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

Bakalářská práce

Průzkum nástrojů a postupů pro zjišťování závislostí a vztahů mezi balíky a třídami v jazyce Java

Plzeň, 2015 Pavel Sikyta

Prohlášení

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

V Plzni dne 24. června 2015

Pavel Sikyta

Poděkování

Děkuji vedoucímu bakalářské práce Ing. Tomášovi Potužákovi, Ph.D. za cenné rady a připomínky.

Abstract

The main purpose of this bachelor's thesis is to compare current tools which analyze dependencies between classes and packages of Java applications. This thesis consists of three main parts. There is a description of dependencies of Java applications together with description what causes them in the first part. The second part consists of survey of existing tools, which analyze these dependencies and a description of process of extracting the dependencies through these tools. Third, significant part of the paper describes a detailed test, a comparison and a review of selected tools from the second part of the thesis.

Abstrakt

Hlavním cílem této bakalářské práce je porovnání existujících nástrojů pro analýzu závislostí mezi třídami a balíky v Java aplikacích. Práce se skládá ze tří hlavních částí. V první části je popsáno, co vlastně závislosti jsou a co je v Java aplikacích způsobuje. V druhé části je proveden průzkum existujících nástrojů, které zkoumají tyto závislosti. V této části je popsán i postup analýzy závislostí aplikací těmito nástroji. Poslední část se věnuje detailnímu testování, porovnání a hodnocení nástrojů vybraných v druhé části této práce.

Obsah

1 Úvod

Existují mnohá doporučení, jak správně konstruovat rozsáhlejší aplikace v Javě a jiných objektově orientovaných jazycích, tak, aby byly jednotlivé třídy správně rozděleny do balíčků a podbalíčků. Zároveň existují další doporučení, čemu je naopak dobré se vyhnout. V rozsáhlejších aplikacích se však často nevyhneme porušení těchto doporučení ať už úmyslně či opomenutím. Nesprávná struktura aplikace a podivné závislosti mezi třídami z různých (na první pohled nesouvisejících) balíčků pak může vést k obtížné údržbě či rozšiřování aplikace.

Hlavním cílem této bakalářské práce je porovnání existujících nástrojů pro analýzu závislostí mezi třídami a balíky v Java aplikacích. Práce se skládá ze tří hlavních částí. V první části je popsáno, co vlastně závislosti jsou a co je v Java aplikacích způsobuje. V druhé části je proveden průzkum existujících nástrojů, které zkoumají tyto závislosti. V této části je popsán i postup analýzy závislostí aplikací těmito nástroji. Poslední část se věnuje detailnímu testování, porovnání a hodnocení nástrojů vybraných v druhé části této práce.

2 Objektové vlastnosti jazyka Java

Java se řadí mezi hybridní jazyky. To znamená, že je současně překládána i interpretována. Programy v jazyku Java se nejprve přeloží do speciálního tvaru nazývaný bajtkód, který pak analyzuje a interpretuje speciální program nazývaný virtuální stroj Javy (Java Virtual Machine – zkratka JVM). Virtuální stroj umožňuje, aby jeden a týž program běžel na různých počítačích a operačních systémech (pro které existuje virtuální stroj). Ten se pak stará o správný běh programů, takže se program (a s ním i uživatel) vůbec nemusí starat o to, na jakém hardwaru a operačním systému zrovna běží [1].

Základním paradigmatem OOP (Objektově Orientovaného Programování) je snaha modelovat při řešení úloh principy reálného světa v počítači pokud možno jedna ku jedné. V praktickém životě otevíráme dveře pořád stejně, bez ohledu na to, zda jsou dřevěné nebo laminované, zda mají kukátko, bezpečnostní vložku nebo řetízek navíc. Stejně tak se můžeme dívat na televizi, přepínat programy a docela dobře ji ovládat, přesto že nevíme vůbec nic o principech jejího fungování. Analogicky při vývoji složitých informačních systémů mohou vývojáři používat již vytvořené komponenty, podle potřeby si je trochu upravit nebo je používat jako stavebnici pro sestavování důmyslnějších a složitějších objektů [2].

V jazyce Java existují primitivní datové typy (celá a reálná čísla, znaky, logické hodnoty) a dále objekty [2].

Objekty – jednotlivé prvky modelované reality (jak data, tak související funkčnost) jsou v programu seskupeny do entit, nazývaných objekty. Objekty mohou mít svůj stav a navenek poskytují operace (přístupné jako metody pro volání) [2].

2.1 Třída

Ve větších programech se mohou vyskytovat tisíce i desetitisíce objektů. Abychom s nimi mohli rozumně pracovat, musíme si je nějak roztřídit. Objekty jdou většinou rozdělit do skupin s velice podobnými vlastnostmi. Tyto skupiny označujeme jako třídy [3]. Třída je šablona, podle které můžeme vytvořit více instancí (objektů), přičemž každý objekt může mít různě nastavené proměnné - atributy.

V objektově orientovaném jazyce je základním stavebním kamenem třída. Třída představuje soubor proměnných, konstant a podprogramy, které s těmito proměnnými a konstantami pracují. Proměnným se říká členské proměnné nebo též datové složky nebo atributy a je v nich uložen stav objektu. Podprogramům se říká metody a manipulují s členskými proměnnými, čímž mění stav objektu [4].

Třída je ale jen jakási šablona (objektový typ) a sama o sobě nemá přidělenou žádnou paměť. Můžeme si ji představit ve velkém zjednodušení např. jako datový typ – pokud nedeklarujeme proměnnou určitého datového typu, není nám tento datový typ k ničemu. Objekt je datový prvek, který je vytvořen podle vzoru třídy. Často se říká, že objekt je instance třídy a často se termíny objekt a instance vzájemně volně zaměňují [4] .

Třídy v Javě mohou být pomocí vyhrazených klíčových slov jazyka zařazeny do různých typů a tím určovat celkovou architekturu aplikace.

2.1.1 Klasická třída

Klasická třída definovaná bez dodatečných modifikátorů (`static` (viz kapitola 2.1.2), `final` (viz kapitola 2.1.3) nebo `abstract` (viz kapitola 2.1.4)) dovoluje vytvářet instance dané třídy, kde je každému objektu přiřazena vlastní paměť. Třída je dostupná uvnitř dané třídy a ve třídách téhož balíku. Třídu je možné dědit a vytvářet tak jejich stromovou strukturu. Ke třídám množiny `public` lze navíc přistupovat i mimo balík, ve kterém jsou definovány.

Takto označené třídy jsou určené k použití kdekoli, tzn., že je možné například pracovat s instancemi takové třídy i v programu, který sestavuje někdo jiný než je autor programu [5].

2.1.2 Statická třída

Třídy určené slovem `static` (možné také pro jednotlivé členské proměnné a metody třídy) nejsou instancovatelné a nelze z nich tedy vytvářet objekty. K proměnným a metodám se přistupuje namísto jména objektu přímo skrze jméno třídy.

2.1.3 Final třída

To, co v Javě označíme modifikátorem `final`, je neměnné a nelze to jakkoli upravovat. V případě třídy to znamená, že z ní nepůjdou odvodit žádné podtřídy, což by také

nemělo smysl, protože všechny metody jsou statické a ty se vztahují pouze k této konkrétní třídě [4].

2.1.4 Abstraktní třída

Abstraktní třídy jsou v Javě definovány klíčovým slovem `abstract`, není možné vytvářet jejich instance. Třídu je však možné dědit – potomek třídy poté metody musí bezpodmínečně implementovat. Smysl abstraktní třídy je v tom, že tvoří společný základ pro více tříd, které od ní budou zděděny.

Jedná se o případ, kdy při konstrukci kořenové třídy již víme, jaké budeme potřebovat metody, ale jejich konkrétní implementace bude silně záviset na povaze zděděných tříd. Z tohoto důvodu není možné tyto metody naprogramovat. Výhodné řešení však je vytvořit abstraktní třídu a abstraktní metodu s jasně definovanými parametry a návratovým typem. Tím je možné donutit programátory zděděných tříd, aby tuto metodu překryli, protože bez implementace všech metod zděděné abstraktní třídy se program nepřeloží [6].

2.2 Rozhraní

Rozhraní shrnuje, co by okolní program měl vědět o dané entitě. Definuje soubor metod, které v něm však nejsou implementovány, tj. v deklaraci rozhraní je pouze hlavička metody, stejně jako je to u abstraktní metody. Třída, která toto rozhraní implementuje, musí poté překrýt všechny jeho metody (abstraktní třída může překrýt jen část z nich). Rozhraní nevynucuje příbuzenské vztahy, jako to činí dědičnost, pouze vnucuje určité dovednosti těm, kteří jsou těchto dovedností principiálně schopni [6].

Rozhraní je v Javě definováno klíčovým slovem `interface` a každá třída, která rozhraní implementuje, musí uvést jeho jméno v hlavičce za klíčovým slovem `implements`.

2.3 Balík

Java umožňuje využít tzv. balíčky (packages). Balíčky představují obdobu jmenných prostorů (namespace) v C++. Balíček je fyzicky uložen v adresáři. Jméno balíčku je tvořeno cestou k souborům se třídami v něm obsaženým. Nezapisuje se však pomocí zpětných lomítek, adresáře jsou odděleny tečkou. Máme-li v adresáři `Pokus` balíček

s názvem `balik`, ve kterém se nachází soubor `Pokus.java`, lze cestu k tomuto souboru zapsat ve tvaru `balik.Pokus`. Při návrhu Javy byl vytvořen takový systém pojmenování balíčků, aby byl v rámci celého světa jednoznačný. Název balíčku by měl vycházet z názvu internetových domén v opačném pořadí. Balík `balik` nacházející se v doméně `moje.cz` bude mít dle této konvence název `cz.moje.balik` [7].

Každá třída s modifikátorem public musí být uložena ve vlastním java souboru, jeho název musí odpovídat názvu třídy (včetně velikosti písmen). Pokud třída není public, může být i více tříd v jednom souboru. Balík vytváří prostor, v rámci něhož musejí být jména tříd jednoznačná; zároveň je to také prostor zajišťující zapouzdření tříd v rámci jednoho balíku. Z jiných balíků jsou přístupné pouze ty třídy, které jsou označeny atributem `public`. Zároveň všechny proměnné a metody tříd, které neobsahují žádný explicitní modifikátor (`private`, `protected` nebo `public`), jsou přímo dostupné všem metodám tříd z téhož balíku. To značně usnadňuje komunikaci v rámci tříd konkrétního balíku, kde by konzistence měla být zajištěna přímo jeho autory a není třeba všechny přístupy k atributům řešit voláním přístupových metod [8].

Pokud chceme nějakou třídu přidat do balíčku, uvedeme tuto informaci do prvního řádku zdrojového kódu příkazem [7]:

```
package nazev_baliku;
```

Stejně jako mohou složky obsahovat podsložky, mohou balíčky obsahovat podbalíčky, Název podbalíčku se skládá z názvu rodičovského balíčku následovaného tečkou a názvem podbalíčku. Podbalíčky mohou mít opět podbalíčky, a to do libovolné hloubky. Pro název platí stejná konvence. Balíčky tak mohou vytvářet stejnou stromovou strukturu jako složky a i pro jejich názvy platí obdobná pravidla jako pro absolutní cestu k dané složce – pouze se lomítka nebo zpětná lomítka nahrazují tečkou [1].

Obsahuje-li balíček nějaké podbalíčky, lze pro jejich zpřístupnění použít zástupný symbol *. V následujícím příkladu má balíček s názvem `balik` dva podbalíčky: `balicek1` a `balicek2`. Lze psát:

```
import balik.balicek1;

import balik.balicek2;
```

Nebo využít zástupný znak * a psát:

```
import balik.*;
```

Což v přeneseném smyslu znamená: připoj všechny podbalíčky balíčku `balik` [7].

2.4 Zapouzdření

V procedurálním programování neexistuje žádná ochrana proti nesprávnému použití atributů a procedur [9].

Jednou z charakteristických vlastností jazyka Java, stejně jako jiných objektových jazyků, je zapouzdření. Zapouzdření je explicitní spojení dat a metod dané konstrukcí třídy. To znamená, že se data vždy vyskytují společně s metodami, které s nimi pracují. Speciálním typem metod (i když z hlediska Javy se jedná o běžné metody) jsou pak *getry* a *setry*, které slouží ke kontrolovanému přístupu k datům z vnějšku třídy. Důsledkem zapouzdření je autorizovaný přístup k datům. Tak je zajištěno, že s daty není možné manipulovat z vnějšku třídy jinak, než pomocí metod této třídy. Tím je preventivně zabráněno některým chybám [4].

2.4.1 Přístupová práva

Modifikátor přístupu je klíčové slovo programovacího jazyka specifikující, jaká část programu může přistupovat k atributům a metodám [9].

Přístupová práva se v Javě nastavují jednak „hrubě“ na úrovni tříd, jemněji potom pro jednotlivé metody a proměnné objektů. Práva jsou určena vždy již při překladu tříd – nelze je měnit za běhu [5].

`public`

Modifikátor `public` definuje třídy, metody a atributy jako veřejné, tzn. je možno k nim přistupovat odkudkoli z kódu programu.

Žádný modifikátor (někdy označován jako friendly)

Přátelská třída nebo metoda je přístupná pouze uvnitř dané třídy a ve třídách téhož balíku. Jedná se o implicitní chování, když není uveden žádný modifikátor přístupových práv.

`protected`

Třídy nebo jejich metody a atributy označené klíčovým slovem `protected` jsou tzv. chráněné. Je možné k nim přistupovat pouze z třídy samotné, ze tříd téhož balíku, nebo ze tříd, které jsou potomkem aktuální třídy.

`private`

Příznak `private` specifikuje úseky kódu, ke kterým je možné přistupovat pouze z jejich mateřské třídy.

2.5 Dědičnost

Dědičnost představuje možnost přidat k rodičovské třídě další vlastnosti a vytvořit tak odvozenou třídu – potomka. Odvozená třída bývá specializovanější než třída základní.

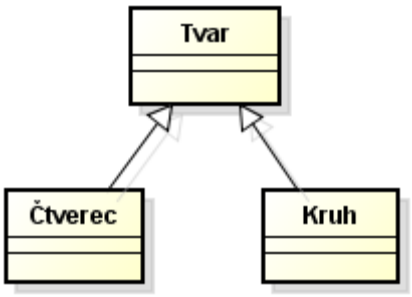
Dědění je i významný nástroj pro vytváření opakovaně využitelných programových částí (modulů).

Vícenásobná dědičnost znamená, že třída vznikla děděním z více rodičovských tříd najednou. Tato možnost přináší různé výhody, ale také různé problémy (nejznámější je dědění od téhož předka dvěma cestami). „Protože autoři Javy byli přesvědčení, že problémů je více než výhod, vícenásobnou dědičnost neumožnili“ [4]. Java tak umožňuje pouze jednoduchou dědičnost, tj. každá třída může mít pouze jednoho přímého předka. Jako náhradu za vícenásobnou dědičnost má ale Java možnost používat *rozhraní*.

Třídy, od kterých není možné dále dědit, lze v Javě specifikovat vyhrazeným slovem `final`. Taková třída může mít instance, ale nesmí být použita jako rodičovská třída pro přípravu jiných tříd [4].

Podobně, jako lze dědit třídy, lze dědit i rozhraní. To znamená, že rozhraní, které je potomkem, přebírá všechny deklarované metody od svého rodiče. Rozdíl od dědění tříd je v tom, že v případě rozhraní může být rodičů více a pak se přebírají všechny metody všech rodičů [4].

Na Obrázku č. 2.1 je v roli rodiče třída `Tvar`. Zbylé třídy (`Čtverec` a `Kruh`) dědí od třídy `Tvar` a jsou tedy v rolích potomků. To znamená, že jsou upřesněním třídy `Tvar` [6].



Obrázek 2.1: příklad dědičnosti tříd

2.6 Polymorfismus

Slovo polymorfizmus lze nejblíže vyjádřit českým slovem *vícetvarost* nebo mnohotvarost. Jedná se o možnost využívat v programovém textu stejnou syntaktickou podobu pro metody s různou vnitřní reprezentací. Jinak řečeno, voláme pořád stejnou metodu, ale ta provádí pokaždé něco jiného. Polymorfismus dovoluje jednotným způsobem manipulovat s prvky příbuzného, ale předem neznámého typu [4].

V Javě může potomek díky polymorfismu nahradit předka. To znamená, že do referenční proměnné, původně deklarované s typem předka, můžeme přiřadit referenci na instanci potomka. Pak lze přes referenční proměnnou předka využívat pouze metody, které byly definovány v předkovi. Je ale možné, že v potomkovi budou mít jinou funkcionalitu. Metody definované pouze v potomkovi (a ne předkovi) nejsou přes referenční proměnnou předka přímo přístupné (je nutné explicitní přetypování na potomka). To má význam hlavně v případě více typů potomků, protože k nim můžeme přistupovat jednotným (typově nezávislým) přístupem pomocí referenční proměnné na jejich společného předka. Pro polymorfismus se využívají například abstraktní třídy a hlavně rozhraní [6].

2.7 Generičnost

Generičnost je možnost programovacího jazyka definovat místo typů jen „vzory typů“, kde typy proměnných, použité v definici typu (typu jako abstraktní datový typ), jsou vyvedeny vně definice jako parametry a jsou určeny později klientskou aplikací. Základním užitím generičnosti jsou třídy kontejnerů [10].

