DOPAD AGILNÉHO VÝVOJA NA KVALITU SOFTVÉRU

Nič nie je dokonalé.

Jakub Calík

Slovenská technická univerzita
Fakulta informatiky a informačných technológií
Ilkovičova 3, 842 16 Bratislava
xcalikj[zavináč]is[.]stuba[.]com

Abstrakt. *Každý softvérový vývojár sa snaží vytvoriť kvalitný softvér. Kvalita však nie je niečo, čo sa dá jednoducho odmerať počtom riadkov alebo podobne. Poznáme viacero aspektov, ktorých súhra a vyváženosť určujú celkovú kvalitu softvéru. Aby sa však na tieto aspekty nezabúdalo, a aby výsledkom vývoja bol naozaj kvalitný softvérový produkt, vzniklo množstvo metód a modelov pre vývoj softvéru [5]. Nie všetky metódy a postupy vývoja softvéru však kladú rovnaký dôraz na všetky aspekty kvality. Niektoré môžu vyzdvihovať až príliš, zatiaľ čo iné úplne zanedbávať. V poslednej dobe sa však stáva veľmi populárny agilný model vývoja, V tejto eseji sa preto budem snažiť objasniť, ako sa agilný vývoj softvéru postavil k aspektom kvality a ako by sa dali vylepšiť jeho nedostatky.*

Kľúčové slová: *softvér, kvalita, prenositeľnosť, výkonnosť, kompatibilita, metodológie vývoja, agilný vývoj, extrémne programovanie, párové programovanie*

Agilný vývoj softvéru

Agilný prístup k vývoju softvéru sa vyznačuje vysokou mierou prispôsobiteľnosti. Či už sa to týka samotných vývojových tímov alebo schopnosti rýchlo a efektívne reagovať na zmenu požiadaviek [1].

Vývojové tímy by mali byť schopné fungovať ako samostatné jednotky schopné vyriešiť a zrealizovať ľubovoľnú požiadavku od používateľa. Na základe toho musia byť zložené z rôzne zameraných ľudí (programátor, tester, analytik...).

Celý vývoj softvéru je prispôsobený tak, aby bolo možné do riešeného problému zapracovať zmeny čo najjednoduchšie. Taktiež snaha dodať funkčný softvér za čo najkratší čas a uspokojiť požiadavky zákazníka majú za dôsledok, že niektorým veciam sa vo vývoji kladie menší dôraz a na niektoré sa úplne zabúda. Je to napríklad tvorba dôkladných špecifikácií a písanie dokumentácie. Nedostatočná špecifikácia a neustále sa meniace názory a požiadavky koncového zákazníka však môžu znamenať nemalé problémy, hlavne v neskorších etapách vývoja projektu.

Na začiatku vývoja je nevyhnutné venovať dostatok času požiadavkám. Požiadavky vznikajú a menia sa aj počas vývoja softvéru. To je jedným z hlavných princípov agilného vývoja [1]. Vďaka tomu však môžu vzniknúť dve riziká, s ktorými treba počítať a predchádzať im. Prvé je vznik nekonečného projektu, kedy zákazník sám nevie čo chce a neustále mení požiadavky alebo pridáva stále nové a nové. Druhým rizikom takéhoto vývoja je ťažšia predvídateľnosť hneď na začiatku o tom, koľko čo bude trvať a čo sa vlastne zákazníkovi dodá na konci vývoja. Toto môže vyústiť do zlého odhadu pevnej ceny projektu a zle stanovenej lehoty dodania softvéru.

Na začiatku vývoja samozrejme existuje vízia zákazníka, ktorý má určitú predstavu o softvéri a o tom, čo všetko musí spĺňať. Zákazník svoju víziu zdieľa s vývojovým tímom, ktorý ju bude realizovať.

Vývojový tím samotný však nie je dostačujúci a do procesu by mal byť zapojený aj expert na oblasť problematiky. Hlavnou úlohou tohto experta je pochopiť a zanalyzovať zákazníkovu víziu. Musí tiež určiť nakoľko sú zákazníkove požiadavky akceptovateľné a aplikovateľné v rámci danej problematiky. Cieľom jeho analýzy je určiť medzery a nejasnosti, ktoré by mohli predstavovať problém v logike softvéru.

Po takejto kontrole realizovateľnosti zákazníkových požiadaviek je nevyhnutné zdokumentovať požiadavky, aby sa zachovala referencia pre budúci vývoj. Už v tomto počiatočnom štádiu vývoja je potrebná spätná väzba od zákazníka, ktorý musí objasniť všetky nezrovnalosti, prípadne, ak je to nevyhnutné na základe analýzy experta alebo odporúčania členov tímu, upraviť svoje požiadavky.

