ESEJE O MANAŽMENTE V SOFTVÉROVOM INŽINIERSTVE

Slovenská technická univerzita
Fakulta informatiky a informačných technológií

ESEJE O MANAŽMENTE V SOFTVÉROVOM INŽINIERSTVE

Editori a grafická úprava
Bc. Rastislav Bertušek
Bc. Marek Fučila

Slovenská technická univerzita
Fakulta informatiky a informačných technológií
2005

Editori a grafická úprava

Bc. Rastislav Bertušek

Bc. Marek Fučila

Prispievatelia

Bc. Rastislav Bertušek
Bc. Peter Dušek
Bc. Marek Fučila
Bc. Pavol Goňo
Bc. Martin Hinka
Bc. Martin Jenčo
Bc. Ľuboš Lečko
Bc. Michal Moravčík
Bc. Martin Niejadlik
Bc. Peter Orosi
Bc. Ján Šnirc
Bc. Bohuslav Szabo
Bc. Peter Trnovský
Bc. Miroslav Vnuk
Bc. Milan Žužo

Publikácia neprešla jazykovou úpravou.

Príhovor

Táto publikácia obsahuje eseje, ktoré vytvorili študenti na predmete Manažment v softvérovom inžinierstve na FIIT STU v školskom roku 2004/2005. Všetky eseje reagujú na aktuálne problémy softvérového inžinierstva a ponúkajú tak prehľad o súčasnom vývoji softvéru a jeho nedostatkoch. Niektoré popisujú samotný softvér, jeho formy, vlastnosti a kvalitu. Ostatné sa viac sústredia na proces tvorby softvérových produktov. Problémy vývoja týchto produktov sú analyzované z rôznych pohľadov a následne sa môžeme dozvedieť o spôsoboch ich riešenia. Rôzne prístupy sú stavané do protikladov na základe ich kladov a rizík, ktoré so sebou prinášajú. Ani jedna z esejí si nekladie za cieľ podať vyčerpávajúce informácie o téme, ktorú prezentuje. Napriek tomu veríme, že inšpirujú čitateľa, aby siahol po ďalších zdrojoch. Priblíži sa tak k objektívnemu pohľadu na súčasné problémy, s ktorými sa stretávajú organizácie vyvíjajúce softvér.

Eseje sme zoskupili podľa piatich rámcových tém, ktorým sa venujú. Začíname pohľadom na tvorbu a vlastníctvo softvéru, kde sa riešia dodávateľské vzťahy, komercia, otvorenosť a zdieľanie softvéru. Tvorba softvéru rozhodne nie je jednoduchá záležitosť, čomu nasvedčuje aj druhá skupina esejí, ktorá sa venuje problémom vývoja softvéru. Popisujú sa tu riziká tvorby projektov a príčiny úspechov a neúspechov vývoja. Nasledujú eseje o tom, ako znížiť náklady na vývoj a obstáť v konkurenčnom prostredí. Rieši sa tu zvyšovanie produktivity vlastných zamestnancov a outsourcing. Ďalšia časť rozoberá problematiku tímovej spolupráce. Dozvieme sa ako je možné zorganizovať a riadiť tímy tak, aby dosahovali požadované výsledky, ako predchádzať konfliktom, a ako ich riešiť v prípade ich vzniku. Nakoniec si priblížime oblasť kvality softvéru. Pôjde o popis pravidiel a spôsobov testovania, s následným určením zodpovedností za chyby v programoch a manažovaním kvality vývoja softvéru.

Ak vás prehľad zaujal, v rukách držíte tú správnu knižku. Ak nie, veríme, že sa pre vás stane hodnotná po prečítaní niekoľkých esejí. Za každou z nich nasledujú odkazy na zdroje, z ktorých autor čerpal. V prípade záujmu o získanie väčšieho množstva informácii k rozoberanej téme vám odporúčame siahnuť po uvedených odkazoch.

Prajeme vám príjemné čítanie!

Obsah

Príhovor ... 3

Obsah ... 5

Úvod ... 7

Vzťahy medzi členmi dodávateľského reťazca ... 11

Softvér a údaje, majetok pre zdieľanie ... 19

Komerčné verzus nekomerčné metódy vývoja ... 29

Udržovateľnosť a Open Source Software ... 37

Vplyv rizík softvérových projektov na výsledky projektu ... 45

Čo zaručuje úspech softvérového projektu ... 53

Zlepšenie produktivity práce softvérového tímu ... 61

Systematické výmeny zamestnancov ... 73

Obstarávanie v softvérových IT projektoch ... 79

Vývojové tímy v softvérovom inžinierstve ... 87

Prístupy k zdieľaniu informácií v distribuovaných projektoch ... 95

Manažment konfliktov medzi programátormi a testermi ... 103

Počítanie riadkov zdrojového kódu ... 111

Podvedomý prístup k testovaniu softvéru ... 117

Kto je zodpovedný za chyby a bezpečnostné slabiny v softvéri ? ... 125

Záver ... 135

Úvod

V súčasnosti už sa ľudstvo nezaobíde bez počítačov. Dalo by sa povedať, že je na nich závislé. Pre laikov je počítač len múdrou skrinkou s pripojeným monitorom, klávesnicou, myšou a inými perifériami. Pomáha im pri práci, no často ani netušia, že hardvér, ktorý pred sebou vidia, vlastne nie je to, čo pre nich počítač predstavuje. Ich počítačom je softvér, ktorý beží v plechovej schránke, ktorý vnímajú cez obraz na monitore, cez zvuk z reproduktorov, ktorý ovládajú myšou či klávesnicou. Softvér je to, čo im uľahčuje prácu, umožňuje zábavu či spôsobuje problémy. Vďaka softvéru sa počítače tak rozmohli. Softvér určuje ich užitočnosť. Softvér však nevzniká sám od seba, ani mávnutím čarovného prútika, ale ide o náročný proces vývoja, za ktorým stoja ľudia. Softvér je dielo softvérových inžinierov.

Na toto inžinierske dielo a proces jeho vývoja sa môžete na nasledujúcich stranách pozrieť očami autorov pätnástich esejí. Eseje zapadajú do piatich okruhov podľa príbuzností tém. Nasleduje krátky prehľad jednotlivých tém.

Tvorba a vlastníctvo softvéru

Softvérom sú programy pre počítače. Softvér nie je hmatateľný, hoci médiá, na ktorých je uložený áno. Je ho možné kopírovať bez poškodenia z média na médium, a tak programy ľubovoľne šíriť. Či už ide o šírenie vykonateľných kódov alebo zdrojových textov programov, je táto činnosť obmedzená platnou legislatívou. Programy je možné tvoriť len tak pre radosť, s nezištným úmyslom, alebo za účelom predaja. Predávať a kúpiť sa dajú programy, ktoré už niekto vyrobil alebo je možné si dať softvér zhotoviť na objednávku.

Eseje

Vzťahy medzi členmi dodávateľského reťazca
Softvér a údaje, majetok pre zdieľanie
Komerčné verzus nekomerčné metódy vývoja
Udržovateľnosť a Open Source Software

Problémy vývoja softvéru

Zvláštnosťou vývoja softvéru je veľmi vysoké percento neúspešných projektov. Vyplýva to pravdepodobne zo zložitosti problémov, ktoré má softvér riešiť, abstraktnosti samotných programov a z následného nedorozumenia medzi odberateľom a dodávateľskou softvérovou organizáciou. Preto je veľmi dôležité zvážiť

riziká tvorby softvérových projektov, pri čom nám môže napomôcť analýza úspechu nasadených a neúspechu nedokončených projektov.

Eseje

Vplyv rizík softvérových projektov na výsledky projektu
Čo zaručuje úspech softvérového projektu

Znižovanie nákladov

Vývoj komerčného softvéru je ekonomická činnosť. Tvorcovia softvéru musia svoj produkt predať, aby prežili. V konkurenčnom prostredí je snaha čo najviac znižovať náklady. Keďže pri vývoji softvéru sú najnákladnejšie ľudské zdroje, je nutné zvyšovať produktivitu vývojárov ako aj celých vývojových tímov. Niekedy je výhodné využiť cudzie zdroje a časť vývoja outsourcovať.

Eseje

Zlepšenie produktivity práce softvérového tímu.
Systematické výmeny zamestnancov
Obstarávanie v softvérových IT projektoch

Tímová spolupráca

Rozsiahlosť softvérových systémov si pri tvorbe vyžaduje spojiť väčší či menší počet vývojárov do tímov. Čím väčší systém sa vyvíja, tým dôležitejšie je správne zorganizovať tímy, aby tímová spolupráca priniesla svoje ovocie. Distribuované projekty si vyžadujú funkčné zdieľanie informácii medzi inžiniermi, aby bola práca efektívna. V záujme vysokej efektivity a dobrej spolupráce je nutné aj riešiť konflikty, ktoré vznikajú medzi vývojármi v tímoch.

Eseje

Vývojové tímy v softvérovom inžinierstve
Prístupy k zdieľaniu informácií v distribuovaných projektoch
Manažment konfliktov medzi programátormi a testermi

Kvalita softvéru

Počítače robia opakovane a rýchlejšie činnosti, ktoré by dokázal vykonávať aj človek, no rýchlosť a bezchybné vykonávanie inštrukcií dovoľujú týmto strojom robiť priam zázraky. Rozsiahlosť a komplexnosť procesov, ktoré vykonávajú programy, spôsobuje, že práca softvérových inžinierov je z pohľadu kvality neporovnateľná s inými inžinierskymi činnosťami. Nie je možné vytvoriť bezchybný softvér, a tak je nutné

vytvorené dielo testovať a opravovať chyby, ktoré sa vyskytnú. Ako pri každej ľudskej činnosti, aj pri vývoji softvéru za každú chybu niekto zodpovedá.

Eseje

Počítanie riadkov zdrojového kódu
Podvedomý prístup k testovaniu softvéru
Kto je zodpovedný za chyby a bezpečnostné slabiny v softvéri ?

Vzťahy medzi členmi dodávateľského reťazca

PETER TRNOVSKÝ

Slovenská technická univerzita
Fakulta informatiky a informačných technológií
Ilkovičova 3, 842 16 Bratislava

Abstrakt. V posledných štyridsiatich rokoch sa techniky, metódy a procesy dodávky softvéru zameriavali najmä na dodávateľa softvéru. No s posunom požiadaviek trhu na čas, funkcionalitu a flexibilitu, je potrebný posun aj v oblasti dodávky softvéru a to smerom k zákazníkovi. Evolúcia dodávateľských prístupov v softvérovom inžinierstve spôsobuje posun od dodávky softvéru ako tovaru, k dodávke softvérových služieb. Je zaujímavé sledovať vývoj dodávateľských vzťahov ako súčasť tejto evolúcie. Nemožno však zabúdať na široko rozšírený spôsob dodávky založený na využití generického softvéru, prípadne na vývoj softvéru na zákazku. Nové prístupy k dodávke softvéru, vytvárané s ohľadom na otvorený trh, prinášajú okrem zmeny organizácie dodávateľských vzťahov aj nové požiadavky na ich kvantitu. Súčasťou nových prístupov sú plne automatizované vzťahy pre koncového zákazníka ako v oblasti vývoja softvéru a komunikácie, tak aj v oblasti obchodu. Plne automatizované vzťahy znamenajú nielen prínos pre skvalitnenie vzťahov v procese dodávky softvéru, ale v konečnom dôsledku umožňujú zrýchliť dodávkový cyklus a evolúciu. Naopak klasické prístupy sú zaujímavé z pohľadu sily vzťahov a vzájomného poznania sa obchodných partnerov. Silný vzťah má stále svoje opodstatnenie vo vzájomnej dôvere, zdieľaní informácií, ktoré umožňujú znížiť riziko spojené s dodávkou softvéru.

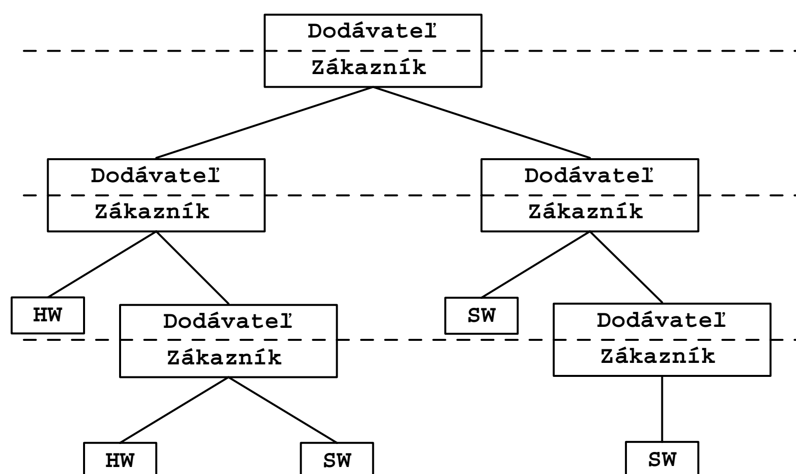
Vzťahy, dedičstvo minulosti

Dôležitosť a fungovania vzťahov medzi zákazníkom a jeho dodávateľom sú veci, ktoré boli známe obchodníkom už pred niekoľkými stáročiami. Rovnaké vzťahy zostávajú platné dodnes a to aj pre oblasti, ktoré svoj vek počítajú v desiatkach rokov. Vzťahy medzi zákazníkom a dodávateľom majú svoje opodstatnenie aj v procese dodávky produktov a služieb v oblasti softvéru. Manažment vzťahov v moderných dodávateľských procesoch má rovnaké základy, ktoré vychádzajú z toku tovarov a služieb od dodávateľa k zákazníkovi [2, 4].

Organizácia zapojená do dodávateľského procesu si nevyhnutne vytvára vzťahy s dodávateľmi a zákazníkmi. Prepojením organizácií vznikajú špecifické dodávateľské

reťazce. Do reťazca môže byť zapojené veľké množstvo organizácií. Každá z organizácií si vytvára vlastné vzťahy, ktorých počet a kvalita je vlastná danej organizácii. Vzájomné vzťahy hrajú dôležitú rolu v úspechu celého dodávateľského reťazca.

Klasický dodávateľský reťazec [5] je hierarchická organizačná štruktúra, ktorej vrcholom je koncový spotrebiteľ. Tento je prostredníctvom maloobchodného trhu napojený na sieť dodávateľov. Vzťahy medzi dodávateľmi sú organizované prostredníctvom veľkoobchodného trhu. Na úplnom spodku klasického dodávateľského reťazca sa nachádzajú primitívne výrobky a služby. Tokom primitívnych výrobkov ku koncovému zákazníkovi postupne narastá ich pridaná hodnota. Uvedený reťazec je možné nájsť aj v procese dodávky softvéru (obr. 1.).



Obr. 1. Prirodzený dodávateľský reťazec

Pri dodávke softvéru, ale aj v iných oblastiach, existujú vysoko dynamické, agilné organizácie s vysokou mierou inovácií, krátkym produkčným cyklom a prispôsobovaním založenom na evolúcií zákazníckych preferencií. U týchto je možné pozorovať klasickú dodávateľskú reťaz, ale táto je vzhľadom na rýchly vývoj, modifikovaná pri každom jej použití. Preto sa zameriame hlavne na podstatu vzťahov v sieti dodávateľov v závislosti od zvolenej vývojovej paradigmy.

V posledných štyridsiatich rokoch sa techniky, metódy a procesy dodávky softvéru zameriavali najmä na dodávateľa softvéru. No s posunom požiadaviek trhu kladených na funkcionalitu, flexibilitu a čas, je potrebný posun aj v oblasti vývoja a dodávky softvéru. Radikálny posun je potrebný najmä od centrického modelu založeného na požiadavkách, k modelu založenému na otvorenom trhu a softvérových službách.

Otvorený trh

V absolútne flexibilnom svete má zákazník možnosť dostať službu kedykoľvek potrebuje jej vykonanie. Medzi problémom a jeho riešením existuje neskoré viazanie, ktoré sa vykonáva až v čase uspokojovania potreby. Predpokladá sa široká škála pre výber riešenia problému, jednoduchá nahraditeľnosť riešenia iným a bezprostredná dostupnosť.

Ideálnu predstavu trhu v praxi nahrádza pragmatickejšia predstava, v ktorej s využitím síl pôsobiacich na trhu vznikajú spojenectvá dodávateľov. Títo poskytujú menšie množstvo, ale populárnych riešení. Existuje tu teda tlak na ich neustále zlepšovanie. Tento stav je pozorovateľný vo viacerých odvetviach najmä v oblasti služieb. Na podpore otvoreného trhu je v súčasnosti v softvérovom inžinierstve založených viacero vývojových postupov (napríklad aj v rámci ESA). Dôvodom pre podporu otvoreného trhu je najmä tlak na rýchlu evolúciu riešení, či vytváranie rozsiahlych globálnych trhov.

Softvér na zákazku

V súčasnosti bežným prístupom k dodávke softvéru je vytváranie softvéru na zákazku. Softvérom na zákazku budeme rozumieť vývoj softvéru pre koncovú organizáciu, kedy dodávateľská organizácia disponuje všetkou potrebnou technológiu, a teda jej fungovanie závisí len od malého množstva dodávateľov. Nosná časť vytváraného riešenia je dielom dodávateľa softvéru. Pre dodávku softvéru si zákaznícka organizácia vyberá jednu alebo malé množstvo dodávateľských organizácií.

Dodávka softvéru sa zväčša uskutočňuje dlhší čas, a preto je tento prístup charakteristický silným vzťahom medzi zákazníkom a dodávateľom. Vzťah medzi zákazníkom a dodávateľom sa spravidla vytvára v priebehu viacerých fáz. Fázy zahŕňajú stretnutia so zákazníkom za účelom špecifikácie požiadaviek, návrh systému, implementáciu, nasadenie, zaučenie používateľov, ladenie v čase prevádzky a následnú technickú podporu. Vzťah pri dodávke softvéru na zákazku preto pretrváva zväčša po dobu dodávky viacerých produktov alebo verzií softvéru.

Dobré poznanie sa oboch strán znižuje riziko neúspechu. Silný vzťah je výhodou aj pri analýze súčasných a budúcich potrieb zákazníka a znižuje riziko uviaznutia problému. Veľkou nevýhodou dodávky softvéru na zákazku je obmedzená konkurencieschopnosť v dobe po výbere dodávateľa. Pre zákazníka nie je jednoduché zmeniť dodávateľa alebo produkt v ďalších fázach dodávky.

Veľkým rizikom pre úspech projektu je nízka angažovanosť ľudí na strane zákazníka. Tento prístup je preto limitovaný na použitie najmä vo veľkých zákazníckych organizáciách, ktoré sú spôsobilé poskytnúť dostatočné množstvo a aktiváciu ľudskej sily.

Ďalšou nevýhodou je obmedzený prístup nových dodávateľov na trh. Prístup k trhu majú najmä silné dodávateľské organizácie. Hlavnou prednosťou týchto organizácií je ich meno, ktoré sa spája s vysokým štandardom kvality a garanciou úspešného ukončenia projektu.

Generický softvér

Iným prístupom k dodávke softvéru, je vytváranie riešenia s využitím generického softvéru alebo COTS komponentov (Commercial off-the-shelf). Prístup COTS je založený na procese zostavovania konečného produktu, výberom z malého množstva nastavení. Stavebnou časťou pri prístupe COTS sú rozsiahlejšie produkty alebo softvérové celky, ktoré sú prispôsobované na poskytovanie špecifickej funkcionality. Takéto softvérové celky často poskytujú a využívajú funkcionalitu v rámci značnej časti podnikových informačných systémov. Zákazníkmi pri dodávke týchto systémov sú priamo koncové organizácie.

Vývojom procesu pre vytváranie COTS systémov sa zaoberá množstvo vývojových skupín. Príkladom úspešných procesov sú procesy PORE [7] a PECA [3]. Súčasťou procesov sú aj postupy pre vytváranie vzťahov medzi dodávateľmi COTS produktov a organizáciami vyhľadávajúcimi COTS produkty.

Dodávka softvéru na zákazku a prístup COTS majú z hľadiska vzťahov dodávateľa a koncového zákazníka mnoho spoločného. Pri týchto prístupoch vstupuje koncový zákazník do dodávateľského vzťahu s jednou alebo malým množstvom dodávateľských organizácií. Medzi zákazníkom a dodávateľom vzniká silný vzťah. Z pohľadu oboch strán sa javí druhá strana ako dlhodobý partner. Medzi oboma stranami sa vytvára silnejší vzťah – partnerstvo. Aby bolo partnerstvo úspešné musia byť vzájomné vzťahy budované na silných základoch. Vzťahy sú budované zväčša dlhodobo a rozširujú sa s priebehom dodávky produktu alebo produktov. Vo vzťahoch je nevyhnutný vzájomný rešpekt a vôľa zdieľať informácie.

Úlohou zákazníka pri COTS prístupe je sledovanie technológie a udržovanie informácií pre vyhodnocovanie a expertízy. Pri prístupe COTS je pre zákazníka žiadúce, aby bol schopný identifikovať a vyhodnotiť alternatívy a iných partnerov. Udržovanie týchto informácií kompenzuje silnú viazanosť zákazníka na dodávaný produkt, prípadne na dodávateľskú organizáciu.

Dodávateľ sústavne konzultuje so zákazníkom jeho aktuálne a budúce požiadavky. Uprednostnením osobných kontaktov pred elektronickou komunikáciou si môže utvrdzovať vzťah so zákazníkom. Široká vzájomná komunikácia s partnerom, charakteristická pre tieto prístupy k dodávke softvéru, umožňuje predísť uviaznutiu problému či už na strane dodávateľa alebo zákazníka. Naopak veľké množstvo zdieľaných informácií nesie so sebou riziko straty potreby služieb dodávateľa. Preto pre správne fungovanie zdieľania informácií je dôležité, aby zákazník ubezpečoval dodávateľa, že mu nehrozí vytlačenie z trhu samotným zákazníkom.

Nevýhodou uvedených prístupov je vysoká viazanosť zákazníka na dodávateľa. Pre zákazníka môže byť veľmi ťažké zmeniť dodávateľa alebo nahradiť produkt. Pre malé a stredné organizácie môže byť ťažké dostať sa medzi dodávateľov produktu, prípadne stať zákazníkom pre tieto produkty.

Jednou z osobitných charakteristík dodávky softvéru na zákazku a COTS prístupu je možnosť manipulovať s druhou stranou. Táto možnosť je ako na strane zákazníka, tak na strane dodávateľa. Túto možnosť využívajú silné zákaznícke organizácie na vytváranie neprimeraného tlaku na svojich dodávateľov. Tieto organizácie sú však

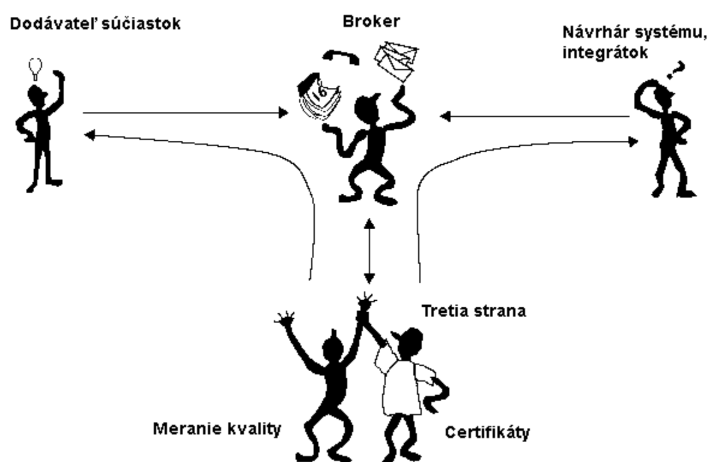
taktiež schopné poskytnúť vysoké nasadenie pracovníkov pre stretnutia s dodávateľom, či sledovanie demonštrácií produktu.

Softvér zo súčiastok

Softvérové inžinierstvo založené na komponentoch CBSE (z anglického Component Based Software Engineering) je založené na vytváraní systémov z už existujúcich častí. V tejto eseji ním budeme rozumieť proces vytvárania systémov z komponentov, ktoré pochádzajú od nezávislých tvorcov, nezávislých dodávateľov a rozmiestnenia, pričom spolupracujú na vytvorení požadovanej funkcionality.

Dodávateľská reťaz v prístupe CBSE (obr. 2.) pozostáva najčastejšie zo štyroch skupín hráčov:

- Návrhár systému alebo integrátor komponentov – vyberá a získava komponenty. V CBSE prístupe predstavuje konečného zákazníka.
- Dodávateľ CBSE komponentov – vyvíja komponenty pre použitie v prístupe CBSE.
- Broker – vystupuje ako sprostredkovateľ. Vytvára automatizovaný trh pre dodávateľov a odberateľov CBSE komponentov.
- Tretia strana – vytvára pridanú hodnotu CBSE komponentov. Táto môže byť vo forme certifikátov alebo vyhodnocovania komponentov.



Obr. 2. Organizácia vzťahov pri prístupe CBSE

Vzťah medzi koncovým zákazníkom (tento raz návrhár systému) je spravidla realizovaný prostredníctvom sprostredkovateľa. Tento poskytuje množstvo služieb a najmä podporu pre rozhodovanie.

Dodávateľ súčiastok vstupuje do vzťahu so sprostredkovateľom s cieľom zaregistrovať svoju súčiastku v katalógu súčiastok, ktorý vedie sprostredkovateľ. Katalóg obsahuje funkcionálne a nefunkcionálne vlastnosti súčiastky a ďalšie informácie, ako napríklad obchodné informácie o dodávateľovi alebo cene. Zákazník si vyberá súčiastky z katalógu a v spolupráci so sprostredkovateľom robí kompromisy medzi ponukou súčiastok a vlastnými požiadavkami. Pri výbere mu pomáhajú nástroje pre usporiadanie a výber súčiastok, certifikáty, vizualizácia súčiastok a vizualizácia podobnosti súčiastok s požiadavkami. Vzťah medzi koncovým zákazníkom a dodávateľom súčiastok je preto takmer automatizovaný.

Nevýhodou CBSE prístupu je riziko spojené s trhom. Zákazník nemá taký silný vzťah so svojim dodávateľom ako v prípade dodávky softvéru na zákazku alebo COTS. Zvyšuje sa teda riziko, že dodávateľ prestane dodávať nájdenú súčiastku alebo úplne odíde z trhu. S tým je spojené riziko, že zákazník nebude vedieť nájsť náhradu súčiastky.

Posun od prístupu COTS nastal najmä v miere zviazanosti produktu so zákazníkom. Nahradenie softvérovej súčiastky v prístupe CBSE je principiálne jednoduchšie ako výmena rozsiahleho COTS komponentu.

Inou nevýhodou CBSE prístupu je vysoká réžia spojená s registráciou súčiastky. Táto časť by mohla byť v budúcnosti realizovaná na vyššej úrovni automatizácie.

Softvér ako služby

SBSE – softvérové inžinierstvo založené na softvérových službách (z anglického Service Based Software Engineering) predstavuje evolučný krok z CBSE. Základom pre SBSE prístup je myšlienka zviazať a nastaviť služby na špecifickú množinu požiadaviek až v čase jej poskytnutia. Ako definíciu služby použijeme definíciu podľa [6] :

"Služba je pôsobenie alebo výkon ponúkaný jednou stranou druhej. Výkon môže byť zviazaný s fyzickým produktom. No v podstate je výkon nehmatateľný a spravidla sa neprenáša do vlastníctva žiadneho faktoru produkcie."

Dodávka služby môže pozostávať z poskytnutia základnej služby, alebo častejšie poskytnutia služby pozostávajúcej z čiastkových služieb. Za služby sa platí až priamo pri ich poskytnutí. Služba nie je mechanický proces, pretože zapája ľudský manažment dodávateľských vzťahov. Dodávka softvéru vo forme služieb umožňuje vytvárať, zhromažďovať a skladať služby spojením veľkého množstva dodávateľov, a to v dobe vzniku požiadavky na vykonanie služby [1].

Podobne ako v CBSE aj pri SBSE vystupujú tri typy hráčov:

- Zákazník – koncová organizácia, ktorá bude službu využívať.
- Dodávateľ SBSE služieb – na základe požiadaviek trhu vytvára a poskytuje služby koncovému zákazníkovi.
- Broker – vystupuje ako sprostredkovateľ. Vytvára automatizovaný trh pre dodávateľov a odberateľov SBSE služieb.

Rovnako ako pri prístupe CBSE, dodávatelia registrujú ponuku služieb v registri vedenom sprostredkovateľom. Zákazníci pristupujú k registru v dvoch krokoch. V prvom kroku zákazník prehľadáva služby s cieľom nájsť také, ktoré korešpondujú s jeho potrebami. V druhom kroku zákazník vybrané služby využíva. Vzťah medzi dodávateľom a zákazníkom je vytváraný dynamicky v čase poskytnutia služby. V súčasnosti je dostupné množstvo alternatívnych technológií, ktoré umožňujú podporu uvedeného modelu (.NET, J2EE a množstvo konkurenčných aplikačných serverov).

Počas návrhu služby, používateľ využíva prostredie pre návrh skladania služieb. Výsledkom je popis zloženej služby alebo šablóna. Návrh zloženej služby je časť procesu vyžadujúca si zapojenie ľudskej aktivity. Popis zloženej služby obsahuje architektúru čiastkových služieb, popis čiastkových služieb a podmienky na ich kvalitu.

V čase potreby vykonania služby je využitý systém dynamického poskytovania služieb. Tento na základe popisu alebo šablóny vykoná výber potrebných čiastkových služieb, automaticky dohodne podmienky, a to vrátane obchodných a zmluvných, s poskytovateľmi služieb a zavolá dohodnuté čiastkové služby. Spôsob viazania služieb sa približuje k ultra neskorému viazaniu, kedy sa zviazanie realizuje až pred poskytnutím služby a je zrušené bezprostredne po jej zrealizovaní. Zrejmou výhodou je jednoduchá nahraditeľnosť čiastkovej služby. Zákazník je minimálne viazaný na konkrétneho dodávateľa.

Ako bolo spomenuté, vzťah medzi zákazníkom a dodávateľom služby je realizovaný prostredníctvom sprostredkovateľa. Tento plní viacero úloh a môže pozostávať z viacerých organizácií. Tieto zabezpečujú registráciu služieb, platby za služby, dohadovanie zmluvných podmienok, monitorovanie kvality služieb a iné činnosti súvisiace s poskytnutím služieb.

Jednou z nevýhod uvedeného prístupu je riziko spojené s veľkou mierou automatizácie vzťahov. Dôvera medzi partnermi je ohraničená. Dôveru ohraničuje najmä slabé poznanie sa partnerov. S nízkou mierou dôvery je spojené riziko zlyhania. Inou nevýhodou, podobne ako pri CBSE prístupe, je vysoká réžia súvisiaca automatizáciou aktivít ako výber dodávateľa, dohadovanie zmluvných podmienok, zmena požiadaviek alebo monitorovanie kvality služieb.

Záver

Jednoznačnou prednosťou poskytovania softvérových služieb pred ďalšími uvedenými prístupmi je priblíženie sa k otvorenému trhu. No každý z uvedených prístupov prináša do vzťahov svoje výhody a špecifiká. Na tieto nemožno zabúdať najmä pri evolúcií a vytváraní nových prístupov k tvorbe softvéru alebo modelov dodávateľských vzťahov.

Pri pohľade do budúcnosti je potrebné si uvedomiť výhody prístupu k dodávke softvéru vo forme služieb najmä v kontexte vytvárania globálnych trhov, dynamických vzťahov v sieti dodávateľov, nezávislého rozmiestnenie malých čiastkových služieb a umožnenie rýchlej evolúcie čiastkových služieb.

Použitá literatúra

1. K. Bennett, P. Layzell, D. Budgen, P. Brereton, L. Macaulay, M. Munro: Service-Based Software: The Future for Flexible Software. APSEC, 2000.
2. P. Brereton: The Software Customer/Supplier Relationship. Communication of the ACM, Vol. 47, No. 2, February 2004, 77-81.
3. S. Comela-Dorda: A process fot COTS software product evaluation. In Proceedings of the First International Conference on COTS-Based Software Systems, february 2002, Springer-Verlag.
4. R.B. Handfiels, E.L. Nichols: Introduction to Supply Chain Management. Prantice Hall, Upper Saddle River, NJ, 1998.
5. M. Jones, A. Mantineo, U.K. Mortensen: Introducting ECSS Software-Enginnering Standards within ESA. Esa bulletin, 111., august 2002, 132-139.
6. C. Lovelock, S. Vandermerwe, B. Lewis: Services Marketing, Prentice Hall Europe, 1996.
7. N.A.M. Maiden, C. Ncubde: Acquiring COTS software selection requirements. IEEE Software 15, 2, 1998, 45-56.

Annotation

Relationships between members of supply chain.

Management of a supply chain is important step to implement more flexible and agile software development process. This paper present some approaches of software procurement. There are summarized advantages and future use of bespoke, component based and software services based software development processes. Major part of this work is dedicated to highlighting benefits of software services that lead to binding of service components just in time they are needed and then the binding may be discarded. Also this approach allows to make other relationships on demand.

Softvér a údaje, majetok pre zdieľanie

MICHAL MORAVČÍK

Slovenská technická univerzita
Fakulta informatiky a informačných technológií
Ilkovičova 3, 842 16 Bratislava

Abstrakt. Softvér je moderným typom majetku v dnešnej, na vlastnícke vzťahy citlivej, spoločnosti. Rozvoj ľudskej spoločnosti bol odpradávna ovplyvnený zdieľaním informácií rôzneho druhu. Nerovnomerným rozširovaním informácií sa ľudia, ktorí sú vlastníkmi informácie, dostávajú do výhody oproti ostatným, ktorí informáciou nedisponujú. Aj takto teda vznikajú rozdiely medzi ľuďmi. Niet ľahšieho spôsobu rozširovania informácií, ako je to v prípade softvéru. Priam magické slovo „kopírovať" je možné nájsť v každom operačnom systéme, v životnom prostredí pre softvér. Je samozrejmé, že ten kto vlastní softvér sa dostáva do výhody oproti ostatným, ktorí ho nemajú. Avšak do väčšej nevýhody sa dostáva autor, ktorý sa živí tvorbou softvéru, keď je jeho dielo voľne, resp. nelegálne rozširované. V tejto eseji sa budem venovať myšlienkam, aký je to vlastne nelegálny softvér, kedy, ako a prečo sa zdieľanie softvéru stáva nelegálnym a aký to má vplyv na softvérové inžinierstvo a na softvérového inžiniera.

Úvod

Softvér je moderným typom majetku v dnešnej, na vlastnícke vzťahy citlivej, spoločnosti. Softvér je vo svojej podstate iba text napísaný človekom v nejakom programovacom jazyku, preložený do strojového jazyka, ktorému rozumie počítač. Algoritmus programu možno zapísať nielen v rozličných programovacích jazykoch, ale aj viacerými spôsobmi v tom istom programovacom jazyku, podobne ako pri vyjadrovaní myšlienok. Softvér teda môžeme prirovnať k rukopisu autora, programátora [2]. Programy teda môžu v podstate fungovať a vyzerať rozdielne, ale môžu zabezpečovať rovnakú funkciu. Tu sa objavuje otázka. Čo a ako treba chrániť, samotnú formu, verziu, text programu, alebo jeho myšlienku, funkciu?

Zdieľanie údajov

Údaje slúžia na prenos informácie. Rozvoj spoločnosti by nebol možný bez zdieľania a rozširovania informácií. Posledné výskumy hovoria [3] že psi oveľa ľahšie vedia rozpoznať význam informácií ako šimpanzy, preto je možné že sú lepšími „priateľmi“ človeka.

Spoločnosť a technológie na prenos údajov sa vyvíjali zároveň. Tí, ktorí disponujú informáciami sa dostávajú do výhody oproti iným, ktorí nemajú prístup k informáciám. Napríklad keď jeden podnikateľ má štatistické informácie o ponuke a o dopyte po nejakom tovare, môže výhodne prispôsobiť svoje podnikanie na svoje obohatenie. Rovnaké princípy môžu platiť aj v prípade softvéru. Firma, ktorá má k dispozícii kvalitný a efektívny nástroj na tvorbu softvéru môže vyprodukovať viac softvéru, s menším množstvom chýb, ako spoločnosť, ktorej vývojári používajú zastarané vývojové prostredia. Forma údajov, v akej sa prenášali, sa menila v závislosti od rozvíjajúcich sa technológií [3].

Vývoj údajov

Prvou technológiou na prenos údajov bol *hovorený jazyk*. Údaje sa prenášali výlučne ústnou formou a keďže boli uložené iba v ľudskej pamäti, mali krátkodobý charakter. Rečou sa šíri predovšetkým myšlienka, poslucháči si zapamätajú informáciu samotnú a nie presnú formu, v akej bola podávaná.

Ďalšou technológiou na prenos údajov bol *písaný jazyk*. Údaje v písanej forme majú niektoré veľmi dôležité vlastnosti. Sú trvalé a je možné reprodukovať ich informačnú hodnotu bez zmeny. Má zmysel identifikovať pôvodného autora informácie.

Dnes sa údaje prenášajú v *elektromagnetickej podobe*. Táto forma zjednodušuje získavanie, ukladanie, uchovávanie, spracovanie, prenos a zobrazovanie informácií. Technológia umožňuje uchovať akúkoľvek informáciu, v priam akomkoľvek množstve a v akejkoľvek forme, ako sú reč, obraz, video, program, text a podobne.

Modernejšie technológie umožňujú jednoduchší prenos a rozširovanie informácií. Avšak na ochranu údajov sú potrebné ešte modernejšie a sofistikovanejšie technológie. Inými slovami, technológia na prenos a uchovanie informácií je vždy krok dopredu pred technológiou na ich ochranu. Preto je veľmi komplikované ustrážiť akékoľvek kopírovanie údajov a nárokovanie si autorských práv.

Údaje ako majetok

Údaje určitým spôsobom reprezentujú myšlienky, fakty, programy. Z pohľadu ochrany majetku sa údaje ochraňujú ako konkrétna reprezentácia myšlienky [3], tzv. *copyright*. Prvý krát bola ochrana údajov aplikovaná na tlač kníh v Anglicku, ktorá však viedla k vytvoreniu monopolu a k vydieraniu. Neskôr boli práva na kontrolovanie kopírovania rozšírené aj na umenie (maľby, divadlo, hudba).

Ľudová hudba bola voľne kopírovaná, dopĺňaná a distribuovaná [5] a malo to pozitívny vplyv v prvom rade na jej zachovanie a v neposlednom rade aj na jej rozvoj.

Anonymní autori vložili do ľudovej muziky svoje hudobné bohatstvo z ktorého mohla čerpať celá spoločnosť.

V prípade softvéru existujú tzv. *open source* projekty, do vývoja ktorých zasahuje množstvo vývojárov a teda všetci spoločne prispievajú k ich vylepšovaniu. Z toho môže, podobne ako z ľudovej muziky, čerpať celá spoločnosť.

Zdieľanie softvéru

Autori softvéru, softvéroví inžinieri, programátori, v drvivej väčšine nevytvárajú softvér výlučne pre svoju potrebu. Softvér slúži ľuďom a teda musí sa k nim nejako dostať. Rozširovanie programov je veľmi jednoduché. Magické slovo „kopírovať“ možno nájsť v každom operačnom systéme, životnom prostredí pre softvér.

Čo je to softvér

Pod pojmom softvér [2] si veľa ľudí predstaví programy ako MS Word, alebo Netscape Navigator. Analogicky k tejto predstave možno povedať, že Shakespearovo dielo Rómeo a Júlia je „anglický jazyk“. Na najnižšej úrovni tvorí softvér formálne iba postupnosť čísel (pozri Obr. 1), podobne ako jazyk je postupnosť písmen a medzier.

```
00000330: cc7c fbf7 89ec 5dc3 4865 6c6c 6f20 776f  .|....].Hello wo
00000340: 726c 6421 0a00 5589 e583 ec08 83e4 f0b8  rld!..U.........
00000350: 0000 0000 29c4 c704 2438 8304 08e8 06ff  ....)...$8......
00000360: ffff b800 0000 00c9 c390 9090 9090 9090  ................
00000370: 5589 e556 5331 dbe8 b4fe ffff b838 9404  U..VS1.......8..
```

Obr. 1. Kód programu Hello world v binárnom tvare

Čísla z tejto postupnosti sa dajú interpretovať ako inštrukcie, elementárne príkazy a údaje s ktorými program pracuje. Program je teda tvorený zoznamom inštrukcií (pozri Obr. 2), podobne ako recept v kuchárskej knihe [2]. Počítač ich číta, interpretuje a vykonáva.

```
.LC0:
    .string "Hello world!\n"
.globl main
    .type   main,@function
main:
    pushl   %ebp
    movl    %esp, %ebp
    subl    $8, %esp
    andl    $-16, %esp
    movl    $0, %eax
    subl    %eax, %esp
    movl    $.LC0, (%esp)
    call    printf
    movl    $0, %eax
    leave
    ret
```

Obr. 2. Kód programu Hello world v jazyku symbolických inštrukcií

Ľudia si tvorbu programov zjednodušili tak, že vývojári nemusia písať programy v číslach, a vymýšľať číslam význam, ani v inštrukciách pre každý typ procesora zvlášť, ale v abstraktnom, obmedzenom programovacom jazyku, ktorý si sami vymysleli a ktorý sa čiastočne podobá ľudskej reči (pozri Obr. 3).

```
#include <stdic.h>
int main(int argc, char** argv, char** envv) {
  prin=f("Hellc world!\n");
  retu=n 0;
}
```

Obr. 3. Kód programu Hello world v jazyku C [2]

Význam softvéru je v komunikácii

Význam, podstata softvéru je v odovzdávaní informácií, teda v komunikácii [2]. Človek komunikuje s počítačom keď vytvára softvér. Ľudia komunikujú medzi sebou, zdieľajú zdrojové kódy a diskutujú o nich. Počítač komunikuje s počítačom pri prenose a vykonávaní programu. A nakoniec, počítač komunikuje s človekom o najlepšom nájdenom riešení problému.

