KYKLOP

Tvorba softvéru v tretom tisícrocí

SLOVENSKÁ TECHNICKÁ UNIVERZITA

BRATISLAVA 2002

Bc. Michal Bigoš
Bc. Peter Diko
Bc. Vladimír Grlický
Bc. Ladislav Gyökeres
Bc. Rastislav Habala
Bc. Jozef Hudec
Bc. Richard Krupa
Bc. Jaroslav Kuruc
Bc. Vladimír Marko
Bc. Martin Šindelar

Tvorba softvéru v tretom tisícrocí

© KYKLOP 2002

Publikácia neprešla jazykovou úpravou.

Predhovor

Vladimír Marko

Softvérové inžinierstvo ako pomerne mladá profesia celí mnohým tlakom, ktoré prináša trh a neustále sa zrýchlujúci výrobný cyklus v tejto oblasti. Zároveň je hospodárstvo najvyspelejších štátov závislé na výsledkoch dosahovaných softvérovými inžiniermi ako na máloktorej inej profesii. Táto závislosť vytvára tlak, ktorý naopak v mnohých aspektoch pôsobí proti zrýchlovaniu výrobného cyklu. Sme svedkami veľkého množstva snáh, ktoré si kladú za cieľ odpovedať na oba tieto tlaky, pricom zároveň so zrýchlením vývoja sa snažia zvýšiť kvalitu pri menšom riziku. Jednou z takýchto odpovedí na zvyšujúce sa požiadavky, ktoré sú kladené na profesiu, sú agilné metódy.

Nejde o presne definovaný súbor metód, ale skôr o novú preferenciu hodnôt a princípov, ktoré vedú pri správnom použití k dosiahnutiu cieľov. Tieto kľúčové pojmy zhrnula skupina „anarchistov“, ako sami seba nazývajú v Manifeste agilného vývoja softvéru [1]. Tým najdôležitejším cieľom je zmena charakteru reakcie na zmenu, pricom táto je oproti klasickým metodológiám pružnejšia, co v dnešnom svete predstavuje nemalú výhodu. Typ tejto zmeny azda vyjadruje slovenský doslovný preklad adjektíva *agile* a tým je *obratný*.

Takzvané nové metodológie tak prinášajú „obratný“ prístup k vývoju softvéru [2]. Tento prístup má mnoho rôznych charakteristík, azda najdôležitejšou je dôraz a dôvera v jednotlivca. Kladenie dôrazu na jednotlivca však neznamená rozbitie kolektívu na izolované individuality, ale práve naopak, zvýraznuje potrebu efektívnej komunikácie vo vnútri tímu. Táto charakteristika sa prejavuje aj v tom, že oproti zaužívaným metodológiám dosahujú agilné metódy vyššiu mieru samoorganizácie. Nie je náhodou, že táto samoorganizácia do istej miery pripomína prístup používaný pri tvorbe slobodného softvéru s otvoreným kódom (Opensource), ktorý je samostatnou metodológiou, ale podobne kladie dôraz na jednotlivca. Odlišný prístup k ľudskému faktoru a pozícii jednotlivca zúčastneného pri tvorbe softvéru vyniesla agilným metódam nemalú popularitu, ale aj zatracovanie.

Iste nejde o všeliek, použitie je určite limitované najmä dostupnými prostriedkami, pricom podľa vierozvestov agilného vývoja sú najkritickejším prostriedkom ľudia. Môžu sa dokonca objaviť oblasti, v ktorých je použitie agilných metód vyslovene kontraproduktívne. To je však znak všetkých prístupov, ktoré

softvérové inžinierstvo pozná. Metódy agilného vývoja softvéru si rozhodne zaslúžia skúmanie a vylepšovanie.

Účelom tejto publikácie je preskúmat a zhrnút poznatky a dosiahnuté výsledky v tejto oblasti. Pritom sa budeme snažit na dostupnom priestore podat charakteristiku širokého pola metodológií a súborov techník, ktoré v nej existujú. Dalej je cielom porovnat adat do súvisu jednotlivé faktory a cinnosti, ktoré sa vyskytujú v procese vývoja softvéru.

[1] Agile Manifesto
http://agilemanifesto.org/

[2] Fowler, M., New Metodology
http://www.martinfowler.com/articles/newMethodology.html

Obsah

Úvod 1

Agilný vývoj softvéru 3

Prehlad vybraných agilných metód a prístupov 11

Agilné metódy a ľudia 31

Vztah k zákazníkovi v agilných metódach 41

Riadenie v agilných metódach 51

Agilné metódy a návrh 61

Dokumentovanie a modelovanie v agilných metódach 73

Úvod

Publikáciu sme rozdelili do dvoch hlavných castí. Prvá je zameraná viac informatívne a zaoberá sa hlavnými pojmami z oblasti agilného vývoja softvéru. Základným kamenom agilných metód je *Manifest agilného vývoja*, ktorý je zaradený na zaciatok publikácie. Obsahom tejto prvej casti je aj prehlad a strucná charakteristika niektorých metodológii (kapitola *Prehlad vybraných agilných metód a prístupov*), ktorej cielom je uviest citatela do problematiky a poskytnút základný pohlad, na ktorom stavia zvyšok publikácie.

V druhej casti sa pozrieme na použitie agilných metód v praxi z rôznych aspektov. Aspektmi, ktoré súvisia s postavením ludí v procese vývoja softvéru sa zaoberajú kapitoly *Agilné metódy a ludia* a *Vztah kzákazníkovi v agilných metódach*. Dalej nasleduje téma, ktorá sa týka tak ludí, ako aj „neživých“ castí projektu, t.j. problematika riadenia (kapitola *Riadenie v agilných metódach*). Nakoniec sme vybrali témy, ktoré sa týkajú samotnej podstaty vývoja softvéru, a to úloha návrhu v agilných metódach (kapitola *Agilné metódy a návrh*) a použitie modelovania a tvorba dokumentácie (*Dokumentovanie a modelovanie v agilných metódach*).

Uvedené aspekty sme zvolili tak, aby do popredia vystúpili odlišnosti agilného vývoja softvéru a klasických (plánom riadených) metodológii.

Agilný vývoj softvéru

Jaroslav Kuruc, Richard Krupa

Popri „klasických“ metódach vývoja softvéru vznikajú aj alternatívne metódy. Velkou skupinou sú tzv. agilné metódy, ktoré sa od klasického prístupu líšia najmä vztahom k zákazníkovi, ale aj spôsobom riadenia projektu. Tieto metódy majú mnoho spolocného, co malo za následok vznik spolocného Manifestu agilného vývoja. Článok analyzuje jeho obsah.

Úvod

Už niekolko desatrocí sú metódy vývoja softvéru založené na princípe vývoja riadeného plánom. Tak ako sa tieto metódy postupom casu vyvíjali a zdokonalovali, prirodzene vznikali aj metódy založené na iných princípoch.

V posledných rokoch ci desatrociach vzniklo niekolko metód, ktoré sa líšia od klasického prístupu ci už vztahom k zákazníkovi alebo spôsobom riadenia projektu. Mnohé z nich sa radia do skupiny tzv. agilných metód. Sú to napríklad *Extrémne programovanie (Extreme Programming – XP)* [3][4], *Metóda vývoja dynamických systémov (Dynamic Systems Development Method – DSDM)* [5], *Scrum* [6], *Charakteristikami riadený vývoj (Feature-Driven Development – FDD)* [7], *Štíhle programovanie (Lean Programming – LP)* [8][9], *Adaptívny vývoj softvéru (Adaptive Software Development – ASD)* [10], *Crystal* [11] a mnohé iné. Tieto metódy a prístupy majú mnoho spolocného.

Aj to viedlo predstavitelov týchto metód k tomu, aby spolu vytvorili manifest [1], v ktorom sa prihlásili k spolocným hodnotám pri vývoji softvéru. V tomto clánku sa pozrieme na obsah manifestu a vysvetlíme si jeho body.

Manifest agilného vývoja

Manifest vznikol vo februári 2001, kedy sa skupina 17 ludí, propagátorov agilných metód ako aj priaznivcov podporujúcich potrebu alternatívnych metód vývoja softvéru, rozhodla napísat vyhlásenie, v ktorom sa prihlásili k urcitým hodnotám pri vývoji softvéru [2]. Manifest v originálnom znení vanglickom jazyku a jeho preklad do slovenského jazyka je nasledovný:

Vyvíjaním a napomáhaním iným pri vývoji objavujeme nové spôsoby vývoja softvéru. *Pri tejto práci sme prišli na tieto hodnoty:*	*We are uncovering better ways of developing software by doing it and helping others do it.* *Through this work we have come to value:*
Jednotlivci a interakcie *pred procesmi a návodmi*	***Individuals and interactions*** *over processes and tools*
Funkcný softvér *pred podrobnou dokumentáciou*	***Working software*** *over comprehensive documentation*
Spolupráca so zákazníkom *pred uzatváraním zmlúv*	***Customer collaboration*** *over contract negotiation*
Odozva na zmenu *pred dodržiavaním plánu*	***Responding to change*** *over following a plan*
Hoci aj ked sú položky na pravej strane výrazu hodnotné, my si ceníme viac tie na lavej strane.	*That is, while there is value in the items on the right, we value the items on the left more.*

Vysvetlime si jednotlivé body manifestu:

Jednotlivci a interakcie *pred procesmi a návodmi*

Pri riešení projektu by mali byt prvoradí ludia a nie ich úlohy v rámci procesu. Pri vývoji riadenom procesom je definovaná postupnost operácií, ktoré je potrebné vykonat. Ak sa nezohladnuje dôležitost jednotlivca, proces nie je jednoduché prispôsobovat novým podmienkam a teda nie je flexibilný.

Potrebný je však aj samotný proces. Na základe jeho existencie je totiž jednoduchšie nahrádzanie clenov tímu, resp. rozširovanie tímu. Ludia sú a musia byt nahraditelní.

Funkcný softvér *pred podrobnou dokumentáciou*

To, co je dôležité, je softvér. Aj ked budeme mat skvelú dokumentáciu, neposkytuje to, co žiadame od softvéru. Prioritou preto musí byt tvorba softvéru a nie dokumentácie. Tá je však tiež velmi potrebná, minimálne nato, aby sa bolo možné spätne vracat k rozhodnutiam. Preto je potrebné urobit kompromis medzi týmito protichodnými požiadavkami.

Spolupráca so zákazníkom *pred uzatváraním zmlúv*

Spolupráca so zákazníkom znamená, že všetci zúcastnení na projekte, ci už zákazník, používatel alebo vývojár, sú clenmi jedného tímu. Spojením ich skúseností dosiahneme lepšie výsledky a lacnejšie riešenia. Uzatváranie zmlúv je síce dôležité, ale nepostacuje. Zmluva definuje požiadavky na výsledný systém, avšak pocas samotného vývoja už nie je v projekte nápomocná.

Odozva na zmenu *pred dodržiavaním plánu*

Najväcšou hodnotou je schopnost prispôsobit sa meniacim sa požiadavkám. Plánovanie je užitocné a mnoho agilných metód obsahuje urcité formy plánovania. Dodržiavanie plánu je tiež užitocné, ale iba dovtedy, kým nie je plán príliš vzdialený od reality.

Princípy agilného vývoja

Súcastou manifestu je aj 12 princípov, ktoré by sme pri agilnom vývoji mali dodržiavat. Slovenský preklad a ich vysvetlenie vyzerá nasledovne:

Našou najvyššou prioritou je uspokojenie zákazníka skorým a castým dodávaním použitelného softvéru.

Naším prvoradým cielom pri vývoji softvéru by mala byt jeho použitelnost pre potreby zákazníka. Vdaka vcasnému a castému prototypovaniu sme schopní skôr zistit prípadné nedostatky, co nám umožnuje lepšie riadit projekt a prispôsobovat jeho priority.

Pojem *castý* sa chápe rôzne pre rôzne projekty. Ak by sa denne ci týždenne menil systém, jeho používatelia by z toho boli zrejme zmätení a v konecnom dôsledku by to projektu neprospelo. Zároven je však potrebné, aby systém pravidelne zvyšoval svoju funkcionalitu.

Uvítajme požiadavky na zmeny aj v neskorších fázach vývoja. Použitie agilného prístupu sa stane výhodou pri získavaní zákazníkov.

Agilné metódy umožnujú zmenu požiadaviek v ktorejkolvek fáze vývoja a to vdaka spomínanému castému prototypovaniu, ale aj technikám iterácie a casových ohranicení, neustálej aktualizácii návrhu a taktiež vôli menit tento návrh. To môže byt velkou výhodou oproti konkurencii.

V jednotlivých agilných metódach sa spôsob reagovania na zmenu líši, všetky agilné metódy však takýmto mechanizmom disponujú.

Dodávajme funkcný softvér co najcastejšie – od niekolkých týždnov po niekolko mesiacov, s možnostou skrátenia tohto casu.

U väčšiny projektov, ktorých vývoj je založený na inkrementálnom spôsobe, trvá cyklus od jedného do troch mesiacov. Kratší cyklus býva nárocný pre používatelov vzhladom na to, že sa musia casto prispôsobovat novým zmenám.

Ak je používatel schopný prijat kratší cyklus a vývojári zároven stihnú zapracovat požiadavky v kratšom case, potom je lepšie tento cyklus skrátit.

Manažéri a vývojári musia dennodenne spolupracovat na projekte.

Existuje mnoho projektov, v ktorých zákazník nie je ochotný venovat dostatok casu na kontrolu ich stavu. Štúdie pritom naznacujú závislost medzi zapojením zákazníka do projektu na jednej strane a jeho úspechom alebo neúspechom na strane druhej. Naskytá sa tu priestor pre manažéra, ktorého úlohou je, aby sprostredkovával informácie medzi používatelmi a vývojármi. To sa dá splnit napríklad pravidelným stretávaním sa manažéra so zákazníkom a vývojármi za úcelom vyhodnocovania potrieb používatelov.

Riešme projekty vdaka motivovaným jednotlivcom. Vytvorme im vhodné prostredie, snažme sa splnit ich požiadavky a verme im, že úlohu splnia.

Je lepšie mat motivovaných šikovných ludí, ktorí navzájom medzi sebou komunikujú ale v podstate nepotrebujú žiadne pravidlá, ako mat dobre definované pravidlá a nemotivovaných ludí. Jednotlivci sú najväcším prínosom pre projekt. Ak im poskytneme to, co potrebujú, vráti sa nám to vo zvýšení efektivity ich práce.

Najrýchlejšia a najefektívnejšia metóda získavania informácií od vývojového tímu alebo v rámci neho je rozhovor „tvárou v tvár“.

Casto je omnoho efektívnejšie v prípade potreby komunikovat priamo s clovekom, ktorý nám pri riešení problému môže pomôct, ako hladat a študovat rôzne casti dokumentácie. Navyše dokumentácia nemusí byt aktuálna a teda nemusí odrážat skutocný stav projektu, resp. môže obsahovat zavádzajúce informácie o predmete projektu.

Funkcný softvér predstavuje najdôležitejšiu mieru napredovania.

Je mnoho spôsobov, ktorými sa dá kontrolovat stav projektu. Môžu to byt rôzne plány ci dokumenty, najdôveryhodnejším je však funkcný kód.

Agilné metódy kladú zvlášt velký dôraz na to, aby v ktorejkolvek fáze vývoja existoval funkcný spustitelný kód. Tento spôsob vývoja vyžaduje rozdrobenie projektu na menšie casti, ktoré by sa postupne vyvíjali. Rozdrobenie projektu nemusí byt vždy priamociare a jednoduché, no vždy je možné.

Agilné procesy zvyšujú udržatelnost napredovania. Zákazníci, vývojári a používatelia by mali byt schopní trvalo zabezpecovat napredovanie.

V agilných metódach by si reakcia na zmenu nemala vyžiadat nadmernú prácu vo forme nadcasov, pretože to výrazným spôsobom zvyšuje únavu vývojárov. Únava so sebou prináša zvýšenie poctu chýb, co znehodnocuje hodnotu navyše vykonanej práce. Tým, že pri agilnom spôsobe vývoja softvéru sú požiadavky na zmenu bežné, ba až ocakávané, aj ich zapracovávanie by sa malo vykonávat v bežnom pracovnom case. Prílišné nadcasy naznacujú, že v projekte nie je všetko v poriadku.

Obratnost znamená nepretržitý dôraz na technickú kvalitu a dobrý návrh.

Tým, že sa má systém vediet rýchlo prispôsobovat zmenám, je neustále potrebné robit zmeny v jeho návrhu. Zmeny sa v návrhu robia tým jednoduchšie, cím je tento návrh jednoduchší. Preto je potrebné neustále zvyšovat kvalitu návrhu systému. Jednoduchý návrh zároven prináša možnost jednoduchšieho zapracovania radikálnejších zmien v návrhu.

Podstatná je jednoduchost, inými slovami maximalizácia práce, ktorú netreba vykonat.

Vytváraný systém by mal byt co najjednoduchší. Je dôležité navrhovat iba potrebné casti, pretože tie, ktoré sú v danom case nepotrebné, sa neskôr môžu ukázat ako zle navrhnuté. V budúcnosti sa navyše ani nemusí objavit požiadavka na takúto funkcionalitu, cím sa predražuje vývoj.

Najlepšie architektúry, požiadavky a návrhy vytvárajú samoorganizované tímy.

V samoorganizovaných tímoch je každý clen rovnocenný a môže byt tvorcom myšlienky, ktorá projektu prispeje. Všetci clenovia tímu tieto myšlienky podrobujú kritike, coho dôsledkom je, že

v danom case majú iba tie najlepšie myšlienky šancu ujat sa. Dobré myšlienky sú základom pre tvorbu dobrej architektúry a návrhu.

Podrobnosti návrhu systému vždy zaskocia aj tých najskúsenejších návrhárov. Preto je dobré s postupom casu prispôsobovat návrh. Príliš uzavretá architektúra, alebo príliš skoro uzavretá, už nebude schopná akceptovat prípadné zmeny zistené pri implementácii alebo pri zmene požiadaviek. Architektúra, ktorá sa vybuduje postupne je flexibilnejšia.

Po urcitom case tím zistí, ako môže byt efektívnejší a primerane prispôsobí svoje správanie.

Vzájomnou interakciou clenov si tím vytvára vnútornú spätnú väzbu, ktorá spolu s potrebami z vonku vytvára samoregulovacie prostredie.

Použitá literatúra

[1] Manifesto for Agile Software Development
http://www.agilemanifesto.org

[2] Cockburn, A.: Agile Software Development, draft version 3b. The Agile Software Development Series, 2000, p. 175-183.

[3] XProgramming.com: An Extreme Programming Resource
http://www.xprogramming.com

[4] Extreme Programming: A gentle introduction
http://www.extremeprogramming.org

[5] DSDM Consortium, Dynamic Systems Development Method
http://www.dsdm.org

[6] Scrum
http://www.controlchaos.com

[7] Feature-Driven Development
http://www.togethercommunity.com/coad-letter/Coad-Letter-0070.html

[8] Poppendieck, M.: Lean Programming. Part 1. May 2001
http://www.sdmagazine.com/documents/s=731/sdm0105e/0105e.htm

[9] Poppendieck, M.: Lean Programming. Part 2. June 2001
http://www.sdmagazine.com/documents/s=730/sdm0106g/0106g.htm

[10] Adaptive Software Development
http://www.adaptivesd.com

[11] Crystal
http://crystalmethodologies.org

Prehlad vybraných agilných metód a prístupov

Richard Krupa, Jaroslav Kuruc

Článok pojednáva o vybraných metódach a prístupoch k agilnému vývoju softvéru. Charakterizuje niektoré prístupy (Extrémne programovanie, Metóda vývoja dynamických systémov, Štíhle programovanie, Scrum, Charakteristikami riadený vývoj) a na záver ich porovnáva navzájom aj vzhladom voci Manifestu agilného vývoja.

Extrémne programovanie

Extrémne programovanie (*Extreme Programming – XP*) [2][3][4] nie je metódou ako takou, ide skôr o súbor praktík, ktorých spôsob použitia nie je explicitne definovaný. Nie je povedané, kedy sa má co a ako urobit a už vôbec nie v akej postupnosti. Tieto praktiky však vzájomne velmi úzko súvisia a nie je možné riadit sa len niektorými z nich, pretože mnohokrát má použitie jednej praktiky význam iba v prípade použitia inej. Sú založené na princípoch jednoduchosti, komunikácii, spätnej väzby a odvahy. Mnohé z agilných metód vychádzajú práve z praktík Extrémneho programovania. Použitie tohto prístupu je však vhodné iba pre malé tímy vyvíjajúce softvér v krátkom case a v prostredí, kde sa casto menia požiadavky. Casto takto fungujú interné tímy informatikov (alebo tiež IT oddelenia) v inštitúciách ako sú banky a poistovne.

Vysvetlime si jednotlivé praktiky XP.

Tím ako celok

Všetci zúcastnení na projekte sú clenmi jedného tímu. V tíme musí byt reprezentant zákazníka, ktorý identifikuje požiadavky, stanovuje priority a riadi projekt. Najlepšie je, ked je to aj koncový používatel, ktorý dobre pozná problémovú oblast. Dalej má tím samozrejme programátorov. Je dobré, ked sú v tíme aj testeri, ktorí môžu pomôct zákazníkovi vytvorit akceptacné testy. Dalej by mal tím obsahovat aj analytikov, ktorých úlohou je pomáhat

zákazníkovi pri identifikovaní požiadaviek. V tíme by mal byt vedúci, ktorého úlohou je zabezpecovat prostriedky a komunikovat za tím smerom navonok.

Žiaden z clenov tímu by však nemal byt špecialistom len vo svojom odbore. Vo väčšine projektov je vhodné, ked majú clenovia tímu aspon aký-taký prehlad o systéme ako o celku. Môže sa to totiž prejavit vkvalitnejšom realizovaní vlastných úloh.

Plánovanie

Úlohou plánovania v XP je stanovit odpovede na tieto otázky:

- Co sa dokoncí do daného dátumu?
- A co sa bude robit potom?

