

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

Bakalářská práce

Nástroje pro zjišťování pokrytí kódu testy

Prohlášení

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

V Plzni dne 6.5.2014

.....

Václav Žák

Poděkování

Rád bych poděkoval vedoucímu své bakalářské práce Doc. Ing. Pavlu Heroutovi Ph.D. za cenné rady, připomínky, informace a čas, který mi věnoval při řešení dané problematiky.

Abstract

The thesis deals with structural testing and code coverage metrics. Currently, at least 14 defined types of coverage can be found in literature. Although, only some of them are being used in practice. Likewise, there is a considerable amount of software tools for measuring various types of coverage. The main goal of the thesis is to find a suitable compromise, i.e. a tool that will be able to work with all the major types of coverage and will also meet several other non-functional requirements. Due to the multi-criteria evaluation it is possible to easily assess other tools as well.

Abstrakt

Práce se zabývá problematikou strukturálního testování a metrikami pokrytí kódu testy. V současné době lze v literatuře najít definice minimálně 14 druhů pokrytí. Ovšem pouze některé z nich se prakticky využívají. Stejně tak existuje značné množství softwarových nástrojů měřících různé typy pokrytí. Cílem práce je najít vhodný kompromis, tj. nástroj, který bude schopen pracovat s nejpoužívanějšími typy pokrytí a bude dále splňovat několik dalších mimofunkčních (non-functional) požadavků. Díky použitému multikriteriálnímu vyhodnocení je možné, aby byly do hodnocení snadno přidávány i další nástroje.

Obsah

1. Úvod.....	8
2. Testování software.....	9
2.1. Co je testování.....	9
2.1.1. Testování při vývoji softwaru.....	9
2.1.2. Testování a fáze životního cyklu vývoje software.....	9
2.1.3 Důvody, proč testovat.....	10
2.2. Chyby.....	11
2.2.1. Vznik chyb.....	11
2.2.3. Náklady na opravy chyb.....	12
2.2.4. Příklad známého selhání.....	13
2.3. Druhy testování.....	13
2.3.1. Pohled na testování dle fáze vývoje.....	13
2.3.2. Pohled na testování dle pohledu na systém.....	14
3. Metriky pokrytí kódu.....	16
3.1. Druhy pokrytí.....	16
3.1.1. Pokrytí příkazů/řádek (statement/line coverage).....	16
3.1.2. Pokrytí hran/rozhodnutí (desicion/branch coverage).....	17
3.1.3. Pokrytí podmínek (condition coverage).....	19
3.1.4. Pokrytí cest (path coverage).....	20
3.1.4.1. Cyklomatická složitost.....	21
3.1.5. Další metriky pokrytí.....	22
3.1.5.1. Pokrytí cyklů (loop coverage).....	22
3.1.5.2. Pokrytí toku dat (data flow coverage).....	23
3.1.5.3. Pokrytí funkcí/metod (function/method/call coverage).....	23
3.1.5.4. Pokrytí relačních operátorů (relation operators coverage).....	23
3.1.5.5. Pokrytí souběžného provádění kódu (race coverage).....	24
3.1.5.6. Pokrytí QMO (Quest Mark Operator coverage).....	24
3.1.5.7. Pokrytí kódu lineární sekvence a skoku (linear code sequence and jump coverage).....	24
3.1.5.8. Pokrytí vstupu/výstupu (entry/exit coverage).....	24
3.1.5.9. Pokrytí bloků synchronized (synchronized statement coverage).....	24
3.1.5.10. Pokrytí křehkých změn (weak mutation coverage).....	24
4. Existující nástroje.....	25
4.1. Komerční nástroje.....	25
4.2. Open-source nástroje.....	26
4.2.1. Quilt.....	26
4.2.2. NoUnit.....	27
4.2.3. InsECT.....	27
4.2.4. Jester.....	27
4.2.5. JVMDI Code Coverage Analyser.....	27
4.2.6. GroboCodeCoverage.....	27
4.2.7. jcoverage/gpl.....	27
4.2.8. JBlanket.....	27
4.2.9. Hansel.....	28

4.2.10. Coverlipse.....	28
4.2.11. PIT.....	28
4.3. Nástroje vybrané k testování.....	28
4.3.1. CodeCover.....	29
4.3.1.1. Základní informace CodeCover.....	29
4.3.1.2. Boolean analyzer pluginu CodeCover pro Eclipse.....	29
4.3.1.3. Korelační matice pluginu CodeCover pro Eclipse.....	30
4.3.2. Cobertura.....	31
4.3.2.1 Základní informace.....	31
4.3.3. EclEmma.....	32
4.3.3.1. Základní informace EclEmma.....	32
5. Ukázky kódu jednotlivých metrik.....	34
5.1. Pokrytí řádek/příkazů.....	34
5.1.1. Vhodný případ.....	34
5.1.2. Nevhodný případ.....	35
5.1.3. Vyhodnocení.....	36
5.2. Pokrytí hran/rozhodnutí.....	36
5.2.1. Vhodný případ.....	36
5.2.2. Nevhodný případ.....	37
5.2.3. Vyhodnocení.....	38
5.3. Pokrytí podmínek.....	38
5.3.1. Vhodný případ.....	38
5.3.2. Vyhodnocení.....	40
5.4. Pokrytí cyklů.....	40
5.4.1. Vhodný případ.....	40
5.4.2. Vyhodnocení.....	41
5.5. Pokrytí ternálních operátorů.....	42
5.5.1. Vhodný případ.....	42
5.5.2. Vyhodnocení.....	43
5.6. Pokrytí metod.....	43
5.6.1. Vhodný případ.....	43
5.6.2. Vyhodnocení.....	44
6. Zhodnocení dosažených výsledků.....	45
6.1. CodeCover.....	45
6.1.1. Příprava nástroje.....	45
6.1.2. Testování podporovaných pokrytí.....	45
6.1.4. Vyhodnocení výsledků.....	51
6.2. EclEmma.....	51
6.2.1. Příprava nástroje.....	51
6.2.2. Testování podporovaných pokrytí.....	51
6.2.3. Generování souhrnného reportu.....	55
6.2.4. Vyhodnocení výsledků.....	56
6.3. Cobertura.....	56
6.3.1. Příprava nástroje.....	56
6.3.2. Testování podporovaných pokrytí.....	56
6.3.3. Generování souhrnného reportu.....	57
6.3.4. Vyhodnocení výsledků.....	58
6.4. Vícekriteriální hodnocení.....	58

6.4.1. Podporované techniky měření pokrytí.....	59
6.4.2. Možnosti spouštění.....	60
6.4.3. Přehlednost reportu.....	60
6.4.4. Funkce navíc.....	61
6.4.5. Vhodnost použití nástroje na danou techniku.....	61
6.4.6. Integrace s JUnit.....	62
6.4.7. Podpora verze Eclipse.....	63
6.4.8. Oblíbenost nástroje.....	63
6.4.9. Výsledky vícekritériálního hodnocení.....	63
7. Závěr.....	66
Seznam použitých zdrojů.....	67
Terminologický slovník.....	69
Seznam obrázků.....	70
Seznam tabulek.....	71

1. Úvod

Software je v určité podobě nedílnou součástí našeho každodenního života, ať přímo či nepřímo. Se stále hlubší integrací a zvyšující se složitostí i nákladností jak podpůrných, tak kritických systémů logicky rostou i požadavky na jejich kvalitu. Tento koncept kvality je závislý na kontextu charakteristik, jako je například funkčnost, použitelnost, bezporuchovost a účinnost. Každá charakteristika je pak předmětem měření, tedy zpravidla testování. Dá se říci, že testování poskytuje informace o kvalitě produktu.

Jedna z úrovní testování, testování jednotek, ve které se provádí testování psaním testů pro jednotlivé části kódu programu samotným programátorem, obsahuje právě techniku měření pokrytí kódu. Tato technika, podle dostupné literatury, v dnešní době obsahuje minimálně 14 druhů metrik pro testování pokrytí. Ty mají za úkol stanovit, jak je daný software otestován.

V současné době existuje mnoho nástrojů pro testování pokrytí kódu, avšak různé kvality. Proto je vhodné nástroje prozkoumat, podle kritérií vybrat tři z nich, na kterých budou následně otestovány vlastní vytvořené kódy pro demonstraci jednotlivých metrik. Samozřejmě mohou nastat i komplikace s vhodným a nevhodným použitím jednotlivých testovacích případů. Hlavním cílem práce je najít vhodný nástroj, který bude schopen pracovat s nejpoužívanějšími typy pokrytí a bude dále splňovat několik mimofunkčních požadavků. Bude také zapotřebí vytvořit kvalifikované posouzení s multikriteriálním hodnocením, aby si každý mohl vybrat dle svých potřeb.

Hlavním důvodem, proč jsem si tuto práci vybral, je seznámit se z technikami testování, které nejsou moc běžně užívané a přitom v rámci vývoje software velmi efektivní. Pokrytí kódu testy, které slouží účelu strukturálního testování, je velmi rozsáhlá oblast této problematiky, která není dosud, alespoň v našich končinách, dostatečně probádaná.

2. Testování software

Pokrytí kódu testy je součástí problematiky testování softwaru. Proto tedy v následující kapitole rozeberu základy této problematiky.

2.1. Co je testování

Testování je proces hodnocení systému nebo nějaké jeho součásti za účelem zjistit, zda splňuje zadané požadavky či nikoliv. Tato činnost má za následek skutečné či očekávané rozdíly jejich výsledků [1]. Jednoduše je tedy testování spouštění systému s cílem identifikovat všechny jeho nedostatky, chyby a chybějící požadavky oproti skutečnému záměru.

2.1.1. Testování při vývoji softwaru

Spolupráce testerů a vývojářského týmu spolu s porozuměním dohodnuté specifikace požadavků produktu se zákazníkem jsou hlavní aspekty při vývoji softwaru. Právě zákazník určuje, jaké požadavky má produkt plnit. Kvalita následného produktu je pak taková, která tyto požadavky do určité míry splňuje.

Testování softwaru při vývoji software lze charakterizovat jako proces potvrzování a ověřování, že počítačový program:

- splňuje požadavky, které řídí jeho návrh a vývoj
- pracuje podle očekávání
- může být implementován se stejnými vlastnostmi
- splňuje potřeby všech zúčastněných stran [2]

2.1.2. Testování a fáze životního cyklu vývoje software

Fáze životního cyklu vývoje software:

1. Analýza a specifikace požadavků

- Přáním zákazníka je potřeba analyzovat a zapsat požadavky v jasné a strukturované podobě. Pozornost je věnována pouze požadavkům samotným, nikoliv jejich realizaci.

2. Návrh systému

- Architektonický návrh - slouží k ujasnění koncepce systému a k jeho rozebrání. Plánuje se akceptační testování a nasazení do provozu (včetně

zaškolení uživatelů).

- Podrobný návrh – soustřeďuje se na podrobnou specifikaci softwarových součástí, na výběr algoritmů, na stanovení struktury dat a ošetřování chybových a neočekávaných stavů. Specifikují se požadavky na lidské zdroje, odhaduje se doba trvání a náklady na projekt. Výsledkem by měl být také návrh testů součástí včetně testovacích dat. [3]

3. Implementace

- Programová realizace softwarových součástí, vypracování dokumentace k součástem a otestování implementovaných součástí.

4. Integrace a testování

- Spojení součástí do jediného celku, otestování celého systému a oprava chyb v součástech, které se projeví až při integraci.

5. Provoz a údržba

- Otestování systému uživatelem/zákazníkem. Po akceptaci systému zákazníkem následuje instalace systému a školení uživatelů.

V každé této fázi je velmi důležité ověřování správnosti a testování!

2.1.3 Důvody, proč testovat

Tak jako jsou chyby běžnou součástí našeho života, chyby v programech budou také existovat vždy.

Software testujeme, protože se snažíme odhalit chyby předtím, než by mohly uškodit.

Snižování nákladů

Přestože jednou z nejčastějších manažerských chyb je při snižování nákladů obětování kvality, právě efektivně nastavené zajišťování kvality přináší největší úspory.

Podle výzkumu, provedeném *National Institute of Standards and Technology* v roce 2002, uvádí, že softwarové chyby stojí ekonomiku USA ročně 59,5 miliard dolarů.

Z ekonomického hlediska by testování mělo trvat tak dlouho, dokud předpokládaná průměrná cena nalezení a opravy chyby objevené v příštím cyklu testů je menší než průměrné náklady na chybu objevenou zákazníkem vynásobené pravděpodobností objevu chyby. Zjednodušeně, testovat by se mělo tak dlouho, dokud je to finančně výhodnější než netestovat. Zajišťování kvality se navíc snaží nastavit procesy tak, aby docházelo k co nejmenšímu počtu chyb a ty byly co nejlevněji odstranitelné. Zajištění

kvality je investice a je tedy vhodné sledovat její návratnost.[4]

Hlavním problémem je velmi obtížné zvládnutí řízení kvality tak, aby bylo maximálně efektivní a aby minimalizovalo náklady. To vyžaduje zkušenosti a výborné znalosti. Aby bylo tedy testování efektivní, je potřeba dělat ho dobře.

2.2. Chyby

Definice chyby by se neměla soustředit pouze na specifikaci, protože pak by nastala situace, že projekty bez specifikace by nemohly být testovány, protože by se nikdy nemohla najít žádná chyba. Také by neměla definice zapomínat na očekávání klienta, která nejsou na první pohled patrná a jejichž ohrožení by klient ani nepoznal.[5]

Definice Rona Pattona [6]:

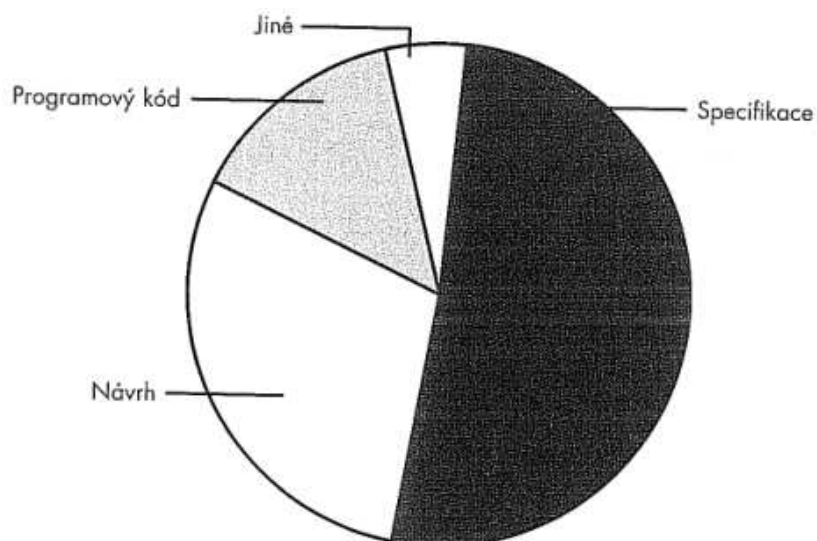
„O softwarovou chybu se jedná, je-li splněna jedna nebo více z následujících podmínek:

- 1. Software nedělá něco, co by podle specifikace dělat měl.*
- 2. Software dělá něco, co by podle specifikace dělat neměl.*
- 3. Software dělá něco, o čem se specifikace nezmiňuje.*
- 4. Software nedělá něco, o čem se produktová specifikace nezmiňuje, ale měla by se zmiňovat.*
- 5. Software je obtížně srozumitelný, těžko se s ním pracuje, je pomalý nebo – podle názoru testera softwaru – jej koncový uživatel nebude považovat za správný. „*

2.2.1. Vznik chyb

U software, na rozdíl od fyzických systémů, kde můžeme narazit na určené množství předvídatelných chyb, se mohou vyskytnout zvláštní případy chyb ve velkém množství. Může to být dáno např. jinou konfigurací (operačního systému), než na které byl program testován, proto není možné provést testování za všech možných podmínek. Jak znázorňuje obrázek *Obr. 1* (Ron Patton, 2002), pomineme-li prvotní problém (tj. specifikace programu), nejedná se u většiny software o výrobní chyby, ale o chyby návrhu programu a přímo programovém kódu. Tyto činí 50 – 70 % [5].

