděleny do dvou hlavních složek podle operačních systémů. Jedna složka *tools_win* slouží pro nástroje, které jsou testované v operačním systému Windows, druhá *tools_linux* pro nástroje testované v Linuxu. Obě složky obsahují soubor *runtools.bat*, respektive *runtools.sh*, kterým lze spustit všechny nástroje z dané složky najednou. Každý nástroj ve složce také obsahuje spouštěcí soubor, kdyby bylo požadováno spustit jej samostatně. V této fázi se hodnotí kritéria *snadnost použití* a *integrovatelnost*.

Po spuštění nástrojů jsou reporty vygenerované do složky *reports*. Vyhodnocení kritérií *odhalení nekompatibilit, výstupní formát* a *přehlednost výstupu* je poté provedeno ručně výstup po výstupu.

Každé z těchto kritérií je ve své podstatě odlišné a není možné je hodnotit stejným způsobem. Některá se porovnávají na základě výstupů nástrojů, u jiných se porovnávají jejich vlastnosti. Je tedy jasné, že se metody vyhodnocení jednotlivých kritérií liší.

Nicméně z tohoto důvodu je nutné pro všechna kritéria stanovit jednotnou metriku, v rámci které naměřené a jinak získané výsledky jsou převedeny na body. U hlavního kritéria *Odhalení nekompatibilit* toto znamená počet všech správně vyhodnocených nekompatibilních, ale také kompatibilních změn v připravených datech. Každá vlastnost nástroje vyhodnocená jako mimofunkční charakteristika může poté být dalším bodem pro daný nástroj navíc. Tedy až na jedinou výjimku *Příklady použití,* kde se body

nástroji odebírají. Výsledné vyhodnocení, který nástroj na kontrolu zpětné kompatibility je nejlepší, bylo poté provedeno jako součet bodů jednotlivých kritérií.

5.2.1 Metriky jednotlivých kritérií

Jak již bylo zmíněno, celkové hodnocení bude součtem bodů získaných v každém z kritérií. Nejsou k dispozici data o tom, které vlastnosti v rámci kritéria jsou důležitější, nejsou tedy stanoveny váhy. Znamená to například, že žádná nalezená změna nemá vyšší váhu v rámci kritéria *Odhalení nekompatibilit*. Stejně tak například výstupní formát XML není důležitější než formát HTML.

Odhalení nekompatibilit

Jednotlivé nástroje je nutné spustit nad samotnými testovacími daty vytvořenými podle kapitoly 4. Jejich výstupy následně zkontrolovat, zda dokáží detekovat veškeré nekompatibility v připravených testech, případně některou z kompatibilit neoznačit jako chybnou. Každá dobře vyhodnocená změna poté představuje jeden bod.

Snadnost použití

Toto kritérium bylo vyhodnoceno na základě, kolika možnostmi je daný nástroj možné použít. Každá možnost použití tedy představuje jeden bod.

Výstupní formát

Výstupní formát je hodnocen u jednotlivých nástrojů na základě počtu typů výstupních souborů, které jsou podporovány. Jako jeden typ je brán v úvahu prostý textový výstup. Dalšími typy jsou XML nebo HTML. Opět každý typ představuje jeden bod.

Přehlednost výstupu

U přehlednosti výstupu se hodnotí, zda obsahuje veškeré informace u výpisů nalezených změn. Tedy umístění, kde se chyba nachází, textový popis chyby, její závažnost a rozpoznání, zda se jedná o binární či zdrojovou nekompatibilitu.

Integrovatelnost do vývojového cyklu

Testovaný nástroj získá bod za každý nástroj, ze kterého jej lze použít ve vývojovém cyklu (Maven, Ant).

Uživatelská dokumentace

Dokumentace je hodnocena u jednotlivých nástrojů na základě příkladů použití. Pokud má nástroj více příkladů použití pro jednu možnost použití, dostane pouze jeden bod.

5.3 Použitý hardware a software

Pro účely zpracování této práce byla použita platforma Windows a desktopový počítač. Konkrétní konfigurace:

- Operační systém: Microsoft Windows 7 Professional 64-bit, Service Pack 1
- Procesor: AMD PhenomTM II X4 955 Processor 3.20 GHz
- Operační paměť: 4.00 GB, DDR
- Základní deska: Gigabyte GA-MA790XT-UD4P
- Pevný disk: Samsung SSD 840 EVO 250 GB

Testovací data jsou psána v programovacím jazyce Java konkrétně ve verzi 1.8.0_91. Bylo použito vývojové prostředí Eclipse Luna verze 4.4.1. Pro sestavení a řízení závislostí jsem použil nástroj Apache Maven 3.3.9. Ke zjištění, zda jsou jednotlivé testy kompatibilní či nekompatibilní bylo použito sestavovacího nástroje Apache Ant verze 1.9.6.

Pro nástroje, které byly testované v Linuxu. Virtualizačním nástrojem VMware Workstation verze 12.1.0. byl vytvořen virtuální stroj se systémem Linux. Konkrétně byl použit Debian 8.4.0 amd64.

6 Nástroje na kontrolu kompatibility

Detekce kompatibility mezi dvěma verzemi téže Java knihovny není nic nového. Existuje několik nástrojů řešící tento problém. Jednotlivá řešení jsou podrobněji popsána níže v této kapitole. Analýza existujících řešení je důležitá pro jejich vzájemné porovnání. Jsou probrány základní informace (autor, licence, možnosti použití, atd.) a výstupy s příklady.

Nástroje:

- Clirr,
- Japicmp
- Japi checker
- Java API comlpiance checker
- Japitools
- Revapi
- Sigtest
- Jour a Javassist
- JaCC (nástroj katedry)

Všechny nástroje jsou testovány za účelem, aby se zjistilo, jaké změny v Javě dovedou odhalit, což v podstatě zjistí i jejich nedostatky. Jsou testovány se stejným vstupem, což jsou dvě verze téže testovací knihovny. Tato knihovna je vytvořena podle kapitoly 4. Každý nástroj má odlišný výstupní formát, což znemožňuje vytvoření automatizovaných testů. U každého nástroje je zobrazena krátká část výstupní zprávy.

6.1 Clirr

Clirr [10] je open source nástroj na kontrolu kompatibility napsaný Larsem Kühnem. Testovaná verze 0.6.0 byla publikována pod licencí GNU LGPL (Lesser

General Public License) v roce 2005. Nástroj je možné použít přes příkazovou řádku, jako Ant task nebo jako Maven plugin.

Formální popis spouštěcího příkazu:

```
java -jar clirr-core-0.6-uber.jar <-o oldJar> <-n newJar> [-f outfile]
```

Ukázka konkrétního CLI příkazu spuštění:

```
java -jar clirr-core-0.6-uber.jar -o testing-lib-v1-0.0.1.jar -n
testing-lib-v2-0.0.2.jar -f clirrReport.txt
```

Výstup nástroje může být ve formě textu, viz příklad níže, nebo ve formátu XML. Maven plugin umožňuje výstup pouze ve formátu XML. Project Mojo Haus vytvořil plugin pro Maven 2, k jehož vytvoření byla použita poslední verze nástroje. Výstup tohoto pluginu je soubor ve formátu HTML. Textový výstup je seznam zpráv popisující změny v knihovně, kde jedna zpráva odpovídá jednomu Java elementu, na kterém změna proběhla. Každá zpráva má následující strukturu:

<*závažnost*>: <*kód zprávy*>: <*umístění*>: <*typ změny*>

Příklad části textového výstupu:

```
INFO: 7011: testing_lib.membersClazzMethodAdd.MembersClazzMethodAdd:
Method 'public void method1()' has been added

ERROR: 7002:
testing_lib.membersClazzMethodDelete.MembersClazzMethodDelete: Method
'public void method1()' has been removed

RROR: 6004:
testing_lib.dataTypeIfazeConstantBoxing.DataTypeIfazeConstantBoxing:
Changed type of field FIELD1 from int to java.lang.Integer

WARNING: 6002:
testing_lib.dataTypeIfazeConstantBoxing.DataTypeIfazeConstantBoxing:
Value of field FIELD1 is no longer a compile-time constant
```

Clirr umožňuje volbu blíže specifikovat, který balík, s jeho podřízenými balíky, bude testován. K parsování byte kódu používá knihovnu ASM.

Hlavní nevýhodou tohoto nástroje je jeho zastaralost. Nepodporuje elementy přidané do Javy verze 5 (generika).

6.2 Japicmp

Japicmp [11] je nástroj na kontrolu kompatibility napsaný Martinem Moisem. Testovaná verze 0.7.2 byla publikována pod licencí Apache 2.0 v roce 2016. Nástroj je možné použít přes příkazovou řádku, jako Maven plugin nebo jako knihovnu přímo v kódu.

Formální popis spouštěcího příkazu:

```
java -jar japicmp-0.7.2-jar-with-dependencies.jar <-o oldJar> <-n
newJar> [-a accessModifier]
```

Ukázka konkrétního CLI příkazu spuštění:

```
java -jar japicmp-0.7.2-jar-with-dependencies.jar -o testing-lib-v1-
0.0.1.jar -n testing-lib-v2-0.0.2.jar -a private > japicmpReport.txt
```

Výstup nástroje může být ve formě textu (tzv. simple diff format), viz příklad níže, ve formátu XML nebo ve formátu HTML. Textový výstup je seznam zpráv popisující změny v knihovně, kde jsou zprávy rozděleny podle úrovně odsazení. První úroveň určuje, jestli se jedná o změnu ve třídě nebo v rozhraní. Druhá úroveň popisuje změny u elementů uvnitř tříd či rozhraní. Každá zpráva má následující strukturu:

<značka + druh změny> <Java element>: *<modifikátory>* [*návratový typ*] *<umístění>*

Příklad části textového výstupu:

```
***  MODIFIED CLASS: PUBLIC
testing_lib.membersClazzMethodAdd.MembersClazzMethodAdd  (not
serializable)
   ===  UNCHANGED CONSTRUCTOR: PUBLIC MembersClazzMethodAdd()
   +++  NEW METHOD: PUBLIC(+) void method1()

***! MODIFIED CLASS: PUBLIC
testing lib.membersClazzMethodDelete.MembersClazzMethodDelete  (not
serializable)
   ===  UNCHANGED CONSTRUCTOR: PUBLIC MembersClazzMethodDelete()
   ---! REMOVED METHOD: PUBLIC(-) void method1()

**** MODIFIED INTERFACE: PUBLIC ABSTRACT
testing_lib.membersIfazeMethodAdd.MembersIfazeMethodAdd  (not
serializable)
   +++* NEW METHOD: PUBLIC(+) ABSTRACT(+) void method1()
```

Japicmp umožňuje volbu blíže specifikovat, který balík, s jeho podřízenými balíky, bude testován. Navíc umožňuje definovat balíky, které nebudou testované.