Komunikácia teda sprevádza celý proces vývoja softvéru, ako aj jeho používanie. Výmenou informácií, podľa nášho predpokladu, dochádza k rozvoju spoločnosti, v prvom rade spoločnosti vývojárov softvéru, ktorí si pomáhajú zdieľaním vlastných skúseností a nápadov, a v neposlednom rade aj spoločnosti používateľov, pre ktorých je softvér určený. Používaním softvéru majú k dispozícií informácie ktoré by bez neho nemali.

Nelegálny softvér

Z právneho pohľadu je softvér nehmotným majetkom, autorským dielom [6]. Na disponovanie so softvérom je potrebný súhlas nositeľa autorských práv na základe licenčnej zmluvy, ktorá je súčasťou dodávky softvéru. Nedodržiavaním licenčných podmienok a voľným kopírovaním sa softvér z právneho hľadiska stáva nelegálnym, tiež nazývaný pirátsky softvér.

Podľa štatistík bolo na Slovensku v roku 1998 až 50% softvéru nelegálne používaného [6], čo bol v porovnaní s rokom 1996 pokles o 16%. Softvérové spoločnosti prišli o zisky v hodnote 11,2 miliónov USD (400 miliónov Sk). V celosvetovom pohľade bolo používaných 38% nelegálneho softvéru, čo predstavuje stratu 11,4 miliardy USD.

Odhalenie tejto trestnej činnosti je veľmi zložité. Postupne však dochádza k zlepšovaniu situácie jednak uvedomovaním si etických princípov používateľmi softvéru a aj úpravou príslušnej legislatívy. Vznikajú organizácie dohliadajúce na používanie legálneho softvéru predovšetkým vo firmách, ale aj v domácnostiach. Známy je prípad Jiřího Herolda, keď neznáma osoba podala oznámenie o nelegálnom používaní softvéru na firmu Českomoravská lekárnická platebna, a.s., alebo prípad z 14.januára 1998 keď súd vyniesol vôbec prvý rozsudok nad obchodníkom na Slovensku, ktorý vyrábal a distribuoval pirátsky softvér spoločností Autodesk, Microsoft a Novell a dosiahol zisk vo výške 400 000 Sk. Obchodník bol odsúdený na 8 mesačný podmienečný trest odňatia slobody a boli mu odobrané CD nosiče a dva osobné počítače.

Zdieľanie softvéru v každodennom živote

Ako to vyzerá v každodennom živote? Bežný používateľ sa dozvie o novom, výbornom softvéri a hneď zatúži si ho vyskúšať. Najprv sa pokúsi získať tento softvér legálnou cestou, teda presvedčí sa, či nie je náhodou voľne prístupný, v podobe *freeware*, *shareware*, alebo *open source* na web stránke autora softvéru. Ak nenájde, pokúsi sa získať aspoň nejakú skúšobnú, obmedzenú verziu programu. Ak sa mu program zapáči, svedomie má v poriadku a je ochotný doň investovať, program si kúpi. Ak po plne funkčnom programe zatúži ešte viac a nemá potrebnú sumu, alebo jednoducho nie je ochotný zaplatiť za program snaží sa získať plnú verziu programu spôsobom, ktorý môžeme označiť za nelegálny. Priamo na internete existuje množstvo stránok, ktoré poskytujú informácie o *sériových číslach* (najbežnejší spôsob ochrany softvéru) množstva bežne používaných programov, ktorými je možné odomknúť skúšobnú verziu programu. Ak je program chránený sofistikovanejšie, tento náš používateľ potrebuje použiť sofistikovanejšie prelomenie tejto ochrany, vo forme malého programu (tzv. *crack* alebo *patch*), ktorý zväčša pozmení kód, t.j. prelomí algoritmus ochrany, samotného programu, čím ho zbaví obmedzení skúšobnej verzie. Tieto malé programy je možné podobne získať z internetu. Potom už môže program ľubovoľne používať a aj kopírovať.

Ak program nie je prístupný na internete v skúšobnej verzii, náš mimozákonný používateľ sa pokúsi získať tento softvér od známeho, alebo z počítačovej siete, kde

viacerí používatelia zdieľajú údaje rôzneho druhu. Ak hľadá bežne používaný program, takmer určite ho u niekoho nájde a skopíruje si ho.

V neposlednom rade informuje svojich známych, že sa mu podarilo získať tento výborný softvér a dá im ho k dispozícii, čím umožní ďalšie nelegálne šírenie programu.

Ako sa softvér rozširuje nelegálne

Ako som už naznačil, šírenie softvéru je z technického hľadiska až príliš jednoduché. Ak aj softvér obsahuje nejakú ochranu, teda pre jeho používanie nestačí iba čisté skopírovanie programu, takmer vždy je možné dopracovať sa k údajom na prelomenie tejto ochrany. Potom už nelegálnej kópii programu naozaj nič nestojí v ceste k novým, nelegálnym používateľom.

Buď jedna osoba priamo poskytne svoju nelegálnu, alebo aj legálnu kópiu softvéru inej osobe, alebo používateľ získa softvér anonymne. V súčasnosti existujú v prostredí internetu veľké P2P (point to point, teda spájajú klientske počítače priamo) siete, ako napr. KaZaA, alebo Napster a podobne. Oficiálne tieto siete nepodporujú zdieľanie nelegálneho softvéru a údajov, ale na druhej strane vyhlasujú, že nenesú žiadnu právnu zodpovednosť za údaje, ktoré si používatelia medzi sebou prenášajú. Niektoré z týchto sietí boli zrušené, práve kvôli šíreniu nelegálneho softvéru a iných údajov ako nelegálnej hudby, filmov a podobne. Proti iným príslušné orgány aktívne bojujú, zatiaľ čo spokojní nelegálni používatelia využívajú „lacný“ prístup k informáciám.

Softvéroví piráti

Nelegálny softvér kopírujú predovšetkým ľudia, ktorí nemajú prostriedky na zakúpenie plnej verzie, ľudia, ktorí vôbec vedia nájsť a využiť tento nelegálny softvér. Väčšiu vinu zodpovednosti majú však ľudia, ktorí obchádzajú zabezpečenia programov a dávajú nelegálny softvér k dispozícii iným ľuďom. Najväčšmi však stoja mimo zákona ľudia, ktorí kopírujú, distribuujú a predávajú nelegálny softvér pre vlastný zisk. Vtedy už skutočne možno hovoriť o okrádaní autora softvéru, pretože niekto iný neprávom inkasuje za rovnaký produkt nezodpovedajúcu sumu. Títo ľudia sú označovaní ako softvéroví piráti a softvérová polícia najviac pátra práve po takýchlo ľuďoch.

Pohľad softvérového inžiniera

Gazda, ktorý pred dažďom nepozhŕňa a neskryje seno, ktoré sa mu suší pre kravu, len preto, že je lenivý, môže na to doplatiť [6]. Môže pršať niekoľko dní a seno mu zhnije. Keď má sena málo, v zime sa mu to vypomstí. Zdochne mu krava, ktorá nemá čo žrať. Uspokojil svoje momentálne „potreby“, ale zabudol na budúcnosť. Podobne je to aj so softvérovými inžiniermi. Ak používajú nelegálny softvér sú sami proti sebe. Náklady výrobcu sa odrážajú v cene softvéru. Za softvér, ktorý používa, výrobcovi nezaplatil. Ak výrobca predá málo kusov svojho produktu, cena jednej kópie programu bude vyššia ako keby ich predal viac. Pri vysokej cene to odradí množstvo potenciálnych

zákazníkov. Výrobca sa snaží znížiť svoje náklady na minimum aby cena softvéru bola čo najnižšia a predalo sa ho čo najviac. Náklady obmedzí tým, že bude vyrábať softvér s chudobnejšou funkčnosťou, alebo nižšou kvalitou. Na tvorbu softvéru s menšou funkčnosťou treba menej a aj menej kvalifikovaných softvérových inžinierov. Preto klesá dopyt zo strany výrobcov po softvérových inžinieroch, čo má za následok nižšiu trhovú hodnotu softvérového inžiniera. Potom sa teda nečudujme, ak nám šéf nebude chcieť zvýšiť plat.

Ochrana softvéru

Softvérový patent

Patent [1] je legálny spôsob ochrany nejakého vynálezu, postupu, myšlienky. Samotný patent je verejne prístupný, teda každý si môže zistiť ako nový vynález funguje a ako ho zostrojiť. Ale jedine držiteľ patentu a ten, komu držiteľ patentu predal práva, môže tento vynález vyrábať a predávať po dobu 20 rokov. Patent teda zaručuje ochranu intelektuálneho bohatstva a umožňuje autorovi živiť sa predávaním svojho vynálezu.

Pôvodne boli patenty zavedené pre priemyselné vynálezy. Preto je na mieste otázka, či má vôbec zmysel používať patenty pre softvér. Softvérový patent zahŕňa tzv. biznis model programu [1], teda to čo program robí a ako to robí. Patent definuje ako program číta vstupy a ako zapisuje výstupy a ako spolupracuje s inými programami. V 70. rokoch bolo rozhodnuté [1], že softvér bude chránený zákonom na ochranu intelektuálneho bohatstva. Proces rozširovania patentov aj na softvér trval v Spojených štátoch v priebehu 80. rokov až po koniec 90. rokov minulého storočia. Následne sa pripájajú aj ostatné krajiny. Softvérové patenty majú z ekonomického hľadiska takéto vplyvy [1]:

- *Malí producenti*: nízke šance na zisky, potenciálne vysoké náklady kvôli využívaniu patentov
- *Veľkí producenti*: nízke šance na zisky, nízke náklady, všeobecne kvôli krížovým licenciám
- *Držitelia patentov*: veľké šance na zisk, žiadne náklady
- *Používateli*a, vrátane vývojárov softvéru: platenie vyššej sumy za softvér obsahujúci patenty

Spomenuli sme, že softvér je akousi formou komunikácie, rukopis autora, umelecké dielo. Dá sa vôbec umenie patentovať? Predstavme si napríklad, že by bol patentovaný obraz *Mona Lisa*, znamenalo by to že počas doby držania patentu by nikto nemohol vytvoriť rovnaké dielo [2]. Tento patent by však nezabránil reprodukovaniu tohto diela, vyrobeniu plagátu, alebo pohľadnice a ich predávanie, používaniu diela v televíznej relácii, rádiu, alebo vystavovaniu podoby obrazu na internete. Podobne,

patent na automobil *Dodge Viper* nezabráni vytvoreniu fotografie, na ktorej bude dokonca množstvo patentovaných prvkov. Patenty, podľa [2] nie sú vhodným nástrojom ochrany pre softvér. Patentovanie softvéru je ako patentovanie kuchárskeho receptu, nie predmetu receptu, ale samotného textu receptu, čo môže vyznievať možno až trochu absurdne. Nanešťastie, patenty sú priznávané na funkcii kódu, napr. slávny patent Amazon.com „nakupovanie na jedno kliknutie". Tento patent nemá nič spoločné s kódom softvéru spomenutej spoločnosti, zahŕňa celý koncept nakupovania na jedno kliknutie. Zatiaľ čo klasické patenty zahŕňajú presný popis komponentov, tvarov a vzťahov jednotlivých objektov, softvérové patenty sú definované veľmi široko, pokrývajúc celé koncepty. Na druhej strane, softvérové patenty nepokrývajú tzv. *synonymné programy*, t.j. programy, ktoré vykonávajú principiálne rovnakú činnosť, napríklad webové servery ako Microsoft IIS a Apache.

Hlasy proti softvérovým patentom disponujú silnými argumentmi [4]. Takáto ochrana intelektuálneho bohatstva je značne nepraktická. Digitálna reprezentácia programov, alebo iných textov môže byť kopírovaná a manipulovaná s triviálnou námahou. Snaha chrániť prenos údajov v digitálnej podobe pôsobí výsmešne. Vážnejším argumentom je fakt, že zákon o intelektuálnom bohatstve je založený na ochrane všeobecného dobra a softvérové patenty, alebo copyright jednoducho nezaručuje dobro pre celú spoločnosť. Navyše voľné šírenie softvéru zaručuje stály rozvoj, znovupoužiteľnosť nápadov, overených riešení, dostupnosť pre všetkých a prínos pre celú spoločnosť.

Copyright a trademark na softvér

Copyright, alebo právo kontroly nad kopírovaním produktu. Toto právo chráni reprezentáciu softvéru, nie samotnú myšlienku softvéru. Trademark zabezpečuje jednoznačnú identifikáciu výrobcu produktu. Obe techniky majú zabrániť nespravodlivému obchodovaniu [4]. Aplikovaním na softvér však vôbec nie je zaručené, že autor programu je správne identifikovaný, a už vôbec nie je zaručené že softvér sa nebude kopírovať v zmysle rozširovania konkrétnej verzie vykonateľného kódu. V prípade kopírovania zdrojových textov softvéru je situácia iná. Prax je taká, že väčšinou potrebujeme časť zdrojového kódu, aby sme mohli využívať znovupoužiteľnosť softvérových súčiastok. V tomto prípade môže byť copyright účinnou „brzdou" pri vývoji softvéru.

Záver

Softvér je umelecké dielo, malý svet, zázrak. Autor mu dal život a podpísal sa na ňom vlastným rukopisom. Softvér je ako recept, postup pre servírovanie tých najlepších informácií pre používateľa, zapísaný jedným z tisíc možných spôsobov.

Dva rozdielne pohľady na softvér. Na jednej strane myšlienka, nápad, funkcia, zmysel programu a na druhej strane štýl, forma, spôsob zapísania. Oba pohľady sú nemenej dôležité a do istej miery treba softvér chrániť z oboch strán. V každom prípade by však mala mať osoh zo softvéru celá spoločnosť, nie monopolná firma,

ktorá predáva svoje produkty vysoko nad cenu a spôsobuje tak nepriamo rozvoj nelegálneho kopírovania a pokles etiky používateľov softvéru. Osoh by mali mať vývojári, mali by mať možnosť spoznať a podeliť sa s novými technikami, originálnymi nápadmi a riešeniami aby mohli tieto poznatky použiť v ďalších softvérových projektoch. Používatelia by mali mať v rukách kvalitný softvér pracujúci podľa požiadaviek, s technickou podporou v prípade problémov a v neposlednom rade mali by byť odmenení autori softvéru. Rovnováha sa hľadá veľmi ťažko, ale ak by sa dosiahla, boli by sme o krôčik bližšie k dokonalej a šťastnej spoločnosti.

Použitá literatúra

1. Barahona, J.M.G.: Free software and patentability of software, *ETUC affiliated organisations ICT meeting*, Madrid, June 24th, 2004
2. Bowers J.: Software and Software Patents. *The Ethics of Modern Communication*, 2005, www.jerf.org/writings/communicationEthics
3. Holmes, W.N.: Data and Information as Property. *Computer*, May 2004, 92, 90-91.
4. Holmes, W.N.: The Evitability of Software Patents. *Computer*, Mar. 2000, 30-34.
5. Santini, S.: Bringing Copyright into the Information Age. *Computer*, Aug. 2003, 104, 102-103.
6. Tím SOWA: *Etika softvérového inžiniera a softvérové procesy*. MSI, KIVT STU, Bratislava, 2000.

Annotation

Software and data, the property for sharing

This paper refers to issues on sharing software and data as property. It tries to describe what software is, what means sharing for the society and the contrast of benefits and negative aspects of free sharing of data and programs. It deals with illegal software gaining and the reasons why people do that. Paper describes ways how the software is protected, which is neither suitable for the software completely. On the other hand it proclaims sharing programs and source codes as a good proposion and tries to draft a fair balance of collaboration of developers and users of software.

Komerčné verzus nekomerčné metódy vývoja

MARTIN JENČO

Slovenská technická univerzita
Fakulta informatiky a informačných technológií
Ilkovičova 3, 842 16 Bratislava

Abstrakt. Vývoj softvéru je dlhý a náročný proces. Pre komerčný vývoj softvéru boli vyvinuté rôzne metódy, ktoré majú za úlohu zvyšovanie efektivity vývojárov a manažérov a znižovanie nákladov potrebných na vývoj softvérového produktu. Postupom času sa vykryštalizovali rôzne metódy vývoja aj v nekomerčnej sfére. Prekvapením môže byť, že niektoré metódy používané Open-Source komunitami (nekomerčná sféra) sú časovo efektívnejšie, ľahšie manažovateľné a menej náchylné na chyby ako mnohé komerčne využívané metódy. Ich použitie však nie je vôbec limitované na nekomerčnú sféru. V tejto práci sú porovnané najpoužívanejšia metódu nekomerčnej sféry s metódami komerčnej sféry z hľadiska manažovateľnosti, efektivity a celkovej použiteľnosti.

Úvod

Vývoj rozsiahlych softvérových produktov nie je vôbec jednoduchá záležitosť. Hlavne vývoj väčších a zložitejších systémov prináša so sebou množstvo problémov, ktoré sa pri vývoji menších produktov nevyskytujú. Medzi takéto problémy patria hlavne problémy s manažovateľnosťou projektov, správa verzií (testovacia verzia, verzia pre zákazníka, ...) a mnohé iné.

Pri komerčnom vývoji sa využívajú metódy, ktoré sa snažia predísť, resp. minimalizovať spomínané problémy. Okrem komerčnej sféry je však softvér vyvíjaný aj nekomerčne, zdrojové kódy sú verejne dostupné a ktokoľvek môže prispieť pri vývoji. Takýto spôsob vývoja softvéru sa nazýva vývoj s otvoreným zdrojovým kódom programu (*Open-Source*). Už na prvý pohľad je jasné, že pokiaľ by ktokoľvek mohol bez akejkoľvek kontroly pridať ľubovoľný kód do zdrojových kódov, bol by projekt veľmi rýchlo odsúdený na zánik. Preto sa časom vyvinuli a zaužívali metódy vývoja Open-Source projektov, pri ktorých je každý pridávaný kód kontrolovaný, čím sa taktiež odhalí väčšina chýb ešte pred ich pridaním do zdrojových kódov.

V tomto dokumente opíšem model vývoja softvéru používanom pri Open-Source projektoch, porovnať ho s metódami používanými v komerčnej sfére a zhodnotiť jeho kladné a záporné stránky.

Organizácia projektu

V komerčnej sfére sa najčastejšie používa hierarchický model organizácie projektu. Projekt je rozdelený na menšie podprojekty, kde za každý podprojekt je zodpovedná iná osoba. Väčšie podprojekty sa podobným spôsobom rozdelia na ešte menšie podprojekty, kde sa zodpovednosť taktiež rozdelí na skupinu ľudí. Keď už sú podprojekty rozumného rozsahu, začnú tímy vývojárov pracovať na projekte. Postupne ako pribúda funkcionalita jednotlivých podprojektov, stáva sa funkčným aj celkový projekt. Vývojári sú zodpovední za funkčnosť vyvinutých súčastí, zatiaľ čo vedúci tímov a manažéri sú zodpovední za správnosť a funkčnosť celého podprojektu, ktorý majú na starosti.

Pri použití takéhoto modelu je veľmi dobrá manažovateľnosť, nakoľko za každý celok, za každú súčasť je zodpovedný niekto konkrétny, kto priebežne informuje o tom, ako postupuje vývoj. Trochu problematickejšia je však komunikácia medzi jednotlivými tímami. Nie je zriedkavé, že tím vývojárov zodpovedný za vývoj jednej časti zistí chybu v inej časti projektu, za ktorú nie sú zodpovedný, ale spolupracujú s ňou. Alebo v prípade, že potrebujú, aby bola v inej časti projektu pridaná nová funkcionalita potrebná pre výsledný produkt resp. jeho časti. V takýchto prípadoch oznámia vývojári svoju požiadavku vedúcemu tímu, ktorý ju oznámi vedúcemu vyvíjaného celku, ten ju následne oznámi osobe zodpovednej za celý podprojekt a tak ďalej, až sa správa dostane k osobe, ktorá je zodpovedná za obidva celky. Tá ju následne odovzdá svojmu podriadenému, ktorý je zodpovedný za spomínaný celok, ten svojim podriadeným až sa správa dostane k druhému tímu vývojárov. Takýto spôsob komunikácie je už na prvý pohľad neefektívny a zdĺhavý. V praxi sa často tento spôsob komunikácie obchádza a tímy komunikujú priamo. Tým sa však znižuje manažovateľnosť projektu, nakoľko nadriadení, zodpovední za celok sa dozvedia o zmenách až dodatočne, prípadne vôbec. Tým sa taktiež ohrozuje funkčnosť celého projektu, nakoľko vývojársky tím nemusí vedieť podrobnosti o ostatných častiach a samoiniciatívnymi úpravami môžu napáchať viac škody ako úžitku.

Pri Open-Source projektoch neexistuje žiadne hierarchické usporiadanie, ale taktiež nie všetci majú rovnaké možnosti ovplyvniť, akým smerom sa bude projekt uberať. Ktokoľvek sa môže pridať k tímu vývojárov a môže navrhovať zmeny zdrojových kódov. Zmeny však môže iba navrhovať, nemôže ich však presadiť. Každý návrh na zmenu je zverejnený spoločným komunikačným kanálom, napríklad mailing-listom. Počas istého časového intervalu sa môže ktokoľvek vyjadriť k funkčnosti, efektívnosti, bezpečnosti, programátorskému štýlu, prípadne čomukoľvek ohľadne navrhovanej zmeny. Navrhovateľ zmeny môže prijať konštruktívnu kritiku a upraviť návrh svojej zmeny tak, aby spĺňal všetky predpoklady na správny, funkčný, efektívny a dobre napísaný kód. Ani keď nikto nevyjadrí negatívny postoj k návrhu, nie je isté, že zmena bude akceptovaná a pridaná do projektu. Rozhodujúce slovo má osoba

zodpovedná za celý projekt, prípadne za časť projektu. Pokiaľ sa rozhodne potvrdiť úpravu, je návrh prijatý a vykoná sa úprava zdrojových kódov projektu. Pokiaľ sa však rozhodne neprijať úpravu, tak sa úprava nedostane do výsledného kódu projektu.

Princípy Open-Source metódy

V tejto kapitole opíšem základné princípy metód vývoja používaných v Open-Source projektoch podľa [1].

Malé úpravy

Základom metódy vývoja používanej pri vývoji nekomerčných projektov sú malé úpravy. Malé zmeny sú vykonávané vo forme malých úprav, veľké zmeny sú taktiež vykonávané vo forme viacerých malých úprav.

Výhodou malých úprav je dobrá prehľadnosť, dobrá kontrolovateľnosť pridávaných úprav a malá pravdepodobnosť spôsobenia veľkej, závažnej chyby. Malé úpravy sa ľahko kontrolujú a na ich pochopenie nie je väčšinou potrebný dlhý čas. Tento čas je ešte možné znížiť vhodne zvolenými komentármi, dokumentáciou a vhodne volenými názvami premenných, funkcií a podobne. Čo sa týka chybovosti, tak pri kontrole malých úprav je oveľa ľahšie všimnúť si chybu alebo nedostatok, než pri veľkých úpravách.

Rozdelenie úloh

Ako som už spomínal v časti *Organizácia*, i napriek tomu, že v Open-Source metódach sa nepoužíva štandardný hierarchický model, existuje tu isté rozdelenie úloh [2]. Najpočetnejšou a z hľadiska vývoja produktu nezanedbateľnou skupinou je tím vývojárov. Vzhľadom na povahu Open-Source projektov sa môže hocikto zaradiť do skupiny vývojárov a pomáhať pri vývoji. Vývojári sa neformálne delia na dve skupiny, z ktorých jedni vytvárajú a pridávajú nové funkčnosti, zatiaľ čo druhí pracujú na odstránení známych chýb.

Druhou skupinou, dôležitou hlavne z dôvodu zabránenia šírenia chýb v projekte je skupina kontrolórov. Táto skupina má za úlohu kontrolu navrhovaných zmien. V prípade, že vývojár považuje svoju úpravu za pripravenú na vloženie do produktu, zverejní ju prostredníctvom spoločného komunikačného kanála a vtedy prichádzajú na radu kontrolóri. Kontrolóri preskúmajú navrhovanú úpravu z hľadiska správnosti, funkčnosti, bezpečnosti, prehľadnosti a podobne. Svoje pripomienky zverejnia v už spomínanom spoločnom komunikačnom systéme. Vývojár môže uznať vhodnosť pripomienky a prepracovať svoj návrh na úpravu.

Treťou a poslednou skupinou je skupina potvrdzovateľov. Potvrdzovateľ je osoba zodpovedná za celý projekt, prípadne jeho časť a má rozhodujúce slovo, či bude úprava prijatá do projektu, alebo nie.

Jednotlivé skupiny úloh nie sú navzájom sa vylučujúce. To znamená, že vývojár môže byť aj kontrolór, kontrolór môže byť vývojár ba dokonca jedna osoba môže byť vo všetkých troch skupinách súčasne.

V Open-Source projektoch je štandardné rozdelenie úloh také, že osoba zodpovedná za projekt (ktorá projekt vymyslela, založila, stará sa o projekt a pod.) prípadne osoby, ktoré tým poverila sú v pozícií potvrdzovateľov. Skupiny vývojárov a kontrolórov sú verejné, čo znamená, že hocikto má možnosť pridať sa do ľubovoľnej z nich, prípadne obidvoch.

Kontrola kódu

Veľmi dobrým a spoľahlivým spôsobom hľadania a odstraňovania chýb je kontrola kódu. Pri kontrole kódu sa podľa štatistických výskumov odhalí 60-90% všetkých chýb, čo je určite nezanedbateľné množstvo.

Kontrola kódu znamená, že niekto iný, ako autor kódu sa pokúsi pochopiť, čo a ako má kontrolovaný kód robiť, pri čom sa snaží zobrať do úvahy všetky prípady, ktoré môžu nastať a kontroluje, či sa v kóde nevyskytne chyba.

Čím viac ľudí kontroluje kód, tým je väčšia pravdepodobnosť odhalenia chýb. V Open-Source projektoch v závislosti od rozsiahlosti projektu kontroluje kód rádovo pár desiatok až pár desiatok tisíc ľudí. Všetci majú možnosť vyjadriť sa ku kontrolovanému kódu, prípadne navrhnúť jeho zmeny, úpravy a modifikácie.

Potvrdzovanie úprav

Úpravy sa do výsledného produktu dostávajú až potvrdením. Potvrdenie je proces, kde osoba zodpovedná za projekt (potvrdzovateľ) potvrdí vhodnosť a správnosť úpravy a pridá ju do výsledného produktu.

Čo sa týka vykonávania zmien výsledného produktu, má vždy posledné slovo potvrdzovateľ. I v prípade, že navrhovaná úprava prejde kontrolou kódu bez vážnejších pripomienok, má potvrdzovateľ možnosť rozhodnúť sa nezaradiť úpravu do výsledného produktu.

Vždy funkčná verzia

Dôležitým faktorom je, aby výsledný produkt bol vždy funkčný. Toto je možné dosiahnuť vďaka už spomínaným malým úpravám. Všetky úpravy musia mať taký charakter, aby neobmedzili celkovú funkčnosť výsledného produktu.

Tým sa zabráni tomu, aby bola posledná aktuálna verzia produktu nepoužiteľná, lebo je potrebné pridať ešte ďalšie nedorobené súčasti. Je lepšie mať funkčnú verziu, ktorá ešte neponúka všetky možnosti, ako mať verziu, ktorá ponúka všetky možnosti, ale je z nejakého dôvodu nepoužiteľná.

Podrobná dokumentácia

Dokumentácia je neoddeliteľnou súčasťou vývoja a bez dobrej dokumentácie je projekt pravdepodobne odsúdený na rýchly zánik.

Každá vykonaná úprava musí byť riadne okomentovaná a zdokumentovaná. Každá úprava v závislosti od závažnosti úpravy ovplyvní subverziu, prípadne verziu komponentu, resp. celého produktu. V prípade uvolnenia vydanej verzie je potrebné podrobne zaznamenať verzie všetkých súčastí a komponent, z ktorých sa výsledný produkt skladá.

Paralelný vývoj

Čas sú peniaze. Túto myšlienku možno s menšou obmenou aplikovať aj na nekomerčné Open-Source projekty. V Open-Source projektoch nejde o peniaze, ale projekty, ktoré sú vyvíjané tak dlho, že keď konečne uzrú svetlo sveta, sú už k ničomu, sú zbytočne vynaloženým úsilím.

Hlavným faktorom ovplyvňujúcim rýchlosť vývoja projektu je počet ľudí, ktorí na projekte pracujú. Pokiaľ však nemôžu pracovať paralelne, tak projekt postupuje veľmi pomaly. Práve vďaka už spomínaným malým zmenám je možné, aby na projektoch využívajúcich Open-Source metódy vývoja pracovalo viacero ľudí súčastne.

Ak sa jeden vývojár, resp. tím vývojárov rozhodne zásadne prepracovať časť projektu, tak do okamihu ako vytvoria použiteľný kód, nemôžu ostatní pracovať na tej istej časti. Avšak vzhľadom na to, že všetky úpravy sú vykonávané vo forme malých a častých úprav, je čas zdržania pri paralelnom vývoji zanedbateľný.

Iteratívny a inkrementálny vývoj

Ako už bolo spomínané v časti rozdelenie úloh, vývoj sa skladá z pridávania novej funkčnosti (inkrementálny vývoj) a z opravy známych chýb (iteratívny vývoj).

Iteratívnym vývojom sa vývojári snažia dosiahnuť bezchybnú a plne funkčnú verziu produktu. O tom, či je možné, alebo nie je možné dosiahnuť 100% bezchybnú verziu sa dá polemizovať. Podstatné je však, že je možné sa iteratívne (postupným približovaním) približovať bezchybnej verzii. Vývojári pracujúci na odstraňovaní chýb využívajú zoznam chýb, nazývaný *bug-list.* V tomto zozname sú uvedené všetky známe chyby a môže tu byť uvedená aj závažnosť chyby. Je logické, že najskôr sa pracuje na odstránení závažných chýb a až potom sa venuje čas odstraňovaniu chýb menej závažných.

Inkrementálny vývoj slúži na pridávanie novej funkčnosti do projektu. Podobne ako pri iteratívnom vývoji, aj v inkrementálnom vývoji existuje zoznam úloh, na ktorých by bolo vhodné pracovať. Tento zoznam sa nazýva *Todo-list.* Todo-list je vytváraný prevažne na základe ohlasu spotrebiteľa produktu, prípadne na základe ohlasu vývojárov.

Výhody

V predchádzajúcej kapitole som naznačil isté výhody modelu vývoja používaného pri Open-Source projektoch. V tejto kapitole sú zosumarizované najdôležitejšie výhody sú popísané, do akej miery sa môžu odraziť na úspešnosti softvérového projektu [2].

Časová úspora

Čas je jedným z rozhodujúcich faktorov úspešnosti projektu. Vo väčšine prípadov platí, že čím skôr je projekt hotový, tým lepšie. Taktiež v komerčnej sfére v zásade platí, že čím dlhšie sa pracuje na projekte, tým je projekt drahší.

Časová úspora opisovanej metódy spočíva prevažne v tom, že väčšina chýb sa objaví vo veľmi skorom štádiu. V závislosti od počtu kontrolórov kódu sa pri kontrole odhalí 60 – 90% všetkých chýb. A pri vývoji softvéru platí, že čím skôr sa chyba objaví, tým menej času a úsilia treba na jej nápravu. Podľa prieskumu uvedenom v [1] vyplýva, že i v prípade, že každú úpravu kontrolujú ďalší traja ľudia, celkový vývoj vyjde lacnejšie, ako v prípade, keď sa kód nekontroluje.

Rýchle odozvy na požiadavky

Ďalším dôležitým faktorom je pružnosť projektu. Pružnosť závisí od toho, ako rýchlo je projekt schopný reagovať na požiadavky.

Keďže pri spomínanej metóde je stále udržiavaná funkčná posledná verzia a sú vykonávané iba malé, časovo nenáročné úpravy, je možné v prípade potreby veľmi pružne zareagovať na požiadavku s vysokou prioritou. Príkladom môže byť napr. oprava závažnej bezpečnostnej chyby, ktorá má bezpochyby vysokú prioritu. V prípade, že by neboli vykonávané drobné úpravy, ale veľké úpravy a chyba by sa vyskytla práve v časti, ktorá sa prerába, bolo by potrebné s novou, opravenou počkať na dorobenie celej časti. Pri drobných úpravách však nie je potrebné čakať na dokončenie celého celku a je možné chybu opraviť takmer okamžite.

Chyby sa nedostanú do vydanej verzie

Chyby sú veľmi zlou vizitkou celého projektu a spôsobujú veľmi veľa problémov. Ideálny stav = nerobiť žiadne chyby sa prakticky nedá dosiahnuť. Keďže sa chybám nedá vyhnúť, otázkou je iba to, kedy a ako chyby zistíme. Pre dobré meno projektu je určite lepšie, keď chybu odhalia vývojári, prípadne kontrolóri, ako keby mal chybu nájsť zákazník / používateľ produktu.

Čím neskôr zistíme chybu, tým viac času a úsilia musíme vynaložiť na to, aby sme ju odstránili. Open-Source metóda sa v značnej miere sústredí na skoré odhaľovanie chýb. Práve vďaka kontrole kódu, ktorá je vykonávaná nad každou navrhovanou zmenou sa prevažná väčšina chýb zistí a odstráni ešte pred tým, než je pridaná do vydanej verzie.

Kontrola šírenia chýb

Ak sa predsa len voľajaká chyba dostane do vydanej verzie, je veľmi užitočné vedieť, ktorých verzií sa chyba týka a čo všetko mohla chyba ovplyvniť.

Pokiaľ máme zdokumentovanú každú vykonanú úpravu, je relatívne jednoduché zistiť, do ktorých verzií sa chyba rozšírila. Následnou kontrolou úprav vykonaných po vzniku chyby je možné zistiť, či sa vplyv chyby nerozšíril aj do iných častí projektu.

Nevýhody

Okrem výhod má spomínaná metóda samozrejme aj nevýhody. V tejto kapitole sa zameriam na najdôležitejšie z nich [2].

Prototyp / vydanie

Keďže vždy je k dispozícií funkčná verzia, je potrebné rozhodnúť, kedy bude verzia označená za vydanú verziu, označiť ju príslušným číslom verzie a dať ju k dispozícií používateľom.

Nie vždy je ľahké zhodnotiť a rozhodnúť, kedy nastal pravý čas na označenie verzie za vydanú. Je potrebné zhodnotiť závažnosť a množstvo vykonaných úprav.

Nepodarený kód

Pri každom väčšom projekte je potrebné veľmi dôsledne sledovať a kontrolovať vykonávané úpravy a zmeny kódu, prípadne nový pridávaný kód.

Okrem funkčnosti a efektívnosti je dôležitý aj programátorský štýl použitý pri vývoji. Pokiaľ na vývoji pracuje veľké množstvo vývojárov, a keďže každý rozmýšľa mierne odlišne, môže sa veľmi ľahko stať, že kód sa postupne stane neprehľadný, neohrabaný a veľmi ťažko modifikovateľný. A práve potvrdzovateľ má za úlohu zabrániť vzniku takéhoto nepodareného kódu. Aby sa zabránilo vzniku nepodareného kódu má potvrdzovateľ možnosť rozhodnúť, že zmena sa do projektu nepridá i napriek tomu, že funguje.

Veľké projekty

Opisovaná metóda je takmer ideálna pre malé projekty, pri väčších však môžu nastať problémy s manažovateľnosťou celého projektu.

Pokiaľ je potvrdzovateľ iba jeden, prípadne malá skupina spolupracujúcich osôb, nie je s manažovateľnosťou projektu veľký problém. Pokiaľ je však projekt takého rozsahu, že malá skupina ľudí nezvláda potvrdzovať všetky vykonávané úpravy, je potrebné vytvoriť v skupine potvrdzovateľov hierarchickú štruktúru.

Záver

I napriek tomu, že Open-Source je veľmi často vyvíjaný zadarmo, nie je vyvíjaný iba nadšencami, amatérmi a študentmi. Veľké množstvo vývojárov pracujúcich na Open-Source projektoch sú profesionálni vývojári vyvíjajúci softvér aj v komerčnej sfére.

Spomínaná metóda však nie je obmedzená iba pre použitie v Open-Source projektoch, ale s menšími úpravami je použiteľná i v komerčných projektoch.

Použitá literatúra

1. Stephane Lussier, Macadamian technologies: *New Tricks: How Open Source Changed the Way My Team Works.* Focus: developing with open source software, IEEE software.
2. MSc Thesis Kim Johnson: *A Descriptive Process Model for Open-Source Software Development*. University of Calgary, June 2001, http://sern.ucalgary.ca/students/theses/KimJohnson/toc.htm (anglicky).

Annotation

Commerce versus non-commerce software development methods

Software development is long and difficult process. There are many methods for commerce software development. They helps to increase developers and managers efficiency and decreasing software development costs. By the time, some methods were developed also in non-commerce sphere. Some methods used by Open-Source communities are more efficient, easier to manage and more error proof than some methods used in commerce sphere. Usage of this methods is not limited only for non-commerce projects. This paper try to compare mostly used non-commerce method with other, commerce methods.

Udržovateľnosť a Open Source Software

PAVOL GONO

Slovenská technická univerzita
Fakulta informatiky a informačných technológií
Ilkovičova 3, 842 16 Bratislava

Abstrakt. Cieľom tejto eseje je poukázať na udržovateľnosť softvérových produktov a faktory vplývajúce na udržovateľnosť softvéru. Porovnáva sa udržovateľnosť medzi softvérom s otovreným zdrojovým kódom a softvérom s uzavretým zdrojovým kódom. Analyzujú sa možnosti merať udržovateľnosť, existujúce metriky, kompozitné metriky. Hodnotia sa vykonané merania v reálnych softvéroch v tejto oblasti.

Úvod

Podnetom pre túto esej bola práca autorov Ioannis Samoladas, Ioannis Stamelos, Lefteris Angelis a Apostolos Oikonomou s názvom „Open source software development should strive for even greater code maintainability“ [1]. Autori sa venujú problematike merania udržovateľnosti Open Source Software, konkrétne piatich populárnych projektov.

V tejto eseji chcem bližšie rozobrať možnosti merania udržovateľnosti, uviesť vhodné metriky a priblížiť ďalšie merania konkrétnych softvérových projektov. Tiež uvediem všeobecné zásady na tvorbu dobre udržovateľných programov.

V prvej časti eseje je definovaná udržovateľnosť a údržba softvéru. Druhá časť rozoberá Open Source Software. V tretej časti sa hovorí o rozdieloch medzi Open Source Software a Closed Source Software. Štvrtá časť analyzuje možnosti merania softvéru. V piatej časti sú uvedené výsledky merania softvérov a v šiestej časti sú zásady tvorby dobre udržovateľných programov.

Čo je to udržovateľnosť

Jedna z definícií udržovateľnosti je: Úsilie, ktoré treba vynaložiť na ďalší vývoj a údržbu výrobku podľa meniacich sa potrieb zákazníka a aj meniaceho sa okolia. [2]

S udržovateľnosťou úzko súvisí samotná údržba softvéru. Ak je dobrá udržovateľnosť, ľahšie a rýchlejšie sa vykonáva údržba. Údržbu rozdeľujeme na [3]:

- opravnú (corrective maintenance): odstraňovanie všetkých chýb, ktoré pretrvávajú v systéme (chyby špecifikácie, návrhu, implementácie, dokumentácie).
- prispôsobovaciu (adaptive maintenance): menenie softvéru v závislosti od zmien okolitého prostredia, tiež zmeny časti softvéru v dôsledku modifikácie inej časti.
- zlepšovaciu (perfective maintenance): vylepšovanie vlastností produktu, ktoré neboli zahrnuté do špecifikácie pôvodného systému, tiež zmeny v závislosti od zmien problémovej oblasti a požiadaviek používateľa.
- preventívnu (preventive maintenance): modifikácie produktu s cieľom zlepšenia ďalšej údržby.

Aký softvér zaraďujeme k Open Source Software

Existuje aj viac definícií systémov s otvoreným zdrojovým textom programu (ďalej OSS – Open Source Software). Najčastejšie používaná definícia sa nachádza na stránke Open Source Iniciatívy [4]. Je rozpísaná v bodoch, ktoré sa venujú voľnej redistribúcii, zdrojovému kódu, odvodenej práci, integrite zdrojového kódu autora a pod. Tu sú však zdôraznené iba požiadavky na licenciu OSS (umožnenie bezplatného prístupu k danému softvéru a jeho zdrojovým textom, každému záujemcovi umožniť modifikáciu daného softvéru, ...). Príkladom OSS licencií je GPL, BSD, apache license, Mozilla Public License. Pre účely tejto eseje bude vhodnejšie chápať OSS ako softvér, pri ktorého vývoji sa použil špecifický proces, najčastejšie nazývaný ako Katedrála a Bazár [5].