Samozřejmě záleží i na typu projektu, jelikož vývoj řízený testy, schopný analytik či zkušený programátoři mohou tyto procenta značně ovlivnit.

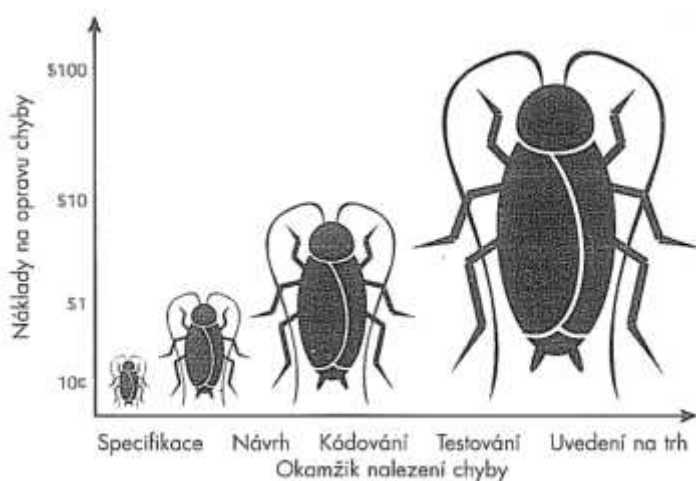


Obr. 1 – Příčiny chyb [6]

Jestli tyto chyby nenalezneme během testování, je velmi pravděpodobné, že budou časem aktivovány.

2.2.3. Náklady na opravy chyb

Jak je psáno v kapitole 2.1.3 *Důvody, proč testovat*, čím dříve na chybu přijdeme, tím její oprava je levnější. Jak rostou náklady na opravu těchto chyb lze vidět na obrázku *Obr. 2*.



Obr. 2 – Náklady na opravu chyb v závislosti na fázi vývoje software [6]

2.2.4. Příklad známého selhání

V závislosti na účelu systému mohou mít selhání způsobená softwarovými defekty různé následky, včetně škodách na lidském zdraví.

Mezi lety 1985 a 1987 se stal tragický případ ve spojitosti s důvěrou v software jednotky Therac-25, určené k ozařování pacientů s rakovinou. Díky nové verzi tohoto přístroje, který měl stejný software, ale oproti předchozí verzi odstraněnou hardwarovou pojistku, umožnilo nastavit mnohonásobně vyšší dávky ozáření. Na následky zemřelo nebo bylo vážně zraněno šest lidí. [7]

2.3. Druhy testování

Na testovaný systém je možné se dívat z různých perspektiv. Jedním z hlavních dělení jsou dle fáze vývoje či dle pohledu na systém.

2.3.1. Pohled na testování dle fáze vývoje

Testy mohou být odvozeny od požadavků a specifikací, návrhu až po samotný zdrojový kód. Různé úrovně testování doprovází odlišné vývojové stupně vytváření software [8].

Na základě toho na jaké úrovni se testování provádí, v jakém časovém horizontu od napsání kódu, se testování dělí na pět stupňů:

1. Testování programátorem (Developer testing)

- Po napsání kódu je prověřen programátorem, většinou jiným, než kód napsal.

2. Testování jednotek (Unit testing)

- U objektově orientovaného programování se jedná o testování jednotlivých tříd a metod. Jednotka se chápe jako samostatná testovatelná část. Testy těchto jednotek se zapisují ve formě programového kódu za pomoci nástrojů na bázi frameworků (JUnit). Na již zaběhlých projektech se velmi špatně dodatečně aplikují. Testy této práce jsou zaměřeny právě na testování jednotek.

3. Integrační testování (Integration testing)

- Integrační testy již nevytváří programátor, ale tým testerů. Hlavním úkolem je ověřit bezchybnou komunikaci jednotlivých modulů. Chyba, která se během testování projeví, se jistě projeví i v dalších fázích.

4. Systémové testování (System testing)

- Během těchto testů je aplikace ověřována jako jeden funkční celek. Ověřují aplikaci z pohledu zákazníka. Slouží jako výstupní kontrola softwaru.

5. Akceptační testování (User acceptance testing)

- Akceptační testy na straně zákazníka. Pokud byl odsouhlasen přenos aplikace k zákazníkovi, zákazník si aplikaci ověří svým týmem testerů. Pokud jsou chyby ve fázi akceptačních testů objeveny, je nutné je v co nejkratším termínu opravit a předat aplikaci zákazníkovi k dalším testům.

Všechny chyby, objevené během testování, mohou vést ke zpoždění termínu nasazení softwaru do provozu a fatální dopad na úspěch celého projektu.

2.3.2. Pohled na testování dle pohledu na systém

V kontextu testování se běžně rozlišují tyto tři základní úrovně pohledu.

1. Černá skříňka

- Při testování černé skříňky se zaměřujeme na vstupy a výstupy programu, bez znalosti, jakým je implementován. Je viditelný pouze zvenčí – z pohledu koncového uživatele. Do této perspektivy patří všechny druhy testů uživatelských rozhraní, akceptační testy, testování podle scénářů.

2. **Bílá skříňka**

- Jsou viditelné detaily systému a je znám způsob jeho fungování. Máme kompletní přístup ke zdrojovému kódu a na základě toho jej testujeme. Vidíme vnější i vnitřní reakce systému. Tím zanedbáváme pohled uživatele, ale lépe se dá odhadnout, kde hledat chyby.

3. **Šedá skříňka**

- O implementaci a vnitřních pochodech produktu něco víme, ale ne tolik, aby to spadalo pod testování „bílých skříňky“. Například nemáme k dispozici celý zdrojový kód, ale html kód webových stránek nebo informace o designu aplikace [5]. Systém je testován stylem „černá skříňka“.

3. Metriky pokrytí kódu

Metrika pokrytí kódu testy zjišťuje, kolik procent zdrojového kódu programu je otestováno souborem testovacích případů. Díky tomuto měření můžeme určit, kdy je kód dostatečně otestován. Hledáme tu část kódu, která není pokryta testy. Proces pokrytí v sobě zahrnuje i vytváření nových testů k zajištění většího pokrytí kódu testy.

Přestože tyto metriky pracují se zdrojovými kódy programu, nelze jimi měřit kvalitu celého produktu. Nacházíme se v úrovni testování jednotek dle fáze vývoje a z pohledu na systém v úrovni bílé skříňky.

3.1. Druhy pokrytí

Běžně se používá asi pět druhů pokrytí. V roce 2000 byly známy druhy pouze tři [6].

Nyní jich lze celkem rozeznat asi 14.

Příklady metrik budou uvedeny v pseudokódu.

3.1.1. Pokrytí příkazů/řádek (statement/line coverage)

Do této kategorie spadá pokrytí jednotlivých příkazů a řádků. Pouze kontroluje, zda daný příkaz/řádek byl spuštěn v průběhu testování, pro pokrytí podmínky tak může stačit její libovolné vyhodnocení. Ve složitějším kódu je toto pokrytí nedostatečné a je dobré ho doplnit jiným pokrytím.

Je-li tedy k dispozici kód, který obsahuje 100 řádků a během běhu programu se spustí (otestuje) řádek 50, výsledné pokrytí bude činit 50%.

Pokud bude každý řádek zpracován pouze jednou, vzorec pro pokrytí příkazů/řádek bude vypadat následovně:

$$\text{Pokrytí [\%]} = \frac{\text{počet spuštěných řádků/příkazů}}{\text{celkový počet řádků/příkazů}} * 100$$

Formálně: T splňuje kritérium pokrytí příkazů pro daný kód K , jestliže pro každý příkaz p náležící kódu K existuje test t z množiny T , že při provedení t bude spuštěn příkaz p . [9]

Příklad:

1. `integer a;`
2. `integer b;`
3. `integer c;`
4. `c = a + b;`

5. `if (c > 10){ vypiš „součet čísel je větší než 10“ }`

V tomto příkladu, pro 100% pokrytí řádků, stačí jeden test, zvolením čísel $a = 5$ a $b = 6$. Je vyhodnocena pouze kladná podmínka větve *if*. Pokud by existovala záporná podmínka, ve které by existoval příkaz, nebo pouze výběrem jiných čísel a , b , jejichž součet by byl menší než 10, tento příkaz se během spuštění již nevyhodnotí, nespadá tedy do pokrytí. Jak je psáno již výše, toto pokrytí je nedostatečné a mělo by být doplněno jinou metrikou pokrytí.

Pokrytí bloků

Variantou pokrytí řádek, je pokrytí bloků (block coverage). Poskytuje stejné výsledky založené na blocích namísto na jednotlivých příkazech. Ve složitějším kódu je toto pokrytí opět nedostatečné.

3.1.2. Pokrytí hran/rozhodnutí (decision/branch coverage)

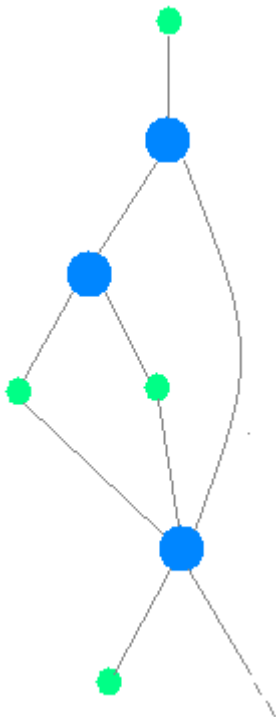
Každá podmínka je uzlem, ze kterého vedou dvě hrany (*true/false*). Je tedy testováno kladné i záporné vyhodnocení během sledu testovacích případů. Pro 100% pokrytí je tedy zapotřebí minimálně dvou testovacích případů. Tato metrika zkoumá vyhodnocení podmínky jako celku a nejsou brány v potaz vyhodnocení jednotlivých podmínek, které obsahuje podmínka složená. Složená podmínka se pořád bere jen jako jedna.

Pokud budeme testovat jednu podmínku, spustíme pro ni dva testy pro kladné a záporné vyhodnocení, každý pokryje kód z 50%, celkové pokrytí tedy bude 100%.

Vzorec pro procentuální výpočet pokrytí hran:

Pokrytí [%] = (počet true a false hran, pro které spustíme testy / 2* celkový počet podmínek) * 100

Formálně: Představme si graf, kdy příkazy tvoří uzly a možné přechody mezi nimi hrany. Pak za sebou provedené příkazy tvoří hranu a podmínka uzlem, ze kterého vedou dvě hrany, jedna pro *true* a jedna pro *false* hodnotu. Pak množina testů T splňuje kritérium pokrytí hran pro daný kód K , jestliže pro každou hranu h výše popsaného grafu existuje test t z množiny T , že při provedení t projde výpočet hranou h . [9]



Obr. 3 – Graf metriky pokrytí hran [9]

Příklad pro vyhodnocení pokrytí jedné podmínky:

1. **integer** a;
2. **integer** b;
3. **integer** c;
4. $c = a + b$;
5. **if** ($c > 10$){ **vypiš** „součet čísel je větší než 10“ }
6. **else** { **vypiš** „součet čísel je menší než 10“ }

Z příkladu je snadné pochopit, že při vytvoření dvou testovacích případů (tj. zadání čísel a , b v součtu větším než 10 a v druhém případě zadáním čísel a , b v součtu menším než 10) bude pokrytí hran 100%.

Příklad pro vyhodnocení pokrytí vnořené podmínky:

1. **integer** a;
2. **integer** b;
3. **integer** c;
4. $c = a + b$;
5. **if** ($c > 10$){ **vypiš** „součet čísel je větší než 10“ }
6. **if** ($a > 5$) { **vypiš** „číslo a je větší než 5“ }
7. **else** { **vypiš** „číslo a je menší než 5“ } }
6. **else** { **vypiš** „součet čísel je menší než 10“ }

Každá hrana podmínky, i vnořené musí být pro 100% pokrytí vyhodnocena. Proto celkový počet testů pro dosažení 100% pokrytí bude čtyři:

1. kladné vyhodnocení první podmínky
2. záporné vyhodnocení první podmínky
3. kladné vyhodnocení podmínky vnořené
4. záporné vyhodnocení podmínky vnořené

Příklad pro vyhodnocení pokrytí složené podmínky:

1. `integer` a;
2. `integer` b;
3. `integer` c;
4. `c = a + b;`
5. `if (c > 10 && a > 5 || b > 5){ vypiš „součet čísel je větší než 10 a číslo a nebo číslo b je větší než 5“ }`
6. `else { vypiš „součet čísel je menší než 10“ }`

Jelikož při tomto pokrytí nebereme v úvahu všechny větve složené podmínky, stačit nám budou pouze dva testovací případy – kladné a záporné vyhodnocení podmínky. Kvůli těmto případům můžou zůstat chyby i při 100% pokrytí rozhodnutí.

3.1.3. Pokrytí podmínek (condition coverage)

Pokrytí podmínek vyhodnocuje, jestli byla ve složených podmínkách testována celá podmínka. Je tedy testováno *true/false* i u podvýrazů ve složené podmínce. Pro všechny možné vstupy je vždy celá podmínka vyhodnocena jako pravdivá nebo nepravdivá. Vylepšení pokrytí podmínek spočívá v tom, že se berou v úvahu všechna možná vyhodnocení podmínky, nejen to, zda je pravdivá, či nikoli.

Jelikož se berou v úvahu všechna možná vyhodnocení, vzorec pro výpočet pokrytí podmínek bude následovný:

Pokrytí [%] = (počet true a false hran, pro které spustíme testy / 2* celkový počet podmínek) * 100

Formálně: Množina testů T splňuje kritérium pokrytí podmínek pro daný kód K , jestliže splňuje kritérium pokrytí hran a pro každou složenou podmínku platí, že pro každou její část p existují testy t, u z množiny T , že při provedení t se p vyhodnotí kladně a při provedení u záporně. [9]

Příklad:

1. `integer a;`
2. `integer b;`
3. `integer c;`
4. `c = a + b;`
5. `if (c > 10 && a > 5 || b > 5){ vypiš „součet čísel je větší než 10 a číslo a nebo číslo b je větší než 5“ }`
6. `else { vypiš „součet čísel je menší než 10“ }`

Mějme tedy opět složenou podmínku. Výběrem čísel $a = 5$, $b = 6$ splňujeme kladné vyhodnocení podmínky. Při zvolení čísel v součtu menších než 10 zase záporné. Pokrytí ale není 100%, jelikož jsme ve složené podmínce nepokryli případ, kdy číslo a je větší než 5. Proto je pro 100% pokrytí potřeba ještě další testovací případ.