6.3 Japi checker

Japi checker [12] je nástroj na kontrolu kompatibility napsaný Williamem Bernardetem. Testovaná verze 0.2.1 byla publikována pod licencí Apache 2.0 v roce 2015. Nástroj je možné použít jako Ant task, jako Maven plugin a od roku 2013 byla provedena integrace s projektem Tomáše Rohovského. Od té doby je možné Japi checker používat jako samostatnou aplikaci přes příkazovou řádku. Nicméně nástroj je nutné klonovat z git repozitáře a sestavit nástrojem Maven.

Formální popis spouštěcího příkazu:

```
java -jar japi-checker-cli-0.2.1-SNAPSHOT-shaded.jar <oldJar> <newJar>
[-bin]
```

Ukázka konkrétního CLI příkazu spuštění:

```
java -jar japi-checker-cli-0.2.1-SNAPSHOT-shaded.jar testing-lib-v1-
0.0.1.jar testing-lib-v2-0.0.2.jar -bin > japiCheckerReport.txt
```

Textový výstup je seznam zpráv popisující změny v knihovně, kde jedna zpráva odpovídá jednomu Java elementů, na kterém změna proběhla. Každá zpráva má následující strukturu:

<závažnost>: *<umístění>*: *<typ změny>*

Příklad části textového výstupu:

```
ERROR:
testing_lib/membersClazzMethodDelete/MembersClazzMethodDelete.java:
Could not find method method1(()V) in newer version.

WARNING:
testing_lib/accessModifierClazzFieldAccessIncrease/AccessModifierClazz
FieldAccessIncrease.java: The visibility of the fieldPrivateToPublic
field has been changed from PRIVATE to PUBLIC
```

Hlavní nevýhodou tohoto nástroje je velmi slabá dokumentace. Dále také, při použití Maven pluginu se vypíše počet problémů, ale bez dalších informací a neprovede se překlad.

6.4 Java API compliance checker

Java API compliance checker [13] (dále jen JAPICC) je open source nástroj na kontrolu kompatibility napsaný v perlu Andreyem Ponomarenkem. Testovaná verze 1.6 byla publikována pod licencí GNU GPL (General Public License) nebo LGPL v roce 2016. Nástroj je možné použít pouze přes příkazovou řádku. Dále je nutné mít nainstalovaný Active Perl 5 nebo vyšší, aby bylo možné nástroj přeložit a spustit.

Formální popis spouštěcího příkazu:

```
japi-checker-cli-0.2.1-SNAPSHOT-shaded.jar <oldJar> <newJar>
```

Ukázka konkrétního CLI příkazu spuštění:

```
japi-compliance-checker testing-lib-v1-0.0.1.jar testing-lib-v2-
0.0.2.jar
```

Vstupní data lze nástroji zadat více způsoby. Prvním a hlavně testovaným vstupem je zadání dvou verzí téže knihovny (jar soubory). Dále je možné zadat do vstupu dva XML soubory, které obsahují cesty k jar souborům. Výstupem je HTML soubor, který obsahuje výsledky testů (např. jestli testované knihovny jsou kompatibilní, procento kompatibility, kolik balíků, tříd a metod je ovlivněno), kolik metod bylo přidáno a kolik odebráno, problémy s datovými typy a problémy s metodami. Dále jsou vypsané a rozebrané jednotlivé nalezené změny.

Nástroj je implementován jako jeden skript soubor. Od definice zpráv, všechna kontrolovaná pravidla, po generování HTML výstupu. To zhoršuje možnosti budoucího rozšíření.

6.5 Revapi

Revapi [14] je open source nástroj na kontrolu kompatibility napsaný Lukášem Krejčím. Testovaná verze 0.4.2 byla publikována pod licencí Apache 2.0 v roce 2016. Nástroj je možné použít přes příkazovou řádku, jako Ant task nebo jako Maven plugin.

Formální popis spouštěcího příkazu:

```
./revapi.sh <--extensions=extension1,extension2…> <--old=oldJar> <--
new=newJar> <-D config>
```

Ukázka konkrétního CLI příkazu spuštění:

```
./revapi.sh \
--extensions=org.revapi:revapi-java:0.8.0,org.revapi:revapi-reporting-
text:0.4.1 \
--old=testing-lib-v1-0.0.1.jar \
--new=testing-lib-v2-0.0.2.jar \
-D revapi.reporter.text.minSeverity=NON_BREAKING > revapiReport.txt
```

Výstupem je seznam změn, kde každá změna je popsána zprávou, která je tvořená minimálně třemi řádky. První řádek je uvozen slovem *old* a obsahuje popis jak daný element, kterého se změna týká, vypadal. Druhý řádek je popis elementu, v nové verzi knihovny, uvozen slovem *new*. Následuje popis změn, které byly na elementu provedeny. Popis změn má následující strukturu:

<typ změny> <informace o kompatibilitě>: *<popis změny>*

Příklad části textového výstupu:

```
old: <none>
new: method void
testing_lib.membersClazzMethodAdd.MembersClazzMethodAdd::method1()
java.method.added BINARY: NON_BREAKING, SOURCE: NON_BREAKING: Method
was added.

old: method void
testing_lib.membersClazzMethodDelete.MembersClazzMethodDelete::method1
()
new: <none>
java.method.removed BINARY: BREAKING, SOURCE: BREAKING: Method was
removed.
```

6.6 Sigtest

Sigtest [15] je kolekce open source nástrojů, zahrnující nástroj na porovnání API a také na měření pokrytí API testy. Kolekce je distribuovaná společností Oracle. Testovaná verze 3.1 byla publikována pod licencí GNU GPLv2 v roce 2015. K našemu účelu slouží nástroj na porovnání API. Nástroj je možné použít přes příkazovou řádku, jako Ant task nebo jako Maven plugin.

Nástroj umožňuje zavolat čtyři příkazy: *Setup*, *SetupAndTest*, *Test* a *Merge*.

Formální popis spouštěcího příkazu:

```
java -jar sigtestdev.jar SetupAndTest <-reference
oldJar;dependencyJars> <-test newJar;dependencyJars> [-package
testedPackage] [-H] [-out outFile]
```

Ukázka konkrétního CLI příkazu spuštění:

```
java -jar sigtestdev.jar SetupAndTest -reference testing-lib-v2-
0.0.1.jar;"c:/Program Files/Java/jdk1.8.0_77/jre/lib/rt.jar" -test
testing-lib-v2-0.0.2.jar;"c:/Program
Files/Java/jdk1.8.0_77/jre/lib/rt.jar" -package testing_lib -H -out
revapiReport.txt
```

Pokud chceme zjistit rozdíly mezi dvěma verzemi knihovny, musíme z jedné vytvořit takzvaný *signature* soubor, což je v podstatě textová reprezentace API. Toho dosáhneme příkazu *Setup*. Formát *signature* souboru je pro člověka dobře čitelný.

Příklad části ze *signature* souboru:

```
CLSS public testing_lib.membersClazzMethodAdd.MembersClazzMethodAdd
cons public <init>()
supr java.lang.Object

CLSS public
testing_lib.membersClazzMethodDelete.MembersClazzMethodDelete
cons public <init>()
meth public void method1()
supr java.lang.Object
```

Následně musíme program použít s příkazem *Test*, kde parametry budou vytvořený *signature* soubor a knihovna, kterou chceme porovnávat, abychom zjistili rozdíly.

Jednoduší možností je zavolat wrapper příkaz *SetupAndTest*, který tyto dva příkazy zavolá za nás.

Dále musíme k oběma porovnávaným souborům přidat na vstupu také jejich závislosti. Proto je nutné k oběma vstupům přidat *rt.jar* knihovnu, která obsahuje všechny základní třídy pro Java runtime enviroment.

U výstupu lze nastavit, zda má být ve formátu čitelném pro stroje nebo pro lidi. U formátu, čitelného pro stroje, jsou změny vypsány jako seznam přidaných či odstraněných elementů. U druhého formátu jsou zobrazeny i změny u jednotlivých elementů. Nejprve je vypsaná třída, ve které změna nastala, a poté jsou vypsány jednotlivé změny.

Příklad části textového výstupu:

```
Class testing_lib.membersClazzMethodAdd.MembersClazzMethodAdd
  Added Methods
    method public void
testing_lib.membersClazzMethodAdd.MembersClazzMethodAdd.method1()

Class testing_lib.membersClazzMethodDelete.MembersClazzMethodDelete
  Missing Methods
    method public void
testing_lib.membersClazzMethodDelete.MembersClazzMethodDelete.method1(
)
```

6.7 Japitools

Japitools [16] je kolekce dvou open source nástrojů (*japize*, *japicompat*), které slouží k porovnání API. Původně byly navrženy na testování volných implementací Javy, ale mohou být použity na testování jakýkoli API. Autorem kolekce je Stuart Bellard. Testovaná verze 0.9.7 byla publikována pod licencí GNU GPL v roce 2006. Na rozdíl od kolekce SigTest zde budeme používat oba nástroje, které kolekce obsahuje. Nástroj je možné použít pouze přes příkazovou řádku.

Formální popis spouštěcího příkazu:

```
japize [unzip] [as <name>] packages <zipfile>|<dir> ... +|-<pkgpath>
...

japicompat [-svqhtjw4] [-o outfile] [-i ignorefiles] <original_api>
<api_to_check>
```

Ukázka konkrétního CLI příkazu spuštění:

```
japize as japizeSigFile packages testing-lib-v1-0.0.1.jar rt.jar
+testing_lib

japize as japizeSigFile2 packages testing-lib-v2-0.0.2.jar rt.jar
+testing_lib

japicompat -o japitoolsReport.txt japizeSigFile.japi.gz
japizeSigFile2.japi.gz
```

Pokud chceme zjistit rozdíly mezi dvěma verzemi knihovny, musíme s nástrojem *japize* z obou vytvořit takzvaný *signature* soubor, což je v podstatě soubor popisující API. K oběma porovnávaným souborům musíme, stejně jako u nástroje Sigtest, přidat na vstupu také jejich závislosti. Proto je nutné k oběma vstupům přidat knihovnu *rt.jar*, obsahující všechny základní třídy pro Java runtime enviroment.