Kontejnery jsou objekty, které jsou určeny k uchovávání jiných objektů. Rozlišujeme kontejnery statické (s konečně přidělenou velikostí) a dynamické (mohou svoji velikost měnit za běhu). V předchozích verzích Javy nebyla možnost specifikovat, co do kontejneru patří, tak, aby to mohl zkontrolovat překladač. Java 5.0 přinesla generické typy, což jsou datové typy, jejichž některé vlastnosti je možno specifikovat až v okamžiku kdy proměnnou daného typu deklarujeme. Za názvem generického typu se ve špičatých závorkách uvádějí typové parametry. Ty v případě kontejnerů specifikují, jakých typů jsou objekty, které do kontejneru ukládáme [3].

např. `new ArrayList<Tvar>`

Kouzlo generičnosti vynikne pak v kombinaci s dědičností, kdy do seznamu mohou být vloženy nejen objekty typu `Tvar`, ale i objekty všech možných potomků třídy `Tvar`. Konkrétní typ, použitelný v textu programovacího jazyka pak vzniká, když `Tvar` nahradíme skutečným existujícím typem (`Ctverec` nebo `Kruh` viz Obrázek 2.1) [10].

2.8 Architektury aplikací

Architektura je všeobecné označení určující celkovou strukturu a základní konstrukci částí nebo kompletního počítačového systému [11]. V jazyce Java i v mnoha jiných jazycích existuje několik architektur aplikace, které se běžně používají. S architekturou aplikací souvisí organizace tříd do balíčků a uspořádání balíčků do vrstev, podsystémů. Jedná se tedy o způsob rozdělení aplikace, aplikačních dat, procesů i datových toků do logických celků, stanovení struktury těchto komponent, vzájemných vztahů a interakcí mezi nimi, a to na dostatečně obecné úrovni [12]. V této práci je architektura aplikací popsána proto, aby byla zjevná možnost vývoje jednotlivých částí nezávisle na sobě.

2.8.1 Klient-server

V této architektuře jsou služby rozděleny do dvou vrstev – prezentační a datové. Část aplikace se spouští na serveru a zbytek na klientovi. Používají se různé varianty rozložení vždy stejných služeb mezi server a klienta jako jsou Klient/server se vzdálenými daty, Klient/server se vzdálenou prezentací nebo Klient/server s rozdělenou logikou. Každá varianta jinak rozkládá zatížení mezi klienta a server. Z této „dvouvrstvé“ architektury vznikla třívrstvá architektura.

2.8.2 Třívrstvá architektura

MVC (Model View Controller) je oblíbená architektura v Java aplikacích. Rozděluje aplikaci na tři logické části a pro každou z nich přesně definuje, za co je odpovědná. Rozdělením do částí jednak aplikaci zpřehledňuje, usnadňuje budoucí vývoj a umožňuje testování jednotlivých části zvlášť.

Model je funkčním a datovým základem celé aplikace. Poskytuje prostředky jak pro přístup k datové základně a stavům aplikace, tak pro jejich ukládání a aktualizaci. Měl by být jako celek zapouzdřený a pro *View* a *Controller* nabízet přesně definované rozhraní [12].

View zobrazuje obsah Modelu, zajišťuje grafický či jiný výstup aplikace. Přes Model přistupuje k datům a stavům aplikace a specifikuje, jak mají být prezentovány [12].

Controller definuje chování aplikace. Zpracovává veškeré vstupy a události pocházející od uživatele. Na jejich základě volá příslušné procesy *Modelu*, mění jeho stav apod. Podle událostí přijatých od uživatele i podle výsledků akcí v *Modelu* pak vybírá *Controller* vhodné *View* pro další zobrazení [12].

Výhody uváděné u této architektury jsou oddělení business logiky od prezentační vrstvy, snadná výměna jednotlivých vrstev a znovupoužitelnost existujícího kódu.

2.8.3 Komponentová architektura

V případě komponentové architektury je program poskládán ze vzájemně nezávislých komponent – modulů. Systém může být jednoduše rozšířen o další funkčnost – komponentu. Stejně tak je možné vyměnit jednu komponentu za jinou – se stejnou funkcionalitou bez toho, aby byly ovlivněny jiné části systému.

Komponenta je softwarový balík, který poskytuje ucelenou, samostatně užitečnou funkčnost. Tento balík je možné samostatně distribuovat a nasadit do provozního prostředí. Každá komponenta má jasně definované rozhraní. Rozhraní popisuje, co komponenta poskytuje a naopak co komponenta potřebuje. Jednotlivé komponenty mezi sebou komunikují přes tato rozhraní, samotná implementace komponenty zůstává pro okolí skrytá. Tomuto principu se také říká *black-box model*. Neexistuje žádný způsob, jak zavolat funkcionalitu, která není definována v rozhraní, to znamená, že komponenta nesmí záviset na svém okolí jinak, než jak explicitně deklaruje v rozhraní [13]. Výhoda

této architektury je díky znuvupoužitelnosti existujících komponent, nebo komponent třetích stran rychlé nasazení a možnost vyvíjení jednotlivých komponent nezávisle na sobě.

2.8.4 Service-oriented architecture (SOA)

Z názvu servisně orientovaná architektura je jasné, že tato architektura klade důraz na služby. Při dodržení principů SOA ve fázi návrhu lze výsledný systém přirovnat ke stavebnici. Výsledný systém je podkladný ze služeb. Architektura se dá z části přirovnat k architektuře komponentové. Existují katalogy služeb popisující funkcionalitu služby, konkrétní implementace je ale ostatním částem utajena a pro dvě stejné služby se může lišit. V katalogu je možné vybrat službu podle konkrétních potřeb. Ze SOA principů je pro důvody této práce důležité zmínit snahu o co nejtenčí vazby mezi jednotlivými služby a kladení důrazu na možnost znovupoužití služby i v jiném projektu již pří jejím návrhu.

2.9 Návrhové vzory

Důležitou zásadou produktivního programování je snaha o znovupoužití již naprogramovaných tříd namísto vymýšlení nových postupů. Návrhové vzory se dají charakterizovat jako praxí ověřené návody na řešení některých typických, často se vyskytujících úloh [1]. S použitím standardních postupů se váže rychlejší řešení i zmenšení pravděpodobnosti chyby [3]. Oblíbenost mezi objektově orientovanými programátory odstartovala v roce 1995 kniha *Desing Patterns* obsahující 23 základních návrhových vzorů všeobecného použití [1]. Od té doby počet zveřejněných návrhových vzorů roste.

V literatuře je asi nejčastěji udávaný návrhový vzor *Singleton* (česky jedináček), který doporučuje postup (znepřístupnit konstruktor a pro získání odkazu na instanci nabídnout *jednoduchou tovární metodu (simple factory method)*, další návrhový vzor, která pokaždé vrátí odkaz na stejnou instanci) jak vytvořit třídu, která má právě jedinou instanci [3].

Dalším návrhovým vzorem je *Messenger* – jeho účelem je umožnit pracovat s několika hodnotami jako s hodnotou jedinou, a umožnit tím snadný přenos této skupiny hodnot mezi jednotlivými instancemi a jejich metodami [1].

2.10 Závislosti v aplikacích

Kdykoli `třída A` používá jinou `třídu` nebo `rozhraní B`, pak `A` závisí na `B`. `Třída A` nemůže pokračovat bez `B` a `A` nemůže být použita bez použití `B`. Dvě třídy, které se navzájem používají, se v angličtině nazývají *coupled* (spojené, svázané). Vazba mezi třídami může být volná, těsná nebo někde mezi tím. Spojení není charakterizováno jako binární ani diskrétní. Závislosti mohou být charakterizovány i jako silné a slabé přičemž těsná vazba vede k silným závislostem a volná vazba ke slabým nebo žádným závislostem [14]. Se závislostmi souvisí i různé druhy metrik, které je hodnotí.

U závislostí závisí i na směru závislosti. Závislosti nejsou automaticky obousměrné. Nelze říct, že když `A` závisí na `B`, pak `B` závisí na `A` [14].

2.10.1 Co vyvolává závislosti

Závislosti vznikají následujícími situacemi.

Interakcí třídy s třídou (`class` to `class`)

- Pokud třída rozšiřuje třídu (`extends`),
- Pokud třída implementuje rozhraní (`implements`),

Interakcí metod nebo proměnných instance s třídou (`feature` to `class`)

- Proměnná instance je typu této třídy,
- Lokální proměnná je typu této třídy,
- Tato třída je parametrem metody,
- Metoda v případě výjimky použije tuto třídu v klauzuli `throws [15]`.

Interakcí metod nebo proměnných instance s metodou nebo proměnných instance jiné třídy (`feature` to `feature`)

- Přístup k proměnné objektu jiné třídy,
- Voláním metody jiné třídy [15].

2.10.2 Závislost balíčku na balíčku

Balíček A závisí na balíčku B právě tehdy, když alespoň jedna třída z balíčku A závisí na alespoň jedné třídě z balíčku B.

2.10.3 Cyklická závislost

Jestliže prvek A závisí na prvku B a zároveň prvek B závisí na prvku A, jedná se o tzv. cyklickou závislost. Nejčastěji jsou sledovány cyklické závislosti u balíčků a tříd.

Je žádoucí, organizovat závislosti mezi balíčky jedním směrem. To umožňuje provedení změny a předpověď jejího dopadu. Například pokud balíček GUI závisí na balíčku logika a nic nezávisí na balíčku GUI, mělo by být možné provést změny v balíčku GUI bez toho, aby došlo k problémům mimo balíček GUI. Cyklická závislost by s využitím stejných balíčků vypadala následovně. Balíček GUI závisí na balíčku logika ale i balíček logika závisí na balíčku GUI. Mohlo by dojít k tomu, že pokud změním něco v balíčku GUI, mohlo by to narušit logickou vrstvu a ta by následně mohla ovlivnit něco zdánlivě úplně nesouvisejícího v balíčku GUI. Při dodržování vrstvení softwaru a organizování závislostí jedním směrem mezi vrstvy nevznikají cyklické závislosti. Cyklické závislosti tedy značí nedodržení architektury. Velmi často se tyto cykly dají odstranit přesunutím třídy nebo několika tříd z jednoho balíčku do druhého [16].

3 Nástroje pro analýzu vztahů mezi třídami a balíky

Postupně jsem začal na internetu vyhledávat nástroje, které by byly použitelné pro analýzu vztahů mezi třídami a balíky v Java aplikacích. Nástroje jsem vyhledával na obecných webech, v diskusních fórech věnujících se programování a na tzv. Questions and Answers (Q&A) webových stránkách.

Úplně první nástroj *Jgrasp* jsem našel ve vědeckém článku [17]. Článek v první polovině pojednává o teorii, jak získat závislosti z `class` souborů, v druhé polovině článku je popsán nástroj *Jgrasp*. Po přečtení článku a zběžném prozkoumání nástroje jsem začal sepisovat množinu kritérií, která budu u všech dalších nástrojů sledovat. Tuto množinu kritérií jsem během průzkumu nástrojů rozšiřoval a u předešlých nástrojů nová kritéria zpětně ohodnotil.

3.1 Sledovaná kritéria

Mezi hlavní sledovaná kritéria v tomto průzkumu patří například druh licence, schopnost nástroje zkoumat závislosti mezi balíčky a závislosti mezi třídami, schopnost nástroje získat závislosti z `class` souborů či pouze ze zdrojových souborů a schopnost nástroje závislosti vizuálně zobrazit. Dále bylo sledováno, jestli nástroj funguje jako samostatná aplikace s grafickým uživatelským rozhraním (GUI) jako aplikace v příkazovém řádku nebo jako plugin do vývojového prostředí *Eclipse*.

U nástroje byly zjišťovány jeho možnosti výstupů analýzy. Dále bylo sledováno, kdy byla vydána poslední verze nástroje, byla zohledněna celková udržovanost projektu případně plány rozšíření nástroje do budoucna, dostupnost a obsáhlost dokumentace k nástroji. Zohledněny byly také ohlasy uživatelů nástroje. V případě, že má nástroj GUI, bylo zhodnoceno jeho zpracování, rychlost a nakolik je ovládání nástroje intuitivní.

3.2 Seznam nástrojů

Celkem bylo do průzkumu zařazeno 30 nástrojů (viz Tabulka 3.1), které se z počátku jevily, jako vhodní kandidáti.

Nástroj č. 1-15	Nástroj č. 16-30
jGrasp	Eclipse Metrics plugin
CodePro AnalytiX	Eclipse Metrics plugin continued
jDepend	Coderu
Degraph	Java Dependency Viewer
DependencyFinder	UCDetector
DepAn	Jetbrains-InteliJ IDEA
iSpace	SonarJ/sonargraph
Classycle + Classycle plugin	eUML2/eDepend
Findbugs	Structure 101
Class Dependency Analyzer (CDA)	Jarchitekt
Highwheel	Nwire
SonarQube + SonarQube plugin	Lattix
Eclipse Project dependencies viewer (epdv)	Stan4j
Byecycle	Dependometer
Code Analysis Plugin	Pasta: Package Structure Analysis Tool

Tabulka 3.1: počáteční seznam nástrojů

U všech těchto nástrojů byla sledována kritéria uvedená v kapitole 3.1. Na základě těchto kritérií bylo rozhodnuto, zda daný nástroj bude prozkoumán podrobněji.

3.3 Vyřazení nástrojů z podrobnějšího testu

Nástroj *jGrasp* nebyl zařazen do podrobnějšího testu, protože neumí zkoumat závislosti mezi balíčky aplikace.

Nástroj *jDepend* nebyl zařazen do podrobnějšího testu, protože nezkoumá závislosti mezi třídami aplikace. Poslední verze nástroje je z roku 2005.

Nástroj *DepAn* nebyl zařazen do podrobnějšího testu, protože neumí zkoumat závislosti mezi balíčky aplikace.

Nástroj *iSpace* nebyl zařazen do podrobnějšího testu, protože je projekt mrtvý, poslední verze nástroje je z roku 2007.

Nástroj *Findbugs* nebyl zařazen do podrobnějšího testu, protože neumí zkoumat závislosti mezi balíčky aplikace.

Nástroj *Highwheel* nebyl zařazen do podrobnějšího testu, kvůli nedostupnosti dokumentace k nástroji.

Nástroj *Eclipse project dependencies viewer (epdv)* nebyl zařazen do podrobnějšího testu, protože zkoumá závislosti jen mezi projekty.

Nástroj *Byecycle* nebyl zařazen do podrobnějšího testu, kvůli špatným ohlasům uživatelů a protože je projekt mrtvý. Projekt byl smazán ze serveru www.sourceforge.net, i z *Eclipse* marketu. Poslední verze nástroje je z roku 2010.

Nástroj *Code analysis plugin* nebyl zařazen do podrobnějšího testu, protože je projekt mrtvý. Poslední verze nástroje je z roku 2005. Uživatelé plugin ohodnotili na 40%.

Nástroj *Eclipse Metrics plugin* nebyl zařazen do podrobnějšího testu, protože neumí zkoumat závislosti mezi třídami aplikace.

Nástroj *Eclipse Metrics plugin continued* nebyl zařazen do podrobnějšího testu, na základě jeho negativního hodnocení a nulového počtu stažení v *Eclipse* marketu.

Nástroj *Coderu* nebyl zařazen do podrobnějšího testu, protože jsem, po prozkoumání nástroje, zjistil, že to není nástroj pro statickou analýzu, ale spíš doplněk sestavovacího procesu aplikace.

Nástroj *Java Dependency Viewer* nebyl zařazen do podrobnějšího testu, na základě velmi špatného hodnocení uživatelů.

Nástroj *UCDetector* nebyl zařazen do podrobnějšího testu, protože neumí zkoumat závislosti mezi balíčky aplikace.

Nástroj *Dependometer* nebyl zařazen do podrobnějšího testu, protože neumí zkoumat závislosti mezi balíčky aplikace a protože vývoj nástroje dál nepokračuje.

Nástroj *iSpace* nebyl zařazen do podrobnějšího testu, protože je projekt mrtvý, poslední verze nástroje je z roku 2002.

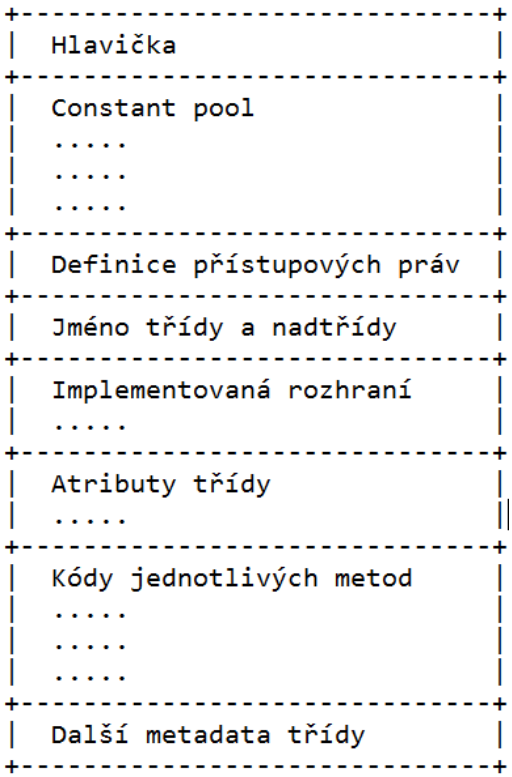
Nástroje *Jetbrains-InteliJ IDEA, SonarJ/Sonargraph, eUML2/eDepend, Structure 101, Jarchitekt, Nwire, Latex a Stan4J* nebyly zařazené do podrobnějšího testu, protože jsou placené.

3.4 Obecný průběh analýzy vztahů mezi třídami a balíky

V této kapitole je popsán zobecněný postup toho, jak nástroje zkoumají závislosti mezi třídami a balíky v Java aplikacích.

Pro účely této kapitoly je zde popsána struktura `class` souboru. Všechny kompilátory musí generovat `class` soubory tak, aby dodržely specifikaci – formát souboru, který je popsán v [18] a znázorněn na Obrázku 3.1. Pro analýzu závislostí je důležitý záznam *Constant pool* obsahující záznamy viz Tabulka 3.2, některé z nich jsou popsány níže.

