



# Meranie v softvérovom inžinierstve

eseje o manažmente softvérových projektov

te@msi



**Meranie v softvérovom inžinierstve  
a eseje o manažmente softvérových projektov**



# Meranie v softvérovom inžinierstve

eseje o manažmente softvérových projektov

te@msi

Slovenská technická univerzita  
2001

© te@Msi

Bc. Peter Agh	Bc. Stanislav Hrk
Bc. Radoslav Kováč	Bc. Dušan Lacko
Bc. Szabolcs Molnár	Bc. Ján Pidych
Bc. Marián Šimo	Bc. Michal Šrámka
Bc. Dušan Šucha	Bc. Marián Teplický
Bc. Vladimír Trgo	Bc. Zoltán Varga

Publikácia neprešla jazykovou úpravou.

## Predslov

Publikácia, ktorú práve držíte v rukách je ďalšou z publikácií o softvérovom inžinierstve, určená študentom tohto oboru. Rozdelená je do dvoch dielov:

Prvý diel obsahuje eseje autorov na tému manažmentu v softvérovom inžinierstve. Každá z esejí sa dotýka určitej tematiky manažmentu. Autori reflektujú eseje známych autorov (ktoré slúžili ako námet), pričom uvádzajú vlastné stanoviská a postoje.

Druhý diel je odborný a venuje sa oblasti *merania v softvérovom inžinierstve*. Určený je pre študentov, ako úvodný študijný materiál o tejto oblasti. Skladá sa z niekoľkých príspevkov. Úvodný príspevok popisuje pojem merania, ciele a význam merania v softvérovom inžinierstve. Popisuje dve hlavné oblasti, kde sa meranie využíva: prvou oblasťou je meranie v jednotlivých etapách životného cyklu (analýza, návrh, implementácia, testovanie, prevádzka a údržba) a druhou je meranie samotného procesu vývoja. V našej publikácii sa zameriavame najmä na prvú časť. Samostatné príspevky sú venované meraniu v jednotlivých etapách, osobitný príspevok je venovaný meraniu pri objektovo-orientovanej paradigme. Okrem toho, stručne sa venujeme aj druhej oblasti, meraniu samotného procesu, ktorej venujeme dva príspevky (meranie procesu a meranie produktivity ako významného atribútu). Záverečný príspevok sa tematike merania venuje z iného pohľadu – z pohľadu zákazníka (špeciálne meraniu produktu zákazníkom). Uvedené zdroje (ktoré uvádzame aj s popismi) možno využiť na ďalšie štúdium tejto oblasti.

Publikácia vznikla v rámci predmetu Manažment v softvérovom inžinierstve, vyučovanom na FEI STU v Bratislave, pod vedením pani docentky Ing. Márie Bielikovej, PhD.



## Obsah

<b>DIEL I. ESEJE O MANAŽMENTE SOFTVÉROVÝCH PROCESOV .....</b>	<b>1</b>
ÚVOD.....	3
CHYBY V PROCESSE TVORBY SOFTVÉRU A ICH PRÍČINY .....	5
JE OTVORENÝ ZDROJOVÝ TEXT RIEŠENÍM PRE INTEGRÁCIU BEŽNE DOSTUPNÉHO SOFTVÉRU? .....	11
ZLATÁ RYBKA.....	17
SOFTVÉROVÝ MANAŽÉR.....	23
KOLÍZIE SOFTVÉROVÝCH MODELOV – AKO SA IM VYHNÚŤ? .....	29
TAJOMSTVO ÚSPEŠNÉHO SOFTVÉROVÉHO PROJEKTU.....	37
ZNOVUPOUŽITIE – ÁNO ČI NIE? AK ÁNO, TAK AKO? .....	43
PRIPRAVTE SA ZLYHAŤ .....	51
TESTOVANIE POŽIADAVIEK A ČLOVEK V ÚLOHE TESTERA. ....	57
ČO ROBIŤ, KEĎ ZÁKAZNÍK NEVIE PRESNE, ČO CHCE?.....	63
AKO ZLEPŠIŤ SOFTVÉROVÉ PROCESY .....	69
AKO URÝCHLIŤ VÝVOJ APLIKÁCIÍ?.....	75
<b>DIEL II. MERANIE V SOFTVÉROVOM INŽINIERSTVE .....</b>	<b>83</b>
ÚVOD.....	85
METRIKY V ŠTÁDIU ANALÝZY .....	91
MERANIE V ETAPE NÁVRHU.....	99
MERANIE V ETAPE IMPLEMENTÁCIE .....	111
MERANIE A PARADIGMY PROGRAMOVANIA.....	119
MERANIE PRI TESTOVANÍ, PREVÁDZKE A ÚDRŽBE SOFTVÉROVÝCH SYSTÉMOV .....	127
MERANIE VYSPELOSTI SOFTVÉROVÉHO PROCESU .....	137
MERANIE PRODUKTIVITY .....	149
MERANIE SOFTVÉRU ZÁKAZNÍKOM (MERANIE PRODUKTU) .....	159



**DIEL I.**  
**Eseje o manažmente**  
**softvérových procesov**

ÚVOD.....	3
CHYBY V PROCESE TVORBY SOFTVÉRU A ICH PRÍČINY .....	5
JE OTVORENÝ ZDROJOVÝ TEXT RIEŠENÍM PRE INTEGRÁCIU BEŽNE DOSTUPNÉHO SOFTVÉRU? .....	11
ZLATÁ RYBKA.....	17
SOFTVÉROVÝ MANAŽÉR.....	23
KOLÍZIE SOFTVÉROVÝCH MODELOV – AKO SA IM VYHNÚŤ? .....	29
TAJOMSTVO ÚSPEŠNÉHO SOFTVÉROVÉHO PROJEKTU.....	37
ZNOVUPOUŽITIE – ÁNO ČI NIE? AK ÁNO, TAK AKO? .....	43
PRIPRAVTE SA ZLYHAŤ .....	51
TESTOVANIE POŽIADAVIEK A ČLOVEK V ÚLOHE TESTERA. ....	57
ČO ROBIŤ, KEĎ ZÁKAZNÍK NEVIE PRESNE, ČO CHCE?.....	63
AKO ZLEPŠIŤ SOFTVÉROVÉ PROCESY .....	69
AKO URÝCHLIŤ VÝVOJ APLIKÁCIÍ?.....	75



## Úvod

Softvérové inžinierstvo je disciplína, ktorá sa neustále rozvíja. Napriek dlhoročnému výskumu a značnému úsiliu je stále ešte mnoho otázok nezodpovedaných, stále je na našej „*mape poznania*“ ešte veľa „*bielych miest*“.

Manažment v softvérovom inžinierstve predstavuje samostatnú oblasť softvérového inžinierstva, pre ktorú to platí takisto. Špecifická vývoja softvéru ho odlišujú od „tradičného“ manažmentu, preto postupy „tradičného“ manažmentu nemožno automaticky aplikovať aj pri vývoji softvéru. Na druhej strane, aj v prípade „tradičného“ manažmentu sa nejedná o preskúmanú a oblasť. Výskum v manažmente v softvérovom inžinierstve v súčasnosti ďalej prebieha, hľadajú sa odpovede na otázky a riešenia problémov.

V tejto časti publikácie práve reagujeme vo forme esejí na niektoré aktuálne problémy manažmentu v softvérovom inžinierstve. Časť predstavuje zborník esejí autorov (esejou prispel každý z autorov publikácie). Každá esej pôvodne vychádzala z témy a úvah určitej eseje známeho autora, avšak nejedná sa o jej recenziu či reprodukciu – ale o určitú reflexiu, zaujatie „osobného“ postoja autora k danej téme a v nej prezentovaným úvahám. Nazdávame sa, že táto časť bude pre čitateľa dozaista zaujímavá a inšpirujúca, veď poznať aj cudzie názory je bezpochyby nevyhnutnou súčasťou poznávania.



## Chyby v procese tvorby softvéru a ich príčiny

Spracované podľa: Boehm, B. – Basili, V. R.: *Software Defect Reduction Top 10 List*. Computer IEEE, január 2001, 135-137.

Zoltán Varga

**Abstrakt.** *Zložitosť softvérových systémov a rýchly vývoj je hlavným faktorom vzniku zbytočných, vyhnuteľných chýb v programoch počas vývoja. Chyby nevznikajú len z dôvodu nepochopenia požiadaviek a funkcií systémov, ale v dosť veľkej miere ich zapríčiňujú aj iné faktory ako nedisciplinovanosť, neperspektívny pohľad, zlé zvyky vývojárov. Ak poznáme zdroje chýb, potom vieme nájsť aj techniku na redukciu chýb. Použitím rôznych techník a dobrých rád počas vývoja softvéru môžeme znížiť vynaložené úsilie a náklady, čím dosiahneme väčšiu efektivitu práce.*

Chyby v produktoch v softvérovom inžinierstve sa vyskytujú dosť často, vyplývajú mnohokrát z vlastností softvéru ako je zložitosť, neviditeľnosť, meniteľnosť. Na tieto faktory jednoznačne vplyvajú aj nedostatočné predstavy zákazníkov, čo požadujú. Každý programátor má cieľ znížiť počet chýb počas tvorby, ale neočakávané zmeny (hlavne zmeny v požiadavkách) nútia ich k stálej zmene systému. Barry Boehm a Victor R. Basili uvádzajú v [Boehm01] zoznam desiatich najzávažnejších chýb počas vývoja softvéru, a tiež dávajú návod na ich odstraňovanie. Pojem chyby treba chápať v širšom slova zmysle. Nehovoríme len o chybe počas vykonávania programu, ale zahrňuje aj nedostatky vo fáze špecifikácií požiadaviek, návrhu a implementácie.

### Na vine je nedostatočná špecifikácia

Vyhľadanie a oprava chýb po zavedení systému do prevádzky je často 100 krát drahšia ako vyhľadanie a oprava týchto chýb vo fáze špecifikácie a návrhu. Pomer 100:1 neplatí vždy, lebo sa dokázalo, že pre menšie systémy je to len 5:1. Hlavným dôvodom na zmenu je zmena požiadaviek používateľov. Avšak tento problém možno jednoducho vyriešiť neustálym prototypovaním a dobrým návrhom modulov, ktoré sú voľne zviazané.

To znamená, že zmena v jednej časti systému nevyžaduje ďalšie zmeny v ostatných častiach. Pán Boehm už v roku 1987 zaznamenal, že ak sa chceme vyhnúť drahým opravám, treba sa zamerať na skorú verifikáciu, validáciu a na prototypovanie zhora nadol. Druhá rada je, že dobrým architektonickým návrhom môžeme značne redukovať faktor stupňovania nákladov aj pre veľké systémy. Dobrým príkladom na to je milión riadkový TRW CCPDS-R systém uvedený v knihe s názvom Software Project Management, ktorej autorom je Walker Royce. Faktor zvýšenia nákladov v uvedenom systéme bol iba 2:1.

