Eseje na námety knihy Freda Brooksa

Táto časť obsahuje vybrané eseje, ktoré vytvorili študenti inžinierskeho štúdia odboru Informatika v predmete Manažment v softvérovom inžinierstve v školskom roku 1998/99 na námety knihy Frederica Brooksa: The Mythical Man-Month (Anniversary Edition, Addison Wesley, 1995). Eseje poskytujú určitý prehľad o manažmente v softvérových projektoch.

Riziká tvorby druhého projektu

Karol SLANINA

Abstrakt. Každý softvérový inžinier venuje spravidla dostatočnú pozornosť rizikám plynúcim z nedostatku skúseností pri tvorbe svojho prvého systému v nejakej oblasti. V prípade jeho úspechu nadobudne väčšina návrhárov klamný pocit ostrieľaných odborníkov a oveľa menej si uvedomuje riziká, ktoré so sebou prináša nasledujúci projekt.

Úvod

Prirodzenou vlastnosťou každého vývojára, ktorý rieši preňho zatiaľ nepoznané úlohy, je opatrnosť. Má tendenciu sústrediť sa na základné funkcie navrhovaného systému a brzdiť fantáziu, nakoľko v danej problematike nemá žiadnu prax. V procese tvorby prvého systému nadobúda jeho tvorca početné skúsenosti a len čo je systém dokončený, má už mnoho ďalších nápadov, ktoré by tento systém vylepšili.

Tvorca systému je vyzbrojený skúsenosťami z návrhu svojho prvého systému a často podlieha lákavej tendencii „vylepšiť“ svoj druhý návrh o celý rad nových funkcií a charakteristík.

Každý chápe riziká návrhu svojho prvého systému prameniace z nedostatku skúseností v príslušnej oblasti. Možno ale práve z dôvodu tejto obozretnosti dokáže mnoho tvorcov navrhnúť na prvý raz jednoduchý, ale plne funkčný a elegantný systém.

Efekt druhého systému

Žiaľ, čo si uvedomuje už oveľa menej vývojárov, je nebezpečenstvo číhajúce pri návrhu druhého systému. Tvorca systému je vyzbrojený skúsenosťami, ktoré získal v procese návrhu svojho prvého systému a často podlieha lákavej tendencii „vylepšiť“ svoj druhý návrh o celý rad nových funkcií a charakteristík, ktoré pri tvorbe prvého systému neuvažoval. Systémový architekt je plný elánu, ktorý podporuje úspech prvého produktu. Výsledná práca je monštruóznym zhlukom prvého jednoduchého systému a množstva nových nápadov, ktoré boli zapracované do druhého systému. V porovnaní s predošlou prácou systém stratil mnoho zo svojej elegancie a jednoduchosti a stal sa akýmsi slonom, obsahujúcim mnoho zbytočností.

Tento efekt dostal názov „efekt druhého systému“, v literatúre sa možno stretnúť aj s pojmom „syndróm druhého systému“. Fred Brooks, manažér projektu OS/360 firmy IBM, opisuje nahradenie jednoduchých a elegantných operačných systémov na počítačoch rady IBM 70XX operačným systémom OS/360 na IBM System/360 [1, 2]. Tento systém bol navrhnutý príliš grandiózne a veľkoryso a obsahoval veľa čŕt, ktoré mu pridávali viac zložitosti, ako úžitku (autor napríklad spomína stále prítomný proces zabezpečujúci korektné vyhodnocovanie posledného dňa priestupných rokov). Prvotnou príčinou, prečo sa v prípade OS/360 prejavil efekt druhého systému, bol fakt, že prevažná väčšina jeho tvorcov mala za sebou práve jednu skúsenosť s vývojom operačných systémov.

Prejavy efektu druhého systému

Brooks identifikoval dva možné dôsledky spomínaného efektu. Prvým z nich je, že druhý systém sa stane „veľkou kopou“, neprehľadným a príliš zložitým zhlukom, na rozdiel od svojho úspešného predchodcu. Druhým, vari ešte horším dôsledkom môže byť fakt, že sa tvorca druhého systému príliš zameria na vylepšovanie techník, ktoré sa medzičasom stali zastaralé, namiesto toho, aby ich v systéme nahradil modernými postupmi. Mnohé príklady, ktoré sa uvádzajú v [1] sa mi zdali pre dnešného čitateľa príliš vzdialené, a preto by som chcel uviesť zopár príkladov, s ktorými som sa stretol v inej literatúre, alebo som si spätne uvedomil vo svojej praxi.

Príklady produktov, v ktorých sa prejavil efekt druhého systému

Multics

Multics [3] bol operačný systém podporujúci vykonávanie viacerých programov súčasne, ktorý prišiel s množstvom nových nápadov (napr. zaviedol myšlienku pristupovania ku všetkým zariadeniam prostredníctvom špeciálnych súborov). Jeho experimentálnym predchodcom bol systém zvaný CTSS (Compatible Time-Sharing System). Multics navrhovalo konzorcium viacerých renomovaných spoločností. Napriek množstvu revolučných inovácií sa nikdy prakticky neuplatnil, práve vďaka následkom efektu druhého systému. V procese návrhu bolo

do systému zapracovaných veľa zbytočností, ako napríklad fakt, že používateľ musel zadávať prístupové heslo dokonca aj pri odhlasovaní sa zo systému. Po tom, čo sa v procese návrhu Multicsu prejavil efekt druhého systému, väčšina zúčastnených firiem na návrhu konzorcium opustila.

C++

Jazyk C je jazykom mocným a efektívnym, ale na programátora číhajú nástrahy najmä v podobe primitívnej práce s pamäťou. S tým sa spája veľmi slabá kontrola správnosti prekladaného zdrojového textu programu kompilátorom a jeho ťažká udržovateľnosť. Zlé jazyky označovali C za „jazyk, ktorý spája efektívnosť a silu asembleru s čitateľnosťou a udržovateľnosťou asembleru“ [3]. Zmenu v tejto oblasti mal priniesť nástupca jazyka C, ktorým sa stal objektovo orientovaný jazyk C++. Nemožno poprieť, že práve objektové črty jazyka C++ umožňujú neporovnateľne elegantnejšie riešenia a poskytujú prostriedky pre odhalenie oveľa väčšieho množstva chýb. C++ je ale typickým príkladom jazyka, ktorého návrh bol zasiahnutý efektom druhého systému.

Tento efekt sa prejavuje najmä vďaka cieľu tvorcov dosiahnuť úplnú spätnú kompatibilitu s C, čo sa síce podarilo, ale iba za cenu straty elegancie jazyka a prieniku všetkých nebezpečenstiev C do C++. C++ je jazykom, ktorý sa snaží spájať čisto procedurálny s objektovo orientovaným návrhom softvéru. Práve tento jeho vnútorný rozkol je príčinou, pre ktorú je jazyk ako celok mohutnou zlepeninou, ktorá v človeku vyvoláva pocit, že jej mohol byť navrhnutý omnoho jednoduchšie. V porovnaní s čisto objektovým jazykom (napr. Java) pôsobí C++ prehnane zložito, poskytuje príliš veľa prostriedkov na dosiahnutie toho istého cieľa. Práve táto jeho nadbytočnosť mu uberá veľa z čistoty a elegancie čisto objektového jazyka.

Moja osobná skúsenosť

Počas mojej relatívne krátkej praxe návrhára softvérových systémov som nemal veľa možností zúčastniť sa tvorby dvoch systémov rovnakého typu, ale myslím, že jedna skúsenosť je hodná spomenutia. Od malička je mojím koníčkom hudba. Možnosť spojiť ju s využitím počítača, bola pre mňa vždy veľmi lákavá. Vo svojom voľnom čase som sa zaujímal o možnosti programovania hudobného softvéru a vytvoril som si jednoduchý systém na prehrávanie zvukových záznamov, ktorý využíva rozhranie DirectSound. Je implementovaný v jazyku C.

Vďaka výmenným tréningovým pobytom študentov v zahraničných firmách sprostredkovaných organizáciou IAESTE som sa dostal na ročnú prax do firmy zameranej na tvorbu počítačových hier v Oslo. Firma nemala vytvorený žiaden hudobný štandard a potrebovala jednotné hudobné rozhranie, ktoré by bolo použiteľné v každej hre. Vzhľadom na moje predchádzajúce skúsenosti mi pridelili tento projekt.

Posmelený skúsenosťami, pustil som sa do návrhu systému, ktorý mal byť na rozdiel od mojej prvotiny objektovo orientovaný. Až spätne som si uvedomil, ako sa v tomto prípade prejavil efekt druhého systému, kombinovaný s mojou dovtedajšou nedostatočnou praxou v C++. Prvotný návrh hudobného rozhrania bol veľmi tesne myšlienkovo previazaný s mojím domácim jednoduchučkým systémom. Základy, na ktorých bol prvý systém postavený, nemohli uniesť zložitý a efektívne pracujúci systém potrebný pre dnešnú počítačovú hru. Uvedomil som si, že v tomto prípade mi moje dovtedajšie skúsenosti slúžili skôr ako akási hranica, s ktorou som bol vnútorne spokojný a neuvedomoval som si, že v daných nových podmienkach budem musieť prehodnotiť mnohé z postupov, ktoré pre mňa v mojom prvom systéme znamenali veľký krok vpred, ale pri daných nových požiadavkách boli už nevhodné.

Moje dovtedajšie skúsenosti mi slúžili skôr ako akási hranica, s ktorou som bol vnútorne spokojný a neuvedomoval som si, že v daných nových podmienkach budem musieť prehodnotiť mnohé z postupov, ktoré pre mňa v mojom prvom systéme znamenali veľký krok vpred …

Našťastie, zodpovedný manažér si včas všimol tento trend a poskytol mi mnohé dobré rady, takže sa zvukový systém neskôr vo firme úspešne uplatnil.

Aktuálnosť efektu druhého systému

Z viacerých príkladov a mojej osobnej skúsenosti by som mohol uzavrieť, že spomínaný efekt platí stále a s najväčšou pravdepodobnosťou ostane trvale prítomným problémom. Navyše si myslím, že v žiadnom prípade sa neohraničuje iba na oblasť softvérového inžinierstva. Za spomínaným efektom totiž zrejme stoja najmä všeobecne platné psychologické príčiny. Každého človeka, ktorý prvýkrát ukončil nejaký úspešný projekt, úspech veľmi silne ovplyvňuje. Pri tvorbe nasledujúceho má tendenciu použiť úspešné časti svojej predošlej práce, chýba mu potrebný nadhľad a môže mu uniknúť, že medzičasom sa niektoré dôležité podmienky zmenili.

V prípade návrhára softvéru tu pristupuje ďalší problém – predošlý úspešný produkt sa vo veľkej miere využíva, ale časom sa hromadia „dobré rady“ od používateľov, aké možné prídavky by systém urobili ešte oveľa lepším. Ak návrhár nezvládne výber len najpotrebnejších pridaných funkcií a posmelený úspechom sa rozhodne vyjsť v ústrety aj zbytočným požiadavkám, môže sa predtým úspešný systém stať prehnane zložitým a neefektívnym.

Predchádzanie efektu druhého systému

Najväčším problémom efektu druhého systému je, že si ho väčšina návrhárov vopred neuvedomuje. Tvorcovia s jednou skúsenosťou podvedome preceňujú svoje sily a potrebný nadhľad sa dá získať iba prácou na viacerých projektoch. Keďže svojimi prvými dvoma projektami musí prejsť každý, jedinou prevenciou je uvedomiť si, že aj napriek vnútornému pocitu skúsenosti treba byť pri návrhu druhého systému nanajvýš opatrný a pozorne ohodnocovať výhody a nevýhody každého kroku. Na podobný

klamný pocit vlastnej skúsenosti doplatili už mnohí a treba sa z toho poučiť. Nemalo by sa nikdy stať, aby bol za vedúceho dôležitého projektu dosadený človek s jedinou skúsenosťou v danej problematike. Takto môže vedúci zachytiť príznaky efektu druhého systému v práci svojich podriadených v skorom štádiu.

Literatúra

1. F.P. Brooks. The Mythical Man-Month, Addison-Wesley Longman, Inc., 1995.
2. M.J. Flynn. Computer Engineering 30 Years After the IBM Model 91, IEEE Computer, April 1998.
3. E. Dumbill. Jargon Lexicon, 1994. ftp.informatik.rwth-aachen.de/jargon300.

Prečo sa nepodarilo postaviť vežu Bábel?

Marek TRABALKA

Abstrakt. Esej v stručnosti približuje problémy, ktoré vznikajú v softvérových projektoch kvôli nedostatočnej komunikácii a zlej organizácii. Načrtnem možné spôsoby komunikácie, ktoré sa v praxi používajú, spolu s možnými perspektívami do budúcnosti. Vysvetlím dôležitosť rozdelenia práce v tíme a spôsoby organizácie vedenia tímu po stránke riadiacej a technickej.

> „*... pomäťme tam ich reč, aby nikto nerozumel reči druhého*." GENEZIS *11, 7*

Prečo sa nepodarilo postaviť vežu Bábel?

Príbeh o veži Bábel je všeobecne známy po celom svete. Keď skúsime odhliadnuť od morálno-náboženského aspektu príbehu, môžeme sa na celý projekt pozrieť ako na prvé veľké zdokumentované inžinierske fiasko. Práve z tohto pohľadu je pre nás veľmi užitočné zamyslieť sa nad tým, prečo vlastne projekt skrachoval z pohľadu manažmentu. Stavba mala:

Akonáhle ľudia prestanú komunikovať, celý systém práce začína upadať.

1. jasný cieľ, hoci zrejme nedosiahnuteľný
2. ľudské zdroje: množstvo
3. materiál: v Mezopotámii bol dostatok materiálu na stavbu
4. čas: prakticky neobmedzený, nebolo tu žiadne časové ohraničenie
5. technológia: použiteľná, architektúra založená na pyramídovitých a kónických tvaroch je dostatočne stabilná

Napriek všetkým týmto dobrým podmienkam však projekt zlyhal. Dôvod bol jednoduchý: zlyhala komunikácia a následne na to organizácia projektu.

Komunikácia vo veľkom softvérovom projekte

Z minulosti, a nielen tej biblickej, ľudia postupne pochopili význam komunikácie pri riešení veľkých projektov, a to nielen softvérových. Hoci myslím, že pri týchto je problém s komunikáciou obzvlášť závažný. Akonáhle ľudia prestanú komunikovať, celý systém práce začína upadať. Z vlastných skúseností viem, aké je nepríjemné, keď vytvoríte nejaký modul a následne zistíte, že váš kolega práve dokončuje tvorbu modulu s rovnakou funkcionalitou. Toto ešte nie je najhorší prípad. Oveľa horšie pre vás bude, keď váš kolega síce naprogramuje knižnicu s funkciami, ktoré potrebujete pre dokončenie vášho modulu, ibaže ich urobí narýchlo v jazyku Visual Basic a vy ich potrebujete na časovo kritickú aplikáciu.