Dôraz sa kladie na riadenie projektu ako také (co je viac priamociarejšie), ako na stanovovanie toho, co bude treba kedy urobit a ako dlho to bude trvat (to je trochu zložitejšie). V XP sú definované dva spôsoby riadenia:

- **Plánovanie vydania** – Je to praktika, kedy zákazník prezentuje programátorom požadované vlastnosti a títo následne zhodnotia ich nárocnost. Na základe odhadu nákladov a dôležitosti zapracovania daných požiadaviek môže potom zákazník stanovit plán. Pociatocné plány sú zvycajne nepresné, spresnia sa až ked tím zacne na úlohe pracovat.
- **Iteracné plánovanie** – Ide o praktiku, kedy si tím stanovuje úlohy na niekolko nasledujúcich týždnov. Tím zvycajne vytvára vydania každé dva týždne, pricom výstupom je funkcný použitelný kód. Po skoncení danej iterácie zákazník navrhne funkcnost, ktorá by sa mala implementovat v nasledujúcej iterácii. Programátori to zhodnotia (na jemnejšej úrovni ako pri Plánovaní vydania) a na základe množstva vykonanej práce naplánujú, co by sa malo stihnút v nasledujúcej iterácii.

Výhodou takéhoto plánovania je fakt, že na konci každej iterácie je zrejmý stav projektu. Neexistuje stav úlohy ako dokoncená na 90%; úloha je bud dokoncená alebo nedokoncená.

Takýto prístup operatívneho plánovania je charakteristický všeobecne pre všetky agilné metódy aprístupy. Vychádza z požiadavky co najrýchlejšej odpovede na zmenu.

Testy od zákazníka

Súcastou požadovanej vlastnosti na vytváraný systém od zákazníka je aj množina akceptacných testov, ktoré vytvorený systém musí splnit. Ak ho splní, je zrejmé, že požadovaná vlastnost je implementovaná postacujúco. Tieto testy sa vykonávajú automatizovane. Je to výhodné aj v prípade potreby zrýchlenia vývoja, kedy sa môže manuálne testovanie vynechat.

Od momentu, kedy sa akceptacné testy po prvý raz vykonajú, musia sa vykonávat už vždy. Zabezpecí sa tým, že funkcionalita systému neklesne, iba narastie. Tento spôsob testovania sa tiež nazýva regresné testovanie. Testy sa vykonávajú pravidelne, napríklad denne, a pri pridaní novej funkcionality. Vždy sa vykonáva celá množina testov, teda nie len tie, ktoré testujú práve pridanú funkcionalitu. Dôvodom je to, aby bolo možné odhalit prípadné vedlajšie úcinky práve vykonanej zmeny.

Malé vydania

Tím vytvára nasledovné druhy vydaní:

- Na konci každej iterácie sa vytvorí funkcný otestovaný systém, ktorý splna vlastnosti požadované od zákazníka. Zákazník môže s daným vydaním nakladat ako uzná za vhodné, bud ho poskytne na odskúšanie, alebo ho dokonca poskytne koncovým používatelom. Co je ale najdôležitejšie je to, že zákazník má dokonalý prehlad o funkcionalite takéhoto systému.
- Systém urcený pre koncového používatela. Vydáva sa tak casto, ako je to možné, záleží to však od druhu systému. Pri WWW aplikácii to môže byt denne, pri iných projektoch to môže byt obdobie týždnov až mesiacov.

Dôležitý je fakt, že tento prístup (rovnako ako väcšina agilných prístupov vývoja softvéru) je založený na iteracnom spôsobe vývoja. Vdaka tomu existuje v ktoromkolvek okamihu projektu funkcná verzia systému s aspon minimálnou požadovanou funkcionalitou.

Jednoduchý návrh

Vytváraný systém by mal byt co najjednoduchší. Zacína sa z najjednoduchšieho návrhu a postupným pridávaním funkcionality sa systém rozširuje. Táto praktika velmi úzko súvisí so

zušlachtovaním kódu. Návrh nie je totiž jednorazovou záležitostou, neustále sa aktualizuje pomocou *refaktorizácie* (vid praktiku Zušlachtovanie návrhu). Pri iteracnom spôsobe vývoja, ako napríklad v XP, je návrh zvlášt dôležitý, preto sa na nom pracuje pocas celého trvania projektu.

To sa samozrejme nedá aplikovat na akýkolvek projekt. Je nieco iné refaktorizovat jednoduchý, resp. stredne zložitý systém a nieco iné refaktorizovat systém obrovského rozsahu, napríklad operacný systém. V systéme takéhoto rozsahu je totiž casto námaha spojená s refaktorizáciou neúmerná jej osohu. Vo všeobecnosti platí, že cím je systém menší, tým je viac otvorenejší zmenám návrhu.

Programovanie v párocH

Kód v XP píše dvojica programátorov pracujúca na jednom pocítaci. Táto technika zabezpecuje, že produkovaný kód je zároven kontrolovaný iným programátorom a výsledkom je lepšie navrhnutý, implementovaný a otestovaný systém.

Môže sa zdat, že nie je efektívne, aby dvaja programátori robili prácu jedného, opak je však pravdou. Prax ukazuje, že výsledkom programovania v párocH je lepší kód za takmer rovnaký cas, ako keby ho vyvíjali dvaja programátori nezávisle. Vyžaduje si to však urcitý cas, kým si programátor zvykne pracovat v páre. Až 90% programátorov, ktorí sa naucia pracovat v párocH, tento spôsob preferuje.

Tým, že programátori sa v párocH striedajú, navzájom sa aj od seba ucia. To je prínosom pre celý tím. Nakolko tento princíp nie je závislý od ostatných princípov XP a jeho použitie si nevyžaduje žiadne dalšie opatrenia, dá sa aplikovat vpodstate v projekte lubovolného typu, aj mimo XP.

Vývoj riadený testovaním

Spätná väzba je pri použití XP životne dôležitá. Dobrá spätná väzba si vyžaduje vhodný spôsob testovania. Tímy casto pracujú spôsobom vývoja riadeného testovaním. Najprv napíšu test, ktorý skontroluje správnost neskôr vytvorenej požadovanej funkcionality systému, a potom implementujú samotnú funkcionalitu systému.

Nestací však testy iba písat. Treba ich aj spúštat. Tieto testy sa zhromaždujú a zakaždým, ked programátor pridá do

systému funkcionalitu, musí spustit všetky testy a všetky musia prejst. Programátor tak má okamžitú spätnú väzbu.

Takýto prístup, ako je zrejmé, má svoje výhody. Je potrebné ale podotknút, že si vyžaduje väčšie nároky na programátorov, pretože okrem funkcionality musia programovat aj testy. Je to ale zvlášt výhodné v projekte väčšieho rozsahu, kedy to môže výrazne urýchlit dobu testovania, pretože základná funkcionalita sa odskúša automatickými regresnými testami a zvyšné casti potom odskúšajú testeri.

Zušlachtovanie návrhu

Cielom XP je funkcný kód po každej iterácii. Preto je potrebné, aby bol systém dobre navrhnutý. Riešením môže byt aj tzv. *refaktorizácia*. Je to proces, kedy sa odstranuje duplicitný kód, alebo sa zhutnuje kód zovšeobecnením a následným parametrizovaním. Taktiež sa odstranujú závislosti v kóde.

Nato, aby sme si boli istí, že refaktorizáciou kódu nedošlo k jeho znefunkcneniu, je nanajvýš vhodné použit automatizované regresné testovanie. Táto technika je teda silne previazané s technikou Testy od zákazníka.

Priebežná integrácia

Vývojári by mali systém integrovat tak casto, ako sa len dá. Obcasná integrácia môže viest k vážnym problémom. Je nárocné „kvalitne“ integrovat casti systému, zvlášt ked sa integrácia nerobí casto, a zároven túto integráciu casto robí osoba, ktorá nemá až taký velký prehlad o podrobnostiach systému. V niektorých prípadoch môžu byt programátori zriedkavým integrovaním dokonca obmedzovaní, napríklad ked ide o funkcnost, ktorá je závislá na rôznych castiach systému a tieto sa navyše casto menia.

Kolektívne vlastníctvo kódu

Pri použití XP vlastnia kód všetci clenovia tímu. Ktokolvek totiž môže kedykolvek upravit akýkolvek kód. Výhodou je, že kód je pod drobnohladom viacerých programátorov a je teda kvalitnejší, lepšie ošetrený od chýb. Má to aj inú výhodu. Ked kód vlastní jediný programátor a iný programátor objaví, že sa mu hodí funkcionalita kódu, ktorý nevlastní, programátor je závislý od

iného programátora. V prípade, že zodpovedný programátor je v danej chvíli zaneprázdnený, tento programátor je nútený si potrebnú funkcionalitu naprogramovat sám. To má za následok duplicitu a neporiadok v celkovom systéme.

Kolektívne vlastníctvo prináša so sebou aj možné problémy. Problém môže vzniknút v prípade, že programátor zacne menit kód, ktorému až tak dobre nerozumie. XP to rieši regresným testovaním a programovaním v párach, pricom jedným v páre je znalec daného kódu.

Štandard písania kódu

Programátori by pri písaní kódu mali dodržiavat spolocné pravidlá. Cielom je, aby text programu vyzeral tak, akoby ho písal jeden clovek. To, aké sú pravidlá, nie je dôležitejšie ako to, že kód vyzerá rovnako. Je to zvlášt dôležité vsúvislosti skolektívnym vlastníctvom kódu. Tým, že sa tieto pravidlá stanovia na zaciatku, text programu sa píše podla dohody a neskôr už nie je nutné ho prerábat.

Metafora

Pocas trvania projektu si clenovia tímu vytvárajú názor na to, akým spôsobom systém funguje. Nazvime to metafora. Je vhodné, aby všetci clenovia tímu mali co najviac podobnú takúto predstavu. Súvisí to napríklad spomenovávaním tried alebo ich metód. Jednoducho je dobré, aby bol každý clen tímu schopný logicky nájst v systéme cast, ktorú hladá a tiež ked nieco pomenúva, aby to pomenoval logicky aj pre ostatných clenov.

Udržatelné napredovanie

Žiaden programátor nie je schopný pracovat 60 hodín týždenne tak, aby sa to neprejavilo na kvalite jeho práce. Preto je potrebné, aby programátori programovali vtedy, ked je to ešte efektívne, aby pracovali nadcas vtedy, ak je to pre dalšiu prácu prínosom a oddychovali vtedy, ked je to potrebné. Cielom nie je na projekte sa znicit, ale zadanie úlohy vyriešit.

Metóda vývoja dynamických systémov

Metóda vývoja dynamických systémov (*Dynamic Systems Development Method – DSDM*) [5] patrí taktiež medzi agilné metódy. Pri klasickom prístupe k vývoju softvéru je cielom uspokojenie požiadaviek, casto aj nepresných. Tieto sa navyše pocas trvania projektu môžu menit. Aj vzhladom na to sa cas a prostriedky potrebné na vývoj casto menia. V prístupe DSDM je to presne naopak. Cas a pokial možno aj prostriedky sú fixné a funkcionalita je variabilná.

Táto metóda opisuje, ako postupovat pri riešení projektu, nepredpisuje však spôsob, ako realizovat ciele v rámci tohto postupu. Je preto vhodné ju použit v kombinácii s inými agilnými prístupmi, napríklad XP.

Princípy DSDM

Mnohé z princípov, na ktorých táto metóda stavia je zhodných s princípmi agilného vývoja deklarovanými v Manifeste agilného vývoja [1]. V rôznych castiach metódy sú podla potreby aplikované tieto princípy:

- Je nevyhnutná aktívna úcast používatela.
- Tím musí byt splnomocnený robit samostatné rozhodnutia.
- Dôraz sa kladie na casté dodávanie produktov.
- Dôležité je splnenie nevyhnutných požiadaviek v stanovenom case.
- Na dosiahnutie ciela je nevyhnutné použitie iteratívneho a inkrementálneho spôsobu vývoja.
- Všetky zmeny pocas vývoja musia byt reverzibilné.
- Musia byt dopredu stanovené hrubé požiadavky na systém.
- Testovanie sa vykonáva pocas celého životného cyklu.
- Je nutná spolupráca medzi všetkými zúcastnenými osobami.

Životný cyklus DSDM

Životný cyklus, ktorý používa DSDM je iteratívny a inkrementálny. Cyklus má 5 fáz a výsledok je použitelný iba po prechode cez všetky fázy. Sú to fázy:

- Prieskum uskutocnitelnosti
- Prieskum problémovej oblasti
- Iterácia vo funkcionálnom modeli
- Iterácia v návrhu a vytváraní systému
- Nasadenie

Prieskum uskutocnitelnosti

Úlohou je zhodnotenie vhodnosti použitia metódy DSDM. Táto fáza zvycajne koncí po niekolkých týždnoch.

Prieskum problémovej oblasti

Úlohou tejto fázy je analýza cinností potrebných na vyriešenie projektu. Predstavuje akýsi technický základ pre budúcu prácu. Stanovia sa požiadavky na vyššej úrovni a nacrtne sa architektúra systému. Rovnako ako fáza Prieskum uskutocnitelnosti, aj táto fáza je krátkodobá a netrvá zvycajne dlhšie ako mesiac.

Iterácia vo funkcionálnom modeli

Táto fáza spocíva v definovaní vysokoúrovnových funkcionálnych požiadaviek. Rovnako ako iterácia v návrhu a vytváraní, aj iterácia vo funkcionálnom modeli spocíva v štyroch fázach:

- identifikácia, co sa má vytvorit
- dohoda ako a kedy to urobit
- vytvorenie produktu
- kontrola, že to bolo vytvorené tak, ako malo byt

Väcšina práce spocíva v dvoch iteracných fázach, kedy sa z prototypu postupne vytvorí otestovaný systém. Prototyp je zástupcom finálneho systému, preto už aj v nom sa riešia nefunkcionálne požiadavky na systém, ako je napríklad výkonnost.

Iterácia v návrhu a vytváraní systému

Cielom tejto fázy je získat otestovaný systém tak, aby mohol byt poskytnutý používatelom. Testovanie nie je explicitne zaznacené v životnom cykle, pretože je súcastou jednak iterácie vo funkcionálnom modeli a jednak iterácie v návrhu a vytváraní systému.

Nasadenie

Nasadenie pokrýva prechod z prostredia vývoja do prostredia prevádzky. Spocíva v zaškolení používatelov v práci so systémom. Taktiež sa zhodnotí projekt z pohladu dosiahnutia výsledkov. Hodnotí sa splnenie požiadaviek identifikovaných pocas vývoja a miera, do akej systém tieto požiadavky splna.

Kedže je to inkrementálny spôsob vývoja, reakcia na hodnotenie môže byt rôzna:

- Všetky požiadavky boli splnené a nie je dôvod na další vývoj.
- Bola objavená nová funkcionalita a vývoj sa vracia do fázy prieskumu problémovej oblasti.
- Z dôvodu nedostatku casu nebola splnená menej dôležitá funkcionalita. Vývoj sa vracia na zaciatok fázy iterácie vo funkcionálnom modeli.
- Nebola splnená nefunkcionálna požiadavka a vývoj sa vracia do fázy iterácie v návrhu a vytváraní systému.

Štíhle programovanie

Štíhle programovanie (*Lean Programming – LP*) predstavuje aplikovanie základných princípov a pravidiel Štíhlej výroby na vývoj softvérových produktov [10] [11]. Štíhla výroba vznikla z dôvodu zvýšenia efektívnosti výroby produktov pomocou montážnych liniek, pricom jeho história siaha až do obdobia krátko po 2. svetovej vojne. Štíhla výroba spolu s Komplexným manažmentom kvality (Total Quality Management) definujú desat základných praktík, pomocou ktorých je možné významne zefektívnit výrobu hmatatelných, nesoftvérových produktov:

1. Eliminácia strát
2. Minimalizácia zásob
3. Maximalizácia toku výroby

4. Produkcia vyvolaná požiadavkami
5. Splnenie požiadaviek zákazníka
6. Eliminácia opráv
7. Splnomocnovanie pracovníkov
8. Zakázanie lokálnej optimalizácie
9. Pestovanie vztahov s dodávatelmi
10. Tvorba kultúry kontinuálneho zdokonalovania

Štíhle programovanie interpretuje tieto praktiky spôsobom, ktorý dovoluje ich použitie aj pri tvorbe softvérových produktov.

Eliminácia strát

Dokumenty, diagramy a modely vyprodukované pocas vývoja softvérového produktu sú casto pocas tohto vývoja „konzumované“ a nie sú castou finálneho produktu. V okamihu dorucenia produktu zákazníkovi, strácajú mnohé dokumenty, diagramy a modely svoj význam a samotný zákazník sa o ne zaujíma iba velmi málo. Každý medziprodukt, ktorý je skonzumovaný pocas vývoja produktu je vhodným kandidátom na detailnejšie preskúmanie jeho úlohy v tomto procese. Medziprodukt musí nielen pridávat hodnotu k finálnemu produktu, ale zároven musí predstavovat najefektívnejší spôsob, ako túto hodnotu dosiahnut.

Minimalizácia zásob

Paralelu zásob uplatnujúcich sa pri výrobe nesoftvérových produktov je pri vývoji softvéru možné vidiet vdokumentácii, ktorá nie je súcastou finálneho produktu. Cas potrebný na tvorbu dokumentov zaberá podstatnú cast životného cyklu softvérového produktu. Porovnanie s výrobnou linkou je v podobe zásob vznikajúcich na jednotlivých stupnoch výroby. Pre zvýšenie toku výroby je nutné tieto zásoby minimalizovat. Takisto pre zrýchlenie vývoja softvérového produktu musia byt zásoby v podobe dokumentácie minimalizované.

Maximalizácia toku výroby

Pri výrobe nesoftvérových produktov Štíhla výroba odporúca princípy malých dávok a hladkého toku. Tieto princípy sa zhodujú

s iteratívnym spôsobom vývoja softvéru, kedy sa malá, ale kompaktná cast softvéru vyvíja a dodáva pocas jedného vývojového cyklu. Každá iterácia prispieva do vyvíjaného systému sadou nových charakteristík, má trvanie niekolkých týždnov až mesiacov a zahrna všetky procesy od získavania požiadaviek až po akceptacné testovanie.

Produkcia vyvolaná požiadavkami

Tento princíp zdôraznuje skutocnost, že softvérové systémy by mali byt navrhované tak, aby sa vedeli flexibilne vyrovnat so zmenami, co ale neznamená, že by sa tieto zmeny mali pri návrhu predvídat. Pocítanie so zmenami, ktoré nie sú súcastou požiadaviek predražuje výsledný systém vzhladom na skutocnost, že nakoniec sa tieto zmeny nemusia uskutocnit.

Splnenie požiadaviek zákazníka

Jednou znajcastejších prícin neúspechu softvérových projektov boli v minulosti chýbajúce, nekompletné alebo nesprávne požiadavky. Výsledkom bolo spisovanie detailnejšie opísaných požiadaviek aich potvrdzovanie podpismi za stranu dodávatela a objednávatela. Ak sa od softvérového systému ocakáva pružné prispôsobovanie sa zmenám, nie je takýto postup vhodný, vzhladom na výrazné procedurálne spomalenie v podobe potreby odobrenia dalších zmien dalšími podpismi. Jedným z efektívnych spôsobov, ako získat presné požiadavky je iteracný spôsob vývoja softvérových systémov, ktorý umožnuje castú a priamu spätnú väzbu s budúcim používatelom už v case vývoja produktu.

Eliminácia opráv

Tento princíp zdôraznuje skutocnost, že najlacnejší spôsob ako vytvorit nejaký produkt, je vytvorit ho správne na prvý raz. Akékolvek dodatocné opravy znacne predražujú výsledný produkt. Štíhle programovanie doporucuje dve techniky, ktoré umožnujú prispôsobenie sa zmenám a vyvarovanie sa pritom zbytocných opráv v budúcnosti. Prvou z nich je zostavenie testov na zaciatku vývoja softvérového produktu. Ak sa v priebehu programovania reaguje na zmeny, prebehne regresné testovanie všetkých jednotiek. V prípade neúspešných testov sa proces zastaví do tej doby, kým sa daný problém neodstráni. Druhá technika prichádza na rad pri refaktoringu a zdokonalovaní

návrhu existujúcich systémov. Pomocou refaktoringu sa pôvodný návrh môže sústredit na pôvodné požiadavky bez toho, aby návrhár musel mysliet na budúce potreby.

Splnomocnovanie pracovníkov

Štíhla výroba vyzdvihuje ludí a tímovú spoluprácu nad „papierovaním“ ašpecifikovaním presných procesov. Treba sa sústredovat na metódy, ktoré formujú tímy tak, aby vedeli pomenovat a pružne riešit svoje vlastné problémy.

Zakázanie lokálnej optimalizácie

Pri výrobe nesoftvérových produktov Štíhla výroba zdôraznuje, že nie je nutné, aby výrobné zariadenia pracovali na plný výkon, práve naopak, niekedy to nie je vhodné, pretože to nepriaznivo ovplyvnuje celkový proces, napríklad vytváranie zásob v medziprocesných skladoch. Dôležitejšou vlastnostou výrobných zariadení ako je maximálna výrobná kapacita je skôr univerzálnost a schopnost rýchleho prispôsobenia sa zmenám vo výrobe. Táto myšlienka naráža ešte aj v súcasnosti na nepochopenie. V tomto kontexte Štíhle programovanie naráža na nepochopenie skutocnosti, že tak, ako všetko, aj rozsah softvérového projektu sa mení a je na škodu projektu ako takého, ked sa pocas celého jeho produktu striktne dodržiava pôvodne stanovený rozsah.

Pestovanie vzt ahov s dodávatelmi

Manažment dodávatelských retazcov pomáha rozvíjat vztahy medzi jednotlivými spolocnostami. Štíhla výroba ucí, že je výhodnejšie pestovat si niekolko dodávatelsko-odberatelských vztahov na dlhšie obdobie, ako si vyberat zponuky mnohých spolocností, napríklad na základe ceny. Spolocnosti si tak môžu navzájom pomáhat v zdokonalovaní svojich produktov a tiež ich toku.

Tvorba kultúry kontinuálneho zdokonalovania

Softvérové spolocnosti sa snažia neustále zvyšovat stupen svojej „softvérovej vyspelosti“. Jednou z ciest je získavanie rôznych ocenení a certifikátov. Týmto spôsobom si spolocnost potvrdí

svoju vyspelost z hladiska dokumentovania vývojového procesu. Ak sa však na vyspelost pozeráme z hladiska schopnosti rýchleho prispôsobovania sa zmenám, nemusia tieto certifikáty zvyšovat kredit u potenciálneho zákazníka. Jednou z možností ako pestovat kultúru kontinuálneho zdokonalovania je použit prístup Plan-Do-Check-Act, ktorý vyhovuje iteratívnemu spôsobu vývoja softvérových produktov. Jednotlivé kroky sú:

- ***Plan:*** Výber problému, jeho analýza a nájdenie pravdepodobných prícin.
- ***Do:*** Vykonanie experimentov na preskúmanie pravdepodobných prícin problému.
- ***Check:*** Analýza výstupných údajov experimentov a validácia prícin problému.
- ***Act:*** Zjemnenie a štandardizácia riešenia v závislosti od výsledku predchádzajúceho kroku.