Tento jav – oprava chýb po predaji – sa môže vyskytnúť aj v iných odvetviach. Zoberme si len nedávny prípad s prvými procesormi typu Pentium, u ktorých výsledky niektorých matematických operácií boli nepresné. Napriek tomu vlastníci procesorov ich používali a často ani nezbadali tú malú chybičku. Opraviť chybu v návrhu procesora určite nestálo veľa peňazí, ale už výmena vydaných procesorov dotýčnú firmu stálo oveľa viac, lebo procesory musela znovu vyrobiť a bezplatne vymeniť. Oprava chýb v softvérových systémoch takisto vyžaduje finančné náklady, ale cena výmeny (zavedenie produktu do prevádzky) je oveľa nižšia v porovnaní s hardvérovými súčiastkami (alebo produktmi z iných odvetví). Na opravu chýb treba vynaložiť dosť veľké úsilie. Autori v [Boehm01] hovoria, že 40-50% vynaloženého úsilia ide na prácu, ktorá nie je nevyhnutná. Táto práca je spôsobená vyskytnutými chybami, nečakanými procesmi. Keď zoberieme pravidlo rozdelenia úsilia na jednotlivé etapy vývoja, t.j. 1/3 návrh, 1/6 implementácia, 1/4 testovanie modulov a 1/4 testovanie systému, vyjde, že 50 percent celkového úsilia treba vynaložiť na testovanie produktu. Je to len náhoda, že odhad týchto dvoch hodnôt je rovnaký? Priznajme si, že na testovanie produktov nevynaložíme toľko úsilia, ako by bolo potrebné. S tým úzko súvisia aj rôzne zanedbania ošetrení chýb pri programovaní, ktoré môžu spôsobiť vážne chyby v správaní programu.

Zo štúdií niekoľkých projektov sa zistil nasledujúci zoznam tried chýb [Basili84]:

- chybné a nepochopené požiadavky
- chybná a nepochopená funkcionálna špecifikácia
- zlý návrh niekoľkých komponentov (nesprávny predpoklad o hodnote štruktúry alebo údajov, nesprávne riadenie, zlý výpočet)
- zlý návrh alebo zlá implementácia v jednom module
- nepochopenie vonkajšieho prostredia
- chyba v implementačnom jazyku, v prekladači
- chyby, ktoré vznikli odstránením predošlých chýb

Z analýzy rozdelenia chýb sa zistilo, že polovica chýb vznikne z chybných špecifikácií a z nepochopenia požiadaviek. Na korekciu týchto chýb je treba vynaložiť vyše 50% z celkového úsilia.

Okolo 80% nepotrebné práce, ktorá vzniká opravou chýb spočíva v odstraňovaní 20% chýb. Číslo 80% samozrejme závisí od veľkosti systému, ale sa netreba hrať na maličké percentá, hlavný je pomer. Rýchla špecifikácia požiadaviek, návrh a vývoj zvyčajne zapríčiňujú hlavné chyby v architektúre, v návrhu a v kóde, ktorých dôvodom je neskoré prispôsobenie požiadaviek. Veľkú úsporu vynaloženého úsilia môžeme získať zdokonaľovaním vyspelosti softvérových procesov, softvérových architektúr a manažmentu rizík, ktorý vzniká z nadbytočnej práci.

Z uvedených pravidiel jednoznačne vyplýva, že presnou špecifikáciou, zameranou na nejednoznačné časti systémov znížime počet vyskytnutých chýb, nedostatkov. Keďže mnoho zákazníkov nevie jednoznačne vyjadriť svoje požiadavky, počas vývoja sa treba zamerať aj na rýchle a skoré prototypovanie. Dobre vytvorené prototypy pomôžu lepšie identifikovať nedostatky (nepreskúmané možnosti) v systémoch a to nielen nám, ale aj zákazníkovi. S tým docielime zníženie počtu chýb, ktorých odstránenie vyžaduje značnú časť celkového úsilia. Tento názor potvrdzuje aj pravidlo, že väčšina chýb vzniká v 20% modulov a asi polovica modulov je bezchybná.

### **Kontrola, ako riešenie**

Je dané, že skoré zachytenie a oprava mnohých chýb v životnom cykle projektu je viac cenovo efektívna, ako zachytenie neskôr. Pokúšame sa teda nájsť chyby čím skôr. Mnoho výskumov potvrdilo tvrdenie pána Boehma, že opätovná kontrola zachytí 60% chýb. Výsledkom výskumov je, že vizuálna kontrola a testovanie zachytí rozličné triedy chýb v rozličných bodoch v životnom cykle. Taktiež sa zistilo, že riadenou kontrolou identifikujeme o jednu tretinu viac chýb, než neriadenou kontrolou. Vzniká teda potreba empirického výskumu na výber najlepšej stratégie pri hľadaní chýb. Basili [Basili97] uvádza metódu snímania založenú na scenároch, ktorá ponúka množinu formálnych postupov pre detekciu nedostatkov. Tieto techniky sa používajú pri odstraňovaní nedostatkov počas vývoja softvéru – pri špecifikácii požiadaviek, pri objektovo-orientovanom návrhu a pri kontrole používateľského rozhrania.

Podobne je potvrdené, že disciplinované osobné prevádzkové predpisy (dobré organizovanie) môžu redukovať výskyt chýb až o 75%. Hypotéza Harlana Millsa hovorí, že ľudia nekontrolujú tak ako by mali, lebo veria, že testovanie odhaľuje chyby za nich. Dokazovanie tejto hypotézy bolo jednoduché: ľudia, ktorí čítali (kontrolovali) a vedeli, že nemôžu testovať, robili to dôslednejšie, pozornejšie než tí, ktorí vedeli, že môžu testovať neskôr. Každý môže cítiť tento vplyv na človeka aj na vlastnej koži. Pri písaní programov (resp. textov) urobíme preklepy, ktoré si pri pozornej vizuálnej kontrole všimneme a opravíme, inak to necháme na kompilátor (resp. zapneme kontrolu pravopisu). Síce syntaktické analyzátory odhalia syntaktické chyby, ale chyby v sémantike už neodhalia. Teda môžeme konštatovať, že vizuálna kontrola je

efektívnejšia aj cenovo, ako testovanie, t.j. technika kontroly je dôležitá. Avšak rôzne prístupy môžu byť rôzne efektívne pre rôzne typy chýb. Preto ľudia musia byť motivovaní, aby kontrolovali lepšie, aby verili v to, že je to dôležité.

Harlan Mills navrhol techniku „Cleanroom“. Technika vychádza z motivovania programátorov, aby vizuálne kontrolovali to čo napísali tým, že nemôžu testovať funkčnosť programov. Výsledky testovania na skupiny študentov potvrdilo základnú myšlienku vizuálnej kontroly:

- „Cleanroom“ programátori cítili väčší význam v opätovnej kontrole ako sústredenie sa na funkcionálne testovanie, strávili menej času pri počítači a použili menej softvérových prostriedkov.
- Ich programy mali nasledovné vlastnosti: menej komplexné, písali viac komentárov, viac priradovacích operácii, ich produkty boli blízko k požiadavkám
- Mnoho Cleanroom vývojárov modifikovalo svoj programovací štýl a chcú ďalej používať tento prístup, teda získané výsledky pozitívne vplývali aj na prístup k tvorbe softvéru.

Údaje o používaní metódy „Cleanroom“ v NASA ukazujú, že počet chýb počas testovania sa znížil o 25 až 75%. Údaje takisto ukazujú, že oprava chýb trvala dlhšie, ako jednu hodinu len u 5% chýb, kým štandardný proces ukazuje mieru až 60%.

Druhá technika je PSP (Personal Software Process – osobný softvérový proces), ktorého autorom je Watts Humphrey. PSP sa sústreďuje na jadro vzniku chýb – analyzovaním chýb jednotlivcov a vytvorením osobných kontrolných zoznamov s cieľom vyhnúť sa opakovaným chybám. Tým docielime k značnému zníženiu chýb. PSP tréningové kurzy ukazujú 10 krát menší počet vyskytnutých chýb po 10 cvičeniach ako počas prvého tréningu. Efekty pri reálnych projektoch sú však viac rozptýlené, pretože efekty závisia aj od faktorov ako je zrelosť organizácie, ochota personálu a organizácie pracovať vo vysoko štruktúrovanej softvérovej kultúre. Keď spájame PSP s úzko súvisiacou TSP (Team Software Process – tímový softvérový proces), miera chýb sa zníži o 10 alebo viackrát v organizácii, ktorá je dostatočne zrelá.

### **Vyvíjajme vysoko spoľahlivé systémy**

Náklady na vývoj jedného riadku programu sú o 50% vyššie pri vývoji softvéru s veľkou prevádzkovou spoľahlivosťou, ako pri vývoji menej spoľahlivých výrobkov. Teda u spoľahlivejších produktov celkové investície sú väčšie ako morálna hodnota produktu. Ak však projekt zahrňuje aj údržbu softvéru, potom tieto hodnoty sa vyrovnávajú. Veď produkty s veľkou spoľahlivosťou potrebujú menej nákladov na údržbu. Analýzou 161 projektov sa zistilo, že polovica projektov mala väčšie náklady kvôli požadovanej spoľahlivosti softvéru, ako sa očakávalo. Avšak odhad nákladov na údržbu v Cocomo II naznačí, že údržba menej spoľahlivého softvérového systému je o 50% drahšia ako vývoj, pričom

tento podiel je len 15% pri systémoch s veľkou spoľahlivosťou. Typické rozdelenie nákladov podľa životného cyklu ukazuje, že 30% z celkových nákladov treba na vývoj a 70% na údržbu. Z týchto analýz môžeme konštatovať, že náklady na vývoj a údržbu menej spoľahlivých a na vývoj a údržbu vysoko spoľahlivých produktov sú skoro rovnaké. Podobne charakteristiky kvality ukazujú, že vysoko spoľahlivé systémy obsahujú 16-krát menej chýb ako systémy s nízkou spoľahlivosťou. Z týchto informácií už každý vývojár by mal uprednostniť vývoj vysoko spoľahlivých systémov, ktoré sú aj pre zákazníka výhodnejšie, lebo pre nich údržba znamená stratu času, pokles predaja, stratu dlhodobých zákazníkov.

Veseji sme sa zaoberali najväznejšími chybami vyskytujúcimi sa počas tvorby softvérových produktov. Tento zoznam je založený na ďalších, niekoľko ročných výskumoch zaoberajúcich sa znižovaním chýb. Samozrejme, veľa horeuvedených informácií neberie do úvahy vzájomné vplyvy jednotlivých faktorov. Radi by sme vedeli aj odpovede na otázky ako sú:

- Ak ja investujem do vizuálnej kontroly, Cleanroom a PSP, potom platím za odstránenie tých istých chýb 3 krát?
- Aké množstvo testovania mi umožňuje vyhýbanie sa chybám?
- Koľkých ďalších chýb sa dopustím odstránením jednej chyby ?

Je potrebný ďalší výskum v tejto oblasti, aby sme boli schopní odpovedať na podobné otázky. V súčasnej dobe sa stále väčší dôraz kladie aj na tvorbu softvéru so znovupoužitím. Dôvody sú veľmi jednoduché: vytvorené komponenty (súčiastky) sú odskúšané, sú spoľahlivejšie. Avšak špecializované komponenty treba rozširovať, prispôbiť k danej problematickej oblasti. Požadované zmeny môžu byť zdrojom chýb, ktoré práve chceme obísť. Teda nemôžeme jednoznačne tvrdiť, že tvorba so znovupoužitím znižuje riziko vývoja a vyrieši všetko za nás. Aj pri takomto prístupe dodržme uvedené dobré rady a tým vylepšujeme procesy tvorby softvéru!

