Slovenská technická univerzita v Bratislave
Fakulta informatiky a informačných technológií

Peter Beňuš

Automatické testovanie softvéru

Bakalársky projekt 1

Študijný program: Informatika
Vedúci bakalárskeho projektu: Ing. Karol Rástočný
December 2015

Slovenská technická univerzita v Bratislave
FAKULTA INFORMATIKY A INFORMAČNÝCH TECHNOLÓGIÍ

Anotácia

Študijný program: Informatika
Autor: Peter Beňuš
Názov bakalárskej práce: Automatické testovanie softvéru
Vedúci bakalárskej práce: Ing. Karol Rástočný
december 2015

Práca sa venuje jednotkovému testovaniu softvéru. Obsahuje analýzu vlastností jednotkového testovania, z ktorých vyplývajú vlastnosti na frameworky určené na jednotkové testovanie. Na základe vlastností jednotkového testovania sú odvodené vlastnosti ideálneho testovacieho frameworku na jednotkové testovanie. Vybraných je niekoľko frameworkov pre jazyk C# a Java a všetky tieto frameworky patria do skupiny xUnit z čoho vyplýva, že majú viacero spoločných vlastností, ale zároveň používajú niekoľko rôznych spôsobov testovania. Každý testovací framework prináša zo sebou nejaké výhody aj nevýhody oproti ostatným softvérom. Preto sú tieto frameworky porovnané s ideálny frameorkom a je vybraný jeden pre jazyk C# a jeden pre jazyk Java, ktorý sa najviac podobá ideálnemu.

Slovak University of Technology Bratislava
FACULTY OF INFORMATICS AND INFORMATION TECHNOLOGIES

Annotation

Degree Course: Informatics
Author: Peter Beňuš
Title of bachelor thesis: Automatic Software Testing
Supervisor: Ing. Karol Rástočný
december 2015
This work devotes to unit testing of software. It includes analysis of unit testing characteristics from which emerge features of unit testing frameworks. Features of ideal unit testing framework are derived from unit testing characteristics. Several of unit testing frameworks for C# and Java was chosen and all of these frameworks, belong to the xUnit group, so they have common features, but also they used different ways how to test units. Every of this testing frameworks have prons and cons compared to the others. Because of this, framework are compared with ideal framework. One for C# and one for Java which is most similar to the ideal is chosen.

Obsah

1 Úvod

Testovanie softvéru v súčastnosti naberá na dôležitosti a prikladá sa mu čoraz väčšia váha počas tvorby softvéru. Napriek tomu, že pri súčasnej komplexnosti rôznych programov nedokážeme zabezpečiť úplnú absenciu chýb má veľký význam sa snažiť tieto chyby minimalizovať.

Testovanie môže prebiehať (a často aj prebieha) počas celej doby životného cyklu programu. Čím skôr je chyba odhalená a opravená tým menšie negatívne následky bude mať na výsledný produkt. Preto je dôležité testovať softvér od začiatku tvorby, od najmenších jednotiek, cez integráciu viacerých jednotiek a komponentov až po celé systémy, ktoré môžu byť zložené z viacerých programov.

Preto, že je dôležité odhaliť chybu čo najskôr a medzi pracovnými rámcami nie je doteraz vytvorené žiadne porovnanie, tak sme sa rozhodli porovnať viacero pracovných rámcov na jednotkové testovanie a vytvoriť ich porovnanie čo najobjektívnejšie a nezohľadňovať len subjektívne pocity, ktoré vznikajú pri práci s každým pracovným rámcom.

Tak ako sa v rôznych častiach životného cyklu softvéru testujú rôzne aspekty produktu, používajú sa aj rôzne prístupy testovania a softvér testujú rôzny ľudia. Jednotkové testovanie vykonávajú typicky programátori a testujú jednotlivé jednotky nezávisle od seba. Integračné a systémové testovanie majú na starosti testeri a ich úlohou je odhaliť chyby v komunikácií medzi komponentami, resp. programom a konkrétnym systémom. Akceptačné testovanie vykonáva už objednávateľ alebo cieľová skupina používateľov a jeho výsledkom je akceptovanie výsledného produktu, resp. neakceptovanie a nutnosť opravy [8].

Jednotkové testovanie je typ testovania, ktorým sa musel zaoberať každý, kto už niečo programoval, pretože vždy začíname tým, že vytvoríme nejakú jednotku programu a postupne sa k tomu pridávajú ďalšie. Dokonca je tento postup úplne nezávislý od toho či sa iba učíme programovať alebo už programovať vieme a vytvárame nejaký program na konkrétny účel. Výrazný rozdiel, ale môže byť v tom, že začiatočníci budú testovať svoje jednotky programu opakovaným spúšťaním, zadávaním rôznych vstupov a sledovaním výsledkov, ale skúsený programátor použije nejaký pracovný rámec, ktorý dokáže otestovať to čo treba za neho a upozorniť ho keď sa niekde nájde chyba.

Pracovný rámec, ktorý by dokázal automaticky vykonávať jednotkové testovanie by bol preto veľkým prínosom pre programátorov, pretože by im zrýchlil a zjednodušil prácu.

2 Testovanie softvéru

Testovanie softvéru je empirická činnosť, ktorá skúma kvalitu testovaného produktu alebo služby vykonávaná na podanie informácií o kvalite všetkým zainteresovaným osobám [6]. V súčastnosti existuje veľa spôsobov testovania a veľa častí životného cyklu softvéru, v ktorých sa aplikujú iné typy testov. Testovanie softvéru môžeme rozdeliť na kategórie podľa postupu, ktorý sa používa pri testovaní, podľa spôsobu testovania a podľa úrovne testu.

