Zabezpečenie kvality v softvérovom projekte

IGOR ANDRUŠKA

Slovenská technická univerzita
Fakulta informatiky a informačných technológií
Ilkovičova 3, 842 16 Bratislava
igor[.]andruska[zavináč]gmail[.]com

Abstrakt. Kvalitu softvéru môžeme rozdeliť na internú a externú. Externá kvalita vyplýva z požiadaviek zákazníka. Kľúčová v jej zabezpečovaní je etapa testovania implementovaného softvéru. Táto etapa má však veľmi nepríjemnú vlastnosť: jej trvanie a efektívnosť sú ťažko predikovateľné. Táto vlastnosť môže úzko súvisieť s internou kvalitou softvéru. Neprehľadný, zložitý a príliš prepojený zdrojový kód komplikuje testovanie. Tiež spôsobuje zlú udržiavateľnosť softvéru, z ktorej vyplýva nadmerná tvorba chýb. V eseji sa autor zamýšľa nad súvislosťami medzi internou a externou kvalitou softvéru. Rozoberá možnosti zvýšenia internej kvality softvéru v jednotlivých fázach vývoja.

Úvod

Riadenie a vyhodnocovanie kvality je v softvérovom priemysle oproti iným odvetviam priemyslu špecifické. Výroba softvéru nie je hromadná. Teda nemôžeme využiť exaktné štatistické metódy riadenia a vyhodnocovania kvality. Každý softvérový produkt je sám o sebe jedinečný a požiadavky na kvalitu sú pre každý produkt rôzne. Softvér má unikátnu a zväčša komplikovanú vnútornú štruktúru. Zákazník túto štruktúru nevidí a ani ho nezaujíma. Softvérový inžinieri pochopili, že s tradičnými definíciami a modelmi kvality si softvérový priemysel nevystačí. Softvérový produkt nestačí posudzovať iba klasicky z pohľadu zákazníka. Treba zvážiť aj vnútornú štruktúru softvéru. Kvalita spracovania vnútornej štruktúry má nezanedbateľné dôsledky na celý softvérový proces a nakoniec na aj kvalitu z pohľadu zákazníka.

Model kvality

Norma ISO 9126 definuje model kvality pre softvérové projekty. Delí kvalitu na tri časti [4]:

1. Interná kvalita
2. Externá kvalita

Manažment projektov softvérových a informačných systémov, október 2008, s. 1-8.

3. Kvalita z hľadiska používania

Interná kvalita je v záujme vývojárov. Odráža kvalitu návrhu a implementácie produktu (vrátane dokumentácie). Hodnotí vnútornú štruktúru softvéru, nie jeho správanie. Sleduje sa v prvých fázach vývoja od analýzy po implementáciu. Naopak, externá kvalita odráža správanie sa softvéru. Je definovaná ako miera, akou softvérový produkt spĺňa požiadavky zákazníka. Externá kvalita sa vyhodnocuje a zlepšuje počas testovania a neskôr počas prevádzky a údržby systému. Model kvality podľa ISO 9126 je založený na šiestich základných charakteristikách (Tab. 1). Každá obsahuje niekoľko podcharakteristík [4].

Tab. 1. Charakteristiky s podcharakteristiky kvality podľa normy ISO 9126.

Interná a externá kvalita					
Funkčnosť	***Spoľahlivosť***	***Použiteľnosť***	***Efektívnosť***	***Udržiavateľnosť***	***Prenositeľnosť***
Účelnosť	Zrelosť	Zrozumiteľnosť	Správanie v čase	Analyzovateľnosť	Adaptabilita
Precíznosť	Prípustnosť chýb	Naučiteľnosť	Spotreba zdrojov	Schopnosť zmeny	Jednoduchosť inštalácie
Schopnosť súčinnosti	Schopnosť zotavenia	Schopnosť Prevádzky		Stabilita	Zhoda
Bezpečnosť		Príťažlivosť		Testovateľnosť	Nahraditeľnosť