Môžeme neefektívny softvér označiť za chybný? Pre mňa aj efektivita je podstatná vec, ktorú často zanedbávame. Ani tak nedávno výkon osobných počítačov oproti dnešnej situácii bol rádovo nižší. Prudký vývoj hardvérových súčiastok prináša neuveriteľný pokrok. Vývoj softvérových produktov taktiež nezaostáva, vzniklo mnoho nových a užitočných produktov, ktoré uľahčujú náš každodenný život. Ale naozaj je vývoj taký prudký, aj keď zarátame do toho efektivitu programov? Softvérové produkty vyžadujú čoraz vyššie a vyššie nároky na použitý hardvér. Kým výkon počítačov neustále rastie, efektivita programov klesá. Hovorí sa, že 90% času vykonávania programov zodpovedá 10% kódu. Značné zrýchlenie by sme mohli dosiahnuť optimalizovaním tých 10% kódu. Príde čas, kedy výrazné zvýšenie výkonnosti počítačov z technických

dôvodov už nebude možné, potom tento zákon bude mať vysokú prioritu pri návrhu softvérových systémov.

Niekoho môžu uvedené vysoké pomery chýb odstrašiť. Alebo môžeme byť šťastní, že máme identifikované problémy? Veď sme ľudia, ktorí sa môžu tiež pomýliť. Dôsledky omylov sú chyby a je to tak nielen v softvérovom inžinierstve, ale aj v každodennom živote. Keď neurobíme veľmi vážne chyby, potom ich odstránenie je určite možné. Každý by mal zefektívniť svoju prácu analyzovaním úsilia na opravu chýb, aby určil ich zdroje a takto znížil množstvo nadbytočných prác. A bezchybné, ideálne softvérové produkty ostávajú len snom...

## Literatúra

- [Basili84] Basili, V. R. – Perricone B.: *Software Errors and Complexity: An Empirical Investigation*. Communication of the ACM, Január 1984, vol. 27, no. 1, s. 42-52.  
<http://www.cs.umd.edu/projects/SoftEng/ESEG/papers/82.19.pdf>. (Február 2001).
- [Basili97] BASILI, V. R.: *Evolving and Packaging Reading Techniques*. Communication of the ACM, Júl 1997, vol. 38, no. 1, s. 3-12.  
<http://www.cs.umd.edu/projects/SoftEng/ESEG/papers/reading.techniques.pdf>. (Február 2001).
- [Boehm01] BOEHM, B. – BASILI, V. R.: *Software Defect Reduction Top 10 List*. Computer IEEE, Január 2001, s. 135-137.  
<http://www.cebase.org/defreduction/top10>. (Február 2001).

## Je otvorený zdrojový text riešením pre integráciu bežne dostupného softvéru?

### (Open Source: a (C)OTS silver bullet?)

Esej inšpirovaná článkom Barryho Boehma a Chrisa Abtsa – *COTS Integration: Plug and Pray?* (IEEE Software, január 1999, 135-138).

Dušan Lacko

**Abstrakt.** *Využívanie generického, bežne dostupného softvéru pri vývoji rozsiahlych systémov sľubuje značné zníženie peňažných a časových nákladov, ale taktiež prináša so sebou aj nové problémy. Správne použitie softvéru s otvoreným zdrojovým textom môže ale zmierniť tieto problémy a zjednodušiť ich riešenie.*

V súčasnej dobe sa v oblasti softvérového inžinierstva čoraz častejšie hovorí o znovupoužitelnosti, integrácii a generických, bežne dostupných systémov. Dôvodom tohoto záujmu je čoraz viac rastúca zložitosť súčasných softvérových systémov. Z tohoto dôvodu stále väčší počet organizácií používa generický, voľne dostupný softvér nie len ako kompletne koncové aplikácie, ale aj ako súčasť pri vytváraní väčších systémov. Táto nová úloha generického softvéru ale prináša aj nové problémy pri jeho integrácii do vytváraného systému.

Dôvodov čoraz častejšieho využívania znovupoužitia na úrovni generického softvéru veľkými spoločnosťami je, aj napriek nutnosti riešiť problémy s integráciou, očakávanie významných úspor času a peňazí. Podľa štatistík, aj najlepší programátori vyprodukujú za deň iba približne 10 riadkov zdokumentovaného a otestovaného zdrojového textu [Voas98]. Táto skutočnosť robí vývoj vlastných rozsiahlych softvérových systémov veľmi nákladným. Ak spoločnosť vyvíja produkt ktorý bude obsahovať približne 300,000 riadkov, tak získanie napríklad 100,000 riadkov „niekde inde“, by výrazne znížilo náklady na vývoj systému.

### **Čo je vlastne komerčný, bežne dostupný softvér (COTS)?**

Existuje mnoho definícií komerčného, bežne dostupného softvéru (COTS – *Commercial off-the-shelf*). Väčšina autorov sa zhoduje na nasledujúcej definícii [Oberndorf98]:

Softvér získaný od externého komerčného dodávateľa a integrovaný do systému je COTS softvér. COTS softvér môže byť abstraktný dátový typ, trieda, knižnica takýchto tried, celý podsystem (napr., databáza) alebo v širšom zmysle aj celá aplikácia. Hlavným znakom COTS softvéru je, že už existuje, je dodávaný viacerým zákazníkom (teda nie je špecifickým riešením pre jedného zákazníka). Ďalšou často uvádzanou vlastnosťou je, že COTS softvér neumožňuje svoju modifikáciu, a zvyčajne nie je voľne prístupný jeho zdrojový text (ak sa to dá modifikovať, tak to nie je COTS [Boehm99]).

Niektorí autori zaoberajúci sa generickým, voľne dostupným softvérom ale považujú túto definíciu za veľmi hrubú a tvrdia, že komerčnosť a modifikovateľnosť sú dva úplne rozdielne a od seba nezávislé atribúty. Generický, bežne dostupný softvér teda môže byť súčasne aj komerčný aj modifikovateľný a mať voľne prístupný zdrojový kód [Carney00]. Ďalej v texte budem preto používať skratku OTS pre každý generický, voľne dostupný softvér (či už komerčný a/alebo modifikovateľný) a skratku COTS pre komerčný, voľne nemodifikovateľný softvér.

### **Čo je systém s otvoreným zdrojovým textom programu (Open Source)?**

Tak ako existuje viac definícií OTS, existuje aj viac definícií systémov s otvoreným zdrojovým textom programu (ďalej OSS – *Open Source System/Software*). Najčastejšie používaná definícia sa nachádza na stránke Open Source Iniciatívy [OSD]. Táto definícia sa ale zaoberá iba požiadavkami, ktoré by mala spĺňať licencia OSS – licencia OSS má umožňovať bezplatný prístup k danému softvéru a jeho zdrojovým textom a každému záujemcovi musí umožňovať modifikáciu daného softvéru.

V tomto článku budem pod OSS chápať softvér, ktorý spĺňa nielen podmienky Open Source Iniciatívy, ale pri jeho vývoji je použitý aj špecifický proces, najčastejšie nazývaný Katedrála a Bazár [Raymond00].

### **Problémy integrácie OTS a ich riešenia**

Pri integrácii OTS softvéru do väčších systémov sa stretávame s novými problémami, ktoré sa nevyskytujú pri vývoji systémov bez použitia OTS. V nasledujúcich odstavcoch rozoberiem niektoré z týchto problémov a pokúsím sa načrtnúť ako môže voľba OSS zjednodušiť ich riešenie.

Problémy integrácie OTS softvéru môžeme rozdeliť do dvoch skupín: technické a manažérske.

## Technické problémy

Najťažšie riešiteľné problémy s integráciou OTS softvéru sú spôsobné architektonickými nezhodami medzi použitými OTS systémami a vyvíjaným systémom [Garlan95]. Každý systém bol vyvinutý s určitými predpokladmi o architektúre, spôsobe spolupráce a integrácie jednotlivých komponentov. Keďže často krát jednotlivé OTS systémy pochádzajú od rozličných dodávateľov, je veľmi nepravdepodobné, že boli vytvorené s rovnakými architektonickými predpokladmi. Tieto predpoklady sú väčšinou implicitné (nezdokumentované) a prídu sa na ne až počas implementácie systému.

Integrácia takýchto architektonicky sa nezhodujúcich systémov väčšinou vyžaduje prispôbenie použitých OTS komponentov vyvíjanému prostrediu a ostatným komponentom. Niekedy sa dá toto prispôbenie zabezpečiť zapuzdrením COTS komponentov (použitie „wrapper“ modulov) [Mehta00]. Často ale treba vykonať zmeny priamo v niektorých OTS systémoch. Pri COTS systémoch je väčšinou potrebné použiť spätné inžinierstvo a zmeny softvéru urobiť v binárnych súboroch systému [Garlan95], čo je veľmi nákladné a po prechode na novú verziu COTS modulu je nutné vykonať túto činnosť odznova.

Použitie OSS významne zjednodušuje vykonávanie týchto úprav. Existujú však námietky [Thomas00], že ak modifikujeme zdrojový kód OTS modulu, potom sa stávame jeho vlastníkom a leží na nás zodpovednosť údržby tohoto modulu. Tento názor platí ale iba v prípade, ak sa nám nepodariť presadiť zmenu u entity, ktorá je zodpovedná za používaný modul. V prípade OSS je však omnoho väčšia pravdepodobnosť, že sa navrhovaná zmena zahrnie do pôvodných zdrojových textov (ak už nie inak, tak aspoň pomocou prostriedkov podmienenej kompilácie – `#ifdef`, `#endif`) ako pri presadzovaní tejto zmeny u komerčného dodávateľa.

Ďalším problémom pri integrácií OTS systémov je hľadanie a odstraňovanie chýb systému („debugging“). Pri bežnom použití COTS bez možnosti prístupu k zdrojovým textom programu, sme odkázaní na testovanie a hľadanie chyby metódou čiernej skrinky. V prípade, že identifikujeme chybu a chybný modul, náprava nepozostáva z opravy zdrojového textu ale z hľadania spôsobu, ako chybu obísť zmenou našej časti systému [Hissam97, 98]. Výhody prístupu k zdrojovým textom modulu pri použití OSS hádam ani netreba opisovať.

Iným problémom je, že komerčné OTS systémy často nepoužívajú štandardné rozhrania/protokoly alebo používajú vlastné nedokumentované rozšírenia štandardných protokolov. Túto stratégiu používajú firmy s veľkým podielom na trhu, ktoré sa takto snažia zabrániť vstupu konkurencie na trh. Používateľ je potom viazaný na riešenia jednej firmy a stráca možnosť rozhodovania. Takéto firmy sa väčšinou bránia tvrdením o potrebe a práve na inováciu. Ich vlastné interné dokumenty (napr. [Halloween]) však napriek tomu priamo

hovorí o uzavretých zmenách (dekomoditizácii) protokolov ako o spôsobe udržania svojho postavenia na trhu.