Táto analýza bude kľúčová pre ďalší vývoj. Aj keď sa vo vývoji počíta so zmenou požiadaviek, táto zmena by sa stále mala držať na začiatku dohodnutých požiadaviek. Zásadná odchýlka od týchto požiadaviek by sa už počas vývoja softvéru nemala vyskytnúť. Inak hrozí, že aj napriek maximálnej flexibilnosti vývojových tímov bude nutné prepracovať základné piliere softvéru, čo bude mať za následok neočakávané natiahnutie vývoja a z toho vyplývajúce predraženie celého softvéru.

Znovupoužiteľnosť a rozšíriteľnosť

Rozšíriteľnosť softvéru o nové moduly a znovupoužitie jednotlivých modulov patria k základným aspektom objektovo orientovaného programovania. Použitý model vývoja však ovplyvňuje, ako ľahko sa vývoj prispôsobí novým požiadavkám a aké následky má zapracovanie zmeny do projektu.

Pri agilnom vývoji sa postupuje po krátkych iteračných cykloch, ktoré sú vždy zamerané na 100% splnenie nejakej konkrétnej požiadavky [4]. V spojení s nedostatočnou dokumentáciou a snahou vytvoriť funkčný produkt v čo najkratšom čase môže vzniknúť problém, že vývojový tím stratí komplexný náhľad nad celým softvérom. A znovu sme sa

dostali k dokumentácii, ktorá je dôležitá a vývojový tím by preto mal zvážiť, či by nebolo vhodné v agilnom vývoji použiť štandardnú dokumentáciu.

Samozrejme sa k tomu pridáva aj priebežná zmena požiadaviek, čo ešte prispieva k nutnosti pravidelne refaktorovať kód a prispôsobovať ho tak novým požiadavkám. Refaktoring by však s výnimkou nových požiadaviek, nemal byť hlavným nástrojom ako opravovať staré chyby, ktorým sa dalo predísť napríklad odlišným rozhodnutím.

Aby sme teda zredukovali nutnosť refaktorovať každý jeden modul vo vyvíjanom systéme, je potrebné, aby vo vývojovom tíme bol zahrnutý niekto, kto si dokáže udržať komplexný náhľad nad celým projektom a je schopný prijímať dôležité rozhodnutia už zo začiatku vývoja softvéru. Takýmto človekom môže byť skúsený softvérový návrhár alebo programátor, ktorý už má skúsenosti s podobnými projektmi a dokáže spoľahlivo viesť aj menej skúsených tímových kolegov.

K udržaniu nadhľadu nad celým softvérom môže tiež prispieť dokumentácia a kvalitný návrh. Pri agilnom vývoji sa na ňu často nekladie taký dôraz a jednotlivé cykly sa iterujú s čo najnižšími požiadavkami. Vývoj sa pri tom spolieha hlavne na komunikáciu so zákazníkom. Zavedenie štandardnej komunikácie do agilného vývoja však môže pozitívne prispieť k vývoju tým, že vývojári by dopredu vedeli čo majú robiť a čo budú musieť robiť. Taktiež nie je dobre obracať sa s každým drobným problémom na zákazníka, ktorého môže prílišné zaintegrovanie do projektu rýchlo omrzieť [3].

Testovanie

Testovanie môžeme označiť ako základný nástroj na zabezpečenie robustnosti a správnosti aplikácie. Zatiaľ čo unit testy kontrolujú funkčnosť jednotlivých modulov aplikácie, akceptačné testy ponúkajú komplexnejší prehľad a kontrolujú, či softvér zodpovedá špecifikácii a požiadavkám zákazníka.

Pri klasických metódach vývoja prebieha testovacia fáza nasledovne: Manažér testovania vytvorí plán testovania. Začnú sa znovu prechádzať požiadavky a špecifikácia. Testeri začnú písať testovacie skripty pre jednotlivé prípady testovania. Následne sa čaká na dokončenie softvéru, často až kým je príliš neskoro na dôkladné otestovanie. Po otestovaní sa začínajú riešiť problémy objavené počas testovania.

Problémom pri takomto vývoji je, že testeri sú zapájaní do vývoja projektu až tesne pre ukončením vývoja (vodopádový model), prípadne pred ukončením jednotlivých iterácií (špirálový model). Agilné metódy vývoja softvéru sa to snažia zlepšiť tým, že testeri sú súčasťou vývojového tímu po celú dobu vývoja a úzko spolupracujú s celým tímom [2].

Pri test-driven development sa testovanie dostáva do čela a tvorí základy pre celý vývoj projektu. Testy sú vytvárané ešte pred začiatkom implementácie samotnej funkcionality. Toto umožňuje implementovať tak, aby funkcionalita vyhovovala testom. Môže sa tak značne znížiť potreba prerábať kód.