Např.:

1. $a = 5, b = 6$
2. $a = 1, b = 2$
3. $a = 6, b = 5$

3.1.4. Pokrytí cest (path coverage)

Toto pokrytí je nejvyšší stupněm pokrytí. Sleduje všechny možné průchody kódem, takže představuje opravdu podrobné otestování. I když nelze odhalit všechny chyby odhalit jen z kódu, pokrytí cest poskytuje jistotu, že byly otestovány všechny možnosti běhu. Prvním problémem je ale praktická nepoužitelnost tohoto pokrytí, jelikož jeho složitost způsobuje exponenciální růst počtu testů. Další nevýhoda spočívá v nemožnosti otestovat některé cesty kvůli vztahům mezi daty [10].

„Obzvláště náročné je v tomto případě testování cyklů, neboť každá iterace cyklu je považována za zvláštní cestu. Testování skutečně všech cest v netriviální funkci je tedy značně nepraktické.“ [15]

Vzorec pro výpočet:

Pokrytí [%] = (počet cest, pro které spustíme testy / celkový počet cest) * 100

Formálně: Množina testů T splňuje kritérium pokrytí cest pro daný kód K , jestliže splňuje kritérium pokrytí podmínek a pro každou cestu C v grafu kódu spojující vstupní a výstupní uzel grafu a obsahující nejvýše n cyklů existuje test t z množiny T , že při provedení t projde výpočet cestou C . [9]

Příklad:

1. `integer a;`
2. `integer b;`
3. `integer c;`
4. `c = a + b;`
5. `if (c > 10){ vypiš „součet čísel a,b je větší než 10“ }`
6. `d = c - a;`
7. `if (b < 5){ vypiš „číslo b je menší než 5“ }`

Pro tento příklad potřebujeme tedy pro 100% pokrytí otestovat čtyři různé cesty:

1. Kladné vyhodnocení první podmínky.
2. Záporné vyhodnocení první podmínky.
3. Kladné vyhodnocení druhé podmínky.
4. Záporné vyhodnocení druhé podmínky.

Pokud bude druhá podmínka stejná jako první, nikdy 100% pokrytí nedosáhneme.

1. `integer a;`
2. `integer b;`
3. `integer c;`
4. `c = a + b;`
5. `if (c > 10){ vypiš „součet čísel a,b je větší než 10“ }`
6. `if (c > 10){ vypiš „součet čísel a,b je větší než 10“ }`

Jelikož kladné vyhodnocení první podmínky = kladné vyhodnocení druhé podmínky a zároveň záporné vyhodnocení první podmínky = záporné vyhodnocení druhé podmínky, pokrytí bude činit pouze 50% i při čtyřech testovacích případech.

3.1.4.1. Cyklomatická složitost

Cyklomatická (podmínková) složitost (cyclomatic complexity či conditional complexity) je metrika vyvinutá Thomasem J. McCabem a určuje složitost programu podle počtu nezávislých cest zdrojovým kódem [7]. Tato veličina nám umožňuje určit minimální počet cest, které bychom měli v programu otestovat.

Je to běžně používaná metrika během vývoje pro vyhodnocování:

- očekávané spolehlivosti
- testovatelnosti
- udržitelnosti

Cyklomatická složitost je užitečná pro testery, kteří ji mohou použít pro určení minimálního počtu testů potřebných pro testování toku řízení. Měří komplexnost rozhodovací logiky metody nebo celého modulu.

Výpočet se provádí spočtením všech jednoduchých podmínek a přičtení 1.

$$v(G) = \text{počet_podmínek} + 1$$

Metody bez podmínky mají hodnotu 1, na otestování stačí jeden test, $v(G) = 0 + 1 = 1$.

V případě jedné podmínky (dvě větve *TRUE* a *FALSE*) bude hodnota $v(G) = 1 + 1 = 2$.

Cyklomatická složitost jako metrika sledovaná u kódu je pro nás důležitá. Obecně se udává, že by se měla pohybovat kolem čísla 8-15 [7]. Čím je toto číslo vyšší, tím je kód obtížněji testovatelný, náchylnější vůči chybám, hůře čitelný a udržitelný. Vysoká cyklomatická složitost (číslo 20 a více) je často důvodem ke zvážení vhodnosti návrhu a přepracování.

3.1.5. Další metriky pokrytí

3.1.5.1. Pokrytí cyklů (loop coverage)

Zjišťuje u každého cyklu v kódu, zda během testovacího procesu došlo k projití cyklem nula krát, právě jednou nebo více než jednou. Pokrytí je použitelné pro *while*, *do-while* a *for*.

Příklad:

```
1. integer maximum;  
2. integer i;  
3. integer cislo = 0;  
4. for(i = 1; i <= maximum; i++){  
5.     cislo += i; }
```

For cyklus potřebuje tři testovací příklady pro 100% pokrytí kódu. Zvolením hodnoty maximum pokryjeme tyto tři testovací případy:

1. Pokud zvolíme 0, cyklus neproběhne ani jednou.

2. Pokud zvolíme 1, cyklus proběhne jednou.
3. Pokud zvolíme číslo větší než 1, cyklus proběhne více než jedenkrát.

3.1.5.2. Pokrytí toku dat (data flow coverage)

Tato varianta pokrytí cestu kontroluje podcesty od přiřazení proměnné po její použití. Výhodou této metriky je, že kontrolované cesty mají přímý vliv na způsob, jak program zpracovává data. Nevýhoda je, že tato metrika nezahrnuje pokrytí rozhodnutí. Další nevýhodou je složitost. Vývojáři navrhli mnoho variant, které zvyšují složitost této metriky. Například variace rozlišovat mezi použitím proměnné ve výpočtu oproti použití v rozhodnutí, a mezi lokálními a globálními proměnnými. Stejně tak jako u analýzy toku dat pro optimalizaci kódu, ukazatele také zapříčiňují potíže.

Příklad:

1. metoda `prevracenaHodnota (double x){`
2. `vrát 1.0/x; }`

Testovací případ:

1. `double y = prevracenaHodnota(5);`
2. `double ocekavana = 1.0/5.0;`
3. `OCEKAVANY_VYSLEDEK (ocekavana, y);`

Tento test dosáhne 100% pokrytí. Přesto je to jen jeden z $1.8446744e+19$ možných vstupů (za předpokladu, že *double* je 64 bitů velký).

3.1.5.3. Pokrytí funkcí/metod (function/method/call coverage)

Tato metrika kontroluje, zda je alespoň jednou volána každá funkce nebo metoda. Je zvláště užitečná při předběžném testování, kde může zaručit alespoň nějaké pokrytí ve všech oblastech softwaru, protože v této metodě odhalíte části kódu, které by nešly pokrýt testy [10].

Chyby v programech se totiž často objevují v rozhraních mezi moduly.

Příklad:

1. `metoda hodnota()`
2. `{vrát 5;}`
3. `metoda hlavni()`
4. `{vytiskni("Hodnota:"+hodnota());}`

3.1.5.4. Pokrytí relačních operátorů (relation operators coverage)

Tato metrika kontroluje, zda při použití relačních operátorů (<, <=, >, >=) nastane krajní situace. Hypotéza je, že v krajních situacích testovacích případů najdete *off-by-one* chyby (chyba spočívající v použití hodnoty o jednotku menší nebo větší) a použití nesprávných relačních operátorů, jako například < místo <=.

Příklad: [10]

1. **if** (a < b)
2. **PŘÍKAZ;**

Pokrytí relačních operátorů kontroluje, zda dojde k situaci a == b. Dojde-li k této situaci a program se chová správně, můžeme předpokládat, že relační operátor nebude <=.

3.1.5.5. Pokrytí souběžného provádění kódu (race coverage)

Tato metrika kontroluje, zda více vláken vykonává ten samý kód ve stejný čas. To pomáhá odhalit chyby při synchronizování přístupu ke sdíleným prostředkům.

3.1.5.6. Pokrytí QMO (Quest Mark Operator coverage)

Metrika kontroluje, zda jsou pokryty obě možnosti ternárního operátoru (? :).

3.1.5.7. Pokrytí kódu lineární sekvence a skoku (linear code sequence and jump coverage)

Tato varianta pokrytí cesty zvažuje pouze podcesty, které mohou být snadno zastoupeny ve zdrojovém kódu programu. Pokrytí skoku kontroluje, v jakém pořadí je spouštěna posloupnost řádků zdrojového kódu. Použitelné pro break a continue.

3.1.5.8. Pokrytí vstupu/výstupu (entry/exit coverage)

Kontroluje, jestli byly provedeny všechny možné návraty z metody. Vstupní bod metody je ve vyšších programovacích jazycích jen jeden.

3.1.5.9. Pokrytí bloků synchronized (synchronized statement coverage)

Metrika kontroluje, zda příkaz synchronized způsobil čekání zbylých vláken, dokud se uzamčená část neodemkne. Užitečné ke kvantifikování účinku zátěžového testování.

3.1.5.10. Pokrytí křehkých změn (weak mutation coverage)

Dochází ke kontrole otestování alternativních operátorů a kontrole záměny proměnných nebo konstant při přiřazování hodnot.

4. Existující nástroje

Existují tři kategorie analyzátorů, se kterými je možné testovat.

- Nástroje, které pracují se zdrojovým kódem programu
- Nástroje, které pracují s byte-kódem Javy (již zkompileovaný kód)
- Nástroje, které se spouštějí v JVM (Java Virtual Machine)

Princip práce nástrojů, které se spouštějí v JVM:

- Zkoumaný kód je testován, nejčastěji pomocí JUnit testů. Tyto testy spouštějí menší části kódu a nedělají se už žádné další speciální testy.
- Analyzátor se spustí nad testy a zjišťuje, jaké části kódu byly v rámci testů spuštěny. Nerozlišuje mezi tím, zda test uspěl, či selhal, je pro něj důležité jen to, kolik z testovaného kódu bylo spuštěno.

Z výsledku práce analyzátoru snadno zjistíme, jaké části programového kódu naše testovací případy nepokrývají, jaké jsou redundantní nebo jaké nové případy je třeba vytvořit. Ty, které testovací případy nepokrývají, musíme doplnit. Redundantní případy jsou takové případy, kde se po provedení série testovacích případů nezvýší procento pokrytého kódu. Pravděpodobně jsou zbytečné. Nové testovací případy je nutné vytvořit, pokud se zaměřujeme na úsek kódu s nízkým procentem pokrytí.

Zkoumáme, jak pracuje a neotestované úseky pokryjeme novými testovacími případy.

V současné době existuje celá řada nástrojů různorodé kvality pro jazyk Java, které pokrývají široké spektrum technik měření pokrytí. V základě lze tyto nástroje dělit na dvě podskupiny. Komerční a open-source, tedy nekomerční.

4.1. Komerční nástroje

Pod pojmem komerční nástroje se rozumí takové nástroje, které nejsou dostupné zdarma.

Přehled aktuálně dostupných komerčních nástrojů:

Nástroj	Webové stránky	Poslední verze
Clover	https://www.atlassian.com/software/clover/overview	02/2014
Sonar Cube	http://nemo.sonarqube.org/	12/2013
Jtest	http://www.parasoft.com/jtest?itemId=14	12/2012

Tab. 1 - Existující komerční nástroje pro zjištění pokrytí kódu testy

Poznámka: Jedna z podmínek bakalářské práce testování nástrojů na pokrytí kódu je, že budou k dispozici zdarma. Proto komerční nástroje nebudou do analýzy zahrnuty.

4.2. Open-source nástroje

Open-source nástroje jsou nekomerční nástroje, tedy nástroje, které mohou být svázány určitými licencemi, ale jsou k dispozici zdarma nebo zdarma pro nekomerční použití. Splňují tedy zadání bakalářské práce.

Aktuálně dostupné nekomerční nástroje: [12]

Nástroj	Webové stránky	Poslední verze
Quilt	http://quilt.sourceforge.net/	10/2003
NoUnit	http://nounit.sourceforge.net/	12/2003
InsECT	http://insectj.sourceforge.net/	09/2005
Jester	http://jester.sourceforge.net/	11/2005
JVM DI Code Coverage Analyser	http://jvmdicover.sourceforge.net/	03/2002
GroboCodeCoverage	http://groboutils.sourceforge.net/codecoverage/	04/2004
jcoverage/gpl	http://jcoverage.com	01/2002
JBlanket	http://csdl.ics.hawaii.edu/research/jblanket/	02/2006
Hansel	http://hansel.sourceforge.net/	10/2006
Coverlipse	http://coverlipse.sourceforge.net/	02/2009
PIT	http://pittest.org/	08/2012
CodeCover	http://codecover.org/	07/2011
Cobertura	http://cobertura.github.io/cobertura/	08/2013
EclEmma	http://www.eclEmma.org/	03/2014

Tab. 2 - Existující open-source nástroje pro zjištění pokrytí kódu testy

4.2.1. Quilt

Quilt je nástroj pro měření pokrytí řádků, podmínek, cest, cyklů a relačních operátorů. Je optimalizován jak pro přímé měření Java kódu, tak pro použití s JUnit s nástroji pro psaní a automatizaci build scriptů softwarových aplikací Apache Ant a Apache Maven. Nevýhoda je neaktuálnost programu, jelikož poslední verze 0.6b vyšla na podzim roku 2003.

4.2.2. NoUnit

NoUnit je nástroj, který optimalizuje měření pokrytí bez použití JUnit testů. Jednoduše graficky zobrazuje, jaké metody nebyly pokryté, tudíž je použitelný snadno pro každého. Kvůli jednoduchosti jej nemohu doporučit pro výběr nástrojů k testování.

4.2.3. InsECT

InsECT je Java framework, který se používá jako plugin ve vývojářském software Eclipse. Měří pokrytí volání metod a pokrytí podmínek.

4.2.4. Jester

Jester hledá kód, který není pokryt testy. Dokáže změnit kód, spustit testy, a pokud testy projdou, zobrazí zprávu, co změnil. Obsahuje skript pro generování html, která ukazuje změny, pokud testy neprojdou.

4.2.5. JVMDI Code Coverage Analyser

Tento malý nástroj je sdílená knihovna, která se načítá přímo do JVMDI (Java Virtual Machine Debug Interface) a zaznamenává všechny řádky, které jsou spuštěny. Je to celkem hrubý způsob testování pokrytí, ale někdy užitečný.

4.2.6. GroboCodeCoverage

GroboCodeCoverage je čistá Java implementace nástroje pro měření pokrytí. Používá Jakarta BCEL platformu pro post-kompilaci tříd k přidání příkazů ke sledování pokrytí.

4.2.7. jcoverage/gpl

Další z volně dostupných nástrojů, plugin pro Apache Maven, použitelný pro měření pokrytí řádků. Vývoj skončil v roce 2002.

4.2.8. JBlanket

Tento projekt se zabývá zásadami nazývanými "extrémní pokrytí" pro měření kvality unit testů nezávisle na fázi vývoje softwaru. Extrémní pokrytí měří pokrytí metod a používá sadu pravidel závislejších na testovacím systému. K použití s JUnit testy a nástrojem Apache Ant.

4.2.9. Hansel

Hansel slouží k testování 100% pokrytí rozhodnutí skrz JUnit testy. Generuje dodatečné testy, které se nazývají "sondy", z nichž každá ověřuje pokrytí pro jednu metodu nebo rozhodnutí. Každá z nich se nezdaří, pokud se nevztahuje ke spuštění funkčních testů.

