VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV INFORMAČNÍCH SYSTÉMŮ
DEPARTMENT OF INFORMATION SYSTEMS

ZDOKONALOVÁNÍ ZDROJOVÉHO KÓDU APLIKACÍ

APPLICATIONS SOURCE CODE IMPROVEMENT

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE **Bc. ALENA OBLUKOVÁ**
AUTHOR

VEDOUCÍ PRÁCE **doc. RNDr. JITKA KRESLÍKOVÁ, CSc.**
SUPERVISOR

BRNO 2017

Abstrakt

Problémem, jímž se práce zabývá, je vylepšení použitelnosti aplikace Classycle, zejména zvýšení srozumitelnosti jejích výstupů. Po nastudování teorie týkající se oblasti refaktorizace, testování, grafů a důkladné analýze původní aplikace Classycle byly vytvořeny zcela nové výstupy aplikace zobrazující výstupní data v grafické podobě. Tato aplikace byla otestována nad reálnými daty a je připravena k nasazení ve firmě. Díky vytvoření nových forem výstupu, popsaných v praktické části diplomové práce, získá programátor silnější nástroj pro detekci závislostí mezi třídami a balíčky v kódu.

Abstract

The problem discussed in this master's thesis is to increase the usability of aplication Classycle, especially to increase the comprehensibility of its outputs. Having studied theories of refactoring, testing, graphs and after thorough analysis of Classycle, it has been created new outputs of the application, displaying the output data in graphics form. The application has been tested with real-life data and it is ready to be deploy in company. Thanks to creation of new forms of outputs, which are discribed in practical part of master's thesis, programmer obtains a powerful tool for detection dependences between classes and packages in code.

Klíčová slova

Classycle, Apache Maven, testování, reengineering, refaktorování, grafové algoritmy

Keywords

Classycle, Apache Maven, testing, reengineering, refactoring, graph algorithms

Citace

OBLUKOVÁ, Alena. *Zdokonalování zdrojového kódu aplikací.* Brno, 2017. Diplomová práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Kreslíková Jitka.

Zdokonalování zdrojového kódu aplikací

Prohlášení

Prohlašuji, že jsem tuto diplomovou práci vypracovala samostatně pod vedením paní doc. RNDr. Jitky Kreslíkové, CSc. Další informace mi poskytli Mgr. Rostislav Mátl, konzultant z firmy. Uvedla jsem všechny literární prameny a publikace, ze kterých jsem čerpala.

.....
Alena Obluková
21. května 2017

Poděkování

Ráda bych velmi poděkovala vedoucí mé práce, paní doc. RNDr. Jitce Kreslíkové, CSc., která se mnou měla velikou trpělivost, měla k mé práci mnoho užitečných komentářů a darovala mi spoustu užitečných rad. Také chci poděkovat konzultantovi Mgr. Rostislavu Mátlovi.

Obsah

Kapitola 1

Úvod

Kvalitní programátor by měl toužit po vytvoření kódu, který splňuje všechny kvalitativní charakteristiky softwarových produktů a jenž se dá nazvat čistým kódem. Důvodů, proč psát čistý kód, je mnoho a jsou rozepsány v kapitole 2. Vytvořit programátorovi čistý kód pomáhá aplikace Classycle, která se zaměřuje na závislosti mezi třídami a balíčky 5.3 v kódu. Díky analýze závislostí a jejich následném zobrazení dokáže programátor nevhodné závislosti odstranit a tím zvýšit srozumitelnost, tedy i celkovou kvalitu jeho produktu.

Předmětem diplomové práce je aplikace Classycle, její analýza, následná modifikace a vznik nových forem výstupu. Nové formy výstupu aplikace Classycle budou použity v reálné firmě při kontrole automatizovaných testů. Díky změně v praktické implementační části projektu se zvyšuje použitelnost aplikace, zejména její srozumitelnost, což povede k vyšší efektivitě při práci s ní. Aplikace Classycle je volně šiřitelná aplikace analyzující kód psaný v jazyce Java a zkoumající závislosti mezi třídami a balíčky. Tyto závislosti následně analyzuje a dále zpracovává. Na základě parametrů při spuštění generuje aplikace textový či grafický výstup těchto závislostí.

Jelikož aplikace Classycle již existovala, v rámci diplomové práce ji bylo nutné nejprve důkladně analyzovat, porozumět neznámému kódu a následně nad některými částmi kódu provést refaktorizaci. Metody porozumění neznámému kódu, definice pojmu refaktorizace a důvody optimalizace kódu jsou popsány v kapitole 2.

Aplikace Classycle se zabývá kontrolou automatizovaných testů. Aby bylo možné rozlišovat druhy testů a pracovat s pojmy chyba či defekt, vznikla kapitola 3 zabývající se úvodem do testování.

Cílem práce bylo vytvořit novou formu výstupu aplikace, a to v grafické podobě. Data jsou zobrazena grafem. V kapitole 4 jsou definovány základní pojmy a popsány algoritmy, které jsou v aplikaci použity. Největší zaměření je zde na definici a využití silně souvislých komponent, jenž aplikace Classycle používá.

Kapitola 5 popisuje nástroje, které byly využity při práci na aplikaci Classycle. Jedná se zejména o nástroj Maven, grafické nástroje yEd a Gephi. Na konci kapitoly je vysvětleno několik základních pojmů z jazyka Java – co je to balíček či bajtkód atp.

V kapitole 6 je podrobně analyzována aplikace Classycle. Tato analýza byla nutností, bez ní nebylo možné aplikaci modifikovat. V kapitole je rozepsáno použití aplikace ve firmě, její struktura a průběh, její podoba před změnou. V sekci Původní vstupy a výstupy zabývající se aplikací Classycle před změnou jsou podrobně popsány i dva módy aplikace, dva způsoby analýzy bajtkódu. Prvním módem je detekce silně souvislých komponent. Druhým módem je kontrola pravidel vytvořených uživatelem sloužících ke kontrole povolených či nepovolených závislostí mezi třídami a balíčky.

Změnou aplikace, se zabývá kapitola 7. Ta je rozdělena na tři části. První část popisuje refaktorizaci aplikace. Druhá část je věnována novým formám výstupu. Nejdříve je definována samotná struktura výstupu a následně je graficky znázorněn příklad výstupu při zobrazení silně souvislých komponent a příklad výstupu při zobrazení porušených pravidel. Poslední částí je využití aplikace a to jak ve firmě, tak i mimo ni.

Závěr obsahuje stručné shrnutí diplomové práce, včetně zhodnocení dosažených výsledků a návrhů na další možná rozšíření.

Diplomová práce navazuje na semestrální projekt. Ten byl zaměřen zejména na analýzu aplikace Classycle, nastudování teoretických podkladů v oblasti refaktorizace a testování a seznámení se s nástroji. Text semestrálního projektu byl po úpravách použit v kapitolách 2, 3, 6 a 5. Všechny kapitoly diplomové práce staví na znalostech získaných během práce na semestrálním projektu a informace z něj dále rozvíjí.

Kapitola 2

Reengineering softwaru, refaktorizace

Kapitola se bude věnovat pojmům jako je již zmíněný reengineering softwaru, refaktorování kódu, způsobům porozumění neznámému kódu, reverzní a dopředný engineering *(reverse engineering, forward engineering)*, výseč kódu *(code slicing)*. Nejdříve je v této kapitole nastíněn teoretický základ – způsoby porozumění kódu a s tím související metody poznávání kódu, s čímž souvisí pojmy jako je například znalostní základ a kognitivní model. Další sekce se věnují mimo jiné pojmu reengineering a refaktorování kódu.

2.1 Způsoby porozumění neznámému kódu

Samotný kód aplikace Classycle je velmi spoře okomentován, chybí základní informace. Kód je to velmi nečistý, proměnné nejsou sebevysvětlující, vyskytují se tu i konstanty s názvem `MAGIC`. Nejtěžší není aplikaci přepsat, upravit. Nejtěžší je zorientovat se v kódu, pochopit, co původní autor chtěl nejspíše říci. Samotným porozuměním kódu jsem se zabývala daleko delší dobu, než jsem původně očekávala.

Vytvoření použitelného kódu, tedy kódu srozumitelného, zvládnutelného, provozovatelného a atraktivního je velmi náročné a zároveň velmi důležité. Je obvyklé, že se jednoho projektu účastní více lidí. Nezřídka se stává, že programátor přijde k výtvoru jiného autora a je nemile překvapen, protože kód je psán zcela jiným stylem, než na který je zvyklý.

V této sekci bude slovo *autor* vyjadřovat původního programátora projektu, slovem *programátor* se pak bude mínit člověk, který se snaží program pochopit a následně s ním pracovat.

Předně je nutné definovat si základní pojmy.

2.1.1 Kognitivní model vs. mentální model

Přídavné jméno kognitivní vyjadřuje získávání obecných poznatků a procesů chápání, zpracování informací. Kognitivní model se od mentálního modelu, který bude popsaný níže, liší v tom, že popisuje celkový kognitivní systém nebo proces použitý při porozumění programu. Veškeré teorie o porozumění neznámému kódu se shodují na tom, že proces porozumění využívá dosud získaných znalostí programátora společně se strategií využívanou k získávání nových znalostí. [10]

Různé strategie porozumění, získávání nových znalostí, mají společný následující postup – v úvodu je nutné formulovat hypotézu, kterou je následně nutné potvrdit, vyvrátit

či přehodnotit a upravit. Formulováním a ověřováním těchto hypotéz díky již získaným znalostem se získávají znalosti nové. Mentální model je interní reprezentace programu, kterou má programátor v hlavě. Tato reprezentace může být rozdrobena na několik menších podčástí, jednotlivých sémantických konstrukcí. Mentální model se samozřejmě neustále mění s tím, jak postupuje proces poznání. [10]

2.1.2 Znalostní základ, získávání znalostí

Znalostním základem se rozumí znalosti, které má osoba již delší dobu, stejně jako znalosti nově získané. Obecně má každý dvě roviny znalostního základu. První je takzvaný všeobecný základ, který získáváme zejména během prvních let studia a pak dále v běžném životě. Druhou skupinou jsou specifické znalosti, které souvisí s naší specializací. Během procesu poznávání potřebuje programátor zejména specifické znalosti související s danou problematikou. Pokud tuto oblast zúžíme čistě na oblast informačních technologií, do všeobecných znalostí je možné zařadit znalost programovacího jazyka, znalost systému.
Do specifických znalostí pak patří porozumění přímo danému úkolu, tedy znalost programovacího stylu, cíl programu, funkce programu. Pokud programátor vidí program poprvé, má pouze všeobecné znalosti a specifické znalosti musí teprve získat. [10]

Znalosti se získávají pomocí asimilačního procesu, což je proces, který programátor používá při snaze porozumět programu. Při asimilačním procesu se na základě zdrojového kódu, dokumentace a znalostnímu základu upravuje programátorův mentální model. Neustálým upravováním mentálního modelu dochází k lepšímu a lepšímu pochopení a porozumění programu. [10]

Mechanismus porozumění se může rozdělit do dvou kategorií a těmi je sjednocování kousků *(chuncking)* a křížové reference *(cross-referencing)*. Sjednocováním kousků je myšleno vytváření větších logických celků, z menších kousků. Na nízké úrovni se jednotlivé kusy programů myšlenkově propojí a vzniknou tak vyšší, abstraktní celky, které se opět dále mohou propojit, až zahrnou celý program. Křížové reference využívají toho, že si programátor dokáže propojit souvislost mezi jednotlivými abstraktními celky. [10]

2.1.3 Porozumění shora dolů

Proces, který se při této metodě používá, začíná vytvořením hypotézy o celkové funkci a cíli programu. Postupně se počáteční hypotéza rozpadne na sub-hypotézy, které se dále rozpadají a programátor proniká hlouběji do programu. [10]

Tento přístup je vhodnější pro situace, kdy se programátor s podobným kusem kódu nebo s podobným stylem kódu již setkal. [10]

2.1.4 Porozumění zdola nahoru

Tato metoda je, jak již název napovídá, opakem výše zmíněné metody. V této metodě začíná proces porozumění tak, že programátor si nejprve přečte zdrojový kód. Kód si rozdělí na jednotlivé sekce. Tyto sekce seskupí do mentálních kousků, ty opět dále spojuje ve větší abstraktní celky. Postupně se vynořuje výš a výš, dokud neporozumí celému programu. [10]

Dle Penningtonové si programátoři na počátku vytvoří abstraktní vývojový diagram. Tento diagram zachycuje jednotlivé sekvence programu. Tento model, diagram, se postupně vyvíjí z kousků mikrostruktur v kódu ve větší logické uskupení a pomocí křížových referencí těchto uskupení. Model je dokončen, jakmile je program programátorem zcela pochopen. [10]

2.1.5 Metody sloužící k porozumění kódu

V této podkapitole stručně popíši některé metody sloužící k porozumění kódu, které byly popsány různými autory. Všechny teorie používají společný základ a to znalostní základ, mentální model a formu asimilace. Způsob tvoření těchto komponent se v různých teoriích liší. [10]

Brookův kognitivní model: program je pochopen, pokud se propojí horní vrstva, problém, se spodní vrstvou, samotným kódem. Brook používá méně tradiční přístup shora dolů, kdy se nejprve programátor snaží pochopit cíl programu a až poté se zaměřuje na konkrétní programovací konstrukce. [10]

Shneidermanův a Mayerův kognitivní model: dle tohoto přístupu se pochopení modelu dělí na dvě roviny, syntaktickou a sémantickou. Syntaktická znalost obsahuje znalost klíčových slov a konstrukcí pro specifický jazyk, tato znalost samozřejmě roste s programátorovou profesní znalostí. Sémantická znalost pak zahrnuje obecné programovací konstrukce jako jsou cykly apod. Proces porozumění musí zahrnovat aplikace syntaktické znalosti k vytvoření sémantického modelu programu. Tento model používá přístup zdola nahoru. Shneiderman používá pro porozumění proces sjednocování kousků kódu ve větší celky, kdy se syntaktické základy uskupují ve větší sémantické celky. [10]

Kognitivní model Letovského: je založen na empirickém studování programátorů, kteří měli za úkol provést nějakou akci s pro ně neznámým kódem. Aby bylo možné provést změny, bylo nutné, aby programátoři neznámému kódu nejprve porozuměli. Při procesu porozumívání měli tzv. přemýšlet nahlas, aby pozorovatelé slyšeli jejich myšlenkový pochod. Kognitivní model rozdělil Letovsky, podobně jako jiní autoři, do třech rovin: znalostní základ, mentální model a asimilační proces. Tyto pojmy jsou vysvětleny výše. Letovsky zároveň naznačil, že je pro zkoumání kódu možné použít dva přístupy a to shora dolů a zdola nahoru. Programátor může mezi těmito přístupy přepínat v závislosti na konkrétní situaci. [10]

Lze vidět, že metody pracují na podobném principu. Používají pojmy jako je znalostní základ, mentální model a asimilační proces. Shodují se na tom, že rychlost porozumění je závislá na velikosti a komplexnosti zkoumaného systému. Závisí také na schopnostech programátorů, experti obecně porozumí neznámému kódu rychleji než nováčci. Porozumění kódu nespočívá pouze v pochopení syntaxe, ale i v porozumění sémantice, celkovému cíli programu. [10]

2.2 Reengineering

Definicí pojmu reengineering je více. V [13] je možné přečíst si následující anglickou definici pojmu reengineering mnou volně přeloženou do češtiny.

„Reengineering je zásadní a hluboké přehodnocení a redesign byznysových procesů, které vede k výraznému vylepšení stávajících faktorů jako je výkonnost, cena, kvalita, služby a rychlost.“[13]

Reengineering je proces, který zahrnuje zkoumání, analýzu a úpravy stávajícího softwarového systému vedoucí k vytvoření nové formy a s tím související implementace této nové formy. Tento proces typicky zahrnuje reverzní inženýrství, opětovné sepsání dokumentace, restrukturalizaci, překlad a dopředné inženýrství. Cílem je porozumět existujícímu softwaru (specifikaci, designu, implementaci) a poté tento software znovu implementovat za účelem

zlepšení systémové funkcionality, výkonu. Záměrem je zachovat stávající funkcionalitu a připravit půdu pro přidání další funkcionality. [10]

Obecně vzato se tedy jedná o zásadní změnu již vytvořeného. Slovo zásadní je zde důležité, protože reengineering nemá za cíl drobně vylepšit chyby, vytvořit záplatu, provést malé změny. Cílem je zapomenout na původní strukturu a vytvořit něco zcela nového se stejnou funkcionalitou.

Dovolím si uvést příklad. Je zcela běžné, že malá firma začne vyrábět software, produkt. Firma se postupně rozrůstá a s ní se zvyšuje i funkcionalita nabízeného produktu. Se zvyšující funkcionalitou ale přibývají zaměstnanci, řádky kódu, s nimiž na počátku nebylo počítáno. Software je pak velmi obtížné spravovat, nemá zavedenou jasnou strukturu, nové prvky jsou přidávány nesystematicky. Je proto dobré v jedné chvíli říci stop, odhodit starý systém a se znalostmi, co již firma během let nastřádala, vyrobit software nový, který bude rychlejší, levnější na údržbu, přehlednější, připravený na další rozšiřování. Složitost provést reengineering tkví ve více věcech. Jednou z nejzásadnějších je odvaha provést tak zásadní změnu. Dalším úkolem, se kterým je nutné se vypořádat, je již zmiňované porozumění současnému systému. Často se stává, že původní požadavky a dokumentace kódu již neexistují nebo jsou příliš zastaralé. Není tedy jasné, jaká je přesná funkcionalita současného systémů.

Pojem reengineering se často používá v souvislosti se změnou podnikových procesů a ne nutně se změnou zdrojového kódu. Reengineering zdrojového kódu se často v literatuře nazývá refaktorování kódu, o kterém je následující sekce.

2.2.1 Kvalitativní charakteristika softwarových produktů

V diplomové práci jsou často použita slova srozumitelnost, použitelnost a podobně. Tato slova jsou jasně definována v rámci kvalitativní charakteristiky softwarových produktů. V rámci této charakteristiky je definováno šest základních ukazatelů, které se dále dělí. Funkčnost *(functionality)*, bezporuchovost *(reliability)*, použitelnost *(usability)*, účinnost *(efficiency)*, udržovatelnost *(maintainability)* a přenositelnost *(portability)*. [9]

Funkčnost je způsobilost softwarového produktu poskytovat funkce, které uspokojují stanovené a předpokládané potřeby, pokud je software používán za specifikovaných podmínek. **Bezporuchovost** je způsobilost softwarového produktu udržovat specifikovanou úroveň výkonu, pokud je používán za specifikovaných podmínek. **Použitelnost** je způsobilost softwarového produktu být srozumitelný, zvládnutelný, používaný a atraktivní pro uživatele, pokud je používán za specifikovaných podmínek. Ta se dále dělí na **srozumitelnost** (způsobilost softwarového produktu umožnit uživateli porozumět, zda je software vhodný a jak může být použit pro konkrétní úlohy a podmínky používání), **zvládnutelnost** (způsobilost softwarového produktu umožnit uživateli naučit se jej používat), provozovatelnost, atraktivnost a soulad použitelnosti. **Účinnost** je způsobilost softwarového produktu poskytovat vhodný výkon s ohledem na množství použitých zdrojů, a za stanovených podmínek. **Udržovatelnost** je způsobilost softwarového produktu být modifikován. Modifikace mohou zahrnovat nápravy, zlepšování nebo adaptace softwaru na změny v prostředí, v požadavcích a ve specifikacích funkcí. Jedním z podbodů udržovatelnosti je i **zaměnitelnost** (způsobilost softwarového produktu umožnit, aby byla specifikovaná modifikace implementována). **Přenositelnost** je způsobilost softwarového produktu být přenesen z jednoho prostředí do jiného prostředí (organizační, hardwarové, softwarové). [9]

2.3 Optimalizace kódu

Pro optimalizaci kódu se užívá spoustu odborných názvů převzatých z anglického jazyka. Nejčastějším užívaným pojmem je *refaktorizace*, případně *refaktorování*. Tyto výrazy lze nalézt také v české odborné literatuře. Nadále v této práci budu užívat pojmy refaktorizace a refaktorování jako vyjádření procesu optimalizace kódu.

„Refaktorování je proces provádění změn v softwarovém systému takovým způsobem, že nemají vliv na vnější chování kódu, ale vylepšují jeho vnitřní strukturu. Je to disciplinovaný způsob pročišťování kódu s minimálním rizikem vnášení chyb.“ [12]

Refaktorizace je častým prostředkem programátora. Typickým příkladem refaktorizace je přejmenování proměnné, rozdělení dlouhé funkce či metody na více kratších a podobně.

Důležitým bodem u refaktorizace jsou kvalitní testy. O testech pojednává kapitola Testování 3.

2.3.1 Čistý kód

V často citovaném zdroji (*Clean Code*) je popsána snaha o definici čistého kódu. Dle autora existuje těchto definic tolik, kolik existuje programátorů. Žádnou definici čistého kódu zde tedy neuvádím. Uvádím však pár citátů známých osobností „čistý kód“.

Bjarne Stroustrup, tvůrce jazyka C++ a autor „The Programming Language“

„Chci, aby můj kód byl elegantní a účinný. Logika by měla být přímočará, aby se chyby neměly kde skrývat, s minimálními závislostmi, aby údržba byla jednoduchá, kompletní ošetření chyb v souladu s jasně formulovanou strategií a s výkonem blízkým optimu, aby lidé nebyli v pokušení zavádět do kódu nepořádek pomocí svévolných optimalizací. Čistý kód plní svou funkci dobře.“ [15]

Grady Booch, autor knihy „Object Orineted Analysis and Design with Applications“

„Čistý kód je jednoduchý a přímočarý. Čistý kód se čte jako dobře napsaná próza. Čistý kód nikdy nezatemňuje záměr návrháře, ale je plný břitkých abstrakcí a přímých toků řízení.“ [15]

Ward Cunnigham, tvůrce „Wiki“, tvůrce nástroje „Fit“, spolutvůrce „eXtrémního Programování“. Motivační síla v pozadí „Návrhových vzorů“. Čelní myšlenkový představitel objektově orientovaného programování a jazyka Smalltalk. Kmotr všech, kteří se zajímají o kód.

