Techniky zabezpečenia a zvýšenia kvality softvéru

TOMÁŠ KRAMÁR

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

Abstrakt. Napriek tomu, že agilné metódy tvorby softvéru sa čoraz viac dostávajú do povedomia programátorov a manažérov, niektoré projekty stále postupujú vodopádovým modelom. Testovanie sa tak dostáva na rad až po skončení implementácie, v čase, kedy sa už len opravujú chyby a výsledná funkcionalita, ktorá z pohľadu zákazníka tvorí kvalitu, sa dá len ťažko ovplyvniť. Navyše, aj akceptačné testovanie odhaľuje množstvo chýb, ktoré sa dali objaviť už počas vývoja. Kvalitný softvér vzniká počas implementácie, je preto dôležité aby sa počas nej aj pravidelne overoval. Ako však možno zvýšiť kvalitu kódu a znížiť chybovosť? Ako predísť nepríjemným prekvapeniam pri integrácii? Aj na tieto otázky ponúka esej odpoveď vo forme prehliadok kódu a tvorbe rôznych druhov automatizovaných testov a ich pravidelným spúšťaním na integračnom serveri (technika známa ako neustála integrácia).

Na začiatku bolo testovanie

Zamysleli ste sa niekedy nad tým, čo znamená pojem „kvalita softvéru“? Na túto otázku by asi inak odpovedal zákazník, inak programátor a inak projektový manažér. Ktorá odpoveď je však správna? Podľa Denninga [2] sa používatelia nepýtajú: „Je kód tohto systému dobre štrukturovaný?“, ale „Pomáha mi tento softvér urobiť viac práce? „Môžem sa naňho spoľahnúť?“. Pre používateľov je teda meradlom kvality spoľahlivosť softvéru. Je preto namieste, aby sme sa ňou pri tvorbe programov patrične zaoberali, pretože len spoľahlivý program (bez chýb) je kvalitný (neplatí to však naopak).

Každý komplikovaný softvér je zložený z obrovského množstva kódu a riadi sa princípom úzkeho hrdla. Program je spoľahlivý do takej miery, do akej je spoľahlivá jeho najmenej spoľahlivá časť. Pre zabezpečenie kvality je teda potrebné otestovať *každú* časť softvéru.

Napísať zložitý softvér je komplikované; je však omnoho zložitejšie otestovať ho. Prečo? Pretože softvér je závislý od množstva externých faktorov a jeho beh je riadený

akciami používateľa a dátami, ktoré sa menia. Človek nie je schopný nasimulovať a otestovať všetky možné stavy, do ktorých sa môže program dostať. Situácia je skomplikovaná tým, že softvér sa časom mení. Každá zmena má potenciál zaniesť do kódu nové chyby, ktoré môžu byť odhalené len opakovaným testovaním. Navyše, chyby sa časom kumulujú a je obtiažne objaviť ich príčinu a opraviť ich, pretože sa navzájom ovplyvňujú. Ich dopad je aj psychologický – ľudia majú menej energie na odstraňovanie chýb ak ich je príliš veľa. Tento fenomén sa označuje ako syndróm rozbitých okien. Ak je na budove rozbité okno a nie je nahradené novým, všetky zvyšné okná budú čoskoro tiež rozbité. Jedno neopravené okno je signálom toho, že nikto sa nestará, takže rozbitie ďalších okien je beztrestné...

Chyby, aj keď vyzerajú malé a neškodné dokážu narobiť veľa problémov – znižujú sebavedomie, ovplyvňujú plány a reputáciu. Chyby v nasadenom softvéri môžu nahnevať používateľov. Chyby objavené počas procesu vývoja sú prekážkami pri práci, stavajú sa do cesty a zabraňujú zvyšku softvéru pracovať korektne. Ak sa do akceptačného testovania dostane príliš chybový produkt, strávi sa veľa času opravovaním veľkých chýb – tých, ktoré sa v produkte ani nemali objaviť. Keď nastane čas hľadať malé a zákernejšie chyby, väčšinou už na to nie je čas.