Tomuto problému sa dá vyhnúť používaním OTS, ktoré spĺňajú podmienky *otvoreného systému* [Oberndorf98]. *Otvorený systém* je systém, ktorého rozhrania a služby sú jasne špecifikované a implementácia tohoto systému túto špecifikáciu naozaj spĺňa. Špecifikácia rozhraní musí byť prístupná verejnosti a musí byť spravovaná na základe dohody v rámci nejakej skupiny (verejnosť, vývojári firiem z danej oblasti, ...). Otvorené systémy chránia používateľa pred pevným „uzamknutím“ k riešeniam jednej firmy a umožňujú jednoduchšie nahrádzanie jednotlivých modulov alternatívnymi riešeniami. OSS v drvivej väčšine spĺňajú tieto podmienky pretože celá filozofia systémov s otvoreným zdrojovým kódom je založená na čo najrozsiahljšom zvoľpoužití, používaní štandardných rozhraní a proces vývoja takýchto systémov je založený na hľadaní riešení, ktoré vyhovujú väčšine zúčastnených.

### **Manažérske problémy**

Manažérske problémy súvisia s výberom dodávateľov OTS komponentov, komunikácie s dodávateľom a získavanie podpory od dodávateľa.

Výber konkrétneho komponentu a dodávateľa je jedným, z najdôležitejších procesov pri vývoji s použitím OTS softvéru. Keďže zvolené komponenty budú súčasťou nášho systému pravdepodobne počas jeho celej životnosti treba veľmi dôkladne zvážiť ich výber. Výber komerčných OTS je skomplikovaný tým, že komerční dodávatelia sa snažia svoje systémy vychváliť a čo najmenej sa zmieňovať o chybách a problémoch. Kvalitnejšiemu výberu by tiež pomohol prístup k internej technickej dokumentácii, ktorá by mohla osvetliť architektonické predpoklady, z ktorých sa pri vývoji systému vychádzalo. Táto dokumentácia ale nie je zvyčajne verejne dostupná.

Na druhej strane, autori OSS nemajú dôvod prikrášľovať vlastnosti svojho systému. Každý OSS projekt si taktiež udržiava verejne prístupnú databázu chýb, pretože čím je väčšie povedomie o chybe, tým rýchlejšie sa nájde nejaký vývojár, ktorý ju opraví. Verejne prístupné sú tiež archívy komunikácie medzi jednotlivými vývojármi (väčšinou ide o konferenciu elektronickej pošty). Tento archív umožňuje zistiť dôvody dôležitých rozhodnutí, a viesť svetlo do architektonických predpokladov, s ktorými sa systém vyvíja.

Dôležitým aspektom manažmentu dlhodobého projektu, ktorý využíva OTS komponenty je ochrana pred evolučným posunom integrovaných komponentov nežiadúcim smerom alebo úplným zrušením vývoja a podpory komponentu dodávateľom. V prípade COTS softvéru začína komponent bez podpory rýchlo zastarávať a integrátor musí veľmi rýchlo nájsť náhradu. V prípade OSS má integrátor možnosť prevziať na seba údržbu komponentu a pokračovať v jeho používaní. OSS majú nižšiu pravdepodobnosť úplného ukončenia vývoja komponentu,

pretože aj keď aktuálny koordinátor/vedúci projektu nemá zdroje alebo vôľu pokračovať v jeho udržiavaní, je pravdepodobné, že starosť o projekt prevezme niekto iný.

### Najčastejšie výhrady voči OSS

Manažéri veľkých softvérových projektov sa často obávajú využívať OSS vo svojich projektoch. Najčastejšie sa obávajú nízkej spoľahlivosti a nedostatku podpory pre OSS systémy.

OSS ale už v dnešnej dobe dosahuje a dokonca v niektorých prípadoch aj prekračuje spoľahlivosť komerčných riešení. Ako príklad môže slúžiť práca [Miller00], v ktorej autori testovali spoľahlivosť nástrojov a služieb operačného systému UNIX a boli prekvapení skutočnosťou, že spoľahlivosť OSS nástrojov operačného systému Linux je význačne vyššia ako u porovnateľných komerčných systémov.

Nedostatok oficiálnej podpory riešia firmy, ktoré poskytujú zmluvnú platenú podporu pre jednotlivé OSS (ako napríklad firmy Red Hat a Caldera pre Linux, alebo firma CodeWeavers, Inc. pre knižnicu Wine). Priama podpora vývojárov poskytovaná vývojármi jednotlivých OSS na sieti Internet, je ale tiež veľmi kvalitná, čo dokazuje aj udelenie ocenenia *Best Technical Support Award* operačnému systému Linux [Foster97].

Práve vďaka týmto skutočnostiam sa OSS čoraz častejšie začína presadzovať aj v komerčnej sfére [Wang01, Weinberg00].

OSS nie je všeliakom na všetky problémy integrácie OTS systémov, ale pri správnom výbere použitého systému a akceptovaní niektorých špecifik vývoja OSS môže významne uľahčiť integráciu OTS softvéru. Najvýznamnejšiu výhodu vidím v slobode rozhodovania, ktorú OSS dáva vývojárovi. Ak sa vývojár nerozhodne modifikovať zdrojové texty, môže OSS systém používať ako bežný komerčný nemodifikovateľný softvér a OSS nebude mať žiadne výrazné nevýhody oproti svojim komerčným ekvivalentom. Ale v prípade, že sa niekedy objaví potreba modifikovať použitý OTS, prináša použitie OSS nespornú výhodu.

### Literatúra

- [Boehm99] BOEHM, BARRY – ABTS, CHRIS: *COTS Integration: Plug and Pray?* IEEE Software, Január 1999, 135-138.
- [Carney00] CARNEY, DAVID – LONG, FRED: *What Do You Mean by COTS? (Finally, a Useful Answer)*. IEEE Software, Marec/Apríl 2000, 83-86.
- [Foster97] FOSTER, Ed: *Best Technical Support Award*. <http://www.infoworld.com/cgi-bin/displayTC.pl?/97poy.supp.htm>.

- [Garlan95] GARLAN, DAVID – ALLEN, ROBERT – OCKERBLOOM, JOHN: *Architectural Mismatch: Why Reuse Is So Hard*. IEEE Software, November 1995, 17-26.
- [Halloween] VALLOPILLIL, VINOD – COHEN, JOSH – RAYMOND, ERIC S.: *The Halloween Documents*. <http://www.opensource.org/halloween/>.
- [Hissam97] HISSAM, SCOT: *Case Study: Correcting System Failure in a COTS Information System*. Software Engineering Institute, Carnegie Mellon University, 1997.
- [Hissam98] HISSAM, SCOT – CARNEY, DAVID: *Isolating Faults in Complex COTS-Based Systems*. Software Engineering Institute, Carnegie Mellon University, 1998.
- [Mehta00] MEHTA, ALOK – HEINEMAN, GEORGE T.: *A Framework for COTS Integration and Extension*. AFS/WPI, 2000.
- [Miller00] MILLER, BARTON P. ET AL: *Fuzz Revisited: A Re-examination of the Reliability of UNIX Utilities and Services*. Computer Sciences Department, University of Wisconsin, 2000.
- [Oberndorf98] OBERNDORF, PATRICIA: *COTS and Open Systems*. Software Engineering Institute, Carnegie Mellon University, 1998.
- [OSD] *The Open Source Definition*. <http://www.opensource.org/docs/definition.html>.
- [Raymond00] RAYMOND, ERIC S.: *The Cathedral and the Bazaar*. <http://tuxedo.org/~esr/writings/cathedral-bazaar/cathedral-bazaar/cathedral-bazaar.ps>.
- [Thomas00] THOMAS, BILL – WEINSTOCK, CHUCK – PLACE, PAT – HISSAM, SCOT – PLAKOSH, DAN: *A Discussion of Open Source Software*. SEI Interactive, Marec 2000.
- [Voas98] VOAS, JEFFREY M.: *The Challenges of Using COTS Software in Component-Based Development*. IEEE Computer, Jún 1998, 44-45.
- [Wang01] WANG, HUIQING – WANG, CHEN: *Open Source Software Adoption: A Status Report*. IEEE Software, Marec/Apríl 2001, 90-95.
- [Weinberg00] WEINBERG, BILL: *Linux: Off-the-Shelf and Open Source*. COTS Journal, Máj/Jún 2000.

## Zlatá rybka

Peter AGH

„Chcem to a to. A to čo najskôr.“

„Jasné, pane. Naše heslo –

**Zázraky na počkanie, Nemožné do týždňa!**

**Informačný systém do piatich dní!**

*Myslím, že splniť Vaše požiadavky nebude žiaden problém!*

*A mimochodom, môžete mi prosím nadiktovať čo presne chcete?“*

**Abstrakt.** Tematikou tejto eseje sú nerealistické očakávania, či už zákazníkov alebo vývojárov. Snažím sa v nej hľadať príčiny „optimizmu“. Zamýšľam sa a navrhujem možné riešenia: jasná a „čistá“ komunikácia so zákazníkom, partnerský vzťah medzi zákazníkom a vývojárom, trpezlivo odhady a precízne plánovanie, ale aj určitá „osveta“ zákazníkov i celej spoločnosti.

Podnetom pre túto esej bola esej od Barryho Boehma nazvaná „The Art of Expectations Management“. V eseji bolo uvedených niekoľko postrehov k vývoju softvéru, ku ktorým by som sa chcel vyjadriť. Najskôr si dovoľm priblížiť obsah spomínanej eseje. V úvode autor uviedol príhodu so svojim podriadeným, za ktorým raz prišiel, aby rýchlo vykonal „malé“ rozšírenie existujúceho produktu ich firmy pre potreby istého zákazníka. Z funkčného hľadiska sa skutočne jednalo o malú zmenu (realizovateľnú niekoľkými jednoduchými funkciami), avšak toto rozšírenie zahrňovalo aj námahu potrebnú na ďalšie súvisiace aktivity, ako napr. rozšírenie používateľského rozhrania, testovanie správnosti nových funkcií, testovanie dôsledkov správania sa nových funkcií na ostatné komponenty systému, doplnenie dokumentácie a rozšírenie používateľskej príručky a iné. Autorov podriadený toto všetko zbral do úvahy, vymenoval šéfovi všetky súvisiace činnosti a dobu realizácie stanovil na dva týždne. Šéf, ktorý očakával odpoveď „okamžite“, prípadne „do zajtra“ síce nebol nadšený, ale uznal oprávnenosť odhadu, pretože pri zadávaní svojej požiadavky si neuvedomil, čo všetko jej realizácia zahŕňa. Nakoniec ho však podriadený príjemne prekvapil, pretože zadanú prácu

stihol za jeden týždeň. Možno súhlasiť s autorom, že tak vznikla oveľa príjemnejšia situácia, ako keby mu jeho podriadený sľúbil, že to bude „do zajtra“, a pritom by to bolo až o týždeň. Vznikla by prirodzená nervozita, všetci (zákazník, podriadený i šéf) by boli svojím spôsobom poškodení, a to pri rovnakej výkonnosti pracovníka ako v predchádzajúcom prípade. Dokonca je zrejmé, že aj pri vyššej námahe a zaťažení pracovníka by napokon došlo k pocitu neúspechu a poškodenia (veď aj keby sa pracovník viac snažil, trebárs aj dvojnásobne, a stihol by to spraviť dajme tomu za štyri dni, aj tak by sme to pokladali za neúspech, veď sme očakávali jeden deň). Chyba by teda nebola v pracovníkovi, ale v nerealistických očakávaniach.