„Že pracujete s čistým kódem poznáte podle toho, že každá procedura, kterou procházíte, se ukáže být tím, co jste do značné míry předpokládali. Kód můžete označit za nádherný, když vypadá, jako kdyby byl použitý jazyk pro daný problém stvořen.“ [15]

Definic čistého kódu je více a každý si může vybrat tu svou. Všechny však spojuje požadavek na vysokou míru srozumitelnosti a zaměnitelnosti. Také je kladen důraz na minimalizaci závislostí, které se aplikace Classycle snaží dosáhnout. Podle autora knihy „strýčka Boba“ je nutné uvědomit si, proč je tak důležité psát čistý kód. Poměr času, stráveného čtením k času strávenému psaním je více než 10:1. Proto je snaha učinit čtení kódu co nejjednodušší, přestože je psaní takového kódu těžší.

2.3.2 Pachy kódu

V [12] jsou vyjmenovány tzv. pachy kódu, tedy případy, které jsou typickými kandidáty na refaktorování, resp. takové části, které nějak “smrdí” a je nutné jejich další prozkoumání. Není jednoduché odpovědět na otázky, kdy už refaktorovat a kdy je to ještě zbytečné. Pachy jsou myšleny typy struktur, které nabádají k refaktorování. Uvádím zde jen ty nejčastější.

Nejčastější a nejoblíbenější kandidát na refaktorování je duplicitní kód. Je-li v programu na více místech podobná struktura, pak je určitě záhodno tyto jednotlivé kousky nahradit jednou metodou a v místech jejich původního výskytu je smazat a nahradit voláním na nově vytvořenou metodu. [12]

Dalším snadno objevitelným pachem je dlouhá funkce/metoda. To stejné platí i pro velkou třídu. Tak je to psáno i v [15], ve které se píše, že prvním pravidlem pro psaní funkcí je, že by měly být malé. Druhým pravidlem je, že by měly být ještě menší. Přesná definice dlouhé funkce není. V osmdesátých letech se tvrdilo, že funkce nemá být delší, než je zaplněná obrazovka. Přepočítáno na současné poměry, řádky by neměly být delší než 150 znaků a funkce by neměly být delší než 100 řádků. Ideální funkce má mít pouze tři nebo čtyři řádky. Podobné pravidlo platí pro třídu. Ve stejném zdroji stojí, že první pravidlo pro psaní třídy je, že by měly být malé. Druhým pravidlem je, že by měly být ještě menší. Ano, jak funkce, tak třída nemá být v čistém kódu příliš velká. Definice velikosti třídy vychází z odpovědností – čím více odpovědností, tím větší třída. Třída by měla být tak velká, aby měla pouze jedinou odpovědnost. Principem jedné odpovědnosti (*Single Responsibility Principle*) se zabývá spousta dalších publikací. [12], [15]

Funkce či metoda by měla dělat jen jednu věc. Měla by ji dělat dobře a neměla by dělat nic jiného. Příklad z aplikace Classycle: funkce `readClassFiles` podle názvu pouze čte soubory. Ve skutečnosti však data ze souborů analyzuje a vytváří finální graf. Tato situace není intuitivní. Ideálním řešením je funkci rozdělit na menší, méně dobrým řešením je funkci alespoň přejmenovat, aby bylo již z názvu patrné, co dělá. [15]

Nepřehlédnutelným kandidátem na refaktorování je i příliš dlouhý seznam parametrů. Dříve se při procedurálním programování předávalo hodně parametrů z důvodu omezení globálních proměnných. Objektově orientovaný přístup situaci mění, objekt může o bližší informaci jiný objekt jednoduše požádat. [12]

Příkaz switch se v objektově orientovaném přístupu příliš nepoužívá, lepší je použití polymorfismu. Někdy se samozřejmě příkazu switch nevyhneme, ale při rychlém pročtení cizího kódu může být tento příkaz často kandidátem k refaktorování. [12]

Posledním pachem, který zde uvádím, je líná třída. Líná třída je třída, která téměř nic nedělá, má nečinné podtřídy a v kódu se vyskytuje zcela zbytečně. Takovou třídu je dobré odstranit, případné zbytky její funkčnosti zahrnout do jiných tříd. [12]

Kapitola 3

Základy testování

Testování je jednou z nejvýznamnějších položek v rámci vývoje softwaru. Průběh samotného psaní kódu je obvykle následující. Samotné tělo kódu napsáno velmi rychle, ale vymýšlení toho, jak kód napsat zabere chvíli čas. Nejdéle pak zabere hledání defektů. Každý programátor to zná. Defekt je sice opravený rychle, nicméně doba jeho nalezení je podstatně delší. Psaní testů dříve než samotného programu pomůže programátorovi ujasnit si myšlenky. Při následném psaní může testy automaticky opakovaně spouštět. Defekt pak není tak těžké najít, protože délka kódu, který vznikl od doby posledního spuštění testu, bude krátká. Defekt se bude hledat v menší oblasti a také nad kódem, který bude mít programátor ještě v hlavě.

Manuální testování není levné na lidské zdroje a je možné ho považovat za neprogramátorskou činnost. I manuální testování má však své místo v rámci zajišťování kvality softwaru. Automatizované testy jsou často používaným nástrojem, díky své úspoře lidských zdrojů při jejich provádění. Navíc kvalitní testy můžou částečně nahradit dokumentaci. V agilním vývoji se tohoto často využívá.

Důvodem vzniku této kapitoly je to, že aplikace Classycle má jako jeden z úkolů otestovat program. Způsob psaní testů se tedy dá aplikovat na úpravu této aplikace. Dalším důvodem je to, že aplikace Classycle testuje automatizované testy.

3.1 Základní pojmy

V úvodu kapitoly je nutné definovat pár teoretických pojmů, jako jsou základní dělení testů, definice defektu a jeho analýza, rozdíly statickou a dynamickou technikou a rozdíl mezi chybou a defektem.

3.1.1 Chyba vs. defekt

V [18] je krátká kapitola o rozdílu mezi defektem a chybou. Sami autoři v knize uvádí, že se tyto dva pojmy velmi pletou, často je i odborná literatura považuje za synonyma. Jedná se však pouze o dva podobné pojmy, rozdíl mezi defektem, chybou a selháním existuje.

Defekt je část kódu, která může vést k chybě. Pokud tato chyba ovlivní další systém přes definované rozhraní, dojde k selhání.

Příklad: programátor napíše kód, který obsahuje defekt. Tento kód provádí většinou času výpočet dobře, při jisté kombinaci validních hodnot však vrátí chybný výsledek, dochází k chybě. Pokud se tato hodnota ze systému A používá v systému B, pak v systému A dochází k selhání a systém B obsahuje defekt, který opět může zapříčinit chybu. [18]

To, že kód obsahuje defekt neznamená, že se musí chyba projevit. Defekt totiž může být v části kódu, která se nikdy nepoužije (ať už se jedná o mrtvý kód nebo souhru náhod vstupních dat). [18]

Platí tedy následující: defekt je příčinou chyby, chyba je stav systému a je příčinou selhání a selhání je nevalidní chování systému oproti specifikaci. [18]

Dále v textu budu rozlišovat termíny chyba a defekt.

3.1.2 Statické a dynamické techniky

Rozdíl mezi statickou a dynamickou technikou v rámci kvality softwaru je popsán níže.

Za statickou technikou považuje literatura statickou analýzu. Statická analýza zkoumá software, aniž by ho jakkoliv spouštěla. Z toho vyplývá, že pojem testování, tak jak je běžné chápán, nepatří do statické analýzy. Statická analýza může být prováděna hned v počátku vývoje. V [18] je následující dělení:

- Nástroje automatické statické analýzy se používají dnes a denně každým programátorem. Může se jednat o speciální nástroj, často jsou však v nějaké podobě součástí IDE, integrovaného vývojového prostředí *(Integrated development environment)*. Automatická statická analýza téměř vždy zkoumá syntaxi zdrojového kódu. Dle druhu IDE pak kontroluje dodržování firemních programovacích konvencí a standardů, kontrolu toku řízení apod.

- Neformální revize patří mezi běžně používané techniky, kdy jeden programátor vysvětluje dalšímu či dalším, co je obsahem jeho kódu. Jakmile se najde defekt, okamžitě se opraví.

- Strukturované procházení je podobné neformální revizi, je však formálnější. Cílem není pouze nalézt defekty, ale také podělit se o kód s ostatními členy týmu, aby měli všichni přehled o aktuálním dění.

- Technická revize je již formální revize. Jejím výsledkem má být zpráva, ve které je doporučení, zda produkt splňuje definované požadavky a nebo zda je nutné ho pozměnit, přepracovat.

- Inspekce je nejpokročilejší metodou, která se v praxi používá již minimálně. Kód se prezentuje před ostatními členy, kteří mají pevně stanovené role (moderátor, průvodce, autor, oponenti, zapisovatel). Už z tohoto popisu je zřejmé, že se jedná o velmi formální aktivitu, která je sice velmi efektivní, ale také náročná na čas a zkušenosti zúčastněných členů.

Pokud člověk pracuje sám či v týmu na malém projektu (příkladem jsou školní projekty), používá v naprosté většině pouze první dva jmenované typy. Někteří z nás si u týmových projektů zavedli programovací konvence, aby každý kus kódu nevypadal zcela jinak. Neformální revize je pak běžnou součástí práce i na malých projektech. Specifickou neformální revizí může být tzv. *"Duck programming"*. Programátor při tomto způsobu programování říká tok svých myšlenek nahlas. Mluvení nahlas je zmíněno i v teorii poznávání neznámého kódu – u Kongitivního modelu Letovského.

3.2 Testování

V této kapitole je vysvětleno co je testování a co není, jaké jsou základní axiomy a principy testování, jaké typy testování rozlišujeme.

3.2.1 Axiomy a principy testování

Různé zdroje se liší v tom, co považují za axiomy, neměnná pravidla testování. Některé body jsou však společné. Z odborné literatury [18], [14], [16] jsem vybrala následující body:

- Není možné testovat program úplně celý, zahrnout všechny případy.
- Programátor by si neměl psát své vlastní testy.
- Test musí otestovat jak validní, tak nevalidní hodnoty.
- Testování nedokáže zajistit naprostou absenci defektů.
- Čím více defektů test odhalí, tím více jich tam je.
- Ne všechny nalezené defekty se odstraní.

Není možné testovat program úplně celý, zahrnout všechny případy. Pokud je program netriviální, není možné zpravidla říci, že je program zcela otestován a není v něm žádný defekt. Každý zná Murphyho zákon o programování: „Každý program obsahuje jeden chybný řádek“. Z toho plyne, že každý program jde zkrátit na jeden řádek, který je chybný. Ve většině případů program obsahuje tolik kombinací vstupních hodnot, že je prakticky nemožné v reálném čase všechny kombinace otestovat. [18]

Jiným případem je pak formální verifikace. Formální verifikace je však velmi specifická oblast, kterou se v diplomové práci nebudu zabývat.

Programátor by si neměl psát své vlastní testy. V úvodu je nutné říci, že touto větou se nemyslí, aby si programátor vůbec nezkontroloval svůj kód. Základní testy jistě provádět musí. Nicméně neměl by provádět testování, o jakém se mluví nyní. Otestovat produkt by měl vždy jiný člověk než ten, který jej programoval. Důvod je zřejmý. Programátor ví (měl by vědět), co napsal. Ví, kde má svá slabá místa. Člověk podvědomě chce odvádět dobrou práci a defekty si hledat cíleně nechce. Je pro něj obtížné se snažit kód takzvaně “rozbít” poté, co ho tak dlouhou chvíli vytvářel. Je obvyklé, že se programátor snaží odvést svou práci co nejlépe a již při vytváření programu se snaží zachytit každou situaci. Pokud ho nenapadlo ošetřit situaci v programu, asi ho nenapadne daná situace ani při testování. Je taktéž možné, že programátor vytvořil výborný program, který však není správný, protože programátor chybně pochopil specifikaci. Pokud by programátor svůj výtvor také testoval, je pravděpodobné, že by specifikaci pochopil opět chybně. [16]

Je mylné domnívat se, že tester odhalí vždy více chyb, než autor programu nebo že specifikaci pochopí oproti programátorovi vždy správně. Nicméně je dobré, když kód zkontroluje více lidí a tester obvykle bývá v hledání defektů v programu úspěšnější než autor sám.

Test musí otestovat jak validní, tak nevalidní hodnoty. Je zřejmé, že testování pouze validních či pouze nevalidních hodnot není správná cesta. Autoři testů musí myslet vždy na obě varianty. V obou případech nepostačí pouze jedna kombinace hodnot, ale je snaha otestovat většinu krajních hodnot. Příkladem může být operace celočíselného dělení.

Mezi validní hodnoty jistě patří dělení kladným celým číslem, záporným celým číslem. Mezi nevalidní pak patří dělení 0, dělení desetinným číslem, dělení nečíselnou hodnotou.

Testování nedokáže zajistit naprostou absenci defektů. Jak již bylo zmíněno výše, testy u netriviálních programů zpravidla nedokážou pokrýt všechny kombinace vstupů.

Čím více defektů test odhalí, tím více jich tam je. Tento bod platí zejména kvůli tomu, že programátor často své defekty opakuje. Pokud tedy udělal defekt na jednom místě, je možné, že ho zopakoval i na místě jiném. Pokud je v jedné části více defektů, je možné, že programátor měl zkrátka "špatný den" a defektů se nachází v daném místě více. [18]

Ne všechny nalezené defekty se odstraní. Často se při vývoji velkého produktu stává, že tester odhalí defekt. Náklady na jeho odstranění jsou však větší, než případné ztráty při projevení chyby. Oprava defektu vyžaduje lidské zdroje, které však můžou být alokovány někde jinde.

3.2.2 Druhy testů

V rámci testování je možné setkat se s více druhy testů. Existují čtyří základní úrovně testování korespondující s jednotlivými fázemi vývoje. [18]

- Testování jednotek.
- Integrační testování.
- Systémové testování.
- Akceptační testování.

Další používané testy jsou:

- Regresní testy.
- Konfirmační testy
- Smoke testování.

Testování jednotek (*unit testing*) vytváří a následně i provádí testy sám programátor. Testuje jednotlivé jednotky, bloky kódu. V závislosti na jazyce se testují jednotlivé funkce nebo metody či třídy. Cílem je dokázat, že jednotka je funkční nezávisle na ostatních, při testu nesmí používat jiné jednotky. [18]

Integrační testování (*integration testing*) má za úkol otestovat skupinu jednotek, které spolu kooperují. Je zřejmé, že funkčnost jednotlivých jednotek nezaručuje funkčnost systému vzniklého jejich spojením. [18]

Systémové testování (*system testing*) zahrnuje více podskupin. Funkční testy mají zaručit, že systém splňuje specifikované požadavky. Sem lze zařadit i bezpečnostní testy. Dalšími testy jsou testy robustnosti testující schopost systému zvládat chybové či neočekávané vstupy. Testy použitelnosti zkoumají, zda systém splňuje požadavky uživatele na použitelnost, tedy zda je dostatečně komfortní, uživatelsky přívětivý. Dále existují testy spolehlivosti, výkonnostní testy. [18]

Akceptační testování (*acceptance testing*) ověřuje, zda systém splňuje tzv. akceptační kritéria. Tato kritéria jsou měřitelná, ověřitelná a jsou předem dohodnutá zákazníkem a dodavatelem. [18].

Regresní a konfirmační testy. V praxi se často uvede na trh produkt, který se časem vylepšuje. S každým vylepšením je možné zavést do sytému defekty. Regresní testy ověřují,

že se změnou jedné části neovlivnila část jiná. Regresní testy by měly být prováděny na všech úrovních, tedy jednotkové, integrační a systémové. To je velice náročné jak časově, tak finančně. Ke zmírnění nákladů se vyvíjí tzv. automatizované testy, které jsou automaticky spouštěny při situaci definované vnitřní politikou firmy. Konfirmační testy slouží k ověření a potvrzení správnosti opravy defektu, odhaleného regresním testováním. Regresní testy tvoří regresní sady. Ty nejsou z kapacitních důvodů schopny otestovat vše. Proto se nejčastěji do regresní sady volí testy ověřující základní a klíčovou funkcionalitu systému, velmi často používané součásti sytému, testy zabývající se oblastí změny. [18]

Smoke testování[1] slouží k ověření, zda systém splňuje základní funkcionalitu, zda je dostatečně stabilní na to, aby bylo možné pokračovat v testování. Pokud by smoke testy selhaly, je jisté, že by selhala i naprostá většina testů z regresní sady, které je z toho důvodu zbytečné pouštět. Spouštění všech testů z regresní sady je totiž zpravidla časově náročné. Smoke testy jsou obvykle prováděny automaticky před každým regresním testováním, v některé literatuře jsou smoke testy zařazeny do regresního testování jako jedna z jeho částí. [18]

[1] „Toto pojmenování v angličtině proniklo do světa testování na základě přeneseného významu. Používání originálního pojmu je i u nás natolik zažité, že nemá smysl pokoušet se o překlad.“ Citace z [18]

Kapitola 4

Grafy, grafové algoritmy

Výstupem aplikace Classycle je orientovaný graf. V závislosti na parametrech jsou v grafu zobrazeny silné komponenty nebo porušená pravidla.

4.1 Definice

Teorie grafů je poměrně mladá disciplína matematiky. Za zakladatele teorie grafů je považován Leonhard Euler. Ten v roce 1736 vyřešil dnes již klasickou úlohu sedmi mostů města Královce (dnešní Kaliningrad).

Níže uvedené definice jsou nutné pro definici silně souvislé komponenty.

4.1.1 Graf neformálně

Graf intuitivně chápeme jako množinu vrcholů (uzlů), které můžou a nemusí být propojeny hranami. Hrana spojuje dva vrcholy. Pokud spojuje jeden a tentýž vrcholy, hovoříme o smyčce.

Pokud jsou hrany orientované, mluvíme o orientovaném grafu. Rozlišujeme počáteční (výchozí) vrchol a koncový vrchol. V orientovaném grafu jsou všechny hrany orientované.

Opakem jsou neorientované grafy, kde hrany nejsou orientované, chápeme je jako symetrické spojení dvou vrcholů. V neorientovaném grafu jsou všechny hrany neorientované. [11]

4.1.2 Orientovaný graf

Definice 4.1.1. „Orientovaný graf je dvojice $G = (U, H)$, tvořená neprázdnou konečnou množinou U, jejíž prvky nazýváme *vrcholy* nebo *uzly* a konečnou množinou H, jejíž prvky nazýváme *orientovanými hranami*[1], $H = \{(u, v)|u, v \in U\}$.“ [20]

„Orientovaný graf je též možno chápat jako neprázdnou konečnou množinu s binární relací.“ [20]

[1] Všechny grafy v této diplomové práci jsou orientované. Všechny hrany jsou orientované, proto dále v textu bude používán pouze termín „hrany“ namísto „orientované hrany“.

4.1.3 Vstupní a výstupní stupeň uzlu

Definice 4.1.2. „Nechť $G = (U, H)$ je orientovaný graf. Pro uzel $u \in U$ grafu G definujeme čísla $deg_+(u) = |M|$, $deg_-(u) = |N|$, kde
$M = h \in H | \exists v \in U : h = (v, u)$ a $N = h \in H | \exists v \in U : h = (u, v)$.
Číslo $deg_+(u)$ se rovná počtu hran, které vedou z nějakého uzlu do uzlu u, a nazývá se *vstupním stupněm uzlu u*.
Číslo $deg_-(u)$ se rovná počtu hran, které vedou z uzlu u do nějakého uzlu, a nazývá se *výstupním stupněm uzlu u*.
Pokud platí $deg_-(u) = 0$, u se nazývá *koncový uzel*, a pokud $deg_+(u) = 0$, u se nazývá *počáteční uzel* grafu G.“ [20]

4.1.4 Sled

Definice 4.1.3. „Posloupnost vrcholů a hran $u_0, h_1, u_1, h_2, ..., h_k, u_k$ nazýváme *orientovaným sledem*, jestliže pro každou hranu h_i z této posloupnosti platí $h_i = \{u_{i-1}, u_i\}, 1 \leq i \leq n$.“ [11], [20]

Definice 4.1.4. „*Uzavřený sled* je sled, který má alespoň jednu hranu a jehož počáteční a koncový vrchol splývají.“ [11], [20]

4.1.5 Cesta

Definice 4.1.5. „Orientovaný sled, v němž se neopakuje žádný vrchol, nazýváme *orientovanou cestou*.“ [11]

Definice 4.1.6. „*Uzavřená cesta* je uzavřený sled, v němž se neopakují vrcholy (kromě $u_0 = u_k$) a navíc se neopakují ani hrany.“ [11]

4.1.6 Cyklus

Definice 4.1.7. „*Cyklus* je orientovaná uzavřená cesta.“ [20]

4.1.7 Podgraf

Definice 4.1.8. „Graf H je podgrafem grafu G, vznikne-li z grafu G vynecháním nějakých (nebo žádných) vrcholů a hran. Podstatné je, že podgraf musí být také grafem: spolu s každou hranou, která je v podgrafu, tam musí být i oba její krajní vrcholy.“ [11]

4.2 Silně souvislá komponenta

4.2.1 Silná souvislost

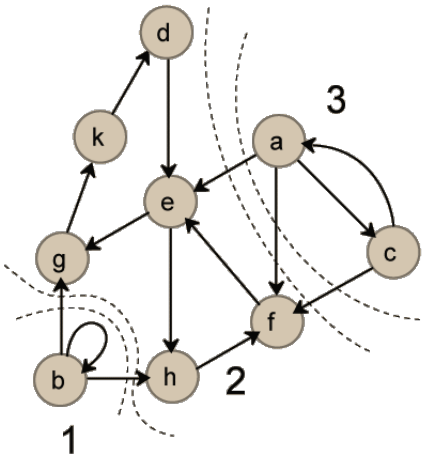
Definice 4.2.1. „Orientovaný graf G nazýváme *silně souvislým*, jestliže pro každou dvojici jeho vrcholů u, v existuje orientovaná cesta z u do v a také zpět z v do u. *Silně souvislou komponentou* grafu G (též *silnou komponentou* nebo *komponentou silné souvislosti*) nazýváme podgraf H grafu G, který je silně souvislý a který je maximální s touto vlastností, tj. není část většího silně souvislého podgrafu.“ [11]

Vlastnosti silných komponent. „Každý vrchol grafu leží přesně v jedné komponentě silné souvislosti.“ [11]

Věta. „Hrana h je obsažena v nějakém cyklu právě tehdy, když oba její vrcholy (počáteční i koncový) leží v téže silně souvislé komponentě.“ [11]

Důsledek. „Každá silně souvislá komponenta, která obsahuje alespoň dva různé vrcholy, obsahuje cyklus.“ [11]

V grafu na obrázku 4.1 lze vidět tři silně souvislé komponenty.



