

Tvorba softvéru v treťom tisícročí

Hobiti



SLOVENSKÁ
TECHNICKÁ
UNIVERZITA
V BRATISLAVE

Tvorba softvéru v tret'om tisícročí

Tvorba softvéru v tret'om tisícročí

Hobiti

Slovenská technická univerzita
2002

© Hobiti, 2002

Bc. Jozef Drgoňa
Bc. Martin Geleneky
Bc. Jozef Krššák
Bc. Marek Kubran
Bc. Radoslav Otipka
Bc. Ivan Polák
Bc. Richard Richter
Bc. Mária Šmigurová
Bc. Anton Weissensteiner

Lektori:

Doc. Ing. Mária Bieliková, PhD.
Ing. Peter Dolog

Publikácia neprešla jazykovou úpravou.

V publikácii sú použité citáty z trilógie

J.R.R.Tolkien: Pán Prsteňov.

Ilustrácie: John Howe.

Obsah

PREDHOVOR	vi
AGILNÉ METÓDY	1
ODEĽAHČENÝ VÝVOJ	7
POŽIADAVKY	10
VÝVOJÁRI.....	11
MANAŽÉRI PROJEKTU	14
ZÁKAZNÍCI	16
ODEĽAHČENÝ VÝVOJ A INÉ AGILNÉ PRÍSTUPY	17
ZÁVER	17
ADAPTÍVNY VÝVOJ SOFTVÉRU.....	21
PREČO ADAPTÍVNY VÝVOJ SOFTVÉRU?.....	23
ČO ZNAMENÁ „SPOLUPRACOVAŤ“?	27
ADAPTÍVNY ZÁKAZNÍK	28
ČO ZNAMENÁ „UČENIE SA“?	30
PREČO ZLYHÁ DOBRÝ „MANAŽÉR“?	31
ZÁVER	32
VÝVOJ RIADENÝ POMOCOU ČRT	35
PRISPÔSOBOVANIE SA STÁLE KRATŠÍM A KRATŠÍM CYKLOM	37
ČRTY	39
VYTVÁRANIE CELKOVÉHO MODELU SYSTÉMU.....	40
PREČO VÔBEC POSTUP?.....	42
ROLY VO VÝVOJOVOM TÍME	44
VEDÚCI PROGRAMÁTORI.....	44
VLASTNÍCI TRIED.....	45
TÍMY PRE JEDNOTLIVÉ ČRTY	45
SLEDOVANIE POSTUPU.....	46

ZÁVER.....	47
PRIEZRAČNÉ TECHNIKY	49
ČO PODNIETILO VZNIK PRIEZRAČNÝCH TECHNÍK?	50
ČO SÚ TO PRIEZRAČNÉ TECHNIKY?	52
ZÁKLADNÉ KAMENE.....	52
ĽUDIA	53
PRIEZRAČNÉ TECHNIKY PRE NAJMENŠIE TÍMY	55
ODPORÚČANIA <i>CRYSTAL CLEAR</i>	55
ČO PRINÁŠA <i>CRYSTAL CLEAR</i> ?	57
KEDY NEPOUŽÍVAŤ <i>CRYSTAL CLEAR</i>	57
AKÝCH ĽUDÍ POTREBUJEME PRI <i>CRYSTAL CLEAR</i>	58
AKÉ ŠTANDARDY POUŽÍVA <i>CRYSTAL CLEAR</i> ?.....	58
AKO PREBIEHA PRÁCA NA PROJEKTE VYUŽITÍM <i>CRYSTAL CLEAR</i>	59
ZÁVER.....	61
EXTRÉMNE PROGRAMOVANIE	63
NA ČOM STOJÍ EXTRÉMNE PROGRAMOVANIE?	64
ZÁKLADNÉ PRAKTIKY	65
KEDY JE EXTRÉMNE PROGRAMOVANIE ROZHODNE DOBRÝ NÁPAD?	68
EXTRÉMNE PROGRAMOVANIE NIE JE LIEK NA VŠETKO.....	69
DOKUMENTÁCIA	70
ZÁVER.....	71
SKRUMÁŽ	75
ČO JE TO SKRUMÁŽ.....	78
APLIKOVANIE METÓDY SKRUMÁŽ.....	78
ITERATÍVNY VÝVOJ SOFTVÉRU	80
DVA ZÁKLADNÉ PILIERE METÓDY SKRUMÁŽ	81
ZÁKLADNÉ ÚLOHY TÍMU A MANAŽMENTU	82
AKO FUNGUJE SKRUMÁŽ.....	83

PREČO SKRUMÁŽ FUNGUJE	84
ZÁVER	86
DYNAMICKÝ VÝVOJ SYSTÉMOV	89
DYNAMICKÝ VÝVOJ SYSTÉMOV A TRADIČNÝ PRÍSTUP	91
PREDSTAVENIE METÓDY	92
DYNAMICKÝ VÝVOJ SYSTÉMOV A INÉ PRÍSTUPY	98
DYNAMICKÝ VÝVOJ A EXTRÉMNE PROGRAMOVANIE	98
DYNAMICKÝ VÝVOJ A <i>RATIONAL UNIFIED PROCESS</i>	99
ZÁVER	101
AGILNÉ MODELOVANIE	105
ČO (NIE) JE AGILNÉ MODELOVANIE	107
HODNOTY AM.....	108
PRINCÍPY AGILNÉHO MODELOVANIA	109
KEDY JE MODEL AGILNÝM	110
KEDY AGILNE (NE)MODELUJETE	111
ZÁVER	112
NAMIESTO ZÁVERU.....	115
LITERATÚRA.....	119
REGISTER.....	123



Predhovor

Softvérové systémy sa chtiac či nechtiac stali neoddeliteľnou súčasťou nášho života. Nájdeme ich takmer všade – v elektronických pokladniciach pri nákupe v obchode, v práčkach, mikrovlnných rúrach, automobiloch alebo dokonca vo výťahoch. Dohliadajú na chod atómovej elektrárne a riadia let človeka do vesmíru. Pomáhajú určovať diagnózu chorých.

Dnešná doba kladie vysoké nároky na softvérové systémy, na ich efektívnosť, spoľahlivosť, jednoduché a príjemné používanie. Keďže sa potreba softvéru stále zvyšuje, je dôležité systematicky a efektívne pristupovať k jeho tvorbe. Na zabezpečenie tohoto cieľa vznikli rôzne postoje, metódy a prístupy. Staršie z nich, ako sú SSADM alebo OMT môžeme označiť ako klasické.

Popri nich nedávno vznikli nové, ktoré sa označujú pojmom *agilné metódy*. A práve tomuto novému postoju k vývoju softvéru, ktorý sa javí ako trend smerovania disciplíny softvérové inžinierstvo v budúcnosti, sa venuje aj naša publikácia. Predkladá pomerne ucelený prehľad jednotlivých metód a prístupov. Zameriava sa na ich významné myšlienky a opisuje ich prínos a rozdiely voči klasickým metódam.

Na rozlíšenie rozsahu jednotlivých postojov používame dva termíny – *metóda* a *prístup*. Metóda označuje komplexnejší postoj k tvorbe softvéru. Okrem pravidiel a odporúčaní definuje pracovný postup, teda aj poradie a podmienky podľa ktorých jednotlivé odporúčania použiť. Prístup možno chápať iba ako množinu pravidiel a odporúčaní.

Čakala nás aj ďalšia zodpovedná úloha. Agilné metódy sú pomerne mladou oblasťou, v ktorej ešte nebola vytvorená jednotná slovenská terminológia. Neexistuje ani publikácia podobného zamerania, z ktorej terminológie by sme mohli vychádzať.

Pri preklade názvov jednotlivých metód a prístupov do slovenčiny sme chceli pôvodný význam pomenovania. No nebolo to vždy jednoduché. Napríklad postoj *Crystal Methods* je označený ako metóda, ale jedná sa o prístup. Nakoniec sme sa rozhodli pre názov *Priezračné techniky*, ktorý podľa nás „neznie až tak krkolomne“.

V úvode publikácie analyzujeme príčiny, ktoré viedli k vzniku agilných metód. Vysvetlíujeme základné rysy týchto metód a porovnávame ich s vlastnosťami klasických metód.

V ďalších kapitolách sú spracované jednotlivé metódy a prístupy. Najskôr sme zaradili nezávislé, ucelené metódy *Odľahčený vývoj* a *Adaptívny vývoj softvéru*. Ďalej nasledujú agilné prístupy zamerané hlavne na samotný vývoj softvérových systémov, a to *Vývoj riadený pomocou črt*, spomínané *Priezračné techniky* a *Extrémne programovanie*. Nakoniec sme zaradili metódy orientované na riadenie – *Skrumáž*, *Dynamický vývoj systémov* a *Agilné modelovanie*.

Jednotlivé metódy a prístupy sme sa snažili priblížiť formou esejí. Keďže v rámci esejí sme charakterizovali jednotlivé metódy, eseje pôsobia ako odborný článok. A hoci sa nám podarilo ujednotiť na slovenských označeniach týchto postojov, pri ich štúdiu sme si natoľko zvykli na pôvodné anglické skratky, že sme ich s obľubou používali aj v texte kapitoly pri ich výklade. V úvode každej kapitoly sme teda uviedli pôvodný anglický názov aj použitú skratku.

V závere publikácie uvádzame pre porovnanie stručné charakteristiky opísaných metód.

Publikácia, ktorá sa Vám dostala do rúk, je svojím spôsobom jedinečná. Nie je dielom skúsených softvérových inžinierov, ktorí sa podieľali na vývoji niekoľkých väčších či menších softvérových systémov a ktorí vedia zasvätené zhodnotiť výhody a prínos nových prístupov k tvorbe softvéru. Vytvorili ju študenti softvérového inžinierstva popri rôznych ďalších študijných povinnostiach počas inžinierskeho štúdia ako svoju prvú publikáciu podobného zamerania a rozsahu.

Pri tejto činnosti nám veľmi pomohli naši pedagógovia, Mária Bieliková a Peter Dolog, ktorí nás viedli počas semestra na predmete Manažment v softvérovom inžinierstve, v rámci ktorého vznikla aj táto publikácia. Chceme im vysloviť vďaku za ich cenné pripomienky a rady, ktoré pomohli skvalitniť obsah našej publikácie. Túto publikáciu by sme chceli venovať hlavne im a tiež študentom softvérového inžinierstva.

Na záver by sme Vám chceli popriať veľa trpezlivosti pri čítaní tejto útlej knižky. Veríme, že bude mať pre Vás aspoň taký prínos, aký mala jej tvorba pre nás, Hobitov...



Agilné metódy

Bc. Anton Weissensteiner

Čo sú to agilné metódy a prístupy k vývoju softvérových systémov? Čo podnietilo ich vznik? Aké výhody majú oproti ostatným, klasickým metódam? Na tieto otázky predkladá autor v nasledujúcom príspevku vlastné odpovede.

„Hobiti - smrteľné plemeno malých ľudí. Málokedy presiahnu výšku 120 centimetrov. Obývajú Kraj, kludnú, utešenú krajinku v zapadnutom kúte Stredozeme. Nemajú radi techniku a veľmi radi jedia, pokiaľ možno šesť krát za deň. Majú veľké, chlpaté chodidlá a nikdy nenosia topánky. Celkovo sú to mierumilovní, okúzľujúci ľud, ktorý si žije sám pre seba a teší sa z jednoduchých životných radostí.“

Všetko sa to začalo na konferenciách NATO v rokoch 1968-1969, kde sa diskutovalo o probléme označovanom ako softvérová kríza. Jedná sa o výrazné zaostávanie vývoja softvéru za hardvérom, ktoré bolo spôsobené viacerými faktormi. Jedným z najzávažnejších bol neefektívny prístup k tvorbe softvéru.

Na základe analýzy mnohých projektov, úspešných aj neúspešných, boli definované metódy a prístupy, ktorých cieľom je systematickým spôsobom zlepšiť akosť softvéru, najmä jeho spoľahlivosť, a zvýšiť produktivitu programátorov. Vznikla disciplína softvérové inžinierstvo [Bielik2000].

Metódy softvérového inžinierstva teda vychádzali zo prístupu k práci pri vývoji, snažili sa ich zefektívniť. Boli nimi ako aj požiadavkami, ktoré sa kládli na vytvárané softvérové systémy v dobe vzniku disciplíny, ovplyvnené. Tieto systémy boli zväčša určené pre konkrétnu hardvérovú architektúru. Boli to (z toho dôvodu) komplexné a väčšinou uzavreté aplikácie.

Tieto metódy dnes už označujeme termínom klasické. Sú charakteristické snahou o podrobnú analýzu problému a špecifikáciu požiadaviek v úvodnej fáze vývoja produktu. Sú pomerne robustné, orientované všeobecne – snažia sa pokryť čo najväčšiu množinu aplikačných oblastí softvéru. Základnými hodnotami, na ktorých sú tieto metódy postavené, sú čas a peniaze. Istú úroveň kvality softvéru, ktorá vyjadruje tretiu základnú hodnotu softvérového produktu a najťažšie sa ohodnocuje, garantuje softvérový proces definovaný tou-ktorou metódou.

Vzhľadom na to, že softvérové systémy sú považované za najzložitejšie ľudské výtvory, má všeobecnosť klasických metód vývoja softvéru aj určitú nevýhodu. Proces definovaný metódou nemôže byť vo všetkých prípadoch optimálny. Klasické metódy predstavujú overenú cestu, ktorá garantuje výstup na určitej úrovni kvality. Redukujú riziko zlyhania projektu.

Zavedenie softvérového systému do praxe upravuje výrobný proces v aplikačnej oblasti, pre ktorú je softvérový systém určený. Toto platí nielen pre aplikačné metódy, ale aj pre softvérové inžinierstvo samotné. Definovaním metód, vývojom pomocných pracovných nástrojov, ktoré automatizujú niektoré činnosti, vznikom vyšších programovacích jazykov sa upravila doba trvania jednotlivých činností, vzťahy medzi nimi aj doba vývoja softvérového produktu.

Zavádzaním nových technológií sa upravuje aj cieľová aplikačná oblasť, menia sa požiadavky na vytváraný produkt. Vznikajúce štandardy zjednodušujú interoperabilitu, uľahčujú znovupoužitie a vytvárajú tak základ pre otvorené, užšie špecializované znovupoužiteľné a jednoducho integrovateľné riešenia. Snáď najväčší vplyv na zmenu požiadaviek na aplikácie má rozšírenie Internetu.

Iste, nemožno tvrdiť, že všetko, čo platilo v klasických metódach, v týchto zmenených podmienkach prestáva platiť. Klasické metódy dajú stále používať úspešne aj dnes. Nové prostredie však vytvára pri použití klasických metód isté problémy. Napríklad je náročné v rýchlom sa vyvíjajúcom prostredí definovať všetky požiadavky a úspešne odhadnúť trend vývoja samotného prostredia. Treba uvedomiť, že softvérová kríza stále pretrváva, takže žiadna metóda ani prístup neboli zatiaľ zázračnou guličkou, liekom, ktorý by nám ju pomohol odstrániť.

Akosi odpoveďou na vzniknutú situáciu je iniciatíva softvérových inžinierov nazývaná *Agilná aliancia (Agile Alliance)*. Ich cieľom je odkrývať lepšie cesty vo vývoji softvéru a pomáhať iným pri ich hľadaní. Svoj zámer vyjadrili v Manifeste agilného vývoja softvéru (*Manifesto for Agile Software Development*) [Agile2001].

Výsledkom ich snaženia sú nové metódy a prístupy (nie vždy sa jedná o ucelenú metódu, väčšinou je to len množina odporúčaní), ktoré sa označujú ako *agilné metódy (agile methods)*. Upravujú proces vývoja softvéru, aby lepšie odpovedal dynamickému prostrediu vytváraných softvérových systémov, prinášajú nový pohľad na ich vývoj.

Treba si uvedomiť, že *softvér vytvárajú ľudia a je určený pre ľudí*. Väčšina agilných metód sa teda viac ako na samotný

proces orientuje na ľudí, ktorí vytvárajú softvérový systém, na komunikáciu medzi nimi, snažia sa zapojiť zákazníka do procesu vývoja softvéru.

Komunikácia je jedným z kľúčových momentov tímovej spolupráce. Aj keď máme v súčasnosti vyspelé komunikačné nástroje, ako sú mobilné komunikačné siete, elektronická pošta, internetové fóra či virtuálne konferencie, stretávanie ľudí, ich fyzický kontakt, je stále najefektívnejšou formou komunikácie, najrýchlejším prostriedkom, ako dospieť k dôležitému rozhodnutiu, prostriedok, ako vytvoriť zo skupiny programátorov vývojový tím.

Zapojením zákazníka do vývoja softvérového systému získa zákazník lepší prehľad o stave projektu, lepšiu predstavu o náročnosti jeho vývoja aj lepšiu možnosť definovať a diskutovať svoje požiadavky na výsledný produkt.

Zmena požiadaviek sa nepokladá za nežiadúci jav. Je prirodzenou súčasťou vývoja produktu v rýchlo sa meniacom prostredí. Namiesto identifikácie všetkých možných požiadaviek vo fáze analýzy sa vývoj produktu orientuje vo fáze návrhu tak, aby bolo možné neskôr jednoducho realizovať zmeny.

Metódy sú zamerané na redukciiu množstva nadbytočnej práce, aby sa skrátila doba od začiatku vývoja po dodanie prvého prototypu. Znamená to zníženie viditeľnosti procesu a vnáša väčšiu pravdepodobnosť rizík spojených s tvorbou produktu. Na druhej strane tým získame zvýšenie efektívnosti.

Tak čo konkrétne agilné metódy prinášajú? V nasledujúcich kapitolách sa nachádza prehľad niektorých z nich.



Odl'ahčený vývoj

Bc. Jozef Drgoňa

Na základe princípov a postupov, ktoré sa používajú v priemyselnej výrobe, vznikla agilná metóda Odl'ahčený vývoj (LD). Jej zámerom je vytvoriť taký typ procesu vývoja softvéru, ktorý v čo najväčšej miere odzrkadľuje požiadavky zákazníkov. LD sa snaží eliminovať všetky nadbytočné procesy, ktoré súvisia s vývojom softvéru, s cieľom skrátiť čas odozvy na požiadavky zákazníkov. Predstavuje alternatívu k zaužívaným prístupom. Je však vhodné porovnávať priemyselnú výrobu s procesom tvorby softvéru? Môže byť takýto prístup úspešný v podmienkach silného konkurenčného boja o zákazníka?

„Rada je nebezpečný dar, aj od múdreho múdremu a každá cesta môže viesť k zlým koncom.“

Tak to už býva. Skúsenosti z jedného odvetvia priemyslu sa môžu aplikovať aj v inom odvetví. Keďže proces vývoja softvéru sa v určitých aspektoch podobá procesu vývoja nových priemyselných produktov, vzniká tu možnosť aplikovať skúsenosti z priemyslu do oblasti manažmentu v softvérovom inžinierstve.

Všetko to začalo na konci druhej svetovej vojny, kedy *Toyota Sakichi* sníval o vývoji áut pre širokú verejnosť. Poveril *Taiichi Ohno* zavedením efektívneho produkčného systému, ktorý by produkoval automobily vysokej kvality. V nasledujúcich troch dekádach bol vyvinutý systém, ktorý sa nazýval *Toyota Production system*, celosvetovo známy pod názvom *Lean Manufacturing* [Jones1990]. Tento systém úplne eliminoval mrhanie zdrojmi. Na vytvorenie takéhoto vysoko efektívneho systému bolo potrebné realizovať niektoré zmeny, ktoré sa pri ich zavádzaní do používania mohli javiť ako drastické, no ich výsledný efekt bol nanajvýš uspokojivý.

Z dôvodu zvýšenia toku produktov boli vytvorené pracovné postupy, ktoré nevytvárali inžinieri za svojimi stolmi, ale ich vyvíjali samotní robotníci, ktorí najlepšie poznali proces výroby. Pre každú súčiastku bol určený štandardný časový cyklus. Taktiež bol definovaný priestor pre každú súčiastku, ktorá sa v procese vývoja používala. Pracovníci pracovali ako výmenný tím, vymieňajúc si produkty „z rúčky do rúčky“. Výmena produktov predpokladala použitie manažmentu totálnej kvality. Je to nový pohľad na plánovanie výroby produktov. Iné metódy pri plánovaní výroby predpokladajú nepodarkovosť okolo desiatich percent. Tá je ale v skutočnosti väčšia a niekedy dosahuje až dvadsať percent. Manažment totálnej kvality predpokladá nulovú nepodarkovosť. Výsledkom je výroba s nepodarkovosťou okolo dvoch percent, čo predstavuje výrazné zlepšenie. Výmena produktov taktiež predpokladá tesné načasovanie jednotlivých činností. V prípade, že dochádzalo k oneskoreniu výroby, od spolupracovníkov sa očakávalo, že si navzájom pomôžu pri nastavovaní alebo opravovaní stroja.

Táto až agresívna eliminácia plytvania mala za následok rapídne zrýchlenie toku produktov. Keďže aj pri zvýšenom tempe je nutné dodržiavať čo najvyšší stupeň kvality, došlo k zabudovaniu manažmentu totálnej kvality priamo do procesu. Takýmto spôsobom sa podarilo zvýšiť kvalitu, znížiť náklady a skrátiť čas zavedenia produktu.

Takéto zefektívnenie práce bolo zaujímavé pre ľudí, ktorí sa zaoberali metódami tvorby softvérových systémov. Preto došlo k hlbšiemu preštudovaniu princípov používaných pri *Lean Manufacturing*. Výsledkom boli výskumy, ktoré potvrdzovali, že väčšina pravidiel, ktoré sú zadefinované pre *Lean Manufacturing*, sa môže použiť aj pri vývoji softvéru.

Požiadavky

Odlahčený vývoj je odpoveďou na neustále sa meniace prostredie. Využitím manažmentu totálnej kvality a stratégie vznikol prístup schopný adaptívne sa prispôsobovať meniacim sa požiadavkám trhu.

V pôvodnom ponímaní marketingu je dôležité čo najpresnejšie predpovedať jednotlivé požiadavky. V súčasnosti sa ešte stále investuje veľa peňazí do vývoja sofistikovaných metód na čo najlepšiu predikciu budúceho vývoja.

Avšak už teraz viacero odborníkov vidí, že riešením nie je perfektná predikcia budúcnosti. Ideálnejšie by bolo, keby sme mohli redukovať našu závislosť od predpovedí do budúcnosti vytvorením produkčného systému, ktorý by eliminoval čas odpovede systému na zmenu požiadaviek. Tým by sa vlastne docielilo, že systém by bol schopný vytvoriť produkt zodpovedajúci aktuálnym požiadavkám zákazníkov. Nemuseli by sem sa už spoliehať na predpovede, ktoré by aj tak nemuseli odrážať všetky požiadavky.

Na trhu, kde stále meniaci sa technológia vyžaduje stále vylepšovanie pôvodného systému, nie je vhodné predpovedať požiadavky, ale na ne reagovať v čo najkratšom čase. Schopnosť robiť rozhodnutia najneskoršie ako to len ide dáva dobré predpoklady uplatnenia sa firmy na rýchlo sa meniacom trhu.

S takýmto prístupom by sa dalo polemizovať. Schopnosť reagovať v čo najkratšom čase na požiadavky zákazníka môže spôsobiť, že manažéri budú menej ostražití pri sledovaní vývoja trhu. Pritom sa môže stať, že trh prejde takým vývojom, ktoré iné firmy mohli predpokladať, no organizácia, ktorá nemá vytvorený monitorovací systém, nestihne na tento vývoj zareagovať. Preto by možno bolo vhodnejšie vytvoriť

konsenzus medzi predpokladaním vývoja na základe sledovania trhu a schopnosťou minimálnej doby reakcie na požiadavky. Takto by mohol vzniknúť systém, ktorý by redukoval náklady potrebné na predvídanie správania sa zákazníka, ale pritom by si zachoval svoju schopnosť zahrnúť čo najväčšie množstvo požiadaviek používateľa do výsledného softvérového systému.