Spomínané problémy sa najjednoduchšie odstránia tým, že programátori budú tvoriť kód, ktorý nebude obsahovať chyby. To naneštastie nie je úplne možné, môžeme sa však snažiť počet chýb minimalizovať a včas ich odhaliť. To je možné dosiahnuť:

- prehliadkami kódu,
- tvorbou automatizovaných testov
- a ich pravidelným spúštaním

Viac očí viac vidí

Pod pojmom „testovanie“ si väčšina z nás predstaví spúsťanie softvéru za účelom overenia jeho správnosti. Alternatívnou formou testovania (alebo, presnejšie kontroly kvality) je ukázanie práce kolegovi za účelom obajavenia chýb a možností zlepšenia.

Kým tradičné testovanie je limitované na overovanie vlastností spustiteľného kódu, prehliadky môžu byť (a mali by byť) použité na všetky časti procesu vývoja: od návrhu až po dokumentáciu.

Za chyby sa platí a v softvérovom inžinierstve sa za chyby platí tým viac, čím neskoršie sa odhalia. Troška matematiky ukáže, že najviac ušetríme na chybách, ktoré odhalíme čo najskôr – a to sú chyby v špecifikácii požiadaviek. Nič však nie je zadarmo, a platí to aj pre prehliadky. Na jednej strane ušetríme na chybách, na druhej strane stratíme čas a prostriedky na ich hľadanie. Prehliadky môžu „zhltnúť“ až 5 – 15% všetkých nákladov na projekt [6]. Napriek tomu sa ukazuje, že výhody ďaleko prevážia náklady.

Prehliadky skracujú dobu potrebnú na vývoj tým, že skracujú dobu potrebnú na integráciu a redukujú čas potrebný na testovanie, pretože do finálneho testovania sa dostáva menej chybový produkt. Vyššia kvalita finálneho produktu tiež skracuje čas

potrebný na údržbu. Čas, ktorý by bolo potrebné venovať hľadaniu a opravovaniu chýb je tak možné venovať vyvíjaniu nových funkcií.

Samotné testovanie softvéru má len obmedzený efekt. Priemerný podiel odhalených chýb je len 25% pre jednotkové testy[1], 35% pre funkčné testy a 45% pre integračné testy. Pre porovnanie, priemerná efektívnosť prehliadok kódu je 55 až 60% [5]. Podľa inej štúdie, typický programátor nájde za hodinu testovaním 2-4 chyby ale 6-10 chýb pri prehliadke kódu [3]. Je to spôsobené aj tým, že ak sa pri testovaní odhalí chyba, je potrebné venovať čas hľadaniu príčiny. Pri prehliadke kódu sa však hľadajú priamo príčiny chyby, nie len symptómy.

Prehliadky kódu však neslúžia ako náhrada testovania, ale len ako doplnok. Tieto dve techniky pomáhajú odhaľovať rôzne druhy chýb, sú aplikovateľné v rôznych fázach projektu a vyžadujú rôzne druhy a úrovne skúseností. Testovanie ukazuje reálne správanie systému, nie modelové správanie, ktoré bolo vydedukované pri prehliadke. Prehliadka kódu nedokáže odhaliť problémy s výkonom, ani nedokáže odhaliť problémy s použiteľnosťou rozhrania. Je preto dôležité aby si obe techniky našli svoje miesto pri zabezpečovaní kvality a pracovali vo vzájomnej súhre.