Rozdiely medzi OSS a CSS z pohľadu udržovateľnosti

Obyčajný začiatok OSS projektu: vývojár alebo malá skupinka vývojárov napíše prvú verziu softvéru, táto sa poskytne na Internet, a pozvú sa ďalší vývojári, aby prispievali kúskami kódu. Niektorí vývojári majú hlavné slovo, ktorí riadia, akým smerom sa bude vývoj uberať, aké úpravy kódu sa uskutočnia, časový plán releasov.

Obyčajný začiatok CSS projektu: manažéri softvérovej spoločnosti sa rozhodnú vyvíjať určitý produkt (napr. na základe ponuky trhu, požiadaviek zákazníka), vytvorí sa tím vývojárov, dohodne sa časový plán a s určitým balíkom peňazí sa začne vyvíjať projekt.

Stáva sa, že CSS vývoj sa zmení na OSS, napr. aby sa dosiahli výhody takéhoto vývoja. Taktiež sa stáva, že OSS sa zmení na CSS, zvyčajne však pre iné príčiny.

V CSS nastáva podľa mňa často nasledujúca situácia. Príde zákazka na vývoj softvérového produktu, stanoví sa pomerne krátka doba, za ktorú je potrebné produkt vytvoriť. Programátori v snahe splniť požiadavky vyprodukujú výsledok rýchlo, ale už sa nemyslí veľmi na nasledujúcu údržbu produktu. Môže existovať podvedomé

odmietanie písanie prehľadného kódu – Ja si túto časť programujem, keď bude treba spraviť zmenu, ja ju budem opravovať, tak načo písať nejaké ďalšie komentáre, veď ja tomu rozumiem. Pri OSS projektoch programátori rátajú (alebo by mali rátať) s tým, že ich kód bude určite čítať aj niekto iný, takže je tu väčší predpoklad, že sa kód napíše prehľadnejšie. Ťažko to však presne potvrdiť a už vôbec sa to nedá zovšeobecniť.

V OSS je často nekompletná dokumentácia (možná výhovorka – dajú sa pozrieť zdrojové texty) alebo neexistujúca technická podpora (hlavne u menších projektov sa vyskytuje prístup - dorob si sám). Často sa ale vyskytujú aj opačné prípady, napríklad z vlastných skúseností - ak som mal problémy s používaním niektorých open-source programov, veľakrát som dostal rýchlu odpoveď na mailing-liste.

Merateľnosť udržovateľnosti

Dostatočne presným meraním udržovateľnosti by bolo merať strednú dobu potrebnú na opravy chyby, strednú dobu potrebnú na pochopenie logiky modulu a strednú dobu potrebnú na sprístupnenie príslušnej informácie v dokumentácii.

Takéto meranie by ale bolo v praxi veľmi nepohodlné, pretože by si programátori museli zapínať a vypínať stopky pri bežnej činnosti. Taktiež by sa nedalo takéto meranie urobiť externe nezúčastnenými osobami. Výhodnejším prístupom sa javí hodnotiť samotný zdrojový text softvéru, pretože ten veľa hovorí o samotnom softvéri. Vlastnosti ako štýl programovania, jazyk, komentáre, samoopisnosť, zložitosť vplývajú na samotné vlastnosti softvéru. V modernom softvérovom inžinierstve platí domnienka, že externé charakteristiky kvality sú vo vzťahu k interným charakteristikám kvality. To znamená, že merania v zdrojových textoch môžu do určitého rozsahu predpovedať externé charakteristiky kvality systému, ako sú udržovateľnosť, spoľahlivosť, rozšíriteľnosť, alebo prenosnosť.

Autori McCall a kolektív už v 70 rokoch definovali takéto faktory ovplyvňujúce udržovateľnosť: jednoduchosť, presnosť, samoopisnosť, modularita.

V súvislosti so zlou udržovateľnosťou môžu vznikať problémy pri údržbe [3]:

- Ťažkosti so sledovaním, riadením vývoja softvéru, ktorý existuje v mnohých verziách.
- Problémy s porozumením programov „niekoho iného“.
- Na údržbu sa často nasadzujú neskúsení softvéroví inžinieri, podceňovanie údržby.
- Udržiavané programy vznikli často pred veľa rokmi, nepoužili sa moderné techniky, nie sú štruktúrované, okomentované.
- Zmeny často spôsobujú chyby a následne ďalšie zmeny.
- Degradácie štruktúry meneného softvéru
- Vo všeobecnosti nízka kvalita dokumentácie
- Udržiavaný systém často nemožno vysadiť z prevádzky

Autori článku [1] používajú tieto metriky pre meranie zdrojových kódov:

- počet riadkov textu programu (LOC) – meria fyzickú veľkosť programu bez prázdnych riadkov a komentárov. (Táto metrika môže vyjadrovať celkovú zložitosť systému, čas potrebný na pochopenie systému)
- Percento riadkov komentárov vzhľadom na počet riadkov textu programu (PerCM). Vyjadruje „samoopisnosť“ zdrojového textu.
- Metrika „Halstead Volume“ (V). n1 = počet odlišných operátorov, n2 = počet odlišných operandov, N1 = celkový počet operátorov, N2 = celkový počet operandov, n = n1 + n2 (slovník programu), N = N1 + N2 (dĺžka programu). Nakoniec $V = N * (\log_2 n)$ je výsledná zložená metrika. Poskytuje alternatívny pohľad na veľkosť programu.
- Cyklomatické číslo V(g). Táto metrika sa zakladá na počte nezávislých ciest v grafe riadenia programu. Jeho hodnota závisí na počte vetiev vytvorených vetvením programu (if-else konštrukcie). Meria štrukturálnu zložitosť programu.

Podľa štúdie 25 miliónov riadkov kódu v National Security Agency [6], metriky na meranie udržovateľnosti a čitateľnosti boli použité tieto:

- Počet riadkov kódu. Nemá prekročiť 62 riadkov kódu na jednu funkciu.
- Počet vykonávateľných príkazov. Nemá prekročiť 50 vykonávateľných príkazov na jednu funkciu.
- Počet riadkov komentárov. Mal by byť najmenej 60% na funkciu. Toto môže trochu udiviť, ale NSA má veľmi veľa zastaraného kódu, ktorý sa niekedy používa aj desiatky rokov. Slabo komentovaný kód je ťažké pochopiť a horšie sa mení.
- Rozpätie odkazu na premennú. Najväčší priemerný počet riadkov medzi priradením hodnoty do premennej a použitím premennej vo funkcii. Nemal by byť väčší ako 10-12 riadkov z rôznych bezpečnostných dôvodov.

Autori článku [7] skúmajú 2 merateľné veličiny

- Globálna zviazanosť. Zviazanosť je miera previazanosti súčiastok, ideálne keď je minimálna. Globálna zviazanosť je jav, keď sa súčiastky (moduly) odvolávajú na spoločné údaje. Je to nepríjemný jav, pretože chyba v jednej súčiastke môže spôsobiť nesprávne správanie v iných súčiastkach. Ťažko sa určuje kto a kedy modifikoval spoločné údaje. Priame väzby medzi súčiastkami sú skryté, z toho vyplývajú problému pri zmenách a údržbe. Je ľahké si predstaviť, že väčšia globálna zviazanosť má nepriaznivý vplyv na udržovateľnosť. Metrika sa určila ako počty odkazov na globálne premenné (nie konštanty), pričom v rámci jedného modulu sa viacnásobný odkaz na tú istú premennú počítal iba raz.
- Počet riadkov kódu (LOC).

Merania zdrojových textov produktov OSS

Autori [1] chceli použiť kompozitnú metriku, ktorá by vyjadrovala jedno číslo, ktorým by sa ľahko porovnávali rôzne softvéry. Použili kompozitnú metriku – Maintainability Index (MI), ktorá bola navrhnutá SEI v [8] ako najviac vhodný nástroj na meranie udržovateľnosti pre systémy, na ktoré sa kladú vysoké nároky na kvalitu. Koeficienty na výpočet tohto indexu boli kalibrované podľa mnohých systémov, ktoré udržoval Hewelt Packard:

MI = 171 – 5.2ln(avgV) – 0.23avgV(g) – 16.2ln(avgLOC) + 50sin(sqrt(2.4avgPerCM))

avgV znamená priemerné Halstead Volume na modul, a podobne aj avgV(g), avgLOC a avgPerCM. Vyššie hodnoty MI znamenajú vyššiu udržovateľnosť. Vzorec berie do úvahy veľkosť, zložitosť a samoopisnosť zdrojového textu.

Sčítanec 50.sin(...) znamená funkciu, ktorá ma lokálne maximum v prípade avgPerCM=1,028. Vyššie hodnoty MI tiež získame, keď avgV bude nižšie (menšia zložitosť podľa Halstead-a), keď avgV(g) bude nižšie (menšie cyklomatické číslo) a keď avgLOC bude nižšie (menšia veľkosť systému).

Predbežné závery autorov [1]:

1. Použitím nástrojov ako MI na meranie, kvalita OSS kódu vyzerá byť rovnaká, prípadne lepšia ako kvalita CSS kódu. Môže to byť zapríčinené motiváciou skúsených OSS programátorov konkurovať CSS programátorom. Vysoká motivácia môže byť považovaná za veľkú výhodu v OSS.
2. OSS projekty potrebujú obozretnú individuálnu analýzu kvôli náhlym zmenám medzi rôznymi verziami softvéru, pre ktoré sa rozhodli koordinátori OSS projektu. Takéto zmeny majú veľký vplyv na usporiadanie OSS softvéru. Štrukturálna analýza kódu môže poukázať na komponenty, ktoré sú najviac rizikové a môžu zapríčiňovať najviac problémov. Takéto komponenty sa dajú identifikovať pomocou uvedených meraní.
3. Kvalita kódu OSS trpí tými istými problémami ako kvalita kódu CSS. Znehodnocovanie udržovateľnosti časom je typickým fenoménom a vznikajú tým zastarané CSS systémy. Je logické, že to isté sa bude diať s OSS systémami. To znamená, že v OSS projektoch je taktiež nutná vhodná preventívna údržba a prípadná reštrukturalizácia.

Závery merania udržovateľnosti linuxového jadra [7]:

- Počet riadkov kódu narastá so vzrastajúcimi verziami jadra lineárne. Každá ďalšia verzia jadra ponúka novú funkcionalitu, preto je logické, že sa počet riadkov zväčšuje. Fakt, že veľkosť jadra vzrastá iba lineárne, by mohla byť indikácia, že jadro a jeho moduly sú dobre navrhnuté – iba malé množstvo prídavného kódu je nutné vložiť do rozhrania jadra a modulov, aby sa dosiahla nová funkcionalita.

- Množstvo výskytu globálnej zviazanosti narastá so vzrastajúcimi verziami jadra exponenciálne. Dá sa predpokladať, že takýto rast sa nezastaví pokiaľ sa linux kompletne nezmení štruktúru na minimálne používanie globálnej zviazanosti. V opačnom prípade sa dá očakávať časom zhoršujúca sa udržiavateľnosť.

Pre programátorov

Existujú všeobecné zásady, ktorými by sa mal riadiť vývojári, aby dosiahli dobrú udržovateľnosť zdrojových textov:

- ak je telo funkcie niekoľko desiatok riadkov dlhé, treba uvažovať nad rozdelením funkcie do viacerých
- aj je ten istý kód napísaný len s miernymi modifikáciami dvomi alebo viacerými spôsobmi, pravdepodobne sa to urobilo nevhodne (treba pouvažovať nad jednou funkciou s parametrami)
- používať šablóny namiesto opakovania kódu (ak to programovací jazyk dovoľuje)
- používať konštanty v čo najväčšej miere (ak to programovací jazyk dovoľuje)
- používať objekty polí namiesto polí definovaných na mieste použitia
- ak sa viaceré premenné používajú často spoločne, zapuzdriť ich do objektu
- ak viaceré funkcie pracujú iba s jedným typom objektov, urobiť v objekte tieto funkcie metódami
- už len základná refaktorizácia urobí zdrojový text čitateľnejším a organizovanejším

Záver

V tejto eseji sa bližšie rozobrali možnosti merania udržovateľnosti, uviedli sa vhodné metriky a priblížili sa ďalšie merania konkrétnych softvérových projektov. Verím, že čitateľom ponúkli lepší prehľad v tejto problematike.

Čo sa týka udržovateľnosti OSS projektov verzus CSS projektov, podľa mňa závisí hlavne na konkrétnom projekte, na konkrétnej situácii a na konkrétnych ľuďoch. Existuje veľa projektov, ktoré majú mizernú kvalitu a udržovateľnosť a sú projekty, ktorých zdrojové kódy je radosť čítať a človek má po chvíli dojem, že tomu rozumie. Rovnako pre OSS, rovnako pre CSS.

Použitá literatúra

1. Ioannis Samoladas, Ioannis Stamelos, Lefteris Angelis, and Apostolos Oikonomou. Open source software development should strive for even greater code maintainability. *Communications of the ACM*, 47(10):83-87, October 2004.
2. Institute of Electrical and Electronics Engineers. IEEE Standard Computer Dictionary: A Compilation of IEEE Standard Computer Glossaries. New York, NY: 1990.
3. Bieliková, M.: *Softvérové inžinierstvo. Princípy a manažement*. Vydavateľstvo STU, Bratislava, 2000.
4. The Open Source Initiative. Open source definition, version 1.9; www.opensource.org/docs/definition.php
5. Raymond, Eric S.: The Cathedral and the Bazaar. www.catb.org/~esr/writings/cathedral-bazaar/cathedral-bazaar/cathedral-bazaar.ps.
6. Drake, T. Measuring software quality: A case study. *IEEE Computer 29,* 11 (1996), 78–87.
7. Schach S.R., Jin B., Wright D.R., Heller D.Z., and Offutt A.J. Maintainability of the Linux kernel. In *IEE Proceedings—Software Engineering 149,* 1 (2002), 18–24.
8. Maintainability index technique for measuring program maintainability. Software Technology Review, SEI; www.sei.cmu.edu/str/descriptions/mitmpm_body.html.

Annotation

Maintainability and Open Source Software

The aim of this essay is to mention maintainability of software products and factors which influence maintainability of software. I compare maintainability between Open Source Software and Closed Source Software. I analyse possibilities of measuring maintainability, existing metrics and composite metrics. Measurements of real software products are evaluated in this section.

Vplyv rizík softvérových projektov na výsledky projektu

BOHUSLAV SZABO

Slovenská technická univerzita
Fakulta informatiky a informačných technológií
Ilkovičova 3, 842 16 Bratislava

Abstrakt. Predkladaný dokument sa zaoberá rôznymi pohľadmi na analýzu rizík v softvérových projektoch a ich účinok na výsledky samotného produktu ako aj procesu jeho tvorby. Text vysvetľuje kategorizáciu rizík na základe predpokladanej závažnosti rizikových faktorov a očakávaných možnostiach ich riadenia zo strany manažéra. Podľa týchto kritérií sú rozobrané vplyvy jednotlivých skupín rizík na výsledky projektu a ich vzájomné interakcie. Podceňovanie riadenia rizík pri vývoji softvérového projektu je častým dôvodom jeho zlyhania. Článok poukazuje na základné aspekty dôležitosti riadenia rizík. Jedna kapitola je venovaná analýze rizík tímového projektu September Project Partnership. Sú identifikované riziká, ktoré projekt ovplyvnili, analýza ich dopadu a spôsoby riešenia. Záver textu sa venuje stručnému popisu metód analýzy rizík, ako sú „RIPRAN“, rozhodovací strom, DELPHI.

Úvod

Mnoho softvérových projektov končí neúspechom. Aby bolo možné vykonať nápravné kroky, je potrebné identifikovať faktory, ktoré vplývajú na úspešnosť projektu a jeho výsledky. Mnohé štúdie poukazujú na fakt, že za veľkou časťou neúspešných projektov stojí nedostatočná analýza a riadenie rizík.

Cieľom tejto eseje je sprístupniť rôzne pohľady na analýzu a identifikáciu rizík. Práve na identifikáciu rôznych rizík v projektoch bolo v posledných rokoch vynaložené veľké úsilie a množstvo prostriedkov, avšak na analýzu ich vplyvov na výsledky projektu len minimum.

Zlyhanie pri pochopení a riadení súvisiacich rizík projektu môže často viesť k zlyhaniu celého projektu. Manažéri môžu len získať lepším pochopením toho, ako riziká softvérového projektu vplývajú na jeho celkové výsledky, a menej projektov tak môže končiť neúspechom.

Identifikácia rizík

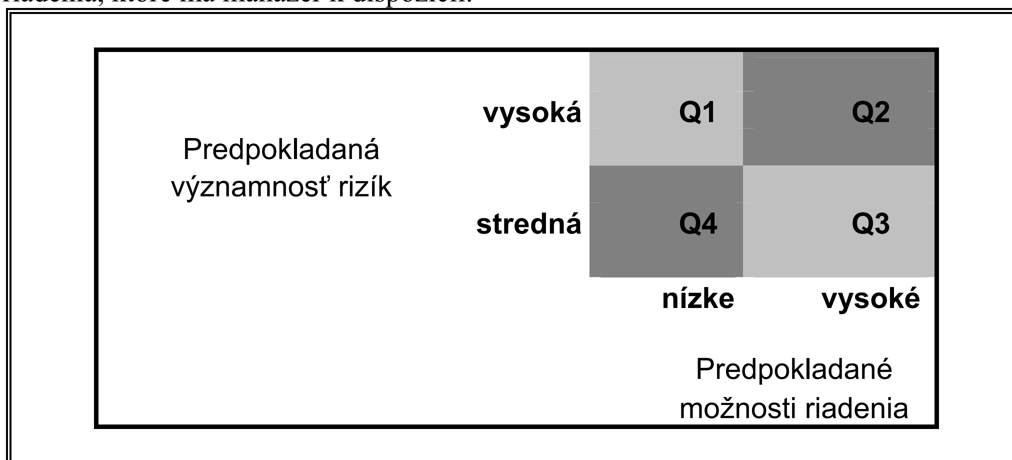
Prvou etapou pri manažovaní rizík je ich identifikácia. V nej je potrebné presne definovať všetky faktory, ktoré môžu nepriaznivo vplývať na projekt, čím sa rozumie popis ich charakteristík. Pre identifikovanie rizík bolo vyvinutých viacero metód, ktorých vhodnou kombináciou je možné dosiahnuť popis širokej bázy rizikových faktorov.

Základnou metódou je zoznam rizík, ktorý sa tvorí prevažne na základe skúseností z predchádzajúcich projektov. Ako ďalší krok je potrebné analyzovať predpoklady, keď najmä optimistické môžu predstavovať riziko. Rovnako tak je potrebné analyzovať všetky rozhodnutia, ktoré boli prijaté napríklad z politických dôvodov. Na tvorbu zoznamu rizík sa často používajú dotazníky alebo interview.

Kategorizácia rizík

Identifikovanie rizík komplikujúcich vývoj softvérových projektov a ich zapracovanie do koherentných stratégií manažovania rizík nie je jednoduchou úlohou. Pokým boli navrhnuté viaceré zoznamy rizikových faktorov, ktoré by mali byť riadené, pomerne málo úsilia bolo venovaného na samotnú organizáciu rizík a štúdiu efektov, ktorými môžu vplývať na celkové výsledky projektu. Výsledkom je, že softvéroví manažéri majú málo formálnych procedúr, ktoré by ich viedli pri práci. Po identifikácii je ďalším krokom k úspešnému riadeniu rizík ich kategorizácia.

Jeden z prvých pohľadov na tvorbu užitočného nástroja pre riadenie rizík je prezentovaný v [2]. Tak ako to ukazuje **Obr. 1**, všetky identifikované riziká sú rozdelené podľa ich predpokladanej významnosti a predpokladaných možností riadenia, ktoré má manažér k dispozícii.



Obr. 1. Kategorizácia rizík

Opíšme si bližšie jednotlivé kvadranty tohto rozdelenia.

Q1 predstavuje rizikové faktory vzťahované na zákazníka. Je zrejmé, že problémy v tejto oblasti budú mať zásadný vplyv na úspech celého projektu. Zahŕňa nedostatky najvyššieho manažmentu a neadekvátne angažovanie používateľa. Tieto faktory sú často mimo kontroly manažéra, a preto ich považujeme za kritické.

Q2 sa zameriava na rizikové faktory spojené s neschopnosťou manažéra pri posudzovaní rozsahu projektu a konkrétnych požiadaviek v danej problémovej oblasti, zahŕňa riziká spojené s požadovanou funkcionalitou. V tejto oblasti by projektoví manažéri mali byť schopní riadiť väčšinu rizík.

Q3 je spojený s konkrétnou realizáciou projektu. Obsahuje najmä riziká spojené s neadekvátnym obsadzovaním projektu, nevhodnou metodológiou vývoja, nedostatočným definovaním rolí a zodpovedností, slabým plánovaním a riadením projektu. Veľká väčšina týchto problémov by mala byť relatívne ľahko riešiteľná, preto ich zaraďujeme do strednej triedy závažnosti.

Q4 rovnako zaraďujeme do strednej triedy závažnosti. Zameriava sa na rizikové faktory v externom aj internom prostredí. Časť problémov zahrnutých v tomto kvadrante pramení z organizačných zmien, prípadne politických zmien, ktoré môžu vplývať na projekt. Je zrejmé, že možnosti riadenia v tejto oblasti sú veľmi malé.

Predloženú kategorizáciu možno pokladať za intuitívnu, avšak je potrebné odpovedať na otázku, či a ako ovplyvňujú riziká predstavované jednotlivými kvadrantmi celkové výsledky projektu, či sa niektoré kvadranty v jednotlivých rizikách neprekrývajú. Je teda potrebné bližšie popísať rôzne typy rizík a ich možné vzájomné interakcie.

Aké faktory teda prislúchajú jednotlivým kvadrantom ?

Q1	nedostatočná spolupráca používateľa, neakceptovanie zmien, konflikty medzi používateľmi, používatelia s negatívnymi očakávaniami, nedostatočná podpora vyššieho manažmentu
Q2	nedefinované kritéria pre úspech projektu, konflikty v jednotlivých požiadavkách systému, neustále sa meniace požiadavky resp. rozsah, nejasné a neadekvátne definované požiadavky, zložité identifikovanie vstupov a výstupov systému
Q3	neadekvátne školení alebo neskúsení členovia tímu, časté konflikty medzi členmi tímu, zlá metodológia vývoja, zlé plánovanie projektu, negatívne očakávania členov tímu, projekt zahŕňa použitie novej technológie, vysoká úroveň technickej komplexnosti, nedostatočné monitorovanie procesu tvorby produktu, neadekvátny odhad rozpočtu, neefektívny manažment projektu, neskúsení manažéri, neefektívna komunikácia
Q4	organizačné zmeny zahŕňajúce presun ľudí a prostriedkov, politické zmeny, závislosť od vonkajších dodávateľov, veľa externých zdrojov podieľajúcich sa na vývojovom procese

Tab. 1. Rizikové faktory rozdelené do kvadrantov

Pôsobenie rizík na výsledky projektu

Pri analýze typov rizík uvedenej v [1] bolo zistené, že len riziká kvadrantov Q2 a Q3 majú významný vplyv na to, či bude projekt ukončený načas a v danom rozpočte, pričom nebola identifikovaná žiadna interakcia medzi týmito dvomi kvadrantmi. Tieto výsledky dávajú intuitívne zmysel, keďže chabo riadené projekty ako aj projekty zaoberajúce sa nestabilnou oblasťou, kde sa požiadavky môžu často meniť, spravidla prekračujú časové plány i rozpočet. Zistilo sa tiež, že riziká realizácie projektu sú až dvakrát významnejšie ako riziká rozsahu a požiadaviek. Keďže riziko realizácie reprezentujú faktory spojené s projektovými tímami, projektovou komplexnosťou a riadením projektu, musia sa manažéri viac zamerať na záležitosti rozpočtu a časového plánu.

Štatistické analýzy ukázali, že najmä kvadranty Q1, Q2 a Q3 majú výrazný vplyv na výsledky projektu, naproti tomu kvadrant Q4 nie je tak významný. Riziká prostredia, ktoré sa s týmto kvadrantom spájajú sú relatívne ojedinelé a takmer vždy sú nepredvídateľné. Mnohí projektoví manažéri v tejto súvislosti dokumentovali ich skúsenosti s mnohými rizikovými faktormi prostredia so záverom, že ich významnosť je možné výrazne potlačiť najmä presne definovanými požiadavkami a vynikajúcou realizáciou projektu. Najmä pre malú početnosť výskytu rizík prostredia tieto identifikujeme ako menej významné pre celkové výsledky projektu.

Medzi jednotlivými kvadrantmi boli identifikované viaceré vzájomné vzťahy podieľajúce sa na výsledku projektu. Najdôležitejším spozorovaným faktom je, že zníženie rizík realizácie prináša aj výrazne zníženie vplyvu rizík ostatných dvoch kvadrantov na celkové výsledky projektu. Na druhej strane, ak ostane riziko realizácie vysoké, efekt ostatných dvoch kvadrantov sa môže až dvojnásobne zvýšiť. Ak teda nie je možné znížiť riziká spojené s realizáciou, je nutné výrazne sa zamerať na riziká spojené s rozsahom, požiadavkami a používateľom. Je teda potrebné si zapamätať, že ak sa nám podarí znížiť riziká realizácie, výrazne sa znížia aj riziká používateľa, rozsahu a požiadaviek.

Ak chceme produkovať úspešnú aplikáciu, musíme sa ako softvéroví manažéri naučiť riadiť riziká realizácie. Ak sú tieto riadené efektívne, minimalizuje sa vplyv ostatných rizikových faktorov na celkové výsledky projektu.

Analýza rizík tímového projektu September Project Partnership

V tejto kapitole by som rád analyzoval riziká a ich vplyv na výsledky projektu tímu September Project Partnership, ktorý mal za úlohu upraviť aplikáciu pre podporu prideľovania, odovzdávania a posudzovania projektov na našej fakulte. Keďže sa jednalo o pokračovanie v už existujúcom projekte, o rizikové faktory nebola núdza. V tejto kapitole by som chcel zhrnúť riziká, s ktorými sme sa stretli a analyzovať ich vplyv na výsledky projektu.

Hneď na úvod musím poznamenať, že skupina rizík, ktorú sme v predchádzajúcich kapitolách identifikovali ako najkritickejšiu (Q1), sa nám vo veľkej miere vyhla. Spolupráca zákazníka bola na vysokej úrovni, keďže jeho záujmom bolo,

aby výsledná aplikácia bola použiteľná. Zákazník mal vedomosti o vývoji produktu z jeho minulých rokov, takže vedel, aké požiadavky môžu byť splniteľné a k výsledkom projektu sa staval optimisticky. Toto bol hlavný základ úspechu celého projektu.

Rovnako z hľadiska druhého kvadrantu možno prácu tímu označiť za vyhovujúcu pre elimináciu rizík spojených s funkcionalitou. Už na jednom z prvých stretnutí sme jasne definovali, že úspechom projektu bude jeho nasadenie pri odovzdávaní projektov na konci letného semestra školského roku 2004/2005. Zákazník mal s produktom bohaté skúsenosti, čo prispelo k presnému definovaniu požiadaviek. Niektoré z nich sa síce menili, ale po dôkladnej analýze alternatív sa vždy vzniknuté rozpory podarilo v adekvátnom čase vyriešiť. Základným problémom boli niektoré menšie zmeny v databáze, ktoré sa počas analýzy javili ako vhodné, avšak najmä z časových posunov v pláne projektu, ktoré vznikli najmä rizikami tretieho kvadrantu, sme od týchto zmien ustúpili. Tu teda možno vidieť, ako riziká tretieho kvadrantu ovplyvnili riziká kvadrantu druhého. A ktoré to teda boli ?

V projekte September Project Partnership by som za zdroj najväčších rizík označil tretí kvadrant. Dôvodom je najmä, že sme pokračovali v už existujúcom projekte. Znamenalo to, že produkt bolo najprv potrebné naštudovať. Čoskoro sme zistili, že skúsenosti členov tímu s použitými technológiami tiež nepostačujú požiadavkám tohto projektu a preto bolo nutné aj ich štúdium. Vyzdvihnúť treba dobré plánovanie projektu a vzornú tímovú spoluprácu, čím sme vzniknuté riziká dostatočne eliminovali. Pomerne veľký časový posun spôsobil fakt, že k projektu, v ktorom sme pokračovali, nebolo možné získať zdrojové texty aktuálne používanej verzie. S týmto sme samozrejme pri plánovaní nerátali. Dekompiláciou projektu sme napokon získali zdrojové texty, ktoré však neobsahovali komentáre autorov a to spôsobilo predĺženie času ich štúdia. Ešte raz by som chcel vyzdvihnúť tímový prístup všetkých členov, efektívnu komunikáciu a optimistický prístup počas celého procesu vývoja.

Kvadrant Q4 nás so svojimi rizikami našťastie obišiel. Zloženie tímu sa nemenilo. Rovnako nás pri riešení neovplyvnili žiadne politické zmeny. Počas projektu zákazníka zastupovali dve osoby, čo však bolo skôr výhodou, pretože sa svojimi požiadavkami vhodne dopĺňali. Niektoré odlišné názory sme prekonzultovali na pravidelných stretnutiach tímu so zákazníkmi.

Celkovo hodnotím projekt September Project Partnership ako úspešný. Z hľadiska analýzy rizík k tomu prispelo najmä takmer úplné eliminovanie rizík druhého kvadrantu. Kvadranty Q1 a Q4 sa nám so svojimi rizikovými faktormi v podstate vyhli a tak bolo hlavnou úlohou vysporiadať sa s rizikami tretieho kvadrantu. Tímovým a zodpovedným prístupom sa nám podarilo potlačiť vzniknuté problémy a doviesť projektu k vytýčenému cieľu.

Metódy analýzy rizík

Po identifikácii rizík (napr. metódami ako zoznam rizík, dekompozícia, analýza rozhodnutí, analýza predpokladov, interview) je potrebné tieto riziká nejakým

spôsobom klasifikovať a analyzovať. Voľba vhodnej metódy analýzy rizík je prvým krokom k ich efektívnemu a úspešnému riadeniu.

RIPRAN

Veľmi účinnou metódou analýzy rizík je metóda RIPRAN (RIsc PRoject ANalysis), vyvinutá v prostredí FSI ČVUT v Brne, ktorej cieľom je poukázať na potencionálne prekážky úspešnosti projektu. Táto metóda definuje procesy pre identifikáciu rizika, jeho kvantifikáciu a návrh opatrení na zníženie rizika. Vplyv jednotlivých identifikovaných rizikových faktorov určuje na základe ich odhadovaných dopadov a predpokladanej frekvencii ich výskytu. Všetky procesy sú navrhnuté podľa zásad riadenia procesu – majú definovaný cieľ, vstup, postup, výstup, kontrolu priebehu a výstupu pre zaistenie kvality.

Rozhodovací strom

Ďalšou vhodnou metódou pre analýzu rizík je metóda rozhodovacieho stromu, ktorá vo všeobecnosti predstavuje proces výberu jednej z viacerých alternatív. Keďže výsledky rozhodovania v tomto prípade možno hodnotiť pomocou viacerých charakteristík, je potrebné riešiť rozhodovacie situácie s vektorovým ohodnotením výsledkov, ktoré sú vždy konfliktné. Pre rozhodovací proces je charakteristická ekonomická závažnosť dôsledkov a časové obmedzenie rozhodnutia. Rozhodovací proces má nasledovné fázy:

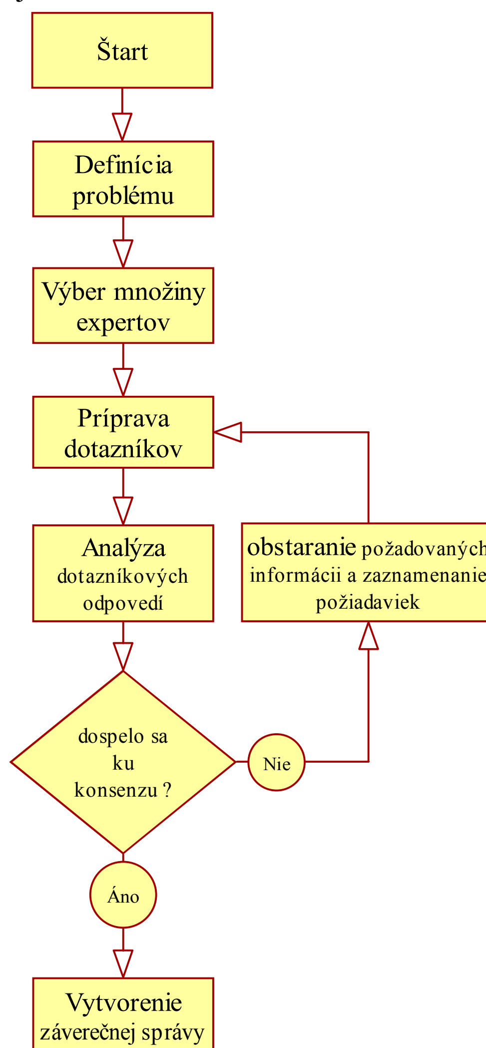
- rozpoznávanie problémov
- konštrukcia rozhodovacieho modelu
- definovanie posudzovacích kritérií
- výber jednej z alternatív

Rozhodovací strom je grafické znázornenie štruktúry problému za účelom voľby postupu riešenia. Celý problém reprezentujeme pojmovým aparátom teórie grafov, kde celý proces zobrazíme ako postupnosť uzlov a hrán orientovaného grafu. Uzly predstavujú okamihy výberu jednej z ponúkaných alternatív (označujeme štvorčekom) alebo okamihy, kedy jednu z alternatív volí okolie (označujeme krúžkami). Takto možno uzly rozdeliť na rozhodovacie a situačné. Hrany potom predstavujú rozhodovanie alebo situačné alternatívy a zvyčajne sú nákladovo alebo výnosovo ohodnotené. Výberom rôznych postupností rozhodovacích a situačných alternatív vznikajú v grafe cesty riešení. Pomocou tejto metódy tak možno stanoviť priority rizík. Viac o rozhodovacích stromoch sa možno dočítať napr. v [3].

DELPHI

Táto metóda bola vyvinutá už v roku 1969 firmou RAND Corporation. Predstavuje proces skupinového rozhodovania o pravdepodobnosti výskytu istých udalostí. V softvérovom inžinierstve nájde uplatnenie najmä pri určovaní expertných odhadov potreby času, nákladov a zdrojov.

Na tejto metóde sa podieľa viacero odborníkov na oblasť, ktorou sa projekt zaoberá. Mali by to byť skúsení experti s množstvom projektov, ktorí určia ďalší postup pri riešení projektu lepšie odhadnú pravdepodobnosť možných situácií. Názory týchto odborníkov sa zozbierajú anonymnou dotazníkovou formu. Štatisticky sa vyhodnotí najviac krát odporúčané riešenie. Stručný postup pri všeobecnej metóde DELPHI najlepšie vystihuje Obr. 2.



Obr. 2. Vývojový diagram metódy DELPHI [4]

Výhodou metódy DELPHI je, že umožní získať viacero expertných pohľadov na budúcnosť projektu, pričom však anonymitou zamedzí ich priamej konfrontácii. Po vyhodnotení dotazníkov sú všetci respondenti oboznámení s celkovými výsledkami a môžu prehodnotiť svoje predchádzajúce uzávery, čím sa zabezpečuje spätná väzba.

Záver

Cieľom tejto eseje bolo zdôrazniť potrebu riadenia jednotlivých rizík a naznačiť jeden z možných prístupov pri ich riadení. Ako bolo uvedené, je potrebné existujúce rizikové faktory kategorizovať a následne určiť strategické postupy čo najefektívnejšieho riadenia. Ako vychádza z analýzy uvedenej v [1], hlavný vplyv na výsledky produktu majú riziká uvedené v prvých troch kvadrantoch, pokiaľ na proces samotný najmä kvadranty Q2 a Q3. Identifikovali sme, že efektívnym riadením rizík kvadrantu Q3 je možné výrazne znížiť vplyv ostatných kvadrantov.

Pre úspech projektu je riadenie rizík určite veľmi dôležité. Samozrejme, že náročnosť tejto činnosti je výrazne ovplyvnená veľkosťou projektu. Mieru, do akej sa jej venovať pre každý projekt musí vždy určiť manažment, a len skúsenosťami je možné dopracovať sa k lepším výsledkom v tejto oblasti.

Použitá literatúra

1. Keil, M., Wallace, L. : *Software project risks and their effect on outcomes. Commun. ACM 47, 4 (Apr 2004) 68 – 73.*
2. Keil, M., Cule, P., Lyytinen, K., Schmidt, R.: *A framework for identifying software project risks. Commun. ACM 41, 11 (Nov 1998) 76 – 83.*
3. Baranová, B. : Rozhodovanie stromy, http://poprad.fei.tuke.sk/~bbarnova/ROZSTROM.HTM, (marec 2005)
4. Joppe, M. : The Delphi Method, http://www.ryerson.ca/~mjoppe/ResearchProcess/841TheDelphiMethod.htm, (marec 2005)

Annotation

Risk factors affecting project's outcomes

In this paper is categorized set of risks factors in software projects. These are divided into four quadrants considering their importance and possibilities of manager's impact. The outcomes of these four sets are analysed in common and on project September Project Partnership. In conclusion are introduced four methods of risk analyze.

Čo zaručuje úspech softvérového projektu

RASTISLAV BERTUŠEK

Slovenská technická univerzita
Fakulta informatiky a informačných technológií
Ilkovičova 3, 842 16 Bratislava

Abstrakt. Skoro každý softvérový projekt je niečím jedinečný. Poznatky získané z jeho vývoja sa ťažko aplikujú na iné softvérové projekty. Napriek tomu sú určité faktory, ktoré pri tvorbe softvérového projektu zohrávajú kladnú alebo naopak zápornú úlohu. Dnešnú dobu charakterizuje stále sa zrýchľujúci proces vývoja softvérových projektov. Preto je priam nevyhnutné, aby sme v tomto procese využívali poznatky z analýzy príčin úspechu a neúspechu softvérových projektov vytvorených v minulosti. Manažment projektu by mal ovládať tieto poznatky a intuitívne sa podľa nich riadiť. Aké sú príčiny, ktoré vedú k úspechu alebo neúspechu softvérového projektu? Následkom akých rozhodnutí manažmentu projektu sú tieto príčiny? Čím lepšie budeme schopní odpovedať na tieto a podobné otázky, tým úspešnejšie budeme vedieť vytvárať softvérové projekty. V eseji sa snažím zachytiť, ale najmä analyzovať tieto príčiny. Vychádzam pritom z prieskumu, ktorý sa zaoberal úspešnosťou softvérových projektov.

Úvod

Úspech softvérového projektu (ďalej iba projektu) je subjektívny pojem. Napríklad pre niekoho je neúspešný projekt taký, ktorý nebol dodaný vo vopred stanovenom čase alebo za daných podmienok, v stanovenej kvalite, v rámci stanoveného rozpočtu, zadefinovaných požiadaviek. Čo pre niekoho znamená úspech alebo aspoň čiastočné naplnenie cieľov projektu, pre iného môže znamenať neúspech. Preto je na každom z nás, kde si určíme pomyseľnú hranicu úspechu a neúspechu.

Takisto softvérový projekt je veľmi široký pojem. Za softvérový projekt je možné považovať skript vytvorený študentom, ktorý sa jednorazovo použije, a na druhú stranu projekt s celosvetovým dosahom (napr. operačné systémy, databázové systémy a iné generické projekty) alebo s niekoľko miliardovým rozpočtom. Preto sa články, zaoberajúce sa manažmentom softvérových projektov, zameriavajú najmä na komerčné projekty s vopred stanovenými a merateľnými podmienkami. Prieskum, ktorý sa zaoberá príčinami úspechu resp. neúspechu projektu, bol publikovaný v článku [5] (ďalej uvádzam iba ako prieskum). Zaoberá sa 122

komerčnými projektmi vytváranými v rámci finančných, bankových a úverových inštitúcií. Tiež nechali ohodnotenie úspešnosti projektu na jednotlivých respondentov prieskumu. V tomto článku sa snažím rozobrať niektoré zaujímavé výsledky prieskumu. V eseji popisujem úspešnosť softvérových projektov vo všeobecnosti. Následne rozoberám niektoré problémové oblasti:

- model vývoja,
- projektový manažment,
- plánovanie,
- požiadavky na projekt,
- centrálna správa,
- manažment rizík a analýza po ukončení projektu.

Úspešnosť softvérových projektov

Pri nekomerčných projektoch je miera úspechu prakticky nestanoviteľná, vzhľadom na osobitné prostredie pri ich tvorbe. Sú to najmä nešpecifikované časové termíny, ciele, požiadavky, testovanie a pod. Preto je vo veľkom množstve takýchto projektov prinajmenšom ťažké určiť mieru ich úspechu/neúspechu a nemá zmysel hľadať jeho príčinu.

Podľa prieskumu iba 55 % vývojových projektov bolo úspešných. Po započítaní projektov určených na údržbu a zlepšovanie už vyvinutých projektov percentuálne číslo stúplo iba na hodnotu 62 %. Z toho vyplýva alarmujúca informácia, že ani silné finančné zázemie a potenciál, ktorý majú inštitúcie zúčastnené v prieskume, nezaručuje úspešné ukončenie projektu. Teda kľúčom k úspechu alebo naopak príčinou neúspechu nie je v prvom rade finančné zabezpečenie projektu.