Vývojári

Môžeme povedať, že Odl'ahčený vývoj je tímové úsilie [Lean2002]. Jedným z hlavných cieľov tohto prístupu je vytvoriť taký tím spolupracovníkov, ktorí sa môžu na seba navzájom spoľahnúť v kritických situáciách. Jednou z kritických situácií pri vývoji softvéru je omeškanie sa za stanoveným plánom. Pri veľkých organizáciách môže dôjsť k omeškaniu plánu iba na nejakom úseku. No aj takéto čiastkové omeškanie môže znamenať omeškanie celého projektu. V tomto prístupe sa takáto situácia rieši tak, že členovia tímu, ktorí stihli dodržať plán, prestanú pracovať na svojej práci a pomôžu tomu úseku, ktorý mešká za plánom. A až po dotiahnutí sklzu sa všetci vrátia k práci na svojich podprojektoch.

Myslím si, že takýto prístup vytvára kvalitné pracovné prostredie, kde vládne spolupatričnosť a vzájomná dôvera. Takýto tím ľudí potom môže vytvárať nové riešenia, ktoré by v tíme individualistov neboli realizovateľné. Možné riziko môže nastať vtedy, keď z dôvodu doťahovania sklzu na jednom úseku dôjde k výraznému sklzu na úseku, ktorý bol v časovom pláne. Potom je už na rozhodnutí manažérov projektu ako sa rozhodnú rozdeliť pracovné sily tak, aby nedošlo k špirálovému efektu. Toto je však už riešenie dôsledkov. Lepšie by však bolo predchádzať takýmto situáciám. Riešením

je vytvoriť viaceré pravidlá, ktoré by zabezpečili rýchly a kvalitný vývoj softvérového systému. Samozrejme, že tieto pravidlá by museli dodržiavať všetci členovia tímu.

V prvom rade je potrebné eliminovať plytvanie tým, že z procesu sa odstráni všetko, čo nepridáva hodnotu do výsledného produktu [Lean2001]. Na zistenie, ktoré činnosti znamenajú plytvanie, je potrebné použiť hodnotovú analýzu. Príkladom môžu byť dokumenty, diagramy a moduly, ktoré vznikajú pri vývoji softvéru. Aj keď tvoria jeho časť, nie sú súčasťou výsledného produktu, ktorý sa distribuuje koncovým používateľom. Táto metóda sa snaží donútiť vývojárov preskúmať všetky takéto vedľajšie produkty, ktoré sú kandidátmi na odstránenie.

V priemyselných podnikoch, kde používajú tento prístup sa snažia eliminovať všetky zásoby, pretože tie spôsobujú zvýšenie doby odozvy na zmenu. Keďže zásoby je potrebné spotrebovať, k zmene môže dôjsť až vtedy, keď sú všetky zásoby spotrebované. „Zásobami“ pri vývoji softvéru je dokumentácia, ktorá nie je časťou výsledného produktu. Samozrejme, nie sú to zásoby v pravom zmysle slova.

Zásoby v priemyselnej výrobe sa nakupujú a neskôr spotrebúvajú, pričom vzniká výsledný produkt. Pri vývoji softvéru dochádza k vytváraniu dokumentácie, no táto dokumentácia sa nespotrebuje. Preto je možné hľadať analógiu medzi zásobami a dokumentáciou iba na vyššej úrovni abstrakcie. Dokumentácia, ako inventár, by mala byť podrobená hodnotovej analýze.

Čas potrebný na vytvorenie dokumentácie k analýze požiadaviek a návrhu systému vlastne determinuje dobu trvania týchto etáp. Strata času pritom nie je spôsobená len samotným písaním dokumentácie, ale aj predĺžením z dôvodu prehliadky

dokumentu. Ak je táto etapa dosť dlhá dôjde k zmene požiadaviek, ktoré je potrebné ešte zahrnúť do dokumentácie. Z dôvodu čo najväčšieho zrýchlenia vývoja je vhodné, ak sa veľkosť dokumentácie, ktorá vzniká pri vývoji softvéru, zredukuje oproti štandardnej veľkosti dokumentácie. Najlepším riešením sa javí zvýšenie úrovne abstrakcie dokumentu. Namiesto 100 strán detailnej špecifikácie je potrebné napísať 10 strán pravidiel a návodov na usmernenie vývoja softvéru.

Zrýchlenie vývoja softvéru umožní aj redukcia práce vykonanej v jednotlivých podprocesoch hlavného procesu [Frailey2002]. Je to z toho dôvodu, že malé ale kompletne časti systému sú rýchlo navrhované a implementované. Potom v malých iteráciách sú pridávané hodnoty do výsledného softvérového produktu. Každá iterácia pritom v sebe zahŕňa celý vývojový proces od získavania požiadaviek až po akceptačné testovanie.

Aby sme mohli zredukovať množstvo práce v jednotlivých podprocesoch, je potrebné zabezpečiť vykonávanie rozhodnutí na čo najnižšej úrovni. Predpokladom je však vytvorenie takých tímov spolupracovníkov, ktorí budú schopní spolupracovať čo najefektívnejšie. Ako už bolo vyššie spomenuté, nezanedbateľným problémom pre vývojárov je dokumentácia. Zvýšenie abstrakcie dokumentácie, v ktorej budú spísané len návody na vývoj systému, uľahčí orientáciu v dokumentácii. Vždy je lepšie povedať vývojárom čo majú robiť namiesto toho, aby im bolo povedané, ako to majú robiť. Vývojári potrebujú pochopiť cieľ svojej práce a ako zapadá do celkového kontextu.

Myslím si, že takýto prístup v plnej miere podporuje kreativitu vývojárov. Vývojár už nie je len osoba, ktorá vykonáva presne stanovenú prácu, za ktorú dostane patričnú odmenu, ale je to tvorivá bytosť, ktorej nápady sa akceptujú.

Takto akceptované metódy môžu zodpovední členovia tímu prehodnotiť. Ak sa ukáže že tento nápad môže priniesť výrazné vylepšenie vývoja alebo samotného produktu, dôjde k jeho zavedeniu. Samozrejme, že každá organizácia musí mať stanovené určité hranice, ktoré nesmie realizácia nápadu presiahnuť. Zavádzanie každého nového nápadu by mohlo vytvoriť chaotický proces vývoja softvéru. Ohraničenia na úrovni vývoja softvéru môžu byť realizované rôznymi prístupmi. Príkladom takýchto prístupov môžu byť dva prístupy, ktoré definuje Odl'ahčený vývoj pre vykonávanie zmien.

Prvým prístupom je zabudovanie testov priamo do procesu vývoja. To nám umožňuje sa uistiť, či zmeny vykonané v kóde neovplyvnia výslednú funkčnosť. Najlepší prístup je taký, že sa najskôr napíšu testy a až potom sa pristúpi k písaniu samotného kódu.

Druhým prístupom umožňujúcim vykonávať zmeny aj neskoro vo vývoji je refaktoring alebo vylepšenie návrhu existujúceho softvéru kontrolovaným a rýchlym spôsobom. Ak je refaktoring časťou procesu, redukujú sa tým špekulácie typu, aké časti softvéru budú potrebné v budúcnosti. To znamená, že v prípade zmeny požiadaviek, vývojári upravia už existujúce zdrojové kódy a nebudú musieť vytvárať nové moduly. Na to, aby použitie refaktoringu bolo úspešné a hlavne efektívne, musí si každá organizácia vytvoriť vlastnú metodológiu, ktorá bude stanovovať postupy nielen pri samotnej úprave zdrojového kódu, ale aj postupy pri jeho prvotnom písaní tak, aby sa v budúcnosti mohol použiť refaktoring.

Manažéri projektu

Projektoví manažéri sú trénovaní na manažovanie určitej časti projektu. Takto sa môže stať, že pri snahe o zvýšenie

efektívnosti procesu vývoja softvéru dôjde k lokálnej optimalizácii podprocesov. Avšak optimalizácia podprocesov nemusí implicitne viesť k optimalizácii procesu ako celku. Výsledkom môže byť ešte horší proces vývoja softvéru, ako bol pred vykonanou optimalizáciou.

Druhým výrazným rizikom pre manažerov môže byť odlišnosť prostredí, pre ktoré sa softvérové systémy vyvíjajú. Softvérové systémy môžeme rozdeliť podľa miery zmeny okolia, v ktorom systém pracuje, do dvoch skupín. Prvou skupinou sú systémy, kde je možné na začiatku stanoviť požiadavky na výsledný produkt a tieto požiadavky sa nebudú meniť počas celej doby vývoja.

Druhú skupinu tvoria systémy, ktoré sa nachádzajú v takom dynamicky sa meniacom prostredí, že počas vývoja aplikácie je potrebné modifikovať alebo pridávať požiadavky. V takýchto systémoch koncentrácia sa na to, čo bolo zamýšľané na začiatku projektu, má pre používateľa malú hodnotu, pretože jeho okolie sa sústavne mení. V skutočnosti to len paralyzuje proces vývoja. Taktiež to nepridáva veľkú hodnotu do konečného systému, ktorý už bude vlastne „zastaraný“ už v čase jeho dodania zákazníkovi. Koncentrácia na veci, ktoré nie sú platné, znamená plytvanie časom a zdrojmi.

Riešením je taký manažment, ktorý pripúšťa zmenu požiadaviek v čase, pričom vždy má na zreteli projekt ako celok. Preto je potrebné zabezpečiť, aby bol projekt riadený nielen časom, ale aj spätnou väzbou. Pomocou spätnej väzby budú mať manažéri ucelený obraz o stave projektu. To si však vyžaduje znalosť problematiky doménovej oblasti. Prístup odl'ahčeného vývoja predpokladá, že splnením všetkých týchto predpokladov je možné dosiahnuť maximálnu optimalizáciu celého projektu.

Je však otázne, do akej miery pripustiť zmenu požiadaviek na výsledný softvérový systém. Ak by sme pripustili, že každá zmena požiadavky zákazníka sa musí odzrkadliť vo výslednom produkte, mohli by sme dospieť do stavu, kedy by došlo k výraznému oneskоровaniu za stanoveným plánom. Vytvárať systém, ktorý by reflektoval všetky požiadavky zákazníka, ale ktorý by bol dodaný o niekoľko „rokov“ neskôr, neprinesie žiadny úžitok ani zákazníkovi a ani organizácii, ktorá systém vyvíjala. Preto hlavnou prioritou je vytvoriť konštruktívny dialóg medzi manažérmi a vývojármi na jednej strane a zákazníkmi na strane druhej.

Zákazníci

Výsledkom štúdie od Standish Group [SG1994], ktorá analyzovala neúspechy implementácie softvérových systémov bolo, že väčšina projektov stroskotala na tom, že v projekte neboli zahrnuté všetky požiadavky, alebo boli nesprávne realizované.

V súčasných metódach vývoja softvéru je zákazník nútený stanoviť požiadavky v počiatočných fázach. Potom mu už nie je dovolené meniť názor aj vtedy, keď nemá predstavu o svojom systéme alebo vtedy, keď nevie ako sa bude v budúcnosti vyvíjať obchodná situácia. Inštinktívnou reakciou každého zákazníka je potom oddŕaľovať rozhodnutia na čo najneskoršiu dobu. To má za následok aj oneskоровanie projektu.

Efektívnym riešením tejto situácie je umožniť zákazníkovi podieľať sa na vývoji systému v každej jeho fáze. Ak budeme akceptovať, že požiadavky sa budú môcť meniť kedykoľvek v čase, musíme začať s podstatnými požiadavkami, ktoré sa nebudú meniť počas celého procesu vývoja. Tieto požiadavky musia byť stanovené na začiatku vývoja softvéru. Potom sa

môže systém lepšie prispôsobovať požiadavkám, ktoré vzniknú v neskoršej dobe, ale súčasne bude zachovávať časové rozpätie odovzdania výsledného softvérového systému.

Odl'ahčený vývoj a iné agilné prístupy

Agilné prístupy vo všeobecnosti znamenajú nový pohľad na vývoj softvéru. Odl'ahčený vývoj je oproti iným agilným prístupom odlišný v dvoch aspektoch.

V prvom rade je najviac orientovaný na stratégiu uplatňovanú vo firme. Je to hlavne z historických dôvodov, pretože celý tento prístup vychádza zo skúseností získaných v priemyselnej výrobe, ktorá bola vždy orientovaná na stratégiu.

Druhým aspektom je orientácia na tímovú spoluprácu, ktorá je voči iným metódam najviac prepracovaná. Aj keď všetky agilné prístupy vychádzajú z hlavného princípu, ktorým je komunikácia v tíme, táto metóda sa môže popýšiť najväčšou orientáciou na tímovú spoluprácu.

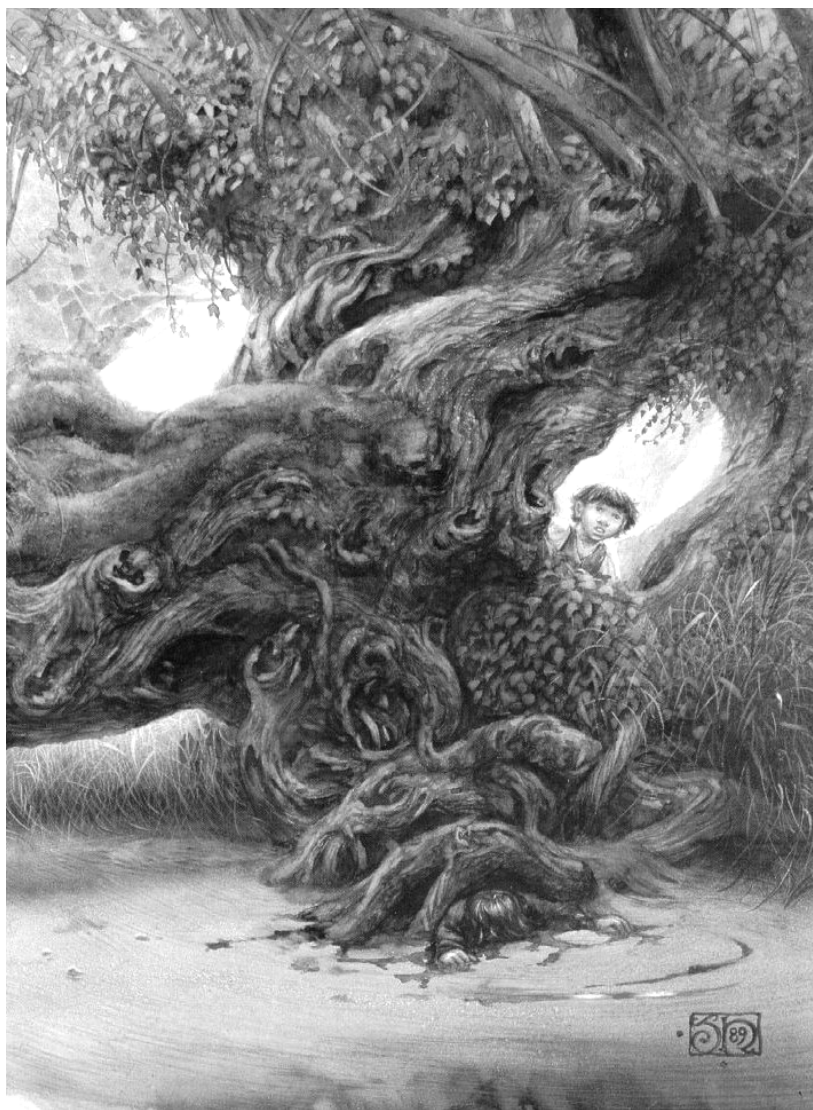
Môžeme povedať, že tento prístup predstavuje akýsi prechod medzi klasickými a agilnými prístupmi. Klasickým prístupom sa podobá hlavne v oblasti strategického rozhodovania počas vývoja softvéru. Agilným prístupom sa približuje v dôraze na tímovú spoluprácu, redukciu nadbytočných aktivít a požiadavky zákazníka.

Záver

Nový závan do skostnatených prístupov k vývoju softvérových systémov prispieva k napredovaniu softvérového inžinierstva. Tento prístup by mohol priniesť zvýšenie kvality a skrátenie času vývoja softvéru. Jeho výraznou črtou je orientácia na

požiadavky zákazníka. Vlastne môžeme povedať, že celý vývoj sa podriaďuje zákazníkovi.

Použitie takéhoto prístupu na vývoj softvérového systému prináša niektoré výhody, ale aj úskalia. Preto až čas ukáže ako obstojí v dynamicky sa rozvíjajúcom odvetví akým je softvérové inžinierstvo. Aj keď v oblasti priemyslu tento prístup zaznamenal výrazný úspech, nemusí to ešte znamenať, že rovnaký úspech zaznamená aj v tejto oblasti. Je to hlavne z toho dôvodu, že oblasť priemyslu už neprechádza takým dynamickým vývojom, aký momentálne zaznamenáva softvérové inžinierstvo.



Adaptívny vývoj softvéru

Bc. Martin Geleneky

Prostredia, v ktorých a pre ktoré je v súčasnosti vyvíjaný softvér „volajú“ po adaptácii. Vyžadujú si adaptívny model vývoja, adaptívny model manažmentu a adaptívny konceptuálny model, ktorým sa venuje nasledujúca esej.

Adaptívny vývoj softvéru (Adaptive Software Development, ASD) kladie dôraz na spoluprácu, rozmyšľanie a učenie sa. ASD vyžaduje zmenu v prístupe k tvorbe softvéru zo strany zákazníka aj zo strany manažéra. Všetko je založené na prispôbení sa (adaptácii) v nespoznanom, nie na optimalizácii spoznaného.

„Nechod' sa pýtať elfov na radu, pretože ti povedia áno aj nie.“

V súčasnosti sa stretávame s rastúcim záujmom o použitie nových postupov a prístupov pri tvorbe softvérových produktov, ktoré za stredobod *procesu vývoja* považujú človeka, nie produkt. Tradičné vývojové stratégie zahrňujú presné plánovanie a kontrolu všetkých procesov. Finančný softvérový trh, veľmi bohatý na rôzne riziká, ktorý si vyžaduje rýchlu produkciu rovnakej nezmenenej a vysokej kvality, však potrebuje inú stratégiu. Tradičné metódy zostávajú zachované, avšak zmení sa priorita a postavenie jednotlivých aspektov.

Adaptívny vývoj softvéru (ASD) je prístup k tvorbe softvérových systémov používaný vo vývojárskych tímoch, kde konkurencia (súťaživosť) vytvára extrémny tlak na rýchlosť aj na zmenu procesu dodania softvérového produktu. Aj keď sa mnoho organizácií snaží zredukovať životný cyklus projektu, rozdiel medzi mnohými týmito projektami a skutočne extrémnymi projektami je analogický rozdielu v horolezectve, rozdielu liezť na Mount Cook a Mount Everest. ASD kladie dôraz skôr na vytvorenie vysoko hodnotného výsledku založeného na rýchlom prispôbení sa vonkajším aj

vnútorným udalostiam a vplyvom, ako na zlepšenie samotného procesu vývoja.

Prečo Adaptívny vývoj softvéru?

V menej komplexnom svete, s pomalším napredovaním ako je dnešné, pozostatky z „Newtonskej éry“ fungujú veľmi dobre. Optimalizácia prostredníctvom prísnych proces zlepšujúcich prístupov, manažment projektu založený na detailných úlohách a hierarchický manažment boli enormne úspešné. V dnešnej dobe „štítom“ optimalizácie môže byť – optimalizované vytvorenie „spoznaného“ (*known*) na vysokej úrovni efektívnosti.

Ale čo s „nespoznaným“ (*unknown*)? Čo ak nevieme presne aký produkt si trh vyžaduje? Čo ak produkt od konkurencie úplne pokazí naše starostlivo vytvorené plány? Čo ak nové technológie spravia z nášho snaženia zastaralú vec, prežitok? Čo ak máme len akúsi hmľistú predstavu o našom smerovaní?

V týchto situáciách má optimalizácia veľmi limitované možnosti. Tvorba softvéru pre rýchlo sa meniaci trh si vyžaduje odlišný prístup. V tomto prostredí je „*adaptácia dôležitejšia ako optimalizácia*“.

Kým prístupy typu RAD (*Rapid Application Development*) umožňovali organizáciám urýchliť vývoj v malých projektoch a prístupy založené na evolúcii sa používali na manažment zmien, kombinovanie rýchlosti a zmien zapríčinené zväčšovaním sa projektov, komplexnosť a distribuované tímy si vyžadujú viac. ASD vychádza z úspechov oboch prístupov ako RAD, tak aj evolučných prístupov, ale zahŕňa v sebe pokročilé prístupy spolupráce a riadenia projektu cielené na extrémne prostredia, ako je

napríklad Internet (Internet = vysoká rýchlosť, rýchle zmeny, nestálosť => udržať cyklus vývoja čo najkratší a podporovať iteráciu).

ASD je podstatne odlišný od tradičných prístupov zlepšenia softvérových procesov a má iný konceptuálny základ. V posledných desiatich rokoch štúdie komplexnosti adaptívnych systémov na poli biológie, ekológie a fyziky sa začali vymykať zo svojho rámca a dospeli až k *manažmentu organizácií* [IA2001]. Niektoré z týchto štúdií pomáhajú vysvetliť, prečo „high-tech“ organizácie uspeli prelomením historických vzorov správania sa a dostali sa na čelo, a poskytujú pohľad dovnútra fungovania organizácie budúcnosti.

Zlepšenie procesu je zakorenené v *deterministickosti* a „Newtonskom“ pohľade na svet, kde kontinuálna optimalizácia a statické riadenie sú základom úspechu. Tieto techniky optimalizácie sa dajú úspešne použiť v miernom prostredí. Adaptívny vývoj je zakorenený v nelineárnom, nedeterministickom pohľade, na hranici chaosu, kde plánovanie je slabé a riadenie nemožné – je to svet adaptácie, nie optimalizácie. Ide o výsledkami a nie procesom riadený prístup.

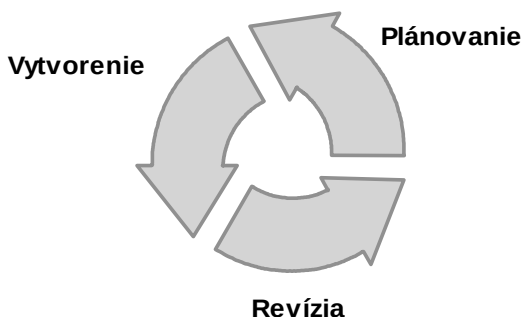
ASD prístup sa skladá z troch modelov [Rokham2001]:

Adaptívny model vývoja (Adaptive Development Model) je založený na adaptívnom životnom cykle podobnom evolučnému alebo špirálovému modelu životného cyklu.

Predstavovaný ASD model sa časom vyvíjal. Roky bol dominantným prístupom vývoja softvéru vodopádový model. Linearita vývojovej fázy a predpoklad výsledku predstavovali základné črty. Na vonkajšie vplyvy a spätné väzby od zúčastnených sa nebral skoro vôbec ohľad. Vznikali produkty,

ktoré boli naplánované, ale možno vôbec nespĺnili to čo ponúkali, vzhľadom na očakávania zákazníka.