Takéto, ale aj omnoho bizarnejšie problémy sú však v oblasti informačných technológií vecou veľmi častou a zdá sa mi, že s postupom času je to stále horšie a horšie, namiesto aby sa to zlepšilo. Pochopiteľne pri tímoch s tromi či štyrmi ľuďmi sa dá komunikácia riešiť ústne pri občasných poradách a v prípade potreby elektronickou poštou či telefonicky.

Situácia však začína nadobúdať úplne iné rozmery s pribúdajúcim počtom ľudí v tíme. Ako sa tím zväčšuje, nároky na komunikáciu nerastú lineárne, ale omnoho rýchlejšie. Pri veľkých tímoch je naviac časté, že jednotliví pracovníci sú rozmiestnení v rôznych fyzických lokalitách, čo taktiež komplikuje vzájomnú informovanosť.

Metódy komunikácie

S akými metódami tímovej komunikácie sa teda stretávame?

- *neformálna komunikácia*
  Táto forma je nevyhnutná pri každom type projektu. Pri veľkých projektoch sa používa ako doplnková, pri malých môže hrať dokonca kľúčovú úlohu. Jej hlavnou úlohou je obvykle objasniť iba malé problémy, či dohodnúť drobné funkčné detaily.
- *stretnutia*
  Pravidelné projektové stretnutia, pri ktorých si tímy navzájom konzultujú technické riešenia sú na nezaplatenie. Týmto spôsobom možno vysvetliť jednoducho a priamo množstvo drobných nezrovnalostí, ako aj objasniť základné myšlienky a koncepty.
- *výkazy práce*
  Dôležitou formou komunikácie sú aj správy o vykonanej práci, určené najmä pre priameho nadriadeného. Pomocou nich možno sledovať nielen prácu jednotlivých ľudí, ale aj stav projektu.
- *pracovná kniha*
  Je to výborný formálny prostriedok nielen na komunikáciu v tíme a medzi tímami, ale aj na dokumentovanie produktu a jeho vývoja.

Počas svojej programátorskej praxe som sa stretol so všetkými uvedenými formami komunikácie. Najčastejšie to bola *neformálna komunikácia*.

Keď som asi pred štyrmi rokmi prvýkrát nastúpil do zamestnania ako programátor, bolo to v malej firme, ktorá sa zaoberala distribúciou finančných informácií v reálnom čase. Hlavnou úlohou nás všetkých (nielen programátorov), bolo udržiavať systém v prevádzke. Komunikácia prebiehala takmer výlučne telefonicky alebo priamym rozhovorom. Iba raz za týždeň alebo dva sme mali schôdzku všetkých pracovníkov, kde sme si porozprávali aké problémy boli a ako sa vyriešili. Tieto schôdzky mávali takmer výlučne informatívny charakter. O nejakej pracovnej knihe nebolo ani reči, a preto keď odchádzala správkyňa systému, jej hlavnou úlohou bolo naučiť spravovať systém druhého kolegu.

Nasledujúca firma, v ktorej som pracoval, bola síce počtom zamestnancov podobná tej predošlej, ale úroveň riadenia a disciplíny bola omnoho vyššia. Pracovali sme na viacerých projektoch, a preto sme boli rozdelení do viacerých tímov. Počtom najväčší tím pracoval na vývoji grafického CAD systému pre odevných návrhárov. Počet ľudí sa s časom menil, od piatich až po ôsmich ľudí. Celý čas však riadil projekt jeden človek, ktorý organizoval *skutočné pracovné schôdzky*. Samozrejme aj na nich sa hodnotilo čo kto urobil, ale ich dôležitou náplňou býval aj tzv. „brainstorming“, pri ktorom sa kolektívne nahlas uvažovalo, diskutovalo aj hádalo o najlepšom riešení rôznych problémov. Čo však bolo veľmi dôležité, neostalo iba pri takejto slovnej výmene, ale po skončení vypracoval vždy náš manažér zápis z porady a poslal ho všetkým členom tímu. Preto sa nikdy nestalo, že by niekto zabudol na úlohu, ktorú mal splniť alebo by nevedel, čo robia ostatní. Nuž a napokon záznamy slúžili aj ako informácia o práci jednotlivcov aj celého tímu pre majiteľa firmy.

Neskôr som sa ocitol vo veľkej softvérovej firme. Hneď od začiatku človek pochopiteľne pocíti zmenu v organizácii, keď zistí že má nad sebou niekoľko úrovní riadenia. Pracovali sme na väčšom projekte, pričom počet ľudí v tíme sa v čase menil – bolo nás od desať do pätnásť ľudí. Určite uznáte, že tu už by komunikácia každého s každým spôsobila veľké problémy. Tu už boli časté technické porady týkajúce sa špecifického problému, pri ktorých neboli všetci členovia tímu, ale iba tí, ktorých sa problém priamo dotýkal. S tým súvisel problém, že nie každý vedel, čo všetko robia ostatní. Preto sa občas stalo, že dvaja ľudia programovali rovnaké pomocné funkcie pre svoje časti.

Práve tu sme museli písať *pravidelné týždenné hlásenia* o vykonanej práci a plán na nasledujúci týždeň. Určite nikoho neprekvapí, že sme sa všetci bránili, pretože sa nám to zdalo zbytočné. Keď sme si však na výkazy zvykli, postupne sme s potešením zistili, že sa nás vedúci tímu menej často pýta na čom pracujeme a koľko sme už z toho dokončili.

Taktiež sa pre náš projekt zaviedla tzv. *pracovná kniha*. Nebola to viazaná papierová kniha, ale iba elektronické dokumenty. Nebol to iba jeden veľký dokument, ale súhrn všetkých dokumentov, ktoré sa týkali projektu, umiestnený na jednom dohodnutom mieste na servri. Obsah týchto dokumentov bol rôznorodý. Bola tu podrobná špecifikácia projektu, opísané rozhrania jednotlivých COM komponentov, vysvetlená spolupráca jednotlivých častí aj spôsob, ako a v akom poradí treba inštalovať jednotlivé časti. Mali sme dokonca aj vlastnú *bázu znalostí*, kde boli napísané problémy, s ktorými sa stretli jednotliví členovia tímu a ako ich nakoniec vyriešili, prípadne obišli.

Zrod elektronickej pošty ovplyvnil vo veľkej miere najmä neformálnu komunikáciu a výkazy práce.

Význam týchto dokumentov je pri väčších projektoch kľúčový. Nie je možné vysvetľovať všetky podrobnosti na stretnutiach a ani nemožno očakávať, že neformálnym rozhovorom bude vývojár opisovať podrobnosti svojej implementácie viacerým kolegom. Práve z týchto dokumentov neskôr vznikne používateľská príručka a príručka pre administrátorov a práve tieto texty budú slúžiť spoločne so zdrojovým textom programov na hľadanie a opravu chýb v neskorších fázach projektu. Veľkou výhodou takejto dokumentácie je aj jednoduchá možnosť výberu relevantných častí. Problém nebýva ani tak v snahe zabrániť prístupu k informáciám, ako skôr upriamiť pozornosť na tie informácie, ktoré sú pre toho ktorého pracovníka dôležité.

Napokon dôležitou funkciou pracovnej knihy je vyriešiť správu verzií dokumentov. Tak ako projekt sám, tak aj dokumenty sa nielen pridávajú, ale aj upravujú a mažú. A je samozrejmé, že potrebujeme, aby všetci členovia tímu videli najnovšie verzie.

Prostriedky komunikácie

Tak ako kedysi hrali pri stručnej komunikácii dominantnú úlohu fax, telefón a papier, preberajú ju dnes postupne elektronická pošta, internet a podporné softvérové prostriedky. Internet ako fenomén dnešných čias významne ovplyvňuje myslenie a správanie sa ľudí všeobecne a profesionálov z oblasti informačných technológií ešte oveľa viac. Z hľadiska manažmentu projektu je pre nás zaujímavý najmä tým, že nám poskytuje viacero nových spôsobov komunikácie.

Skúsme sa pozrieť, akým spôsobom sa menia prostriedky, ktoré sa používajú pri jednotlivých formách komunikácie.

Elektronická pošta je najpoužívanejšou službou na Internete. Založená je na princípe klasickej pošty, umožňuje teda jednostrannú komunikáciu, pri ktorej príjemca správy nemusí priamo reagovať a byť prítomný. Hodí sa najmä na informovanie o stave a na posielanie správ a dokumentov. Menej vhodný je tento spôsob pre interaktívnu komunikáciu.

Dokumenty poslané faxom či obyčajnou poštou dostane príjemca na papieri, zatiaľ čo pri elektronickej pošte obdrží priamo údaje v elektronickej podobe. Tým môže ľahko manipulovať s obsahom a prípadne ho upravovať či inak ďalej spracúvať. Samozrejmou výhodou je aj možnosť prenášať nielen text či obrázky, ale napr. aj celé aplikácie či databázy údajov a pod.

Zrod elektronickej pošty ovplyvnil vo veľkej miere najmä neformálnu komunikáciu a výkazy práce. Práve pomocou elektronických správ si programátori vymieňajú rady a príklady, narýchlo objasňujú význam jednotlivých metód v rozhraniach súčiastok či dohadujú termíny pracovných porád.

A dnes sa už takmer výhradne pomocou elektronickej pošty posielajú nadriadeným výkazy práce.

Elektronická pošta sa používa aj na distribúciu softvérových súčiastok v rámci tímu. Tento spôsob však nebýva práve najšťastnejší pri väčších projektoch. V takomto prípade je vhodnejšie posielať iba upozornenia, že existuje novšia verzia tej-ktorej súčiastky, pričom samotné súčiastky sú uložené centrálne.

Tu sa práve dostávame k softvérovým prostriedkom na správu verzií. Zdieľanie dokumentov, zdrojových textov ale aj preložených modulov či dát bolo vždy spojené s mnohými problémami. Klasickým spôsobom je uloženie takýchto častí projektu na jednom mieste, prístupnom všetkým členom tímu, teda obvykle v nejakom zdieľanom adresári na serveri. Z praxe však viem, že toto riešenie sa pri väčšom projekte zákonite stane nepoužiteľným. Prečo? Dôvod je jednoduchý: kvôli rôznym verziám jednotlivých častí. Pri desiatich ľuďoch je jednoducho nemožné, aby všetci efektívne pracovali a mali pritom rovnaké verzie všetkých častí systému. Niekto, kto má začať robiť na novej časti bude samozrejme chcieť pracovať s najnovšími súčiastkami. Avšak programátor, ktorý ladí jadro systému a používa ostatné súčiastky iba na testovanie a hľadanie závad vo svojich moduloch, nebude potrebovať najnovšie súčiastky, kvôli ktorým by musel vždy mierne upraviť svoje rozhrania. Práve na vyriešenie tohto závažného problému sa používajú podporné prostriedky, ktoré umožňujú prácu s rozličnými verziami v podstate ľubovoľných súčiastok, reprezentovaných textom aj binárne.

Účelom organizácie projektu je redukovať množstvo komunikácie a koordinovať ju.

Správa verzií z hľadiska manažmentu projektu sa týka predovšetkým pracovnej knihy. Užitočnosť týchto prostriedkov vám ľahko priblížim porovnaním so spôsobom, akým sa riešila pracovná kniha v minulosti. Brooks [1] opisuje prostriedky komunikácie v projekte OS/360, ktorý riešili pred vyše dvadsiatimi rokmi. V projekte pracovali stovky ľudí na rôznych miestach. Preto existencia dobre štruktúrovanej pracovnej knihy bola absolútnou nevyhnutnosťou. Krátko po začatí prác zistili, že treba, aby každý programátor videl všetok materiál. To znamenalo, že každý musel mať kópiu pracovnej knihy dostupnú v mieste svojho pracoviska. Kritickým bolo najmä dopĺňanie a obnova dokumentu, pretože každý pochopiteľne potreboval najnovšiu verziu. To bolo veľmi náročné, pretože bolo treba nielen napísať zmeny, ale aj zabezpečiť výmenu zmenených stránok vo všetkých kópiách.

Keď dostali programátori zmenené stránky, zaujímali ich najmä dve otázky: „Čo sa zmenilo?“ a „Aká je teda aktuálna definícia?“. Preto sa na zmenených stránkach používali vertikálne čiary na označenie zmenených odsekov.

Po pol roku však vznikol ďalší veľký problém: jedna kópia pracovnej knihy bola hrubá asi jeden a pol metra! Naviac každý deň prišlo ku každej kópii asi 150 zmenených strán! Starostlivosť o pracovnú knihu začala postupne zaberať podstatnú časť pracovného dňa. Tieto okolnosti prinútili manažment prejsť na technológiu mikrofišov, ktorá ušetrila milióny dolárov. Objem knihy sa prudko zmenšil a celková manipulácia zjednodušila. Nevýhodou však bolo, že už nebola možnosť jednoducho písať poznámky priamo do knihy a označovať si dôležité časti.

Keď sa na problémy s pracovnou knihou projektu pozrieme z dnešného pohľadu, riešenie je v podstate jednoznačné: počítačová sieť a softvér na správu verzií. Takto odpadnú problémy s distribúciou, pretože dokumenty sú vždy prístupné pomocou siete v najnovšej podobe. Dobrý systém na správu verzií vie priamo zvýrazniť riadky textu, ktoré niekto pridal, vymazal alebo zmenil, a to medzi ľubovoľnými dvomi verziami dokumentu. Naviac celková správa knihy je priamočiara a je jednoduché zálohovať ju ako celok aj s uchovaním informácie o zmenách. Veľmi dôležitou výhodou je nesporne skutočnosť, že pri takomto prístupe netreba žiadnych osobitných pracovníkov na správu knihy, pretože ju priamo spravujú všetci členovia tímu. Zmeny sa premietajú všetkým pracovníkom okamžite.