Model vývoja

Jednou z hlavných príčin úspechu softvérového projektu je jeho model vývoja. Viac ako polovica projektov bola vytváraná vodopádovým modelom (57 %), čo svedčí o konzervatívnosti politiky manažérov a manažmentu inštitúcie. Naopak, modely prototypovania (5 %), inkrementálny (2 %) a špirálový (2 %) mali spolu iba 9 %. Je zjavné, že novšie vývojové modely ako prototypovanie a inkrementálny, ktoré ešte nie sú dostatočne podrobne zadefinované a podporované vývojovými nástrojmi, nemajú u manažérov zelenú. Ich používanie sa presadzuje napríklad v agilnom programovaní alebo v extrémnom programovaní (ang. extreme programming). Pravdou však je, že vodopádový model má veľmi dobrú čitateľnosť, kontrolu kvality, časových ohraničení. Tieto požiadavky na projekt sú pre finančné inštitúcie veľmi dôležité. Myslím si, že ak by sa prieskum konal v iných organizáciách, výsledok percentuálneho rozloženia modelov by mohol byť odlišný. Napriek tomu, že modelom prototypovania a JAD

modelom (joint application design) sa riadilo iba 13 projektov, 9 projektov bolo úspešných, čo je výrazne viac ako priemerný počet úspešných projektov. Zaujímavosťou bolo, že iba 3 projekty z 8, ktoré používali UML pri dokumentovaní požiadaviek, boli úspešné. Príčinou toho podľa môjho názoru bolo nedostatočné zvládnutie novej terminológie. Pri využívaní UML a obzvlášť pri využívaní RUP procesu vývoja, je nevyhnutné si jasne stanoviť jednotlivé kroky vývoja projektu. Jeden z mnohých príkladov podrobného rozpracovania procesu vývoja je v dokumente [4].

Projektový manažment

Model vývoja v organizácii sa stanovuje najmä projektovým manažmentom. Ďalším výsledkom prieskumu bolo zistenie, že zmena projektového manažmentu výrazne negatívne ovplyvnila úspech projektu. Preto je veľmi podstatné, aby sa kládol veľký dôraz na výber správneho projektového manažmentu ešte pri jeho formovaní. Pritom by sa mal projektový manažér vyberať najmä vzhľadom na jeho schopnosti riadenia. Jeho znalosti v odbore sú v tomto prípade minoritnou záležitosťou. Je dôležité mať skôr všeobecný náhľad než vynikajúce znalosti v jednej špecifickej oblasti. Manažér musí mať schopnosť komunikácie a musí vedieť vytvoriť priateľskú atmosféru v tíme. Tieto vlastnosti manažéra posúvajú projekt bližšie k úspechu.

Keď sa projekt vytvára podľa dobre stanoveného plánu a jasnou víziou cieľov, každý člen tímu si ľahšie a presnejšie naplánuje svoj individuálny plán splnenia požiadaviek na neho kladených. Projekt musí mať preto čo najpresnejšie a najpodrobnejšie stanovený harmonogram jednotlivých krokov. Z tohto pohľadu je tragické, že väčšina prvotných časových a rozpočtových špecifikácií projektu vyjednáva a schvaľuje vrcholový manažment. Nielenže projektovým manažérom nie je na začiatku projektu umožnené dohodnúť sa o časových rámcoch, obsahu, cieľoch a finančných otázkach projektu, ale čo je horšie, je im aj v priebehu riešenia softvérového projektu bránené tieto otázky otvárať a modifikovať. Napokon sa projektu pridelí projektový manažment, ale ten už má nadiktované rámcové mantinely. Nanešťastie je zrejmé, že sa časové a rozpočtové mantinely musia až príliš často meniť v priebehu projektu. Vrcholový manažment by mal čo najskôr ustanoviť manažment projektu a následne mu prenechať hlavnú úlohu pri prvotnej komunikácii so zákazníkom a pri stanovovaní prvotnej špecifikácie projektu.

Samozrejme aj priebežná komunikácia o zásadných otázkach, ako je dátum odovzdania projektu do testovania, do prevádzky alebo rozpočet projektu, by sa mala vykonávať čo najpriamejšie medzi manažmentom projektu a zákazníkom bez sprostredkovateľa ako napr. vrcholový manažment. Vrcholový manažment by do tohto procesu mal zasahovať v čo najmenšej nevyhnutnej miere. S tým úzko a neoddeliteľné súvisí zhromažďovanie a spracovanie požiadaviek na projekt. Ak manažment projektu nemá už od začiatku projektu priamy kontakt so zákazníkom, nie je schopný vytvoriť prvotný katalóg požiadaviek a potom nemá dostatočné podklady pre stanovenie rozpočtu a časového harmonogramu. Z toho logicky vyplýva, že je nevyhnutné mať pri plánovacích rozhodnutiach relevantné informácie a požiadavky zákazníka.

Pracovníci v tíme nebudú môcť plniť dobre stanovený časový harmonogram, ak pracujú v stresovom prostredí. Nevhodné vedenie a nedostatočná komunikácia zo strany manažmentu projektu má za následok postupný nezáujem o projekt alebo naopak nadmernú psychickú záťaž pracovníka. Pre dosiahnutie dobrých výkonov by mali pracovníci pociťovať príjemný prístup zo strany manažmentu. Musia byť priebežne povzbudzovaní či už jednoduchou pochvalou alebo finančnou odmenou. Ak pracovníci pociťujú priebežnú oporu, sú pozitívne naladení, dokážu lepšie komunikovať s okolím, pracovať tvorivejšie a tým pádom aj efektívnejšie a rýchlejšie.

Plánovanie

Projektový manažment by tiež mal byť jednou zo základných zložiek projektového plánovania. Jednou z hlavných príčin zlého plánovania je optimistické stanovenie nákladov a časového harmonogramu. Prieskum [5] potvrdil nutnosť mať jasnú víziu o budúcom smerovaní projektu ako najlepší predpoklad úspechu zo strany finančného manažmentu. Tento problém je už dlhšiu dobu známy, ale ťažko riešiteľný. Nie je totiž jednoduché vyjednávať, ak nemáme dostatočnú špecifikáciu projektu, a preto sa snažíme započítať riziká a stanoviť náklady a časový plán s rezervou. Zákazník sa zasa naopak snaží minimalizovať časový harmonogram a cenu, za ktorú sa daný projekt má realizovať.

Vo viac ako dvoch tretinách projektov vrcholový manažment, zákazník alebo používateľ stanovovali náklady a časový harmonogram projektu, čo samozrejme spôsobilo, že z toho viac ako dve tretiny boli zle stanovené náklady a harmonogram. Ďalším prekvapujúcim zistením prieskumu bolo tvrdenie, že nedostatočne stanovené požiadavky na začiatku projektu nespôsobili vo výraznej miere neúspech projektu. Podľa môjho názoru, ale táto zdanlivá nezávislosť bola spôsobená iba dodatočnou komunikáciou a došpecifikovaním, resp. korigovaním prvotnej špecifikácie projektu manažmentom projektu. Teda v konečnom dôsledku sa potvrdzuje tvrdenie, že manažment projektu musí byť schopný komunikácie na vysokej odbornej, rečníckej a psychologickej úrovni.

Požiadavky na projekt

Jedným zo základných podkladov pre projektové plánovanie sú požiadavky na projekt. Pri veľkých projektoch sa prejavuje efekt nedocenenia prvotnej špecifikácie požiadaviek. Väčšina z viac ako polovice projektov, ktorým sa výrazne menili požiadavky v priebehu projektu boli veľké (v kontexte prieskumu) projekty. Spôsobil to fakt, že veľké projekty sú omnoho viac náchylné na nedostatočnú špecifikáciu požiadaviek. Zmena čo len jednej požiadavky môže vyvolať zmenu obrovského množstva súvisiacich požiadaviek a znehodnotiť množstvo už vykonanej práce. Pri veľkých projektoch je taktiež nutné rátať s rôznymi aspektmi, ktorých dôležitosť sa bohužiaľ zanedbáva. Ich dôležitosť ale narastá s veľkosťou projektu. Zjednodušene povedané, požiadavky a manažment rizík, ktorý sa vykonáva pri malých

projektoch, nie je zďaleka postačujúci a priamo aplikovateľný na veľké projekty. Tento prístup je obdobný aj v iných oblastiach manažmentu ako napr. manažment bezpečnosti, manažment riadenia (viď. [1]). Zároveň ale treba dbať pred stanovovaním požiadaviek, aby sa pochopil problém do čo najväčšej hĺbky.

Tab. 1 zobrazuje niektoré role v softvérovom projekte podľa článku [2].

Rola	Motivácia	Expertízna oblasť
Zákazník	Zmena s maximálnym úžitkom	Firemné a informačné systémové stratégie, trendy v priemysle
Používateľ	Zmena s minimálnym narušením	Firemný proces, operačné procedúry
Projektový manažér	Úspešné ukončenie projektu s danými zdrojmi	Projektový manažment, softvérový vývoj a proces dodania
Analytik	Špecifikácia požiadaviek načas a s daným rozpočtom	Metódy a prostriedky inžinierstva požiadaviek
Vývojár	Vytvorenie technicky vynikajúceho systému, použitie najnovších technológií	Najnovšie technológie, návrhové metódy, programovacie prostredia a jazyky
Sledovanie kvality	Vyhovenie procesovým a produktovým štandardom	Softvérový proces, metódy, štandardy

Tab. 1. Motivácia a expertízna oblasť niektorých rolí v projekte.

Z tabuľky vyplýva, že práve analytik má na starosti čo najlepšiu identifikáciu požiadaviek. Analytik používa metódy a nástroje inžinierstva požiadaviek. Na jeho pleciach preto stojí jedna z najdôležitejších a najťažších úloh v softvérovom projekte. Od jeho úsudku identifikovania správnych a vhodných požiadaviek vzhľadom na poskytnuté financie závisí osud celého projektu. Úspešní analytici poznajú do hĺbky sledovanú oblasť. Tento cieľ dosahujú dvoma cestami. Prvá cesta je vlastná skúsenosť v danej problémovej oblasti. Druhá v praxi omnoho viac používaná metóda je komunikácia s expertom v danej problémovej oblasti, teda najmä s používateľom, zákazníkom a/alebo zadávateľom softvérového projektu. Ďalej analytici musia mať prepracovaný systém procesu inžinierstva požiadaviek, aby boli schopní efektívne komunikovať s expertom v problémovej oblasti a získané poznatky vhodným a prehľadným spôsobom zaznamenávať. Podľa článku [2] úspešné projekty vynaložili na inžinierstvo požiadaviek až 28 % svojich zdrojov. S alokáciou zdrojov úzko súvisí priorita jednotlivých požiadaviek. Musíme odstupňovať jednotlivé požiadavky podľa ich dôležitosti, aby sme na relatívne nepodstatnú požiadavku zo strany zákazníka zbytočne neplytvali zdrojmi. Toto všetko by sa nedalo naplniť, ak by analytici nedocenili zásadnú vec. Analytici musia často a úzko komunikovať so zákazníkom.

Centrálna správa

Nedocenenie požiadaviek je podľa prieskumu vážny problém pri vývoji informačného systému. Iba 60 percent projektov používalo centrálnu správu požiadaviek, chýb a nedostatkov. Z vlastnej skúsenosti viem, že správa požiadaviek, chýb a nedostatkov spôsobuje vážne komplikácie pri vývoji systému. Centrálna správa umožňuje jednoduchšie a najmä prehľadnejšie zadeľovanie úloh. Manažment projektu, ale aj členovia tímu rýchlo vkladajú do centrálnej správy požiadavky na novú funkcionalitu, opravu chýb, zmenu atď. Na druhej strane nielen člen tímu má jasný prehľad o jeho úlohe, o ďalšej práci, ktorú má vykonať. Prehľad o dosiahnutých výsledkoch a priebehu vytvárania projektu má taktiež manažment projektu a vrcholový manažment. Pracovník je poloautomatickým spôsobom upozorňovaný resp. oboznamovaný, ako má ďalej postupovať na projekte v rámci tímu a nemusí po každej splnenej úlohe žiadať o novú. Zároveň v takejto centrálnej správe je možné konzultovať jednotlivé požiadavky. Po uzavretí úlohy sa do centrálnej správy zapíše, ako bola uzavretá a nakoľko bola splnená. Centrálny manažment požiadaviek má aj množstvo ďalších výhod, o ktorých sa je možné dočítať napr. na stránke [3].

Manažment rizík a analýza po ukončení projektu

Úspešnosť softvérového projektu tiež závisí od manažmentu rizík a vykonaní analýzy po ukončení projektu. Manažment rizík je najviac zanedbávanou oblasťou manažmentu v softvérovom projekte. Väčšina projektov, ktoré nemali manažment rizík bolo neúspešných. Manažment rizík v priebehu projektu bol pritom podľa prieskumu tiež identifikovaný ako výrazný ukazovateľ úspechu alebo neúspechu projektu.

Iba jedna tretina softvérových projektov vykonala analýzu po ukončení projektu (angl. postmortem review). Podľa môjho názoru tento problém úzko súvisí s vedením podnikovej informačnej databázy otázok a riešení. Takáto podniková databáza uľahčuje prácu zamestnancom. Nemusia pri riešení každodenných otázok znovu „vymýšľať koleso“, ale ak sa už nachádza v takejto centrálnej databáze riešenie, použijú ho. Nevyhnutnou podmienkou využívania centrálnej databázy je jej neustála zmena a dopĺňanie v priebehu riešenia projektov a pri analýze po ukončení projektov. Tým sa tiež docieli lepšie zdieľanie vedomostí jednotlivých zamestnancov v rámci spoločnosti.

Záver

Zaručí nám niečo alebo niekto úspech projektu? Nie, nezaručí. Neviem o žiadnom softvérovom projekte, ktorý by poistila poisťovňa proti rizikám. Nevezme na seba riziko neúspechu. Je to pre ňu veľký hazard, lebo ako prieskum ukázal, iba niečo viac ako polovica projektov je úspešných. Tiež neexistuje jednotlivec, ktorý dokáže zaručiť úspech, ak s ním v tíme spolupracujú kolegovia, ktorí nie sú schopní splniť požiadavky na nich kladené. Dalo by sa teda povedať, že úspech je súhra okolností, ktorým je nutné zo všetkých síl napomáhať. Lebo nedostaví sa automaticky, ani veľkým rozpočtom, ani silnou vôľou. Preto sa musíme snažiť čo najlepšie pochopiť a spoznať príčiny úspechu a neúspechu softvérových projektov.

Použitá literatúra

1. Bieliková M.: *Softvérové inžinierstvo - Princípy a manažment*, Vydavateľstvo STU, Vazovova 5, Bratislava, 109-173.
2. Hofmann H. F., Lehner F.: *Requiremenst Engineering as a Success Factor in Software Projects*, IEEE Software, July/Aufust 2001, 58-66.
3. IBM Rational RequisitePro: *Introduction to Requirements Management*, http://www.pts.com/wp2297.cfm, 24. 3. 2005.
4. Polášek I.: *Objektovo orientovaný prístup a jeho vlastnosti*, http://www.gratex.com /Download/OOANS01Postupnost.pdf, 24. 3. 2005.
5. Verner J. M., Evanco W. M.: *In-House Software Development: What Project Management Practices Lead to Success?*, IEEE Software, Vol. 22, No. 1 (January/February 2005), 86-93.

Annotation

What guarantee success of software project

Almost every software project is special with in usage, development methodology, programming language, and so on. Therefore is hard applying knowledge from one project to another. Despite this, we should try find some useful factories that lead to success or failure of software project. Project management would be able to apply piece of knowledge from analyzing software project from past. What cause success or failure of software projects? What bad do project management? When we will able to better resolve some of causes of failure, then more projects will be success. I try to answer in my essay to these questions and reasons describe to details.

Zlepšenie produktivity práce softvérového tímu

MARTIN HINKA

Slovenská technická univerzita
Fakulta informatiky a informačných technológií
Ilkovičova 3, 842 16 Bratislava

Abstrakt. V posledných niekoľkých desiatkach rokov došlo k rozsiahlemu rozvoju informačných technológií. Zatiaľ čo výkonnosť hardvéru a kapacita sietí urobila pôsobivo dlhé kroky pomocou automatizácie výroby a technologickou inováciou, vývoj softvéru sa nezlepšil na rádovo rovnakej úrovni. Následkom toho softvérové komponenty informačných systémov chronicky zapríčiňujú meškanie projektov, prekročenie nákladov a nespokojnosť zákazníkov.
Je všeobecne uznávané, že zlepšenie produktivity vývoja softvéru požaduje rovnomerný prístup k trojici pilierov softvérového manažmentu: technológia, ľudia, proces.
Existuje niekoľko prístupov koordinácie problémov pri vývoji softvérových systémov. Opíšeme tri súčasné prístupy, ktoré sa používajú: veľký tresk, častá integrácia a periodická synchronizácia, a závadami-poháňaný prístup. Prístupy sa líšia v rozdielnom načasovaní a intenzite koordinácie. Taktiež sa budeme venovať faktorom, ktoré ovplyvňujú intenzitu koordinácie. Tieto sme zoradili do štyroch skupín: projekt, tím, systém, technológia.
Ďalej si opíšeme efekt učenia sa tímu a povieme si niečo o fenoméne mýtického človeko-mesiaca, softvérových charakteristikách systému, technológiách a nástrojoch.
Na záver si spomenieme niečo o ekonomike a manažmente počas vývoji softvérových systémov.

Úvod

Ako môžeme zlepšiť produktivitu práce ľudí a tímov? Pozrieme sa do minulosti a poučíme sa z chýb, znalostí vedomostí, ktoré sme tam získali. A tiež tak, že pozrieme do budúcnosti a budeme sa snažiť pomocou týchto skúseností vyhnúť chybám, ktoré by sme mohli urobiť [3].

Pokiaľ sa všetci budú snažiť pracovať tak, aby sme si mohli v budúcnosti povedať, že pracovali najlepšie ako vedeli a najefektívnejšie ako sa dalo bude zlepšenie produktivity práce prirodzene zabezpečené. Zatiaľ máme pred sebou ešte

dlhú cestu, pretože neustále je čo zlepšovať a dokonalá práca neexistuje. Môže byť dokonalá, resp. optimálna, len vzhľadom na niekoľko málo požiadaviek. Vzhľadom na všetky požiadavky nebude vždy dokonalá, pretože to vyplýva priamo z protichodnosti niektorých požiadaviek.

Informačné technológie a vývoj softvéru

Počas niekoľkých posledných desaťročí nastal prudký a rozsiahly rozvoj informačných technológií (IT). Tento rozvoj pomohol organizáciám dosiahnuť mnoho pracovných aj strategických úspechov.

Nielen organizácie, ale aj bežní spotrebitelia majú úžitok z IT, pretože redukuje faktory trhu spôsobené geografickou separáciou, cenovú neprehľadnosť a oneskorenie informácií.

Ľudia si uvedomili že informácie majú dnes čím ďalej vyššiu cenu, a preto sa snažia ich mať čím viac v čo najkratšom čase. Oneskorený príchod informácií už môže byť zbytočný, pretože informácie už nemusia byť aktuálne

Z toho vyplývajúce zvýšenie dopytu po IT produktoch a službách tvorí nové výzvy pre dodávanie IT riešení umocnených vývojom hardvéru, softvéru a sieťových komponentov a to pri neustále sa znižujúcich časoch na vývoj.

Zatiaľ čo rýchlosť hardvéru a kapacita sietí urobila pôsobivo dlhé kroky pomocou automatizácie výroby a technologickou inováciou, vývoj softvéru sa nezlepšil na rádovo rovnakej úrovni.

Následkom toho softvérové komponenty informačných systémov chronicky zapríčiňujú meškanie projektov, prekročenie nákladov a nespokojnosť zákazníkov.

Je tu niekoľko faktorov, ktoré spôsobujú obtiažny vývoj softvéru, zvlášť zložitosť softvéru, ktorá požaduje zásah človeka počas jeho tvorby. Následkom toho nie je možné realizovať plne automatizovaný vývoj softvéru. Napríklad použitím automatického generovania kódu.

Okrem toho inovačná povaha softvéru spôsobuje jeho ťažké podvolenie sa skúsenostiam a vedomostiam tímu počas projektov.

Nakoniec - nehmatateľnosť softvéru - komplikuje merania a kvantitatívne analýzy, ktoré sú nutné pre neustále sa zlepšujúcu produktivitu.

Piliere softvérového manažmentu

Je všeobecne uznávané, že zlepšenie produktivity vývoja softvéru požaduje rovnomerný prístup k trojici pilierov softvérového manažmentu: technológia, ľudia, proces [1].

Technológia

Zjemneniu stavebných technológií softvéru bolo venované veľké úsilie s významnými výsledkami. Napríklad, sofistikované kompilátory a skriptovacie technológie zvýšili rýchlosť programovania. Robustné pomocné nástroje dovolili jednoduché

vyhľadávanie a odstraňovanie chýb, ladenie systému, jednoduché nastavenie a konfiguráciu.

Komunikačné aplikácie a vytváranie sietí umožnili udržať projekt a systémové informácie prehľadné a rozčlenené, čo je pri obrovských množstvách informácií jeden z veľmi dôležitých faktov. Schopnosť vedieť udržať prehľad a rozdelenie medzi kvantami informácií tak, aby sa v nich vedelo orientovať veľké množstvo ľudí je vzácna, ale dá sa nadobudnúť praxou. Používanie dobrých nástrojov tomu prirodzenou cestou napomáha. Je dôležité, aby sa informácie dali členiť takmer prirodzene zo svojej podstaty, pretože umelé, násilné členenie vyvoláva pocit dezorientácie a vnáša neprehľadnosť.

Ľudia

Ľudia sú dnes nutnou a neodmysliteľnou súčasťou pri získavaní informácií a pri vývoji projektov. Kvôli človeku - smerujúcej povahe vývoja softvéru, prínos z technologických zlepšení nemôže byť plne realizovaný bez spôsobilých zamestnancov. Základom je teda získať, alebo si vychovať spoľahlivých a schopných zamestnancov.

Avšak, investovanie do ľudských zdrojov vyžaduje dlhodobé plánovanie a záväzok no nevytvára okamžitú odmenu. Existuje však CMU / SEI People Capability Maturity (P-CMM) rámcový systém, ktorý poskytol odporúčania na to, ako sa majú uskutočňovať organizačné zmeny tak, aby umožňovali lepší manažment a vývoj zamestnancov.

Proces

Tretím spôsobom ako dosiahnuť stúpanie produktivity je zjemnenie a zušľachtenie samotného vývojového procesu. Prínosy z tohto spôsobu zlepšenia procesu nie sú limitované iba obmedzením sa na zrýchľujúcu sa prácu na vývoji, ale taktiež znižujú úsilie na veľmi náročné a drahé opravné aktivity. Bez správneho vývojového procesu, projektový tím môže operovať iba chaotickým spôsobom, ktorý má následok nízku produktivitu a slabú kvalitu systému.

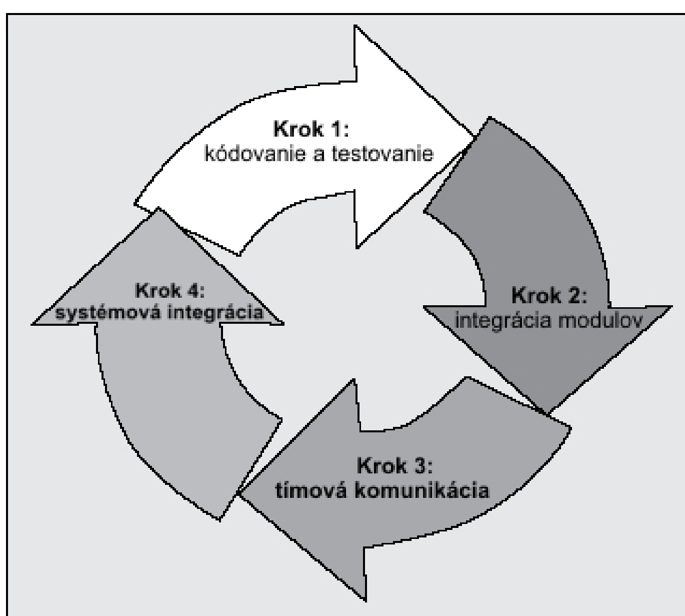
Procesné modely a politiky

Literatúra z minulých rokov ako napríklad *Brooksov inkrementálny vývoj* a *Boehmov špirálový model* predpokladá, že vývoj systému by mal byť radšej skôr akt pozvoľného zlepšovania ako násilné spájanie softvérových súčiastok. Brooks objavil že inkrementálny vývoj, v ktorom sa prelína práca na vývoji s obdobím testovania a hľadania chýb, vedie k jednoduchému spätnému sledovaniu a prirodzenému prototypovaniu.

Rámcové systémy pre inkrementálny vývoj, aj pokiaľ sú kvalitné, často neponúkajú presný postup ako inkrementálny vývoj môže byť optimálne implementovaný. Zvlášť v prípadoch, keď zoberieme do úvahy široké spektrum systémov, zamestnancov a technických faktorov.

Prikročíme teda k množstvu kvantitatívnych procesných modelov a politík na zlepšenie organizácie inkrementálneho vývoja. Hlavná ťarcha spočíva na fáze *konštrukcie systému*, ktorá je stupňom projektu v ktorom sa uskutočňuje kódovanie, čiže implementácia systému. Špecifické činnosti, ktoré sa vyskytujú v priebehu konštrukcie systému sú *kódovanie a testovanie*, *integrácia modulov* (resp. integrácia modulov so zvyškom systému), *tímová komunikácia* (napríklad príklady, príručky, náhľady, stretnutia k projektom atď.) a *systémová integrácia* (systémové oživovanie).

Tieto činnosti sú zobrazené na Obr. 3 a sú sformované do *konštrukčného cyklu.* Na konci každého cyklu sa hotová časť systému (pozostávajúca z kolekcie stabilných softvérových modulov) stane základom, na ktorom môže byť založený ďalší vývoj resp. ďalšia činnosť. Typicky je potrebných niekoľko konštrukčných cyklov, kým je implementovaný a otestovaný celý systém [1].



Obr. 3. Činnosti počas konštrukčného cyklu.

S výnimkou kroku 1, činnosti zobrazené na Obr. 3 sú vzájomne koordinované. Väčšina konštrukcií reálne používaných systémov na svete vyžaduje koordináciu medzi projektovými účastníkmi, akými sú vývojári, testeri, návrhári, a systémový používatelia.

V tomto kontexte vývoja softvéru, Kraut a Streeter popísali koordináciu, zamerajúc sa na činnosti, ktoré umožňujú prácu rozdielnych ľudí na projekte majúcom presnú definíciu a presnú špecifikáciu, čiže presne vieme čo sa ide vytvárať. Delenie a vzájomné poskytovanie si informácií a súlad úsilia týchto ľudí medzi sebou, čiže koordinácia ľudí, informácií a procesu. Teda koordinácia vývoja softvérového systému a problémov s tým spojených.

Koordinácia problémov a jednotlivé prístupy

V tejto časti si vysvetlíme rôzne prístupy koordinácie a rozhodnutí v skutočných softvérových organizáciách. Špecificky si prediskutujeme otázku: *"Kedy je najlepší čas na koordináciu?"*

Správne zodpovedať na túto otázku je veľmi dôležité, pretože oneskorenie načasovania koordinácie za bod určitý bod, alebo príliš malá koordinácia, vedie k drahému prepracovaniu. Súčasne, predčasná, alebo prehnaná koordinácia, môže byť kontraproduktívna, pretože môže narušiť prácu na vývoji.

Opíšeme tri prístupy, ktoré sa v súčasnosti používajú: veľký tresk (Big-Bang), častá integrácia a periodická synchronizácia (frequent integration and periodic synchronization), a závadami-poháňaný (fault-driven) prístup. Líšia sa rozdielnym načasovaním spustením koordinácie [1].

Big-Bang

Big-Bang je prístup, kde celá koordinácia nastáva na konci projektu. V tomto prístupe sú kroky 2, 3, a 4 z Obr. 3 pozdržané až pokiaľ nie je kompletný krok 1. Tento prístup nasleduje *vodopádový model* a tak samo o sebe nejde o inkrementálnu vývojovú techniku.

Od tohto času koordinácia nie je aktívne organizovaná. tento prístup má mimoriadne nízke nároky na organizáciu projektu a preto je veľmi vhodný pre malé a schopné tímy pracujúce na dobre definovanom projekte. Pri tomto je schopnosť učenia sa tímu nižšia ako pri inkrementálnom modely [2].

Hlavným nedostatkom Bing-Bang koordinácie je rozšíriteľnosť. Softvérové súčiastky v systéme sa môžu vzájomne ovplyvňovať. Ak nie je tento nedostatok odstránený včas, tak následne sa stáva ťažko opraviteľný a zapríčiňuje ďalšie nedostatky, ktoré sa môžu vyskytnúť neskôr v projekte. Takéto vedľajšie efekty robia Bing-Bang prístup veľmi drahým na použitie pre väčšie projekty [1].

Častá integrácia a periodická synchronizácia

Softvérové organizácie teraz uskutočňujú integráciu modulov (tak ako v kroku 2 na Obr. 3) oveľa častejšie - často denne. Je to dobre dokumentovaný a publikovaný prístup adoptovaný firmou Microsoft. Ide o tzv. "*Daily Build and Smoke Test*" a bol použitý pri vývoji veľkého množstva projektov v mnohých firmách [1]. Mimo častej integrácie modulov, pravidelnej tímovej komunikácie a systémovej integrácie ktoré sú vykonávané zabezpečuje kvalitu produktu už jeho samotná konštrukcia.

Zatiaľ čo častejšia koordinácia na modulovej a systémovej úrovni pomáha zmierňovať efekt degradácie, dôležitá otázka zostáva nezodpovedaná: *"Ako dlho má trvať vývoj v každom cykle?"*

Všimli sme si, že časovanie koordinácie často prebieha spôsobom *ad hoc* v mnohých organizáciách. V dôsledku toho, by sa však tieto organizácie mali riadiť aspoň základnou koordinačnou politikou, kde koordinácia je intenzívnejšia na začiatku projektu, uvoľňuje sa počas stredu, a stáva sa zase intenzívnou blízko konca projektu.

Takáto koordinačná politika môže byť vysvetlená ako výsledok dvoch faktorov: *tímového učenia sa a systémovej stability* [1].

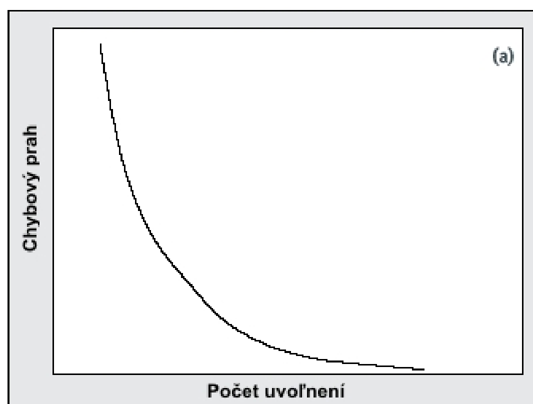
Závadami-poháňá koordinácia

Zatiaľ čo časovo-založené koordinačné politiky majú jasné organizačné prínosy, nejavia sa plne schopnými pokryť dynamiku vyvíjajúceho sa projektu.

Pri závadami-poháňanej koordinačnej politike špecificky platí, že koordinácia je naliehavejšia, keď sa zdá že systém nie je synchronizovaný, inak je vhodné v práci na vývoji pokračovať. V dôsledku toho koordinačné rozhodnutia by mali byť nejako zviazané k aktuálnemu stavu systému [2].

S pokročilým vývojom a nástrojmi riadenia projektu, je teraz možné získať údaje o chybách systému (*system fault data*) a ďalšie príbuzné metriky pomerne v skorých fázach projektu. Použitím aktuálnej početnosti týchto chýb a prísnych metrík, projektoví manažéri môžu plánovať koordináciu vždy, keď priemerné náklady na opravu chýb presahujú očakávané hodnoty resp. začnú stúpať.

Obr. 4 ukazuje krivku chybových prahov, ktorá môže byť odvodená s týmto zdôvodnením. Tvar prahovej krivky závisí na špecifických faktoroch projektu akými sú napríklad, zložitosť systému, veľkosť a skúsenosti tímu, vývojové prostredie, doména resp. oblasť projektu, a čas dosiahnuteľný pri konštrukcií systému.

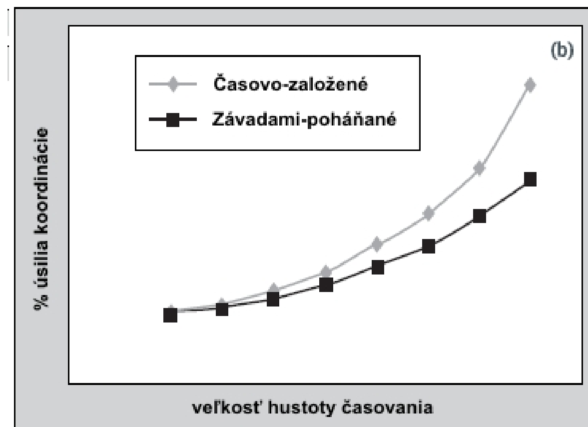


Obr. 4. Závadami-poháňaná koordinácia

(a) krivka chybových prahov (fault treshold curve)

Pri závadami-poháňanej politike, koordinácia nastáva pri uvoľnení modulu, keď počet chýb presiahne prahovú hodnotu stanovenú pre toto uvoľnenie. Zostupný svah krivky zaisťuje, že na začiatku je síce veľké množstvo chýb, ale po uvoľneniach niekoľkých modulov tento počet klesá. Koordinácia nemôže nastať až kým sa neuvoľní niekoľko málo modulov. Naproti tomu, prahy sa znižujú pri veľkom počte uvoľnení, čo je povzbudením pre koordináciu, lebo práca na vývoji má extrémne vysokú kvalitu.

Obr. 5 porovnáva časovo-založenú (*timed-based*) a závadami-poháňanú (*fault-driven*) koordinačnú politiku a ukazuje, že závadami-poháňaná koordinácia je často efektívnejšia. Použitie údajov o chýb systému pre vytvorenie koordinačných rozhodnutí je z čiastočne priaznivé pre projekty bez nahusteného časovania (čo nastáva, keď sa požaduje vysoký pomer práce na dosiahnutý čas) [1].



Obr. 5. Závadami-poháňaná koordinácia

(b) zisk zo závadami-poháňanej koordinácia (benefit of fault-driven coordination)

Faktory ovplyvňujúce koordináciu

Konceptuálne, intenzita koordinácie je pomer úsilia použitého na koordináciu v jednom konštrukčnom cykle. Zoradili sme teda faktory ovplyvňujúce intenzitu koordinácie do štyroch skupín: projekt, tím, systém, technológia [1].

Projekt

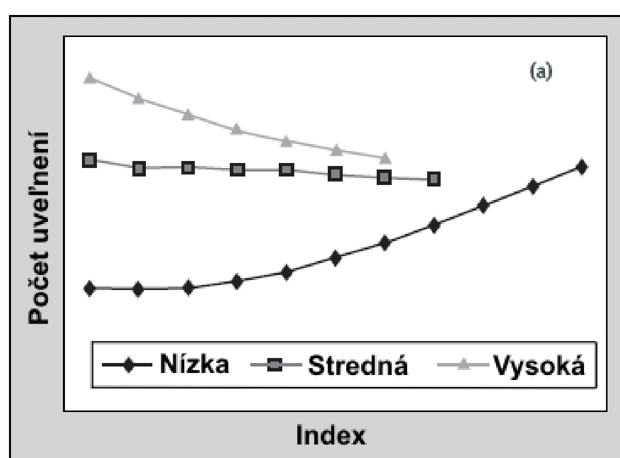
Kritickým faktorom ovplyvňujúcim produktivitu tímu je dovolený čas vývoja systému. Pre danú množinu požiadaviek, môžeme dosiahnuť kratší konštrukčný čas pri väčšom tíme a teda pri drahšej koordinácií medzi členmi tímu. Na druhej strane, zvyšovanie veľkosti tímu zmenšuje produktivitu na jedného člena.

Tento fenomén mýtického človeko-mesiaca (mythical man-month phenomenon) naznačuje, že pridávaním členov do tímu, za určitý bod môžeme skutočne predĺžiť trvanie projektu. Liek na konfrontáciu s veľkým tímom je zaviesť hierarchickú komunikačnú štruktúru a tak sa relatívne malé skupiny vývojárov môžu sústrediť na dobre prepojiteľné systémové komponenty.

Skúsenosti tímu a efekt učenia sa

Krivka na Obr. 6 predpokladá, že projektové tímy, ktoré môžu stabilizovať moduly veľmi rýchlo si môžu dovoliť menej častú koordináciu na projekte už skôr. Naproti tomu s narastajúcou voľnosťou, je potrebná častejšia koordinácia pokiaľ nie je dosiahnutá dostatočná spoľahlivosť a stabilita. Napríklad projekt uskutočnený menej skúseným tímom môže dosiahnuť stabilitu oveľa pomalšie, takže tím musí byť koordinovaný oveľa častejšie.

Indikáciou pomalej stabilizácie je, že hlavné prvky systému (také ako systémové rozhrania a moduly riadiace moduly) pokračujú v podstupovaní zmien až do neskorých fáz projektu.



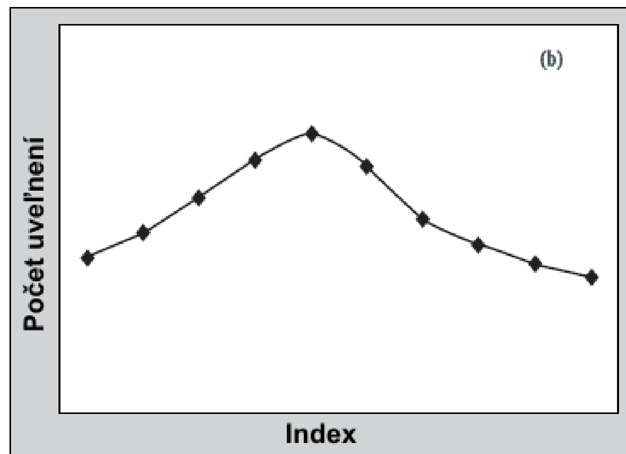
Obr. 6. Zmeny v krivke koordinačnej politiky

(a) efekt systémovej stabilizácie (effect of system stabilization rate)

Mnohé štúdie ukázali až desaťnásobnú priepasť v produktivite medzi nováčikom a zručným vývojárom. Jednou z čŕt prvotriednych vývojárov je ich schopnosť získavať projektovo-špecifické znalosti a vyhnúť sa tak väčšiemu prepracovaniu, dokonca aj v neznámej projektovej doméne resp. v neznámom projektovom prostredí.

Dôkazom tohto *"procesu učenia"* je, že vývojový tím stabilizuje novo vyvíjané moduly oveľa efektívnejšie ako predošlé a dochádza tak k pokroku schopností tímu a k rýchlejšiemu a kvalitatívne lepšiemu pokroku v projekte.

Zmeny v krivke koordinačnej politiky na Obr. 7 ukazujú, že pre takýto tím je vhodná relatívne úzka koordinácia na začiatku projektu, čo urýchľuje proces učenia. Tím by sa mal potom sústrediť na programovanie úloh s menšími zákrokmi takmer až do konca projektu, kedy je zase potrebná úzka koordinácia na zmenšenie rizík plánovania. Koordinačná schéma na Obr. 7 je odporúčaná pre skúsený tím pracujúci na projekte v novej doménovej oblasti.



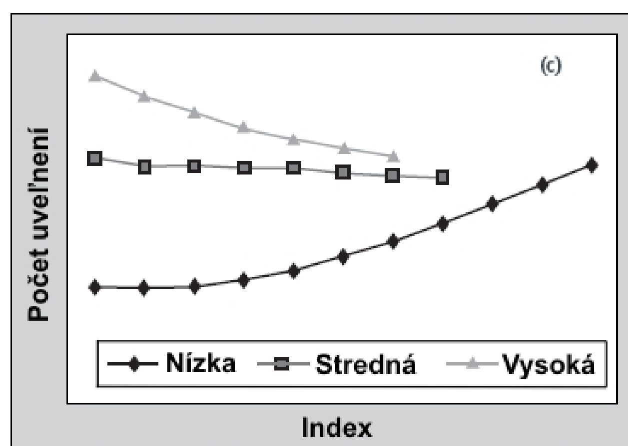
Obr. 7. Zmeny v krivke koordinačnej politiky

(b) efekt procesu učenia tímu(effect of team learning)

Softvérové charakteristiky systému

Čím vyššia je zložitosť systému, tým by mala byť intenzita koordinácie vyššia. Tento nárast však nie je v žiadnom prípade priamoúmerný. V zložitejších systémoch sa nesúlad v jednotlivých moduloch stáva rádovo ťažšie opraviteľný ako pri systémoch s menšou zložitosťou. Intenzívnejšia koordinácia preto pomáha udržať chybovosť pod kontrolou.

Obr. 8 ukazuje optimálnu krivku koordinácie pre rôzne úrovne systémovej zložitosti. Náklady na celkovú koordináciu taktiež narastajú so zložitosťou systému. Jeden z aspektov na Obr. 8 je, že nám poskytuje základ pre navrhovanú ekonomiku. Pokiaľ čelíte obmedzeným prostriedkom a zdrojom alebo neprávnemu načasovaniu projektu, tak projektový manažéri sú často v pokušení preskočiť návrh systému, aby sa viac venovalo produktívnej implementácií. Často by bolo pre nich lepšie investovať čas do prvotriedneho návrhu systému aby predišli náročnému a drahému "haseniu", resp. prerábaniu systému v priebehu konštrukcie.



