UDRŽANIE KVALITY ZA POCHODU

Kde nie je chyba, nie je kvalita.

Marián Hönsch

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

Abstrakt. *Meniace sa podmienky sú častým úkazom pri tvorbe softvéru v tíme. Zmena špecifikácie či ďalšie želania zákazníka nás nesmú prekvapiť. Je to fakt, s ktorým sa musí každý vývojár a manažér stotožniť. Časy, kedy sa testovanie nechávalo ako posledná fáza vývoja, už nie sú aktuálne. Kladieme dôraz na to, aby sme chyby podchytili v skorom štádiu. Takýto objav nám poskytne neoceniteľný prínos ku kvalite produktu. V softvérovom vývoji dlhšie tvoríme postupy ako zabezpečiť kvalitu už od inicializácie projektu a súčasne sa vyrovnať stálym zmenám prostredia. V tejto eseji sa pozerám práve na možnosti ako zabezpečiť a udržať kvalitu projektu v malom tíme od začiatku až po koniec. Vyberiem vhodné postupy testovania a poskytnem ich hodnotenie ako sa dokážu uplatniť v malom tíme. Medzi ne patria napríklad metóda stálej integrácie, automatické testy a vývoj riadený testami. Zhodnotím správnu disciplínu a prístup vývojára pri tvorbe kvalitného produktu. Zamyslím sa nad možným vývojom testovania do budúcnosti.*

Kľúčové slová: *kvalita, test, testovanie v malom tíme, disciplína, stála integrácia, automatický test, agilný vývoj, vývoj riadený testom, in vivo*

Kde nie je chyba, nie je kvalita

Chyby vznikajú pri každej činnosti človeka. Už od momentu, keď sa vzpriamil a možno aj skôr, robí pri svojej činnosti chyby. Je to viac ako prirodzené a inak tomu nie je ani pri tvorbe softvéru. Ak sa to pokúsime aspoň okrajovo pochopiť, môžeme tvorbe chýb

efektívne predísť alebo ich úspešne odhaliť. Teda postupy na obmedzenie nedostatkov môžeme rozdeliť na tie, ktoré chybám určitými opatreniami predchádzajú a tie, ktoré ich odhalia. Môžeme sa pýtať, či nám bude stačiť veľká množina opatrení, ktorá znemožní vznik akejkoľvek chyby. Ak by tieto opatrenia pochádzali z ľudskej dielne, sme opäť pri tvrdení: ľudia robia chyby. Vznikne začarovaný kruh a prelomíme ho testovaním našich výtvorov.

Môžeme sa teda spoľahnúť, že nejaká chybička vznikne vždy. Nie je to však žiadna dráma. Horšie bude, ak pri testovaní objavíme len menší počet chýb relatívny k veľkosti projektu. To by totiž znamenalo, že tam niekde sú ešte ukryté. Preto sa chýb netreba báť, sú súčasťou nášho života. Isteže ich neradi vidíme a hlavne, ak sme ich spôsobili. Dokážu nás znechutiť a vziať chuť do ďalšej práce. Avšak nikto nie je bezchybný. Osobný strach z chýb musíme prekonať sami a aby sme ich obmedzili pri našej práci, zvolíme správne techniky na zlepšenie kvality.

Vplyv na kvalitu softvéru má správny prístup k vývoju softvéru, spolupráca v tíme a v neposlednom rade aj jednotlivec. Takto rozdelím metodiku na odhalenie a predchádzanie chýb. Na testovanie sa budem pozerať ako na možnosť jednotlivca zlepšiť kvalitu softvéru. Pôjde o nasledovné metódy:

- Stála integrácia projektu
- Prehľad kódu, párové programovanie
- Agilný vývoj a vývoj riadený testom
- Jednotkové, integračné automatické a „in vivo“ testovanie

Disciplinovanosť tímu, ale aj jednotlivca je tiež jeden z vplyvných faktorov. Takto tomu je nie len pri tvorbe softvéru, ale aj pri stavbe mosta, či operácii slepého čreva. Dovolím si tiež poukázať na techniky zabezpečenia kvality v minulosti a na sny budúcnosti.

Čo nás naučila história

V skorých štádiách softvérového vývoja sa zvolil praktický prístup odvodený od iných inžinierskych odvetí. Proces zlepšenia kvality bol zadefinovaný ako cyklus aktivít: plánuj, rob, testuj a vyhodnoť [10]. Vo svojej podstate tento proces používame dodnes, i keď prešiel výrazným vývojom.