2.1 Podľa postupu testovania

2.1.1 Testovanie formou čiernej skrinky

Testuje funkcionalitu bez informácií o tom ako je softvér implementovaný. Tester dostane iba informácie o tom, aký by mal byť výsledok testu po zadaní vstupných dát a kontroluje výstup softvéru či sú výstupné dáta totožné s očakávanými [5]. Test je konštruovaný z funkcionálnych vlastností, ktoré sú špecifikované požiadavkách na program [7]. Výhodou tohto typu testovania je, že tester nie je ovplyvnený štruktúrou zdrojového kódu a tým môže odhaliť chyby aj tam, kde to programátor nehľadal lebo to považoval za správne pri pohľade na zdrojový kód, ale bola tam chyba, ktorá zo zdrojového kódu nemusí byť viditeľná (napríklad nesprávne, resp. nedostatočné ošetrenie nekorektných vstupov). Hlavnou výhodou je, že tester nemusí poznať zdrojový kód a preto môže oveľa rýchlejšie vytvoriť testy. Nevýhodou je úroveň otestovania systému, pretože tvorca testov nevie ako program funguje, a preto veľmi pravdepodobne nebude schopný vytvoriť test, ktorý by testoval všetky vetvy programu. Používa sa pri jednotkovom, integračnom, systémovom a akceptačnom testovaní. Okrem použitia pri rôznych úrovniach testov sa využíva aj na validáciu softvéru [5].

2.1.2 Testovanie formou bielej skrinky

Pri tomto spôsobe testovania je testerovi známa vnútorná štruktúra softvéru a aj konkrétna implementácia. Test sa tvorí tak, aby bola otestovaná každá vetva zdrojového kódu [5]. Používa sa pri jednotkovom testovaní na skoré odhalenie čo najväčšieho množstva chýb, pri integračnom testovaní na testovanie správnej spolupráce rôznych jednotiek programu a aj pri regresnom testovaní, kde sa používajú recyklované testovacie prípady z integračného a jednotkového testovania a taktiež slúži na verifikáciu. Výhodou je schopnosť otestovať komplexne všetky vetvy, ktoré program na danej úrovni testu vykonáva, ale nevýhodou je, že tester musí mať dobré vedomosti o zdrojovom kóde a v niektorých prípadoch tvorenia testov môže byť znalosť zdrojového kódu nevýhoda.

2.1.3 Testovanie formou sivej skrinky

Spôsob testovania, pri ktorom je známy zdrojový kód (nemusí byť sprístupnený úplne celý), ale testy sa vykonávajú rovnako ako pri testovaní formou čiernej skrinky. Používa sa napríklad pri integračnom testovaní ak máme dva moduly od rôznych vývojárov a odkryté sú len rozhrania [5]. Poskytuje výhody obidvoch predchádzajúcich prístupov, ale má oproti nim aj nejaké nevýhody. Oproti testovaniu čiernou skrinkou má výhodu v lepšom pokrytí rôznych vetiev zdrojového kódu, ale je časovo náročnejšie na tvorbu testov. Oproti testovaniu bielou

skrinkou je menej náročné na znalosť zdrojového kódu, pretože ho nemá prístupný celý, ale nepokrýva všetky vetvy programu a preto je menej komplexné.

2.2 Podľa spôsobu testovania

2.2.1 Statické testovanie

Statické testovanie je často implicitné. Zahŕňa napríklad kontrolu zdrojového kódu programátorom jeho čítaním hneď po napísaní, kontrolu štruktúry a syntaxe kódu nástrojom alebo editorom, v ktorom sa zdrojový kód píše. Program nie je potrebné spúšťať, ale analýza zdrojového kódu založená na upravovacích pravidlách zistí v zdrojovom kóde rôzne možné chyby, ktoré sa zvyčajne objavujú v spravovaní pamäte, neinicializovaných premenných, výnimke nulového smerníku, porušení prístupu k poľu a taktiež pretečení vyrovnávacej pamäti [12].

Veľmi dôležitou súčasťou statického testovania je posudzovanie zdrojového kódu (angl. codereview). Je to posudzovanie zdrojového kódu iným vývojárom za účelom dosiahnutia čo najvyššej kvality. Formálny variant tohto posudzovania bol veľmi dlho efektívnym kvalitatívnym prínosom pri tvorbe softvéru, ale za relatívne vysokú cenu. Preto sa v poslednej dobe začala používať odľahčené a neformálne posudzovanie založené na nástrojoch. Táto forma sa na nazýva moderné posudzovanie softvéru (angl. modern code review). Tieto nástroje sú kolaboratívne a umožňujú všetkým spoluprácovníkom vidieť stav každého, jeho komentáre a značky pri zdrojovom kóde a pod. [2]

Výhodou je, že v súčasnosti je už vo väčšine vývojových prostredí statické testovanie automatické. Je rýchle a dokáže odhaliť niektoré chyby v zdrojovom kóde aj pred spustením programu (napr. použitie neinicializovaných premenných). Okrem toho ešte dobré statické testovanie môže odhaliť logické chyby v zdrojovom kóde, ktoré by sa inak odhalili až pri dynamickom testovaní. Staticky môže testovať aj neúplný a ešte nespustiteľný program. Nevýhodou je, že nedokáže zistiť, či sa počas behu vyskytne nejaká chyba, pretože prebieha len na základe zdrojového kódu a pri testovaní sa program nespúšťa.

2.2.2 Dynamické testovanie

Dynamické testovanie prebiaha už na spustiteľnom programe. Program nemusí byť ešte kompletný, ale podmienkou je, že musí byť skompilovateľný a spustiteľný na stroji, na ktorom chceme testovať. Do dynamického testovania spadá buď testovanie formou čiernej skrinky (spustíme program, dáme mu potrebné vstupné dáta a skontrolujeme čo je na výstupe) alebo aj formou bielej skrinky pri **(debuggingu)**. Pri debuggingu môžeme sledovať správanie programu a hodnoty premenných na každom riadku zdrojového kódu. K dynamickému testovaniu sa viaže validácia.

Výhodou dynamického testovania je, že vieme zistiť rýchlo a jednodycho ako sa program naozaj správa, keď je už spustený a odhaliť tak chyby, ktoré nemusia byť alebo sú len veľmi ťažko viditeľné zo zdrojového kódu. Nevýhodou je, že niektoré funkcie programov sú závislé od konfigurácie stroja a my vieme jednoznačne otestovať funkčnosť len na stroji, ktorý máme dostupný. Keďže testovať program na všetkých možných konfiguráciach je nereálne vznikajú v praxi problémy s kompatibilitou.