Obr. 8. Zmeny v krivke koordinačnej politiky

(c) efekt zložitosti sytémov(effect of system complexity)

Technológie a nástroje

Zmeny v prínosoch možno merať tým ako ľahko členovia tímu vedia skĺbiť v rámci svojho pracovného prostredia vývoj a koordináciou. Ďalším dôležitým merítkom je podpora chápania projektu z danej doménovej oblasti. Inak povedané tím má presne vedieť, čo, kde, kedy a prípadne ako bude robiť. Preto podľa Obr. 8 by mala byť koordinácia intenzívnejšia, keď nastavajú efektívnejšie resp. väčšie zmeny projekte.

Napríklad v organizácií so sofistikovanou CASE podporou, je tím vedený ku koordinácií oveľa častejšie. Technologické inovácie ponúkané novými nástrojmi sa častokrát považujú za pomôcky umožňujúce zrýchlenie implementácie a celého vývoja projektu. V prípade správneho použitia, môžu taktiež znížiť nadbytočnú koordináciu.

Organizácie zaoberajúce sa vývojom softvéru sa teraz tešia zo širokého výberu nástrojov a technológií, od ktorých si sľubujú zvýšenie produktivity vývoja. Aj keď v priemysle, ktorý praktizuje tzv. "kreatívnu deštrukciu" sú rýchlosť a kvalita, teda výhody zo získaných technológií prenasledované vyššími očakávaniami používateľov.

Je preto dôležité vytvoriť dobrý a efektívny manažment projektového procesu, aby sme pomocou neho dokázali čo najlepšie využiť zamestnancov a technológie, ktoré máme k dispozícií.

Ekonomika vývoja projektu.

Jednou zo zaujímavých sfér budúceho výskumu je vývoj modelov a teórií rôznorodých softvérových systémov. V predchádzajúcich štúdiách sa na softvérových systém pozerá ako na súbor (viac či menej) identických modulov. Zatiaľ čo tento homogénny prístup

má jasné analytické výhody, je veľa situácií, kde softvérové moduly môžu obsahovať významné rozdiely vo veľkosti, zložitosti a funkcionalite.

Možnosťou je vytvoriť postupnosť, v ktorej sú moduly vyvíjané vzhľadom na obchodné faktory, ako napríklad funkcionalita, finančné plány a náklady na vývoj. Inak povedané, je potrebné vyvíjať ekonomickú základňu pre uvoľnenie softvérových modulov. Je teda dôležité založiť vývoj projektu na aktuálnych a premyslených zdrojoch ekonomických a neekonomických zdrojoch.

Mierkou pre zrelosť softvérových organizácií je či dokážu podávať konzistentnú, resp. rovnakú produktivitu počas dlhšieho obdobia. Veľa štúdií o produktivite sa zameriava na znižovanie priameho implementačného úsilia.

Produktivita tímu je výrazne ovplyvnená tým ako dobre je rozdelené koordinačné úsilie. Pokiaľ nie je koordinácia kontrolovaná, je málo pravdepodobné, že kvalita softvéru bude dosiahnutá rýchlejšie jednoduchým pridaním viacerých ľudí do projektu. Správny proces koordinácie a návrhu je preto kľúčom k lepšej produktivite a vyššej kvalite výsledného produktu.

Mnoho organizácií skôr nazerá na proces vývoja ako na zdržujúci ako na urýchľujúci produktivitu. Je to preto, že proces vývoja je často použitý cez donútenú postupnosť projektových úloh, bez explicitného zamyslenia sa nad koordinačnými aktivitami.

Slepo hľadať produktívne zlepšenia v snahe vyhovieť zákazníkovým očakávaniam je len sen, pokiaľ je len malá šanca ich splniť. Veľa projektových manažérov je neschopných spraviť citlivú analýzu pre štúdium následkov zmenených požiadaviek na projektové náklady, alebo plánovanie. Vlastnosti projektu ako systémová zložitosť, faktor stabilizácie a učenie sa tímu nám poskytujú smernice ako stanoviť dopad zmien projektu v plánovaní projektu [1].

Záver

Tento článok sa pokúša spojiť priepasť medzi analytickým a empirickým pohľadom na manažment projektu, aby poskytol smernice ako najlepšie riadiť softvérový projekt a pomohol tak zlepšiť produktivitu práce v softvérovom tíme.

Opisuje základné piliere softvérového manažmentu ako aj prístupy, ktoré sa dnes používajú pri koordinácií projektov.

Ďalej sa venuje faktorom ovplyvňujúcim koordináciu projektu, efektu učenia sa tímu, fenoménu mýtického človeko-mesiaca, softvérovým charakteristikám systému, technológiám a nástrojom používaným pri koordinácií a vývoji projektu.

Na záver je opísaná ekonomika a manažment projektu počas vývoja softvérových systémov.

Použitá literatúra

1. Robert Chiang, Vijay S. Mookerjee: Improving Software Team Productivity. In Communications of the ACM, Vol. 47, No. 5 (May 2004), 89-93.
2. Liu Xanfeng, Liu Guanguiu: Coordination in software develepment, review of “Improving Software Team Productivity, www presentation, (Februar 2005), 28 pages
3. Walt Scacchi: Uderstanding and improving software productivity, Institute for Software Research, University of California, Irvine USA, www presentation, (Februar 2004), 44 pages

Annotation

Improving software team productivity

Over the last few decades the widespread adoption of information technology (IT) was come. While hardware speed and network capacity have made impressive strides through manufacturing automation and technological innovation, software development has not improved under the same order of magnitude. As a result, the software component of information systems chronically causes project delays, cost overruns, and customer dissatisfaction.

It is widely recognized that improving software development productivity requires a balanced approach toward the three pillars of software management: technology, people, and process.

Many approaches of software development coordination problems exist. We describe three current approaches being used: Big Bang, frequent integration and periodic synchronization, and fault-driven. In each approach, there is different coordination trigger and coordination intensity. We describe the factors that have affect to coordination intensity. These have we clustered into four groups: project, team, system, and technology.

Next we describe the team learning effect and we said something about mythical man-month phenomenon, software characteristics, technologies and tools.

At the end we note something about software system development economics and management.

Systematické výmeny zamestnancov

MAREK FUČILA

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

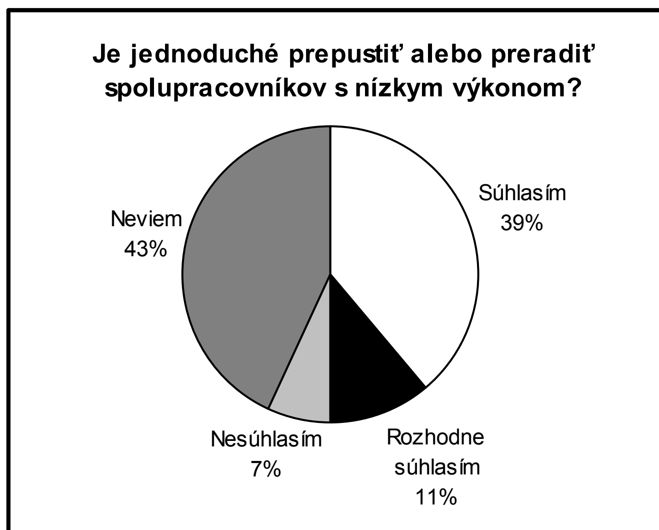
Abstrakt. Ak sa v konkurenčnom prostredí prestáva spoločnosti dariť a chce sa udržať na trhu, musí znižovať náklady. Spravidla dochádza k veľkému prepúšťaniu zamestnancov. Takéto riešenie je však nepopulárne a navyše skúsenosti ukazujú, že zamestnanci sú aj po ozdravnom procese schopní identifikovať slabších spolupracovníkov, ktorí ich brzdia.
Táto esej sa venuje systematickému prístupu k výmene zamestnancov. Takýto prístup uplatňujú s rôznymi obmenami napríklad v spoločnostiach General Electric, Toyota či Microsoft. Ide vlastne o vyraďovanie najslabších zamestnancov na základe hodnotenia ich výkonov. Vychádza sa z výskumov, ktoré ukazujú, že sú až priepastné rozdiely v efektivite práce jednotlivcov, a preto je vhodné sa zbavovať najslabších pracovníkov postupne. Najlepších zamestnancov je možné vhodne motivovať, a tak sa každý rok zvyšuje efektivita práce danej organizácie.
Problémom stratégie je hlavne otázka korektnej klasifikácie zamestnancov. Väčšie softvérové spoločnosti totiž tvoria tímy vedené rôznymi manažérmi s rôznymi kritériami hodnotenia, pričom nie je ľahké dosiahnuť konzistenciu. Niektorí schopní vývojári môžu mať nižšiu produktivitu práce napríklad kvôli zlej organizácii.

Úvod

Každá spoločnosť stojí a padá na svojich zamestnancoch. Úspešnosť riešenia projektov závisí od efektivity práce tímov ako aj jednotlivých zamestnancov. Ak je efektivita práce príliš nízka, spoločnosť prestáva konkurovať ostatným spoločnostiam na trhu, pretože vyvíja softvér s vyššími nákladmi.

Podstatnú časť nákladov tvoria mzdy zamestnancov. Aby bolo možné znížiť náklady na vývoj a obstáť v konkurenčnom boji, musí spoločnosť v kríze prepúšťať ľudí. Spravidla dochádza k veľkému prepúšťaniu, ktoré nebolo dlhodobo plánované, a tak manažéri obvykle nemajú k dispozícii dostatok informácií, aby vedeli správne vybrať najschopnejších zamestnancov, ktorí ostanú. Potvrdzujú to prieskumy robené formami ankiet a stretnutí s vývojármi, ktorí by mali po masívnom prepúšťaní byť práve tí najlepší.

Ako Middleton a kolektív uvádzajú vo svojom článku [1], zamestnanci sú aj po takomto ozdravnom procese schopní identifikovať svojich slabších spolupracovníkov. Po dlhšej dobe opäť naberajú pocit, že medzi nimi zbytočne zotrvávajú slabší kolegovia. Autori uvádzajú odpoveď zamestnancov na anketovú otázku *„Je jednoduché prepustiť alebo preradiť spolupracovníkov s nízkym výkonom?“* v grafe uvedenom na Obr. 1.



Obr. 1. Iba polovica vývojárov si myslí, že je jednoduché zbaviť sa pracovíkov s nízkym výkonom

Celoplošné prepúšťanie teda nezabráni opakovanému trvalému poklesu efektivity práce celku, a preto je len otázkou času, kedy nastane potreba ďalšej nepopulárnej reorganizácie spoločnosti. Manažéri, ktorí sa snažia predísť tejto perspektíve využívajú niektorú z foriem systematického prístupu k výmene zamestnancov.

Výskumy

Opodstatnenosť snahy o neustále zvyšovanie efektivity práce inžinierov pri vývoji softvéru dokazujú výskumy. Podľa autorov článku [1], Barry Boehm a Richard Turner zistili, že dobrí ľudia a zohrané tímy tromfnú ostatné faktory. Je dôležité mať kvalitných kvalifikovaných inžinierov, ktorí dokážu spolupracovať so svojimi kolegami. Rovnako je nevyhnutné ich prácu efektívne organizovať a tým celý proces vývoja softvéru zefektívniť. Ak sa nedarí spoluprácu koordinovať, alebo ak sa v tímoch vyskytujú ľudia, ktorí výrazne zaostávajú za kolektívom, a tak brzdia ostatných, degraduje to výkon celku. Barry Boehm a Richard Turner zaznamenali až priepastné rozdiely výkonov jednotlivcov. V rámci jednej softvérovej spoločnosti

môže pomer výkonov rôznych vývojárov podľa ich zistení dosahovať až 26:1, prípadne 10:1 v závislosti od metriky.

Tieto výsledky potvrdzujú aj Tom DeMarco a Tim Lister, ktorí podľa Middletonovho článku [1] taktiež uverejnili údaje ukazujúce rozptyl schopností skúsených softvérových profesionálov približne 10:1. Navyše, zistili aj rozdiel 10:1 v produktivitách rôznych softvérových firiem. To poukazuje na opodstatnenosť úsilia o zlepšovanie procesov vývoja softvéru, s čím súvisí dobrá organizácia práce a premyslené štruktúrovanie tímov. Autori štúdie zistili, že vysoko výkonní zamestnanci robili v organizáciách s presnejším vymedzením úloh, s väčším súkromím a pokojom, pričom trpeli menším počtom prerušení. Každé prerušenie práce totiž znižuje výkon.

Ak sú závery, ktoré nám výskum ponúka správne, potom je logické podporiť systematické výmeny slabých pracovníkov spolu s reorganizáciou tímov v prípade zníženia efektivity práce.

Možné prístupy zvyšovania produktivity

Prístup systematického zvyšovania produktivity práce zamestnancov uplatňujú s rôznymi obmenami napríklad v spoločnostiach General Electric, Microsoft či Toyota.

Jednou z možností, ako udržať vo firme rast produktivity práce, je prístup podľa Jacka Welcha, ktorý zaviedol v General Electric. Tajomstvom úspechu je hodnotenie zamestnancov. Manažéri, ktorí vedú vývojový tím, majú záujem na tom, aby bol celý ich tím dobre hodnotený, pretože je to mierou ich vlastných schopností. V praxi sa preto často stáva, že nadhodnocujú kvalitu práce svojich podriadených, a zároveň zahladzujú nedostatky jednotlivcov. Navonok sa potom zdá, že tím pracuje efektívne, a pritom existujú skryté rezervy. Časom sa problém zhoršuje. Tomu Welch zabránil tak, že uložil manažérom povinnosť hodnotiť svojich podriadených podľa gaussovej krivky. Hodnotenie tak zodpovedá normálnemu rozloženiu schopností zamestnancov. Identifikujú sa tak najslabší členovia tímu, ktorí brzdia prácu ostatných a tým aj celej spoločnosti. Welch stanovil hranicu na najslabších 10 percent. Títo ľudia dostanú šancu zlepšiť sa. K tomu im môžu pomôcť napríklad rôzne školenia. Zamestnanci, ktorí nedokážu obhájiť svoje zlepšenie, musia firmu opustiť. Naopak najlepší pracovníci môžu byť za svoje výkony odmenení. Takýmto spôsobom sa potom darí manažérom posunúť gaussovu krivku hodnotenia smerom k vyššej efektivite a produktivite práce.

V spoločnosti Microsoft sú vývojári softvéru podobne ako v General Electric nútení zlepšovať svoje výkony, pretože pravidelne musí najslabších 5 percent zamestnancov odísť. Zaujímavé je, že takéto množstvo vývojárov obvykle odchádza dobrovoľne. Relatívne malé personálne výmeny pravdepodobne súvisia s kvalitným prijímacím procesom. Microsoft sa stará o motiváciu svojich najlepších ľudí, aby nestrácal aj ich. Na rozdiel od General Electric sa teda Microsoft nemusí veľmi snažiť podporovať zlepšovanie výkonu najslabších pracovníkov. Vývojári softvéru obvykle často striedajú zamestnanie, a kvôli tejto fluktuácii sa treba zamerať najmä na to, aby neodchádzali tí najlepší. Kľúčová je teda motivácia. Napríklad nadštandardné ohodnotenie.

Spoločnosť Toyota Motor Manufacturing v USA má na rozdiel od General Electric či Microsoftu stupňovitý proces zvyšovania pracovných výkonov, a to bez pevných kvót. Zjednodušene sa dá hlavná myšlienka tejto stratégie opísať sloganom: „ponechať, premiestniť, odstrániť“.

Pri zvyšovaní produktivity práce jednotlivcov môže napomôcť systém navrhnutým Wattsom Humphreyom [3] nazvaný Osobný softvérový proces (Personal Software Process - PSP), z ktorého bol odvodený Tímový softvérový proces (Team Software Process - TSP) určený pre použitie vo veľkých projektoch, ktorý je možné využiť na úrovni celého tímu.

Proces PSP je založený na nasledujúcich princípoch:

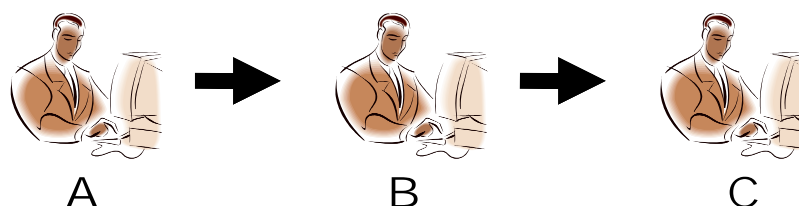
- Každý zamestnanec je iný. Aby mohli pracovať čo najefektívnejšie, musia si stanoviť plány a tie musia byť založené na údajoch, ktoré získali počas práce na predchádzajúcich projektoch.
- Na dôsledné zlepšovanie výkonnosti zamestnancov je nevyhnutné aby používali dobre definované a merateľné procesy.
- Aby sa mohli vyvíjať kvalitné produkty, je potrebné aby zamestnanci cítili osobnú zodpovednosť za kvalitu svojich výrobkov. Zamestnanci sa musia snažiť vytvárať kvalitné produkty.
- Nájdenie a oprava chyby na začiatku procesu je lacnejšia ako v neskorších fázach.
- Je efektívnejšie predchádzať chybám ako ich hľadať a odstraňovať.
- Správny spôsob ako vykonať prácu je taký, ktorý je najrýchlejší a najlacnejší.

Problémy

Využiť proces gaussovej krivky bolo v General Electric každým rokom ťažšie, pretože ostávali len schopní zamestnanci. Tiež nebolo jednoduché použiť ho konzistentne v celej spoločnosti, pretože niektoré odvetvia boli lepšie ako iné. Napriek tomu sa v každoročnom anonymnom zamestnaneckom prieskume zamestnanci tejto spoločnosti stále sťažovali, že očistný proces nebol implementovaný dosť prísne. Tento záver prekvapil samého Welcha, ktorý tvrdo bojoval za to, aby jeho manažéri túto iniciatívu implementovali.

Zamestnanci boli kritickejší k svojim slabším kolegom ako ich manažéri. Možným vysvetlením je výskum dynamiky softvérových projektov Tareka Abdel-Hamida a Stuarta Madnicka, spomínaný v Middletonovej práci [1]. Tento výskum poukazuje pomocou formálneho modelu procesu vývoja softvéru na problémy riešenia ohraničených úloh. S využitím teórie spracovania úloh v rade autori identifikovali slabé miesto, kde dochádza k znižovaniu výkonu tímu.

Príkladom je práca troch vývojárov v tíme. Na obrázku č.2 sú označení ako A, B a C.



Obr. 2. Vývojár B sa môže ocitnúť v pasci

Ak vývojár B pracuje nepretržite, ale dostáva prácu od vývojára A nepravidelne a vývojár C pracuje tiež nevyrovnane, výsledok celého tímu tým bude trpieť. V praxi je nemožné pre vývojára B zlepšiť produktivitu tímu bez toho, aby sa zlepšili ľudia pred a za ním. Autori túto skutočnosť potvrdili simuláciou.

Simulácia ďalej ukázala, že ak je práca dodávaná vývojárovi B príliš variabilne, teda veľké časti práce prichádzajú nepravidelne, produktivita samotného vývojára B klesne. Ak je však také isté množstvo práce dodávané v priebehu rovnakého času s menšími odchýlkami a v menších častiach, potom sa produktivita vývojára B značne zlepší.

Bez zmeny pracovných postupov konkrétneho pracovníka sa tak môže stať, že stúpa alebo klesá jeho produktivita, a to len kvôli vyrovnaným respektíve nerovnomerným dodávkam práce. Ak to nadriadený nepostrehne, mylne tým identifikuje vývojára, ktorého výkon poklesol ako slabý článok reťaze. Zamestnanec v pasci síce môže dostať možnosť zlepšiť sa, to mu však bez zmeny organizácie práce tímu takmer vôbec nepomôže, nakoľko nedokáže ovplyvniť čas nečinnosti, keď nedostal výstupy práce vývojára v rade pred ním.

Záver

Systematické výmeny zamestnancov vyzerajú byť vhodnou prevenciou pred znižovaním produktivity práce spoločnosti. Zvyšovanie výkonu tímov ale nemôže byť postavené len na hodnotení zamestnancov zhora. Samotní vývojári totiž vedia medzi sebou určiť zdroj problémov lepšie ako ich manažéri. Preto by manažéri mali byť schopní na základe podnetov svojich podriadených reorganizovať prácu v tíme, pridelenie úloh, komunikáciu, a tým zvýšiť efektivitu práce jednotlivcov, čím stúpne produktivita tímu. Gaussova krivka hodnotenia totiž odráža reálny stav len pri optimálne zohranom tíme.

Použitá literatúra

1. Middleton, P., Lee, H.W., Irani, S.A.: Why Culling Software Colleagues Is Popular. In IEEE Software, Vol. 21, No. 5 (September/October 2004), 28-32.
2. Madachy, R.J.: Software Process Dynamics - Portions from 4/03 Draft Version, IEEE Computer Society Press (1999)
3. Humphrey, W.S.: The Personal Software Process. CMU/SEI Technical Report 2000-TR-022, November 2000

Annotation

Systematic stuff culling

If the company in the competitive environment goes into a recession and it wants to stay in the marketplace, it has to reduce costs. Commonly a big firing is coming up. But this solution is not popular and the experience shows personnel are able to identify poor colleagues also after this cleaning process.

This essay is concerned about systematic stuff culling. This approach is variedly used in companies like General Electric, Toyota and Microsoft. It is about stuff culling based on ranking their productivity. Studies show us there are huge differences in productivity of stuff members. Therefore it is desirable to cull out weakest workers systematically. It is also worthy to motivate best workers. The productivity of work in the organization is than higher year by year.

The main problem of this strategy is the appropriate classification of stuff. Big software companies composite of teams leaded by different managers with different criteria. It is hard to achieve consistency. Some skilled developers may have lower productivity for bad work organization

Obstarávanie v softvérových IT projektoch

MARTIN NIEJADLIK

Slovenská technická univerzita
Fakulta informatiky a informačných technológií
Ilkovičova 3, 842 16 Bratislava

Abstrakt. Táto práca opisuje outsourcing z pohľadu manažmentu v oblasti IT. Rozoberá problematiku outsourcingu od vzniku až po zánik celého tohto procesu. Takmer vo všetkých odvetviach prevláda požiadavka na znižovanie nákladov. Personálne náklady sú jednými z najvyšších nákladov spoločností a preto je outsourcing v dnešnej dobe stále viac a viac diskutovanou témou. V tejto práci sa venujem predovšetkým problematike určenia biznis procesov, ktoré je potrebné alebo vhodné outsourcovať. Dôraz je však kladený aj na prostredie softvérových spoločností. Ide hlavne o outsourcing procesov súvisiacich s vývojom informačných systémov. Tu sa venujem problematike outsourcingu jednotlivých etáp vývoja softvéru ako aj "zapožičiavaniu" zamestnancov na tieto činnosti od iných firiem. Jednotlivé procesy sú zaradené do dvoch kategórií podľa priority na outsourcing. Tieto skupiny sú podrobne opísané a vysvetlené ich výhody a nevýhody súvisiace s outsourcingom konkrétnych procesov z týchto skupín.

Úvod

Outsourcing je definovaný ako delegácia práce od internej produkcie k nejakej externej entite. V posledných rokoch to znamená elimináciu lokálnych zamestnancov firmy na úkor externých dodávateľov, kde sú nižšie náklady na ľudské zdroje. Toto je dôvod prečo sa v spojení s outsourcingom väčšinou myslí na zahraničnú spoluprácu. Cieľom outsourcingu je teda znižovanie nákladov ako aj možnosť orientovať všetky zdroje firmy na odvetvie, v ktorom podniká a nie na podporné procesy.

Celý proces outsourcingu predstavuje dlhý a namáhavý boj pre obe strany. V tomto procese vystupujú obe strany s pomerne protichodnými očakávaniami. Dodávateľ, ktorý bude zodpovedný za externé vykonávanie procesov chce mať z tejto činnosti čo najväčší zisk pri vydaní čo najmenšieho úsilia. Naopak odberateľ, ktorý si objednáva outsourcing u dodávateľa, chce mať za čo najmenšie náklady čo najviac dodanej práce.

Proces outsourcingu sa dá rozdeliť do troch štádií. Prvé štádium predstavuje výber vhodného dodávateľa, dohodnutie čo a ako sa bude outsourcovať. Druhé štádium predstavuje dobu počas ktorej sú vybrané procesy firmy vykonávané u dodávateľskej

firmy. Tretia fáza predstavuje ukončenie outsourcingu, teda dobu keď sa všetky procesy ktoré boli outsourcované presúvajú buď naspäť do danej firmy alebo k inému dodávateľovi. Prvej fáze sa budem venovať podrobnejšie, pretože je zo všetkých troch fáz najdôležitejšia. Druhú a tretiu fázu spomeniem len okrajovo.

Čo outsorcovať

Prvoradou otázkou každého IT manažéra je, či má vôbec význam outsorcovať niektoré podnikové procesy. Ak je odpoveď na túto otázku kladná, tak sa vynára ďalšia otázka a to ktoré procesy sa môžu outsorcovať a ktoré ostanú vo firme.

IT manažéri musia pozerať produktívne a hlavne proaktívne na svoje vlastné IT operácie a zistiť, ktoré z nich sú strategické a ktoré sú v podstate nederivované služby. Tieto posledné služby sú kandidátmi na outsourcing. Väčšina manažérov vidí systémy a služby, ktoré priamo komunikujú so zákazníkom, ako strategické a ich vývoj aj údržbu je potrebné vykonávať v rámci podniku. Na druhej strane údržba aplikácií alebo help desk sú záležitosti, ktoré navzájom nediferencujú firmy a je možné ich považovať za kandidátov na outsourcing. Dôležité je teda zachovať služby a procesy ktoré sú strategické z hľadiska konkurencie, teda tie ktoré robia firmu jedinečnou – konkurencie schopnou, zachovať v podniku. Predsa by nebolo dobré keby sa firma vzdala v prospech niekoho iného toho čo ju živí a robí jedinečnou.

Začiatok, priebeh a ukončenie outsourcingu

Začiatok outsourcingu

Pri výbere procesov ktoré sa plánujú outsourcovať je dôležité brať ohľad na to čo sa ide outsourcovať. V tomto prípade platí pravidlo pomaly ďalej zájdeš.

Nie je vhodné vzdať sa hneď tých najväčších procesov, ale začať tými najjednoduchšími. Takto si firma môže otestovať služby a schopnosti dodávateľa a na základe výsledkov mu odovzdať do správy ďalšie procesy.

Dôležitým faktorom z hľadiska kvalitného outsourcingu sú znalosti dodávateľa. Tento by mal poznať oblasť, v ktorej firma podniká ako aj jednotlivé súvzťažnosti tohto biznisu. Nie je nič horšie ako odovzdať časť svojho podniku do rúk niekoho kto nevie nič o danej doméne podnikania. V procese plánovania outsourcingu by sa preto nemalo zabúdať na zaškolenie a zaučenie zamestnancov dodávateľa, ktorí budú zodpovední za vykonávanie outsourcovaných procesov. Teda vykoná sa prenos vedomostí, ktoré daná firma nadobudla počas predošlej doby, smerom k dodávateľovi outsourcingu.

Práve tu nastáva jeden zo základných kameňov rozhodnutia či outsourcovať alebo nie. Spomenutým prenosom vedomostí sa v podstate prenáša know-how firmy, teda to, čo ju robí jedinečnou. Riešením je presne diferencovať čo je základom spoločnosti (core business), tento v žiadnom prípade neoutsourcovať a outsourcovať podporné procesy. Napríklad firma, ktorá vytvára svoje vlastné softvérové aplikácie, by nemala

outsourcovať celý softvérový proces. Minimálne návrh a počiatočný vývoj by mal ostať v jej rukách. Týmto sa zabezpečí požadovaná kvalita programov. Predsa len jej zamestnanci vedia najlepšie čo ich zákazníci žiadajú a toto žiadny outsourcing nenahradí. Zároveň je zabezpečený aj rýchly reakčný čas na zmeny návrhu.

Potrebné zaškolenie sa dá najlepšie dosiahnuť tým, že zamestnanci dodávateľa sa aktívne zúčastnia jednotlivých činností na strane objednávateľa. Pred príchodom týchto ľudí je výhodné vytvoriť plán, ktorý zahrňuje všetky procesy, ktoré sú predmetom prenosu vedomostí. Jeho časťou je takisto aj identifikácia kľúčových zamestnancov odberateľa, ktorí sú expertmi na procesy ktoré sa idú outsourcovať. Zároveň sa identifikujú aj zamestnanci dodávateľa ktorí získajú tieto vedomosti, časový plán a záverečné testy. Takýto plán potom znižuje riziko stratenia prehľadu o tom, ktoré informácie už dodávateľ dostal a ktoré ešte má dostať.

Každý outsourcing má okrem svojej kladnej stránky aj negatívnu stránku. Keďže výsledkom outsourcingu je predanie interných procesov externej entite, tak v najhoršom prípade dochádza k znižovaniu ľudských zdrojov – prepúšťaniu zamestnancov. V tom lepšom prípade dôjde iba k ich preradeniu do iných oblastí. Počas príprav na outsourcing je teda dôležité brať ohľad aj na takýchto zamestnancov. Ak by spanikárili, tak sa môže jednoducho stať, že z firmy odídu kľúčoví zamestnanci aj so svojimi vedomosťami. V procese outsourcingu je teda potrebné stimulovať postihnutých zamestnancov tak, aby ostali vo firme a svoje vedomosti mohli predať dodávateľovi daných služieb. Firma by mala takýmto zamestnancom umožniť presadenie v inom oddelení firmy alebo im pomôcť pri hľadaní druhého zamestnania. Týmto sa môže radikálne zlepšiť prechod na outsourcované procesy.

Ako som už spomenul, dôležitou činnosťou v procese prechodu na outsourcing je prenos vedomostí. Toto však nie je len jednostranná činnosť a je potrebné aby aj na strane dodávateľa boli zamestnanci odberateľa, ktorí kontrolujú korektnosť vykonávania jednotlivých funkcií na strane dodávateľa. Zároveň môžu manažéri kontrolovať plnenie vytvoreného plánu ako aj vykonávať testy vedomostí a kvality jednotlivých procesov u dodávateľa.

Proces vzdelávania však nekončí splnením tohto plánu. Pokračuje počas celej spolupráce oboch firiem. Je to dané tým, že procesy odberateľa a biznis sa menia. Pri každej takejto zmene sa musia zase zaškoliť zamestnanci dodávateľa, aby dokázali držať krok s meniacim sa trhom a rýchlo reagovať na jeho zmeny.

Priebeh outsourcingu

Outsourcing následne prebieha počas obdobia dohodnutého oboma stranami. Počas tohto obdobia obe strany spolu komunikujú za účelom dosiahnutia požadovanej kvality služieb. Odberateľ aktualizuje vedomosti zamestnancov dodávateľa aby mohli držať krok s novo vyvinutými postupmi. Aby odberateľ nebol 100% závislý od dodávateľa, tak je zároveň vhodné, aby zamestnanci odberateľa boli školený na činnosti ktoré sú outsourcované.

Ukončenie outsourcingu

Po uplynutí doby outsourcingu nastáva ďalšia fáza, počas ktorej sa odberateľ rozhoduje či bude tieto procesy ďalej outsourcovať alebo ich prevezme do plnej miery. Rozhodnutie závisí od niekoľkých faktorov. Sú to hlavne tie, ktoré rozhodovali o prvotnom outsourcingu. Teda či sú náklady na outsourcing stále nižšie ako náklady na prevádzku vo vlastnej réžii alebo nie.

Outsourcing vývoja softvéru

Outsourcovanie vývoja softvéru je jednou z najviac outsourcovaných činností firiem. Dôvod je rovnaký ako v každom inom prípade outsourcingu. Nižšie náklady na vývoj pri dosiahnutí vyššej kvality. Outsourcing vývoja softvéru je vhodný pre firmy, ktoré sa primárne nezaoberajú jeho tvorbou. Príkladom môže byť firma zaoberajúca sa výrobou pneumatík. Udržiavať tím softvérových vývojárov len kvôli údržbe a vývoju softvéru pre vlastnú potrebu je značne nákladné. Zároveň by takáto firma musela investovať do rôznych školení aby udržala vedomosti týchto zamestnancov na najnovšej technologickej úrovni.

Proces vývoja softvéru je možné rozdeliť do štyroch kategórií:

- Vývoj v rámci vlastných možností
- Outsourcing vývoja komponentu
- Outsourcing procesu vývoja softvéru
- Kompletný outsourcing vývoja softvéru

Vývoj v rámci vlastných možností predstavuje vývoj softvéru v rámci firmy s nasadením vlastných zamestnancov na vývoj softvéru. Výhodou takéhoto vývoja je komplexná znalosť problémovej oblasti vývojárov ako aj možnosť rýchle reagovať na zmeny v danej oblasti. Táto výhoda predstavuje jeden z hlavných dôvodov proti voľbe outsourcovania. Zamestnanci, ktorí majú priame skúsenosti s predmetom podnikania firmy dokážu určite lepšie pochopiť požiadavky používateľov na softvér ako nejaká iná firma, ktorá sa špecializuje len na tvorbu softvéru a z danej doménovej oblasti má minimálne alebo dokonca žiadne znalosti. Nevýhodou je naopak nízka kvalifikovanosť vývojárov ako aj ich obmedzený „akčný rádius“ v ktorom tvoria programy. Postupným programovaním si totiž vytvorili už svoje overené postupy, ktoré používajú stále dookola a teda do softvéru neprichádza zmena, ktorá by ho mohla prípadne vylepšiť alebo priniesť nejakú kladnú zmenu.

Outsourcovanie vývoja určitého komponentu alebo komponentov je výhodné v prípade, ak firma nemá dostatočné kapacity na vývoj celého systému. V tomto prípade je dodávateľ zaviazaný vytvoriť dané komponenty podľa požiadaviek zákazníka. Tu sa už začínajú uplatňovať výhody outsourcingu, pretože dané komponenty sú vyvíjané kvalifikovanými pracovníkmi s dostatočnou praxou vo vývoji softvéru. Pri takomto vývoji sa dajú očakávať vhodné výsledky, pretože návrh celého systému stále vykonávajú ľudia v danom obore.

Podobne je na tom aj outsourcing procesu vývoja softvéru. Tu sa však neoutsourcuje vývoj komponentov ale celých procesov vývoja softvéru. Napríklad návrh, implementácia prípadne vývoj prototypu. Výhody takéhoto outsourcingu som už spomenul v predchádzajúcich častiach tejto práce. Tu je hlavnou výhodou krátky reakčný čas na požiadavky zmeny softvéru.

Poslednou kategóriou je kompletný outsourcing vývoja softvéru. Tento je používaný firmami, ktoré nepotrebujú mať rýchlu odozvu na zmeny v ich podnikateľskej sfére a teda nie je potrebná okamžitá reakcia vo forme nového vývoja prípadne zmeny existujúceho softvéru. Naopak, firma ktorá potrebuje rýchlo zareagovať na zmeny v jej oblasti aj svojím softvérom asi využije outsourcing procesu vývoja, konkrétne niektorých záverečných procesov – návrh a implementácia finálnej verzie.

Outsourcing vývoja softvéru prebieha podobne ako každý iný outsourcing. Na začiatku si obe strany dohodnú čo bude predmetom outsourcingu. Teda od odberateľa sú to špecifikácie požiadaviek na nový softvér. Dodávateľ zase dodá predbežnú cenovú kalkuláciu. Táto kalkulácia je tým presnejšia, čím presnejšie sú požiadavky klienta. Platby na outsourcovaný vývoj sa zvyčajne platia počas celej doby outsourcovania. Typicky sa 25% zaplatí na začiatku projektu po stanovení požiadaviek a ďalšieho postupu. 50% sa platí počas priebehu projektu, zvyčajne po splnení nejakých dielčích požiadaviek. Zvyšných 25% sa zaplatí po ukončení projektu.

Kontrola kvality outsourcovaných služieb je kapitola sama o sebe. Ak odberateľ nemá s dodávateľom ešte žiadne skúsenosti, je veľmi dôležité aby bol každý krok vývoja kontrolovaný. Pri dohadovaní kontraktu by si mali obe strany stanoviť určité okamihy počas vývoja kedy odberateľ skontroluje a ohodnotí činnosť dodávateľa aby sa predišlo sklamaniu na oboch stranách. Týmto sa zabezpečí aj požadovaná kvalita softvéru zo strany odberateľa.

Ďalšou kapitolou outsourcingu vývoja softvéru je použitie „on-site" vývojárov. V tomto prípade sú odberateľovi dodaní zamestnanci, ktorí pracujú v jeho firme na jeho projektoch. Tento spôsob outsourcingu sa používa vtedy, ak firma nemá dostatočný počet ľudí na daný projekt. Títo pracovníci sú stále platení svojím zamestnávateľom, s tým rozdielom, že pracujú na inom mieste. Pri tomto spôsobe však vznikajú aj určité problémy a to hlavne v komunikácii. Trvá určitú dobu kým takýto človek zapadne do kolektívu a začne naplno pracovať. Takisto tu môže vzniknúť problém s adaptáciou tohto pracovníka na procesy a postupy, ktoré sú v rámci firmy zabehané. Či už ide o jednotlivé stupne vývoja, formát a obsah dokumentácií ako aj o samotný štýl programovania. Takýto postup je teda nevhodný pre krátke projekty, kde by sa vývojár nestihol adaptovať na novú situáciu. Naopak, pri dlhodobejšom projekte sa môže adaptovať až natoľko, že bude rovnako výkonný ako každý jeho „nový" kolega.

Podobný princíp je možné realizovať aj opačne a to v prípade, keď odberateľ chce dohliadnuť na kvalitu vyvíjaného softvéru a pošle jedného zo svojich zamestnancov na pracovisko dodávateľa, kde dohliada, prípadne riadi, proces vývoja softvéru. V prípade zabehaných a overených partnerov je však takýto postup zbytočne nákladný.

Záver

Outsourcing je proces, ktorý sa uplatňuje už od 70-tich rokov 20. storočia. Predstavuje prostriedok na zníženie nákladov firiem a pomáha im odbremeniť sa od „zbytočných“ činností, ktoré nesúvisia s ich podnikaním. Namiesto toho tieto činnosti presúva spoľahlivým a skúseným partnerom, ktorí ponúkajú dlhoročné skúsenosti s dodávaním požadovaných služieb a teda často krát môžu priniesť aj niečo nové do už zabehanej firmy.

Outsourcing je vynikajúci nástroj na zníženie nákladov, ale len v prípade, že sa správne použije. Outsourcovanie nekritických softvérových procesov firmy je správne riešenie použitia outsourcingu. Pri každom outsourcingu je však potrebné dávať pozor na presné stanovenie požiadaviek outsourcingu a takisto vykonať kvalitne prenos vedomostí smerom k dodávateľovi outsourcingu. Ak firma potrebuje mať rýchly reakčný čas na zmenu špecifikácie svojho softvéru, tak nie je vhodné outsourcovať celý proces tvorby softvéru ale len jeho konečnú časť. Teda vývoj prototypu je lepšie vykonávať vo vlastnej réžii.

Pri zvážení všetkých kladov a záporov outsourcingu dokáže firma znížiť náklady na vývoj softvéru a zároveň zvýšiť jeho kvalitu ako aj kvalitu svojich služieb.

Použitá literatúra

1. Donald J. Reifer: Seven Hot Outsourcing Practices. *In IEEE Software*, Vol. 21, No. 1 (January/February 2004), 14-16.
2. Outsourcing research center, Články o outsourcingu, http://www.cio.com/research/outsourcing
3. Govexec, Články o outsourcingu, http://www.govexec.com/outsourcing
4. The Outsourcing Institute, Články o outsourcingu, http://www.outsourcing.com
5. Wesley Bertch: How Offshore Outsourcing Failed Us. *Network computing*, Oct 16, 2003
6. National outsourcing association, Články o outsourcingu, http://www.noa.co.uk

Annotation

Outsourcing

This paper describes the outsourcing from the viewpoint of IT management. It discusses the outsourcing problematic from the beginning up to the end of the whole process. The requirement to cut down the costs can be seen in almost every industry segment. Personnel costs are the major costs of every company which is the main reason why outsourcing is more and more discussed theme. In this paper I address especially the problematic of identifying the business processes which are needed/desirable to outsource. The attention is also put on the environment of software companies. Main attention is paid to processes involved in software systems development. In this manner I address the problems of software development processes and also to dedicated workers. These processes are divided into two categories according to

their priority for outsourcing. These groups are detailed described and explained their pro and contras in reference to outsourcing.

Vývojové tímy v softvérovom inžinierstve

JÁN ŠNIRC

Slovenská technická univerzita
Fakulta informatiky a informačných technológií
Ilkovičova 3, 842 16 Bratislava

Abstrakt. Vývoj softvéru je zložitý proces, ktorého úspech záleží od viacerých aspektov. V technickej rovine sú to použité technológie, nástroje a metodológia, na druhej strane spôsob organizácie tímu a komunikácie medzi jednotlivými členmi. Cieľom tejto eseje je prezentovať základné typy organizácie tímov, ich výhody, nevýhody a vhodnosť pre jednotlivé metodológie. Na záver uvádzam vlastné skúsenosti z pôsobenia v jednom tíme.