Deming vo svojej teórii poukázal, že pri kvalitnom produkte vynaložíme menej úsilia na jeho údržbu. Opisuje jav, v ktorom údržba nekvalitného produktu spôsobuje negatívny postoj zákazníka k nemu, kde naopak kvalitný produkt pomôže zväčšiť výrobcov podiel na trhu [10]. Teda námaha, ktorú investujeme pri vývoji kvalitného produktu je omnoho menšia ako problémy, ktoré dokáže spôsobiť menej kvalitný produkt na trhu. Platí táto skutočnosť aj pri niečom inom ako je softvér? Samozrejme, ak si chceme budovať značku a udržať sa na trhu dlhší čas.

Crosby prišiel s myšlienkou, že kvalita je zadarmo. Tento prístup môžeme charakterizovať heslom: „Robme veci správne od začiatku“. Uvádzal vývoj, pri ktorom by nemali vznikať žiadne chyby. Ako hlavným zdrojom chýb uvádza nedostatok vedomosti alebo nedostatok záujmu od personálu [10].

Ak si tento vývoj premietneme do dnešných metodológií zabezpečenia kvality uvidíme určité prepojenia. Hlavné myšlienky Crosbyho pripomínajú vývoj riadený testom

a agilný vývoj. V skutočnosti tieto dva prístupy v softvérovom inžinierstve nie sú nové. Hlavne vývoj riadený testom sa propagoval aj v minulom storočí, len jeho nasadenie muselo dozrieť. Podobne vznikali aj metodiky, kde sa menej dbalo na formálny proces vývoja a namiesto toho za zdôrazňovala osobnosť programátora a komunikácia v tíme. Môžeme tvrdiť, že vývoj v softvérovom inžinierstve smeruje k metodike, ktorá zabezpečí vyššiu kvalitu softvéru. Len je na zváženie nakoľko sú programátori a prostredie pripravené tieto metodiky adaptovať. Pravdepodobne tlak trhu a zákazníka bude prirodzene smerovať k tomu, aby sa takto udialo postupne, ale čím skôr.

Nechcem aby vyznelo, že programátor nechce zlepšiť kvalitu, lebo je lenivý, skôr si myslím, že veľa ľudí nevie ako presne na to. Takto to bolo aj v minulosti, keď sa vyvinuli lepšie nástroje na obrábanie pôdy, skvalitnila sa aj výsledná úroda. Nahliadnime kam speje vývoj testovania.

Vízia v testovaní

Podľa Bertolinovej sny a vízie v testovaní môžeme zhrnúť do štyroch nasledujúcich bodov [2]:

1. Univerzálna testovacia teória
2. Testovaním založené modelovanie
3. 100 % automatické testy
4. Maximálne efektívne testovanie

Jednotlivé kroky na seba nadväzujú. Prvý krok je zlúčiť a normovať všetky testovacie techniky a vytvoriť jednotný prístup k testovaniu. Ďalší krok zahŕňa vývoj modelov pre testovanie. Namiesto otázok, ako najlepšie otestovať tento model, sa spýtame, ako navrhnúť tento model, aby bol najlepšie testovateľný. V [2] je tento jav opísaný ako prechod od „model-based testing“ ku „test-based modeling“. Ak by sa nám podarilo takéto modely preniesť do praxe, ďalším krokom by bolo odvodiť od týchto modelov automatické testy. Môžeme to označiť ako testovanie podľa vzorov. Aplikácia by bola sto percentne pokrytá automatickými testami. Testy by už neboli vykonávané separátne, ale boli by aktívnou súčasťou aplikácie. Posledným krokom a najvzdialenejším snom je tvorba vysoko kvalitného softvéru s minimalizáciou všetkých známych rizík.

Opísaný postup predpokladá, že softvérové inžinierstvo bude v ďalších generáciách spieť k univerzálnosti a prototypovateľnosti. Z príkladu, ako sa vyvíjali iné vedné disciplíny, môžeme usúdiť, že nastane podobný vývoj. Do bližšej budúcnosti však predpokladám, že testovanie sa bude stávať centrálnejším prvkom vývoja. V mnohých prípadoch sa to stáva. Avšak to nie je všetko. Viac vynaloženej námahy na testovanie nemusí zabezpečiť vždy v rovnakej miere kvalitnejší produkt. Nakoniec zvíťazia techniky, ktoré dokážu pri nižšej námahe na testovanie poskytnúť lepšie výsledky. Tieto zároveň určia ďalší smer vývoja v testovaní.