4.2.10. Coverlipse

Plugin pro Eclipse, pro vizualizaci měření pokrytí kódu JUnit testy. Podporuje pokrytí bloků (příkazů) a pokrytí toku dat.

4.2.11. PIT

PIT je nástroj, pracující s byte-kódem, který umožňuje testovat efektivitu JUnit testů. Lze použít jako automatizované testování testů. Zahrnuje v sobě testování pokrytí řádků, rychlosti spouštění a testování konvence pojmenování. K použití s nástroji Apache Maven, Ant.

4.3. Nástroje vybrané k testování

Nástroje vybrané k testování pokrytí musely, dle zadání, splňovat většinu následujících kritérií:

- Co největší množství podporovaných technik měření pokrytí
- Možnosti spouštění – příkazová řádka, Apache Ant/Maven, IDE
- Přehlednost reportu
- Užitečné funkce navíc pro přehlednost výsledků testů
- Vhodnost použití nástroje na danou techniku
- Integrace s JUnit
- Podpora nástroje v Eclipse
- Oblíbenost nástroje
- Aktuálnost nástroje

Ze všech nástrojů splňují kritéria tyto tři: CodeCover, Cobertura, EclEmma.

4.3.1. CodeCover

4.3.1.1. Základní informace CodeCover

Odkaz ke stažení	http://codecover.org/
Aktuální verze	1.0.1.2 - 7.7. 2011
Podpora verze Javy	1.5 a vyšší
Podpora verze Eclipse	3.3 a vyšší
Podpora verze JUnit	3 a vyšší
Kategorie nástroje	Pracuje se zdrojovým kódem programu

Tab. 3 - Základní informace o nástroji CodeCover

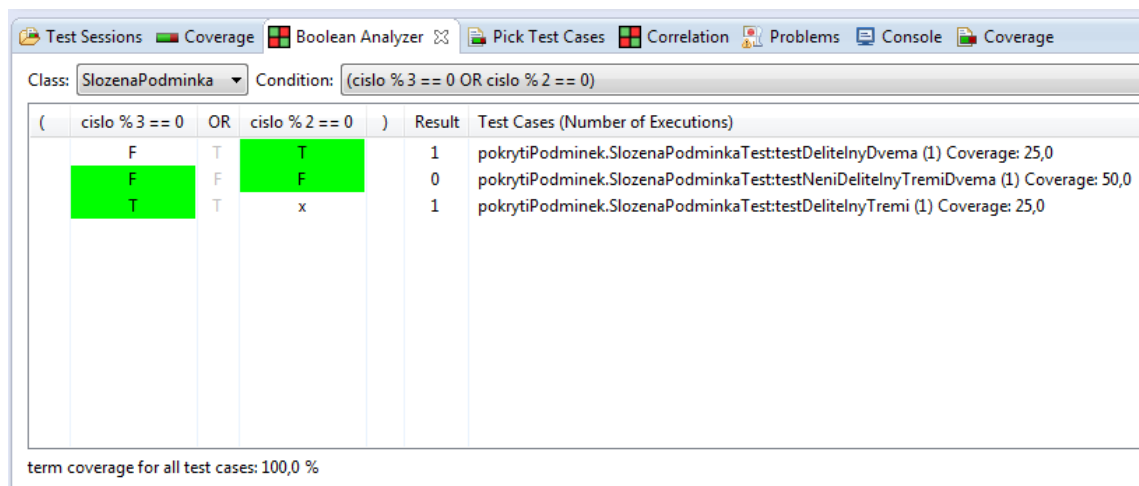
CodeCover je rozšiřitelný open source nástroj pro měření pokrytí kódu vyvinutý v Institutu pro Softwarové Technologie (ISTE) na Univerzitě ve Stuttgartu v roce 2007.

Vlastnosti CodeCover:

- Podporuje pokrytí příkazů, podmínek, rozhodnutí, cyklů, QMO a bloků synchronized.
- Pracuje se zdrojovým kódem pro nejpřesnější měření pokrytí.
- Přizpůsobitelný export do HTML a CSV, i zvlášť pro každou metriku.
- Spustitelný samostatně v Apache Ant, Maven nebo jako plugin v Eclipse.
- Plná integrace s JUnit
- Živá oznámení – k zaznamenávání testů či stažení na vzdáleném počítači.
- Boolean analyzátor, který pomáhá testovacím případům zlepšit pokrytí podmínek.
- Korelační matice, která pomáhá maximálně využít celé testovací sady.
- Záložka *coverage* dokáže zobrazit i zvolené procentuální pokrytí pro snazší přehlednost v testech.
- Barevné označení kódu ve výsledcích pokrytí
- Nastavení jednotlivých tříd pro testování.

4.3.1.2. Boolean analyzer pluginu CodeCover pro Eclipse

Užitečná funkce navíc je tzv. boolean analyzátor. Dokáže přehledně graficky zobrazit pokrytí podmínek pro lepší orientaci ve větvení podmínek při testování pokrytí.



(cislo % 3 == 0	OR	cislo % 2 == 0)	Result	Test Cases (Number of Executions)
	F	T	T		1	pokrytiPodminek.SlozenaPodminkaTest:testDelitelnyDvema (1) Coverage: 25,0
	F	F	F		0	pokrytiPodminek.SlozenaPodminkaTest:testNeniDelitelnyTremiDvema (1) Coverage: 50,0
	T	T	x		1	pokrytiPodminek.SlozenaPodminkaTest:testDelitelnyTremi (1) Coverage: 25,0

term coverage for all test cases: 100,0 %

Obr. 4 - Příklad znázornění boolean analyzátoru nástroje CodeCover

4.3.1.3. Korelační matice pluginu CodeCover pro Eclipse

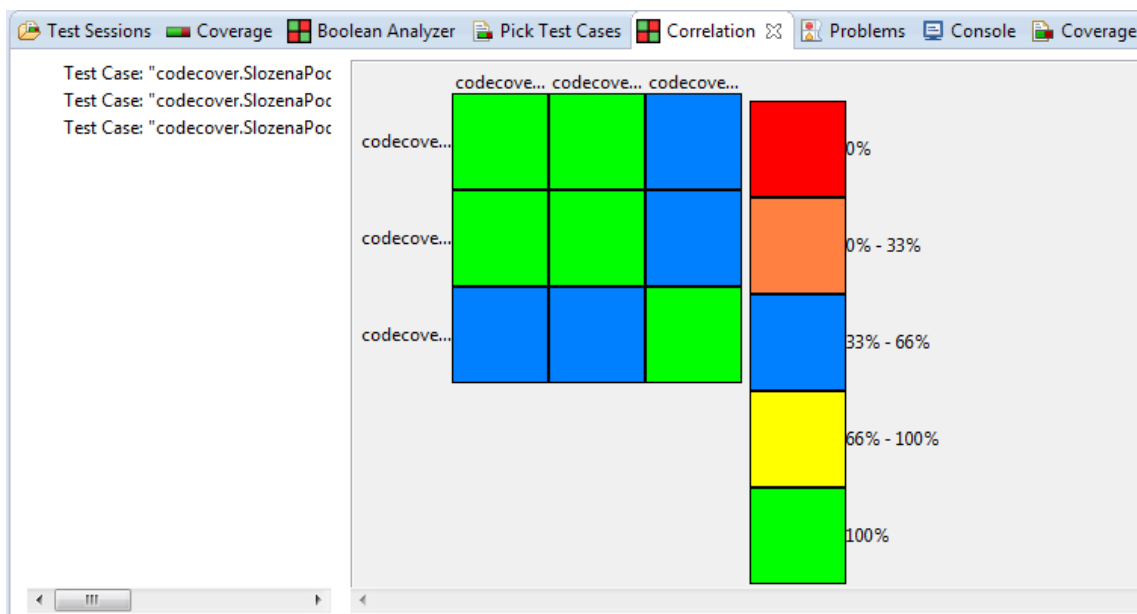
Korelační matice, tedy matice závislostí, je tabulka, která nám ukazuje procentuální shodnost pokrytí jednotlivých testovacích případů.

Testovací případ	1.	2.	3.
1.	100%	100%	33,3%
2.	100%	100%	33,3%
3.	33,3%	33,3%	100%

Tab. 4 - Korelační matice

U prvního a druhého testovacího případu pro pokrytí vidíme, že každý z nich pokrývá stejné tři jednotky, které jdou tímto pokrytím pokrýt. Jejich vzájemná shodnost je tedy 100%. Každý první (druhý) a třetí testovací případ pokrývají tři jednotky, ale společně pokrývají pouze jednu. Je tu tedy 33,3% shodnost.

Korelační matice se tedy zabývá závislostí testovacích případů, které nejsou úplně pokryté na každém jednom stupni pokrytí. Tyto stupně se značí různými barvami. Velikost matice určuje počet testovacích případů (v našem případě je matice řádu 3 – 3 testovací případy). Matici je potřeba číst zleva, například třetí testovací případ pokrývá první testovací případ v rozmezí od 33 % - 66 % (modrá barva, první sloupec, třetí řádek na obrázku *Obr. 5*). Hlavní diagonála nám bude vždy udávat 100%. Pokud budeme mít pouze jeden testovací případ, matice bude řádu 1 a bude udávat vždy 100%.



Obr. 5 - Příklad znázornění korelační matice pluginem CodeCover v prostředí Eclipse.

4.3.2. Cobertura

4.3.2.1 Základní informace

Odkaz ke stažení	http://cobertura.sourceforge.net/download.html
Aktuální verze	Cobertura 2.0.3 - 12.8. 2013 a eCobertura 0.9.8 - 7.11.2010
Podpora verze Javy	1.5 a vyšší
Podpora verze Eclipse	3.3 až 4.2
Podpora verze JUnit	3.8 a vyšší
Kategorie nástroje	Pracuje s již zkompilevaným kódem – byte kód Javy

Tab. 5 - Základní informace o nástroji Cobertura

Cobertura je bezplatný Java nástroj, který měří procentuální pokrytí kódu testy. Je založen na starším projektu jcoverage, na jehož základech se vyvíjí od roku 2005. Cobertura byla založena z důvodu chybovosti poslední verze, která obsahovala větší množství chyb při použití pro testování většího množství tříd. Navíc se vývojáři snaží pro otevřenější vývoj, než tomu tak bylo u původního nástroje.

Slovo „cobertura“ pochází ze Španělštiny jako ekvivalent slova „pokrytí“. [13]

Vlastnosti Cobaturity:

- Spuštění s příkazové řádky, Ant/Maven a plugin pro Eclipse
- Práce s byte kódem Javy

- Report do HTML nebo XML (jen knihovna) – HTML výsledky dokáže různě řadit (sestupně/vzestupně podle jména, procent pokrytí, atd...)
- Měří pokrytí řádek a podmínek
- Barevné označení kódu
- Ukazuje cyklomatickou složitost pro každou třídu, průměrnou pak pro každý balíček tříd a poté i souhrn pro celý projekt.

4.3.3. EclEmma

4.3.3.1. Základní informace EclEmma

Odkaz ke stažení	http://www.eclEmma.org/
Aktuální verze	EclEmma 2.3.0 - 19.3. 2014 a JaCoCo 0.7.0 18.3. 2014
Podpora verze Javy	1.5 a vyšší
Podpora verze Eclipse	3.5 a vyšší
Podpora verze JUnit	3 a vyšší
Kategorie nástroje	Pracuje s již zkompilevaným kódem – byte kód Javy

Tab. 6 - Základní informace o nástroji EclEmma

EclEmma je pokračování open-source projektu Emma pro měření pokrytí kódu. V podstatě se dělí na dva nástroje. Prvním je JaCoCo knihovna, která je samostatně spustitelná v příkazové řádce nebo v Apache Ant/Maven a druhým právě EclEmma jako plugin pro vývojářské prostředí Eclipse. Odlišuje se od ostatních nástrojů tím, že podporuje unikátní kombinaci funkcí, a to podporu pro vývoj v měřítku podnikového softwaru a širší podporu technik měření pokrytí.

Vlastnosti EclEmma:

- Měří pokrytí instrukcí, řádek, podmínek, metod/tříd
- Ukazuje cyklomatickou složitost

- Barevné znázornění pokrytí kódu v Eclipse
 - Ztvárněno třemi symboly:
 - Žádné pokrytí: červený diamant
 - Částečné pokrytí: žlutý diamant
 - Plné pokrytí: zelený diamant
- Spuštění s příkazové řádky, Ant/Maven a plugin pro Eclipse
- Práce s byte kódem Javy

5. Ukázky kódu jednotlivých metrik

V kapitole 3 jsou vysvětleny metriky testování pokrytí kódu. V této kapitole rozeberu ty nejpoužívanější, které jsou zároveň obsažené v nástrojích CodeCover, Cobertura, EclEmma. Výsledky testů budou zobrazeny pomocí všech zvolených nástrojů v kapitole 6. Veškerý kód příkladů je napsán v jazyce Java.

5.1. Pokrytí řádek/příkazů

5.1.1. Vhodný případ

Testování příkazů je vhodné na ošetřování výjimek, protože během jiných testů většinou nedojde k provedení kódu výjimky. Pokrytí testem na tuto skutečnost upozorní.

Vytvoříme si testovanou třídu *vhodnyPripad*, která nám převádí řetězec na kladné číslo typu `short`. Pokud řetězec nejde převést, nebo je číslo záporné, vrací `-1`. Výjimka zadání formátu čísla je ošetřena příkazem *catch*.

```
public class vhodnyPripad {  
  
    public static short prevodNaKladneCislo(String vstup) {  
        try {  
            short pom = Short.parseShort(vstup);  
            if (pom >= 0) {  
                return pom;  
            }  
            else {  
                return -1;  
            }  
        } catch (NumberFormatException e) {  
            return -1;  
        }  
    }  
}
```

Testovací třídu nestačí vytvořit pouze dvěma testovacími případy po pokrytí větví IF a ELSE, ale je zapotřebí napsat speciální testovací případ pro výjimku *catch*. Je tedy zapotřebí třech testovacích případů pro 100% pokrytí řádek/příkazů.

```
import static org.junit.Assert.*;  
import org.junit.Test;  
  
public class vhodnyPripadTest {  
  
    @Test  
    public void testPrevodNaKladneCislo_if() {  
        short pom = vhodnyPripad.prevodNaKladneCislo("123");  
    }  
}
```

```

        assertEquals(123, pom);
    }

    @Test
    public void testPrevodNaKladneCislo_else() {
        short pom = vhodnyPripad.prevodNaKladneCislo("-123");
        assertEquals(-1, pom);
    }

    @Test
    public void testPrevodNaKladneCislo_catch() {
        short pom = vhodnyPripad.prevodNaKladneCislo("abc");
        assertEquals(-1, pom);
    }
}

```

5.1.2. Nevhodný případ

Následující příklad ukazuje nevhodnost používání metriky pokrytí řádek, příkazů.

Vytvořím si jednoduchou testovanou třídu na testování výsledku součtu.

```

public class nevhodnyPripad {

    public static int secti2P(boolean podminka1, boolean podminka2) {
        int x = 0, y = 0;
        if (podminka1 == true && podminka2 == true) {
            x = 1;
            y = 2;
        }
        return x + y;
    }
}

```

Následně vytvoříme i testovací třídu.

```

public class nevhodnyPripadTest {

    @Test
    public final void testSecti2P_true() {
        int vysledek = nevhodnyPripad.secti2P(true, true);
        assertEquals(3, vysledek);
    }
}

```

K naměření 100% pokrytí stačí jeden testovací případ. Pokud nebude při testování spuštěna větev podmínky, ve které neexistuje příkaz/řádek, výsledné pokrytí bude pořád 100%.

