Snaha o tvorbu kvalitného softvérového produktu

JOZEF SLEZÁK

Slovenská technická univerzita
Fakulta informatiky a informačných technológií
Ilkovičova 3, 842 16 Bratislava
jozef.slezak@gmail.com

Abstrakt. Jednou zo základných požiadaviek na softvér je jeho kvalita. Ako docieliť požadovanú kvalitu softvérového produktu? Základnou ideou môže byť podpora vývoja tvorbou testov a použitím metrík kvality softvéru. V tomto článku sa zaoberám niektorými zo súčasných trendov, akými sú unit testy, TDD (Test Driven Developement), IoC (Inversion Of Control). Vyjadrím sa k výhodám a nevýhodám daných prístupov.

Úvod

Kvalita softvéru závisí od mnohých faktorov akými sú dobrý plán, riadenie, prepracovaná dokumentácia a v neposlednej rade aj zdrojový kód. V tomto dokumente sa sústredím práve na to akými spôsobmi je možné udržať, prípadne zvýšiť kvalitu zdrojového kódu.
Prvá časť tohto článku je venovaná overeniu funkčnosti softvéru. Správna funkčnosť je nevyhnutnou podmienkou na to, aby sa vôbec dalo konštatovať, že softvér je kvalitný. Zakladaným nástrojom na dosiahnutie tohto požadovaného stavu je testovanie. Táto časť pojednáva o súčasných trendoch testovania v procese vývoja softvéru.
Nasledujúca časť pojednáva o ďalších kvalitatívnych vlastnostiach zdrojových kódov. Ide o vlastnosti, ktoré je možné zistiť inšpekciou zdrojového kódu a aplikáciou vhodných metrík. Najhlavnejšiu úlohu bude hrať vhodná štruktúrovanosť, ktorá je dôležitá pre ďalšie rozširovanie a údržbu softvéru.

Overenie funkčnosti softvéru

V praxi sa už mnohokrát potvrdilo, že čím je chyba odhalená v neskoršej fáze vývoja softvéru, tým nákladnejšie je jej odstránenie - pričom náklady rastú exponenciálne. [2] Preto by sme sa pri vývoji softvéru mali snažiť odhaliť chybu podľa možnosti čo najskôr. Tento fakt určite presvedčí mnohých ekonomicky zmýšľajúcich ľudí, aby sústredili svoju pozornosť na testovanie funkčnosti.

Manažment v softvérovom inžinierstve, október 2006, s. 1-6.

Uvediem ešte citát z knihy Nástroje pre platformu Java: "V prípade, že nejakej vlastnosti programu chýba existencia automatizovaného testu, potom môžme predpokladať, že nie je funkčná. Takéto tvrdenie znie totiž omnoho bezpečnejšie než častejšie používane tvrdenie, že vývojár si je istý, že daná vlastnosť je a bude vždy funkčná."

Tieto tvrdenia by mali programátorov dostatočne motivovať k tomu, aby k problematike testovania nepristupovali ľahostajne.

Automatizované testovanie poskytuje riešenie ako testovať správnosť funkčnosti softvéru efektívne. Automatizovanými testami sa bude ďalej zaoberať.

Unit Test

Unit testy pomáhajú odkrývať takzvané regresné chyby [6]. Ide napríklad o chyby vznikajúce neúmyselne ako dôsledok zásahu do systému. Inými slovami, unit testy identifikujú časti zdrojového kódu, ktoré ešte pred časom správne plnili svoju funkciu.

Unit test je úsek kódu, napísaný špeciálne pre účely overenia toho, že určitá časť kódu spĺňa dané požiadavky na funkcionalitu.

Ide teda o automatizované testy, čo znamená, že je ich možne spúšťať automaticky a nie je nevyhnutné, aby tím testerov musel daný softvér preklikávať.

Unit testy sú neoddeliteľnou súčasťou pri refaktoringu softvéru. Refaktoring taktiež napomáha zvýšiť kvalitu softvéru tým, že zlepšuje štruktúru softvéru. Unit testy v tomto prípade máju za úlohu určiť, či aj po zmene štruktúry, plní softvér tú istú funkcionalitu.