Organizácia veľkého softvérového projektu

Základnou úlohou softvérového vývojárskeho tímu je vytvoriť ilúziu jednoduchosti – na skrytie podrobností systému pred jeho používateľom. V dnešných časoch sa dodávajú systémy, ktorých veľkosť sa pohybuje v stovkách tisícok až v miliónoch riadkov zdrojového programu vo vyššom programovacom jazyku! Jednoducho nie je možné, aby jeden človek úplne pochopil takýto zložitý systém. Hoci dobre dekomponujeme projekt, stále nám ostanú rádovo stovky samostatných modulov. A ako dokázali experimenty psychológov, maximálne množstvo skupín informácií, ktoré dokáže človek simultánne zvládnuť je sedem plus mínus dva. Preto sa na tvorbu rozsiahlych systémov jednoducho musia vytvárať tímy ľudí. Snahou je vždy, aby bol tím čo najmenší, pretože čím je väčší, tým je komunikácia medzi jeho členmi zložitejšia.

Keď si predstavíme projekt s n pracovníkmi, zistíme, že počet možných navzájom komunikujúcich dvojíc je $n*(n-1)/2$. Ak máme teda napríklad desať ľudí na projekte, predstavuje to 45 možných spojení, pri 100 ľuďoch je to 4950 spojení!

Účelom organizácie projektu je práve redukovať množstvo komunikácie a koordinovať ju. Vo väčšine organizácií vidíme hierarchickú, tzv. stromovú štruktúru riadenia. Táto štruktúra vznikla na základe úrovne zodpovednosti a autority, pričom sa vychádzalo z princípu, že každý pracovník má iba jedného nadriadeného. Preto sa veľké organizácie členia zvyčajne na divízie, ktoré sa skladajú z oddelení, v ktorých funguje niekoľko tímov.

Komunikácia ako taká sa však nemôže obmedziť len na takúto štruktúru. Pokiaľ dva tímy navzájom spolupracujú na jednom projekte, určite nebudú navzájom komunikovať výhradne prostredníctvom vedúceho celého oddelenia. Práve naopak, v skutočnosti budú často komunikovať práve pracovníci na najnižšej úrovni medzi sebou. Neadekvátnosť modelu riadenia firmy na projektové riadenie sa odráža vo vytváraní rôznych pracovných skupín zameraných na jednotlivé úlohy alebo dokonca vo vytvorení tzv. maticovej organizácie, kde sa kombinuje zadelenie ľudí z pohľadu riadenia firmy a z pohľadu riadenia projektov.

Každý správny tím je založený na dvoch princípoch: deľba práce a špecializácia funkcie. V softvérovom projekte zastávajú pracovníci tieto roly: manažér projektu, architekt projektu, analytik, návrhár, implementátor, dokumentátor, tester, administrátor.

Z hľadiska organizácie a manažmentu projektu sú pre nás zaujímavé najmä roly manažéra a architekta projektu. Veľmi často sa stretávame so zamieňaním týchto pojmov, či dokonca s ich nesprávnym chápaním. Spôsobuje to zrejme skutočnosť, že obidve funkcie často zastáva jedna a tá istá osoba. Skúsme si teda bližšie vysvetliť funkciu manažéra a funkciu architekta a porovnať spôsoby ich vzájomného vzťahu z hľadiska výhodnosti pre rôzne typy projektov.

Manažér projektu

Čo je hlavnou úlohou manažéra projektu? V prvom rade je to riadenie tímu, rozdeľovanie práce a tvorba plánu projektu. Taktiež musí zaobstarávať zdroje pre projekt, či už ľudské vo forme ďalších pracovníkov, alebo softvérové, hardvérové a iné. Z toho vyplýva, že veľká časť jeho komunikácie smeruje mimo tím. Práve manažér je zodpovedný za posielanie výkazov o postupe prác nadriadeným. Musí mať preto prehľad o práci jednotlivých ľudí v tíme. Musí dohliadať na postup prác, porovnávať aktuálny stav s plánom projektu a prípadne modifikovať plán a rozdelenie zdrojov podľa potrieb projektu.

Keď si rozoberieme úlohy manažéra, zistíme, že na túto pozíciu sa hodí človek s ekonomickým, resp. manažérskym vzdelaním a skúsenosťami, ktorému stačia častokrát len hrubé vedomosti z technickej oblasti.

Architekt projektu

Úloha architekta projektu je kľúčová z hľadiska technickej realizácie. Architekt predstavuje hlavnú hybnú silu, pričom rozhoduje o kľúčových otázkach analýzy a návrhu systému, špecifikuje ako sa bude správať systém navonok aj aká bude jeho vnútorná štruktúra. Musí mať neustály prehľad o celkovom fungovaní systému a musí udržiavať celkovú integritu vyvíjaného produktu. Ako sa objavujú jednotlivé technické problémy, musí zabezpečiť ich riešenie alebo upraviť návrh systému.

Keď si uvedomíme náročnosť úlohy architekta projektu, zistíme, že práve on ako osoba je v podstate hlavným limitujúcim faktorom zložitosti celého systému. Dobrým architektom môže byť len človek s výbornými technickými znalosťami a skúsenosťami. Na druhej strane jeho skúsenosti s riadením ľudí a zdrojov nie sú natoľko dôležité.

Manažér a architekt

Keď porovnáme úlohy manažéra a architekta, zistíme celkom jednoduchý rozdiel: manažér sa stará o to kto a kedy na niečom pracuje, architekt má za úlohu povedať čo a ako sa má robiť. Preto manažér zodpovedá najmä za postup projektu, zatiaľ čo architekt zodpovedá za jeho výsledok, teda produkt.

Keby sme prirovnali softvérový tím ku klasickým hodinkám, architekt by zrejme fungoval ako pružinka, ktorá poháňa celý kolotoč vývojárov – do seba zapadajúcich koliesok. A manažér by bol práve kľúčikom, pomocou ktorého sa celý systém nielen spúšťa do chodu, ale sa v ňom aj udržiava.

Rôzni ľudia bývajú rôzne nadaní a z toho vyplýva aj ich vhodnosť na jednotlivé úlohy.

Myslím, že teraz je už úplne jasný rozdiel medzi týmito dvomi úlohami. Čo je však dôležité uvedomiť si, je veľký rozdiel medzi potrebnými schopnosťami a skúsenosťami pre tieto úlohy.

Rôzni ľudia bývajú rôzne nadaní a z toho vyplýva aj ich vhodnosť na jednotlivé úlohy. Pretože obidve tieto roly majú v tíme riadiacu úlohu a obvykle chceme dodržať pravidlo jedného vedúceho, môžeme dostať tri rôzne vzťahy medzi manažérom a architektom projektu:

- manažér a architekt je tá istá osoba
- manažér je vedúci, architekt je jeho pravá ruka
- architekt je vedúci a manažér je jeho pravá ruka

Manažér a architekt je tá istá osoba

V praxi je situácia, kedy jeden človek zastáva obidve riadiace funkcie, veľmi častá. Treba si však uvedomiť, že toto funguje iba pri malých projektoch s najviac päť-šesť člennými tímami.

Pri väčších projektoch toto nemôže fungovať hneď z dvoch dôvodov. V prvom rade je veľmi zriedkavé nájsť človeka, ktorý má technický talent a pritom je aj výborný manažér. Myslitelia sú zriedkaví, realizátori ešte zriedkavejší, ale myslitelia-realizátori v jednej osobe sú najzriedkavejší.

Druhý dôvod prečo toto nemôže pri rozsiahlejších projektoch fungovať je celkom prozaický: obidve úlohy sú časovo veľmi náročné a vyžadujú človeka na plný úväzok. Preto ak sa niekto pokúša o oboje, obvykle na to doplatí koncept systému alebo riadenie ľudí a zdrojov, v tom najhoršom prípade oboje.

Keď je „hlavným mužom“ manažér

Pri väčších projektoch je zrejme najčastejší model práve tento. Hlavnou autoritou je manažér, ktorý rozdeľuje úlohy a plánuje projekt. Pri tom mu asistuje architekt, ktorý funguje ako jeho pravá ruka.

Správne je, ak manažér navonok veľmi jasne deklaruje architektovu autoritu čo sa týka technického riešenia a s ním spojených problémov. Práve týmto sa dá vyhnúť problému dvoch nadriadených, pretože pokiaľ má pracovník technický problém, vie že ho má riešiť s architektom a pokiaľ je problém netechnického rázu, jeho nadriadeným je manažér.

Keď projekt vedie architekt

Menej častá je situácia, keď architekt funguje ako hlava projektu a manažér je jeho pravá ruka. Manažér je v tomto prípade čosi ako „dievča pre všetko“ a jeho hlavnou úlohou je odbremeniť architekta od všetkých netechnických povinností.

Ako to teda vyriešiť?

Podobne ako vo väčšine iných problémov týkajúcich sa ľudí a práce s nimi, ani tu nemožno jednoducho povedať „Toto riešenie je správne, takto to urobte a bude to dobre.“ Napriek tomu však možno odporučiť vhodnosť jednotlivých vzťahov medzi manažérom a architektom pre jednotlivé typy projektov.

Ak architekt a manažér sú rozdielne osoby, je veľmi dôležité, aby ich vzájomný vzťah bol na dobrej úrovni.

Keď je tím veľmi malý, teda tak do päť ľudí, môžeme si celkom dobre predstaviť, že jeden človek sa stará o celkové riadenie projektu a teda funguje ako architekt aj manažér súčasne.

Pri malých, prípadne stredne veľkých projektoch sa dá výhodne použiť schéma, pri ktorej má hlavnú úlohu architekt. Riešenie projektu sa vtedy zameriava najmä na tvorbu produktu. Tvorba hlásení a správ, zisťovanie potrieb zákazníka či zabezpečovanie prostriedkov pre vývoj ostáva na pleciach manažéra. Tým sa odbremeňuje architekt aj ostatní vývojári od „nepodstatných maličkostí, bez ktorých by to nešlo“. V takýchto prípadoch sa však často stáva, že plánovanie projektu a určovanie práce pre členov tímu robí architekt systému a nie manažér.

Vo všeobecnosti je pre stredné a veľké projekty zrejme najvhodnejšie, ak celý projekt vedie manažér a asistuje mu pri tom architekt. Pre manažéra je potom oveľa ľahšie zaobstarávať potrebné zdroje, pretože má nielen najväčší prehľad o projekte, ale aj väčšie právomoci.

Pri schémach, kde architekt a manažér sú rozdielne osoby, je veľmi dôležité, aby ich vzájomný vzťah bol na dobrej úrovni. Pokiaľ totiž nebudú dobre komunikovať títo kľúčoví ľudia tímu, nemôže nikdy fungovať tím ako celok.

Ako by sme vežu Bábel stavali dnes?

Vežu Bábel sa nepodarilo postaviť. Avšak podarilo sa postaviť pyramídy v Gize, tunel pod kanálom La Manche a mrakodrapy na Manhattane. Podarilo sa naprogramovať úspešné vesmírne sondy, technicky nesmierne náročné vojenské systémy, aj počítače, ktoré porazili majstra sveta v šachu. Tieto a obrovské množstvo ďalších projektov dokazujú, že ľudstvo sa poučilo z neúspechov. Stále existuje alarmujúce množstvo projektov, ktoré sa nikdy nedokončia, alebo ak sa aj dokončia, nikdy sa v praxi nepoužijú. S každým takýmto projektom však zisťujeme ďalšie príčiny neúspechu, ktorých sa môžeme neskôr vyvarovať.

Zistili sme už, že kľúčovú úlohu pri softvérových projektoch hrajú komunikácia a organizácia a preto vieme, že problémy v tomto smere musíme riešiť ako prvé. To nám dáva do budúcnosti šancu na skvalitnenie práce a zvýšenie úspešnosti softvérových systémov.

Literatúra

1. F.P. Brooks. The Mythical Man-Month: Essays on Software Engineering. Anniversary Edition. Addison Wesley 1995.
2. G. Booch. Object-Oriented Analysis and Design with Applications. Second Edition. The Benjamin/ Cummings Publishing Company, Inc. 1994.

Zmeny v živote programu

Michal KADLEC

Abstrakt. Zmeny, či už ide o vynútené zmeny vplyvom zlej funkčnosti systému, o reakciu na požiadavky používateľa alebo o nutnosť prispôsobiť systém, zamestnancov, organizáciu na nové podmienky, sa stali neodmysliteľnou súčasťou životného cyklu softvérového produktu. Táto esej sa venuje zmenám, ich predvídaniu, zapracúvaniu a riadeniu počas života softvéru. Venuje sa nutnosti ich výskytu a nutnosti považovať ich nie za niečo výnimočné, neočakávané alebo nebezpečné, ale za neoddeliteľnú súčasť života softvéru, tak ako sú zmeny neoddeliteľnou súčasťou našich životov.

24 rokov po

Podkladom tejto eseje je iná 24 rokov stará esej Freda Brooksa „Plan to Throw One Away“ z jeho knihy „The Mythical Man-Month“. Brooksova esej sa zaoberá nutnosťou plánovania pri vytváraní tzv. pilotnej verzie softvérového systému. Táto verzia má slúžiť na overenie a odskúšanie správnosti navrhnutého riešenia. Podľa autora, prvé riešenie je málokedy (pri veľkých systémoch takmer nikdy) správne a okamžite použiteľné. Obyčajne treba úplné prepracovanie tejto prvej verzie systému na novú verziu, ktorá odstráni nedostatky prvej. Podľa autora základná otázka nestojí, či sa má vytvárať pilotný systém, ktorý sa potom zahodí – pretože to tak určite bude, ale či sa má rovno plánovať, že sa vytvorí systém, ktorý sa zahodí a na jeho základe sa vybuduje funkčný systém alebo sa má systém na zahodenie predať používateľovi, ktorý sa s ním bude trápiť, pričom novú verziu bude nutné vytvoriť tak či tak.

V ďalších častiach eseje sa autor zaoberá celkovým vplyvom zmien na tvorbu softvérového produktu od plánovania zmien počas návrhu, cez zmeny organizačnej štruktúry až po zmeny, ktoré prichádzajú po nasadení systému do prevádzky.

Keď som čítal túto esej po 24 rokoch, zamrazila ma jej nadčasovosť. Keby som nevedel, že táto esej bola napísaná pred 24 rokmi (tých 24 rokov zdôrazňujem preto, že predstavujú celý môj život) ani by som si to neuvedomil. Ide o jedno z najrýchlejšie sa rozvíjajúcich odvetví priemyslu, kde sa nové technológie, postupy, techniky, prostriedky a názory objavujú s neuveriteľnou rýchlosťou a pokojne by mohla byť uverejnená dnes a na jej platnosti by to nič nezmenilo.

Aj dnes sa hovorí o nutnosti plánovania systémov pre zmenu a stále sa viac hovorí ako praktikuje. Aj dnes sa hovorí o nutnosti vedomého riadenia zmien akoby išlo o novú prevratnú myšlienku, o nutnosti prispôsobovania zaradenia pracovníkov podľa zmeny projektu tak, ako pred 24 rokmi.