Constant pool je obvykle objemově nejrozsáhlejší část `class` souborů. Nachází se zde přesné typové informace o všech použitých třídách i o volaných metodách těchto tříd. Důležitý je záznam obsahující řetězec (kódovaný s využitím UTF-8), protože na tento typ záznamu se odkazují mnohé další záznamy. To se týká i záznamu typu *`String`*, který se odkazuje právě na záznam typu *Utf8* (odkaz není nic jiného, než index záznamu v *constant poolu).* Záznam typu *Class* obsahuje jména tříd (přesněji řečeno odkazy na řetězce se jmény tříd) – konkrétně se jedná o plné názvy (názvy i s příslušným balíkem – například java/lang/Object) [19].



Obrázek 3.1: Struktura `class` souboru; převzato z *[19]*

Constant Type	Value
CONSTANT_Class	7
CONSTANT_Fieldref	9
CONSTANT_Methodref	10
CONSTANT_InterfaceMethodref	11
CONSTANT_String	8
CONSTANT_Integer	3
CONSTANT_Float	4
CONSTANT_Long	5
CONSTANT_Double	6
CONSTANT_NameAndType	12
CONSTANT_Utf8	1
CONSTANT_MethodHandle	15
CONSTANT_MethodType	16
CONSTANT_InvokeDynamic	18

Tabulka 3.2: seznam Constant pool tagů; převzato z [18]

3.4.1 Analýza dostupných java nebo class souborů

Například nástroj *Classycle* (viz kapitola 5.1.4) popisuje svoji analýzu závislostí a rozděluje ji do šesti hlavních částí. Následuje popis čtyř z těchto částí, které jsou dostatečně obecné tudíž s velkou pravděpodobností i společné pro ostatní nástroje zkoumající závislosti z `class` souborů. Zbývající dvě části jsou specifické pro tento nástroj (popisují výpočet indexu vrstvy a generování reportu analýzy) a nebudou zde uvedeny. Zmíněné části jsou:

1. Přečtení `class` souborů

Přečte `class` soubory a extrahuje z nich *Constant pooly*, které obsahují odkazy na ostatní třídy [20].

2. Vytvoření grafu závislostí tříd

Aby mohl nástroj odhalit závislosti tříd, analyzuje extrahované *Constant pooly*. Nástroj analyzuje dva nebo tři typy záznamů z *Constant poolu* - *Class constant*, *Utf8 constant* a volitelně *String constant* viz kapitola 3.4. Výstupem této analýzy je orientovaný graf závislostí tříd. [20]

3. Vytvoření grafu závislostí balíčků z grafu závislostí tříd.

Je jednoduché vytvořit z grafu závislostí tříd graf závislostí balíčků. Balíček A závisí na balíčku B tehdy, když alespoň jedna třída z A závisí alespoň na jedné třídě z B [20].

Detekce cyklických závislostí pomocí *Tarjanova algoritmu* [20].

Tarjanův algoritmus je grafový algoritmus sloužící k vyhledávání silných komponent orientovaného grafu. Silná komponenta je maximální množina uzlů orientovaného grafu taková, že mezi každými dvěma uzly existuje cesta [21].

Pro práci s `class` soubory existuje například i knihovna *The Byte Code Engineering Library (BCEL)*, kterou, za účelem analýzy `class` souborů, využívá například nástroj *Class Dependency Analyzer (CDA)* (viz kapitola 5.1.6).

3.4.2 Volitelný export modelu

Po dokončení analýzy umožňují některé nástroje exportovat vypočtený model závislostí např. ve formátu XML (Extensible Markup Language). Tyto nástroje pak, bez nutnosti opětovné analýzy, umožňují tento model znovu použít popřípadě na něj aplikovat omezení (například zobrazit jen závislosti konkrétního typu).

3.4.3 Prezentování závislostí

Analýzou získané závislosti je potřeba uživateli nástroje nějakým způsobem prezentovat. Některé nástroje volí formu prezentace textovým výpisem seznamů závislostí. Jiné nástroje závislosti graficky znázorňují – grafem závislostí. Další možností prezentace závislostí je forma reportu (například v XML nebo HTML (HyperText Markup Language) formátu), který obsahuje jeden nebo oba předešlé body a ideálně klade důraz na nežádoucí závislosti společně s dalšími metrikami. Ideálním případem jsou nástroje, které umí závislosti prezentovat všemi výše zmíněnými možnostmi, nebo kombinací z nich.

4 Aplikace pro testování nalezených nástrojů

Aby bylo možné nalezené nástroje co nejdůkladněji otestovat, bylo potřeba připravit sadu několika Java aplikací, na kterých půjde provést testy všech nástrojů. Bylo vhodné, aby aplikace byly co možná nejvíce různorodé – velikostí, strukturou, počtem tříd a balíčků. Potřeba byla jednoduché aplikace s malým počtem balíčků a tříd, na které bylo možné jednoduše manuálně ověřit závislosti a jejich počet a naopak byla potřeba několik složitějších aplikací s velkým počtem balíčků a tříd, ale různorodou strukturou, aby bylo možné nástroje dostatečně zatížit.

Bylo potřeba otestovat, s jakými typy souborů umí nástroje pracovat, jestli nástroje umí získávat závislosti ze zdrojového kódu (soubory s koncovkou `.java`) či z bajtkódu (soubory s koncovkou `.class`). Dále bylo potřeba zjistit, jestli nástroje umí získávat závislosti i z jiných typů souborů jako například Java Archive (koncovka `.jar`), který se používá k distribuci programů a knihoven napsaných v Javě, Web Archive (koncovka `.war`) používaný v Javě EE nebo Enterprise Archive (koncovka `.ear`).

Bylo otestováno, zda nástroje umí pracovat s adresářovou strukturou, popřípadě s archivem ve formátu `zip`. Nástroje bylo potřeba otestovat také na detekci cyklických závislostí.

Jako hotové aplikace pro otestování nástrojů jsem použil veřejně dostupné bakalářské a diplomové práce. K aplikacím obsaženým v bakalářských a diplomových pracích obecně bývají dostupné zdrojové kódy i bajtkód. Vzhledem k tomu, že jsou práce veřejně dostupné, nedochází k porušování žádných autorských práv.

Od obou typů prací jsem také očekával jinou kvalitu zpracování aplikací. Aplikace naprogramované v rámci bakalářských prací by teoreticky měly s větší pravděpodobností porušovat zvyklosti programovacího jazyka a mohly by častěji obsahovat zmíněné cyklické závislosti. Aplikace naprogramované v rámci diplomové práce by naopak mohly obsahovat menší množství chyb závislostí.

Celkem jsem prozkoumal aplikace z jedenácti kvalifikačních prací (6 bakalářských prací a 5 diplomových prací). Informace o struktuře aplikací (počet balíků a tříd, viz Tabulka 4.1) jsem získal pomocí nástroje *OO Metrics – Extractor*.

id aplikace	Zdroj	počet balíků	počet tříd
1	Bakalářská práce	3	12
2	Bakalářská práce	10	110
3	Bakalářská práce	16	124
4	Bakalářská práce	6	49
5	Bakalářská práce	8	50
6	Bakalářská práce	20	238
7	Diplomová práce	39	557
8	Diplomová práce	690	11586
9	Diplomová práce	9	80
10	Diplomová práce	10	87
11	Diplomová práce	11	89

Tabulka 4.1: struktura všech nalezených aplikací

Z aplikací, obsažených v Tabulce 4.1, jsem vyřadil podobně strukturované aplikace. Nakonec jsem se rozhodl vybrat pro testování aplikace obsažené v Tabulce 4.2. Jedná se o dvě aplikace, které vznikly v rámci bakalářské práce a dvě aplikace vytvořené jako součást diplomových prací.

id aplikace	Zdroj	počet balíků	počet tříd
1	Bakalářská práce	3	12
5	Bakalářská práce	8	50
7	Diplomová práce	39	557
8	Diplomová práce	690 (5)	11586 (51)

Tabulka 4.2: sada aplikací, na které budou prováděny testy v rámci kapitoly 5

Později se ukázalo, že do počtu balíků a tříd u aplikace id8 byly započítány i veškeré knihovny přibalené do webového archivu – souboru, na kterém byl prováděn test nástrojem *OO Metrics – Extractor*. Ukázalo se, že správný počet balíků 5 a správný počet tříd je 51. Aplikace id7 s 39 balíčky a 557 třídami by však měla být pro zátěžový test nástrojů dostačující.

5 Podrobné testování vybraných nástrojů

Nyní důkladně otestuji zbývající následující nástroje, které jsem, na základě kriterií, zmíněných v kapitole 3, nevyřadil z podrobného testování. V průběhu testování jsem ověřoval proklamované funkcionality nástrojů, rozšiřoval jsem množinu funkcionalit, které budu u všech nástrojů prověřovat a hodnotit. Jednotlivé funkcionality z finální množiny sledovaných funkcionalit jsem podle významu rozřadil do šesti kategorií.

Kategorie *nástroj* obsahuje ty funkcionality, které popisují nejvíce vypovídající vlastnosti a možnosti práce s nástrojem jako takovým.

Kategorie *výstupy* obsahuje ty funkcionality, které popisují možnosti výstupů analýzy závislostí provedené nástrojem.

Kategorie *GUI* obsahuje funkcionality, které hodnotí a vypovídají o možnosti práce v GUI.

Kategorie *graf* obsahuje funkcionality popisující práci s grafem.

Kategorie *„umí najít“* obsahuje funkcionality vypovídající o schopnostech nástroje detekovat závislosti na různých úrovních aplikace.

Kategorie *seznam* obsahuje funkcionality, které v rámci reportu, přímo v GUI nebo jiným způsobem umožňují vypsat seznam různých závislostí.

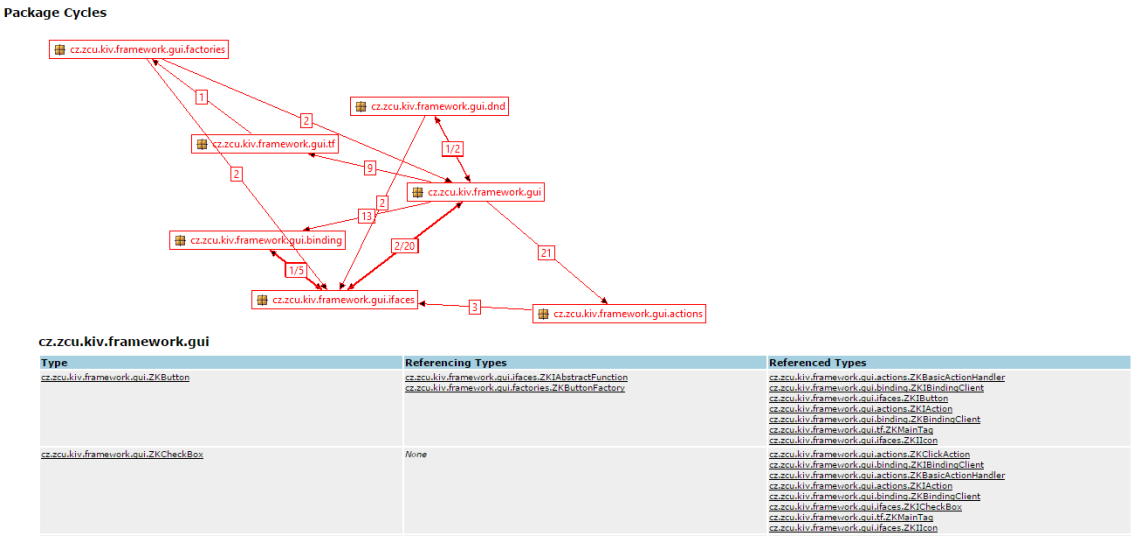
Všechny následující testy probíhaly na notebooku o konfiguraci Intel® Core™ i5-2450M CPU @ 2.50GHz, 4GB RAM, Windows 8.1 Pro N (64-bit).

5.1 Představení nástrojů

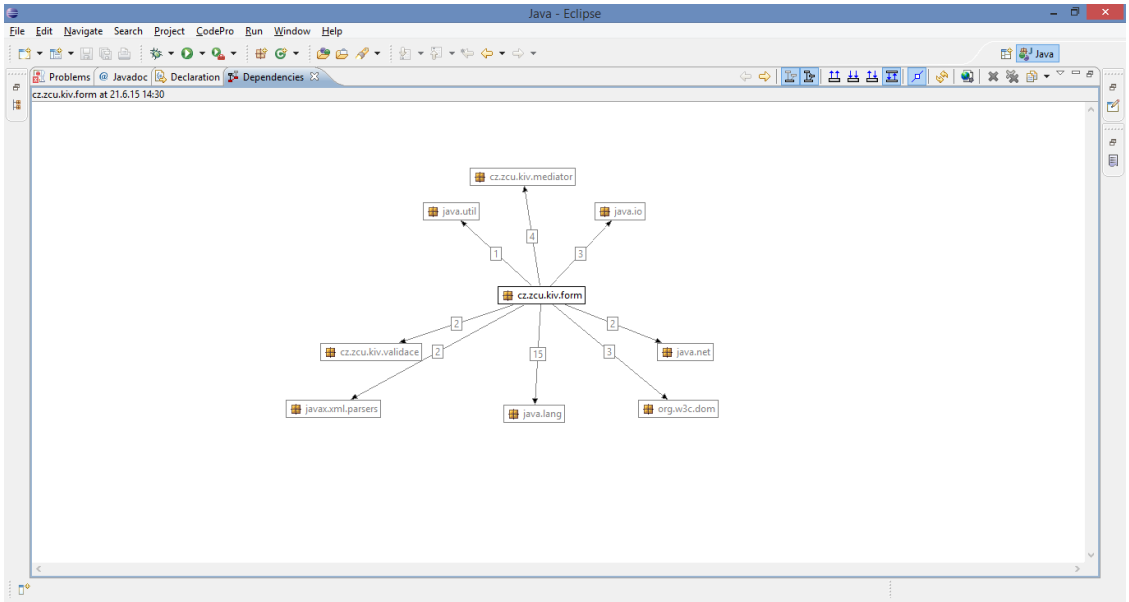
5.1.1 CodePro Analytix

Jedná se o nástroj pro zlepšování kvality softwaru, který funguje jako plugin pro vývojové prostředí *Eclipse* (viz Obrázek 5.2 a Obrázek 5.3). Mezi hlavní funkce tohoto nástroje patří analýza kódu, výpočet metrik, generování *JUnit* testů, měření pokrytí kódu testy, hledání duplikátů v kódu, analýza a zobrazení závislostí v Java aplikacích. Já se samozřejmě zaměřil na otestování funkčnosti analýzy a zobrazení závislostí. Dále se budu věnovat jen těmto dvěma funkcím. Funkcionality nástroje popisuji také v tabulce 5.1.

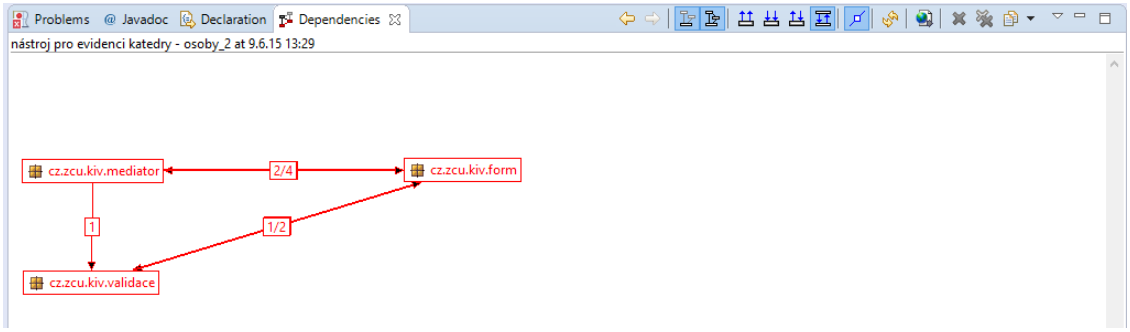
Poslední a zároveň testovaná verze 7.1.0 nástroje byla vydána v roce 2010 pro vývojové prostředí *Eclipse* verze 3.7. Momentálně je poslední dostupná verze 4.4.2. Plugin v nejnovější verzi *Eclipse* nefungoval, proto jsem dále pokračoval v testování nástroje v *Eclipse* 3.7. Nic nenasvědčuje tomu, že by autoři projektu měli v plánu pokračovat s vývojem nástroje. Možnosti práce s nástrojem bych rozdělil do dvou hlavních bodů – možnost práce s grafem závislostí v rámci prostředí *Eclipse* a možnost pracovat podle vygenerovaného reportu viz Obrázek 5.1. Možnosti prostředí a práce s grafem jsou uvedené v tabulce 5.3, možnosti výstupů nástroje jsou uvedené v tabulkách 5.2 a 5.4. Nástrojem provedená analýza závislostí zkoumá závislosti viz Tabulka 5.5.