Kontinuálne zdokonalovanie produktu je v iteracnom procese výrazne umocnené využitím refaktorizácie.

Scrum

Scrum je variácia Sashimiho prístupu k softvérovému inžinierstvu „all-in-once“ [6] [7]. Jedná sa o iteratívny, inkrementálny prístup k tvorbe softvérových produktov. Jeho najdôležitejšie piliere sú:

- ***Splnomocnenie tímu***: Akonáhle tím dostane nejakú prácu, je zodpovedný za to, ako si s nou poradí. Pocas každej iterácie robí tím to najlepšie, co vie. Kým tím pracuje, jeho jediná interakcia s manažmentom je informovanie o priebehu iterácie.
- ***Prispôsobivost:*** Jednotlivé iterácie majú maximálnu dlžku 30 dní, takže tím a manažment môže vyhodnotit výsledky iterácie a definovat, aké dalšie kroky by sa mali uskutocnit pocas dalšej iterácie. Pri definovaní dalších krokov sa berú do úvahy zmeny pochádzajúce z vonkajšieho prostredia.

Scrum sa snaží zabránit vzniku nasledujúcich „chorôb“, ktoré vznikajú použitím zlej filozofie a techník pri vývoji softvérových produktov:

- Pocas doby vývoja softvérového produktu sa požiadavky nan zmenia do takej miery, že výsledný produkt casto nevyhovuje

aktuálnym požiadavkám a sú nutné výrazné zmeny. Príčinou je prijímanie informácií o požiadavkách iba pocas etapy plánovania.

- Manažment „verí“, že dokáže predpovedat cenu, dátum dodania a funkcionalitu výsledného produktu.
- Vývojári a projektoví manažéri žijú v klamstve, predstierajúc, že vedia plánovat, predpovedat a dodat produkt. Vytvárajú produkt nejakým spôsobom a predstierajú pritom, že používajú iný spôsob. Výsledkom je, že neexistuje dostatocná kontrola.

Princípy

Scrum rozdeluje vývoj softvéru na krátke casové úseky, ktoré nazýva Šprinty. Jeden šprint je casovo ohranicený na maximálne 30 dní. Jednotlivé etapy vývoja softvérového produktu (Požiadavky, Analýza, Návrh, Implementácia, Testovanie) sú priradené jednému alebo viacerým šprintom. Šprint je riadený dennými stretnutiami, na ktorých sa vyhodnocuje dosiahnutý pokrok a vykonávajú sa tu všetky potrebné rozhodnutia. Pred zaciatkom šprintu sa definuje kniha nedokoncenej práce pre tento šprint (Sprint Backlog). Úlohou šprintu je vyriešit všetky úlohy, ktoré sú zahrnuté v jeho knihe nedokoncených prác. Pocas trvania šprintu nie je možné doplnat obsah jeho knihy nedokoncených prác mimo prostredia samotného šprintu.

Implementácia procesu

Scrum zdôraznuje použitie zdravého rozumu. Jeho najvyššie priority sú pracovat a vydávat produkty. Jeho implementácia zahrna nasledujúce body:

1. Spustenie procesu
2. Menovanie predsedu
3. Identifikácia knihy nedokoncených prác
4. Ustanovenie a vedenie denných stretnutí

Spustenie procesu

Pre spustenie procesu je potrebné definovat pracovný tím. Tento by nemal mat viac ako 6 až 9 clenov. Ak je k dispozícii viac clenov ako je možné manažovat, malo by sa vytvorit viac tímov.

Tím musí mat k dispozícii vlastný pracovný priestor, pricom by ho mal využívat každý jeho clen.

Menovanie predsedu

Predseda je osoba, ktorá vedie denné stretnutia, empiricky meria dosiahnutý pokrok a vykonáva rozhodnutia. Na stretnutí sa predseda pýta každého clena tímu tri otázky:

1. Co si spravil od minulého stretnutia?
2. Na aké problémy si odvtedy narazil?
3. Co spravíš do budúceho stretnutia?

Predseda musí byt schopný na základe všetkých odpovedí vykonávat okamžité rozhodnutia, a to priamo na stretnutí. Takisto rieši všetky pracovné prekážky.

Identifikácia knihy nedokoncených prác

Kniha nedokoncených prác predstavuje všetky úlohy, ktoré je potrebné vykonat, aby sa splnili podmienky ukoncenia projektu. Tieto úlohy pritom nemusia byt dobre definované a môžu mat dokonca charakter vízie. Ako prvý krok je identifikácia všetkých úloh. Jednotlivé úlohy sa združia do skupín tak, aby predpokladaná doba ich realizácie nepresahovala 30 dní. V prípade, že nie je možné stanovit skupinu úloh s trvaním približne 30 dní, urcí sa skupina skratším termínom. Existuje iba jediná osoba, ktorá priraduje jednotlivým úlohám priority. Pred zaciatkom šprintu si tím vyberie knihu nedokoncených prác pre daný šprint. V priebehu šprintu sa vykonávajú iba úlohy z knihy nedokoncených prác pre daný šprint.

Denné stretnutia

Vzhladom na to, že stretnutia sú denné, existuje tu možnost vzniku réžie v súvislosti s definovaní jeho miesta a casu. Táto réžia súvisí so samotnou aktivitou hladania vhodného miesta a casu a tiež s možným obmedzením niektorých clenov tímu v snahe prispôsobit sa ostatným. Samotné stretnutie by nemalo presiahnut 30 minút. V jeho priebehu sa predseda pýta jednotlivých clenov tímu tri otázky a na základe odpovedí vykonáva potrebné rozhodnutia. Akákolvek diskusia, ktorá nesúvisí s danými tromi otázkami sa odkladá na dalšie stretnutie.

Charakteristikami riadený vývoj

Charakteristikami riadený vývoj (*Feature-Driven Development – FDD*) predstavuje iteracný proces s krátkou dobou iterácie [8] [9]. Používa objektovo-orientovaný prístup, co potvrdzuje aj ním definovaný pojem „vlastník triedy“. Proces sa skladá z nasledujúcich castí:

1. Tvorba celkového modelu
2. Tvorba zoznamu charakteristík
3. Plánovanie po charakteristikách
4. Návrh a implementácia po charakteristikách

Tvorba celkového modelu

Znalci domény vyvíjaného softvérového produktu a zástupcovia vývoja vytvárajú celkový model systému. Znalci domény prezentujú pociatocný pohlad, poukazujúc na podstatné charakteristiky súvisiace s danou doménou, pomocou coho sa definuje pociatocný rozsah. Spolu so zástupcami vývojového tímu vytvárajú detailnejší pohlad. Znalci domény a zástupcovia vývoja pracujú v malých tímoch, ktorých výsledky sa zosumarizujú a vytvoria tak základný model. Tvorba celkového modelu prebieha pod dohladom hlavného architekta.

Tvorba zoznamu charakteristík

Na základe informácií získaných pocas tvorby celkového modelu sa skonštruuje komplexný zoznam charakteristík. Charakteristika predstavuje malú cast celkovej funkcnosti, ktorá je hodnotná pre zákazníka. Vyjadruje sa v tvare <akcia> <výsledok> <objekt>. Napríklad <vypocítaj> <celkovú sumu> <predaja>. Pri tvorbe zoznamu sa berú do úvahy zdokumentované požiadavky ako sú prípady použitia alebo funkcná špecifikácia. Ak nie sú tieto k dispozícii, použijú sa neformálne informácie z prvého kroku. Charakteristiky sa rozdelia do skupín podla funkcnosti. Pri velkých systémoch sa tieto skupiny môžu spájat do väcších celkov. Spolu so znalcami domény sa všetkým charakteristikám pridelia priority a definuje sa ich minimálna sada tak, aby predstavovala minimálnu funkcnost vhodnú pre zákazníka.

Plánovanie po charakteristikách

Pocas plánovania sa skupiny charakteristík, alebo ich celky v prípade väčšieho systému zoradia do casového harmonogramu a priradia sa hlavným programátorom. Triedy identifikované pocas tvorby celkového modelu sa priradia jednotlivým vývojárom.

Návrh a implementácia po charakteristikách

Hlavný programátor vyberie malú skupinu charakteristík, o ktorej predpokladá, že jej vývoj bude trvat jeden až dva týždne. Pocas návrhu po charakteristikách hlavný programátor identifikuje triedy, ktorých sa pravdepodobne vybrané charakteristiky týkajú. Tak vytvorí tím vývojárov, ciže vlastníkov tried, pracujúcich v danej iterácii. Vytvorený tím vypracuje sekvencný diagram pre vybrané charakteristiky a deklaruje potrebné triedy a metódy. Pred tým ako zacne implementácia po charakteristikách, vývojový tím vykoná inšpekciu návrhu. V priebehu implementácie po charakteristikách jednotliví vlastníci tried doplnajú definíciu deklarovaných tried a metód, vykonávajú testovanie a prípadnú integráciu dovtedy, kým nie je hlavný programátor spokojný. Hlavný programátor sa môže starat súcasne o dva až tri tímy a jeden vlastník tried môže byt clenom dvoch až troch tímov.

Porovnanie metód a prístupov

Vyššie opísané metódy a prístupy nedefinujú proces vývoja softvéru resp. charakteristiky tohto procesu na rovnakej úrovni, preto je ich možné kombinovat, ako v prípade DSDM a XP, kde DSDM definuje proces ako taký, nedefinuje však spôsob realizácie jednotlivých krokov postupu. Pri samotnej realizácii je tak možné, a casto aj vhodné, použit praktiky XP. V niektorých prípadoch sa však spomínané prístupy vylucujú. Je to tak v prípade XP a FDD, kde XP uprednostnuje kolektívne vlastníctvo kódu a naproti tomu objektovo-orientované praktiky FDD zavádzajú pojem „vlastník triedy“, ktorý je zodpovedný za danú triedu.

Manifest agilného vývoja obsahuje zoznam princípov, ktoré by sa mali dodržiavat pri agilnom vývoji softvéru. Pred vydaním manifestu existovali aj iné princípy, ktoré sa snažili prispiet k efektívnejšiemu vývoju softvéru. Takéto princípy definuje napríklad LP. Princípy ako také nedefinujú spôsoby, ktorými je možné ich dodržiavat, to je úlohou konkrétnych metód (DSDM, Scrum, FDD). Metódy pri realizácii môžu používat rôzne existujúce praktiky a postupy (XP), niekedy navyše definujú vlastné praktiky (Scrum, FDD).

Uvedené postupy môžeme porovnat aj vzhladom na manifest a jeho princípy.

Jednotlivci a interakcie pred procesmi a návodmi

LP, DSDM a Scrum zvýraznujú jednotlivca takým spôsobom, že dovolujú tímom samostatne rozhodovat s cielom splnenia stanovenej úlohy. Objektovo-orientované praktiky FDD uprednostnujú jednotlivca tým, že vlastník triedy je zodpovedný za stav danej triedy a vykonáva samostatné rozhodnutia týkajúce sa tejto triedy. Azda najväcším spôsobom je jednotlivec preferovaný pri praktikách XP, kde je kód kolektívnym vlastníctvom tímu a teda ktorýkolvek clen tohto tímu don môže zasahovat spôsobom, ktorý považuje za vhodný.

Funkcný softvér pred podrobnou dokumentáciou

Väcšina zo spomínaných prístupov (XP, Scrum, FDD, LP) sa snaží obmedzit tvorbu podrobnej dokumentácie, pricom ju nahrádza aktívnou komunikáciou zúcastnených na projekte. Funkcnost softvéru je pritom prioritou vo všetkých prístupoch. Fakt, že vývoj prebieha iteratívnym spôsobom casto zabezpecuje existenciu funkcného softvéru, ktorý je aspon v minimálnej miere použitelný pre zákazníka.

Spolupráca so zákazníkom pred uzatváraním zmlúv

Iteratívny spôsob vývoja, ktorý je použitý v uvedených prístupoch, dovoluje aktívnu úcast zákazníka na tvorbe softvéru tým, že

každá iterácia môže prechádzat procesom akceptácie zákazníkom (DSDM, Scrum, FDD). LP obsahuje princíp pestovania si dodávatelsko-odberatelských vztahov, ktorého dodržanie takisto umožnuje aktívne zapojenie sa zákazníka do procesu tvorby softvéru.

Odozva na zmenu pred dodržiavaním plánu

Potreba promptného zareagovania na zmeny bola prícinou vzniku všetkých opisovaných prístupov. Väčšina prístupov používa iteratívny spôsob vývoja softvéru, co umožnuje definíciu vstupného bodu požiadaviek od zákazníka na rozhraní iterácií.

Použitá literatúra

[1] Manifesto for Agile Software Development
http://www.agilemanifesto.org

[2] XProgramming.com: An Extreme Programming Resource
http://www.xprogramming.com

[3] Extreme Programming: A gentle introduction
http://www.extremeprogramming.org

[4] Maurer, F.; Martel, S.: Extreme Programming – Rapid Development for Web-Based Applications. IEEE Internet Computing, January/February 2002, p. 86-90.

[5] DSDM Consortium, Dynamic Systems Development Method
http://www.dsdm.org

[6] Scrum
http://www.controlchaos.com

[7] Rising, L., Janoff, N.: The Scrum Software Development Process for Small Teams
http://www.brock.dk/bsc/info3011/SCRUM.pdf

[8] Feature-Driven Development
http://www.togethercommunity.com/coad-letter/Coad-Letter-0070.html

[9] Palmer, S., Felsing, J.: Feature-Driven Development - Practices
http://vig.pearsoned.com/samplechapter/0130676152.pdf

[10] Poppendieck, M.: Lean Programming. Part 1. May 2001
http://www.sdmagazine.com/documents/s=731/sdm0105e/0105e.htm

[11] Poppendieck, M.: Lean Programming. Part 2. June 2001
http://www.sdmagazine.com/documents/s=730/sdm0106g/0106g.htm

Agilné metódy a ludia

Jozef Hudec

Agilné metódy sa oproti klasickým zameriavajú na ludí zainteresovaných v postupe viac ako na použité procesy. V tomto pohlade sú ludia najdôležitejší zdroj pri tvorbe systému, ale taký zdroj, ktorý má špecifické vlastnosti. Článok sa zaoberá analýzou ludských vlastností, vymenúva vlastnosti dôležité pre agilné metódy. Ukazuje možnosti, ako využit známe fakty o ludských schopnostiach, aby sme proces tvorby zefektívnili, uvažuje nad vhodnou velkostou tímu pre použitie agilných metód. Pohlad sa zameriava na ludí vo vývojovom tíme, zákazníkom sa venuje iba okrajovo.

Úvod

Možno najzaujímavejší pohlad, ktorý agilné metódy prinášajú, je pohlad na ludí zúcastnujúcich sa tvorby softvéru. Na softvérový systém sa dá nazerat ako na množinu vzájomne interagujúcich súciastok. Agilné metódy rozširujú tento pohlad aj na ludí, ktorí samotný softvér robia. Na rozdiel od softvérových súciastok sú ludia „súciastky“ nedeterministické, ich vlastnosti a správanie sa nedajú presne urcit , ale iba do istej miery predpokladat [1].

Co nám táto zaujímavá sebareflexia softvérových inžinierov prináša? Uvedomenie si, že softvér tvoria ludia, a preto sa treba na ludí orientovat, treba pochopit ich vlastnosti. Prvoradý nie je proces, alebo predmet práce, prvoradí sú ludia, ktorí prácu robia a prispôsobenie procesu práce tak, aby daní ludia prácu spravili co najefektívnejšie. Samozrejme, nejde o to, aby ludské záležitosti zatienili potreby projektu.

Pozerat na ludí cisto ako na súciastky by bolo chybou, každý clovek je odlišný, a teda takéto komponenty sú velmi variabilné. Navyše, oproti funkcionalite komponentu má clovek iba isté „schopnosti“, no a týmto slovom vyjadrujeme iba istú pravdepodobnost (hoci aj vysokú), že daný „komponent“ bude v procese vykazovat žiadané vlastnosti. Pri použití agilných metód sa preukazuje dôvera v schopnosti cloveka, a práve preto je nutné tieto schopnosti poznat.

Alistair Cockburn predpokladá, že „prvoradý ludský efekt“ pri tvorbe softvéru bude najskúmanejšou oblastou v budúcich desatrociach [1]. Už dnes môžeme badat orientáciu

na ludí ako na základný zdroj výrobkov a to sa netýka iba agilných metód a softvérových produktov. Oproti kapitalistickému pohladu na ludí ako na zložku kapitálu sa agilné metódy zameriavajú na cloveka ako na individualitu a nie ako na zamenitelnú súciastku.

Agilné metódy vyžadujú agilných ludí

Pozrime sa bližšie na ludské vlastnosti, ktoré nás pri použití agilných metód najviac zaujímajú. Nejde iba o to, aby „naši ludia“ mali dané vlastnosti. V súlade s orientáciou na povahu ludských zdrojov sa snažíme zároven urcit, aké vlastnosti majú mat metódy, aby sa využili známe ludské vlastnosti. Metódy totiž upravíme lahšie, ako ludí.

Cockburn a Highsmith požadujú, aby pracovníci boli priatelskí, talentovaní, zrucní a komunikatívni [2]. V iných prácach sa požadujú aj: prispôsobivost, zainteresovanost, angažovanost, kreativita [8].

Agilné metódy nevyžadujú nutne ludí s výbornými schopnostami, ako by sa mohlo na prvý pohlad zdat, vela projektov používajúcich agilné metódy skoncilo úspechom, aj ked v tíme boli málo skúsení ludia. Asi by sa tažko zostavoval celý tím z ludí vysokých kvalít, napokon, agilné metódy iba vylepšujú produktivitu a definujú iný pohlad na prácu tímu, nejde tu o prírucku na výber tých najkompetentnejších ludí. Skôr ako technické hladajú sa isté ludské vlastnosti. Pozrime sa teraz na to, aké vlastnosti hladáme a preco sú ony dôležité, ak používame agilné metódy.

Komunikatívnost

Všetky agilné metódy sa snažia znížit cas vývoja zlepšením nutnej komunikácie. Tam, kde sa to dá (a je to rozumné), sa používa rozhovor tvárou v tvár, prípadne aj bez tvorby dokumentácie. Pri Extrémnom programovaní sa dokonca nezvykne vytvárat ani zoznam požiadaviek, tieto sa udržujú v mysli tvorcov neustálou komunikáciou o systéme. Zase však treba použit agilný prístup: tvorit tolko dokumentácie, aby táto bola využívaná, nie iba odkladaná.

Dokumentácia je tvorená podla hesla: „Cielom nie je dokumentácia – cielom je porozumenie.“ A na ozajstné porozumenie je potrebné, aby komunikácia bola fyzická – tvárou v tvár. Je notoricky známe, že každé komunikacné médium má iba istú kapacitu a že najviac informácií si dvaja ludia dokážu odovzdat, práve ked spolu komunikujú pri tabuli. Dôležitý je pocet – dvaja – pre viac ludí sa znižuje efektivita takéhoto prenosu; taktiež spomenutá tabula je dôležitá, dokonca až natolko, že vjednom z projektov sa vytvorilo pravidlo: „Na všetkých dôležitých miestach vo firme majte tabulu.“ [1] Práve kvôli výhodám fyzickej blízkosti pri komunikácii sa dá spravit mnoho zjednodušení, ktoré sa pri agilných metódach dajú využit, ale práve kvôli tomuto je obmedzený rozsah projektu, co sa týka ludí.

Ludia si rýchlejšie odovzdajú myšlienky tvárou v tvár, ako pri písaní a cítaní dokumentácie. Viacero návrhárov „sediacich“ spolu dokáže vytvorit lepší návrh, ako keby pracoval každý sám. Podobne aj rozhovory so zákazníkmi a sponzormi prebiehajú hladšie, ked sa ich zúcastní viacero vývojárov, ako ked by každý z nich mal viest individuálny rozhovor.

Agilné metódy si vyžadujú ludí, ktorí sa neboja komunikovat, a tiež sa neboja povedat, kedy veciam nerozumejú a comu nerozumejú. Aj toto je jeden z aspektov „dobrej“ komunikácie a požadovanej „priatelskosti“. Ak sa vám pohodlne komunikuje, nebojíte sa ozvat s nejasnostami, lebo sa nebojíte, že „vyjdete na posmech“, ale viete, že nejasné bude vysvetlené.

Dokumentácie sa síce pri použití agilných metód vytvára menej, ale zato je viac cielená, preto sú hodnotnejší ludia, ktorí vedia cielene vysvetlit a zhodnotit veci - a najmä identifikovat, co bude potrebné pre další vývoj. Takisto komunikácia s nadriadenou vrstvou (prezentácia stavu projektu) by mala byt intenzívna.

Podobne aj technika programovania v párocH využíva komunikáciu dvoch ludí, aj ked nie za tabulou, ale za pocítacom. Aj z vlastnej skúsenosti viem, že ak mi pri programovaní pomáha ten správny clovek, som viac koncentrovaný, výsledný kód je citatelnejší, obsahuje menej chýb, rýchlejšie napredujeme. Na druhej strane, na takýto spôsob práce si treba zvyknút a byt otvorený, ved málokto má rád dozorcu za chrbtom. Ludský faktor je už raz taký, nie každý dokáže komunikovat s každým – sú dvojice, pri ktorých spolocné programovanie nebude fungovat, pretože sa nezhodnú, aj ked by trebárs každý z nich takúto techniku s hocikým iným zvládol.

V porovnaní s klasickými metódami sa pri agilnom prístupe požaduje, aby „naši ludia“ boli viac komunikatívni.

Samostatnost a zainteresovanost

Je tažké vyclenit jedinú ludskú vlastnost, ktorá by definovala, preco sa agilné metódy dostali práve do svojej terajšej podoby. Ide zrejme o niekolko základných právd o cloveku: [1]

- ludia sú zvycajne radi „dobrými obcanmi“
- vždy sa nájde niekto, kto preberie iniciatívu
- ludia majú dobrú schopnost rozhladu

Vela projektov sa spätne hodnotí ako úspešných len preto, že sa pocas práce vyskytlo „zopár dobrých mužov“ (alebo žien, samozrejme). Ak jadro tímu tvoria zainteresovaní, angažovaní ludia, ktorí vedia prebrat iniciatívu - „potiahnut“ projekt, odstráni sa týmto vela negatívnych prejavov, ktoré cakajú takmer každý projekt: opadnutie záujmu clenov, únava, ponorková choroba. Ale dobrí tahúni ešte nerobia dobrých obcanov, je dôležité zostavit tím tak, aby „mesto“ nebolo roztrieštené ani etnicky, ani názorovo. Toto je prostý fakt odpozorovaný z praxe, kde sa vyskytlo viac prípadov, že tím bol rozdelený na viac skupín a nikdy sa nedal dokopy tak, aby mohol súdržne fungovat [1].