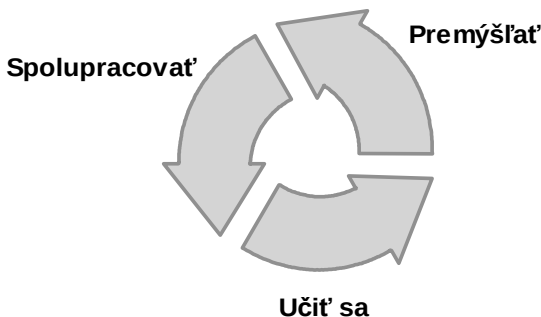
Iteratívny vývojový model (Obr.1) vznikol okolo roku 1985. V tomto modeli sa plánuje aj to čo nazývame *revízia* výsledkov. Vidíme tu v popredí presné plánovanie procesov ako kratšie časové úseky nejakého vývojového cyklu, teda stále sú zreteľné deterministické formulácie.



Obr. 1: Iteratívny vývojový cyklus

ASD sa nezakladá na tomto deterministickom spôsobe, ale vychádza z výskytu neustálych zmien, teda všetko bude aj zvážené nielen naplánované. To znamená, že cieľov bude veľmi veľa a nie všetky musia byť konečné. Spolupráca bude cielená ako hnací mechanizmus kreativity. Prostredníctvom spolupráce sa získajú potrebné praktické skúsenosti, ktoré budú potom použité v ďalších vývojových cykloch. Pre vývoj softvéru „*Premýšľať*“ znamená nielen zvážiť plánovanie celých projektov, ale aj plánovanie nových cyklov. „*Spolupráca*“ ručí za implementáciu softvéru, v ktorej má každý člen tímu dôležitú a nezastupiteľnú úlohu, a „*Učenie sa*“ za kvalitatívnu skúšku produktov po jednom vývojovom cykle, ktorá bude

realizovaná prostredníctvom rôznych kontrol (Obr.2: Adaptívny vývojový cyklus). Na všetko spolu sa však môžeme pozerat' aj ako na nejakú učiacu sa slučku, pretože každá z týchto častí má nejaký učiaci efekt.



Obr. 2: Adaptívny vývojový cyklus

Adaptívny model manažmentu (Adaptive Management Model) zdôrazňuje vedúce postavenie, spoluprácu a zodpovednosť a nie príkazy a kontroly.

- V dnešnom extrémne rýchlom a meniacom sa prostredí je rovnováha výnimkou a zmena pravidlom.
- Adaptácia závisí skôr na vedúcom postavení a spolupráci ako na príkazoch a kontrole.

Prvý bod hovorí o tom, že nikdy nedosiahnete stav, v ktorom budete spokojní, resp. v ktorom bude zákazník spokojný, s vytvoreným softvérovým produktom vzhľadom k zmenám, ktoré sa udiali počas vývoja. Dal by sa tu použiť aj trochu pozmenený Murphyho zákon v tvare: „Čo sa môže zmeniť sa určite zmení!“

Druhý bod hovorí o tom, že striktné dodržiavanie, vydávanie a kontrola rôznych príkazov a nariadení, nezdravá autoritatívnosť nie sú zdrojom správnej adaptácie, ba práve naopak, môžu od nej odvádzať pozornosť. Všetko závisí od manažéra a ostatných vedúcich osôb, ktoré riadia daný proces, a od toho ako sa s problémom správnej autority vysporiadajú.

Adaptívny konceptuálny model (Adaptive Conceptual Model) je odvodený z novej vedeckej práce venujúcej sa teórii Komplexných adaptívnych systémov (Complex Adaptive Systems theory).

Komplexné adaptívne systémy (Complex Adaptive Systems) je množina nezávislých agentov (spoločností, organizácií),

- ktorí spolupracujú na vytvorení ekosystému.
- ktorých spolupráca je definovaná výmenou informácií.
- ktorých individuálne akcie sú založené na interných pravidlách.
- ktorých samoorganizácia je nelineárna, aby dosiahli naliehavé výsledky.
- ktoré majú preukázateľné charakteristiky poriadku aj chaosu.
- ktoré sa s časom vyvíjajú.

Čo znamená „spolupracovať“?

Ako môže tvorivá spolupráca poháňať? Dôležité je predovšetkým to, aby došlo k výmene potrebného množstva informácií. Aj v čase kávičkovej prestávky alebo zábavy na chodbe môžu vzniknúť neuveriteľné tvorivé nápady. Každý člen projektu by mal vysloviť svoj názor, aby diskusia mala

pozitívny vplyv na kreativitu. Malé rozdiely v hierarchii môžu takisto napomôcť pozitívnej pracovnej morálke. Nadriadený v ASD modely by mal nechať svojim podriadeným čo najväčší priestor na tvorivú spoluprácu a vývoj. Nesmie mať ku všetkému pripomienky a nesmie do detailov kontrolovať všetky postupy vývoja. Na druhej strane stále musí byť jasný, kto je tu šéfom, a jeho nariadenia presne dodržiavať.

Avšak nesmieme zabúdať, že trochu rešpektu pred nadriadeným a takisto trochu tlaku z jeho strany je tiež spôsob, akým možno spoluprácu a kreativitu podporiť. “Zo strachu” pred šéfom sa budú všetci pracovníci snažiť vykonávať svoju prácu ku spokojnosti všetkých. Ako pri ostatných fázach adaptívneho vývojového cyklu (Obr.2 Adaptívny vývojový cyklus), tak aj pri spolupráci, musí predovšetkým vedúci projektu alebo manažér zvážiť, aký spôsob je v danej situácii najvhodnejší. Tento spôsob by mal dať za vznik atmosfére na pokraji chaosu. Malo by vzniknúť pracovné prostredie, v ktorom neplynie všetko deterministicky, ale v ktorom vznikajú stále nové a nové nápady. To však ani zďaleka neznamená, že všade panuje zmätok. Stále existujú určité hranice, ktoré treba dodržiavať. Práve takéto pracovné prostredie je zo skúseností optimálne pre vznik tvorivých nápadov.

Adaptívny zákazník

Vyššie opísaný spôsob adaptívnej spolupráce si vyžaduje aj odlišný typ vzťahu so zákazníkom, ako ten ktorý je často používaný hlavne keď vývoj vykonáva nezávislá firma. Keď vývoj softvéru zveríme nezávislej firme, väčšina zo zákazníkov bude samozrejme uprednostňovať pevne stanovenú cenu za produkt. Zistíme od vývojárov čo požadujú, pýtame sa na podmienky, potom tieto podmienky akceptujeme a všetka

zodpovednosť je potom už len na organizácii, ktorá softvér vyvíja.

Pevná cena za produkt si však vyžaduje stabilné požiadavky a prediktívny proces. Adaptívny proces a nestále požiadavky však implikujú, že nemôžeme pracovať s obvyklým pojmom pevnej ceny. Pokus o stanovenie pevnej ceny adaptívneho procesu skončí „bolestivým výbuchom“. Nepříjemný dôsledok tohto výbuchu je, že zákazníci sú rovnako zranení ako samotná vývojová organizácia. Predsa zákazník nebude chcieť nejaký softvér, kým si to biznis nevyžaduje? Ak ho potom nedostane, biznis trpí! Takže aj keď spoločnosti za vývoj nezaplatí nič, stále stráca. V skutočnosti však stratí viac než by zaplatil za softvér (zákazník sa pýta: prečo by som mal platiť za softvér, keď jeho obchodná cena bola nižšia?).

Takže v prípade, že nemožno použiť prediktívny proces s pevne stanovenou cenou, tu hrozí nebezpečenstvo nepodpísania kontraktu z oboch strán. To znamená, že zákazník sa musí správať trochu inak.

V adaptívnom procese má zákazník lepšiu kontrolu nad procesom vývoja softvéru. V každej iterácii dostane možnosť vývoj skontrolovať a prípadne zmeniť smer vývoja. Toto však vedie k bližšiemu vzťahu so softvérovými vývojármi, ku skutočnému obchodnému partnerstvu medzi zákazníkom a vývojárskou firmou. Tento spôsob spolupráce nevyhovuje každému zákazníkovi, ani každej firme vyvíjajúcej softvér, ale je nevyhnutný pre správne fungovanie adaptívneho procesu.

Kľúčovým prínosom pre zákazníka je potom vývoj softvéru, ktorý sa uskutočňuje presne podľa jeho pokynov a požiadaviek. A použiteľný systém sa potom môže začať veľmi rýchlo vyvíjať. Zákazníci potom môžu meniť jeho správanie sa

v závislosti od zmien v ich biznise a takisto v závislosti od toho, čo sa naučili pri reálnej prevádzke systému.

Čo znamená „učenie sa“?

Položme si teraz otázku: Čo znamená učenie sa v adaptívnom vývojovom cykle, poprípade ako sa uskutočňuje a ako nás bude poháňať? Tu môžeme teraz rozlišovať medzi učením sa pri spolupráci a učením sa pri samostatnej práci bez pomoci kolegov. V našom prípade je učenie sa vec postoja a nie schopnosť. Musíme byť pripravení, musíme mať nazbierané potrebné skúsenosti a tie musíme potom spracúvať. Výrok: „Človek sa učí na vlastných chybách.“, sa zdá ako samozrejмый, avšak pravdivý je len v prípade, keď sme pripravení urobiť chyby odhaľovať a akceptovať ich, teda zmieriť sa s nimi a skutočne sa z nich poučiť.

Pri procese učenia sa v ASD modely je dôležité, aby sme hneď od začiatku projekt chápali ako celok a cítili sa ako podstatná súčasť celého vývojového tímu. Týmto pocitom si zvyšujeme nielen podvedomie ako člena tímu, ale aj vnímavosť pre skúsenosti a kladný postoj k prijatiu chýb, na ktoré sme boli upozornení od ostatných, čo má za následok efektívne učenie sa. Kto zapochybuje sám o sebe alebo o projekte, stáva sa menej vnímavým a hrozí nebezpečenstvo neúspechu projektu. Rovnako to platí o pracovníkoch, ktorí sú vystavovaní prílišnému tlaku a stresu. Je úlohou vedúceho tímu a manažéra, aby nevytvárali zbytočne veľký tlak a stres, a aby nabádali ľudí, aby si verili. Každý by mal zároveň vyhľadávať kontakt s kolegom a baviť sa s ním nielen o veciach ohľadne projektu, ale aj o ostatných jeho problémoch alebo radoostiach, a tak s ním vlastne spolucítiť, pomáhať mu a tešiť sa s ním. Pasívna účasť je v ASD modely nežiadúca a odporuje myšlienke spolupráce a spoločného učenia sa.

Učenie sa potom neznamená len nutne získanie nových skúseností, ale aj zlepšenie už existujúcich.

Prečo zlyhá dobrý „manažér“?

Tradičné manažérske metódy používajú katalóg riešení. Ak sa vyskytne problém nájdú riešenie v katalógu. V ASD modely je potrebné postupovať šikovnejšie a okamžite zistiť, že niet už čo optimalizovať, a že táto metóda nevedie k nájdeniu optimálneho riešenia. Je potrebné nájsť novú cestu, ktorej zodpovedá tvorivé spontánne riešenie. Manažéri v ASD modely musia mať teda schopnosť tvorivého myslenia. Presne toto je však problém pri tradične vyškolených manažéroch. Často sa spoliehajú na zdanlivo osvedčené metódy riešenia problému, namiesto toho, aby sa pokúsili spontánne prispôbiť daným skutočnostiam. Veľmi nebezpečné môžu byť takzvané „deštruktívne technológie“.

Vysvetlime si tento pojem na nasledujúcom príklade.

Firma Plextor vytvorila veľmi drahé SCSI rozhranie. Vtedy novo zavedené rozhranie IDE bolo firmou Plextor označené ako nestabilné a nespoľahlivé. Vtedy sa vyrábala len unifikovaná disketová jednotka, čo nebolo s IDE rozhraním možné. Cenovo výhodné IDE rozhranie sa však aj napriek tomu stalo akýmsi kvázi štandardom a stále sa rozvíjalo ďalej. Plextor tak stál pred rozhodnutím, či ďalej optimalizovať a zlepšovať kvalitu SCSI rozhrania, alebo predsa len zobrať do úvahy aj vývoj IDE rozhrania. IDE rozhranie tu predstavovalo „deštruktívnu technológiu“. Nová technológia, ktorá stojí proti firemnej politike a ktorú firma neprevezme, môže viesť ku bankrotu. Medzičasom existuje viac IDE ako SCSI produktov od firmy Plextor. Aj keby sa mnoho firiem, aj keď nedobrovoľne preorientovalo na váš koncept, stále budete dosť

často tlačení počítačovým odvetvím. Koniec koncov vždy o úspechu nejakého výrobku rozhodne zákazník.

Záver

Pri ASD modely sú stredobodom pozornosti ľudia a nie vývoj produktu. Stavia sa na tom, že dobrá spolupráca tímov vedie k najlepšiemu produktu. Manažéri sa snažia, aby vývojové prostredie a pracovné postupy čo najviac prispôbili projektovým nárokom a ideám ASD modelu. Vývojový tím sa potom stará o čo najlepšiu adaptáciu produktu vzhľadom na požiadavky. ASD vychádza z toho, že všeobecné vzťahy medzi členmi tímu pozitívne stimulujú inováciu a kreativitu. Adaptácia tímov na zmeny, ktoré sa v projekte vyskytnú, sa potom uskutočňuje prostredníctvom komunikácie, spolupráce a súťaživosti napríklad pri spoločných diskusiách členov tímu.



Vývoj riadený pomocou črt

Bc. Marek Kubran

Znížiť riziko prekročenia časového plánu a rozpočtu na minimum, motivovať vývojárov a pracovné tímy, vytvárať realizovateľné plány, zapojiť používateľa do procesu vývoja softvérového systému, vychádzať v ústrety novým a meniacim sa požiadavkám zákazníka. To je sen každého projektového manažéra. Je to iba naozaj sen a niečo nedosiahnuteľné, alebo existujú prístupy vývoja softvéru, ktoré dokážu vyhovieť uvedeným kritériám?

"Biela látka sa dá ofarbiť. Biela stránka sa dá popísať a biele svetlo sa dá rozlámať. A potom už nie je biele. A ten, kto rozláme vec, aby zistil, čo je zač, opustil cestu múdrosti."

Oblasť ekonómie v mnohom zasahuje do softvérového inžinierstva, najčastejšie v podobe tzv. e-biznis riešení. Procesy prebiehajúce v trhovej ekonomike podliehajú neustálym zmenám. Na podporné softvérové nástroje a programy sa tak zo strany zákazníkov kladú nové a nové požiadavky. Aby ich bolo možné splniť, je potrebné hľadať a využívať nové spôsoby v procese vývoja softvéru.

V nasledovných častiach je opísaný prístup k vývoju softvéru pomocou črt, ktorá umožňuje presné plánovanie, hodnotenie výsledkov, preukazovanie funkčných výsledkov a meranie progresu s vysokou presnosťou. Autor článku taktiež uvažuje nad potrebou postupu pri agilných prístupoch k vývoju softvéru a opisuje, ako by mal takýto postup vyzeráť.

Prispôsobovanie sa stále kratším a kratším cyklom

Napriek pokroku v technike tvorby softvéru, sa v praxi môžeme ešte stále stretnúť s projektmi, ktoré trvajú niekoľko rokov a používajú tradičný vodopádový model: od funkčnej špecifikácie (prípady použitia) cez návrh, implementáciu, testovanie až po zavedenie a údržbu systému [Coad1999]. Hoci v tomto prístupe došlo k niekoľkým zmenám, napríklad prototypovanie, aj tak sa v týchto softvérových projektoch mnohokrát prekročí rozpočet, predĺžia termíny, nedodrží plán a výsledný produkt nezodpovedá súčasným požiadavkám, lež potrebám spred niekoľkých rokov.

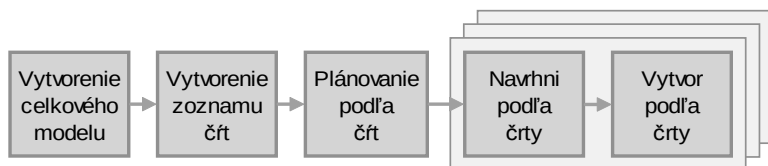
Neustále sa zväčšujúci pokrok v technologickom napredovaní odsudzuje v mnohých prípadoch projekty dlhšie ako dva roky na neúspech.

Čoraz častejšie sa presadzujú projekty s celkovým časovým plánom rozvrhnutým na 90, 120, alebo 180 dní, prípadne 9, 12, alebo 18 mesiacov. V niektorých prípadoch možno považovať projekty trvajúce viac ako pol roka za rizikové – napr. v trhovej ekonomike konkurencia významne ohrozuje úspech projektu. Dobrým príkladom z nášho každodenného života je zavádzanie nových služieb prevádzkovateľov mobilných sietí. V tomto prípade je lepší prístup iterovania a prispôsobovania sa zmenám postupne počas vývoja systému.

Jedným zo štandardov pre projekty s krátkodobými cyklami je prístup k vývoju softvéru pomocou črt, takzvaný *Feature-Driven Development* (FDD). FDD je modelom riadený postup vývoja softvéru používajúci krátke cykly. Ide o inkrementálny (prírastkový) vývoj, ktorý začína s celkovým modelom systému a zoznamom črt, za ktorým nasleduje séria

dvojtyždňových cyklov návrhu a implementácie jednotlivých čŕt (pozri obr. 3).

FDD možno schematicky zobrazit' ako nasledovnú postupnosť procesov:



Obr. 3: Postupnosť aktivít v rámci FDD [Coad2000].

FDD dosahuje v praxi významný úspech, a to u všetkých strán zúčastnených na projekte.

Vývojári sú naklonení FDD, pretože pomocou tohoto prístupu dosahujú každé dva týždne niečo nové, hotové a fungujúce. Takéto pravidelné výsledky pôsobia motivačne a prinášajú uspokojenie z práce, pretože častá možnosť vyslovenia vety „som hotový“ každého iste povzbudí.

Manažéri uprednostňujú FDD preto, lebo presne vedľa, čo naplánovať a ako určiť významné a zmysluplné kontrolné body. FDD znižuje riziko, ktoré prináša projekt dodávajúci časté, konkrétne a funkčné výsledky. Pomocou FDD projektanti získavajú reálne čísla vyjadrené v percentách o napredovaní postupu vývoja, t.j. presne vedľa určiť, koľko percent z vývoja softvéru je hotových. Prezenterovanie týchto konkrétnych čísel poskytuje zákazníkovi a vrcholovému manažmentu presný obraz o tom, ako ďaleko sa projekt nachádza, čo prirodzene znižuje riziko prekročenia rozpočtu.

Zákazníci inklinujú k FDD tiež, pretože majú pred sebou jasný a „čitateľný“ plán s míľnikmi (kontrolnými bodmi). Už

počas vývoja vidia včasné a časté výsledky, ktorým rozumejú. V neposlednom rade vedia v ktoromkoľvek časovom okamihu presne zistiť, ako je projekt ďaleko. Navyše zákazník podobne, ako vo všetkých agilných prístupoch vývoja softvéru, participuje na riešení, a nie je iba v roli pasívneho zákazníka, ktorý čaká na výsledok. Zákazník je teda spolutvorcom softvéru.

Črty

Črty sú malé ciele, resp. výsledky, ktoré sú „užitočné v očiach zákazníka“. Väčšina iteratívnych postupov predstavuje niečo krátke a prospešné pre zákazníka. Iterácia ako napr. „vytvor účtovný podsystém“ spotrebuje veľké množstvo času, a na druhej strane iterácia ako „vytvor rozhranie databázového modulu“ nie je (aspoň priamo) cenná pre zákazníka, i keď jej vývoj môže zabráť aj niekoľko človekomesiacov.

Navyše dlhé iterácie so zložitým výsledkom prinášajú mnoho ťažkostí. Je v nich obtiažne sledovať postup a určiť, čo je v danom okamihu hotové a čo nie. Zároveň je obtiažne zapojiť do procesu zákazníka, pretože k dispozícii nie sú priebežné výsledky, ktoré by mu bolo možné ukázať, prípadne dať zhodnotiť.

Naopak, malá črta, ako napr. „priradiť faktúre jedinečné číslo“ je aj krátka aj užitočná. Zákazník presne vie, čo táto črta znamená, môže jej priradiť prioritu, môže o nej diskutovať, či je potrebná a či vôbec splňa jeho požiadavky.

Črta teda predstavuje malý, ale dôležitý stavebný prvok v plánovaní, vykazovaní a sledovaní vývoja softvérových systémov. Je zrozumiteľným a merateľným prírastkom v tvorbe systému, ktorý je možné navrhnuť a implementovať v krátkom časovom horizonte.

Podobne, ako ostatné vývojové postupy, FDD predpisuje postupnosť krokov, ktoré sú však jedinečné v nasledovnom [Coad1999]:

- používa veľmi malé funkčné bloky nazývané črty, ktoré sú pre zákazníka významné a užitočné. Črty dovoľujú používateľovi v krátkosti opísať, čo naozaj potrebuje, namiesto toho, aby boli jeho predstavy obmedzené na jednoduché „používateľ robí to, systém robí to“,
- tieto malé bloky sa združujú do príbuzných skupín, tzv. množín črt (pričom sa rieši dilemma, pre ktorú úroveň treba napísať prípady použitia),
- vývojári vytvárajú každé dva týždne funkčné výsledky,
- podporuje kontrolnú činnosť pri návrhu, implementácii aj riadení, čo uľahčuje akceptáciu a prevádzkovanie systému,
- poskytuje detailné plánovanie vrátane naplánovania kontrolných bodov,
- podporuje súbežný vývoj v rámci každého prírastku „navrhni podľa črty, vytvor podľa črty“, a
- sleduje a vykazuje progres s veľmi veľkou presnosťou, a to detailne pre vývojárov a riadiacich pracovníkov, ako aj sumárne pre vrcholový manažment a zákazníka.

Vytváranie celkového modelu systému

Celkový model systému v podstate predstavuje objektový model sveta, ktorý vytvárajú používatelia z aplikačnej oblasti a vývojári spolu pod vedením vedúceho architekta. Už počas vytvárania celkového modelu si vývojári počas diskusií

s používateľmi vytvárajú neformálny zoznam črt a taktiež z dokumentov, s ktorými pracujú.

Detailný zoznam črt sa vytvára až po vytvorení celkového objektového modelu. Niektoré črty vznikajú transformáciou metód v modeli [Coad1999], a iné zas tým, že vývojári uvažujú aj nad ďalšími črtami, ktoré by mohli lepšie opísať požiadavky zákazníka, prípadne splniť jeho budúce potreby. Napríklad pre črtu „vypočítaj celkový obrat z predaja“ bude zákazník pravdepodobne potrebovať aj vypočítať daň, a pokiaľ ju explicitne neuviedol, môže ju vývojár zaradiť do zoznamu. Pre každú dodatočnú črtu sa doplní príslušná metóda v objektovom modeli.

Vytváranie objektového modelu a neformálneho zoznamu črt pred samotnou fázou vytvorenia podrobného zoznamu črt prináša niekoľko výhod:

- zbližuje používateľov z aplikačnej oblasti tým, že musia komunikovať jeden s druhým, aby zostavili spoločný model oblasti, pre ktorú sa softvérový systém vytvára,
- vývojári lepšie porozumejú aplikačnej oblasti, lepšie pochopia vzájomné súvislosti, a to aj vtedy, ak už v tejto oblasti systém vytvárali,
- podporuje väčšiu tvorivosť a vylepšovanie systému (pri vizuálnych modeloch, akými objektové modely nesporne sú, sa viac zapája priestorové zmýšľanie a tvorivosť, než pri obmedzení sa do jazykového, matematického alebo logického zmýšľania pri tradičnej špecifikácii) a
- preferuje skúmanie systému spôsobom „čo by sa mohlo spraviť, čo by malo byť vykonané“

a objavovanie nových črt pred uzavretím sa do pevných hraníc systému („používateľ robí to, systém robí to“ typicky v prípadoch použitia), resp. pasívnym spisovaním požiadaviek pre „iba ďalší“ systém.

Prečo vôbec postup?