Pri prehliadke je možné postupovať viacerými spôsobmi [8]:

- **Inšpekcia** je formálny, najsystematickejší a najrigoróznejší typ prehliadky. Prebieha na základe definovaného procesu, so špecifickými rolami priradenými jednotlivým zúčastneným. Inšpekcie sú efektívnejšie pri hľadaní chýb ako bežné neformálne prehliadky.
- **Tímová prehliadka** je osekanou, menej formálnou verziou inšpekcie. Niektoré časti procesu sú zjednodušené alebo vynechané a role sa prekrývajú.
- **Prehliadka** je neformálna; autor popisuje kód kolegom a prijíma komentáre. Autor tu hrá dominantnú rolu, žiadne iné role nie sú definované.
- Pri **párovom programovaní** pracujú dvaja vývojári na jednej pracovnej stanici. Jeden programátor programuje a druhý neustále kontroluje jeho prácu. Párovému programovaniu však chýba perspektíva osoby, ktorá nie je osobne zainteresovaná v skúmanom kóde.
- Pri **posudku** skúma kód iba jediná osoba.

Bez ohľadu na typ prehliadky by však mali byť všetci zúčastnení pripravení. Prehliadka bez predošlej prípravy je neefektívna. Až 75% všetkých chýb sa odhalí počas prípravy [7].

Vo všeobecnosti platí, že tím by si mal zvoliť taký typ prehliadky, ktorý mu umožňuje efektívne odhaľovať chyby a zároveň minimalizovať náklady. Napriek tomu, že existuje veľa druhov prehliadok, len inšpekcie sú uznávané ako „best practice“.

Zavedenie prehliadok kódu do procesu vývoja nie je jednoduché. Vyžaduje si prekonanie mnohých psychologických bariér a predsudkov. Programátori sa zvyknú spočiatku prehliadkam kódu brániť. Myslím si však, že toto odmietanie je úplne

[1] angl. unit tests

prirodzené a indikuje, že programátor si je vedomý toho, že v jeho kóde existujú chyby. Prehliadka však vyžaduje, aby si ich aj priznal, čo nie je pre každého úplne ľahké, ani intuitívne. Človek sa prirodzene bráni; nikto predsa nechce, aby sa jeho „špinavé prádlo“ prepieralo na verejnosti.

Pri prvom styku s prehliadkou kódu si to zrejme málokto uvedomí, ale jej cieľom je odhalenie chýb *a kritika kódu autora, nie autora* samotného. Prehliadka by sa teda mala niesť v konštruktívnom duchu, v snahe objaviť chyby a nie riešiť osobné problémy.

Podľa Wiegersa [8] je ďalším problémom, ktorý bráni zavedeniu prehliadok zainteresovanosť manažmentu. Počet nájdených chýb by nemal slúžiť ako podklad pre odmeňovanie zamestnancov – z toho dôvodu by sa manažéri ani nemali zúčastňovať prehliadok. Tento problém však nemusí byť až taký čiernobiely. Ak má manažér dostatočné technické znalosti a chce pomôcť, neexistuje dôvod prečo by sa nemal zúčastniť. Všetko je založené na vzájomnej dôvere a rešpekte.

Podľa mňa by mali byť prehliadky súčasťou vývojového procesu v každom tíme. Ich výhody sú zrejmé. Akákoľvek forma prehliadky má dramatický dopad na kvalitu produktu. Nielen že sa hľadajú a opravujú chyby, čo skvalitňuje kód po stránke stability, ale odhaľujú sa a komenutujú aj rôzne zlé praktiky, čo zvyšuje kvalitu aj z pohľadu programátora. Každá prehliadka je navyše dobrou lekciou pre autora, ktorý sa nabudúce (ak je schopný prijať konštruktívnu kritiku) podobnej chybe vyhne. Existujú isté chyby, najmä tie, spôsobené neznalosťou jazyka, alebo platformy, ktoré majú tendenciu opakovať sa. Je omnoho efektívnejšie zaraziť chybnú praktiku už v zárodku, ako potom opravovať niekoľko podobných chýb odhalených pri testovaní. Alebo, v horšom prípade, opraviť niekoľko nájdených chýb a niekoľko iných nenájdených, nechať čakať na „svoju príležitosť“.

Ďalším pozitívnym vedľajším efektom je, že všetci zúčastnení spoznávajú kód. Aj keď to priamo nevplýva na kvalitu produktu, umožňuje tímu rýchlejšie zotaviť sa (aj) z (trvalého) odchodu autora. Kód sa tak ešte viac stáva spoločným vlastníctvom tímu a oprava chyby bez prítomnosti pôvodného autora zaberá podstatne menej času.