Minulosť a sny budúcnosti nám nezlepšia kvalitu softvéru. Pozrime sa teda, aké možnosti máme dnes.

Čo dokáže správny prístup

Ako pri každej práci, tak aj pri vývoji softvéru si musíme zvoliť svoj „plán boja“. Ako programátori vieme, že existuje viacero prístupov. Cieľom každého z nás je, aby na konci vznikol softvér alebo aspoň by mal. Čím väčší je tím a projekt, tým sa detaily, prípadne výhody a nevýhody daného prístupu prejavia vo väčšom rozpätí. Univerzálny prístup k tvorbe softvéru sme ešte nenašli. Niektoré umožňujú lepšie prototypovanie, iné pomalé dodávanie častí systému. Navzájom sa dopĺňajú, ale aj vylučujú. Samozrejme existuje prístup ako je vývoj riadený testom a agilné programovanie, ktoré tlačia do svojho popredia kvalitu softvéru. Prečo teda každý programátor na svete takto nepracuje a svojej firme nezabezpečí kvalitný softvér, ktorý ovládne celý trh?

Stručne sa tento vývoj dá opísať v štyroch krokoch. Prijmi požiadavku od zákazníka, navrhni test, implementuj, testuj a opravuj, kým program neprejde testom. Problémom je, že ak sa v poslednom cykle zasekneme pri hľadaní chyby, náklady spojené s touto požiadavkou prevýšia náklady pri inom vývojovom modeli [7]. Tu môžeme argumentovať, že náklady na vývoj riadený testom sú prirodzene vyššie, ale vráti sa nám to omnoho viac, ak produkt bude kvalitný. To je pravda, ale náš produkt nemusí ešte stále uspieť na trhu z iných dôvodov. Niekto vyvinie niečo krajšie a rýchlejšie a náš projektový manažér stratí prácu, lebo na vývoj investoval priveľa.

Zábranou nie sú iba vyššie náklady. Kto z nás rád skúša niečo nové, čo chce veľa pozornosti a urobí to dobrovoľne. V [11] autori opisujú, ako sa zavádzal vývoj riadený testom v stredne veľkom tíme pri tvorbe mobilnej aplikácie. Opisujú jav, že kód široko pokrytý testami je porovnateľný s dobre štruktúrovaným kódom od skúseného programátora. Taktiež postup, najskôr vyviň test, menej skúseným programátorom nepomohol vytvoriť kód oveľa kvalitnejšie. Veľa programátorov pocítilo rozčarovanie z počiatočného entuziazmu o tom, ako testom riadený vývoj zlepší ich výsledok. Príliš dogmaticky aplikovali prístup a nechali sa pohltiť drobnými testami a zažili „burnout“, keď po niekoľkých týždňoch dospeli k výsledku, že ich napredovanie je omnoho menšie ako pri overených metódach. Neskôr testovanie potláčali viac a viac do úzadia a k celej metodike si vybudovali negatívny prístup.

Isteže, všetky tieto nedostatky vznikali aj preto, lebo sa prístup aplikoval prvýkrát. Pri druhom a treťom opakovaní nežiaduce efekty oslabnú. Koniec koncov, ak programátor získa potrebné skúsenosti pri vývoji riadenom testom a prakticky ich aplikuje, tieto efekty naozaj stratia silu. Projekt z opisovaného testu bol úspešný, ale váha úspešnosti sa nepripisovala novej zázračnej metóde, ale skôr odhodlaniu programátorov testovanie vpustiť do ich každodennej práce.

Osobne si myslím, že programátor ako každý iný človek je pohodlný a zmenu robí, keď motivácia je podstatne vyššia ako námaha. Inými slovami, vývoj riadený testom nie je natoľko vyvinutý, aby bol široko akceptovaný ako lepšie riešenie. Veľa ľudí priznáva, že je atraktívne to skúsiť, ale ku konkrétnym krokom chýba odhodlanie. Na druhej strane, jeho vylepšenie si vyžaduje viac skúsenosti z praxe. Je to nepochybne prístup, ktorý sa ešte dožije svojho uznania.