2.3 Podľa úrovne testu

2.3.1 Jednotkové testovanie

Jednotkové testovanie je metóda testovania softvéru, pri ktorej sa testujú individuálne komponenty (jednotky) zdrojového kódu. Zvyčajne nie je testovacou fázou v zmysle nejakého obdobia na tvorbe projektu, ale skôr je to posledný krok písania časti zdrojového kódu [1]. Programátori takmer vykonávajú jednotkové testovanie takmer stále, či už pri testovaní vlastného zdrojového kódu alebo kódu iného programátora [1]. Kvalitné testovanie na tejto úrovni môže výrazne znížiť cenu a čas potrebný na vývoj celého softvéru [5].

2.3.2 Testovanie komponentov

Počas testovania komponentov sa testeri zameriavajú na chyby v ucelených častiach systému. Vykonávanie testu zvyčajne začína, keď je už prvý komponent funkčný spolu so všetkým potrebným (napr. ovládače) na fungovanie tohto komponentu bez zbytku systému [1].

Testovanie komponentov má sklon viezť k štrukturálnemu testovaniu alebo testovaniu formou bielej skrinky. Ak je komponent nezávislý môže sa použiť aj testovanie formou čiernej skrinky [1].

2.3.3 Integračné testovanie

V integračnom testovaní sa testeri zameriavajú na hľadanie chýb vo vzťahoch a rozhraniach medzi pármi a skupinami komponentov. Integračné testovanie musí byť koordinované, aby sa správna množina komponentov spojila správnym spôsobom a v správnom čase pre najskoršie možné odhalenie integračných chýb [1].

Niektoré projekty nepotrebujú formálnu fázu integračného testovania. Ak je projekt množinou nezávislých aplikácií, ktoré nezdieľajú dáta alebo sa nespúšťajú navzájom, môže byť táto fáza preskočená [1].

2.3.4 Systémové testovanie

Systémove testovanie je vykonávané na úplnom a integrovanom systéme za účelom vyhodnotenia súladu systému z jeho špecifikovanými požiadavkami [4]. Niekedy, napríklad pri testovaní inštalácie a použiteľnosti, sa na tieto testy pozerajú na systém z pohľadu zákazníka alebo koncového používateľa. Inokedy sú testy zdôrazňujú konkrétne aspekty, ktoré môžu byť nepovšimnuté používateľom, ale kritické pre správne fungovanie systému [1].

2.3.5 Akceptačné testovanie

Akceptačné testovanie je formálne testovanie zamerané na potreby používateľa, požiadavky a biznis procesy vedúce k rozhodnutiu či systému vyhovuje alebo nevyhovuje akceptačným kritériám a umožniť používateľovi, zákazníkovi alebo inému splnomocnenému subjektu či má alebo nemá byť systém akceptovaný [11]. Narozdiel od predchádzajúcich foriem testovania, akceptačné testovanie demonštruje, že systém spĺňa požiadavky [1].

V komerčnej sfére sú niekedy tieto testy nazývané aj podľa toho kým sú vykonávané älfa testy"(používateľmi vo firme) alebo "beta testy"(súčasnými alebo potenciálnymi zákazníkmi v prevádzke) [1].

3 Kľúčové vlastnosti jednotkového testovania

Per Runeson, profesor na univerzite v Švédskom Lunde robil prieskum medzi 50 firmami, ktorých hlavným produktom je softvér [10]. Firmy boli rôznej veľkosti od firiem tvorených jedným človekom až po firmy so stovkami zamestnancov a taktiež aj rôzneho cieľového odboru, v ktorom sa ich softvér používa. Cieľom tohto prieskumu bolo zistiť, kde sú silné stránky firiem v používaní jednotkového testovania a čo podľa nich jednotkové testovanie zahŕňa. Na základe tohto môžeme odvodiť najdôležitejšie vlastnosti jednotkového testovania využívané v praxi a následne podľa nich porovnať rôzne nástroje umožňujúce jednotkové testovanie.

Jednotkové testovanie je podľa prieskumu testovanie najmenších samostatných jednotiek s vnútornými/vonkajšími parametrami. Takisto sa účastníci prieskumu zhodli na tom, že testovanie sa zameriava na samostatné funkcie avšak už v tom či má byť vykonávané samostatne do zbytku systému sa nezhodli.

Testy by mali byť založené na štruktúre programu (to znamená testovanie formou bielej alebo sivej skrinky), vykonávané automaticky a vedené vývojármi, ktorí zároveň určujú ako by mali byť vykonávané. Toto neplatí pre testom riadený vývoj, ale tým sa nezaoberáme, pretože v ňom sa testy píšu ešte pred písaním zdrojového kódu a práca sa zameriava na testovanie už napísaných jednotiek. Silný nesúhlas bol s tým, že by malo viesť jednotkové testy oddelenie testovania alebo kvality. Špecifikované by mali byť v testovacom kóde a nemali by byť špecifikované v texte.

Pri otázke ako často by mali byť vykonávané sa názory dosť líšili a podľa výsledkov si väčšina myslí, že by mali byť vykonávané niekoľkokrát denne a po každej kompilácii. U väčšiny firiem, ktoré sa zúčastnili prieskumu vykonávanie všetkých jednotkových testov trvá niekoľko minút.

Vo väčšine firiem sú jednotkové testy vykonávané aby sa vývojári presvedčili, že daná jednotka vykonáva to čo od nej očakávali a vo všeobecnosti jednotkové testovanie zvyšuje kvalitu výsledného produktu. Neslúžia na akceptovanie jednotiek a nezvyknú byť požiadavkou klientov.

Medzi silné stránky jednotkového testovania zaradili účastníci prieskumu to, že jednotkové testy dobre identifikujú jednotky a dobre sa udržuje ich testovací kód. Dobre špecifikujú testovacie prípady a sú vykonávané automaticky. Ďalšou výhodou je množstvo pracovných rámcov a dobrá integrácia s hotovými systémami.

Slabou stránkou je určite testovanie grafického používateľského rozhrania. Za slabé bolo označené tiež pokrytie kódu a hlásenie chýb. Veľmi nejasné bolo označené posúdenie kedy je jednotkové testovanie ukončené.