Výhody unit testov

Výhodou tvorby unit testov je možnosť overenia správnosti funkcionality. Nemôže sa stať, že doplnením funkcionality do softvéru znefunkčníme inú časť, bez toho, aby sme o tom vedeli. Znižujú sa nároky na čas testerov pretože množstvo testov prebehlo ešte pred tým ako by danú funkcionalitu musel testovať tím testerov.

Nevýhody unit testov

- Vznikajú problémy so zmenou alebo pridaním funkcionality. Často sa stava, že je nutné zmeniť aj automatizovane testy. Preto by sa mali prevažne testovať len dôležité triedy. Štandardne sú nimi servisy[1], triedy ktoré spolupracujú s inými triedami a triedy ktoré sa plánujú znovupoužiť.
- Vývojári softvéru musia stráviť ďalší čas vytváraním unit testov.
- Ďalším problémom môže byť časová náročnosť spustenia sady automatizovaných testov.
- Stáva sa, že test nejakej funkcionality zaberá viac riadkov kódu ako funkcionalita samotná.

1 Servisy sú triedy, ktoré poskytujú služby iným triedam

Výhody verzus nevýhody

Vytváranie automatizovaných testov zaberá len malo času programátora [1]. Výhody, ktoré získame ich vytvorením výrazne prevyšujú nad ich nevýhodami.

Samotný proces vytvárania týchto testov je podporený širokou sadou rámcových systémov pre množstvo programovacíh jazykov[2]. Tieto rámcove systémy sa význačujú tým, že sú jednoducho pochopiteľné a ľahko použiteľné so zabudovanou podporou vo vývojových prostrediach.

Automatizované testy a automatický preklad

Osvedčenou praktikov pri vývoji softvéru je mať nástroj, ktorý automaticky prekladá zdrojové súbory.

V dnešnej dobe sú takéto nástroje doplnené aj o možnosť automatického spustenia testov. Ak sa naše testy spúšťajú automaticky zabránime chybe ľudského faktoru, ktorý by mohol zabudnúť spustiť dané testy.

Tento prístup taktiež rieši problém časovej náročnosti spúšťania testov. Pri spúšťaní jednotlivých testov zo sady je potrebná inicializácia testovaného podprogramu. V dnešnej dobe je takáto inicializácia spojená napríklad s inicializovaním objektovo relačných maprov, pripadne inicializáciou spojení na vzdialene servisy.

Zvykom býva, že po spustení všetkých testov sa rozošle mail, o tom či všetky testy prebehli úspešne. Udržiavame si tak povedomie o tom či je naša aplikácia spoľahlivá. Šetríme úsilie vývojárov, ktorí by si inak tieto testy museli spúšťať sami. [3]

Programovanie riadene testami (TDD - Test Driven Developement)

V tejto kapitole sa budeme zaoberať metódou vývoja softvéru zvanou programovanie riadene testami (TDD). TDD sa používa aj v súvislosti s agilným vývojom softvéru[3].
TDD je nový prístup, ktorý má potenciál výrazne zvýšiť produktivitu vývojárov. Zaujímavé na TDD je to, že predpisuje disciplinovane tvoriť unit testy ešte pred samotným implementovaním funkcionality pre ktorú test píšeme. To umožňuje sústrediť pozornosť programátora na rozhranie softvéru a jeho správanie sa, zatiaľ čo buduje jeho architektúru.[4]

Na prvý pohľad je to prekvapujúce, že TDD umožňuje zvýšiť produktivitu vývojárov. TDD pridáva síce zanedbateľne málo práce na vytvorenie testov, ale núti programátorov sústrediť sa na malý podproblém, ten vyriešiť, otestovať a pokračovať ďalším.

Tento prístup k tvorbe softvéru umožňuje výrazne zvýšiť jeho kvalitu. Skupina vývojárov IBM Retail Store Solutions prehlásila [1], že TDD im umožnilo znížiť faktor chybovosti pri funkcionálnom testovaní na 50% a výrazne znížiť prácu testerov v porovnaní s predchádzajúcou verziou toho istého systému.

---

2 Ide o skupinu rámcových systémov xUnit. Pre jazyk Java je to JUnit a pre jazyk PHP zase PHPUnit.

3 Jednou z agilnych metód je aj Extrémne programovanie (XP). [XP, SoftArchImprovTDD]. Táto metóda je pripravená na zmeny špecifikácie softvérového produktu počas jeho vývoja.