5.1.3. Vyhodnocení

U jednoduchých metod lze dosáhnout velmi vysokého procenta pokrytí. Snadno i 100%, jako v nevhodném příkladu. Je nutné si ale uvědomit, že mezi pokrytím příkazů kódu a kvalitou kódu není žádná přímá úměra. Souvisí spolu pouze nepřímo. Pokrytí je příliš slabé pro robustní testy složitějších programových konstrukcí, snadno se přehlédnou chyby. Toto pokrytí je tedy nedostatečné a mělo by být doplněno jinou metrikou pokrytí.

5.2. Pokrytí hran/rozhodnutí

5.2.1. Vhodný případ

V podkapitole 3.2.1 je vysvětleno pokrytí hran, rozhodnutí. Vytvoříme si testovanou třídu *DelitelnostCisel*. V metodě *delitelnostTremi* je podmínka, ze které vedou dvě hrany. Pokud bude zadané číslo dělitelné třemi, program zpracuje tuto skutečnost, stejně tak pokud dělitelné třemi nebude.

```
public class DelitelnostCisel {  
  
    public boolean delitelnostTremi(int cislo) {  
        if (cislo % 3 == 0) {  
            System.out.println("" + cislo + " je delitelne tremi");  
            return true;  
        }  
        else {  
            System.out.println("" + cislo + " neni delitelne tremi");  
            return false;  
        }  
    }  
}
```

Je tedy zapotřebí dvou testovacích případů, abychom dosáhli pokrytí obou větví podmínky. Vytvoříme si tedy testovací třídu *DelitelnostCiselTest*, která otestuje obě větve.

```
import org.junit.Before;  
import org.junit.Test;  
import static org.junit.Assert.*;  
  
public class DelitelnostCiselTest {  
  
    private DelitelnostCisel delitelnost;  
  
    @Before  
    public void setUp() {  
        delitelnost = new DelitelnostCisel();  
    }  
}
```

```

    }

    @Test
    public void testDelitelnyTremi() {
        assertEquals("Chyba - 9 je dělitelné 3", true,
            delitelnost.delitelnostTremi(9));
    }

    @Test
    public void testNeniDelitelnyTremi() {
        assertEquals("Chyba - 7 není dělitelné 3", false,
            delitelnost.delitelnostTremi(7));
    }
}

```

První testovací případ nám pomocí čísla 9 otestoval dělitelnost třemi. Druhý nám zase otestoval *else* větev podmínky pro číslo, které dělitelné třemi není, tj. číslo 7. Testovací metody v testovací třídě nám 100% pokryly obě větve podmínky.

5.2.2. Nevhodný případ

Jak bylo řečeno v kapitole 3.2.1, toto pokrytí netestuje složené podmínky. Upravím testovanou třídu o složenou podmínku, v našem případě o další dělitelnost, dělitelnost dvěma.

```

public class DelitelnostCiselNevhodnyPripad {

    public boolean delitelnostTremi(int cislo) {
        if (cislo % 3 == 0 || cislo % 2 == 0) {
            System.out.println("" + cislo + " je delitelne tremi nebo
                dvema");
            return true;
        }
        else {
            System.out.println("" + cislo + " není delitelne tremi
                ani dvema");
            return false;
        }
    }
}

```

Při zanechání testovací třídy nám první testovací případ splní jen polovinu složené podmínky, dělitelnost třemi. Výsledek druhého testovacího případu zůstává stejný. I když není splněna druhá část podmínky, dělitelnost dvěma, pokrytí rozhodnutí přesto zůstane 100%.

5.2.3. Vyhodnocení

Tato technika je efektivní v případě jednoduchých podmínek. Na žádnou větev nezapomene. Bohužel dostatečně nepokrývá relační operátory použité v podmínkách a samotné složené podmínky. Špatně použití relačních operátorů v okolí okrajových hodnot je jednou z nejčastějších příčin chyb.

5.3. Pokrytí podmínek

5.3.1. Vhodný případ

Pokrytí podmínek vyhodnocuje, zda byly ve složených podmínkách testovány všechny části podmínky. Vezměme si jako ukázkou příklad založený na nevhodném případu testování pokrytí rozhodnutí. Na jeho základě vytvoříme novou testovanou třídu *SlozenaPodminka*.

```
public class SlozenaPodminka {  
  
    public boolean delitelnostTremiDvema(int cislo) {  
        if (cislo % 3 == 0 || cislo % 2 == 0) {  
            System.out.println("" + cislo + " je delitelne tremi nebo  
                dvema");  
            return true;  
        }  
        else {  
            System.out.println("" + cislo + " není delitelne tremi  
                nebo dvema");  
            return false;  
        }  
    }  
}
```

Při zachování testovací třídy a testovacích případů bude pokrytí podmínek činit pouze 75%.

The screenshot shows the Eclipse IDE with two tabs: SlozenaPodminka.java and SlozenaPodminkaTest.java. The code in SlozenaPodminka.java is as follows:

```

1 package codecover;
2
3 public class SlozenaPodminka {
4
5
6     public boolean delitelnostTremiDvema(int cislo) {
7
8         if (cislo % 3 == 0 || cislo % 2 == 0) {
9             System.out.println("" + cislo + " je delitelne tremi nebo dvema");
10            return true;
11        }
12        else {
13            System.out.println("" + cislo + " není delitelne tremi ani dvema");
14            return false;
15        }
16    }
17 }
18
19

```

The Coverage view at the bottom shows the following table:

Name	Statement	Branch	Loop	Term
ukazkyCodeCover	?	100,0 %	?	75,0 %
codecover	?	100,0 %	?	75,0 %
SlozenaPodminka	?	100,0 %	?	75,0 %

Obr. 6 - Rozdíl mezi pokrytím podmínek a rozhodnutí, znázorněný pluginem CodeCover ve vývojářském prostředí Eclipse

Pro 100% pokrytí musíme přidat jeden testovací případ. V tomto případě pro dělitelnost dvěma. Testovací třída bude vypadat následovně.

```

import static org.junit.Assert.*;

import org.junit.Before;
import org.junit.Test;

public class SlozenaPodminkaTest {
    private SlozenaPodminka podminka;

    @Before
    public void setUp() {
        podminka = new SlozenaPodminka();
    }

    @Test
    public void testDelitelnyTremi() {
        assertEquals("Chyba - 9 je dělitelné 3", true,
            podminka.delitelnostTremiDvema(9));
    }
}

```

```

@Test
public void testNeniDelitelnyTremiDvema() {
    assertEquals("Chyba - 7 není dělitelné 3 ani 2", false,
        podminka.delitelnostTremiDvema(7));
}

@Test
public void testDelitelnyDvema() {
    assertEquals("Chyba - 8 je dělitelné 2", true,
        podminka.delitelnostTremiDvema(8));
}
}

```

Pokryli jsme všechny možnosti podmínky testovacími případy. Pokrytí je tedy již 100%.

5.3.2. Vyhodnocení

Toto pokrytí dokáže zajistit otestování všech pod-podmínek za předpokladu zkráceného vyhodnocování logických výrazů. Toto vyhodnocování je v Javě obecné chování, které se nedá změnit.

Příklad:

```

if (x == 0 && y == 0) {
    System.out.println("cisla x,y jsou rovny 0");
}

```

V tomto příkladě je znázorněno, že má-li v logickém součinu první operand hodnotu false, druhý operand se nevyhodnocuje a hodnota celého výrazu je false.

V některých programovacích jazycích by bylo nutné použít více testů v případě úplného vyhodnocování logických výrazů.

5.4. Pokrytí cyklů

5.4.1. Vhodný případ

Jako příklad pro testování si zvolíme for-cyklus. Pro otestování pokrytí tohoto cyklu si vytvoříme testovanou třídu *PokrytiCyklu*, v jejíž metodě *cyklusFor* se právě takový cyklus bude nacházet.

```

public class PokrytiCyklu {

    public int cyklusFor(int pocetcyklu){

```



```

        int sum = 0;
        for(int i = 1; i <= pocetcyklu; i++){
            sum += i;
        }
        return sum;
    }
}

```

Jak je rozebráno v kapitole 3.1.5., musíme vytvořit tři testovací případy pro spuštění cyklu právě jednou, nula krát a více než jednou.

```

import static org.junit.Assert.*;

import org.junit.Before;
import org.junit.Test;

public class PokrytiCykluTest {

    private PokrytiCyklu cyklus;

    @Before
    public void setUp() {
        cyklus = new PokrytiCyklu();
    }

    @Test
    public void testPruchodJednou() {
        assertEquals(1, cyklus.cyklusFor(1));
    }

    @Test
    public void testBezPruchodu() {
        assertEquals(0, cyklus.cyklusFor(0));
    }

    @Test
    public void testPruchodVicekrat() {
        assertEquals(3, cyklus.cyklusFor(3));
    }
}

```

Testovací případy nám tedy ošetřily všechny možnosti pokrytí cyklu, které je nyní pokryto 100%.

5.4.2. Vyhodnocení

Pokrytí cyklu ohlídá všechny základní možnosti průchodu cyklem. Z hlediska jednoduchosti nejsou nevhodné případy použití této metriky známy.

5.5. Pokrytí ternálních operátorů

5.5.1. Vhodný případ

Ternální operátor je vlastně možnost, jak zkrátit podmínku pomocí operátoru pro podmíněný výraz.

Příklad podmínky:

```
if (podmínka) {  
    výraz1;  
}  
else {  
    výraz2;  
}
```

Lze zapsat jako:

```
podmínka ? výraz1 : výraz2;
```

Vyhodnotí se podmínka. Je-li podmínka nenulová (true), vyhodnotí se výraz1 a ten bude výsledkem celé operace. Je-li podmínka nulová (false), vyhodnotí se výraz2 a ten bude výsledkem celé operace.

Pro testování tohoto pokrytí si vytvoříme testovanou třídu *PokrytiQMO* na podobném principu.

```
public class PokrytiQMO {  
    public static final int MAX = 10;  
  
    public static boolean mensiNez(int hodnota) {  
        return (hodnota < MAX) ? true : false;  
    }  
}
```

Pro 100% pokrytí musíme v testovací třídě vytvořit dva testovací případy, které pokryjí obě větve podmínky.

```
import static org.junit.Assert.*;  
  
import org.junit.Test;  
  
public class pokrytiQMOTest {
```

```

@Test
public void testMensiNez_true() {
    boolean vysledek = pokrytiQMO.mensiNez(5);
    assertTrue("Chyba - je menší", vysledek);
}
@Test
public void testMensiNez_false() {
    boolean vysledek = pokrytiQMO.mensiNez(15);
    assertTrue("Chyba - je větší", vysledek);
}
}

```

První testovací případ pokrývá kladné vyhodnocení podmínky (pro číslo 5, které je menší než *MAX* číslo 10). Druhý nám zase pro vstupní číslo 15 vyhodnotí opak, tj. zápor pro druhou větví podmínky.

5.5.2. Vyhodnocení

Testování pokrytí ternárního operátoru je v podstatě shodné s testováním podmínek pro pokrytí rozhodnutí. Musí se vyhodnotit obě větve pro 100% pokrytí.

5.6. Pokrytí metod

5.6.1. Vhodný případ

Vytvoříme si testovanou třídu *PokrytiMetod*, obsahující tři metody. Metody *secti* a *odecti* vrací výsledek součtu a rozdílu čísel *x* a *y*. Metoda *vysledek* volá obě metody a vrací součet jejich výsledků

```

public class PokrytiMetod {

    public static int vysledek(int x, int y){
        int z = 0;
        z = secti(x,y) + odecti(x,y);
        return z;
    }

    public static int odecti(int x, int y) {

        return x - y;
    }

    public static int secti(int x, int y) {

        return x + y;
    }
}

```

Pokud tedy vytvoříme jeden testovací případ v testovací třídě pro otestování metody *vysledek*, mělo by být pokrytí všech metod 100%. Testovací případ otestuje metodu *vysledek* v testované třídě s čísly $x = 2$ a $y = 1$. Jako ověření výsledků skrz JUnit testy bude požadovat návratovou hodnotu 4 ($(2 + 1) + (2 - 1) = 4$).

```
import static org.junit.Assert.*;

import org.junit.Before;
import org.junit.Test;

public class PokrytiMetodTest {

    @Test
    public final void testSecti_true() {
        int vysledek = PokrytiMetod.vysledek(2,1);
        assertEquals(4, vysledek);
    }
}
```

5.6.2. Vyhodnocení

Jelikož tato metrika kontroluje, zda je alespoň jednou volána každá metoda, nelze najít nevhodný případ použití. Je užitečná, pokud chci zjistit, které části kódu by nešly pokrýt testy.

6. Zhodnocení dosažených výsledků

Testování proběhlo na PC s operačním systémem Windows 7 a Java Development Kit verze 1.7. Nástroje byly testovány ve vývojovém prostředí Eclipse a některé generování reportu proběhlo nástrojem Apache Ant z příkazové řádky.

6.1. CodeCover

6.1.1. Příprava nástroje

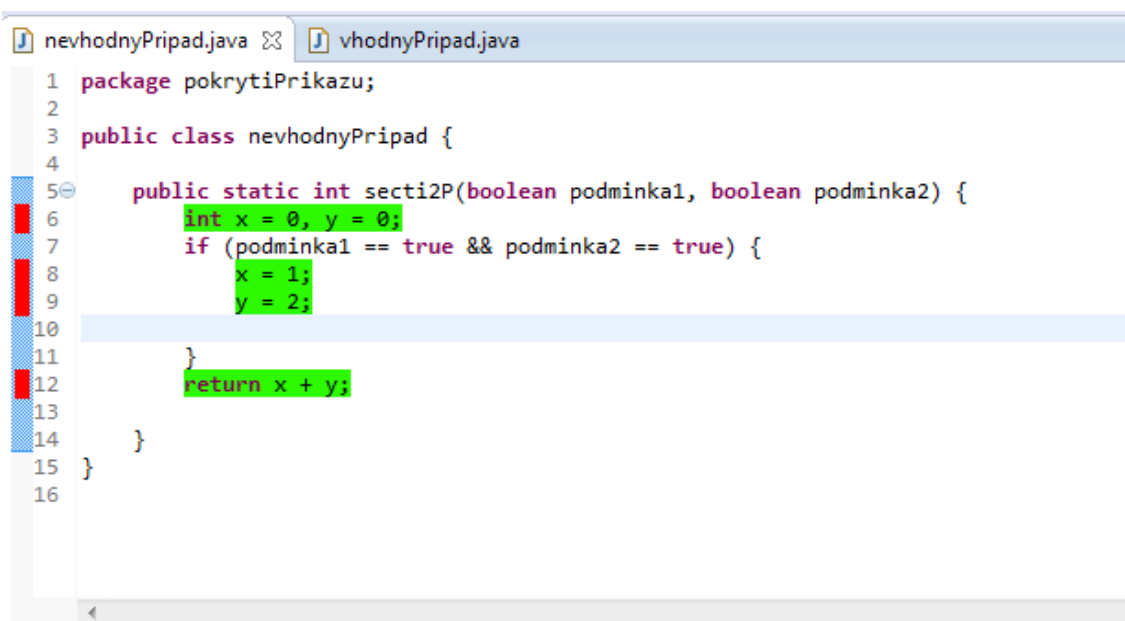
K testování byl použit plugin CodeCover verze 1.0.1.2 pro vývojové prostředí Eclipse v nejnovější verzi Eclipse Kepler. Instalace proběhla bez problémů přes aktualizací repozitář CodeCover.