Z prieskumu [10] sme identifikovali tieto základné vlastnosti jednotkových testov:

- Zamerané na funkcie testovaného programu.
- Sú založené na štruktúre programu. To znamená, že programátor pozná kód a píše ich tak, aby boli pri testovaní vykonané všetky vetvy zdrojového kódu, ktoré potrebuje otestovať.
- Testy sú špecifikované v zdrojovom kóde.

- Testy sa vykonávajú automaticky a často (niekoľkokrát denne alebo po každej kompilácií).
- Testovanie by malo trvať len krátko, maximálne niekoľko minút.
- Zvyšuje kvalitu a znižuje cenu výsledného produktu lebo vďaka nemu skoro a rýchlo odhalíme chyby.

3.1 Vlastnosti ideálneho pracovného rámca pre jednotkové testovanie

Hlavnou úlohou jednotkového testovania je otestovať, či jedna alebo viac testovaných jednotiek plnia svoju funkciu správne. Zjednodušene môžeme povedať, že pri písaní programov používame dva typy operácií a to sú logické a výpočtové (zmeny dát v dátových typoch). Výpočtové môžeme považovať za správne, pretože počítač vykoná akúkoľvek výpočtovú operáciu s oveľa väčšou pravdepodobnosťou správne ako človek (je len minimálna šanca, že počítač sa môže "pomýliť") a keď sa už stane, že výsledok takejto operácie je nesprávny ide zväčša len o nejakú chybu programátora (napr. pretiekol dátový typ alebo práca s rôznymi typmi v jednej operácií a počítač pri pretypovaní zmenil hodnotu inak ako predpokladal programátor a pod.). Teda môžeme povedať, že ťažisko správnej funkcionality je v logických operáciach a správnom použití výpočtových operácií. Ideálny pracovný rámec na testovanie by preto mal ponúkať možnosti na kontrolu hodnoty premenných, aby sme vedeli skontrolovať či jednotka robí to čo má robiť.

Môže sa zdať, že druhý bod v zhrnutí vlastností s pracovným rámcom nijako nesúvisí a ide len o to, aby programátor poznal zdrojový kód, ku ktorému píše test, ale neplatí to až tak úplne. Správne jednotkové testovanie pokrýva 100% zdrojového kódu a teda každá vetva je vykonaná počas testovania aspoň raz. Framework by preto mal aspoň vedieť zistiť, koľko percent zdrojového kódu je daným testovaním pokrytých, prípadne určiť, ktoré vetvy pokryté nie sú. Pri predstavách o ideálnom pracovnom rámci, ale môžeme ísť ešte ďalej, pretože ideálny pracovný rámec by dokázala sám napísať testy, ktoré by mali 100% pokrytie kódu. Toto zatiaľ síce nie je úplne reálne, ale už teraz existujú pracovné rámce, ktoré dokážu aj samé generovať testy a preto ak by sme uvažovali o ideálnom a v súčasnosti vytvoriteľnom pracovnom rámci tak by mal vedieť generovať testy aspoň do takej miery, aby programátorovi stačilo skontrolovať, resp. prispôsobiť si ich.

Špecifikácia testov by v ideálnom pracovnom rámci mala byť v zdrojovom kóde. Z toho vyplýva, že forma, v ktorej sú testy špecifikované je rovnaká v akej je napísaný aj zdrojový kód testovanej jednotky. Teda aj jazyk by mal byť rovnaký (prípadne s drobnými odchýlkami ak si to špecifikovanie testov vyžaduje), ale jednoznačne by to nemal byť iný programovací jazyk. Vzhľadom na blízkosť testov k zdrojovému kódu je ideálne ak sú testy a zdrojový kód programu prístupné z toho istého vývojového prostredia.

Programátor po napísaní zdrojového kódu, prípadne po nejakej jeho úprave, keď už dokončí všetko čo chcel spraviť kód spustí kompiláciu. Ak kompilácia prebehne úspešne tak potrebuje zistiť, či jednotka pracuje správne, inými slovami chce ju otestovať. Ideálny pracovný rámec by mal teda programátorovi umožňovať vybrať si či chce po každej kompilácií púšťať testy automaticky. Okrem automatického spúšťania by mal vedieť vyhodnotiť, ktorých testov sa zmeny týkali a spúšťať len tie. Vzhľadom na veľmi malú pravdepodobnosť toho, že počítač urobí pri výpočte nejakú chybu (nesúvisiacu so zdrojobým kódom

programátora) môžeme predpokladať, že každá jednotka pri rovnakých vstupných hodnotách vykoná vždy to isté. Preto tie jednotky, v ktorých sa nijako neupravoval zdrojový kód a neupravoval sa ani v jednotkách vytvárajúcich vstupy pre tieto jednotky nie je nutné opätovne testovať. Samozrejme vždy by mala byť možnosť manuálneho spustenia vybraných testov.

Jednotkové testovanie by nemalo dlho trvať, ale dĺžka testovania z väčšej časti nezávisí od pracovného rámca, ale od počtu a dĺžky testov. To ale neznamená, že pracovný rámec s tým nemôže nič spraviť. Každé testovanie pomocou pracovného rámca si vyžaduje nejakú réžiu počas testovania. Ideálny pracovný rámec by žiadnu réžiu okolo nepotreboval, ale to je nemožné a preto za ideálny môžeme považovať ten, ktorého réžia nie je väčšia ako naozaj nutná na poskytnutie všetkých potrebných vlastností, ktoré by testovanie malo mať.

Ideálny pracovný rámec je teda taký, ktorý umožní programátorovi otestovať funkcie jednotiek, vygeneruje testy pre programátora, ktorý ich už len prispôsobí, zistí koľko percent kódu je pokrytého testami, používa rovnaký jazyk na písanie testov ako sa používa v zdrojovom kóde, testy sa píšu v rovnakom vývojom prostredí, púšťajú sa automaticky vtedy kedy chce programátor a len v rozsahu v akom sa robili zmeny v kóde a časová réžia testu je čo najmenšia.

4 Hodnotenie pracovných rámcov

Pre účely tejto práce si definujeme hodnotenie pracovných rámcov, aby sme ich vedeli čo najobjektívnejšie porovnať a rozhodnúť, ktorý má najbližšie k ideálnemu a čo je jeho najslabšia stránka oproti ostatným.