Obrázek 5.1: ukázka části reportu generovaného nástrojem CodePro Analytix



Obrázek 5.2: GUI nástroje CodePro Analytix



Obrázek 5.3: cyklická závislost mezi balíčky v prostředí nástroje CodePro Analytix

kategorie	název funkcionality
nástroj	funguje jako plugin
nástroj	má GUI
nástroj	umí rozpoznat závislosti ze souborů s koncovkou `.java`
nástroj	umí pracovat se soubory s koncovkou .jar (obsahující soubory s koncovkou `.java`)
nástroj	umí pracovat s adresářovou strukturou obsahující soubory s koncovkou `.java`
nástroj	umí zobrazit graf závislostí

Tabulka 5.1: představení nástroje CodePro Analytix

kategorie	název funkcionality	poznámka k funkcionalitě
výstupy	lze generovat report o projektu	
výstupy	report svádí k řešení důležitých věcí	
výstupy	lze exportovat seznam závislostí	
výstupy	lze exportovat graf	`.gif`
výstupy	lze exportovat část grafu	`.gif`
výstupy	zobrazuje další metriky	

Tabulka 5.2: možnosti výstupů nástroje CodePro Analytix

kategorie	název funkcionality	poznámka k funkcionalitě
graf	lze zobrazit graf závislostí	
graf	lze zobrazit graf závislostí + použít filtry	
graf	uzly grafů lze manuálně rozmístit	
graf	na graf lze použít filtry	externí prvky, prvky které nejsou obsaženy v cyklu, závislosti směrem k, závislosti směrem od, závislosti vybraného prvku
graf	lze zobrazit všechny závislosti na vybraném prvku	
graf	zobrazit graf hierarchie vybraného prvku	balíčku
GUI	zobrazuje počet závislostí	

Tabulka 5.3: práce s grafem + prostředí nástroje CodePro Analytix

kategorie	název funkcionality	poznámka k funkcionalitě
seznam	závislostí balíčku na balíčkách	v rámci reportu
seznam	závislostí třídy na třídách	v rámci reportu
seznam	závislostí na vybraném prvku	v rámci reportu
seznam	dalších metrik	v rámci reportu

Tabulka 5.4: seznamy, které poskytuje nástroj CodePro Analytix

kategorie	název funkcionality	poznámka k funkcionalitě
umí najít	závislosti balíčku na balíčkách	
umí najít	závislosti třídy na třídách	
umí najít	cykly mezi balíčky	
umí najít	závislosti balíčku na balíčkách + filtr	
umí najít	závislosti balíčku na třídách	ručně
umí najít	závislosti třídy na balíčkách	ručně
umí najít	počet závislostí	
umí najít	závislosti balíčku na třídách + filtr	ručně
umí najít	závislosti na vybraném prvku (seznam)	v rámci reportu
umí najít	závislosti vybraného prvku (seznam)	v rámci reportu
umí najít	závislosti projektu	externí balíčky/třídy

Tabulka 5.5: závislosti, které detekuje nástroj CodePro Analytix

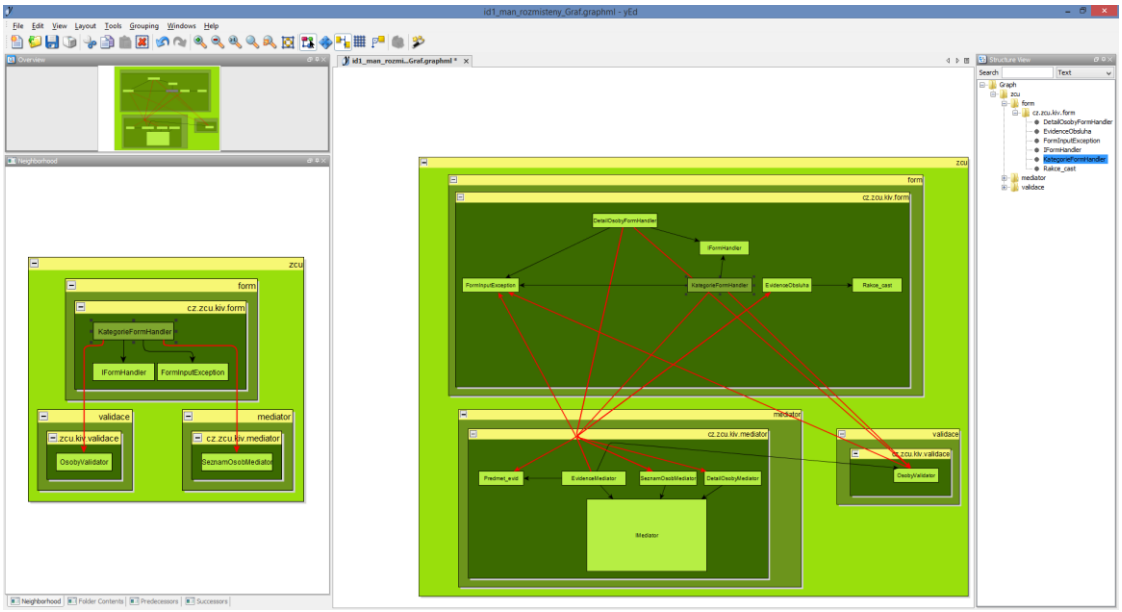
5.1.2 Degraph

Je nástroj, který se spouští v příkazové řádce, věnuje se grafickému zobrazení závislostí mezi balíčky a třídami Java aplikací. Poslední a zároveň testovaná verze 0.1.2 nástroje, byla vydána v roce 2015. Autor zůstává v problematice závislostí aktivní. Projekt se jeví udržovaný, u projektu jsou zmíněny i plány vylepšení nástroje do budoucna – například vlastní GUI. Vlastnosti nástroje jsou zmíněné v Tabulce 5.6.

Autor tohoto nástroje zmiňuje dva základní způsoby použití nástroje, přičemž otestování nástroje se bude věnovat jen jednomu-prvnímu z nich. Druhý způsob použití nástroje vyžaduje definování omezení závislostí a následné testování porušení těchto omezení.

První-testovaný způsob použití nástroje je provedení analýzy závislostí aplikace a následný výstup analýzy ve formě *graphml* souboru (Graph Markup Language). *Graphml* formát, používaný pro práci s grafy, je založený na XML formátu. Analýza závislostí v aplikacích se tedy provádí v příkazové řádce, nástroj umožňuje z analýzy vyjmout třídy, balíčky nebo obojí, které odpovídají zadanému vzoru. Ve výstupném souboru analýzy-grafu závislostí je obsažena hierarchie aplikace a závislosti jejích

prvků. Další možnosti výstupů jsou uvedeny v Tabulce 5.7. Autor nástroje se, výstupem analýzy ve standardním formátu, spoléhá na zobrazení grafu závislostí programem třetích stran. Další možnosti se tedy odvíjí od funkcionalit těchto programů. Autor doporučuje nástroj *yEd Graph Editor* viz Obrázek 5.4. Možnosti zobrazení grafu závislostí a práce s ním pomocí editoru *yEd* jsou popsané v Tabulce 5.8. Tento způsob využití nástroje je založen na tom, že menší prvky - třídy jsou graficky vnořené do větších prvků – balíčků viz Obrázek 5.4. Vy pak můžete věnovat pozornost závislostem přesahujícím do cizích balíčků nebo cyklickým závislostem. Z grafu závislostí lze vypozorovat závislosti (viz Tabulka 5.9) mezi různými prvky aplikace.



Obrázek 5.4: prostředí editoru yEd, vzhled uzlů grafu a struktura vynucená nástrojem Degraph

kategorie	název funkcionality	poznámka k funkcionalitě
nástroj	funguje jako samostatná aplikace	příkazová řádka + sw 3. stran
nástroj	umí rozpoznat závislosti ze souborů s koncovkou `.class`	
nástroj	umí pracovat se soubory s koncovkou .jar (obsahující soubory `.class`)	
nástroj	umí pracovat s adresářovou strukturou obsahující soubory s koncovkou `.class`	
nástroj	umí zobrazit graf závislostí	export + použití programu třetí strany

Tabulka 5.6: představení nástroje Degraph

kategorie	název funkcionality	poznámka k funkcionalitě
výstupy	lze exportovat graf	`.graphML`
výstupy	lze exportovat model	`.graphMl`
výstupy	lze exportovat část grafu	filtr analýzy

Tabulka 5.7: možnosti výstupů nástroje Degraph

kategorie	název funkcionality	poznámka k funkcionalitě
graf	lze zobrazit graf závislostí	exportovat do `.graphml`, samotné zobrazení grafu pak za pomoci programu 3.stran
graf	uzly grafů lze manuálně rozmístit	
graf	lze automaticky rozmístit uzly grafu	
graf	lze automaticky rozmístit uzly grafu více způsoby	
graf	na graf lze použít zoom	yEd (pohled 1:1, rozmístit na 1 obrazovku, zoom do vybrané oblasti, lupa u myši)
graf	je možné využít více pohledů na graf zároveň (např. další okno jiná perspektiva)	yEd má dva přídavné pohledy (perspektiva, sousedé)
graf	lze zobrazit všechny závislosti na vybraném prvku	
graf	lze zobrazit graf hierarchie vybraného prvku	

Tabulka 5.8: práce s grafem v editoru yEd

kategorie	název funkcionality	poznámka k funkcionalitě
umí najít	závislosti balíčku na balíčkách	ručně
umí najít	závislosti třídy na třídách	ručně
umí najít	závislosti balíčku na třídách	ručně
umí najít	závislosti třídy na balíčkách	ručně
umí najít	závislosti na vybraném prvku (graf)	
umí najít	závislosti vybraného prvku (graf)	
umí najít	závislosti projektu	externí balíčky/třídy
umí najít	závislosti balíčku na kontejneru	
umí najít	závislosti třídy na kontejneru	ručně
umí najít	závislosti kontejneru na kontejneru	ručně
umí najít	závislosti kontejneru na balíčkách	ručně
umí najít	závislosti kontejneru na třídách	ručně

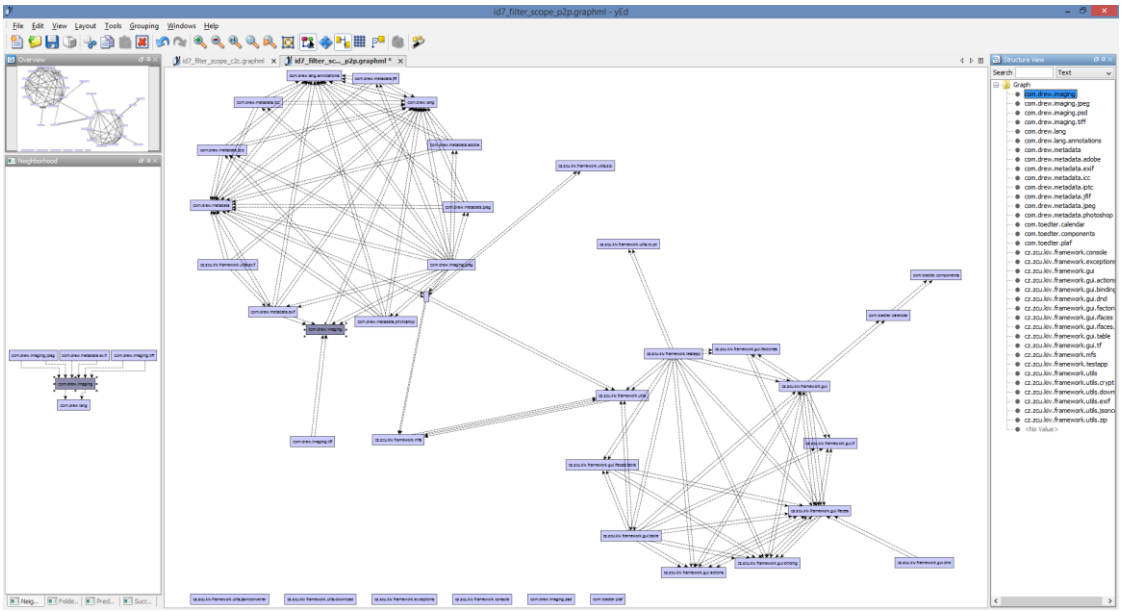
Tabulka 5.9: závislosti, které detekuje nástroj Degraph

5.1.3 DependencyFinder

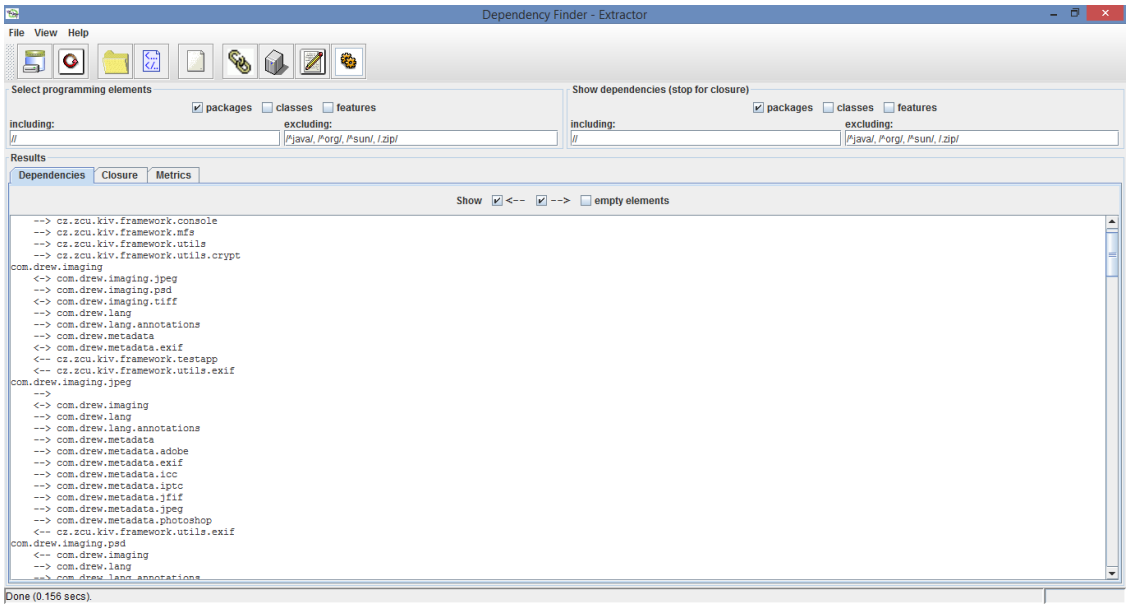
DependencyFinder je nástroj tvořený souborem aplikací, který za pomoci těchto aplikací získává závislosti z Java aplikací. Nástroj umí vypočítat různé druhy metrik a rozdíly mezi dvěma verzemi aplikace. Já se v rámci mého testu soustředím na prvně zmíněnou funkcionalitu nástroje. Další funkcionality nástroje jsou uvedené v Tabulce 5.10.

Poslední a zároveň testovaná verze 1.2.1-beta4 nástroje byla vydána v roce 2010. Projekt vypadá neudržovaně, nic nenasvědčuje tomu, že by autor projektu měl v plánu pokračovat s vývojem nástroje.

Nástroj je kombinací aplikace s GUI a podpůrných aplikací fungujících v rámci příkazové řádky. Za pomoci GUI nástroje, nebo přímo aplikací v příkazové řádce, lze po analýze závislostí exportovat model aplikace, který lze následně, podobně jako u předchozího nástroje, převést do formátu *graphml*. Další postup je stejný jako u předchozího nástroje. Ukázka grafu závislostí v prostředí nástroje yEd viz Obrázek 5.5, možnosti práce s grafem jsou uvedené v Tabulce 5.12. Možnosti výstupů nástroje jsou uvedeny v Tabulce 5.11. Analýzu závislostí je také možné provádět v rámci GUI nástroje viz Obrázek 5.6. Analýzou přímo v GUI jsme schopni získat seznamy závislostí na různých úrovních viz Tabulka 5.13. Kombinací výstupů z jednotlivých aplikací je nástroj *DependencyFinder* schopný detekovat závislosti viz Tabulka 5.14.