Interná kvalita softvéru sa asi najviac prejavuje v *udržiavateľnosti* softvéru. Je jasné, že kvalitný návrh a implementácia podporí *analyzovateľnosť* zdrojového kódu. Taktiež zjednoduší či už zmeny funkčnosti alebo štruktúry softvéru. Zlá *analyzovateľnosť* zdrojového kódu má spravidla za následok vnášanie chýb do softvéru pri zmenách. Tento faktor vyjadruje *stabilita*. No a v neposlednom rade môže interná kvalita výrazne podporiť *testovateľnosť* softvéru. Samozrejme, interná kvalita viac alebo menej ovplyvňuje aj iné charakteristiky. Existuje mnoho metrík na jej vyhodnocovanie. Typicky sa prihliada na tieto [5]:

1. *Zložitosť* – delí sa na dve metriky:
   - *Cyklomatická zložitosť* – počet nezávislých vetiev behu programu
   - *Hĺbka vnorených blokov* – maximálna hĺbka vnorených blokov (riadiace konštrukcie) v podprograme
2. *Prepojenosť* – vyjadruje mieru, v akej sú moduly previazané mimo štandardného rozhrania. Inými slovami v akej miere sú previazané vnútornými implementáciami.
3. *Súdržnosť* – vyjadruje ako sú metódy nejakého modulu logicky (účelovo) podobné. Napr. ak má trieda za úlohu robiť syntaktickú analýzu jazyka, sú v nej iba metódy týkajúce sa procesu syntaktickej analýzy (a nie sú v nej metódy vykonávajúce lexikálnu analýzu a pod.). Hovoríme, že trieda je vysoko súdržná.

Interná kvalita softvéru je pre zákazníka nezaujímavá. Zákazníka zaujíma do akej miery produkt spĺňa jeho požiadavky. To je vlastne definícia *externej kvality*. Externá kvalita sa zlepšuje počas testovania modulov a testovania systému. Používateľa produktu zaujíma *kvalita z hľadiska používania*. Charakteristiky, ktorých sa týka, sú *efektívnosť*, *bezpečnosť*, *produktivita* a *spokojnosť*. Vyhodnocuje a zlepšuje sa počas akceptačného testovania.

Požiadavky na kvalitu spravidla explicitne vyplývajú z analýzy a požiadaviek zákazníka. Na druhej strane niektoré charakteristiky kvality môžu vyplynúť implicitne. Teda nepriamo. Napr. prostredie rýchlo podlieha zmenám a softvér treba adekvátne prispôsobovať. To implikuje nutnosť vysokej udržiavateľnosti softvéru. Zákazník spravidla priamo nepožaduje vysokú udržiavateľnosť softvéru. To je záležitosť vývojárov. Identifikovať a určiť prioritu charakteristikám kvality je úlohou manažéra kvality.

Ak sa softvér nespráva podľa špecifikácie, hovoríme že je v ňom chyba. Chyby znižujú v očiach zákazníka a používateľa kvalitu softvéru. Za účelom hľadania a odstraňovania chýb je v softvérovom procese zaradená fáza testovania.

Testovanie

Etapa testovania je pre zabezpečenie externej kvality a kvality z hľadiska používania softvérového produktu kľúčová [1][3]. Cieľom testovania je odhaliť chyby a preukázať požadované vlastnosti softvéru [2]. Hovorí sa, že bezchybný softvér neexistuje. Aj keby sme pripustili jeho existenciu, nemôžeme si byť istý, že tým bezchybným je práve náš. Etape testovania sa teda nevyhneme. Zameriam sa testovanie súčiastok a testovanie systému. Tieto sú zaujímavé z hľadiska overovania funkčných vlastností softvéru.