Zmeny existujú

Zmeny vstupujú do procesu vývoja softvérového produktu v každej časti jeho životného cyklu, od špecifikácie požiadaviek a architektonického návrhu cez implementáciu a testovanie až po integráciu, nasadenie a údržbu. Ich význam, ale najmä dopad, nie je v jednotlivých krokoch životného cyklu rovnaký. Zatiaľ čo zmeny špecifikácie požiadaviek majú dopad na všetky ďalšie etapy vývoja, dopad zmien v implementácii nemusí presiahnuť rozsah implementovaného modulu.

V súčasnosti pri požiadavkách na čo najrýchlejší vývoj, pri skracujúcich sa cykloch dodávania nových verzií programových systémov na trh, prevláda pokušenie považovať procesy a nástroje na riadenie zmien ako zbytočnú réžiu, ktorá spomaľuje rýchlosť vývoja. Skúsenosti však ukazujú, že vhodné techniky manažmentu zmien môžu zrýchliť softvérový vývoj zlepšením komunikácie a predchádzaním znovuvytváraniu už existujúcich softvérových súčiastok.

Pri vývoji nového softvérového systému, existuje snaha predvídať zmeny tak, aby systém bol jednoducho modifikovateľný. Predvídanie zmien nie je jednoduché, pretože existuje veľa dôvodov prečo sa systém môže alebo musí meniť. Napr.: zmení sa okolie softvérového systému, nájde sa chyba, treba zlepšiť výkonnosť alebo iné jeho vlastnosti.

Typy softvérových systémov podľa citlivosti na zmeny

Manny Lehman zaviedol klasifikáciu programových systémov vzhľadom na ich závislosť od prostredia, v ktorom pracujú [3]. Lehman rozdelil systémy na S-, P- a E- systémy. Tieto systémy sú charakterizované spôsobom, akým softvér spolupracuje so svojím prostredím a stupňom možnosti zmeny prostredia a riešeného problému.

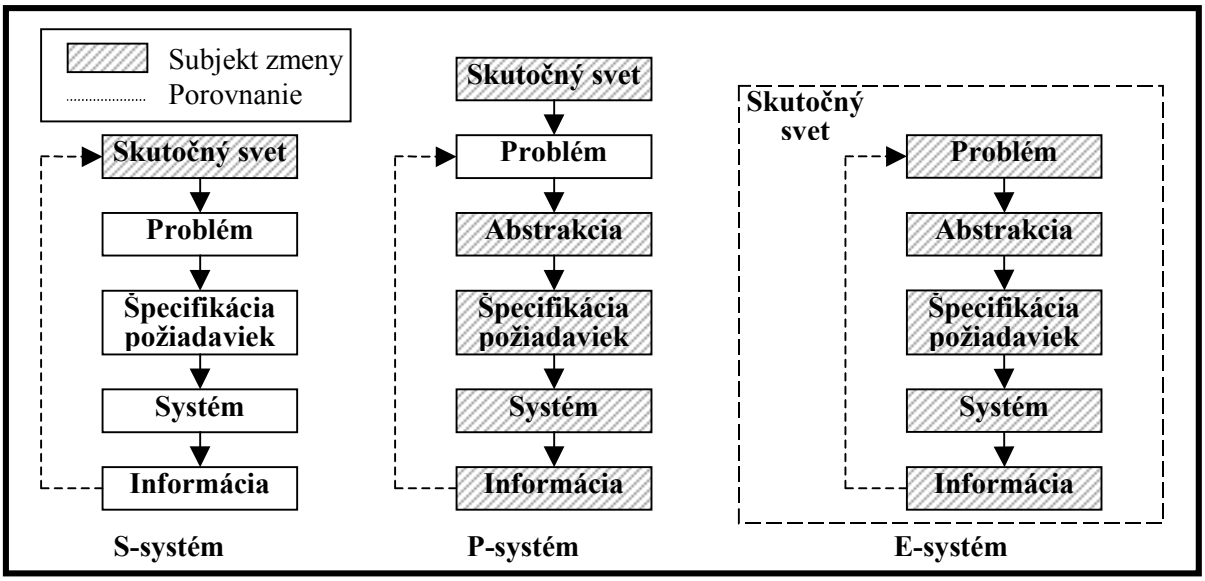
S-systémy

S-systémy sú formálne definované a odvoditelné zo špecifikácie. Problém, ktorý riešia je úplne opísaný a existuje aspoň jedno jeho správne riešenie. Toto riešenie je veľmi dobre známe, takže programátor sa nemusí zaoberať hľadaním správneho riešenia, ale len hľadaním správnej implementácie známeho riešenia (napr. operácie s maticami).

Takýto systém je pomerne statický. Vzhľadom na to, že závisíme od skutočného sveta (obr. 1), ktorý sa môže zmeniť, môže sa stať, že riešený problém už nie je súčasťou skutočného sveta. Výsledkom zmeny skutočného sveta bude nový problém.

P-systémy

P-systém je taký, ktorý rieši také problémy, ktoré nemožno úplne opísať. Aj keď teoretické riešenie problému existuje je z mnohých príčin nepraktické alebo dokonca nemožné toto riešenie implementovať. (Napr. hra šach; aj keď pravidlá sú úplne definované a teda sa dajú určiť všetky možné ťahy a ich vplyv na ďalší vývoj hry, nie je možné pri súčasnej technológii implementovať systém, ktorý preverí všetky možné ťahy až do konca hry a vyberie naj-



Obr. 1: Porovnanie S-, P- a E- systémov vzhľadom na možné zdroje zmien v systéme
(Zdroj: Pfleeger, 1998)

lepší). Pri P-systémoch vytvárame praktickú abstrakciu namiesto úplného opisu reálneho problému.

V P-systéme sa môže zmeniť mnoho vecí. Ak sa porovná výstup systému s problémom môže dôjsť ku zmene abstrakcie problému, zmene požiadaviek, čo ovplyvní aj jeho implementáciu.

E- systémy

Zatiaľ čo v S- a P-systémoch, zostáva situácia v skutočnom svete stabilná, E-systémy zahŕňajú aj meniaci sa svet a menia sa zároveň s ním. Systém je integrálnou súčasťou sveta, ktorý modeluje. Vlastnosťou E-systémov je neustála zmena. Každá časť takéhoto systému sa môže zmeniť a zároveň výstup takéhoto systému môže zmeniť aj skutočný svet.

Problémy riešené S-systémami sú úplne definované a väčšinou nemenné. P-systémy sú približným riešením problému a môžu si vyžiadať zmenu. Modifikácie sa môžu vyskytnúť počas všetkých krokov vývoja. E-systémy používajú abstrakcie a modely na priblíženie skutočnosti tak isto ako aj P-systémy, teda na ne vplývajú aj rovnaké zdroje zmien. Okrem toho sa v E-systémoch môže zmeniť aj samotný problém, ktorý majú riešiť.

Nikdy nebudeme mať istotu, že sa systém už nebude meniť. Musíme predpokladať, že sa bude meniť a vytvoriť ho tak, aby sa dal meniť jednoducho.

Riadenie zmien

Riziká, ktoré sprevádzajú zmeny môžeme prekonať iba aktívnym riadením zmien. Musíme rozumieť dôsledkom každej zmeny. Toto vyžaduje

- poznať zmeny, ktoré sa môžu vyskytnúť a celkový rozsah zmien v projekte
- odhadnúť dôsledky každej navrhnutej zmeny
- sústavné rozhodovanie v súvislosti s povolením alebo zakázaním zmeny, založené na funkciách určujúcich cenovú výhodnosť
- okamžitú komunikáciu o každej zmene s každým, kto by mal o nej vedieť.

Riadenie zmien si môžeme predstaviť ako uzavretý systém so spätnou väzbou, ktorý sa reálnom čase prispôsobuje meniacim sa vstupom a sledovaným cieľom. Cesta spätnej väzby obsahuje všetky zmeny, ktoré sa vyskytujú v systéme a ich dopad na systém.

Využitie tejto cesty tiež vyžaduje metriky, ktoré merajú množstvo a frekvenciu zmien – zmenené požiadavky, zmenené riadky programu a pod. Riadiaca funkcia zahŕňa vykonávanie rozhodnutí o povolení alebo zakázaní zmien. V prípade inkrementálneho vývoja softvéru, sa musíme rozhodovať či zaradiť zmenu do tejto alebo až do nasledujúcej verzie.

Bez plánu, procesu a informačného systému na riadenie zmien sa všetky projekty (snáď okrem veľmi malých a jednoduchých) vystavujú riziku straty kontroly na projektom a prekročenia časových limitov pre projekt.

Plánovanie systému pre zmenu

Spôsoby návrhu systému pre zmenu sú všeobecne známe a široko diskutované v literatúre – možno viac diskutované ako používané. Zahŕňajú v sebe vhodnú modularizáciu, podrobný a úplný opis rozhraní medzi modulmi a úplnú dokumentáciu.

Dobre premyslená otvorená architektúra redukuje dôsledky kritických zmien čŕt systému, ktoré sa môžu vyskytnúť neskôr. Zahŕňa znovupoužitie podsystémov, súčiastok z predchádzajúcich projektov alebo vonkajších zdrojov. Návrh, pri ktorom uvažujeme možné zmeny trvá dlhšie ako keď zmeny neuvažujeme. Tento čas navyše venovaný návrhu sa však môže niekoľkonásobne vrátiť počas ďalšieho vývoja a údržby produktu.

V súčasnosti sa veľa úsilia v softvérovej komunite venuje princípu znovupoužitia. Existuje snaha o čo najväčšie využitie už existujúcich softvérových súčiastok, ale veľa času a prostriedkov sa vynakladá aj na návrh a implementáciu nových znovupoužiteľných súčiastok.

Pojem znovupoužitia sa nevzťahuje len na samotné fyzické znovupoužitie existujúcich súčiastok, ale aj znovupoužitie získaných skúseností. Medzi najsľubnejšie teórie dnes patria vzory (architektonické – zaoberajú sa členením systémov na podsystémy, návrhové – riešia vzťahy a správanie sa objektov, a mnohé iné. Existujú aj vzory, ktoré sa zaoberajú zmenami, či už návrhom systému pre zmeny, alebo riešením situácii výskytu zmeny.).

Znovupoužitie úzko súvisí so zmenami systému. Znovupoužitím súčiastky môžeme „použiť“ aj chyby, ktoré obsahuje. V prípade, ak sa takéto chyby nájdu a opravia, treba vymeniť všetky inštancie chybnej súčiastky za správnu. Toto môže niekedy znamenať veľký problém. Jedným z možných, často využívaným riešením je rozdelenie výsledného produktu na viac samostatných častí, ktoré spolu komunikujú cez presne definované rozhrania. V takomto prípade možno pri zmene súčiastky (a pri zachovaní rozhrania) jednoducho vymeniť danú časť (často súbor) za novú.

Nekonečný vývoj

Program sa neprestáva meniť ani po jeho odovzdaní zákazníkovi. Zmeny po odovzdaní spadajú do údržby. Údržbu softvéru tvoria najmä opravy návrhových chýb, pridávanie nových funkcií do systému a vylepšovanie vlastností systému.

Celkové náklady na údržbu široko používaného programu obyčajne predstavujú 40 % alebo aj viac celkových nákladov vynaložených počas života softvérového systému [1]. Tieto náklady závisia aj od množstva používateľov systému – viac používateľov nájde viac chýb.

Betty Campbell poukázala na cyklus výskytu chýb v programe (pozri obr. 2). Na začiatku, po odovzdaní novej verzie programu, sa objavia chyby nových funkcií pridaných do novej verzie a chyby, ktoré vznikli pri oprave chýb v predchádzajúcej verzii.

Po vyriešení týchto chýb nasleduje niekoľko mesiacov, v ktorých je výskyt chýb relatívne malý. Potom výskyt chýb znovu začne rásť. Je to spôsobené príchodom nových používateľov s malou úrovňou znalosti systému, ktorí skúšaním čo všetko systém dokáže, objavia nové, predtým skryté chyby.

Základným problémom s údržbou programu je, že oprava chyby má veľkú šancu spôsobiť chybu(y) nové, ktoré bude treba skôr či neskôr tiež opraviť. Tieto chyby väčšinou spôsobuje neuvedomenie si, že oprava lokálnej chyby môže mať dosah na viaceré časti systému. Ak si aj uvedomíme možnosť ovplyvnenia iných častí systému, nemusíme poznať všetky závislosti.

Opravy chýb vyžadujú mohutnejšie testovanie ako testovanie programu po jeho prvej implementácii. Teoreticky, po oprave každej chyby by sa malo vykonať úplné testovanie systému, aby sme sa uistili, že táto oprava nespôsobila nové chyby systému.

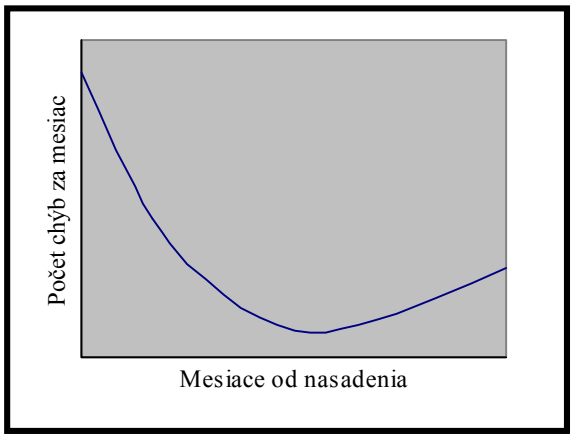
Všetky opravy majú tendenciu znehodnotiť pôvodnú štruktúru systému. Menej a menej úsilia sa venuje oprave chýb pôvodného systému, viac a viac sa venuje oprave chýb, ktoré spôsobili predchádzajúce opravy. Časom je systém čím ďalej, tým menej prehľadný. Skôr alebo neskôr bude nutné systém kompletne prepracovať.

Veľké systémy nikdy nie sú kompletné, neustále sa vyvíjajú. Systémy rastú ako sa pridávajú nové črty, rastie množstvo ohraničení, spolupracujú s viacerými inými systémami, podporujú viac používateľov a podobne. Tiež sa menia, pretože sa mení prostredie: prenášajú sa na iné platformy, prepisujú do nových programovacích jazykov.

Takáto sústavná zmena vedie k neustálemu zväčšovaniu zložitosti systému a pokračuje až pokiaľ nie je cenovo výhodnejšie nahradiť starý systém novým, alebo aspoň prepracovaným systémom.

