Slovenská technická univerzita v Bratislave
Fakulta informatiky a informačných technológií
Študijný program: Informatika

Peter Beňuš

Automatické testovanie softvéru

Bakalársky projekt 1

Vedúci bakalárskeho projektu: Ing. Karol Rástočný
December 2015

Slovenská technická univerzita v Bratislave
FAKULTA INFORMATIKY A INFORMAČNÝCH TECHNOLÓGIÍ

AUTOMATICKÉ TESTOVANIE SOFTVÉRU

Študijný program: Informatika
Autor: Peter Beňuš
Bakalárska práca:
Vedúci bakalárskej práce: Ing. Karol Rástočný
december 2015
V práci sa zaoberám automatickým testovaním softvéru a konkrétne jednotkovým testovaním. Jednotkové testovanie je prvým testovaním spustiteľného programu a je neoddeliteľnou súčasťou životného cyklu akéhokoľvek softvérového produktu. V analýze prostredia sú analyzované viaceré dostupné frameworky pre jazyky C# a Java. Ku každému z vybraných frameworkov je v práci krátky opis základných vlastností a zhodnotenie či spĺňa vlastnosti, ktoré by malo jednotkové testovanie mať a ak áno tak do akej miery sú tieto vlastnosti splnené. Vlastnosti sú definované z praktických skúseností firiem, ktorých produktom je softvér.

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.

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.

2 Testovanie softvéru

Testovanie softvéru je empirická činnosť, ktorá skúma kvalitu testovaného produktu alebo služby vykonávaná na podanie informácií o kvalite všetkým zainteresovaným osobám [5]. V súčastnosti existuje veľa spôsobov testovania a veľa častí životného cyklu softvéru v ktorých sa aplikujú iné typy testov. Testovanie softvéru môžem rozdeliť na kategórie podľa postupu, ktorý sa používa pri testovaní, podľa spôsobu testovanie 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 [4]. Test je konštruovaný z funkcionálnych vlastností, ktoré sú špecifikované požiadavkách na program [6]. 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 úrovní testov sa využíva aj na validáciu softvéru [4].

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 [4]. Používa sa pri jednotkovom testovaní na skoré odhalenie všetkých 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 [4]. 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 [10].

2.2.2 Dynamické testovanie

Dynamické testovanie prebieha už na spustenom softvéri. Začína skôr ako je úplne dokončený softvér, pretože sa počas vývoja testujú aj menšie spustiteľné časti. K dynamickému testovaniu sa viaže validácia.

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 [4].

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 [3]. Niekedy, napríklad pri testovaní inštalácie a použiteľnosti, sa 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 [9].
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) [1].

3 Kľúčové vlastnosti jednotkového testovania

Per Runeson, profesor na univerzite v Švédskom Lunde robil prieskum medzi 50 firmami, ktorých hlavným produktom je softvér [8]. 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. 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 kompilácií. 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 frameworkov 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.

3.1 Zhrnutie vlastností jednotkového testovania

Základné vlastnosti vyplývajúce z prieskumu:

- 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.

4 xUnit

xUnit je označenie pre skupinu frameworkov, 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 framework na jednotkové testovanie a mnoho z nich je odvodených práve od xUnit. [1]

Spoločné znaky frameworkov patriacich do skupinu xUnit:

- **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. [2]

4.1 Nástroje patriace do skupiny xUnit pre .NET

Fixie patrí k novším frameworkom a umožňujem programátorovi vytvárať a vykonávať jednotkové testovanie. Výhoda, ktorú Fixie prináša, že je založený na konvencií. Preto programátor pri písaní testu nemusí používať atribúty na označovanie tried a metód. Keď je

[1] http://www.martinfowler.com/bliki/Xunit.html

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. Na internete je možné aj nájsť plugin pre ReSharper, avšak zatiaľ len beta verziu. [2]

MbUnit je rozšíriteľný framework, 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. [3]
- **Paralelizovateľné testy** Každý test, ktorý je označený ako paralelný bude pri vykonávaní spustený spolu z ostatnými paralelizovateľnými testami. [4]
- **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 framework. čo znamená že dokáže z jednoduchého jednotkového testu urobiť niekoľko ďalších. Od roku 2013 uz ale nie sú žiadne commity na GitHube a preto ho môžeme považovať za už nevyvíjaný softvér. [5]

Mock je to simulovaný objekt, ktorý imituje správanie reálneho objektu. Zvyčajne sa používajú pri jednotkovom testovaní. Mockovanie je celé o imitovaní (faking) reálnych objektov a robení operácií kontrolovaným spôsobom. [6]

Moq je podľa tvorcov jediná mockovacia knižnica, ktorá je vytváraná od začiatku, tak aby využila naplno výhody .NET Ling (Language-Integrated Query) strom výrazov a lambda výrazy.

- 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 mock, 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.