Samotné testovanie môžeme realizovať dvoma základnými technikami: statickým a dynamickým testovaním [2]. Statické testovanie nevyžaduje vykonanie programu. Zväčša sa jedná o formálnu prehliadku zdrojového kódu alebo dokumentácie. Klasickým príkladom je lokalizácia chyby v zdrojovom kóde. Prehliadka môže byť veľmi sťažená v zložitom kóde (vysoká cyklomatická zložitosť, hlboké vnorené bloky). Vyšším stupňom statického overovania je formálny dôkaz správnosti programu. Je zrejmé, že tento typ overovania je použiteľný len pre veľmi špecifické typy projektov.

V praxi častejšie používaným je dynamické overovanie. Podstatou je samotné vykonávanie programu s množinou „vhodne“ zvolených testovacích vstupov a overovanie výstupov. Čo znamená „vhodná“ voľba vstupov? Na to poznáme dva prístupy. Prvou je metóda čiernej skrinky. Tu nemáme nijakú informáciu o vnútornej štruktúre programu. Návrh testovacích vstupov vychádza zo vstupno-výstupnej špecifikácie (modulu, algoritmu, funkcie, ...) Ide o pokrytie všetkých kombinácií vstupov. Je jasné, že množiny testovacích vstupov môžu narásť do značných rozmerov. Ako zvládnuť testovanie? Ideálnym riešením je automatické testovanie. Myslím si ale, že takéto testovanie je reálne len na nižších úrovniach. Teda pri testovaní algoritmov,

súčiastok a čiastočne pri testovaní systému na úrovni protokolov a rozhraní medzi modulmi. Automatické testovanie celkového systému môže vyžadovať samostatný vývoj nástrojov na realizáciu testov. Tie môžu byť špecifické pre daný projekt. A to sú nemalé výdavky navyše.

Ak poznáme vnútornú štruktúru programu, môžeme využiť prístupov tzv. bielej skrinky. Tu sa množina testovacích vstupov môže značne redukovať. Už nás nezaujímajú hodnoty z celej domény premennej *x*, ale iba subdomény, ktoré majú vplyv na riadiace konštrukcie programu (napr. kladné a záporné hodnoty). Problém môže nastať keď je program príliš zložitý. Moduly alebo iné menšie časti softvéru sú neprehľadné a silne prepojené. Množina testovacích vstupov môže opäť narásť do veľkých rozmerov. Takýto program sa nielenže ťažko štrukturálne testuje, ale spravidla obsahuje viac chýb. Dotkli sme sa internej kvality softvéru. Teda testovateľnosť programu závisí priamo od jeho internej kvality.

Ak by sa počas testovania neobjavili chyby (resp. zanedbateľne málo), niečo nie je v poriadku so samotným procesom testovania (dobré testovanie vždy odhalí chyby). Na druhej strane, ak testovanie odhaľuje priveľké množstvá chýb, tiež nie je niečo v poriadku. Veľké množstvá chýb neúmerne predlžujú proces testovania. Nestačí chybu iba nájsť. Treba ju stabilizovať (presne identifikovať okolnosti za ktorých nastáva), lokalizovať v zdrojovom kóde a odstrániť. Práve lokalizácia môže trvať značne dlho. V tomto prípade by sme sa mali zamyslieť nad internou kvalitou softvéru.

Vo väčšine prípadov je prakticky nemožné vykonať úplné dynamické testovanie a odhaliť všetky chyby. Lacnejšie a efektívnejšie je chybám predchádzať. Kľúčom je možno práve zlepšenie internej kvality.

Interná kvalita

Chyby teda znižujú externú kvalitu softvéru. Prečo ale chyby vznikajú? Myslím si, že hlavná príčina je v probléme človeka vysporiadať sa so zložitosťou. Softvér vo všeobecnosti je jedným z najkomplikovanejších ľudských intelektuálnych výtvorov. Úlohou veľkých softvérových systémov je spravidla modelovanie procesov v reálnom svete. Vzťahy medzi entitami reálneho sveta sú veľmi komplikované. A požiadavky na detailnosť modelu môžu byť veľmi vysoké. Z toho vyplýva komplikovanosť modelujúceho softvéru.