Väčšina programátorov využívajúcich agilné metódy sa domnieva, že postupy a dokumentácia sú zbytočné. Na druhej strane, mnohokrát sa stáva, že manažéri a vývojárske tímy sa „ponoria“ do vykonávania jednotlivých krokov postupu až tak, že zabudnú, že sú platení za výsledky a nie za vykonávanie postupu.

Postupy pre postupy spôsobujú viac škody ako úžitku. Stostranový postup demoralizuje členov tímu až do takej miery, že dobrovoľne „vypnú“ rozum a jednoducho nasledujú kroky postupu. Už samotné vytvorenie postupu spotrebuje veľa času, ktorý by sa ináč dal využiť na vývoj softvéru.

Asi pred desiatimi rokmi istý projektant napísal 110 stranový postup pre jeden veľký tím. Nezáležalo na tom, ako sa veľmi zastával postupu a obhajoval v ňom každé slovo ako veľkú myšlienku, všetci členovia tímu si vždy pozreli iba štvorstranové zhrnutie na konci a zvyšok, ktorý autor považoval za významný a cenný, ignorovali. Z tejto skúsenosti sa tento človek naučil: bez ohľadu na to, aký dokonalý opis procesu dokážete ako vedúci projektu vytvoriť, pre vývojárov je v skutočnosti potrebný len stručný jedno alebo dvojstranový postup.

Žiadny (akokoľvek dobrý) postup však nenahradí nespoľahlivých ľudí. Projekt treba obsadiť „dobrými“ pracovníkmi a na ich riadenie použiť jednoduchý a dobre

definovaný postup. Takýto postup dokáže tímu pomôcť pri spolupráci a dosiahnuť mimoriadne výsledky.

Najdôležitejšie dôvody pre vytvorenie a zavedenie postupu do FDD (vo všeobecnosti do ľubovoľnej prístupu ASD1) sú [Coad1999]:

Vývoj väčších systémov a opakovaný úspech. Aby tím mohol pracovať na vývoji rozsiahlych systémov, keďže ide o riadený proces vývoja a nie o tzv. *hacking*, potrebuje dobrý postup = systém na budovanie systémov. Jednoduchý a dobre definovaný postup funguje najlepšie. Tím ho dokáže aplikovať niekoľkokrát, upravovať pre konkrétny vývoj a zapamätá/osvojí si ho. Namiesto realizovania krokov postupu sa tím zameriava na podstatu vytváraného systému a výsledky.

Uvedenie nových pracovníkov. Dobre definovaný a jednoduchý postup umožňuje ľahké uvedenie nového pracovníka do projektu, pričom sa rapídne zníži čas jeho zaučenia. Nováčik tak rýchlejšie porozumie ako sú veci navrhnuté, implementované, aký štandard v programovaní sa používa, namiesto zdĺhavého vysvetľovania resp. študovania kopy dokumentácie.

Zameranie sa na podstatné výsledky. Často sa stáva, že niektorí vývojári idú za hranice toho, čo je potrebné, a v extrémnom prípade sa usilujú o dokonalosť jednej časti bez uvažovania zvyšku systému. Je veľmi dôležité, aby sa tím zameriaval iba na podstatné a „výnosné“ výsledky. Tzn. treba stanoviť a dodržať priority, určiť nevyhnutné črty a označiť v nich hranicu, čo je postačujúce. Táto hranica by sa nemala pri žiadnej črte prekročiť skôr, kým ju nedosiahnú aj všetky

¹ Agile Software Development

ostatné črty. Toto je možné dosiahnuť napr. týždenným vykazovaním progresu všetkých členov tímu.

Roly vo vývojovom tíme

Vo vývojovom tíme FDD sú podstatné dve roly (vedúci programátor a vlastník triedy) a jedna spoločenská štruktúra (tím pre jednu črtu) [Coad1999]. Keďže celkový model systému predstavuje objektový model, FDD používa pojem vlastník triedy. Ak sa však celkový model systému skonštruuje iný ako objektový model, možno použiť iný pojem na označenie vlastníka triedy, napr. vlastník procedúry, modulu, funkcie, atď.

Vedúci programátori

Fázy FDD „navrhni podľa črty, vytvor podľa črty“, riadenie implementácie črt v postupnosti za sebou, vzorový návrh a programovanie, a poradenstvo (najmä formou kontroly) si vyžaduje niekoho veľmi produktívneho, produktívnejšieho ako sú ostatní členovia tímu – vedúceho programátora.

Počet vedúcich programátorov ovplyvňuje rýchlosť realizácie projektu. Ak je potrebné zvýšiť rýchlosť projektu, treba zapojiť ďalších vedúcich programátorov. Multiplikačný efekt možno dosiahnuť kombináciou prirodzeného talentu, kvalifikácie, schopností a skúseností, zriedka vyskytujúcich sa v jednej osobe.

Pridávanie ďalších programátorov vedie často k spomaleniu projektu tak, ako to opísal Fred Brooks vo svojej eseji „Mýtický človekomesiac“ [Brooks1995]. Toto pravidlo platí vo všeobecnosti. V prípade FDD však pridaním vedúceho programátora zapojiť ďalších ľudí sústredených okolo neho, a v skutočnosti tak zvýšiť rýchlosť realizácie

projektu tým, že sa zvýši počet paralelne sa vyvíjaných črt. Počet vedúcich programátorov je však aj v tomto prípade zhora ohraničený.

Vlastníci tried

Vlastník triedy (objektov) je zodpovedný za návrh a implementáciu triedy. Tento spôsob určenia zodpovednosti funguje spoľahlivo a účinne. Po prvé, vývojári získavajú pocit vlastníctva určitej časti kódu, čo je prínosom v motivačnej oblasti – každý je hrdý na svoje dielo. Po druhé, tento spôsob prináša lokálnu konzistenciu v rámci triedy – kód modifikuje iba jeden programátor.

Štandardne sa používa pravidlo: jedna trieda, jeden vlastník. Avšak pri zložitejších metódach možno použiť: jedna trieda, jeden vlastník triedy a jeden programátor algoritmov.

FDD usporadúva aktivity podľa črt, nie podľa tried, pretože FDD je o vytváraní častých, konkrétnych a funkčných výsledkoch – zákazníci používajú črty (!), nie štruktúry, ktoré vývojári použijú na implementáciu malých kúskov určitej črty.

Tímy pre jednotlivé črty

Črty sa priradujú vedúcim programátorom, z ktorých každý identifikuje vlastníkov tried, ktorí sa budú podieľať na vývoji konkrétnej črty. Vedúci programátor sformuje dočasný tím pozostávajúci z vlastníkov tried, ktorý spoločne pracuje na jednej črte jeden až dva týždne.

Vlastníci tried pracujú naraz na vývoji viacerých črt. Členstvo v tíme sa môže meniť v každom cykle „navrhni podľa črty, vytvor podľa črty“.

Interakcie v rámci tímu sú najmä medzi vedúcim programátorom a vlastníkmi tried. Tento prístup urýchljuje

postup v takto riadenom vývoji, zaist'uje priebežné poradenstvo a podporuje uniformitu návrhu a implementácie.

Vo všeobecnosti vedúci architekt riadi vedúcich programátorov, a vedúci programátorov riadia vlastníkov tried v rámci tímu.

Sledovanie postupu

Z hľadiska riadenia FDD poskytuje dosť presné sledovanie a vykazovanie postupu vývoja. Najviac času (až 77% [Coad1999]) spotrebuje fáza „navrhni podľa črty, vytvor podľa črty“, ktorá sa vykonáva v dvojtýždňových cykloch. Každému kroku v rámci tohto cyklu sa priradí váha podľa schémy znázornenej na obr. 4:

Rekapitulácia doménovej oblasti	Kontrola návrhu	Návrh	Programovanie	Kontrola kódu programu	Zakomponovanie do finálnej verzie
1%	40%	3%	45%	10%	1%

Obr. 4: Časová náročnosť jednotlivých krokov v rámci cyklu „navrhni podľa črty“ a „vytvor podľa črty“ [Coad2000].

Každý krok predstavuje míľnik vo vývoji určitej črty. Napr. ak sa v rámci realizácie danej črty zrekapituluje aplikačná oblasť a vytvorí návrh tejto črty, možno povedať, že 41% z črty je hotových. Takýto prístup si je možné dovoliť iba vďaka tomu, že systém je rozložený na drobné zložky (črty), ktoré sa dajú implementovať v priebehu krátkeho času.

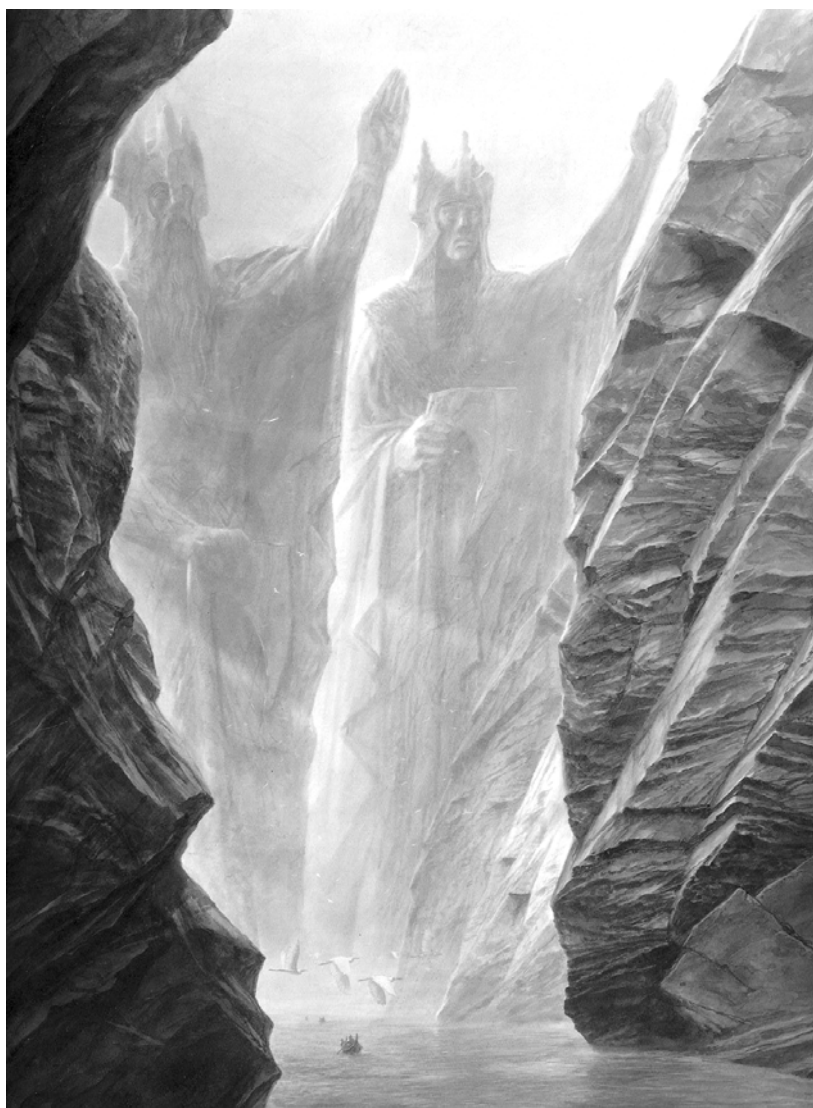
Vedúci programátor označuje každý krok vo vývoji črty v prehľadnom diagrame. Pokiaľ sa vývoj črty ukončí, vyfarbí sa na zeleno, pokiaľ vývoj ešte pokračuje, vyfarbí sa na žltu,

pokiaľ ešte nezačal, vyfarbí sa na bielo, a pokiaľ črta vyžaduje pozornosť, vyfarbí sa na červeno. Takto vymaľovaný zoznam črt sa umiestni na jednu plochu, čo poskytuje dobrý vizuálny efekt o celkovom postupe a zároveň poskytuje podrobnosti o realizácii konkrétnych črt.

Pre vrcholový manažment a zákazníkov sa vykazuje mesačný postup o vývoji priamočiarou transformáciou týchto percent na skupiny črt a hlavné skupiny črt. Pre každú hlavnú skupinu sa nakreslí rámček, do vnútra ktorého sa zakreslia rámčeky pre každú skupinu. Tieto vnútorné rámčeky obsahujú meno skupiny; počet črt, z koľkých sa skupina skladá; počet percent vyjadrujúci počet ukončených črt; a plánovaný dátum ukončenia. Každý rámček má príslušnú farbu podľa toho, v akom štádiu sa skupina nachádza. Keď je skupina úplne dokončená, celá bunka sa vyfarbí na zeleno.

Záver

FDD ako jedna z agilných metód predstavuje vhodný prístup vývoja softvéru v treťom tisícročí, ktoré je poznačené neustále sa meniacim prostredím a novými požiadavkami zo strany zákazníkov. Výber tejto metódy prináša časté a funkčné výsledky, ktoré uspokojujú ako zákazníkov, tak aj vývojárov a manažment. Pre firmu zaoberajúcu sa tvorbou softvéru na zákazku predstavuje tento prístup značné zníženie rizika prekročenia časového plánu a rozpočtu.



Priezračné techniky

Bc. Mária Šmigurová

Na vývoj softvérového systému pôsobí veľa faktorov, jedným z najdôležitejších sú ľudia. Práve od nich vo veľkej miere závisí úspech softvérového systému. Táto skutočnosť podnietila vznik prístupu Priezračné techniky (Crystal methods), ktoré sa zameriavajú hlavne na ľudí, na prácu v tíme, na komunikáciu medzi členmi tímu. Ponúkajú možnosti ako optimalizovať a uľahčiť prácu v tímoch pri vývoji softvérových systémov. Taktiež sa snažia prispôbovať konkrétnym projektom a priniesť do vývoja softvéru ľahkosť a priezračnosť spojenú s vytváraním softvéru.

„Svet sa predsa mení: Cítim to vo vode, cítim to v pôde, cítim to vo vzduchu. Myslím, že sa už nestretneme.“

Priezračné techniky (Crystal methods, odteraz CM) sú súčasťou agilných metód vývoja softvérových systémov. Agilné metódy sú v súčasnosti najperspektívnejšie metódy vývoja softvérových systémov, pretože ich hlavnou charakteristikou je prispôsobivosť, schopnosť reagovať na zmeny, čo je v dnešnom rýchlo sa meniacom svete takmer neodmysliteľná vlastnosť.

Tento článok predstavuje priezračné techniky vývoja softvéru. Vysvetľuje možnosti ich použitia, ich význam a výhody. Zaoberá sa ľuďmi ako základným kameňom CM a stratégiami práce v tíme.

Čo podnietilo vznik Priezračných techník?

Vznik CM podnietilo niekoľko faktorov, ktoré sa stali základnými kameňmi CM. Sú to: potreba prispôbovať sa, zefektívniť tímovú prácu, sprehľadniť a uľahčiť proces vývoja softvéru.

Aj napriek tomu, že bolo vyvinutých nesmierne veľa softvérových systémov, neexistuje zovšeobecnenie spôsobu vývoja na jednu metódu vývoja softvérového systému. Preto nie je veľmi vhodné prispôbovať projekt nejakej metóde, ale radšej *prispôbiť* metódu projektu, pretože každý projekt je iný, má odlišné požiadavky, sú preň vhodnejšie iné postupy, stratégie, metódy, prístupy.

Na vývoj softvérového systému pôsobí veľa faktorov, jedným z najdôležitejších sú *ľudia*. Od nich vo veľkej miere závisí úspech softvérového systému. Na projektoch väčšinou nepracuje jeden človek, ale *tím* ľudí. Práca v tíme je veľmi náročná a závisí najmä od schopností jednotlivcov pracovať v tíme. Každý člen tímu je *individualita*, prináša svoje zručnosti a schopnosti, jeho rast (a to nielen v profesionálnej oblasti), ovplyvňuje aj jeho prácu v tíme.

Pri tímovej práci je veľmi dôležitá *komunikácia*. Bez kvalitnej komunikácie tím nemá veľké šance na úspech. Lepšia a častejšia komunikácia znižuje množstvo práce vynaloženej na projekte, teda zvyšuje efektivitu tímu.

Nie vždy sa dá alebo nie vždy je vhodné pri vývoji softvéru použiť *rozsiahly proces* vývoja softvéru. V prípade potreby šetriť so zdrojmi (čas, peniaze, ľudia), si nemôžeme dovoliť luxus vo forme písania rozsiahlej dokumentácie pre požiadavky, analýzu, návrh, plánovanie, testovanie, a pod. Ale aj napriek tomu chceme vyrobiť kvalitný softvér, nechceme, aby naša práca v tíme bola ad hoc, aby sa na nič nezabudlo, nič sa nevynechalo.

Jednou z reakcií na tieto skutočnosti, ktorá sa snaží o proces vývoja softvéru zameraného na ľudí, tím, prispôbovanie, prehľadnosť a ľahkosť prinášajú CM.

Čo sú to Priezračné techniky?

CM sú prístupy k vývoju softvéru, ktoré sa zameriavajú hlavne na ľudí (*human-powered*), na *prispôsobovanie* sa projektu (*adaptive*), na *ľahkosť* (*ultralight*) a *priezračnosť* (*crystal*) procesu vývoja softvéru a snažia sa *optimalizovať* jeho veľkosť. Pomáhajú ľuďom uľahčiť a zefektívniť prácu v tímoch pri vývoji softvérových systémov [Cockburn1].

Základné kamene

Ludia – úspech projektu závisí hlavne od ľudí, ktorí pracujú na projekte. Je založený na kvalite a množstve komunikácie medzi nimi a schopnosti jednotlivých členov pracovať v tíme. (Niektoré iné prístupy môžu byť zamerané na proces vývoja softvérového systému (SS) alebo na architektúru SS alebo na použité nástroje pri tvorbe SS, ale CM sú je zamerané na ľudí).

Ľahkosť - CM sa zameriavajú na zmenšovanie (redukciu) „papierovej roboty“ na čo najmenšiu možnú veľkosť. Samozrejme berie sa ohľad na predpokladané použitie a parametre projektu. Zahŕňajú všetko čo súvisí s veľkosťou projektu a s jeho prioritami.

Optimálnosť – snaha všetko optimalizovať, zmenšovať, robiť "fit".

Priezračnosť – programovať prehľadne a jednoducho, radšej ubrať z efektívnosti, aby sa funkciám dalo ľahko porozumieť.

Prispôsobivosť – prispôsobovanie sa konkrétnemu projektu, podmienkam. CM sú len odporúčania, nie presne definované postupy, preto si ich každý tím môže „ušiť na mieru“.

Ľudia

Tímová práca je neoddeliteľnou súčasťou každého väčšieho projektu. Tím tvoria ľudia, ktorý sú jedinečné osobnosti, majú rôzne schopnosti a zručnosti. Spoločnou prácou však môžu dosiahnuť, čo by ani jeden z nich sám nedosiahol.

Tím sa stáva lepším tak ako sa každý z jeho členov stáva lepším a ako sa zlepšuje ich vzájomná spolupráca a interakcia. Dôležité je budovať vzájomnú dôveru a spoluprácu. Netreba zabúdať na stretnutia tímu, niekedy krátke stretnutia môžu byť lepšie ako dlhé. Dôležitá je motivácia ľudí, snaha vytvoriť niečo dobré, hľadať riešenia a možnosti zlepšenia.

Pri tímovej práci je dôležité stanoviť základné pravidlá na prácu v tíme, stratégie tímu a tieto dôsledne dodržiavať a priebežne sledovať ich dodržiavanie. Takúto dohodu predstavuje metodológia. Je to dokument, ktorý presne stanovuje dohody a stratégie tímu. Dôležité je, aby všetci členovia tímu s nimi súhlasili, stotožnili sa s nimi a dodržiavali ich pri práci [Cockburn2].

Aké tímové stratégie použiť? Tímy zostavené z 3 až 8 ľudí môžu sedieť v tej istej miestnosti a rozprávať sa medzi sebou. Každý člen tímu má prehľad o práci celého tímu, uvedomuje si jednotlivé zmeny v projekte, ale aj atmosféru v tíme, to pomáha projekt posunúť ďalej. Pri vyššom počte ľudí v tíme to už nie je také ľahké a ani možné. So vzrastajúcim počtom ľudí rastie aj počet odlišností medzi nimi, mení sa charakter ich práce a aj veľkosť spolupráce. Bude potrebné použiť iné stratégie ako pri malom tíme.

Podobné prístupy ako CM, zamerané na ľudí a ľahký vývoj softvéru sa nezvyknú používať v tímoch kde je viac ako 50 ľudí. V oblasti CM sa používa toto farebné označenie typov projektov (obr. 5): clear (tím = 3-8 ľudí), **yellow** (10-20),

orange (25-30), **red** (50-100), **maroon** (100-200), **blue** (200-500), **violet** (viac ako 500 ľudí) [Cockburn1].

3-8	10-20	25-50	50-100	100-200	200-500	500+
-----	-------	-------	--------	---------	---------	------

Obr. 5: Farebné označenie CM podľa počtu členov v tíme [Cockburn1].

Neznamená to, že pre každú farbu existuje presne určená stratégia. Existujúce stratégie sú iba odporučené pre tímy danej veľkosti. CM sú založené na tom, že každý tím si vyberá stratégiu, ktorá mu najviac vyhovuje. Farebnou schémou sa nedá zobraziť fyzické rozmiestnenie tímu. Priame doporučenie CM je dať ľudí na to isté miesto. Potom sú oveľa lepšie možnosti na komunikáciu, nájde sa najlepšia cesta na ľahkú metodológiu.

Jednotliví ľudia majú individuálne charakteristiky, slabé a silné stránky. Jednotlivé charakteristiky ich lepšie predurčujú pre niektoré roly a pre niektoré zasa nie, zaradenie človeka do role môže mať podľa toho rôzny efekt na projekt (pozitívny alebo negatívny). Preto treba brať ohľad na celkovú charakteristiku človeka pri určovaní jeho roly v tíme.

Taktiež je vhodné zosumarizovať si čo sa očakáva od jednotlivých rolí, v akých druhoch projektov je vhodné použiť aké roly, aké výsledky od nich môžeme očakávať, aké prístupy môžeme použiť na zvýšenie produktivity.

Priehľadné techniky pre najmenšie tímy

Keďže ako väčšina študentov som ešte nemala možnosť pracovať vo väčšom tíme, rozhodla som sa priblížiť filozofiu CM na najjednoduchšej z nich a tou je CM pre najmenšie tímy (*Crystal Clear*, odteraz CC). Je zaujímavé konfrontovať CC s prácou na tímovom projekte, na ktorom pracuje približne 6 ľudí, pretože tento počet spadá práve pod CC.

Viac podrobností o CC sa môžete dočítať v [Cockburn3].

Odporúčania *Crystal Clear*

CC nedefinujú presný postup a pravidlá vytvárania softvérového systému, dávajú iba odporúčania ako postupovať pri práci v malých tímoch.

Umiestniť tím do jednej miestnosti. Keďže tím tvorí 3-8 ľudí nie je problémom umiestniť ich do jednej miestnosti. Vďaka tomu sa zjednoduší komunikácia medzi nimi. Keď si chcú niečo povedať alebo oznámiť, stačí len zakričať. Možno si niekto povie: „Prečo dávať ľudí do jednej miestnosti, keď máme predsa vyspelú techniku, môžeme používať elektronickú poštu, telefón, videotelefón a pod., čo sú tiež veľmi rýchle formy komunikácie. Alebo len jednoducho zaskočím ku kolegovi do vedľajšej kancelárie.“ Nie je to však celkom tak, pretože počas toho kým zdvihnete telefón alebo sa vyberiete za niekým sa môže všeličo stať, niekto vás zastaví na chodbe, zazvoní telefón, atď.

Zvýšiť rozsah komunikácie. Keď tím sedí v jednej miestnosti môžu neustále medzi sebou komunikovať. Vedia o všetkom čo sa deje. Ak má niekto problém stačí sa spýtať a každý z tímu mu môže poradiť, pretože majú prehľad o všetkom a o všetkých zmenách.