Talent, zrucnost, kreativita

Larry Constantine napísal: „Všetky agilné metódy nadpriemerne profitujú z nadpriemerných ludí.“ [3] Agilné metódy umožnujú talentovaným jedincom odbremenenie od únavných cinností a súcasne dávajú možnost ovplyvnit svojím talentom zvyšok tímu. V [3] sa tiež spomína: „Osobnosti lahko dominujú projektu.“

Talent, ktorý ludia prejavia a ktorý sa odrazí v uznaní okolitých osôb, môže byt silným motivacným faktorom projektu. Klasicky sa talent ocenuje, pri agilnom prístupe sa však takmer až požaduje.

Prispôsobivost, nekonzervatívnost

Agilné metódy od svojho jadra predpokladajú zmenu pocas celého životného cyklu softvéru - menia sa požiadavky, pracovný proces,

prepracúva sa návrh... Preto aj ludia zaclenení do tohto procesu by mali vediet zmeny akceptovat. Je prirodzené, že velké zmeny nevítame. Ako ludia sme prirodzene konzervatívni (asi nás k tomu vedie pud sebazáchovy – naco sa vrhat do neznáma...). Ludia neradi menia svoje zvyky, neradi pracujú podla nových schém.

Avšak ked sa ukáže potreba zmeny v našom procese tvorby, je potrebné ju vykonat. Niekedy to chce iba trocha sa premôct. Stým súvisí aj prispôsobovanie sa zvyšku tímu, ak chceme efektívne pracovat, niektoré veci treba mat dohodnuté rovnako. Napríklad pri tvorbe kódu je toto pravidlo dôležité, hlavne ked si uvedomíme, že dokumentácie sa tvorí menej, ako zvycajne, takže niektoré veci, ktoré clovek môže implicitne predpokladat (kedže boli explicitne dohodnuté), dokážu zjednodušit prácu.

V pravidelných intervaloch tím zhodnotí, aké zmeny treba spravit, aby jeho cinnost bola efektívnejšia. Návod na to neexistuje, ale spravidla samotné metódy obsahujú pravidlá co a kedy urobit, aj ked len do istej úrovne podrobnosti. O tom je samotná podstata agilnosti – problém sa rieši *flexibilne* vzhladom na svoj charakter a na vlastnosti riešitelského tímu. Ale je to riadený postup - postupuje sa podla metodológie (alebo podla pravidiel), iba je to inak definovaný postup, ako pri vývoji riadenom plánom.

Disciplína

Ak postavíme vedla seba na ciaru agilné metódy, „kovbojský štýl“ a plánom riadený vývoj, agilné metódy budú dost blízko „kovbojom“ („hackerom“ v pôvodnom zmysle tohoto slova) [6]. Kovbojský štýl však nie je našim cielom, aj ked sú agilné metódy menej striktné, neumožnujú úplne „volnú jazdu“. A aj ked sme bližšie ku kovbojom, nesmieme zabúdat na to, že aj agilné metódy majú svoje pravidlá a disciplína je preto dôležitá.

Agilitu nicí prepracovanost. Clovek nemôže efektívne pracovat pridlhý cas, oddych musí byt súcastou prípravy na prácu. Agilné metódy sa spoliehajú na ludí, ktorí sú pozorní a kreatívni a dokážu si tieto vlastnosti zachovat po dlhý cas, nie sa „vyštavit“. Je dobré nájst hranicu (pocet hodín), ktoré tím venuje práci a jeho vlastnosti zostanú zachované. Napríklad extrémne programovanie definuje túto hranicu ako 40 hodín týždenne. Cielom je doviest projekt stabilným tempom k jeho výsledku a nie „pochod smrti“.

Spolahlivost , zodpovednost

Dôvera v ludí je v agilných metódach silným predpokladom. Aj ked komunikácia dokáže odhalit dost skoro, že clovek nerobí to co má, samotné odovzdanie dôvery je silne motivujúcim krokom a aj predpokladom, vdaka ktorému fungujú niektoré princípy – napríklad menšia dokumentácia. Veríme našim pracovníkom, že dokážu zhodnotit, co je pre zákazníka dôležité v kontexte nákladov na vytvorenie a neprecenit pritom vlastné sily.

Sme iba ludia...

Okrem požadovania istej úrovne pozitívnych vlastností je dôležité uvedomit si, že ludia majú aj svoju negatívnu stránku, „zlé“ vlastnosti (zlé z pohladu procesu tvorby softvéru). Sú to však fakty, sktorými treba rátat, nie ocakávat, že sa ludia zmenia (nieto ešte im to prikazovat). V súvislosti s agilnými metódami sa takto spomínajú hlavne konzervatívnost a „opakovatelnost správania“ [1].

Ako sme si definovali naše „zdroje-ludí“ – sme nespolahliví a nedeterministickí. Dajte cloveku urobit nejakú zložitejšiu vec viackrát, ani raz ju neurobí úplne rovnako [4]. Tento fakt musí metodológia (alebo jej tvorca) brat do úvahy, napríklad programovanie v pároch vylepšuje výsledok, hoci tento aj tak zostáva v case neopakovatelný. Ludia nie sú radi, ked majú prikázaný striktný proces, je vich prirodzenosti robit veci „po svojom“.

Na to, aby sme sa vyrovnali s takýmito vlastnostami ludí, existuje vela dalších techník a postupov (používajú sa nielen v agilných metódach), navyše sa do súvisu dáva aj dôvera v schopnosti ludí – povedzte ludom, co majú urobit a nechajte ich, aby sa rozhodli, ako to spravia.

Tímy

Ludia majú dôležité vlastnosti pre prácu v tíme a inú skupinu vlastností, ktoré sa zvycajne uplatnujú samostatne. Agilné metódy umožnujú vylepšit tieto individuálne vlastnosti tímovou prácou. Napríklad programovanie v pároch vylepšuje kvalitu kódu a rýchlost jeho vytvárania, kolektívne zdielanie kódu, umožnuje odstránit urcitú komunikacnú bariéru. Táto technika je však urcená pre práve dvoch ludí (pri väcšom pocte zúcastnených

efektivita prudko klesá), podobne sú na tom viaceré metódy, ale co ked rozsah projektu požaduje využit viac pracovníkov?

Aj ked stále hovoríme o tíme, je jasné, že velký pocet ludí sa nebude správat ako malý tím, bude klesat komunikatívnost, budú stúpat nároky na dokumentáciu a podobne. Co potom s metódami, ktoré ako svoj základ požadujú vysoké porozumenie?

Agilný vývoj je vo velkých tímoch zložitejší. Pre malé tímy je atraktívny, ale ak pocet ludí v tíme rastie nad 30, prestávajú tieto metódy mat zodpovedajúci základ. Aj ked vniektorých prípadoch agilného vývoja boli úspešné aj tímy, ktoré mali 250 ludí [2]. Záleží zrejme na type projektu a na zúcastnených ludoch – projekt môže prosperovat aj za pomoci agilných metód, ak sa v tíme nachádza viacero vodcovských osobností, ktoré svojou agilitou urcujú smer a udržujú chod projektu. Pre toto zatial neexistujú žiadne pravidlá, iba pozorovania z praxe, na ktorých sa ukazuje, že malé tímy preferujú a úspešne používajú agilné metódy. Kedže agilné metódy preferujú komunikáciu tvárou v tvár, stúpa komunikacná bariéra pri väcších tímoch, ak každý chce komunikovat tvárou v tvár. Toto je možné do istej miery eliminovat, ked agilná metóda adoptuje nejakú pomôcku z plánom riadených metód. Napríklad správne udržiavaná nástenka vo fyzickej, alebo virtuálnej podobe umožní redukovat množstvo informácií, ktoré by bolo tažké odovzdat tvárou v tvár.

Na väcších tímoch by bolo zaujímavé sledovat použitie meta-metodík, ako napríklad „Konštrukcia metodológie na mieste“ [5], ktoré navrhujú, ako vytvorit metodológiu urcenú priamo pre daný projekt a daných ludí, urcit, kto s kým má komunikovat. Podla môjho názoru by cesta pre velké tímy mohla viest aj tadialto, pokial sa definuje štruktúra komunikácie zodpovedajúca velkosti tímu.

V [5] autor okrem iného navrhuje tvorit metodiku priamo pre konkrétny projekt na základe skúseností z predchádzajúcich projektov podobného typu, ale tiež aj zo skúseností priamo z prebiehajúceho projektu. Požaduje zároven aspon dvakrát za projekt urobit revíziu postupov použitých v projekte.

Autor zároven porovnáva takýto postup s „rodinou“ metód Crystal [7], podla ktorých sa na konkrétny projekt uplatnuje práve taká zo skupiny metód, ktorá je najvhodnejšia – uvažuje sa podla typu, rozsahu projektu, atd. V skupine Crystal sú definované metódy aj pre projekty, na ktorých sa môže zúcastnit až do 80 ludí.

Manažovanie ludí

Morálka a kooperácia tímu sú prvé veci, ktoré by sa mali sledovat. Kooperácia je nevyhnutná a je základom pre všetko, preto je nutné zabezpecit, aby fungovala. Popri silnej požiadavke na komunikáciu a komunikatívnost netreba zabudnút na to, že clovek potrebuje cas aj na samostatné myslenie a konanie.

Morálka je tiež predpokladanou velicinou pocas práce na projekte, takisto ako pri klasických metódach.

Ludia sa musia naucit hladat rezervy, spôsoby práce, motiváciu sami v sebe. Nie je to len úlohou manažmentu, agilnost ludí je aj v hladaní v sebe. S tým je spojená maximálna dôvera pracovníkom, cielom je zostavit projektový tím na základe niekolkých schopných ludí, dat im prostredie a podporu v práci a dôverovat im, že odvedú dobrú prácu. Dôležité rozhodnutia musia urobit ludia, ktorí vedia o situácii najviac, manažéri musia dôverovat pracovníkom, napokon, tí sú platení za to, aby o danej situácii vedeli co najviac. Tento prístup by som vzhladom na klasické pracovné procesy prirovnal kotvoreným a zatvoreným ociam: klasicky cestuje rozhodnutie zhora nadol. Manažéri sa síce rozhodujú podla informácií zdola, avšak ludia dole už nedostanú vysvetlenie, preco došlo práve k takémuto rozhodnutiu. Pri použití agilného prístupu musia mat všetci pracovníci otvorené oci. Ale nie v každom projekte je možné použit agilný prístup všetkých zúcastnených. Produkt pri agilnom prístupe tvoria ludia špecifických kvalít, ktorým sa oplatí dôverovat a ktorí si zaslúžia byt castou procesu rozhodovania. Popri tom sa neznižuje význam kontroly pracovníkov, ale táto ciastocne prechádza na nich aj so zvyšujúcou sa zodpovednostou. Na zníženie rizík, spojených s nedostatocnou viditelnostou procesu z pohladu manažéra sú vytvorené v mnohých procesoch mechanizmy a techniky - napr. SCRUM obsahuje pravidelné schôdze, kde každý clen tímu hovorí o dosiahnutom pokroku a s cím má problémy.

Pre manažéra je dôležité pýtat sa svojich ludí, ako napreduje projekt (okrem iných metód merania postupu projektu). Dobrým pravidlom je, aby sa všetci snažili dat co najviac realistický odhad, nie aby sa snažili potešit manažéra správami o dobrom postupe projektu. Predpokladajme teda, že sa budú snažit byt realistickí, ale aj tak každý clen bude mat iný odhad. Ak je v tíme vela optimistov alebo pesimistov, celkový odhad bude posunutý oproti ideálnemu. Dôležité je vediet, aké zloženie „presvedcení“ je v našom tíme, aby sa jednotlivé subjektívne pohlady mohli spojit do objektívneho. Takto by sa zdalo výhodné

mat tím iba z realistov, na druhej strane je však dobré, ak máme v tíme silnejších pesimistov aj optimistov, títo najskôr môžu upozornit na slabé aj silné stránky vyvíjaného softvéru, ale aj procesu vývoja. (Alebo skôr upozornia na hranice odhadov.)

Zákazník

Vztahy so zákazníkmi musia byt svojím spôsobom tiež agilné a to na oboch stranách – vývojový tím aj zákazník. Softvér sa produkuje na základe želaní zákazníka a ak sa tento nebude správat agilne, tažko možno na základe jeho práce agilne vyvíjat systém. (Bližšie sa vztahmi so zákazníkmi zaoberá iná esej v tejto publikácii.)

Záver

Agilné metódy nie sú všeliekom na problémy vznikajúce pri tvorbe softvéru. Sú vhodné pre isté typy projektov a inak definujú požiadavky na ludské zdroje. Oproti klasickým metódam, kde prevažujú technické požiadavky na schopnosti ludí, tu sú nevyhnutné aj ludské kvality.

Takisto je podstatná aj velkost tímu - je málo pravdepodobné, že agilné metódy budú použitelné pre velké tímy (aj ked toto závisí od zavedeného stupna „agilnosti“ v skupine metód aplikovaných v projekte).

Ludia sú nepochybne zaujímavý „zdroj“ pri vývoji softvérového systému. Je dobré, že zo samotnej komunity expertov–informatikov vyšlo volanie po lepšom porozumení samým sebe. Máme už za sebou istú históriu, pocas ktorej sme sa snažili porozumiet samotnej podstate procesu tvorby softvéru. Aj ked sa vždy pri tvorbe systémov nazeralo aj na ludí, ich kvality a ich spojitost s procesom tvorby, až agilné metódy a ich potreba v dnešnej dobe nám odhalujú vela z našich vlastností. Tieto metódy priamo na niektorých vlastnostiach stavajú a na nich majú založenú svoju agilitu.

V dobe, ked sa paradoxne zvyšuje kapacita technických komunikacných kanálov, ale ludská komunikácia slabne, je príjemné vidiet pohlad, ktorý navrhuje využit v ludoch to dobré a „posadit ich spolu do jednej miestnosti.“

Použitá literatúra

[1] Cockburn, Alistair: „Characterizing People as Non-Linear, First-Order Components in Software Development", Humans and Technology Technical Report, október 1999 http://members.aol.com/humansandt/papers/nonlinear/nonlinear.htm

[2] Cockburn, A. – Highsmith, J.: „Agile Software Development: The People Factor", in *Computer*, november 2001, str. 131-133

[3] Constantine, L.: „Methodological Agility", in *Software Development*, jún 2001, str. 67-69

[4] Highsmith, J.: „Adaptive Software Development", Dorset House, 1999

[5] Cockburn, Alistair: „Just-In-Time Methodology Construction", Humans and Technology, 2001 http://members.aol.com/humansandt/papers/jitmethy/jitmethy.htm

[6] Boehm, B.: „Get Ready for Agile Methods, with Care", in *Computer*, január 2002, str. 64-69

[7] Cockburn A., „Crystal(Clear): A human-powered software development methodology for small teams", Addison Wesley, 1999 http://members.aol.com/humansandt/crystal/clear http://crystalmethodologies.org

[8] Domino, M. – Collins, R.W. - Hevner, A.: „Software Testing Processes in Pair-Programming", 2001 http://www.coba.usf.edu/departments/isds/nistp/behavioral/docs/pairprogrammingWP.doc

Vzt ah k zákazníkovi v agilných metódach

Vladimír Grlický

Oproti klasickým metódam vývoja sa pri použití agilných metód dostáva zákazník do nového postavenia. Z úlohy štatistu, pozorovatela procesu sa dostáva do úlohy spolutvorcu budúceho diela. Článok je pohladom na vztah programátor – zákazník s dôrazom na zákazníka, najprv v porovnaní agilného prístupu s prístupom klasickým, neskôr je tento vztah bližšie predstavený cez praktiky Extrémneho programovania. Na príklade XP sú opísané jeho nové možnosti ovládania vývoja softvéru, no i cinnosti, ktoré od neho agilné metódy vtomto procese ocakávajú. Článok sa zaoberá aj problémami, ktoré agilné metódy do tohto vztahu prinášajú a snaží sa nacrtnút ich možné riešenia.

Úvod

Oproti klasickým metódam vývoja sa pri použití agilných metód dostáva zákazník do nového postavenia. Z úlohy štatistu, pozorovatela procesu sa dostáva do úlohy spolutvorcu budúceho diela. To samozrejme kladie nové nároky na ludí a funkcie, ktoré reprezentujú prienik zákazníka s clenom vývojového tímu. Predmetom clánku je aj pripravenost prostredia na takúto zmenu. Snahou bude poukázat na nové možnosti a nástroje, ktoré sa ako zákazníkovi tak aj programátorovi použitím agilných metód dostali do rúk, no zároven tiež zdôraznenie nových úloh ci ocakávaní, ktoré tento agilný prístup kvývoju softvéru vztahu programátor - zákazník prináša. Článok bude tiež akýmsi subjektívnym pohladom na tento vztah, bude sa snažit poukázat na najväcšie problémy a nacrtnút možné riešenia.

V nasledujúcich kapitolách podrobnejšie opíšem nový, tesnejší vztah medzi programátorom a zákazníkom. Zákazník už nebude len akousi externou entitou, pasívnym pozorovatelom vývoja systému, stane sa zneho aktívny prvok tohto systému, ktorý sa bude musiet takisto spolupodielat na tvorbe budúceho diela. Agilné metódy mu vkladajú do rúk nové nástroje, ktoré mu dávajú moc usmernovat vývoj projektu. Tieto nástroje však vyžadujú aj isté zrucnosti, ktoré tradicné metódy od zákazníka nevyžadovali, a tým ho nútia výraznou mierou zmenit svoje

doterajšie správanie. Agilný prístup k vývoju softvéru teda vyžaduje aj nový typ zákazníka – nazvime ho agilný zákazník.

Pri vzájomnom porovnávaní softvérových produktov, pri vývoji ktorých boli použité tradicné metódy, sa najcastejšie používajú tri vlastnosti, ktorými sú cas, cena a rozsah. Dôvod ich použitia je evidentný – jasne charakterizujú softvérový výrobok ako celok, slúžia ako metafora, ktorou sú oznacené najdôležitejšie ciele procesu, na konci ktorého vznikne výsledný softvérový systém. Agilné metódy navyše pridávajú k týmto charakteristikám ešte jednu – jeho kvalitu, a tým akoby pridávali hodnote softvéru štvrtý rozmer. Ludia okolo agilného vývoja softvéru tieto štyri charakteristiky nazývajú tiež „štyri premenné“ s dôrazom na slovo „premenné“, cím chcú naznacit možný omyl, ktorého sa dopúštajú tradicné metódy, ked cas, cenu, rozsah ci kvalitu považujú za fixné parametre hodnotenia softvéru.

Úloha zákazníka v procese vývoja

Aby sme mohli lepšie pochopit rozdiely medzi tradicným zákazníkom a zákazníkom agilným, bolo by dobré osviežit si tradicný prístup k vývoju softvéru:

Zákazník (jednotlivec alebo skupina ludí) sa dohodne s tímom vývojárov, že pre projekt urcia analytika. Pocas niekolkých týždnov až mesiacov sa analytik stretáva so zákazníkom na niekolko hodín týždenne. Jeho úlohou je zostavenie série dokumentov, ktoré sa snažia podat detailný obraz o požadovanom systéme. Zákazník (a samozrejme aj programátori) si tieto dokumenty prezrie a odsúhlasí. Okrem samotného špecifikovania požiadaviek sa v tejto fáze obidve strany dohodnú na cene za dodanie celého systému, ako aj na case jeho uvedenia do prevádzky. Tu úloha zákazníka koncí a nastupujú programátori, ktorí si tieto informácie zoberú a o niekolko mesiacov neskôr (presnejšie o vopred dohodnutý cas) vytvoria systém, viac alebo menej podobný ocakávanému. Nie je výnimkou, ak ku koncu vývoja zákazník zistí, že na nieco zabudol a uvedomí si, ako sa zmenilo prostredie budúceho systému od vytvorenia spomínaných dokumentov. Vývojári potom musia zapracovat tieto zmeny do systému. Nakoniec zákazník vykoná akceptacné testy a ak vyvíjaný systém nimi „prejde“, považuje sa za dokoncený a je uvedený do prevádzky. Casto však celý tento proces trvá ovela dlhšie ako sa ocakávalo a kvalita systému tiež nie je taká, akú chcel zákazník mat.

Výsledkom sú vycerpaní a znechutení programátori a nespokojný zákazník.

Preco došlo k takejto situácii? Kde treba hladat chybu? Podla mna jedným zhlavných dôvodov neúspechu pri použití tradicných metód vývoja softvéru je nekompletnost požiadaviek, a tiež nepredvídatelnost ich zmeny pocas samotného vývoja softvéru.

Je takmer nemožné špecifikovat všetky požiadavky už na zaciatku prác na projekte. Tu môže byt problémom samotná velkost plánovaného systému, pri ktorej mnohokrát zlyháva ludská predstavivost (hlavne tá zákazníkova). Výsledkom býva slabá špecifikácia požiadaviek, ktorá ponúka programátorom velkú volnost v ich došpecifikovaní (a prácu naviac). Tí sa však nie vždy „trafia“ do nevyslovených požiadaviek zákazníka, co v konecnom dôsledku vyústuje do nespokojnosti oboch zúcastnených strán. Neželaný efekt ale prináša aj snaha o co najpodrobnejšiu špecifikáciu. V tomto prípade môže dôjst k znechuteniu zákazníka (hlavne technicky menej zdatného), od ktorého sa vyžaduje príliš vela podrobných informácií už na zaciatku životného cyklu projektu. Znechutená však býva aj druhá strana a nezriedka možno pocut po stretnutiach so zákazníkom z úst programátorov vety typu „Zákazník nevie, co chce.“ Další vývoj má podobný spád ako v predchádzajúcom prípade.