Aby boli zmeny kódu bez strachu

Kód je ako živý organizmus. Časom sa rastie a mení. Každá zmena má však nepríjemnú tendenciu pokaziť existujúcu funkcionalitu. Ak je kódu málo, je možné otestovať každú zmenu ručne. Časom však prestane byť ručné testovanie použiteľné – pri každej zmene je potrebné dôkladne otestovať nielen nový kód, ale aj jeho interakciu so starým kódom.

Je preto prirodzené, že programátori vytvárajú automatizované testy. Jednotkový test skúma „jednotku“ kódu, volá funkcie alebo metódy kódu a porovnáva aktuálny výstup s očakávaným výstupom. Definícia „jednotky“ sa líši v každom programovacom jazyku, či paradigme. V objektovo orientovaných jazykoch sa za jednotku považuje objekt a testy skúmajú jeho jednotlivé metódy.

V čom teda spočíva výhoda jednotkových testov? Každý test je program, ktorý je možné spustiť, a ktorý automaticky vyhodnotí korektnosť testovanej jednotky.

Priebežným vytváraním testov pre každú novú funkcionalitu tak získame komplexnú sadu testov, ktorá preverí funkčnosť kódu. Ak testy prejdú, všetko je v poriadku. Ak testy neprejdú? Ukážu, kde je chyba. Ak sú testy dostatočne jemne granulované a navzájom nezávislé, každý test overuje práve jednu malú vlastnosť kódu. Test, ktorý zlyhá, tak presne ukáže na chybné miesto v kóde, aj s postupom, ako chybu vyvolať. Ako dlho by to trvalo človeku?

Jednotkové testy nám navyše umožňujú nasimulovať rôzne podmienky, ktoré počas bežného „človeko-testovania" nikdy nenastanú, prípadne je obtiažne ich nasimulovať. Ako sa zachová náš systém, ak sa externý systém zachová tak či onak? Po vytvorení testu je odpoveď otázkou pár sekúnd.

Práca s kódom, ktorý má len minimálne, alebo žiadne pokrytie testami, je ako prechádzať sa po tenkom ľade. Nikdy neviete, kedy sa ľad praskne a vy sa prepadnete. Táto paralela však nie je úplne presná. Pri vývoji softvéru nie je chybu vidno ihneď, ale prejaví sa až dlho potom, čo ľad praskol. Kód, ktorý je pokrytý testami umožňuje zmeny kódu bez strachu. Po každej zmene kódu sa spustí sada testov, ktorá odhalí nové chyby a zaistí, že starý kód nebol poškodený novými zmenami.

Podobne ako prehliadky kódu, ani jednotkové testy nie sú náhradou za ručné testovanie. Jednotkové testy môžu nedokážu odhaliť niektoré druhy chýb, ktoré odhalí len manuálne testovanie. Rovnako nedokážu odchytiť všetky chyby, mali by sa preto používať ako doplnok k ostatným druhom testov.

Napriek zrejmým výhodám sa spája s tvorbou testov mnoho kontroverzií. Väčšina vývojárov podvedome tuší, že by mali tvoriť testy. Len málo ich ich však aj naozaj tvorí. Univerzálnou odpoveďou na otázku „Prečo nie?" je „Mám málo času". To však vedie k vytvoreniu bludného cyklu. Čím menej času programátor má, tým menej testov tvorí. Čím menej testov tvorí, tým je menej produktívny a jeho kód je menej stabilný [1]. Čím je menej produktívny a čím menej stabilný kód tvorí, tým má menej času a dostáva sa pod väčší tlak. Začarovaný kruh.