Úvod

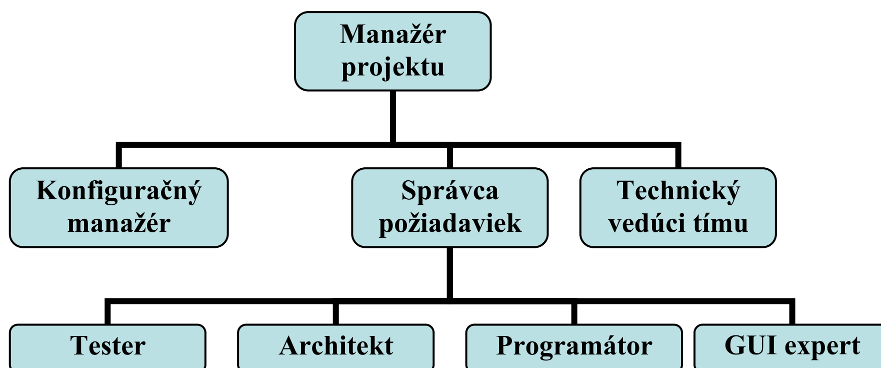
Spôsob organizácie vývojového tímu v softvérových projektoch závisí od viacerých faktorov. Od komplexnosti projektu, ľudí, veľkosti tímu, dĺžky jeho existencie až po použitú metodológiu pri vývoji softvéru. Práve použitá metodológia výrazným spôsobom určuje štruktúru tímu. Avšak treba povedať, že jej výber ovplyvňujú tiež predošlé faktory, takže ako to už býva, všetko so všetkým súvisí.

V hrubom priblížení rozoznávame tri základné typy vývojových tímov [2]: sekvenčný, skupinový a sieťový typ. Každý z nich reprezentuje odlišný pohľad na organizáciu tímu, komunikáciu medzi jednotlivými členmi ako aj samotný vývoj softvéru. Jednotlivé typy podrobnejšie opíšem v nasledujúcom texte.

Sekvenčný typ

Dobrý proces vedie k dobrému výsledku. To je základná myšlienka, ktorá ovplyvňuje formovanie vývojárskeho tímu do sekvenčného typu. Vývoj softvéru vychádzajúci z tohto presvedčenia je prísne lineárny a riadený úlohami, čo sa odráža aj na štruktúre tímu (Obr. 1). Tá je formalizovaná a založená na rolách. Jednotliví pracovníci sú špecialistami na konkrétne oblasti, v ktorých samostatne riešia pridelené úlohy a sú ohodnotení presne za to, čo vyprodukujú. Keďže vývoj je riadený úlohami, ktoré sú dopredu dané, potreba diskusie medzi ostatnými členmi tímu je minimalizovaná a v prípade potreby prebieha iba cez formálne kanály vertikálnym smerom. To implikuje slabé sociálne väzby medzi členmi tímu a ich jednoduchú výmenu. Ďalšou

výhodou takejto organizácie, ktorá sa začne prejavovať pri väčšom počte ľudí je jednoduché rozdelenie tímu na oddelenia a riadenie, čím je nedochádza k nedorozumeniam pri riadení a komunikácii. Z pohľadu členov tímu je takáto štruktúra vhodná ak sú všetci skúsenými odborníkmi na pridelenú oblasť. V prípade menej skúsených ľudí, ktorý potrebujú dopĺňať svoje znalosti a jednotlivé kroky konzultovať nastáva značná neefektivita, pretože komunikácia v sekvenčnom type je už z jeho podstaty obmedzená a pomalá.



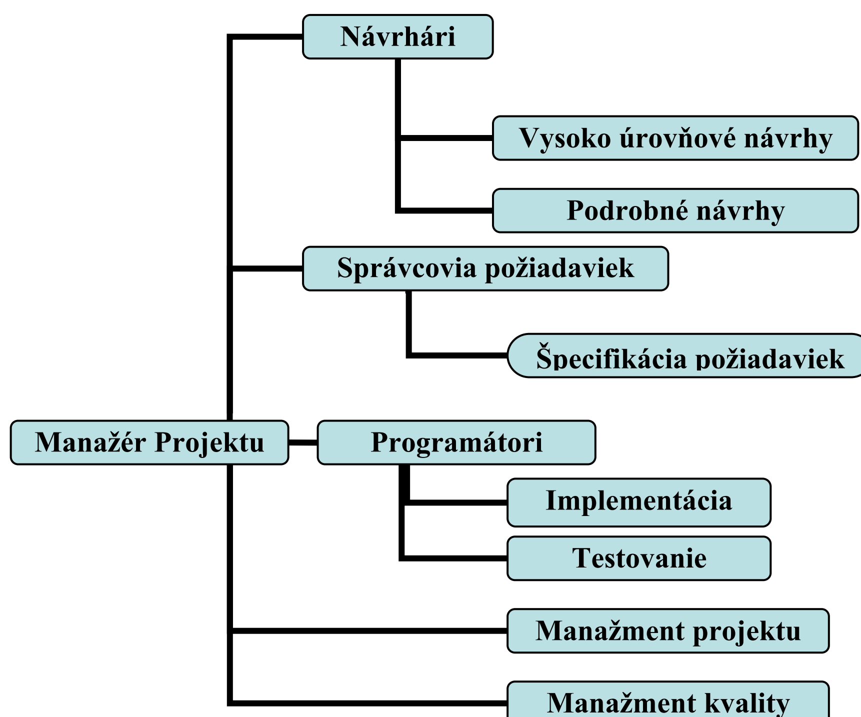
Obr. 1 Centrálne riadený tím

Typickým prípadom kedy je táto forma organizácie použitá je vodopádový model vývoja softvéru. Jednotlivé fázy vodopádového modelu sa vykonávajú v striktnom poradí, bez prekrývania či iteratívnych krokov.

Skupinový typ

Vývoj softvéru v skupine je založený na úlohách, avšak na rozdiel od sekvenčného typu ich riešia všetci členovia skupiny a to iteratívne. Už samotný fakt, že vývojári pracujú v skupine na spoločnej úlohe implikuje silné sociálne väzby, ktoré okrem formálnej komunikácie ústia aj do neformálnej. Sila skupiny spočíva práve v komunikácii medzi členmi a vzájomnom zdieľaní svojich znalostí. Vývoj v skupine však môže znamenať aj vznik konfliktov medzi členmi a zníženie produktivity. Z tohto dôvodu je potrebné zaviesť pravidlá, ktoré predchádzajú vznikom konfliktov a venovať dodatočné úsilie na vytvorenie priateľskej atmosfére v skupine

Príkladom skupinového typu sú špirálový a evolučný prístup k vývoju softvéru. Ten je rozdelený na iterácie, ktorých výstupom je pridaná nová funkcionalita. Výsledkom iterácie sú tiež získané poznatky, ktoré sú použité v nasledujúcej iterácii.



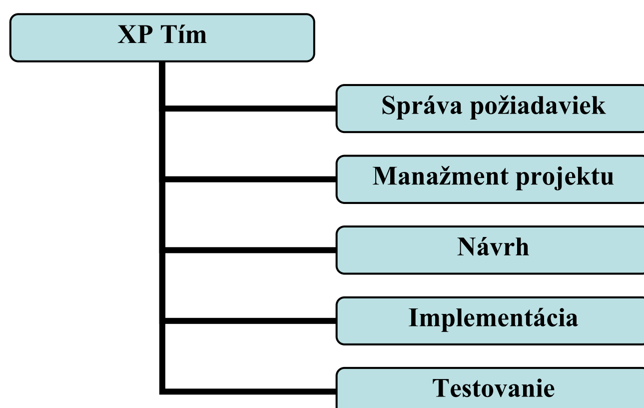
Obr. 2 Decentralizovane riadený tím

Vývoj softvéru je riadený decentralizovane (Obr. 2). Na čele tímu je definovaný koordinátor, ktorý riadi niektoré úlohy a lídrov zodpovedných za príslušné úlohy. Komunikácia medzi skupinami a jednotlivcami prebieha horizontálne, medzi koordinátorom a lídrami vertikálne.

Relatívne nový prístup k vývoju softvéru predstavujú agilné metódy. Základné myšlienky sú vyjadrené v manifeste agilného vývoja softvéru [3]. Z pohľadu organizácie softvérových tímov je dôležitým faktom, že agilné metódy upriamujú pozornosť na jednotlivcov a ich vzájomnú komunikácia. Preferované sú blízke vzťahy v tíme a uzavreté pracovné prostredie a iné aktivity, ktoré zvyšujú morálku v tíme.

Jedným z derivátom agilných metód je extrémne programovanie, ktorého autorom je Kent Beck. Ide o zjednodušená metódu vývoja softvéru založenú princípoch jednoduchosti, komunikácie a spätnej väzby. K zaujímavým prvkom tejto metódy patrí myšlienka zdieľaného kódu. Každý člen vlastní celý kód tímu, rozumie mu a môže ho flexibilne meniť. Na to, aby princíp zdieľaného kódu fungoval je potrebné samozrejme zaviesť konvencie písania kódu. Ďalším prvkom, ktorý ovplyvňuje vnútornú štruktúru skupiny je zavedenie dvojíc programátorov, ktorí spolu úzko spolupracujú pri implementácii a vzájomne sa dopĺňajú. XP tím je organizovaný demokraticky [Obr. 3], t.j. neexistuje žiadny stály líder a komunikácia medzi členmi tímu prebieha horizontálne. Výhodami takejto organizácie je prispievanie každého člena k rozhodnutiam, učenie sa jeden od druhého, čím sa zvyšuje spokojnosť s prácou.

Okrem štandardných nevýhod, ktoré so sebou prináša práca v skupine sa pridáva aj zníženie osobnej zodpovednosti v dôsledku demokratického rozhodovania.

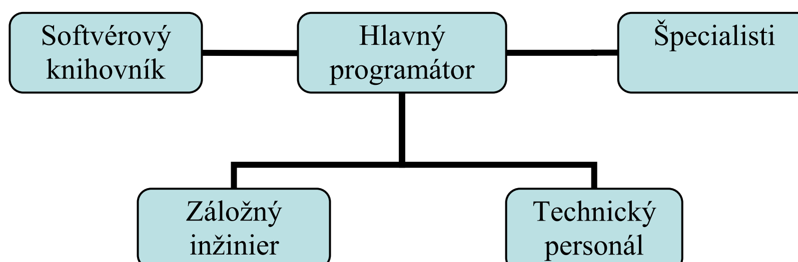


Obr. 3 Organizácia XP tímu

Sieťový typ

V centre pozornosti sa nenachádza proces, ale produkt. Sieťový typ softvérového tímu vychádza z presvedčenia, že skúsení ľudia tvoria dobrý softvér. Tím tvorí sieť s uzlami, ktorými sú skúsení programátori a koncovými bodmi, ktoré tvoria menej zdatný členovia tímu, pričom vytvorená sieť nemusí mapovať organizáciu spoločnosti, či geografickú polohu členov. Úlohy, ktoré treba riešiť nie sú dopredu dané, ale vznikajú z komunikácie v sieti, ale existujú aj centrálne úlohy ako je napr. správa verzií, tvorba dokumentácie. Výhodou tohto typu je rýchly vývoj aplikácii v tíme kde sú skúsenosti jednotlivých členov na rôznej úrovni a fluktuácia menej skúsených členov je relatívne vysoká. Na druhej strane odchod človeka, ktorý tvorí uzol siete znamená pre projekt významnú stratu, nielen pre jeho odborné znalosti, ale aj pre väzby s ostatnými členmi tímu, ktoré bude musieť jeho nástupca nanovo nadväzovať. Ďalšou nevýhodou je komunikácia na diaľku, ktorú treba dodatočne podporiť nástrojmi a môže byť zdrojom nedorozumení.

Typickým príkladom môže byť organizácia tímu vyvíjajúca open source softvér, alebo tím hlavného programátora [1] (Chief programmer team). Ten vznikol v prostredí IBM v 70. rokoch ako odpoveď na existenciu značných rozdielov medzi vývojármi a vysokou fluktuáciou tých menej skúsených. Tím je postavený okolo skúseného hlavného programátora a ostatný členovia tímu tvoria pre neho technickú a netechnickú podporu (Obr. 4).



Obr. 4 Organizácia tímu hlavného programátora

Riadenie tímu je prísne centralizované. Riešenie hlavných problémov a koordinácia ostatných členov je v kompetencii hlavného programátora, ktorý tvorí jadro tímu. Ako podporu má pridelených dvoch až piatich pracovníkov na jeho podporu. Jedná sa ostatných programátorov, dokumentaristov a softvérového knihovníka, ktorý môže pracovať pre viaceré tímy a stará sa o softvérovú konfiguráciu, asistuje pri výskume a príprave dokumentácie.

Výhodou je centralizované rozhodovanie a redukcia zbytočnej komunikácie. Na druhej strane je celý tím závislý od hlavného programátora a jeho správania sa, čo znižuje morálku v tíme.

Všeobecné princípy organizovania tímu

Bez ohľadu na typ organizácie tímu existujú všeobecné princípy, ktoré môžu zvýšiť efektivitu tímu.

- Menej ľudí, ale kvalitných
- Plniť úlohy vzhľadom na možnosti a motiváciu v tíme
- Vyberať ľudí tak aby výsledkom bol harmonický tím
- Ak niekto nezapadne do tímu, radšej ho vymeniť
- Brooksov zákon – pridanie nových ľudí do projektu, ktorý mešká spôsobí jeho ďalšie oneskorenie.

Osobné skúsenosti z práce v tíme

V súčasnosti som členom tímu, ktorý vyvíja softvér a preto som sa rozhodol prezentovať spôsob jeho organizácie a výhody resp. nevýhody z nej prameniace.

Projekt predstavuje zadanie riešené v rámci predmetu Tímový projekt inžinierskeho štúdia FIIT STU. Samotné zadanie projektu ako aj fakt že medzi členmi tímu existovali už predtým neformálne vzťahy vyústil do organizácie tímu, ktorá najlepšie zodpovedá typu skupina. Počas riešenia projektu používame princípy definované v agilných metódach a to najmä úzku spoluprácu medzi členmi tímu, tvorbu softvéru bez dokumentácie a pripravenosť na zmeny. Riadenia tímu je demokratické bez stáleho tím lídra. Samotný projekt bolo možné intuitívne rozdeliť na viaceré pod úlohy a k nim prislúchajúceho lídra, ktorý viedol diskusiu ale rozhodnutie bolo nakoniec tímové a súhlasili s ním všetci členovia.

Výhodou takejto organizácie je jednoznačne zdieľanie informácií v tíme a pri výskyte nových problémov brainstoring, ktorý generuje množstvo zaujímavých spôsobov riešenia. Problémom, ako som sa sám presvedčil môže byť demokratické rozhodovanie o riešení problému. V našom prípade išlo konkrétne o návrh komunikačného rozhrania. Predstavy členov tímu boli značne odlišné a všetky navrhované riešenia v princípe správne. To spôsobilo približne dvojtýždňové meškanie tejto úlohy a pri pohľade späť by v tomto prípade bolo autoritatívne riadenie vhodnejšie.

Záver

Určenie najlepšieho spôsobu organizácie vývojového tímu, ako vyplýva z prechádzajúceho textu nie je triviálna záležitosť. Voľba resp. kombinácia uvedených typov záleží od riešeného projektu, znalostí členov tímu ich schopnosti vzájomne komunikovať, osobnej skúsenosti z práce v tíme atď. Cieľom príspevku bolo preto najmä priblíženie faktorov, ktoré tento výber ovplyvňujú.

Použitá literatúra

1. Baker, F. Chief programmer team management of production programming. IBM Systems Journal 11, 1 (Jan. 1972), 56–73.
2. Sawyer, S. : Software development teams. Communications of the ACM December 2004 Vol. 47, No. 12
3. Manifesto for Agile Software Development, http://agilemanifesto.org/ Marec 2005

Annotation

Software development teams

Software development is a complex process and its result depends on many aspects. In technical level that are technologies, tools and methodology. On other side the way of organizing development team and communication among team member. The goal of this work is to present basic archetypes of development team structure, their advantages, disadvantages and suitability for different

methodologies. At the end I introduce my own experiences from working in two different team types.

Prístupy k zdieľaniu informácií v distribuovaných projektoch

MIROSLAV VNUK

Slovenská technická univerzita
Fakulta informatiky a informačných technológií
Ilkovičova 3, 842 16 Bratislava

Abstrakt. Práca rozoberá a analyzuje rôzne možné prístupy k uchovávaniu a zdieľaniu informácií, ktoré boli nadobudnuté počas riešenia rôznych na sebe viac či menej závislých projektov, riešených v distribuovanom prostredí. Tieto informácie kategorizuje a na nich popisuje klady a zápory jednotlivých prístupov. Na záver popisuje reálne systémy, ktoré implementujú uvedené prístupy.

Úvod

Dnešný svet patrí informatike a informáciám. V tomto svete majú informácie vysokú cenu, keď sa nachádzajú na správnom mieste v správnych rukách. Nastáva však problém, ako žiadanú informáciu vhodne rozdistribuovať medzi ľudí, ktorí majú o nej vedieť. Touto problematikou sa zaoberá táto práca a pokúša sa vniesť pohľad na prístupy k zdieľaniu informácií (nielen) v distribuovaných projektoch.

Informácie

Na začiatok by som chcel uviesť význam slova informácia v akom je v práci použitý. Význam informácie je chápaný v zmysle užitočných významných poznatkov nadobudnutých riešením rôznych projektov. Môže sa jednať o poznatky nielen iba z určitej časti riešeného projektu, ale o ucelené poznatky nadobudnuté počas riešenia projektu počas celého jeho životného cyklu. Na základe takto definovaného významu informácie je možné samotné informácie deliť do troch skupín [3].

Prvá skupina predstavuje *informácie nadobudnuté počas riešenia projektov*. Zahŕňa technickú dokumentáciu k projektu, ďalej rôzne použité študijné materiály použité k vyriešeniu projektu a časť riadiacej dokumentácie – záznamy z porád (hlavne čo sa týka technickej realizácie projektu).

Druhá skupina, do ktoré možno klasifikovať *informácie reprezentuje poznatky získané z manažovania projektov*. V tomto prípade je to najmä riadiaca dokumentácia vytváraná k príslušnému projektu zahŕňajúca najmä informácie o časovom plánovaní jednotlivých úloh, stave ich plnenia, deadlines. Ďalej zahŕňa rozdelenie príslušných počtov ľudí na príslušné typy úloh vrátanie ich zaradenia na konkrétne dielčie úlohy. Táto skupina informácií by mala obsahovať aj informácie týkajúce sa financií celého projektu (napríklad: návratnosť investícii, finančné prostriedky vynaložené na riešenie dielčích úloh, ...).

Tretia posledná kategória, do ktorej je možné informácie klasifikovať je predstavuje *informácie získané po ukončení príslušného projektu*. Jedná sa o rôzne post analýzy a audity úspešnosti projektu zamerané či už na menšie časti, alebo na celý projekt.

Delenie prístupov k zdieľaniu informácií

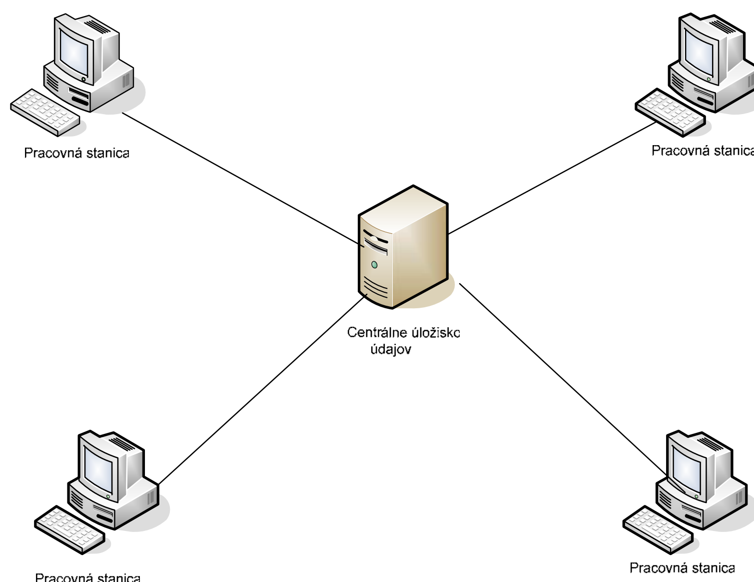
Takto kategorizované informácie je potrebné zdieľať medzi rôznymi riešiteľmi projektov (čím sa uľahčí riešenie „už raz" riešených častí, aj keď boli riešené na inom projekte). Samotné zdieľanie týchto informácií je možné realizovať rôznymi spôsobmi, ktoré sa líšia neskôr uvedenými vlastnosťami.

Prístup k zdieľaniu informácií pri riešení projektov v distribuovanom prostredí (rôzne tímy riešia rôzne projekty v rôznych, vzdialených lokáciach) je možné rozdeliť do troch skupín. Prvý prístup je možné nazvať *centrálny model* a ako už názov napovedá centralizovane uchováva informácie. Druhý prístup vychádza z P2P (peer-to-peer) architektúry, z toho je odvodené aj jeho meno – *P2P model*. Tretí posledný prístup k zdieľaniu informácií reprezentuje *hybridný model*, ktorým reprezentuje istú kombináciu prvých dvoch uvedených prístupov. V tejto práci bude uvedený jeden konkrétny hybridný prístup, s konkrétnymi vlastnosťami, pričom vo všeobecnosti v tomto prístupe návrhu môžu prevažovať viac či menej vlastnosti prvého alebo druhého prístupu.

Centrálny model

Na opis vlastností uvedeného prístupu, reprezentovaného týmto modelom, sa dá pozerať z rôznych pohľadov - abstraktný pohľad, nepozerá na samotnú technickú implementáciu a jej vlastnosti, ktoré sa môžu v konkrétnych systémoch líšiť, ale popisuje všeobecné vlastnosti prístupu, ktoré pokrývajú takmer všetky konkrétne systémy založené na príslušnom modeli. Na druhú stranu je to konkrétny pohľad na jednotlivé systémy alebo na skupinu rovnakých riešení. Predstavuje v podstate opak abstraktného pohľadu.

Na začiatok sa budem venovať abstraktnému pohľadu na vymenované prístupy a začnem centrálnym modelom.



Obr. 9. Centrálny model zdieľania dát

Aj v tomto prípade sa dá na výhody a nevýhody tohto prístupu pozerať z viacerých hľadísk: *zdieľanie dát*, *riadenie prístupu* a *štruktúra dát*. Z pohľadu *zdieľania dát* vyplýva jednoznačná výhoda a tou je oddelenie informácií od používateľa a ich centrálne uchovávanie. Jednou z výhod z toho vyplývajúcich je nezávislosť informácií príslušného používateľa od aktuálneho stavu jeho počítača (či je vypnutý, zapnutý, funkčný, nefunkčný). Ďalšou výhodou centrálneho uchovávania je fakt, o ktorom vedia všetci, ktorí systém používajú, a tým je jednoznačné miesto, kde požadované informácie hľadať a tým sa v podstate jedná o štandard, ktorí všetci používajú.

Nakoľko všetko má aj isté nevýhody, aj tento prístup z pohľadu zdieľania dát má isté nevýhody. Jednou z nich je relatívne dlhý časový úsek medzi nadobudnutím informácie a jej zdieľaním medzi používateľmi, čím hrozí v istom zmysle strata, alebo znehodnotenie informácie. Ďalšou nevýhodou je istá strata kontroly používateľa nad ním vytvorenými informáciami, ktorá sa dá riešiť zavedením istých prístupových práv.

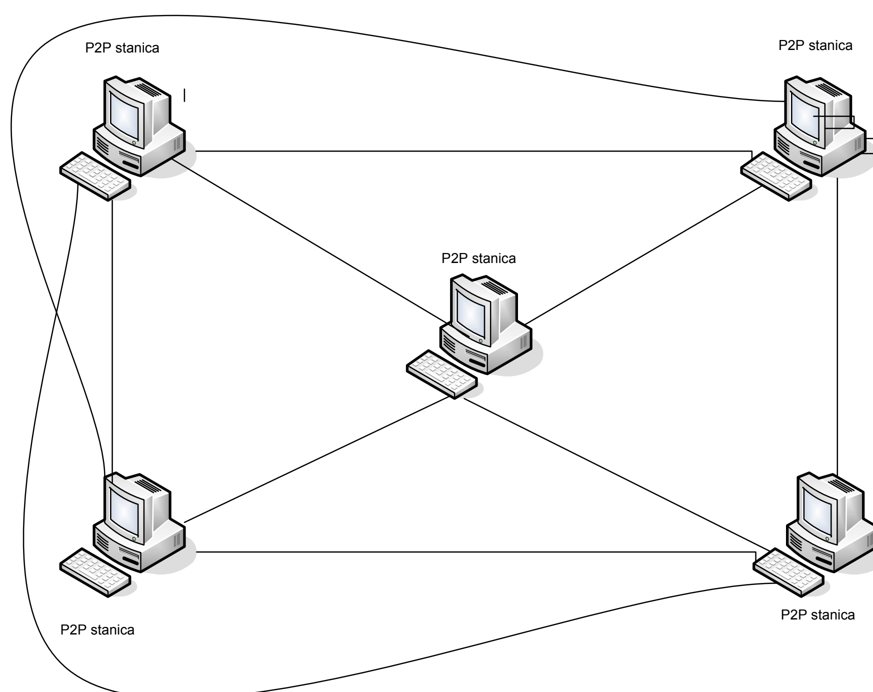
Z pohľadu riadenia prístupu vzišla azda len jedna už spomenutá nevýhoda, a tou je riadenie prístupu kvôli spätnej kontrole nad informáciami používateľom, ktorý príslušnú informáciu vytvoril. Z pohľadu štruktúry informácií má tento prístup asi najvýznamnejšie žiadané vlastnosti, a tými sú pevná organizácia štruktúry informácií, optimalizovaná najmä na rýchlosť, použitie rôznych druhov filtrov a vyhľadávacích schém. S tým sú ale spojené určité nevýhody, najmä komplikovanosť návrhu dátovej štruktúry použitej na uchovávanie rôznych informácií. Ďalšia nevýhoda je spôsobená samotnou pevnou štruktúrou informácií, a tou je nedostačujúca variabilita a tým

nemožnosť zaznamenávať všetky typy informácií, čím môže v určitom zmysle dôjsť k strate informácie.

Z uvedených výhod a nevýhod vyplýva použitie tohto prístupu na uchovávanie informácií, ktoré sa príliš často nemenia a nemajú „príliš veľkú“ variabilitu, a to sú *informácie získané z manažovania* a *po ukončení projektov*.

P2P (peer-to-peer) model

Ďalším možným, už spomínaným prístupom k zdieľaniu informácií, je model P2P (peer-to-peer).



Obr. 10. P2P model zdieľania dát

Z pohľadu zdieľania informácií sa tento prístup ukazuje ako vhodnejší aj oproti predchádzajúcemu prístupu v minimalizovaní časového úseku medzi nadobudnutím informácie a jej zaznamenaním a určením pre zdieľanie pre ostatných používateľov. Ukazuje sa, že osoba, ktorá nadobudne nejakú informáciu, ju radšej a hlavne rýchlejšie zdieľa u seba ako v nejakom centrálnom dátovom sklade (rozdiel oproti *centrálnemu modelu*). Z hľadiska riadenia centrálneho prístupu je tento model oproti predchádzajúcemu hlavne jednoduchší na realizáciu, keďže netreba centrálne určovať a riadiť prístup. Každý používateľ je zodpovedný za svoje informácie, ktoré zdieľa

a kto má k nim prístup. Pri zmene informácie znovu nenastáva žiadny problém, jednoducho si svoje informácie prepíše, keďže každý sám si spravuje svoje informácie.

Z hľadiska štruktúry dát je tento prístup komplikovanejší ako predchádzajúci, nakoľko jednotliví prispievatelia informácií nepoužívajú jednotnú štruktúru ukladania a zdieľania informácií. Z toho v podstate plynú už uvedené výhody a ďalej vymenované nevýhody. Ako výhodu je ešte možné označiť vlastnosť zdieľať informácie s veľkou variabilitou a tým je možné citlivejšie určovať informačnú úroveň príslušnej zdieľanej informácie. Nevýhoda spôsobená vysokou variabilitou a nejednoznačnou štruktúrou je znemožnenie kvalitného a plnohodnotného vyhľadávania. Na vyhľadávanie je potrebné použiť rôzne špecializované agenty, ktoré sa každý zameriavajú na určitý typ informácií. Tým je samozrejme predĺžená dĺžka vyhľadávania a ochudobnená možnosť použitia filtrov a vyhľadávacích schém.

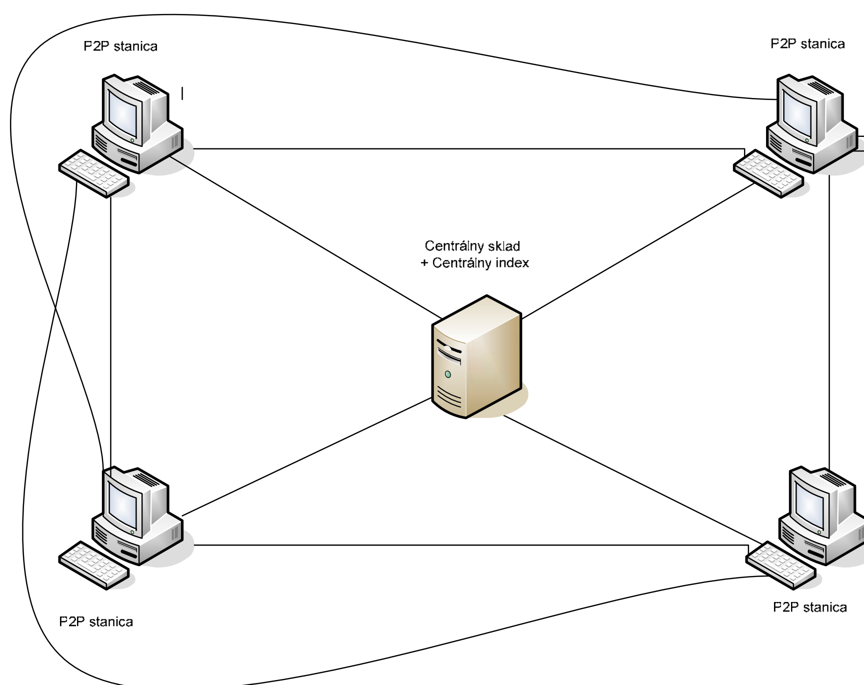
Z uvedených výhod a nevýhod vyplýva, že tento prístup k zdieľaniu informácií je najvhodnejšie použiť na zdieľanie informácií vykazujúcich veľkú variabilitu a časté obmieňanie. Z uvedenej klasifikácie typov informácií je tento systém najvhodnejšie použiť na zdieľanie *informácií získaných počas riešenia projektov*.

Hybridný model

Posledným možným, aj keď nie až tak jednoznačným prístupom z hľadiska zdieľania dát, je hybridný model. Nejednoznačnosť modelu vyplýva z možnosti existovania množstva hybridných modelov kombinujúcich predchádzajúce prístupy, pričom každý môže mať iné vlastnosti. Následne je uvedený jeden z možných modelov v ktorom je snaha o čo najideálnejšiu kombináciu predchádzajúcich modelov, s ohľadom na uvedenú klasifikáciu typov informácií. Pokiaľ by sa zmenili typy informácií, možno by bolo vhodné zvoliť iný spôsob kombinácie modelov.

Konečná podoba systému obsahuje tri časti: *centrálny sklad* informácií (časť prevzatá z *centrálneho modelu*), stanice prepojené P2P obsahujúce *distribuované sklady* informácií (časť prevzatá z *modelu P2P*) a *centrálny inde*x. Centrálny index je v podstate dátový sklad riešený rovnako ako *centrálny model*, ale majúci formu slovníka, ktorého jednotlivé položky odkazujú najmä na informácie uložené v dátových skladoch na P2P staniciach. Najvýznamnejšou nevýhodou tohto popísaného prístupu je komplikovanejšia fyzická realizácia.

Tento typ hybridného modelu využíva výhody centrálneho modelu na zdieľanie informácií, ktoré sa nemenia príliš často a majú relatívne pevnú dátovú štruktúru (*informácie získané z manažovania projektov* a *informácie získané po ukončení projektov*). Tieto výhody kombinuje s výhodami P2P modelu, čiže zdieľanie informácií s variabilnou štruktúrou, čo v tomto prípade predstavujú *informácie získané počas riešenia projektov*.



Obr. 11. Hybridný model zdieľania dát

Použitie v praxi

Popísané prístupy v zdieľaní informácií neexistujú len v teoretickej rovine, ale existujú ale aj ako reálne systémy. Samozrejme, nemusia mať všetky vymenované vlastnosti uvedených prístupov.

Uvedené prístupy vôbec nemusia byť použité pri zdieľaní informácií z projektov v distribuovanom prostredí, ale môžu byť použité na zdieľanie akýchkoľvek informácií v distribuovanom prostredí, ale v tomto prípade poslúžia ako vhodné ukážky z praxe.

Ako príklad centrálneho modelu zdieľania dát môže byť prezentovaný systém FTP (File Transfer Protocol), v ktorom sa komunikácia uskutočňuje iba medzi FTP serverom a klientom. Ak si potom dvaja používatelia chcú zdieľať súbor s využitím protokolu FTP, najskôr musí jeden z nich pomocou svojho FTP klienta nahrať tento súbor na FTP server a cieľový prijímateľ si ho zas musí pomocou svojho FTP klienta stiahnuť. Priame spojenie medzi FTP klientmi nie je možné . [2]

Príkladom P2P modelu by mohol byť systém Gnutela. V ňom jednotliví používatelia siete Gnutela predstavujú rovnocenné uzly. Ak chce používateľ – uzol A získať prístup do siete, musí nájsť aspoň jeden ďalší uzol siete - uzol B. Na hľadanie funkčných uzlov môže poslúžiť zoznam dodávaný priamo s aplikáciou alebo

vyhľadávací systém uzlov. Po nadviazaní spojenia pošle uzol B uzlu A svoj vlastný zoznam a uzol A sa pokúsi nadviazať spojenie aj s nimi. Pri vyhľadávaní v sieti potom pošle uzol A požiadavku všetkým uzlom, s ktorými je aktívne spojený, a tie smerujú jeho požiadavku ďalej. Teoreticky by sa táto požiadavka mala dostať postupne ku všetkým klientom, ale v praxi väčšina z nich neobsiahne ani polovicu všetkých uzlov. Ak používateľ nájde viac uzlov, ktoré poskytujú ten istý súbor, môže využiť segmentové sťahovanie a sťahovať súčasne časti súboru z rôznych uzlov, čím sa zvýši celková rýchlosť. [2]

Posledný príklad zdieľania informácií reprezentovaný hybridným modelom predstavuje systém DC++, ktorý sa spolieha na špeciálne servery, po anglicky nazývané huby (rozbočovače), ktoré súžia ako zhromaždisko používateľov. Jednotlivé huby medzi sebou nekomunikujú. Hub udržiava zoznam všetkých prihlásených používateľov a nimi poskytovaných, zdieľaných súborov a zabezpečuje vyhľadávanie medzi týmito súbormi. Výmena súborov sa vykonáva už priamo medzi jednotlivými používateľmi.[2]

Záver

Ako vidieť uvedené prístupy k zdieľaniu informácií nemusia sa na sto percent odrážať v konkrétnych technických realizáciách. Majú poslúžiť na načrtnutie hlavných prúdov v prístupe k zdieľaniu informácií. To, ktorý si príslušná osoba, tím, firma vyberie, závisí od charakteru zdieľaných informácií a od požiadaviek, ktoré sa od toho očakávajú. Na to potom nadväzuje úspech či neúspech príslušného prístupu v danom prostredí a z toho čiastočne vyplývajúci úspech alebo neúspech riešených projektov.

Použitá literatúra

1. Kevin C. Desouza, J. Roberto Evaristo: Managing Knowledge in Distributed Projects. In *Communications of the ACM*, Vol. 47, No. 4 (April 2004), 87-91
2. Ondík M.: P2P – história a súčasnosť. *PC Revue*, No. 3 (2005) 12-15.
3. Schindler, M. and Damm, D. Security issues of a knowledge medium for distributed project work. *Intern. J. of Project Management 20* (2002), 37–47.

Annotation

Approaches to sharing information in distributed projects

This document analyses and describes various approaches in storing and sharing informations generated by projects solved in distributed environment. It categorizes informations and describes pros and cons of various approaches of sharing these informations. In the end it describes real systems, using analyzed approaches.

Manažment konfliktov medzi programátormi a testermi

ĽUBOŠ LEČKO

Slovenská technická univerzita
Fakulta informatiky a informačných technológií
Ilkovičova 3, 842 16 Bratislava

Abstrakt. Konflikty sú bežnou súčasťou ľudského života a nevyhýbajú sa ani softvérovým projektom. Najčastejším dôvodom konfliktu býva nedostatok nejakých zdrojov. Uvedené sú najčastejšie zdroje konfliktov medzi členmi vývojárskych tímov spolu s potrebným pozadím. Rozobraté sú konflikty z rôznych pohľadov, vplyvy pracovného prostredia na atmosféru a vzťahy v tíme. Práca obsahuje informácie o agilných metódach vývoja softvéru, z ktorých priamo alebo nepriamo vyplývajú možnosti a potreby testovania, a s nimi súvisiace prístupy a opatrenia na riešenie konfliktu. Spomenuté sú odporúčané manažérske prístupy k riešeniu konfliktov v štandardných aj agilných softvérových procesoch.

Úvod

Organizácie stále tvoria jednotlivci, a tí, či už chceme alebo nie, prinášajú so sebou do spoločného prostredia rôzne zvyky, návyky, morálne, sociálne a náboženské hodnoty. V takomto mnohonázorovom prostredí preto stačí len malá iskierka na to, aby sa rozhorel konflikt, ktorý bude mať neodvratný dopad na všetko a všetkých, ktorých sa týka.

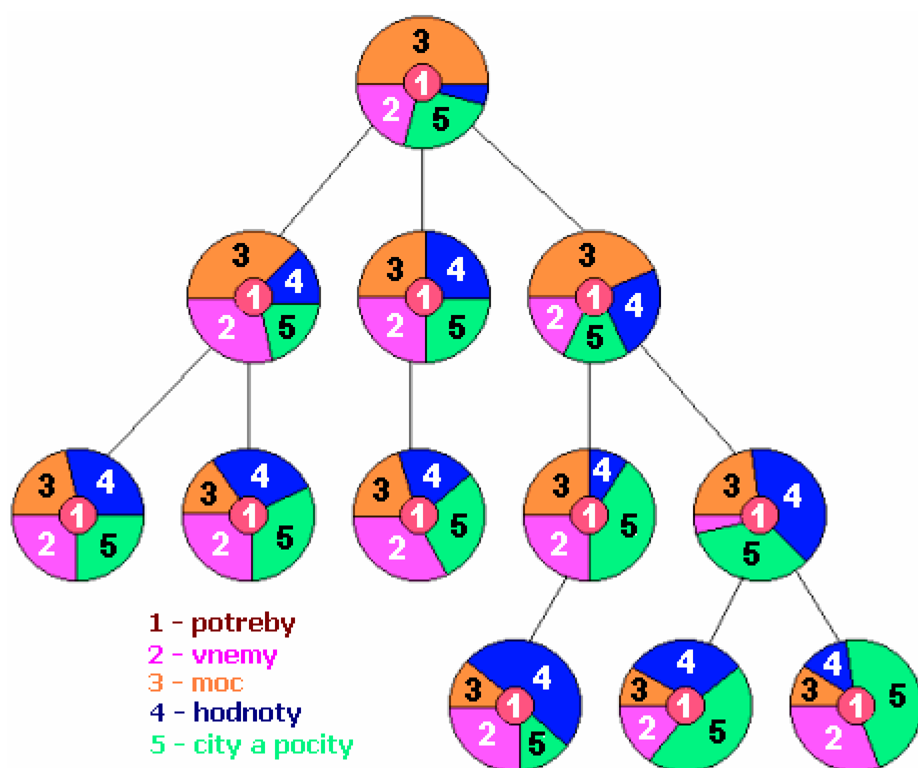
Konflikt - definícia

Konflikt je prirodzený nesúhlas vznikajúci medzi skupinami alebo jednotlivými osobami, ktoré sa líšia v postojoch, náboženstvách, hodnotách alebo potrebách. Zárodok tiež môže mať v sporoch z minulosti a personálnych rozdieloch. Ďalšie príčiny konfliktu zahŕňajú snahu o vyjednávanie v nepravú chvíľu alebo skôr než sú dostupné potrebné informácie[4].

Prísady konfliktu

Konflikt ako taký je len dôsledok nerovnováhy medzi nasledovnými charakteristikami jedincov v organizácii – Obrázok 1:

- *Potreby* – všetko, čo je nevyhnutné k našej telesnej a duševnej pohode; ignorovanie vlastných potrieb, potrieb druhých ľudí alebo potrieb skupiny vedie ku konfliktu.
- *Vnemy* – ľudia vnímajú realitu rôzne, rozdielne vidia závažnosť, príčiny a dôsledky problémov.
- *Moc* – dôležitý vplyv na počet a typ konfliktov v závislosti od toho, ako ju ľudia definujú a najmä používajú; ovplyvňuje aj to, ako je konflikt zmanažovaný.
- *Hodnoty* – všetko, čomu my prikladáme osobitý význam; konflikty vznikajú, keď ľudia majú nezlúčiteľné alebo nejasné hodnoty.
- *City a pocity* – veľký počet ľudí necháva city a pocity ovplyvniť ich vysporiadanie sa s konfliktom.



Obrázok 1: Prísady konfliktu a možné rozloženie ich vplyvu v hierarchii riadenia organizácie.