Kvalitným návrhom však môžeme zložitosť do značnej miery redukovať. Moja snaha pri návrhu softvéru je, aby štruktúra v čo najväčšej miere odrážala modelovanú realitu. Potom je zdrojový kód pre človeka, ktorý pozná problémovú oblasť na dostatočnej úrovni abstrakcie, relatívne ľahko analyzovateľný. Programátor, ktorý potrebuje urobiť zmenu (a pozná problémovú oblasť), potom nemá problémy s pochopením štruktúry programu. Ťažkosti s chápaním štruktúry majú podľa mňa najväčší podiel na tvorbe chýb v programoch. Ak si programátor neuvedomuje detailne vplyv vykonanej zmeny na ostatné časti softvéru, veľmi ľahko urobí chybu. To sa zväčša stáva, ak je program príliš prepojený alebo málo súdržný. Ak sa modifikáciami programu opakovane vnášajú do programu chyby, problém už nie je v programátoroch.

Aj oni sú iba ľudia, a teda majú ťažkosti s chápaním zložitých systémov. Problém je v samotnom návrhu softvéru.

Navrhovať softvér modelovaním reality nie je vždy vhodné. Občas stačí zachytiť komplikovanú realitu len veľmi povrchne. Výkon modelu verne odrážajúceho štruktúru skutočnosti by mohol byť neadekvátne malý. V každom prípade si myslím, že zdrojový kód treba udržiavať na vhodnej úrovni abstrakcie.

Refaktoring

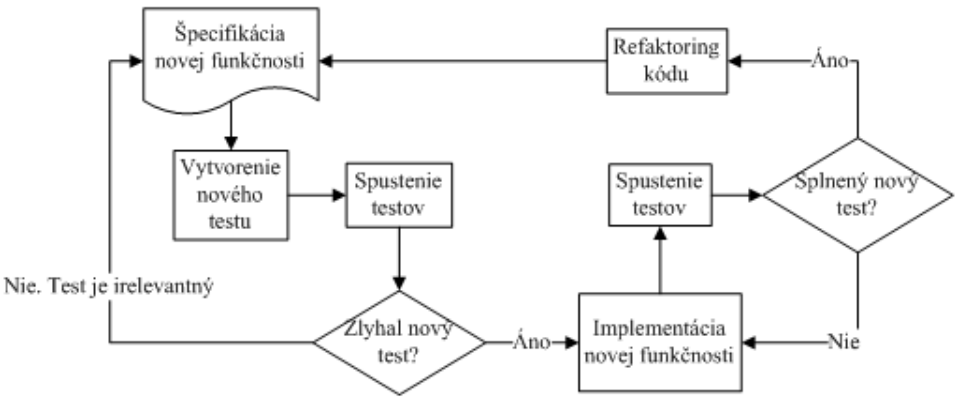
Aj kvalitný návrh však môže časom degradovať. K degradácii vedie nesprávne pridávanie nových funkčností do systému. V praxi sa často nová funkčnosť do programu iba „prilepí“ bez ohľadu na vnútornú filozofiu návrhu. To je podľa mňa veľmi nebezpečné. Ušetrí sa síce čas a prostriedky a dokonca to môže budiť dojem, že softvér je vysoko udržiavateľný. Časom sa ale systém stane komplikovaný, jednotlivé predtým oddelené súčasti príliš prepojené alebo moduly málo súdržné. Ďalšími modifikáciami sa vnášajú chyby a v konečnom dôsledku utrpí externá kvalita, zákazník a naša povesť. Narážam na pojem zvaný *refaktoring*. Teda zlepšenie vnútornej štruktúry programu pri zachovaní vonkajšieho správania. Náklady na mohutný refaktoring sú nezanedbateľne vyššie ako súčet nákladov na priebežné malé úpravy štruktúry. Dôvod je zrejmý. Programátori musia znovu naštudovať a pochopiť časti kódu, s ktorými už dlho nepracovali (ak vôbec sú vo firme programátori, ktorý niekedy s danými časťami pracovali). Navyše sa manažéri často s obavami (ak vôbec) púšťajú do rozsiahlejšieho refaktoringu. Majú pocit, že sa do už prácne „odladeného“ systému vnesú znovu chyby. Čiastočne obavy chápem. Myslím si ale, že ak sa refaktoring zverí programátorovi s detailnými znalosťami problémovej oblasti, úspech je zaručený. Ak taký človek v spoločnosti nie je, nie je ohrozená iba udržiavateľnosť softvéru, ale celý projekt.