Obrázek 4.1: Silně souvislé komponenty. Zdroj [2].

Cyklus v silné komponentě. Pro nalezení cyklů v grafu se často využívá silných komponent, respektive algoritmu pro nalezení silných komponent. Silně souvislá komponenta podle definice cyklu výše nemusí obsahovat pouze jeden cyklus, ale může jich obsahovat více. Na obrázku 4.1 lze vidět, že v silně souvislé komponentě č. 2 jsou cykly dva – e-h-f-e a e-g-k-d-e, kde symbol „-“ vyjadřuje hranu mezi vrcholy. Dle definice je cyklus uzavřená orientovaná cesta a uzavřená cesta je sled, v níž se neopakují vrcholy (kromě počátečního a koncového). Správně definováno, silně souvislá komponenta na obrázku 4.1 označena číslem 2 neobsahuje jeden cyklus, ale dva. Pro jednodušší popis dále v práci předpokládám, že každá silně souvislá komponenta obsahuje vždy jeden cyklus, který vznikne složením menších cyklů. Pro použití dále v textu upravuji definici cyklu tak, že je možné projít jedním vrcholem vícekrát, ač se nejedná o uzel počáteční. Silně souvislou komponentu č. 2 v tomto vyjádření tvoří jeden cyklus a to cyklus e-h-f-e-g-k-d-e.

4.2.2 Hledání silně souvislých komponent

Princip spočívá v tom, že se zvolí vrchol, který neleží v žádné silné komponentě a najdeme silnou komponentu, která tento vrchol obsahuje. Výpočet se opakuje do té doby, dokud nejsou všechny vrcholy zařazeny do komponent. [11]

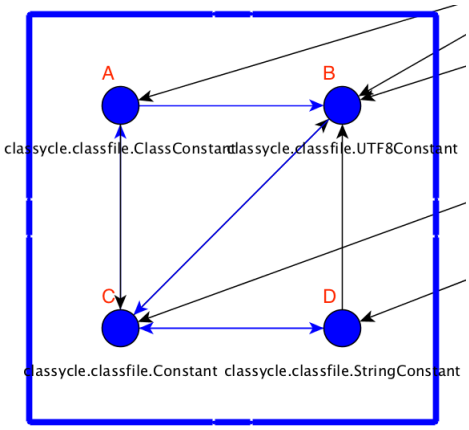
Tarjanův algoritmus je algoritmus pro hledání silně souvislých komponent, který je v této souvislosti vždy zmiňován jako první a často i jediný algoritmus. Dalším algoritmem je například Kosarajuův, který však není uváděn v žádné z knih o grafech, které mám k dispozici. Tento fakt je zřejmě důvodem, proč si autor aplikace Classycle vybral pro hledání silně souvislých komponent právě Tarjanův algoritmus. Aplikace již před mým zásahem dokázala silně souvislé komponenty najít, tento algoritmus jsem neimplementovala. Proto

zde není blíže rozebírán. Je založen na prohledávání grafu do hloubky (dále v textu DFS – *deep search first)*, v němž se však navíc provádí pomocné operace vždy při příchodu do nového vrcholu, při zpracování hrany a při návratu z vrcholu. Protože je založen na DFS, má asymptotickou složitost $O(m + n)$, kde m je počet hran a n je počet vrcholů. [11]

4.2.3 Hledání cyklů v silně souvislé komponentě

Classycle umí najít silně souvislé komponenty. Hledá je pomocí Tarjanova algoritmu. Ten však nezobrazuje hrany, díky kterým je jsou uzly seskupeny do jedné silně souvislé komponenty. Z pohledu programátora je však nutné vědět, které vazby způsobují, že jsou uzly v jedné silné komponentě. Proto vznikl algoritmus, který tyto hrany hledá a následně je zvýrazňuje.

Na obrázku 4.2 lze vidět, že modře jsou zvýrazněny hrany, díky kterým se z libovolného uzlu grafu dá dostat do libovolného jiného. Pro lepší čitelnost jsou uzly označeny v grafu písmeny A, B, C a D. Například z uzlu A do uzlu B se lze dostat pomocí jedné hrany. Z uzlu A do uzlu D vede cesta přes uzly B a C.



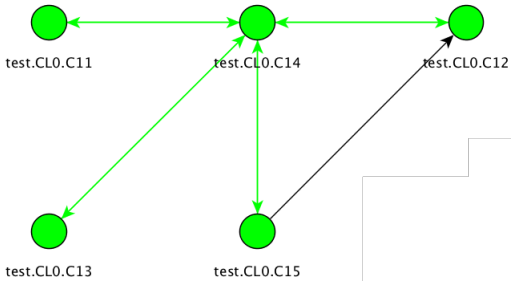
Obrázek 4.2: Jedna silně souvislá komponenta, výřez z analýzy aplikace Classsycle.

Z obrázku lze vyčíst, že z uzlu A do uzlu D vede cesta také přes vrchol C, vynecháním vrcholu B. Hrana z A do C však zvýrazněna není. Důvod je následující. Můj cíl je vytvořit co nejpřehlednější výstup pro programátora. Snažila jsem se zobrazit pouze jedno z možných řešení. Odstraněním všech těchto vazeb zanikne silná komponenta, více vazeb není nutné odstranit. Pokud by bylo zobrazeno variant více, programátor by nevěděl, které hrany jsou nutné a postačující k odstranění tak, aby silná komponenta zanikla. Pokud programátor odstraní pouze jednu zvýrazněnou hranu, která nepovede k odstranění silné komponenty, a spustí analýzu pomocí aplikace Classycle znovu, algoritmus zvýrazní jiné hrany způsobující silnou komponentu.

Algoritmus k nalezení této cesty je založen taktéž na algoritmu DFS. Aplikace má interně uloženy jednotlivé silně souvislé komponenty. Pro každou silně souvislou komponentu, která obsahovala více než jeden uzel, jsem sestrojila list hran, které ji způsobují. Tento list představuje výsledný cyklus.

V rámci algoritmu jsem řešila následující problém. Algoritmus DFS má tzv. zarážku, díky které se nenavštěvuje jeden a tentýž uzel neustále dokola. Jako zarážka funguje proměnná `navštíveno`, která se nastaví na hodnotu `true`, jakmile je uzel navštíven. Uzly, které již jednou navštívené byly se znovu nenavštěvují. Tento postup jsem nemohla zvolit.

Je totiž validní, aby jeden uzel byl navštíven vícekrát a to maximálně tolikrát, kolik z něj vede hran. Na obrázku 4.3 lze vidět, že uzel uprostřed musí být navštíven ne pouze jednou, ale hned čtyřikrát.



Obrázek 4.3: Jedna silně souvislá komponenta, výřez z analýzy aplikace Classsycle.

Díky způsobu definice zarážky pomocí maximálního počtu z uzlu vedoucích hran mohou vznikat při hledání cesty smyčky. Například na obrázku 4.2 chci hledat cestu z uzlu C do uzlu D. Pomocí DFS se nalezne sled[2] z vrcholu C-A-C-D, kde „ - “ představuje hranu. Propojení C-A-C není žádané, je třeba nalézt pouze cestu C-D. Proto je z výsledku, který vydal DFS nutné počátek odstranit a zachovat pouze cestu C-D.

Pseudoalgoritmus pro nalezení sledu je následující:

```
1. Udělej pro každou silnou komponentu obsahující více než jeden uzel:
   2. for (int i = 0; i < počet uzlů jedné komponenty; i++
      3. Vezmi i a i+1 uzel komponenty a nalezni pro ně sled pomocí DFS
      4. Odstraň smyčky a přidej ho do sledu
   5. Vezmi poslední a první prvek a nalezni pro ně sled pomocí DFS
```

[2]Nelze už mluvit o cestě, protože z definice cesty plyne, že se nesmí opakovat vrcholy.

Kapitola 5

Použité nástroje

V této kapitole jsou přiblíženy nástroje, se kterými jsem se během své práce na diplomové práci setkala. Je zde podrobně popsán nástroj Apache Maven, definovány základní pojmy z jazyka Java a představeny nástroje Gephi a yEd.

5.1 Maven

Tato kapitola se zabývá nástrojem Apache Maven, který aplikace Classycle, tak jak je napsaná, používá. Původní aplikace Classycle tzv. mavenizována nebyla, tento krok udělal až konzultant z firmy. Tento nástroj je velmi rozšířen a používá se pro jednodušší řízení, správu, překlad a sestavování aplikací. Lze jej použít pro aplikace v různých programovacích jazycích, prakticky se užívá pro jazyk Java. Běžně při zmiňování tohoto nástroje se používá místo Apache Maven pouze slovo Maven. I já si dovolím nadále v textu používat zkrácenou variantu, pouze Maven. Maven se dá použít na většině operačních systémů, počínaje Mac OS X, Microsoft Windows, konče Linuxem, FreeBSD, OpenBSD.

Na oficiálních stránkách Mavenu je popsán i význam slova. Původ slova Maven pochází z židovském jazyce Jidiš a volně přeloženo znamená „pramen poznání“. [6]

5.1.1 Hlavní cíle nástroje

Lidé často prohlašují o Mavenu, že je to tzv. sestavovací nástroj *(build tool)*[1], nástroj pro překlad a vše co s překladem souvisí. Je sice pravdou, že je Maven zejména sestavovací nástroj, ale není to jeho jediná funkcionalita. Nástrojem na překlad je i Ant. Ale právě proto, že Ant nebyl dostačující, vznikl Maven. Ant není dostačující, protože je pouze pro přípravu, kompilaci, vytváření balíčků, testování a distribuci. Maven toho umí daleko víc. Mimo toho, co již bylo zmíněno, umí vytvářet zprávy, generovat webové stránky, zjednodušit komunikaci mezi programátory při vývoji. [19]

Hlavní cíl nástroje Maven je pomoci programátorovi, aby dokázal co nejjednodušeji a v co nejkratším čase dokončit vývoj. Aby se tak dělo, Maven si dává za cíl:

- Udělat sestavovací proces jednoduchý.
- Poskytnout jednotný systém sestavování.
- Poskytnout kvalitní informace o projektu.

[1] Sestavovácí nástroje *(build tools)* jsou programy k automatickému vytvoření spustitelné aplikace ze zdrojového kódu. Sestavení zahrnuje kompilaci, propojování a balíčkování kódu do spustitelné formy.

- Poskytnout návod pro “best practice” vývoj.
- Umožnit přehlednou a transparentní integraci nových funkcí do projektu.

Sestavení aplikací psaných v jazyce Java může být občas neintuitivní. Maven zastřešuje detaily sestavení a usnadňuje tím programátorovi práci, neboť není nutné znát detaily překladu.

Maven používá soubor POM – pom.xml. Tento soubor obsahuje vše, co je pro sestavení potřebné. Obsahuje také tzv. rozšiřující moduly, rozšíření (*plugin*[2]). Struktura souboru POM je pro všechny projekty stejná, takže uživatel se nemusí při každém projektu učit zcela novou věc. [6]

Maven kromě hlavní funkce sestavení poskytuje také mnohé další informace o projektu. Díky souboru POM dokáže Maven mimo jiné poskytnout zprávy z jednotkových testů, seznam emailů, seznam závislostí. Navíc se do souboru POM dá přidat spoustu dalších vlasností, které dodají další informace o projektu. [6]

5.1.2 Rozdíl mezi nástroji Ant a Maven

Čtenář by mohl nabýt dojmu, že Apache Maven je nejlepší a Apache Ant se nedá na nic použít. Tak to samozřejmě není. Jak Ant, tak Maven, má své kladné a záporné stránky. Ant je výborný sestavovací nástroj, je stále ve velkém používán. Pokud se uživatel rozhodne pro Ant, tak musí popsat každou instrukci pro kompilaci a vytvoření balíčku. Používá příkazy `javac`, `jar`. Ant musí mít ve své struktuře jasně napsané, kde jsou zdrojové soubory a kam se mají uložit cílové soubory. Maven naopak má jasně definovanou strukturu, cestu k souborům tedy není nutné ručně specifikovat. Ant je procedurální, je tedy nutné, aby mu uživatel krok po kroku popsal, co má dělat, jak to dělat a kdy to dělat. Je nutné mu říct, že je nejprve nutná kompilace, pak kopie a pak komprese. Naopak Mavenu stačí jednoduchý pom.xml, soubory ve správném adresáři a o vše ostatní se Maven postará. Ant nemá žádný životní cyklus. Nedefinuje žádné cíle a závislosti. Vše je nutné provést manuálně. Na druhou stranu Maven definovaný životní cyklus má. Invokuje se, když uživatel provede příkaz `mvn install`. Maven se o zbytek postará. Nevýhodou může být to, že jako vedlejší efekt se spustí plno implicitních rozšíření. [19]

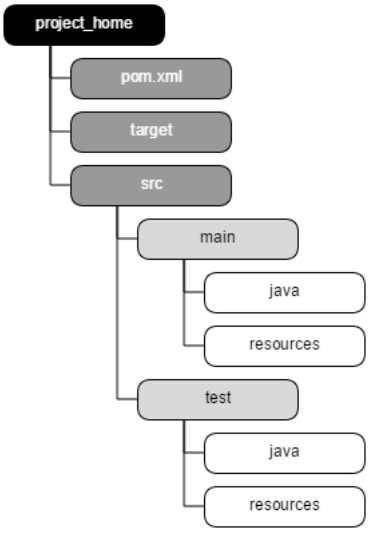
5.1.3 Struktura Mavenu

Maven očekává, že zdrojový kód bude umístěn v `${basedir}/src/main/java`, zdroje v `${basedir}/src/main/`resources a testy v `${basedir}/src/test`. Maven dále předpokládá, že zkompilovaný kód chce uživatel umístit do `${basedir}/target/classes` a distribuovatelný `WAR` soubor do `${basedir}/target`. Pokud by uživatel chtěl webovou aplikaci místo aplikace typu `JAR`[3], pak je nutné změnit projekt na projekt typu `.war` a soubory uložit do `${basedir}/src/main/webapp`. Na rozdíl od nástrojů založených na Antu se tyto lokace nemění, jsou jasně definované a nemusí se tak definovat pro každý projekt. A co víc, Maven má jasně definovaný životní cyklus, set běžných modulů, rozšíření. Pokud uživatel správně založí projekt a dodrží výše zmíněné konvence, Maven se už o vše postará a uživatel nemusí vyvinout žádné další úsilí. Pokud ovšem uživatel touží mít svou vlastní strukturu, vlastní nastavení, může si Maven přizpůsobit k obrazu svému. [19]

Pro jednodušší představu je celá struktura zachycena na obrázku 5.1.

[2] *Plugin* je anglické slovo překládané jako doplněk, rozšíření, rozšiřující modul. Je to modul, který přidává softwaru další funkcionalitu.

[3] Souborový formát, který používá platforma Java podobný ZIP kompresi.



Obrázek 5.1: Struktura souborů. Vytvořeno na základě [19] a [6]

K vytvoření `JAR` souboru stačí mít nějaký zdrojový kód v jazyce Java a ten umístit do `${basedir}/src/main/java`. Dále je nutné mít vytvořený jednoduchý `pom.xml`. Potom již stačí pouze spustit `mvn install` z příkazové řádky. Nejjednodušší pom.xml je zobrazen níže na příkladu.

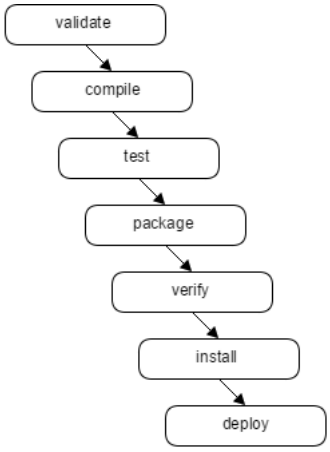
```
<project>
   <modelVersion>4.0.0</modelVersion>
   <groupId>org.sonatype.mavenbook</groupId>
   <artifactId>my-project</artifactId>
   <version>1.0.</version>
</project>
```

Samozřejmě se jedná o nejjednodušší příklad, projekt nemá nic víc než zdrojový kód a vytvoří pouze JAR soubor. [19]

5.1.4 Životní cyklus nástroje Maven

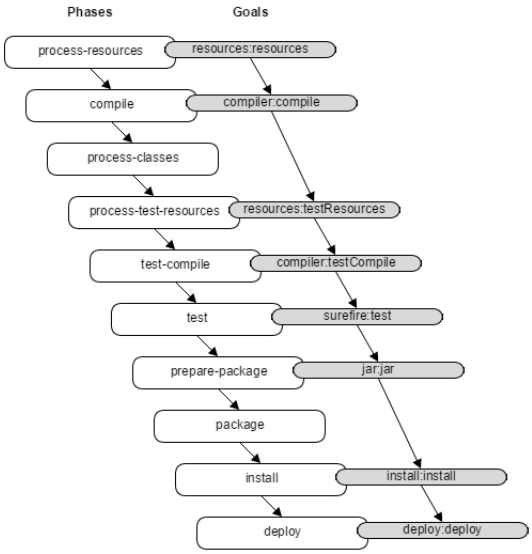
Maven má, na rozdíl od zmíněného nástroje Ant, jasně definovaný životní cyklus. Životní cyklus se může lišit podle druhů balíčku, ten základní se skládá ze základních 7 fázích a jsou vidět na obrázku 5.2. Fáze a cíle jsou popsány anglicky, vysvětleny budou níže.

V první fázi, fázi *validate*, se zkontroluje projekt a dostupnost všech potřebných informací. Ve druhé fázi, fázi *compile*, se projekt kompiluje. Ve třetí fázi *test* se otestuje kompilovaný kód pomocí jednotkových testů frameworku. Ve fázi *package* se kompilovaný kód přetvoří do balíčku vhodného formátu, nejčastěji se jedná o formát `JAR`, může se však jednat i o typ `WAR`, `EAR` a jiné. Pátá fáze *verify* má za úkol zkontrolovat právě vytvořený balíček. Předposlední šestá fáze *install* instaluje balíček vytvořený ve čtvrtém kroku do lokálního repozitáře, aby byla možnost použít tento balíček pro další lokální projekty. Poslední fáze *deploy* zkopíruje finální balíček do repozitáře, odkud může být sdílen dalšími programátory a projekty. [19], [6]



Obrázek 5.2: Životní cyklus Mavenu. Vytvořeno na základě [19] a [6]

Pro jednotlivé fáze, kterými Maven prochází v rámci životního cyklu, se vykonávají cíle, které jsou s nimi spojeny. Každá fáze může mít takových cílů nula až nekonečně mnoho. Níže je popsáno, co vše je potřeba splnit, pokud uživatel chce vytvořit JAR balíček. Je zřejmé, že je nutné provést i předchozí kroky a s tím spjaté cíle předchozích fází. Na obrázku 5.3 lze vidět, co vše je nutné provést, pokud se má vytvořit JAR balíček.



Obrázek 5.3: Fáze Mavenu a s tím související cíle. Vytvořeno na základě [19] a [6]

5.2 Grafické nástroje

V rámci diplomové práce jsem používala dva nástroje, Gephi a yEd. Níže je krátký popis obou nástrojů, jejich výhody a nevýhody. Taktéž uveden výstup pro každý z nich.

5.2.1 Gephi

Tento nástroj byl zvolen mým konzultantem z firmy. Jedná se o volně šiřitelný *(open-source)* nástroj vydáván pod GPL 3 *(„GNU General Public License“)*. Soustředí se na síťovou vizualizaci a analýzu. Používá 3D render pro zobrazení velkých grafů v reálném čase. Umí prozkoumat, analyzovat, filtrovat, shlukovat, manipulovat a exportovat všechny typy sítí. [7]

Nástroj je volně ke stažení na webových stránkách [7]. Dle webových stránek je funkční jak na Windows, tak na Linuxu a Mac OS X. Bohužel z vlastních zkušeností se mi podařilo spustit a používat tento nástroj pouze na platformě Windows.

Gephi je opravdu silný nástroj a je na mnohých internetových fórech vychvalován. Má spoustu výhod, bohužel nevýhody pro mou potřebu převažují.

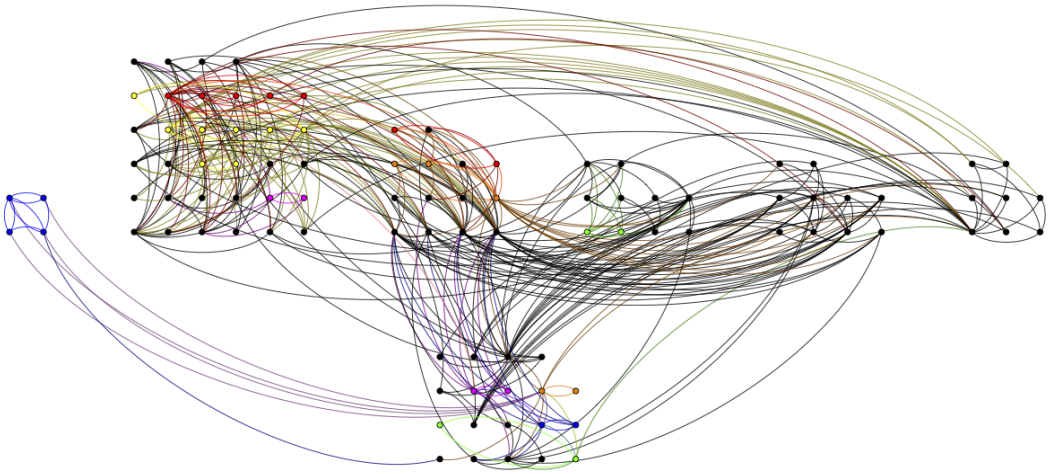
Výhody:

- Jedná se o opravdu silný nástroj, který má integrováno spoustu funkcí, jako je shlukování uzlů, rozmístění uzlů podle klíče, nalezení nejkratší cesty apod.
- Je možné vytvořit vlastní modul rozšiřující funkcionalitu nástroje, spoustu rozšíření je možné stáhnout si z webových stránek od dalších vývojářů.
- Lze jednoduše importovat jednoduchý graf.
- Nástroj je zdarma a je volně šiřitelný.