TDD a plánovanie

Testy sú dopredu naplánované. Programátor potom implementuje postupne bod po bode každú položku plánu. Počas implementácie môže programátor plán rozširovať o ďalšie testy, ktoré plán nezahŕňal. Tým, že sú všetky testy naplánované je viditeľný stav vývoja softvéru. Tuto vlastnosť zrejme privíta každý manažér softvérového projektu.

Ďalšou výhodou TDD je, že programátori píšu kód, ktorý je automaticky testovateľný. Práca programátora prebieha v krátkych iteráciach tak, že napíše jeden automatizovaný test. V tomto bode program nemusí byť vôbec preložiteľný. Potom napíše implementačný kód tak, aby pokrýval práve napísaný test.

Zavádzanie TDD do praxe

Pri zavadzaní TDD do praxe vzniká problém spojený s tým, že programátori musia prijať za svoju túto metódu vývoja. Je potrebné im vysvetliť, že TDD ma vysoký dopad na ich produktivitu a z dlhodobého hľadiska získavajú.

Výhody oproti ad-hoc prístupu

Vo väčšine prípadov proces vytvárania unit testov nemal stanovené presné pravidlá. Testy sa zvyčajne vytvárali až dodatočne. Potom sa stavalo, že mnoho testov nebolo vytvorených, hlavne ak testovanie nebolo striktne naplánované.

IoC (Inversion Of Control) a testovanie

V kontexte tohoto článku sa budeme zaoberať IoC z pohľadu ako prispieva k testovateľnosti softvéru. IoC odstraňuje závislosti softvéru. Tým znižuje zviazanosť softvérových súčiastok [5]. Jednotlive súčiastky implementujú rozhranie pomocou, ktorého s nimi potom komunikujú iné súčiastky. Toto má výrazný dopad na testovanie [5]. IoC nám umožňuje otestovať funkcionalitu izolovanej časti objektov. Jednotlivé súčiastky izolujeme tak, že použijeme tzv. "Mock Objects". „Mock Objects“ sú akési objekty, ktoré neplnia žiadnu užitočnú funkcionalitu. Len implementujú rozhranie tých objektov, ktoré chceme odizolovať, teda práve netestovať. Použitie IoC si vyžaduje určite úsilie. Toto úsilie sa dá minimalizovať použitím, tzv. IoC kontainerov.

Testovanie Webových Aplikacii

Webové aplikácie sú široko rozšírené, vzniká potreba zaistiť ich kvalitu testovaním. Tradične metódy testovania však nie sú v mnohých prípadoch vhodné. [8] Pri vývoji web aplikácii musíme dbať na mnohé aspekty akými sú okrem funkcionality aj výkonnosť, kompatibilita a bezpečnosť. Všetko to má za následok náročnejšie testovanie a údržbu v porovnaní s tradičným softvérom.

Tradičný beh web aplikácie vyzerá tak, že užívateľ vyplní vstupné polia a následne sa na strane serveru zavolá príslušná biznis logika na strane serveru. [BuildinWebApps]

Biznis vrstvu teda vieme testovať štandardnými metódami na ostatné časť musíme použiť špecializované nástroje[4].

Statická analýza zdrojového kódu

Inšpekcia zdrojového kódu

Tak isto ako testovanie aj inšpekcia kódu ma za cieľ zvýšiť kvalitu zdrojového kódu. Inšpekcia zdrojového kódu prebieha tak, že sa analyzuje zdrojový kód a hľadajú sa v ňom nedostatky. Kontroluje pri tom to, čo automatické testy nedokázali. Túto inšpekciu môže robiť človek alebo sa použije špecializovaný nástroj určený na inšpekciu zdrojového kódu. Od toho potom závisí, aké chyby je možné nájsť v zdrojovom kóde. V prípade, že inšpekcia zdrojového kódu je vykonávaná človekom, je možné odhaliť napríklad chyby súvisiace s bezpečnosťou zdrojového kód. Veľmi často je tento proces dopĺňaný refaktoringom.