Obrázek 5.5: GUI nástroje yEd, automatické rozmístění uzlů grafu – circular; vzhled uzlů grafu a struktura vynucená nástrojem DependencyFinder



Obrázek 5.6: GUI nástroje DependencyFinder, závislosti mezi balíčky

Kategorie	**název funkcionality**	**poznámka k funkcionalitě**
Nástroj	funguje jako samostatná aplikace	
Nástroj	má GUI	
Nástroj	umí rozpoznat závislosti ze souborů s koncovkou `.class`	
Nástroj	umí pracovat se soubory s koncovkou .jar (obsahující soubory `.class`)	
nástroj	umí pracovat s adresářovou strukturou obsahující soubory s koncovkou `.class`	
nástroj	další podporované formáty souborů pro analýzu	`.war`
nástroj	umí zobrazit graf závislostí	exportovat do `.graphml` samotné zobrazení grafu pak za pomoci programu 3.stran

Tabulka 5.10: představení nástroje DependencyFinder

kategorie	**název funkcionality**	**poznámka k funkcionalitě**
výstupy	lze exportovat seznam závislostí	
výstupy	lze exportovat graf	(`graphML, HTML, RDF, txt`)
výstupy	lze exportovat model	(`xml, html, txt`)
výstupy	lze exportovat část grafu	filtr analýzy
výstupy	zobrazuje další metriky	

Tabulka 5.11: možnosti výstupů nástroje DependencyFinder

kategorie	název funkcionality	poznámka k funkcionalitě
graf	lze zobrazit graf závislostí	exportovat do `.graphml`, samotné zobrazení grafu pak za pomoci programu 3.stran
graf	uzly grafů lze manuálně rozmístit	
graf	lze automaticky rozmístit uzly grafu	
graf	lze automaticky rozmístit uzly grafu více způsoby	
graf	na graf lze použít zoom	yEd (pohled 1:1, rozmístit na 1 obrazovku, zoom do vybrané oblasti, lupa u myši)
graf	je možné využít více pohledů na graf zároveň (např. další okno jiná perspektiva)	yEd má dva přídavné pohledy (perspektiva, sousedé)
graf	lze zobrazit všechny závislosti na vybraném prvku	
graf	lze zobrazit graf hierarchie vybraného prvku	
GUI	zobrazuje počet závislostí	
GUI	umožňuje "proklik" přímo na konkrétní závislost	

Tabulka 5.12: práce s grafem, prostředím nástroje DependencyFinder a editorem yEd

kategorie	název funkcionality	poznámka k funkcionalitě
seznam	závislostí balíčku na balíčkách	
seznam	závislostí balíčku na balíčkách + filtr	
seznam	závislostí balíčku na třídách	
seznam	závislostí balíčku na třídách + filtr	
seznam	závislostí třídy na balíčkách	
seznam	závislostí třídy na balíčkách + filtr	
seznam	závislostí třídy na třídách	
seznam	závislostí třídy na třídách + filtr	
seznam	závislostí na vybraném prvku	
seznam	závislostí na vybraném prvku + filtr	
seznam	dalších metrik	

Tabulka 5.13: seznamy, které poskytuje nástroj DependencyFinder

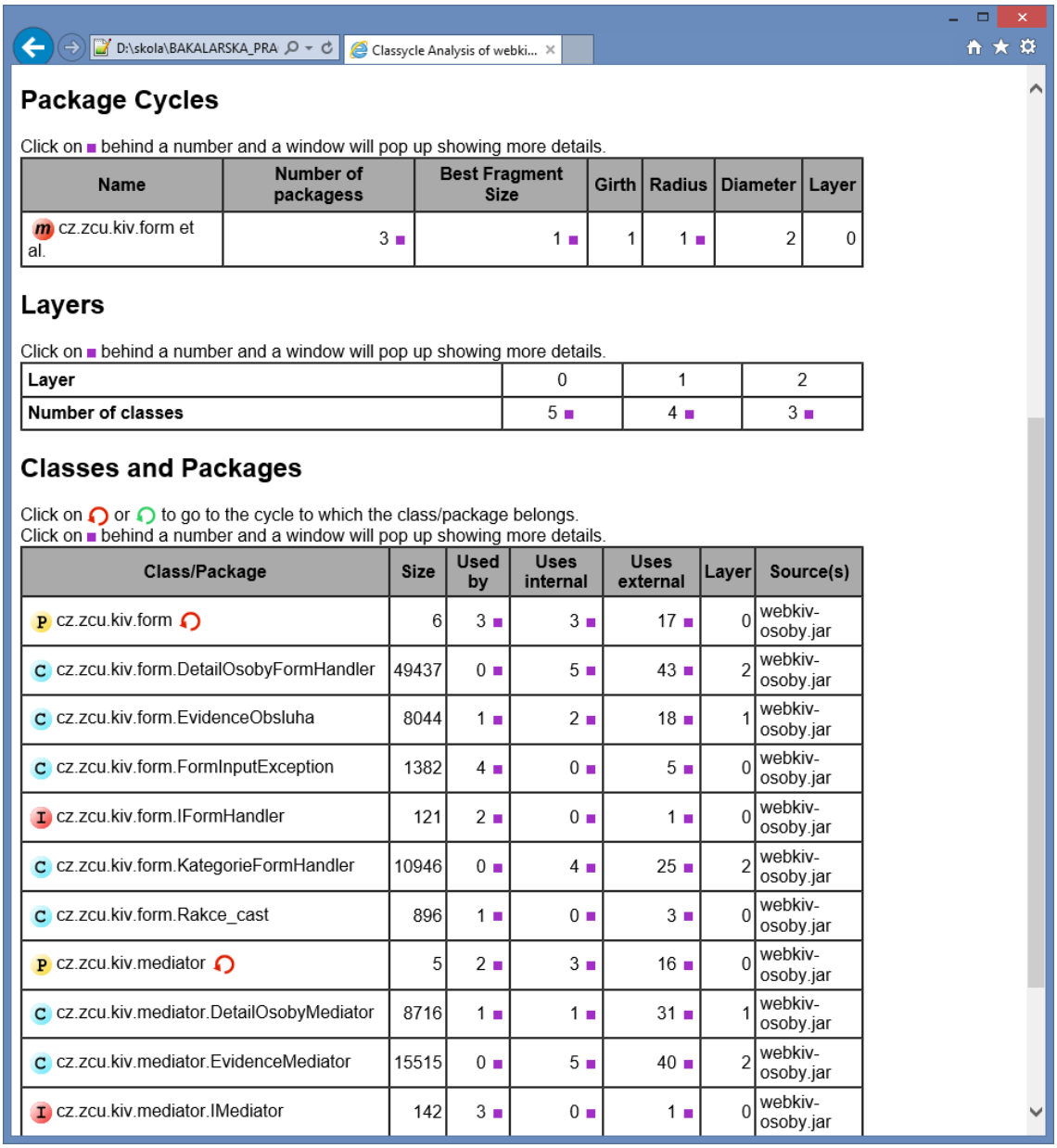
kategorie	název funkcionality	poznámka k funkcionalitě
umí najít	závislosti balíčku na balíčkách	
umí najít	závislosti třídy na třídách	
umí najít	cykly mezi balíčky	
umí najít	cykly mezi třídami	
umí najít	závislosti balíčku na balíčkách + filtr	
umí najít	závislosti balíčku na třídách	
umí najít	závislosti třídy na třídách + filtr	
umí najít	závislosti třídy na balíčkách	
umí najít	závislosti balíčku na třídách + filtr	
umí najít	závislosti třídy na balíčkách + filtr	
umí najít	závislosti na vybraném prvku (graf)	
umí najít	závislosti vybraného prvku (graf)	
umí najít	závislosti na vybraném prvku (seznam)	
umí najít	závislosti vybraného prvku (seznam)	
umí najít	závislosti projektu	externí balíčky/třídy
umí najít	závislosti metody na metodě	

Tabulka 5.14: závislosti, které kombinací výstupů aplikací detekuje nástroj DependencyFinder

5.1.4 Classycle

Classycle je nástroj, který se věnuje výhradně závislostem mezi třídami a balíčkami v Java aplikacích viz Tabulka 5.17. Hlavní funkcionality nástroje jsou detekce cyklických závislostí, XML report a kontrola dodržování vrstvené architektury aplikace. Nástroj se spouští z příkazové řádky. Další funkcionality z kategorie *nástroj* jsou uvedené v Tabulce 5.15.

Poslední a zároveň testovaná verze 1.4.2 nástroje byla vydána v roce 2014. Analýza závislostí v aplikacích se provádí v příkazové řádce, nástroj je možné spustit s více parametry, umožňuje z analýzy vyjmout třídy, balíčky nebo obojí, které odpovídají zadanému vzoru. Hlavním výstupem analýzy závislostí je XML report (viz Obrázek 5.7), ten je pak možné pomocí XSLT (eXtensible Stylesheet Language Transformations) souboru převést na HTML soubor a zobrazit jej v prohlížeči. Možnosti výstupů jsou uvedené v Tabulce 5.16. V reportu jsou obsaženy metriky a seznamy uvedené v Tabulce 5.18.



Package Cycles

Click on ■ behind a number and a window will pop up showing more details.

Name	Number of packagess	Best Fragment Size	Girth	Radius	Diameter	Layer
m cz.zcu.kiv.form et al.	3 ■	1 ■	1	1 ■	2	0

Layers

Click on ■ behind a number and a window will pop up showing more details.

Layer	0	1	2
Number of classes	5 ■	4 ■	3 ■

Classes and Packages

Click on ↻ or ↻ to go to the cycle to which the class/package belongs.
Click on ■ behind a number and a window will pop up showing more details.

Class/Package	Size	Used by	Uses internal	Uses external	Layer	Source(s)
P cz.zcu.kiv.form ↻	6	3 ■	3 ■	17 ■	0	webkiv-osoby.jar
C cz.zcu.kiv.form.DetailOsobyFormHandler	49437	0 ■	5 ■	43 ■	2	webkiv-osoby.jar
C cz.zcu.kiv.form.EvidenceObsluha	8044	1 ■	2 ■	18 ■	1	webkiv-osoby.jar
C cz.zcu.kiv.form.FormInputException	1382	4 ■	0 ■	5 ■	0	webkiv-osoby.jar
I cz.zcu.kiv.form.IFormHandler	121	2 ■	0 ■	1 ■	0	webkiv-osoby.jar
C cz.zcu.kiv.form.KategorieFormHandler	10946	0 ■	4 ■	25 ■	2	webkiv-osoby.jar
C cz.zcu.kiv.form.Rakce_cast	896	1 ■	0 ■	3 ■	0	webkiv-osoby.jar
P cz.zcu.kiv.mediator ↻	5	2 ■	3 ■	16 ■	0	webkiv-osoby.jar
C cz.zcu.kiv.mediator.DetailOsobyMediator	8716	1 ■	1 ■	31 ■	1	webkiv-osoby.jar
C cz.zcu.kiv.mediator.EvidenceMediator	15515	0 ■	5 ■	40 ■	2	webkiv-osoby.jar
I cz.zcu.kiv.mediator.IMediator	142	3 ■	0 ■	1 ■	0	webkiv-osoby.jar

Obrázek 5.7: část XML reportu aplikace classycle

kategorie	název funkcionality	poznámka k funkcionalitě
nástroj	funguje jako samostatná aplikace	Příkazový řádek + report v prohlížeči
nástroj	funguje i jako plugin	v pluginu nejsou implementovány změny ze třech novějších verzí samostatné aplikace classcycle (používá jádro verze 1.3.3 (2008))
nástroj	umí rozpoznat závislosti ze souborů s koncovkou `.class`	
nástroj	umí pracovat se soubory s koncovkou `.jar` (obsahující soubory `.class`)	
nástroj	umí pracovat s adresářovou strukturou obsahující soubory s koncovkou `.class`	
nástroj	další podporované formáty souborů pro analýzu	`.zip`, `.war`, `.ear`(neotestováno)

Tabulka 5.15: představení nástroje Classycle

kategorie	název funkcionality	poznámka k funkcionalitě
výstupy	lze generovat report o projektu	
výstupy	report svádí k řešení důležitých věcí	
výstupy	lze exportovat seznam závislostí	
výstupy	zobrazuje další metriky	

Tabulka 5.16: možnosti výstupů nástroje Classycle

kategorie	název funkcionality	poznámka k funkcionalitě
umí najít	závislosti balíčku na balíčkách	
umí najít	závislosti třídy na třídách	
umí najít	cykly mezi balíčky	
umí najít	cykly mezi třídami	
umí najít	závislosti třídy na balíčkách	ručně v rámci reportu
umí najít	počet závislostí	
umí najít	závislosti na vybraném prvku (seznam)	
umí najít	závislosti vybraného prvku (seznam)	v rámci reportu
umí najít	závislosti projektu	externí balíčky/třídy

Tabulka 5.17: závislosti, které detekuje nástroj Classycle

kategorie	název funkcionality	poznámka k funkcionalitě
seznam	závislostí balíčku na balíčkách	v rámci reportu
seznam	závislostí třídy na třídách	v rámci reportu
seznam	závislostí na vybraném prvku	v rámci reportu
seznam	dalších metrik	

Tabulka 5.18: seznamy, které poskytuje nástroj Classycle

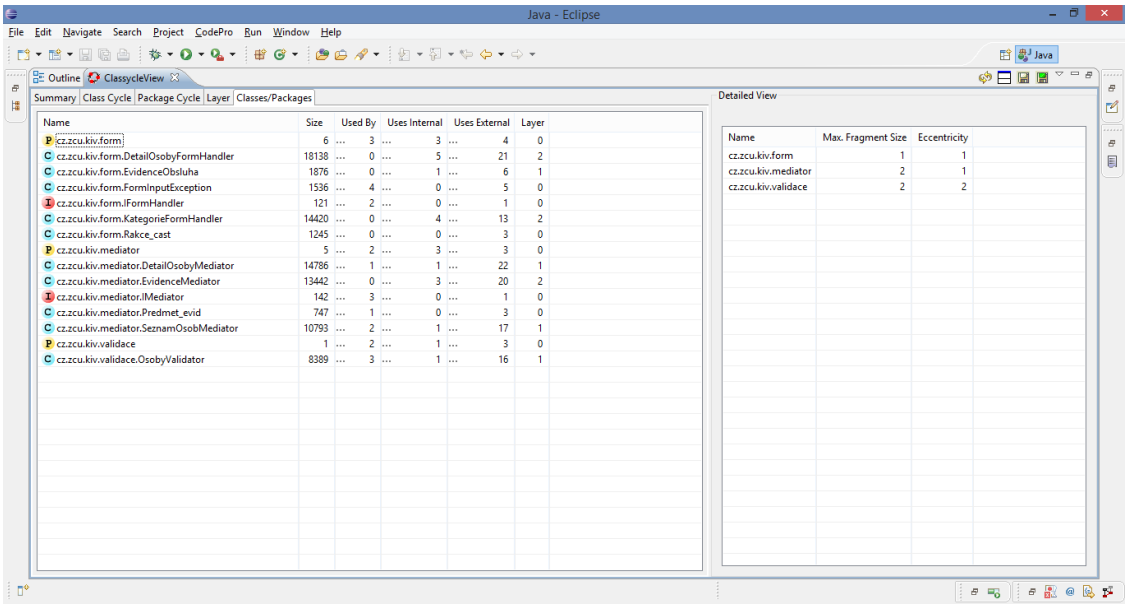
5.1.5 Classycle plugin

Je nástroj, který funguje jako plugin pro *Eclipse*. Všechny hlavní rysy a funkcionality nástroje jsou shodné jako u předchozího nástroje.

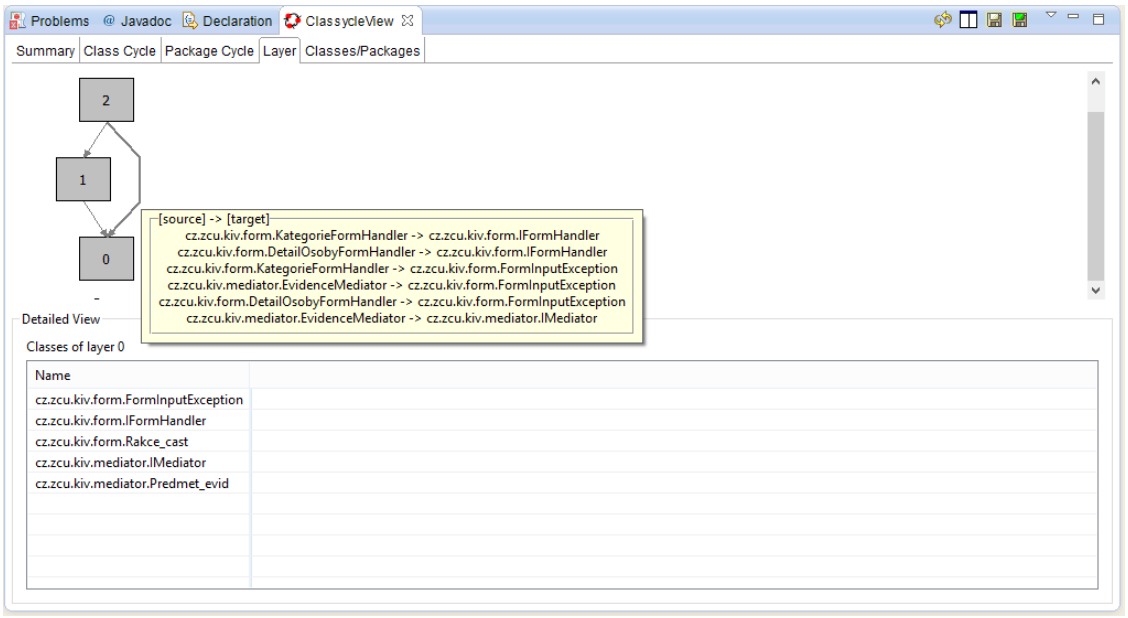
Poslední a zároveň testovaná verze 1.1.2 pluginu byla vydána v roce 2009. Tato verze pluginu používá jádro nástroje Classycle verze 1.3.3 z roku 2008. Od té doby jsou k dispozici 3 novější verze nástroje Classycle. Tyto novější verze, kromě fukce d*ependentOnlyon*, neobsahují žádné zásadní změny. Autor mi potvrdil, že vývoj pluginu nepokračuje.

Požadavky pluginu na verzi vývojového prostředí *Eclipse* jsou 3.0.0 a vyšší, plugin ale v nejnovější verzi *Eclipse*(Luna SR2 (4.4.2)) nefunguje. Plugin jsem otestoval ve verzi 4.3.2 vývojového prostředí *Eclipse*.

Plugin umí generovat stejný report závislostí (viz Obrázek 5.7) jako nástroj Classycle. Uživatelské rozhraní pluginu je v podstatě stejné viz Obrázek 5.8. Oproti reportu je zde navíc graf navrhované struktury vrstvené architektury aplikace viz Obrázek 5.9.