Krúžky na Obrázok 1 predstavujú jednotlivcov v organizácií, prepojenia krúžkov pracovné vzťahy medzi nimi. Každý krúžok má iné delenie vnútornej plochy, rovnako ako je iný pomer prísad v každom zamestnancovi.

Každý zamestnanec si z každej uvedenej prísady prináša rôzne množstvo, čo sa prejaví na jeho individuálnej povahe a prístupe. Keby všetci zamestnanci boli rovnocenní z pohľadu vzájomných pomerov jednotlivých prísad, určite by dochádzalo k menšiemu množstvu konfliktov. Na základe starej známej pravdy – oko za oko, zub za zub – každý by sa snažil vyhnúť konfliktu, lebo vie, že rovnako sa zachová celé jeho okolie. Samozrejme môžeme použiť aj úvahu, že každý by sa snažil okoliu čo najviac robiť napriek, pretože rovnako by sa okolie správalo k nemu, lenže potom by takýto systém asi nedosiahol ciele, za účelom ktorých bol vytvorený.

Konflikty v softvérových tímoch

Vývoj softvéru, tak ako ostatné ľudské činnosti, obsahuje veľa potencionálnych zdrojov konfliktu, ktoré môžu poškodiť výkonnosť činnosti. Rovnako ako môže vzplanúť konflikt medzi vývojármi a používateľmi, môže sa objaviť aj vo vnútri tímu, najčastejšie medzi vývojármi a testermi.

Pretože konflikt často ovplyvní pracovnú výkonnosť a kvalitu výsledného produktu, identifikácia jeho zdrojov je kritická nielen v súkromnom sektore, ale aj na akademickej pôde. Na základe informácií uvedených v [1] možno na konflikt v softvérových procesoch nahliadať v troch rovinách ilustrovaných na Obrázok 2:

- Proces
- Ľudia
- Organizačná štruktúra

Obrázok 2: Roviny konfliktu[1].

Konflikt z pohľadu procesu

Aj keď nedostatok času nie je unikátnym javom v softvérových procesoch, najčastejšie spomínaným zdrojom konfliktu podľa programátorov a manažérov je rozdelenie času medzi fázy vývoja a testovania. Organizácie sa snažia dodať na trh komplikované, bezchybné produkty a čas sa jednoducho stáva menej prístupným a teda viac hodnotným. Testovanie je často oneskorené a jeho dĺžka skrátená za účelom dodržania termínov dodávky.

Testeri a programátori súperia navzájom o čas na splnenie svojich záväzkov, výsledkom čoho je konflikt. Vďaka prirodzenej následnosti testovania po programovaní testerom často zostáva málo času. Testeri vidia najväčší problém vo fakte, že manažéri nechávajú programátorov produkovať kód až do posledného momentu, často až do večera pred finálnou dodávkou. A keďže programátori nie sú tí, ktorých starosťou je dodať finálny produkt zákazníkovi, pri pohľade na harmonogram nepociťujú potrebu rýchlo dokončiť to, s čím už meškajú. Skrátka ako keby nemali rovnaký zmysel pre povinnosť. A čím viac sa implementácia oneskoruje, tým viac rastie napätie medzi programátormi a testermi.

Ako ďalší dôvod na konflikt sa uvádza rozdielne chápanie požiadaviek. Zatiaľ čo testeri bývajú viac zameraní na požiadavky používateľov, programátori upriamujú svoju pozornosť na technické aspekty výsledného produktu. Napriek tomu, že určite ide o dôležité pohľady na proces, výsledkom nemusí byť len lepšie pochopenie a zvládnutie problémovej oblasti, ale aj veľa problémov.

Čas sám osebe je požiadavkou na efektívne testovanie. Aj keď niektorí testeri očakávajú nedostatok času na výkon svojej profesie, stále to môže vyvolať stres a konflikt. Skrátenie času na testovanie interpretujú tak, že ich práca nie je dostatočne rešpektovaná. Manažéri musia pre každú fázu softvérového procesu naplánovať dostatočne dlhý čas a neuberať z neho, ak nejaká skupina nestihne svoje termíny.

Konflikt často vznikne, ak programátori a testeri nezdieľajú rovnaké pracovné ciele. Programátori sú zamestnávaní najmä preto, že nemajú problém vymýšľať nové riešenia. V záujme ich realizácie nie vždy dodržia používateľské požiadavky. A to je tŕňom v oku testerov, ktorých úloha v tíme je jednak overiť funkčnosť a kvalitu riešení od programátorov, a jednak čiastočne striehnuť ne dodržiavanie používateľských požiadaviek. Z toho vyplýva, že je potrebné jasne definovať skupinové a individuálne ciele pre potreby vytvorenia softvéru, ktorý sa na trh dostane načas a súčasne bude kvalitný.

Konflikt z pohľadu ľudí

Základným stavebným kameňom organizácií sú stále ľudia – individualisti, z ktorých každý prináša individuálny prístup, vnímanie problémov, mentálne pochody atď.

Testeri samých seba zväčša opisujú ako puntičkárov, pedantov, ktorí sú schopní vytvoriť súdržnú skupinu. Programátori sú manažérmi opisovaní ako kreatívni, temperamentní individualisti. Aj keď každá táto charakteristika jednotlivým profesiám len prospieva, v rámci tímu sa môže prejaviť ako rušivý element.

Podľa vyjadrení testerov programátori napíšu kód aplikácie a pre vlastné potreby ho otestujú spôsobom, akým by sa mala aplikácia používať[1]. Myšlienkové pochody

programátora sú často veľmi odlišné od pochodov testera, týka sa to aj ich rozdielneho chápania potreby testov. Testeri teda program otestujú spôsobom, akým sa aplikácia bude používať, čo je pre programátorov ťažko pochopiteľná záležitosť. Môže to byť sčasti spôsobené aj faktom, že niektorí programátori vnímaju svoj kód ako predĺženie svojej osobnosti a v ňom objavenú chybu chápu osobne. Konflikt naberá na intenzite so vzrastajúcim počtom odhalených chýb.

Aj keď niektorí testeri sa postupom času naučili pomocou metódy pokus-omyl komunikovať s programátormi, keď očakávajú konflikt, tento predpoklad nemožno použiť na všetkých testerov. Stále častejšie si manažment organizácií uvedomuje, že trénovanie členov tímu v komunikačných a psychologických zručnostiach prispieva k zníženiu rizika vzniku konfliktov, a tiež k ich rýchlejšiemu riešeniu.

Konflikt často vychádza z faktu, že programátori a testeri nevnímajú softvérový proces rovnako. No čo je ešte dôležitejšie, svoje rozdielne pohľady si neuvedomujú. Organizované pracovné aktivity, ktorých úlohou je oboznámiť programátorov a testerov s podstatou práce druhej strany, pomáhajú v prevencii pred problémami.

Konflikt z pohľadu organizačnej štruktúry

Aj keď väčšina organizácií si uvedomuje, že potrebujú vysoko kvalifikovaných testerov a ich zručnosti, testeri stále zápasia o zisk zaslúženého rešpektu, o rešpekt porovnateľný s programátormi.O veľkosti ega niektorých programátorov hovorí výrok:“Ak by ste mali diagram s Bohom na vrchu, programátori by svoju pozíciu umiestnili nad neho.“ Aj vďaka tomu musia testeri dennodenne vyvíjať úsilie na to, aby si udržali rešpekt porovnateľný s programátormi. Nedostatok podpory zo strany manažérov robí prácu testera náročnejšou z fyzickej, psychickej a časovej stránky.

Tento zápas im nijako neuľahčuje ani pokrok v komunikačných technológiách, keď problémy, ktoré by vyriešili priamou komunikáciou za 10 minút, riešia cez mail alebo fax celý deň.

Softvérové metódy a ich riešenie konfliktu

V súčasnosti najrozšírenejšími metódami vývoja softvéru sú tie plánom riadené, v posledných rokoch sa však stále viac presadzujú agilné metódy. Stále je však ťažké určiť, či je lepšie, jednoduchšie alebo lacnejšie plánovať alebo prerábať.

Plánom riadené metódy

Čistý plánom riadený vývoj softvéru predpokladá, že architektúra, používateľské požiadavky a celkový dizajn systému môžu byť vytvorené už v skorých fázach procesu a implementácia je ponechaná na neskoršie fázy.

Metódy využívajúce plánovanie sú náchylnejšie na nedodržanie termínov, stačí, aby sa oneskorila čo i len jedna fáza. Postupy, ako zabrániť konfliktu alebo ho riešiť, zahŕňajú:

- Zlepšenie komunikácie v rámci tímu – zamestnanci musia byť už od začiatku trénovaní v riešení konfliktov, rozvíjaní svojich komunikačných schopností a využívaní neformálnych sprostredkovacích techník.
- Využitie roly nestranného arbitra – ak konflikt prerastie do neúnosných rozmerov, neodkladne ho treba riešiť. Nestranný arbiter je nie príliš často používanou možnosťou na jeho riešenie. Vypočuje obe strany, predloží návrhy riešenia a vyjedná zmierlivé riešenie.
- Podpora tímových aktivít – organizácia z režijných nákladov financuje aktivity pre zamestnancov vo voľnom čase; zamestnanci si na seba rýchlejšie zvyknú a komunikácia medzi nimi prejde do menej strohej a formálnej roviny, menej formálna komunikácia často zrýchli riešenie problémov.
- Ekvivalentný prístup k testerom v porovnaní s programátormi.

Agilné metódy

Stratégie agilných metód vývoja softvéru sa usilujú zredukovať cenu neustálych zmien pomocou vývoja jednoduchých riešení, pravidelných zmien návrhu a stáleho testovania. Medzi najvýznamnejšie môžeme zaradiť:

XP – Extreme Programming

Spomedzi všetkých agilných metód si vyslúžila najviac pozornosti. Ide o proces založený na experimentovaní a vylepšovaní, na základe používateľských požiadaviek sa stanoví séria testov, ktorú musí implementácia splniť, aby bola považovaná za riešenie.

Scrum

Iteratívny, inkrementálny proces. Výsledkom každej iterácie je časť funkcionality, ktorá môže byť eventuálne samostatne expedovaná k zákazníkovi. Dĺžka iterácie je 30 dní[5].

Feature Driven Development

Proces navrhnutý najmä pre objektovo orientovaný prístup. Výhodou sú krátke iterácie v dĺžke 14 dní, ktorých úlohou je rýchlo doručiť hmatateľnú funkcionalitu zákazníkovi.

Spoločnou vlastnosťou agilných metód je skrátenie dodacích cyklov, čím sa ale tiež skracuje doba, počas ktorej je možné vytvorené riešenie testovať. Preto manažéri, ktorých organizácie používajú niektorú z agilných metód, musia s ešte väčším dôrazom podporovať testovaciu fázu projektu a vyhýbať sa konfliktom. Napríklad *XP* používa na obranu pred konfliktmi v tíme kolektívne vlastníctvo kódu. Povzbudzuje každého člena tímu prispieť k procesu novými nápadmi, nech už ide o fázu implementácie alebo testovania. Ak každý člen tímu má rovnaký podiel snahy na výslednom produkte, zaslúži si aj rovnaký diel úcty a rešpektu = účinné predchádzanie konfliktu. Tímy

bývajú tvorené menším počtom ľudí, čím sa v porovnaní s ostatnými metódami vytvorí lepšie prostredie na neformálnu komunikáciu, zlepší prehľad o možnostiach a schopnostiach jednotlivých členov. Vylepšovanie kódu, ktorého vlastníkom je každý člen tímu, je spoločný cieľ a stanovenie spoločného cieľa pre programátorov a testerov predchádza konfliktom.

Záver

Na konflikt medzi programátormi a testermi možno nazerať v troch rovinách, všetky sú špecifické pre prostredie vývojárskych tímov. Jednotlivci v pozíciách programátorov a testerov niekedy majú problém spolu vychádzať. V skupine ľudí to nie je nič neočakávané. Nepriateľská povaha testovania softvéru a rozdiely medzi jeho jednotlivými účastníkmi sú istou zárukou konfliktov. Úlohou manažérov je týmto konfliktom predchádzať, na výber majú široké spektrum metód, sprostredkovaných, manažovaných aj nemanažovaných. Samozrejme je potrebné konflikty riešiť tak, aby ich dopad na projekty bol minimálny. Záleží len od ich skúseností, prostriedkov alebo zvolenej metódy, aký výsledok riešenia konfliktu sa dostaví.

Použitá literatúra

1. Cynthia F. Cohen, Stanley J. Birkin, Monica J. Garfield, Harold W. Webb: Managing Conflict in Software Testing. In Communications of the ACM, Vol. 47, No. 1 (January 2004), 76-81.
2. Martin Fowler: The New Methodology, http://www.martinfowler.com/articles/newMethodology.html, 7.5.2005
3. businesslistening.com: A Simple Process for Resolving Business Conflicts, http://www.businesslistening.com/conflict_resolution-2.php, 7.5.2005
4. Managing conflict, http://www.ctic.purdue.edu/KYW/Brochures/ManageConflict.html, 7.5.2005
5. Scrum, http://www.controlchaos.com/about/, 7.5.2005

Annotation

Managing conflicts between developers and testers

Conflicts are natural part of human life, including software projects. Lack of resources is the most frequent source of conflict. The most frequent sources of conflict among development teams are given including necessary background. Various layers of conflict are considered. The paper contains information on agile methods of software development with testing requirements and conflict resolving precautions. Management methods in standard and agile processes are given.

Počítanie riadkov zdrojového kódu

MILAN ŽUŽO

Slovenská technická univerzita
Fakulta informatiky a informačných technológií
Ilkovičova 3, 842 16 Bratislava

Abstrakt. V súčasnosti sa metóda počítania riadkov zdrojového kódu bežne používa na odhadovanie času (najčastejšie v jednotke človeko-mesiace) potrebného na úspešné dokončenie projektu. Taktiež sa využíva na meranie efektivity práce programátora alebo tímu. V tejto práci opisujem rôzne pohľady na počítanie riadkov zdrojového kódu. Zameriavam sa hlavne na problémy spojené s používaním tejto metódy v spomenutých dvoch oblastiach a podrobnejšie rozoberám rôzne prístupy k počítaniu týchto riadkov. Podávam náčrt alternatívnych možností merania produktivity. Každá metóda má nejaké nedostatky a preto je mojím cieľom v tejto práci popísať spôsob, akým sa dá táto metóda používať vzhľadom na jej nedostatky.

Úvod

Počítanie riadkov zdrojového kódu (LOC – lines of code) je jedna z najstarších, ak nie práve najstaršia metóda merania veľkosti softvérových produktov. Aj napriek tomuto faktu sa ešte stále bežne používa na odhadovanie veľkosti projektu alebo na meranie produktivity práce či už programátora alebo celého tímu. Nastáva otázka, či je táto metóda stále použiteľná. Vývoj softvéru predsa prešiel výraznými zmenami. Má riadok kódu v asembleri, ktorý sa písal pred niekoľkými rokmi, porovnateľnú hodnotu s riadkom v niektorom z vyšších programovacích jazykov? Jedným z riešení je napríklad prevodová tabuľka, ale je jasné, že pri používaní tejto metódy vo vyvíjajúcom sa prostredí nastanú isté komplikácie. Takéto a podobné problémy viedli k vzniku nových metód, ako je napríklad metóda funkčných bodov (angl. Function Points). Aj keď sa snažia nahradiť počítanie riadkov zdrojového kódu, stále nie sú univerzálne použiteľnými metódami – majú svoje výhody, ale aj nevýhody. V tomto článku sa pokúsim priblížiť problémy, ktoré vznikajú pri počítaní LOC a prípadne spôsoby odstránenia ich vplyvu.

Odhad veľkosti projektu

V dnešnej dobe sa projekt robí pre zákazníka a ten chce vedieť, kedy bude produkt hotový. Konkurencieschopnosť firmy závisí od schopnosti dodať výrobok včas a v požadovanej kvalite. Preto sa manažment, ktorý je zodpovedný za daný projekt, musí zaoberať problémom stanovenia termínu dokončenia projektu. Tieto odhady sa opierajú o metódy, ktoré merajú veľkosti projektov a snahou je aproximovať podobné projekty na náš konkrétny pripravovaný projekt. Ale aj v súčasnosti používané metódy nie sú bezchybné – vždy je to len odhad.

LOC

Ako Phillip Armour popisuje vo svojich článkoch [2] a [3], použitie počítania LOC na odhadovanie času potrebného na úspešné dokončenie projektu nás privádza k dileme. Keď chceme odhadovať veľkosť projektu, tak sme ešte len v počiatočnom štádiu projektu a nemáme ešte žiaden kód – teda nemáme čo počítať. A keď už máme kód, tak je projekt už hotový a nepotrebujeme odhadovať. Takže to vyzerá tak, že LOC pri odhadovaní sú presné, ak ich nepotrebujeme a nie sú k dispozícii, ak ich potrebujeme.

Hlavným problémom je to, že naším cieľom nie je získať LOC ale vedomosti. Ak riadky zdrojového kódu neobsahujú tie „správne“ vedomosti, je jedno koľko ich je. Oveľa horší je fakt, že nepotrebujeme vedieť množstvo vedomostí, ktoré máme, ale to, ktoré musíme získať (ktoré nám chýba, aby bol projekt hotový). Ľudia ešte stále nevymysleli jednotku pre množstvo vedomostí ani spôsob ako ich odvážiť, takže takéto odhady nie je ľahké urobiť. LOC možno vyjadrí nejaký priemer, ale použiť ju ako jednotku na meranie množstva vedomostí je skoro ako odváženie knihy za účelom zistenia, koľko vedomostí obsahuje.

Projekt však nepozostáva iba zo zdrojového kódu. Na jeho zvládnutie treba prejsť rôznymi fázami jeho životného cyklu, ako je zber požiadaviek, analýza, návrh, testovanie a pod. Niekedy sa požiadavky zbierajú týždeň, inokedy dva mesiace. A táto informácia sa v počte riadkov zdrojového textu programu nevyskytuje.

Funkčné body

Štandardný prístup International Function Points User Group (IFPUG) zahŕňa počítanie a váženie vstupov, výstupov a základných prvkov na uchovávanie dát s úpravami zvolenými s ohľadom na prostredie, v ktorom bude systém pracovať [1]. Tieto prvky sú skoro vždy prístupné v skorej časti projektu, keď je potrebné spraviť odhad. Ale pri tejto metóde môžme pozorovať zopár zaujímavostí. Napr. bežným krokom procedúry metódy funkčných bodov je preklad medzi LOC a funkčnými bodmi prostredníctvom procesu, ktorý sa nazýva „backfiring“ [4]. Ak vezmeme do úvahy, že pôvodným zámerom bolo odísť od LOC, pôsobí to trochu ironicky. Niektoré procedúry odhadovania umožňujú merať veľkosť systému podľa modulov, komponentov, objektov a iných častí systému. Ale vo väčšine prípadov musia byť tieto počty sprevádzané faktorom, ktorý určuje, koľko LOC je na daný prvok.

Ak odhliadneme od toho, že niektoré procedúry sa vracajú k LOC, vyzerajú funkčné body na odhadovanie veľkosti projektu o trocha lepšie ako samotné počítanie

LOC. Ale aj tu nastáva problém s tým, že podľa Phillipa Armoura nemeriame vedomosti. Napr. zložitosť telekomunikačného systému nemusí vôbec súvisieť s jeho vstupmi a výstupmi. V takomto prípade nedostaneme správny odhad.

Podrobnejší opis iných metód ako LOC je mimo rozsah tohto článku, preto sa im nebudem venovať. Chcel som iba ukázať, že aj keď funkčné body pôvodne chceli nahradiť počítanie LOC (keďže počet riadkov ešte na začiatku projektu nemáme), nakoniec sa aj tak spravil prevod medzi týmito dvoma metódami.

Meranie produktivity práce

LOC sa často používa na meranie produktivity práce zamestnancov. Nemusí to byť vždy za účelom zisťovania, kto ako pracuje, ale taktiež sa touto metódou dá sledovať pokrok projektu. Ak sa výrazne zmení počet LOC za dané skúmané obdobie oproti predošlým obdobiam, je to indikáciou toho, že niečo možno nie je v poriadku a manažment projektu má šancu včas zareagovať a identifikovať problém.

Veľa programátorov má však voči tejto metóde predsudky, pretože zastávajú názor, že nie je objektívna. Zavedenie tejto metódy do takéhoto prostredia môže byť niekedy kontraproduktívne. Na jednom fóre som sa stretol s prípadom, kedy manažér zaviedol do tímu počítanie LOC. Programátori vtedy začali rozťahovať formátovanie kódu a písať zbytočne rozsiahle komentáre. Keď na to manažér prišiel a začal počítať bodkočiarky (písali vtedy v jazyku C), programátori začali dávať bodkočiarky do komentárov. Možno ich prístup nebol veľmi profesionálny a vzťah manažéra s tímom možno tiež nebol najlepší, ale vyplýva z toho isté ponaučenie. Nie je vždy ľahké zaviesť do praxe metódu, o ktorej nie je každý presvedčený, že je objektívna.

Meranie produktivity so sebou nesie aj iné komplikácie. Občas sa optimalizáciou zníži počet riadkov kódu. Vezmime si situáciu, kedy programátor potrebuje istú časť kódu optimalizovať a zaberie mu to celý deň. Na konci dňa úspešne dokončí svoju prácu, ale má menej riadkov kódu ako na začiatku dňa. Nemôžme predsa povedať, že nepracoval. Preto výkyvy v počte riadkov treba zaznamenávať s poznámkami o tom, čo bolo príčinou daného poklesu. Možno lepším spôsobom by bolo počítanie riadkov, ktoré sú odlišné od napr. predchádzajúceho dňa. Takáto metóda by zahrnula aj optimalizáciu kódu, či opravovanie chýb. Na druhej strane sa to dá takisto zneužiť lenivými členmi tímu. Ale pri správnej aplikácii do tímu by táto metóda bola pravdepodobne o niečo presnejšia ako počítanie LOC.

Ďalšou možnosťou pre sledovanie pokroku tímu je percentuálne vyjadrenie toho, koľko tím z rôznych súčastí svojej práce/projektu dokončil. Či sa jedná o vylepšovanie jednotlivých častí programu (napr. COM alebo DLL) alebo celý program, vývojár má mapu (aspoň vo svojej hlave) toho, čo treba urobiť, ako je to rozdelené do menších častí, ako zložitá môže daná časť byť, v akom časovom rozmedzí by sa daná časť dala dokončiť, atď. S realizáciou je to už ťažšie, pretože takýto odhad spraví človek najlepšie iba sám pre seba. Na posúdenie schopností tímu treba nejakú prax, spoluprácu s ním na niekoľkých projektoch a takisto treba poznať znalosti jednotlivých členov tímu.

Pri počítaní LOC sa niekedy počítajú aj riadky binárnych súborov a dokumentácie. Tak sa prípadné väčšie množstvo administratívnej práce prejaví v počte riadkov. Ale považovať to za „jedinú správnu" metódu nie je správne. Myslím, že manažér, ktorý sa o to pokúsi, časom zistí, že jeho najmenej produktívny človek tvrdo zapracuje na počte riadkov, ale napriek tomu zostane málo produktívny. K tomu vedie aj problém kvality oproti kvantite. Počet riadkov neurčuje koľko práce programátor spravil, hovorí iba koľko v ten deň napísal. Myslím, že dávať prednosť kvantite pred kvalitou pri produkte alebo službe je dosť chabý manažment. Dalo by sa to prirovnať k stavaniu múru z tehiel. Ak niekto postaví múr, ktorý sa mal stavať dva dni, za jeden deň, ale múr nie je taký rovný, ako ho chcel, stálo to za tie peniaze?

Ako počítať

Ak zoberieme do úvahy všetky spomenuté príklady počítania LOC v praxi, volá to po zavedení jednotného spôsobu počítania. Spôsobu, ktorý by bol najviac objektívny pre obe strany – programátora i vedenie. Táto úloha je však do dnes nevyriešená, pretože názory na to čo a ako počítať sa líšia.

Príkladov nájdeme v praxi hneď niekoľko. Už pri spomínaných komentároch je na jednej strane názor, že komentár nepridáva žiadnu novú funkcionalitu, preto by sa nemali počítať. Proti je takisto argument, že programátor môže komentáre „zneužívať" na zbieranie ďalších riadkov, aby navonok preukazoval vyššiu produktivitu. Na druhej strane je názor, že komentár je potrebný pre lepšiu čitateľnosť a zrozumiteľnosť kódu. A keďže sa ním programátor musí zaoberať a venuje mu istý čas, malo by sa to odraziť aj vo veľkosti zdrojového kódu (počítaného podľa metódy LOC).

Podobne je to s testovacím kódom. Pri vyvíjaní softvéru programátor často napíše kód, ktorý slúži iba na umožnenie alebo aspoň uľahčenie testovania hotových častí projektu. Takýto kód sa píše „na zahodenie", a vo výslednom produkte sa nikdy nevyskytuje. Znovu sú tu dve strany – jedna tvrdí, že čo nepridá žiadnu novú hodnotu ku koncovému výstupu, preto sa nepočíta. Druhá strana tvrdí, že testovací kód je potrebný na úspešné zvládnutie projektu. Je jeho neoddeliteľnou súčasťou, tak prečo ho vlastne nepočítať?

Ako posledný príklad uvediem ešte znovupoužitie kódu. Podľa jednej teórie sa počítajú iba riadky, kde sa na tento kód odkazuje (teda kde sa volá). Tu už veľmi neplatí dôvod, že programátor musí znovupoužitému kódu venovať „plnohodnotný" čas. Ale existujú zástancovia názoru, že čas ušetrený na tomto mieste sa minie niekde inde (napr. pri podrobnejšom študovaní riešenej problematiky) a vytvára istú rovnováhu medzi počtom LOC a veľkosťou projektu, čo v konečnom dôsledku by malo zvýšiť presnosť odhadu pomocou tejto metódy. Ak si spomenieme na nápad s počítaním rozdielov (alebo zmien) v zdrojovom kóde, niekedy môže spôsobiť ťažkosti nezarátať znovupoužitý kód (ak by sme to mali aspoň automatizované, čo asi je pravdou vo väčšine prípadov).

LOC ako jeden z ukazovateľov

Ako z vidíme z vyššie opísaných príkladov, počítanie riadkov nie je jednoznačnou metódou, ktorá objektívne vyjadruje veľkosť projektu alebo meria produktivitu práce. Môžeme to tvrdiť už len preto, lebo neexistuje jednotný spôsob počítania, na ktorom by sa zhodli všetci, ktorí túto metódu používajú.

Jedna firma si spravila v tejto oblasti nasledovný experiment [2]. Zozbierala 20 rôznych potenciálnych ukazovateľov a sledovala aký vplyv budú mať na čas, potrebný na dokončenie projektu. Zistila, že iba 12 z týchto 20-tich ukazovateľov malo vôbec nejaký vzťah a iba 8 z nich malo vzťah, ktorý bol dostatočný na zváženie tohto ukazovateľa pri odhadovaní veľkosti projektu. Najvýznamnejším ukazovateľom sa stal počet zmien, s ktorými prišiel zákazník počas prvého mesiaca projektu. To očividne nemá veľa spoločného s počtom riadkov zdrojového kódu...

Armour dáva vo svojom príspevku [2] pekný príklad na porovnanie použitia počítania LOC s návštevou u lekára. Píše, že počítanie LOC a metóda funkčných bodov sú iba indikáciou nejakej zmeny, ale nič bližšie o nej nehovoria. Je to ako keď navštívime lekára s teplotou 38°C. Čo nám je? To nám lekár nebude vedieť povedať z tohto jediného ukazovateľa, ale nazbiera ďalšie ukazovatele a až keď ich bude mať „dostatočný“ počet, povie, že nazbierané ukazovatele indikujú nejaký konkrétny stav (napr. chorobu). Niekedy nebudú ukazovatele indikovať správne a musia byť vysvetlené. Podobne je to s LOC a ostatnými metódami. Ak sa od jedného systému očakáva, že bude mať 2-krát toľko riadkov ako druhý, indikuje to, že jeho vytvorenie asi potrvá oveľa dlhšie (zvyčajne proporcionálne k počtu LOC), ale nemusí to byť pravidlom.

V konečnom dôsledku robí odhad iba človek (väčšinou stanovený expert), ktorý pri zbieraní rôznych ukazovateľov môže dôjsť k iným výsledkom ako keby tie isté ukazovatele zbieral nejaký iný človek. Dvaja experti môžu dôjsť k úplne iným odhadom. Jeho posudzovanie je subjektívne vzhľadom na jeho znalosti a skúsenosti. Ak už robil viac odhadov, pravdepodobne spraví lepší odhad ako niekto, kto robí svoj prvý odhad. Niekedy sa termín určuje dokonca z druhej strany – od zákazníka. Zákazník si určí, že v tento termín to potrebuje a softvérová firma sa prispôsobí. Toto nie je najlepší prístup z hľadiska kvality, ale väčšinou sa termín dodrží, aj keď je niečo trochu ináč, ako sa plánovalo, alebo sú v systéme stále nejaké menšie chyby. Ale to je už iná téma.

Záver

V každom prípade počítanie LOC má zmysel. Nie ako „jediný správny“ ukazovateľ, ale ako jeden z množiny ukazovateľov. Odhad veľkosti projektu nemusí byť nutne správny (veď je to iba odhad). Naznačí nám, koľko by trvalo realizovanie priemerného projektu v oblasti, ktorú riešime. Máme aspoň približnú predstavu o tom, koľko to *môže* trvať a môžeme sa pokúsiť oprieť o iné metódy, aby sme sa priblížili skutočnej hodnote. Podobne je to pri meraní produktivity práce. LOC nám indikuje pokles alebo

vzrast produktivity, ale je na nás zistiť prečo vznikli tieto výkyvy a posúdiť, či sú relevantné.

Použitá literatúra

1. Albrecht, A.J.: Measuring application development productivity. In: *Proceedings of the IBM Application Development Symposium*, Monterey, CA (Oct. 1979) 83-92.
2. Armour, P.G.: The business of software: Beware of Counting LOC. *Communications of the ACM,* Vol. 47, No. 3 (2004) 21-24
3. Armour, P.G.: The business of software: Ten Unmyths of Project Estimation. *Communications of the ACM,* Vol. 45, No. 11 (2002) 15-18
4. Jones, C.T.: Backfiring: Converting lines of code to function points. *IEEE Computer 28,* Vol. 11 (Nov. 1995)

Annotation

Counting lines of code

In the present the method of counting lines of code is commonly used for time estimation (usually in the man-month unit) needed for a successful project completion. Also it is used for work production measurement of a programmer or a team. In this paper I describe the various ways to look at the counting of lines of code method. I focus mainly on the problems which arise by using this method the two areas I mentioned before. I explain in more detail the various approaches of how to count the lines of code and present a draft of alternative possibilities of measuring productivity. Each method has some disadvantages. That is why my goal in this paper is to describe a way to use this method considering its disadvantages.

Podvedomý prístup k testovaniu softvéru

PETER DUŠEK

Slovenská technická univerzita
Fakulta informatiky a informačných technológií
Ilkovičova 3, 842 16 Bratislava

Abstrakt. Testovanie je dôležitou fázou v etape vývoja softvéru. V tejto eseji opíšem vplyv ľudského podvedomia pri testovaní. Rozoberiem rozdiel medzi testovaním a inšpekciou, kedy a ako ich použiť. Už definícia procesu testovania býva často krát nesprávna, preto sa na tento pojem pozriem bližšie. Pri testovaní je vhodné dodržiavať určité princípy, ktoré sú vodítkom pre dosiahnutie požadovaných výsledkov. V článku ich popíšem podrobnejšie. Ďalej sa zameriam na roly v tíme testerov a ich zodpovednosť. Taktiež opíšem vhodnosť oddelenia vývojového a testovacieho prostredia.

Úvod

Testovanie je dôležitou fázou vývoja softvéru. Pán Myers [1] tvrdí že „...najdôležitejšie vplyvy na testovanie softvéru má psychológia a ekonomika“. Obzvlášť psychológia. Testovanie je výzvou, je viac o objavovaní ako iné disciplíny vo vývoji softvéru. Časy, kedy boli ľudia trestaní za objavené chyby sú už minulosťou a je snaha odhaliť ich čo najskôr.

Psychológia a ekonomika testovania programov

Jednou z primárnych príčin slabého testovania softvéru je fakt, že veľa programátorov začína s nesprávnou definíciou samotného testovania.
Vychádzajú z úvah:

- Testovanie je proces, ktorý preukáže, že v programe sa nevyskytujú chyby.
- Cieľom testovania je ukázať, že program vykonáva požadovanú funkciu správne.
- Testovanie je proces, ktorý vytvára presvedčenie, že program robí to, čo sa od neho očakáva.

Takéto definície sú ale nesprávne. Testovaním chceme softvéru pridať určitú hodnotu. Pridaná hodnota pomocou testovania znamená zvýšenie kvality a spoľahlivosti programov. Pričom zvyšovanie spoľahlivosti znamená hľadanie a odstraňovanie chýb.

Z toho dôvodu, netestujme programy, aby sme preukázali že pracujú, ale mali by sme vychádzať z predpokladu, že program obsahuje chyby (býva to pravdivý predpoklad takmer pre každý program) a pokúsiť sa nájsť z nich čo najviac.

Vhodnejšia definícia je potom:

Testovanie je proces vykonávania programu s úmyslom nájsť chyby.

Ľudia sú cieľavedomí, s úmyslom dosiahnuť čo si stanovia a stanovenie primeraných cieľov má významný psychologický efekt. Ak je naším cieľom preukázať, že program nemá chyby, potom budeme k tomu podvedome vedení. Môže to byť pri výbere testovacích dát a pravdepodobnosť, že objavíme chybu sa zníži.

Naopak, pokiaľ je naším cieľom ukázať, že softvér obsahuje chyby, je väčšia pravdepodobnosť, že naše testovacie dáta zistia chybu.

Iný spôsob ako upevniť definíciu testovania je analyzovať používanie slova „úspešný“ a „neúspešný“ najme v konkrétnych prípadoch kedy projektoví manažéri zaraďujú výsledky ich testovacích prípadov. Mnohí z nich volajú test, ktorý neodhalí chybu ako „úspešný test“ a tie ktoré odhalia nedostatok ako „neúspešný test“.

Skúsme si toto predstaviť na príklade kedy pacient navštívi svojho lekára s tým, že sa celkovo necíti dobre, je slabý a pravdepodobne je chorý.

Lekár vykoná niekoľko laboratórnych testov, ktoré nepotvrdia chorobu a neurčia diagnózu. Takéto testy by sme nenazvali „úspešné“, pretože boli neúspešné a výdavky na vykonanie testov mohli byť neprimerane vysoké. Pacient je naďalej chorý a môže ho zaujímať odbornosť lekára pri určovaní diagnózy. Ak testy niečo preukázali, boli úspešné a lekár môže začať s adekvátnou liečbou. Analogicky si môžeme pri testovaní predstaviť softvér v úlohe chorých pacientov.

Testovaním je prakticky nemožné ukázať, že chyba sa v programoch nevyskytuje, aj v prípade väčšiny triviálnych programov. Väčšie pokroky je možné očakávať v začiatočnej fáze testovania a čím je dlhšie proces testovania trvá, tým menšie pokroky možno pozorovať.

Aj podľa tretej uvedenej definície „Testovanie je proces, ktorý vytvára presvedčenie, že program robí to, čo sa od neho očakáva“, a softvér spĺňa požiadavky používateľa i napriek tomu môže obsahovať chyby. Tie sa môžu prejaviť neskôr v kritických situáciách.

Z iného pohľadu je testovanie hľadaním prípadov, kedy nie je splnená špecifikácia. Aktivity ako prehliadka, inšpekcia a statická analýza zdrojového kódu môžeme nazvať testovanie, kedy program nie je vykonávaný. Takéto aktivity môžeme nazvať statické testovanie.

Dynamické testovanie je počas vykonávania programu. Sem patrí funkcionálne testovanie, kedy zisťujeme či vstupno-výstupné správanie vyhovuje špecifikácii. Bez ohľadu na logiku systému môžeme zostaviť množinu testovacích vstupov. Štruktúrované testovanie je na základe vnútornej štruktúry, pričom ide o pokrytie tokov

riadenia aplikácie alebo o údaje. Ďalším je testovanie rozhraní jednotlivých podsystémov a overenie komunikácie medzi jednotlivými modulmi.

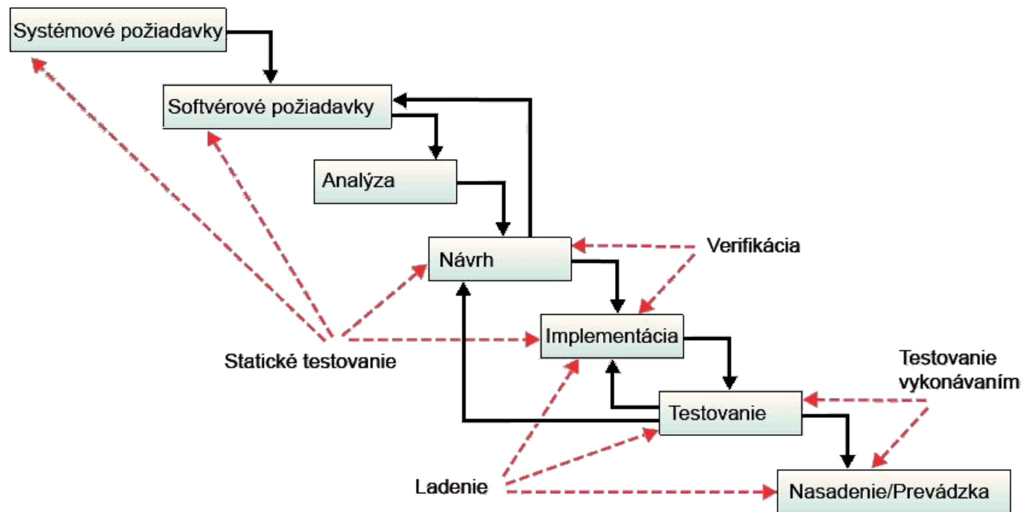
Životný cyklus a testeri

Testeri by mali veľmi dobre rozumieť produktu, aby boli schopní dobre navrhnúť testovacie plány, návrhy, procedúry a jednotlivé testovacie prípady [3]. Účasť testovacích tímov v skorších fázach vývoja eliminuje zmätok okolo funkcionálnych požiadaviek, ktorý by neskôr mohol vzniknúť. Rovnako testeri získajú prehľad o aspektoch, ktoré sú pre koncových používateľov kritické.

Na vodopádovom modeli životného cyklu softvérového projektu (pozri Obr. 1) si ukážeme do ktorých fáz zasahuje testovanie.

Cieľom ladenia je opraviť chybu v implementácii - uplatňuje sa počas samotnej implementácie, pri testovaní, počas prevádzky a údržby. Pri verifikácii ide hlavne o overenie funkcionálnej správnosti, ktorá závisí od kvalitného návrhu a implementácie.

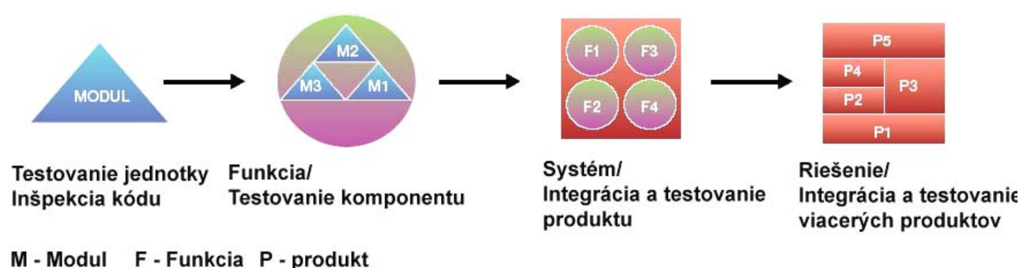
Testovanie v prvých fázach je statické, jedná sa o formálne prehliadky dokumentov a zdrojového kódu. Testovanie vykonávaním prihliada na aspekty ako bezpečnosť, výkon systému, požiadavky na hardvérovú a softvérovú konfiguráciu, zaťaženie systému a prípadné zotavenie. Typickým pri nasadení sú zákaznícke akceptačné kritériá.



Obr.1. Aktivity zahŕňajúce ladenie, testovanie a verifikáciu v typickom softvérovom vývojovom procese [3]

Rozsiahle a zložité systémy možno jemnejšie rozčleniť na podsystémy, komponenty a moduly, ktoré sú spojené v jeden celok. Na Obr. 2. je znázornené takéto rozčlenenie, ktoré nám určuje stratégie testovania. Testovanie v neskorších fázach na úrovni systému alebo riešenia je typu „čierna skrinka“ a nevyžaduje detailné znalosti o implementácii.

Pokiaľ nás zaujíma štruktúra programu ide o testovanie typu „biela skrinka“. Je možné testovať napríklad súdržnosť a zviazanosť jednotlivých modulov, komponentov.



Obr.2. Typické stratégie testovania na jednotlivých úrovniach

Testovanie a inšpekcia

Oboje, testovanie aj inšpekcia sú súčasťou verifikácie a validácie počas vývoja softvéru. V istých prípadoch plnia rovnakú úlohu, obe majú rovnaký cieľ, zvýšiť kvalitu produktu. Je veľmi dôležité odhaliť chyby čo najskôr, pretože často krát sa 40 až 50 percent vývoja venuje následnej oprave nedostatkov. Čo predstavuje veľké množstvo času ktoré možno získať. Inšpekcia je zameraná na odhaľovanie chýb, zatiaľ čo testovanie je zamerané hlavne na hľadanie zlyhaní [2].