Hodnotené vlastnosti pracovných rámcov:

- čas potrebný na napísanie testov a naučenie sa syntaxe pracovného rámca
- prehľadnosť jazyka a prostredia
- pomer času, ktorý jednotka potrebuje na vykonanie práce počas testovania v pracovnom rámci a času vykonanie mimo neho
- univerzálnosť podporavaných testo (použiteľnosť testov v inom pracovnom rámci)
- využitie mockov pri testovaní
- správa spúšťania testov (možnosť rozdelenia testov do skupín, automatické spúšťanie a pod.)
- zobrazenie výsledkov testovania (prehľadnosť, ako rýchlo sa dá zistiť, ktorý test skončil s chyba a aká bola chyba a pod.)
- automatické generovanie testov

Vyhodnocovanie vlastností sa bude rozlišovať podľa toho, či je daná vlastnosť merateľná (napr. čas) alebo nemerateľná (napr. prehľadnosť). Merateľné vlastnosti budú zoradené od najhoršieho výsledku po najlepší v pozorovanej vlastnosti a tento interval bude rozdelený na niekoľko rovnakých častí. Podľa toho v akej časti pracovný rámec bude toľko bodov dostane (čím viac tým lepšie). Nemerateľné vlastnosti sa nedajú hodnotiť tak objektívne ako merateľné a preto pri nich budeme najprv diskutovať výhody a nevýhody každého pracovného rámca pri pozorovanej vlastnosti a podľa toho budú prideľované body.

5 xUnit

xUnit je označenie pre skupinu pracovnývh rámcov, ktoré slúžia na jednotkové testovanie. Vznikol pôvodne pre programovací jazyk Smalltalk a veľmi rýchlo sa stal známym a úspešným. Dnes už majú všetky bežne používané programovacie jazyky minimálne jeden vlastný pracovný rámec na jednotkové testovanie a mnoho z nich je odvodených práve od xUnit. [1]

Spoločné znaky pracovných rámcov patriacich do skupinu xUnit odvodené zo znakov SUnit [3]:

- **Spúšťač testov (Test runner)** - Je to spustitešný program, ktorý vykoná test a zároveň vytvorí správu o výsledku testu.
- **Testovacie prípady (Test case)** - Je to základná trieda, od ktorej sú odvodené všetky testy. Reprezentuje test alebo skupinu testov.
- **Podmienky pre spustenie testov (Test fixtures)** - Množina podmienok definovaných programátorom, ktoré musia byť splnené pred vykonaním testu. Po teste by mali byť vrátené do pôvodného stavu.
- **Zostavy testov (Test suites)** - Množina testov, ktoré zdieľajú podmienky potrebné pre spustenie testu. Je to množina niekoľkých testovacích prípadov.
- **Vykonanie testu** - Vykonanie individuálneho jednotkového testu.
- **Výsledok testu** - Obsahuje informácie o výsledkoch testu ako napríklad počet úspešných testov, počet neúspešných testov a počet zastavených testov, kvôli chybe programu.
- **Assertion** - Je to funkcia alebo makro, ktorá definuje stav testovanej jednotky. Zvyčajne je to logická podmienka, ktorá pravdivá ak je výsledok testu správny. Zlyhanie väčšinou končí volaním výnimky, ktorá ukončí vykonávanie testu.

5.1 Nástroje patriace do skupiny xUnit pre Javu

Arquillian je inovatívna a rozšíriteľná testovacia platforma pre JVM (Java virtual machine), ktorá umožnuje vytvárať integračné, akceptačné a funkcionálne testy. [2] Arquillian nevyužíva mocky, ale testy spúšťa za behu programu (brings tour test to the runtime), je možné testy debugovať a obsahuje množstvo pluginov pre rôzne iné nástroje. [3] Projekty dodržiavajú tri základné princípy:

- Test by mal byť prenositeľný do akéhokoľvek podporovaného kontajneru.

[1] http://www.martinfowler.com/bliki/Xunit.html
[2] http://arquillian.org/invasion/
[3] http://arquillian.org/invasion/

- Test by mal byť spustiteľný aj s integrovaného vývojového prostredia aj s kompilačného nástroja.
- Platforma by mala rozširovať alebo integrovať existujúce pracovné rámce.

HavaRunner je voľne dostupný testovací pracovný rámec. Jeho najväčším rozdielom oproti ostatným rozšíreným (napr. JUnit a TestNG) je, že testy sú predvolene paralelné, čo prináša nezanedbateľnú zmenu v rýchlosti testov [4]. Okrem spomínaných má aj tieto vlastnosti [5]:

- Dokáže vytvárať skupiny testov.
- Test môže bežať s rôznymi skupinami vstupných dát.
- Každý test má vlastnú inštanciu.
- Model behu je úplne asynchrónny.
- HavaRunner je spúšťač JUnit, to znamená že je jednoduché použiť ho tam, kde sa už používajú JUnit testy.

JExample je testovací pracovný rámec na písanie jednotkových testov, ktoré sú stavané jeden na druhý. Predstavuje vzťah producent-konzument v jednotkových testoch. Producent je testovacia metóda, ktorej test vracia nejakú hodnotu. Konzument je metóda, ktorá závisí od jedného alebo viacerých producentov [6].

Ak producent nejakej metódy zlyhá, tak táto metóda je pri testovaní preskočená. Výstupné hodnoty producentov sú vložené do konzumentov a sú opakovane používané.

JExample je tvorený ako rozšírenie JUnit a je bezproblémovo integrovaný v JUnite aj plugine do Eclipse.

JUnit je jednoduchý testovací pracovný rámec a zároveň základ mnohých ďalších pracovných rámcov, z ktorých sú niektoré spomínané aj v tejto práci. Okrem vlastností vyplývajúcich z toho, že patrí do skupiny xUnit pracovných rámcov umožňuje nastaviť čas pre každý test, po ktorého uplynutí test končí neúspechom, ignorovať testy, spúšťam testy s rôznymi parametrami (parametrizované testy), nastaviť testu očakávanú výnimku a ak vznikne test je akceptovaný, možnosť nastaviť poradie testov a aj testovanie viacvláknových programov [7].