Příklad části ze *signature* souboru:

```
testing_lib.membersClazzMethodDelete,MembersClazzMethodDelete! Pcsnur
class:java.lang.Object
testing_lib.membersClazzMethodDelete,MembersClazzMethodDelete!()
Pcinur constructor
testing_lib.membersClazzMethodDelete,MembersClazzMethodDelete!clone()
pcinur Ljava/lang/Object;*java.lang.CloneNotSupportedException
testing_lib.membersClazzMethodDelete,MembersClazzMethodDelete!equals(L
java/lang/Object;) Pcinur Z
testing_lib.membersClazzMethodDelete,MembersClazzMethodDelete!finalize
() pcinur V*java.lang.Throwable
testing_lib.membersClazzMethodDelete,MembersClazzMethodDelete!getClass
() Pcifur Ljava/lang/Class<{Ljava/lang/Object;>;
testing_lib.membersClazzMethodDelete,MembersClazzMethodDelete!hashCode
() Pcinur I
testing_lib.membersClazzMethodDelete,MembersClazzMethodDelete!method1(
) Pcinur V
testing_lib.membersClazzMethodDelete,MembersClazzMethodDelete!notify()
Pcifur V
testing_lib.membersClazzMethodDelete,MembersClazzMethodDelete!notifyAl
l() Pcifur V
testing_lib.membersClazzMethodDelete,MembersClazzMethodDelete!toString
() Pcinur Ljava/lang/String;
testing_lib.membersClazzMethodDelete,MembersClazzMethodDelete!wait()
Pcifur V*java.lang.InterruptedException
testing_lib.membersClazzMethodDelete,MembersClazzMethodDelete!wait(J)
Pcifur V*java.lang.InterruptedException
testing_lib.membersClazzMethodDelete,MembersClazzMethodDelete!wait(J,I
) Pcifur V*java.lang.InterruptedException
```

Následně musíme tyto dva vygenerované *signature* soubory porovnat programem japicompat.

U výstupu lze nastavit, zda má být ve formátu čitelném pro stroje, pro lidi nebo ve formátu HTML. U formátu čitelného pro lidi je nejprve zobrazený výčet balíků s procenty zobrazujícími, jak závažné změny jsou v nich provedeny. Následuje výčet chyb a za ním je jejich detailní popis. Nejprve je vypsán balík, ve které změna nastala. Následuje závažnost změny a poté jsou vypsány jednotlivé změny.

Příklad části textového výstupu:

```
testing_lib.membersClazzMethodDelete:
Missing
method
testing_lib.membersClazzMethodDelete.MembersClazzMethodDelete.method1(
): missing in testV2
```

6.8 Jour a Javassist

Jour [17] je navržen, aby zjednodušil používání Javassist knihovny při práci s více třídami. Jednoduše řečeno se jedná o AOP (aspect oriented programming) Framework nad knihovnou Javassist. Autorem jsou Vlad Skarzhevskyy a Michael Lifshits. Testovaná verze 2.0.3 byla publikována pod licencí GNU LGPL v roce 2008. Nástroj je možné použít přes příkazovou řádku nebo jako Maven plugin.

Formální popis spouštěcího příkazu:

```
java –cp jour-instrument-2.0.3.jar;javassist.jar
net.sf.jour.SignatureGenerator <--src oldJar> [-jars
oldJarDependencies] [--packages testedPackage] [--dst signatureFile]
[--level accessModifier]

java –cp jour-instrument-2.0.3.jar;javassist.jar
net.sf.jour.SignatureVerify <--src newJar> [-jars newJarDependencies]
[--signature signatureFile] [--level accessModifier]
```

Ukázka konkrétního CLI příkazu spuštění:

```
java -cp jour-instrument-2.0.3.jar;javassist.jar
net.sf.jour.SignatureGenerator --src testing-lib-v1-0.0.1.jar -jars
rt.jar --packages testing_lib --dst testLib1apiSignature.xml --level
private

java -cp jour-instrument-2.0.3.jar;javassist.jar
net.sf.jour.SignatureVerify --src testing-lib-v2-0.0.2.jar -jars
rt.jar --signature testLib1apiSignature.xml --level private >
jourReport.txt
```

Pokud chceme zjistit rozdíly mezi dvěma verzemi knihovny, musíme jedné nejprve vytvořit takzvaný *signature* soubor, což zde je v podstatě XML soubor popisující API. Při jeho vytváření musíme ke zdrojovému souboru přidat na vstupu také jeho závislosti. Proto je nutné přidat hnihovnu *rt.jar*, obsahující všechny základní třídy pro Java runtime enviroment. Také je nutné nastavit jaké balíky, s jejich podřízenými balíky, budou v *signature* souboru. Jour navíc umožňuje definovat, jaká úroveň přístupových modifikátorů elementů se budou testovat.

Příklad části ze *signature* souboru:

```
<class modifiers="public"
name="testing_lib.membersClazzMethodAdd.MembersClazzMethodAdd">
    <constructor modifiers="public"/>
</class>

<class modifiers="public"
name="testing_lib.membersClazzMethodDelete.MembersClazzMethodDelete">
    <constructor modifiers="public"/>
    <method modifiers="public" name="method1" return="void"/>
</class>
```

Následně musíme vygenerovaný *signature* soubor porovnat s druhou verzí testované knihovny.

Textový výstup je seznam zpráv popisující změny v knihovně, kde jedna zpráva odpovídá jednomu Java elementu, na kterém změna proběhla. Každá zpráva má následující strukturu:

<umístění>: *<popis změny>*

Příklad části textového výstupu:

```
testing_lib.membersClazzMethodAdd.MembersClazzMethodAdd number of
Methods, Extra method(s) [method1()] expected:<0> but was:<1>

testing_lib.membersClazzMethodDelete.MembersClazzMethodDelete.method1(
)V is Missing

testing_lib.membersClazzMethodDelete.MembersClazzMethodDelete number
of Methods expected:<1> but was:<0>
```

6.9 Java Compatibility Checker (JaCC) - nástroj katedry

JaCC [18] je nástroj na kontrolu kompatibility knihoven vyvíjený na katedře informatiky a výpočetní techniky na Západočeské universitě v Plzni. Testovaná byla verze 1.0.9 a lze ji použít jako Maven plugin, jako eclipse plugin nebo jako knihovnu přímo v kódu.

K tomu, aby bylo možné nástroj testovat, bylo zapotřebí vytvořit klient (Jar), který je teprve možné používat z příkazové řádky. Textový výstup, viz příklad níže, je seznam zpráv popisující změny v knihovně, kde jedna zpráva odpovídá jednomu Java elementu, na kterém změna proběhla.

Příklad části textového výstupu:

```
Class: testing_lib.membersClazzMethodAdd.MembersClazzMethodAdd
(Classification: C2)
 Method: void method1() (Added) (Extension: source)

Class: testing_lib.membersClazzMethodDelete.MembersClazzMethodDelete
(Classification: C2)
 Method: void method1() (Not found) (Classification: M1) (Extension:
source Invocation: binary+source)
```

Nástroj se skládá z jádra nazývaného JaCC a sady klientských nástrojů pracujících s tímto jádrem. Mimo kontrolu zpětné kompatibility je možné nástroj použít pro testování kompozice klient – knihovna. Nástroj využívá takzvaného reverzního inženýrství, kdy z binární kódu knihovny získá modely v paměti a ty poté testuje.

6.10 Souhrn

Nástroj japi-checker je nutné nejdříve klonovat z git repozitáře a poté následně sestavit nástrojem Maven. U nástroje JAPICC je zapotřebí ve Windows nainstalovat Active Perl 5 nebo vyšší, aby jej šlo přeložit a spustit. U nástroje JaCC je zapotřebí vytvořit klient (Jar), který je teprve možné zavolat z příkazové řádky. Japitools je zase nutné stáhnout do Linuxu jako rozšiřující balík.

Pro lepší přehlednost je v následujících tabulkách (Tabulka 10, Tabulka 11) vypsaný souhrn základních informací, výstupních formátů a možností jakým způsobem lze jednotlivé nástroje použít. Detailnější popis jednotlivých nástrojů je uvedeny v předchozích kapitolách.

	Clirr	Japicmp	japi checker	JAPICC	Revapi
Základní info					
Autor	Lars Kühne	Martin Mois	William Bernardet	Andrey Ponomarenko	Lukas Krejci
licence	GNU LGPL	Apache 2.0	Apache 2.0	GNU GPL or LGPL	Apache 2.0
verze	0.6.0	0.7.2	0.2.1	1.5	0.4.2
release	27.9.2005	20.3.2016	3.10.2015	8.4.2016	30.3.2016
Výstup:					
txt	●	●	●	×	●
XML	●	●	×	×	×
HTML	●	●	×	●	×
Použití:					
CLI	●	●	●	●	●
Maven	●	●	●	×	●
Ant Task	●	×	●	×	●
java knihovna	×	●	×	×	×
eclipse plugin	×	×	×	×	×

Tabulka 10 - Souhrn nástrojů 1

	Sigtest	Japitools	Jour	JaCC
Základní info				
Autor	Oracle	Stuart Ballard	Vlad Skarzhevskyy	
licence	GNU GPLv2	GNU GPL	GNU LGPL	
verze	3.1	0.9.7	2.0.3	1.0.9
release	8.4.2016	13.11.2007	12.12.2008	
Výstup:				
txt	●	●	●	●
XML	×	×	×	×
HTML	×	×	×	×
Použití:				
CLI	●	●	●	×
Maven	●	×	●	●
Ant Task	●	×	×	×
java knihovna	×	×	×	●
eclipse plugin	×	×	×	●

Tabulka 11 - Souhrn nástrojů 2

7 Vyhodnocení kritérií

V této kapitole jsou vyhodnocená jednotlivá kritéria. Proces se řídí pravidly určenými v kapitole *Metodika*. Jsou zde uvedeny dvě podkapitoly popisující vyhodnocení šesti vybraných kritérií. V první podkapitole je vyhodnoceno hlavní kritérium, tedy *Odhalení nekompatibilit* a v druhé podkapitole jsou dále vyhodnoceny zbylá mimofunkční kritéria.

7.1 Odhalení nekompatibilit

Jak již bylo zmíněno dříve v předchozích kapitolách, porovnání nástrojů v rámci tohoto kritéria vychází z toho, jak jednotlivé nástroje obstojí při vyhodnocování změn provedených v testovací knihovně. Nejen, zda dokáží nalézt a vhodně vyhodnotit dané nekompatibilní testy, ale jestli dovedou také případně správně vyhodnotit nalezené kompatibilní testy. V případě nástrojů vypisujících všechny testy vypsat, že změna je binárně kompatibilní nebo vypsat, že neproběhla změna. U nástrojů, které vypisují pouze nekompatibilní testy poté kompatibilní změnu nevypisovat vůbec.

Porovnávané nástroje si v rámci jednotlivých kategorií často vedou velmi podobně, viz Tabulka 12. Například v kategorii *Datové typy* všechny nástroje správně vyhodnotili veškeré připravené testy.