Čo dokáže tím

Práca v tíme je ako lov mamutov. Ak chceme to isté a ťaháme za rovnaký koniec, môžeme dosiahnuť svoj cieľ. Jednotlivec nemá nikdy toľko možností ako skupina. V našom prípade platí, cieľ nie je vždy to najdôležitejšie, ale cesta. Zhodnotím teda aké výhody môže mať spolupráca pri zvýšení kvality a čo sa nesmie zanedbať.

Spoločná práca

Systémový architekt rozdelí produkt do menších častí a programátori sa pustia do svojej práce. Po piatich mesiacoch intenzívneho tvorenia všetky zdrojové kódy spoja do jedného celku. Vďaka dômyselnej architektúre a disciplinovanému prístupu všetko do seba zapadá a projekt prejde do ďalšej fázy. Ak ste sa pri tomto opise úplné zarazili, je to určite oprávnené.

Čiastočné výsledky každodennej práce musíme združovať do celku, ak nechceme zažiť nepríjemne prekvapenia. Keď programátor píše svoje riadky, tak raz začas kompiluje svojho kódu. Prostredie mu vzápätí ohlási koľko preklepov, syntaktických chýb a nesprávnych referencií stihol zabudovať. Na opravu týchto chýb nečaká do konca projektu, ale čím skôr to urobí. Ako aplikovať tento jednoduchý a účinný princíp do praxe? Odpoveďou je metóda stálej integrácie.

Stála integrácia znamená pravidelne integrovať zdrojový kód od členov vývojového tímu na centrálne úložisko. Každá integrácia je overená automatickým zostavením programu, a tak sa okamžite objaví veľké množstvo integračných chýb [5]. V praxi to znamená zostavenie servera pre kontrolu zdrojového kódu. Jednotliví vývojári pravidelne aspoň raz za deň uložia časť svojho „funkčného“ zdrojového kódu na server. Server spustí vzápätí, alebo neskôr, raz za noc celú integráciu častí a zostaví produkt. Prípadne nezostavenie je jasným náznakom nedostatkov integrácii, no úspešné zostavenie je len nutnou podmienkou bezchybného produktu. Na tomto mieste sa priam ponúka vykonať automatické testy na overenie výsledku. Jediná ďalšia námaha spojená s touto technikou je pre programátora potreba si pravidelne sťahovať ostatné časti produktu od kolegov do svojho lokálneho úložiska.

Tento prístup je asi jeden z najefektívnejších prostriedkov ako predísť riziku chýb pri integrácii zdrojového kódu. Je to asi najlacnejší prostriedok spojený s minimálnou námahou, ktorý môže tím využiť pri svojom vývoji. Umožňuje mať vždy poruke spustiteľnú a otestovanú verziu produktu, čo nám zároveň dovolí častejšie podávať výsledky zákazníkom.

Týmto spôsobom vieme technicky podporiť vývoj v tíme a zlepšiť výslednú kvalitu. Dokáže aj kolega , ktorý sedí oproti mne to isté?

Pomoc od kolegov

Každý pracovný tím tvorí aj sociálnu komunitu. Nebudem sa zaoberať týmito zákonitosťami, len chcem povedať, že skúsenejší poradí menej skúseným. Ak by tomu tak nebolo, tak tento tím už pravdepodobne nebude produktívny. Ale ako tieto rady a ochotu využiť čo najlepšie? Asi aj ten najtrpezlivejší a kamarátsky kolega stratí trpezlivosť, ak ho príliš zaťažíme.

Pri nekomerčných projektoch označovaných taktiež „Open Source“ je typické, že komunita robí pred vydaním novej verzie produktu revíziu kódu. Práca na týchto projektoch je zväčša zadarmo a preto motivácia poskytnúť svoj kód a robiť revíziu je bezúhonná. Je to podstate dôkaz, že komunita si vie takto zlepšiť kvalitu produktu sama [8]. V komerčnej sfére sa takéto revízie robia tiež. Fakt, že sa reviduje časť zaplatenej práce, prináša so sebou pár negatívnych vlastností.