Znížiť zbytočnú prácu. Tým, že majú prehľad o všetkom, netreba písať dokumentáciu pre členov tímu, stačí si robiť poznámky na veľkej bielej tabuli, ktorá je uprostred miestnosti. Netreba napríklad písať požiadavku na zmenu, stačí keď sa stretne celý tím, prediskutujú možnú zmenu a zaznačia ju na tabuli. Ďalším príkladom môže byť kreslenie obrázkov. Ako vieme kreslenie rôznych modelov funkčnosti, údajov, správania sa systému zaberie veľmi veľa času a aj napriek tejto veľkej snahe sa môže stať, že používateľ týmto komplikovaným modelom vôbec neporozumie. CC odporúča radšej urobiť niečo jednoduchšie ako napr. prípady použitia vo forme krátkych postupov, ktoré vznikli počas rozhovorov so zákazníkom a sledovania jeho práce.

Zapojiť používateľa. Keďže používateľ zadáva požiadavky na systém je veľmi vhodné ho čo najviac zapojiť do procesu vývoja, pretože môže okamžite korigovať všetky zmeny alebo dávať nové požiadavky. Vďaka tomu, že spolupracuje vytvára sa systém, ktorý používateľ spoznáva takpovediac od narodenia a môže ovplyvňovať jeho vývoj a formovanie, a tak sa nemôže stať, že nakoniec bude prekvapený a nespokojný s výsledným produktom.

Zvýšiť spätnú väzbu. Spätná väzba pomáha zefektívneniu práce. Poučíme sa čo robíme zle čo dobre, ako postupovať, čo kam vedie.

Vytvárať funkcionality. CC odporúča stanoviť si počas vývoja niekoľko míľnikov a tými má byť práve dodaná funkcionality systému. Takýto proces je dostatočne viditeľný a jasne sa dá sledovať jeho progres. Ďalším odporúčaním je nesnažiť sa vyvíjať čo najefektívnejšie funkcie systému, ale čo najprehľadnejšie a najjednoduchšie, aby boli zdrojové kódy prehľadné a ľahko pochopiteľné, potom netreba vytvárať ani

zložitú dokumentáciu o vytvorených funkciách, pretože sú dostatočne jasné.

Čo prináša *Crystal Clear*?

CC je vhodné použiť na projekty pre 6 ľudí, najvhodnejšia je však pre 3-4 ľudí. CC je vhodný pri vytváraní mainframe systémov, systémov klient-server, aj pre všetky databázové systémy – centrálné aj distribuované.

CC prináša *silnú komunikáciu*, bohatú na informácie, *priezračný a ľahký vývoj*, prehľadné a jednoduché programovanie na úkor efektívnosti funkcií, *doručiteľnosť*, pretože je tu veľa komunikácie aj so zákazníkom, zákazník vie čo sa vytvára, *frekventované doručovanie*, každé tri mesiace sa vytvárajú nové verzie. Viacej doručovaní a veľká komunikácia so zákazníkom, ale aj v tíme, znižujú potrebu vytvárania medzi produktov, *redukovanie sa nadbytočnosti*, vytvára sa menej dokumentácie v rámci tímu, stačí len niekoľko náčrtkov a poznámok, vytvára sa len dokumentácia pre zákazníka – používateľská príručka, dokument o testovaní a pod.

Malé tímy dávajú dobré výsledky používaním ľahkých prístupov zahŕňajúcich rozsiahlu interakciu medzi ľuďmi a *spätnú väzbu* od zdrojového kódu k ľuďom.

Kedy nepoužívať *Crystal Clear*

CC nie je vhodné *pre väčšie tímy a projekty*. Pri veľkých tímoch sa stráca koordinácia medzi jednotlivými členmi tímu. Vôbec nie je vhodné pre systémy, od ktorých závisia ľudské životy, pretože *chýba verifikácia* a plánovanie. Takéto systémy by mali byť navrhované veľmi systematicky s overením každej etapy.

Akých ľudí potrebujeme pri *Crystal Clear*

Tím CC by mal mať *skúseného návrhára*, je to človek, ktorý vie čo treba urobiť, má skúsenosti, tvorí jadro návrhu systému, určuje štruktúru tímu, prideluje jednotlivé roli členom tímu. Bez tohto človeka nemôže tím fungovať. Ďalším členom tímu by mal byť *programátor návrhár*, ktorý navrhuje obrazovky, programuje zdrojový kód, pripravuje údaje na testovanie.

Pri CC je dôležité, aby členom tímu bol aj *používateľ*, používateľ pomáha s vytváraním prípadov použitia a s návrhom obrazoviek. Keď je používateľ členom tímu, nemôže sa stať, že bude nespokojný s finálnym výsledkom. *Sponzor* udáva smerovanie tímu, základnú stratégiu, má na starosti financie tímu. Tieto roly by nemal vykonávať jeden človek, mali by byť rozdelené medzi štyroch ľudí, pretože ich úloha je veľmi dôležitá a musia sa na ňu plne sústrediť.

Ďalšie možné roly v tíme CC sú už zlučiteľné aj s predchádzajúcimi: *expert na používateľa*, môže ním byť aj sponzor, používateľ alebo starší návrhár, *expert na používateľa* je odborníkom na prostredie nasadenia systému, vytvára informatívne prípady použitia a dokument požiadaviek. *Koordinátor* zabezpečuje následnosť, plánovanie a kontrolu verzií, vytvára zoznam rizík, dáva informácie o stave projektu, môže ním byť aj návrhár. *Tester* robí testovanie systému, o priebehu testovania vytvára hlásenie, *pisár* vytvára dokumentáciu, používateľskú príručku, tieto dve úlohy môže vykonávať aj návrhár.

Aké štandardy používa *Crystal Clear*?

V tíme CC sa používa inkrementálna stratégia vytvárania verzií, míliaiky tvorí dodaná funkcionálna systému.

Na tvorbu špecifikácie požiadaviek sa vytvárajú jednoduché prípady požitia. Na celý systém sa používa regresné testovanie a prehliadka kódu.

Každý tím si určuje svoje lokálne štandardy: štýl programovania, návrh používateľského rozhrania, testovanie.

Každý štandard má určitú mieru tolerancie, treba ho buď dodržiavať striktnie alebo je len odporúčaním. CC vyžaduje striktnie dodržiavať postupy pri regresnom testovaní, vydávaní verzií, pri prefixoch tried a pri menách metód. Odporúčania sú pri tvorbe používateľskej príručky, komentovaní zdrojového kódu a pod.

Ako prebieha práca na projekte využitím *Crystal Clear*

Predstavme si projekt na ktorom pracuje tím 3 až 8 ľudí. Preto aby si boli čo najbližšie, sedia v jednej miestnosti. Veľmi blízky kontakt je jednou z najdôležitejších vecí. Komunikujú jednoducho – kričia na seba cez celú miestnosť. Snažia sa vytvoriť tzv. „tím jednej mysle“. Najskôr si spolu sadnú a snažia sa pochopiť problém, ako ho riešiť a až potom začnú pracovať. Vďaka tomu, že sú v jednej miestnosti, môžu neustále komunikovať, oznámiť si dôležité skutočnosti, rozhodnutia, každý člen tému vie čo sa deje.

Ako návrhové médium používajú bielu tabuľu. Ak sú v strese a nestíhajú tak do dokumentáciu používajú odfotenú tabuľu. Nevýhodou tabule je, že si ju nemôžu zobrať so sebou alebo vytlačiť.

Najskôr prebiehajú rozhovory so zákazníkom a so sponzormi. Jeden z členov tímu ide k zákazníkovi a pozoruje jeho prácu, aby lepšie pochopil jeho požiadavky, stáva sa expertom na používateľovu prácu. Na tvorbu požiadaviek

používa veľmi jednoduché prípady použitia (*light use case*). Sú to krátke postupy, ktoré opísal používateľ. S používateľom udržujú veľmi blízku a častú komunikáciu. Najskôr sa napíšu nejaké návrhy, ktoré potom používateľ upresní. Spíše požiadavky ako jednoduché prípady použitia, určí si priority, čo treba urobiť najskôr.

Ďalší člen tímu sa stáva technickým expertom, podľa požiadaviek a technických možností určí čo sa dá a čo je možné realizovať. Vytvorí architektonický návrh.

Potom urobia plán vydávania verzií, vo všeobecnosti sa uvažuje o vydávaní verzií každá dva alebo tri mesiace. Po vydaní každej novej verzie prebieha diskusia so zákazníkom.

Po návrhu jadra systému si prácu v tíme rozdelia na jednotlivé časti, ale nepracujú na všetkom oddelene, niektoré časti vytvárajú spoločne. CC a jej podobné prístupy predpokladajú, že v tíme je vždy aspoň jedna osoba, ktorá vie čo treba robiť, nejaký skúsený odborník. A ten má zvyčajne na starosti aj nováčikov. V prípade, že sú dvaja nováčikovia celý tím musí mať rovnomerne rozložené skúsenosti, musí byť dobrý.

Prvé demo vytvoria v priebehu niekoľkých týždňov, obsahuje hlavne obrazovky systému naplnené nejakými skúšobnými údajmi. V priebehu mesiaca sa potom vytvorí prvé funkčné demo.

Míľnikmi pre prácu v tíme je dodaná funkcionálna. Pri tvorbe dokumentácie sa zameriavajú hlavne na tvorbu používateľskej príručky, dokumentu o testovaní. Dokumentácia v rámci tímu je minimálna, používajú sa jednoduché náčrtky, obrázky, poznámky a pod.

Záver

Už z názvu CM (*Crystal methods*) vyplýva, že ide o veľmi priezračný, priehľadný a ľahký proces vývoja softvérových systémov. CM prináša veľa zjednodušení a optimalizácie práce na vývoji, ale hlavne zaoberá sa ľuďmi a tímovou prácou. práve táto vlastnosť posúva CM dopredu, pretože ľudský faktor je pri vývoji softvéru veľmi rizikový.

CM sú vlastne iba určité odporúčania, ktoré sa zameriavajú hlavne na ľudí. Ponúkajú rôzne stratégie na prácu v tímoch, postupy ako zefektívniť tímovú prácu. Nepredpisujú žiadne striktné postupy, a preto sú veľmi ľahko prispôsobiteľné konkrétnemu projektu a tímu [Crystal2000].

Veľkou výhodou CM je, že je ich možné použiť aj v rámci inej metódy alebo prístupu na zlepšenie tímovej práce. A teda keď nechcete používať CM, skúste aspoň použiť niektoré z prístupov alebo stratégií pre prácu v tíme. Určite sa vám podarí zefektívniť prácu, zlepšiť komunikáciu a vzťahy v tíme.



Extrémne programovanie

Bc. Richard Richter

Jedným z najradikálnejších prístupov, ktoré si môžete osvojiť s cieľom zvýšiť produktivitu tímu, je extrémne programovanie (Extreme programming, XP). Nasledujúca esej sa zaoberá princípmi, na ktorých stojí XP, a opisuje výhrady, ktoré voči nemu majú jeho oponenti. Máte malý tím čínorodých ľudí, ktorí spolu dobre vychádzajú? Potom možno práve pre Vás bude tento prístup, ktorý môže skupinu hackerov zmeniť na efektívne pracujúci tím.

*„O túto úlohu sa môže pokúsiť slabý rovnako ako silný.
A predsa sa často veľké skutky odohrávajú tak: malé ruky
ich konajú, pretože musia, kým oči veľkých sú obrátené
inam.“*

Extrémne programovanie (XP) je jednou z odpovedí na problémy, ktoré prináša vývoj softvéru v dnešnom svete. Znalec agilných metód zrejme nebude namietat', ak povieme, že nie nadarmo má XP vo svojom názve prívlastok „extrémne“. Nie náhodou ho Barry Boehm [Boehm2002] zaradil v plánovacom spektre vedľa tzv. hackerov, často nazývaných aj programátori-kovboji. Napriek tomu, že je XP od metód založených na plánovaní skutočne ďaleko, má s nimi ale jeden podstatný menovateľ – a tým je tímová práca. Aj v tomto smere má však XP svoje špecifické (ak chcete „extrémne“) črty.

Na čom stojí extrémne programovanie?

Autorom metodológie je Kent Back – významným momentom v histórii XP je vydanie jeho knihy *Extreme Programming Explained*. Back za základné piliere XP zvolil štyri pojmy:

- jednoduchosť (*simplicity*);
- komunikácia (*communication*);
- spätná väzba (*feedback*);
- guráž (*courage*).

Jednoduchosť nám káže spraviť len to, čo je naozaj nutné, len to, čo zákazník skutočne chcel – nie viac. Komunikácia je potrebná na spoluprácu v tíme – o tom, že pre XP to platí dvojnásobne, sa ešte presvedčíme. Spätná väzba je zabezpečovaná malými vývojovými iteráciami – tak vieme pružnejšie reagovať na zmeny, máme projekt viac pod kontrolou a rovnako tak vieme, kedy nastane ten správny čas na prerobenie časti kódu (tzv. *refactoring*). A na samotný refactoring potom potrebujeme spomínanú guráž.

Na prvý pohľad najmä posledné vety zaváňajú skutočným hazardom. Aby odvaha potrebná k refaktoringu nevyústila v katastrofu, opiera sa XP veľmi silno o testovanie ako súčasť tvorby softvéru (teda nie len na záverečné testovanie). Poďme sa bližšie pozrieť na praktiky, ktoré robia XP tým, čím je.

Základné praktiky

Základných praktík, ktoré charakterizujú XP, je trinásť (uvádzam pôvodný termín a voľný preklad):

- *Jeden tím (Whole Team)*: Tím sa formuje okolo zástupcu zákazníka, ktorý sa tak stáva súčasťou tímu. Ak je niekto za zákazníka členom tímu, dosahuje sa užšia spolupráca; prípadné odchýlky od skutočných predstáv zákazníka sa rýchlo odstraňujú. Je to rozhodne efektívnejšie ako nekonečné vymieňanie si špecifikácií. Dôležitým faktorom je aj lepšie vybudovaný vzťah so zákazníkom.

- *Plánovanie hrou (Planning Game)*: Plánovanie je jednoduché, určujú sa len najbližšie úlohy a odhaduje sa čas dokončenia projektu. Tím sa riadi podľa potrieb zákazníka a rieši požiadavku za požiadavkou.
- *Malé vývojové cykly (Small Releases)*: Časté verzie zabezpečujú minimálny odklon od potrieb zákazníka, ako aj možnosť častého testovania celku. Viditeľnosť projektu – najmä pre zákazníka – je jedným z najväčších problémov klasického modelu vývoja softvéru. Časté prinášanie nových funkčných verzií tento problém rieši, čo zákazník rozhodne uvíta.
- *Zákaznícke testy (Customer Tests)*: Sú dané testy, ktorými musí „release“ prejsť (a tým je vlastne určené, kedy už ide o „release“). Na základe špecifikácie (hoci nepísanej) sa ešte pred implementovaním zostavujú testy – tie napokon sú tiež istou formou špecifikácie. Oproti týmto testom sa potom vždy produkt validuje.
- *Jednoduchý návrh (Simple Design)*: Jednoduchý návrh, ktorý pokrýva momentálne potreby, ale nič viac. XP tento prístup zhŕňa do vety „*you ain't gonna need it*“ (akronym YAGNI). Faktom je, že veľmi často programátor robí zbytočné veci, pripravuje sa na možnosti, ktoré nenastanú – preto ono „toto nebudeš potrebovať“.
- *Programovanie v pároch (Pair Programming)*: Hoci sa môže na prvý pohľad zdať programovanie v pároch neefektívne, nie je to pravda – ba dokonca až opak môže byť pravdou. Ak za jedným počítačom pracujú dvaja ľudia, ich spoločný výkon môže byť lepší ako výkon dvoch samostatne pracujúcich programátorov.

Tento paradox sa prejaví najmä, ak sa posudzuje kvalita výsledného kódu.

- *Vývoj riadený testami (Test-Driven Development)*–: Každá trieda / metóda / časť kódu (*unit*) musí prejsť testom. Akonáhle je napísaná jednotka kódu, hneď sa testuje, pričom *testy sa píšu ako prvé!* Spolu s párovým programovaním ide o dva najzásadnejšie prvky XP. Sada testov sa stáva súčasťou projektu.
- *Vylepšovanie návrhu (Design Improvement Refactoring)*: Keď je potreba vylepšiť návrh pre nové požiadavky, spraví sa to. Keď je potreba preprogramovať časť projektu kvôli novému návrhu, spraví sa to.
- *Neustála integrácia (Continuous Integration)*: Integrácia je neustávajúci proces, takže systém je vlastne stále integrovaný a schopný behu.
- *Spoločné vlastníctvo kódu (Collective Code Ownership)*: Nakoľko sa píše v pároch, kód patrí kolektívu. Z toho vyplýva aj možnosť zmeniť ho v prípade potreby. Možnosť meniť (a istá bezstarostnosť pri tejto činnosti) sa opiera o existenciu sady testov.
- *Spôsob písania kódu (Coding Standard)*: Aby bolo možné vlastniť kód kolektívom, je nutné zhodnúť sa na potrebných zásadách – spôsob, akým sa píše kód, musí byť jednotný.
- *Vízia systému (Methaphor)*: Tím má vytvorený jeden názor, jednu víziu systému. Tým, že zdieľa jeden pohľad na systém, je jednoduchšia komunikácia v tíme.

- *Vlastné tempo (Sustainable Pace)*: Pracovať viac neznamená nutne viac spraviť (obdobne to platí pre kvalitu). Nadčasy sa robia vtedy, keď to má význam – vývojár nutne musí naraziť na dni, kedy sa mu nedarí, nech robí, čo robí. Každý pozná svoje tempo. Neustále pracovanie v jednom tempe môže jednému vyhovovať, inému nie.

Kedy je Extrémne programovanie rozhodne dobrý nápad?

XP – podobne ako iné agilné metódy – veľmi silno stavia na ľuďoch a ich ľudských kvalitách. Teda nie len profesionálne zdatnosti, ale aj ľudské kvality sú *nutnou* podmienkou na nasadenie XP. Ide najmä o dostatok dobrej vôle „robiť správne veci“ – často bez príkazov. Táto *agilita* (môžeme povedať čínorodosť), ktorá je všetkým týmto metódam vlastná, je pre XP absolútne kľúčová.

Dobré medziľudské vzťahy v tíme sú nutné – viete si predstaviť párové programovanie „s nepriateľom“? Rozhodne nie!

Kent Beck navrhoval použitie XP pre dvoch až desiatich ľudí. Faktom je, že väčší tím už má problém udržať vzájomnú komunikáciu na dostatočne efektívnej úrovni, aby ostala zachovaná jednotná vízia projektu. Pri extrémnom programovaní nie je nič nezvyčajné, ak po nájdení chyby ľubovoľný člen tímu túto lokalizuje a opraví. Takémuto prístupu napomáha jednak zdieľané vlastníctvo kódu, tak aj s tým súvisiaca filozofia *všetci všetko vedia*.

Aj napriek minimálnej dokumentácii a faktu, že jeden človek nemôže obsiahnuť celý projekt, XP funguje – a to práve na komunikácii medzi ľuďmi. Kent Beck postavil XP na

niečom, čo je pre nás prirodzené, ľudia sa stretávajú, rozprávajú, polemizujú a ešte aj pri programovaní sú vždy dvaja.

Osobne považujem práve ľudské kvality človeka za rozhodujúce pre úspešne nasadené XP, napriek tomu však vznikla vtipná poznámka, že XP je vhodné pre *malé tímy nadpriemerných vývojárov*. A nakoľko nadpriemerných nemôže byť viac ako polovica, nemôže byť extrémne programovanie pre každého. Pravdou aj je, že pre každého rozhodne nie je. Pre niektorých ľudí je jednoduchšie byť tlačенý normami a metodikami než používať vlastný vnútorný motor.

Extrémne programovanie nie je liek na všetko

Za istých okolností je možné dopredu odhadnúť architektúru systému. V takom prípade môže byť efektívnejšie zapracovať najprv len na architektúre a následne implementovať funkcie systému. Takýto postup je ale v rozpore so zásadami XP, ktoré hovoria, aby sa implementovali len požadované funkcie s tým, že architektúra sa modifikuje podľa potreby (*refactoring*). Pracovať istý čas len na architektúre je prekážkou skorého dodania prvých výsledkov zákazníkovi.

Rob len to, čo musíš a nie viac – ako už bolo spomenuté, XP túto skúsenosť formuluje do slov *you ain't gonna need it*. Presne tento princíp bráni programátorovi vyhrať sa s architektúrou, hoci môže vedieť, že je to dobrá investícia do budúcnosti.

Druhým problémom XP je jeho použiteľnosť pre malé tímy. Znalosť o systéme je zdieľaná vedomosť celého tímu, čo kladie vysoké nároky na fungujúcu komunikáciu. Ide pritom predovšetkým o komunikáciu typu každý s každým. Je ľahké

predstaviť si tak veľké tímy, v ktorých takáto komunikácia už nie je možná (alebo proste nie je efektívna).

Na tento problém našťastie jestvujú riešenia. Väčšinou je nutné problém rozdeliť na menšie celky – vytvoriť podprojekty, ktoré zvládnu tímy s typickou veľkosťou do desať ľudí. Jestvujú prístupy, ktoré počítajú s XP na najnižších úrovniach (napriek tomu by som nerád zjednodušil XP len na programovanie – ide totiž skutočne o vývoj).

Tretím problémom XP – ale vlastne všetkých agilných metód – je nižšia akceptovateľnosť u zákazníka. Mnohí zákazníci lipnú na konzervatívnych prístupoch a nie vždy im je jedno, aké sú procesy prebiehajúce vnútri organizácie, ktorá im má produkt vyvíjať. A v prípade extrémneho programovania stačí niekedy vidieť len ten názov (najmä jeho prvé slovo).

Ak teda zhrnieme veci, ktoré sú XP vytýkané, ide najmä o:

- Princíp YAGNI – ak je známa (alebo ľahko dedukovateľná, napríklad z veľkej množiny požiadaviek) architektúra systému, môže byť efektívnejšie venovať sa chvíľu jej miesto množstva refaktoringu.
- Malá škálovateľnosť tímu.
- Zavše odmietnutie zákazníka (má extrémizmus v názve).

Dokumentácia

Spôsob, akým sa XP stavia k dokumentácii, je typickou ukážkou extrémneho prístupu. Extrémne programovanie nehovorí o vnútornej projektovej dokumentácii prakticky nič. Nehovorí, ako ju treba písať, čo má zahŕňať, apod. Miesto toho XP hovorí: „Ak nemusíš dokumentovať, nedokumentuj.“ Ak

tím projektom žije, nepotrebuje vnútornú dokumentáciu na to, aby si v ňom udržal prehľad. To je napokon jeden z dopadov kolektívneho vlastníctva kódu.

Pri návrhu operujú členovia tímu s kartičkami (typicky reprezentujúcimi napríklad triedy), ale aj keď navrhnú „rozloženie“ týchto kartičiek, výsledok si udržiavajú v hlave, nerobia dokument.

Napriek vyššie uvedeným typickým prípadom, XP tím tiež môže dokumentovať. Buď takáto potreba vznikne vnútri tímu, alebo dokumentáciu požaduje zákazník. V druhom prípade ide o požiadavku, ktorá sa proste vypracuje a nik nešpekuluje nad tým, že miesto kódovania dokumentuje. Samozrejme, aj pri dokumentácii sa prejavuje zásada jednoduchosti.