V klasických metódach vývoja je oficiálne každý zodpovedný za kvalitu. Keď sa však niečo pokazí (softvér nevyhovuje špecifikácii, neobjavené chyby), vzniká problém, kto konkrétne je zodpovedný. Návrhár, ktorý nepočítal s komplexnou funkcionalitou? Vývojár, ktorý naimplementoval málo rozšíriteľný kód? Alebo tester, ktorý vytvoril nedostatočné testy.

Pri agilnom vývoji však testy vytvára a vykonáva celý tím. Vývojár, analytik, dokonca aj zákazník. Prenáša sa tak zodpovednosť za požadovanú funkcionalitu z testerov na celý tím. Keďže každý vývojár vytvára testy, zvyšuje sa celková testovateľnosť softvéru.

Dá sa teda povedať, že testovanie predstavuje najsilnejšiu stránku agilného vývoja softvéru. Nie všetky požiadavky na softvér sa však musia vzťahovať na funkcionálnu stránku softvéru a dajú sa ľahko otestovať.

Prenositeľnosť

Pri agilnom vývoji, hlavne pri extrémnom programovaní, sa využíva rýchle prototypovanie a následná priebežná integrácia vyvíjaného softvéru. Problémy s kompatibilitou so softvérom či hardvérom sa tak objavia omnoho skôr ako pri iných metódach vývoja softvéru. Nestačí však vedieť o problémoch a rizikách, ale treba ich identifikovať dopredu a vyvíjať softvér tak, aby nevznikali.

Požiadavka na prenositeľnosť softvéru však nie je priamo zadeklarovaná v manifeste agilného vývoja [1]. Napriek tomu na ňu softvérový vývojári musia myslieť, obzvlášť, ak má zákazník špecifické požiadavky na cieľové platformy softvéru. Dobrým prístupom je zapracovať požiadavky na portabilitu už pri počiatočnej analýze softvéru.

Ak teda vývojári a softvérový dizajnéri nezanedbajú tento aspekt a myslí sa na neho od začiatku vývoja, prenositeľnosť môže byť ďalšou výhodou agilného vývoja. Hlavne vďaka včasnej integrácii je potom možné otestovať už naimplementované požiadavky súvisiace s prenositeľnosťou a nie je ďalej potrebné vykonávať väčšie zmeny, prípadne prerábať celý systém.

Avšak aj softvéroví vývojári sú len ľudia. Ako teda zabezpečiť, aby na požiadavku prenositeľnosti mysleli, aj keď nie je explicitne zadefinovaná v zákazníkových požiadavkách?

Jedným z možných prístupov môže byť, že vývojový tím si zaužíva sám vytvoriť požiadavky súvisiace s prenositeľnosťou v každom vyvíjanom projekte. Vytvorené požiadavky taktiež musia naplánovať do jedného zo šprintov. Do ktorého šprintu ich zahrnú už závisí od samotného projektu a vývojového tímu. Malo by to však byť už na začiatku vývoja, najlepšie už počas implementácie prvého prototypu softvéru.

Výkonnosť

Vďaka skorému prototypovaniu a včasnej integrácii je výkonnosť ďalší aspekt, ktorý je možný zmerať a následne aplikovať potrebné zmeny. Rýchle prototypovanie v agilnom vývoji však môže mať aj negatívny efekt na výsledný softvér.

Samozrejme, každý softvérový vývojár by mal byť schopný písať čo najefektívnejší kód a to bez ohľadu na to, či vyvíjaný softvér pobeží na superpočítači alebo menej výkonných mobilných zariadeniach. V extrémnom programovaní sa však, pri snahe čo najrýchlejšie naimplementovať funkčný prototyp s minimálnymi požiadavkami, môže zanedbať efektivita naimplementovaných algoritmov. Netreba preto zanedbať návrh a potrebu vychádzať pri návrhu z efektívnych algoritmov.

Z praxe je však známe, že návrhári si často nerobia svoju prácu tak dôsledne, ako by mali. Pri návrhu architektúry sa často nezamýšľajú podrobne nad konkrétnymi algoritmami a táto zodpovednosť napokon ostane na programátoroch.

Tento problém môže pomôcť vyriešiť tzv. pair-programming (programovanie vo dvojiciach, „párové programovanie“). Aj keď je tento prístup nákladný na ľudské zdroje, myšlienka, že dvaja programátori riešia spoločne jeden problém a navzájom si vymieňajú nápady a skúsenosti, môže v konečnom dôsledku ušetriť ďaleko viac prostriedkov. Samozrejme, len za predpokladu, že sú programátori priraďovaní do dvojíc rozumným spôsobom (skúsený programátor s menej skúseným a nie dvaja úplní nováčikovia pohromade), čo nebýva ľahká úloha.