Samozrejme je to vec prístupu k revízii a postoj manažmentu. Revíziu môžeme postaviť ako spôsob ako sa môžeme naučiť od kolegu jeho overené postupy pri kódovaní alebo ako hľadanie vinníka. Jedna aj druhá situácia je samozrejme extrém, ktorý v praxi nenájdeme. Ale asi súhlasíte, že revízie, ktoré sa držia v prvom duchu, zachovajú tím aj do ďalších projektov. Keďže objektom revízie je práca jednotlivca, ktorý cíti prepojenie so svojou prácou, musíme toto prepojenie zohľadniť. Programátor si musí uvedomiť, že sa hodnotí jeho kód a nie jeho osobnosť. Hodnotiaci sa sústredí, aby sa jeho komentár týkal kódu a nie programátora [4]. Toto pravidlo treba zohľadniť asi len pri objavení nedostatkov, v opačnom prípade treba programátora pochváliť za dobrú prácu.

Pri revízii sa nepozerá len na funkčné vlastnosti kódu, ale aj jeho formálnu stránku a čitateľnosť. Čítanie cudzieho netriviálneho kódu je asi jedna z najnepríjemnejších činností aké poznám. Programátor musí pred revíziou okomentovať svoju prácu a dôkladne preveriť, či dodržiava interné pravidlá kódovania. Inak hodnotiaci nemá šancu preniknúť do kódu a jeho spätná väzba bude príliš všeobecná. Je plytvanie času, ak by prehliadajúci musel opravovať štruktúru kódu, aby súhlasila s dohodnutými konvenciami. Túto stránku by mal zvládnuť aj menej skúsený programátor sám. Takéto opatrenia pomôžu inému programátorovi opraviť alebo dotvoriť jeho prácu v neskorších fázach.

Zaujímavým úkazom pri revízii je, že sa nebáda po chybných výsledkoch alebo náznakoch chybného správania, ale priamo sa vyhodnotia použité algoritmy v kontexte celého programu. Takúto techniku zlepšenia kvality nám neposkytne žiadna strojová forma. Preto revízia kódu iným programátorom musí patriť do techník zabezpečenia kvality pri práci v tíme.

Určitou formou revízie je párové programovanie [1]. Túto techniku môžeme zaradiť aj medzi extrémne programovanie. Pri tomto modeli ako aj pri revízii kódu platí heslo „viac očí viac vidí“. Princíp je jednouchý. Dvaja programátori, jeden programuje druhý naviguje a potom sa vymenia. Zo skúsenosti z praxe sa jeden od druhého veľa naučia a vzniknutý kód je objektívne kvalitnejší [1]. Je to metóda ako zaučiť nováčika. Produkuje čitateľnejší a kvalitnejší kód a súčasne je pár produktívnejší ako jednotlivec [6].

Takáto práca mi však pripadá ako programovanie na ihlách. Určite má zmysel dočasne vytvoriť takéto páry, ktoré vyriešia danú úlohu. V skorých fázach projektu, kedy sa poukladajú základné kamene, to vylepší a objasní ďalšie napredovanie implementácie. Pri kritických alebo nejasných častiach zabezpečí kvalitnejšiu implementáciu. Avšak pretrvávanie v takomto režime si neviem predstaviť. Programovanie vidím ako určitý druh umenia, a preto aj samostatná práca je nevyhnutná pri tvorbe kvalitného softvéru. Niektoré veci si treba s rozvahou premyslieť. Druhá strana sa potom zíde pri konzultácii výsledku ako na samotný postup riešenia. Taktiež faktor učenia s pribúdajúcim časom opadne, páry sa môžu striedať. Avšak raz nastane moment kedy sa začne od začiatku.

Čo dokáže jednotlivec

Ako dokáže programátor pri každodennej práci ovplyvniť kvalitu? Asi najviac, správny prístup k práci a spolupráca s kolegami vytvoria správne prostredie, kde sa dbá na kvalitu softvéru. Programátor však priamo pôsobí na produkt, keďže je to jeho kód, ktorý sa bude spúšťať. Toto odvodenie nie je nové a platí aj pri výrobe „hodiniek", je ale dôležité poukázať, že celý koncept udržania kvality súvisí s osobným nasadením programátora.

ABC Programátora

Ako som už spomínal skôr, programátor je v mojich očiach druhom umelca. Ako každý umelec má svoj vlastný štýl tvorby. Je zaujímavé, že v prostredí ako je informatika, ktoré navonok pôsobí diferencovane a štruktúrovane, existuje miesto pre umenie. Ak sa pozerám na obrazy v galérii, myslím si, že niektorým rozumiem, iným zasa nie. Toto si ale v projekte dovoliť nemôžeme. Aj keď kód je duševným dielom, mal by byť čitateľný pre každého iného programátora. Vznikajú vo firmách interné protokoly a návody ako kódovať. Správne označenie tried, premenných a metód. Začiatočníci si pomyslia, že tento kód je jednoduchý, určite v ňom nie je chyba a prečo by ho mal niekto niekedy čítať. Tu by trebalo poukázať na chybné zmýšľanie. Dodržiavanie určených konvencií nielen zlepší čitateľnosť kódu, ale aj psychologický aspekt štruktúrovaného prístupu vedie programátora premyslieť si dôkladnejšie svoje kroky.