Napriek tomu, že tvorba dokumentácie nie je v XP takpovediac obľúbená (kde napokon je?), celý problém je možné zhrnúť do konštatovania: *Dokumentuj, keď je potreba*. Nezáleží na tom, odkiaľ táto potreba vychádza.²

XP sa napokon na dokumentáciu pozerá zo širšieho pohľadu – za dokumentáciu sa považujú napísané testy (už sme spomínali, že test môže byť aj špecifikačným nástrojom – to spolu súvisí), dobre napísaný kód sa dokumentuje sám, apod. Na postoji XP k dokumentácii je teda vidieť, že sa dá skĺbiť extrémnosť s rozmyslom.

Záver

XP priťahuje mnohých programátorských nadšencov, ktorí ešte nezapadli do konzervatívneho programovania – dôvodom môže

² Na margo tejto zásady existuje krásny výrok: *We're Extreme, not stupid*, t.j. sme extrémni, nie však hlúpi.

byť, že oproti klasickým prístupom tu ide (hneď po zákazníkovi) o programovanie, vývoj. Programovanie tu nie je len písanie kódu, člen tímu sa aktívne podieľa na analýze, návrhu. Programovanie je *state of Art*, tak ako napokon vždy bolo.

Na druhú stranu má XP aj kritikov, často aj medzi programátormi-individualistami. Nemožno poprieť, že párové programovanie nie je pre každého (aj keď takmer každý programátor už riešil problém vo dvojici). Nemožno napokon poprieť, že v istých prípadoch nemusí byť XP tým najrýchlejším spôsobom (to by bolo, aby všetky metódy boli najrýchlejšie). Osobne sa mi na XP páči ten *pocit*, aký musí byť v tíme, aby to takto vôbec fungovalo. Nie je zas tak veľa skvelých ľudí okolo – a vytvoriť dobrý XP tím asi nie je jednoduché. No potom to musí byť úžasná vec.

Každopádne je XP veľmi inšpirujúci prístup k programovaniu a v mnohom ho je možné použiť, aj keď hneď nebude celý tím extrémny. Dúfam, že z prečítaných riadkov je zrejmé, že aj táto extrémnosť je v mnohom založená (pominúc ideály, s ktorými do toho musel niekto ísť) na zdravom rozume.



Skrumáž

Bc. Ivan Polák

Neaktuálnosť niekedy vedie až k nepoužiteľnosti vytvorených softvérových systémov. Rýchla zmena požiadaviek zákazníka a zmena vlastností cieľového prostredia určeného pre nasadenie softvérových systémov sú len niektoré základné súčasné problémy, ktoré vznikajú pri použití klasických zaužívaných prístupov k riadeniu tvorby softvérových systémov. Metóda Skrumáž (SCRUM) sa snaží odstrániť tieto problémy pomocou iteratívneho vývoja softvéru, tímovej práce a zmeny vzťahu medzi manažmentom a vývojovým tímom.

„Zdá sa, že sú trochu nad tým, aby sa mi páčili alebo nie. Sú iní, než som čakal - tak starí a tak mladí, a tak veselí a smutní.“

Čas a peniaze. Dva pojmy, ktoré ovplyvňujú všetko v 21. storočí. A inak to nie je ani v oblasti informačných technológií a softvérového inžinierstva. Ako v každej oblasti aj tu je pánom zákazník. Zákazník, ktorý v mnohých prípadoch nevie presne čo chce, ale chce to rýchlo a čo najlacnejšie. Tvorcovia softvérových systémov musia byť schopní neustále a rýchlo sa prispôbiť meniacim sa podmienkam a požiadavkám zákazníka.

V prípade využitia klasických prístupov k tvorbe softvérových systémov sa často stávalo, že hoci softvérový systém bol vytvorený presne podľa požiadaviek zákazníka, ale počas jeho tvorby došlo k tak rozsiahlym a zásadným zmenám v požiadavkách, ktoré boli na neho kladené, alebo v prostredí, pre ktoré bol vytvorený softvérový systém určený, že bol prakticky nepoužiteľný. Tento neželaný jav nastal v dôsledku spolupráce so zákazníkom na špecifikácii požiadaviek vytváraného softvérového systému len na začiatku jeho tvorby.

Počas tvorby softvérového systému nebolo možné pružne reagovať na zmeny, ktoré požadoval zákazník a podobne ani na zmeny prostredia, v ktorom sa predpokladalo nasadenie vytvoreného softvéru. Následkom tohto javu sa tvorcovia softvérového systému museli okamžite po jeho uvedení do prevádzky vrátiť k jeho tvorbe a to buď vo forme jeho údržby alebo vytvárania ďalšej verzie, aby bol softvér aspoň čiastočne použiteľný, čoho dôsledkom bol neúmerný rast finančných nákladov na jeho prevádzku a údržbu.

Veľmi dobrým príkladom je tvorba softvérového systému pre štátne inštitúcie, ktorých fungovanie ovplyvňujú zákony. Podstatná zmena zákona zvyčajne spôsobuje úplnú nepoužiteľnosť vytvoreného softvérového systému.

Tento nevyhovujúci stav mal za následok, že vodopádový model, ktorý sa veľmi často používal pri vývoji softvérových systémov, a ktorý nevedel dostatočne rýchlo reagovať na zmenu požiadaviek a podmienok, bolo nutné nahradiť niečím novým – pružnejším, rýchlo reagujúcim na zmeny. Nový model vychádza z pochopenia, že rýchlosť a flexibilita je rovnako dôležitá ako vysoká kvalita a nízka cena, a preto bolo nutné redukovať vodopádový model na štyri základné fázy - špecifikácia požiadaviek, návrh, vytvorenie a odovzdanie produktu.

Na základe spomínaných skutočností sa mnoho firiem rozhodlo aplikovať pri vývoji softvérových systémov napríklad prístup Extreme Programming (XP), o ktorom sa dočítate v kapitole Extrémne programovanie. Ale ako riadiť takéto projekty? Odpoveď na túto otázku dáva metóda Skrumáž (SCRUM).

Čo je to Skrumáž

Je to inkrementálna, iteratívna metóda tvorby softvérových systémov, ktorej základnou myšlienkou je iteratívny vývoj softvéru a tímová práca. Je to množina pravidiel, pomocou ktorých dokážeme optimalizovať celé vývojové prostredie, znížiť množstvo komunikácie a čo je najdôležitejšie, synchronizovať potreby trhu a zákazníka s iteratívnym vývojom jednotlivých prototypov softvéru.

Pri jeho použití využijeme všetky dostupné zdroje, vytvoríme produkt v požadovanej kvalite, a to najdôležitejšie - dodržíme dátum dodania výsledného softvérového systému zákazníkovi. SCRUM je možné použiť v projektoch ľubovoľného zamerania, či už v jeho začiatku alebo počas vývoja, a dokonca je možné jeho nasadenie až keď sa projekt dostane do problémov, čo sú dve z jeho niekoľkých podstatných výhod.

Aplikovanie metódy Skrumáž

Ako bolo povedané SCRUM je založená na tímovej práci a iteračnom vývoji softvérového systému. Najdôležitejším prvkom je tímová práca a správne zostavenie tímu (*Scrum team*). Potrebujeme vytvoriť tím tak, aby sme na každú etapu vybrali práve toho správneho človeka na základe jeho doterajších skúseností.

Tvorca tímu musí mať dobrý prehľad o jednotlivých zamestnancoch spoločnosti a mal by ich poznať aj osobne, pretože nie všetci ľudia sú vhodní na každý projekt a na každú jeho etapu, čo nezáleží len od ich vzdelania a ovládaných technológiách.

Napríklad na špecifikáciu požiadaviek, ktorá samozrejme vyžaduje stretnutia so zákazníkom vyberieme človeka, ktorý je

vnímavý, komunikatívny, dobre vychádza s ľuďmi a má príjemné vystupovanie.

Po zostavení tvorca tímu určí vedúceho tímu, zvolá vytvorený tím a oboznámi ho s projektom. Opíše časovú, finančnú náročnosť a všetko čo sa očakáva od tohto projektu, čo bude jeho úlohou a on ako tvorca tímu je tu na to, aby mu poskytol všetky potrebné zdroje. Netreba byť prekvapený, ak niektorí členovia tímu, ktorí sa ešte nestretli s metódou SCRUM, povedia, že to za daných podmienok nie je možné a odídu.

Dôležitá je aj veľkosť tímu. Aj tu platí pravidlo sedem plus-mínus dva. Teda počet členov tímu by mal byť päť až deväť. Tento počet spĺňa všetky podmienky kladené na vlastnosti tímu a možnosti jeho riadenia a manažmentu. Ak vytvárame väčší softvérový systém, radšej ho rozdelíme na niekoľko pomerne nezávislých častí a vytvoríme príslušný počet tímov ako vytvoriť jeden veľký tím.

Tím musí mať vlastný pracovný priestor, kde budú pracovať všetci členovia tímu a kde budú prebiehať pravidelné stretnutia. Táto metóda je aj o komunikácii medzi jednotlivými členmi tímu a spoločnom riešení problémov, čo by v prípade rozdelenia tímu napríklad do viacerých budov bolo pomerne zložité. Priamy osobný kontakt nenahradí žiadna forma komunikácie, ani tak v poslednej dobe preferovaná video-konferencia ako moderný spôsob komunikácie medzi jednotlivými spolupracovníkmi.

Spoločné umiestnenie jednotlivých členov tímu upevňuje vzťahy v tíme a podporuje aj napríklad praktiku programovania v pároch, ktorú využíva extrémne programovanie. Podobne v týchto miestach by mala byť aj miestnosť určená na spoločné

stretávanie, čo ako uvidíme neskôr je veľmi dôležitý prvok tejto metódy.

Iteratívny vývoj softvéru

Celý projekt (celú prácu potrebnú vykonať na uspokojenie požiadaviek zákazníka - *Backlog*) rozdelíme na tridsaťdňové etapy – iterácie (*Sprint*) [ADM2002]. Samozrejme etapa môže skončiť aj za menej ako tridsať dní.

Pýtate sa prečo práve tridsať dní? Odpoveď je jednoduchá. Tridsať dní je dostatočne dlhá doba na to, aby tím pochopil problematiku, objasnil si požiadavky, navrhol, vytvoril a otestoval riešenie. Ak by to bolo viac ako tridsať dní, pravdepodobnosť, že dôjde k zmene požiadaviek zákazníka alebo prostredia, pre ktoré je systém určený, prípadne k zmene technológie, ktorú systém používa, sa veľmi zvyšuje. Hodnota tridsať dní vychádza zo skúseností firiem využívajúcich metódu SCRUM.

Ak spoločnosť zameriava napríklad na projekty istého špecifického typu môže podľa vlastných skúseností túto hodnotu upraviť. Podľa skúseností veľkého množstva spoločností venujúcich sa tvorbe softvérových systémov v rôznych oblastiach a využívajúcich metódu SCRUM, práve hodnota tridsať dní je tá pravá a univerzálna pre ľubovoľné projekty. Pred začiatkom tridsaťdňovej etapy sa presne definuje práca, ktorú je potrebné vykonať počas tejto etapy (*Sprint Backlog*).

Keďže v oblasti informačných technológií sa podmienky veľmi rýchlo menia, rozdelením projektu na tridsaťdňové etapy sa snaží SCRUM zabezpečiť, že softvérový produkt po dokončení bude spĺňať aktuálne podmienky kladené zo strany

zákazníka a bude vyhovovať aktuálnym podmienkam prostredia kde bude nasadený.

Ak by to bolo menej ako tridsať dní, pravdepodobnosť, že tím stihne vykonať všetky fázy v tejto etape sa znižuje. Pri väčšine softvérových projektov postačuje tri až osem takýchto etáp (iterácií). Každá zo základných fáz tvorby softvérového systému je namapovaná na jednu alebo viac etáp. Tak napríklad, na špecifikáciu požiadaviek zvyčajne postačuje jedna etapa, na analýzu a návrh dve etapy a vývoj softvérového systému tri až päť etáp.

Dva základné piliere metódy Skrumáž

Metóda SCRUM stojí na dvoch základných pilieroch [ADM2002]. Prvým základným pilierom je splnomocnenie tímu. Ak sa tím pustí do práce, najlepšie vie ako má danú úlohu v tejto iterácii produktu splniť a je plne zodpovedný za jej splnenie počas tridsiatich dní. Jedinou komunikáciou medzi tímom a manažmentom by malo byť upozorňovanie manažmentu tímom na problémy stojace mu v ceste, a ktoré by mal manažment odstrániť pre zvýšenie produktivity tímu.

Druhým základným pilierom je prispôsobivosť. Medzi každou etapou – iteráciou sa stretne tím s manažmentom, aby sa dohodli čo sa bude robiť v nasledujúcej iterácii, podľa aktuálnych požiadaviek, podľa toho čo dosiahol tím v predchádzajúcej iterácii a čo je najdôležitejšie pre úspešné ukončenie projektu. Na dodržanie týchto dvoch základných pilierov metódy SCRUM je nutné presne definovať úlohy tímu a manažmentu.

Každý programátor sa určite stretol s nutnosťou vytvárania správ, prezentácii a hlásení o stave projektu, týždenných správ pre nadriadených a podobne. Toto všetko jednak zaberá čas,

ale aj rozptyľuje a prerušuje prácu jednotlivých členov tímu. Nezaberie to len čas potrebný na vytvorenie správy alebo prezentácie, ale aj čas nutný aby sa člen tímu vrátil späť do problematiky, kde predtým skončil.

Takže je ľahké povedať, že je to za chvíľu hotové, ako manažment zvyčajne obhajuje svoje požiadavky na vytvorenie takýchto správ. Hotové to je, ale návrat späť do problému, napríklad pri implementácii, vrátiť sa späť k nápadu ako riešiť daný problém, spomenúť si na myšlienku, býva niekedy zložité a preto vytváranie takýchto správ zaberá omnoho väčší čas ako si manažment zvyčajne uvedomuje. V prípade metódy SCRUM, to však tak nie je.

Základné úlohy tímu a manažmentu

Manažment je tu pre vývojový tím a nie naopak, ako to zvyčajne býva [Beedle2002]. Manažment sa musí zamerať na dve základné otázky. Je tu niečo, s čím by sme mohli vývojovému tímu pomôcť v jeho ceste k úspešnému ukončeniu tejto etapy alebo existuje niečo, čo bráni v pokračovaní práce tímu? Ak také niečo existuje, je prvoradou úlohou manažmentu, to niečo najrýchlejšie odstrániť. Druhou otázkou je, aké vývojové prostredie a aký softvér je najdôležitejší a potrebný pre vývojový tím. Z toho plyní úloha pre manažment čo najrýchlejšie zabezpečiť potreby tímu.

Tím sa musí zamerať na jednu jedinou vec – vytvorenie softvéru a ignorovanie všetkého ostatného. Ako bolo spomínané vyššie, dodržiavanie týchto úloh manažmentom a tímom prispieva k efektivite a produktivite tímu a úspešnému ukončeniu práce v danej etape vývoja softvérového systému.

Ako funguje Skrumáž

Prvé čo nastane po začatí etapy je, že vedúci tímu sa stane hlavným hovorcom tímu – jeho najdôležitejšie úlohy sú vedenie pravidelných denných stretnutí (*Scrum daily meetings*), zaoberanie sa meraním dosiahnutého pokroku a dôležitá je jeho schopnosť robiť rýchle rozhodnutia. V prípade existencie viacerých tímov spolupracujúcich na jednom väčšom projekte, by mal zabezpečovať aj interakciu tímu s ostatnými tímami.

Počas tridsiatich dní sa všetci členovia tímu každý deň stretnú (nemalo by to byť na viac ako tridsať minút, napríklad pri raňajšej káve), aby každý člen odpovedal na tri základné otázky. Čo robil od posledného stretnutia, či má nejaké problémy pri riešení a čo bude robiť do ďalšieho stretnutia [ADM2002].

Je výhodné, ak sa na týchto stretnutiach pasívne zúčastňuje aj niekto z manažmentu, aby vedel odhaliť prípadné prekážky, ktoré ako už bolo spomenuté, by mal manažment odstraňovať. Často je veľmi výhodné, aby sa v prípade potreby na stretnutiach zúčastňoval aj zástupca zákazníka, s ktorým je možné konzultovať prípadné zmeny v špecifikácii a podobne.

Veľmi skoro po začatí etapy sa jednotliví členovia tímu naučia ako riešiť problémy, splniť požiadavky a budú aktívne prispievať k riešeniu jednotlivých problémov. Po ukončení každej fázy, si bude celý tím vedomý, že práca sa nespravila sama, ale za každou vlastnosťou vytváraného softvérového systému bude vidieť tím a jeho jednotlivých členov.

Ak by nezávislý pozorovateľ sledoval činnosť jednotlivých členov tímu, bol by prekvapený, že počas jedného dňa môže jeden človek robiť špecifikáciu požiadaviek, návrh aj implementáciu. Nie je prekvapením, že počas jednej etapy sa

viackrát zmení návrh, jednotliví členovia prichádzajú s novými myšlienkami, na základe ktorých môže dôjsť aj k dosť podstatným zmenám – nové myšlienky prídu a staré myšlienky, cesty a riešenia, budú zavrhnuté.

Vytvorený tím je úplne samostatný a jednotliví členovia tímu sa „zažerú“ do práce a preto sa môže stať, že budú inklinovať k prekročeniu pôvodne stanovených cieľov, čomu by mal zabrániť vedúci tímu [ADM2002]. Po ukončení projektu sa mnohým členom môže stať, že budú mať pocit ako keby stratili zmysel života – až tak veľmi sa môže tím zahĺbiť do svojej úlohy.

Prečo Skrumáž funguje

Ak člen tímu je angažovaný na projekte a jeho nadriadený mu skutočne dôveruje a verí, tým pádom je oslobodený od vytvárania správ a prezentácií o dosiahnutom pokroku pri riešení úloh a môže sa viac času venovať práve tvorbe softvérového systému - čo znižuje potrebu stretávania sa a šetrí čas, ktorý môže venovať tvorivej práci na projekte.

Na druhej strane manažment má menší prehľad o vykonanej práci a stráca možnosť priamej kontroly vykonanej práce, čo môže v prípade neznalosti metódy SCRUM byť mylne považované manažmentom za nevýhodu a z toho plynúcu nechotu púšťať sa do zavedenie tejto metódy.

Samozrejme aj tu existuje istý typ kontroly. Je malá, hlavne nenápadná a väčšinou nepriama [ADM2002]. Na kontrolu musíme nájsť správneho človeka, ktorý okrem poznania riešenej problematiky musí vedieť aj vychádzať s ľuďmi a tak trocha musí byť aj psychológ. Nikoho nepoteší, ak sa mu niekto neustále pozerá cez plece čo robí alebo ho

kritizuje, že nestíha a podobne. Treba vytvoriť príjemné prostredie, kde tento človek ale aj jednotliví členovia tímu majú na seba povzbudzujúci účinok a spolu vyriešia prípadné problémy a chyby.

Úlohou kontroly nie je neustále upozorňovanie na chyby, ale ich tolerancia a pomoc pri ich riešení a odhaľovaní. Každá osoba vybraná do tímu prichádza z určitými vlastnosťami a skúsenosťami z predchádzajúcich projektov. Každý člen tímu má tendenciu spájať si vytváraný systém s podobným systémom, na ktorého vývoji sa podieľal alebo hoci ho len videl. Táto ľudská vlastnosť má za následok znižovanie počtu návrhov na riešenie jednotlivých problémov pri vytváraní softvérového systému a obmedzovanie ich len na také, ktoré jednotliví členovia tímu poznajú, preto je nevyhnutné vždy na začiatku jednotlivých etáp projektu podporenie nových myšlienok a nových prístupov.

Pri metóde SCRUM navrhnutie správneho systému hodnotenia nie je nikdy jednoduché. Je vhodné vytvoriť systém hodnotenia založený na výkone celej skupiny.

Pre manažment je ťažké vzdať sa riadenia projektu a nebyť informovaný o každom detaile a probléme pri jeho vývoji. Tento prístup je riskantný, ale ako sa hovorí, risk je zisk a platí to aj v tomto prípade. Jednotliví členovia tímu pocítia vašu dôveru a budú sa môcť plne sústrediť na riešenie projektu. Je dôležité aby každý člen porozumel všetkým problémom, ktoré sa vyskytli pri jeho riešení a aby pochopil všetky kroky, ktoré viedli k ich odstráneniu, čo do istej miery môže mať za následok obmedzenie veľkosti problému, pre ktoré je táto metóda vhodná.

Záver

Ako každá metóda aj táto prináša určité výhody aj nevýhody. Veľkou výhodou je, že dokážeme pružne reagovať na zmeny požiadaviek zákazníka a podmienok prostredia, pre ktoré je vytváraný softvérový systém určený. Rozdelenie procesu tvorby softvérového systému na tridsaťdňové etapy prinieslo možnosť pružne zareagovať na požadované zmeny prípadne prehodnotiť ďalší postup prác na projekte a tak zabezpečiť aktuálnosť vytváraného softvéru.

Použitím tejto metódy bude vytvorený softvérový systém po jeho dokončení skutočne použiteľný a bude spĺňať aktuálne požiadavky zákazníka.

Manažment sa stará o maximálnu podporu tímu a tím sa orientuje len na tvorbu softvérového systému – to sú dva základné princípy tejto metódy, ktoré zabezpečujú maximálnu efektivitu pri tvorbe softvéru.

Pre manažmenty softvérových firiem zastavajúcich staré prístupy k tvorbe softvérových systémov bude najväčším problémom absencia priamej kontroly postupu prác a zmena vzťahu medzi manažmentom a tímom (zvyčajne mylne považované za nevýhody), ale podľa môjho názoru, každý, kto raz vyskúšal nasadenie metódy SCRUM, to nikdy neolutoval a nevrátil sa k starým metódam riadenia.



Dynamický vývoj systémov

Bc. Radoslav Otipka

Metóda dynamického vývoja systémov (Dynamic System Development Method, DSDM) je jednou z množstva agilných metód vývoja softvéru. V metóde sú presne definované postupy a pravidlá vývoja softvéru. DSDM je definovaná na abstraktnej úrovni a vo svojich procesoch podporuje iné konkrétne agilné prístupy tvorby softvéru, s ktorými spolupracuje napr. extrémne programovanie (XP). Nasledujúci článok sa venuje opisu princípov DSDM a porovnáva DSDM s klasickým (sekvenčným) vývojom softvéru, s XP a s metódou Rational Unified Process.

„Neznášam zmeny v mojom veku, a ešte k tomu zmeny k najhoršiemu. Najhoršie, to je nepekne slovo - a dúfam, že toho sa nedožijete.“

V prvej polovici 90. rokov sa objavila v IT priemysle nová metóda vývoja softvéru *Rapid Application Development* (RAD). RAD bol odlišný od klasických metód vývoja softvéru založených na vodopádovom modeli. RAD metódy vznikli v dôsledku neschopnosti klasických metód, rýchleho prispôbovaniu softvérového produktu často sa meniacim požiadavkám. Ako sa postupne RAD vyvíjal, neexistovala spočiatku všeobecná dohodnutá definícia procesov vývoja RAD. Už v roku 1993 sa objavilo na trhu veľké množstvo prostriedkov podporujúcich RAD. Množstvo spoločností však nepotrebovalo na zlepšenie vývoja softvéru len podporné prostriedky pre RAD, ale aj definovanie nových procesov vývoja samotného softvéru.

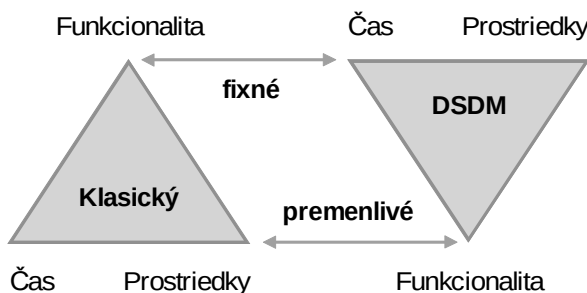
Preto bolo v roku 1994 založená metóda s názvom Dynamický vývoj systémov (*Dynamic Systems Development Method*, DSDM) konzorcium, ktoré si dalo za úlohu navrhnúť RAD metódu vývoja softvéru, ktorá by nedefinovala len prístupy, ale aj samotný proces vývoja softvéru. Prvú verziu

DSDM metódy vydalo konzorcium vo februári 1995. Posledná verzia s poradovým číslom 4.1 bola vydaná v októbri 2001.