Optimizmus nepochybne je pri vývoji softvéru prevládajúca emócia. Teda aspoň v počiatočných štádiách projektu.

Autor uvádza niekoľko postrehov: Po prvé, vývojári chcú potešiť, čo vedie k snahe vyhnúť sa hocijakej konfrontácii. Ľudia sľúbia viac, než čo sú schopní. Po druhé, niekedy vládne „v tábore vývojárov“ veľký entuziazmus, ako veľmi určité nástroje a metódy pomôžu pri vývoji softvéru pre zákazníka. Príručky zväčša uvádzajú ako príklady len jednoduché vzorové aplikácie, čo môže zavádzať a viesť k chybným očakávaniam. Výsledkom je práve prílišný optimizmus a nerealistické očakávania. Napokon, zadávatelia i vývojári majú odlišné predstavy o svojich schopnostiach a možnostiach. Príkladom môžu byť vedci a politici. Ich prístupy a vzájomné predstavy o sebe sú tak rozdielne, že možno priam hovoriť „o dvoch kultúrach“.

Ako riešiť uvedené problémy? Autor zvyrazňuje potrebu efektívnej komunikácie, realistických odhadov a plánovania. Hovorí, že „jasná komunikácia, opatrné a uvážlivé názory a odhady a precízne plánovanie pomôžu dosiahnuť realistické očakávania (a nádeje)“. Zamyslime sa bližšie nad týmito odporúčaniami.

### **Komunikácia.**

Potreba jasnej, jednoznačnej a efektívnej komunikácie je zrejmá a dá sa

*Na brehu mora chytil rybár  
s jednou chromou rukou ryby.  
Pošťastilo sa mu chytiť zlatú rybkú.  
Pustil ju a tá mu sľúbila, že mu  
splní tri želania. Rybár sa smutne  
pozrel na svoju chromú ruku i  
riekol: „Chcem mať obe ruky  
rovnaké!“. Rybka začarovala, a  
stalo sa. Aj druhá, zdravá ruka*

s ňou len súhlasiť. Zákazník sa pohybuje väčšinu času v svojom prostredí. Komunikuje zväčša s ľuďmi zo svojej oblasti, jazykom tejto oblasti. Prirodzene, nehovoríme o manažéroch zákaznickej firmy, ale napríklad o jeho vývojových pracovníkoch alebo robotníkoch, ktorí systém používajú pri výrobe. Avšak práve od nich prichádzajú požiadavky na systém. Následky na vyvíjaný systém môžu byť

ďalekosiahle – pretože sú zvyknutí, že ľudia, s ktorými komunikujú, sa v danej oblasti vyznajú, tak sa môže stať, že náležite nevysvetlia určité pojmy, situácie, hraničné prípady, resp. neopíšu kontext. Analogicky toto môže byť aj problémom vývojárov. Zákazník sa zväčša v IT nevyzná, s čím

treba počítať a tak sa aj vyjadrovať. Navyše, zákazník môže vzbudiť mylný dojem, že sa v IT (resp. v určitej oblasti IT) vyzná, napr. tým, že pri vyjadrovaní svojich požiadaviek použije niekoľko „IT slov“, ktoré niekde počul, no odborníkom nie je. Tým sa môže stať, že slovom, ktorého význam a kontext bližšie nepozná vyjadrí odlišnú skutočnosť, než by chcel. Dôležitých je preto niekoľko schopností zákazníkov i vývojárov. Prvá je schopnosť komunikovať jasne a jednoznačne. Je potrebné vedieť sa vyjadriť dostatočne formálne, komunikovať s uvedovaním si zákonov logiky. Ďalej, potrebná je snaha o pochopenie toho druhého. Pri vývoji projektu si zákazník i vývojár musia vedieť v krátkom čase naštudovať a osvojiť množstvo nových vedomostí z oblasti svojho projektového partnera. Analytik musí vedieť osvojiť si v krátkom čase veľa nových pojmov z domény zákazníka. Podobne ale aj zákazník sa dozvie veľa nového o niektorých oblastiach IT a je ideálne (a niekedy aj nutné), aby niektoré strategické rozhodnutia, ktoré tieto znalosti vyžadujú, urobil sám. Ideálne by bolo predpokladať, že zákazník sa vyzná v oblasti IT a analytik má aspoň základy doménovej oblasti. Toto však zväčša neplatí, a preto ostáva požiadavka vedieť si rýchlo osvojiť nové poznatky a vedieť s nimi operovať, pričom pri ich získavaní má nezanedbateľný význam práve komunikácia so svojím partnerom.

*sa stala chromou. Rybár zakrútil očami a skríkol: „Nie tak, naopak!“. Rybka začarovala, a stalo sa. Tá ruka, ktorá bola na počiatku zdravá sa stala chorou a naopak. Rybár zalomil rukami: „ja sa z toho asi zbláznim!“. A stalo sa.*

### Prototypovanie.

Významnou pomôckou pri špecifikovaní požiadaviek je využitie techniky prototypovania. Prototypovanie môže vhodne poslúžiť na viacero významných účelov. Jednak si vývojári overia, že správne pochopili zákazníkove požiadavky (na základe reakcií zákazníka na prototyp). Ďalej, zákazník sám bude môcť lepšie pochopiť svoje potreby (na základe toho, ako ich napĺňa prototyp) a môže pridať nové požiadavky alebo naopak, niektoré pôvodné (ktorých realizácia je drahá a pritom zisk pre jeho potreby je nevýznamný) môže redukovať. Príklad poslednej situácie je uvedený v [Boehmoo], keď zákazník mal požiadavku na „rýchly“ systém (t.j. systém s krátkou dobou odozvy) , ktorého realizácia by bola príliš drahá. Keď sa mu však dodal prototyp systému, ktorý bol pomalší, tak zistil, že tento plne postačuje jeho potrebám. Ušetrili sa finančné prostriedky a skrátil sa čas vývoja.

*Na brehu rieky sedí deduško a chytá ryby. Chytí zlatú rybku a tá mu povie: „ak ma pustíš, splním ti tri želania!“. On ju však len škodoradostne odhodí preč od hladiny, na súš, a zakričí za ňou: „ja som kúzelný deduško!“*

### Vzťah zákazníka a vývojára. Partnerstvo a spoluzodpovednosť.

Dôležité je, že zákazník nemôže mať len „pasívnu úlohu“ na projekte, nemôže očakávať, že prosto odpovie na pár otázok vývojára a dostane to, čo si predstavuje. Zákazníka vývojára nemožno prirovnať ku kupujúcemu

a predajcovi. Oboja, zákazník i vývojár musia sledovať a kontrolovať, či sa navzájom chápu a skúmať, či všetko dôležité bolo zodpovedané. Označenie oboch strán ako „partnerov“ nebolo použité náhodne. Páči sa mi filozofia metodiky vývoja PRINCE (Project In Controlled Environment), ktorá bola vyvinutá v Británii ako metodika, podľa ktorej sa musí vyvíjať softvér pre britské vládne inštitúcie. Stanovuje, že za vývoj sú spoluzodpovední vývojár i zákazník [Červenka98]. Teda zákazník nemôže byť len pasívnym článkom, prostým zadávateľom projektu, ale chápe sa ako spolutvorca projektu, z čoho plynú jeho povinnosti a zodpovednosť.

Toto nie je vždy samozrejmé. Poznám firmu, ktorá robila zakázku pre istý švajčiarsky podnik. Túto zakázku mala pôvodne iná firma, tá ju však nedokončila a tak sa k nej dostala spomínaná firma. Komunikácia oboch strán však nebola skoro žiadna – zákazník odovzdal haldu (pomerne nezrozumiteľných)diagramov „prípadov použitia“ (use-case diagramov), ktoré vypracoval s predchádzajúcim riešiteľom a s novým riešiteľom komunikoval iba zriedka, pretože „nemal čas“. Vývojári namiesto komunikácie museli „lúštiť“ use-case diagramy a hádať odpovede na niektoré otázky. Celý postup bol pre nich oveľa namáhavejší a navyše sa objavili zbytočné chyby. A prípadov, keď zákazník nie je ochotný podieľať sa na realizácii projektu a do komunikácie s vývojárom investuje len minimum času, je stále neúrekom.

### **Realistické odhady.**

Precízne plánovanie, triezve odhady. Boehm v [Boehm00] radí, že je potrebné dať zákazníkovi reálne a triezve odhady. Hovorí, že problémom pre „dostatočnú reálnosť“ je prílišná sebadôvera vývojárov a snaha „zapáčiť sa“ zákazníkovi. Nepochybne, dnešné techniky a metodiky softvérového inžinierstva sú pomerne dobrým „liekom“ proti nerealistickým odhadom. Vývojári vďaka nim a svojim skúsenostiam môžu pomerne presne odhadnúť svoje sily.

Druhou vecou však ostáva snaha zapáčiť sa. Autor sa tejto téme nevenuje, resp. automaticky predpokladá jej nahradenie „reálnym pohľadom“. V tomto smere však stále vidím problém, vychádzajúci z predstáv zákazníka.

Žijeme v komerčnom svete. Zákazník chce za projekt zaplatiť čo najmenej a mať ho hotový čo najskôr. Pokiaľ firma predloží na konkurze svoj odhad nákladov a doby realizácie, ktorý je pomerne reálny, môže sa stať, že zákazku nedostane, ale dostane ju práve iná, „optimistická firma“, ktorá sa domnieva, že danú úlohu vďaka šikovnosti svojich ľudí a „nenáročnosti“ bez väčších ťažkostí zvládne.

Možno samozrejme namietat, že zákaznícke firmy nie sú naivné, že nedajú automaticky ponuku tomu, kto ponúka zdanlivo najlepšie, ale že sami skúmajú, či je dodávateľ schopný svoj sľub uskutočniť a zaujímajú sa o jeho renomé. Týchto však nie je až tak veľa (hlavne na Slovensku), prevláda dojem „optimizmu“ a „všemocnosti IT“. Potrebné sú k tomu aj

určité znalosti a skúsenosti. Ďalej, zákazníci často nechápu obtiažnosť dodatočných zmien požiadaviek, neuvedomujú si náročnosť projektu a z toho vyplývajúce možnosti prekročenia časových plánov a nevidia obtiažnosť údržby.

Treba však pripomenúť, že tieto problémy si často neuvedomujú ani „optimistickí“ vývojári [Brooks95]. Pre nich sú možnými riešeniami metódy a postupy softvérového inžinierstva. V prípade zákazníkov sa však jedná o určité „nepochopenie softvéru“.

Zákazníci si neuvedomujú reálne možnosti a ohraničenia, nevidia zložitost' softvéru a náročnosť procesu jeho vývoja. Možno len zákazník, pre ktorého bol už kedysi vyvíjaný nejaký špeciálny produkt na objednávku, navyše zložitý, môže mať predstavu o náročnosti, pretože práve „zložitým produktom na zakázku“ softvér je.