Dvojica programátorov funguje tak, že zatiaľ čo jeden programátor implementuje algoritmus, druhý programátor nad ním vykonáva dohľad, či algoritmy implementuje podľa dohodnutých vzorov a používa dohodnuté nástroje a knižnice. Okrem takéhoto dohľadu má čas zamýšľať sa nad ďalšími algoritmami, možnými rizikami a novými úlohami. Spolupráca dvoch programátorov na jednej spoločnej úlohe tak môže ušetriť značné množstvo času, ktorý by bol inak potrebný na refaktoring a prerábanie kódu.

Tento spôsob vývoja sa používa napríklad pri extrémnom programovaní a iné agilné metódy, ako napríklad SCRUM alebo Kanban, ho nevyužívajú. Napriek tomu si myslím, že možnosti a výhody, ktoré pair-programming prináša, môžu obohatiť a aj zlepšiť tieto metódy.

Riešením, menej náročným na ľudské zdroje, môže byť priradenie aspoň jedného experta (skúseného vývojára) do každého tímu. Úlohou experta bude dohliadnuť na menej skúsených tímových kolegov a zabezpečiť, aby sa používali dopredu dohodnuté algoritmy a nevytvárali zbytočný alebo neefektívny kód. Problémom však je, že schopných ľudí je všade málo a teda, nemôžeme sa spoľahnúť na to, že každý tím bude mať k dispozícii takého experta.

Aj napriek tomu, že niektoré nedostatky systému je možné odhaliť pomocou záťažových testov, väčšina slabín sa prejaví až po nasadení systému v reálnych podmienkach. Tu má zas agilný vývoj jasnú výhodu v skorej integrácii prototypov a včasnom zapracovaní zmien a opráv.

Použiteľnosť

Pre koncového používateľa má najväčší význam použiteľnosť softvéru. Vývojári a softvéroví návrhári by taktiež mali dopredu myslieť na naučiteľnosť a čo najintuitívnejšiu použiteľnosť softvéru a navrhovať jednotlivé komponenty tak, aby ich dokázal použiť aj najmenej kvalifikovaný používateľ v organizácii. Vzhľadom k tomu, že pri agilnom vývoji je zákazník súčasťou vývojového tímu, je schopný podávať spätnú väzbu o vyvíjanom produkte priebežne priamo vývojárom.

Záver

Agilný model vývoja softvéru, aj napriek svojej obľúbenosti a veľkému množstvu predností, nie je dokonalý. Keď sa na neho pozeráme z pohľadu kvality a hodnotíme, ako pristupuje k jednotlivým aspektom kvality, môžeme vidieť nedostatky a vznikajúce riziká

skoro pri každom aspekte. Vzhľadom na obmedzený rozsah tejto práce som nemohol popísať všetky aspekty. Zanalyzoval som preto len vyššie spomenuté, pri ktorých som si všimol väčšie nedostatky. Tieto nedostatky sa však dajú zlepšiť, prípadne úplne odstrániť, napríklad pomocou mnou spomínaných návrhov, ako napríklad štandardnou dokumentáciou a párovým programovaním. Vývojový tím, ktorý sa rozhodne prejsť na agilný model vývoja, by však rozhodne nemal zanedbať tieto aspekty.

Použitá literatúra

1. Mnkandla, E., Dwolatzky, B.: "Agile Software Methods: State-of-the-Art." Agile Software Development Quality Assurance. IGI Global, 2007. 1-22. Web. 15 Oct. 2012. doi:10.4018/978-1-59904-216-9.ch001
2. Shubhangi, Anand N., Manisha V. and G V Garje. Article: Improving Software Quality with Agile Testing. International Journal of Computer Applications 1(22):66–71, February 2010.
3. Ming Huo, June Verner, Liming Zhu, Muhammad Ali Babar, "Software Quality and Agile Methods," compsac, vol. 1, pp.520-525, 28th Annual International Computer Software and Applications Conference (COMPSAC'04), 2004
4. Abrantes, J.F., Travassos, G.H., "Common Agile Practices in Software Processes", Empirical Software Engineering and Measurement (ESEM), 2011 International Symposium on, On page(s): 355 – 358
5. Govardhan, A.: A Comparison Between Five Models Of Software Engineering, International Journal of Computer Science Issues, Vol. 7, Issue 5, 2010
6. William R. Duncan: A guide to the project management body of knowledge, Project Management Institution, 2004Annotation

Annotation

Agile software development impact on software quality

Every software developer is trying to create software with best quality. Quality is not something that can be easily measured by the number of lines of code. There are several aspects that determine balance of the overall quality of software. Several methodologies and models for software development was made in order to create software product of good quality. However, not all of the methods and procedures of software development, place equal emphasis on all aspects of quality. While some aspects can have more weight in certain model, others could be put asside completly. Lately, become very popular agile development model. In this essay, I will try to explain how agile software development influence software quality aspects and how it can be improved.