Takmer každý softvérový systém je vyvíjaný s cielom urýchlit, zefektívnit, zautomatizovat, jednoducho ulahcit prácu ludí pri riešení problémov reálneho sveta. Na to, aby tieto problémy mohol systém riešit co najpresnejšie, musí mat vytvorené rovnaké podmienky alebo podmienky aspon blížiace sa tým v reálnom svete. Tieto (samozrejme nie všetky) podmienky sa však v reálnom svete s casom menia, a mnohokrát sa menia práve pocas vývoja systému. Za vytváranie podmienok sú zodpovední programátori, no ich zmenu by mal sledovat práve zákazník, ktorý dostatocne rýchlo o tejto zmene vývojový tím informuje, a tým zamedzí „starnutiu“ softvéru už pocas jeho vývoja. A práve tu tradicné metódy na tesnejšiu spoluprácu so zákazníkom akosi pozabudli, ci skôr neboli dostatocne agresívne v jeho zaangažovaní aj do dalších etáp životného cyklu projektu. Spolupráca zákazníka s tímom vývojárov vychádzajúca z jeho vlastnej iniciatívy a „vloženie“ sa do vývoja softvéru je skôr výnimkou ako pravidlom. Zákazník sa totiž casto mylne domnieva, že v dalších fázach projektu už nie je potrebný. Nie vždy si však dokáže uvedomit, že je to práve on, od ktorého tím vývojárov ocakáva informácie, tak potrebné pre úspešné vytvorenie softvérového systému. Tieto informácie sú však

nevyhnutné nielen pri zapocatí prác na vývoji softvéru, takisto dôležité sú aj v jeho priebehu.

Agilné metódy odpovedajú na obidva problémy (nekompletnost a nepredvídatelnost zmeny požiadaviek) na prvý pohlad jednoduchým spôsobom, a to zainteresovaním zákazníka aj do dalších fáz vývoja projektu. Pod zainteresovaním treba rozumiet aktívnu spoluprácu zákazníka s vývojovým tímom, jeho tesnejšie previazanie s clenmi vývojového tímu, v ideálnom prípade sa zákazník stáva dokonca clenom tohto tímu a zotrváva v nom až do ukoncenia projektu a jeho uvedenia do prevádzky.

Jedným z pozitívnych efektov zapojenia zákazníka do všetkých castí vývoja je jeho odbremenenie od pociatocnej zátaže, ktorá vyplýva z detailnej špecifikácie požiadaviek v úvodnej fáze životného cyklu projektu. Tu zákazník priebežne špecifikuje požiadavky a urcuje tie funkcné body, ktoré sú mu v danej fáze známe alebo majú najvyššiu prioritu. Zákazník má tiež právo kedykolvek pocas vývoja pridávat nové, odstranovat už naplánované funkcie systému ci podla potreby menit priority dalšieho vývoja jednotlivých funkcií. Len ako poznámku uvádzame, že slovo *kedykolvek* má takmer v každej metóde agilného vývoja softvéru iný casový interval (napr. u metódy Scrum po 30 dnoch, v XP pri použití tzv. *zákazníka priamo na mieste* je to naozaj kedykolvek). Splnenie týchto požiadaviek zákazník kontroluje na pravidelných stretnutiach formou testov. Ak vyvíjaný systém „prejde“ aj akceptacným testovaním, je uvedený do prevádzky.

Štyri premenné

Vychádzajúc z výroku „Ak nemožno merat, nemožno ani riadit.“ [1] nám vyplýva, že ak chceme softvérový projekt viest, musíme ho vediet urcitým spôsobom merat a hodnotit. Projekt môžeme hodnotit z rôznych uhlov pohladu, podla rôznych kritérií, na rôznych úrovniach podrobností. Výber konkrétnych vlastností, charakteristík ci atribútov, ktoré si na projekte budeme všímat, však vždy záleží aj od množstva dalších faktorov (kto sa o merané výsledky zaujíma, v akom prostredí sa tieto charakteristiky merajú a podobne). Ak sa na softvérový projekt budeme pozerat ako na tovar, ktorý treba dodat zákazníkovi, základnými premennými pre hodnotenie tohto projektu urcite bude štvorica kritérií - cas, cena, rozsah a kvalita. Významnú úlohu pri ich výbere má aj fakt, že so spomenutými kritériami sa do kontaktu dostal takmer každý, ci je

to programátor, vedúci projektu, manažér alebo zákazník, a teda ich význam netreba dalej podrobne vysvetlovat.

Potrebuje zákazník dostat vyvíjaný systém do prevádzky v co najkratšom case? Alebo je limitovaný obmedzeným rozpoctom? Potrebuje mat v systéme množstvo funkcií alebo ich môže byt menej ale podrobne prepracovaných? Hladanie a nájdenie odpovedí na tieto a množstvo dalších podobných otázok takmer nikdy nepatrí medzi jednoduché záležitosti, a to z jednoduchého dôvodu. Spomenuté charakteristiky nie sú totiž navzájom nezávislé (niekedy sú dokonca protichodné). Je samozrejmé, že ak chcem dosiahnut nejaký ciel daného rozsahu v krátkom case, bud oželiem kvalitu, alebo investujem - pridám zdroje (viac ludí, viac penazí) a pod. Nájdenie odpovedí na vyššie spomenuté otázky je teda mnohokrát založené na dohode medzi zákazníkom (jeho požiadavkami) a programátorom (jeho reálnym pohladom na problém). Ak by sme mali zapísat výsledok ich dohody matematicky, mal by tvar:

hodnota projektu = f(cas, cena, rozsah, kvalita)

co znamená, že hodnotu projektu môžeme vyjadrit ako funkciu štyroch premenných. Ako však možno regulovat jednotlivé premenné tejto funkcie? Kto má v rukách nástroje na ich ovládanie?

V agilných metódach je odhad casu potrebného na vývoj systému, ci jeho casti plne v rukách vývojového tímu. Spolu s casom do ich kompetencií vstupuje aj cena, ktorú bude musiet zákazník zaplatit za implementovanie požadovanej casti systému.

Pod rozsahom mám na mysli velkost celého systému, ktorá sa casto stanovuje ako pocet funkcných bodov, ktoré sú potrebné alebo ktoré sa podarilo splnit. Práve v korigovaní tejto vlastnosti systému má zákazník ovela väcšie možnosti ako u tradicných metód. Tam bol rozsah projektu špecifikovaný v úvodných fázach jeho životného cyklu a programátori boli tým clánkom, ktorý vprípade sklzu projektu voci plánu rozhodoval o tom, do akej miery a co bude vytvorené. Agilné metódy dovolujú zákazníkovi plne ovládat rozsah vyvíjaného softvéru, a to nielen v jeho finálnej fáze, ale aj v jednotlivých iteráciách stanovením priorít jednotlivých funkcných bodov.

Treba si však uvedomit, že proces stanovenia rozsahu, ceny za implementovanie rozsahu, a casu potrebného na jeho implementovanie nie je jednoprechodový. Naopak, výsledkom tohto procesu je zvycajne kompromis medzi zákazníkom ako

zadávatelom úlohy a programátorom, jej vykonávatelom, ktorý má sám niekolko iterácií.

Ako príklad riedenia jednotlivých charakteristík softvéru možno uviest Extrémne programovanie, ktoré považuje za dôležité správne pochopenie úloh ako zákazníka tak aj programátora. Ak by sme mali ich úlohy zjednodušene vyjadrit pár slovami, výsledok by bol nasledovný - zákazník zodpovedá za to „co chceme“, kým programátor rozhoduje o tom, „kolko to bude stát“[5].

Zákazník teda rozhoduje:

- *o rozsahu* – co daný systém musí robit
- *o priorite* – co je viac alebo naopak menej dôležité
- *o zložení jednotlivých verzií softvéru* – co musí daná verzia obsahovat, aby bola použitelná
- *o dátume uvedenia do prevádzky* – kedy je „uvolnenie“ vybranej verzie softvérového systému žiadúce

Programátor naopak rozhoduje:

- *o odhade casu*, ktorý je potrebný na pridanie danej funkcie do systému
- *o technických dôsledkoch* rozhodnutia zákazníka (výsledné rozhodnutie o dalšom postupe je ale stále na zákazníkovi)
- *o procese* – o tom, ako bude tím pracovat,
- *o podrobnom plánovaní* v rámci iterácie

Rozdelenie zodpovedností pomáha celému vývojovému tímu držat sa tej „správnej“ cesty, a to okamžitým vytváraním technických dôsledkov rozhodnutí zákazníka. Ak napríklad zákazník chce dostat správu o projekte tento týžden, vývojový tím ju bez problémov vypracuje. Najprv sa však naznacia všetky riziká spojené s oddialením prác na projekte, a taktiež sa odhadne, co to bude stát. Potom znovu nastupuje zákazník a rozhoduje o tom, co bude z prác na vývoji softvéru vypustené, prípadne odložené na neskôr. Samozrejme, že v jeho kompetencii je aj prehodnotenie a následné stiahnutie tejto požiadavky, ci už z dôvodu vysokej ceny alebo neakceptovatelných rizík.

Co však robit, ak nastane konflikt? Co ak zákazník požaduje implementovanie novej funkcie do urceného dátumu a programátori odhadnú, že to bude trvat ovela dlhšie? Tu sa ponúka hned niekolko riešení:

- zákazník zníži rozsah systému
- zákazník posunie dátum ukoncenia danej funkcie
- zákazník vydá viac casu alebo penazí na hladanie alternatívy
- zákazník nájde iný vývojový tím

Hoci posledné riešenie je trochu agresívne, uvedený príklad ilustruje polohu, do ktorej sa zákazník dostáva v agilných metódach v case vývoja systému – polohu cloveka, ktorý rozhoduje o dalšom smerovaní projektu.

Cinnosti zákazníka

Väcšina agilných metód je založená na iteráciách a inkrementálnom vývoji projektu, všetky metódy v ich manifestoch hlásajú ovela castejší a tesnejší kontakt so zákazníkom ako je tomu umetód tradicných, nie všetky sa však rovnako detailne zameriavajú na opísanie vztahu programátor – zákazník. Pre niektoré metódy stací, ak sa zákazník aktívne zúcastnuje len na dopredu dohodnutých pravidelných stretnutiach (Scrum), iné vyžadujú od zákazníka byt nápomocný pri vývoji po celý jeho pracovný cas (XP). Aj tu je možno vidiet, že nájdenie toho najlepšieho riešenia je velmi komplikované, a ocividne vo velkej miere závisí aj od povahy samotného projektu. Predmetom tohto clánku je však vzájomný vztah programátor - zákazník. Kedže týmto vztahom sa najdetailnejšie zaoberá klúcový súbor techník agilných metód - Extrémne programovanie, rozhodol som sa priblížit vzt ah programátor - zákazník a poukázat na prednosti no i nedostatky tohto vztahu práve cez XP.

Extrémne programovanie prináša do vývoja softvéru množstvo novátorských myšlienok a radikálnych postupov, výnimkou nie je ani prístup k zákazníkovi a vymedzenie jeho úloh. Tu slovo „extrémne“, ktoré sa nachádza v samotnom názve metódy, je naozaj na mieste, kedže zákazníka presúva z polohy pasívneho diváka do úlohy aktívneho clena vývojového tímu – a vyžaduje od neho plnú úcast na vývoji – potrebuje ho priamo na mieste vývoja.

Vari najradikálnejšou zmenou oproti tradicnému chápaniu úlohy zákazníka v procese vývoja softvéru je jeho postavenie priamo do centra „diania“, a tým vytvorenie novej pozície – tzv. zákazníka priamo na mieste. Je to zákazník, ktorý spolupracuje s tímom pocas celej pracovnej doby. Sedí s ostatnými clenmi

vývojového tímu, píše testy, odpovedá na otázky a urcuje priority. Ak chce tím napredovat tak rýchlo, ako je to len možné, zákazník priamo na mieste je klúcovým faktorom k dosiahnutiu tohto ciela.

Problémy?

Takmer žiadne riešenie nie je absolútne správne aaj tu možno nájst situácie, kedy môžu nastat problémy. Extrémne programovanie predpokladá plnú zákazníkovu prítomnost pocas celej práce na projekte. Jedným z castých problémov je však neochota zákazníka byt k dispozícii celý pracovný cas. Co vtedy robit? Podla Toma Petersa „Ak vám klient neposkytne jeho najlepších clenov na plný pracovný cas, odstúpte od projektu. Klient nie je seriózny.“ [2]. Toto síce môže byt pravda, no co ak nie je možné v tejto fáze od projektu odstúpit? Alebo je projekt pre nás velmi dôležitý a nechceme ho stratit?

Zákazník sa casto doslova hrozí situácie, ked by mal byt ešte tesnejšie zatiahnutý do vývoja softvéru, nechce byt totiž konfrontovaný s jeho vlastnou neznalostou technológie, a najmä si nechce pripustit, že neposkytuje vývojárom dostatok informácií. Takéto myslenie je ale nesprávne. Ved znalost technológie je predsa to, kvôli comu sú tu programátori, ovládanie domény – problémovej oblasti – je dôvod, kvôli ktorému je v procese vývoja potrebný práve zákazník [4].

Ako teda presvedcit zákazníka, že cas, ktorý venuje vývojárom, je jeho profitom a nie stratou? Ak ste nedosiahli prítomnost zákazníka okamžite, podla Arie Van Deursen to môžete prekonat tromi spôsobmi:

- Opätovne sa pokúšajte zákazníka vtiahnut do vývoja. Ukážte mu, že byt s tímom je zábava, že projekt napreduje ovela rýchlejšie, ked je nablízku, ako v prípade jeho neprítomnosti.
- Vytvárajte odborné odhady a naucte sa žit so spomalením práce a frustráciou zaprícinenou opravovaním tých nesprávnych odhadov. Zaznamenávajte si, kolko casu bolo strateného, pretože zákazník nebol nablízku.
- Odídte (už spomenuté ideálne riešenie, no zvycajne nie velmi praktické).

Zaujímavým riešením problému neochoty zákazníka podielat sa na vývoji systému je zameranie sa na povzbudenie zákazníkovej

zainteresovanosti už v prvotných fázach vývoja projektu [3]. Na týchto dvoj až trojdnových intenzívnych stretnutiach zákazník spolu so skupinou programátorov súťaživou formou vytvára prvotný prototyp, a po ich skoncení sám zistí, s akými problémami sa programátori musia stretávat, a tieto nové skúsenosti ho pravdepodobne prinútia prehodnotit svoj prístup k programátorom ako aj k dalšiemu vývoju softvéru.

Záver

Ci chceme alebo nie, pri vývoji softvéru bude vždy jedným z najdôležitejších článkov tohto procesu zákazník. Ved je to on, pre koho je softvér vyvíjaný, kto bude v konecnom dôsledku celý systém hodnotit a vyjadrí svoju spokojnost alebo zostane nespokojný.

Myslím si, že by to mali byt programátori, ktorí prejdú cez bariéru komunikácie, urobia ten prvý, tak potrebný krok k zákazníkovi a sami uznajú, že jeho pomoc potrebujú. Obidve strany by si však v prvom rade mali uvedomit, že nestoja proti sebe, ale sú na rovnakej strane, majú rovnaký ciel, tahajú za ten istý koniec povrazu. No svoj krok musí urobit aj zákazník, ktorému by na výsledku vývoja malo záležat najviac, a tomu by mal podriadit svoje dalšie správanie a rozhodovanie sa.

Agilné metódy sa pokúšajú svojimi postupmi vniest i do tohto vztahu nový, lepší, a možno aj ludskejší prístup k riešeniu problémov, ktorý, ak si ho obidve strany osvoja, môže výrazne zefektívnit a v konecnom dôsledku doviest do úspešného konca nejeden softvérový projekt.

Použitá literatúra

[1] Bieliková, M.: Softvérové inžinierstvo: Princípy a manažment. STU Bratislava. 80-227-1322-8. 2000.

[2] Peters, T.: The Professional Service Firm 50. Knopf Alfred A. 228 s. 0375407715. 1999

[3] Ayerst, T: Getting the Customer Off Site and On the Team *http://www.cwi.nl/~arie/wci2001/ayerst.html*

[4] Deursen van, A.: Lack of On Site Customer *http://c2.com/cgi/wiki?LackOfOnSiteCustomer*

[5] How to Be a Customer of XP Software. XPlorations. Jún 2000 *http://xp123.com/xplor/xp0006c/index.shtml*

Riadenie v agilných metódach

Peter Diko

Riadenie je jednou z najdôležitejších castí pri vývoji softvéru. Článok sa zaoberá možnostami riadenia pri agilných metódach. Pozerá sa na riadenie z viacerých úrovní, na základe hierarchickej štruktúry v skutocných firmách. Taktiež sa zaoberá vztahom zákazníka k riadeniu. Vo všetkých úrovniach porovnáva s klasickým, plánom riadením prístupom a upozornuje na výhody a nevýhody jednotlivých prístupov. Článok nedáva jednoznacnú odpoved na otázku, ktorý prístup je lepší, iba ich porovnáva, pricom sa hlavne sústredí na agilný prístup. Článok ale dáva odpoved na otázku, ci je softvér vyvíjaný agilnými metódami riadený dostatocne.

Úvod

Z príchodom agilných metód (informácie nájdete v [1] a v [3]), lepšie povedané s ich praktickou aplikáciou, sa objavili rôzne dohady, diskusie a názory na ich využívanie. Pri prvom oboznámení sa s agilnými metódami môžeme získat dojem, že sú príliš volné pri svojom postupe a nechávajú privelkú volnost tvorcom projektu na najnižšej úrovni (teda programátorom). Táto volnost môže byt až taká velká, že programátori nakoniec vytvoria produkt, ktorý celkom nezodpovedá predstavám manažéra. Prípadne, budú mat tolko volnosti, že nakoniec nebudú vediet, co majú vlastne urobit a výsledok bude podobný, ak nie horší ako v predchádzajúcom prípade.

V com je vlastne problém?

Ako je známe, rozdiel medzi plánom riadeným a agilným prístupom pri tvorbe softvéru je v reagovaní na zmenu. Zatial co klasický prístup sa zmenám snaží v co najväcšej miere vyhýbat, pri agilnom prístupe sa so zmenami pocíta ako s niecím prirodzeným. V prvom prípade je dopredu urcený relatívne podrobný plán a celý projekt je rozclenený do jednotlivých fáz. V druhom prípade nieco také nie je. Vzniká tu teda otázka, ci je vôbec možné riadit

projekt, ktorý sa stále mení a ak áno, tak ako. Vzhladom na úspech agilných metód pri riešení projektov je odpoved na prvú otázku jasná. Keby nebolo možné riadit projekt, tažko by sme sa dopracovali k žiadanému výsledku. Je teda jasné, že projekt, ktorý vzniká pomocou agilného prístupu je možné riadit. Zostáva otázka ako.

Zodpovedanie tejto otázky nie je lahké a zrejme každý manažér má svoje vlastné metódy. Zvládnut takúto úlohu urcite nie je jednoduché a záleží to aj na schopnostiach osoby, ktorej pripadla táto úloha. Urcite je to velká výzva a bolo by nemožné a naivné si mysliet že v tomto krátkom clánku odpoviem na túto otázku (to napokon nie je ani jeho cielom), iba naznacím možné spôsoby.

Úrovne riadenia

Budem sa sústredit na možnosti riadenia z troch základných pohladov (úrovní). Prvý pohlad bude z hladiska riadenia celého projektu. Týka sa najvyššieho manažmentu v rámci daného projektu. Druhý pohlad bude z hladiska manažérov, vedúcich jednotlivých castí projektu alebo jednotlivých vývojových tímov. Tretí pohlad bude z hladiska samotných vývojárov, programátorov. Samozrejme takéto rozdelenie do troch úrovní nemusí zodpovedat každému skutocnému projektu. Záleží aj na velkosti konkrétnej firmy. Úlohy a úrovne sa môžu prekrývat a jeden clovek taktiež môže zastávat funkcie vo viacerých úrovniach, prípadne sa niektorá úroven vynechá. V skutocných spolocnostiach, ktoré tvoria komercné softvérové riešenia musí do istej miery existovat hierarchická štruktúra. Takáto štruktúra musí existovat aj v hierarchii funkcií, ale aj v rámci projektu.

Riadenie celého projektu

Spolocným spôsobom vývoja projektu v agilných metódach je iteratívny prístup. Zjednodušene povedané, stanoví sa casové obdobie, za ktoré má vývojový tím dosiahnut urcené ciele. Potom sa vyhodnotia dosiahnuté ciele a urcia sa nové, s ohladom na dosiahnuté výsledky. Pracuje sa v iteracných cykloch, pricom pre každý cyklus sa urcujú kontrolné body, kedy sa referuje o stave projektu, zapracovávajú sa zmeny a urcuje sa další vývoj.

Zdá sa, že aj riadenie projektu (v zmysle usmernovania vývoja, nie funkcnosti jednotlivých zložiek) sa dá realizovat jedine pocas dosiahnutia urcitého kontrolného bodu. V zásade sa to zdá

ako pravda, ale len ak sa na to pozeráme z hladiska plánom riadeného prístupu, kde je (malo by byt) dopredu jasné, ako sa bude daný projekt vyvíjat. Pri agilnom prístupe sa pri urcovaní spôsobu vývoja zúcastnujú všetci. To dáva možnost pre manažéra projektu vysvetlit všetkým, co vlastne idú vyvíjat. Zároven môže dostat odpoved, že sa to nedá a projekt je ukoncený. Touto poznámkou chcem naznacit, že predstavy manažérov (zákazníkov) sú niekedy v rozpore s možnostami technickej realizácie.

Je len na prospech celého projektu, ak sa nieco, co sa nedá realizovat, zistí hned na zaciatku. Manažér má možnost okamžitej reakcie smerom ku zákazníkovi, vysvetlit co sa dá a co nie. Cím vlastne riadi celý projekt, lebo stanovuje jeho ciel. Je tu výhoda oproti plánom riadenému vývoju. Nemôže sa stat, že návrhár vypracuje velmi pekný návrh pre projekt, s množstvom funkcií a iných skvelých vlastností, ale tento návrh bude technicky nerealizovatelný. Vyzerá to ako znacná prevaha agilných metód, no na druhej strane treba povedat, že návrhári sú väcšinou skúsení ludia a takýchto hrubých chýb sa nedopúštajú (inak by ich nikto nezamestnal). No pri zacínajúcich návrhároch, alebo pri urcitých detailoch v projekte sa urcitá strnulost klasického prístupu môže prejavit. Typický príklad je, ked návrhár nepozná najnovšie technické prostriedky a naplánuje použitie starších, síce kvalitných, ale menej efektívnych. Samozrejme môže nastat aj opacný prípad, pri navrhnutí supermoderného, efektívneho prostriedku, ale vo firme nie je nikto, kto by s ním mal nejaké skúsenosti (nikto, kto by ho vedel použit). Ak neexistuje okamžitá reakcia na navrhovaný prostriedok, predstavuje to pre manažéra (a nie len prenho) riziko, ak schváli takýto postup. Pri agilných metódach, kde je okamžitá reakcia na vytvorený návrh, sa toto riziko zmenšuje.