Refaktoring sa dá podľa mňa využiť aj tak trochu netradične. V situáciách keď sme stabilizovali chybu, nepodarilo sa nám ju presne lokalizovať (často práve kôli nízkej internej kvalite danej časti softvéru). Vieme len, že sa nachádza v nejakom module, funkcii alebo inej menšej časti. Termín vydania sa blíži no chybu aj napriek dlhým hodinám pátrania (možno až týždňom) stále nevieme lokalizovať. Refaktoring časti kódu môže byť riešením problému. Prepíšeme časť a chyba sa viac neobjaví. Alebo počas opätovnej implementácie (a premýšľaní nad riešeným problémom) chybu objavíme. V praxi som tento prístup viackrát s úspechom použil (a často nikdy nepochopil v čom chyba spočívala).

Testami riadený vývoj

Myslím si, že modely vývoja softvéru, kde sa softvér najskôr celý navrhne, potom implementuje a nakoniec testuje, majú jednu podstatnú nevýhodu. Objasním na príklade: programátor napíše mnoho dôležitých súčastí softvéru (a s nízkou internou kvalitou) a ešte pred fázou testovania opustí spoločnosť. Projekt dospeje do fázy testovania a odhalí sa mnoho chýb v spomínaných súčastiach. Lokalizácia chýb iným programátorom môže byť veľmi zdĺhavá a „bolestivá“. Alternatívou ku „klasickým“ modelom vývoja sú agilné metodiky, konkrétne testami riadený vývoj (angl. Test-

driven development - TDD). Účelom tejto metodiky nie je zlepšenie procesu testovania ako by názov mohol napovedať. Motiváciou je zlepšenie návrhu, teda internej kvality softvéru [5]. TDD sa týka fázy návrhu a implementácie produktu. Typicky vývoj prebieha podľa nasledovnej schémy (Obr. 1):



Obr. 1. Schéma vývoja metodikou TDD.

Okamžité testovanie (typicky automatické testovanie) sčasti rieši vyššie spomínaný problém s lokalizáciou množstva chýb odhalených v záverečnom testovaní. Avšak, prínos je hlavne v zlepšenej internej kvalite softvéru. Štúdia [5] ukazuje, že metodika TDD vedie programátorov k písaniu kratších modulov (funkcií, tried). Funkcie sú tiež jednoduchšie (nižšia cylkomatická zložitosť, menšia hĺbka vnorených blokov) a moduly menej prepojené, čo podporuje znovupoužitie a testovateľnosť zdrojového kódu. Softvér vytváraný metodikou TDD je teda podľa štúdie interne kvalitnejší.

Nevýhodou TDD je podľa mňa fakt, že podrobný návrh sa tvorí zároveň s pridávaním funkčností. Čo ak si pridanie nejakej špeciálnej funkčnosti vynúti rozsiahly refaktoring? Toto riziko by sa dalo zmierniť už spomínaným verným modelovaním reality. Ideálne potom každá nová funkčnosť spôsobí iba zovšeobecnenie modelu a posunie ho na vyššiu úroveň abstrakcie.