[2] https://visualstudiomagazine.com/articles/2015/04/22/fixie-c-sharp-testing.aspx
[3] https://vkreynin.wordpress.com/2010/07/18/test/
[4] http://blog.bits-in-motion.com/2009/03/announcing-gallio-and-mbunit-v306.html
[5] http://stackoverflow.com/questions/3678783/mbunit-vs-nunit
[6] http://www.agile-code.com/blog/mocking-with-moq/

- Granulovaná kontrola (granular control) nad správaním mocku s jednoduchou MockBehavior enumeráciou (netreba vedieť teoretické rozdiely medzi mockom, stubom, imitáciou (fake), dynamickým mockom a pod.)
- Je možne mockovať rozhrania aj triedy
- Override expectations: can set default expectations in a fixture setup, and override as needed on tests
- Posielať argumenty pre mockované triedy
- Ohraničiť (Intercept) a vyvolať (raise) akcie (events) na mockoch
- Intuitívna podpora pre out/ref argumenty. [7]

Nunit je testovací framework pre C# a je to v podstate to isté, čo je Junit pre Javu (o Junite neskôr v Java frameworkoch). V Nunite je možné pridávať testy do rôznych kategórií a potom spúšťať testy spadajúce len do vybraných kategórií. Môžeme tak ušetriť dosť času ak nepotrebujeme testovať úplne všetko. 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.[8] Testy je možné spúšťať aj s obmedzeným časom na beh. Po uplynutí stanoveného času je test označený za neúspešný, ale vieme nastaviť či sa má test okamžite ukončiť alebo sa má nechať dobehnúť. [9]

Nbi je doplnok do Nunit pre Microsoft obchodné služby a prístup k dátam (Microsoft Business Intelligence platform and Data Access). Jeho výhodou je, že netreba vôbec poznať C# alebo mať nainštalované Microsoft Visual Studio lebo testy sa píšu v Xml. Pomocou frameworku 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. [10]

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

[7] https://github.com/Moq/moq4
[8] http://www.slideshare.net/ShirBrass/nunit-features-presentation
[9] https://lukewickstead.wordpress.com/2013/02/09/howto-nunit-features/
[10] http://fr.slideshare.net/CdricCharlier1

zdrojový kód a vytvoriť ďalšie testy. To môže pomôcť programátorovi nájsť chyby v kóde, ktoré si sám nevšimol. [11] [12]

Randoop.NET je framework, ktorý náhodne generuje testy, ale je riadený spätnou väzbou. Testy vytvára inkrementálne, teda každý ďalší je rozšírením predchádzajúceho, ale najprv vygenerovaný test vykoná a vyhodnotí 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. [7]

Rhino.Mocks je .NET mokovací framework veľmi užitočný na vytváranie falošných objektov tak, aby ste testovali len presne to čo chcete 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.

Mockovať je však možné iba virtuálne členov reálnych tried alebo mockovať celé rozhrania. [13]

xUnit.net je voľný, rozšíriteľný, open-source framework dizajnovaný pre programátorov [14]. Výhodou je aj, že je veľmi dobre integrovaný do .NET ekosystému a preto sa netreba báť vážnejšieho problému s kompatibilitou [15]. Niekoľko hlavných dôvodov, pre ktoré sa rozhodli autori vytvoriť xUnit.net [16] :

- Samostatné inštancia objektu pre každú triedu. Niektoré frameworky (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.

[11] http://research.microsoft.com/en-us/projects/pex/
[12] http://www.pexforfun.com/Documentation.aspx#HowDoesPexWork
[13] http://www.wrightfully.com/using-rhino-mocks-quick-guide-to-generating-mocks-and-stubs/
[14] https://www.pluralsight.com/courses/xunitdotnet-test-framework
[15] http://www.codeproject.com/Articles/1011753/Moving-to-xUnit-net
[16] https://xunit.github.io/docs/why-did-we-build-xunit-1.0.html

- Nezávislý spúšťač testov. Všetky spúšťače sú nezávislé od verzie a teda akýmkoľvek xUnit.net spúšťačom testov je možné spustí minulé, ale aj budúce testy.
- Rozšíriteľnosť testovacej triedy. Testovacia je trieda je rozšíriteľná napríklad o testy z Nunitu pomocou atribútu [RunWithNUnit].

4.2 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. [17] 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. [18] Projekty dodržiavajú tri základné princípy:

- Test by mal byť prenositeľný do akéhokoľvek podporovaného kontajneru.
- Test by mal byť spustiteľný aj s integrovaného vývojového prostredia aj s kompilačného nástroja.
- Platforma by mala rozširovať alebo integrovať existujúce frameworky.

HavaRunner je voľne dostupný testovací framework. Jeho najväčším rozdielom oproti ostatným rozšíreným (napr. JUnit a TestNG) je, že testy sú predvolene paralelné, čo prináša nezanedbateľnú zmenu v rýchlosti testov [19]. Okrem spomínaných má aj tieto vlastnosti [20]:

- 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 už sú JUnit testy.

JExample je testovací framework 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 [21].

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

[17] http://arquillian.org/invasion/
[18] http://arquillian.org/invasion/
[19] http://lauri.lehmijoki.net/write-concurrent-java-tests-with-havarunner/
[20] https://github.com/havarunner/havarunner
[21] http://scg.unibe.ch/research/jexample

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

JUnit je jednotuchý testovací framework a zároveň základ mnohých ďalších frameworkov, z ktorých sú niektoré spomínané aj v tejto práci. Okrem vlastností vyplývajúcich z toho, že patrí do skupiny xUnit frameworkov 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 [22].

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

Sprytest je komerčný nástroj založený na používateľskom rozhraní a výrazne ovplyvňuje tvorbu jednotkových testov. Tvorba testov a nastavovanie mockov, nastavovanie asercií (assertions) môžu trvať len zlomok času. Taktiež pomáha zistiť mimovoľné zmeny správania a zjednodušuje izoláciu a opravu chýb [23].

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 framework 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)

[22] http://junit.org/
[23] https://marketplace.eclipse.org/content/sprytest

- Podpora rôznych nástrojov a plug-inov.

Je navrhnutý tak, aby pokryl všetky kategórie testov: jednotkové, funkcionálne, integračné, end-to-end. atď'.[24]

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

[3] Standard Computer Dictionary. *Standard Computer Dictionary*.

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

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

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

[7] 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.

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

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

[10] 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.

[24] http://testng.org/doc/index.html