Nástroje určené na statickú analýzu zdrojového kódu[5]

Takéto nástroje nám umožnia ešte pred spustením zdrojového kódu hľadať takzvaný podozrivý zdrojový kód. Ide o široké spektrum podozrivého kódu od nepoužitých premenných v kóde a nedosiahnuteľných miest v zdrojovom kóde. Cez chyby pretečenia pamäte, ale aj synchronizácie vlákien v programe. Moderné programovacie jazyky umožňujú už na úrovni jazyka deklarovať synchronizáciu vlákien.

Metriky

Nakoniec by som ešte rád spomenul niečo o vyhodnocovaní kvalitatívnych ukazovateľoch prostredníctvom metrík. Množstvo metrík je založených na vyčísľovaní miery zviazanosti a súdržnosti softvérových súčiastok a takisto aj metrík merajúcich vlastnosti objektovo orientovaného sofvtvéru. [7].

Záver

Tak ako aj v iných odvetviach tak aj pri vývoji softvéru je neustály trend zefektívňovať a skvalitňovať proces tvorby softvéru. Vývoj softvéru sa odlišuje oproti klasickým inžinierskym disciplínam tým, že produkt, ktorý vyvíjame je nehmotný. Takisto aj teórie okolo sú relatívne mladé.

Do určitej mieri je možné nájsť paralely medzi klasickými inžinierskými disciplínami a vývojom softvéru. Predstavme si paralelu v tom, že pre novo postavený most sa robia záťažové testy takým spôsobom, že na most prídu ťažko naložené

4 Takýmto nástrojom je napr. Cactus (http://jakarta.apache.org/cactus/)

5 Takými to nástrojmi sú napr.: PMD (http://pmd.sourceforge.net/), FindBugs (http://findbugs.sourceforge.net/)

nákladné autá a meria sa, či to most zvládne. Takúto metaforu môžeme aplikovať aj na softvér s tým, že nákladnými autami budú rôzne testy (automatizované unit testy, záťažové testy a iné).

Keďže táto disciplína je mladá neustále prichádzajú nové metódy ako vyvíjať softvér, jednou z nich bola aj spomínaná TDD. Tento prístup je momentálne považovaný za jeden z najlepších. Môže sa stať, že už za krátku dobu sa objavý nová metóda.

Stotožňujem sa s tým, že metódy, ktoré boli spomenuté v tomto článku sú v súčastnosti považované za jedni z najlepších a umožňujú vytvárať kvalitnejší softvér. Sú naozaj najlepšie? Nebudú už o rok nové, ešte lepšie?

Zavádzaní každej novej metódy do praxe vzniká problém s tým, že metóda musí byť prijatá danou komunitou ľudí. Neostáva však nič iné len sledovať aktuálne trendy a vždy si z nich zobrať to čo by náš proces mohlo zefektívniť.

Taktiež si myslím, že aj keď použijeme viac či menej kvalitnú metódu, za každým bude bez pochyby nevyhnutné daný proces vhodne naplánovať.

Použitá literatúra

1. E. Michael Maximilien, Laurie Williams, Assessing Test-Driven Development at IBM, ACM digital library.
2. Bieliková, M.: *Princípy softvérového inžinierstva*. STU Bratislava, ISBN 80-227-1322-8
3. Dustin, E., Rashka, J., and Paul, J., *Automated Software Testing. Reading, Massachusetts.* Addison Wesley, 1999.
4. David S. Janzen University of Kansas, Software Architecture Improvement through Test Driven Development. ACM digital library.
5. http://www.martinfowler.com/articles/injection.html. Navštívené dňa 24.10.2006.
6. *http://en.wikipedia.org/wiki/Regression_test.* Navštívené dňa 24.10.2006.
7. Jozef Slezák, *Ohodnocovanie softvérových systémov*. Záverečný projekt bakalárskeho štúdia. Máj, 2006 FIIT STU.
8. *Testing Web Applications Focusing on Their Specialties.* ACM digital library.
9. Jim Conallen (2000): *Building Web Applications with UML*. Addison-Wesley Publishing Company, Reading, MA, 2000.

Annotation

An attempt for creating quality software

There are many requiremets on software behaviour. One of most important is software quality. How to reach required software quality?. The main idea is to support software developement by using tests and quality metrics. In this paper I discuss the nowdays trends or methods for example unit tests, TDD (Test Driven Developement), IoC (Inversion Of Control). I will write my opinion about this methods and discuss advantages and disadvantages.