Zmeny infraštruktúry organizácie

Organizačné štruktúry projektu sa musia meniť takisto ako sa mení systém. Treba zabezpečiť, aby všetci členovia vývojárskeho tímu boli v „obraze“.



Obr. 2: Výskyt chýb ako funkcia času
(Zdroj: Brown, 1995)

Zamedzí sa tým časovým stratám a opätovnému prepracúvaniu existujúcich častí systému.

Je vhodné použiť jednoducho prístupné centrálne úložisko, ktoré umožní všetkým členom tímu prístup k informáciám dôležitým pre ich prácu a jednoduchú obnovu dokumentov pri zachovaní synchronizácie medzi členmi tímu. Jedným z možných riešení je využitie vnútropodnikového intranetu.

Zmeny budú

Úspešný rast v súčasnosti významne závisí od schopnosti organizácie čo najrýchlejšie zlepšiť a rozšíriť jej produkty ako odpoveď na spätnú väzbu od používateľa. Ale keď sa organizácie snažia skrátiť cykly vývoja a zároveň držať krok s novými nápadmi a vývojom používateľských požiadaviek, javia sa zmeny ako nepriateľ rýchleho vývoja.

Kombinácia zložitosti, rýchlosti cyklov vývoja a odovzdávania systému a rýchlej zmeny vytvára hlavné riziko vývoja softvéru. Aby organizácia udržala krok s konkurenciou musí brať zmenu ako faktor, ktorý treba riadiť rovnako ako technológie, prostriedky a čas. Aj keď si zavedenie riadenia zmien vyžaduje úsilie, čas a investície, môže zrýchliť softvérový proces najmä zamedzením výskytu mnohých zdrojov rizík, nejednoznačností, chýb a znovuvytvárania existujúceho.

Ako sa softvéroví inžinieri snažia vytvárať udržovateľné produkty, vynára sa otázka či je vôbec možné vytvoriť takýto systém správne hneď na prvýkrát. Ak aj použijeme silne súdržné a voľne viazané softvérové súčiastky, ak aj máme úplnú a aktuálnu dokumentáciu, ešte stále budeme potrebovať fázu údržby. Dôvod je v prirodzenosti samotného systému. Nikdy nebudeme mať istotu, že sa systém už nebude meniť. Musíme predpokladať,

Dobre premyslená otvorená architektúra redukuje dôsledky zmien systému, ktoré sa vyskytnú neskôr.

že sa bude meniť a vytvoriť ho tak, aby sa dal meniť jednoducho.

Literatúra

1. F.P. Brooks. The Mythical Man-Month. Anniversary Edition, Addison-Wesley, 1995.
2. G. Forte. Managing Change for Rapid Development. IEEE Software, May/June 1997, 120-122.
3. S.L. Pfleeger. The Nature of System Change. IEEE Software, May/June 1998, 87-90.
4. A.I. Wasserman. Toward a Discipline of Software Engineering. IEEE Software, Nov. 1996, 23-31.

Meškáme, čo s tým?

Marek KOČAN

Abstrakt. Táto práca podáva pohľad na problematiku meškania úspešného ukončenia softvérového projektu, na metódy jeho detekcie, príčiny jeho vzniku. Diskutuje opatrenia, ktoré zabránenia meškaniu, či už počas vývoja projektu alebo po jeho skončení. Hlavným podkladom pre túto prácu je kniha Freda Brooksa [1], ktorá sa zaoberá problémami veľkých softvérových projektov. Napriek tomu, že prvé vydanie tejto knihy vyšlo pred viac ako 20 rokmi, nestráca na aktuálnosti ani dnes.

Ako dochádza k meškaniu projektu?

Softvérové spoločnosti sa dnes dostávajú do veľkého tlaku, keď majú za málo peňazí a krátky čas vyvinúť dokonalý softvérový systém. Pod týmto tlakom a tlakom konkurencie organizácie skracujú čas vývoja softvérových systémov. Následkom je, že sa dostávajú do časového sklzu a s ním súvisiacich problémov. Ako dochádza k meškaniu projektu?

Pri vývoji softvérového systému môže dôjsť ku dvom druhom časového sklzu. Prvý, spôsobený väčšími "katastrofami", je ľahšie pozorovateľný. Nastáva pri udalostiach, kde sú okamžite známe dôsledky – projekt sa dostane do časového sklzu. Vtedy sa musia rýchlo vykonať potrebné protiopatrenia. Odpoveďou budú rázne opatrenia, napríklad reorganizácia tímu, spôsobu práce, komunikácie alebo aj výmena nekompetentných ľudí.

Druhý druh časového sklzu, ktorý sa kumuluje postupne a nenápadne po dňoch, sa omnoho ťažšie zisťuje, ťažšie sa proti nemu bráni a ťažšie sa dá zvládnuť. Napríklad včera sa neuskutočnilo dôležité stretnutie pretože kľúčový človek bol chorý; dnes sa nedostavila polovica zamestnancov do práce, pretože hromadná doprava štrajkuje a zajtra sa nebude môcť začať testovať, lebo ešte nie sú k dispozícii testovacie stroje. Počasie, rodinné problémy, stretnutia so zákazníkom a pod., sú udalosti, ktoré posúvajú ďalšie činnosti o deň, dva, tri a projekt sa dostáva do časového sklzu.

Ako zistíme, že projekt meška?

Rozvrh

Ak chceme zistiť, či projekt meška, musíme mať určitý časový plán – rozvrh. Rozvrh je zoznam úloh, míľnikov (kontrolných bodov), ktoré treba splniť v určenom čase. Voľba dátumov, resp. času plnenia, je náročná úloha, ktorá záleží najmä od skúseností ľudí, ktorí rozvrh vytvárajú. Kontrolné body rozvrhu musia byť jednoznačne a presne definované udalosti. Napríklad: "Podpísanie špecifikácie návrhármi a programátormi", "Program prešiel všetkými predpísanými testami". V prípade, ak by neboli takto definované, mohli by vzniknúť nepresnosti a nejasnosti v ich interpretácii a ťažšie by sa určovalo, či projekt ide podľa plánu alebo nie.

Ďalším dôvodom, prečo musia byť kontrolné body presne a jednoznačne definované je, že postup projektu sa nedá falšovať. Presný a jednoznačný rozvrh je vlastne pomocou pre celý vývojový tím. Tím vie presne čo, odkedy a dokedy treba urobiť. Nepresne a nejasne zvolený rozvrh vyvoláva nejednoznačnosť v jeho interpretácii a vedie k rôznym konfliktom, ktoré potom spôsobujú ďalší sklz projektu.

Sklz projektu

Čo robiť, ak meškáme o jeden deň? Doženieme to neskôr alebo je to jedno, pretože tá časť, do ktorej patrí náš modul meška tiež? Ako rozlíšiť, či je nutné dobehnúť časový sklz alebo nie?

Otázka sa môže na prvý pohľad zdať zbytočná, lebo ak máme časový sklz, tak by sme sa ho mali určite snažiť vyrovnať. Lenže nie každý sklz v rozvrhu znamená katastrofu a sklz celého projektu. Ako teda určíme, či náš sklz znamená katastrofu alebo nie? Najvhodnejšími metódami, ktoré nám to umožňujú je PERT diagram (angl. Program Evaluation and Review Technique) alebo CPM (angl. Critical Path Method). (Prehľad metód vytvárania rozvrhu poskytuje napr. [2]). Obidve metódy sa používajú na zistenie minimálneho času trvania projektu a kritickej cesty. Kritická cesta obsahuje tie činnosti, ktorých oneskorením sa oneskorí celý projekt. Metódy ďalej ukazujú aj o koľko sa môže určitá činnosť oneskoriť, kým sa stane súčasťou kritickej cesty.

Kontrolné body rozvrhu musia byť jednoznačne a presne definované udalosti.

Pri vytváraní PERT diagramu sa určuje sieť činností projektu, hľadajú sa závislosti medzi činnosťami a odhaduje sa ich dĺžka trvania. Počas práce na projekte dáva PERT diagram odpoveď na otázky ako môžeme oneskorenie jednej časti projektu vynahradiť v inej časti a či nám oneskorenie niektorej časti ovplyvní aj trvanie celého projektu – či sa úloha nachádza na kritickej ceste.

Manažér a nadriadený

Manažér tímu málokedy hneď uteká za nadriadeným, ak jeho tím meška. Dôvodov na jeho konanie je hneď viacero. Jeho tím môže byť schopný situáciu ešte zvládnuť alebo dokáže niečo vymyslieť, či reorganizovať tím tak, aby vyriešil daný problém. Potom nie je dôvod na to, aby išiel ihneď za nadriadeným, pretože funkcia manažéra projektu je práve na to, aby riešil takéto problémy.

Na druhej strane aj nadriadený sleduje ako projekt postupuje. Ak sa dozvie od manažéra, že projekt meška, môže sa stať, že začne riešiť problémy on a tým bude konať namiesto manažéra. Takéto konanie môže znížiť autoritu a opodstatnenosť roly manažé-

ra. Nastáva konflikt úloh medzi manažérom a jeho nadriadeným. Samozrejme, na druhej strane manažér nemôže všetko vyriešiť sám (na všetko nemá ani kompetencie) a nastanú situácie, kedy do problému musí vstúpiť nadriadený.

Nadriadený musí vedieť rozlišovať medzi informáciami o stave projektu a informáciami o riešení aktuálneho stavu, ktoré mu poskytujú manažéri. Nadriadený nesmie konať, ak manažér vie vyriešiť daný problém a nesmie nikdy konať, ak si prezerá informácie o stave projektu. Naopak, ak manažér vie, že jeho nadriadený bude akceptovať správu o stave projektu a nebude konať namiesto neho, tak aj on bude dávať do správy čestné a poctivé informácie. Vhodným riešením je označovanie stretnutí ako *informácia o stave projektu*, kde sa informuje iba o stave a *riešenie aktuálneho stavu projektu*, kde sa stanovujú konkrétne akcie a opatrenia. Riešenie aktuálneho stavu môže byť následkom informácie o stave projektu, ak si nadriadený myslí, že problém sa vymkol kontrole alebo že ho manažér nezvládne. Výhodou tohoto prístupu je, že každý vie na čom je a o čom sa rokuje.

Napriek takémuto riešeniu je dôležité, aby existoval prehľad stavu projektu, ktorý reprezentuje skutočný stav riešenia. Ako základ pre jeho vytvorenie môže slúžiť PERT diagram. Takýto prehľad by mal obsahovať jednotlivé kontrolné body činností a dátumy ich ukončenia. Z takejto správy sa dá ľahko zistiť, čo v projekte zaostáva a čo ide podľa plánu. Ak sa v tejto správe vyskytnú činnosti, ktoré zaostávajú, manažér musí vedieť vysvetliť prečo zaostávajú, určiť nový koniec činnosti, ako tento problém hodlá riešiť a akú pomoc pri tom potrebuje od nadriadeného alebo od iných pracovných skupín.

PERT diagram vytvára nadriadený a jeho manažéri projektov, ktorí mu podávajú správy. Počas vývoja projektu si PERT diagram vyžaduje aktualizáciu a revíziu, ktorú môže zabezpečovať malá skupina ľudí, ktorá podlieha nadriadenému. Jej úlohou nie je nič iné, iba zbierať informácie od manažérov, či už určili činnosti a ich termíny začatia a ukončenia. Táto skupina odoberá prácu manažérom a tí sa potom môžu venovať tomu základnému a to hľadaniu riešení problémov a riadeniu projektu.

Aké sú možné riešenia meškania projektu?

Aké sú možné riešenia, ak projekt začína zaostávať za rozvrhom? Aké ponaučenia si zoberieme z takéhoto projektu, aby sme sa v budúcnosti vyvarovali oneskorenia projektu? Tieto otázky boli položené niekoľkým firmám v Anglicku a ich odpovede rozoberáme v ďalších riadkoch [3].

Riešenia počas projektu

Na otázku, ako riešili zaostávanie projektu, uviedli spoločnosti odpovede uvedené v tab.1 [3].

Ako vidieť z tabuľky medzi najčastejšie používané opatrenie patrí *predĺženie doby trvania činností*, teda natiahnutie časového rozvrhu projektu. Toto opatrenie môže vychádzať z dvoch predpokladov. Produktivita vývojárov je príliš nízka alebo (čo býva častejšie) časový rozvrh bol nadhodnotený. Predĺženie doby trvania projektu je jednoduchá záležitosť, ale iba ak nebol rozvrh podstatnou súčasťou projektu (zmluvy). Ale ak je rozvrh podstatnou súčasťou projektu, napríklad: ak sa výsledok projektu integruje do iného projektu, tak natiahnutie rozvrhu bude pravdepodobne dosť problematické.

1.	Predĺženie doby trvania činností a projektu	85 %
2.	Zlepšenie manažérskych postupov	54 %
3.	Pridanie ľudí	53 %
4.	Pridanie peňazí	43 %
5.	Tlak na dodávateľov zadržiavaním platieb	38 %
6.	Redukcia rozsahu projektu	28 %
7.	Pomoc zvonku	27 %
8.	Zlepšenie vývojových metód	25 %
9.	Tlak na dodávateľov hrozbou súdu	20 %
10.	Zmena technológie	13 %
11.	Prerušenie projektu	9 %
12.	Iné	9 %

Tab.1: Prehľad činností vykonaných počas projektu pri zaostávaní za rozvrhom.

Druhé najčastejšie používané opatrenie je *zlepšenie manažérskych postupov*. Predpokladom pre toto opatrenie je, že sa doteraz používali nevhodné metódy a postupy riadenia alebo sú nesprávni ľudia na manažérskych miestach. Toto opatrenie, ako uvidíme neskôr, patrí medzi najčastejšie používané opatrenie proti sklzu ďalšieho projektu v budúcnosti.

Medzi ďalšie, viac-menej intuitívne opatrenia, patrí *pridanie ľudí a peňazí*. Pridanie ľudí je síce jednoduchá reakcia na sklz projektu, ktorá však prináša aj dosť nevýhod. Pri pridaní ľudí do projektu môže nastať jeho ďalšie oneskorenie z dôvodu nutnosti zaškolenia nových ľudí a zapracovania ich do problematiky. Zrýchlenie práce je reálne vtedy, ak noví ľudia poznajú problematiku. Na druhej strane, pridanie ľudí implikuje ďalšie opatrenie a tým je zvýšenie spotreby peňazí. V skutočnosti všetky uvedené opatrenia, až na dve – redukcia rozsahu projektu a prerušenie projektu, zvyšujú náklady na projekt.