Dále je možné z tabulky vysledovat, že některé nástroje nepodporuji vyhledávání změn v celé šíři dané kategorie. Příkladem zde mohou být *Generika*, které nepodporují čtyři resp. skoro pět z devíti testovaných nástrojů. V nástrojích tedy není implementována detekce změn v kategorii.

Ze získaných výsledků nástrojů v jednotlivých kategoriích je vidět, že nejlépe si s připravenými testovacími daty poradil nástroj Sigtest od Oraclu. Podobně, s pár resty, dopadl i nástroj Japitools a Revapi. Čtvrtým v pořadí je JaCC (nástroj katedry informatiky na ZČU).

	Sigtest	Japitools	Revapi	JaCC	JAPICC	Japicmp	Jour	japi checker	Clirr
Datové typy [%]	100	100	100	100	100	100	100	100	100
Přístupové mod. [%]	100	100	100	100	94	100	78	100	100
Ostatní mod. [%]	97	90	87	97	100	97	90	83	90
Členové [%]	100	100	100	100	96	100	89	93	100
Generika [%]	100	100	100	32	6	0	0	0	0
Dědičnost [%]	100	100	94	100	94	88	81	81	94
Výjimky [%]	100	100	100	0	100	81	100	75	0
Ostatní [%]	100	100	100	100	83	100	100	100	100
Celkem [%]	**99,60**	**98,80**	**98,01**	**69,32**	**65,34**	**62,55**	**59,76**	**59,36**	**56,97**

Tabulka 12 - Vyhodnocení nástrojů v %

Nástroj Sigtest ze všech připravených testů nedokázal odhalit nebo špatným způsobem vyhodnotil pouhý jeden test. Změnu modifikátoru tzv. effectively final třídy (má `private` konstruktor) na `final`. Tímto testem lze zjistit, zda nástroj při vyhodnocování změny elementu bere ohled, jak jsou definované jeho ostatní elementy.

Dále je vidět, že nástroj vyvíjený na katedře nemá implementovanou podporu pro změny týkající se kategorie *Výjimky* a v *Generikach* dokázal vyhodnotit pouze testy týkající se speciálního typu Wildcards.

Následující tabulka zobrazuje detailnější pohled na úspěšnost nástrojů v kategoriích. Je možné získat přehled, jakou část všech vytvořených testů jednotlivé kategorie tvořili. Například generika tvořila zhruba 36% všech vytvořených testů.

	Sigtest	Japitools	Revapi	JaCC	JAPICC	Japicmp	Jour	japi checker	Clirr	Celkem
Datové typy	49	49	49	49	49	49	49	49	49	**49**
Přístupové mod.	18	18	18	18	17	18	14	18	18	**18**
Ostatní mod.	29	27	26	29	30	29	27	25	27	**30**
Členové	28	28	28	28	27	28	25	26	28	**28**
Generika	88	88	88	28	5	0	0	0	0	**88**
Dědičnost	16	16	15	16	15	14	13	13	15	**16**
Výjimky	16	16	16	0	16	13	16	12	0	**16**
Ostatní	6	6	6	6	5	6	6	6	6	**6**
Celkem nalezeno	**250**	**248**	**246**	**174**	**164**	**157**	**150**	**149**	**143**	**251**

Tabulka 13 - Vyhodnocení nástrojů detail

Na následující tabulce lze vidět, jak si nástroje poradili pouze se změnami, ve kterých došlo ke zdrojové či binární nekompatibilitě. Lze si všimnout, že nástroj Japicmp si zde vede lépe než nástroj JAPICC, na rozdíl od vyhodnocení všech změn. Zbylé nástroje jsou však ve stejném pořadí.

	Sigtest	Japitools	Revapi	JaCC	Japicmp	JAPICC	Jour	japi checker	Clirr	Celkem
Datové typy	47	47	47	47	47	47	47	47	47	47
Přístupové mod.	6	6	6	6	6	5	5	6	6	6
Ostatní mod.	11	9	8	11	11	11	10	7	8	11
Členové	19	19	19	19	19	18	16	17	19	19
Generika	49	49	49	17	0	3	0	0	0	49
Dědičnost	5	5	5	5	5	4	5	5	5	5
Výjimky	8	8	8	0	8	8	8	8	0	8
Ostatní	4	4	4	4	4	3	4	4	4	4
Celkem nalezeno	**149**	**147**	**146**	**109**	**100**	**99**	**95**	**94**	**89**	**149**

Tabulka 14 – Porovnání nástrojů (zdrojová či binární nekompatibilita)

Z této tabulky však není poznat, jak si jednotlivé nástroje vedou v rámci každé z kompatibilit. Toho si můžeme všimnout v následujících tabulkách.

	Sigtest	Japitools	Revapi	JaCC	Japicmp	JAPICC	Jour	japi checker	Clirr	Celkem
Datové typy	29	29	29	29	29	29	29	29	29	29
Přístupové mod.	6	6	6	6	6	5	6	6	6	6
Ostatní mod.	9	7	6	9	9	9	8	6	7	9
Členové	19	19	19	19	19	18	19	17	19	19
Generika	49	49	49	17	0	3	0	0	0	49
Dědičnost	5	5	5	5	5	4	5	5	5	5
Výjimky	8	8	8	0	8	8	8	8	0	8
Ostatní	4	4	4	4	4	3	4	4	4	4
Celkem nalezeno	**129**	**127**	**126**	**89**	**80**	**79**	**79**	**75**	**70**	**129**

Tabulka 15 - Porovnání nastrojů (zdrojová nekompatibilita)

Při porovnání nástrojů z pohledu odhalení zdrojové nekompatibility se pořadí od tabulky 14 nezměnilo. V pohledu odhalení co se týče binární nekompatibility v následující tabulce, se však již pořadí mění. Nástroj Revapi, který byl doposud třetí, má v odhalování binárních nekompatibilit drobné nedostatky.

	Sigtest	Japicmp	JaCC	Jour	Japitools	Clirr	Revapi	japi checker	JAPICC	Celkem
Datové typy	46	46	46	46	46	46	46	46	46	46
Přístupové mod.	6	6	6	6	6	6	6	6	5	6
Ostatní mod.	11	11	11	10	9	8	8	7	11	11
Členové	14	14	14	14	14	14	14	14	13	14
Generika	0	0	0	0	0	0	0	0	0	0
Dědičnost	5	5	5	5	5	5	5	5	4	5
Výjimky	0	0	0	0	0	0	0	0	0	0
Ostatní	4	4	4	4	4	4	4	4	3	4
Celkem nalezeno	**86**	**86**	**86**	**85**	**84**	**83**	**83**	**82**	**82**	**86**

Tabulka 16 - Porovnání nástrojů (binární nekompatibilita)

Z předchozích tabulek je vidět, že nástroje dokáží lépe odhalovat binární nekompatibility. Prohlédnout si, jak nástroje uspěli s jednotlivými testy v rámci definovaných kategorií, je možné v tabulkách v Příloze A.

7.2 Mimofunkční kritéria

Zde budou nejprve vyhodnoceny jednotlivé mimofunkční kritéria. Celkové výsledky jsou vidět v tabulce na konci této kapitoly. Tabulka s detaily, kde je vidět proč daný nástroj dostal určité množství bodů je v Příloze A.10.

Snadnost použití

Prvním hodnoceným mimofunkčním kritériem je snadnost použití. Jedná se o možnosti, jakými lze nástroj použít. Téměř všechny nástroje je možné použít přes příkazovou řádku. Jedinou výjimkou je zde nástroj JaCC, u kterého bylo potřeba klienta spouštěného přes příkazovou řádku vytvořit. Navíc od ostatních jej můžeme použít jako Eclipse plugin. Další možností, u nástrojů, které lze spustit více jak jedním způsobem je použití jako Maven plugin. Poslední možností je poté pro nástroje Japicmp a JaCC použití jako knihovna přímo v kódu, je tedy nutné vytvořit klienta. U ostatních nástrojů je poslední možností použití jako Ant task.

Výstupní formát

Téměř všechny nástroje podporují vygenerování výstupu v textové podobě. Výjimkou je nástroj JAPICC, který umožňuje výstup pouze ve formátu HTML. Nástroje Clirr a Japicmp navíc mimo příkazové řádky podporují výstup ve formátu XML a HTML.

Přehlednost výstupu

Ať už se jedná o výstup na konzoli nebo přímo do souboru, vždy je nutné, aby výstup obsahoval určité prvky. Prvním takovým je prvek umístění, který je podle očekávání u všech výstupních souborů nástrojů skoro identický. U některých nástrojů je například vypsaný pouze do detailu třídy a změna týkající se například proměnné je popsána až v textovém popisu. Zatímco u jiných je měněná proměnná již v prvku umístění.

Dalším prvkem je textový popis nalezené chyby. U nástrojů, které ve výstupu mají tento prvek, je okamžitě po přečtení jasné, co jsme v provedené změně udělali za chybu. U nástrojů Japicmp, Sigtest a JaCC toto musíme pochopit z více řádků, které se týkají jedné změny.

Dále se hodnotil výskyt prvku závažnost. Jedná se o to, zda nástroj u jednotlivých změn definuje, jestli se jedná například o Warning nebo přímo Error, nebo nějakým jiným způsobem určuje závažnost. Tento prvek nevyhodnocují nástroje Jour, Revapi, Japicmp, Sigtest a JaCC.

Posledním prvkem je rozpoznání zda se jedná o binární či zdrojovou nekompatibilitu. Tento prvek je velmi dobře definovaný u nástroje Revapi. Přímo u zprávy o změně je vypsáno, zda porušuje binární či zdrojovou kompatibilitu. Nástroj Japicmp má tento prvek taky pěkně zpracovaný. K vyhodnocení, zda se jedná o změnu, přidání, odebrání nebo jestli element nebyl změněný, přidá navíc znak “!“ v případě, že se jedná o binární nekompatibilitu, nebo znak “*” pokud se jedná o zdrojovou nekompatibilitu. U některých nástrojů je také možné definovat, zda výpis má obsahovat změny binárně nebo zdrojově nekompatibilní, přepínačem při jejich spuštění. Jedním z nich je například Sigtest.

Nástroje Japitools a JAPICC umožňují v tomto kritériu větší možnosti. Před výpisem jednotlivých změn mají souhrn (např. kolik bylo přidaných metod, kolik jich bylo změněno, atd.).

Integrovatelnost do vývojového cyklu

Jak již bylo zmíněno v kritériu *Snadnost použití*, nástroje, které mají více, jak jednu možnost použití je možné použít také jako Maven plugin a tím je v podstatě integrovat do vývojového cyklu ve formě pluginu. Z toho vyplývá, že u všech nástrojů je jeden bod za tuto možnost. Druhý bod u některých nástrojů je poté za možnost použití jako Ant task.