Dynamický vývoj systémov a tradičný prístup

Pri tradičnom (sekvenčnom) vývoji softvéru sa najprv zostavil dokument všetkých požiadaviek na softvérový produkt. Na základe týchto požiadaviek sa navrhol a implemetoval produkt. Často sa však stáva, že požiadavky, ktoré zadá používateľ môžu byť nepresné a neúplné, preto sa treba vracieť k počiatočným fázam vývoja softvéru a aplikovať nové požiadavky na návrh, implementáciu, testovanie atď.

V dôsledku zmien požiadaviek dochádza potom k zvyšovaniu potrebných prostriedkov na vývoj a predlžovaniu času vývoja. Všeobecne môžeme povedať, že pri klasickom, sekvenčnom vývoji softvéru je funkcionlita systému pevne stanovená a zdroje a čas potrebný na dokončenie projektu, sa menia [DSDMHistory]. V metóde DSDM je to naopak prostriedky a čas sú fixne stanovené a mení sa funkcionlita softvéru na základe požiadaviek od používateľa [DSDMHistory]. Porovnanie tradičného (klasického) prístupu a DSDM je zobrazené na obr. 6.



Obr. 6: Porovnanie tradičného (klasického) vývoja softvéru a DSDM [DSDMdiff].

Flexibilita požiadaviek má významný dopad na vývoj, riadenie a v konečnom dôsledku aj na prijatie produktu zákazníkom. „Hlavnou myšlienkou DSDM je, že nič nie je perfektne postavené na prvý raz, ale 80 percent funkčnosti navrhovaného systému, môže byť vytvorená za jednu pätinu času z celkového času určeného na vývoj systému“ [DSDMdiff].

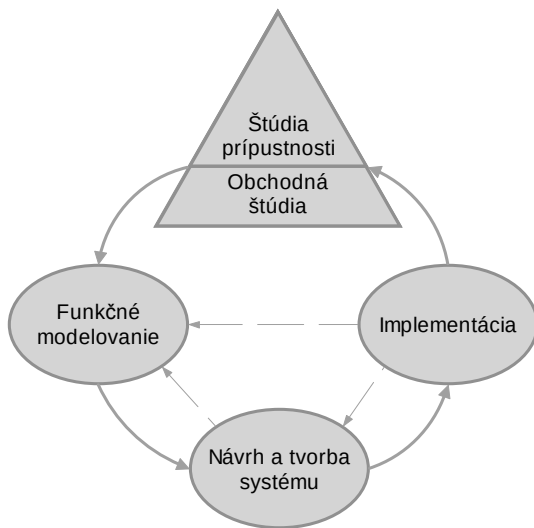
Predstavenie metódy

Ako už bolo spomínané, DSDM je RAD metódou vývoja softvéru. V DSDM je definovaný životný cyklus vývoja systému a tiež deväť princípov, ktoré by mali byť dodržiavané počas jednotlivých fáz životného cyklu. Počas vývoja sa vykonáva kontrola, či sa vyvíja systém podľa požiadaviek používateľa. Vývoj sa riadi tak, aby bol ukončený do stanoveného času a za stanovené prostriedky.

Životný cyklus vývoja softvéru v DSDM je inkrementálny a iteratívny a skladá sa z piatich fáz. Výsledný systém nebude zákazníkovi dodaný naraz, ale v niekoľkých iteráciách. Počas jednotlivých iterácií sú spracovávané jednotlivé požiadavky používateľa. V prvých iteráciách sú zapracované do systému tie najdôležitejšie požiadavky a v ďalších iteráciách tie menej dôležité. Najdôležitejším faktom je, že počas každej z fáz vývoja softvéru musí byť do samotného vývoja aktívne zapojený aj používateľ.

Fázy životného cyklu DSDM sú nasledovné (obr. 7):

- Štúdia prípustnosti
- Obchodná štúdia
- Funkčné modelovanie a Návrh a tvorba systému
- Implementáciu systému



Obr. 7: Životný cyklus v DSDM [DSDMCycle].

Štúdia príпустnosti (ŠP) nachádza vhodnosť použitia DSDM prístupu na vývoj systému a zisťuje, ktoré technické a manažérske postupy je vhodné pri vývoji použiť. Väčšinou netrvá dlhšie ako niekoľko týždňov nanajvýš mesiac.

Obchodná štúdia (OŠ) zahŕňa oblasť celkových aktivít projektu a poskytuje obchodné a technické základy pre ďalšiu prácu. Definujú sa v nej funkčné a nefunkčné požiadavky na najvyššej úrovni a na základe nich je vytvorený model systému na najvyššej úrovni. Už v tejto fáze sa načrtne hrubá architektúra systému. Trvá približne tak dlho ako ŠP.

Funkčné modelovanie (FM) a *Návrh a tvorba systému (NTS)* sú aktivity, ktoré sa vykonávajú počas fáz. Funkčné modelovanie a Návrh a tvorba systému, navzájom na seba nadväzujú. Vykonávané aktivity sú nasledovné:

- Identifikovanie, čo budeme vyrábať
- Rozhodnutie ako a kedy sa to urobí
- Vytvorenie produktu
- Kontrola, či bol produkt vytvorený správne (demonstráciou prototypu, testovaním časti systému, kontrolou dokumentov...)

Vo fáze *Funkčný model* sa robí zjemňovanie aspektov systému, ktorý je postavený na funkčných požiadavkách na najvyššej úrovni.

Hlavný dôraz vo fáze *Návrh a tvorba systému* sa kladie na uistenie sa, že systém spĺňa aj tie najprísnejšie požiadavky a štandardy a či môže byť systém používateľovi dodaný.

Najviac práce je vykonaných práve v týchto dvoch fázach, v nich sú postupne vytvárané a testované prototypy systému. Všetky prototypy sú vyvíjané, tak ako keby to už bol finálny produkt.

Implementáciu systému (IS) transformuje systém z vývojového prostredia do operačného prostredia [DSDMCycle]. Systém sa nasadzuje do prevádzky u zákazníka. Fáza zahrňuje aj školenie používateľov.

V tejto fáze sa vytvára aj dokument o stave projektu. Dokument obsahuje všetky požiadavky, ktoré boli identifikované počas vývoja systému. Tiež obsahuje zoznam tých identifikovaných požiadaviek, ktoré sú už zahrnuté v systéme.

Vo fáze implementácia sa rozhodujeme, ako sa bude ďalej pokračovať vo vývoji systému, t. j. či sa vývoj ukončí alebo bude potrebné vykonať niektorú z predchádzajúcich fáz životného cyklu.

Nastávajú štyri možnosti ďalšieho vývoja systému:

- Všetko bolo vytvorené a dodané a už nie je potrebný ďalší vývoj. Ukončí sa vývoj systému.
- Nové funkčné požiadavky boli objavené počas vývoja a vývoj sa posunie do fázy Obchodná štúdia.
- Málo podstatná časť funkcionality systému bola vypustená počas vývoja, v dôsledku časového ohraničenia. Vývoj sa vráti do fázy Funkčné modelovanie a pridá sa vypustená funkcionality do systému.
- Ak nefunkčné požiadavky nie sú splnené, vývoj sa vráti do fázy Návrh a tvorba systému a pridajú sa nefunkčné požiadavky do systému.

Okrem toho, že DSDM má definovaný životný cyklus vývoja softvéru, má definovaných deväť základných princípov, ktoré by sa mali dodržiavať počas vývoja systémov:

- Aktívna účasť používateľa je nevyhnutná – používateľ by sa mal aktívne zúčastniť vývoja systému. Ak by sa používateľ nezúčastňoval veľmi tesne vývoja, na konci môže mať pocit, že výsledné riešenie je iba podľa predstáv manažérov a vývojárov. Skutočný výsledok vývoja, by mal byť podľa predstavy používateľa.
- Tím musí mať splnomocnenie, aby mohol robiť rozhodnutia – DSDM tímy musia byť zložené z používateľov a tiež samozrejme z vývojárov. Tím by mal byť schopný robiť rozhodnutia, akým spôsobom budú požiadavky upravené alebo zmenené.
- Hlavný dôraz je kladený na rýchle dodanie produktov (softvérových) – Všetka práca DSDM tímu je

sústredená na produkty, ktoré majú byť dodané a schválené používateľom za predpísaný časový úsek. Udržovaním krátkych časových úsekov je vývojový tím nútený sa rozhodnúť, ktorá z aktivít je nevyhnutná a postačujúca na vytvorenie tých správnych produktov.

- Dôraz je kladený na akceptovanie a spracovanie nevyhnutných požiadaviek v požadovanom čase.
- Iteratívny a inkrementálny vývoj je nevyhnutný a prostredníctvom neho vývoj systému konverguje k správne riešeniu. DSDM dovoľuje aby systém rástol inkrementálne. Vývojári majú spätnú väzbu od používateľov, ktorý po každej iterácii skontrolujú výsledok. Vývojári sa môžu uistiť či vývoj prebieha podľa predstáv používateľov.
- Všetky zmeny urobené počas vývoja musia byť vratné – V DSDM sa počas vývoja vykonáva kontrola všetkých produktov. V každom čase vývoja, musí byť produkt v známom stave. Možnosť vrátenie zmien späť je jednou z hlavných charakteristík DSDM. Pri vývoji určitých systémov je jednoduchšie vrátiť zmeny späť a pokračovať iným smerom ako napr. zahodiť celý systém a vyvíjať ho odznova. Takýto postup môže ušetriť čas potrebný na vývoj systému.
- „Zmrazenie“ návrhu systému na takej úrovni, ktorá dovoľuje detailný opis požiadaviek, ktoré sú v prvom rade nevyhnutné pre používateľa. Pri použití metódy DSDM je podstatné, aby sa v prvom rade našli a implementovali do systému najdôležitejšie požiadavky. Menej dôležité požiadavky môžu byť implementované dodatočne.

- Testovanie je vykonávané počas každej fázy životného cyklu. Testovanie nie je oddelené a neberie sa ako samostatná aktivita. Systém je testovaný nielen vývojármi ale používateľmi, na uistenie sa, že vývoj systému prebieha správnym smerom.
- Spolupráca medzi zainteresovanými ľuďmi do projektu je veľmi dôležitá. Medzi zainteresovaných ľudí do projektu nepatrí len vývojový tím, ktorý na projekte pracuje ale aj technická podpora a manažéri zdrojov.

DSDM má implementované aj kontrolovanie postupu vývoja projektu - „Timeboxes“. „Timeboxes“ reprezentujú určité míľniky alebo kontrolné body v projekte, v ktorých musia byť splnené určité požiadavky zákazníka na výsledný systém. Priority požiadaviek, ktoré musia byť implementované v jednotlivých kontrolných bodoch určujú pravidlá, v DSDM sa nazývajú „MoSCoW Rules“ [MoSCoW]. V prvom rade musia byť splnené požiadavky s najvyššou prioritou, to sú také požiadavky, ktoré je nevyhnutné implementovať, aby mohol byť projekt úspešný a ktoré sú pre zákazníka najdôležitejšie. Potom nasledujú tie menej dôležitejšie požiadavky, až nakoniec tie najmenej dôležité. Aká priorita sa prideli jednotlivým požiadavkám, to závisí už na konkrétnom projekte.

Jednotlivé kontroly sa robia v časovom úseku od dvoch do šiestich týždňov. Kontrola sa vykonáva v nasledujúcich troch fázach:

- Prieskum – rýchla kontrola, ktorá zistí či sa tím pri vývoji ubera správnym smerom
- Zjemnenie – vytvorenie záverov, z výsledkov kontroly z fázy Prieskum.

- Konsolidácia – na základe záverov z fázy Zjemnenie, stanovenie ďalšieho postupu, pri vývoji softvéru.

Dynamický vývoj systémov a iné prístupy

V predchádzajúcich častiach sme opísali základné charakteristiky metódy DSDM.

DSDM je nezávislá metóda vývoja softvéru a môže byť použitá pri vývoji v súlade aj s inými prístupmi alebo metódami napr. XP (Extrémne programovanie) alebo RUP (*Rational Unified Process*). Základným rozdielom medzi RUP, XP a DSDM je v úrovni abstrakcie. Metóda DSDM je definovaná na vyššej úrovni abstrakcie ako XP alebo RUP.

Dynamický vývoj a Extrémne programovanie

Množstvo ľudí sa domnieva, že DSDM a XP sú navzájom konkurujúce metódy vývoja softvéru. V prvom rade musíme povedať, že XP neberieme ako metódu skôr ako prístup k vývoju softvéru. XP a DSDM sa môžu používať spoločne a navzájom sa môžu veľmi vhodne dopĺňať.

Spoločné vlastnosti XP a DSDM:

- Zákazník (používateľ) je zainteresovaný do vývoja produktu (je súčasťou tímu).
- Časté testovanie. (XP požaduje aby testovacie prípady boli napísané ešte pre začatím kódovania) Rozdiel je však v úrovni formalizácie testovacích prípadov, ktorá je väčšia pri DSDM.
- Inkrementálny spôsob vývoja
- Mechanizmus rýchlej odozvy na zmeny požiadaviek
- Rýchle dodanie systému

- Zameranie sa na hlavne to čo musí byť teraz vykonané
- Malé tímy
- Vývojári v tíme musia byť všestranní (mali by byť dobrí návrhári, programátori a testéri)

XP, neurčuje presnú zodpovednosť za jednotlivé činnosti, preto môže byť použité hlavne pri vývoji malých softvérových systémov. Kým DSDM je tiež dobre použiteľné na malé projekty, ale súčasne jednotlivé roly a zodpovednosti definované v DSDM sa dajú jednoducho škálovať na veľké projekty.

DSDM a XP sa tiež môžu navzájom dopĺňať. XP sa nezaobera podrobnosťami riadenia projektu a nezahŕňa kompletnú podporu životného cyklu vývoja softvéru ale veľmi dobre podporuje samotné programovacie prístupy (napr. párové programovanie). Na druhej strane DSDM zase menej podporuje konkrétne programovacie prístupy, ale viacej riadenie („timeboxes“, životný cyklus DSDM, princípy DSDM). Kombináciou oboch môžeme teoreticky dostať metódu s dobrou podporou riadenia a dobrými programovacími prístupmi.

DSDM a XP sa snažia riešiť ten istý problém: vyvinúť dobrý systém v krátkom časovom úseku. XP sa hlavne zameriava na samotný akt programovania, ktorý je len zľahka načrtnutý v DSDM. DSDM poskytuje hlavne metódu riadenia projektu, do ktorej môže byť XP dobre zahrnuté [MoSCoW].

Dynamický vývoj a *Rational Unified Process*

V nasledujúcej časti sa budem venovať stručnému porovnaniu DSDM a RUP.

Spoločné črty DSDM a RUP:

- DSDM poskytuje deväť základných princípov, ktoré sú tiež obsiahnuté aj v metóde RUP.
- Obidve metódy poskytujú inkrementálne a iteratívne životné cykly a podporu rýchleho dodania softvéru.
- RUP poskytuje detailnú definíciu modelu procesov, ktoré sa majú vykonávať. Tento model môže byť uvažovaný ako implementácia DSDM metódy.
- Obidve metódy jasne definujú zodpovednosti za jednotlivé aktivity. DSDM má navyše definíciu rolí ako Používateľ Veľvyslanec a Vizionár. RUP má podrobnejšie opísaný vývojový tím a roly, zodpovednosti jednotlivých členov vývojového tímu.
- DSDM poskytuje množstvo prístupov na najvyššej úrovni. RUP tiež poskytuje množstvo prístupov a detailne opisuje ako jednotlivé prístupy použiť, to už DSDM nerobí.
- Obidve metódy poskytujú smernice (metodiky) na rôzne aspekty procesu vývoja. RUP zahŕňa kompletne metodiky na modelovanie v UML, na manažment požiadaviek, vývoj a na testovanie. DSDM obsahuje metodiku pre podporné porady, časové škálovanie projektu a kontrolu implementácie používateľských požiadaviek.
- RUP obsahuje veľké množstvo šablón. DSDM úmyselne nepoužíva šablóny, aby sa nestratil širší nadhľad na vývoj systému.
- RUP poskytuje komplexnú dokumentáciu ako jednotlivé podporné vývojové prostriedky (napr. *Rational Rose*) integrovať s procesmi. DSDM

poskytuje poradenstvo aká vhodná množina prostriedkov sa má použiť pre jednotlivé procesy. Pri každom type projektov môže byť vhodné použitie iných prostriedkov.

DSDM a RUP zdieľajú spoločný prístup k vývoju softvéru. „Obidva procesy používajú iteratívny prístup k vývoju a hlavne sa zameriavajú na vývoj softvéru, v ktorom dominujú používateľské požiadavky“ [Tuffs1999]. Výhodou DSDM je, že používa všeobecné pravidlá (9 princípov DSDM, „*MoSCoW Rules*“), ktoré sa dajú elegantne napasovať (prispôbiť) na konkrétne prostredie alebo projekt. RUP zase poskytuje detailné metodiky v špecifických oblastiach a zároveň poskytuje detailný opis nielen toho čo sa má vykonať, ale aj ako sa to má vykonať.

„RUP je veľmi dobre a podrobne spracovaná metóda vývoja softvéru a obsahuje tiež v sebe princípy DSDM. RUP je vo svojej podstate detailná implementácia DSDM“ [Tuffs1999].

Záver

DSDM je orientované na rýchle dodanie softvéru. Zahŕňa v sebe riadenie, ktoré určuje prioritu s akou majú byť splnené jednotlivé požiadavky používateľa a za aký čas. Je to metóda, na rozdiel napr. od XP a má definované procesy a zodpovednosti za vykonávanie jednotlivých činností.

Najlepšie definované zodpovednosti pre vývojárov má RUP, kladie asi aj najmenšie požiadavky na všestrannosť vývojára. Najlepšie schopnosti by mali mať vývojári používajúci metódu XP. DSDM je to niekde medzi týmito dvoma krajnými hranicami. Požiadavky na vývojára sú väčšie ako je to pre RUP, pretože zodpovednosti nie sú v DSDM, až

na takej úrovni podrobnosti ako v metóde RUP, ale vývojári v DSDM určite nemusia byť až tak všestranne zručný ako je to potrebné pri XP.

V DSDM tvoria zákazníci súčasť vývojového tímu, podobne ako je to aj v iných agilných metódach a prístupoch.

DSDM je všeobecne použiteľné na malé projekty ale dá sa veľmi vhodne naškálovať aj na veľké projekty. DSDM môže byť použité spolu s inými prístupmi vývoja softvéru. Príkladom použitia metódy DSDM na vývoj veľkých informačných systémov (IS) je vytvorenie IS pre firmu Fujitsu's European Repair Centre's [FastSolution] alebo pre spoločnosť Ladok [Ladok].



Agilné Modelovanie

Bc. Jozef Krššák

Agilné modelovanie (Agile modeling, AM) predstavuje metódu na efektívne modelovanie a dokumentovanie softvérovo založených systémov. Ide o zbierku hodnôt, princípov a praxe na modelovanie softvéru, ktorú možno aplikovať na vývoj softvéru efektívnym a ľahkým spôsobom. Jej základnými hodnotami sú komunikácia, jednoduchosť, odozva, guráž a pokora. Snahou AM je vytvárať čo najjednoduchšie riešenia, a pritom kvalitne a promptne reagovať na požiadavky zákazníka.

„Prísaha môže posilniť roztrasené srdce. Alebo ho zlomiť. Nedívajte sa príliš ďaleko dopredu!“

Agilné modelovanie (AM) je prakticky orientovaná metóda na efektívne modelovanie a dokumentáciu softvérovo založených systémov. Je to zbierka hodnôt, princípov a praktických rád na modelovanie softvéru, ktorá môže byť aplikovaná na vývoj softvéru efektívnym a ľahkým spôsobom [Ronin].

AM nie je nariaďujúci proces, tzn. nedefinuje detailne procedúry ako vytvoriť daný druh modelu. Namiesto toho ponúka radu ako byť efektívnym modelárom.

AM uznáva hodnoty, ktoré prezentuje aj *Agile Alliance*, pretože sú kľúčom k efektivite. Sú to:

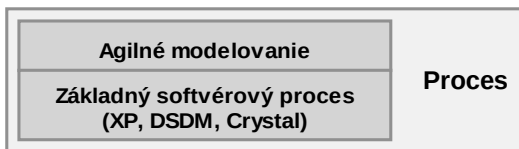
- Jednotlivci a vzájomné ovplyvňovanie cez procesy a nástroje
- Pracujúci softvér cez úplnú dokumentáciu
- Spolupráca zákazníka cez kontraktové jednanie
- Reagovanie na zmenu cez nasledovanie plánu

Čo (nie) je Agilné modelovanie

Ak sa popisuje rozsah niečoho, mali by sme popísať oba pohľady, teda nielen čo to je ale aj čo to nie je. Nasledujúce body preto popisujú tento rozsah.

AM je postoj nie nariaďujúci proces. Zahŕňa zbierku hodnôt, ktorej sa agilný modelár drží, princípov, ktorým verí a praktických rád, ktoré aplikuje. Popisuje štýl modelovania, pomocou ktorého sa zlepšia vlastnosti softvéru a vyhne sa nerealistickým očakávaniam.

AM je dodatok k existujúcim metódam, nie kompletná metóda. Primárne sa sústreďuje na modelovanie, sekundárne na dokumentáciu. Prístup AM môžu byť použité na pozdvihnutie úsilia a vytvorenie zjednocovacieho modelu v projektoch, kde sa používa XP (Extrémne programovanie), DSDM (Dynamický vývoj systémov) a Crystal (Priezračné techniky) (pozri obr. 8).



Obr. 8: Vzťah AM k ostatným agilným metódam.

AM je cesta ako spolu a účinne vyhovieť potrebám projektových investorov. Agilní vývojári pracujú ako tím spolu s projektovými investormi. Tu neplatí „Ja“.

AM je efektívny. Dokumentácia a model sa vytvára vtedy, keď je jasne stanovený účel a publikum.

AM pracuje v praxi, nie je to akademická teória. Cieľom je popísať prístupy na modelovanie systému efektívnym

spôsobom, ktoré sú ako dostatočné tak aj výkonné pre danú úlohu.

AM nie je magická strieborná guľička, ktorá nám vyrieši všetky vývojové problémy. Je to účinný prístup na zlepšenie softvérového vývojového úsilia. Presne toto, nič viac. Ak si ale zoberiete k srdcu hodnoty, princípy a praktiky AM, pravdepodobne dosiahnete zlepšenie efektivity.

AM je pre priemerného vývojára, ale nie je to náhrada za schopných ľudí. Najťažšie na AM je, že núti ľudí, aby sa učili široký rozsah modelovacích prístupov, čo je dlhá a kontinuálna aktivita. Najprv to môže byť ťažké, ale prístupy je možné sa učiť postupom času a tak to jednoducho zvládnuť.

AM nie je útok na dokumentáciu. Agilná dokumentácia je čo najviac jednoduchá, minimálna, má prím určenie na vyvíjaný systém a definované publikum, ktoré jej potrebuje porozumieť.

AM nie je útok na CASE nástroje. Agilný modelár používa nástroje, ktoré mu prinášajú pozitívne hodnoty, snaží sa ale používať najjednoduchšie nástroje, ktoré vykonajú potrebnú prácu.

AM nie je pre každého. Nie vždy existujú vhodné podmienky, nie každému to vyhovuje.