Opatrenia pomocou *tlaku na dodávateľov zadržiavaním platieb a hrozbami súdu* sú reakciou na neplnenie zmlúv dodávateľských firiem. Ak niektoré firmy nedodajú načas dohodnuté práce, tak to vedie k nechcenému predlžovaniu projektu, ktoré nevie projektový tím ovplyvniť. Zaujímavé je, podľa výskumov uvedených v [3], že v minulosti (1989) nedochádzalo k takýmto konfliktom medzi organizáciami. Môže to spôsobiť napríklad aj rýchly rozvoj technológie, zvýšený dopyt po softvérových systémoch, silnejúca konkurencia na trhu.

Redukcia rozsahu projektu. Toto relatívne jednoduché opatrenie sa nepoužíva často. Väčšinou možno ustúpiť od určitých požiadaviek a tým zabrániť oneskoreniu celého projektu. Ale asi väčšina zákazníkov nemôže, resp. nechce ustúpiť od požiadaviek projektu. Toto opatrenie sa zrejme používa ako krajné riešenie, až keď zlyhajú iné opatrenia.

V súčasnosti je zvýšený nárast dopytu po *externej pomoci* pri vývoji. Ide najmä o firmy, ktoré ponúkajú konzultačné služby, postupy, riešenia. Tímy si najímajú ľudí z týchto firiem, ktorí pomáhajú riešiť projekt. Toto opatrenie vyvoláva niektoré už analyzované opatrenia a to pridanie ľudí, zvýšenie nákladov alebo aj spotrebu väčšieho množstva času – natiahnutie rozvrhu.

Použitie lepších vývojových metód a zmena technológie súvisí priamo s opatrením zlepšenia manažérskych postupov. Tieto opatrenia môžu priniesť veľké problémy, najmä ak sa použijú na rozbehnutom projekte. Podľa mňa, vhodný moment na zmenu vývojových metód alebo technológie je na začiatku projektu. Takéto zmeny si totiž vyžadujú preškolenie tímu, zvládnutie nových prostriedkov. Ak sa toto opatrenie vykonáva počas riešenia projektu, tak môže viesť k výraznejšiemu zvýšeniu nákladov, predĺženiu rozvrhu projektu a dokonca až k neúspechu celého projektu.

Posledným spomínaným opatrením je *prerušenie projektu*. Je zaujímavé, že jeho výskyt je dosť nízky. Z toho vyplýva, že väčšina meškajúcich projektov zásadne nekončí vždy jeho predčasným ukončením. V zásade sa toto opatrenie používa až ako posledná možnosť ako zabrániť ešte väčším stratám.

Aby projekt nezaostával

Tab. 2 uvádza činnosti, ktoré uviedli opýtané firmy na otázku, aké zmeny vykonali, aby sa vyhli problémom s oneskorením dokončenia projektu [3].

1.	Zlepšenie projektového manažmentu	86 %
2.	Štúdia vhodnosti	84 %
3.	Väčšie zapojenie používateľa do projektu	68 %
4.	Viac externej pomoci	56 %
5.	Ani jedno z uvedených	4 %

Tab.2: Prehľad opatrení vykonaných po skončení projektu, ktorý zaostával za rozvrhom.

V tejto tabuľke si môžeme všimnúť viaceré zaujímavé rozdiely. Prvým rozdielom je 11 opatrení pri riešení zaostávania projektu počas jeho riešenia a iba 4 opatrenia po jeho skončení. Samozrejme, že niektoré opatrenia vykonané počas projektu nemajú zmysel po jeho skončení. Viac ľudí, viac peňazí, tlak na dodávateľov sú opatrenia týkajúce sa iba aktuálneho projektu.

Druhým zaujímavým momentom je, že opatrenia týkajúce sa zlepšenia vývojových metód alebo zmien technológie, ktoré sa uplatňujú počas projektu, sa neobjavujú v tab. 2. Vyzerá to, akoby problémy s vývojovými metódami a technológiou boli otázkami daného okamihu riešenia projektu a

Najvhodnejší moment na zmenu metód a technológie je po skončení projektu, keď sa niektoré postupy alebo metódy ukázali ako nevhodné.

potom sa na to akosi pozabudlo, pretože to nie je po skončení projektu až tak viditeľné. Myslím, že najvhodnejší moment na zmenu metód a technológie je práve po skončení projektu, kde sa niektoré postupy alebo metódy ukázali ako nevhodné. Cena týchto zmien hneď na začiatku nového projektu je určite nižšia ako cena zmeny počas rozbehnutého projektu, keď sa časť práce vykonala starými metódami. Prepracovanie môže byť omnoho drahšie.

Iný, veľmi zaujímavý rozdiel je zvýšenie zainteresovanosti používateľa. Toto opatrenie sa nenachádza medzi opatreniami vykonanými počas riešenia projektu, i keď niektoré by sa nemali aplikovať bez používateľa (napríklad zmenšenie rozsahu projektu).

Medzi často používané dlhodobé opatrenia na zabránenie omeškania projektu patrí *štúdia vhodnosti*. Toto opatrenie môže vyplývať z dvoch skutočností. Prvá je tá, že štúdia vhodnosti nebola rozpracovaná vôbec a druhá, že bola vypracovaná len povrchne. Pravdepodobne bol na začiatku projektu až príliš veľký optimizmus a manažment si zvolil racionálnejší prístup do budúcnosti.

V obidvoch tabuľkách sa na poprednom mieste nachádza jedno opatrenie – *zlepšenie manažmentu projektu.* Manažment projektu je zrejme, podľa spoločností vyvíjajúcich softvérové systémy, vo veľkej miere zodpovedný za oneskorenia softvérových projektov. Z toho vyplýva, že súčasné smery na zabránenie predlžovania termínu dokončenia projektov sú v zlepšovaní manažérskych postupov.

Literatúra

1. F.P. Brooks. The Mythical Man-Month: Essays on Software Engineering. Anniversary Edition, Addison Wesley, 1995.
2. N. Noel. Harroff. Scheduling in a Nutshell, http:://nnh.com/ev/nut2.html (marec 1998).
3. R.L. Glass. Short-Term and Long-Term Remedies for RunAway Projects, Comunications of the ACM, Vol. 41, No. 7, July 1998, 13-15.

Cena človeka

Radovan SEMANČÍK

Abstrakt. Reakcia na esej „Ten Pounds in a Five-Pound Sack“ od F.P. Brooksa. Reakcia na platnosť alebo skôr neplatnosť spomenutých faktov po viac ako 20-tich rokoch. Úvahy o postavení dnešných softvérových inžinierov, cene ich práce a korelácií vývoja ceny softvérových inžinierov a Moorovho zákona.

Úvod

Technológia postupuje dopredu míľovými krokmi. Gordon Moore roku 1965 zverejnil zaujímavé pozo-

rovanie, že počet tranzistorov integrovaných obvodov sa zdvojnásobuje zhruba každých 18 mesiacov [2]. Toto pozorovanie je dnes známe ako Moorov zákon a naznačuje trend rýchleho rozvoja hardvérových základov počítačov. Kapacita pamätí sa zdvojnásobí každých 18 mesiacov, výkon procesorov sa zdvojnásobí každých 18 mesiacov a celkový výkon počítačových systémov takto veľmi prudko narastá. Prudkým rozvojom výroby osobných počítačov klesá cena ich komponentov a takto sa hardvérové prostriedky stávajú široko dostupné.

Cena priemernej počítačovej zostavy zostáva stále konštantná v čase (Tak tvrdí Machronov zákon; podľa neho je cena priemernej zostavy asi $5 000), aj keď výkon počítačov rastie exponenciálne. Pred zhruba 20-timi rokmi bolo 2 MB operačnej pamäte veľmi nákladné a programátori museli šetriť každým kilobajtom pamäte. Dnes sa za minimálnu hranicu pre osobný počítač považuje 32 MB operačnej pamäte. Výkon procesorov tiež stúpa po veľkých krokoch. Taktovacie frekvencie dnes dosahujú hranice 300 – 400 MHz. So zvyšovaním taktovacích frekvencií sa však približuje hranica rýchlosti svetla. Rýchlosť svetla časom nerastie, preto sa musí prispôsobovať dráha, ktorú signály musia prejsť [3].

Zmenšovanie procesorov a počítačov vôbec nie je len kvôli pohodliu ich používateľov. Aj napriek hranici rýchlosti svetla sa očakáva, že Moorov zákon zostane v platnosti ešte niekoľko desiatok rokov.

Softvérový inžinier

Spolu so zvyšovaním výkonu hardvérových systémov rastie aj zložitosť softvérových systémov, ktoré hardvér využívajú. Treba väčšie množstvo dobre vyškolených a kvalifikovaných odborníkov. Počet odborníkov však nestúpa rovnako ako počet tranzistorov na čipe. Školenie odborníkov je časovo náročné a samotný čas štúdia, ktorý musia venovať problematike, sa predlžuje.

V minulosti bolo bežné, že odborník na niektorý typ počítača poznal do detailov hardvér počítača, operačný systém aj bežné aplikačné systémy počítača. Dnes je už len samotný hardvér počítačových systémov nad možnosti jedného človeka. So softvérovými systémami je to podobne.

Keďže výkon počítačových systémov rastie exponenciálne a dostupné ľudské zdroje sotva lineárne, vývojári už nie sú naďalej schopní využívať hardvérové prostriedky naplno. Značná časť výkonu počítačových systémov sa „obetuje“ v prospech zníženia zložitosti riešenia, modifikovateľnosti, viditeľnosti a podobných vlastností softvérových systémov. Tieto vlastnosti napomáhajú jednoduchšej práci vývojárov a tým podstatne redukujú náklady na vývoj a prevádzku aplikácií. Dnes je odborná ľudská práca podstatne nákladnejšia ako hardvérové prostriedky.

Plytvanie výkonom

Dôležité je, aby sa prostriedkami určenými na zjednodušenie riešenia neplytvalo. Jedným príkladom z praxe môže byť použitie výkonného servra na báze procesora Pentium II ako DNS servra pre malú sieť obsahujúcu asi 30 počítačov. Neskôr, keď firma potrebovala výkonný testovací server, nahradili tento „nafúknutý“ DNS server starou pracovnou stanicou s procesorom i486, ktorá nemala ani desatinový výkon oproti pôvodnému serveru. Aj táto stará pracovná stanica však bola vo svojich výkonových špičkách zaťažená asi na 20 %. Takýto zlý odhad potrebných zdrojov takmer viedol k desaťnásobnému nadhodnoteniu potrebných finančných prostriedkov. Návrhári však neboli nútení o vhodnosti umiestnenia servra rozmýšľať, pretože server bol „štandardného typu“ a jeho parametre boli v súlade s požiadavkami operačného systému. Nie je nutné šetriť, keď všetkého je dosť.

Softvérový inžinier sa dostáva do pozície strategického objektu a manažment práce softvérových inžinierov sa radí medzi strategické funkcie každého výrobcu a prevádzkovateľa softvéru.

Podobný prístup môžeme pozorovať aj pri tvorbe softvérových systémov. Pre programátora je oveľa jednoduchšie použiť veľkú knižnicu, aj keď z nej využije len jedinú funkciu. Pamäte je dosť a použitím knižnice sa skráti čas vývoja a tým aj náklady na tvorbu aplikácie. Málokto sa dnes zaoberá optimálnym využitím prostriedkov. Pripadá nám normálne, že každá nová verzia programu vyžaduje viac miesta na disku, výkonnejší procesor, viac pamäte.

Toto správanie vystihuje ďalší zákon: Parkinsonov zákon o údajoch. Tento zákon hovorí, že údaje sa budú rozširovať až kým nezaplnia akékoľvek miesto určené na ich uloženie. Skúste si spomenúť koľko máte voľného miesta na disku vašej pracovnej stanice. Väčšina úložných médií počítačov je zaplnených takmer na maximum. Obdobne je to aj so softvérovými systémami. Vývojári dokážu naplno zahltiť prostriedky akéhokoľvek hardvérového systému.

Softvérové systémy sú dimenzované podľa hardvéru, ktorý zákazník vlastní alebo ktorý sme schopní spolu so softvérovým systémom dodať a podľa požiadaviek samotného softvérového systému. Rovnako bežné je, že počas vývoja softvérového systému sa zistí, že na jeho prevádzku bude treba ďaleko výkonnejší hardvér ako sa pôvodne plánovalo. Na šetrenie prostriedkami nikto nemá dosť ľudí.

Tento trend sa ukazuje aj na cene hardvéru a softvéru. V posledných niekoľkých rokoch cena softvérových systémov niekoľkonásobne prevýšila cenu hardvéru. Tento trend sa týka generického aplikačného softvéru a aj zákazníckych softvérových systémov. To znamená, že cena práce softvérových inžinierov a programátorov je drahšia ako hardvérové prostriedky. Preto sa oplatí skrátiť dobu realizácie projektu alebo jeho personálnu náročnosť na úkor efektivity využitia hardvérových prostriedkov.

Okrem veľkého pokroku v oblasti počítačového hardvéru existuje aj ďalší faktor, ktorý ovplyvňuje

prácu softvérových inžinierov. Celková koncepcia a štruktúra informačných systémov sa mení. Od pôvodných, silne centralizovaných, informačných systémov založených na veľkých počítačoch a terminálových sieťach sa vývoj posunul k distribuovaným systémom založeným na globálnych sieťových technológiách. Softvéroví inžinieri nie sú školení na rýchle zvládnutie týchto nových technológií, preto nie sú v krátkom čase schopní využiť ich výhody.

Aj v dnešných časoch veľkých podnikových sietí a Internetu nie sú zriedkavé aplikácie, ktoré využívajú tieto siete len na transport terminálovej relácie s výkonným centrálnym počítačom, degradujúc celú multimediálnu sieť len na inteligentnú terminálovú sieť. Návrhári softvérových systémov sa ešte úplne neprispôsobili novým podmienkam.

Na jeden problém navyše existuje niekoľko rôznych riešení, ktoré sa nezriedka líšia len nekompatibilnými formátmi správ, protokolov alebo súborov. Zorientovať sa v množstve rôznych riešení a technológií je časovo nesmierne náročné. Softvéroví inžinieri sa preto väčšinou držia riešení od jedného dodávateľa, ktorý je často aj dodávateľom hardvéru a operačného systému. Takto sa riešitelia stávajú otrokmi svojich dodávateľov. Samotný riešiteľ nakoniec ani nevie ako systém, ktorý integroval, pracuje. On vie iba to, z akých súčiastok sa systém skladá a čo jednotlivé súčiastky navonok robia. Chýba tu hlbšie pochopenie funkcie jednotlivých zložiek systému a aj systému ako celku.

Otvorené systémy podobne ako štandardizované systémy napomáhajú spolupráci počítačových systémov.