Nevýhody:

- Nefunguje na všech platformách. Vystihuje to názor mého spolužáka: „Gephi se zdá, že je mnohem mocnější nástroj, který zvládá velké množství dat, ovšem na Ubuntu 16.04 byl tak nestabilní (padal skoro při každé akci), že jsem musel preferovat yEd.“
- Na většině internetových fór se diskutující shodují, že nástroj je to sice mocný, ale jeho naučení se trvá dlouhou dobu. V tomto ohledu lze Gephi přirovnat k nástroji Gimp, který je taktéž mocný a taktéž volně šiřitelný, nicméně naučit se jeho ovládání zabere dost času.
- Ovladatelnost není intuitivní.
- Při importování dat nelze nastavit některé parametry - barvu hran, směr hran apod.

Výstup z nástroje Gephi je uveden na obrázku 5.4. Analyzována je aplikace Classycle, hledají se silné komponenty. Z obrázku lze vidět, že nelze nastavit barva hrany při importování, hrany mají vždy barvu uzlu, ze kterého vycházejí. Hrany nejsou ve tvaru přímky, ale křivky, což taktéž snižuje přehlednost. Nelze zobrazit jména vrcholů, to je možné až manuálně pro každý uzel v nástroji Gephi. Gephi má jiný souřadnicový systém než yEd a proto je nejvyšší balíček umístěn dole místo nahoře.



Obrázek 5.4: Ukázka výstupu nástroje gephi

Soubor, který tvoří aplikace Classycle jako vstup do programu Gephi je typu .gexf a ukázka je vidět na obrázku obrázku 5.5. Oproti yEd formátu je kratší, má 910 řádků. Důvodem je to, že se ve výsledném grafu nezobrazuje tolik věcí (orámování silných komponent chybí), dále také chybí bližší specifikace hran a uzlů (chybí barva, typ, název...).

```
<?xml version='1.0' encoding='UTF-8'?>
<gexf xmlns="http://www.gexf.net/1.2draft" xmlns:viz="http://www.gexf.net/1.1draft/viz" xmlns:xsi="http://
www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="http://www.gexf.net/1.2draft http://
www.gexf.net/1.2draft/gexf.xsd" version="1.2">

  <meta lastmodifieddate="2017-04-22">
    <creator>Classycle</creator>
    <description> Classycle title='/Users/alenaoblukova/IdeaProjects/classycle/target/classycle-1.0.jar'</
    description>
  </meta>
  <graph defaultedgetype="directed">
<!-- CLASSES -->
    <nodes>
      <node id="0" label="classycle.Analyser">
        <viz:color r="0" g="0" b="0" a="1"/>
        <viz:position x="2640" y="540" z="0.0"/>
        <viz:size value="15"/>
      </node>
...
    </nodes>
<!-- EDGES -->
    <edges>
      <edge id="0" source="0" target="1" />
      <edge id="1" source="0" target="2" />
      <edge id="2" source="0" target="6" />
...
    </edges>
  </graph>
</gexf>
```

Obrázek 5.5: Ukázka souboru typu .gexf importovaného do nástroje Gephi

5.2.2 yEd

Nástroj yEd jsem zvolila jako druhý v pořadí, protože Gephi není aktuálně podporováno na jiné platformě než na Windows. Tento nástroj mi byl konzultantem schválen, protože je zdarma a je multiplatformní. Je možné si zaplatit premium verzi pro pokročilejší funkcionalitu. Není zdaleka tak silný, jako Gephi. Neexistuje tak jednoduchá možnost tvorby

vlastních rozšíření, modulů. Sám obsahuje pár algoritmů na zobrazení uzlů ve tvaru stromu, kruhu, apod. Soubor pro import je ve složitějším formátu, než Gephi formát. Velkou výhodou je jednoduché a intuitivní ovládání a možnost online verze. Ke stažení je na webových stránkách [8], online verze je dostupná na [1].

yEd je nástroj, který svou funkcionalitou plně splňuje mé požadavky.

Výhody:

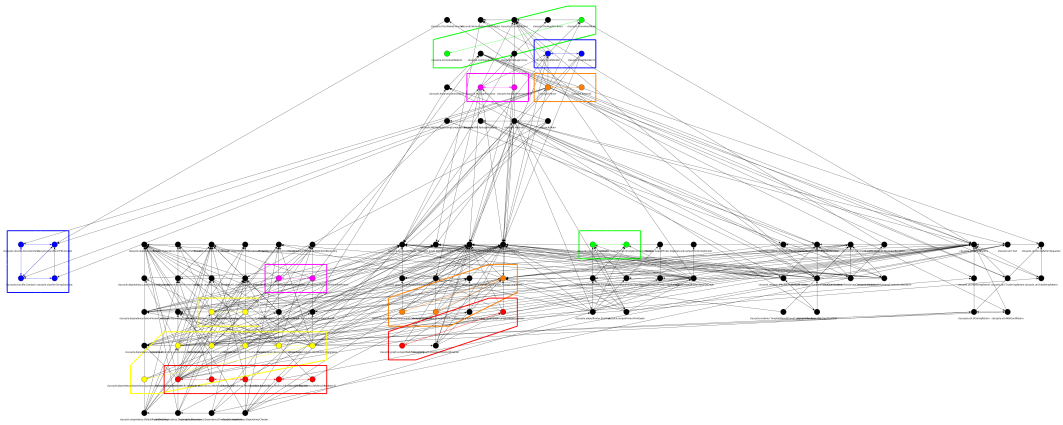
- Multiplatformní.
- Existuje online verze.
- Ovládání je jednoduché, intuitivní.
- Při importování dat je možné definovat si spoustu věcí - přes barvu a typ hrany, po přesnou pozici uzlu, jeho pojmenování, pozici a barvu pojmenování.
- Základní, postačující verze je zdarma.

Nevýhody:

- yEd není tak silný nástroj, jako Gephi.
- Formát importovaného souboru je složitější.
- Není možné si ho pomocí vlastních rozšíření upravit a jednoduše rozšířit jeho funkcionalitu.
- Není to volně šiřitelný software.

Přestože yEd není bez chyb a má také své nevýhody, mnoho nevýhod není pro moje řešení příliš důležitých. Například není možné si ho jednoduše rozšířit pomocí vlastních rozšíření, jeho současná funkcionalita však pro mé potřeby bohatě stačí.

Výstup z nástroje yEd je uveden na obrázku 5.6. Analyzována je aplikace Classycle, hledají se silné komponenty mezi balíčky. Tento obrázek je v práci již ukázán 7.5, proto je zde ve zmenšené podobě. Je vidět, že lze ovlivnit jak barvu vrcholů, tak barvu hran, typ hran, hrany jsou rovné, vrcholy jsou označeny popisky apod.



Obrázek 5.6: Ukázka výstupu nástroje yEd

Soubor, který tvoří aplikace Classycle jako vstup do programu yEd je typu .xml a ukázka je vidět na obrázku 5.7. Oproti Gephi formátu je výrazně delší – má přes osm a půl tisíce řádků. Důvodem je nejen náročnější formát, ale také možnost formátem ovlivnit spoustu důležitých věcí, jako již zmiňovaný detailní popis hrany a podobně.

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<graphml ... >
  <!--Created by yEd 3.17-->
  <key attr.name="Description" attr.type="string" for="graph" id="d0"/>
  <key for="port" id="d1" yfiles.type="portgraphics"/>
  ...
  <graph edgedefault="directed" id="G">
    <data key="d0">/Users/alenaoblukova/IdeaProjects/classycle/target/classycle-1.0.jar</data>
    <node id="0">
      <data key="d5">classycle</data>
      <data key="d6">
        <y:ShapeNode>
          <y:Geometry height="30" width="30" x="2640" y="540"/>
          <y:Fill color="#000000" transparent="false"/>
          <y:BorderStyle color="#000000" raised="false" type="line" width="1.0"/>
          <y:NodeLabel alignment="center" autoSizePolicy="content" fontFamily="Dialog" fontSize="12"
          fontStyle="plain" hasBackgroundColor="false" hasLineColor="false" horizontalTextPosition="center"
           iconTextGap="4" modelName="custom" textColor="#000000" verticalTextPosition="bottom" visible="
          true">classycle.Analyser<y:LabelModel>
              <y:SmartNodeLabelModel distance="4.0"/>
            </y:LabelModel>
            <y:ModelParameter>
              <y:SmartNodeLabelModelParameter labelRatioX="0.0" labelRatioY="-0.5" nodeRatioX="0.0"
              nodeRatioY="0.5" offsetX="0.0" offsetY="10.0" upX="0.0" upY="-1.0"/>
            </y:ModelParameter>
          </y:NodeLabel>
          <y:Shape type="ellipse"/>
        </y:ShapeNode>
      </data>
    </node>
...
    <edge id="ee511" source="325" target="324">
      <data key="d10">
        <y:PolyLineEdge>
          <y:Path sx="0.0" sy="0.0" tx="0.0" ty="0.0"/>
          <y:LineStyle color="#FF0000" type="line" width="5"/>
          <y:Arrows source="none" target="none"/>
          <y:BendStyle smoothed="false"/>
        </y:PolyLineEdge>
      </data>
    </edge>
```

Obrázek 5.7: Ukázka souboru typu .xml importovaného do nástroje yEd

5.3 Jazyk Java

Java je jedním z mnoha programovacích nástrojů. Jejími klíčové vlastnosti spočívají v tom, že je objektově orientovaná, jednoduchá na pochopení, multiplatformní, Java je jak jazyk, tak i platforma (díky virtuálnímu stroji Java (*Java Virtual Machine*), dále jen *JVM*).
V rámci tohoto dokumentu jsou použity pojmy jako je objekt, třída, balíček. Proto je nutné v této části krátce definovat tyto pojmy.[17]

Bajtkód je jazyk virtuálního stroje. Java bajtkód, který aplikace Classycle analyzuje, je instrukční sada JVM.

Třída popisuje společné vlastnosti svých instancí a definuje také nástroj pro jejich vytváření. Definuje své proměnné a metody. Tříd může být více typů – může se jednat o abstraktní třídu, třídy povolující vytvořit pouze jednu instanci, tzv. jedináčka apod.
V jazyce Java neexistuje žádná třída tříd, jako je tomu například v jazyce Smalltalk. Názvy tříd jsou složeny z názvu třídy a jména balíčku, v němž se třída nachází. [17]

Objekt je instance třídy, tedy konkrétní realizace obecného vzoru definovaného v příslušné třídě. Třída může mít obecně libovolný počet instancí. Pojmy objekt a instance jsou synonyma. Objekty si navzájem posílají zprávy, které jsou zpracovány metodami [17]. Metody jsou funkce objektu, proto při definování ideální velikosti v kapitole pachy kódu 2.3.2 můžeme pojmy funkce a metoda v tomto kontextu zaměňovat.

Balíček slouží ke sdružování tříd. Balíčky jsou hierarchicky uspořádány. Balíček slouží za účelem sloučení těch tříd, které spolu logicky souvisejí. Zdrojový text musí na začátku obsahovat jméno balíčku, ve kterém se nachází, název balíčku by měl být stejný, jako název složky, v níž je soubor umístěn. Je také zavedenou konvencí, že balíčky obsahují pouze malá písmena. Balíček může obsahovat další balíčky – podbalíčky. Ty by spolu měly logicky souviset, takový je alespoň dobrý zvyk. Podbalíčky mohou mít opět další podbalíčky. Název podbalíčku se skládá z názvu rodiče a názvu podbalíčku odděleného tečkou. [17]

Aplikace Classycle ja napsána v jazyce Java verze 8. V této verzi vzniká aktuálně nejvíce projektů. Jazyk byl zvolen autorem aplikace Classycle.

Kapitola 6

Classycle

Aplikace Classycle slouží ve firmě v jedné fázi vývoje automatizovaných testů, ve kterých kontroluje pravidla. Aplikace má však širší využití, o čemž je zmínka v kapitole Implementace, sekce Využití aplikace 7.3.

Všechny níže uvedené informace jsem musela nastudovat proto, abych mohla implementovat změny požadované firmou, požadované zadáním diplomové práce. Studium těchto informací bylo náročné, neboť samotná aplikace byla napsána jiným člověkem. Jiný programovací styl, jiné uvažování, chybějící komentáře u důležitých částí kódu, ne vždy vhodné pojmenování proměnných... Všechny tyto body byly velkou překážkou a zpomalovaly pochopení aplikace.

V této kapitole je vysvětleno, k čemu se aplikace používá ve firmě, pro kterou tuto aplikaci upravuji. Dále je popsána struktura aplikace a následně i samotný běh aplikace.

6.1 Použití aplikace Classycle při vývoji ve firmě

Aby bylo možné popsat zařazení aplikace Classycle ve firmě, je nutné popsat, co předchází jejímu užití a co mu následuje.

Aplikace Classycle se ve firmě používá ke kontrole závislostí mezi třídami a balíčky. Kontroluje se kód, který napsali programátoři automatizovaných testů.

Následující obrázek 6.1 popisuje bod, ve kterém se aplikace spouští.

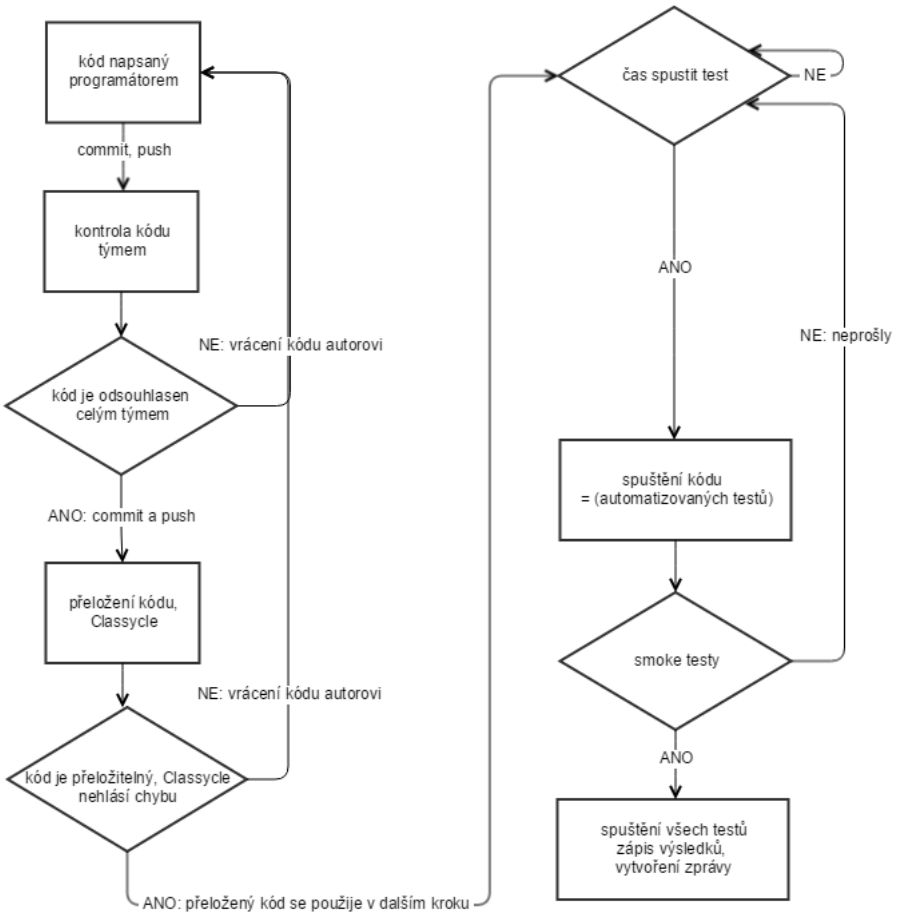
6.1.1 První krok

Jak je vidět, prvním krokem je vytvořit vůbec kód, který má aplikace testovat. Tímto kódem jsou myšleny již zmíněné automatizované testy. Automatizované testy jsou psány ve firmě v oddělení QA *(Quality Assurance)* , oddělení zabývající se zajištěním kvality.
V rámci QA oddělní probíhá i QC *(Quality Control)*, řízení kvality, což zahrnuje testování. Testováním se zabývá kapitola 3.

Testeři napíší automatizovaný test. Na svém počítači si tento test spustí, zkontrolují. Když jsou s ním spokojeni, publikují ho pro ostatní programátory.

6.1.2 Druhý krok

Aby bylo možné popsat druhý krok, je nutné zmínit, že firma, ve které se Classycle používá, využívá oblíbený verzovací nástroj Git.



Obrázek 6.1: Ukázka zařazení Classycle při vývoji

Git, jako další verzovací nástroje, stojí na principu, že každý programátor pracuje na své části na svém počítači. Když má hotový ucelený kus kódu, svou změnu propaguje mezi ostatní programátory. Pro promítnutí změny se používají příkazy `commit`, `push` a `pull`. Tyto výrazy nemají v českém jazyce vhodný ekvivalent. Dále v textu budu používat tyto anglické tvary.

Jakmile programátor provede `commit` a `push` své změny v kódu, ostatní programátoři mají prostor se ke kódu vyjádřit. Tento bod je důležitý, protože při jakékoliv činnosti se stává, že člověk je jistým způsobem zaslepen, omezen jen na svoji část a nějaké zjevné chyby přehlíží. Ostatní ještě nejsou problémem natolik ovlivněni a můžou chybu objevit. Pokud některý z programátorů objeví chybu, kód zamítne a autor kódu musí chybu odstranit. Jakmile jsou všichni v týmu s kódem spokojeni, schválí ho, programátor s právy na `push` do hlavní větve jej provede.

6.1.3 Třetí krok

V tomto kroku se ke slovu konečně dostává aplikace Classycle. Jako všechny předchozí kroky, i tento je spuštěn automaticky po každém příkazu `push` nového kódu. Tento třetí krok je jakousi záchrannou brzdou, která má zabránit v tom, aby automatizované testy obsahovaly nějakou hrubou chybu, která by bránila v jejich automatickému spuštění.

V tomto kroku se provádí překlad testů a spouští se aplikace Classycle. Pokud by se zjistilo, že jsou testy nepřeložitelné a nebo obsahují zakázané vazby, nebude se s touto verzí testů v dalším kroce pracovat a je nutné chyby manuálně opravit. Classycle kontroluje pouze automatizované testy, nekontroluje aplikaci, která je ve firmě vyvíjena – tu kontrolují mimo jiné právě automatizované testy.

6.1.4 Čtvrtý krok

Tento krok je spuštěn nezávisle na jakékoliv aktivitě programátorů, je spouštěn časovačem. V tomto kroku se spouští automatizované testy, které testují aplikaci vyvíjenou firmou. Spouští se vždy poslední validní kód z kroku tři. Pokud aplikace Classycle objeví v kroku tři chybu, bude se tento poslední krok spouštět stále dokola s neaktuálním kódem, dokud se chyba z kroku tři neodstraní. Toto spuštění testů má hned několik kroků.

V první řadě se pustí takzvané *Smoke testy* 3.2.2, tedy testy, které mají za úkol otestovat základní funkcionalitu – aplikaci lze spustit, uživatel se může přihlásit a odhlásit. Šíře Smoke testů závisí vždy na konkrétní situaci. Smoke testy jsou rychlé, v řádu minut, a mají otestovat, zda má cenu spouštět kompletní testovací sadu. Celá testovací sada trvá pak v řádech několika hodin. Automatizované testy odhalují chyby aplikace, která se ve firmě vyvíjí. Jakmile se objeví chyba, testy chybu zaznamenají, vytvoří kopii obrazovky ve stavu, kdy chyba nastala, a zaznamenají si chybovou hlášku. Jakmile všechny testy proběhnou, vytvoří se zpráva o úspěšných a neúspěšných testech. V případě neúspěšných testů se vypíší bližší informace.

6.2 Původní vstupy a výstupy aplikace

Aplikace Classycle pracuje s bajtkódem 5.3. Ten analyzuje a hledá závislosti mezi třídami a díky tomu závislosti mezi balíčky.

Ve firmě, pro kterou aplikaci upravuji, se v současné době používá aplikace pouze v módu, který detekuje porušení pravidel. O pravidlech je psáno více ve struktuře aplikace 6.4.6.

Aplikace se spouští nepřímo, spouští ji modul nástroje Maven. Modul je upraven pro konkrétní potřeby firmy. Tento modul není k diplomové práci přiložen, aplikaci lze používat i bez něj. Ukázka spuštění:

```
$ mvn verify -DskipTests=true
[INFO] Scanning for projects...
[INFO]
[INFO]------------------------
[INFO] Building Schoolwires System Tests 1.0-SNAPSHOT
[INFO] ------------------------
Downloading: https://oss.sonatype.org/content/repositories/snapshots/org/
bithill/aport/0.2-SNAPSHOT/maven-metadata.xml
Downloaded: https://oss.sonatype.org/content/repositories/snapshots/org
/bithill/aport/0.2-SNAPSHOT/maven-metadata.xml
(2 KB at 1.0 KB /sec)
[INFO]
[INFO] - maven-resources-plugin:2.6:resources (default-resources)
@ systemtests -
```