Prečo používať inšpekciu

Inšpekcia dokumentov je dôležitá od začiatku projektu. Jej úlohou je nájsť všetky chyby v každej fáze procesu a tak pokračovať v ďalšej fáze so správnymi podkladmi.

Inšpekciu možno vykonávať mnohými rôznymi spôsobmi, rôznymi technikami čítania a podobne. Ak prídeme k hľadaniu chýb v zdrojových kódoch, môže byť ťažký výber medzi inšpekciou a testovaním. Správna voľba závisí od toho, čo hľadáme. Pomocou inšpekcie môže byť oveľa jednoduchšie odhaliť chyby v štandardoch programovania, než pomocou testovania. Rovnako v prípadoch ladenia programu, keď zisťujeme čo je zle. Podľa pána Sommervilla [4] je inšpekcia oveľa efektívnejšia. Spomína dva dôvody prečo je tomu tak. Po prvé, pri jednom sedení možno odhaliť veľa nedostatkov. Je to preto, že rôzne chyby často ovplyvnia vykonávanie programu a niektoré iné chyby sa potom neobjavia ak program testujeme.

Druhým dôvodom je skúsenosť inšpektorov v problémovej oblasti. Samozrejmosť je skúsenosť programovacieho jazyka a znalosť bežných chýb v zdrojových kódoch s ktorými sa už v minulosti stretli. Kompilátor striktne sleduje logickú postupnosť kódu, ale ľudia sa zameriavajú aj na obsah, ktorý sa môže vyskytnúť mimo normálneho vykonávania programu. Ale ľudia tiež robia chyby... Inšpekcia je lepšia pri hľadaní chýb v návrhu, v dokumentoch požiadaviek, v zdrojových kódoch a podobne. Záleží na znalostiach a predchádzajúcich skúsenostiach členov tímu a od ich celkových poznatkov z problémovej oblasti, inak je možné niektoré dôležité skutočnosti prehliadnuť.

Prečo používať testovanie

Testovanie môže v istých prípadoch predstavovať až polovicu času vývoja softvéru. Dôležitým faktorom je aj to, či je zákazník schopný akceptovať chyby, ktoré sú opravované počas používania vo fáze údržby, čím sa môžu často krát znížiť náklady na vývoj.

Testovanie je jediný spôsob ako odhaliť operačné zlyhania a ako overiť, že nefunkcionálne požiadavky sú splnené na požadovanej úrovni. Na rozdiel od inšpektorov, testeri sa väčšinou riadia jednotlivými testovacími prípadmi, ktoré by mali dodržiavať. Výhodou testovania je blízkosť s používaním po nasadení. Koncoví používatelia majú lepší dojem z produktu a cítia vyššiu kvalitu, pretože zlyhania sú mimo bežného používania, ktoré bolo testované. Produkt sa po vylúčení zlyhaní počas bežnej prevádzky stáva výhodnejším pre nasadenie. Často krát sa stáva, že fáza testovania má menšiu prioritu než ostatné. Ak sa projekt omešká, testovanie sa skracuje. Špeciálne ak je testovanie výlučne poslednou fázou a nie počas celého trvania projektu. Testovanie je niekedy jedinou možnosťou, ak napríklad nemáme prístup k zdrojovým kódom programov, využívame komponenty od tretích strán a rôzne API funkcie. Tak isto je potrebné vykonávať testovanie v rozdielnych prostrediach.

Vzhľadom na uvedené skutočnosti je lepšie vykonať najprv inšpekciu a potom testovanie. Samozrejme v praxi obľúbenejším prístupom býva práve opačný, najprv testovania a potom inšpekcia, ak vôbec.

Princípy testovania softvéru

Pri testovaní je vhodné dodržiavať určité zásady, ktoré prispejú k zvýšeniu kvality testovania. [4]

1. Nevyhnutnou časťou testovacích prípadov je definovanie očakávaných výstupov a výsledkov.
2. Programátor by sa mal vyhnúť pokusom testovať svoj vlastný program.

3. Organizácia zaoberajúca sa tvorbou softvéru, by nemala testovať svoje vlastné produkty.
4. Vykonávať dôkladne inšpekciu výsledkov každého testovania.
5. Testovacie prípady musia byť napísané pre vstupné podmienky, ktoré sú neplatné a neočakávané, tak isto ako pre platné a očakávané vstupy.
6. Preskúmanie či program vykonáva požadované úlohy je len polovičná práca, rovnako dôležité je zistiť, či nevykonáva to čo nemá.
7. Vyhnite sa jednoúčelovým testovacím prípadom, pokiaľ to nie je nevyhnutné, teda ak program tiež nie je jednoúčelový.
8. Neplánujte si úsilie testovať softvér s predpokladom, že chyby nebudú odhalené.
9. Pravdepodobnosť výskytu viacerých chýb vo funkčnom bloku programu je priamo úmerná počtu úž nájdených chýb v danom bloku.
10. Testovanie je veľmi kreatívnou a intelektuálnou výzvou.

Definovanie rolí a zodpovednosti testerov

Schopnosti testovacích tímov môžu výrazne ovplyvniť úspech prípadný neúspech testovania. V závislosti od problémovej oblasti je vhodné mať v tíme členov, ktorí detailne ovládajú túto oblasť. Tieto vedomosti im umožňujú efektívne vytvoriť testovacie prípady, skripty a iné mechanizmy pre overenie požiadaviek, funkčnosti, správnosti a iných kritérií softvérových produktov. Takéto tímy sa zväčša skladajú z niekoľko desiatok pracovníkov, ktorí majú svoje úlohy. Ich zodpovednosť vyplýva z ich pozície a schopností.

Manažér

Mal by rozumieť procesom a metodológiám testovania, byť schopný určiť ciele, objekty a stratégie. Tak isto rozumieť manuálnym a automatickým technikám testovania. Byť oboznámený s rôznymi podpornými nástrojmi, byť dobrý v plánovaní vrátane ľudských zdrojov a zariadení.

Vedúci tímu

Mal by rozumieť obchodným požiadavkám v danej oblasti a aplikačným požiadavkám, byť expertom v rôznych technických prostriedkoch, v programovacích jazykoch, databázových technológiách a operačných systémoch. Taktiež pokročilé znalosti a skúsenosti v procesoch testovania.

Tester

Mal by byť schopný v návrhu testovacích prípadov, profesionál v návrhu GUI štandardov, profesionál v jednotlivých fázach testovania softvéru.

Tester, odborník na siete

Mal by byť odborníkom na databázové a sieťové technológie, tak isto aj správu operačných systémov.

Tester, odborník na bezpečnosť

Mal by byť odborníkom na nástroje a techniky ohľadom počítačovej bezpečnosti.

Oddelenie vývojového a testovacieho prostredia

V niektorých prípadoch je vhodné oddeliť vývojové prostredie o testovacieho prostredia. Testovacie centrá bývajú špecializované a vedia vytvoriť niekoľko rôznych prostredí, či už rôzne platformy, konfigurácie zostáv, prípadne nasimulovať rôzne druhy záťaže systémov. Pre vývojárov je vhodné nechať otestovať svoje produkty na takýchto pracoviskách, kde pracuje tím profesionálov.

Záver

V tomto článku som sa pokúsil zhrnúť základné myšlienky o testovaní softvéru, napísal a zdôvodnil som definíciu testovania, ktorú by sme si mali uvedomiť pri každom testovaní. Ľudí je vhodné nasadiť na testovanie už v počiatočných fázach procesu vývoja, čo nám prinesie úsporu nákladov, času a zvýši kvalitu produktu. Testovanie a inšpekcia sa nevylučujú, je vhodné si uvedomiť ich vhodnosť použitia, pretože sa navzájom dopĺňajú.

Použitá literatúra

1. Phillip G. Armour: *The Unconscious Art of Software Testing*, COMMUNICATIONS OF THE ACM January 2005/Vol. 48, No.1
2. Gustav Evertsson: *Inspection vs. Testing*, Blekinge Institute of Technology, Software Verification and Validation (PAD001), 2002-11-20
3. Hailpern and Santhanam: Software debugging, testing, and verification. IBM Systems journal, vol 41, no 1,2002
4. Dustin, Elfriede: *Effective software testing : 50 specific ways to improve your testing* (Addison-Wesley) December 2002
5. I. Sommerville, Software Engineering, Excerpt: *Verification & Validation*, Adison-Wesley, 2000

Annotation

Subconscious approach to software testing

According to article in ACM I will to contribute to theme „*Subconscious approach to software testing*“. Testing is an important phase in the software development process. In this article I will inscribe impact of human been subconscious by testing. I will discuss the difference between testing and inspection and when to use them. Often the definition of testing is wrong, I'll try to formulate a proper one. It is good to respect some principles of testing, when performing it. Next I will focus on roles in testers team and their responsibility. Also I would like to describe suitability of development and testing environment separation.

Kto je zodpovedný za chyby a bezpečnostné slabiny v softvéri ?

PETER OROSI

Slovenská technická univerzita
Fakulta informatiky a informačných technológií
Ilkovičova 3, 842 16 Bratislava

Abstrakt. Či už chceme alebo nie, chyby sú neoddeliteľnou súčasťou softvéru. Preto sa zamyslíme nad príčinou vzniku týchto chýb a nad možnosťami ako im predchádzať. Spomenieme niektoré metódy manažmentu kvality softvéru, napr. inšpekciu alebo normy rodiny ISO 9000. Zamyslíme sa aj nad mierou, do akej sú softvérové spoločnosti zodpovedné za chyby vo svojich softvérových produktoch a do akej miery je im možné pripisovať právnu zodpovednosť za prípadné škody. Pritom vezmeme do úvahy aj druhú stranu mince, t.j. zlomyseľné kódy, ktoré sa úmyselne snažia o spôsobovanie škôd, najmä využívaním bezpečnostných slabín v softvéri. Na záver si stručne predstavíme základné koncepty bezpečného operačného systému EROS a zamyslíme sa nad jeho nasadením do reálnej prevádzky.

Úvod

Určite každý z nás pozná tzv. "modrú smrť" operačného systému Windows 98. Preslávená modrá obrazovka sa stala námetom už nejedného vtipu. V dobe najväčšej slávy tohto systému nejednému používateľovi privodzovala občas až infarktové stavy a vyvolávala lavínu expresívnych výrazov na stranu Microsoftu. Bol však Microsoft naozaj až taký zlý ? Je jasné, že chybám v softvéri sa nedá vyhnúť ani pri maximálnej opatrnosti. Pokým budú programy písať ľudia, budú tieto naďalej plné chýb. Rôzne zdroje udávajú rôzny počet chýb na 1000 riadkov zdrojového kódu. Ak by sa jednalo len o jednu chybu na 1000 riadkov, je jasné, že také zložité programové systémy, akými sú aj operačné systémy musia obsahovať stovky chýb. Nemenej zlé sú chyby v bežných aplikáciách. Určite nikoho neteší práca s nestabilnou aplikáciou, ktorá každú chvíľu „padá". Často sa vyskytujú aj chyby, ktoré síce zásadným spôsobom neohrozujú vykonávanie aplikácie, ale používateľovi dokážu znepríjemniť prácu. Jedná sa o chyby ktoré neboli odhalené pri testovaní produktu a musia byť odstránené dodatočne v rámci údržby, na základe podnetov zákazníkov. Osobitnou kapitolou sú škodlivé kódy ako vírusy, trójske kone, červy a podobne. Tieto sa väčšinou snažia zneužiť bezpečnostné

chyby v operačnom systéme na prienik a spôsobenie škody. Bez ich pričinenia by sme sa o týchto chybách nedozvedeli, pretože na prvý pohľad sa nám zdá, že systém funguje normálne. V nasledujúcich častiach preberieme problematiku chýb vo vývoji softvéru podrobnejšie. Z dôvodu obmedzeného rozsahu sa zameriam iba na niektoré časti tejto širokej problematiky.

Zdroje a charakterizácia chýb

Napriek niekoľko desaťročnej histórii tvorby softvéru, proces jeho vývoja stále nie je imúnny voči chybám. Zlepšili sa ale metódy na odhaľovanie chýb, prípadne na prevenciu voči chybám. Napr. existencia typových jazykov (Java), testovacie nástroje (xUNIT), statická analýza kódu atď. Zaujímavé však je, že počet chýb skúseného programátora a programátora začiatočníka je približne na rovnakej úrovni [2]. Samozrejme tu zrejme treba zohľadniť, že kód, na ktorom pracuje skúsený programátor je väčšinou oveľa komplikovanejší ako u začiatočníka, preto sú aj chyby ktoré urobí sofistikovanejšie a oveľa ťažšie rozpoznateľné. Takisto pri existencii testovacích nástrojov si musíme dať pozor, či tie samé v sebe neobsahujú chyby, pretože inak je výsledok našej námahy nepoužiteľný. Jedným z najčastejších problémov býva oprava chýb, kedy odstránením problému sa často zavedú nové chyby. Ak neotestujeme všetky funkcie v systéme po oprave chyby (testovanie vykonaných zmien na všetkých moduloch, tzv. regresné testovanie) riskujeme zavedenie nových chýb. Ak hovoríme o kvalite softvéru máme na mysli najmä mieru do akej softvér vyhovuje špecifikácia, ale aj s ohľadom na počet chýb. Manažment kvality má za cieľ dosiahnutie kvalitného produktu.

Tak ako poznamenal Mark Paulk z univerzity Carnegie Mellon [2]: „Základný problém s kvalitou softvéru je, že programátori robia chyby". Táto ľudská omylnosť sa výraznejšie prejavuje s rastúcou zložitosťou systému. Čím zložitejší systém, tým väčší počet chýb možno predpokladať. Užitočné, ale postrádateľné funkcie systému zapríčiňujú väčšinu z tejto zložitosti. Nie všetky chyby sa navonok prejavujú (tzv. latentné chyby). Chyby sa môžu vyskytnúť aj v skorších etapách životného cyklu ako pri implementácii, napr. nesprávnym pochopením požiadaviek zákazníka, alebo v dôsledku návrhu, ktorý je v rozpore so špecifikáciou a pod. Ak má byť ale projekt úspešne ukončený, tieto chyby musia byť vo výslednom projekte odstránené. Väčšina chýb vo výslednom produkte je dôsledok omylov programátorov (preklepy a iné chyby z nepozornosti).

Niektoré vlastnosti softvéru, aj keď nemôžu byť považované priamo za chyby, obsahujú predpoklady na vznik bezpečnostných slabín. Jedná sa napr. o populárny systém pluginov, podporovaný množstvom aplikácií ako prostriedku na rozšírenie funkcionality, prípadne používanie skriptovania v rámci aplikácie. V prípade ak je plugin alebo skript napísaný s úmyslom spôsobiť škodu, aplikácia ho nevie jednoducho rozlíšiť od užitočného obsahu a preto sa stáva zraniteľnou. Možné spôsoby ochrany sú napr.:

– Analýza – zamietnutie kódu ak tu je potenciál, že kód môže spôsobiť škodu.

- Prepísanie – modifikácia kódu pred spustením tak, aby nemohol byť škodlivý.
- Monitorovanie – monitorovanie kódu počas jeho vykonávania a následné zastavenie pri pokuse spáchať škodu.
- Auditovanie – kontrola program a ak spáchal nejakú škodu, vykonanie príslušnej akcie.

Otázka je, do akej miery sú chyby v softvéri škodlivé, resp. akceptovateľné. Napr. Richard Clarke, poradca prezidenta Busha sa vyjadril k otázke chýb nasledovne: „Už nemôže byť dlhšie akceptovateľné, aby sme si kúpili softvér, ktorý obsahuje defekty" [4]. V praxi je však význam slova akceptovateľný softvér úplne iný. Zväčša nie je v silách softvérových spoločností dodať svoje produkty v bezchybnom stave už od prvej verzie, skôr sa stalo dobrým zvykom zverejňovanie rôznych patchov a vychytanie najzávažnejších chýb až od verzie 2.0. Ekonomický prínos tohto zaužívaného postupu je výrazný. Na otázku, či je aj správny je však odpoveď skôr nie ako áno. Správny prístup je vývoj softvéru už od počiatku s cieľom systematicky predchádzať vzniku chýb. Nie je správne iba podľa potreby opravovať chyby, ale dôraz by sa mal klásť na pochopenie príčin týchto chýb a ich odstránenie (napr. pretečenie zásobníka)

Takisto si treba uvedomiť rozdiel medzi robustným softvérom a korektným softvérom. Softvér korektný vzhľadom na špecifikáciu vždy môže skončiť chybou, stačí ak napríklad používateľ zadá chybný vstup (taký, ktorý nezodpovedá špecifikácii).

Možnosti prevencie chýb a manažment kvality

Inšpekcia

Jedným z prostriedkov na zabezpečovanie kvality softvéru je aj inšpekcia. Hlavným cieľom inšpekcie je otestovať program do detailu, nezaujímame sa pri tom o samotný proces tvorby softvéru. Podľa skúseností odborníkov, aj keď formálna verifikácia produktu používa matematické požiadavky, ani tieto nemusia byť vždy schopné správne zachytiť návrhárov alebo zákazníkov úmysel. Inšpekcia dokumentácie požiadaviek zaručuje, že zachytené požiadavky zodpovedajú skutočnosti.

Takisto platí, že systémy, ktoré reagujú na množstvo okolných udalostí sú vo všeobecnosti menej dôveryhodné ako čisto sekvenčné systémy, pretože paralelizmus spracovania prináša so sebou určitú mieru nedeterminizmu. Typickým príkladom je operačný systém, kde by sme potrebovali otestovať všetky možné stavy, do ktorých sa možno dostať (čo v praxi predstavuje problém). Nedeterminizmus môže spôsobovať komplikácie aj z dôvodu, že rovnaký test môže v jednom prípade prejsť, a v inom prípade za zdanlivo rovnakých okolností (iba miernej zmeny stavu) rovnaký test zlyhá. Jedným z možných riešení je prispôsobenie návrhu požiadavkám testovania, napr. preto, aby bolo možné zabezpečiť časovanie podľa našich predstáv.

Osobitné problémy nás čakajú pri inšpekcii objektovo orientovaného kódu z dôvodu, že metódy v rámci rôznych objektov sa navzájom môžu volať komplikovaným spôsobom. Prieskumy ukázali, že zdroj veľkého počtu ťažko odhaliteľných chýb v OO kóde má svoj pôvod v chybách šírených v systéme, t.j. jedno volanie metódy na určitom mieste obsahuje chybu a táto chyba sa prešíri cez reťaz volaní ďalších pomocných metód, až do miesta kde spôsobí problémy. Autori v [5] túto vlastnosť nazvali delokalizácia. Jedným zo spôsobov ako sa pri inšpekcii vyhnúť delokalizácii je systematické štúdium kódu a testovanie, ktoré je však časovo náročne a pri rozsiahlych projektoch nereálne. Preto býva používaný skôr spôsob „podľa potreby", programátor sa zaoberá iba štúdiom kódu, ktorý považuje za opodstatnený, čo je však spojené s určitým rizikom. Metódy inšpekcie, ktoré môžeme použiť sú napr.:

- Technika riadená abstrakciou – vyžaduje od inšpektorov reverzným spôsobom vytvoriť abstraktnú špecifikáciu každej triedy a metódy, jedná sa vlastne o efektívne riešenie problému delokalizácie.
- Technika prípadov použitia – zaoberá sa kódom z dynamického hľadiska, zabezpečuje aby každý objekt reagoval korektne na všetky možné prípady svojho použitia. Dôraz sa kladie najmä na volanie správnych metód používateľom triedy a na to aby zmeny stavu objektu pri volaní každej metódy boli konzistentné
- Metóda kontrolného zoznamu – je založená na sérii špecifických otázok zameraných na typické zdroje chýb. Môže sa napríklad jednať o otázky: Ak sa vyžaduje volanie metódy rodiča v konštruktore, je toto volanie prítomne ? Sú všetky inštancie premenných inicializované na rozumnú hodnotu ?

Samotná inšpekcia zvyčajne nemá trvať dlhšie ako 2 hodiny, a počet preverovaných riadkov kódu by sa mal pohybovať okolo 200. Výsledným počtom odhalených a neodhalených chýb sa metóda zaraďuje medzi klasické testovacie postupy (viacej neodhalených chýb) a metódy formálnej verifikácie (minimum chýb).

ISO 9000

Ďalším z prostriedkov na zabezpečenie kvality je skupina noriem rodiny ISO 9000. Vychádza sa z predpokladu, že kvalitný proces zabezpečí kvalitný produkt. Jedná sa o všeobecne použiteľnú normu, v zásade nezávislú od oblasti nasadenia. Ako preukázateľný dôkaz, že spoločnosť spĺňa požiadavky na proces, slúži certifikát udelený certifikačnou inštitúciou. Tá je zase akreditovaná pomocou akreditačnej inštitúcie. Udelené certifikáty sú medzinárodne uznávané. Normy, ktoré si spoločnosť môže dať certifikovať sú ISO 9001 (ak organizácia vykonáva návrh, vývoj, inštaláciu a údržbu produktu), ISO 9002 (ako ISO 9001 ale bez etapy návrhu a vývoja), ISO 9003 (testovanie), ISO 9004 (zlepšovanie výkonnosti). Proces získania certifikátu je však veľmi zdĺhavý a vhodný iba pre dostatočne veľké spoločnosti. Základnou požiadavkou týchto noriem je pravidelné auditovanie, rozoznávame dva typy auditov:

- vykonaný externou certifikačnou inštitúciou

– vykonávaný interne v rámci spoločnosti vyškoleným pracovníkom

Cieľom auditov je nepretržitý proces zlepšovania kvality, overuje sa, či činnosti vzťahujúce sa na kvalitu a v súvislosti s ňou dosahované výsledky zodpovedajú ustanoveniam ako aj to, či sú tieto ustanovenia dostatočne vhodné. Stručne by sa dali činnosti auditora charakterizovať takto:

– Povedz mi čo robíš (popíš firemný proces)

– Povedz mi kde sa to spomína (odvolanie na manuál procesu)

– Dokáž, že je to to, čo sa vykonalo (predlož dôkaz v podobe dokumentácie, na kvalitu dokumentácie sa kladú vysoké nároky)

Napriek svojej komplikovanosti má norma svoje nesporné výhody, ak ku jej dodržiavaniu pristupujeme zodpovedne, je vysoká pravdepodobnosť, že výsledný produkt bude dobre zdokumentovaný, v súlade so špecifikáciou a vysoko kvalitný. Držiteľ certifikátu ISO 900x má takisto konkurenčnú výhodu v porovnaní so spoločnosťou bez certifikátu.

XP – extrémne programovanie

Zaujímavé sú aj metódy, ktoré zavádza extrémne programovanie (XP). Pri písaní kódu pomocou XP programátori najskôr venujú niekoľko minút príprave testov (napr. pomocou automatizovaných xUNIT testov) pre zatiaľ prázdne implementácie tried a vzápätí implementujú tieto metódy tak, aby prešli testami. Takýto postup, ktorý sa nazýva TDD (test driven development) zachováva návrh systému na najjednoduchšej možnej miere, pretože sa nezaťažujeme funkciami, ktoré možno bude treba v budúcnosti realizovať. Ku kvalite produktu prispievajú aj niektoré ďalšie zásady, napr. jeden programátor píše kód, druhý ho kontroluje, dodržiavanie pracovného režimu max. 40 hodín za týždeň (únava je tiež zdrojom chýb). Takisto proces vývoja býva nepretržite konzultovaný so zákazníkom, čím sa znižuje možnosť objavenia chýb až vo fáze implementácie (v porovnaní s klasickými softvérovými procesmi). Aj keď postupy XP nie sú vhodné pre každý typ projektov, v praxi ukazujú svoju opodstatnenosť. Výsledná kvalita a potrebný čas vývoja pre isté typy projektov môže byť vhodnejšia pri použití XP ako klasických metód vývoja softvéru.

Súvis medzi kvalitou, zložitosťou, spoľahlivosťou a prostriedkami na vývoj softvéru

Exitujú dve hlavné metódy zabezpečenia softvéru

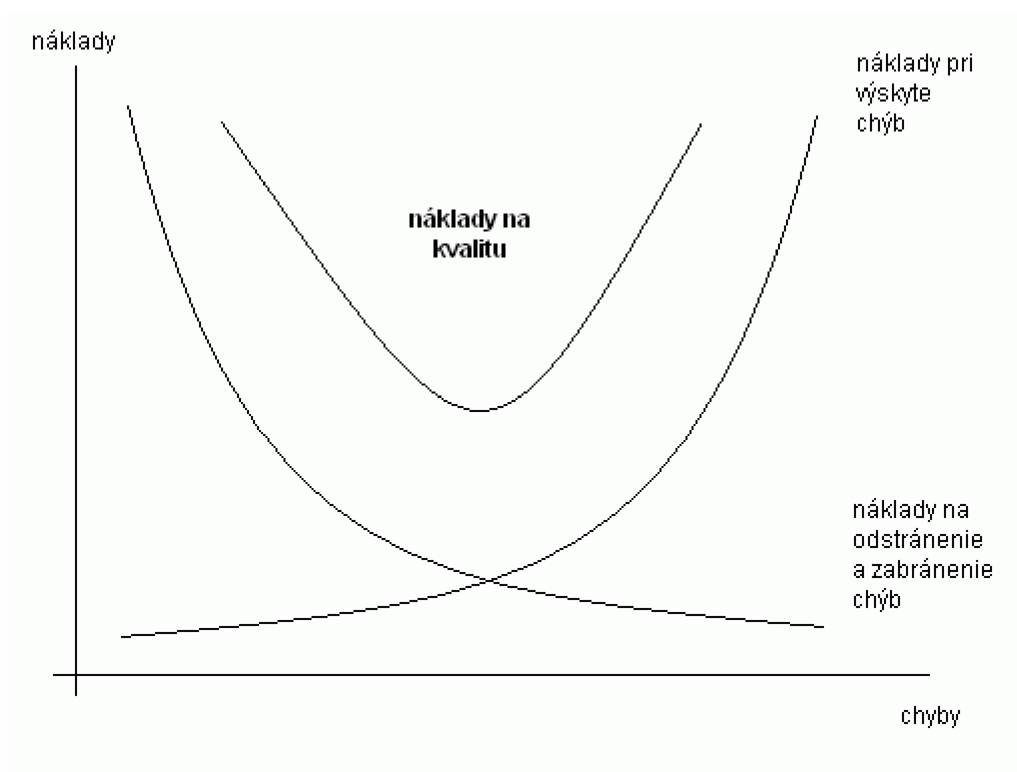
4. zabránenie chýb prostredníctvom formálnych prostriedkov špecifikácie a verifikácie
5. odolnosť voči chybám prostredníctvom diverzity, pod odolnosťou voči chybám rozumieme schopnosť počítačového systému pokračovať v činnosti aj po výskyte chýb

Prvý spôsob formálnych metód sa zatiaľ podarilo úspešne použiť iba na jednoduchšie systémy a jeho nevýhodou je aj veľká náročnosť.

Druhý spôsob pomocou diverzity ráta s viacerými implementáciami rovnakej časti systému. Jedná sa predovšetkým o metódu N-verzií a zotavovacích blokov. Vo všeobecnosti je metóda zotavovacích blokov účinnejšia, pretože stačí, aby bola jedna verzia rovnakej časti správna, kým pri N-verziách sa vyžaduje správnosť väčšiny častí.

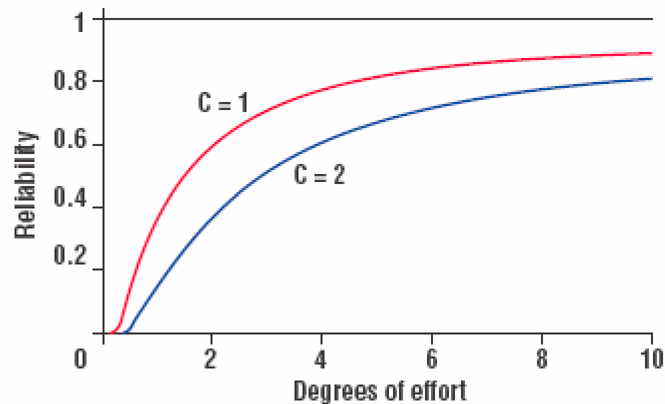
Pri moderných systémoch spôsobuje problémy znovupoužitie súčiastok, pokiaľ nemáme istotu, že súčiastky neobsahujú chyby. Rozdelenie zdrojov pre diverzitu môže viesť k zvýšenej ako aj zníženej spoľahlivosti, v závislosti od architektúry systému. Kľúč k zvýšeniu spoľahlivosti nie je v zvyšovaní stupňa diverzity (počet rôznych implementácií rovnakej veci), ale v existencii jednoduchých základných komponentov, ktoré zabezpečujú funkčnosť kritických častí aj v prípade zlyhania komponentov mimo jadra.

Očividne čím je systém zložitejší, tým ťažšie sa dá zaručiť jeho spoľahlivosť a tým viac je náchylný na chyby. Náklady na softvér sú obmedzené, preto s nimi treba narábať rozumne. Kvalitou softvéru rozumieme mieru splnenia požiadaviek a zároveň absencie chýb v produkte. Súvis medzi nákladmi, chybami, a kvalitou zachytáva obrázok 1.



Obr. 4 Súvis medzi nákladmi, chybami a kvalitou

Predpokladajme, že spoľahlivosť softvéru predstavuje exponenciálnu funkciu času *R(t)*, a ďalej predpokladajme, že početnosť zlyhaní je priamoúmerná zložitosti softvéru (C) a nepriamo úmerná úsiliu na vývoj (E). Vo výsledku sa spoľahlivosť R javí ako funkcia C a E, R = f(C, E). Pre najjednoduchší prípad bez použitia diverzity je situácia zobrazená na obrázku 2.



Obr. 5. Súvis medzi spoľahlivosťou (R) systému, vynaloženým úsilím (E) a zložitosťou softvéru (C).

Zodpovednosť za chyby

Softvéru sa často vyčíta jeho problematická „bezchybnosť". Vzhľadom na vysokú zložitosť softvéru býva prakticky nemožné objaviť všetky chyby ešte pred nasadením. Softvér, ktorý sa bežne využíva obsahuje chyby a používateľov tieto chyby niekedy môžu vyjsť draho. Odhaduje sa, že len v USA sa tieto chyby pohybujú každoročne na úrovni 50 miliárd dolárov. Vyvstáva preto otázka, či by softvérové spoločnosti nemali niesť zodpovednosť za svoje chybné produkty. Nedá mi nespomenúť známy vtip o autách, keby ich vyvíjal Microsoft. Autá, ktoré fungujú bezchybne len na 90% diaľnic, občas sa bezdôvodne zastavujú a na svojho majiteľa kladú rôzne nezmyselné požiadavky. Takéto autá vykazujú väčšinu nepekných vlastností, s ktorými sa môžme u (nielen) Microsoftu stretnúť a ich nasadenie do praxe si zrejme nevieme predstaviť. Jednoducho by si nikto nezobral na zodpovednosť prípadne dôsledky zlyhania.

Z pohľadu softvérových spoločností je však všetko v najlepšom poriadku. Stačí, keď si prečítame licenciu väčšiny programov. Takmer vždy objavíme magickú vetu o tom, že výrobca nezodpovedá za prípadné škody. V praxi častokrát najväčšie problémy spôsobujú latentné bezpečnostné chyby, ktoré sa na prvý pohľad nejavia ako chyby, len napr. umožňujú zlomyseľným kódom prienik do systému. Je snáď Microsoft aj iné firmy preto zodpovedný za svoje chyby ?

Ako ukazuje už obrázok 1, odstránenie všetkých chýb je možné iba za teoretického predpokladu (takmer) nekonečných nákladov. Preto sú v súčasnosti požiadavky na bezchybný softvér ďaleko od toho, čo možno reálne dosiahnuť. Bezchybnosť softvéru treba požadovať, ale iba pri absolútne kritických podmienkach nasadenia, napr. v prípade, ak softvér zodpovedá za riadenie kritických častí jadrovej elektrárne, prípadne riadenie lietadla. V podobných prípadoch je na mieste aj formálna verifikácia. Vo všetkých ostatných prípadoch sa jedná o kompromis medzi kvalitou a cenou.

Vzhľadom na veľkú zložitosť softvéru môžu spoločnosti niesť zodpovednosť iba do určitej miery, v niektorých prípadoch je to však dostatok na zaplatenie pokuty poškodenému zákazníkovi. Takéto prípady sa už reálne odohrali. Microsoft zatiaľ podobným súdnym sporom odoláva. A to aj napriek tomu, že práve chyby v jeho operačnom systéme umožňujú vykonávať autorom zlomyseľných kódov veľké škody. Situáciu možno prirovnať k zabezpečenému domu (bezpečný operačný systém) a jeho obyvateľovi (používateľ operačného systému). Autor zlomyseľného kódu (zlodej), by sa chcel vlámať do domu používateľa a spôsobiť mu škodu. Ak je dom dostatočne zabezpečený (napr. robustný plot s ostnatým drôtom a pod vysokým napätím okolo celého domu) bude zdolanie takejto prekážky zväčša nad zlodejove sily. Ak naopak obyvateľ nechá otvorené ešte aj vchodové dvere, s najväčšou pravdepodobnosťou bude vykradnutý. Analogicky platí, že operačný systém plný dier láka k zneužitiu. Aj keď hlavná zodpovednosť leží na zlodejovi, ponechávať systém nezabezpečený je nerozvážne. Preto aj keď pisateľ vírusov nesie priamu zodpovednosť za spôsobené škody, výrobca operačného systému takisto nesie určitý stupeň nepriamej zodpovednosti. Keď sa teraz pokúsim odpovedať, či by mal byť za túto nepriamu zodpovednosť výrobca operačného systému právne postihnuteľný, moja odpoveď je nie, pravdepodobne by nemal, pretože je v súčasnosti nereálne očakávať úplne bezchybný softvér a hlavné bremeno zodpovednosti je aj tak na autoroch zlomyseľných kódov.

Aj keď sa často zdôrazňuje výhodnosť iných, alternatívnych operačných systémov na báze Linux, ani tieto systémy nie sú voči chybám imúnne. Jedným z možných dôvodov menšej chybovosti týchto systémov je zrejme aj ich nižšia rozšírenosť a potom aj iná filozofia bezpečnosti. To, že je softvér dodávaný aj s voľnými zdrojovými kódmi, však ešte neznamená, že musí byť aj kvalitnejší. Napr. v zdroji [6] sa uvádza: „Je naozaj otázne, aj z pohľadu spoľahlivosti, či mať väčší počet inšpektorov iba s bežnými znalosťami, dáva lepší výsledok“. To na čo sa tu naráža, je aj kvalita inšpektorov ako aj fakt, že nie všetci, ktorí pracujú s takýmito voľnými zdrojovými kódmi, nutne hľadajú v nich chyby. Pravdepodobne výsledný kód bude mať menej očividných chýb, ale vo výsledku aj tak veľa chýb ostáva neodhalených. Hlavná výhoda open-source je však istota, že do kódu nebol primiešaný žiadny trójsky kôň prípadne iné škodlivé a postranné činnosti. A nemenej významný je aj spôsob opravovania chýb. Ak sa nejaká chyba objaví, zväčša odstránenie chyby je otázkou hodín, alebo dní. Oproti klasickému prístupu softvérových spoločností je tento spôsob pružnejší.

Pri hľadaní zdrojov som našiel zaujímavý článok [7], ktorý sa zaoberá tvorbou bezpečného operačného systému s názvom EROS (Extremely Reliable Operating

System). Zatiaľ sa jedná o projekt čisto na akademickej pôde, avšak svojimi výsledkami predstavuje do budúcnosti minimálne zaujímavú alternatívu k existujúcim operačným systémom (Windows, Linux). Jeho hlavná črta je jednoduchá, vnútorne konzistentná architektúra, kedykoľvek je nejaká aktivita v rozpore s prísnymi pravidlami, je požiadavka na túto aktivitu zamietnutá. Takisto bola vykonaná formálna verifikácia jeho kritických bezpečnostných častí. Medzi jeho ďalšie vlastnosti patria:

- bezpečný reštart, nemožnosť vzniku inkonzistencií v súborovom systéme
- bezstavový kernel , stav vykonávania sa cache-uje z priestoru alokovaného používateľom, zabezpečuje sa konzistencia cache so stavom v používateľskom priestore
- štruktúra aplikácií, aplikácie sú stavané z ako skupina spolupracujúcich jednoduchých komponentov, pričom každý z komponentov (aj v rámci jednej aplikácie) beží pod inými prístupovými oprávneniami (napr. právo zápisu na disk) - schopnosťami. Tieto schopnosti sú chránené v rámci kernelu. Ak napríklad vírus dosiahne kontrolu nad jedným z týchto komponentov, škody ktoré môže spôsobiť sú minimálne, pretože nemajú dostatok oprávnení na ovplyvnenie systému ako celku (na rozdiel od klasických operačných systémov). Návrh aplikácii ako malých jednoduchých komponentov uľahčuje testovanie. Pomocou dobre zvolených testov je zväčša možné otestovať všetky stavy komponentu, takisto kompilátor sa stará predchádzanie rôznych pretečení zásobníkov.
- adaptácia aplikácií, v súčasnosti prebiehajú snahy prepísať bezpečným spôsobom na túto platformu podporu pre najdôležitejšie protokoly HTTP, SMTP, FTP atď. (jedná sa o rozhrania vykonávané s vysokým stupňom právomocí), neskôr sú naplánované aplikácie, ktoré spúšťajú rôzne skriptovacie jazyky (prehliadače, word procesory atď.). Plánuje sa zviesť aj emulácia Linuxu

Ak by sa tento systém niekedy v budúcnosti dočkal reálneho nasadenia aj širokej programátorskej podpory pri tvorbe aplikácií, bol by to významný krok v pred na poli softvérovej bezpečnosti.

Záver

Chyby sú neoddeliteľnou súčasťou softvéru, bolo tomu tak v minulosti a určite to tak bude aj v blízkej budúcnosti. Určitú nádej v boji proti chybám predstavujú rôzne metódy manažmentu kvality, ako aj precízne metódy tvorby softvéru, ani tieto však nepredstavujú úplne riešenie problému. Vždy sa nájdu chyby, ktoré zostanú neodhalené. Určitú nádej na zmenu tohto stavu predstavujú systémy ako EROS, operačný systém novej generácie, kde je bezpečnosť kladená nekompromisne na prvé miesto. To či sa niekedy stanú podobné operačné systémy a rovnako aj aplikácie pre tieto operačné systémy bežnou praxou alebo sa jedná iba o naivnú víziu sa ukáže až v budúcnosti. V každom prípade by to bol dobrý začiatok.

Použitá literatúra

1. Michael A. Cusumano: Who is liable for bugs and security flaws in software? *Communications of the ACM*, Vol. 47, No. 3 (March 2004), 25-27
2. Gerard J. Holzmann: The Logic of Bugs. *ACM SIGSOFT Software Engineering Notes*, Vol. 27, No. 6 (November 2002), 81-87
3. Bieliková, M.: Softvérové inžinierstvo – Princípy a manažment, 2000
4. Tish Keefe: Software Insecurity**.** *Computerworld* (August 2002), No. 5
5. Alastair Dunsmore, Marc Roper, Murray Wood: Practical Code Inspection Techniques for Object-Oriented Systems: An Experimental Comparison. *IEEE Software*, Vol. 20, No. 4 (August 2003), 21-29
6. Ross Anderson, Terry Bollinger, Dowg Brown, Erique Draler, Philip Machankk, Gary McGraw, Art Pyster, Howard Schidt, Tim Shimeall, Nancy Mead: Information Security Policy. *IEEE Software*, Vol. 17, No.5 (September 2000), 26-32
7. Jonathan S. Sharpio, Norm Hardy: EROS: A Principle-Driven Operating System from the Ground Up. *IEEE Software*, Vol. 19, No. 1 (January 2002), 26 - 33

Annotation

Who is liable for bugs and security flaws in software ?

Regardless we want or not, erros are inherent part of sotware. That's why we discuss causes of these errors and the possibilities how to prevent them. We remark few methods of software quality management, e.g. inspection or ISO 9000 family standards. We consider extent of responsibilty by software companies by which they should be held liable for bugs in their own software. We also take into our consideration the reverse site of the coin, especially malicious codes which are intentionally trying to cause damage by using security flaws in software. Finally we notice basic concepts of secure operating system EROS and we contemplate its application in real operation.

Záver

Dúfame, že ste sa z tejto publikácie dozvedeli o súčasných problémoch softvérového inžinierstva. Teší nás, ak bola pre vás prínosom. Ďalšie štúdium problematiky môžete začať napríklad pri zdrojoch, ktoré uviedli autori. Ak ste tak už urobili, knižka splnila svoj cieľ.

Názov	Eseje o manažmente v softvérovom inžinierstve
Editori a grafická úprava	Bc. Rastislav Bertušek Bc. Marek Fučila
Prispievatelia	Bc. Rastislav Bertušek Bc. Peter Dušek Bc. Marek Fučila Bc. Pavol Goňo Bc. Martin Hinka Bc. Martin Jenčo Bc. Ľuboš Lečko Bc. Michal Moravčík Bc. Martin Niejadlik Bc. Peter Orosi Bc. Ján Šnirc Bc. Bohuslav Szabo Bc. Peter Trnovský Bc. Miroslav Vnuk Bc. Milan Žužo
Vydané	Ilkovičova 3, FIIT STU v Bratislave
Vydanie	Prvé
Náklad	1 výtlačok
Počet strán	136
Rok vydania	2005