6.1.2. Testování podporovaných pokrytí

Jak je psáno v kapitole 4.3.1.1. *Základní informace CodeCover*, CodeCover podporuje pokrytí příkazů, podmínek, rozhodnutí, QMO, cyklů a bloků synchronized.

- **Pokrytí příkazů**

Testování pokrytí nástrojem CodeCover proběhlo bez problémů přesně dle očekávaných výsledků.



```
1 package pokrytiPrikazu;
2
3 public class nevhodnyPripad {
4
5     public static int secti2P(boolean podminka1, boolean podminka2) {
6         int x = 0, y = 0;
7         if (podminka1 == true && podminka2 == true) {
8             x = 1;
9             y = 2;
10
11         }
12         return x + y;
13     }
14 }
15 }
16 }
```

Obr. 7 - Barevné znázornění pokrytí třídy *nevhodnyPripad.java*

```

1 package pokrytiPrikazu;
2
3 public class vhodnyPripad {
4
5     public static short prevodNaKladneCislo(String vstup) {
6         try {
7             short pom = Short.parseShort(vstup);
8             if (pom >= 0) {
9                 return pom;
10            }
11            else {
12                return -1;
13            }
14        } catch (NumberFormatException e) {
15            return -1;
16        }
17    }
18 }
19

```

Obr. 8 - Barevné znázornění pokrytí třídy vhodnyPripad.java

Name	Statement	Branch	Loop	Term	?-Operator	Synchronized
testovaniPokryti	100,0 %	?	?	?	?	?
pokrytiPrikazu	100,0 %	?	?	?	?	?
nevhodnyPripad	100,0 %	?	?	?	?	?
vhodnyPripad	100,0 %	?	?	?	?	?

Obr. 9 - Procentuální zobrazení výsledků pokrytí příkazů obou tříd

- **Pokrytí rozhodnutí**

Dle obrázku *Obr. 10* proběhly výsledky testování pokrytí rozhodnutí správně. Byly pokryty obě větve podmínky.

```

1 package pokrytiRozhodnuti;
2
3 public class DelitelnostCisel {
4
5     public boolean delitelnostTremi(int cislo) {
6         if (cislo % 3 == 0) {
7             System.out.println("" + cislo + " je delitelne tremi");
8             return true;
9         }
10        else {
11            System.out.println("" + cislo + " neni delitelne tremi");
12            return false;
13        }
14    }
15 }

```

Test Sessions Coverage Boolean Analyzer Pick Test Cases Correlation Problems

Show methods with Statement Coverage <= 90,5 %

Name	Statement	Branch	Loop	Term
testovaniPokryti	?	100,0 %	?	?
pokrytiRozhodnuti	?	100,0 %	?	?
DelitelnostCisel	?	100,0 %	?	?

Obr. 10 - Report výsledku pokrytí rozhodnutí nástrojem CodeCover

A na obrázku *Obr. 11* vidíme i správné vyhodnocení nevhodného případu pokrytí rozhodnutí. Pokrytí zůstalo i přes složenou podmínku 100%.

The screenshot shows a Java IDE with the following code in a file named `DelitelnostCiselNevhodnyPripad.java`:

```

1 package pokrytiRozhodnuti;
2
3 public class DelitelnostCiselNevhodnyPripad {
4
5     public boolean delitelnostTremi(int cislo) {
6         if (cislo % 3 == 0 || cislo % 2 == 0) {
7             System.out.println("" + cislo + " je delitelne tremi nebo dvema");
8             return true;
9         }
10        else {
11            System.out.println("" + cislo + " není delitelne tremi ani dvema");
12            return false;
13        }
14    }
15 }

```

Below the code editor, the CodeCover coverage report is displayed. It shows a tree view of the project structure and a table of coverage metrics.

Report settings: Show methods with **Statement Coverage** \leq **90,5 %**

Name	Statement	Branch	Loop	Term
testovaniPokryti	?	100,0 %	?	?
pokrytiRozhodnuti	?	100,0 %	?	?
DelitelnostCiselNevhodnyPripad	?	100,0 %	?	?

Obr. 11 - Report výsledku pokrytí rozhodnutí na nevhodném případě nástrojem CodeCover

- **Pokrytí podmínek**

Vyhodnocení složené podmínky 100% pokrytí podmínek na obrázku *Obr. 12*.


```

1 package pokrytiPodminek;
2
3 public class SlozenaPodminka {
4
5
6     public boolean delitelnostTremiDvema(int cislo) {
7
8         if (cislo % 3 == 0 || cislo % 2 == 0) {
9             System.out.println("" + cislo + " je delitelne tremi nebo dvema");
10            return true;
11        }
12        else {
13            System.out.println("" + cislo + " neni delitelne tremi ani dvema");
14            return false;
15        }
16    }

```

Test Sessions Coverage Boolean Analyzer Pick Test Cases Correlation Problems Co

Show methods with Statement Coverage <= 90,5 %

Name	Statement	Branch	Loop	Term
testovaniPokryti	?	?	?	100,0 %
pokrytiPodminek	?	?	?	100,0 %
SlozenaPodminka	?	?	?	100,0 %

Obr. 12 - Report výsledku pokrytí podmínek nástrojem CodeCover

- Pokrytí cyklů

```

1 package pokrytiCyklu;
2
3 public class PokrytiCyklu {
4
5
6     public int cyklusFor(int pocetcyklu){
7
8         int sum = 0;
9         for(int i = 1; i <= pocetcyklu; i++){
10            sum += i;
11        }
12        return sum;
13    }
14 }

```

Test Sessions Coverage Boolean Analyzer Pick Test Cases Correlation

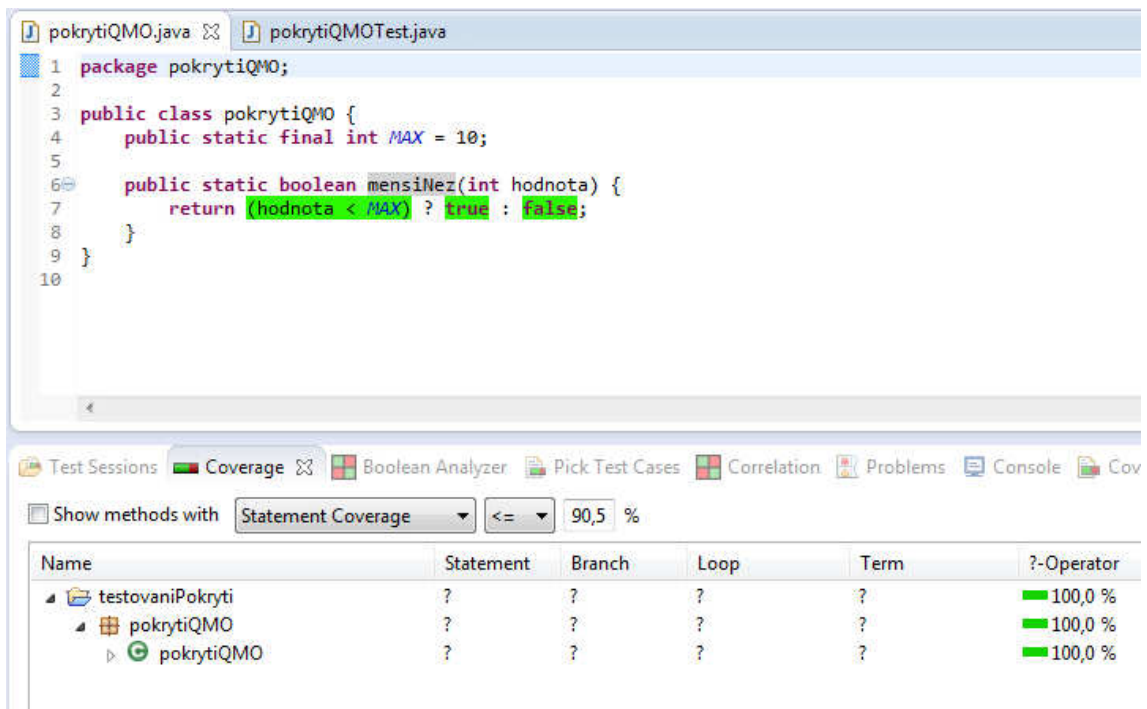
Show methods with Statement Coverage <= 90,5 %

Name	Statement	Branch	Loop
testovaniPokryti	?	?	100,0 %

Obr. 13 - Report výsledku pokrytí cyklů nástrojem CodeCover

Pokrytí cyklů tří testovacích případů proběhlo také správně.

- Pokrytí ternálních operátorů



Obr. 14 - Výsledek pokrytí ternálních operátorů nástrojem CodeCover

6.1.3. Generování souhrnného reportu

Generování reportu lze provést do dvou odlišných formátů. Buď se dá vygenerovat jako HTML nebo CSV.

Coverage Report
16.4.2014 16:56:48
measured on April 16, 2014 4:56:48 PM CEST

Description:

Name	Amount
default package	1
package	5
class	7
method	9

	Statement Coverage	Branch Coverage	Loop Coverage	?-Operator Coverage	Term Coverage
pokrytiRozhodnuti	7 / 7 100%	2 / 2 100%	0 / 0 ---	0 / 0 ---	2 / 2 100%
pokrytiQMO	2 / 2 100%	0 / 0 ---	0 / 0 ---	4 / 4 100%	0 / 0 ---
pokrytiPrikazu	8 / 8 100%	5 / 6 83%	0 / 0 ---	0 / 0 ---	4 / 6 66%
pokrytiPodminek	4 / 4 100%	2 / 2 100%	0 / 0 ---	0 / 0 ---	4 / 4 100%
pokrytiCyklu	3 / 3 100%	0 / 0 ---	3 / 3 100%	0 / 0 ---	2 / 2 100%

Obr. 15 - Vygenerovaný report HTML nástrojem CodeCover

Generovat report lze i do formátu CSV, což je souborový formát pro výměnu tabulkových dat, ve kterém jsou jednotlivé řádky či hodnoty odděleny znaky. Report CSV je součástí této práce jako soubor *ReportCodecoverCSV.csv*, který se dá otevřít v libovolném tabulkovém procesoru.

6.1.4. Vyhodnocení výsledků

Všechny testované metriky proběhly dle očekávaných výsledků. Výhodou nástroje CodeCover je vyhodnocení pro každou podporovanou metriku zvlášť přímo v Eclipse, obarvení řádků či nastavení jednotlivých tříd pro testování. Jako zvlášť užitečné funkce mohou označit analyzátor boolean nebo korelační matici. Celkově je tento nástroj velmi přehledný a snadno nastavitelný. Během instalace či testování neproběhly žádné problémy.

6.2. EclEmma

6.2.1. Příprava nástroje

K testování byl použit plugin EclEmma 2.3.0 pro vývojové prostředí Eclipse v nejnovější verzi Eclipse Kepler. Instalace proběhla bez problémů přes Eclipse Marketplace klienta.

6.2.2. Testování podporovaných pokrytí

EclEmma měří pokrytí příkazů/řádek, podmínek, metod/tříd a cyklomatickou složitost.

- **Pokrytí řádek/příkazů**

Spouštění testování pokrytí příkazů a řádek vyvolalo při testování ukázkového příkladu nečekanou chybu. Pro výsledek 100% pokrytí příkazů a řádek skrz JUnit testy tímto nástrojem je potřeba zavolat bezparametrový konstruktor pro vytvoření objektu do testovací třídy, i když není pro vlastní testování zapotřebí.

The screenshot shows an IDE with two tabs: 'vhodnyPripad.java' and 'vhodnyPripadTest.java'. The code in 'vhodnyPripad.java' is as follows:

```

1 package pokrytiPrikazu;
2
3 public class vhodnyPripad {
4
5     public static short prevodNaKladneCislo(String vstup) {
6         try {
7             short pom = Short.parseShort(vstup);
8             if (pom >= 0) {
9                 return pom;
10            }
11            else {
12                return -1;
13            }
14        } catch (NumberFormatException e) {
15            return -1;
16        }
17    }
18 }
19

```

The 'Coverage' tab shows the following report:

Element	Coverage	Covered Lines	Missed Lines	Total Lines
pokrytiPrikazu	55,2 %	16	13	29
vhodnyPripadTest.java	100,0 %	10	0	10
vhodnyPripad.java	85,7 %	6	1	7
vhodnyPripad	85,7 %	6	1	7
prevodNaKladneCislo(String)	100,0 %	6	0	6

Obr. 16 - Chyba zobrazení pokrytí řádek/příkazů nástroje Eclemma

Vytvoříme jej přidáním následujícího kódu do testovací třídy:

```

@Before
public void setUp() {
    vhodnyPripad pripad = new vhodnyPripad();
}

```

Na obrázku *Obr. 17* je již report výsledku dle očekávání správný.

```

1 package pokrytiPrikazu;
2
3 public class vhodnyPripad {
4
5     public static short prevodNaKladneCislo(String vstup) {
6         try {
7             short pom = Short.parseShort(vstup);
8             if (pom >= 0) {
9                 return pom;
10            }
11            else {
12                return -1;
13            }
14        } catch (NumberFormatException e) {
15            return -1;
16        }
17    }
18 }
19

```

Element	Coverage	Covered Lines	Missed Lines	Total Lines
pokrytiPrikazu	61,3 %	19	12	31
vhodnyPripad.java	100,0 %	7	0	7
vhodnyPripad	100,0 %	7	0	7
prevodNaKladneCislo(String)	100,0 %	6	0	6
vhodnyPripadTest.java	100,0 %	12	0	12

Obr. 17 - Správný report pokrytí řádek/příkazů nástroje EclEmma

Pro zobrazení správného výsledku nevhodného případu je potřeba opět vytvořit objekt testované třídy.

```

@Before
public void setUp() {
    nevhodnyPripad pripad = new nevhodnyPripad();
}

```

```

1 package pokrytiPrikazu;
2
3 public class nevhodnyPripad {
4
5     public static int secti2P(boolean podminka1, boolean podminka2) {
6         int x = 0, y = 0;
7         if (podminka1 == true && podminka2 == true) {
8             x = 1;
9             y = 2;
10        }
11        return x + y;
12    }
13 }
14
15 }
16

```

Element	Coverage	Covered Lines	Missed Lines	Total Lines
pokrytiPrikazu	38,7 %	12	19	31
nevhodnyPripad.java	100,0 %	6	0	6
nevhodnyPripad	100,0 %	6	0	6
secti2P(boolean, boolean)	100,0 %	5	0	5
nevhodnyPripadTest.java	100,0 %	6	0	6

Obr. 18 - Report pokrytí řádek/příkazů u nevhodného příkladu nástroje EclEmma

- **Pokrytí podmínek**

```

1 package pokrytiPodminek;
2
3 public class SlozenaPodminka {
4
5
6     public boolean delitelnostTremiDvema(int cislo) {
7
8         if (cislo % 3 == 0 || cislo % 2 == 0) {
9             System.out.println("" + cislo + " je delitelne tremi nebo dvema");
10            return true;
11        }
12        else {
13            System.out.println("" + cislo + " neni delitelne tremi ani dvema");
14            return false;
15        }
16    }
17 }
18
19

```

Element	Coverage	Covered Branches	Missed Branches	Total Branches
pokrytiPodminek	100,0 %	4	0	4
SlozenaPodminkaTest.java		0	0	0
SlozenaPodminka.java	100,0 %	4	0	4
SlozenaPodminka	100,0 %	4	0	4
delitelnostTremiDvema(int)	100,0 %	4	0	4

Obr. 19 - Report pokrytí podmínek nástroje EclEmma

Test pokrytí podmínek proběhl správně, všechny čtyři větve byly testy pokryty. Testování podmínek je zde vyobrazeno jako *branches* (větve). Tento název se může snadno zaměnit s pokrytím rozhodnutí, které bývá označováno i jako *branch coverage*.