Obrázek 5.8: uživatelské rozhraní nástroje Classycle plugin



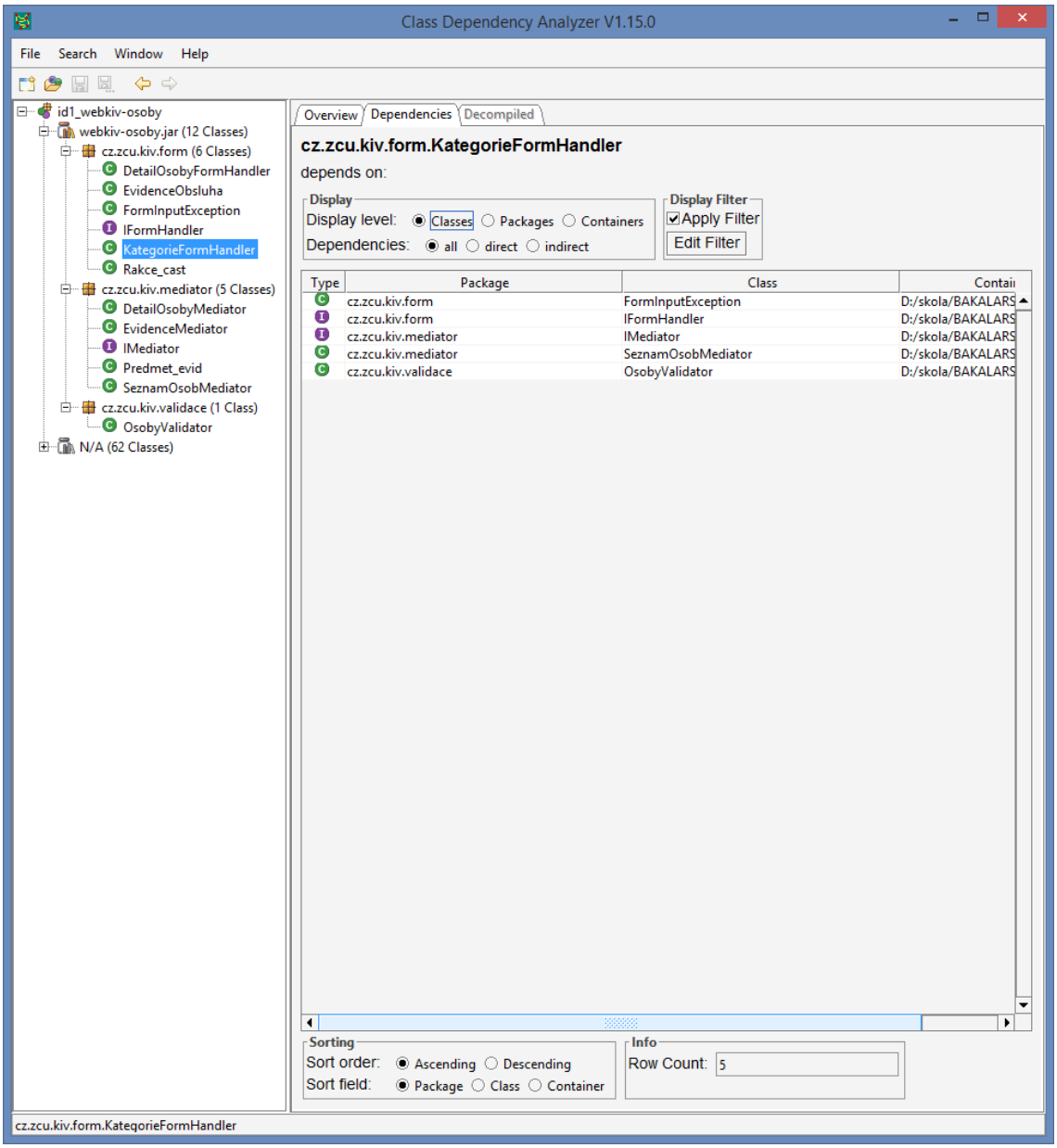
Obrázek 5.9: graf navrhované vrstvené architektury aplikce

5.1.6 Class Dependency Analyzer (CDA)

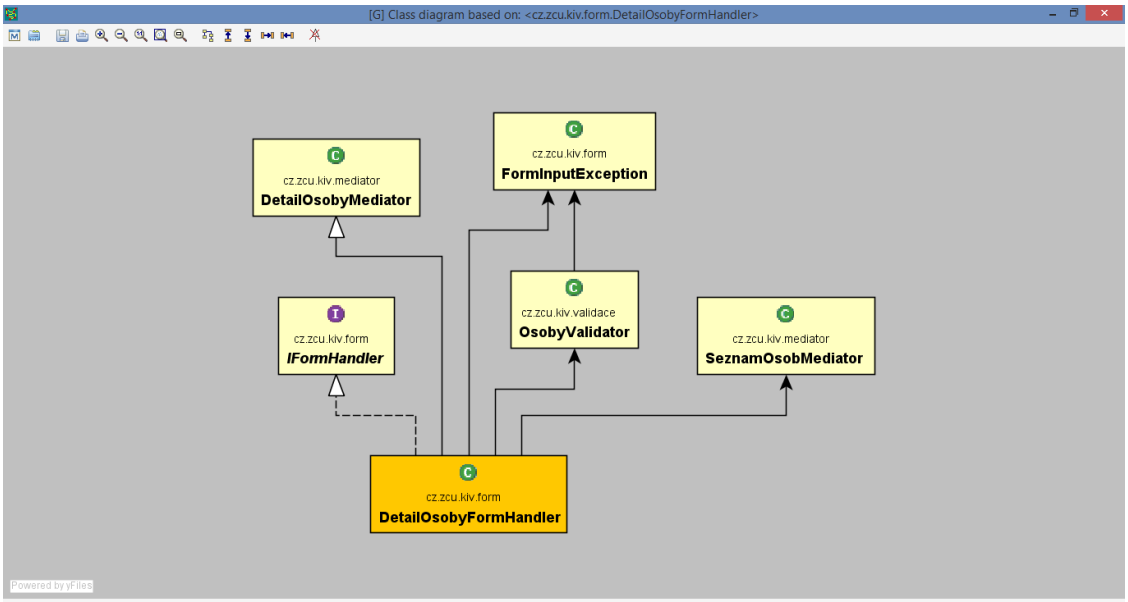
Nástroj se věnuje výhradně závislostem mezi třídami a balíčkami v Java aplikacích viz Tabulka 5.19 a grafickému zobrazení těchto závislostí. Další funkcionality z kategorie *nástroj* jsou uvedené v Tabulce 5.21.

Poslední a zároveň testovaná verze 1.15.0 nástroje byla vydána v roce 2014. Projekt vypadá udržovaně, na stránkách projektu jsou uvedené plány rozšíření nástroje do budoucna.

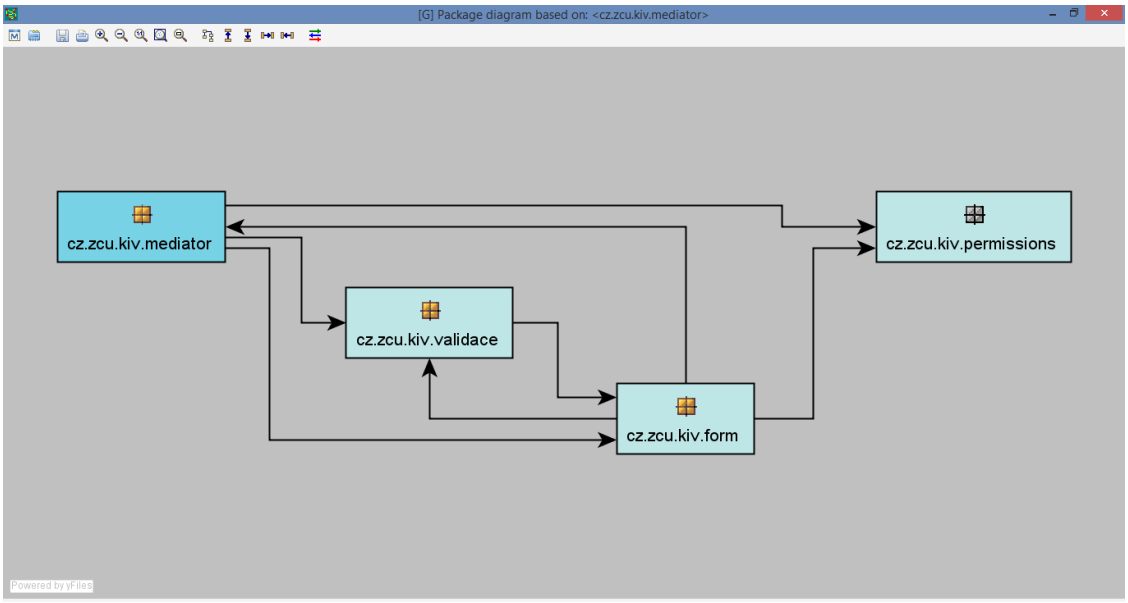
Analýza závislostí probíhá přímo v prostředí nástroje viz Obrázek 5.10. Nástroj umí v prostředí zobrazit seznamy závislostí viz Tabulka 5.23 a umožňuje se seznamem na různých úrovních dále pracovat. Nástroj umí zobrazit graf závislostí viz Tabulka 5.20. Například graf závislosti vybraného prvku - třídy viz Obrázek 5.11, balíčku viz Obrázek 5.12, nebo tzv. kontejneru viz Obrázek 5.14. Zobrazený graf závislostí lze pak exportovat – viz Tabulka 5.22. Nástroj umí detekovat i cyklické závislosti viz Obrázek 5.13.



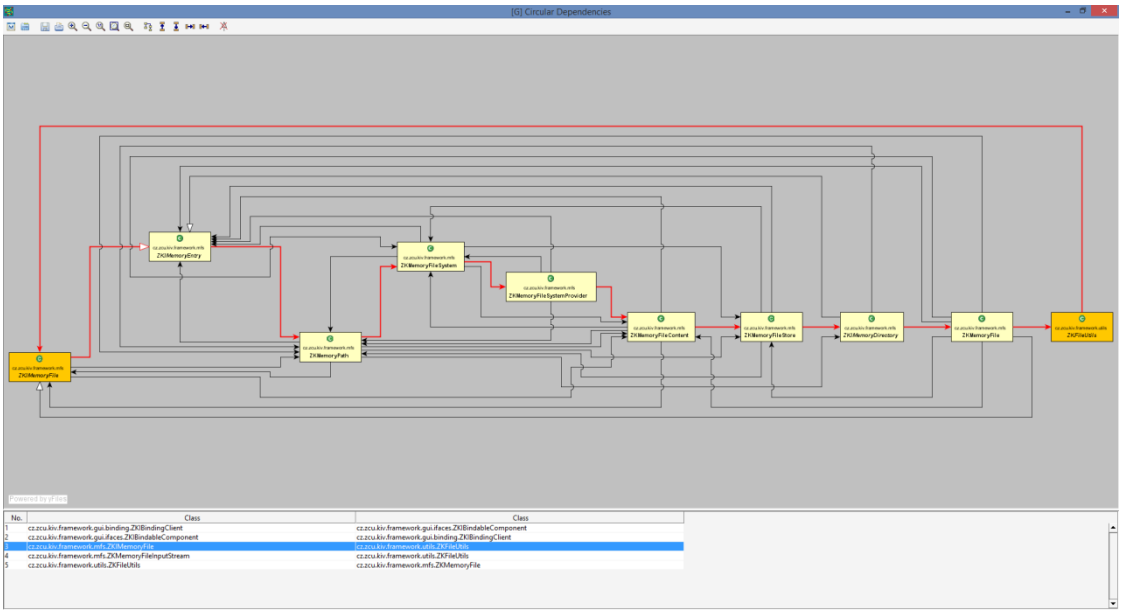
Obrázek 5.10: uživatelské rozhraní nástroje CDA



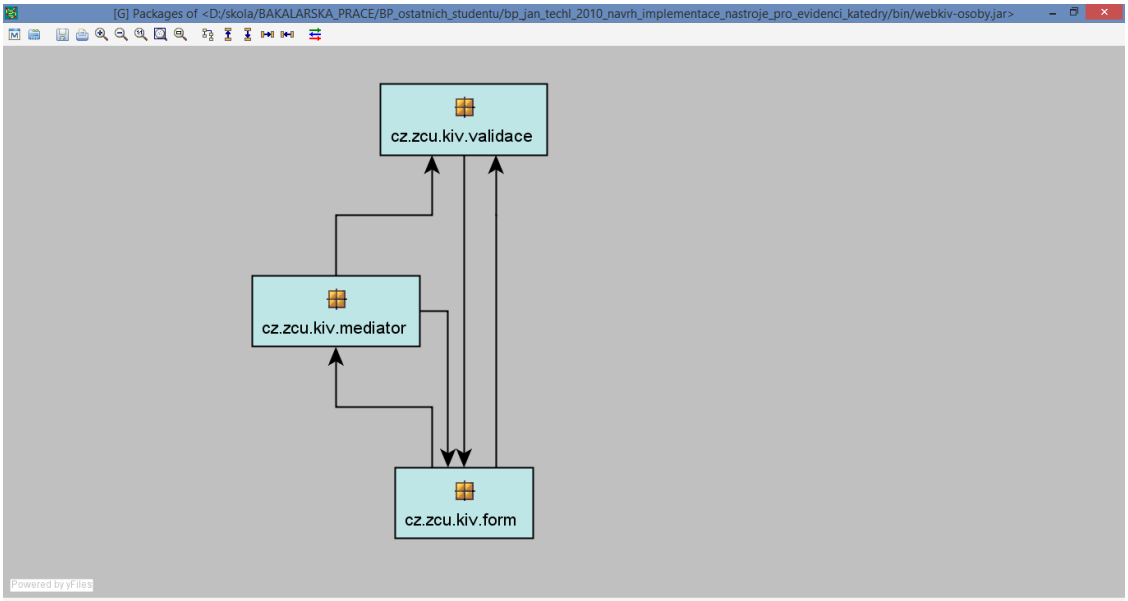
Obrázek 5.11: graf závislosti třídy v nástroji CDA



Obrázek 5.12: graf závislosti balíčku v nástroji CDA



Obrázek 5.13: seznam cyklických závislostí a zobrazení jedné z nich pomocí nástroje CDA



Obrázek 5.14: graf závislosti kontejneru (celého projektu) v nástroji CDA

kategorie	název funkcionality	poznámka k funkcionalitě
umí najít	závislosti balíčku na balíčkách	
umí najít	závislosti třídy na třídách	
umí najít	cykly mezi balíčky	mezi třídami jiných balíčků
umí najít	cykly mezi třídami	
umí najít	závislosti balíčku na balíčkách + filtr	
umí najít	závislosti balíčku na třídách	
umí najít	závislosti třídy na třídách + filtr	
umí najít	závislosti třídy na balíčkách	
umí najít	počet závislostí	
umí najít	závislosti balíčku na třídách + filtr	
umí najít	závislosti třídy na balíčkách + filtr	
umí najít	závislosti na vybraném prvku (graf)	
umí najít	závislosti vybraného prvku (graf)	
umí najít	závislosti na vybraném prvku (seznam)	
umí najít	závislosti vybraného prvku (seznam)	
umí najít	závislosti projektu	externí balíčky/třídy
umí najít	závislosti balíčku na kontejneru	
umí najít	závislosti balíčku na kontejneru + filtr	
umí najít	závislosti třídy na kontejneru	
umí najít	závislosti třídy na kontejneru + filtr	
umí najít	závislosti kontejneru na kontejneru	
umí najít	závislosti kontejneru na kontejneru + filtr	
umí najít	závislosti kontejneru na balíčkách	
umí najít	závislosti kontejneru na balíčkách + filtr	
umí najít	závislosti kontejneru na třídách	
umí najít	závislosti kontejneru na třídách + filtr	

Tabulka 5.19: závislosti, které detekuje nástroj CDA

kategorie	název funkcionality	poznámka k funkcionalitě
graf	lze zobrazit graf závislostí	
graf	lze zobrazit graf závislostí + použít filtry	
graf	uzly grafů lze manuálně rozmístit	
graf	lze automaticky rozmístit uzly grafu	
graf	lze automaticky rozmístit uzly grafu více způsoby	
graf	na graf lze použít zoom	(pohled 1:1, rozmístit na 1 obrazovku, zoom vybrané oblasti)
graf	na graf lze použít filtry	zvlášť vybraný prvek, třídy, balíčky, kontejnery; filtr dědičnosti
graf	lze zobrazit všechny závislosti na vybraném prvku	
graf	lze zobrazit graf hierarchie vybraného prvku	
GUI	zobrazuje počet závislostí	

Tabulka 5.20: práce s grafem + prostředí nástroje CDA

kategorie	název funkcionality	poznámka k funkcionalitě
nástroj	umí dekompilovat třídu (z `.class`) a zobrazit její zdrojový kód	
nástroj	funguje jako samostatná aplikace	
nástroj	má GUI	
nástroj	umí rozpoznat závislosti ze souborů s koncovkou `.class`	
nástroj	umí pracovat se soubory s koncovkou .jar (obsahující soubory `.class`)	
nástroj	umí pracovat s adresářovou strukturou obsahující soubory s koncovkou `.class`	
nástroj	další podporované formáty souborů pro analýzu	(`.zip`, `.war`, Eclipse projekt)
nástroj	umí zobrazit graf závislostí	

Tabulka 5.21: představení nástroje CDA

kategorie	název funkcionality	poznámka k funkcionalitě
výstupy	lze exportovat graf	(`.jpg`, `.gif`, `.png`, `.bmp`, `.gml`)
výstupy	lze exportovat model	(`.odem`)
výstupy	lze exportovat část grafu	(`.jpg`, `.gif`, `.png`, `.bmp`, `.gml`)