Časť dôvodov je možno daných biologickými aspektami podstaty a psychiky nás ľudí – v reálnom živote pracujeme s hmotnými objektami, preto aj o softvéri uvažujeme a prisudzujeme mu niektoré atribúty hmotného objektu.

Ďalšou z príčin je nepochybne aj prudký rozvoj v oblasti informačných technológií. Počas tisícročnej existencie ľudstva sa počítače objavili až pred necelými šesťdesiatimi rokmi. Pritom žiadna iná oblasť ľudského poznania neprešla tak prudkým vývojom. Týka sa to nielen hardvéru, ale aj softvéru. Možnosti, ktoré nám dnes softvér, resp. IT ponúkajú sú neuveriteľné – možno hovoriť až o zmene niektorých vžitých paradigiem spoločnosti a vzniku novej, *informatickej spoločnosti*. Prudkosť vývoja však môže vyvolávať dojem, že „počítače dokážu všetko“ a ich možnosti sú neobmedzené. To, že softvéru je neuveriteľné množstvo a to na tie najrôznejšie oblasti, je možno dôvodom predstavy, že vývoj softvéru nie je zložitý a vyvinutý softvér bude „presne podľa predstáv“ zákazníka. To, že sa jedná o nehmotný objekt je možno príčinou mýtu o ľahkých zmenách, resp. udržiavateľnosti („veď nemusíte kopať kanály, ani klásť tehly, iba ťuknete párkrát do klávesnice, tak čo“). Spomínali aj, že zákazník si musí uvedomiť, že je spoluzodpovedný za vývoj projektu, prejsť od „konzumného“ pohľadu k partnerskému, „spoluautorstvu“. Pokračovať by sme mohli ďalej.

*Rybár pustí zlatú rybku, ktorá mu sľúbi, že mu splní tri želania. „Chcem mať pekný dom!“ Stane sa. „Chcem mať pekné auto!“ Stane sa. A „A aké je tvoje posledné želanie?“ Rybár sa zamyslí. „Nemohla by si zariadiť, aby moja žena bola krajšia?“ Rybka mu povie, aby ju dovedol. Keď ju však uvidí, smutne povie: „zabudni na to. Ja robím čary, nie zázraky!“*

### **Osveta zákazníkov a spoločnosti.**

Nazdávam sa, že je preto potrebná určitá „osveta“ zákazníkov a celej spoločnosti vôbec. Je potrebné uvedomiť si spomenuté skutočnosti, určite je to nevyhnutné pre vytvorenie informatickej spoločnosti, ktorá je našou ďalšou vývojovou etapou.

Pri „odbúravaní“ mýtov by mali pomôcť najmä pracovníci z oblasti vývoja informačných technológií. Pre spoločnosť by to malo zaručený prínos. Pre samotných pracovníkov takisto, a to hneď z dvoch dôvodov. Jednak, priblížili by „svoju kultúru“ ostatným. Ďalej, novší (menej skúsený) pracovníci by sa tým „pocvičili“ v schopnosti komunikovať. Ak by prednášali, nielen študentom informatických odborov, nielen študentom iných odborov či zamestnancom a širokej verejnosti, viedli by kurzy, prednášky a diskusie, reagovali na ich otázky, sledovali odozvu, skúmali pochopenie svojich myšlienok poslucháčmi a snažili sa vnímať ich „tok myšlienok“, neboli by len v pozícii odovzdávania svojich vedomostí a skúseností, ale sami by sa učili, napr. spomínanej komunikácii.

**M**ožno, že raz kurz komunikácie a rétoriky bude súčasťou štúdia softvérového inžinierstva. Možno, že raz skutočne budú existovať spomínané kurzy a prednášky odborníkov. Potom bezpochyby dôjde k tomu, že softvéroví inžinieri nebudú v pozícii zlatej rybky, ale partnera, ktorý vie seriózne komunikovať, je si vedomý svojich možností a schopností a s ktorým zákazník chce vytvoriť požadovaný produkt. Ušetrí sa veľa práce a menej projektov sa skončí neúspešne.

## Literatúra

- [Boehm00] BOEHM, B.: *The Art of Expectations Management*. Computer, Január 2000, s. 122-124.
- [Brooks95] BROOKS, FREDERICK P.: *The Mythical Man-Month: Essays on Software Engineering*. 2. vyd. Addison-Wesley, 1995. 336 s. ISBN 020135959.
- [Červenka98] ČERVENKA, R. – MEDERLY, P.: *Softvérové inžinierstvo*. prednášky MFF UK, 1998.

## Softvérový manažér

Ján Pidych

**Abstrakt.** *Zamyslenie sa nad úlohou softvérového manažéra, jeho postavením a možnosťami. Táto práca diskutuje o rozsahu vplyvu manažéra a jeho riadiacich postupov na projekt, ktorý vedie. Hlavným podkladom tejto eseje je článok „A Tale of Three Developers“ od Donalda J. Reifera [Reifer99].*

Rozvoj hardvéru napreduje míľovými krokmi (viď Moorov zákon), ale rozvoj softvéru viditeľne zaostáva a nestíha držať krok s požiadavkami, ktoré sú naň kladené. V dnešnej dobe softvérové spoločnosti, dokonca môžeme povedať, že celý softvérový priemysel čelí veľkému tlaku počas tvorby softvérových produktov. Požiadavky na softvér stále rastú, pričom čas na ich vývoj sa neustále skracuje. Tento tlak zvyšuje Internet, ktorý vytvára globálne konkurenčné prostredie a nedostatok kvalifikovaných softvérových inžinierov. Výsledkom týchto faktorov sú oneskorenia dodávok produktov, ich chybovosť či neúplná funkčnosť. V tejto situácii rastie význam riadenia projektu a z toho vyplývajúce nároky na osobu, ktorá túto pozíciu zastáva – manažér softvérového projektu.

### Problém

Donald Reifer opísal vo svojom článku tri príbehy z praxe, kde poukázal na rozličné problémy, ktoré sa vyskytli na úrovni riadenia tímu. Problémy boli odlišného charakteru. Raz sa dotýkali organizácie práce, inokedy prevažoval problém komunikácie medzi členmi tímu. Napriek tomu mali tieto príbehy niečo spoločné – problémy spôsobili manažéri softvérových projektov a výrazným spôsobom zhoršovali výkonnosť tímu.

### Príbeh prvý: Hrubá sila nerieši všetko.

Náš prvý príbeh nás zavedie do spoločnosti vyvíjajúcej produkty pre elektronické obchodovanie. Do tejto spoločnosti nastúpil nový programátor Ján. Ján sa ocitol v obklopení schopných a talentovaných spolupracovníkov a jeho šéf bol workoholik. Takéto prostredie Jána

motivovalo, tvorilo pre neho výzvu. Čoskoro mu, ale vysoké tempo vo firme prerástlo cez hlavu a každý deň bol pre Jána nová kríza. Napriek všetkému úsiliu ich projekt meškal a tak začali pracovať 12 až 14 hodín denne. Neskôr aj cez víkendy. Navyše sa počas testovania objavilo množstvo chýb. Vrcholový manažment spoločnosti požadoval od tímu týždenne správy o postupe prác a odstraňovaní chýb. Členom tímu odkázali : “Pracujte usilovnejšie a inteligentnejšie”. Ďalšie zvýšenie pracovnej doby však neurýchlilo práce na projekte.

Ako skončil tento projekt? Na základe odporúčaní konzultanta, nový manažér projekt preplánoval, zvolil realizovateľné termíny odovzdávania prác, zaviedol pravidelné kontroly postupu prác. Takisto sa zamedzilo pridávanie nových požiadaviek od marketingu. Zmeny však nevitáli všetci s otvorenou náručou. Jánov šéf považoval odporúčania konzultanta za smiešne. Rôzne prijali zmeny aj členovia tímu. Niektorí s nimi súhlasili, iní nie. Prečo projekt dospel až do štádia zlyhania a odstúpenia pôvodného manažéra? Veď tím, ako sme sa dozvedeli bol schopný a usilovný. No aj napriek tomu sa môže stať, že projekt začne meškať. Vedenie sa rozhodlo riešiť sklz nadčasmi, pracovnými víkendmi. Nepoužili nič iné. Ako sme videli sklz sa nedobehol, ale naopak, situácia sa ešte zhoršila.

Jedna z mylných predstáv o riadení tímu hovorí, že nadčasy je možné použiť vždy [Racos98]. Zvýšene úsilie môže slúžiť na preklopenie nanajvýš krátkeho obdobia, ale nie je to zázračná guľka na všetko. Pokiaľ budú členovia tímu nútení pracovať nadčasy dlho, skôr či neskôr sa to negatívne odrazí na ich výkonnosti. Pod úpadok sa podpíše viacero faktorov – únava, či stres. Nasledujú problémy v súkromnom živote. V neposlednom rade sa u pracovníkov môže prejaviť strata motivácie plniť dohodnuté termíny – veď načo, keď už aj tak celé týždne meškáme, veď aj tak to nestihneme urobiť. Na druhej strane nie je pre manažéra jednoduché posunúť termín odovzdania (zmluvy sú už podpísané, hrozia penále za nedodržanie termínov), tak ako stalo v tomto príbehu.

Medzi ďalšie možné opatrenie patrí redukcia rozsahu projektu, alebo zmena technológie. Tieto opatrenia je možné realizovať ešte zriedkavejšie ako posunutie dodávky. Zákazník nechce neúplný produkt, zmena technológie zase prináša so sebou množstvo problémov (niekedy je úplne nerealizovateľná). Manažérovi ostáva buď prídanie zdrojov – peňazí a ľudí alebo zmena riadiacich postupov. Účinnosť prídania zdrojov je prinajmenšom diskutabilná, podľa F. Brooksa prídanie ľudí do oneskoreného projektu ho ešte viac oneskorí [Brooks95]. Naopak zmena manažérskych postupov môže priniesť svoje ovocie, tak ako sa to stalo v našom prípade. Buď sú na riadiacich pozíciách nesprávni ľudia alebo sa používajú nesprávne postupy.

Ako by skončil náš príbeh bez zmeny metód riadenia a výmeny manažéra nevieme. Dobré asi nie. Manažér, ak chce byť úspešný, musí postupovať citlivo aj v otázke pracovného nasadenia. Nie hrubou silou. Napokon, aj ľudové príslovie hovorí, že hlavou múr neprerazíš.

### **Príbeh druhý: Nepodceňujte komunikáciu.**

V druhom príbehu vystupuje mladá manažérka Zuzana, ktorej prideliť projekt ešte počas jej štúdií. Jej prácou zavalený šéf ju požiadal o dozeranie nad veľkým projektom, ktorý realizoval dodávateľ. Žiaľ, dodávateľ jej zabarikádoval prístup k sledovaniu postupu prác na projekte. Jej šéf chcel, aby len hovorila čo sa má robiť a nie ako. Keďže podľa auditu projekt potreboval manažéra, dodávateľská firma si zobrala Zuzanu na tento post. Zuzana sa pustila do práce s plným nasadením. Po krátkom prehľade vo vnútri spoločnosti zistila, že používajú zastaralé metódy. Pracovníci projektu venovali celý svoj čas implementácii, aktualizácii a zjemňovaniu špecifikácie. Ich dokumentácia bola nečitateľná a nepoužiteľná. Zuzana navrhovala vytvoriť rýchly prototyp s používateľským rozhraním na ukážku pre zákazníka. V spoločnosti jej návrh odmietli ako nepraktický. Nepodržal ju ani jej šéf, ktorý tvrdil, že sa nerozumie softvéru.