Disciplína pri práci má väčšinou pozitívny vplyv na výsledok. Môžeme sa pýtať, či dodržiavanie pravidiel nespútava tvorivého ducha programátora a tým pádom je viac na škodu. Ja by som tvrdil opak, ale môže byť na tom niečo pravdy. Dodržiavanie formálnych pravidiel ako sú konvencie na pomenovanie objektov dokážeme aj statickou kontrolou kódu [2]. Takéto odbremenenie programátora však nemusí znamenať hneď lepšie výsledky. Stále ostáva veľká požiadavka pre disciplinovaný prístup k svojej práci.

Jedným aspektom disciplinovaného prístupu je aj po sebe upratať. V jazyku programátorov to môže znamenať otestovať či to naozaj robí čo má. Prejdime ku poslednej ale asi najdôležitejšej technike udržania kvality: testovanie.

Test od smietka po mrakodrap

Metodológia testovania nám poskytuje testy v rôznej granularite. Záleží od programátora, ktoré si zvolí. Samotné testovanie môže prebiehať automaticky alebo iným špecializovaným kolegom, ale návrh testu by mal pochádzať od autora kódu. Aký test zvoliť? Povedal by som, že záleží od projektu a jeho budúceho nasadenia. Ale ak práve nepracujeme pre armádu, alebo nevyvíjame firmware pre mikročipy je lepšie siahnuť po teste, ktorý namiesto vymedzenej funkcionality testuje konkrétny prípad použitia. Automatické testy sú pri tomto postupe jasnou výhodou. Napíšeme ich raz a spustia sa pri každej zmene kódu na overenie funkcionality. No netreba sa nechať oklamať zdaním bezchybného kódu. Pri tvorbe automatických testov je asi najťažšie formalizovať priebeh testu a správne určiť množinu vstupov. Tu sa ponúka spôsob náhodného testovania. Táto metodika opisuje testovanie sekvencií s náhodne generovaným vstupom. Takto odhalíme chyby a scenáre, ktoré programátor ani nepredpokladal.

Naše laboratórium otestuje všetky scenáre, prípady použitia a vytrápi sa s automatickými testami. Produkt sa dostane do série a po týždni sa zákazník pýta, čo sme

mu to odovzdali, veď to nejde. Naše testy neukazujú žiadnu chybu. Čo teraz? Dá sa testovať aj u zákazníka? Metóda „in vivo“ presne opisuje takýto postup [9].

Táto testovacia technika môže byť použitá na odhalenie chýb u zákazníka pri:

- neočakávaných stavoch aplikácie
- netestovaných konfiguráciách systému
- nepredpokladaných krokoch používateľa

Všetky body pokrývajú situácie, na ktoré sme nemysleli alebo sme nedokázali myslieť pri testovaní. Treba dodať, že tento postup je viac ako statické monitorovanie systému. Je to aktívne testovanie vybraných častí kódu aplikácie u zákazníka. Samotný test zákazník nespozoruje a ani nezmení stav aplikácie. Počas priebehu, keď zákazník zavolá funkciu, ktorá je pokrytá testom, s určitou pravdepodobnosťou sa spustí v separátnom procese test. Jeho výsledok sa uloží a aplikácia pokračuje ďalej [9]. To znamená, že kód sa testuje viac krát, aby sa otestovala jeho funkcionalita a stabilita pri rôznych vstupných parametroch a meniacich sa východiskových stavoch aplikácie. Preto je potrebné testom pokryť len netriviálne časti aplikácie.

Táto technika je medzi prvými, ktoré sa snažia testovať aplikáciu v aktívnych podmienkach v poli. Testovanie aplikácie u zákazníka je identifikované ako jedna z výziev testovacej komunity do budúcnosti [3]. Jej priame nasadenie si vyžaduje úpravu kódu a umožniť vyčítať uložené testovacie záznamy. Tieto akcie si vyžadujú priveľké zapojenie programátora. Je to však sľubná technika a technickú realizáciu by mohlo umožniť priamo aj vývojové prostredie. Programátor by vytvoril funkciu a prostredie by pokrylo túto funkciu testom a pripravilo na testovanie v poli.