Tabulka 5.22: možnosti výstupů nástroje CDA

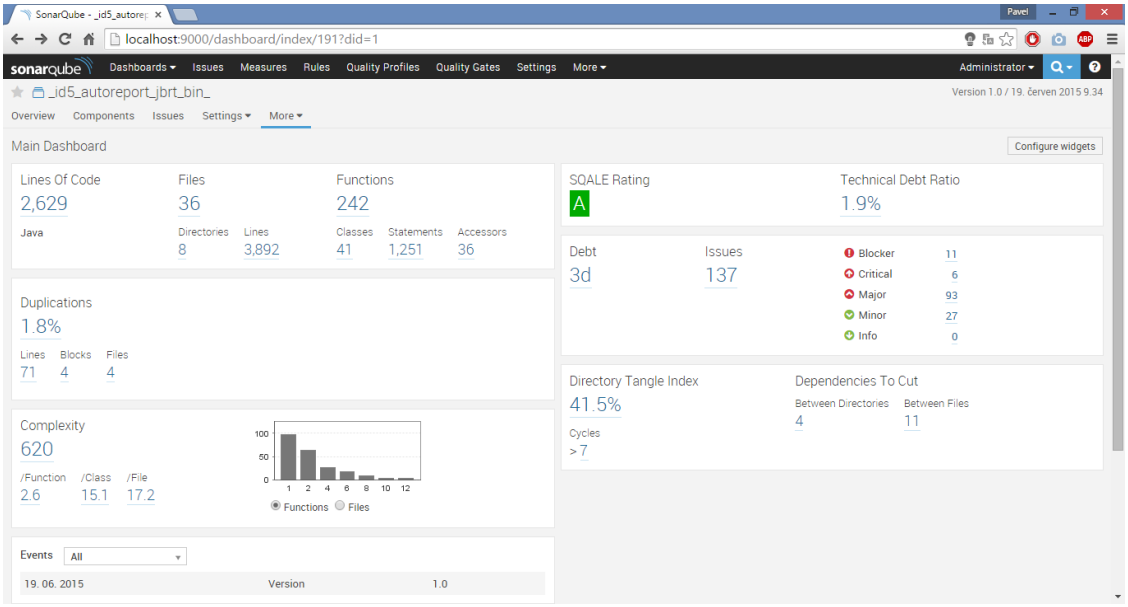
kategorie	název funkcionality
seznam	závislostí balíčku na balíčkách
seznam	závislostí balíčku na balíčkách + filtr
seznam	závislostí balíčku na kontejneru
seznam	závislostí balíčku na kontejneru + filtr
seznam	závislostí balíčku na třídách
seznam	závislostí balíčku na třídách + filtr
seznam	závislostí kontejneru na balíčkách
seznam	závislostí kontejneru na balíčkách + filtr
seznam	závislostí kontejneru na kontejneru
seznam	závislostí kontejneru na kontejneru + filtr
seznam	závislostí kontejneru na třídách
seznam	závislostí kontejneru na třídách + filtr
seznam	závislostí třídy na balíčkách
seznam	závislostí třídy na balíčkách + filtr
seznam	závislostí třídy na kontejneru
seznam	závislostí třídy na kontejneru + filtr
seznam	závislostí třídy na třídách
seznam	závislostí třídy na třídách + filtr
seznam	závislostí na vybraném prvku
seznam	závislostí na vybraném prvku + filtr

Tabulka 5.23: seznamy, které poskytuje nástroj CDA

5.1.7 *SonarQube + SonarQube plugin*

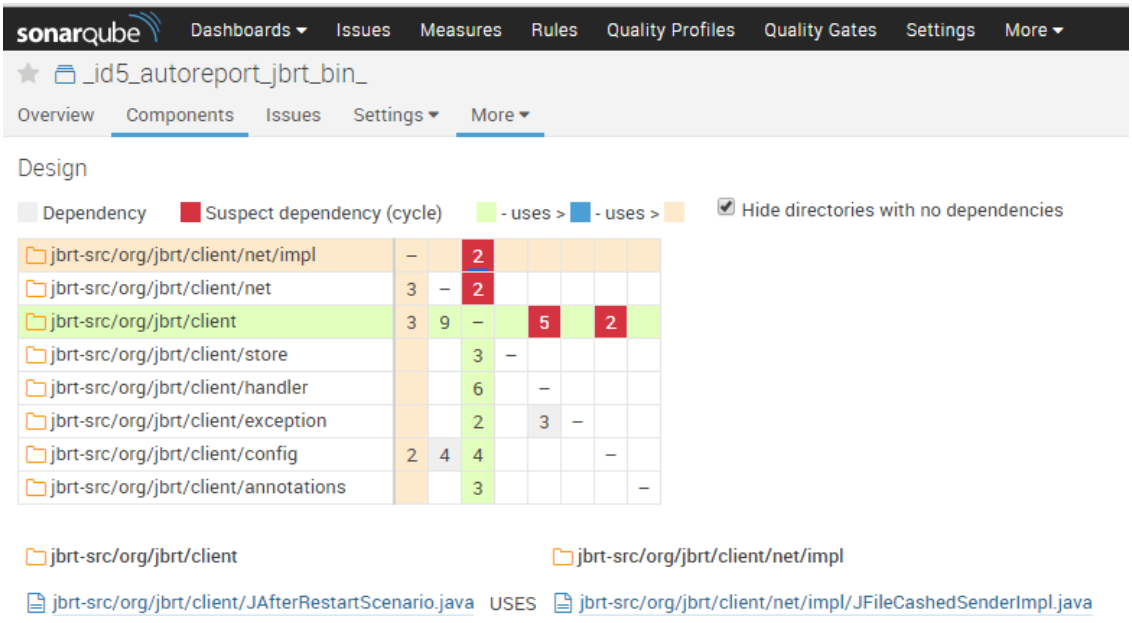
SonarQube je velmi komplexní nástroj, podporující týmovou spolupráci, pro kontrolu a zvyšování kvality kódu nejen v Java aplikacích. Podporuje více než 20 programových jazyků. Funguje na bázi klient-server. Poslední a zároveň testovaná verze nástroje byla vydána v roce 2015, projekt je velmi udržovaný.

Na server se dá připojit přes webový prohlížeč, nebo pomocí pluginu vývojového prostředí *Eclipse* či *IntelliJ*. Na serveru je spuštěna webová aplikace, server může spravovat více projektů zároveň. Úvodní stránka každého projektu (viz Obrázek 5.15) slouží k zobrazení důležitých informací a metrik projektu. Výstupy projektu jsou také uvedeny v Tabulce 5.25.



Obrázek 5.15: úvodní stránka projektu - webová aplikace nástroje sonarQube

Jednou z mnoha funkcí tohoto nástroje je i schopnost zobrazení matice závislostí viz Obrázek 5.16. V matici závislostí jsou uvedeny počty závislostí všech kombinací balíčků či tříd (viz Tabulka 5.27). Každou ze závislostí (viz Tabulka 5.28) lze zobrazit. Závislosti nad diagonálou značí cyklické závislosti. Funkcionality nástroje, které se netýkají závislostí, nebyly prozkoumány. Sledované funkcionality z kategorie *nástroj* jsou uvedené v Tabulce 5.24. V rámci testu sledované funkcionality z kategorie *GUI* jsou uvedené v Tabulce 5.26.



Obrázek 5.16: matice závislostí nástroje sonarQube

kategorie	název funkcionality	poznámka k funkcionalitě
nástroj	umí zobrazit matici závislostí	
nástroj	funguje jako samostatná aplikace	client-server
nástroj	funguje jako plugin	
nástroj	má GUI	
nástroj	umí rozpoznat závislosti ze souborů s koncovkou `.class`	jen v kombinaci se soubory `.java`
nástroj	umí rozpoznat závislosti ze souborů s koncovkou `.java`	jen v kombinaci se soubory `.class`
nástroj	umí pracovat s adresářovou strukturou obsahující soubory s koncovkou `.class`	
nástroj	umí pracovat s adresářovou strukturou obsahující soubory s koncovkou `.java`	

Tabulka 5.24: představení nástroje sonarQube

kategorie	název funkcionality	poznámka k funkcionalitě
výstupy	lze generovat report o projektu	
výstupy	report svádí k řešení důležitých věcí	
výstupy	zobrazuje další metriky	

Tabulka 5.25: možnosti výstupů nástroje sonarQube

kategorie	název funkcionality	poznámka k funkcionalitě
GUI	zobrazuje počet závislostí	
GUI	umožňuje "proklik" přímo na konkrétní závislost	

Tabulka 5.26: prostředí nástroje sonarQube

kategorie	název funkcionality	poznámka k funkcionalitě
umí najít	závislosti balíčku na balíčkách	
umí najít	závislosti třídy na třídách	
umí najít	cykly mezi balíčky	
umí najít	cykly mezi třídami	
umí najít	závislosti balíčku na třídách	
umí najít	počet závislostí	
umí najít	závislosti na vybraném prvku (seznam)	
umí najít	závislosti vybraného prvku (seznam)	

Tabulka 5.27: závislosti, které detekuje nástroj sonarQube

kategorie	název funkcionality	poznámka k funkcionalitě
seznam	závislostí balíčku na balíčkách	ručně
seznam	závislostí balíčku na třídách	ručně
seznam	závislostí třídy na třídách	ručně a jen v rámci balíku
seznam	závislostí na vybraném prvku	
seznam	dalších metrik	
seznam	dalších metrik	

Tabulka 5.28: seznamy, které poskytuje nástroj SonarQube

5.2 Hodnocení nástrojů

Každé funkcionalitě z množiny sledovaných funkcionalit jsem na základě vlastního uvážení přiřadil důležitost. Sloupec D v tabulkách této kapitoly značí důležitost funkcionality, která nabývá hodnot 1-6, kde 1 je nejdůležitější.

Struktura aplikací (počet balíčků a tříd) nakonec významně neprodloužila dobu analýzy závislostí. Doby běhů analýz se pohybovaly vždy do únosné jedné minuty. Nejdéle, 50 sekund, trvala analýza cyklických závislostí na úrovni tříd u aplikace id7 nástrojem *DependencyFinder*.

5.2.1 *CodePro Analytix*

Nástroj nezvládá (nebo zvládá s připomínkami) funkcionality uvedené v Tabulce 5.29. Tyto funkcionality řadím do prvních dvou kategorií důležitosti.

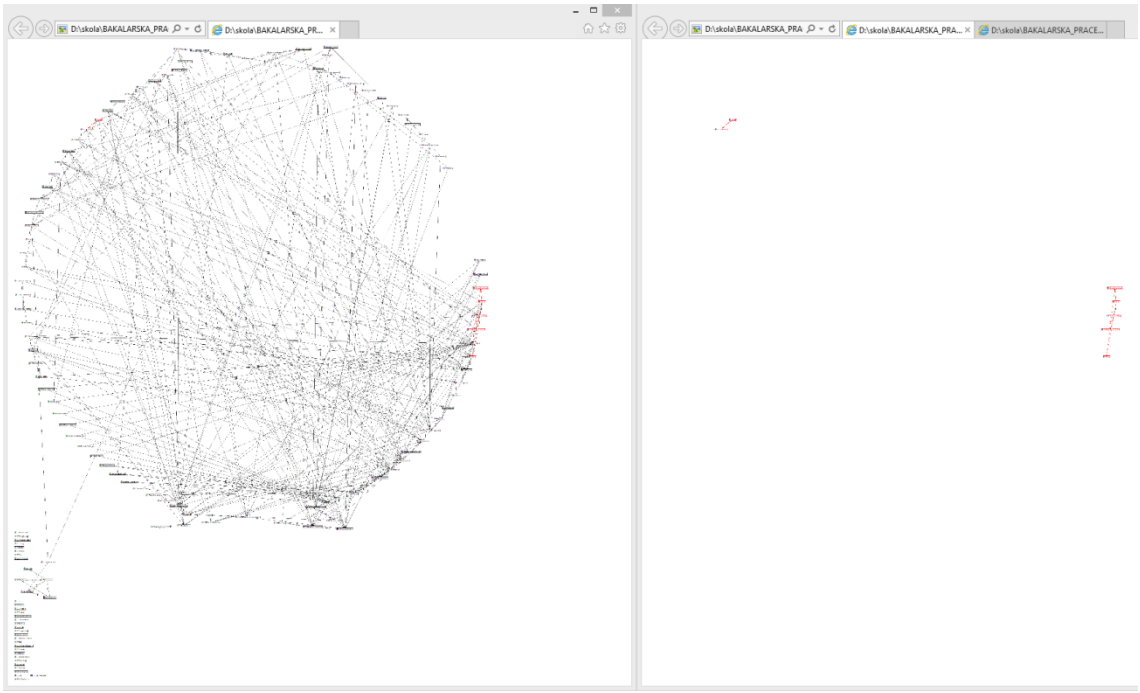
kategorie	D	umí / je pravda	název funkcionality	poznámka k funkcionalitě
nástroj	1	ne	umí zobrazit matici závislostí	
nástroj	1	ne	umí rozpoznat závislosti ze souborů s koncovkou `.class`	
umí najít	1	ne	cykly mezi třídami	
nástroj	2	ne	funguje jako samostatná aplikace	
nástroj	2	ne	umí pracovat se soubory s koncovkou .jar (obsahující soubory `.class`)	
nástroj	2	ne	umí pracovat s adresářovou strukturou obsahující soubory s koncovkou `.class`	
seznam	2	ne	závislostí balíčku na balíčkách + filtr	
seznam	2	ne	závislostí balíčku na třídách	lze ručně v rámci reportu
seznam	2	ne	závislostí třídy na balíčkách	lze ručně v rámci reportu
seznam	2	ne	závislostí třídy na třídách + filtr	
umí najít	2	ne	závislosti třídy na třídách + filtr	jen v rámci balíku

Tabulka 5.29: důležité funkcionality, které nástroj CodePro Analytix nezvládá

U nástroje jsem uvítal jednoduchost instalace a konfigurace nástroje. U nástroje se mi líbil výstup analýzy v rámci HTML reportu viz Obrázek 5.1. Tento report je konfigurovatelný – například je možné do reportu nezahrnout informace o třídách a ponechat jen informace o balíčkách. Report je velmi obsáhlý, u každého prvku – třídy, nebo balíčku jsou informace o prvcích, které prvek využívá a zároveň, které prvky využívají tento prvek. V reportu jsou červeně zvýrazněné cyklické závislosti mezi balíčky a to jak uzly grafu v grafu závislostí, tak jednotlivé balíčky v textové části. Zvýraznění cyklických závislostí svádí k jejich vyřešení.

Ovládání pluginu přímo ve vývojovém prostředí *Eclipse* bych nenazval jako intuitivní. Nebyl jsem spokojen ani se svižností prostředí. Několikrát se mi stalo, že Graf závislostí, který měl velké množství uzlů (nad 500), se nepodařilo vyexportovat. Také se stávalo, že po té, co byl vygenerován report, nebyla práce s grafem přímo v prostředí korektní a bylo nutné analýzu závislostí opakovat. Na grafu závislostí se mi naopak líbil zobrazený počet závislostí mezi uzly grafu a to v obou směrech – od uzlu i k uzlu (viz Obrázek 5.3).

Dále jsem nebyl spokojený s automatickým rozmístěním uzlů grafu závislosti. Pokud měl graf velké množství uzlů, rozložení grafu o ničem nevypovídalo (viz Obrázek 5.17). Na graf závislostí není možné použít zoom, na velký graf závislostí se tedy nelze podívat ze vzdálené perspektivy. Práci s velkým grafem v podstatě umožňovaly hlavně možnosti nástroje filtrovat uzly grafu viz možnosti filtrů uvedené v Tabulka 5.3.



Obrázek 5.17: Aplikování filtru nástrojem CodePro Analytix– zobrazení jen uzlů grafu obsažených v cyklu – žádné automatické přeskupení uzlů

5.2.2 Degraph

Nástroj nezvládá (nebo zvládá s připomínkami) funkcionality uvedené v Tabulce 5.30. Tyto funkcionality řadím do prvních dvou kategorií důležitosti. Tučně zvýrazněné funkcionality v Tabulce 5.30 nástroj nezvládá jako jediný. Jako výhodu tohoto nástroje shledávám schopnost vynutit ve výstupném graphML souboru strukturu tříd vnořených do balíčků viz Obrázek 5.4. Líbila se mi možnost práce v editoru *yEd*. Konkrétně více pohledů na graf zároveň.

Samotnou práci s nástrojem hodnotím jako nepohodlnou a neintuitivní, u testování některých aplikací bylo problematické vynutit správnou strukturu grafu. Ta se vynucuje přes konfigurační soubor, poté se příkazem v příkazovém řádku spustí analýza.

kategorie	D	umí / je pravda	název funkcionality	poznámka k funkcionalitě
nástroj	1	ne	umí zobrazit matici závislostí	
nástroj	1	ne	umí rozpoznat závislosti ze souborů s koncovkou `.java`	
seznam	**1**	**ne**	**závislostí balíčku na balíčkách**	
seznam	**1**	**ne**	**závislostí třídy na třídách**	
umí najít	**1**	**ne**	**cykly mezi balíčky**	**lze vypozorovat z grafu**
umí najít	1	ne	cykly mezi třídami	lze vypozorovat z grafu
graf	2	ne	lze zobrazit graf závislostí + použít filtry	
nástroj	2	ne	má GUI	
nástroj	2	ne	umí pracovat se soubory s koncovkou .jar (obsahující soubory s koncovkou `.java`)	
nástroj	2	ne	umí pracovat s adresářovou strukturou obsahující soubory s koncovkou `.java`	
seznam	2	ne	závislostí balíčku na balíčkách + filtr	
seznam	2	ne	závislostí balíčku na třídách	
seznam	2	ne	závislostí třídy na balíčkách	
seznam	2	ne	závislostí třídy na třídách + filtr	
umí najít	2	ne	závislosti balíčku na balíčkách + filtr	
umí najít	2	ne	závislosti třídy na třídách + filtr	
výstupy	2	ne	lze generovat report o projektu	
výstupy	2	ne	lze exportovat seznam závislostí	

Tabulka 5.30: důležité funkcionality, které nástroj Degraph nezvládá

5.2.3 DependencyFinder

Nástroj nezvládá (nebo zvládá s připomínkami) funkcionality uvedené v Tabulce 5.31. Tyto funkcionality řadím do prvních dvou kategorií důležitosti.