Uživatelská dokumentace

Zde je hodnoceno, zda nástroje mají u každé možnosti použití uveden příklad použití. Jediné nástroje, které toto kritérium porušují, jsou Clirr a japi checker. U nástroje Clirr chybí příklad pro použití jako Maven plugin. U nástroje japi checker chybí příklad pro příkazovou řádku a Ant task. Dá se říci, že u japi checkeru je z celkového pohledu dokumentace nedostatečná.

	Clirr	Japicmp	Revapi	Sigtest	japi checker	JaCC	JAPICC	Jour	Japitools
Snadnost použití	3	3	3	3	3	3	1	2	1
Výstupní formát	3	3	1	1	1	1	1	1	1
Přehlednost výstupu	3	2	3	2	3	2	4	2	3
Integrovatelnost do ž.c.	2	1	2	2	2	1	0	1	0
Uživ. dokumentace	-1	0	0	0	-2	0	0	0	0
Celkem	**10**	**9**	**9**	**8**	**7**	**7**	**6**	**6**	**5**

Tabulka 17 - Mimofunkční kritéria

7.3 Celkové vyhodnocení

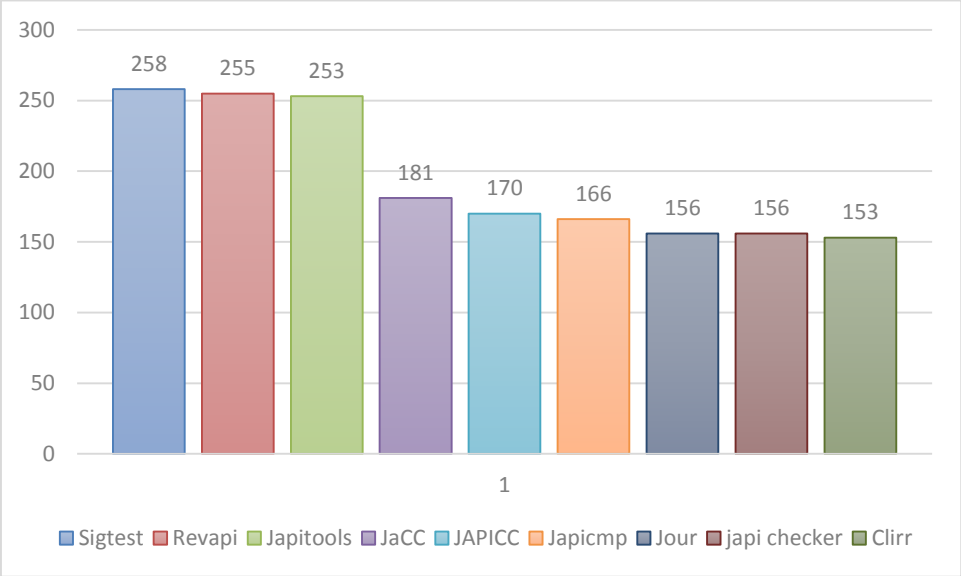
Po konečném součtu bodů vychází nejlépe nástroj Sigtest s celkovým počtem 258 bodů. Sigtest byl po vyhodnocení podle kritéria *Odhalení nekompatibilit* již první.

Po přičtení 8 bodů z mimofunkčních kritérií je tedy tento nástroj vyhodnocen jako nejlepší.

Podobného počtu bodů dosáhli i nástroje Revapi a Japitools, kde druhý jmenovaný nástroj vychází, co se týče odhalení nekompatibilit lépe než nástroj Revapi. Jelikož je možné jej použít pouze z příkazové řádky, dostal tedy ve vyhodnocení mimofunkčních kritérií pouze 5 bodů.

Ostatní nástroje se poté pohybují s celkovým počtem kolem 160 bodů. Takto velká ztráta je způsobena tím, že nemají implementovanou podporu pro vyhledávání změn v některé z testovaných kategorií. Nejčastěji touto kategorií jsou generika a v některých případech i výjimky. Nástroj katedry informatiky na ZČU (JaCC) je podle celkového počtu bodů 181 na čtvrté pozici.

Jako nejhorší byl vyhodnocen nástroj Clirr se 153 body. Clirr nebyl od 27. 9. 2005 aktualizován. Nástroj nemá podporu pro kontrolu změn v kategorii generika a výjimky. Nicméně co se týče bodů získaných z mimofunkčních charakteristik vychází jako nejlepší.



Graf 1 - Celkové vyhodnocení

7.4 Nástroji nenalezené změny

Zde jsou u jednotlivých nástrojů vypsány pouze nejzásadnější nenalezené změny. Seznam všech podporovaných změn by byl velmi dlouhý, je proto uveden v tabulkách v Příloze A. Pokud některý z nástrojů nedetekuje v dané kategorii všechny připravené testy, je ve výpisu uvedena celá kategorie. Nástroje jsou seřazeny podle kapitoly 5.

U nástroje Clirr je nedostatkem, že nepodporuje kategorií generik a výjimek. Dalším nedostatkem je poté, že neodhalil změny v kategorii ostatní modifikátory. Konkrétně změny metody, které jsou přidání modifikátoru `abstract` a přidání nebo odstranění modifikátoru `static`.

Nástroj Japicmp si vedl velice dobře, co se týče jednotlivých kategorií. Nevýhodou jsou však celá generika a pár testů v kategorii výjimky. U výjimek konkrétně přidání, generalizace a specializace unchecked výjimky.

Japi checker si již tak dobře nevedl. Nepodporuje celou kategorií generika. V ostatních modifikátorech nedokázal odhalit nejvíce testů ze všech nástrojů. Jedna ze změn je přidání modifikátoru `final` k proměnné. Zbylé se týkají přidání modifikátoru `abstract` a přidání či odstranění modifikátoru `static` u metody.

Téměř jako jediný poté měl pár nedostatků, co se týče kategorie členové. Zde nedokázal odhalit přidání `abstract` metody ani do rozhraní ani do abstraktní třídy. V kategorii *výjimky* nedovedl odhalit změny týkající se unchecked exception. Jedinou, kterou z nich odhalil, byla změna mutation, kdy se jedná o převod z unchecked na checked exception.

Nástroj JAPICC v generikách dovedl nalézt pár změn, nicméně má nějaké nedostatky i v dalších kategoriích. V přístupových modifikátorech neodhalil snížení viditelnosti zahnízděného rozhraní ve třídě. V kategorii členové smazání zahnízděného rozhraní z jiného rozhraní. V kategorii ostatní poté neodhalil smazání rozhraní z balíku.

U nástroje Revapi je to poté celkem jednoduché, avšak je zde pár nedostatků, které jsou celkem překvapivé. Zásadní nedostatky jsou zde v kategorii ostatní modifikátory. Prvním, který nástroj nenalezl, je přidání `abstract` k elementu metoda. Dalšími jsou přidání či odstranění modifikátoru `static` v případě zahnízděné třídy.

U nástroje Sigtest se jedná pouze o zákeřně postavený test. Kdy se přidal modifikátor `final` ke třídě s privátním konstruktorem.

Japitools poté nenalezl změny přidání či odstranění static modifikátoru u zahnízděné třídy.

Nástroj Jour nedovedl odhalit odstranění static modifikátoru u zahnízděné třídy a poté Generika. Dále některé z testů vyhazovali výjimku, proto je bylo nutné při testování smazat. V tabulkách v příloze jsou poté označeny křížkem. Testy, které byly smazány a jsou zařazené v hodnocení u jiných nástrojů:

- Přístupové modifikátory
  - Snížení viditelnosti zahnízděného rozhraní ve třídě
  - Zvýšení viditelnosti zahnízděného rozhraní ve třídě
  - Snížení viditelnosti zahnízděného rozhraní v rozhraní
  - Zvýšení viditelnosti zahnízděného rozhraní v rozhraní
- Ostatní modifikátory
  - Proměnná – `public transient` → `public`
  - Zahnízděná třída – `public static` → `public`
- Členové
  - Smazání defaultní metody z rozhraní
  - Smazání zahnízděného rozhraní ze třídy
  - Smazání zahnízděného rozhraní z rozhraní
- Dědičnost
  - Přepsání (Override) zděděné defaultní metody
  - Smazání (Override) zděděné defaultní metody z potomka

Nástroj JaCC vyvíjený na ZČU poté nedovedl odhalit Generika pouze základní část a celou kategorii výjimek.

8 Závěr

Výsledkem této práce je porovnání nástrojů pro kontrolu zpětné kompatibility Java knihoven. Porovnání bylo provedeno na základě správnosti vyhodnocení testů z testovacích dat a vyhodnocení mimofunkčních charakteristik. Testovací data simulují inkrementální vývoj s nejrůznějšími změnami v knihovně porušující i zachovávající zpětnou kompatibilitu, neboť je důležité, aby nástroje odhalili porušení, ale zároveň neoznačili zachovávající změnu. U jednotlivých změn tedy bylo zapotřebí určit, jaký mají vliv na zdrojovou či binární kompatibilitu. Na základě zjištění této vlastnosti změn bylo poté možné vyhodnotit, jakým způsobem se nástroj při vyhodnocování má zachovat. Nástroje byly bodově hodnocené podle předem definovaných kritérií, které jsou určené v kapitole *Metodika.*

V celkovém porovnání jednotlivých nástrojů nejlépe skončil nástroj Sigtest, který špatným způsobem vyhodnotil pouze jediný z připravených testů. V hodnocení mimofunkčních kritérií skončil na čtvrté pozici. V součtu dosažených bodů jsou za ním v těsné blízkosti nástroje Revapi a Japitools, které získali jen o pár bodů méně. Čtvrtým je poté nástroj JaCC vyvíjený na ZČU, který však již na přední trojici ztrácí více bodů. Ostatní nástroje jsou téměř vyrovnané a dopadly v následujícím pořadí: JAPICC, Japicmp, Jour, Japi checker. Na poslední pozici po vyhodnocení je nástroj Clirr, který ve vyhodnocení testů dopadl nejhůře, avšak v hodnocení mimofunkčních charakteristik dopadl nejlépe.

V testovacích datech jsou testy ve své podstatě převážně základní, jako je přidání nebo odstranění některého z modifikátorů u jednotlivých elementů, odstranění proměnné, metody nebo odstranění třídy či rozhraní. Dalším rozšířením testů může být vytvoření kombinace těchto již vytvořených testů. Dále pak doplnění, pokud se Java rozšíří o některé vlastnosti, jako tomu bylo ve verzi 1.5 s *Generiky* či ve verzi 1.8 s default metodou u rozhraní, tak jejich doplnění.