Tak isto po jednotlivých kontrolných bodoch je možnost riadenia. Ak sa zmenilo nieco na požiadavkách zákazníka, dosiahla sa zásadná funkcia projektu, alebo sa nieco nepodarilo, má manažér projektu možnost okamžitej reakcie. Ci už v komunikácii so zákazníkom, alebo v zabezpecení nových zdrojov pre projekt a pod. Samozrejme, treba zvolit správne casové rozpätie medzi jednotlivými kontrolnými bodmi, ale to záleží aj od vybranej metódy amôže byt rôzne pre rôzne projekty. Od referovania každý den, cez týžden až po mesiac (napr. pre proces *Scrum* [5] sa kontrolné body uskutocnujú s periódou 30 dní). Všeobecne platí pravidlo: co najskôr ako sa dá (tak casto, ako dosiahneme další pokrok). Takto dokáže nenásilným spôsobom riadit celý projekt, pricom sa nemusí spoliehat len na svoj vlastný úsudok, ale má

názor prakticky každého zainteresovaného v projekte. Pri dôslednom dodržiavaní princípov agilných metód má k dispozícii dokonca aj názor zákazníka pocas vyvíjania projektu, co je nesporne velká výhoda. Podmienkou ale je mat stále na zreteli ciel projektu.

Riadenie castí projektu

V tejto casti sa zameriam na možnosti riadenia manažérov alebo vedúcich jednotlivých tímov. Títo ludia tvoria stupen medzi manažérom projektu a samotnými vývojármi. Samozrejme, že v spolocnosti môže byt takýchto stupnov aj viac. Ale väcšinou má každá vývojová skupina svojho vedúceho aten je zodpovedný svojim nadriadeným.

Spomínaný vedúci alebo manažér vývojového týmu sa môže, alebo nemusí priamo podielat na vývoji softvéru na najnižšej úrovni. V každom prípade má velmi blízko k implementacnej úrovni projektu. To znamená, že pozná pomerne dobre ako anielen co robí cast projektu, ktorej je vedúci.

Má za úlohu zabezpecit, aby jeho tím splnil úlohy urcené do najbližšieho kontrolného bodu. Musí teda riadit svoj tím, cím vlastne riadi projekt alebo aspon jeho cast. Musí zabezpecit rozdelenie práce v tíme tak, aby boli splnené úlohy pre kontrolný bod. Pri rozdelovaní práce a urcovaní dalšieho postupu ale musí zachovávat princípy agilných metód. Pri urcovaní, ako sa bude dalej pokracovat pri vývoji, je potrebné dat možnost vyjadrit sa celému tímu.

Môže, ale nemusí, mat právomoc najímat potrebných ludí pre projekt. Minimálne musí mat možnost urcovat (za spolupráci existujúceho tímu), akých pracovníkov potrebuje – špecialistov na zvolený jazyk, grafikov atd. Samozrejme nie len pracovníci sú potrební pre projekt, ale aj materiálne zabezpecenie. Cíže má mat možnost urcovat potrebné zdroje pre projekt. Výhoda oproti plánom riadeným prístupom je vtom, že nemusí dopredu vediet všetky prostriedky, ktoré bude potrebovat, lebo nie je urcené ako sa bude projekt vyvíjat. Problém však môže nastat, co je nevýhoda oproti plánom riadeným prístupom, ak sa pocas projektu rozhodne pre nejaký nepredpokladaný alebo neštandardný prostriedok, že sa nemusí podarit jeho získanie. Táto skutocnost môže vniest minimálne zdržanie do projektu.

Tu je zrejmý dôvod, preco je potrebný takýto medziclánok medzi manažérom celého projektu a vývojármi.

Vrcholový manažér nemusí a pravdepodobne ani nevidí dopodrobna do technickej realizácie projektu, takže nemá potrebné technické informácie na rozhodnutie o využití takéhoto zdroja. Vývojár má za úlohu vyvíjat a nie zhánat potrebné vybavenie. Vedúci skupiny pritom má potrebné technické informácie a je podrobne oboznámený s problematikou. Má možnosti zabezpecit požadovaný zdroj (pretože má na to právomoc), alebo urcit niekoho kto mu požadovaný zdroj zabezpecí. Preto je najvhodnejšou osobou na rozhodnutie ci sa daný prostriedok bude snažit získat, alebo nie. To je dalšia forma riadenia samotného projektu, lepšie povedané, spôsobu jeho vývoja.

Pozorný citatel si iste všimol, že táto úroven by sa dala rozdelit do minimálne dvoch oblastí. Prvá je riadenie samotného tímu z hladiska co, ako a kto bude riešit. Druhá je zabezpecenie potrebných zdrojov (ludia, materiál) a rozhodnutie ci požadované zdroje budú k dispozícii. Tu len treba zopakovat to, co bolo spomínané na zaciatku tejto casti, že nikde nie je povedané, aby sa riadenie projektu rozdelilo na tri úrovne. Ide mi o to, aby som naznacil, že je potrebná urcitá úroven, ktorá zabezpecuje „podporné“ služby. Aj ked výraz podporné služby, pre vedúceho tímu, asi nevystihuje presne jeho úcel. Na druhej strane výraz „riadiace“ služby, budí dojem direktívneho prístupu vedúceho tímu, co nie je pri agilných metódach pravda. Je to nieco medzi tým. Najlepšie bude, ak sa aktívne zapojíte a vytvoríte si vlastný názov (co je koniec koncov úcel agilného prístupu).

Riadenie projektu z vývojového hladiska

Jednou z najdôležitejších súcastí celého projektu sú vývojári (programátori). Zameriam sa na ich možnosti riadit priebeh projektu. Ich možnosti sa zdajú obmedzené vzhladom na to, že nemajú žiadnych podriadených.

Najskôr sa pozriem na plánom riadené prístupy. Vývojári v nich nemajú velkú možnost ovplyvnit priebeh projektu. Majú urcený plán, ktorý treba dodržiavat. Samozrejme, ak sa vyskytne neocakávaná zmena, ktorá by znemožnila dalšie pokracovanie v projekte, má možnost zasiahnut do riadenia, ale aj to iba formou oboznámenia nadriadených so situáciou. Pricom sa skonštatuje, že návrh, prípadne iná dopredu predpokladaná cast je zlá a potrebuje prepracovat. Vzásade má možnost ovplyvnit

hlavne kvalitu projektu, vzhladom na to, že je sám jeho spoluautorom.

Pri agilných metódach má vývojár omnoho väčšie možnosti. Môže sám navrhovat postup pri riešení projektu a podiela sa na urcovaní jednotlivých kontrolných bodov. Nevytvára iba to, co mu niekto zadal, ale sám urcuje (podiela sa na urcovaní), ako sa bude daný problém riešit. Dostáva obrovský priestor na sebarealizáciu, zvyšuje svoju zainteresovanost v projekte, co vedie k tomu, že mu na projekte zacne viac záležat, tým pádom sa snaží zvyšovat jeho kvalitu.

Ked si uvedomíme co to znamená, zistíme, že je to nesporná výhoda pri vývoji softvéru. Zaujímavý clánok (The New Metodology) o spomínaných vlastnostiach a postupoch (a nie len o tom) napísal Martin Fowler [4]. Ved predstavte si, že by robotník napríklad pri stavbe mostu, ku ktorému sa dá programátor v hierarchii firmy prirovnat, mal možnost ovplyvnit projektové plány. Cisto teoreticky je to možné, mohol by vniest urcité poznatky priamo z praxe, ale prakticky je to nemožné a ani sa tak nepracuje. Samozrejme, môžete namietat, že nemôžme porovnávat stavbu mostu, alebo dialnice s vývojom softvérového produktu. Robotník, podla všetkých predpokladov, nemá potrebné vedomosti – väčšinou sú to ludia s nižšou úrovnou vzdelania. No programátori, vývojári softvérového produktu, sú väčšinou vzdelaný ludia. Tak preco ich nevyužit pri navrhovaní, plánovaní a riadení samotného projektu. Je to jedinecná výhoda pri softvérových dielach, ktorú iné odvetvia priemyslu nemajú. A práve klasické metódy využívajú podobný proces vývoja ako napríklad pri spomínanej stavbe mostu. Najskôr podrobný návrh, plán a až potom samotná stavba. Celé riadenie je organizované smerom zhora nadol. Podriadený prakticky nemá možnost ovplyvnit to, ako sa bude daný projekt realizovat. Pri agilných prístupoch je zvolený iný prístup, ktorý využíva jedinecné možnosti softvéru a dáva možnost na riadenie aj samotnému programátorovi.

Aby sme ale len nevyzdvihovali do neba agilné metódy, treba povedat, že nie každému takýto prístup vyhovuje. V predchádzajúcom texte bolo spomenuté, že vývojár je od toho, aby vyvíjal. A práve tu môže byt problém. Kto sa chce zúcastnovat na riadení projektov (a má na to všetky predpoklady), sa v konecnom dôsledku posunie (presunie) z programátorských funkcií vyššie a vyššie. No nie každý má takéto ambície. Pre takýchto ludí sa potom ich aktívna úcast na riadení projektu stáva zbytocným vyrušením od ich práce. Treba si uvedomit, že pri softvérových projektoch neplatí pravidlo, že nadriadený musí

zákonite poberat vyššie financné odmeny ako jeho podriadený. Ak je programátor na slovo vzatý odborník na urcitú oblast (napríklad špecialista pre programovací jazyk) aje na špickovej úrovni, môže zarábat viac ako jeho nadriadený. Náhrada za takýchto ludí sa hladá tažko, hlavne ked sú už zžití sfirmou, v ktorej pracujú. Je pravdepodobné, že sa pre nich zatahovanie do riadenia stáva prácou navyše. Naštastie, agilné metódy nikoho nenútia, aby sa podielal na urcovaní smeru projektu, dávajú mu iba možnost. No tu je dôvod, preco nie je možné jednoznacne povedat, že plánom riadený prístup je horší. Treba si uvedomit akých ludí máme vtíme a zbytocne ich nezatažovat, aby zas nemali pocit, že oni musia rozhodovat o všetkom.

Riadenie z pohl adu zákazníka

Aj ked v agilných metódach sa pri vývoji zúcastnuje (mal by sa) aj zákazník, nezaradil som ho ako dalšiu úroven. Zákazník predsa len pôsobí mimo firmy. Nemusia ho zaujímat napríklad problémy ako bude projekt implementovaný. V jeho záujme je, aby projekt splnal ním požadované vlastnosti. Zameriam sa teraz na jeho možnosti pri riadení projektu. Zaujímavý clánok (How to Manage Agile/Lightweight Projects) zaoberajúci sa aj touto problematikou napísal Hugh Bawtree[2].

Zdá sa, že možnosti zákazníka sú v podstate iba konzultacného charakteru. Vo všeobecnosti, softvérové firmy nemajú radi, ak sa zákazník snaží ovplyvnit spôsob, ako sa bude daný softvér implementovat. Tento problém je pri zákazníkoch, ktorí si myslia, že sa „rozumejú do pocítacov“. Preto softvérové firmy nezatažujú svojich zákazníkov podrobnostami jednotlivých riešení. Ako teda zaclenit zákazníka do riadenia? V princípe zostáva už len spomenutý konzultacný charakter. Ak si ale uvedomíme, co všetko castými konzultáciami dokáže zákazník zmenit, zistíme že môže výraznou váhou zasiahnut do riadenia projektu.

Na zaciatku, zákazník definuje svoju hrubú predstavu, o tom, co by mal jeho softvér vlastne robit. Pri plánom riadených prístupoch, potom prebieha intenzívna komunikácia so zákazníkom až po vytvorenie podrobnej špecifikácie a plánu. Potom sa zákazník kontaktuje pri testovaní prvých prototypov, prípadne pri zistení vážnych nedostatkov pocas vývoja. Pri agilných metódach má zákazník možnost konzultácií pri každom kontrolnom bode. Na základe dosiahnutého pokroku sa môže rozhodnút pre dalšie smerovanie projektu, lebo konecné slovo

má vždy zákazník. Môže ovplyvnit zdroje projektu a to najmä v podobe financií, cím môže jednoznacne rozhodnút aj o tom, ako sa bude realizovat daný projekt. Ak povie, že jednu alternatívu, síce kvalitnejšiu a jednoduchšiu pre vývojárov, ale financne nárocnejšiu nepodporí, vývoj projektu sa musí uberat iným smerom (alebo skoncí, ale takéto riešenie nie je spravidla v záujme ani jednej zo zúcastnených strán).

Ako vidíte, z obycajných konzultácií pre zákazníka sme sa nakoniec dostali k tomu, že má rozhodujúce slovo. To pravdepodobne predpokladal každý, ale ide o možnost, kedy môže zákazník uplatnit svoje rozhodujúce slovo. Ak je to pocas vývoja projektu a nie len na jeho zaciatku a pri konci, môže to softvérovej firme ušetrit množstvo úsilia a prostriedkov. Na druhej strane sa ale môže stat, že zákazníka bude obtažovat castá konzultácia (aj ked je v jeho prospech) alebo sa bude snažit neprimerane a neodborne zasahovat do projektu (ak si myslí, že všetkému rozumie).

Celkový pohl ad

Pozreli sme sa na riadenie v agilných metódach z pohladu jednotlivých úcastníkov zainteresovaných pri vývoji projektu. Teraz sa pozrieme na riadenie z pohladu celého projektu a budem sa snažit zovšeobecnit princípy riadenia.

Ako som už spomínal, pri vývoji softvéru agilnými metódami sa postupuje iteratívne. Uviedol som, že po každej takejto iterácii (kontrolnom bode) sa referuje o stave projektu, zapracovávajú sa zmeny a urcuje sa další vývoj. No budem ešte konkrétnejší a uvediem konkrétne kroky pri jednotlivých kontrolných bodoch.

V každom cykle sme mali stanovenú urcitú úlohu, ktorú sme plánovali realizovat. Na konci každého cyklu si musíme položit nasledujúce otázky:

- Co sme spravili dobre?
- Co sme sa naucili?
- Co môžeme zlepšit?
- Co je nevyriešené?

Po zodpovedaní týchto otázok si urcíme ciele pre najbližší cyklus, poprípade aj dlžku cyklu, ak nie je pevne stanovená. Stanovenie dlžky cyklov závisí od predpokladaného termínu

ukoncenia vývoja softvéru (ukoncenie je väčšinou zmluvne podchytené). Aké konkrétne kroky sa vykonávajú pocas cyklu a dlžky jednotlivých cyklov sú rôzne pre viaceré prístupy agilných metód.

Záver

Hlavný princíp agilných metód je umožnit každému aktívne prispiet k plánovaniu a vývoju projektu. Z toho vyplýva, že urcitou formou, môže každý riadit projekt. Hlavný dôraz sa kladie na zaciatok projektu, ked sa urcí jeho ciel a potom pri jednotlivých kontrolných bodoch. Tu môže každý prispiet k riadeniu aktívnym zapojením sa a vyjadrením svojho názoru. Výhoda je, ako som už spomenul, že projekt je schopný velmi rýchlo a vhodne reagovat na zmeny. Zainteresovanost zákazníka zabezpecí, aby sa projekt vyvíjal správnym smerom, manažér projektu vytvorí kvalitné podmienky pre vývoj projektu, vedúci tímov vyberú správnych odborníkov, ktorých priama úcast na riadení je zárukou použitia najnovších a najkvalitnejších prostriedkov. Ak sa všetko dodrží, je projekt na najlepšej ceste k úspešnému koncu. Problémy môžu nastat, ak niektorá zo zložiek bude brat svoju prílišnú zainteresovanost v projekte ako obtažovanie a prácu naviac. Potom hrozí neúspech celého projektu. Preto treba uvážit, do akej miery budeme dodržiavat agilný prístup. Urcite treba dodržat vlastnost, že najdôležitejší sú ludia. A ak niektorým z nich takýto prístup nevyhovuje, treba použit aj niektoré metódy pre niektorých klasických, pre niektorých zastaralých prístupov.

Na záver by som chcel dopovedat myšlienku zo zaciatku clánku o volnosti programátorov. Každý kto pozorne cítal celý clánok, musí prist k presvedceniu, že síce majú znacnú volnost v urcovaní ako sa bude daný projekt realizovat, no v žiadnom prípade sa nedá povedat, že majú prílišnú volnost. Musia plnit úlohy zadané pre jednotlivé kontrolné body, pri ktorých sú neustále kontrolovaní a riadení. To však neplatí len pre nich, ale pre všetkých zúcastnených. Dá sa teda povedat, že projekt vyvíjaný agilnými metódami je riadený velmi dôkladne, velmi casto a na všetkých úrovniach.

Použitá literatúra

[1] Boehm, B. 2002. Get Ready for Agile Methods, with Care
http://sunset.usc.edu/events/2002/arr/Get Ready fo Agiel Methods, with Care.pdf

[2] Bawtree, H. 2001. How to Manage Agile/Lightweight Projects
http://www.spc.ca/essentials/nov1301.htm#3

[3] Beck, K., Beedle, M., Van Bennekum, A. et al. 2001. Manifesto for Agile Software Development
http://www.agilemanifesto.org

[4] Fowler, M. 2001. The New Methodology
http://www.martinfowler.com/articles/newMethodology.html#N56

[5] ADM, Inc. 1997 – 2002. Welcome to ControlChaos.com, home of Scrum!
http://www.controlchaos.com

Agilné metódy a návrh

Ladislav Gyökeres

Pre mnohých profesionálov, pozerajúcich sa so zatracovaním na agilné metódy, je trnom v oku neexistencia striktne oddelenej fázy návrhu, ktorý môžeme považovat za klúcovú fázu plánom riadeného vývoja softvéru (aj ked ani jedna fáza vývoja softvéru nemá oproti iným vyslovene vyššie postavenie, návrh je to, co má azda najvýznamnejší vplyv na plán vývoja softvéru). Pri agilnom návrhu je tento distribuovaný do množstva nadväzujúcich a interagujúcich minifáz. Takéto rozdelenie prináša svoje výhody, ale aj nevýhody. Cielom clánku je zhrnút poznatky o postavení návrhu v agilnom vývoji softvéru v porovnaní s postavením v plánovitom vývoji.

Úvod

Jedna z najdôležitejších fáz vývoja softvérového produktu, ktorá má vplyv skoro na celý životný cyklus vývoja softvéru je návrh. Podobne, ako v každej oblasti života, aj pri vývoji softvéru sa kladie velký dôraz na správnost výsledného produktu. Tento fakt musíme mat na zreteli už pri návrhu a eliminovat co najviac chýb. V dnešnej dobe, kedy skracujúce sa termíny a zvyšujúce sa ocakávania na kvalitu predstavujú výzvu nielen pre manažérov, ale aj pre vývojárov, už existuje viac pohladov na návrh, ktoré uprednostnujú rôzne aspekty resp. klúcové elementy aokrem iných sa líšia aj v efektívnosti a v spôsobe zaobchádzania so zmenami, ktoré nastávajú pocas vývoja.

Plány na zahodenie

Asi najrozšírenejším typom návrhu v súcasnosti je návrh, ako ho poznáme z v evolucného životného cyklu. V niektorých pramenoch sa používa termín evolucný návrh. Kedže som vychádzal z literatúry [1], ktorá uprednostnuje túto terminológiu, v dalšom budem používat tento výraz aj ja. Podstata evolucného návrhu spocíva v tom, že návrh sa postupne vylepšuje a doplna vždy v nasledujúcej iterácii (v nasledujúcom cykle) pocas celej fázy implementácie produktu. Kedže návrh nepredstavuje uzavretú fázu, je lahšie do neho zakomponovat zmeny, a preto je tento typ návrhu s oblubou využívaný agilnými metódami. Evolucný návrh vo všeobecnosti však má aj svoje nevýhody,

pretože prispôsobivost, ktorú poskytuje, predstavuje aj isté riziká. Predstavme si, že po každej iterácii vykonávame ďalšie a ďalšie zmeny, ktoré sú väčšinou výsledkom taktických rozhodnutí. Navrhnutý systém sa postupne stane neprehľadným, pridávanie ďalších zmien sa stane coraz ťažším a výsledný produkt zvycajne už je veľmi slabo štruktúrovaný. Toto nesie v sebe ďalšie nebezpecenstvá, lebo softvér sa stane náchylným na chyby (rovnako na logické ako aj na syntaktické), ktoré sa potom ťažšie odstranujú. A aj v tomto prípade platí, že cena odstránenia chyby rastie exponenciálne s postupom projektu. Niektorí ľudia tento typ návrhu ani neuznávajú ako ozajstný návrh. Keďže každý softvér sa dá vylepšiť, hrozí nebezpecenstvo, že budeme iterovať do nekonecna v dôsledku stále nových a nových požiadaviek od zákazníka. Tomuto riziku sa treba vyhnúť alebo aspoň rejako zredukovať, napr. spracovaním požiadaviek na zmenu.

Plánovitý návrh oproti evolucnému predstavuje prístup, ktorý je „vypožicaný“ od ostatných inžinierskych disciplín - najprv plánovať a potom stavať. Návrhári naplánujú softvér, ale nemusia ho implementovať, na to sú programátori. Takto majú návrhári volnejší priestor a môžu využívať abstraktnejšie prostriedky ako programovacie jazyky, napr. modelovacie jazyky, ktorých typickým predstaviteľom je UML (*Unified Modeling Language*). Majú dopredu premyslené všetko súvisiace s návrhom, co im umožnuje vyhnúť sa taktickým rozhodnutiam zo strany programátorov a preto návrh zostane pomerne jednoduchý a prehľadný. Programátori potom iba postupujú podľa návrhu a nemusia robiť rozhodnutia, ktoré sa týkajú celého systému.

Plánovitý návrh predstavuje optimistický prístup k vývoju softvéru a možno preto neberie do úvahy niektoré fakty, ktoré môžu mať velký vplyv na vývoj konkrétneho produktu. Tým, že na vývoji pracuje viacero skupín ľudí – predovšetkým návrhári a programátori – je projekt automaticky rozdelený do viacerých castí. Návrhári však nemôžu predvídať (aspoň nie na 100%), aké problémy sa objavia pocas implementácie, a preto ani nevedia vyriešiť tieto problémy. Dokonca pocas programovania môžu byť objavené aj také problémy, ktoré spochybnia návrh. Niekedy je takáto situácia riešená programátormi, inokedy návrhárom, ak je ešte k dispozícii, v každom prípade však zmena prebieha v casovej tiesni a oneskoruje projekt, pretože je potrebné zistiť, kde všade sa táto zmena ešte prejaví. Velmi závažný problém vzniká v prípade, ked návrhár už pracuje na inom projekte alebo dokonca už nepracuje vo firme, v ktorej sa daný produkt vyvíja. Proti takýmto rizikám plánovitý vývoj nasadzuje silnú zbraň - obsažnú dokumentáciu. Ak je však neadekvátne použitá a tvorená,

môže vo významnej miere brzdit vývoj softvéru. Dalšia vec, ktorá môže byt považovaná za chybu plánovitého návrhu je spojená s rolami. Návrhári sú väčšinou vyberaní podla svojich schopností a skúseností a v dalšom sa venujú iba tejto cinnosti – návrhu. To im niekedy zaberie tolko casu, že už ani neprogramujú. V dnešnom, rýchlo sa vyvíjajúcom svete však dennodenne pribúdajú dalšie nástroje a nové metódy a návrhár, ktorý sa venuje iba navrhovaniu, môže stratit prehlad o nových trendoch. V dôsledku toho môže stratit aj rešpekt v okruhu programátorov. Takéto prípady môžu nastat aj v ostatných inžinierskych disciplínách, ale je tu podstatný rozdiel. V skupine vývojárov nie je až taký rozdiel medzi schopnostami návrhárov a realizátorov ako v ostatných oblastiach. Programátori by mali byt velmi dobrými odborníkmi, natolko dobrými, aby vedeli spochybnit návrh.