Zuzana bola v ťažkej situácii. V dodávateľskej firme je nebrali vážne, nemala dôveru ani vlastného nadriadeného. Na prelomenie ich odporu potrebovala prísť s niečím novým. Jej nápad spočíval v podstrčení vlastných nápadov spolupracovníkom, aby ich považovali za svoje. Počas stretnutí jej pracovnej skupiny zaznamenala úspech pri komunikácii s dodávateľom. Kľúč k prelomeniu bariéry spočíval v uplatnení pravidla pravej ruky. Takýto spôsob diskusie posilňuje práva rečníka, ostatní ho nesmú prerušovať. Nasledujúci rečník je napravo sediaci od pôvodného. Tento spôsob naučil ľudí v skupine počúvať jeden druhého a ponúkol najlepšie možnosti na dosiahnutie dohody. Na zmenu situácie v našom prípade prispela (dokonca rozhodla) zmena v kultúre komunikácie.

Manažér by mal napomáhať takému spôsobu komunikácie, kde sa členovia tímu rešpektujú, počúvajú a uznávajú aj názor druhého. Ľudia sú od prírody rôzni, niektorí sú utiahnutí, iní sa naopak v spoločnosti cítia ako ryba vo vode. Nie všetci sa preto pri stretnutiach môžu rovnako prejaviť. A práve manažér by mal zabráňovať vzniku nerovnocenného postavenia pri diskusii. Bolo by škoda nezaznamenať nápad len preto, že sa jeho autor nedostal k slovu.

Dobrý manažér sa takisto nemôže obmedziť len na pravidelné (formálne) stretnutia s tímom. K svojim podriadeným musí byť k dispozícii v čo najväčšej miere. Zamestnancov zvyčajne trápia rôzne problémy a preto by mali mať možnosť ich riešiť inde ako na pravidelnom stretnutí, kde sa aj tak preberá stav projektu. Pevnejšiemu vzťahu medzi manažérom a tímom prispieva neformálna komunikácia. Neformálne stretnutie sa nemusí uskutočniť len na pracovisku, ale aj v oveľa príjemnejšom prostredí – reštaurácia, športový klub. V uvoľnenejšej atmosfére možno vyriešiť mnohé problémy, možno ľahšie dospieť k dohode. Neformálne stretnutia upevňujú priateľstvá a budujú vzájomnú dôveru, ktorá je pre tímovú prácu veľmi dôležitá. Manažér, ktorý riadi projekt, musí byť počúvaný členmi tímu. Ale takisto musí počúvať svojich ľudí. Ale ani to nestačí. Aj členovia tímu sa musia vedieť

počúvať, ak ich tímová práca má byť efektívna. Ani na túto úroveň komunikácie by dobrý manažér nemal zabúdať.

### **Príbeh tretí: Manažér je tu pre projekt a nie projekt pre manažéra.**

Tento príbeh sa odohráva v prostredí veľkej telekomunikačnej spoločnosti. Náš sprievodca príbehom sa volá Pradeep a práve k tejto firme nastúpil. Jeho nadriadený ho na začiatok priradil k údržbe 20 ročného softvéru, napísaného v C a assembleri. Počas "tréningu" sa Pradeep zdokonalil v technickej oblasti a získal vedomosti o modele vyspelosti procesu (CMM). Zamestnanci spoločnosti boli hrdí na to, že ich spoločnosť bola ohodnotená CMM stupňom 3. Aj Pradeep bol hrdý na to, že pracuje v spoločnosti, ktorá používa metódy, ktoré jeho vyučujúci na univerzite označili ako najlepšie. Tréning skončil a Pradeep bol presunutý k malému tímu. Koncom roka spoločnosť získala dokonca CMM úroveň 4 a začala aplikovať rovnaký model práce na všetky projekty.

A kde je problém v tomto prípade? Organizácia práce je na skvelej úrovni, nikde nie je zmienka o meškaní projektu alebo iných problémov. Lenže nie je všetko zlato, čo sa blyští. Presun myšlienok z teórie do praxe nie je jednoduchý. V Pradeepovej spoločnosti manažéri, ktorí dozerali na kvalitu procesu tvorby projektu (v článku nazývaní "project police") nútili každého dodržiavať štandardy do písmena. Takýto spôsob tvrdo dopadol najmä na malé projekty. Situácia sa dostala až do štádia, keď členovia tímov úmyselne zavádzali chyby do programu, ak sa pri testovaní ukázalo, že ich je menej ako sa očakáva. T.j. menej ako tabuľková hodnota, ktorou sa riadili manažéri. Každá odchýlka bola podozrivá.

Manažéri sa pustili jednou cestou, nezohľadňujúc špecifické vlastnosti jednotlivých projektov. Začali aplikovať jeden formát na všetky projekty – veľké, malé, údržbu, výskum či vývoj. Manažér má napomáhať projektu, je to on kto "služi" projektu a nie projekt manažérovi. Formálne postupy, merania napredovania prác, správy o stave projektu sú tu na uľahčenie práce pri riadení a realizácii projektu. Všetky tieto manažérske postupy sú prostriedkom a nie cieľom. Prostriedkom k riadeniu a sledovaniu projektu. Ak sa na toto zabudne, z prostriedkov na stanú ciele činnosti manažéra. Riadiace postupy skostnatejú a namiesto uľahčenia práce ju členom tímu zhoršujú. Pokiaľ sa manažéri zmenia na kontrolórov, ktorí len sledujú dodržiavanie pravidiel, mali by byť vymenení. A opäť sa dostáva do popredia otázka citlivého, uváženeho prístupu vedúceho pracovníka. Aj projekty z jednej oblasti môžu (a zvyčajne) sú rôzne. Nemožno ich zaradiť do jednej škatuľky. K tomu je nevyhnutné, aby manažér mal znalosti a hlavne skúsenosti z problémovej oblasti. Preto sa ako ideálne javí, dosadzovať na miesta riadiacich pracovníkov bývalých odborníkov – návrhárov, programátorov a ďalších. Lenže kto je dobrý programátor nemusí byť ešte dobrý manažér. A nie každý dobrý manažér projektu je dobrý manažér kontroly procesu, tak ako sa to stalo v našom poslednom príbehu.

Na manažéra (nielen softvérového) číha množstvo nebezpečenstiev. Tak ako sa postupne zvyšuje komplexnosť softvérových produktov, tak stúpajú aj nároky na riadenie takéhoto projektu. V ľuďoch je často zakorenená predstava, že hlavné problémy sú technického rázu [Racos98]. Nie je to pravda. Mnohé problémy skončili neúspešne kvôli organizácii práce, či problémom v komunikácii. Všetci isto poznáme príbeh o Babylonskej veži [Brooks95]. Členovia tímu sú jedinečné ľudské bytosti, dosahujú nielen rôznu pracovnú výkonnosť, ale majú aj rôzne spoločenské ctenie. Niektorí spolupracujú s ostatnými rád, iní radšej pracujú sám. Medzi členmi tímu sa môžu vyskytnúť problémy, ktoré nesúvisia, resp. nevznikli na pracovisku. Na to, aby manažér zvládol ukontrolovať vývoj vo svojom tíme, aby ľudia dokázali motivovať, usmerniť ich, aby ťahali za jeden povraz, musí byť dobrý psychológ. Tu mu jeho znalosti z problémovej oblasti nepomôžu. Musí so svojím tímom „žiť“, aby zachytil vznikajúce problémy a nie až keď sa prejavia v plnej sile. Manažéri často zasahujú, až keď morálka tímu dosiahla dno [Mirantes97]. Vtedy je už ale neskoro.

### Literatúra

- [Brooks95] BROOKS, FREDERICK P.: *The Mythical Man-Month: Essays on Software Engineering*. 2. vyd. Addison-Wesley, 1995. 336 s. ISBN 020135959.
- [Mirantes97] MIRANTES, A.: *Keeping a good software team*. 1997. <http://www.baz.com/kjordan/swse625/htm/tp-am.htm>.
- [Racos98] RACOS, J.: *Project Teams – Myths and Reality*. 1998. <http://www.dot.ca.gov/hq/projmgmt/CHRONOLOGY/Internet/PMI/PDF/PD27.PDF>.
- [Reifer99] REIFER, D. J.: *A tale of three developers*. Computer IEEE, November 1999, s. 128-130.



## Kolízie softvérových modelov – ako sa im vyhnúť?

Podnetom na tvorbu tejto eseje je článok *Avoiding the Software Model-Clash Spiderweb*, ktorého autormi sú Barry Boehm, Dan Port a Mohamed Al-Said, publikovaný v časopise IEEE Computer, november 2000.

Stanislav Hrk

**Abstrakt.** *Tvorcovia softvérových systémov sú prinútení balansovať medzi viacerými, potenciálne konfliktnými záujmami. Projekty, v ktorých sa to nepodarí sa dostávajú do situácie, ktorú veľmi výstižne môžeme opísať ako uviaznutie v dechtovej diere, z ktorej sa aj mohutné a silné šelmy (výrobcovia softvéru) ťažko vyslobodia. Táto esej pojednáva o príčinách problémov, ktoré vedú k uviaznutiu softvérového projektu a spôsobe, ako tieto problémy včas odhaliť a vysporiadať sa s nimi, skôr ako ich dôsledky spôsobia neúspech celého projektu. Spomenieme aj prístup k vývoju softvéru, ktorý pomáha identifikovať a vyhnúť sa kolíziám medzi triedami modelov, podľa ktorých sa softvérový produkt riadi a vyvíja.*

Rovnako názorne ako metaforu dehtovej diery, je na projekt v ťažkostiach použiteľná metafora pavučiny. Problematický projekt je zosobnený hmyzom, ktorý zletel do lepkavej pavučiny a snaží sa odtiaľ dostať skôr ako príde pavúk, zobrazujúci vypršanie časového rozvrhu alebo rozpočtu. Pavučina reprezentuje rôzne ohraničenia a problémy zapríčinené kolíziou medzi softvérovými modelmi. Kolízie modelov sú odvodené z úspechových modelov daných zainteresovanými stranami v projekte. Hlavní držitelia podielu sú vo väčšine prípadov sami užívatelia systému, kupci systému, vývojári a údržbári, a možný je aj výskyt ďalších zainteresovaných strán v závislosti od typu projektu. Pre vysvetlenie kolízií medzi modelmi, treba najprv vyjasniť, čo sú to vlastne modely a aký majú význam pri vývoji softvérových produktov.