```
[INFO] Using 'UTF-8' encoding to copy filtered resources.
[INFO] Copying 17 resources
[INFO]
[INFO] - maven-compiler-plugin:3.3:compile (default-compile)
@ systemtests -
[INFO] Changes detected - recompiling the module!
[INFO] Compiling 566 source files to C:...path...[1]
[INFO]
[INFO] - maven-resources-plugin:2.6:testResources (default-testResources)
@ systemtests -
[INFO] Using 'UTF-8' encoding to copy filtered resources.
[INFO] skip non existing resourceDirectory C:...path...
[INFO]
[INFO] - maven-compiler-plugin:3.3:testCompile (default-testCompile)
@ systemtests -
[INFO] No sources to compile
[INFO]
[INFO] - maven-surefire-plugin:2.12.4:test (default-test) @ systemtests -
[INFO] Tests are skipped.
[INFO]
[INFO] - maven-jar-plugin:2.4:jar (default-jar) @ systemtests -
[INFO] Building jar: C:...path...
[INFO]
[INFO] - maven-failsafe-plugin:2.19.1:integration-test (integration-test)
@ systemtests -
[INFO] Tests are skipped.
[INFO]
[INFO] - Classycle-maven-plugin:0.5:check (verify) @ systemtests -
[INFO]
[INFO] - maven-failsafe-plugin:2.19.1:verify (verify) @ systemtests -
[INFO] Tests are skipped.
[INFO] ------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------
[INFO] Total time: 24.517 s
[INFO] Finished at: 2016-12-05T14:28:48+01:00
[INFO] Final Memory: 31M/476M
[INFO]------------------------
```

Výše je uvedené spuštění aplikace pomocí modulu nástroje Maven. Pokud je aplikace Classycle v pořádku, spustí se samotná analýza bajtkódu. Výstup aplikace se přesměruje do nově vzniklého souboru `target/Classycle/checkresults.txt`. Níže uvedený výstup ukazuje, že vše proběhlo v pořádku, firemní automatizované testy neobsahovaly žádné zakázané závislosti.

[1] Cesta je ve výpisech nahrazena sekvencí „...path...“, neboť obsahovala název firmy a vyvíjený produkt

```
show onlyShortestPaths allResults
check ....image.* independentOf org.testng.* OK[2]
check .....image.* independentOf com.google.common.truth.* OK
check .....image.* independentOf org.openqa.selenium.* OK
check .....image.* independentOf org.bithill.selenium.* OK
check .....time.* independentOf org.testng.* OK
check ....time.*
independentOf com.google.common.truth.* OK
check ....time.* independentOf org.openqa.selenium.* OK
check ....time.* independentOf org.bithill.selenium.* OK
check ....string.* independentOf org.testng.* OK
check ....string.* independentOf com.google.common.truth.* OK
check ....string.* independentOf org.openqa.selenium.* OK
check ....string.* independentOf org.bithill.selenium.* OK
check ....testing.testmanagement.* independentOf org.testng.* OK
check ....testing.testmanagement.* independentOf com.google.common.truth.* OK
check ....testing.testmanagement.* independentOf org.openqa.selenium.* OK
check ....testing.testmanagement.* independentOf org.bithill.selenium.* OK
check ....testing.db.* independentOf org.testng.* OK
check ....testing.db.* independentOf com.google.common.truth.* OK
check ....testing.db.* independentOf org.openqa.selenium.* OK
check ....testing.db.* independentOf org.bithill.selenium.* OK
check ....image.* dependentOnlyOn javax.media.jai.* com.sun.media.jai.*
java.awt.* java.io.* OK
check ....string.* dependentOnlyOn java.lang.* java.util.* OK
check ....time.* dependentOnlyOn java.lang.* java.util.*
java.time.* org.apache.logging.log4j.* OK
check ....testing.selenium.* independentOf org.testng.* OK
check ....testing.selenium.* independentOf com.google.common.truth.* OK
check ....testing.testng.* independentOf ....automation.* OK
check ....testing.testng.* independentOf ....testing.usecases.* OK
check ....testing.testmanagement.*
dependentOnlyOn java.lang.* java.util.* javax.annotation.* OK
check ....automation.* independentOf ....testing.usecases.* OK
```

Může se stát, že programátor vytvoří či upraví automatizované testy tak, že poruší pravidla a vytvoří nepovolenou závislost. V takovém případě Classycle skončí chybou, jeho návratová hodnota je 1. Výpis z textového souboru `target/Classycle/checkresults.txt` ukazuje příklad toho, že vstup obsahuje nepovolené závislosti. Dává dokonce podrobnější popis toho, co přesně za zakázanou závislost se ve vstupu vyskytuje.

```
show onlyShortestPaths allResults
check ....image.* independentOf org.testng.*
   Unexpected dependencies found:
   ....image.Thumbnailer
```

[2] Začátek cesty ve výpisech je nahrazen sekvencí ..., neboť obsahovala název firmy a vyvíjený produkt

```
        -> org.testng.asserts.Assertion
check ....image.* independentOf com.google.common.truth.* OK
...
```

Aplikace před změnou pracovala ve dvou módech, výše je uvedený mód první.

První mód analyzoval závislosti mezi třídami a jeho výstupem byl seznam jednotlivých silně souvislých komponent. Výstup byl ve formátu xml, csv a nebo nezpracovaný výstup. Ukázky jednotlivých výstupů jsou uvedeny níže. Aby bylo možné srovnávat formáty, analýza je vždy prováděna nad samotnou aplikací Classycle. V kapitole Implementace 7 jsou uvedeny výstupy, které byly vytvořeny v rámci této diplomové práce. I tyto výstupy vznikly analýzou aplikace Classycle. Čtenář si tedy můžete jednotlivé způsoby výstupu objektivně porovnat. Výstup ve formátu xml má 6176 řádků. 6.2 Výstup ve formátu csv má pouze 98 řádků, ale je velice špatně čitelný. 6.3 Nezpracovaný *(raw)* výstup má 919 řádků. 6.4

```
<?xml version='1.0' encoding='UTF-8'?>
<?xml-stylesheet type='text/xsl' href='reportXMLtoHTML.xsl'?>
<classycle title='/Users/alenaoblukova/IdeaProjects/classycle/target/classycle-1.0.jar'
date='2017-04-09'>
  <cycles>
    <cycle name="classycle.outputPrinter.Node et al." size="2" longestWalk="0" girth="2" radius="1"
    diameter="1" bestFragmentSize="1">
      <classes>
        <classRef name="classycle.outputPrinter.Node" eccentricity="1" maximumFragmentSize="1"/>
        <classRef name="classycle.outputPrinter.StronglyConnectedComponent" eccentricity="1"
        maximumFragmentSize="1"/>
      </classes>
      <centerClasses>
        <classRef name="classycle.outputPrinter.Node"/>
        <classRef name="classycle.outputPrinter.StronglyConnectedComponent"/>
      </centerClasses>
      <bestFragmenters>
        <classRef name="classycle.outputPrinter.Node"/>
        <classRef name="classycle.outputPrinter.StronglyConnectedComponent"/>

        ...

    </cycle>
  </cycles>
  <classes numberOfExternalClasses="63">
    <class name="classycle.Analyser" sources="/Users/alenaoblukova/IdeaProjects/classycle/target/
    classycle-1.0.jar" type="class" innerClass="false" size="14619" usedBy="2" usesInternal="21"
    usesExternal="23" layer="8" cycle="">
      <classRef name="classycle.Main" type="usedBy"/>
      <classRef name="classycle.dependency.DependencyChecker" type="usedBy"/>
      <classRef name="classycle.graph.StrongComponentAnalyser" type="usesInternal"/>
      <classRef name="java.io.IOException" type="usesExternal"/>
      <classRef name="java.lang.RuntimeException" type="usesExternal"/>

      ...
```

Obrázek 6.2: Ukázka výstupu v xml

```
class name,type,inner class,size,used by,uses internal classes,uses external classes,layer
index,cycle,source
classycle.Analyser,class,false,14619,2,21,23,8,,/Users/alenaoblukova/IdeaProjects/classycle/target/
classycle-1.0.jar
classycle.AnalyserCommandLine,class,false,3130,2,1,5,5,,/Users/alenaoblukova/IdeaProjects/classycle/
target/classycle-1.0.jar
classycle.ClassAttributes,class,false,2446,9,1,2,3,,/Users/alenaoblukova/IdeaProjects/classycle/target/
classycle-1.0.jar
classycle.ClassNameExtractor,class,false,3639,1,1,6,1,,/Users/alenaoblukova/IdeaProjects/classycle/
target/classycle-1.0.jar
classycle.CommandLine,abstract class,false,3444,2,6,5,4,,/Users/alenaoblukova/IdeaProjects/classycle/
target/classycle-1.0.jar
classycle.GraphBuilder,class,false,3833,3,7,8,5,classycle.GraphBuilder and inner classes,/Users/
alenaoblukova/IdeaProjects/classycle/target/classycle-1.0.jar
classycle.GraphBuilder$1,class,true,1263,1,4,3,5,classycle.GraphBuilder and inner classes,/Users/
alenaoblukova/IdeaProjects/classycle/target/classycle-1.0.jar
classycle.Main,class,false,2255,0,5,8,10,,/Users/alenaoblukova/IdeaProjects/classycle/target/
classycle-1.0.jar
classycle.NameAndSourceAttributes,abstract class,false,1529,4,1,7,2,,/Users/alenaoblukova/IdeaProjects/
classycle/target/classycle-1.0.jar
classycle.PackageAttributes,class,false,1349,1,2,4,4,,/Users/alenaoblukova/IdeaProjects/classycle/target
/classycle-1.0.jar
classycle.PackageProcessor,class,false,3987,3,8,8,6,classycle.PackageProcessor and inner classes,/Users/
alenaoblukova/IdeaProjects/classycle/target/classycle-1.0.jar
classycle.PackageProcessor$Arc,class,true,876,1,3,1,6,classycle.PackageProcessor and inner classes,/
Users/alenaoblukova/IdeaProjects/classycle/target/classycle-1.0.jar
classycle.PackageVertex,class,false,684,1,4,1,5,,/Users/alenaoblukova/IdeaProjects/classycle/target/
classycle-1.0.jar

...
```

Obrázek 6.3: Ukázka výstupu v csv

```
class classycle.Analyser (14619 bytes) sources: /Users/alenaoblukova/IdeaProjects/classycle/target/
classycle-1.0.jar
    class classycle.graph.StrongComponentAnalyser (2120 bytes) sources: /Users/alenaoblukova/IdeaProjects/
    classycle/target/classycle-1.0.jar
    unknown external class java.io.IOException
    unknown external class java.lang.RuntimeException
    unknown external class java.util.HashSet
    class classycle.ClassAttributes (2446 bytes) sources: /Users/alenaoblukova/IdeaProjects/classycle/
    target/classycle-1.0.jar
    class classycle.PackageProcessor (3987 bytes) sources: /Users/alenaoblukova/IdeaProjects/classycle/
    target/classycle-1.0.jar
    class classycle.renderer.TemplateBasedClassRenderer (2414 bytes) sources: /Users/alenaoblukova/
    IdeaProjects/classycle/target/classycle-1.0.jar
    unknown external class java.lang.StringBuilder
    class classycle.renderer.PlainStrongComponentRenderer (1613 bytes) sources: /Users/alenaoblukova/
    IdeaProjects/classycle/target/classycle-1.0.jar
    unknown external class java.lang.IllegalStateException
    unknown external class java.util.Date
    class classycle.renderer.XMLStrongComponentRenderer (4570 bytes) sources: /Users/alenaoblukova/
    IdeaProjects/classycle/target/classycle-1.0.jar
    class classycle.renderer.XMLClassRenderer (1064 bytes) sources: /Users/alenaoblukova/IdeaProjects/
    classycle/target/classycle-1.0.jar
    class classycle.XMLPackageStrongComponentRenderer (837 bytes) sources: /Users/alenaoblukova/
    IdeaProjects/classycle/target/classycle-1.0.jar
    class classycle.XMLPackageRenderer (1029 bytes) sources: /Users/alenaoblukova/IdeaProjects/classycle/
    target/classycle-1.0.jar
    unknown external class java.lang.Integer
    unknown external class java.util.ArrayList
    class classycle.graph.StrongComponent (6298 bytes) sources: /Users/alenaoblukova/IdeaProjects/
    classycle/target/classycle-1.0.jar
    class classycle.outputPrinter.PrintGephi (5592 bytes) sources: /Users/alenaoblukova/IdeaProjects/
    classycle/target/classycle-1.0.jar

...
```

Obrázek 6.4: Ukázka nezpracovaného „raw“ výstupu

Druhý mód pouze vypisoval odpověď o porušení či neporušení na standartní výstup. Tento výstup je pomocí modulu nástroje Maven 5.1 přesměrován do souboru `checkresult.txt`, který je zmiňován výše v textu.

Oba dva módy měly svou vlastní metodu `main`. V aplikaci existovala ještě třetí metoda `main`, která nebyla nikde zdokumentována, její použití jsem nikde v aplikaci nenašla.

Výstup z prvního módu je značně rozsáhlý (výstup ve formátu xml má 6176 řádků), obsahuje pro nezasvěceného uživatele příliš informací, které nejsou rozumně strukturované. Pro člověka je tento výstup hůře uchopitelný, než grafické zobrazení. Tento dojem nemám jen já, ale má ho i konzultant z firmy. Právě to byl důvod pro vznik této diplomové práce.

Tato část aplikace se dle požadavků konzultanta z firmy musela modifikovat. Upravila jsem Classycle tak, aby dával výstup, ze kterého by se daly lépe vyčíst jednotlivé vazby mezi třídami a balíčky. Proto novým způsobem výstupu je grafické zobrazení dat. Změny, které jsem v aplikaci provedla, jsou popsány v kapitole Implementace 7.

6.3 Struktura aplikace

Aby bylo možné změny požadované zadáním implementovat, bylo nutné nejprve celou aplikaci analyzovat a pochopit, jak funguje. Analyzování aplikace je značně náročný úkol. Abych mohla do kódu Classyclu jakkoliv zasáhnout, bylo nutné provést důkladnou analýzu stávajícího řešení. Analýza byla složitá z více důvodů:

- Aplikace je rozsáhlá, před změnou obsahovala 6 balíčků[3], tříd bylo před změnou 84. Počty po změně jsou uvedeny níže v textu.
- Aplikaci jsem nepsala celou já, psal ji jiný programátor. Na tohoto programátora nemám žádný kontakt. Jeho programovací styl je jiný, jeho uvažování a vyjadřování je jiné.
- Spuštění aplikace není triviální záležitostí. Jelikož Classycle používá sestavovací nástroj Maven 5.1, bylo potřeba se s tímto nástrojem důkladně seznámit a naučit se s ním pracovat.
- Kód neobsahoval dostatečný počet komentářů. Každá metoda okomentována byla, jednotlivé kroky v metodě však téměř žádný komentář neobsahovaly.
- Classycle se používá ve firmě, jejíž postupy jsem se musela naučit, abych pochopila praktické využití aplikace a její začlenění při vývoji.
- Aplikace pracuje s Java bajtkódem. Jeho strukturu bylo též nutné nastudovat.

Z výše uvedených bodů lze vidět, že náročnost mé diplomové práce nespočívala pouze v nastudování teoretických informací a implementaci vedoucí k řešení problému. Nezanedbatelný podíl na náročnosti mělo i seznámení se s aplikací a pochopení jejího fungování.

Poznatky, které jsem při studiu aplikace získala, popisuje tato kapitola.

6.3.1 Balíčky

Původní aplikace obsahovala 6 balíčků: `classycle`, ten obsahoval dalších 5: `classfile`, `dependency`, `graph`, `renderer` a `util`. Nejvyšší třída `classycle` obsahovala celkem 14 tříd, které se používají v prvním a druhém módu aplikace. Jednou z nejdůležitějších tříd zde je třída `Analyser`, která obsahovala metodu `main` a byla tak jedním ze vstupních bodů do programu. Balíček `classfile` obsahuje třídy reprezentující jednotlivé konstanty

[3] definice pojmu balíček je v kapitole Použité nástroje, v sekci Jazyk Java 5.3

z pole `ConstPool`, o kterém je řeč v sekci Analýza bajtkódu 6.4.4. Tento balíček prošel refaktorizací. Balíček `dependency` obsahoval další vstupní bod do programu a to ve třídě `DependencyChecker`. Balíček sdružuje třídy, které mají na starost kontrolu pravidel při spuštění aplikace v druhém módu. V balíčku `graph` jsou třídy reprezentující graf z pohledu teorie grafů. Předposlení jmenovaný balíček `renderer` sdružuje třídy starající se o správný formát výpisu výsledků. Třídy, které sdružují různé styly výpisu (prostý text a různé typy řetězců), jsou ukryté v balíčku `util`. Během refaktorizace jsem vytvořila nový balíček `outputPrinter`, o kterém bude zmínka v rámci refaktorizace a zejména implementace. Tento balíček sdružuje třídy vytvořené v rámci diplomové práce.

6.3.2 Třídy

Třídy main: původní aplikace obsahoval celkem tři vstupní body do programu – tři třídy `main`. O dvou je psáno výše, poslední třída `main` byla v původní verzi ukryta v balíčku `classfile` ve třídě `ConstantPoolPrinter`. Tato třída však byla v rámci refaktorizace odstraněna.

Aplikace obsahuje několik důležitých tříd, se kterými dále v textu pracuji. První je již zmíněná třída `Analyser`, která obsahuje metody a privátní proměnné, které se používají napříč celým projektem. Jedná se zejména o `_classAnalyser`[4] a `_packageAnalyser` a metody, které je plní či navracejí. V těchto proměnných se ukrývá analyzovaný bajtkód reprezentovaný formou grafu. Zajímavostí je, že v případě druhého módu, tedy módu zkoumajícího porušení vstupních pravidel, se spouštěla třída `main` z balíčku `dependency`. Přesto však se v rámci metody `main` vytvářela instance třídy `Analyser`, protože bylo nutné používat její metody. Třída `Parser`, která se též nachází na úrovni třídy `Analyser` v balíčku `classycle`, se analyzuje, rozebírá, vstup. Pro každou třídu ze vstupu se vytváří struktura, která reprezentuje jeden uzel v grafu. Naplnění struktury se děje v rámci analýzy třídy, bližší popis analýzy bajtkódu je v sekci Analýza bajtkódu 6.4.4. Struktura obsahuje informace o jméně třídy, o typu třídy, o třídách, které je využívají a také o třídách, které využívá ona sama. Další důležitá třída se jmenuje `Constant`. V ní se analyzuje samotný bajtkód. Každá třída je reprezentována třídou `Vertex`, do češtiny přeloženo `Vrchol`. V této třídě je uloženo pole vektorů hlav (*heads*) a ocasů (*tails*). Výrazy hlava a ocas se často používají v teorii grafů pro označení vrcholu, ze kterého směřuje hrana a vrcholu, do kterého hrana míří. V interní reprezentaci neexistuje množina hran. Dále obsahuje atributy, které se liší pro balíček, třídu, graf. V případě balíčků je vnitřní reprezentace totožná.

6.4 Běh aplikace

6.4.1 Parametry aplikace: původní stav

Původní struktura aplikace obsahovala tři metody `main`. Jedna byla ve třídě `Analyser` a druhá ve třídě `DependencyChecker`. Implicitně se spouštěl `main` ze třídy `Analyser`, který reprezentoval první mód aplikace. V tomto prvním módu měla aplikace následující parametry: `[-raw] [-packagesOnly] [-cycles|-strong] [-xmlFile=<file>] [-csvFile=<file>] [-title=<title>] [-mergeInnerClasses] [-includingClasses=<pattern1>,<pattern2>,...] [-excludingClasses=<pattern1>,<pattern2>,...]`

[4] Autor programu Classycle často používá proměnné začínající symbolem „ _ “, ne však vždy. Nepřišla jsem na pravidlo, podle kterého se autor rozhodoval, zda proměnná začne či nezačne symbolem „ _ “.

`[-reflectionPattern=<pattern1>,<pattern2>,...]`
`<class files, zip/jar/war/ear files, or folders>.`
Parametry `-raw, -xmlFile, -csvFile` určují druh výstupu. Díky parametru `-packagesOnly` se zkoumají pouze balíčky a ne třídy. Parametrem `-title` se dá nastavit titulek výstupu v xml, díky `-mergeInnerClasses` se vnitřní třídy sloučí se svými nadřazenými třídami a budou se zkoumat jako jedna velká třída. Parametry `-includingClasses, -excludingClasses` se dají přidat nebo vyřadit třídy, které aplikace analyzuje. Pomocí `-reflectionPattern` se dá nastavit, aby i řetězce byly zkoumány jako třída. Díky tomu například řádky kódu *Class clazz = Thread.class;* a *String className = "java.lang.Thread";* mají stejný efekt. Výběrem `-cycles, -strong` se dá nastavit, zda se na výstup budou tisknout pouze cykly nebo také silné komponenty obsahující pouze jednu třídu (vytisknou se tedy všechny třídy). O silných komponentách je zmínka v kapitole Grafy 4. Posledním parametrem je cesta ke zkoumanému souboru.

Druhý mód aplikace zkoumající povolené a nepovolené závislosti se spouštěl následujícím způsobem: `java -cp classycle.jar classycle.dependency.DependencyChecker`. Dále obsahoval podobné parametry, jako první mód. Navíc musel povinně obsahovat parametr `-dependencies`. Za tímto parametrem následovala pravidla nebo cesta k souboru s pravidly. Pravidla popisují povolené a nepovolené závislosti tříd a balíčků ve zkoumané aplikaci. Způsob definování pravidel je rozepsán v 6.4.6.

6.4.2 Ukončení aplikace

V případě nevalidně zadaných parametrů skončí aplikace chybou, návratová hodnota je 1. Nevalidní parametry mohou být z více příčin. První je, že programátor zadá neexistující parametry (včetně pravopisných chyb). Druhou je chyba v cestě k souboru či souborům. Aby mohla aplikace úspěšně skončit (úspěšností je myšlena návratová hodnota 0), musí být parametry validně zadány. Návratovou hodnotu 1 může způsobit i nedodržení formátu psaní pravidel. Aplikace spuštěná v prvním módu vrací v případě výše splněných podmínek návratovou hodnotu 0. Aplikace spuštěná v druhém módu vrací návratovou hodnotu pouze tehdy, pokud jsou splněny jak výše uvedené podmínky, tak pokud nejsou porušena pravidla zadaná programátorem pomocí parametru na vstupu. Pokud zadaná pravidla porušena jsou, aplikace končí chybou, návratovou hodnotou 1.

6.4.3 Spuštění aplikace

Vstupem do aplikace je metoda `main`. Nejdříve se zpracují parametry. Dle parametrů se zavolá metoda `run` ze třídy `Anaylyser` nebo `runDependency` ze třídy `DependencyChecker`.

Classycle pracuje ve dvou módech. První mód hledá silné komponenty v grafu. Druhý mód kontroluje, zda jsou splněna pravidla

První mód začne zavoláním metody `run` ze třídy `Anaylyser`. Ta ihned na počátku zavolá metodu `readAndAnalyse`, ve které začne výpis na standartní výstup. Výpis vypadá například takto:

```
============= Classycle V 1.5.0 =============
========== by Franz-Josef Elmer ==========
========== edited by Alena Oblukova ==========
read class files and create class graph ...
   done after 33 ms: 10 classes analysed.
create package graph ... done after 5 ms: 7 packages.
condense package graph ... done after 6 ms: 3 strong components found.
```

`calculate package layer indices ... done after 1 ms.`
Během výpisu těchto informací probíhá samotná analýza zkoumaného vstupu, hned po slovech `read class files and create class graph` se volá metoda `createClassGraph`. Jakmile je veškerá analýza hotová a data jsou zpracovány, nastaví se dle zadaných parametrů jeden z možných výstupů – výstup na standartní výstup (`stdout`), výstup ve formátu `.csv`, `.xml` apod.

Druhý mód na počátku nevypíše nic. Nejprve je zavolána metoda `runDependency` ze třídy `DependencyChecker`. V této třídě se vytvoří výstup na standartní výstup a pokud je parametrem zadán výstup do formátu přijímaného aplikací yEd, vytvoří se i tento výstup. Následně se zavolá funkce `check` a ta volá ihned metodu `checkResults`, ve které se volá metoda třídy `Analyser` a to metoda `getClassGraph`. Zde se zjistí, že zatím žádný graf vytvořen není a tak se začne vytvářet a to v metodě `createClassGraph`.

6.4.4 Analýza bajtkódu

V metodě `createClassGraph` se volá metoda `readClassFilesAndCreateGraph` ze třídy `Parser`. Zde se teprve zkusí otevřít vstupní data. Vstupem může být více typů soubor – třídy, .zip, .jar soubor, složky obsahující výše uvedené typy souborů apod. Vždy se však musí jednat o soubory napsané v jazyce Java. Analýza totiž probíhá na bajtkódem. Ve třídě `Parser` se postupně prochází všechny třídy aplikace. Každá třída znamená interně jeden uzel, který odpovídá pojmu uzel v teorii grafů. Pro každý tento uzel se volá metoda `extractConstantPool` ze třídy `Constant`. Zde se prochází celý bajtkód. Aby byla jistota, že zkoumaný soubor je opravdu v bajtkódu, hledá se na počátku souboru tzv. MAGIC konstanta. Slovo magic konstanta může být přeloženo do českého jazyka jako kouzelná, magická či zázračná konstanta. Každý typ souboru začíná jinou magickou konstantou a díky tomu je možné rozlišit, o jaký typ souboru se jedná. Níže je tabulka magických konstant některých často používaných typů souborů 6.1. Hned na prvním řádku tabulky je vidět, že Java bajtkód začíná konstantou *CA FE BA BE*.

Popis souboru	**Přípona souboru**	**Hex vyjádření**
Java bytecode file	CLASS	CA FE BA BE
Word 2.0 file	DOC	DB A5 2D 00
Adobe Portable Document Format	PDF	25 50 44 46
Portable Network Graphics file	PNG	89 50 4E 47 0D 0A 1A 0A
Rich text format word processing file	RTF	7B 5C 72 74 66 31
Java archive	JAR	50 4B 03 04

Tabulka 6.1: Tabulka magických kostant některých často používaných typů souborů. Zdroj [5]

Poté, co se zkontroluje, že zkoumaný vstup opravdu obsahuje bajtkód, vstup se čte po jednotlivých bajtech vstup. Bajtkód má svůj formát, vidět je na obrázku 6.5.

```
ClassFile {
            u4   magic;
            u2   minor_version;
            u2   major_version;
            u2   constant_pool_count;
     cp_info     constant_pool[constant_pool_count-1];
            u2   access_flags;
            u2   this_class;
            u2   super_class;
            u2   interfaces_count;
            u2   interfaces[interfaces_count];
            u2   fields_count;
   field_info    fields[fields_count];
            u2   methods_count;
 method_info     methods[methods_count];
            u2   attributes_count;
attribute_info   attributes[attributes_count];
}
```

Obrázek 6.5: Formát java bajtkódu. Zdroj [4]

Na počátku je již zmíněna magická konstanta. Následuje verze a poté pole konstant *(Constant Pool)*. Právě toto pole konstant se v aplikaci Classycle zkoumá a díky tomu se určují závislosti mezi jednotlivými třídami. Pole konstant může obsahovat následující typy konstant 6.6.

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

Obrázek 6.6: Typy konstant v poli konstant. Zdroj [4]

Pro analýzu jsou důležité pouze kurzívou vyznačené konstanty, tedy *CONSTANT_Class*, *CONSTANT_Utf8*, *CONSTANT_String*. Jakmile se v bajtkódu narazí na jednu z těchto konstant, načte se jejich struktura a vytvoří se instance odpovídající třídy. Tento nově vytvořený objekt se vloží do pole `pool`, se kterým se dále v kódu pracuje.

6.4.5 Tvorba grafu

Výstup z analýzy bajtkódu se využije pro vytvoření interní reprezentace ve formě grafu. Pro každou třídu, balíček, se vytvoří struktura reprezentující jeden vrchol grafu. Každý vrchol obsahuje zejména jméno třídy/balíčku a typ. Dále obsahuje seznam vrcholů, které jsou užívány daným vrcholem a též které využívají daný vrchol.

Pro lepší představu struktury je níže vidět příklad reprezentace třídy `ClassA`, která je používána třídou `ClassB` a zároveň využívá externí třídu `java.lang.Object` 6.7.

```
vertex: class A sources: /Users/Documents/Application.jar: 1 incoming
arc(s) 1 outgoing arc(s))
  ▷ _graphVertex = true
  ▷ _head
      ▷ [0]: unknown external class java.lang.Object: 1 incoming arc(s) 0
        outgoing arc(s)
          ▷ _graphVertex = false
          ▷ _heads
          ▷ _tails
          ▷ _atributes
              ▷ _type = unknown external class
              ▷ _name = java.lang.Object
  ▷ _tails
      ▷ [0]: class B sources: /Users/Documents/Application.jar: 0 incoming
        arc(s) 1 outgoing arc(s)
          ▷ _graphVertex = true
          ▷ _heads
          ▷ _tails
          ▷ _atributes
              ▷ _type = class
              ▷ _name = classB
  ▷ _atributes
      ▷ _type = class
      ▷ _name = classA
```

Obrázek 6.7: Příklad reprezentující strukturu jednoho vrcholu – třídy A

Tento příklad je záměrně zkrácen a neobsahuje veškeré parametry.

6.4.6 Kontrola pravidel

Pravidla jsou kontrolována pouze pokud je aplikace spuštěna v módu kontroly pravidel. Každé pravidlo je nejdříve zkontrolováno, zda má odpovídající formát.

Pravidla se můžou aplikaci předložit buď odkazem na soubor s pravidly nebo se mohou pravidla vepsat přímo do parametru. Dále v textu budu pracovat pouze s možností definovat pravidla v souboru a parametrem zadat aplikaci Classycle odkaz na tento soubor. Pro druhou možnost definování pravidel platí stejné požadavky a stejný formát a stejný výstup jako pro možnost definovat pravidla v souboru.

Soubor s pravidly má přísně daný formát. Pro jednodušší definici pravidel je možné využít následujících definic:

Preference zobrazení definuje, co vše se má zobrazit `show <preference>`. Preferencí zobrazení je více typů. `onlyShortestPaths` zpracuje pouze nejkratší cesty v případě zobrazení nepovolených závislostí, `allPaths` zobrazí všechny cesty nepovolených závislostí, `onlyFailures` zobrazí pouze porušení pravidel, `allResults` zobrazí všechny výsledky včetně potvrzení splnění pravidel.

Vytváření proměnných formou `<property name> = <any string>` a jejich následné použití `$<property name>`.

Tvorba množin pomocí `[<set name>] = <term>0..* excluding <term>1..*` a jejich následné použití `[<set name>]`.

Je možné i **definování vrstev** a následně kontrola vrstev. Touto problematikou se však tato práce nezaobírá, protože tato funkcionalita není ve firmě nijak využívána a nebylo nutné výstup této formy jakkoliv modifikovat.

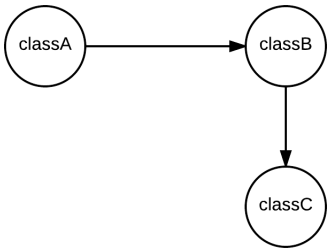
Pravidla samotná je pak možné definovat následovně:

Kontrola množin, zda nejsou prázdné, tedy zda obsahují alespoň jednu třídu kontroluje příkaz `check sets <set>1..*`.

Kontrola cyklů, tedy hledání silných komponent, je možné aplikovat na třídy `check absenceOfClassCycles > <maximum size> in <set>` nebo na balíčky `check absenceOfPackageCycles > <maximum size> in <set>`. Maximální velikostí je míněn maximální počet silně souvislých komponent mezi třídami/balíčky v dané množině.

Kontrola závislosti množiny na množině, tedy kontrola závislosti třídy či skupině tříd na jiné třídě či skupině tříd. Druhů závislosti je více typů. Prvním pravidlem je `check <set>1..* dependentOnlyOn <set>1..*`, které říká, že daná množina může být závislá jen a pouze na jiné množině. Druhým pravidlem je `check <set>1..* directlyIndependentOf <set>1..*`, pomocí kterého se definuje podmínka, že množina nesmí být přímo závislá na jiné množině. Posledním pravidlem je `check <set>1..* independentOf <set>1..*`, který zakazuje i nepřímou závislost. Pro lepší pochopení pravidel závislosti poslouží následující příklad.

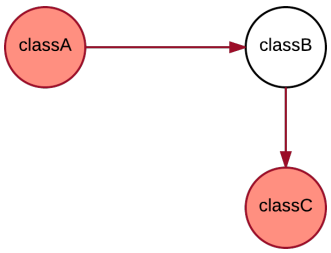
Mějme obrázek zobrazující tři třídy, `ClassA`, `ClassB` a `ClassC`. Třída `ClassA` využívá `ClassB` a ta následně využívá `ClassC` 6.8.



Obrázek 6.8: Příklad závislostí mezi třídami `ClassA`, `ClassB` a `ClassC`

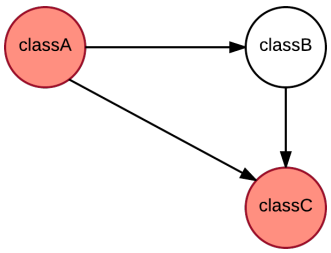
Mějme pravidlo `check ClassA directlyIndependentOf ClassC`. Toto pravidlo je neporušeno.

Mějme pravidlo `check ClassA independentOf ClassC`. Toto pravidlo je porušeno, porušení je zobrazeno na obrázku 6.9, červené uzly a hrany jsou elementy porušující pravidlo.



Obrázek 6.9: Demonstrace porušení pravidla `check ClassA independentOf ClassC`

Je vytvořena další vazba mezi třídou `ClassA` a `ClassC`. Pravidlo `check ClassA dependentOnlyOn ClassB` je pak porušeno 6.10.



Obrázek 6.10: Demonstrace porušení pravidla `check ClassA dependentOnlyOn ClassB`

Příklad souboru s pravidly:

```
show allResults

{package} = classycle

[dependency] = ${package}.dependency.*
```

```
[non-dependency] = ${package}.* excluding [dependency]

check sets [dependency] [non-dependency]

check [dependency] independentOf [non-dependency]
check ${package}.classfile.* independentOf java.util.* java.awt.* \
   javax.swing.*