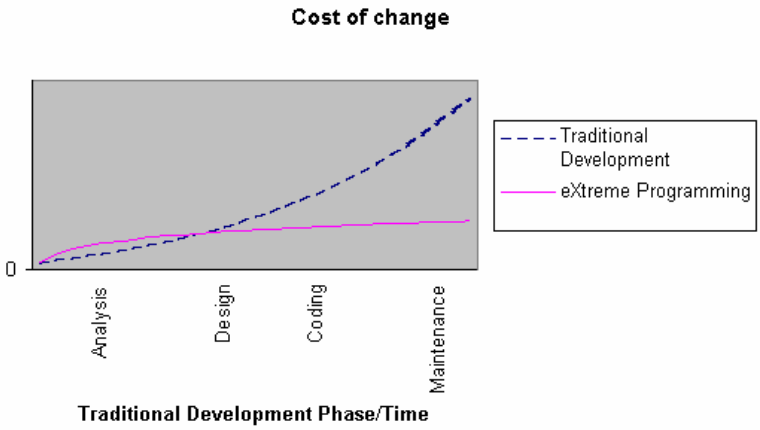
Hore uvedené problémy aj ked tažko, ale môžu byt vyriešené. Možno vieme navrhnút systém v dostatocnej kvalite a možno vieme zabezpecit aj velmi dobrých návrhárov, ale tak ci tak zostáva jeden problém a tým sú meniace sa požiadavky. V súcasnosti predstavujú najväcší problém apráve preto sú na túto problematiku zamerané agilné metódy. Jedným zo spôsobov, ako vyriešit problém meniacich sa požiadaviek, aj ked iba ciastocne, je zabudovanie flexibilnosti do návrhu, aby neskoršie zmeny mohli byt lahšie doplnené. Na to je potrebné podrobne analyzovat problémovú oblast, s cielom hladania možných budúcich zmien. Tým pádom sa môžeme pripravit na zmeny, ale iba na tie predvídané. Proti nepredvídaným zmenám sme ešte stále bezbranní. Niektoré zmeny v požiadavkách sú dôsledkom toho, že sa neporozumelo požiadavkám v dostatocnej miere. Tento fakt spôsobil, že vela ludí sa zameriava na porozumenie požiadavkám v nádeji, že takto môžu predíst zmenám vbudúcnosti. Tým sa však nedá predíst, ale tieto opatrenia aspon skvalitnia analýzu požiadaviek.

Na základe hore uvedených tvrdení sa môže zdat, že plánovitý návrh je v mnohých prípadoch tažko schodná cesta. Je evolucný návrh lepší? Ani to nemôžeme tvrdit. Ktorý je potom lepší? Asi je najvhodnejšia kombinácia týchto dvoch prístupov vo vhodnom pomere. V nasledujúcich castiach tohto clánku sa pokúsime odhalit tento pomer.

Ako používat evolucný návrh?

Použitie extrémneho programovania (dalej XP - *Extreme programming*), ako jedného z predstavitelov agilných metód, je

predmetom na diskusiu z viacerých dôvodov. Jeho hlavnou zvláštnostou je, že podporuje evolucný návrh namiesto plánovitého. Ako vieme, evolucný návrh teoreticky nemôže fungovat v dôsledku taktických rozhodnutí. Treba sa pozriet na tento problém zpohladu krivky zmien (Obr. 1), ktorá ukazuje závislost ceny zmien od casu.



Obrázok c. 1 - Krivka zmien

Na základe tejto krivky sa postupom casu exponenciálne zvyšujú náklady na realizáciu zmien. Je ironiou, že vela projektov funguje ešte stále bez poriadnej analýzy, pricom je zachovaná exponenciálna povaha vývoja krivky zmien. Táto krivka poukazuje na to, preco takýto prístup nemôže byt v praxi úspešný. Tiež to vysvetluje, preco je potrebné použit plánovitý návrh opatrne: lebo v etapách vývoja softvéru, ktoré nasledujú po návrhu, náklady na odstránenie chýb sú už ovela väcšie. Väcšina agilných metód vychádza z toho, že krivku zmien je možné vyhladit a takto zefektívnit evolucný návrh dostatocne na to, aby mohol byt funkcný aj v praxi. Ale pozor. Casti resp. praktiky agilných metód, ktoré sú založené na tomto vyhladení a priamo ho využívajú, nemožno použit bez dalších castí, ktoré toto vyhladenie umožnujú. Toto je castou prícinou polemík. Napr. niektorí ludia kritizujú využitie vyhladenia krivky zmien bez toho, aby rozumeli cinnostiam, ktoré vyhladenie umožnujú. V niektorých prípadoch je to podložené aj vlastnou skúsenostou kritizujúcej osoby, kedy táto osoba použila iba tie casti, ktoré umožnujú vyhladenie bez toho, aby zabezpecila k tomu nevyhnutné predpoklady. Preto ani nemohla uspiet a vidí agilné metódy cierne.

Zabezpecenie predpokladov na umožnenie vyhladenia krivky zmien pozostáva z viacerých cinností. Základ tvorí

testovanie a neustála integrácia, no najviac dôrazu sa pritom kladie na testovanie. Niektoré metódy, ako napríklad aj XP, by ani nemohli existovat bez zabezpecenia, ktoré im poskytuje práve testovanie. A neustála integrácia je potrebná na to, aby vývojový tím bol stále zosynchronizovaný. Spolocné využitie týchto cinností spôsobuje významný synergický efekt, ktorý má velký vplyv na krivku ceny zmien.

Refaktorizácia, ako dalšia cinnost zameraná na vyhladenie krivky zmien, má jednoduchý efekt. Vo významnej miere zefektívnuje prácu. John Highsmith vo svojej práci [2] používa analógiu váh. Na jednu stranu je postavený plánovitý návrh, na druhú návrh podporený refaktorizáciou. V tradicných prístupoch pri vývoji softvéru dominuje plánovitý návrh. Je to dôsledkom predpokladu, že clovek nebude menit svoje názory. Ak cena zmien klesne na istú úroven, je už výhodnejšie používat refaktorizáciu. Evolucný návrh profituje z neustálej integrácie, testovania a refaktorizácie. Napriek tomu, plánovitý návrh má svoje miesto aj dnes pri vývoji softvéru. Ide napríklad o oblasti, kde sa kladú vysoké nároky na prevádzkovú spolahlivost. Len treba nájst rovnováhu medzi týmito dvoma prístupmi. Hlavnou otázkou je, kde je bod rovnováhy. Ved ani XP nie je len testovaním, kódovaním a refaktorizáciou. Je v nom miesto aj pre návrh. Istá cast návrhu sa vykoná pred zacatím samotného kódovania, ale väcšina návrhu sa robí v jednotlivých iteráciach pred implementovaním. V každom prípade, cinnosti spojené s agilnými metódami významne posunuli bod rovnováhy smerom k návrhu podporeného refaktorizáciou.

Jednoduchost: klúc k úspechu

Manifesto jednoduchého návrhu Extrémneho programovania možno zhrnút do dvoch formúl:

„Do the simplest thing that could possibly work.“

“You aren’t going to need it. (YAGNI)“

„Urob najjednoduchšiu vec, ktorá by mala fungovat.“

„Nebudeš to potrebovat.“

Prostredníctvom, týchto formúl získali agilné metódy a zvlášt Extrémne programovanie velkú popularitu. YAGNI je potrebné chápat tak, že nemáme pridat žiadny kód, ktorý bude použitý iba v budúcnosti nejakou novou castou systému, ak to nie je potrebné už teraz. Vyzerá to dost jednoducho. Problémy s

jednoduchostou prichádzali s nástupom znovupoužitelných komponentov a flexibilného návrhu, ktoré sú velmi nárocné na návrh. Ich vývoj je drahší, ale vývojári vychádzajú z predpokladu, že vynaložené náklady sa vrátia v budúcnosti. Vyzerá to tak, že takýto prístup je klúcom k efektívnemu vývoju softvéru.

Ale je to naozaj tak, že pridaná cena vývoja pre znovupoužitie a flexibilitu sa vráti pri dalšom vývoji? Agilné metódy napriek horeuvedeným predpokladom odporúcajú dat prednost co najskoršej funkcionalite pred flexibilitou. Potom, ked systém už funguje, môžeme ho nechat rást podla potreby. Martin Fowler vo svojej práci [1] uviedol ako príklad triedu Peniaze. Ak potrebujem, aby táto trieda vedela scítat, tak zabudujem do nej iba túto vlastnost. Aj ked dobre viem, že v nasledujúcej iterácii budem potrebovat aj násobenie, túto vlastnost v aktuálnom kroku vynechám. Nechám to na nasledujúcu iteráciu a to aj napriek tomu, že viem ako to urobit jednoducho a rýchlo. Jeden z dôvodov na to je aj efektívnost. Ak pracujem na niecom, co bude využité iba v budúcnosti, stratím cas na prácu nad castou, ktorú potrebujem dokoncit v tejto iterácii. Plán vydaní jasne stanovuje, co musí byt hotové v tejto iterácii a co v budúcnosti. Práca na castiach, ktoré musia byt vypracované v budúcnosti je v rozpore stýmto plánom a tiež s dohodou medzi zákazníkom a vývojárom. Navyše hrozí nebezpecenstvo, že ciele aktuálnej iterácie nebudú splnené.

Aj ked takéto riziko nehrozí, zákazník ešte stále môže požiadat o prácu navyše (a tiež sa to nemusí týkat násobenia). Túto ekonomickú hrozbu zhoršuje aj to, že to, co sme vytvorili, nemusí byt správne. Jedným z dôvodov môže byt, že požiadavky ešte nemusia byt celkom jasné. Dá sa to zhrnút nasledovne. Z hladiska manažmentu projektu je nesprávne, ak pracujeme na nejakom probléme, ktorý nie je naplánovaný na aktuálnu iteráciu. Od tohto je horšie už iba to, ak tento problém riešime nesprávne. Experti XP tvrdia, že ludia sú viac náchylní urobit nieco nesprávne ako urobit to dobre.

Druhým dôvodom je, že zložitému návrhu je tažšie porozumiet, a preto je tažšie donho zakomponovat zmeny. Toto zvyšuje náklady medzi iteráciou, kedy bola vlastnost zapracovaná a iteráciou, kedy bola takáto zmena potrebná.

Strucne povedané, nemali by sme zakomponovat casti, ktoré budú potrebné až v budúcnosti. Dokonca ani v prípade, že cena takýchto zmien je nulová, lebo zvyšujú zložitost celého systému a tým pádom aj cenu zakomponovania dalších zmien.

Vychádzajme teda z toho, že chceme jednoduchý program. Zatial sme ale nedefinovali, co to znamená „jednoduchý program“. Kent uviedol štyri kritériá jednoduchosti. Tie sú v poradí dôležitosti:

- vyhovuje všetkým testom
- odhaluje všetky úmysly
- neobsahuje žiadne duplikácie
- obsahuje co najmenší pocet tried resp. funkcií

Vyhovovat testom je prirodzenou ajednoduchou požiadavkou. Vyhnút sa duplikáciám je tiež dost logické, ale v tomto prípade by už niektorí programátori potrebovali usmernenie, ako na to.

Komplikovanejšie je to s odhalovaním úmyslov. Co to presne znamená? Vo všeobecnosti velkú hodnotu predstavuje cistota resp. jednoznacnost kódu. Agilné metódy kladú tiež velký dôraz na citatelnost kódu, najviac asi *Open source*, kde je táto vlastnost životne dôležitou podmienkou na vývoj softvéru. V niektorých metódach sa to nazýva inteligentný kód (*clever code*) alebo kód odhalujúci úmysly (*intention revealing code*). Vo svojej práci XP 2000 Josh Kerievsky na to uviedol jeden dobrý príklad, asi najznámejší XP kód – *JUnit*. Je v tom však malý hácik. Pre programátorov, ktorí poznajú návrhový vzor „*Decorator*“, je síce dobre citatelný, ale tí ktorí ho nepoznajú, sa v nom velmi tažko orientujú.

Na elimináciu duplikátov skoro každá metóda používa nejaký prístup. Napr. XP používa „*Once and only once*“ (Raz a iba raz), v prípade *Pragmatic programmer* sa táto technika volá DRY (*Don't repeat yourself* - Neopakuj sa). Väčšinou ide o tie isté pravidlá, len sú inak pomenované. Ich podstata spocíva v tom, že každá informácia (funkcie, dáta, atd.) musí byt uložená iba raz a na jednom mieste. Ak sa zameriavame iba na eliminaciu duplikátov, aj to nám pomôže vo významnej miere zjednodušit kód, ale ani splnenie hore uvedených štyroch kritérií nie je všetko. Naozajstnú jednoduchost je tažké dosiahnut.

Niektorí návrhári sú priam šokovaní zhore uvedených pravidiel a pomaly postupujú návrhom, lebo sa boja, že návrh nebude vyhovovat týmto pravidlá. V prípade velmi zložitých systémov velký krok dopredu znamená aj refaktorizácia a to sa v takýchto prípadoch aj využíva. Ako M. Fowler píše v jeho práci [1], „je lepšie refaktorizovat zložitý návrh ako refaktorizovat žiadny“. Najlepšie je, ked vytvoríme trochu jednoduchší návrh ako je to potrebné, ale nie je žiadna tragédia, ked návrh bude

trochu komplikovaný. Najlepšia rada pochádza od Roberta Martina. Nemali by sme vela premýšlat o tom, ci je to jednoduchý návrh. Tak ci tak to môžeme resp. mali by sme refaktorizovat neskôr. Podstatné je, aby sme chceli refaktorizovat.

Refaktorizácia teda zaberá cas, ale nepridá funkcionalitu navyše. V súvislosti s tým môže vniknút jedna zaujímavá otázka. Kedže princípy agilných metód radia, aby sme navrhovali pre súcasnost a nie pre budúcnost, nie je to v protiklade s refaktorizáciu? YAGNI hovorí, aby sme nezaviedli komplexnost, ak to nie je potrebné v súcasnosti. To je súcastou aj jednoduchého návrhu. Refaktorizáciu potrebujeme, aby sme návrh vedeli udržiavat taký jednoduchý, ako je to možné a preto je potrebné refaktorizovat kedykolvek, ked je to vhodné. Jednoduchý návrh je možné vytvorit efektívne súcasným použitím testovania, neustálej integrácie a refaktorizácie. Takto sa zabezpecuje aj hladkost krivky zmien. Každá nepotrebná komplexnost, ktorú vnesieme do systému spôsobí, že systém sa stane tažšie modifikovatelným. Výnimkou je vkladanie komplexnosti kvôli získaniu flexibility. Ludia však väcšinou nevedia dobre predvídat a preto je lepšie, ak ostaneme pri jednoduchosti. Túto však len zriedka dosiahneme na prvý pokus a preto je potrebné aj refaktorizovat.

Návrhové vzory v agilných metódach

Návrhové vzory sú v agilných metódach spravidla podhodnotené. Je to v dôsledku toho, že pre niektorých ludí sa zdajú byt v protiklade s agilnými metódami. Dalším dôvodom je, že niekedy sú návrhové vzory použité až príliš casto, bez odôvodnenia. Svet už má plné zuby legendárneho programátora, ktorý motivovaný známym katalógom návrhových vzorov (*GOF – Gang of four*) uvedie šestnást vzorov v 32 riadkovom kóde. Ale to ešte neznamená, že by tieto návrhové vzory boli zlým nápadom. Len ich treba používat primeraným spôsobom a s rozvahou.

Zaujímavé je, že jednoduchý návrh aj sám vedie k vytvoreniu návrhu, ktorý je velmi podobný návrhu vytvorenému na základe niektorého z návrhových vzorov. Refaktorizácia to robí explicitne, ale aj bez nej, sledovaním pravidiel jednoduchého návrhu, sa dopracujeme k riešeniu, ktoré sa bude podobat na návrh, vytvorený podla niektorého návrhového vzoru, aj ked o nich nic nevieme. Je to síce pravda, ale je to naozaj najlepší

spôsob, ako na to? Samozrejme, existujú jednoduchšie spôsoby. Ak vieme približne k comu sa dopracujeme, je rýchlejšie vyhladat riešenie z knihy (napr. GOF) na tento problém, ak existuje, ako sa k tomu prácne dopracovávat sám. Agilné metódy používajú návrhové vzory trochu iným spôsobom, ako klasické metódy, napr. nepoužívajú návrhové vzory silou-mocou, len ked ich použitie sa vyplatí v hladiska nákladov a casu. To ale neznižuje ich hodnotu.

Dalšou zaujímavostou je, že ludia si myslia o agilných metódach, že odmietajú návrhové vzory. Je to irónia, lebo napr. vela zástancov XP bolo vedúcim hnutia zástancov návrhových vzorov. Oni asi vidia hlbšie do problematiky, alebo vzory majú už tak hlboko zakotvené vzmysli, že to ani nezbadajú. Návrhové vzory sú ešte stále esenciálnou súcastou návrhu. Agilné metódy opisujú proces vývoja softvéru, ale kostru návrhu ešte stále tvoria návrhové vzory, ktoré môžu byt použité nezávisle od metódy. Samozrejme, každá metóda používa vzory inak. XP napríklad používa návrhové vzory iba ak sú naozaj potrebné a uprednostnuje vytvorenie vzoru jednoduchou implementáciou. Niekolko príkladov, ako použit návrhové vzory správne:

- Investovat cas do ucenia sa o vzoroch.
- Sústredit sa na to, kedy používame vzory (nie príliš casto).
- Najprv implementovat návrhové vzory co najjednoduchšie a až potom zvyšovat komplexnost.
- Ak zistíme, že vzor , ktorý sme pridali skôr už nesplna svoju úlohu, nebojme sa ho odstránit

Agilné metódy by mali lepšie zdôraznit potrebu oboznámit sa s existujúcimi návrhovými vzormi.

Záver

Agilné metódy predstavujú relatívne novú vývojovú vlnu v softvérovom inžinierstve. Ich vznik je prirodzeným následkom šírenia softvérových systémov do každej oblasti nášho života a potreby vyvíjat coraz mohutnejšie aplikácie pri meniacich sa podmienkach za co najkratší cas. Pre vela ludí sa môže zdat, že agilné metódy so svojou jednoduchostou môžu znamenat zánik návrhu a že klasické metódy ako UML, flexibilné konštrukcie (*flexible frameworks*) ale aj návrhové vzory (*design patterns*) sú prekonané a ignorované.

V skutocnosti však agilné metódy vsebe vo významnej miere zahrnajú návrh, len iným spôsobom ako klasické. Nevyužívajú síce plánovitý návrh ako taký, ale je to zahrnuté v rámci jednotlivých ciastkových úloh. Aby však mohol byt použitý evolucný návrh, je potrebné vykonat dalšie cinnosti, ktoré sú typické pre agilné metódy. Tými sú predovšetkým testovanie, neustála integrácia a refaktorizácia. Bez týchto cinností evolucným návrhom nemôžeme dosiahnut požadovaný efekt – zrýchlenie vývoja softvéru. Agilné metódy, ako bolo uvedené vyššie, v záujme zrýchlenia vývoja radia, aby sme návrh udržiavali co najjednoduchší. To sa dá tiež dosiahnut dodržiavaním niekolkých základných pravidiel. Takisto sa tu využíva refaktorizácia. Posledná z kapitol sa zaoberala návrhovými vzormi. Tie, na rozdiel od všeobecne rozšíreného názoru, nie sú v agilných metódach ignorované. Používajú sa stále, iba iným spôsobom, ako by sme mohli byt zvyknutí.

Okrem iných, existencia agilných metód spôsobila znovuzrodenie evolucného návrhu tým, že pomocou nich sa stal návrh naozaj realizovatelným. Práve v tom spocíva ich sila. Agilný návrh sa využíva najmä v prípadoch, ked sa casto menia požiadavky na softvér a projekt je preto rizikový. Jeho využitie je výhodné pre malé tímy. Agilné metódy však majú aj svoje nevýhody. Asi najpodstatnejšou je, že od návrhárov sa vyžadujú nové kvalitatívne vlastnosti. Návrhári by mali v prvom rade ovládat jednoduchý návrh, použitie návrhových vzorov v evolucnom návrhu a efektívne využitie refaktorizácie.

Použitá literatúra

[1] Fowler, M.: Is design dead?
http://www.martinfowler.com/articles/designDead.html

[2] Highsmith, J.: Summary of XP
http://www.cutter.com/ead/ead0002.html

Dokumentovanie a modelovanie v agilných metódach

Martin Šindelár

Esej sa zaoberá agilným modelovaním, jeho hodnotami, princípmi a praktikami. Dalej sa zameriava hlavne na dokumentovanie procesu vývoja softvérového systému pri použití agilných metód. Vysvetluje preco a v akých prípadoch je dokumentovanie potrebné, problémy spojené s dokumentovaním a ako možno dokumentovat agilne. Na príklade Extrémneho programovania vysvetluje dokumentovanie v jednotlivých fázach tvorby softvérového systému.

Úvod

Zmenšenie množstva dokumentácie je jedným z princípov agilného vývoja, ktorým si získava mnoho nasledovníkov. Nejde pritom o jednoduché odbúranie dokumentácie, ktoré by rýchlo viedlo ku krachu projektu vyvíjaného týmto spôsobom, ale o odstránenie tých castí, ktorých cena vytvorenia a udržiavania je najvyššia a zároven ich hodnota v meniacom sa prostredí je nízka. Podobne novou výzvou je aj hladanie spôsobu efektívneho využitia modelovania v agilných procesoch.