Štandardy a otvorené systémy

Aspoň čiastočným riešením tohoto problému je štandardizácia a vývoj „otvorených systémov“. Štandardizačná snaha vedie k zjednoteniu technológií v snahe zaručiť ich vzájomnú spoluprácu a kompatibilitu. Samotný proces štandardizácie je však pomalý a nie vždy vedie k výsledkom použiteľným v praxi. Napríklad protokoly rodiny ISO OSI sa rozšírili len veľmi málo aj napriek ich štandardizácii. Na druhej strane veľa oficiálne neštandardizovaných systémov sa stala *de facto* štandardami. Medzi takéto „priemyselné štandardy“ patrí napríklad sada AT príkazov modemov firmy HAYES.

Otvorené systémy podobne ako štandardizované systémy napomáhajú spolupráci počítačových systémov. Otvorené systémy majú dobre špecifikovanú architektúru a rozhrania a predpokladajú začleňovanie súčiastok od iných výrobcov ako je autor systému. Vrcholom v „otvorenosti“ systémov sú voľne šíriteľné systémy a systémy s otvoreným zdrojovým textom programu (angl. open-source). Tieto systémy umožňujú vývojárom do podrobností pochopiť ich funkciu a vytvoriť spolupracujúce softvérové súčiastky. Navyše odberateľ takýchto systémov nezávisí od dodávateľa, keďže vlastní zdrojový program.

Aj tie najotvorenejšie systémy však potrebujú na svoje efektívne využitie človeka. Človeka, ktorý im porozumie, dokáže využiť ich dobré vlastnosti a vyhnúť sa tým zlým. Týmto človekom je softvérový inžinier. Organizácia, ktorá takéhoto človeka zamestnáva by mu mala poskytnúť dostatočné podmienky na prácu a zabezpečiť jeho stály odborný rast. Kvalitný softvérový inžinier je najlepšia investícia softvérovej firmy.

Cena kvalifikovaného odborníka je podstatne vyššia ako hardvérového systému, na ktorom pracuje. Táto podstatná zmena oproti minulosti značne ovplyvní vývoj budúcich projektov. Nástup nových technológií ešte viac vyzdvihne cenu inžiniera, ktorý touto technológiou vládne. Takáto situácia stavia softvérového inžiniera do pozície strategického objektu a manažment práce softvérových inžinierov radí medzi strategické funkcie každého výrobcu a prevádzkovateľa softvéru.

Literatúra

1. F.P. Brooks. The Mythical Man-Month: Essays on Software Engineering. Addison Wesley, 1995.
2. Jargon Lexicon. Moore's Law, Parkinson's Law of Data.
3. E. Schmidt. The Future of Networks and Computers. SUN Microsystems.
4. Processor hall of fame: What is Moore's Law?, Intel corporation.

Hra na slovíčka

Ivan TARAPČÍK

Abstrakt. Komunikácia patrí k najdôležitejším zložkám práce v tíme. V dnešnej dobe, keď sa vývoj softvéru zrýchľuje, treba zavádzať moderné a efektívne metódy komunikácie medzi členmi tímu ako aj medzi vývojármi a zákazníkmi. V eseji sa zamýšľam nad spôsobmi komunikácie v prípade rôznych typov softvérových produktov a nad dôležitosťou presnej špecifikácie a jej pochopenia všetkými členmi vývojárskeho tímu.

Úvod

Existuje mnoho metód, ktoré sa zaoberajú životným cyklom softvérového systému. Keď nebudeme brať do úvahy chaotický vývoj, vhodný iba pre maličké programy, ktoré vytvárajú maximálne traja ľudia, je pri všetkých metódach veľmi dôležitá komunikácia. Dobrý architektonický návrh je nanič, keď ho programátori nepochopia a nezrealizujú. Komunikácia je dôležitá na všetkých úrovniach riadenia projektu a vo všetkých smeroch.

Bohužiaľ, o praktických metódach komunikácie medzi ľuďmi a skúsenostiach s nimi sa príliš veľa nehovorí. Toto tajomstvo si, zdá sa, väčšina firiem uchováva – buď ako konkurenčnú výhodu alebo preto, že komunikácie vo firme nefunguje zďaleka tak, ako by mala.

Rôzne produkty, rôzne riešenia

Nie je nepodstatné aký softvér vyvíjame a akým spôsobom. Od toho či vytvárame softvérový systém na zákazku, generický systém, hromadne používaný softvér alebo systém s otvoreným zdrojovým textom (angl. open-source software), budú závisieť metódy a smery komunikácie v tíme aj navonok. Spôsob komunikácie sa pre jednotlivé typy softvérových systémov mení z najformálnejšej v prípade zákazníckych riešení po úplne neformálnu v prípade otvorených systémov.

Proces tvorby zákazníckeho softvéru patrí k najprepracovanejším oblastiam softvérového inžinierstva. Väčšina odborných štúdií prakticky neuvažuje o tom, že by mohol existovať vývoj softvéru inak ako na požiadavku zákazníka a hromadnú produkciu považuje iba za špeciálny prípad, keď zákazníkom je trh – akýsi priemerný šedivý používateľ výsledného produktu. Vývoj na zákazku začína uzavretím kontraktu, špecifikáciou a jej ďalším rozvinutím, návrhom a implementovaním. Tento postup má niekoľko nevýhod. Stručne možno povedať, že je nepružný.

V úvodných fázach sa urobí analýza požiadaviek a podrobná špecifikácia, dohodnú sa termíny a uzavrú sa zmluvy. Akákoľvek neskoršia zmena do špecifikácie – aj keď sa ukáže ako rozumná a pre obe strany akceptovateľná – si vyžaduje zložitý administratívny a komunikačný proces. Komunikácia v tomto prípade prebieha predovšetkým zhora nadol, pričom k iteráciám dochádza najmä lokálne – samostatne pri špecifikácii, návrhu a implementácii. Takto sa ľahko môže stať, že programátor nepochopí niektorú myšlienku architekta, čím sa do systému vnášajú chyby a niekedy ťažko odstrániteľné nevhodné riešenia problémov. Bohužiaľ, takáto striktnosť je v tomto prípade napriek svojim nevýhodám zrejme nutná, pretože zvoľnenie týchto postupov môže viesť k nedodržaniu termínov a nespokojnosti zákazníka aj dodávateľa.

Programy vyvíjané bazárovým spôsobom v niektorých prípadoch napredujú rýchlejšie ako ich komerčné ekvivalenty vďaka silnému paralelizmu najmä pri testovaní a odstraňovaní chýb.

Iná situácia je v prípade vývoja generického a hromadne používaného softvéru. V tomto prípade sa požiadavky a špecifikácia tvoria priamo v tíme, ktorý produkt vyvíja. Väčšinou neplatia také prísne časové a nákladové ohraničenia. Tieto faktory umožňujú zaviesť omnoho pružnejšie modely komunikácie v tíme. V tomto prípade možno ľahšie zabezpečiť celkové pochopenie systému a súvislostí medzi jeho jednotlivými súčiastkami v rámci celého tímu.

Dobrý príklad na takýto spôsob vývoja dáva esej „Passing the Word“ v knihe [1] a článok Toma van Vlecka [3]. V oboch prípadoch sa kladie dôraz na súlad špecifikácie a výsledného produktu, pochopenie dôsledkov zmien všetkými členmi tímu a na vyriešenie rozporov medzi rôznymi myšlienkovými prúdmi v rámci tímu. Základom úspešného dokončenia produktu je presná špecifikácia spolu s rýchlym, pružným a dokumentovaným procesom zmeny špecifikácie v prípade výskytu chýb alebo návrhu nových funkčností.

Úplne odlišné podmienky platia v prípade bazárového vývoja softvéru. Tento termín použil Raymond [2]. Označuje ním špeciálny „režim“ vývoja otvorených systémov, pri ktorom sa používa (do určitej miery) živelný princíp vymieňania programu medzi jednotlivými vývojármi. Vývojárom sa môže stať prakticky ktokoľvek. Raymond vo svojom článku porovnáva vývoj softvéru klasickou metódou (Cathedral), kde je uzavretá skupina vývojárov, komunikujúca sporadicky s používateľmi s bazárovou metódou, pri ktorej sa softvér čím skôr uvoľní do používania, pričom hocikto môže upozorniť na chyby, poslať ich opravy alebo navrhnúť úplne nové funkčnosti.

Úspech bazárového vývoja softvéru spôsobuje najmä vysoká motivácia jeho tvorcov.

V celom tomto procese je dôležitá osoba koordinátora. Jeho úlohou je začleňovať opravy a nové funkčnosti do produktu alebo vyberať z viacerých alternatív. Pri takomto spôsobe vývoja softvéru sa prakticky nedodržiavajú klasické postupy softvérového inžinierstva. Produkt sa vyvíja živelne, pričom komunikácia prebieha verejne prostredníctvom internetu. Požiadavky a špecifikáciu tvoria priebežne používatelia, ktorí sú mnohokrát aj vývojármi alebo aspoň testermi produktu.

Podľa klasických poučiek softvérového inžinierstva nemôže takýto postup fungovať pre väčšie systémy. Prax však na viacerých projektoch (napríklad Linux, Gnome a ďalšie projekty väčšinou pod záštitou GNU) ukázala, že poučky neplatia absolútne. Pri vývoji dochádza k silnému paralelizmu najmä pri testovaní produktu a odstraňovaní chýb, takže programy vyvíjané bazárovým spôsobom v niektorých prípadoch napredujú rýchlejšie ako ich komerčné ekvivalenty. Zdá sa, že v tomto prípade prestáva platiť Brooksov zákon, že množstvo komunikácie medzi členmi tímu stúpa s ich počtom exponenciálne, zatiaľ čo množstvo vykonanej práce stúpa lineárne. Rozvoj tejto metódy vývoja umožnil najmä prudký rozvoj internetu v posledných rokoch.

Rôzne metódy vývoja sa hodia pre rôzne typy produktov. Asi ťažko bude niekto budovať bankový informačný systém bazárovým spôsobom. V mnohých prípadoch – najmä pri generickom softvéri – je možnosť vybrať si medzi rôznymi vhodnými metódami vývoja.

Rýchlo, rýchlo, rýchlejšie...

Požiadavky na nové verzie softvéru sú čoraz väčšie. Používatelia požadujú viac funkčností, príjemnejšie ovládanie a väčšiu rýchlosť aplikácií za čoraz kratšiu dobu. Vzájomná spolupráca aplikácií patrí tiež k základným požiadavkám na súčasný softvér. Toto núti firmy nielen aplikovať znovupoužiteľnosť a rýchly vývoj aplikácií, ale aj klásť dôraz na zavádzanie účinných foriem komunikácie v rámci tímov a aj celej firmy. Je dôležité dokázať pružne reagovať na požiadavky zákazníkov, ale zároveň udržať konzistentnú špecifikáciu a architektúru systému. Technická úroveň komunikačných prostriedkov je dnes na tak vysokej úrovni, že umožňuje udržovať informovanosť členov tímu bez ich nadmerného zaťažovania vzájomnou komunikáciou.

Vývoj možno urýchliť viacerými metódami. Okrem znovupoužitia softvérových súčiastok možno zaviesť paralelizmus pri vývoji aj testovaní. Tento spôsob sa nemusí aplikovať len pri bazárovom vývoji, ale aj v prípade inak klasického postupu pri vývoji aplikácie. Dôležitým faktorom je motivácia pracovníkov. Veľký úspech bazárového vývoja je spôsobený práve vysokou motiváciou tvorcov tohto softvéru, pretože väčšina z nich rieši problém, ktorý ich priamo „páli“.

...a spolu

Požiadavka na vzájomnú spoluprácu aplikácií kladie vysoký dôraz na presnú špecifikáciu rozhraní programu a dodržiavanie noriem. Bohužiaľ boj firiem o trh častokrát vedie k vytváraniu neverejných noriem a špecifikácií rozhraní, čím vznikajú proprietárne normy určené implementáciou toho-ktorého systému. Dodržiavanie noriem sa tak stáva doménou otvorených systémov a niekoľko málo firiem. Následkom toho dokážu mnohokrát navzájom spolupracovať iba riešenia dodané kompletne od jednej firmy a aj v tomto prípade dochádza k problémom s kompatibilitou. Vývoj otvorených systémov v poslednej dobe však začal dobiehať mnohé komerčné produkty. Týmto sa vytvára konkurenčný tlak na komerčné firmy, ktoré sú nútené dodržiavať normy a štandardy.

Dodržiavanie štandardov a účinná komunikácia v tíme je v dnešnej dobe dôležitejšia ako kedykoľvek v minulosti. Skúsenosti a postupy, ktoré uvádza Brooks v eseji „Passing the Word“ sa ukazujú plne platné ešte aj dnes. Prípad, ktorý opisuje Brooks v eseji možno považovať za ideál riadenia projektu a spolupráce architektov, programátorov a testerov. Keby takéto prepracované metódy spolupráce a dôslednej špecifikácie dodržiavala väčšina firiem, softvéroví inžinieri by mohli spávať omnoho pokojnejšie.

Literatúra

1. F.P. Brooks. The Mythical Man-Month. Addison-Wesley, 1995.
2. E.S. Raymond. The cathedral and the bazaar. http://www.tuxedo.org/~esr/writings/cathedral-bazaar/.
3. T. van Vleck. The multics system programming process. reprinted in IEEE Tutorial on Software Maintenance. http://www.best.com/~thvv/tvvswe.html.

Dajte mi ľudí a ja to urobím

Ľubo DROBNÝ

Abstrakt. Zamyslenie sa nad možnosťami štruktuализácie malého tímu s cieľom úspešnej práce na rozsiahlych softvérových projektoch. Dôraz sa kladie nielen na funkcie jednotlivých členov tímu, ale aj na typy osobností.

Úvod

Náročnosť a rozsah vyvíjaných softvérových produktov sa neustále zvyšuje. Je pravda, že to čo sa zdalo pred dvadsiatimi rokmi nedosiahnuteľné je momentálne k dispozícii a každému po ruke. Ako klasický príklad uvediem veľkolepé sny o celosvetovej sieti, osobných počítačoch a dokonalých filmových trikoch. Na mnohé projekty sme však stále príliš krátki a tieto projekty zostávajú stálou výzvou.

Zásadným problémom pri prijímaní takejto výzvy je vždy to isté: Ako na to? Môžeme to skúsiť masívnym útokom. Ak rozsiahly projekt vyžaduje tisíc ľudí tak ich jednoducho dodáme a viac nad tým neuvažujeme. Ak treba tisíc výkonných počítačov, tak použijeme Internet. Ak treba tisíc slonov tak zájdeme do Afriky a pochytáme ich? O efektivite a riskantnosti takéhoto brutálneho prístupu môžeme uvažovať ako o jednom mantineli možných prístupov.