Úplne opačným extrémom je posadnutosť testami. Jednou z metrík kvality testov je „pokrytie testami", ktorá popisuje stupeň otestovanosti kódu. 100% pokrytie kódu znamená, že test vykonáva testované funkcie takým spôsobom (s takými argumentami) a simuluje také podmienky, že testovaný kód prejde pri testoch všetkými možnými cestami. Takéto pokrytie kódu sa vyžaduje pri kritických aplikáciách (pre armádu), je však veľmi ťažko dosiahnuteľné. Myslím, že je zbytočné snažiť sa dosiahnuť čo najväčšie pokrytie kódu. Pri bežných projektoch je to zbytočné – je dôležité aby každá jednotka kódu mala svoj test a aby test preveril aspoň základnú funkčnosť. Nie je však potrebné snažiť sa tvoriť zbytočne veľa testov za každú cenu. Námaha vynaložená na tvorbu testov by mohla vynulovať zisk produktivity, ktorá je nimi získaná.

Čo bolo skôr: test, alebo kód?

V súvislosti s tvorbou jednotkových testov sa vynára jedna otázka: „Je lepšie napísať najprv test a potom kód, alebo najskôr kód a k nemu test?". Pre väčšinu ľudí je prirodzené napísať najprv kód a potom podľa neho vytvoriť test. Existujú však techniky ako „test-first development" alebo „test driven development", ktoré propagujú tvorbu testu ešte pred vytvorením kódu. Zástancovia tejto techniky tvrdia, že

tento prístup zvyšuje kvalitu softvéru, skeptici zasa tvrdia, že takýto prístup je jednak kontraproduktívny a jednak ťažko naučiteľný [3].

Ktorá technika je však správna? Erdogmus s kolegami [3] dospeli empirickým pozorovaním k záveru, že tí programátori, ktorí písali najskôr testy, ich písali aj viac. Viac testov viedlo v konečnom dôsledku k väčšej produktivite. Nepodarilo sa však preukázať, že by „test-first" viedlo k väčšej kvalite. Vyššia produktivita je podľa nich spôsobená viacerými efektami:

- *Lepšie porozumenie zadaniu.* Tvorba testu pred tvorbou kódu vyžaduje od programátora vytvorenie rozhrania – a na to potrebuje dôkladne pochopiť úlohu kódu.
- *Zameranie sa len na jednu úlohu.* Pri „test-first" vývoji sa postupuje po jednotlivých testoch. Test pokrýva kód, ktorého funkcionalita má len obmedzený rozsah. To umožňuje programátorovi sústrediť sa len na jednu malú úlohu.
- *Menšia námaha pri oprave chýb.* Keďže rozsah jedného testu je malý, ak test zlyhá, prepracovanie kódu je jednoduchšie. Pri tvorbe testov až po napísaní kódu majú programátori tendenciu napísať väčší celok a až potom test. V prípade ak test zlyhá je potom potrebné prepracovať viac kódu.

Verím však, že existuje ešte jeden dôvod zvýšenej produktivity. Tvorba testov je niekedy príliš komplikovaná. Častokrát vyžaduje testovaný kód externé závislosti a zinicializované iné časti kódu. To aké ťažké bude kód otestovať sa dá výrazne ovplyvniť už návrhom rozhraní („design for test"). Ak tvoríme najprv test, musíme sa týmito problémami zaoberať a vyriešiť ich skôr ako vznikne samotný kód. Ak však začneme písať test až po napísaní kódu, môžeme naraziť na problémy pri tvorbe testu, ktoré by sa dali vyriešiť lepším návrhom. Takáto situácia väčšinou vedie k vynechaniu testu, alebo prepracovaniu pôvodného kódu, čo sa v konečnom dôsledku prejaví stratou produktivity.

Myslím, že z hľadiska kvality softvéru nakoniec nezáleží na tom, či bol test napísaný pred, alebo až po samotnom kóde. Dôležité je, že nejaký test existuje, a že je možné ho automaticky spúšťať. Je dôležité, aby si každý programátor vybral taký prístup, aký mu vyhovuje. Pre niekoho môže byť psychologická bariéra „test-first" vývoja taká silná, že nútením sa do jej použitia nezíska, ale stratí.