Nástroji bych vytkl, že není možné provést kompletní analýzu závislostí – včetně detekce cyklických závislostí přímo v GUI. Je nutné zkombinovat výstupy několika nástrojů spouštěných v příkazovém řádku. Jinak bych práci s nástrojem v GUI vyzdvihl. Není zde sice zobrazen graf závislostí, závislosti jsou ale přehledně textově vypsány (viz Obrázek 5.6). Tento textový popis závislostí je díky rozsáhlým filtrům, včetně možnosti využití regulárních výrazů, zejména u menších aplikací velmi efektivní. Jedním z výstupů tohoto nástroje, stejně jako nástroje *Degraph* je graf závislostí ve formátu *graphml*. Výstup nástroje *Degraph* využívá navíc funkci vnořených prvků.

Nástroj jako jediný z testovaných umí vypsat i závislosti metod.

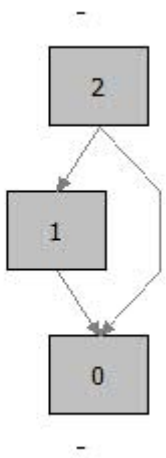
kategorie	D	umí / je pravda	název funkcionality	poznámka k funkcionalitě
nástroj	1	ne	umí zobrazit matici závislostí	
nástroj	1	ne	umí rozpoznat závislosti ze souborů s koncovkou `.java`	
graf	2	ne	lze zobrazit graf závislostí + použít filtry	
nástroj	2	ne	umí pracovat se soubory s koncovkou `.jar` (obsahující soubory s koncovkou `.java`)	
nástroj	2	ne	umí pracovat s adresářovou strukturou obsahující soubory s koncovkou `.java`	
výstupy	2	ne	lze generovat report o projektu	

Tabulka 5.31: důležité funkcionality, které nástroj DependencyFinder nezvládá

5.2.4 Classycle + classycle plugin

Nástroj *Classycle* nezvládá (nebo zvládá s připomínkami) funkcionality uvedené v Tabulce 5.32. Tyto funkcionality řadím do prvních dvou kategorií důležitosti. Oběma nástrojům schází možnost zobrazení grafu závislostí. Nástroje se soustředí výhradně na hledání závislostí v Java aplikacích, práce s nimi byla velmi intuitivní a rychlá. Během testování nenastal s nástroji, na rozdíl od nástrojů *CodePro Analytix* a *SonarQube*, žádný problém. Chválím generovaný XML report (viz Obrázek 5.7). Report je přehledný, pokud je třída, nebo balíček členem některého z cyklů (cyklická závislost), je u prvku znázorněný symbol cyklu, navíc odlišený barvou symbolu.

Classycle plugin sice umožňuje export grafu navrhované vrstvené architektury aplikace (viz obrázek 5.9), jeho využitelnost je ale značně omezená, protože oproti prostředí pluginu neobsahuje žádný popis obsahu navržených vrstev (viz Obrázek 5.18).



Obrázek 5.18: nevypovídající exportovaný graf navrhované vrstvené architektury aplikace

kategorie	D	umí / je pravda	název funkcionality	poznámka k funkcionalitě
graf	1	ne	lze zobrazit graf závislostí	
nástroj	1	ne	umí zobrazit matici závislostí	
nástroj	1	ne	umí rozpoznat závislosti ze souborů s koncovkou `.java`	
nástroj	1	ne	umí zobrazit graf závislostí	
graf	2	ne	lze zobrazit graf závislostí + použít filtry	
nástroj	2	ne	má GUI	
nástroj	2	ne	umí pracovat se soubory s koncovkou `.jar` (obsahující soubory s koncovkou `.java`)	
nástroj	2	ne	umí pracovat s adresářovou strukturou obsahující soubory s koncovkou `.java`	
seznam	2	ne	závislostí balíčku na balíčkách + filtr	lze vyloučit prvky z analýzy (ty jsou pak v seznamu externích závislostí)
seznam	2	ne	závislostí balíčku na třídách	lze ručně v rámci reportu
seznam	2	ne	závislostí třídy na balíčkách	lze ručně v rámci reportu
seznam	2	ne	závislostí třídy na třídách + filtr	lze vyloučit prvky z analýzy (ty jsou pak v seznamu externích závislostí)
umí najít	2	ne	závislosti balíčku na balíčkách + filtr	lze vyloučit prvky z analýzy (ty jsou pak v seznamu externích závislostí)
umí najít	2	ne	závislosti balíčku na třídách	
umí najít	2	ne	závislosti třídy na třídách + filtr	lze vyloučit prvky z analýzy (ty jsou pak v seznamu externích závislostí)
výstupy	2	ne	lze exportovat graf	

Tabulka 5.32: důležité funkcionality, které nástroj Classycle nezvládá

kategorie	D	umí / je pravda	název funkcionality	poznámka k funkcionalitě
graf	1	ne	lze zobrazit graf závislostí	
nástroj	1	ne	umí zobrazit matici závislostí	
nástroj	1	ne	umí rozpoznat závislosti ze souborů s koncovkou `.class`	
nástroj	1	ne	umí zobrazit graf závislostí	
graf	2	ne	lze zobrazit graf závislostí + použít filtry	
nástroj	2	ne	umí pracovat se soubory s koncovkou `.jar` (obsahující soubory `.class`)	Eclipse
nástroj	2	ne	umí pracovat s adresářovou strukturou obsahující soubory s koncovkou `.class`	Eclipse
seznam	2	ne	závislostí balíčku na balíčkách + filtr	lze vyloučit prvky z analýzy (ty jsou pak v seznamu externích závislostí)
seznam	2	ne	závislostí balíčku na třídách	lze ručně v rámci reportu
seznam	2	ne	závislostí třídy na balíčkách	lze ručně v rámci reportu
seznam	2	ne	závislostí třídy na třídách + filtr	lze vyloučit prvky z analýzy (ty jsou pak v seznamu externích závislostí)
umí najít	2	ne	závislosti balíčku na balíčkách + filtr	lze vyloučit prvky z analýzy (ty jsou pak v seznamu externích závislostí)
umí najít	2	ne	závislosti balíčku na třídách	
umí najít	2	ne	závislosti třídy na třídách + filtr	lze vyloučit prvky z analýzy (ty jsou pak v seznamu externích závislostí)

Tabulka 5.33: důležité funkcionality, které nástroj Classycle plugin nezvládá

5.2.5 Class Dependency Analyzer (CDA)

Nástroj nezvládá (nebo zvládá s připomínkami) funkcionality uvedené v Tabulce 5.34. Tyto funkcionality řadím do prvních dvou kategorií důležitosti.

kategorie	D	umí / je pravda	název funkcionality	poznámka k funkcionalitě
nástroj	1	ne	umí zobrazit matici závislostí	
nástroj	1	ne	umí rozpoznat závislosti ze souborů s koncovkou `.java`	
nástroj	2	ne	umí pracovat se soubory s koncovkou `.jar` (obsahující soubory s koncovkou `.java`)	
nástroj	2	ne	umí pracovat s adresářovou strukturou obsahující soubory s koncovkou `.java`	
výstupy	2	ne	lze generovat report o projektu	
výstupy	2	ne	lze exportovat seznam závislostí	lze ale seznam vykopírovat do excelu (bez názvů sloupců)

Tabulka 5.34: důležité funkcionality, které nástroj CDA nezvládá

Práce s nástrojem byla i přes širokou škálu funkcionalit velmi pohodlná intuitivní a svižná. Nástroj poskytuje ideální kombinaci grafického zobrazení závislostí na vysoké úrovni – jako jediný z testovaných nástrojů zobrazuje graf závislostí pomocí UML (Unified Modeling Language) diagramu, navíc ve vektorové grafice (viz Obrázkek 5.11, Obrázek 5.12 a Obrázek 5.13) a zobrazení seznamu závislostí a práce s ním přímo v GUI (viz Obrázek 5.10).

Nástroj jako jediný umí dekompilovat `class` soubor a zobrazit jeho zdrojový kód.

5.2.6 SonarQube + SonarQube plugin

Nástroj jako jediný umí zobrazit matici závislostí (viz Obrázek 5.16). Nástroj naopak nezvládá (nebo zvládá s připomínkami) funkcionality uvedené v Tabulce 5.35. Tyto funkcionality řadím do prvních dvou kategorií důležitosti.

kategorie	D	umí / je pravda	název funkcionality	poznámka k funkcionalitě
graf	1	ne	lze zobrazit graf závislostí	graf je podle mě dostatečně nahrazen Maticí závislostí
nástroj	1	ne	umí zobrazit graf závislostí	
graf	2	ne	lze zobrazit graf závislostí + použít filtry	
nástroj	2	ne	umí pracovat se soubory s koncovkou `.jar` (obsahující soubory `.class`)	
nástroj	2	ne	umí pracovat se soubory s koncovkou `.jar` (obsahující soubory s koncovkou `.java`)	
seznam	2	ne	závislostí balíčku na balíčkách + filtr	ručně
seznam	2	ne	závislostí třídy na balíčkách	
seznam	2	ne	závislostí třídy na třídách + filtr	
umí najít	2	ne	závislosti balíčku na balíčkách + filtr	
umí najít	2	ne	závislosti třídy na třídách + filtr	
výstupy	2	ne	lze exportovat seznam závislostí	
výstupy	2	ne	lze exportovat graf	

Tabulka 5.35: důležité funkcionality, které nástroj sonarQube nezvládá

Nástrojem nešlo otestovat dvě ze 4 aplikací, přičemž se všemi ostatními nástroji bylo možné analýzu závislostí se stejnými vstupními daty provést. Nástroj je oproti všem ostatním testovaným nástrojům velmi komplexní.

5.3 Výsledky testů

Na základě výborných výsledků (viz Tabulka 5.36 a hodnocení v kapitole 5.2.5), konceptu nástroje (věnuje se výhradně zkoumání závislostí v Java aplikacích), plánů

s nástrojem do budoucna a velmi pohodlné práce s nástrojem vyplynul jako vítěz testu nástroj *Class Dependency Analyzer (CDA)*.

Z nedostatků zmíněných v Tabulce 5.34 mě osobně schází nejvíce funkcionalita - matice závislostí (tu uměl z testovaných nástrojů jen nástroj *SonarQube*) a nemožnost vygenerování reportu o projektu.

Nástroj	zvládá x % z důležitých (1-3) funkcionalit	z kategorie	přičemž průměr kategorie je
class Dependency Analyzer (CDA)	78%	graf	69%
class Dependency Analyzer (CDA)	80%	GUI	88%
class Dependency Analyzer (CDA)	55%	nástroj	55%
class Dependency Analyzer (CDA)	100%	seznam	57%
class Dependency Analyzer (CDA)	100%	umí najít	69%
class Dependency Analyzer (CDA)	40%	výstupy	57%
classcycle plugin	80%	GUI	88%
classcycle plugin	55%	nástroj	55%
classcycle plugin	33%	seznam	57%
classcycle plugin	53%	umí najít	69%
classcycle plugin	80%	výstupy	57%
Classycle	45%	nástroj	55%
Classycle	33%	seznam	57%
Classycle	53%	umí najít	69%
Classycle	60%	výstupy	57%
CodePro AnalytiX	56%	graf	69%
CodePro AnalytiX	80%	GUI	88%
CodePro AnalytiX	55%	nástroj	55%
CodePro AnalytiX	33%	seznam	57%
CodePro AnalytiX	80%	umí najít	69%
CodePro AnalytiX	80%	výstupy	57%
Degraph	67%	graf	69%
Degraph	45%	nástroj	55%
Degraph	40%	umí najít	69%
Degraph	40%	výstupy	57%
DependencyFinder	78%	graf	69%
DependencyFinder	100%	GUI	88%
DependencyFinder	55%	nástroj	55%
DependencyFinder	100%	seznam	57%
DependencyFinder	93%	umí najít	69%
DependencyFinder	60%	výstupy	57%
sonarQube	100%	GUI	88%
sonarQube	73%	nástroj	55%
sonarQube	44%	seznam	57%
sonarQube	60%	umí najít	69%
sonarQube	40%	výstupy	57%

Tabulka 5.36: porovnání kolik procent důležitých funkcionalit zkoumané nástroje splňují

6 Závěr

Práci považuji za úspěšnou, neboť se podařilo splnit všechny cíle práce.

V první části práce jsem pronikl do problematiky závislostí v Java aplikacích. Tyto poznatky jsem využil při průzkumu nástrojů a postupů zkoumání závislostí protože již bylo z části jasné, jaká kritéria u nástrojů bylo zapotřebí sledovat. Z 31 nástrojů nalezených během tohoto průzkumu webu bylo, na základě jasných a zmíněných kritérií, 24 nástrojů vyřazeno z důkladného testu.

Sadu různorodých Java aplikací pro otestování nástrojů tvořily kvalifikační práce studentů. Díky použití veřejně dostupných prací nemohl nastat žádný problém s porušením autorských práv. Na zvolených aplikacích se zároveň potvrdilo očekávání existence, pro test nástrojů žádoucích, cyklických závislostí.

Na základě průzkumem nabytých znalostí o nástrojích zkoumajících závislosti byly provedeny cílené testy vybraných nástrojů následované hodnocením těchto nástrojů a zvolením nejlepšího neplaceného nástroje zkoumajícího závislosti v Java aplikacích. Po vyhodnocení všech sledovaných kritérií vyšel z testu ze všech neplacených testovaných nástrojů nejlépe nástroj *Class Dependency Analyzer*.

Seznam použitých zkratek

HTML	HyperText Markup Language
JVM	Java Virtual Machine - speciální program nazývaný virtuální stroj Javy
MVC	Model View Controller
OOP	Objektově Orientované Programování
Q&A	Questions and Answers - webové stránky založené na komunitě uživatelů
SOA	Service Oriented Architecture – servisně orientovaná architektura
UML	Unified Modeling Language
XML	eXtensible Markup Language, značkovací jazyk
XSLT	eXtensible Stylesheet Language Transformations, slouží pro převod dat z XML formátu na libovolný jiný formát

Seznam použité literatury

[1] **Pecinovský, Rudolf.** *Příručka programátora myslíme objektově v jazyku Java.* Praha : Grada, 2004. 80-247-0941-4.

[2] **Keogh, J.** *Java bez předchozích znalostí.* Praha : Computer Press, 2005. ISBN 80-251-0839-2.

[3] **Pecinovský, Rudolf.** *Naučte se myslet a programovat objektově.* Brno : Computer Press, a.s., 2010. 978-80-251-2126-9.

[4] **Herout, Pavel.** *Učebnice jazyka Java.* České Budějovice : KOPP, 2007. 978-80-7232-323-4.

[5] **Pitner, Tomáš.** *Java – začínáme programovat.* místo neznámé : Grada Publishing, spol. s r.o., 2002. 80-247-0295-9.

[6] **Zicha, Vojtěch.** Java tutoriál – Objektově orientované programování. *Programujte.com.* [Online] 2. květen 2007. http://programujte.com/clanek/2007043001-java-tutorial-objektove-orientovane-programovani-3-dil/.

[7] **Koval, Filip.** Banan.cz. *www.banan.cz.* [Online] e-BAAN Net s.r.o. https://www.banan.cz/serialy/Java/Java-balicky-6-dil.

[8] cs.vsb.cz. [Online] http://www.cs.vsb.cz/navrat/vyuka/esf/java/ch01s06.html.

[9] **James Edward Keogh, Mario Giannini.** *OOP bez předchozích znalostí.* místo neznámé : Computer Press, a.s., 2006. 80-251-0973-9.

[10] **512.cz.** Generičnost. *FAV wiki.* [Online] 20. 2 2014. [Citace: 23. 6 2015.] http://www.512.cz/index.php?title=Generi%C4%8Dnost.

[11] **Woodcock, J.** *Slovník výpočetní techniky.* Praha : Microsoft Press, 1993. 80-85297-48-5.

[12] **Tichý, Jan.** Architektura aplikace. *jantichy*. [Online] [Citace: 28. 6 2015.] http://www.jantichy.cz/diplomka/pozadavky/architektura.

[13] Úvod do komponent. *kivwiki*. [Online] [Citace: 28. 6 2015.] http://wiki.kiv.zcu.cz/UvodDoKomponent/HomePage.

[14] **Jenkov, Jakob.** Understanding Dependencies. *jenkov.com*. [Online] [Citace: 28. 6 2015.] http://tutorials.jenkov.com/ood/understanding-dependencies.html.

[15] **Tessier, Jean.** Tutorial for Dependency Finder. *Dependency Finder*. [Online] [Citace: 29. 6 2015.] https://docs.google.com/presentation/d/12OSxyz6m22gPbQDVSEhcs_miMhZWeQcYgk55dYaG6fk/present?slide=id.i46.

[16] **K., Neville.** *stackoverflow*. [Online] 10. 3 2013. [Citace: 6. 28 2015.] http://stackoverflow.com/questions/15321702/what-does-package-tangle-index-data-indicate-in-sonar.

[17] **Barowski, L.A. and Cross, J.H., II.** *Extraction and use of class dependency information for Java.* 2002.

[18] **Oracle.** Java Virtual Machine Specification. *Chapter 4. The class File Format.* [Online] [Citace: 25. 6 2015.] https://docs.oracle.com/javase/specs/jvms/se7/html/jvms-4.html.

[19] **Tišnovský, Pavel.** Pohled pod kapotu JVM. *root.cz*. [Online] 13. 12 2011. [Citace: 27. 6 2015.] http://www.root.cz/clanky/pohled-pod-kapotu-jvm-1-cast-prohlizeni-a-modifikace-bajtkodu/#ic=serial-box&icc=text-title.

[20] **Elmer, Franz-Josef.** Classycle: Analysing Tools for Java Class and Package Dependencies. *How Classycle works.* [Online] [Citace: 24. 6 2015.] http://classycle.sourceforge.net/works.html.

[21] **Mička, Pavel.** Tarjanův algorimus. *Algoritmy.net.* [Online] [Citace: 24. 6 2015.] http://www.algoritmy.net/article/1515/Tarjanuv-algoritmus.