Druhý prístup spočíva v tom, že daný problém zadáme nejakým „garážovým“ firmám, väčšinou „dvojchlapovým“ so svojským prístupom k veci. Ak sa to niektorej náhodou poradí, tak danú firmu kúpime aj s topánkami a hotovo. Toto by som označil za druhý mantinel.

Dobre vieme, že medzi mantinelmi je ihrisko, na ktorom sa hrá tak, aby sme dospeli k nejakému výsledku. Či to bude radostná výhra alebo trpká prehra závisí najmä od kvality hráčov, ale nielen ako individualít. Podstatná je aj celková zohranosť a dôvera v tíme, ktorej sa hovorí priateľstvo – ako spieva Jožo Ráž: „Len kamarát ti prihrávku vráti“. Dôležitá je aj podpora hráčov zo strany zabezpečovacieho tímu

Pointa podľa mňa je najmä v uvedomení si, že ľudia okolo mňa nie sú len perfektní programátori, ale osobnosti s duchovným a duševným bohatstvom, ktoré treba integrálne rozvíjať a nie potlačovať.

a ich zázemia a veľa ďalších vecí, ktoré pri hre priamo nevidno. Stačí si predstaviť ustarostené tváre trénerov, manažérov a všetkých fanúšikov pri dôležitom zápase o záchranu alebo v záverečnom finále majstrovstiev sveta.

Chcem pripomenúť, že sa zaujímame o rozsiahle projekty, ktoré zvyčajne zahŕňajú celý životný cyklus softvérového systému a nie o niečom typu: „Tam klikneš, tu ťukneš a hotovo“.

Problém

Pán Brooks vo svojej eseji videl zásadný problém v tom, že ak naozaj treba tých tisíc ľudí, tak ako to urobíme len s desiatimi? Opísal dnes už klasické riešenie zostavenia hráčov podľa úzkeho chirurgického tímu. V tejto súvislosti pripomeniem zaujímavé štatistiky, ktoré hovoria o výkonnosti rovnako skúsených programátorov toto: „Najlepší programátor je desaťkrát produktívnejší ako najslabší a navyše produkuje program, ktorý má päťkrát nižšie pamäťové a časové nároky“ [1]. Prekvapujúce, ale je na tom kus pravdy.

Tak zoberme teda desať ľudí, ktorí sú naozaj skvelí v tom, čo robia a sú priemerne sedemkrát produktívnejší ako ich ostatní kolegovia. Keďže je ich iba desať, tak komunikačný problém nech je sedemkrát menší oproti mase tisíc ľudí (čísla podľa štatistík [1]). Výsledný vzťah: $10 * 7 * 7 = 490$. Z toho vyplýva, že ak tisíc priemerných ľudí bude pracovať na projekte jeden rok tak, náš malý tím desiatich statočných bude usilovne pracovať približne niečo cez dva roky. Je to samozrejme čierno-biely pohľad, ale dáva isté nádeje.

Riešenie – chirurgický tím

Základná otázka je však akú štruktúru tohoto malého tímu zvoliť, aby to mohlo dobre fungovať? Podľa pána Milla je ideálny chirurgický tím. Jednotlivé úlohy v tíme sa sústreďujú okolo jednej osoby [1].

Touto osobou je chirurg, ktorý je šéfom tímu. Kladú sa naňho vysoké nároky, pretože nesie zodpovednosť za všetky fázy projektu (definícia projektu, špecifikácia, návrh, implementácia a tvorba dokumentácie). S obrovskou zodpovednosťou však má právo na všetky prostriedky, ktoré uzná za vhodné a taktiež má vždy posledné slovo. Aby to dokázal musí to byť génius, ktorý musí mať dlhoročné skúsenosti s tvorbou rozsiahleho softvéru a s riadením tímu. Dôležitá je aj všeobecná rozhľadenosť.

Ďalším členom tímu je druhý pilot. Je pravou rukou chirurga. Musí dokázať to čo chirurg, ale nemusí mať toľko skúseností. Jeho hlavnou úlohou je byť konzultantom a oponentom pre chirurga, ktorý si na ňom môže otestovať svoje nápady. Druhý pilot je pre chirurga poistka, keby sa mu náhodou pošmykol skalpel, je tiež spojovací článok medzi chirurgom a zvyškom tímu. Podieľa sa priamo na vývoji produktu, ale nenesie priamu zodpovednosť.

Každý šéf potrebuje dobrého správcu, chirurg nie je výnimkou. Správca má na starosti hlavne administratívu projektu. Čiže riadi zdroje: ľudí, financie, priestory a pod. Správca sprostredkúva komunikáciu s administratívnym zvyškom organizácie.

Chirurg musí kvôli maximálnej „čistote“ písať všetku dokumentáciu sám. Potrebuje však šikovného editora, ktorý preberá rozpracovanú alebo diktovanú dokumentáciu a dáva jej „ľudskú podobu“.

Keďže práce je veľa a každý človek má len dve ruky, tak editor a správca majú navyše ďalšie dve: každý má totiž sekretára/ku.

Na pokrytie činností spojených s údržbou technických záznamov a programových knižníc slúži programový úradník. Životaschopnosť tejto organizácie je najmä v hesle: „Zo súkromného umenia robíme verejnú zručnosť“. Každý člen tímu všetko vidí a všetko počuje. Z tohoto pohľadu výsledok nie je nejaké súkromné vlastníctvo, ale vystupuje pojem tím.

V tíme ďalej musí byť človek, ktorý zodpovedá za techniku a vývojové nástroje. Tohoto nazveme nástrojár. Musí urobiť všetko čo vidí chirurgovi na očiach a musí mať pripravené všetky nástroje v perfektnom stave. Viete si predstaviť operáciu s tupým skalpelom a bez dostatočného objemu krvi?

V rozprávke „A je to...“ bolo všetko fajn kým to, čo hlavní hrdinovia vyrábali neotestovali. Aby nedošlo k totálnemu krachu, tak treba skúseného testera. Tento človek má na starosti oponentúru produktu sťaby rozmarný a nespokojný zákazník, ale aj prípravu testov pre jednotlivé moduly pre bežné ladenie a hľadanie chýb, ktorých je neúrekom.

Posledným človekom v tíme je takzvaný jazykový expert, ktorý má na starosti vylepšovanie programov z hľadiska jeho požadovaných vlastností. Robí to zväčša voľbou vhodného programovacieho jazyka s následným využitím jeho výhodných čŕt.

Ako to funguje?

Aká je základná myšlienka chirurgického tímu? Môžem ju zhrnúť do jednej vety: „Desať ľudí pracuje na produkte, ktorý vyšiel z jednej mysle alebo z dvoch, ktoré pracujú *uno animo* (jedna duša)“ [1]. V porovnaní napríklad s modelom rozdelenia práce na dvoch programátorov je výhoda najmä v dodržaní konceptuálnej integrity. Ďalšou výhodou je jasnosť pri rozhodovaní a deľbe zdrojov priradených na projekt, kde chirurg má vždy posledné slovo.

Pri obrovských projektoch sa použije viacero chirurgických tímov, ktoré koordinujú jednotliví chirurgovia. Stačí si predstaviť operáciu veľryby, kde jeden tím rieši problém s očami, iný sa stará o žalúdočné ťažkosti, ďalší sa snaží odstrániť vrodenú poruchu na srdci a pod. Jednoducho každý tím sa venuje jednotlivým podsystémom podľa špecializácie chirurga. Ak by sa niektorí členovia tímu nudili, tak môžu slúžiť viacerým tímom, v štýle hesla: „Človek pracujúci na jednom projekte nie je dostatočne vyťažený“.

V tejto chvíli si každý dokáže predstaviť svoju koncepciu tímu na ihrisku a nájsť mnohé pre a proti. Ja osobne mám dobré skúsenosti s tímom, kde sa práca rozdelí medzi jednotlivých členov, ale zodpovednosť a záverečné slovo má vždy len jeden človek. Čiže sa jedná o určitú obdobu chirurgického tímu.

Problémy však vznikajú na ľudskej úrovni. Ako jednoduchý príklad uvediem jeden možný dialóg.

Chirurg: „Urobíš to takto!“ Člen tímu: „Je to hlúposť, urobím to radšej inak.“ Chirurg: „Urobíš to takto alebo si to radšej urobím sám!“

Dôležité nie je za každú cenu sa zaradiť do nejakej klasifikácie, ale samotný proces spoznávania samého seba a svojich predností.

Ďalším problémom pre manažéra je, ako takého génia udržať v tíme. Je zrejmé, že takýto človek má veľmi vysokú hodnotu, ktorá sa nedá podľa mňa vyvážiť peniazmi alebo prázdnymi rečami. Poznám veľa kvalitných ľudí, ktorí sú ochotní za relatívne smiešne peniaze zostávať stále na tom istom mieste jednoducho preto, že ich to baví.

Ľudia ako osobnosti – súčasný trend

V dnešnej dobe mnoho manažérov prišlo na toto isté pomocou štatistík a psychologických prieskumov motivácie a rozvoja osobnosti. Myslím si, že i pri tvorbe softvérového tímu je dôležité hľadieť na ľudí ako na osobnosti, ktoré majú svoje vedomostné, ale aj duchovné a duševné bohatstvo. V našej kultúre motivovanej úspechom, sa na túto skutočnosť príliš často zabúdalo, a potom sa veľmi čudujeme, keď kvalitní ľudia odchádzajú a nemajú žiadnu chuť viacej spolupracovať, pretože sa cítia iba využívaní. Ak ste pre manažéra alebo šéfa len číslo alebo prostriedok, tak sa necítite príjemne.

Moja úvaha momentálne nechce pokračovať diskusiou ďalších modelov štruktúry tímu, z pohľadu jednotlivých funkcií, ale z pohľadu typu osobností, pretože to je podľa mňa súčasný trend v štrukturalizácii tímu.

Rozdelenie osobností do typov môže byť rôzne a na základe rôznorodých kritérií. Klasické antické rozdelenie je na sangvinika, cholerika, melancholika a flegmatika. Každému je asi jasné, že tím zložený z cholerikov sa „zožerie“ a tím flegmatikov zase nič neurobí.

Iný spôsob rozdelenia osobností je pomocou tzv. Enneagrammu [4]. Osobnosti sa rozdeľujú do deviatich typov podľa rôznych kritérií. Zameriava sa na hľadanie talentov a fixácií (nútiacej sily) človeka a integrálny rozvoj osobnosti ako indivídua v spoločenstve. V tejto eseji vzhľadom na rozsah uvediem len niektoré zaujímavosti, aby bolo zhruba jasné o čo ide.

Typy sa označujú číslom od jedna po deväť a na identifikáciu sa používajú kritériá. Nie je však samozrejmé, ktoré kritérium je najdôležitejšie. Suché vymenovanie všetkých kritérií môže pôsobiť nezáživne, preto na ukážku uvediem len niektoré, ktoré sú podľa mňa zaujímavé.

Čomu sa vyhýbam? Každý z nás sa niečoho bojí a preto sa tomu snaží vyhnúť. Niekto sa bojí hnevu a konfliktov, iný závislosti od iných alebo obyčajnosti, ďalší zase zlyhania, či prázdnoty.

Aký mám obranný mechanizmus? Keď na mňa útočia, tak potláčam reakcie, či radšej sa prejavím v súboji alebo sa utiahnem, či utečiem.

Čo ma môže dostať do pasce? Trebárs niekto mi polichotí a stratím zdravý úsudok, či narážky typu: si zbabelec, ma vyvedú z miery alebo nerozhodnosť, ktorá vo mne zabíja tvorivosť.

Môj štýl reči? Rozprávam poučujúco alebo lyricky, som monotónny alebo táravý.

Týchto kritérií je veľmi veľa a na určenie typu sa používajú ďalšie postupy. Dôležité však nie je za každú cenu sa niekde zaradiť, ale samotný proces spoznávania samého seba a svojich predností [3].

Ako sa toto dá použiť pri tvorbe štruktúry v tíme je zrejmé. Napríklad chirurg by nemal byť človek nezodpovedný, lenivý, táravý a utekajúci pred problémami, ale práve naopak, pretože jeho vedomosti a dlhoročné skúsenosti nemusia vždy stačiť.

Nielen negatívne vlastnosti rozhodujú o zložení dobrého tímu, ale skôr tie pozitívne. Napríklad osobnosť „Jedna“ sa dokáže horlivo angažovať za ľudí a za spravodlivosť a najmä dokáže vidieť veci v reálnom svetle. Ľudia tohoto typu sú dobrí ako oponenti alebo správcovia. Osobnosť „Tri“ zase vie byť veľmi tvorivá a dokáže sa so svojimi nápadmi aj podeliť. Práve takýto by mal byť jazykový expert. „Šestky“ zase dokážu intuitívne vycítiť riziká na rôznych úrovniach a sú vítaní v každom tíme.

Záver

Treba si však uvedomiť, že ani toto nie je zázračný liek na všetko. Psychologické poznatky pochádzajú zo štatistík, priamych skúseností a analýz úspešných tímov, či osobností. Každému je jasné, že takéto tímy vznikli aj bez hory psychológov alebo psychoanalytikov. Chcem poukázať na to, že čisto technický pohľad na vec nie je celkom vhodný. Pretože, keď preženieme všetkých pracovníkov cez Anneagramm (realizovaný ako jednoduchý aplet v Jave) a sucho ich roztriedime podľa softvérových skúseností a typov osobností, môžeme pokaziť aj to čo funguje. Pointa podľa mňa je najmä v uvedomení si, že ľudia okolo mňa nie sú len perfektní programátori, ale aj osobnosti, ktoré majú svoje duchovné a duševné bohatstvo, ktoré treba integrálne rozvíjať a nie potlačovať. Tím, v ktorom si ľudia dôverujú, poznajú svoje talenty, ale i ohraničenia a sú dostatočne profesionálne podkutí sa môže podľa mňa pustiť do rozsiahlych projektov s pokojným svedomím.

Literatúra

1. F.P. Brooks. The Mythical Man-Month, Addison-Wesley 1995.
2. D.W. Oliver. Optimizing the Organization's Structure, IEEE Computer, Vol. 30, No. 7, 1997, 110-112.
3. M. Bubák. SVD: Ľudské vzťahy a komunikácia, http://www.upc.uniba.sk.
4. B. Bahrát. SVD: Enneagramm, Misijný dom Božského Srdca – Vidiná 1995.