Agilné modelovanie

Agilné modelovanie (AMo) [1] je súbor praktík tvoriacich metodológiu pre efektívne modelovanie a dokumentovanie softvérových systémov. Je to súbor hodnôt, princípov a praktík pre modelovanie softvéru, ktoré môžu byt efektívne alahko použité pre softvérový projekt. Avšak nie je to kompletný softvérový proces. Neobsahuje aktivity spojené s programovaním alebo testovaním, aj ked hovorí o overení modelu kódom a zvážení testovania. Treba ho použit s dalšou metódou, ako sú napr. XP, DSDM alebo SCRUM.

Nedáva podrobný návod ako urcitý typ modelu vytvorit, ale skôr dáva odporúcania ako byt dobrým tvorcom modelov.

AMo neznamená, že sa modeluje menej, ale niektorí tvorcovia modelov dokonca zistia, že pri použití AMo dokonca modelujú viac než predtým.

Hodnoty

Medzi hodnoty AMo patria hodnoty z XP: komunikácia, jednoduchost, odozva a odvaha spolu so skromnostou.

Modely podporujú komunikáciu medzi tímom a vedúcim projektu a aj medzi vývojármi v rámci tímu.

Vývojári si musia uvedomit, že modely zjednodušujú softvér aj softvérový proces. Je teda jednoduchšie zamýšlat sa nad jednoduchým modelom, ktorý je rozšírením pociatocného nápadu ako zacat písat desiatky ci stovky riadkov zdrojového kódu.

Odozvu dobre vyjadril Kent Beck v [2]: “Optimizmus je pracovné riziko programovania, odozva je na to liekom”. Diskusiou o nápadoch s pomocou modelov teda získame rýchlu odozvu a môžeme tieto rady hned využit.

Odvaha je dôležitá, lebo musíme robit dôležité rozhodnutia a tiež vediet zmenit smerovanie bud zahodením alebo prepracovaním výsledku práce, ak tieto výsledky nezodpovedajú ocakávaniam.

Najlepší vývojári majú skromnost v tom zmysle, že vedia, že nevedia všetko; že ich kolegovia vývojári, zákazníci a další zúcastnení na projekte tiež majú svoje oblasti znalostí a že majú svoju hodnotu pre softvérový projekt. Preto jednajú s každým, kto je nejakým spôsobom zainteresovaný do projektu rovnako a s rešpektom.

Princípy

AMo definuje princípy, ktoré sú pri použití v procese vývoja softvéru základom pre použitie praktík v AMo. Niektoré z princípov boli prebraté z XP a sú dobre opísané v [2], ktoré ich prebralo zo všeobecných techník softvérového inžinierstva. Nasleduje výber z princípov AMo, tak ako sú uvedené na [1]:

Jednoduché riešenie je to najlepšie riešenie. Preto v modeloch nezobrazujeme funkcie, ktoré dnes nepotrebujeme. Neskôr, ako sa požiadavky budú spresnovat, môžeme model

upravit alebo rozšírit. Tým sa podla mna vyhneme tomu, aby sme neskôr venovali zbytocné úsilie prerábaniu takéhoto modelu.

Modelujeme so zretelom na ciel. Mnoho vývojárov rozmýšla nad tým, ci sú nimi vytvorené prostriedky, teda modely, zdrojové kódy alebo dokumentácia, ktoré ešte neskôr použijú, dostatocné alebo príliš detailné a ci sú dostatocne presné. Treba však najprv vediet preco apre koho vlastne tento prostriedok vytvárame. Rovnako to platí aj o zmene v existujúcom modeli – treba mat pre nu dôvod (napr. kvôli zmene požiadaviek). Potom môžeme vytvorit model presný a detailný "na mieru". S tým súvisí aj rozsah, treba vediet komu stací 5 stránkový dokument (manažment – ci už na strane zákazníka alebo dodávatela) a kto potrebuje detailný 100 stránkový (další vývojári).

Myslíme jednoducho. Každý model alebo dokument, ktorý vytvoríme, je potrebné upravovat kedykolvek sa vyskytne zmena (nové alebo zmenené požiadavky alebo nová technológia použitá pri vývoji). Preto by sme ich mali vytvárat do takej miery detailnosti, aká je skutocne potrebná. Inak by sme strávili príliš mnoho casu modelovaním alebo tvorbou dokumentácie na úkor vlastného kódovania.

Obsah je dôležitejší než jeho reprezentácia. Je teda jedno, ci model nakreslíme na papier alebo použijeme nejaký CASE nástroj. Vždy to bude len vstup pre dalšiu fázu, ktorou zvycajne býva kódovanie. Dôležité je uvedomit si výhody modelovania, bez prílišného ohladu na jeho cenu. Zbytocne teda budete strácat cas kreslením modelu v CASE nástroji, ktorý ani nemusíte podrobne ovládat, ak model na papieri postacuje a nie je potrebný niekde inde.

Každý sa môže ucit od niekoho iného. V dnešnej dobe neustálej zmeny a príchodu nových technológií (XML, C#, .NET, JavaOne) treba využit každú príležitost naucit sa nieco nové. Vaši kolegovia vám v tom tiež môžu pomôct a je jednoduchšie, ak vám niekto vysvetlí urcité základy tak, aby ste de facto mohli hned pracovat, než od zaciatku študovat obsiahle príručky.

Modely, ktoré vytvárame, musíme dobre poznat. To vám pomôže ich efektívne použit v dalších fázach vývoja.

Taktiež musíme dobre poznat nástroje, používané pri vývoji. Softvér, ktorý na modelovanie používame má urcite vela funkcií a možností. Preto by ste ho mali ovládat, aby ste vedeli kedy a akú funkciu použit, prípadne nepoužit.

Praktiky

Praktiky AMo sú založené na princípoch AMo. Niektoré sú prevzaté z XP [2].

Musíme vediet použit ten správny prostriedok. Každý prostriedok má svoju vlastnú úlohu. Ak je obrázok lepší ako tisíc slov, potom je model lepší ako 1024 riadkov zdrojového kódu. Musíte poznat silné a slabé stránky jednotlivých prostriedkov a vediet, kedy je vhodné ktorý z nich použit alebo nepoužit.

Viacero modelov vytvárame naraz. Žiadny model nie je sám osebe dostatocný. V kombinácii s praktikou *prechodu k dalšiemu prostriedku* urcite zistíme, že sme ovela viac produktívni, ked budeme pracovat na viacerých modeloch paralelne, než ked sa zameriame len na jeden.

Ak si nevieme rady, prejdeme k dalšiemu prostriedku. Ak máme problémy stvorbou urcitého prostriedku, skúsime prejst na iný. Zmenou pohladu tak prípadne môžeme prist na problémy v tom prvom prostriedku.

Modelujeme s ostatnými. Ak *modelujeme so zretelom na ciel*, casto krát zistíme, že modelujeme preto, aby sme niecomu porozumeli alebo kvôli dalšej komunikácii s inými clenmi tímu. Ak chce potom takýto tím vytvorit jadro modelov potrebných pre projekt, musí pracovat spolocne.

Model overujeme kódom. Model je abstrakcia odrážajúca aspekty toho, co sa snažíme vytvorit. Ale je tento model správny, dostatocný a presný? To zistíme, ak ho overíme kódom. Pri iteracnom procese je modelovanie len jednou zcastí, po nom nasleduje kódovanie a testovanie.

Modelujeme pre porozumenie. Najdôležitejšou úlohou modelovania je preskúmanie modelovaného problému, identifikácia a analýza požiadaviek, ale aj porovnanie potenciálnych alternatív pre výber najvhodnejšej alebo najjednoduchšej z nich.

Používame štandardy. Tak, ako ked sa držíme urcitých štýlových konvencií pri kódovaní, aj pri modelovaní je potrebné používat dohodnuté štandardy (napr. UML). Rozdiel medzi štandardom a štýlovou konvenciou je ten, že napríklad použitie obdlžnika pre model triedy je štandard a umiestnenie podtriedy pod jej nadtriedu je štýlovou konvenciou.

Nepotrebné modely zahadzujeme. Väcšina modelov, ktoré vytvoríme má len docasný charakter. Podla nich vytvorený

zdrojový kód sa môže zmenit a už tomuto modelu nemusí zodpovedat. Ak by aktualizácia tohto modelu mala svoje opodstatnenie a hodnotu pre projekt, tak to urobíme. Inak je lepšie tento model vzhladom k nákladom zahodit.

Modelujeme pre komunikáciu. Dôvodom pre modelovanie je aj potreba komunikácie s "okolitým svetom", teda ludmi mimo tímu pracujúceho na softvérovom projekte. Preto sa oplatí investovat trochu casu a mierne tieto modely "skrášlit" použitím špecializovaného CASE nástroja.

Potreba dokumentácie

Dokumentácia je dôležitou súcastou vývoja softvérového systému. Jej tvorba a údržba je ale casto pre väcšinu vývojárov „nutné zlo". Môže sa však stat agilným aspektom vývoja softvéru a je viacero dôvodov, preco ju vytvárat.

Kedže využívame zákazníkove peniaze aj na tvorbu dokumentácie, mal by to byt on, kto rozhodne, že sa tieto peniaze použijú práve takto. Ak dokumentáciu vyžaduje, musíme ju vytvorit. Znamená to teda, že dokumentáciu treba vytvárat, len ak ju zákazník vyžaduje? Mohlo by sa zdat, že áno. Ale skutocnost je asi iná. Zástupca dodávatela jedná s viacerými ludmi, nielen so zákazníkom a má teda dôvody, preco dokumentáciu vyžaduje, pre koho a v akej miere detailnosti.

Dalším opodstatnením dokumentácie je komunikácia s externou skupinou ludí zainteresovaných na projekte. Ak potrebujete pracovat s takouto skupinou ludí, zdielaná dokumentácia je spolu s komunikáciou, ci už priamou alebo elektronickou poštou mnohokrát súcastou riešenia. Dokumentácia však nie je dobrá ako primárny spôsob komunikácie, môže sa totiž stat, že ju citatel zle pochopí; je ale dobrým podporným mechanizmom.

Ak si potrebujeme niektoré veci ujasnit, nie je nic lahšie ako nazriet do dokumentácie, aj preto bola vytvorená. Vela ludí tvorí dokumentáciu preto, že si chcú potvrdit výsledky svojej práce, alebo pri jej tvorbe môžu samotnému problému viac porozumiet.

Problémy spojené s dokumentovaním

Vytvorenie efektívnej dokumentácie, teda presnej, správnej, v postacujúcom a zároven dostatocnom množstve v správny cas a pre správne osoby závisí od rôznych faktorov.

Základným problémom je "vývoj dokumentácie" verzus vývoj softvéru. Cas strávený sdokumentáciou chýba pri vývoji. Treba vidiet primárny ciel, ktorým je tvorba softvéru.

Softvéroví vývojári majú znalosti, technickí pisatelia majú schopnosti. Pravdou je, že málo technicky nadaných ludí má schopnost problém vysvetlit iným, teda napísat dobrú dokumentáciu. Ale najlepšou osobou pre napísanie dokumentácie k softvérovému produktu je práve vývojár, ktorý o tomto produkte vie najviac. Preto môže byt urcitým kompromisom, ak pociatocnú verziu dokumentácie napíše vývojár, ktorý ju dá "ucesat" technicky zdatnejšiemu pisatelovi.

Mali by sme dokumentovat pocas vývoja, alebo až po nom? Dokumentovanie pocas vývoja má tú výhodu, že všetky podrobnosti máme práve v hlave a to co je dôležité hned zdokumentujeme. Nevýhodou však je, že ak sa produkt mení, s meniacimi sa požiadavkami je potrebné menit aj túto dokumentáciu. Takže nás tento spôsob nielen spomaluje, ale aj mrháme casom aprostriedkami. Opacným extrémom je tvorba dokumentácie, až ked je vytváraný produkt hotový. Výhodou je, že opisujeme hotový a otestovaný produkt. Nevýhodou ale je, že môžeme zabudnút na dôvody urcitých rozhodnutí a taktiež už môžu byt niektorí z ludí, ktorí produkt vyvíjali alebo by mohli písat dokumentáciu nedostupní. Zlatou strednou cestou je potom robenie si poznámok o dôležitých myšlienkach a rozhodnutiach popri vývoji napríklad do zdrojového kódu (a nemyslí sa tým štandardné komentovanie kódu). Výhody tohto prístupu si zvlášt uvedomíme vtedy, ak by sa náhodou zmenilo obsadenie tímu vývojárov, alebo keby v projekte mal pokracovat niekto iný. Ak by sa dokumentácia robila až na konci, nemali by sme okrem zdrojového kódu nic.

Dalším problémom je umiestnenie dokumentácie. Dokumentujeme len zdrojový kód, alebo je všetka dokumentácia naopak externá? Tu opät platí, že treba nájst nejakú optimálnu strednú cestu v závislosti od toho, komu je dokumentácia urcená. Ak pôjde ovývojárov, potom je samozrejme vhodné umiestnit väcšinu dokumentácie priamo do zdrojového kódu. Okrem toho

ale bude potrebná dokumentácia obsahujúca architektúru systému. Ak sú však vývojári múdri, tak tejto dokumentácii, ba ani tej v zdrojovom kóde nebudú slepo dôverovat. Zostávajú ale skupiny ludí, pre ktorých treba "normálnu" dokumentáciu vytvorit, ci už preto, že k zdrojovému kódu nemajú prístup, alebo mu nemôžu porozumiet (napr. používatelia alebo manažéri).

V neposlednom rade tiež závisí na kvantite a kvalite dokumentácie. Urcite je lepšia kvalitná dokumentácia menšieho rozsahu 20-30 strán odkazujúca na zdrojový kód alebo dalšie dokumenty než dokumentácia obsahujúca stovky, ci dokonca tisícky strán. Samozrejme, že v tomto prípade velmi závisí aj od rozsahu softvérového projektu, kde pri väcších projektoch asi nebudú 1000 stránkové prírucky výnimkou.

Agilná dokumentácia

Aj ked sa to možno na prvý pohlad nezdá, dokumentácia môže byt efektívna. Pre to, aby bola dokumentácia agilná existujú rôzne kritériá.

Dokumentácia musí maximalizovat investície. Výhody a zisk z agilnej dokumentácie sú casto vyššie, než investície vložené do jej tvorby. Dokumentácia teda musí priniest aspon tolko, kolko do jej tvorby vložíme. Ak by napríklad dokumentácia mala priniest len 50% do nej vložených investícii a školenie 100%, potom niet co rozmýšlat.

Dokumentácia musí byt jednoduchá a prostá. Teda musí obsahovat len to, co je pre daný úcel skutocne potrebné a tento úcel aj splnat. Niektoré casti sú napríklad namiesto uceleného textu lepšie ako jednotlivé body alebo odrážky. Ak neviete, preco vytvárate urcitý dokument alebo máte o jeho úcele urcité pochybnosti, bude lepšie ak si to ešte premyslíte.

Ako už bolo spomenuté na inom mieste, dokumentácia okrem úcelu má aj svojho adresáta, teda toho komu je urcená. A s tým treba pri jej tvorbe pocítat.

Dokumentovanie v Extrémnom programovaní

Mimo softvérového projektu je dokumentácia urcite potrebná. Vnútri softvérového projektu je však tolko verbálnej

komunikácie, že nebudete potrebovat nic iné. Samozrejme ak však nie je váš tím zložený z navzájom vzdialených ludí.

Dokumentovanie požiadaviek

XP používa na vysvetlenie toho co je potrebné verbálnu diskusiu. Tá je podporená rôznymi tabulkami alebo výnatkami z dokumentácie požiadaviek z iného projektu.

Dokumentácia pozostáva zdvoch castí. Prvou je karta obsahujúca rôzne nápady a požiadavky s odkazmi na podpornú dokumentáciu a druhou, asi najdôležitejšou castou, je sled rozhovorov medzi zákazníkom a vývojármi. Tieto rozhovory budú zapísané ako dodatky ku karte.

Niektoré projekty majú potrebu diskusie o požiadavkách mimo tímu. Toto sa však netýka procesu v XP. Ak je na business úrovni potrebná dokumentácia, tak by ju mal zákazník vyžadovat rovnako ako vyžaduje nové vlastnosti produktu. Tím odhadne cenu takejto dokumentácie a zákazník ju bude vyžadovat v case, kedy mu to bude najviac vyhovovat.

Dokumentovanie návrhu

Tak, ako aj iné inkrementálne metódy, aj v XP je návrh postupný. Systém je tvorený v dvojtýždnových iteráciách a funkcné vlastnosti sú vytvorené za pár dní.

V takomto prostredí majú jednotlivé prostriedky návrhu krátky život. Tím môže nakreslit UML diagram na tabulu a potom si sadnút a vytvorit novú vlastnost produktu.

Pre tímy XP je typické, že majú na tabuli nakreslené casti návrhu systému, o ktorých predtým spolu diskutovali. Pri nasledovnom kódovaní ich tak majú stále na ociach.

Vývojári používajúci XP vedia, že ak tím potrebuje dokumentáciu, ci už diagram, tabulku alebo opis, tak by ju mali vytvorit. Tiež vedia, že ak urcitú vytvorenú dokumentáciu nepoužijú, tak asi vôbec nie je potrebná.

Dokumentovanie kódu

XP sa zameriava na inkrementálny vývoj. Vývojový cyklus zacína jednoduchým návrhom, ktorý pokracuje programovaním v páre. Dalej nasleduje testovanie modulov a potom cyklus ide znova tzv. refaktorizáciou.

Kód musí byt cistý a jasný, co sa dosahuje urcitými pravidlami. Základom je to, že každú novú myšlienku pri programovaní treba vyjadrit v kóde.

Komentáre tvoria hlavnú cast dokumentácie kódu. Cokolvek nejasné v kóde treba okomentovat. Komentáre je vhodné tvorit už popri kódovaní, aby ste sa vyhli prípadným neskorším nejasnostiam a prípadne aj strate casu opätovným štúdiom svojho vlastného kódu, co z vlastnej skúsenosti môžem len potvrdit.

Ako pri návrhu, aj tu je potrebné nieco viac než len okomentované dôležité casti kódu, ak ide kód mimo tímu. Takže treba viac než komentáre k "nejasným" castiam kódu.

Prírucky

Každý, kto už vytvoril nejaký softvérový produkt vie, že takmer nikto necíta používatelskú prírucku (Aj ked to asi platí len v komunite programátorov. Pri špeciálnych softvérových systémoch na objednávku to asi neplatí.). Vieme to, lebo nikto z nás ich tiež necíta.

Aplikácie na Internete nemajú prírucky a predsa sú lahko použitelné. Niektoré majú zopár stránok spomocou alebo len krátky návod. Jediným vodítkom je potom sled stránok alebo tlacidiel.

Cím dalej, tým viac softvéru sa dodáva na CD bez tlacených príruciek. Príkladom môže byt firma Microsoft, ked manuály ku svojim produktom úplne prestala dodávat. Rozhodla sa tak asi z financných dôvodov, ale možno aj z výsledkov nejakého výskumu "cítanosti" a potrebnosti takýchto príruciek. Pre tých, ktorí predsa len manuál potrebujú založila nakladatelstvo Microsoft Press, v ktorom vydáva knihy od základných príruciek až po tzv. Resource Kit-y. Býva však dobrým zvykom, že prírucka je na CD ako PDF súbor alebo objemnejšia pomoc v podobe HTML stránok. Neocenitelnou výhodou takéhoto riešenia oproti tlacenej prírucke je možnost rýchleho vyhladávania, hlavne v prípade tzv. referencných príruciek. Na druhú stranu používatelská prírucka sa predsa len lepšie cíta v tlacenej podobe ako z monitora.

Snažme sa preto vytvorit lahko použitelný produkt, ktorý žiadnu prírucku nepotrebuje. Používatelské rozhranie by malo byt "samovysvetlujúce", takmer ako web stránka. Poteší to väcšinu zákazníkov, ktorí prírucky vôbec necítajú.

Podobne ako pri komentovaní kódu aj prírucku vytvárajte priebežne, tak ako pridávate jednotlivé funkcie systému.

Záver

V agilnom modelovaní ide o zmenšenie množstva vytváraných modelov a dokumentácie. V súcasnom rýchlo sa vyvíjajúcom prostredí to ani inak nejde. Nemôžeme zbytocne strácat cas tvorbou niecoho, co ani príliš potrebné nie je a nebude.

Za každú cenu sa nemusíme snažit dodržat model, budme prístupní k jeho zmene a prípadne aj k jeho zahodeniu.

Na zreteli treba mat hlavný ciel, teda vytváraný softvérový systém a jeho použitelnost. Cielom urcite nie je dokumentácia, preto by ani tá nemala byt prvoradá, je to iba pomôcka, ktorej množstvo sa dá znížit napríklad aj osobnými rozhovormi.

Záleží však aj od konkrétneho projektu, kto na nom pracuje akto je zákazníkom. Dokumentácia, ak je vyžadovaná, môže totiž v niektorých prípadoch celý projekt do istej miery spomalit. Treba vediet odhadnút, co skutocne zákazník potrebuje.

Ak by som mal porovnat klasický, teda plánom riadený vývoj a agilné metódy z pohladu modelovania a dokumentovania, potom jasne vedú práve agilné metódy. Šetria cas a náklady a scasti aj ludí, ktorí nemusia tvorit nieco, o com neskôr zistia, že to nebolo vôbec potrebné, alebo že je to nepoužitelné.

Samozrejme ale agilné metódy nie je možné použit pri všetkých druhoch projektov. Stále zostávajú urcité oblasti, v ktorých sa klasickému vývoju softvéru nevyhneme.

Použitá literatúra

[1] The Official Agile Modeling site
http://www.agilemodeling.com

[2] Beck, Kent: Extreme Programming Explained: Embrace Change. 1999. Addison-Wesley Pub Co. ISBN 0-201-61641-6.

[3] Jeffries, Ron: Essential XP: Documentation
http://www.xprogramming.com/xpmag/expDocumentationInXP.htm

[4] Jeffries, Ron: Manuals in Extreme Programming
http://www.xprogramming.com/xpmag/manualsInXp.htm

KYKLOP

Názov: Tvorba softvéru v trtom tisícrocí

Vedúci tímu: Bc. Rastislav Habala
Editor:: Bc. Vladimír Marko
Redaktori: Bc. Peter Diko
Bc. Vladimír Grlický
Bc. Ladislav Gyökeres
Bc. Jozef Hudec
Bc. Richard Krupa
Bc. Jaroslav Kuruc
Bc. Martin Šindelar
Integrácia a grafická úprava: Bc. Michal Bigoš

Vydalo: KYKLOP, Ilkovicova 3, Bratislava
Rok vydania: 2002
Vydanie: Prvé
Náklad: 1 výtlacok
Rozsah: 82 strán