Štýl programovania

Raz mi známy v diskusii o štýle programovania povedal: „Nikto mi nemôže prikázať ako mám písať program.“ Je to síce pravda, ale prístup to správny rozhodne nie je. Programovanie už dávno nie je umením. Je súčasťou inžinierskej disciplíny. Nevýhodou softvérového priemyslu je, že nemá pevné normy. Chápem, že zaviesť ich by bolo prakticky nemožné. Je to práve kvôli jedinečnosti každého softvéru, časti kódu alebo algoritmu. Máme návrhové vzory, ale tie sú stále iba akýmsi odporúčaním a skôr sa týkajú návrhu. Myslím si ale, že majú obrovský význam. Umožňujú programátorom rýchlo začať pracovať s akýmisi „blokmi“ informácie namiesto aby pri každom

systéme prácne analyzovali ťažkopádne zapísané celé štruktúry kódu. Implementácia samotných algoritmov zostáva stále v plnej moci programátora. V praxi sa často nepoužíva ani pomenovanie premenných podľa účelu použitia. Aj tu sa presadzujú štandardy. Ale tie sa skôr týkajú formátovania kódu ako jeho štruktúre. Myslím, že zavedenie vzorov pre zápis riadiacich konštrukcií by výrazne prispelo k analyzovateľnosti a čitateľnosti programov. Podobne ako návrhové vzory by posúvali analýzu kódu na vyššiu úroveň abstrakcie. Z praxe som nadobudol pocit, že programátori trávia neúmerne veľa času analyzovaním triviálneho kódu, avšak zapísaného neprehľadne a komplikovane. Často ho ani úplne nepochopia a následnými modifikáciami vnášajú do programov chyby.

Záver

Touto esejou som chcel podať svoj hrubý pohľad na najdôležitejšie príčiny nízkej internej kvality softvéru a možnosti jej zlepšenia. Samozrejme faktorov je viac a súvislosti medzi externou a internou kvalitou oveľa hlbšie.

Zložitosť softvéru bude časom iba narastať. Kľúčom k zabezpečeniu kvality v budúcnosti je podľa mňa posúvať programovanie a samotné programovacie nástroje na vyššiu úroveň abstrakcie. Tiež je potrebné v čo najväčšej miere proces implementácie automatizovať a odstrániť tak v čo najväčšej miere ľudský faktor.

Použitá literatúra

1. Bertolino, A.: Software Testing Research: Achievements, Challenges, Dreams. In: *2007 Future of Software Engineering* (May 23 - 25, 2007). International Conference on Software Engineering. IEEE Computer Society, Washington. 2007.
2. Bieliková, M.: *Softvérové inžinierstvo: Princípy a manažment*. FEI STU Bratislava, 2000. Diel I.
3. Feldman, S. *Quality assurance: much more than testing*. Queue 3, 1 (Feb. 2005), 26-29, 2005
4. Ho-Won Jung, Seung-Gweon Kim, Chang-Shin Chung: *Measuring Software Product Quality: A Survey of ISO/IEC 9126*. IEEE Software, vol. 21, no. 5, 88-92, Sep/Oct, 2004.
5. Janzen, D. S., Saiedian, S.: *Does Test-Driven Development Really Improve Software Design Quality?*. IEEE Software, vol. 25, no. 2, 77-84, Mar/Apr 2008.

Annotation

Quality assurance in software project

Software quality can be divided into internal and external. External quality is implied by customers' requirements on product. Key activity in external quality assurance is testing. However testing stage has a serious drawback: it may be hard to predict its duration and

efficiency. This can be tightly related with low internal software quality. Unclean, complex and coupled source code makes testing harder. It also causes low maintainability especially low analyzability. This may be the source of program defects and errors. In this essay author deals with relationship between internal and external software quality. Author also aims at possible reasons of low software quality and it's improvement in different stages of software development cycle.