check absenceOfClassCycles > 1 in [util]
```

Tvorbou pravidel se zabývá sekce Využití aplikace 7.3.

6.4.7 Výpis

Jakmile je vše analyzováno, aplikace vypíše výsledky. Způsob výpisu záleží na parametrech – je možné definovat jak množství informací, tak typ výstupního souboru. Výpisem se blíže zabývá sekce popisující původní výstupy 6.2 a nově vytvořené formy výstupů jsou popsány v 7.2.

Kapitola 7

Implementace

V této kapitole jsou popsány změny, které jsem na aplikaci provedla. Věnovala jsem se dvěma věcem, refaktorizaci aplikace a vytvoření nové formy výstupů. Nová forma výstupů byla potřebnější a proto je jí věnována větší část této kapitoly. V rámci tvorby nových výstupů je detailněji rozvedeno zobrazení silných komponent a porušení pravidel. Ke každému bodu je kromě nového výstupu uveden i způsob využití těchto vylepšení.

7.1 Refaktorizace

V sekci Optimalizace kódu 2.3 je obsažena teorie, která vysvětluje pojem refaktorizace a taktéž vysvětluje důvody, proč se refaktorizace provádí.

Kromě úpravy výstupu aplikace Classycle jsem provedla také její refaktorizaci. Důvodů bylo více:

- Některé třídy se vůbec nevyužívaly.
- Kód neměl sebevysvětlující třídy, metody a proměnné.
- Třídy a metody měly více druhů zodpovědnosti, prováděly spolu nesouvisející úkony.
- Metody měly často vedlejší efekt (například metoda `readClassFiles`) kromě čtení vstupu zároveň vytvářela graf).

Aby mohla být aplikace Classycle považována za aplikaci s čistým kódem, je potřeba provést další refaktorizaci. Aplikace je rozsáhlá a čas strávený na refaktorizaci se odvíjí od více parametrů. Prvním je rozsah refaktorizace, druhým jsou zkušenosti a schopnosti programátora. Čím větší rozsah refaktorizace a menší zkušenosti a schopnosti programátora, tím je doba strávená na refaktorizace delší. Refaktorizací aplikace Classycle se dá strávit ještě velké množství času, které by vydalo na další diplomovou práci.

Já jsem provedla několik úprav, které odstranily největší problémy. Poučení, které jsem si z refaktorizace aplikace odnesla je zejména to, že refaktorizaci má provádět zkušenný programátor, který již sám napsal tisíce a tisíce řádků kódu v daném jazyce. Ne nadarmo mají refaktorizaci na starost ti nejlepší programátoři v týmu. Já zatím tolik zkušeností nemám a to byl také důvod, proč provedených změn není v kódu tolik, aby mohl být prohlášen za čistý kód.

K provedení hlubší refaktorizace je potřeba vytvořit také více testů, konkrétně více jednotkových testů 3. Testů je v současné době 16 a zdaleka nepokrývají veškerou funkčnost

projektu. V případě pokračování refaktorování této aplikace musí být tvorba testů jedním z prvních kroků.

První úpravou bylo odstranění tříd, které se nevyužívaly. Při analýze bajtkódu se pro každou konstantu z pole `ConstPool` vytvářela instance příslušné třídy. Celkem tedy existovalo 15 tříd pro každý typ konstanty. Ve výsledku se však pro analýzu závislostí používaly pouze konstanty `CONST_Class`, `CONST_String` a `CONST_Utf8`, viz metoda `createNode` ve třídě `Parser`. Ostatní konstanty nebyly pro analýzu podstatné. Proto bylo zbytečné vytvářet instance tříd těchto konstant. Celkem bylo odstraněno 12 tříd, nechala jsem pouze tři třídy pro reprezentaci `CONST_Class`, `CONST_String` a `CONST_Utf8`. Dále byla odstraněna třída `ConstantPoolPrinter` a s ní i jednu metodu `main` a to z toho důvodu, že se o této třídě `main` nikde v dokumentaci nezmiňovalo, nenašla jsem nikde její použití. V rámci této metody se mělo pouze vypsat pole `ConstPool`, které je v bajtkódu obsaženo.

V rámci třídy `Constant`, ve které se analyzuje bajtkód, jsem odstranila na tvrdo napsané (*hard-coded*) konstanty, které reprezentovaly konstanty z pole `ConstPool` a nahradila jsem je konstantami z knihovny `javassist`. Tato knihovna je zaměřená na analýzu bajtkódu a je proto vhodnější její užití, než vytvářet konstanty vlastní. Knihovna `javassist` se neustále vyvíjí, aktuální verze, která obsahuje všechny konstanty z pole `ConstPool`, je verze 3.22. Já jsem však importovala verzi 3.18.1., která neobsahuje dvě novější konstanty. Vyšší verze však při překladu způsobuje známou, již popsanou chybu `MethodParameters attribute introduced in version 52.0 class files is ignored in version 50.0 class files`. Díky této chybě nelze aplikaci sestavit, protože v rámci silnější kontroly se aplikace sestavuje s parametrem `-Werror`. Po odstranění tohoto parametru při překladu lze importovat i novější verzi knihovny `javassist`. Já však parametr `-Werror` ponechala, protože používání tohoto parametru je důležitější, než užití novější knihovny. Jakmile bude tato známá chyba z knihovny odstraněna, bude se moci použít a dvě zbývající na tvrdo zadané konstanty se budou moci z kódu odstranit.

V rámci refaktorizace jsem vytvořila pouze jeden společný `main`. Důvodem je zvýšená přehlednost projektu a tím zvýšení kvality softwaru. Druhým důvodem je skutečnost, že metoda `main` ze třídy `DependencyChecker` a metoda `main` ze třídy `Analyser` měly společnou část kódu (což je také jeden z důvodů k refaktorování kódu). První jmenovaná metoda dokonce využívala třídu `Analyser` a musela si tedy vytvořit její instanci, což působilo zmatečně vzhledem k tomu, že ve třídě `Analyser` existovala jiná metoda `main`. Přidáním výstupu se navíc další dva parametry, `-packagesOnly` a `-yedFile`, staly pro obě metody společné a tak nebyl důvod je nadále nechat rozdělené, nadále nechávat duplikovaný kód.

Proběhlo přejmenování několika tříd (např. `readClassFiles` na `readClassFilesAndCreateGraph`), ujednotilo se odřádkování.

7.1.1 Parametry aplikace: nový stav

Prvním rozdílem při použití aplikace je to, že neexistují tři metody `main`, ale pouze jedna. Původní metody byly sloučené do jedné z více důvodů. Prvním důvodem je přehlednost. Z vlastní zkušenosti vím, že není obvyklé hledat metodu `main` schovanou v třídě, která není v balíčku hierarchicky nejvýše a že je ve třídě, která svým názvem vůbec nepřipomíná vstup do programu. Druhým důvodem je to, že metoda `main` ze třídy `DependencyChecker` stejně využívala instanci třídy `Analyser`, v níž byl obsažený první main. Nová metoda `main` slučuje ty řádky kódu, které byly pro původní metody společné. Dle parametrů pak spouští metodu ze třídy `Analyser` nebo ze třídy `DependencyChecker`. Automaticky se však zkoumají všechna pravidla, jak kontrola cyklů v balíčcích, tak kontrola cyklů ve třídách je

udělána, nehledě na parametr `-packagesOnly`. Pokud je však zadán parametr `-yedFile` a výstup se požaduje i v grafové podobě, zobrazí se porušená pravidla pouze na jedné úrovni – na úrovni balíčků nebo na úrovni tříd, způsob zobrazení se reguluje pomocí parametru `-packagesOnly`.

Oproti původnímu stavu přibyly také dva nové parametry a to parametr `-yedFile` a `-gephiFile`. První parametr způsobí, že výstupem je soubor, který lze otevřít v aplikaci yEd, druhý lze otevřít v aplikaci Gephi. Obě dvě aplikace patří mezi grafové nástroje, každá má trochu jinou funkčnost a jiné zaměření. Obě dvě zobrazují výstup z aplikace v Classycle pomocí grafu. Tyto výstupy budou podrobněji rozepsány dále v textu.

7.2 Nové výstupy aplikace

Cílem práce bylo zejména vytvořit novou formu výstupu tak, aby to vyhovovalo požadavkům konzultanta z firmy. Nejdůležitějším požadavkem bylo vytvořit výstup takový, aby byla výstupní data reprezentována v grafické podobě. Jelikož aplikace používá grafovou reprezentaci závislostí, tak i nově vzniklý výstup měl být ve formě grafu. Pro samotnou reprezentaci bylo nutné vybrat vhodný již existující nástroj, který se pro zobrazení dat využije. Cílem práce nebylo vytvářet nástroj nový, protože nástrojů na zobrazení grafů je na trhu již mnoho a jsou plně postačující.

První věc, na které bylo nutné dohodnout se s konzultantem firmy, bylo domluvit si aplikaci, kterou na zobrazování dat použiji.

Nejprve byl vybrán nástroj **Gephi**. Nástroje jsou podrobněji popsány v kapitole Použité nástroje 5. Nástroj Gephi mi byl doporučen konzultantem z firmy. Je to velice silný nástroj, je vyvíjen jako volně šiřitelný *(open-source)* software. Poté, co jsem analyzovala funkci aplikace Classycle, jsem se začala věnovat nové formě výstupu, která bude zpracovatelná nástrojem Gephi. Během vývoje jsem však narazila na jeden dost zásadní problém. Gephi není podporovatelný na systému MacOS. Vzhledem k tomu, že jsem během vývoje přešla na tento systém, mě tento problém velice omezoval. Co více, podmínkou zvoleného nástroje mělo být, že je multiplatformní a zdarma. Podmínka multiplatformosti byla bohužel porušena. Výstup v Gephi není dotažený do konečné podoby, je však ponechán pro možné další rozšíření. Aby bylo možné nástroj Gephi pro můj účel využít, bylo by ho nejprve nutné upravit. Jelikož je nástroj volně šiřitelný, jak již bylo výše zmíněno, úprava by možná byla. Po dohodě s konzultantem i s vedoucím diplomové práce jsem se úpravě aplikace Gephi nevěnovala, protože bych se tím odklonila od řešení původního zadání. Protože nástroj Gephi nevyhovoval mým požadavkům, bylo nutné vybrat nástroj jiný a to nástroj yEd. Tento nástroj netrpí problémy s kompatibilitou, navíc je i v online verzi.

K vytvoření nových výstupů jsem musela vytvořit nové třídy a nové metody. Jelikož jsem vytvořila tříd více, vytvořila jsem také nový balíček jménem `outputPrinter`. V tomto balíčku je celkem 9 nově vzniklých tříd. Třídy se dají rozdělit na skupinu tříd popisující stavební elementy grafu, pomocné třídy a třídy vytvářející graf. Dále jsem modifikovala již existující třídy analyzující vstupní parametry, přidala jsem parametry pro volbu nástroje a taktéž proběhla úprava dovolující u volby kontroly závislostí vytvořit výstup do nástroje yEd.

Do skupiny tříd popisující **stavební elementy grafu** patří třídy `Edge`, `Node`, `Package` popisující hranu, třídu a balíček. Třída popisující balíček se nejmenuje *Class*, protože takového klíčové slovo v jazyce Java již existuje. Třída `Edge` je abstraktní třída, ze které dědí třídy pro hranu mezi třídami a pro hranu mezi balíčky. Při původním návrhu aplikace jsem chtěla vytvořit jednu třídu, která by sdružovala popis pro třídu a popis pro balíček. Ač mají

tyto dvě věci spoustu věcí společných, každá z nich má jinak implementované porovnávání objektů. Nenašla jsem lepší řešení, než podobné třídy s částečně duplikujícím se kódem přesto vytvořit odděleně. Na rozhraní skupiny stavebních elementů grafu
a pomocných tříd leží třída `StronglyConnectedNodes`. Tato třída reprezentuje jednu silnou komponentu, tedy skupinu uzlů cyklicky propojených. Do skupiny tříd popisující **pomocné třídy** patří `PrintYed` a `PrintGephi`, které dědí ze třídy `ParseGraph`, jejich funkcionalita spočívá ve volání metod postupně tvořící graf a v následném vytištění dat ve vhodném formátu. Dále se zde vyskytuje výčet `NodeColor`, který sdružuje barvy. Důvodem nepoužití barev, které nativně nabízí jazyk Java je to, že barev takto nabízených je málo, navíc se mi vizuálně tolik nelíbily. Nejdůležitější skupinou je skupina tříd **vytvářející graf**. Tato skupina obsahuje pouze jednu třídu, třídu `ParseGraph`. V ní jsou schované metody na načtení všech uzlů a balíčků z interní reprezentace aplikace a vytvoření instancí příslušných tříd, vytvoření hran mezi těmito uzly, nastavení pozic v grafu jednotlivým elementům
a s tím související tvorba stromové struktury, obarvení hran a uzlů, seskupení uzlů do silně souvislých komponent a vytvoření tvarů rámující tyto skupiny, obarvení hran a uzlů podle porušení pravidel. Třída `ParseGraph` patří k algoritmicky nejsložitějším nově vytvořeným třídám.

Některé důležité metody z třídy `ParseGraph`:

- `getAllPackages` vytvoří ke všem balíčkům v grafu instanci třídy `Package`. V interní reprezentaci grafu nejsou zaznamenány balíčky nebsahující žádnou třídu. Tyto balíčky jsou v této metodě též tvořeny. Dále je zde volána metoda pro vytvoření stromové struktury celé analyzované aplikace.

- `getAllNodes` vytvoří ke všem třídám v grafu instanci třídy `Node`, nastaví atribut odkazující na balíček, v němž se nachází.

- Vrcholy musí být nějak propojeny a k tomu slouží metoda `createEdges`, která mezi nimi vytvoří příslušné hrany.

- `getAllStronglyConnectedComponents` vytvoří ke všem silným komponentám v grafu instanci třídy `StronglyConnectedNodes`.

- `setPositions` přiřadí každému uzlu v grafu jeho pozici. Uzly jedné silně souvislé komponenty jsou uskupeny k sobě. Výsledný graf je buď na úrovni balíčků – zobrazeny jsou v něm pouze balíčky, a nebo je na úrovni tříd – zobrazeny jsou pouze třídy. Pokud si uživatel vygeneruje grafy oba, pozice balíčků a tříd jsou nastaveny tak, aby balíčky byly vizuálně na počátku skupiny tříd příslušného balíčku a naopak.

- `createStrongComponentSurrounding` tvoří kolem každé skupiny uzlů v jedné silně souvislé komponentě útvar tak, aby bylo z grafu lépe poznat jednotlivé silně souvislé komponenty. Protože nástroj yEd neumožňuje vytvářet jednoduše uživatelem definované tvary, okolí silně souvislých komponent je tvořeno tak, že se seskládá z mnoha vizuálně malých uzlů, jenž jsou vzájemně propojeny. Tvorba těchto pomocných uzlů probíhá ve třídě `createNewSurroundingNodes`.

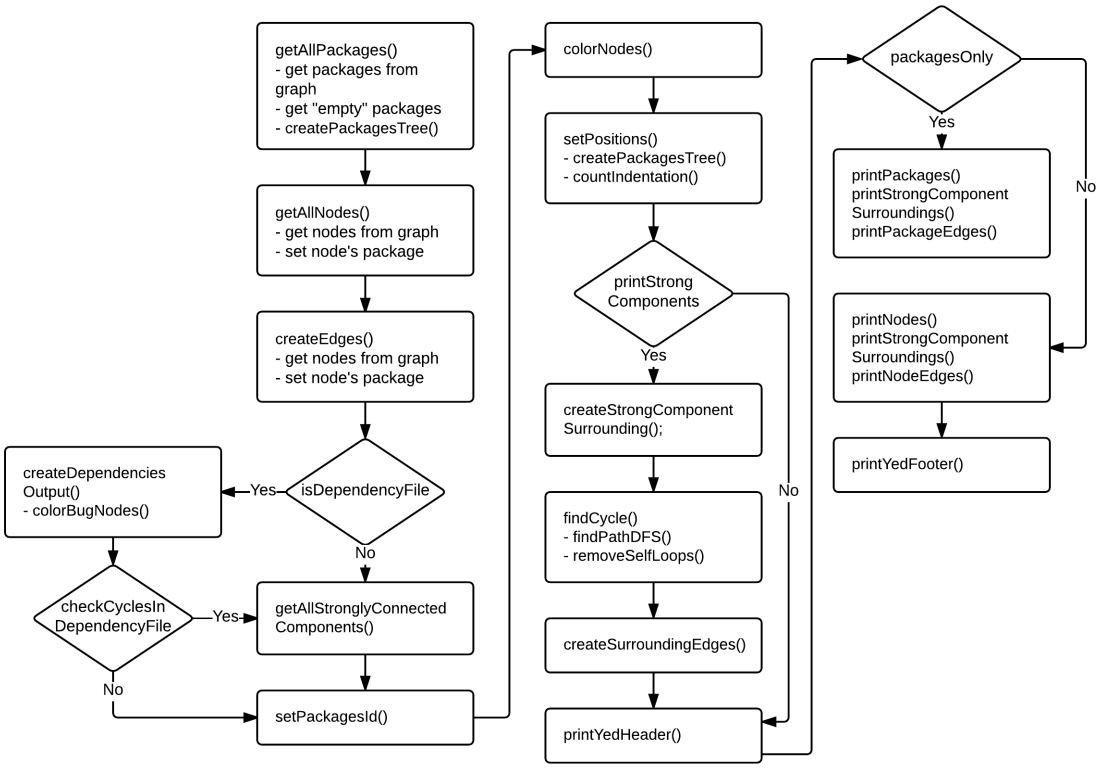
- Jak uzly, tak okolí silně souvislých komponent je třeba obarvit. K tomu slouží metoda `colorNodes`, která obarvuje uzly v jednotlivých silně souvislých komponentách. Obarvuje však pouze ty uzly, které jsou v silně souvislé komponentě o velikosti 2
a více. Ostatní uzly zůstávají černé. Z definice silně souvislé komponenty je totiž každý

uzel v silně souvislé komponentě, která může obsahovat pouze a právě jen tento uzel. Balíčky neobsahující žádnou třídu se obarví na bílo, aby bylo na první pohled rozlišitelné, že jsou prázdné. Ukázka je níže na obrázcích výstupu. Barvy jsou zvoleny tak, aby od sebe byly snadno odlišitelné a aby se zároveň nejednalo pouze o červenou, zelenou a modrou. Použité barvy odpovídají primárním a sekundárním barvám [3] – žlutá, oranžová, červená, fialová, modrá a zelená.

- `createDependenciesOutput` prochází pravidla ze souboru s pravidly, která jsou v aplikaci již zpracována a interně uložena. Pokud obsahuje pravidlo požadavek závislost či nezávislost mezi třídami, vyhledají se příslušné vrcholy a hrany porušující pravidlo a tyto vrcholy a hrany se červeně zvýrazní. Pokud třída závisí na externí třídě a tím porušuje pravidlo, obarví se obrys uzlu reprezentující tuto třídu červeně a vyplní se bílou barvou. Důvod je ten, že externí třídy nejsou pro přehlednost v grafu zobrazovány. Bílá barva naznačuje, že uzel ukazuje někam mimo samotnou aplikaci, závisí na něčem externím. V případě kontroly cyklů, tedy vyhledávání silných komponent, je situace složitější. První problém plyne z toho, že graf zobrazuje pouze balíčky a nebo pouze třídy, zobrazení jak balíčků, tak tříd, není možné. Důvodem byla zhoršená přehlednost grafu. Pokud tedy pravidla obsahují požadavek na kontrolu cyklů, je podstatné, zda byla aplikace spuštěna s parametrem `-packagesOnly` nebo nikoliv. Pokud byl zadán parametr a pravidla obsahují požadavek na kontrolu cyklických závislostí v rámci tříd, kontrola sice proběhne, na standartní výstup bude vypsána informace
  o případném porušení, v grafu se ale případné porušení nezobrazí. Stejný výsledek bude mít i opačná situace. Pokud není aplikace spuštěna s parametrem `-packagesOnly`, graf obsahuje pouze třídy. Pokud je v pravidlech požadavek na kontrolu cyklů v rámci balíčků, na standartní výstup se případné porušení vypíše, v grafu se však nezobrazí. Tato vlastnost může být námětem na další vylepšování aplikace. Interní reprezentace silně souvislé komponenty sice obsahovala informace o uzlech, které jsou v dané komponentě. Nebyla však uložena nikde posloupnost hran tvořící cyklus, díky němuž byla skupina uzlů za silně souvislou komponentu prohlášena. Silná komponenta je totiž vytvářena pomocí Tarjanova algoritmu a ten si tuto informaci neuchovává. Abych mohla tento cyklus vytvořit, bylo nutné vytvořit algoritmus, který cyklus detekuje. Cyklus se detekuje ve metodě `findCycle`. Princip je podrobněji popsán v kapitole 4.

Třída dále obsahuje pomocné metody, jako je `getNodeByLabel`, `colorNodesEdges`, `findPathDFS`, `colorBugNodes`, `createPackagesTree` a další.

Pro lepší názornost je na obrázku 7.1 vidět posloupnost volání metod s pomocnými metodami.

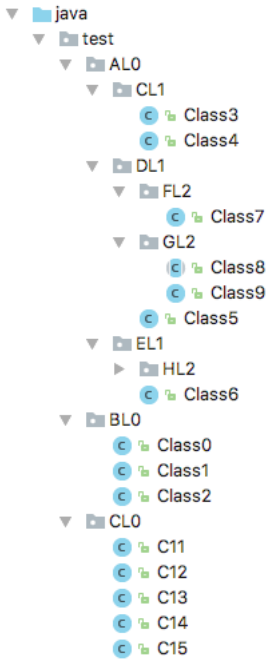


Obrázek 7.1: Posloupnost tvorby výstupu

7.2.1 Struktura výstupu

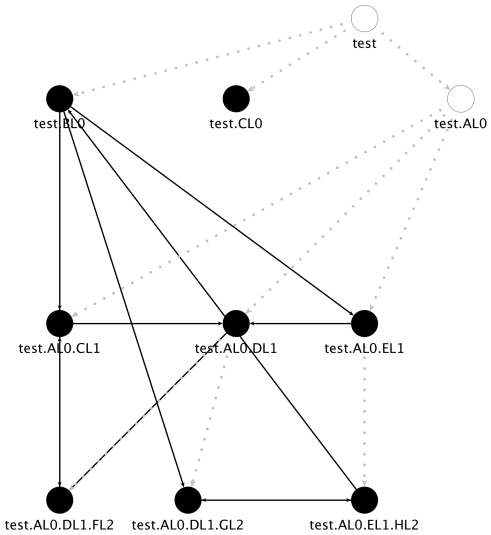
Výstup v grafické podobě jsem původně tvořila pro nástroj Gephi a Classycle dokáže vytvořit výstup i do tohoto nástroje. Brzy jsem z důvodů, jež jsou uvedeny výše, změnila výstup na výstup zobrazovaný v nástroji yEd. Zobrazit v nástroji Gephi lze jen zlomek toho, co v nástroji yEd, protože práce na tomto výstupu byla přerušena. Obrázky níže jsou všechny výstupy nástroje yEd.

Pro účely testování a lepší názornosti jsem si vytvořila aplikaci skládající se z několika balíčků a tříd. Aplikace nemá žádnou speciální funkcionalitu, pouze tiskne pár předem daných konstant na výstup. Lze na nich však demonstrovat druhy výstupu. Struktura aplikace níže zobrazených příkladů je vidět na obrázku 7.2.



Obrázek 7.2: Struktura testovací aplikace, exportováno z aplikace yEd

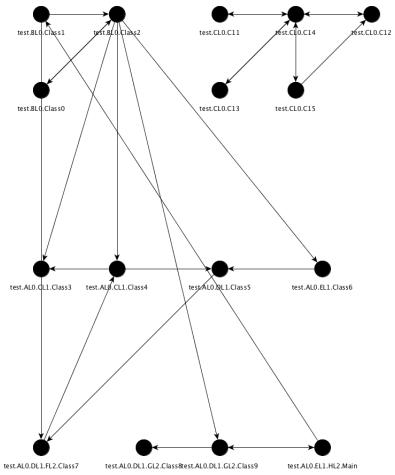
Na obrázku 7.3 lze vidět strukturu výše uvedené aplikace na úrovni balíčků. Černé uzly reprezentují jednotlivé balíčky. Bíle zobrazené balíčky neobsahující třídu, tedy jsou buď zcela prázdné nebo obsahují pouze další balíčky. Černé plné hrany reprezentují vazby mezi balíčky. Šedé tečkované hrany znázorňují hierarchii balíčků – například balíček `test` obsahuje balíčky `AL0` [1], `BL0` a `CL0`. Jednotlivými úrovněmi je naznačeno zanoření balíčků. Na nejvyšší úrovni je balíček `test`, balíček `FL2` je na nejnižší úrovni. Balíčky jsou popsány celou svou cestou, zmíněný balíček `FL2` je pojmenován `test.AL0.DL1.FL2`.



Obrázek 7.3: Způsob zobrazení na úrovni balíčků, exportováno z aplikace yEd

[1] Pro potřeby diplomové práce jsou obrázky zmenšeny a pojmenování může ve vytištěné podobě být hůře čitelné. V rámci elektronické verze se dají obrázky přibližovat a oddalovat, což čitelnost zvyšuje.

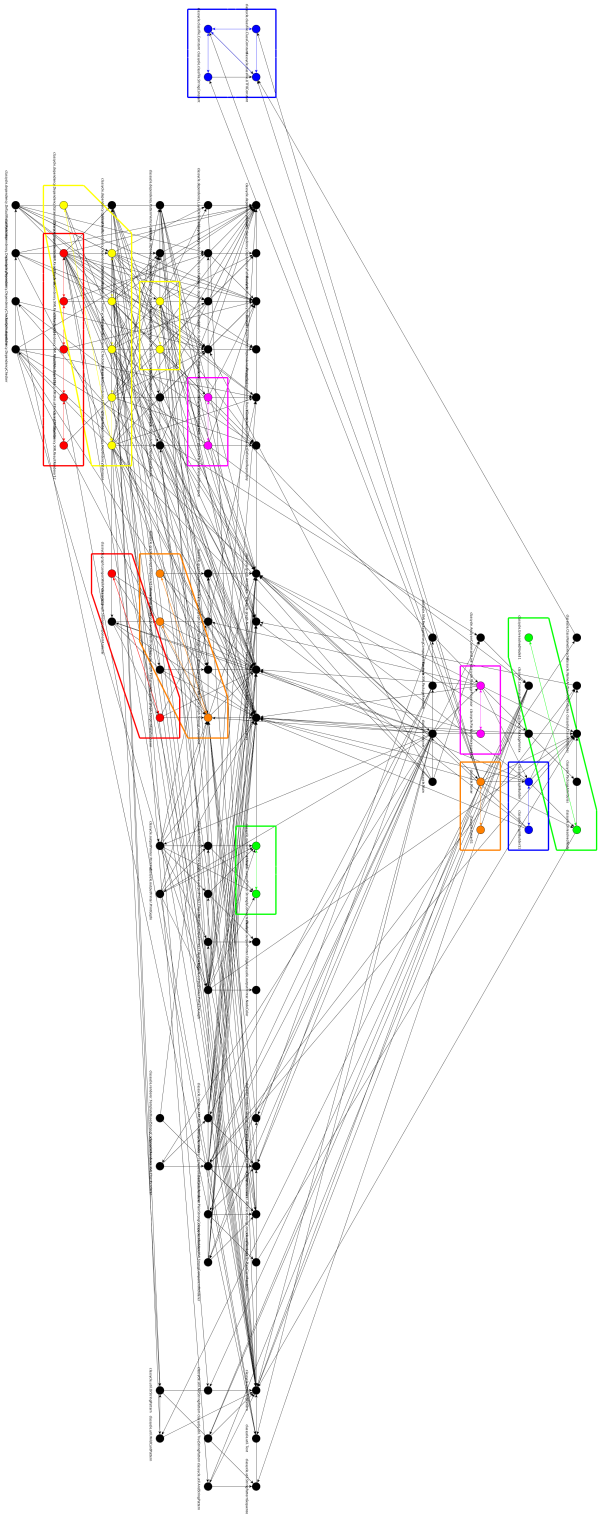
Na obrázku 7.3 lze vidět struktura výše uvedené aplikace na úrovni tříd. Je možné si povšimnout, že chybí nejvyšší úroveň, jelikož balíček `test` neobsahoval žádné třídy. Třídy se seskupují podle jednotlivých balíčků. Názorné je to u tříd patřící o balíčku `CL0`, které jsou všechny seskupeny pohromadě. Jméno tříd je opět dáno celou cestou, například třída `C11` je v grafu vyznačena jako `test.CL0.C11`. Je zachována stromová struktura, hierarchie však již není explicitně zvýrazněna. Hrany jsou pouze černé, stejně jako uzly reprezentující třídu.



Obrázek 7.4: Způsob zobrazení na úrovni tříd, exportováno z aplikace yEd

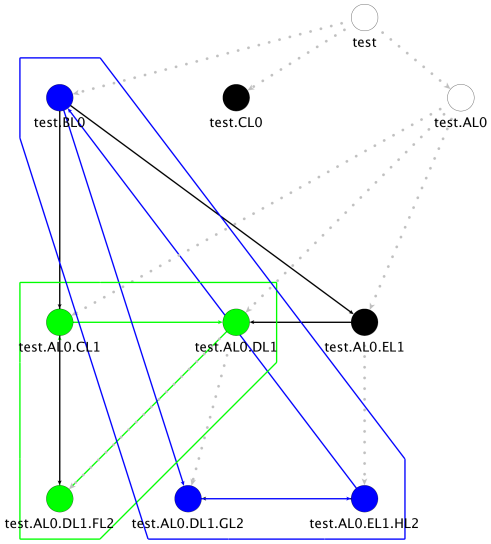
7.2.2 Zobrazení silných komponent

Classycle vytváří dva hlavní výstupy. Prvním je zobrazení silných komponent, druhým je zobrazení porušených pravidel. Na obrázku 7.5 lze vidět analýzu samotné aplikace Classycle. Lze vidět, že silných komponent je více, jsou však vždy v rámci jednoho balíčku. Skupiny uzlů patřící do jedné silné komponenty mají jednotnou barvu a stejnou barvou jsou i ohraničeny. Obrázek je velký a je proto natočen. Pro demonstraci dalších možností výstupu je použita opět testovací aplikace, jejíž analýza nezabírá v grafové podobě tolik místa.



Obrázek 7.5: Způsob zobrazení silně souvislých komponent na úrovni tříd, exportováno z aplikace yEd

Zobrazení silných komponent může být zobrazeno také na úrovni balíčků. Na obrázku 7.6 je ukázka analýzy testovací aplikace, zobrazují se silné komponenty na úrovni balíčků.

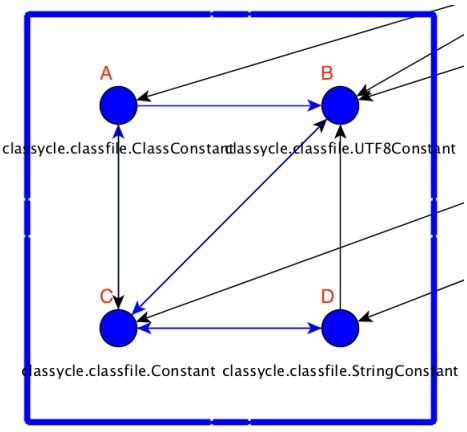


Obrázek 7.6: Způsob zobrazení silně souvislých komponent na úrovni balíčků, exportováno z aplikace yEd

Silné komponenty se zkoumají pouze na úrovni tříd nebo balíčků. Nižší granularita není možná, aplikace to neumí, nestačila by pouze analýza bajtkódu, musel by se zkoumat nepřeložený zdrojový kód. Granularita na vyšší úrovni není možná, protože Java aplikace neobsahuje žádný další pojmenovaný útvar, který by byl hierarchicky výše, než balíček.

Vrcholy v silně souvislé komponentě jsou zvýrazněny více způsoby.

- Vrcholy v silně souvislé komponentě jsou obarveny stejnou barvou.
- Celá silně souvislá komponenta je ohraničena stejnou barvou, jakou mají vrcholy.
- Stejnou barvou je zobrazen cyklus, díky němuž jsou vrcholy v jedné silné komponentě. Příklad je vidět na výřezu 7.7 z výše uvedeného obrázku. Vrcholy jsou nazvány A, B, C a D z důvodu lepší čitelnosti. Z každého vrcholu musí existovat možnost dostat se do libovolného jiného uzlu v rámci silně souvislé komponenty. Například z vrcholu A se do vrcholu D je cesta přes vrcholy B a C, z D do A vede hrana přes vrchol C. Stejný obrázek je i v kapitole Grafy 4, kde je podrobněji rozepsán.



Obrázek 7.7: Způsob zobrazení silně souvislých komponent na úrovni balíčků, exportováno z aplikace yEd

7.2.3 Zobrazení porušených pravidel

Druhý výstup aplikace Classycle se zabývá kontrolou pravidel ze vstupu. Výstup je tisknut vždy na standartní výstup (stdout). Množství vytištěných informací je dáno klíčovými slovy v pravidlech 6.4.6. Dále v textu je uvažována situace, že se mají vypsat všechny výstupy (`allResults`). Pokud je v parametrech aplikace zadán také výstup do formátu zobrazitelného v programu yEd, vytvoří se kromě výpisu na standartní výstup i ten. V grafu jsou zobrazeny pouze porušená pravidla, splněná pravidla nijak vyznačeny nejsou. Vycházela jsem z předpokladu, že při užívání aplikace zajímají uživatele pouze chyby, problémy, protože jen na ty je potřeba se zaměřit.

Graf může zobrazovat třídy nebo balíčky, opět nelze zobrazovat obě dvě úrovně granularity. Z této podmínky plyne, že porušené pravidlo maximálního počtu cyklů mezi balíčky může být zobrazeno pouze pokud je graf na úrovni balíčků. Stejně tak porušené pravidlo maximálního počtu cyklů mezi třídami může být zobrazeno pouze na úrovni tříd. Úrovně závisí na parametru `-packagesOnly`. Je-li zadán a je-li zároveň zadán požadavek na kontrolu cyklů, jejich porušení bude pouze vypsáno na standartní výstup a nebude v grafu zobrazeno. Na tuto skutečnost je uživatel upozorněn výpisem na standartní výstup. To stejné platí i pro opačný případ. Cyklem jsou míněny silně souvislé komponenty. Pokud počet cyklů přesáhne maximální množství zadané pravidlem, zobrazí se všechny silně souvislé komponenty v dané oblasti. Silně souvislé komponenty jsou všechny červené, již se neodlišují více barvami. Důvodem je snaha sjednotit zobrazení chyb – všechny chyby jsou zobrazeny červeně.

Porušená pravidla závislosti lze zobrazit jak v grafu na úrovni balíčků, tak na úrovni tříd. Porušené pravidlo je zobrazeno tak, že vrcholy, kterých se pravidlo týká, jsou obarveny na červeno. Stejně tak hrana mezi nimi je červená. Pokud třída nebo balíček používá externí třídu, není možné červeně označit hranu porušující toto pravidlo – externí třídy totiž nejsou v grafu kvůli lepší čitelnost zobrazeny. Proto je tento typ pravidla zobrazen jinak. Vrchol, který je na externí třídě závislý, má červený okraj, ale je vybarven bíle. Tím je naznačena závislost na neexistující (přeneseně vzato prázdný) vrchol.

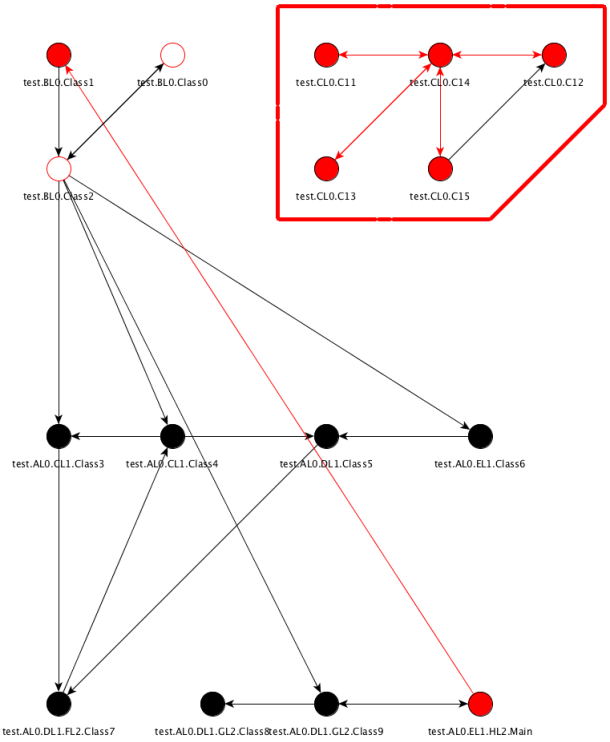
Následující obrázek 7.8 zobrazuje porušení pravidel zadaných v následujícím formátu:

```
show allResults

package = test
[classAclassB] = $package.AL0.* $package.BL0.*
[classC] = $package.* excluding [classAclassB]

check [classC] independentOf [classAclassB]
check [classAclassB] independentOf java.io.*
check test.AL0.* independentOf test.BL0.*

check absenceOfClassCycles > 1 in [classC]
```



Obrázek 7.8: Zobrazení porušených pravidel, exportováno z aplikace yEd

Na standartní výstup je vytištěna následující zpráva:

```
show onlyShortestPaths allResults
check [classC] independentOf [classAclassB] OK
check [classAclassB] independentOf java.io.*
    Unexpected dependencies found:
    test.BL0.Class0
        -> java.io.PrintStream
    test.BL0.Class2
        -> java.io.PrintStream
check test.AL0.* independentOf test.BL0.*
    Unexpected dependencies found:
    test.AL0.EL1.HL2.Main
        -> test.BL0.Class1
check absenceOfClassCycles > 1 in [classC]
    test.CL0.C14 et al. contains 5 classes:
        test.CL0.C11
        test.CL0.C14
        test.CL0.C12
        test.CL0.C13
        test.CL0.C15
```

Pravidla jsou vytvořena tak, aby demonstrovala všechny možné způsoby zobrazení porušení pravidel. Pravidla mají za úkol pouze demonstrovat způsob jejich zápisu a jejich následné zobrazení.

Na grafu lze vidět porušená pravidla.

První porušené pravidlo `check [classAclassB] independentOf java.io.*` je porušeno v `test.BL0.Class0` a `test.BL0.Class2`. Obě dvě třídy závisí na externí třídě. Proto jsou uzly bílé s červeným ohraničením.

Druhé porušené pravidlo `check test.AL0.* independentOf test.BL0.*` je porušeno tím, že v třída `test.AL0.EL1.HL2.Main` používá třídu `test.BL0.Class1`. Toto porušení je znázorněno tak, že oba dva vrcholy jsou červeně vybarvené a hrana mezi nimi je také červená. Pokud by třída `test.AL0.EL1.HL2.Main` závisela na `test.BL0.Class0`, jenž využívá externí třídu a je proto zvýrazněna bíle, hrana by mezi těmito dvěma třídami byla červená, třída `test.AL0.EL1.HL2.Main` by též zůstala červená, ale třída `test.BL0.Class0` by si zachovala bílou barvu na důkaz, že závisí na externí třídě. Zobrazení závislosti na externí třídě je má větší prioritu než zobrazení porušení pravidla. Důvodem je to, že jinak než barevně nelze v grafu naznačit závislost na externí třídě. Závislost mezi dvěma interními třídami však zobrazuje jak červená barva vrcholu, tak červená barva hrany.

Pravidlo `check absenceOfClassCycles > 1 in [classC]` je taktéž porušeno, protože balíček `test.CL0` obsahuje cyklus, tedy silně souvislou komponentu. Tato komponenta je vyznačena v grafu červeně a to jak zvýrazněním všech jejich uzlů, tak jejím orámováním.

7.3 Využití aplikace

V této sekci jsou popsány možnosti použití aplikace. Definuje se zde důvod použití, čas použití i způsob použití aplikace Classycle.

7.3.1 Závislosti, cyklické závislosti mezi třídami

Aplikace odhaluje cyklické závislosti ve formě silných komponent. Dále hledá porušená pravidla, která byla podle jedné z metodik popsaných níže sestrojena. Závislosti mohou vznikat jak mezi balíčky, tak mezi třídami. Příklady budou uváděny pro třídu, pro balíčky jsou principy analogické.

Problém závislostí obecně, nejen cyklických, tkví v tom, že při změně jedné třídy může být porušena funkčnost třídy druhé, na první třídě závislé. Změna v jedné třídě má dopad na všechny závislé třídy, ať už třídy závislé díky dědičnosti nebo kompozici. Závislost nemusí být na první pohled v kódu patrná, snižuje se tak míra čitelnosti kódu. Manipulace s třídami, které jsou nějakým způsobem závislé na jiných, vyžaduje vyšší náklady, než manipulace s na sobě nezávislými třídami.

Problém cyklických závislostí: více cyklů přináší více nákladů. Pokud je potřeba přesunout třídu, obvykle se musí přesunout celá silná komponenta skládající se z cyklicky na sobě závislých tříd. Cyklickým závislostem se v kódu často nedá vyhnout. Je však silně vhodné mít cyklickou závislost pouze v rámci jednoho balíčku. U tříd v různých balíčcích může přesunutí komponenty vést k nutnosti měnit třídy z důvodu přístupových práv k metodám a atributům.

Zvýšené náklady vznikají zejména z následujících důvodů.

- Cyklické závislosti způsobují horší čitelnost kódu a tím pomalejší vývoj.
- Jakákoliv úprava zasahuje více tříd, tím se opět prodlužuje doba práce a tím zvyšují náklady.
- Zvyšování nákladů při testování - při změně jedné ze závislých tříd je nutné otestovat nejen danou třídu, a spolu s ní všechny další na ni závislé třídy.

- Při nasazování a vydávání aktualizací se v praxi často exportuje pouze část aplikace, která byla zasažena změnou – v případě závislostí je zasaženo změnou více tříd a exportovat je nutné větší část kódu, což může opět zvýšit náklady.

Výstup z aplikace, který zobrazuje silně souvislé komponenty, pomáhá programátorovi při dvou skutečnostech.

- Při psaní nového projektu je možné opakovaně spouštět analýzu pomocí Classyclu a tím kontrolovat, že v průběhu vytváření nových komponent se nevytvořila silně souvislá komponenta mezi třídami. Pokud se vytvořila, autor projektu se může vrátit v programovaní o pár kroků zpět a vymyslet jiné řešení problému tak, aby cyklická závislost nevznikla a pokud vzniknout musí, aby vznikla pouze v rámci jednoho balíčku.

- Častější využití vidím při refaktorizaci projektu, aplikace. Je běžné, že se programátor dostane do styku s projektem, která nemá žádnou vnitřní strukturu, všechny třídy jsou obsaženy pouze v jednom či několika málo balíčcích a je tudíž těžké se v projektu orientovat. Classycle pomůže odhalit silné komponenty v projektu. Ty můžou být vodítkem k rozdělení tříd do balíčku – v extrému z každé silné komponenty může vzniknout jeden balíček. Programátorovi tudíž pomůže při prvních krocích refaktorizace, vytvoření základní struktury projektu.

7.3.2 Metodiky pro tvorbu pravidel

Pomocí pravidel se kontrolují závislosti v projektu. Nevznikají samovolně, vždy je pro jejich tvorbu nějaký důvod. Pravidla můžou být definována na základě více skutečností:

- Je nutné dodržet firemní směrnice.

- V rámci kolektivu je domluvena štábní kultura kódu pro zvýšení celkové čitelnosti kódu.

- Jednotlivec chce vytvořit další možnost kontroly uchování jím definované struktury svého kódu.

Ve všech případech je dodržování pravidel další kontrolou při vývoji, podobně jako testování. Pravidla kontrolují domluvené konvence a směrnice, funkčnost kontroluje spíše testování.

Tvorba pravidel vzniká na základě následujících situací.

Ucelené jednotky pro testování: V rámci firemních směrnic je definována velikost celků pro testování. Každý tester či testovací oddělení je zodpovědné za otestování jednoho či více ucelených celků. Pomocí pravidel lze zabránit tomu, aby na sobě byly jednotlivé celky závislé a díky tomu budou testeři testovat ucelenou část, která není závislá na vnějších vlivech. *Příklad:* Firma má dva testery. Vyvíjí aplikaci, jejíž třídy se nachází ve třech balíčcích – A, B a C. První tester má za úkol testovat funkcionalitu tříd v balíčku A a B. Druhý tester testuje třídy z balíčku C. Pravidlo pro tuto situaci by dovolovalo závislost tříd mezi balíčkem A a B, ale zakazovalo by závislost tříd z balíčků A a B na třídách z balíčku C.

Rozdělení práce. Každý programátor ve firmě má na starost jinou část projektu. V případě vzniku závislostí mezi částmi projektu vzniká nejednoznačná zodpovědnost za ty komponenty, které jsou z různých částí projektů na sobě závislé.

Dodržování struktury. Firemní směrnice může definovat strukturu kódu, tedy i přesnou hierarchii balíčků. Díky pravidlům je možné vynutit si dodržování této hierarchie. *Příklad:* aplikace firmy je rozsáhlá a skládá se z několika logicky oddělených částí – část přístupná všem uživatelům a část přístupná přihlášeným uživatelům, každá část má jinou funkčnost. Automatizované testy se taktéž rozdělují podle toho, co testují, tedy na testy pro veřejnou část a pro část dostupnou až po přihlášení. Testy musí být rozděleny do dvou základních podskupin, do dvou základních balíčků. Pravidlo bude vynucovat, aby třídy z jednoho logického celku, balíčku obsahující veřejnou část, neodkazovaly na druhý logický celek, druhý balíček obsahující funkčnost pro přihlášené uživatele.

Ucelené jednotky pro export kódu. Firemní směrnicí je definována maximální velikost aktualizace z důvodu nákladů. Aktualizace se obvykle vydává na úrovni balíčků. Pravidla zabrání tomu, aby na sobě nezávisely komponenty z balíčků, jejichž součet velikostí je větší než maximální velikost aktualizace.

Čistota kódu. Každá třída má název, který odpovídá její funkcionalitě. Balíček seskupuje třídy se stejným účelem. Pravidlo zabrání tomu, aby třída obsažena v balíčku s daným účelem nevyužívala třídy zabývající se jinou problematikou. *Příklad:* třídy tisknoucí něco na výstup jsou sdruženy v balíčku A, ostatní třídy jsou v balíčku B. Všechny třídy v balíčku B nesmí být závislé na knihovně `java.io`, která zastřešuje vstup-výstupní funkce.

Prázdné balíčky. Tento bod souvisí s čistotou kódu. Pomocí pravidel si lze vynutit, že nebudou existovat prázdné balíčky. Prázdné balíčky pouze zvyšují nepřehlednost – jsou nepoužívané a zbytečně tak zvyšují velikost aplikace.

Cykly. V rámci snižování nákladů je dáno směrnicí, kolik maximálně cyklických závislostí může v projektu existovat. Je možné vyčíslit, o kolik je dražší modifikovat či přesunout cyklickou komponentu než jednotlivé nezávislé komponenty. *Příklad:* Empiricky je dáno, že v dané firmě s danou strukturou projektu trvá přesun silně souvislé komponenty o několik procent času déle než přesun bez cyklických závislostí. Každý programátor je zodpovědný za dokončení jedné úlohy. Pokud by bylo silně souvislých komponent v projektu více než určité množství, programátorovi by změny zabraly o tolik více času, že by nestihl splnit svůj úkol, a vznikly by výdaje spojené s pozdním dodáním.

Výčet výše uvedený není kompletní. Každý jednotlivec, kolektiv či firma může přijít s dalším důvodem pro vytvoření nového pravidla, které bude zajišťovat dodržení určité podmínky.

7.3.3 Nasazení ve firmě

Ve firmě není současná verze aplikace Classycle plně nasazena. Její funkčnost je otestována manuálně. Důvod nenasazení Classyclu spočívá v tom, že byl modifikován výstup a způsob spuštění. Tuto změnu je nutné promítnout do Maven modulu, který nyní aplikaci Classycle automaticky spouští. To zatím nebylo možné, protože kolegové zodpovědní za tuto změnu jsou alokováni jinde a nemůžou se této změně z časových důvodů věnovat.

Kapitola 8

Závěr

V diplomové práci byla čtenářům představena aplikace Classycle. V úvodu je čtenáři nastíněn úvod do porozumění cizího textu, neboť se v rámci diplomové práce upravovalo již existující řešení. Je zde vysvětlen důvod pro optimalizaci kódu a zdůrazněna důležitost tzv. čistého kódu 2. Dále se čtenář seznámí se základy testování, zejména s hierarchií testů 3. Aplikace nově vytváří grafické výstupy, které zobrazují analyzovaná data ve formě grafu. Základní pojmy z teorie grafů jsou uvedeny v kapitole 4. Zde je také popsána silně souvislá komponenta, její definice, použití silně souvislých komponent v teorii grafu a algoritmy pro nalezení silně souvislých komponent. Použité nástroje Maven, Gephi, yEd a Java jsou popsány v 5.

Díky získaným teoretickým znalostem, které jsou rozepsány v prvních kapitolách, byla aplikace v kapitole 6 detailně analyzována a popsána. Je zde kladen důraz na detailní popsání původního řešení, jehož nastudování bylo nutnou podmínkou pro následnou modifikaci aplikace. Nastudování aplikace před změnou se zabývá kapitola 6, ve které jsou v rámci původních vstupů a výstupů popsány oba módy, ve kterých aplikace umí analyzovat vstupní bajtkód. Modifikací aplikace se zabývá praktická část diplomové práce, kapitola 7.

V diplomové práci je popsáno vytvoření nových forem výstupu aplikace. Čtenář si může prohlédnout ukázky výstupů v sekci 7.2 a to jak pro analýzu na úrovni tříd, tak pro analýzu aplikace na úrovni balíčků. Velmi důležitou součástí diplomové práce je sekce 7.3, ve které je popsáno praktické využití aplikace. Je zde vysvětlen problém závislostí a cyklických závislostí v kódu. Dále jsou zde vytvořeny metodiky pro tvorbu pravidel, díky kterým se kontrolují povolené a zakázané závislosti mezi třídami a balíčky. Díky nově vytvořeným výstupům, které jsou přehlednější než výstupy původní, bude Classycle pomáhat uživatelům snižovat počet závislostí a cyklických závislostí mezi třídami a balíčky a tím snižovat náklady spojené s vývojem analyzované aplikace. Jelikož se jedná o volně šiřitelnou aplikaci, bude modifikovaná aplikace použita jak ve firmě, tak může být použita i mimo ni.

Všechny body zadání diplomové práce byly splněny. Vhodným rozšířením, kterým je možné aplikaci vylepšit, je možné modifikovat výstupy tak, aby ještě více usnadnili programátorovi rozhodování při odstraňování nevhodných závislostí. Aplikaci je vhodné dále refaktorizovat.

Literatura

[1] *Nástroj yEd online*. [Online; navštíveno 23.1.2016].
URL https://gephi.org/

[2] *Obrázek silně souvislých komponent*. [Online; navštíveno 23.4.2017].
URL http://ramos.elo.utfsm.cl/~lsb/elo320/aplicaciones/aplicaciones/CS460AlgorithmsandComplexity/lecture11/COMP460%20Algorithms%20and%20Complexity%20Lecture%2011.htm

[3] *Primární, sekundární a terciální barvy*. [Online; navštíveno 15.3.2017].
URL https://designschool.canva.com/color-theory/

[4] *Struktura java bajtkódu*. [Online; navštíveno 1.4.2017].
URL https://docs.oracle.com/javase/specs/jvms/se7/html/jvms-4.html

[5] *Tabulka typů souboru*. [Online; navštíveno 1.4.2017].
URL http://www.garykessler.net/library/file_sigs.html

[6] *The Apache Software Foundation*. [Online; navštíveno 20.11.2016].
URL https://maven.apache.org/

[7] *Webové stránky nástroje gephi*. [Online; navštíveno 21.10.2016].
URL https://gephi.org/

[8] *Webové stránky nástroje yEd*. [Online; navštíveno 23.1.2016].
URL https://www.yworks.com/downloads#yEd

[9] *Norma ČSN ISO/IEC 9126-1. . Softwarové inženýrství - Jakost produktu - Část 1: Model jakosti*. Český normalizační institut, 2002.

[10] Alam, A.; Padenga, T.: *Application Software Reengineering*. Dorling Kindersley, 2010, ISBN 978-81-317-3185-7.

[11] Demel, J.: *Grafy a jejich aplikace*. Academia, 2002, ISBN 80-200-0990-6.

[12] Fowler, M.: *Refaktoring: zlepšení existujícího kódu*. Grada, 2003, ISBN 80-247-0299-1, [Online; navštíveno 20.12.2016].
URL https://books.google.cz/books?id=hsvzc-XE3nYC&printsec=frontcover#v=onepage&q&f=false

[13] Květoňová, S.: *Materiály k předmětu Strategické řízení informačních systémů: Business Process Reengineering*.

[14] Mancoridis, S.: *Slidy univerzity Drexel University: The Realities of Software Testing.* [Online; navštíveno 26.12.2016].
URL https://www.cs.drexel.edu/~spiros/teaching/SE320/slides/testing-realities.pdf

[15] Martin, R. C.: *Čistý kód.* Computer Press, 2009, ISBN 987-80-251-2285-3.

[16] Myers, G. J.; Badgett, T.; Sandler, C.: *The Art Of Software Testing.* Wiley, 1946 — 3rd ed., ISBN 978-1-118-03196-4.

[17] Pecinovský, R.: *Java 7.* Grada, 2012, ISBN 978-80-247-3665-5.

[18] Roudenský, P.; Havlíčková, A.: *Řízení kvality softwaru.* Computer Press, 2013, ISBN 978-80-251-3816-8.

[19] Sonatype: *Maven: The Definitive Guide.* Sonatype, 2008, ISBN 978-0-596-51733-5.

[20] Šlapal., J.: *Opora k předmětu Matematické struktury v informatice.*