- **Pokrytí metod/tříd**

Tato metrika nástroje EclEmma kontroluje také volání tříd. Opět se zde objevuje problém s vytvořením nepotřebného objektu, který je zapotřebí implementovat pro 100% pokrytí metod.

```

1 package pokrytiMetod;
2
3 public class PokrytiMetod {
4
5     public static int vysledek(int x, int y){
6         int z = 0;
7         z = secti(x,y) + odedcti(x,y);
8         return z;
9     }
10
11     public static int odedcti(int x, int y) {
12
13         return x - y;
14     }
15
16     public static int secti(int x, int y) {
17
18         return x + y;
19     }
20 }

```

Element	Coverage	Covered Methods	Missed Methods	Total Methods
pokrytiMetod	100,0 %	7	0	7
PokrytiMetod.java	100,0 %	4	0	4
PokrytiMetod	100,0 %	4	0	4
odedcti(int, int)	100,0 %	1	0	1
secti(int, int)	100,0 %	1	0	1
vysledek(int, int)	100,0 %	1	0	1
PokrytiMetodTest.java	100,0 %	3	0	3

Obr. 20 - Report pokrytí metod nástroje EclEmma

Element	Coverage	Covered Methods	Missed Methods	Total Methods
pokrytiPodminek	100,0 %	7	0	7
SlozenaPodminka.java	100,0 %	2	0	2
SlozenaPodminkaTest.java	100,0 %	5	0	5

- Show Projects
- Show Package Roots
- Show Packages
- Show Types
- Instruction Counters
- Branch Counters
- Line Counters
- Method Counters
- Type Counters
- Complexity
- Hide Unused Elements

Obr. 21 - Špatně umístěné tlačítko pro výběr metriky pokrytí

6.2.3. Generování souhrnného reportu

Generování reportu lze provést do HTML, XML nebo CSV. HTML export je celkem přehledný a lze poměrně detailně naměřené pokrytí proklikávat dle jednotlivých balíčků v projektech. U reportu do CSV je nutné každou session pokrytí v projektu spustit zvlášť, aby se do tohoto formátu vygenerovala.

Merged (16.4.2014 20:06:11) > testovaniPokrytiEmma > src Sessions

src

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
pokrytiPríkazu		100%		67%	2	15	0	31	0	12	0	4
pokrytiPodminek		100%		100%	0	9	0	15	0	7	0	2
pokrytiMetod		100%		n/a	0	7	0	12	0	7	0	2
Total	0 of 197	100%	2 of 10	80%	2	31	0	58	0	26	0	8

Merged (16.4.2014 20:06:11) Created with JaCoCo 0.7.0.201403182114

Obr. 22 - Vygenerovaný report HTML nástroje EclEmma

6.2.4. Vyhodnocení výsledků

Samotná instalace proběhla bez problémů. Problém se zbytečným vytvořením objektu u pokrytí řádek/příkazů a metod/tříd při použití JUnit testů, které jsou nesmírně důležité při testování pokrytí kódu, shledávám jako nedokonalost. Ačkoliv je třeba technika měření pokrytí řádek/příkazů nedostatečná, neměly by být výsledky jakkoliv ovlivněny vytvářením zbytečných objektů. Ostatní testy proběhly v pořádku.

Další nevýhoda nástroje EclEmma je zobrazení výsledků pokrytí. Výsledky pokrytí nejsou zobrazeny přehledně. Nelze nastavit samostatné zobrazení výsledků jednotlivých tříd a nastavení zobrazení jednotlivých pokrytí je umístěno nesmyslně na skrytém místě (viz obrázek *Obr. 21*). S tím samozřejmě souvisí generování souhrnného reportu.

6.3. Cobertura

6.3.1. Příprava nástroje

K testování byl použit plugin eCobertura ve verzi 0.9.8 pro Eclipse Juno a plugin Cobertura verze 2.0.3. pro vygenerování souhrnného reportu. Instalace do prostředí proběhla přes aktualizací repositář eCobertura.

6.3.2. Testování podporovaných pokrytí

Plugin pro Eclipse umí měřit pouze pokrytí řádek a pokrytí podmínek, což značně snižuje jeho případnou použitelnost. Cyklomatickou složitost umí měřit jen knihovna pro spuštění v příkazovém řádku či s programem Apache Ant nebo Maven.


```

package pokrytiPodminek;

public class SlozenaPodminka {

    public boolean delitelnostTremiDvema(int cislo) {

        if (cislo % 3 == 0 || cislo % 2 == 0) {
            System.out.println("" + cislo + " je delitelne tremi nebo dvema");
            return true;
        }
        else {
            System.out.println("" + cislo + " není delitelne tremi ani dvema");
            return false;
        }
    }
}

```

Obr. 23 - Barevné označení pokrytého kódu nástrojem eCobertura v Eclipse

Name	Lines	Total	%	Branches	Total	%
All Packages (2014-04-16 21:03:36)	46	46	100,00 %	8	10	80,00 %
pokrytiPodminek	15	15	100,00 %	4	4	100,00 %
SlozenaPodminka	6	6	100,00 %	4	4	100,00 %
SlozenaPodminkaTest	9	9	100,00 %	0	0	-
pokrytiPrikazu	31	31	100,00 %	4	6	66,67 %
nevhodnyPripad	6	6	100,00 %	2	4	50,00 %
nevhodnyPripadTest	6	6	100,00 %	0	0	-
vhodnyPripad	7	7	100,00 %	2	2	100,00 %
vhodnyPripadTest	12	12	100,00 %	0	0	-

Obr. 24 - Report měření pokrytí řádek a podmínek.

Pokrytí řádek i pokrytí podmínek proběhlo správně.

6.3.3. Generování souhrnného reportu

Pro generování reportu je zapotřebí samostatná knihovna. Nelze jej vyexportovat přes Eclipse. Pro testování tohoto exportu jsem použil nástroj Apache Ant ve verzi 1.6.3. Testování proběhlo na stejných ukázkových příkladech jako v případě použití s Eclipse. I se stejnými výsledky.

Package /	# Classes	Line Coverage	Branch Coverage	Complexity
All Packages	3	100% 15/15	80% 8/10	4
pokrytiPodminek	1	100% 6/6	100% 4/4	4
pokrytiPrikazu	2	100% 12/12	66% 4/6	4

Report generated by Cobertura 2.0.3 on 16.4.14 21:35.

Obr. 25 - Ukázka HTML reportu nástroje Cobertura spuštěním s nástrojem Ant

Oproti pluginu pro Eclipse umí Cobertura, spuštěná pomocí Ant, spočítat cyklomatickou složitost.

6.3.4. Vyhodnocení výsledků

Plugin pro Eclipse je zastaralý, nepodporuje novou verzi Kepler. Problémy verze pro Eclipse nastaly i s použitím Javy verze vyšší než 1.5, kde bylo zapotřebí přidat výchozí argument Java Virtual Machine *-XX:-UseSplitVerifier*. Tyto problémy při testování s knihovnou pro Apache Ant nenastaly.

Nástroj podporuje málo metrik, které jsou v pluginu nepřehledně zobrazeny. Stejně jako s nástrojem EclEmma nelze testovat jednotlivé třídy a k tomu zobrazit výsledky pro jednotlivé metriky. Plugin navíc nepodporuje export výsledku do reportu, tudíž jeho použitelnost klesá.

6.4. Vícekriteriální hodnocení

Vícekriteriální hodnocení variant neposkytuje jedno jediné řešení a výsledné řešení je ovlivněno volbou vah a použitými metodami. Váhy číselně odlišují jednotlivá kritéria z hlediska jejich významnosti. Čím je kritérium důležitější, tím větší váhu má přiděleno. [14]

Jako metodu stanovení volby vah jsem si vybral metodu bodovací. Tato metoda udává podstatnou informaci o preferencích jednotlivých kritérií. Každému kritériu přiřadím počet bodů podle preference (čím více bodů, tím je preference větší). Poté sečtu počet přidělených bodů a výsledné váhy získám dělením přidělených bodu jejich součtem.

Celkový počet kritérií k hodnocení nástrojů, které byly vybrány k testování a poté testovány, jsem shrnul do osmi bodů:

1. Podporované techniky měření pokrytí
2. Možnosti spouštění
3. Přehlednost reportu
4. Funkce navíc
5. Vhodnost použití nástroje na danou techniku
6. Integrace s JUnit
7. Podpora verze Eclipse
8. Oblíbenost nástroje

Jednotlivými testovanými nástroji jsou tedy následující:

1. CodeCover plugin pro Eclipse
2. EclEmma plugin pro Eclipse
3. Cobertura plugin pro Eclipse a knihovna pro Apache Ant

Váhy jednotlivých kritérií jsou hodnoceny na stupnici od 1 do 10. Od nejméně po nejvíce důležitou.

6.4.1. Podporované techniky měření pokrytí

Jako první z kritérií, které by mělo mít největší váhu, jsem vybral podporované techniky měření pokrytí. Podle počtu metrik, podporovaných nástrojem, bude mít odpovídající metrika výslednou váhu kritéria. Celková váha kritéria bude dána součtem těchto podporovaných metrik.

Následující tabulka ukazuje jaké a kolik jednotlivých metrik jednotlivé nástroje podporují. Z existujících metrik jsou do testování zahrnuty jen ty, které nástroje podporují.

Druhy pokrytí	CodeCover	EclEmma	Cobertura
Pokrytí řádek/příkazů	ano	ano	ano
Pokrytí rozhodnutí	ano	ne	ne
Pokrytí podmínek	ano	ano	ano
Pokrytí cyklů	ano	ne	ne
Pokrytí metod/tříd	ne	ano	ne
Pokrytí ternálních operátorů	ano	ne	ne
Cyklomatická složitost	ne	ano	ano
Počet technik pokrytí (váha)	5	4	3

Tab. 7 - Výsledné váhy kritéria podporovaných metrik

Nejvíce metrik podporuje nástroj CodeCover, nejméně zase Cobertura. Jediná nevýhoda nástroje CodeCover je v úvahu tohoto kritéria výpočet cyklomatické složitosti, který podporují oba ostatní nástroje. Tento výpočet by měl být tedy měřený nástrojem EclEmma, který se ve výsledku nachází v počtu měřitelných metrik na druhém místě.

Celková váha tohoto kritéria ohodnocuji číslem 8.

6.4.2. Možnosti spouštění

Každý z testovaných nástrojů má více možností spouštění.

Jako nejhlavnější z těchto možností je spouštění ve vývojovém nástroji Eclipse, který dává prostor programátorům, kteří nepotřebují sestavovat buildy, ale naopak potřebují mít okamžitý přehled o pokrytí. Každá možnost spouštění daným nástrojem bude mít hodnotu 1.

Možnosti spouštění	CodeCover	EclEmma	Cobertura
Plugin pro Eclipse	ano	ano	ano (jako eCobertura)
Apache Ant	ano	Ano (jako JaCoCo)	ano
Apache Maven	ano	Ano (jako JaCoCo)	ano
Příkazová řádka	ano	ne	ano
Výsledná váha	4	3	4

Tab. 8 - Výsledné váhy kritéria možnosti spouštění

Celkovou váhu tohoto kritéria ohodnocuji číslem 4.

6.4.3. Přehlednost reportu

Pro vyhodnocení tohoto kritéria je zapotřebí zahrnout všechny možnosti reportu. Hlavní by měly být: formát souhrnného reportu, přímé zobrazení v Eclipse, přímé zobrazení ve zdrojovém kódu v Eclipse, nastavení měření jednotlivých metrik, nastavení měření jednotlivých tříd. Každá tato možnost bude dle předchozího testování ohodnocena na stupni od 1 do 5 (menší znamená horší).

Přehlednost reportu	CodeCover	EclEmma	Cobertura
Formát souhrnného reportu	4	5	2
Přehlednost souhrnného reportu	4	3	3
Přímé zobrazení v Eclipse	5	3	1
Přímé zobrazení ve zdrojovém kódu v Eclipse	5	4	3

Nastavení měření jednotlivých metrik	5	2	1
Nastavení měření jednotlivých tříd	5	1	1
Výsledná váha	28	18	11

Tab. 9 - Výsledné váhy kritéria přehlednosti reportu

Celkové nejlepší výsledky přehlednosti či generování reportu dosahuje nástroj CodeCover. Naopak Cobertura kvůli nutnosti použít pro generování reportu knihovnu dopadla nejhůře.

Z hlediska důležitosti bude váha tohoto kritéria 10.

6.4.4. Funkce navíc

Jakákoliv funkce navíc, která nesouvisí přímo s měřením pokrytí, může být užitečná pro pochopení či přehlednost jejich výsledků. Někdy může také dosti zjednodušit práci samotného měření. Všechny funkce vybraných nástrojů jsou popsány v základních informacích v kapitole 4.3. .

CodeCover má funkce navíc následující: barevně značený kód, korelační matice, boolean analyzátor, živá oznámení, nastavitelná záložka coverage. Nástroje EclEmma a Cobertura mají jako funkci navíc bohužel jen barevně označený kód.

Pro vyhodnocení tohoto kritéria použijí počet nestandardních funkcí jako vah kritéria.

	CodeCover	EclEmma	Cobertura
Funkce navíc (váha)	5	1	1

Tab. 10 - Výsledné váhy kritéria funkcí navíc

Jelikož se nedá říci, že by toto kritérium přímo souviselo nebo nějak ovlivňovalo výsledky měření pokrytí, uděluji mu váhu 2.

6.4.5. Vhodnost použití nástroje na danou techniku

Dalším zajímavým kritériem z hlediska vyhodnocení testovaných nástrojů je vhodnost použití na danou techniku měření.

Celkem jsem vybral sedm technik měření pokrytí napříč podporou testovanými nástroji.

Každá metrika bude dle výsledků použití vyhodnocena od 1 do 5 (více je lépe). Do výsledků bude zahrnuty také problémy při testování. Nepodporované metriky nebudou hodnoceny.

Vhodnost použití	CodeCover	EclEmma	Cobertura
Pokrytí řádek/příkazů	5	2	2
Pokrytí rozhodnutí	5	-	-
Pokrytí podmínek	5	3	3
Pokrytí cyklů	5	-	-
Pokrytí metod/tříd	-	5	-
Pokrytí ternálních operátorů	5	-	-
Cyklomatická složitost	-	5	3
Počet technik pokrytí (váha)	25	15	8

Tab. 11 - Vhodnost použití nástroje na danou techniku

Váha tohoto kritéria bude 8.