Přehled zkratek

API	*Application Programming Interface*
ABI	*Application Binary Interface*
LGPL	*Lesser General Public Licence*
XML	*Extensible Markup Language*
HTML	*HyperText Markup Language*
AOP	*Aspect Oriented Programming*
JAR	*Java Archive*
DDR	*Discoidin domain receptor*
SSD	*Solid state drive*

Literatura

[1] Jim de Riviers. - *Evolving Java - based APIs.*[online]. 2016, [cit. 2016-04-04]. <http://wiki.eclipse.org/Evolving_Java-based_APIs>

[2] *The Java Language Specification - Chapter 13. Binary Compatibility*. [online]. 2016, [cit. 2016-04-04]. <http://docs.oracle.com/javase/specs/jls/se7/html/jls-13.html>

[3] *Kinds of Compatibility*: *Source, Binary, and Behavioral*. [online]. 2016, [cit. 2016-04-04]. <https://blogs.oracle.com/darcy/entry/kinds_of_compatibility>

[5] *Apache license 2.0 - License 2.0* [online]. 2016, [cit. 2016-04-04]. <http://www.apache.org/licenses/LICENSE-2.0.html>

[6] LINDHOLM, T. - YELLIN, F. - BRACHA, G. - BUCKLEY, A. *The Java Virtual Machine Specification*, Java SE 8 Edition [online], 2015, [cit. 2016-04-28]. <https://docs.oracle.com/javase/specs/jvms/se8/html/index.html>

[7] *SDLC Tutorial* [online]. 2016, [cit. 2016-04-28]. <http://www.tutorialspoint.com/sdlc/index.htm>

[8] *Apis and Abis* [online]. 2016, [cit. 2016-04-29]. <https://www.safaribooksonline.com/library/view/linux-system-programming/0596009585/ch01s02.html>

[9] *The Java Tutorials* [online]. 2015, [cit. 2016-04-30]. <https://docs.oracle.com/javase/tutorial/index.html>

[10] Lars Kühne - *Clirr* [online]. 2005, [cit. 2016-05-02]. <http://clirr.sourceforge.net/index.html>

[11] *Japicmp* [online]. 2016, [cit. 2016-05-02]. <https://siom79.github.io/japicmp/index.html>

[12] *japi checker* [online]. 2016, [cit. 2016-05-02]. <https://github.com/williambernardet/japi-checker>

[13] ISPRAS - *Java API Compliance Checker* [online]. 2016, [cit. 2016-05-02]. <http://ispras.linuxbase.org/index.php/Java_API_Compliance_Checker>

[14] Lukas Krejci - *Revapi* [online]. 2016, [cit. 2016-05-02]. <http://revapi.org/>

[15] Oracle – *Sigtest* [online]. 2016, [cit. 2016-05-02]. <https://wiki.openjdk.java.net/display/CodeTools/SigTest>

[16] Java API compatibility testing tools – *Japitools* [online]. 2006, [cit. 2016-05-02]. <http://www.sab39.org/japi/>

[17] *Jour* [online]. 2008, [cit. 2016-05-02]. <http://www.sab39.org/japi/>

[18] JEZEL, K., HOLZ, L., DANEK, J. *Preventing Composition Problems in Modular Java Applications*, 2015.

[19] JEZEK, K., BRADA, P., DANEK, J., Dietrich, J. *Understanding the Types of Software Compatibility and their Impact in Java, (Nepublikováno)*

Příloha A: Výsledky nástrojů

A.1 Legenda tabulek

V následujících kapitolách přílohy se nachází tabulky s výsledky, jak jednotlivé vytvořené testy dopadli při testování zdrojové či binární kompatibility. Také je v nich vidět, jak jednotlivé nástroje vyhodnotili dané testy. Nástroje musí vypsat jak nekompatibilní změny, tak nesmí označit změnu kompatibilní jako chybu.

U kompatibility:

- ● - změna JE kompatibilní
- × - změna NENÍ kompatibilní

Výsledky nástrojů:

1 – nástroj danou změnu vyhodnotil dobře (nekompatibilní testy vypsal, v případě kompatibilních nevypsal)

0 – nástroj danou změnu vyhodnotil špatně (nekompatibilní testy nevypsal, v případě kompatibilních vypsal)

x – nástroj při daném testu vyhazuje výjimku (nutno odstranit z knihovny při testování daného nástroje)

A.2 **Datové typy**

	Zdrojová	Binarní	Clirr	Japicmp	japi checker	JAPICC	Revapi	Sigtest	Japitools	Jour	JaCC
dataTypeClazzConstructorParamBoxing	●	×	1	1	1	1	1	1	1	1	1
dataTypeClazzConstructorParamGeneralization	●	×	1	1	1	1	1	1	1	1	1
dataTypeClazzConstructorParamMutation	×	×	1	1	1	1	1	1	1	1	1
dataTypeClazzConstructorParamNarrowing	×	×	1	1	1	1	1	1	1	1	1
dataTypeClazzConstructorParamSpecialization	×	×	1	1	1	1	1	1	1	1	1
dataTypeClazzConstructorParamUnboxing	●	×	1	1	1	1	1	1	1	1	1
dataTypeClazzConstructorParamWidening	●	×	1	1	1	1	1	1	1	1	1
dataTypeClazzFieldBoxing	●	×	1	1	1	1	1	1	1	1	1
dataTypeClazzFieldGeneralization	×	×	1	1	1	1	1	1	1	1	1
dataTypeClazzFieldMutation	×	×	1	1	1	1	1	1	1	1	1
dataTypeClazzFieldNarrowing	●	×	1	1	1	1	1	1	1	1	1
dataTypeClazzFieldSpecialization	●	×	1	1	1	1	1	1	1	1	1
dataTypeClazzFieldUnboxing	●	×	1	1	1	1	1	1	1	1	1
dataTypeClazzFieldWidening	×	×	1	1	1	1	1	1	1	1	1
dataTypeClazzMethodParamBoxing	●	×	1	1	1	1	1	1	1	1	1
dataTypeClazzMethodParamGeneralization	●	×	1	1	1	1	1	1	1	1	1
dataTypeClazzMethodParamMutation	×	×	1	1	1	1	1	1	1	1	1
dataTypeClazzMethodParamNarrowing	×	×	1	1	1	1	1	1	1	1	1
dataTypeClazzMethodParamSpecialization	×	×	1	1	1	1	1	1	1	1	1
dataTypeClazzMethodParamUnboxing	●	×	1	1	1	1	1	1	1	1	1
dataTypeClazzMethodParamWidening	●	×	1	1	1	1	1	1	1	1	1
dataTypeClazzMethodReturnTypeBoxing	●	×	1	1	1	1	1	1	1	1	1
dataTypeClazzMethodReturnTypeGeneralization	×	×	1	1	1	1	1	1	1	1	1
dataTypeClazzMethodReturnTypeMutation	×	×	1	1	1	1	1	1	1	1	1
dataTypeClazzMethodReturnTypeNarrowing	●	×	1	1	1	1	1	1	1	1	1
dataTypeClazzMethodReturnTypeSpecialization	●	×	1	1	1	1	1	1	1	1	1
dataTypeClazzMethodReturnTypeUnboxing	●	×	1	1	1	1	1	1	1	1	1
dataTypeClazzMethodReturnTypeWidening	×	×	1	1	1	1	1	1	1	1	1

	Zdrojová	Binarní	Clirr	Japicmp	japi checker	JAPICC	Revapi	Sigtest	Japitools	Jour	JaCC
dataTypeIfazeConstantBoxing	●	●	1	1	1	1	1	1	1	1	1
dataTypeIfazeConstantGeneralization	×	×	1	1	1	1	1	1	1	1	1
dataTypeIfazeConstantMutation	×	×	1	1	1	1	1	1	1	1	1
dataTypeIfazeConstantNarrowing	●	●	1	1	1	1	1	1	1	1	1
dataTypeIfazeConstantSpecialization	●	×	1	1	1	1	1	1	1	1	1
dataTypeIfazeConstantUnboxing	●	×	1	1	1	1	1	1	1	1	1
dataTypeIfazeConstantWidening	×	●	1	1	1	1	1	1	1	1	1
dataTypeIfazeMethodParamBoxing	×	×	1	1	1	1	1	1	1	1	1
dataTypeIfazeMethodParamGeneralization	×	×	1	1	1	1	1	1	1	1	1
dataTypeIfazeMethodParamMutation	×	×	1	1	1	1	1	1	1	1	1
dataTypeIfazeMethodParamNarrowing	×	×	1	1	1	1	1	1	1	1	1
dataTypeIfazeMethodParamSpecialization	×	×	1	1	1	1	1	1	1	1	1
dataTypeIfazeMethodParamUnboxing	×	×	1	1	1	1	1	1	1	1	1
dataTypeIfazeMethodParamWidening	×	×	1	1	1	1	1	1	1	1	1
dataTypeIfazeMethodReturnTypeBoxing	×	×	1	1	1	1	1	1	1	1	1
dataTypeIfazeMethodReturnTypeGeneralization	×	×	1	1	1	1	1	1	1	1	1
dataTypeIfazeMethodReturnTypeMutation	×	×	1	1	1	1	1	1	1	1	1
dataTypeIfazeMethodReturnTypeNarrowing	×	×	1	1	1	1	1	1	1	1	1
dataTypeIfazeMethodReturnTypeSpecialization	×	×	1	1	1	1	1	1	1	1	1
dataTypeIfazeMethodReturnTypeUnboxing	×	×	1	1	1	1	1	1	1	1	1
dataTypeIfazeMethodReturnTypeWidening	×	×	1	1	1	1	1	1	1	1	1

A.3 Přístupové modifikátory

	Zdrojová	Binární	Clirr	Japicmp	japi checker	JAPICC	Revapi	Sigtest	Japitools	Jour	JaCC
accessModifierClazzAccessDecrease	×	×	1	1	1	1	1	1	1	1	1
accessModifierClazzAccessIncrease	●	●	1	1	1	1	1	1	1	1	1
accessModifierClazzConstructorAccessDecrease	×	×	1	1	1	1	1	1	1	1	1
accessModifierClazzConstructorAccessIncrease	●	●	1	1	1	1	1	1	1	1	1
accessModifierClazzFieldAccessDecrease	×	×	1	1	1	1	1	1	1	1	1
accessModifierClazzFieldAccessIncrease	●	●	1	1	1	1	1	1	1	1	1
accessModifierClazzMethodAccessDecrease	×	×	1	1	1	1	1	1	1	1	1
accessModifierClazzMethodAccessIncrease	●	●	1	1	1	1	1	1	1	1	1
accessModifierClazzNestedClazzAccessDecrease	×	×	1	1	1	1	1	1	1	1	1
accessModifierClazzNestedClazzAccessIncrease	●	●	1	1	1	1	1	1	1	1	1
accessModifierClazzNestedIfazeAccessDecrease	×	×	1	1	1	0	1	1	1	x	1
accessModifierClazzNestedIfazeAccessIncrease	●	●	1	1	1	1	1	1	1	x	1
accessModifierIfazeFieldAccessDecrease	●	●	1	1	1	1	1	1	1	1	1
accessModifierIfazeFieldAccessIncrease	●	●	1	1	1	1	1	1	1	1	1
accessModifierIfazeMethodAccessDecrease	●	●	1	1	1	1	1	1	1	1	1
accessModifierIfazeMethodAccessIncrease	●	●	1	1	1	1	1	1	1	1	1
accessModifierIfazeNestedIfazeAccessDecrease	●	●	1	1	1	1	1	1	1	x	1
accessModifierIfazeNestedIfazeAccessIncrease	●	●	1	1	1	1	1	1	1	x	1

A.4 Ostatní modifikátory

V názvu jednotlivých testů byl vynechaný začátek `modifier`.