Čo sa oplatí robiť?

Vymenoval a hodnotil som rôzne postupy a techniky, ktoré dokážu zlepšiť kvalitu počas celého vývoja softvéru a aj potom. Sami o sebe nemajú taký veľký vplyv ako keď ich dokáže správne kombinovať a previesť do praxe. Pre malý tím, ktorý sa práve spoznal, je vhodné vyberať techniky, ktoré neznamenajú príliš veľkú zmenu pre jednotlivcov a ako sú zvyknutí pracovať, lebo inak by mohli ohroziť stabilitu projektu. Medzi tieto techniky patrí napríklad metóda stálej integrácie alebo revízia kódu.

Asi najlepšou technikou zabezpečenia kvality je správny osobný prístup programátora. Ak jednotlivci nemajú záujem o udržanie kvality, tak ani overené postupy celkový priemer výrazne nezlepšia. Treba si uvedomiť, že za kvalitu nie je zodpovedný projektový manažér alebo manažér kvality, ale každý člen tímu.

Použitá literatúra

1. Atwood, J.: Pair Programming vs. Code Reviews. *Coding Horror* [online]. 2009 [cit. 2009-10-12].
   Dostupný z WWW: < http://www.codinghorror.com/blog/archives/000999.html>.
2. Ayewah, N., Hovemeyer, D., "Using Static Analysis to Find Bugs," *IEEE Software*, vol. 25, no. 5, pp. 22-29, September/October, 2008.

3. Bertolino, A.: "Software Testing Research: Achievements, Challenges, Dreams," *Future of Software Engineering*, pp. 85-103, Future of Software Engineering (FOSE '07), 2007.
4. Bogue, R: Effective Code Reviews Without the Pain. Developer.com [online]. 2006 [cit. 2009-10-12].
   Dostupný z WWW: <http://www.developer.com/tech/article.php/3579756/Effective-Code-Reviews-Without-the-Pain.htm>.
5. Fowler, M: Continuous Integration. XP and Agile Methods [online]. 2006 [cit. 2009-10-12].
   Dostupný z WWW: <http://martinfowler.com/articles/continuousIntegration.html>.
6. Hulkko, H., Abrahamsson, P.: A multiple case study on the impact of pair programming on product quality. *In Proceedings of the 27th international Conference on Software Engineering* (St. Louis, MO, USA, May 15 - 21, 2005).
7. Karamat, T., Jamil, A. N. "Reducing Test Cost and Improving Documentation In TDD (Test Driven Development)," *Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing, International Conference on & Self-Assembling Wireless Networks*, International Workshop on, pp. 73-76, Seventh ACIS International Conference on Software Engineering, Artificial Intelligence, Networking, and Parallel/Distributed Computing (SNPD'06), 2006.
8. Lussier, S. "New Tricks: How Open Source Changed the Way My Team Works," *IEEE Software*, vol. 21, no. 1, pp. 68-72, January/February, 2004.
9. Murphy, CH., Kaiser, G., Vo, I., Chu, M.: "Quality Assurance of Software Applications Using the In Vivo Testing Approach," *Software Testing, Verification, and Validation, 2008 International Conference* on, pp. 111-120, 2009 International Conference on Software Testing Verification and Validation, 2009.
10. O' Regan, C. G.: *A practical approach to software quality*, 2002. s. 8-18.
11. Rendell, A.: "Effective and Pragmatic Test Driven Development," *AGILE Conference*, pp. 298-303, Agile 2008, 2008.

Annotation

Sustainable quality on the move

Changing environment is a common case in teamwork on a software project. Changing of specification or another customer wishes should not surprise us at all. It is a fact and every developer and manager should get used to it. Times, where testing was the last phase in software development, are long gone now. We prefer to track and correct bugs in early states of development. Such a discovery is an invaluable advantage to software quality. In software development we try our best to find the best practices to ensure quality from project start and at the same time deal with permanent environment changes. In this essay I focus on possibilities of a small team to ensure quality from project initialization. I sort out applicable test practices and their evaluation how they can ensure quality in a small team. To them belong the Continuous integration technique, automatic tests and test driven development. I evaluate the right attitude of a developer by creating quality software. Take a thought of possible evolution of testing in future.