Integrovať, integrovať, integrovať

Časy, kedy softvér tvoril jeden človek sú už dávno preč. Tvorba softvéru je komplikovaná činnosť, ktorá vyžaduje zapojenie viacerých ľudí, rovnako ani softvér nie je jeden monolitický celok, ale skladá sa z viacerých nezávislých modulov. Tento fakt je potrebné zohľadniť aj pri procese vývoja. Ak by sa všetky moduly vyvíjali a testovali nezávisle, pri finálnej integrácii a testovaní by zrejme nastali problémy. V záujme zvýšenia kvality softvéru a zníženia času potrebného na vývoj je preto

potrebné, aby sa jednotlivé moduly integrovali čo najčastejšie. Takáto technika sa označuje ako neustála integrácia[2].

Myšlienka neustálej integrácie spočíva v tom, že členovia tímu integrujú svoju prácu v pravidelných intervaloch niekoľkokrát za deň. Veľkosť intervalov nie je presne definovaná, musí však byť dostatočne malá na to, aby sa chyby odhalili včas. Fowler vo svojom článku [4] odporúča interval desiatich minút.

Integrácia spočíva v skompilovaní a zostavení celého produktu a spustení sady jednotkových a integračných testov. Podľa Fowlera [4] je pre nasadenie neustálej integrácie potrebné:

- Udržiavať všetok kód v jednom, spoločnom repozitári. Všetci vývojári do tohto úložiska zapisujú a všetci si voči nemu synchronizujú kód.
- Zabezpečiť automatizované zostavovanie produktu – to umožní CI nástroju rýchlo zostaviť produkt bez manuálneho zásahu
- Zabezpečiť automatizované testy; bez nich nemá CI význam, súčasťou CI procesu musí byť overenie kvality
- Zabezpečiť časté nahrávanie kódu do repozitára. Je to kritickou súčasťou celého procesu. Produkt môže úspešne prejsť všetkými testami na pracovnej stanici vývojára, je však možné že medzitým iný programátor zmenil časť kódu a nastal konflikt. Ak vývojári nahrávajú často, konflikt je odhalený skoro; v opačnom prípade zostáva skrytý dlhú dobu, čo v dôsledku len sťažuje hľadanie chyby.
- Zabezpečiť krátku dobu zostavovania – cieľom CI je poskytovať rýchle informácie o stave produktu a to je možné len vtedy ak zostavenie produktu a testovanie bude dostatočne rýchle.

Ukazuje sa, že projekty, ktoré používajú neustálu integráciu obsahujú dramaticky menej chýb [4]. Tento efekt je však priamo viazaný na kvalitu sady testov. Vo všeobecnosti platí, že nie je zložité vyrobiť lepšiu sadu testov, s ktorou sa dajú dosiahnuť lepšie výsledky. Zvyčajne však tímu chvíľu trvá kým dosiahne takú sadu testov, ktorá umožňuje odchytiť dosiahnuteľné chyby. Vyžaduje si to neustálu prácu na sade testov a jej zlepšovanie.

Ani neustála integrácia nie je liekom na chyby. Nedokáže nás chýb zbaviť, umožňuje ich však odhaliť dramaticky rýchlejšie. V tomto smere sa veľmi podobá na jednotkové testy. Ak sa do kódu zavedie chyba a podarí sa ju skoro odhaliť, je omnoho jednoduchšie zbaviť sa jej. Keďže bola zmenená iba malá časť systému, pri hľadaní chyby stačí zvažovať len malú časť kódu. Navyše, chyba sa vyskytuje v kóde, ktorý bol naposledy menený, takže dotknutý kód má programátor čerstvý v pamäti – čo znovu uľahčuje jej odstránenie.

[2] angl. continuous integration

Ktorú z techník teda použiť?