Hodnoty AM

Hodnoty AM zahŕňajú hodnoty Extrémneho Programovania: *komunikácia, jednoduchosť, odozva a guráž* a navyše rozširujú ich o piatu hodnotu a tou je *pokora (humility)*. Prvé štyri opisuje kapitola o XP.

Pokora. Najlepší vývojári majú pokoru priznať, že nepoznajú všetko. Potom ich partneri vývojári, zákazníci

a vlastne všetci projektoví investori majú oblasť ich vlastnej kvalifikácie a majú odvahu pridať sa k projektu. Každý, kto je zapojený do projektu, má potom ekvivalentnú hodnotu a zaslúži si, aby sa s ním zaobchádzalo s rešpektom.

Princípy Agilného modelovania

AM definuje zbierku koreňových a doplňujúcich princípov. Časť z nich bola prevzatá z XP ale pre AM sú prezentované v odlišnom svetle.

Je potrebné si osvojiť jednoduchosť a naučiť sa predpokladať, že najjednoduchšie riešenie je najlepšie. Doplňujúce rozšírenia systému nemusia byť zahrnuté okamžite, už dnes.

Byť pripravený na prijatie zmien, pretože ľudia v tíme sa môžu zmeniť ako aj požiadavky investorov.

Budúca námaha je druhotným cieľom, preto môžete posudzovať poruchu v systéme aj keď tento systém ste práve doručili zákazníkovi

Zmena sa vykonáva inkrementálne. Dôležité je uvedomiť si, že na prvýkrát nedostanete to pravé. Model nemusí obsahovať každý detail, na to potrebujete dostatok času.

Maximalizácia spoluúčasti investora, pretože ak niekto investuje peniaze, tak tie by mali byť investované do jeho produktu a nie byť rozhádzané v tíme.

Model má splniť svoj účel. Prvým krokom je poznať publikum, pre ktoré je model vyvíjaný a potom určiť, nakoľko má byť presný. Inak sa vývojár zbytočne obáva, či je model dostatočne alebo až príliš presný.

Používanie rôznorodých modelov, pretože niektoré sa hodia viac a niektoré menej na daný problém. Preto je

nevhodné používate všetky modely pre daný systém ale napr. iba podmnožinu z nich.

Kvalitná práca je dôležitá, pretože nik nemá rád nedbalú prácu a užívateľovi sa páčiť nebude, pretože výsledný systém nesplní jeho očakávania

Rýchla odozva a byť blízko k zákazníkovi, porozumieť jeho požiadavkám a reagovať na ne.

Softvér je hlavným a primárnym cieľom, dokumentácia je sekundárny cieľ.

Kedy je model agilným

Je potrebné porozumieť rozdielu medzi modelom a agilným modelom. Agilný model (Am) je model, ktorý je tak akurát dostatočne dobrý (kvalitný) a to je vtedy, keď má nasledujúce vlastnosti:

- *Am* spĺňa svoj účel,
- *Am* je pochopiteľný pre publikom, pre ktoré je určený,
- *Am* je dostatočne presný, nemusí byť presný na 100%,
- *Am* je dostatočne konzistentný, nemusí byť perfektný,
- *Am* je dostatočne detailný, pretože ani automapa neukazuje každý dom,
- *Am* prináša pozitívnu hodnotu (investícia do nejakého architektonického modelu prináša taký zisk, ktorý zodpovedá, v určenom pomere, tejto investícii, alebo investícia nie je zbytočne vysoká na dosiahnutie požadovaného výsledku),
- *Am* je čo najjednoduchší.

Kedy agilne (ne)modelujete

Najväčším problémom agilného vývoja je keď vývojári reklamujú túto metódu, pretože sa oboznámili iba z jej časťou alebo sa s ňou neoboznámili poriadne. Preto je dobré definovať, v ktorom prípade agilne modelujete a kedy nie.

Keď agilne modelujete, tak:

- Vaši zákazníci / používatelia sa aktívne zúčastňujú na vašich požiadavkách (na nich) a/alebo analýze úsilia pri modelovaní,
- pracujete na projektových požiadavkách, ktoré majú najvyššiu prioritu a potom až na požiadavkách z nižšou prioritou,
- pri modelovaní používate interagujúci a inkrementálny prístup,
- primárne sa sústreďujete na vývoj softvéru, nie na dokumentáciu alebo samotný model,
- vo Vašom tíme je vítaný vstup od každého člena,
- aktívne sa snažíte robiť veci čo najjednoduchšie a používate najjednoduchšie dostupné nástroje a vytvárate najjednoduchší model, ktorá spĺňa svoju úlohu,
- zákazníci / obchodníci robia obchodné rozhodnutia, vývojári robia technické rozhodnutia,
- uznávaný je obsah vašich modelov, nie ich formát / reprezentácia.

Naopak, keď agilne nemodelujete, vtedy:

- Vaším cieľom je dokumentácia k produktu,

- používate CASE nástroj aby špecifikoval architektúru a / alebo návrh vášho softvérového produktu, ale nepoužívate špecifikáciu na vytvorenie časti alebo celého produktu,
- Vaši zákazníci / používatelia majú limitovanú možnosť spoluúčasti na úsilí Vášho tímu,
- Váš pohľad v čase je zaostrený na jeden model,
- pracujete pomocou pozastavenia jedného alebo viacerých Vašich modelov - používate sériový prístup,
- odovzdávate modely a / alebo dokumentáciu ďalšiemu tímu, ktorý vyvíja systém ďalej – inak povedané dávate „ruky preč“ od práce pri sériovom prístupe.

Aj keď nemôžete byť agilným modelárom kvôli podmienkam okolia, ešte stále môžete použiť časť praxe AM vo vašom projekte.

Záver

Agilné modelovanie nie je všeliek, ktorý pracuje v každej situácii a pre každého, aj keď sú splnené všetky podmienky. Ak ale platia nasledujúce faktory, AM môže byť vo vašej organizácii účinné:

- Váš prístup k vývoju softvéru je agilný / živý,
- pracujete interakčne a inkrementálne, pretože efektívna komunikácia a odozva vyžaduje práve takýto prístup k vývoju,
- Váš projekt je prieskumný a preto požiadavky zákazníkov sú neisté a nestále
- Vaším cieľom je vyvinúť softvér,

- máte aktívnu podporu od investora a viete presne, kto je vaším projektovým investorom,
- vyžadujete zodpovedných a motivovaných vývojárov,
- ľudia v tíme pracujú blízko seba a spolupracujú medzi sebou.

Čo urobiť ak jeden alebo viac faktorov neplatí? Môžete zmeniť Vašu situáciu, vyjednávať s Vaším vedením. Akonáhle sa nachádzate v najlepšej možnej situácii, máte tieto možnosti:

- Čiastočne prijať AM a keď Vaše vedenie uvidí výsledky, AM sa bude môcť u Vás adaptovať.
- Vzdať sa použitia AM, pretože pre Vašu organizáciu to nie je vhodné riešenie.
- Hľadať si nové zamestnanie, kde bude možné AM používať.

Ak ale u Vás platia nasledujúce faktory, použitie AM môže byť nevhodné:

- jeden alebo niekoľko vyššie spomenutých faktorov AM nie je k dispozícii,
- Vaša organizačná štruktúra sa orientuje na nariadenie procesy a máte mnoho organizácií ktoré sa nezaujímajú o agilný / živý prístup,
- Váš tím je veľký a priestorovo rozložený tím a preto nie je možné rýchlo a často komunikovať a pracovať v nejakej *shared work room*.

Taktiež aplikovanie AM na vývoj *life-critical* systémov (systémy na kontrolu leteckej dopravy, monitorovacie systémy pre pacientov v nemocniciach) nie je najvhodnejšie riešenie.



Namiesto záveru

Cieľom softvérové inžinierstva je redukovať riziko zlyhania projektu a systematickým spôsobom zefektívňovať proces tvorby softvéru. Proces vývoja softvéru je komplexná činnosť, na ktorú sa dá pozerat' z rôznych uhlov pohľadu.

Agilné metódy tvorby softvéru predstavujú alternatívu ku klasickým metódam. Zhrňme si ešte ich základné aspekty a myšlienky.

Výraznou črtou prístupu *Odl'ahčený vývoj* je orientácia na požiadavky zákazníka, ktorému sa podriaďuje celý vývoj. Snaží sa adaptívne prispôbovať meniacim sa požiadavkám trhu. Proces tvorby softvéru zefektívňuje hlavne redukovaním nepotrebných a nadbytočných zdrojov, ktoré tvorí hlavne priebežná dokumentácia. Predpokladá vytvorenie takého tímu spolupracovníkov, ktorí sa môžu na seba navzájom spoľahnúť v kritických situáciách.

Ponúka dva rôzne prístupy: prvým je zabudovanie testov priamo do procesu vývoja, ktoré sledujú, či zmeny v kóde neovplyvnia výslednú funkčnosť. Druhým prístupom umožňujúcim vykonávať zmeny aj neskoro vo vývoji je refactoring, alebo vylepšenie návrhu existujúceho softvéru kontrolovaným a rýchlym spôsobom.

Odl'ahčený vývoj sa klasickým prístupom podobá hlavne v oblasti strategického rozhodovania počas vývoja softvéru. Agilným prístupom sa približuje v dôraze na tímovú spoluprácu, redukcii nadbytočných aktivít prístupom k požiadavkám zákazníka.

Pri prístupe *Adaptívny vývoj softvéru* sú stredobodom pozornosti ľudia a nie vývoj produktu. Stavia sa na tom, že dobrá spolupráca tímov vedie k najlepšiemu produktu. Manažéri sa snažia, aby vývojové prostredie a pracovné postupy čo najviac prispôsobili projektovým nárokom a ideám modelu Adaptívneho vývoja softvéru. Vývojový tím sa potom stará o čo najlepšiu adaptáciu produktu vzhľadom na požiadavky. Adaptívny vývoj softvéru vychádza z toho, že všeobecné vzťahy medzi členmi tímu pozitívne stimulujú inováciu a kreativitu. Adaptácia tímov na zmeny, ktoré sa v projekte vyskytnú, sa potom uskutočňuje prostredníctvom komunikácie, spolupráce a súťaživosti napríklad pri spoločných diskusiách členov tímu.

Vývoj softvéru pomocou črt (FDD) je prístup, ktorý umožňuje presné plánovanie, hodnotenie výsledkov, preukazovanie funkčných výsledkov a meranie progresu s vysokou presnosťou. FDD je modelom riadený postup vývoja softvéru používajúci krátke cykly. Ide o inkrementálny (prírastkový) vývoj, ktorý začína s celkovým modelom systému a zoznamom črt, za ktorým nasleduje séria dvojtýždňových cyklov návrhu a implementácie jednotlivých črt. Črty sú malé ciele, resp. výsledky, ktoré sú „užitočné v očiach zákazníka“.

Použitie FDD má výhodné vlastnosti pre vývojárov, manažérov aj zákazníkov. Vývojári dosahujú každé dva týždne niečo hotové, manažéri vedia dobre napláňovať projekt, znižuje sa riziko neúspechu projektu, pretože má konkrétne funkčné medzivýsledky. Pre zákazníkov je proces riadený prístupom FDD dobre viditeľný vďaka krátkym vývojovým cyklom.

Priezračné techniky sú prístupom k vývoju softvéru, ktorý sa zameriava v prvom rade na ľudí aj ako jednotlivcov, ale hlavne ako tím. Ponúka rôzne stratégie na zefektívnenie a

optimalizáciu tímovej práce. Odporúča zvýšenie rozsahu komunikácie v rámci tímu, vytvorenie tzv. tímu jednej mysle.

Všetky odporúčania a stratégie sa prispôbujú konkrétnemu projektu a jeho požiadavkám. Pri vývoji sa kladie dôraz na ľahkosť a priezračnosť, radšej sa odporúča vytvárať jednoduché a ľahko čitateľné funkcie ako vytvárať efektívne a zložité funkcie, ku ktorým treba vytvárať veľa dokumentácie.

Dokumentácia sa vytvára iba pre zákazníka, v rámci tímu je tvorba dokumentácie minimálna, pretože sa vytvára len to čo je nevyhnutné a potrebné pre projekt. Veľmi dôležitá je aj spolupráca so zákazníkom, ktorý je vlastne členom tímu a spolupracuje na špecifikácii projektu.

Extrémne programovanie je prístup, ktorý pokrýva hlavne návrh a implementáciu (ako aj ďalšie vylepšovanie projektu). Vývoj iteruje v cykloch dlhých asi dva až tri týždne. V prístupe sa testuje pravidelne, najskôr sa napíšu testy až potom sa programuje. Test je istou formou špecifikácie. Extrémne programovanie je určené pre malé tímy 2- 10 ľudí, stavia na dobrých medziludských vzťahoch. Tesne spolupracuje so zákazníkom, pretože jeden človek od zákazníka je členom tímu. Veľmi zaujímavé je využitie párového programovania.

Veľkou výhodou metódy Skrumáž je schopnosť pružne reagovať na zmeny požiadaviek zákazníka a podmienok prostredia, pre ktoré je vytváraný softvérový systém určený. Základom je inkrementálny vývoj, teda rozdelenie procesu tvorby softvérového systému na tridsaťdňové etapy. Manažment sa stará o maximálnu podporu tímu a tím sa orientuje len na tvorbu softvérového systému. Pre manažmenty softvérových firiem zastavajúcich staré prístupy k tvorbe softvérových systémov bude najväčším problémom absencia

priamej kontroly postupu prác a zmena vzťahu medzi manažmentom a tímom.

Dynamický vývoj systémov je metóda orientovaná na riadenie. Prostriedky a čas určené na vývoj produktu sú fixne stanovené a mení sa funkcionality vytváraného softvéru na základe požiadaviek od používateľa. Metóda je postavená na myšlienke, že nič nie je perfektne postavené na prvý raz, ale 80 percent funkčnosti navrhovaného systému, môže byť vytvorená za jednu pätinu času z celkového času určeného na vývoj systému. Životný cyklus vývoja softvéru v metóde je inkrementálny a iteratívny. Zákazník má možnosť aktívne zúčastniť sa vývoja systému. Vývojári softvérového systému by mali byť všestranne zručný. Metóda, kladie dôraz na komunikáciu medzi členmi tímu a definuje mechanizmus kontroly stavu projektu pomocou kontrolných bodov.

Agilné modelovanie predstavuje metódu na efektívne modelovanie a dokumentovanie softvérových systémov. Ide o zbierku hodnôt, princípov a praxe na modelovanie softvéru, ktorú možno aplikovať na vývoj softvéru efektívnym a ľahkým spôsobom. Jej základnými hodnotami sú komunikácia, jednoduchosť, odozva, guráž a pokora. Snahou metódy je vytvárať čo najjednoduchšie riešenia, a pritom kvalitne a promptne reagovať na požiadavky zákazníka.

Literatúra

- [ADM2002] ADM, INC.: *Home of Scrum*.
<http://www.controlchaos.com>, (26.3.2002).
- [Agile2001] AGILE ALLIANCE: Manifesto for Agile Software Development. 2001.
<http://www.agilemanifesto.org/>. (27.5.2002).
- [Beedle2002] BEEDLE, M. et al.: *SCRUM: An extension pattern language for hyperproductive software development*. 18 s.
<http://www.controlchaos.com/scrum.pdf>.
121 401 B. (26.3.2002).
- [Bielik2000] BIELIKOVÁ, M.: *Softvérové inžinierstvo. Princípy a manažment*. Bratislava: Vydavateľstvo FEI STU, 2000.
- [Boehm2002] BOEHM, B.: *Get Ready for Agile Methods, with Care*.
<http://dlib.computer.org/co/books/co2002/pdf/r1064.pdf>. (25.5.2002).
- [Brooks1995] BROOKS, FREDERICK P. JR.: *The Mythical Man Month – Essays on Software Engineering*. Anniversary Edition. Addison Wesley, 1995.
- [Coad1999] COAD, P., LEFEBVRE, E., DE LUCA, J.: *Java Modeling in Color with UML*. Prentice Hall, 1999.
<http://www.togethersoft.com/files/services/jm/cu/chapter6.pdf>, 20 p. (26.3.2002).

- [Coad2000] COAD LETTER: *Feature Driven Development and Extreme Programming*. Júl, 2000.
<http://www.togethercommunity.com/coad-letter/Coad-Letter-0070.html> (26.3.2002).
- [Cockburn1] COCKBURN, A.: *Crystal Methodologies: Philosophy*.
<http://www.crystallmethodologies.org/philosophy.html>, (26.3.2002).
- [Cockburn2] COCKBURN, A.: *Humans and Technology's Manifesto for software development*. Humans and Technology.
<http://members.aol.com/acockburn/manifesto.html>, (26.3.2002).
- [Cockburn3] COCKBURN, A.: *Crystal "Clear": A human-powered software development methodology for small teams*,
<http://members.aol.com/humansandt/crystal/clear>, (26.3.2002).
- [Crystal2000] *Welcome To Crystal Discussions*.
<http://www.crystallmethodologies.org/cgi/wiki?WelcomeToCrystalDiscussions>, (26.3.2002).
- [DSDMCycle] *The DSDM Lifecycle*.
<http://www.dsdm.org/en/about/lifecycle.asp>, (20. 5. 2002).
- [DSDMdiff] *Why is DSDM different*.
<http://www.dsdm.org/en/about/overview.asp>, (20. 5. 2002).
- [DSDMHistory] *The History of DSDM*,
<http://www.dsdm.org/en/about/history.asp>, (20. 5. 2002).

-
- [DSDMvsXP] *DSDM and Extreme Programming.*
<http://www.dsdm.org/kss/details.asp?fileid=66>, (20. 5. 2002).
- [FastSolution] *Fujitsu Finds the Right Solution Fast.*
<http://www.dsdm.org/kss/details.asp?fileid=140>, (20. 5. 2002).
- [Frailey2002] FRAILEY, D.: *Reducing Cycle Time, Software Development Magazine.* August, 2002.
- [IA2001] INFORMATION ARCHITECTS, Inc.: *Adaptive Software Development.* 1997-2001.
<http://www.adaptivesd.com/serv05.htm>, (13.3.2002).
- [Jones1990] JONES, D., ROOS, D., WOMACK, J.: *The Machine That Changed the World: The Story of Lean Production.* New York: Rawson and Associates. ISBN 08-925-6350-8. 1990.
- [Ladok] *Ladok Case Study.*
<http://www.dsdm.org/kss/details.asp?fileid=212>, (20. 5. 2002).
- [Lean2001] POPPENDIECK.LLC : *Article of Lean Programming.* Software Development Magazine. Jún 2001.
<http://www.poppendieck.com/programming.htm>, (26.3.2002).
- [Lean2002] POPPENDIECK.LLC : *Lean Development.*
<http://www.poppendieck.com/development.htm>, (26.3.2002).
- [Lugsch2000] LUGSCH, J., NOWICKI, M.: *Adaptive Software Development Overview.* University of

- Technology Darmstadt, Nemecko. 5 s. 2000.
<http://www.darmstadt.gmd.de/concert/people/schuemmePrivate/coopse2000/Ausarbeitungen/asd.pdf>. 45 354 B. (13.3.2002).
- [MoSCoW] *MoSCoW RULES*.
<http://www.dsdm.org/en/about/moscow.asp>,
(20. 5. 2002).
- [Rokham2001] ROKHAM, M.: *Adaptive Software Development*. CS511, Group 2. 2001.
- [Ronin2002] RONIN INTERANTIONAL: *The Official Agile Modeling (AM) Site*.
<http://www.agilemodeling.com>, (36.3.2002).
- [SG1994] THE STANDISH GROUP INTERNATIONAL:
Charting the Seas of Information Technology – Chaos, 1994.
- [Timeboxes] *Timeboxes*.
<http://www.dsdm.org/en/about/timeboxes.asp>,
(20. 5. 2002).
- [Tuffs1999] TUFFS, D. ET AL.: *Inter-operability of DSDM with the Rational Unified Process*. Október 1999.
<http://www.dsdm.org/kss/details.asp?fileid=4>,
(20. 5. 2002).
- [Underlying] *The Underlying Principles*.
<http://www.dsdm.org/en/about/principles.asp>,
(20. 5. 2002).

Register

A

adaptívny konceptuálny
model27
adaptívny model
manažmentu.....26
adaptívny model vývoja..24
Adaptívny vývoj softvéru
.....21, 22, 116
agilita68
Agilná aliancia.....4
Agilné metódy4, 50
Agilné modelovanie.....105,
106, 112, 118
agilný model110

B

Backlog.....80

C

črty39, 40, 45, 116

D

doručiteľnosť57
Dynamický vývoj systémov
.....90, 107, 118

E

Extrémne programovanie
.....63, 64, 70, 77, 79, 98,
107, 117

F

finančný softvérový trh...22
funkčné modelovanie.....93
funkčný model94

G

guráž65, 108

H

hacking.....43

I

implementácia systému...94
individualita51
Internet.....24
interoperabilita.....4
iteratívny vývoj softvéru.78
iteratívny vývojový model
.....25

J

jeden tím (Whole Team).65
jednoduchosť65, 108

jednoduchý návrh (Simple Design)..... 66

K

klasické metódy..... 3

komplexné adaptívne systémy..... 27

komunikácia.. 5, 17, 51, 55, 57, 60, 65, 67, 70, 105, 108, 118

L

ľahkosť..... 52

Lean Manufacturing..... 8

ľudia.. 4, 32, 49, 51, 52, 53, 54, 66, 69, 109, 113, 116

M

malé vývojové cykly (Small Releases)..... 66

manažment organizácií .. 24

Manažment totálnej kvality 9

Manifest agilného vývoja softvéru..... 4

míľnik..... 46

N

návrh a tvorba systému . 92, 93, 94, 95

neustála integrácia (Continuous Integration) 67

O

obchodná štúdia..... 93

Odl'ahčený vývoj. 7, 10, 11, 14, 17, 115

odozva 108

optimalizácia 15, 23, 24

optimálnosť 52

P

plán vydávania verzií 60

plánovanie hrou (Planning Game)..... 66

pokora..... 108

Priezračné techniky 50, 116

blue..... 54

clear..... 53, 120

maroon 54

orange..... 54

violet..... 54

yellow..... 53

priezračnosť 52

prispôsobivosť.... 50, 52, 81

programovanie v pároch (Pair Programming).... 66

R

Rapid Application

Development, RAD... 23, 90

red..... 54

refactoring..... 65, 69, 115

RUP..... 98, 99, 100, 101

S

Scrum daily meetings	83
Scrum team	78
Skrumáž	77
softvérová kríza	2
softvérové inžinierstvo	2
spätná väzba	65
splnomocnenie tímu	81
Spoločné vlastníenie kódu (Collective Code Ownership)	67
spôsob písania kódu (Coding Standard)	67
Sprint	80
štúdia prípustnosti	93

T

tímová práca	64, 78
--------------------	--------

V

vízia systému (Methaphor)	67
------------------------------------	----

vlastné tempo (Sustainable Pace)	68
vlastník triedy	45
vylepšovanie návrhu (Design Improvment Refactoring)	67
vývoj riadený testami (Test-Driven Development)	67
Vývoj softvéru pomocou čít	36, 37, 116

Y

YAGNI	66, 70
-------------	--------

Z

zákaznícke testy (Customer Tests)	66
znovupoužitie	4

HOBITI

Bc. Jozef Drgoňa – Bc. Martin Geleneky

Bc. Jozef Krššák – Bc. Marek Kubran

Bc. Radoslav Otipka – Bc. Ivan Polák

Bc. Richard Richter – Bc. Mária Šmigurová

Bc. Anton Weissensteiner

TVORBA SOFTVÉRU V TREŤOM TISÍCROČÍ

1. vydanie

Náklad až 1 výtlačok

142 strán, 19 obrázkov

Edičné číslo 1

Tlač Star LC-10 kompatibilná v Bratislave

Rok vydania 2002

Hodnota: pre nás nevyčísliteľná