6.4.6. Integrace s JUnit

Výběr vhodného nástroje pro testování souvisí s jeho integrací s JUnit. Během testování byly zjištěny drobné problémy s nástroji Cobertura a EclEmma. Plugin eCobertura má navíc problém s novými verzemi.

Na již ověřené stupnici od 1 do 5 (menší horší) ohodnotím bezproblémovost chodu a testování nástroje s JUnit.

	CodeCover	EclEmma	Cobertura
Integrace s JUnit	5	4	3

Tab. 12 - Vyhodnocení integrace s JUnit

Integrace s JUnit je při testování softwaru velmi důležitá, byl to jeden s hlavních předpokladů pro výběr nástroje. Váha tohoto kritéria bude 8.

6.4.7. Podpora verze Eclipse

CodeCover ani EclEmma neměly s Eclipse žádný problém. Kvůli neaktuálnosti nepodporuje plugin eCobertura novou verzi Eclipse Kepler. Navíc je zapotřebí tento plugin testovat se starší verzí JDK anebo přidat spouštěcí argument. Pro testování toto kritérium není nezbytně nutné, váhu mu přiděluji 4. Podporu jednotlivých nástrojů ohodnotím od 1 do 5 (menší horší).

	CodeCover	EclEmma	Cobertura
Podpora verze Eclipse	5	5	2

Tab. 13 - Vyhodnocení podpory verze Eclipse

6.4.8. Oblíbenost nástroje

Zkoumáním oblíbenosti a ohlasů nástrojů na internetu hodnotím na stupnici od 1 do 5 (větší lepší) v následující tabulce takto:

	CodeCover	EclEmma	Cobertura
Oblíbenost nástroje	4	5	3

Tab. 14 - Oblíbenost nástroje - vyhodnocení

Oblíbenost nástroje nebo jeho ohlasy nemají velký vliv. Váhu tohoto kritéria hodnotím číslem 4.

6.4.9. Výsledky vícekritériálního hodnocení

Jako výsledek všech předem stanovených kritérií nyní vyhodnotím nástroj, který dopadl nejlépe.

Určení kritérií dle vah je znázorněno v tabulce *Tab. 15*.

Pro výpočet váhy každého kritéria z hlediska vícekritériálního hodnocení je zapotřebí sečíst přidělené váhy a vydělit každou jednotlivou váhu tímto číslem.

	Dílčí váha	Celková váha	Váha v [%]
Podporované techniky měření pokrytí	8	8/48	16,67
Možnosti spouštění	4	4/48	8,33

Přehlednost reportu	10	10/48	20,83
Funkce navíc	2	2/48	4,17
Vhodnost použití nástroje na danou techniku	8	8/48	16,67
Integrace s JUnit	8	8/48	16,67
Podpora verze Eclipse	4	4/48	8,33
Oblíbenost nástroje	4	4/48	8,33
	48		100

Tab. 15 - Váhy jednotlivých kritérií

K jednotlivým nástrojům přiřadíme hodnoty výsledků.

	CodeCover	EclEmma	Cobertura
Podporované techniky měření pokrytí	5	4	3
Možnosti spouštění	4	3	4
Přehlednost reportu	28	18	11
Funkce navíc	5	1	1
Vhodnost použití nástroje na danou techniku	25	15	8
Integrace s JUnit	5	4	3
Podpora verze Eclipse	5	5	2
Oblíbenost nástroje	4	5	3

Tab. 16 - Ohodnocení nástrojů z hlediska jednotlivých kritérií

Podle váhy každého kritéria pronásobíme každou hodnotu z tabulky *Tab. 16*. Součtem všech hodnot ve sloupci daného nástroje získáme konečné číslo hodnocení každého nástroje (více bodů, lepší hodnocení).

	CodeCover	EclEmma	Cobertura	Celková váha
Podporované techniky měření pokrytí	0,83	0,67	0,50	8/48
Možnosti spouštění	0,33	0,25	0,33	4/48
Přehlednost reportu	5,83	3,75	2,29	10/48
Funkce navíc	0,21	0,04	0,04	2/48
Vhodnost použití nástroje na danou techniku	4,17	2,50	1,33	8/48
Integrace s JUnit	0,83	0,67	0,50	8/48
Podpora verze Eclipse	0,42	0,42	0,17	4/48
Oblíbenost nástroje	0,33	0,42	0,25	4/48
Celkové hodnocení nástroje	12,96	8,72	5,41	

Tab. 17 - Celkové hodnocení nástrojů

7. Závěr

V předkládané práci jsem řešil problematiku pokrytí kódu testy z mnoha různých aspektů. V počáteční (druhé) kapitole jsem obecně popsal problematiku testování software, ve třetí jsem rozebral všechny existující metriky pokrytí kódu. Ze čtvrté kapitoly, obsahující sled a popis dosud existujících nástrojů na měření pokrytí, jsem vybral tři nástroje, na kterých jsem následně prováděl testování měřitelných metrik. Vytvořil jsem příklady testů napsaných v jazyce Java (pátá kapitola), které jsem následně na jednotlivých nástrojích otestoval. V šesté kapitole jsem vytvořil vícekriteriální hodnocení a postupně popsal výsledky jednotlivých kritérií. Ve finální sedmé jsem vypracoval závěrečné shrnutí a zhodnotil dosažené výsledky

Existuje poměrně dost open source nástrojů na měření pokrytí kódu, ale bohužel jen hrstka z nich je natolik kvalitních či aktuálně vyvíjených, že dokážou bez problémů nějaká pokrytí otestovat. Ze třech vybraných nástrojů, splňující kritéria výběru pro testování, jednoznačně zvítězil nástroj CodeCover. Tabulka *Tab. 17* nám dokazuje, že ač se jednalo o množství podporovaných technik měření, vhodnost jejich použití, integrace s podpůrnými programy, přehledností reportu nebo funkcemi navíc, neměl tento nástroj konkurenci. Shledávám ho v dnešní době jako naprostou špičku mezi všemi open source nástroji.

Jako hlavní (a možná jedinou) nevýhodu oproti ostatním nástrojům vidím v neschopnost nástroje CodeCover vypočítat cyklomatickou složitost, která je horšími nástroji jednoznačně zvládnuta.

Práce sice byla zaměřena na poměrně specifické požadavky dané možnostmi předmětu KIV/OKS, ale díky navrženému a použitému multikriteriálnímu hodnocení může mít mnohem širší dopad. Tabulky multikriteriálního hodnocení - dostupné ve formátu tabulkového kalkulátoru Excel - si může případný zájemce relativně snadno (tj. po sérii svých experimentů s již připravenými ukázkami zdrojových kódů) doplnit hodnotami i dalších nástrojů. Další možností je změna váhových koeficientů, dle vlastních/firemních preferencí. Tak je možné s relativně nevelkým úsilím přidat do porovnání například i komerční nástroje v případě, že by se uvažovalo o jejich firemním využití.

Seznam použitých zdrojů

Tištěná literatura

- [2] GRUBB, Penny a TAKANG Armstrong. *Software Maintenance: Concepts and Practice (Second Edition)*. New Jersey: World Scientific Publishing, 2003. 372 s. ISBN: 978-98-123-8426-3.
- [3] MCCONNELL, Steve. *Dokonalý kód: Umění programování a techniky tvorby software*. Vyd. 1. Brno: Computer Press, 2005. 896 s. ISBN: 978-80-251-0849-9
- [6] PATTON, Ron. *Testování softwaru*. Vyd. 1. Praha: Computer Press, 2002. 313 s. ISBN: 80-7226-636-5.
- [7] ROUDENSKÝ Petr a HAVLÍČKOVÁ Anna. *Řízení kvality softwaru – průvodce testováním*. Brno: Computer Press, 2013. 208 s. ISBN: 978-80-251-3816-8.
- [8] AMMANN Paul a OFFUTT Jeff. *Introduction to Software Testing*. Vyd. 1. Cambridge University Press, 2008. 344 s. ISBN: 978-05-218-8038-1.
- [11] PAGE, Alan a kolektiv. *Jak testuje software Microsoft*. Vyd. 1. Brno: Computer Press, 2009. ISBN: 978-80-251-2869-5.

Ostatní zdroje

- [1] BEHENBEIM, Ronald. *Software testing tutorial* [online]. 28. října 2012 [cit. 2014-1-28]. Dostupné z: http://actoolkit.unprme.org/wp-content/resourcepdf/software_testing.pdf
- [4] HAVLÍČKOVÁ, Anna. *Proč bychom měli testovat* [online]. 3. prosince 2009 [cit. 2014-2-4]. Dostupné z: <http://testovanisoftwaru.blogspot.cz/2009/12/proc-bychom-meli-testovat.html>
- [5] BOROVCOVÁ, Anna. *Testování webových aplikací : Část II: Základy testování* [online]. Srpen 2008 [cit. 2014-2-4]. Dostupné z: http://www.poeta.cz/Zaklady_testovani.pdf
- [9] HAVLÍČKOVÁ, Anna. *Problémy s pokrytím kódu* [online]. Říjen 2008 [cit. 2014-3-3]. Dostupné z: <http://testovanisoftwaru.blogspot.cz/2009/10/problemy-s-pokrytim-kodu.html>
- [10] CORNETT, Steve. *Code Coverage Analysis* [online]. 2008 [cit. 2014-3-10]. Dostupné z: <http://www.bullseye.com/coverage.html>
- [12] *Open Source Code Coverage Tools in Java*. 2011 [cit. 2014-3-18]. Dostupné z: <http://java-source.net/open-source/code-coverage>

- [13] *Cobertura Wiki FAQ*. Červenec 2013 [cit. 2014-3-21]. Dostupné z:
<https://github.com/cobertura/cobertura/wiki/FAQ>
- [14] KLICNAROVA, Jana. *Předmět Operační analýza - Vícekriteriální hodnocení*. JČU, Ekonomická fakulta, Katedra aplikované matematiky a informatiky, 25. března 2011 [cit. 2014-4-20]. Dostupné z:
http://home.ef.jcu.cz/~janaklic/oa_zsf/VHV_I.pdf
- [15] SHAHID Muhammad a SUHAIMI Ibrahim. *An Evaluation of Test Coverage Tools*. Advanced Informatics School (AIS), Universiti Teknologi Malaysia 2011 [cit. 2014-4-18]. Dostupné z:
<http://www.ipcsit.net/vol5/39-ICCCM2011-A10014.pdf>

Terminologický slovník

Termín	Zkratka	Význam
Framework		Soubor technologií určených pro vývoj aplikací.
HyperText Markup Language	HTML	Značkovací jazyk pro dokumenty publikovatelné na internetu.
JUnit		Framework pro jednotkové testy psaný v programovacím jazyce Java.
Jednotkové testy		ověřování správné funkčnosti dílčích částí neboli jednotek zdrojového kódu.
Java Virtual Machine	JVM	Sada počítačových programů, která využívá modul virtuálního stroje ke spuštění dalších programů v jazyce Java.
Java Virtual Machine Debug Interface	JVMDI	Ladění rozhraní JVM.
Apache Ant		Nástroj pro sestavování softwarových aplikací.
Apache Maven		Nástroj pro správu, řízení a automatizaci buildů aplikací.
Extensible Markup Language	XML	Rozšiřitelný značkovací jazyk.
Comma-separated values	CSV	Jednoduchý souborový formát určený pro výměnu tabulkových dat.
Eclipse		open source IDE
Eclipse Marketplace		Klientské řešení pro instalaci pluginů přímo v Eclipse.
Integrated Development Environment	IDE	Vývojové prostředí

Seznam obrázků

Obr. 1 – Příčiny chyb.....	12
Obr. 2 – Náklady na opravu chyb v závislosti na fázi vývoje software.....	12
Obr. 3 – Graf metriky pokrytí hran.....	18
Obr. 4 - Příklad znázornění boolean analyzátoru nástroje CodeCover.....	30
Obr. 5 - Příklad znázornění korelační matice pluginem CodeCover v prostředí Eclipse.	31
Obr. 6 - Rozdíl mezi pokrytím podmínek a rozhodnutí, znázorněný pluginem CodeCover ve vývojářském prostředí Eclipse.....	39
Obr. 7 - Barevné znázornění pokrytí třídy nevhodnyPripad.java.....	45
Obr. 8 - Barevné znázornění pokrytí třídy vhodnyPripad.java.....	46
Obr. 9 - Procentuální zobrazení výsledků pokrytí příkazů obou tříd.....	46
Obr. 10 - Report výsledku pokrytí rozhodnutí nástrojem CodeCover.....	47
Obr. 11 - Report výsledku pokrytí rozhodnutí na nevhodném případě nástrojem CodeCover.....	48
Obr. 12 - Report výsledku pokrytí podmínek nástrojem CodeCover.....	49
Obr. 13 - Report výsledku pokrytí cyklů nástrojem CodeCover.....	50
Obr. 14 - Výsledek pokrytí ternálních operátorů nástrojem CodeCover.....	50
Obr. 15 - Vygenerovaný report HTML nástrojem CodeCover.....	50
Obr. 16 - Chyba zobrazení pokrytí řádek/příkazů nástroje EclEmma.....	52
Obr. 17 - Správný report pokrytí řádek/příkazů nástroje EclEmma.....	53
Obr. 18 - Report pokrytí řádek/příkazů u nevhodného příkladu nástroje EclEmma.....	54
Obr. 19 - Report pokrytí podmínek nástroje EclEmma.....	54
Obr. 20 - Report pokrytí metod nástroje EclEmma.....	55
Obr. 21 - Špatně umístěné tlačítko pro výběr metriky pokrytí.....	55
Obr. 22 - Vygenerovaný report HTML nástroje EclEmma.....	56
Obr. 23 - Barevné označení pokrytého kódu nástrojem eCobertura v Eclipse.....	57
Obr. 24 - Report měření pokrytí řádek a podmínek.....	57
Obr. 25 - Ukázka HTML reportu nástroje Cobertura spuštěním s nástrojem Ant.....	57

Seznam tabulek

Tab. 1 - Existující komerční nástroje pro zjištění pokrytí kódu testy.....	25
Tab. 2 - Existující open-source nástroje pro zjištění pokrytí kódu testy.....	26
Tab. 3 - Základní informace o nástroji CodeCover.....	29
Tab. 4 - Korelační matice.....	30
Tab. 5 - Základní informace o nástroji Cobertura.....	31
Tab. 6 - Základní informace o nástroji EclEmma.....	32
Tab. 7 - Výsledné váhy kritéria podporovaných metrik.....	59
Tab. 8 - Výsledné váhy kritéria možnosti spouštění.....	60
Tab. 9 - Výsledné váhy kritéria přehlednosti reportu.....	61
Tab. 10 - Výsledné váhy kritéria funkcí navíc.....	61
Tab. 11 - Vhodnost použití nástroje na danou techniku.....	62
Tab. 12 - Vyhodnocení integrace s JUnit.....	62
Tab. 13 - Vyhodnocení podpory verze Eclipse.....	63
Tab. 14 - Oblíbenost nástroje - vyhodnocení.....	63
Tab. 15 - Váhy jednotlivých kritérií.....	64
Tab. 16 - Ohodnocení nástrojů z hlediska jednotlivých kritérií.....	64
Tab. 17 - Celkové hodnocení nástrojů.....	65