Každá z techník opísaných v tejto eseji pomáha zvyšovať kvalitu výsledného produktu tým, že pomáha predchádzať chybám a odstraňovať ich. Neexistuje však jednoduchá „strieborná guľka“, zázračná technika, ktorá by nám pomohla chybám úplne predísť, alebo ich všetky odhaliť. Pre zabezpečenie kvality softvéru nám tak nezostáva nič iné, ako kombinovať všetky techniky. Každá z nich je totiž špecifická a pomáha odstraňovať špecifický typ chýb a zvyšovať tým kvalitu.

Myslím, že je v záujme všetkých zainteresovaných, aby sa pri overovaní kvality nespoliehali len na záverečné testovanie. Kvalita je citlivá vec; nemožno ju zanedbávať a dúfať že sa o seba sama postará. Nepostará! Treba ju pravidelne overovať a merať a v prípade nepriaznivých výsledkov podniknúť nápravu. Táta snaha by však nemala byť len vecou špecializovaného tímu zodpovedného za zaistenie kvality. Zodpovednosť je na pleciach každého člena tímu. S využitím tu opísaných techník môžu ku kvalite prispieť programátori, je ale úlohou ich manažérov, aby tieto techniky zaviedli do procesu vývoja. Verím, že to nie je ľahká cesta, zavedenie každej novej techniky, ktorá na prvý pohľad znamená len prácu navyše sa stretne s odporom. Je to však cesta, ktorá je hodná toho, aby sme sa na ňu vydali. Na jej konci totiž čaká kvalitnejší softvér a väčšia produktivita. Vynaložená námaha sa vráti niekoľkonásobne.

Ak sa vrátime späť k pôvodnej otázke, ako vníma používateľ kvalitu, zistíme, že táto esej odpovedá len na polovicu otázky – ako zabezpečiť stabilitu softvéru. Na druhú polovicu treba hľadať odpoveď najmä v lepšej komunikácii s používateľmi; tak aby sme tvorili nielen stabilný softvér, ale najmä *použiteľný* softvér. Taký, ktorý bude pomáhať a bude sa na neho dať spoľahnúť.

Použitá literatúra

1. Beck K., Gamma E.: *Test infected: Programmers love writing tests.* http://members.pingnet.ch/gamma/junit.htm [cit. 17.10.2008]
2. Denning, P.J.: *Editorial: What is software quality?* Commun. ACM 35*,* 1 (Jan. 1992), 13-15.
3. Erdogmus H., Morisio M., Torchiano M.: *On the effectivness of the test-first approach to programming.* Proceedings of the IEEE Transactions on Software Engineering, 31(1). January 2005.
4. Fowler M.: *Continuous integration.* 2006 http://martinfowler.com/articles/continuousIntegration.html [cit. 17.10.2008]
5. McConnel, S: *Code Complete: A Practical Handbook of Software Construction.* Microsoft Press (Jul. 2004)
6. Watts S. Humphrey: *Introduction to the Personal Software Process*. Addison-Wesley, Pearson Education, Inc. 2002:159-163
7. Wiegers, K.E.: *Improving quality through software inspections.* Software Development, vol. 3, no. 4 (April 1995)

http://www.processimpact.com/articles/inspects.html [cit. 17.10.2008]

8. Wiegers, K.E.: *Seven Truths About Peer Reviews*. Cutter IT Journal, vol. 15, no. 7 (July 2002)
   http://www.processimpact.com/articles/seven_truths.html, [cit. 17.10.2008]

Annotation

Techniques for software quality assurance and improvement

Although agile development methods are becoming more recognized among developers and managers, there are still many projects following a waterfall development model. Testing takes place only after the implementation is over, at the time when bugs are being fixed and the resulting functionality, which is a synonym of quality from the customer's point of view is hard to change. Even the acceptance testing discovers many bugs, which could have been fixed at the time of development. Quality software originates in implementation, it is thus important that it is also regularly assessed during that phase. But how can we improve software quality and decrease error rate? How to avoid unpleasant surprises during integration phase? This essay tries to answer these questions by leveraging code reviews and various types of automated tests and their regular execution on integration server (known as continuous integration).