Randoop je testovací pracovný rámec, ktorý automaticky generuje testy v rovnakom formáte ako JUnit. Má rovnaké funkcie ako .NET verzia, o ktorej sa môžete dočítať v predchádzajúcej časti práce.

[4] http://lauri.lehmijoki.net/write-concurrent-java-tests-with-havarunner/
[5] https://github.com/havarunner/havarunner
[6] http://scg.unibe.ch/research/jexample
[7] http://junit.org/

Sprytest je komerčný nástroj založený na používateľskom rozhraní a výrazne ovplyvňuje tvorbu jednotkových testov. Tvorba testov a nastavovanie mockov, nastavovanie asercií (assertions) sú v ňom rýchle vďaka jednoduchému rozhraniu. Taktiež pomáha zistiť mimovoľné zmeny správania a zjednodušuje izoláciu a opravu chýb [8].

Niektoré z kľúčových vlastností:

- Ľahká konverzia na štandardné JUnit testy.
- Zabudované zobrazenie pokrytia kódu, ktoré zobrazuje celú cestu vykonávania.
- Rýchlo vytvára výkonné assercie testov (Test Assertions).
- Bezproblémová synchronizácia medzi testovacími prípadmi a zdrojovým kódom.

TestNG je testovací pracovný rámec inšpirovaný JUnit a NUnit ale prináša aj novú funkcionalitu ako napríklad:

- Spúšťanie testov s rôznymi podmienkami pre vlákna (všetky metódy vo vlastnom vlákne, celá testovacia trieda v jednom vlákne a pod.).
- Testovanie, či program správne funguje aj na viacerých vláknach.
- Flexibilná konfigurácia testov.
- Podpora pre dátovo riadené testovanie.
- Podpora pre parametre.
- Výkonný model vykonávania (žiadne sady testov)
- Podpora rôznych nástrojov a plug-inov.

Je navrhnutý tak, aby pokryl všetky kategórie testov: jednotkové, funkcionálne, integračné, end-to-end. atď.[9]

5.2 Nástroje patriace do skupiny xUnit pre .NET

Fixie patrí k novším pracovný rámcom a umožňujem programátorovi vytvárať a vykonávať jednotkové testovanie. Výhoda, ktorú Fixie prináša, že rozlišovanie metód a tried je na konvencií. Preto programátor pri písaní testu nemusí používať atribúty na označovanie tried a metód. Keď je dodržaná konvencia tak Fixie vie podľa názvu zistiť či ide o metódu alebo triedu. Ak by predvolená konvencia nebola vyhovujúca, je možné vytvoriť si vlastnú a následne sa riadiť ňou. Rozšírenia nemá, ale existujú pluginy do vývojových prostredí. [10]

[8] https://marketplace.eclipse.org/content/sprytest
[9] http://testng.org/doc/index.html
[10] https://visualstudiomagazine.com/articles/2015/04/22/fixie-c-sharp-testing.aspx

MbUnit je rozšíriteľný pracovný rámec, ktorý okrem toho, že prijíma vzory xUnit ide ešte ďalej a poskytuje programátorovi viac, ako napríklad:

- **Porovnávanie XML (XML assertions)** MbUnit obsahuje metódy pomocou, ktorých sa dajú porovnávať aj hodnoty v XML súboroch. [11]
- **Paralelizovateľné testy** Každý test, ktorý je označený ako paralelný bude pri vykonávaní spustený spolu z ostatnými paralelizovateľnými testami[12]. To môže výrazne skrátiť čas potrebný na testovanie, ktorý by mal byť podľa vlastností jednotkového testovania čo najkratší.
- **Externé zdroje dát** Dáta používaného v testoch môžu byť uložené v rôznych typoch súborov (XML, CSV, a pod.) a počas testu používané priamo z nich.

Okrem tohto je MbUnit aj generatívny pracovný rámec, čo znamená že dokáže z jednoduchého jednotkového testu urobiť niekoľko ďalších. Od roku 2013 už, ale nie sú žiadne commity v jeho GitHub repozitári a preto ho môžeme považovať za už nevyvíjaný softvér. [13]

Moq je podľa tvorcov jediná mokovacia knižnica, ktorá je vytváraná od začiatku, tak aby využila naplno výhody .NET Ling (Language-Integrated Query) strom výrazov a lambda výrazy. Funkcie a vlastnosti, ktoré Moq ponúka[14] :

- Strong-typed: no strings for expectations, no object-typed return values or constraints.
- Neprekonaná integrácia s intellisense vo Visual Studio: všetko je plne podporované intellisense vo Visual Studio.
- Žiadne nahraj/prehraj idiómy(jazyk) (No Record/Replay idioms to learn.) na učenie. Stačí vytvoriť mok, nastaviť ho, použíť ho a voliteľne potvrdiť ich volania (netreba potvrdzovať mocky, keď vystupujú len ako stuby (stubs) alebo keď sa robí klasickejšie stavovo-založené (state-based) testovnie kontrolovaním navratových hodnôt testovaného objektu).
- Veľmi nízka učiaca krivka (learning curve). Pre väčšinu častí ani netreba čítať dokumentáciu.
- Granulovaná kontrola (granular control) nad správaním mocku s jednoduchou MockBehavior enumeráciou (netreba vedieť teoretické rozdiely medzi mokom, stubom, imitáciou (fake), dynamickým mokom a pod.).
- Je možne mokovať rozhrania aj triedy.

[11] https://vkreynin.wordpress.com/2010/07/18/test/
[12] http://blog.bits-in-motion.com/2009/03/announcing-gallio-and-mbunit-v306.html
[13] http://stackoverflow.com/questions/3678783/mbunit-vs-nunit
[14] https://github.com/Moq/moq4

- Override expectations: can set default expectations in a fixture setup, and override as needed on tests.
- Posielať argumenty pre mokované triedy.
- Ohraničiť (Intercept) a vyvolať (raise) akcie (events) na mokoch.
- Intuitívna podpora pre out/ref argumenty.

Vďaka mokom je test rýchlejší a nezávislý od prostredia alebo iných častí systému, lebo netreba komunikovať so skutočnými objektami a teda spĺňa vlastnosti jednotkového testovania.