	Zdrojová	Binární	Clirr	Japicmp	japi checker	JAPICC	Revapi	Sigtest	Japitools	Jour	JaCC
modifierClazzAbstractToNonAbstract	●	●	1	1	1	1	1	1	1	1	1
modifierClazzEffectivelyFinalToFinal	●	●	1	0	0	1	0	0	0	0	0
modifierClazzFinalToEffectivelyFinal	●	●	1	1	1	1	1	1	1	1	1
modifierClazzFinalToNonFinal	●	●	1	1	1	1	1	1	1	1	1
modifierClazzNonAbstractToAbstract	×	×	1	1	1	1	1	1	1	1	1
modifierClazzNonFinalToFinal	×	×	1	1	1	1	1	1	1	1	1
modifierClazzNonStrictfpToStrictfp	●	●	1	1	1	1	1	1	1	1	1
modifierClazzStrictfpToNonStrictfp	●	●	1	1	1	1	1	1	1	1	1
modifierFieldFinalToNonFinal	●	●	1	1	1	1	1	1	1	1	1
modifierFieldNonFinalToFinal	×	×	1	1	0	1	1	1	1	1	1
modifierFieldNonStaticToStatic	●	×	1	1	1	1	1	1	1	1	1
modifierFieldNonTransientToTransient	●	●	1	1	1	1	1	1	1	1	1
modifierFieldNonVolatileToVolatile	●	●	1	1	1	1	1	1	1	1	1
modifierFieldStaticToNonStatic	×	×	1	1	1	1	1	1	1	1	1
modifierFieldTransientToNonTransient	●	●	1	1	1	1	1	1	1	x	1
modifierFieldVolatileToNonVolatile	●	●	1	1	1	1	1	1	1	1	1
modifierMethodAbstractToNonAbstract	●	●	1	1	1	1	1	1	1	1	1
modifierMethodFinalToNonFinal	●	●	1	1	1	1	1	1	1	1	1
modifierMethodNativeToNonNative	●	●	1	1	1	1	1	1	1	1	1
modifierMethodNonAbstractToAbstract	×	×	0	1	0	1	0	1	1	1	1
modifierMethodNonFinalToFinal	×	×	1	1	1	1	1	1	1	1	1
modifierMethodNonNativeToNative	●	●	1	1	1	1	1	1	1	1	1
modifierMethodNonStaticToStatic	●	×	0	1	0	1	1	1	1	1	1
modifierMethodNonStrictfpToStrictfp	●	●	1	1	1	1	1	1	1	1	1
modifierMethodNonSynchronizedToSynchronized	●	●	1	1	1	1	1	1	1	1	1
modifierMethodStaticToNonStatic	×	×	0	1	0	1	1	1	1	1	1
modifierMethodStrictfpToNonStrictfp	●	●	1	1	1	1	1	1	1	1	1
modifierMethodSynchronizedToNonSynchronized	●	●	1	1	1	1	1	1	1	1	1
modifierNestedClazzNonStaticToStatic	×	×	1	1	1	1	0	1	0	1	1
modifierNestedClazzStaticToNonStatic	×	×	1	1	1	1	0	1	0	0	1

A.5 Členové

	Zdrojová	Binární	Clirr	Japicmp	japi checker	JAPICC	Revapi	Sigtest	Japitools	Jour	JaCC
membersClazzConstructorAdd	●	●	1	1	1	1	1	1	1	1	1
membersClazzConstructorDelete	×	×	1	1	1	1	1	1	1	1	1
membersClazzConstructorParamAdd	×	×	1	1	1	1	1	1	1	1	1
membersClazzConstructorParamDelete	×	×	1	1	1	1	1	1	1	1	1
membersClazzFieldAdd	●	●	1	1	1	1	1	1	1	1	1
membersClazzFieldConstantAdd	●	●	1	1	1	1	1	1	1	1	1
membersClazzFieldConstantDelete	×	×	1	1	1	1	1	1	1	1	1
membersClazzFieldDelete	×	×	1	1	1	1	1	1	1	1	1
membersClazzMethodAbstractAdd	×	●	1	1	0	1	1	1	1	1	1
membersClazzMethodAbstractDelete	×	●	1	1	1	1	1	1	1	1	1
membersClazzMethodAdd	●	●	1	1	1	1	1	1	1	1	1
membersClazzMethodDelete	×	×	1	1	1	1	1	1	1	1	1
membersClazzMethodParamAdd	×	×	1	1	1	1	1	1	1	1	1
membersClazzMethodParamDelete	×	×	1	1	1	1	1	1	1	1	1
membersClazzNestedClazzAdd	●	●	1	1	1	1	1	1	1	1	1
membersClazzNestedClazzDelete	×	×	1	1	1	1	1	1	1	1	1
membersClazzNestedIfazeAdd	●	●	1	1	1	1	1	1	1	1	1
membersClazzNestedIfazeDelete	×	×	1	1	1	1	1	1	1	x	1
membersIfazeConstantAdd	●	●	1	1	1	1	1	1	1	1	1
membersIfazeConstantDelete	×	×	1	1	1	1	1	1	1	1	1
membersIfazeMethodAdd	×	●	1	1	0	1	1	1	1	1	1
membersIfazeMethodDefaultAdd	●	●	1	1	1	1	1	1	1	1	1
membersIfazeMethodDefaultDelete	×	×	1	1	1	1	1	1	1	x	1
membersIfazeMethodDelete	×	×	1	1	1	1	1	1	1	1	1
membersIfazeMethodParamAdd	×	●	1	1	1	1	1	1	1	1	1
membersIfazeMethodParamDelete	×	●	1	1	1	1	1	1	1	1	1
membersIfazeNestedIfazeAdd	●	●	1	1	1	1	1	1	1	1	1
membersIfazeNestedIfazeDelete	×	×	1	1	1	0	1	1	1	x	1

A.6 Dědičnost

V názvu jednotlivých testů byl vynechaný začátek `inheritance`.

	Zdrojová	Binarní	Clirr	Japicmp	japi checker	JAPICC	Revapi	Sigtest	Japitools	Jour	JaCC
inheritanceClazzContractSuperClassSet	×	×	1	1	1	1	1	1	1	1	1
inheritanceClazzExpandSuperClassSet	●	●	1	0	0	1	1	1	1	0	1
inheritanceClazzMethodMovedFromSuperClass	×	×	1	1	1	1	1	1	1	1	1
inheritanceClazzMethodMovedToSuperClass	●	●	1	1	0	1	1	1	1	1	1
inheritanceClazzMethodOverrideAdd	●	●	1	1	1	1	1	1	1	1	1
inheritanceClazzMethodOverrideDelete	●	●	1	1	1	1	1	1	1	1	1
inheritanceClazzStartInherite	●	●	1	1	1	1	1	1	1	1	1
inheritanceClazzStopInherite	×	×	1	1	1	1	1	1	1	1	1
inheritanceIfazeContractSuperinterfaceSet	×	×	1	1	1	1	1	1	1	1	1
inheritanceIfazeDefaultMethodOverrideAdd	●	●	0	0	1	1	1	1	1	x	1
inheritanceIfazeDefaultMethodOverrideDelete	●	●	1	1	1	1	0	1	1	x	1
inheritanceIfazeExpandSuperinterfaceSet	●	●	1	1	1	1	1	1	1	1	1
inheritanceIfazeMethodMovedFromSuperInterface	●	●	1	1	1	1	1	1	1	1	1
inheritanceIfazeMethodMovedToSuperInterface	●	●	1	1	0	1	1	1	1	1	1
inheritanceIfazeStartInherite	●	●	1	1	1	1	1	1	1	1	1
inheritanceIfazeStopInherite	×	×	1	1	1	0	1	1	1	1	1

A.7 Generika

Tato tabulka představuje testy v třídě. V názvu jednotlivých testů byl vynechaný začátek `genericsClazz`.

	Zdrojová	Binarní	Clirr	Japicmp	japi checker	JAPICC	Revapi	Sigtest	Japitools	Jour	JaCC
ConstructorTypeAdd	●	●	0	0	0	0	1	1	1	0	0
ConstructorTypeAddSecond	×	●	0	0	0	0	1	1	1	0	0
ConstructorTypeBoundsAdd	×	●	0	0	0	0	1	1	1	0	0
ConstructorTypeBoundsAddSecond	×	●	0	0	0	0	1	1	1	0	0
ConstructorTypeBoundsDelete	●	●	0	0	0	0	1	1	1	0	0
ConstructorTypeBoundsDeleteSecond	●	●	0	0	0	0	1	1	1	0	0
ConstructorTypeBoundsGeneralization	●	●	0	0	0	0	1	1	1	0	0
ConstructorTypeBoundsMutation	×	●	0	0	0	0	1	1	1	0	0
ConstructorTypeBoundsSpecialization	×	●	0	0	0	0	1	1	1	0	0
ConstructorTypeDelete	●	●	0	0	0	0	1	1	1	0	0
ConstructorTypeDeleteSecond	×	●	0	0	0	0	1	1	1	0	0
ConstructorTypeSwap	●	●	0	0	0	0	1	1	1	0	0
MethodTypeAdd	●	●	0	0	0	0	1	1	1	0	0
MethodTypeAddSecond	×	●	0	0	0	0	1	1	1	0	0
MethodTypeBoundsAdd	×	●	0	0	0	0	1	1	1	0	0
MethodTypeBoundsAddSecond	×	●	0	0	0	0	1	1	1	0	0
MethodTypeBoundsDelete	●	●	0	0	0	0	1	1	1	0	0
MethodTypeBoundsDeleteSecond	●	●	0	0	0	0	1	1	1	0	0
MethodTypeBoundsGeneralization	●	●	0	0	0	0	1	1	1	0	0
MethodTypeBoundsMutation	×	●	0	0	0	0	1	1	1	0	0
MethodTypeBoundsSpecialization	×	●	0	0	0	0	1	1	1	0	0
MethodTypeDelete	●	●	0	0	0	0	1	1	1	0	0
MethodTypeDeleteSecond	×	●	0	0	0	0	1	1	1	0	0
MethodTypeSwap	●	●	0	0	0	0	1	1	1	0	0
TypeAdd	●	●	0	0	0	1	1	1	1	0	0
TypeAddSecond	×	●	0	0	0	1	1	1	1	0	0
TypeBoundsAdd	×	●	0	0	0	0	1	1	1	0	0
TypeBoundsAddSecond	×	●	0	0	0	0	1	1	1	0	0
TypeBoundsDelete	●	●	0	0	0	0	1	1	1	0	0
TypeBoundsDeleteSecond	●	●	0	0	0	0	1	1	1	0	0
TypeBoundsGeneralization	●	●	0	0	0	0	1	1	1	0	0
TypeBoundsMutation	×	●	0	0	0	0	1	1	1	0	0
TypeBoundsSpecialization	×	●	0	0	0	0	1	1	1	0	0
TypeDelete	×	●	0	0	0	1	1	1	1	0	0
TypeDeleteSecond	×	●	0	0	0	1	1	1	1	0	0
TypeSwap	●	●	0	0	0	0	1	1	1	0	0