NUnit je testovací pracovný rámec pre C# a je tým, čím je JUnit pre Javu. Poskytuje tiež možnosť mať rôzne nastavenia pre test a rôzne pre tvorbu programu. Vstupné hodnoty testov môžu byť zadané rôznymi spôsobmi. Buď jednoduchou množinou parametrov, náhodne alebo výberom z daného rozsahu alebo z nejakého externého zdroja pomocou metódy. Pomocou atribútov môžeme ovplyvňovať testovacie prostredie. Môžeme tak deklarovať, že daný mest má bežať na samostatnom vlákne, nemá byť spustený, resp. spúšťať sa bude len manuálne alebo má byť len pre špecifickú platformu[15]. Testy je možné spúšťať aj s obmedzeným časom na beh. Po uplynutí stanoveného času je test označený za neúspešný, ale je možné nastaviť, či sa má test okamžite ukončiť alebo sa má nechať dobehnúť [16].

Nbi je doplnok do Nunit pre Microsoft obchodné služby a prístup k dátam (Microsoft Business Intelligence platform and Data Access), ale čiastočne podporuje aj iné platformy[17]. Jeho výhodou je, že netreba vôbec poznať C# alebo mať nainštalované Microsoft Visual Studio lebo testy sa píšu v Xml. Pomocou pracovného rámca sa potom dajú spúšťať aj bez C# kompilátora. Dokáže pracovať s databázovými dotazmi, Etl (Extract, transform, load) balíčkami a aj s viacrozmernými a tabuľkovými modelmi. Testy sa pomocou genbi alebo genbiL dajú generovať aj automaticky. [18] Keďže v prípade Nbi ide do veľkej miery o testovanie databáz vďaka pracovnému rámcu vieme zaručiť aspoň čiastočnú nezávislosť jednotky od okolia, lebo sa nemusíme pripájať na skutočnú databázu na serveri, ale testujeme len na lokálnej.

Pex je v podstate rozšírením Visual Studia a jeho hlavnou funkciou je generovanie testov. Pre každú podmienku, ktorú napíšeme do testu, Pex vytvorí samostatnú vetvu, viackrát spustí program a na základe toho, ktoré vetvy uspeli a neuspeli v testoch vie analyzovať zdrojový kód a vytvoriť ďalšie testy. To môže pomôcť programátorovi nájsť chyby v kóde, ktoré si sám nevšimol. [19] [20]

[15] http://www.slideshare.net/ShirBrass/nunit-features-presentation
[16] https://lukewickstead.wordpress.com/2013/02/09/howto-nunit-features/
[17] http://www.nbi.io/docs/home/
[18] http://fr.slideshare.net/CdricCharlier1
[19] http://research.microsoft.com/en-us/projects/pex/
[20] http://www.pexforfun.com/Documentation.aspx#HowDoesPexWork

Randoop.NET je pracovný rámec, ktorý náhodne generuje testy, ale je riadený spätnou väzbou. Testy vytvára inkrementálne, teda každý ďalší je rozšírením predchádzajúceho. Pred rozšírením testu sa daný vygenerovaný test najprv vykoná a vyhodnotí a následne Randoop rozhodne, či má zmysel daný test rozširovať. To znamená, že ak vygeneruje test, ktorý obsahuje nejaké nezmyselné volanie funkcie (napr. nastavMesiac(-1)) tak test označí za nesprávny a už ho rozširovať nebude.

Okrem toho, že Randoop vie fungovať samostatne, generuje testy ako samostatný súbor, ktorý je použiteľný pre ostatné testovacie nástroje a tým sa dá odhaliť viacej chýb a dosiahnuť tak lepší výsledok testovania. [9] Randoop spĺňa vlastnosti jednotkového testovania a okrem toho ešte aj mimo nich prispieva ku skráteniu doby potrebnej na testovanie tým, že sám vygeneruje testy a nemusí ich vytvárať všetky programátor.

Rhino.Mocks je .NET mokovací pracovný rámec veľmi užitočný na vytváranie falošných objektov tak, aby sme testovali len presne to čo chceme a dovoľuje kontrolovať prostredie a stavy v ktorých test prebieha. Poskutuje tri druhy mokov:

- **Presný (strict) mok** Vyžaduje alternatívnu implementáciu pre každú metódu/vlastnosť. Ak nejaká chýba tak vyvolá výnimku.
- **Dynamický mok** Ak neexistuje alternatívna implementácia pre metódu alebo vlastnosť tak vráti predvolenú hodnotu daného typu.
- **Čiastočný (partial) mok** Ak neexistuje alternatívna implementácia tak sa použije implementácia pôvodného (underlying) objektu.

Mokovať je však možné iba virtuálnych členov reálnych tried alebo celé rozhrania. [21]

xUnit.net je voľný, rozšíriteľný, open-source pracovný rámec dizajnovaný pre programátorov [22]. Výhodou je aj, že je veľmi dobre integrovaný do .NET ekosystému a preto sa netreba báť vážnejšieho problému s kompatibilitou [23]. Niekoľko hlavných dôvodov, pre ktoré sa rozhodli autori vytvoriť xUnit.net [24] :

- Samostatné inštancia objektu pre každú triedu. Niektoré pracovné rámce (napr. Nunit) vytvoria najprv inštancie objektov a ničia sa až po skončení všetkých testov. Preto môžu vznikať problémy kvôli nedostatočnej izolácií testov.
- Žiadne [SetUp] alebo [TearDown] metódy. Tie sa vykonávajú vždy pred testom, resp. po teste, ale spôsobovali problémy a preto sa tvorcovia xUnit.net rozhodli nevytvoriť žiadnu vstavanú podporu pre tieto metódy.
- Nezávislý spúšťač testov. Všetky spúšťače sú nezávislé od verzie a teda akýmkoľvek xUnit.net spúšťačom testov je možné spustí minulé, ale aj budúce testy.

[21] http://www.wrightfully.com/using-rhino-mocks-quick-guide-to-generating-mocks-and-stubs/
[22] https://www.pluralsight.com/courses/xunitdotnet-test-framework
[23] http://www.codeproject.com/Articles/1011753/Moving-to-xUnit-net
[24] https://xunit.github.io/docs/why-did-we-build-xunit-1.0.html

- Rozšíriteľnosť testovacej triedy. Testovacia je trieda je rozšíriteľná napríklad o testy z Nunitu pomocou atribútu [RunWithNUnit].

5.3 Porovnanie vlastností pracovných rámcov patriacich do skupiny xUnit s ideálnym pracovným rámcom

Všetky pracovné rámce testujú funkčnosť a sú písané v jazyku podobnom alebo rovnakom ako aj zdrový kód okrem pracovného rámca Nbi, ktorý je písaný v jazyku XML. Frameworky v skupine xUnit musia vedieť rozdeľovať testy do kategórií tak táto vlastnoť je splnená pri všetkých, aj keď v praxi sa môžu vyskytnúť rôzne aplikácie tejto vlastnosti. Okrem MbUnit, Pex, Randoop a Randoop.NET pracovné rámce nie sú generatívne a teda treba všetky testy písať manuálne. JExample používa v testovaní vzťah viacerých tried a preto by sa už je možné ho zaradiť niekam na hranu medzi jednotkovým a integračným testovaním.

K časovej réžií nie je možné sa vyjadriť bez praktického testovania a preto túto vlastnosť pracovných rámcov pri tomto hodnotení zanedbávame.

Podľa podobnosti s ideálnym pracovným rámcom by som za najlepší označil Randoop(resp. Randoop.NET), pretože patrí ku generatívnym pracovným rámcom, stále sa vyvíja a venujú sa mu aj rôzne vedecké práce. Hneď za neho by som zaradil pracovné rámce TestNG, JUnit, NUnit, xUnit.net pretože sú populárne, každý z nich priniesol niečo nové do jednotkového testovania. Na základe zatiaľ získaných informácií ich podľa mňa nie je zatiaľ možné objektívne zoradiť a určiť, ktorý je lepší alebo horší.

6 Ciele a záver práce

V práci sme analyzovali testovanie softvéru a zamerali sme sa na analyzovanie jednotkového testovania. Analýzou vlastností jednotkového testovania sa nám podarilo vytvoriť teoretický koncept ideálneho pracovného rámca pre jednotkové testovanie. Snažili sme sa vlastnosti určovať tak, aby sme hovorili o uskutočniteľných vlastnostiach, resp. už skutočných vlastnostiach, ale zatiaľ nie spojených do jedného pracovného rámca, a nebolo to niečo nemožné v realite. Tieto vlastnosti sme porovnali s dostupnými informáciami o vlastnostiach niekoľkých existujúcich pracovných rámcov zo skupiny xUnit. Vybrali sme viacero pracovných rámcov pre jazyk C# a aj viacero pre jazyk Java. Vlastnosti týchto pracovných rámcov sme porovnali s vlastnosťami ideálne pracovného rámca. Zhodnotili sme, ktoré vlastnosti pracovné rámce implementujú a ktoré naopak ešte nie alebo ich implementujú len čiastočne. Na základe zhodnotenia sme potom vedeli určiť, ktorý pracovný rámec má najviac spoločných vlastností s ideálnym a teda má k nemu najbližšie. Aj pre jazyk C# aj pre jazyk Java bol najlepší pracovný rámec Randoop, resp. Randoop.NET.

Cieľom našej práce je vytvoriť projekt v takom rozsahu, aby sme mohli otestovať a zhodnotiť tento pracovný rámec aj v skutočnosti. Našim cieľom je vyskúšať vlastnosti tie vlastnosti pracovného rámca, o ktorých sa v práci diskutuje. Na jednom projekte sa nám pravdepodobne nepodarí do hĺbky otestovať všetky jeho vlastnosti, ale chceme otestovať každú aspoň čiastočne a tie pri ktorých to bude možné aj do hĺbky. Vzhľadom na to, že projekt ešte nemáme špecifikovaný nevieme konkretizovať ktoré vlastnosti a ako bude možné otestovať. Testovať budem pracovné rámce Randoop a Randoop.NET a projekt bude implementovaný v oboch jazykoch (C#, Java).

7 Časový plán do odovzdania práce v letnom semestri

- **december 2015** - špecifikácia projektu, na ktorom budem implementovať testovanie
- **január a február 2016** - implementácia projektu, implementácia testov vo vybraných pracovných rámcoch
- **marec 2016** - testovanie projektu pracovnými rámcami a zhromažďovanie výsledkov testov pre účely zhodnotenia
- **apríl 2016** - zhodnotenie a porovnávanie pracovných rámcov
- **máj 2016** - dokončovanie práce a odovzdanie do 10.5.2016

Literatúra

[1] Enrique Alba and Francisco Chicano. Observations in using parallel and sequential evolutionary algorithms for automatic software testing. *Computers & Operations Research*, 35(10):3161–3183, 2008.

[2] Amiangshu Bosu, Michaela Greiler, and Christian Bird. Characteristics of Useful Code Reviews: An Empirical Study at Microsoft. *2015 IEEE/ACM 12th Working Conference on Mining Software Repositories*, pages 146–156, 2015.

[3] Johannes Brauer. *Programming Smalltalk – Object-Orientation from the Beginning*.

[4] Standard Computer Dictionary. *Standard Computer Dictionary*.

[5] Elfriede Dustin. *Effective Software Testing: 50 specific ways to improve your testing*. 2002.

[6] Cem Kaner. Exploratory Testing. *Quality Assurance International*, (c), 2006.

[7] A.A. Omar;F.A. Moha. A survey of software functional testing methods. *Software Engineering Notes*, page 75, 1991.

[8] Glenford J. Myers, Tom Badgett, and Corey Sandler. *The Art of Software Testing*.

[9] Carlos Pacheco, Shuvendu K. Lahiri, Michael D. Ernst, and Thomas Ball. Feedback-directed random test generation. *Proceedings - International Conference on Software Engineering*, pages 75–84, 2007.

[10] P Runeson. A survey of unit testing practices. *IEEE Software*, 23(4):22–29, 2006.

[11] E. V Veenendaal. Standard glossary of terms used in Software Testing, Version 1.2. *International Software Testing Qualification Board*, 1:1–51, 2010.

[12] Wang Wei. From source code analysis to static software testing. *2014 IEEE Workshop on Advanced Research and Technology in Industry Applications (WARTIA)*, pages 1280–1283, 2014.