Tato tabulka představuje testy v rozhraní. V názvu jednotlivých testů byl vynechaný začátek `genericsIfaze`.

	Zdrojová	Binární	Clirr	Japicmp	japi checker	JAPICC	Revapi	Sigtest	Japitools	Jour	JaCC
MethodTypeAdd	●	●	0	0	0	0	1	1	1	0	0
MethodTypeAddSecond	×	●	0	0	0	0	1	1	1	0	0
MethodTypeBoundsAdd	×	●	0	0	0	0	1	1	1	0	0
MethodTypeBoundsAddSecond	×	●	0	0	0	0	1	1	1	0	0
MethodTypeBoundsDelete	●	●	0	0	0	0	1	1	1	0	0
MethodTypeBoundsDeleteSecond	●	●	0	0	0	0	1	1	1	0	0
MethodTypeBoundsGeneralization	●	●	0	0	0	0	1	1	1	0	0
MethodTypeBoundsMutation	×	●	0	0	0	0	1	1	1	0	0
MethodTypeBoundsSpecialization	×	●	0	0	0	0	1	1	1	0	0
MethodTypeDelete	●	●	0	0	0	0	1	1	1	0	0
MethodTypeDeleteSecond	×	●	0	0	0	0	1	1	1	0	0
MethodTypeSwap	●	●	0	0	0	0	1	1	1	0	0
TypeAdd	●	●	0	0	0	0	1	1	1	0	0
TypeAddSecond	×	●	0	0	0	0	1	1	1	0	0
TypeBoundsAdd	×	●	0	0	0	0	1	1	1	0	0
TypeBoundsAddSecond	×	●	0	0	0	0	1	1	1	0	0
TypeBoundsDelete	●	●	0	0	0	0	1	1	1	0	0
TypeBoundsDeleteSecond	●	●	0	0	0	0	1	1	1	0	0
TypeBoundsGeneralization	●	●	0	0	0	0	1	1	1	0	0
TypeBoundsMutation	×	●	0	0	0	0	1	1	1	0	0
TypeBoundsSpecialization	×	●	0	0	0	0	1	1	1	0	0
TypeDelete	×	●	0	0	0	0	1	1	1	0	0
TypeDeleteSecond	×	●	0	0	0	0	1	1	1	0	0
TypeSwap	●	●	0	0	0	0	1	1	1	0	0

V názvu jednotlivých testů byl vynechaný začátek `genericsWildcardsClazz`.

	Zdrojová	Binární	Clirr	Japicmp	japi checker	JAPICC	Revapi	Sigtest	Japitools	Jour	JaCC
ConstructorParamAdd	●	●	0	0	0	0	1	1	1	0	1
ConstructorParamDelete	×	●	0	0	0	0	1	1	1	0	1
ConstructorParamLowerBoundsAdd	×	●	0	0	0	0	1	1	1	0	1
ConstructorParamLowerBoundsDelete	●	●	0	0	0	0	1	1	1	0	1
ConstructorParamLowerBoundsGeneralization	×	●	0	0	0	0	1	1	1	0	1
ConstructorParamLowerBoundsMutation	×	●	0	0	0	0	1	1	1	0	1
ConstructorParamLowerBoundsSpecialization	●	●	0	0	0	0	1	1	1	0	1
ConstructorParamLowerBoundsToUpperBounds	×	●	0	0	0	0	1	1	1	0	1
ConstructorParamUpperBoundsAdd	×	●	0	0	0	0	1	1	1	0	1
ConstructorParamUpperBoundsDelete	●	●	0	0	0	0	1	1	1	0	1
ConstructorParamUpperBoundsGeneralization	●	●	0	0	0	0	1	1	1	0	1
ConstructorParamUpperBoundsMutation	×	●	0	0	0	0	1	1	1	0	1
ConstructorParamUpperBoundsSpecialization	×	●	0	0	0	0	1	1	1	0	1
ConstructorParamUpperBoundsToLowerBounds	×	●	0	0	0	0	1	1	1	0	1
MethodParamAdd	●	●	0	0	0	0	1	1	1	0	1
MethodParamDelete	×	●	0	0	0	0	1	1	1	0	1
MethodParamLowerBoundsAdd	×	●	0	0	0	0	1	1	1	0	1
MethodParamLowerBoundsDelete	●	●	0	0	0	0	1	1	1	0	1
MethodParamLowerBoundsGeneralization	×	●	0	0	0	0	1	1	1	0	1
MethodParamLowerBoundsMutation	×	●	0	0	0	0	1	1	1	0	1
MethodParamLowerBoundsSpecialization	●	●	0	0	0	0	1	1	1	0	1
MethodParamLowerBoundsToUpperBounds	×	●	0	0	0	0	1	1	1	0	1
MethodParamUpperBoundsAdd	●	●	0	0	0	0	1	1	1	0	1
MethodParamUpperBoundsDelete	●	●	0	0	0	0	1	1	1	0	1
MethodParamUpperBoundsGeneralization	●	●	0	0	0	0	1	1	1	0	1
MethodParamUpperBoundsMutation	×	●	0	0	0	0	1	1	1	0	1
MethodParamUpperBoundsSpecialization	×	●	0	0	0	0	1	1	1	0	1
MethodParamUpperBoundsToLowerBounds	×	●	0	0	0	0	1	1	1	0	1

A.8 Výjimky

V názvu jednotlivých testů byl vynechaný začátek `exception`.

	Zdrojová	Binarní	Clirr	Japicmp	japi checker	JAPICC	Revapi	Sigtest	Japitools	Jour	JaCC
ClazzMethodCatchBlockAdd	●	●	0	1	1	1	1	1	1	1	0
ClazzMethodCatchBlockDelete	●	●	0	1	1	1	1	1	1	1	0
ClazzMethodFinallyBlockAdd	●	●	0	1	1	1	1	1	1	1	0
ClazzMethodFinallyBlockDelete	●	●	0	1	1	1	1	1	1	1	0
ClazzMethodThrowCheckedAdd	×	●	0	1	1	1	1	1	1	1	0
ClazzMethodThrowCheckedDelete	×	●	0	1	1	1	1	1	1	1	0
ClazzMethodThrowCheckedGeneralization	×	●	0	1	1	1	1	1	1	1	0
ClazzMethodThrowCheckedMutation	×	●	0	1	1	1	1	1	1	1	0
ClazzMethodThrowCheckedSpecialization	×	●	0	1	1	1	1	1	1	1	0
ClazzMethodThrowCheckedToTryCatch	×	●	0	1	1	1	1	1	1	1	0
ClazzMethodThrowUncheckedAdd	●	●	0	0	0	1	1	1	1	1	0
ClazzMethodThrowUncheckedDelete	●	●	0	1	0	1	1	1	1	1	0
ClazzMethodThrowUncheckedGeneralization	●	●	0	0	0	1	1	1	1	1	0
ClazzMethodThrowUncheckedMutation	×	●	0	1	1	1	1	1	1	1	0
ClazzMethodThrowUncheckedSpecialization	●	●	0	0	0	1	1	1	1	1	0
ClazzMethodTryCatchToThrowChecked	×	●	0	1	1	1	1	1	1	1	0

A.9 Ostatní

	Zdrojová	Binarní	Clirr	Japicmp	japi checker	JAPICC	Revapi	Sigtest	Japitools	Jour	JaCC
otherClazzAdd	●	●	1	1	1	1	1	1	1	1	1
otherClazzDelete	×	×	1	1	1	1	1	1	1	1	1
otherClazzToIfaze	×	×	1	1	1	1	1	1	1	1	1
otherIfazeAdd	●	●	1	1	1	1	1	1	1	1	1
otherIfazeDelete	×	×	1	1	1	0	1	1	1	1	1
otherIfazeToClass	×	×	1	1	1	1	1	1	1	1	1

A.10 Mimofunkční charakteristiky

	Clirr	Japicmp	japi checker	JAPICC	Revapi	Sigtest	Japitools	Jour	JaCC
Snadnost použití:									
Příkazový řádek	1	1	1	1	1	1	1	1	0
Maven	1	1	1	0	1	1	0	1	1
Ant Task	1	0	1	0	1	1	0	0	0
Java knihovna	0	1	0	0	0	0	0	0	1
Eclipse plugin	0	0	0	0	0	0	0	0	1
Výstupní formát									
Txt	1	1	1	0	1	1	1	1	1
XML	1	1	0	0	0	0	0	0	0
HTML	1	1	0	1	0	0	0	0	0
Přehlednost výstupu									
umístění	1	1	1	1	1	1	1	1	1
textový popis chyby	1	0	1	1	1	0	1	1	0
závažnost	1	0	1	1	0	0	1	0	0
rozlišení druhu nekompatibilit	0	1	0	1	1	1	0	0	1
Integrovatelnost do ž. c.									
Maven	1	1	1	0	1	1	0	1	1
Ant Task	1	0	1	0	1	1	0	0	0
Uživatelská dokumentace									
CLI	1	1	0	1	1	1	1	1	0
Maven	0	1	1	0	1	1	0	1	1
Ant Task	1	0	0	0	1	1	0	0	0
Java knihovna	0	1	0	0	0	0	0	0	1
Eclipse plugin	0	0	0	0	0	0	0	0	1