### **Všadeprítomné modely**

Prvým krokom vo vývoji softvérového produktu je jeho vizualizácia. Predstaviť si výsledný produkt z daných, často nejasných alebo nepostačujúcich poznatkov z danej problémovej oblasti je zložitá úloha pre všetky zainteresované stránky, zoskupené okolo projektu. Pomôckou, ktorú pri tomto používame, často celkom prirodzene a nevedomujúc si to, sú rôzne modely. Modely môžu byť vzory, ktoré sledujeme alebo analógie, ktoré používame pri usudzovaní o tom, čo vlastne vytvárame a akú funkciu to má vykonávať. Bez ohľadu na formu v ktorej sa javia, modely sú všadeprítomné. Vývojári ich využívajú pri stavbe systémov ľubovoľnej veľkosti, zákazníci na zobrazenie toho, čo si myslia že dostávajú od vývojárov.

Vytváranie, zlepšovanie a údržba informačného systému zahŕňa výstavbu štyroch základných modelov: úspechu, produktu, procesu a vlastnosti. Tieto modely sa nevyvíjajú jeden po druhom, ale súbežne a počas celého vývoja produktu.

Modely úspechu sú najvýznamnejšie, aj keď je samotný úspech projektu často zložitý definovať. Je úspech iba keď sa v určitej miere vráti investícia a dosiahne profit? Alebo je úspech iba keď sa uspokojia všetky zainteresované strany v danom projekte? Na tieto otázky zvyčajne nie je jednotný názor medzi držiteľmi podielu, a je možné, že sa bude líšiť u každého z nich. Modely úspechu ktoré sa najčastejšie používajú sú model správnosti, model výhra-výhra (stakeholder win-win), bussiness-case model a model IKIWISI (I'll Know It When I See It, alebo budem to vedieť, keď to uvidím). Posledný model je odvodený zo skúseností v prípadoch, keď bolo požadované od používateľov špecifikovať požiadavky na užívateľské rozhranie. Model výhra-výhra sa často volí ako flexibilnejšia alternatíva definovania tvrdých požiadaviek na úspech projektu. Úspech projektu sa definuje pomocou súboru podmienok výhry, o ktorých sa môže vyjednávať, radšej ako tvrdými požiadavkami okolo ktorých jednanie nie je možné.

Modely produktu zahŕňajú rôzne spôsoby ako špecifikovať pracovné koncepcie, požiadavky, architektúry, návrhy a zdrojový kód, a taktiež aj ich vzájomné vzťahy.

Modely procesu popisujú ako sa projekt bude vyvíjať. Medzi známe procesné modely patria evolučný model, model inkrementálneho vývoja, špirálový model, model rýchleho vývoja aplikácií (Rapid Application Development - RAD), model adaptívneho vývoja a mnohé iné.

Modely vlastnosti definujú požadovanú alebo prijateľnú mieru kompromisov pre faktory ako sú výkon, spoľahlivosť, bezpečnosť, prenosnosť, vyvíjateľnosť a znovupoužiteľnosť.

### **Keď sa modely zrážajú**

Modely sú veľmi silným a hlboko zakoreneným nástrojom vo vedomí ľudí. Keď sa však stane, že sa medzi sebou zrážajú a ako následok tých

zrážok sa projekt dostane do problémov, je zložité odhaliť pravý zdroj problémov. Zriedkavé je, že sa pre problémy obviňujú modely. Namiesto toho sa hľadajú povrchové lieky pre daný problém, ako sú redukcia cieľov, pridanie ľudí do projektu, zmena manažéra, zakúpenie ďalších vývojových prostriedkov. Tieto snahy väčšinou zostávajú bez výsledkov a často situáciu ešte zhoršia.

Modely sa spoliehajú na rôzne predpoklady, ktoré však nie vždy musia byť správne a zhodné s predpokladmi iných modelov. Kolízia modelov odzrkadľuje nezrovnalosti medzi predpokladmi rozličných modelov úspechu, procesu produktu a vlastnosti, ktorými sa daný projekt riadi. Keď príde ku ich kolízii, vytvára sa atmosféra zmätku, nedôvery, stresu. Chybné časti je potrebné prerábať alebo celkom zahodiť, čo vytvára ďalšie problémy s časom a finančnými prostriedkami. Všetko toto prispieva k tomu, aby vývojár, ale aj iní zúčastnení v projekte mali pocit, akoby sa s obrovskou námahou ťahali cez projekt ako cez dechtovú diery.

Názorný príklad nesprávnych predpokladov o ktoré sa model opiera je známe Zlaté Pravidlo. Aj keď sa na prvý pohľad zdá, že toto tvrdenie vždy musí byť správne, keď sa ono uplatní pri tvorbe softvérových produktov stáva sa zdrojom mnohých nedorozumení, konfliktov a neúspechov. Zlaté Pravidlo, ktoré znie *Rob pre iných, ako si praješ aby oni robili pre teba*, sa v kontexte vývoja softvéru môže pretransformovať na *Vyvíjaj systém pre iných, predpokladajúc že oni tiež vyvíjajú softvér a vedia mnoho o počítačoch*. Softvérové produkty vyvíjané podľa tohto modelu úspechu sú síce efektívne a flexibilné, ale nezrozumiteľné pre obvyčajných používateľov a považované za neúspešné. Aj keď vyhovelí modelu úspechu Zlaté Pravidlo ktorým sa viedli, nevyhoveli modelu úspechu víťazné podmienky držiteľov podielu (stakeholder win-win). Dôvod neúspechu je v predpoklade, ktorým bolo Zlaté Pravidlo podložené: *Všetci sú podobní*. Zrejme, toto tvrdenie je nesprávne a väčšina používateľov ma celkom iný pohľad a požiadavky na softvér, ako ľudia ktorý ho vyvíjajú. Ako riešenie ponúka sa Platinové Pravidlo: *Rob pre iných, ako si to oni želajú*.

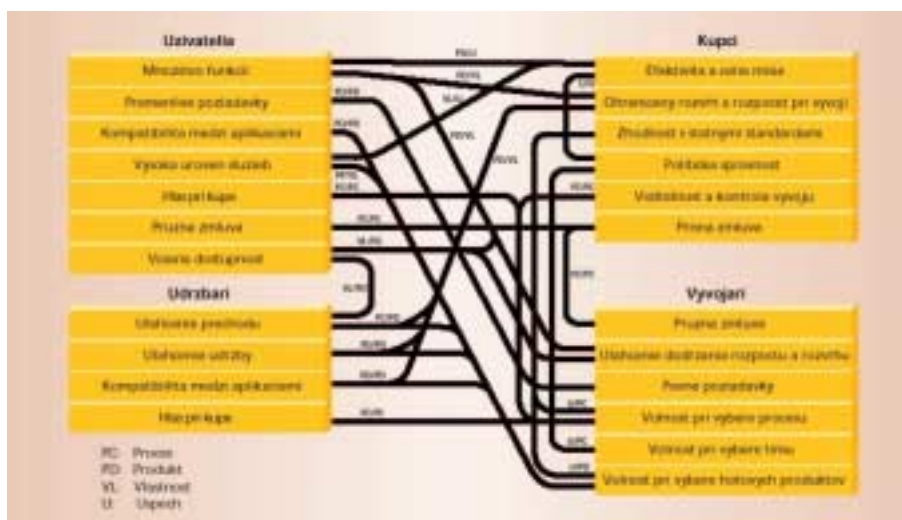
Kolíziu medzi modelmi typu proces a produkt dobre ilustruje príklad kolízie medzi vodopádovým modelom a modelom využitia hotových produktov. Predpoklady, o ktoré sa opiera vodopádový model sú, že požiadavky a zdroje sú vopred pevne definované a neexistujú žiadne vysoko rizikové dôsledky týchto požiadaviek. Model využitia hotových produktov predpokladá, že sú požiadavky formované na základe možností hotových produktov. Možné problémy vyplývajúce z týchto predpokladov sú, že vopred zadané požiadavky vo vodopádovom modeli nemusia byť realizovateľné pomocou dostupných systémov alebo sa to nemusí oplácať. Ak sú požiadavky dané v zmluve, prichádza k ďalším problémom a drahým súdnym sporom pokiaľ sa nedosiahne kompromis a nereálne požiadavky sa zrušia. Riešenie, ktoré sa môže použiť v tomto prípade je súbežne vyvíjať požiadavky a manažment rizík pri používaní hotových produktov.

Príklad kolízie medzi modelmi typu proces a úspech je kolízia medzi vodopádovým a IKIWISI modelom. Vodopádový model predpokladá, že sú požiadavky známe skôr ako sa začne s implementáciou a nemenia sa. Predpoklad IKIWISI je, že používatelia najlepšie formulujú požiadavky na používateľské rozhranie interakciou s fungujúcimi prototypmi. Požiadavky sa menia v súlade s reakciami používateľa na prototyp. Keď si vývojár na začiatku stanoví tvar používateľského rozhrania ako časť pevných požiadaviek, takýto prístup často produkuje výsledný systém, ktorý síce spĺňa stanovené požiadavky, ale zlyháva pri skúške používateľským IKIWISI modelom úspechu.

Kolízie medzi modelmi úspechu sú asi najzávažnejšie. Keď modely úspechu navzájom kolidujú a tieto kolízie sa včas neodstránia, je priam nemožné priviesť projekt k úspešnému koncu tak, aby sa vyhovelo podmienkam výhry všetkých držiteľov podielu v projekte. V prípade, že niektorý z modelov úspechu padne, často sa javí tzv. Dominový efekt, že so sebou potiahne aj iné modely úspechu. Príklad tohto je projekt, pre ktorý boli stanovené dva konfliktné modely úspechu, a to veľkú priepustnosť systému a ohraničený vývojový rozpočet. Výsledný systém síce vyhovelo požiadavke ohraničeného rozpočtu, ale priepustnosť systému bola taká nízka, že celý systém bol prakticky nepoužiteľný. Na konci už nikoho nezajímalo, že sa podarilo udržať nízku cenu vývoja, keď projekt bol označený ako neúspech.

### **Pavučina modelových kolízií**

Aj keď v podstate môže vzniknúť kolízia medzi každým typom modelov, pri vývoji informačných technológií sa niektoré kolízie vyskytujú častejšie ako iné. To pramení v opakujúcej sa kombinácii najčastejších modelov úspechu zadávaných najčastejšie sa vyskytujúcimi držiteľmi podielu pri informačných projektoch. Títo držitelia podielu sú používatelia, zákazníci, vývojári a údržbári systému. Štúdiu mnohých prípadov projektov, ktorý sa skončili neúspechom, autori v [Boehm00] identifikovali niektoré najčastejšie vyskytujúce sa kolízie medzi modelmi, a graficky ich vyjadrili v podobe pavučiny modelových kolízií (Model-Clash Spiderweb) na obr. 1, ktorý zobrazuje najčastejšie modely úspechu definované najčastejšie sa vyskytujúcimi držiteľmi podielu. Taktiež sú vyznačené aj najčastejšie sa vyskytujúce kolízie medzi týmito modelmi.



Obr. 1: Pavučina modelových kolízií [Boehmoo].

### Ako sa vyhnúť kolíziám modelov?

Včasným odhalením kolízií modelov, skôr ako sa urobia formálne záväzky, je možné predísť problémom a zabrániť stroškotaniu projektu. Aj pri projektoch, pri ktorých sa problémy už vyskytli, je možná záchrana identifikáciou kolízií medzi modelmi a podniknutím opatrení, ktorými sa tieto kolízie odstránia.

Autori v [Boehmoo] ponúkajú dve techniky na vyhnutie sa problémom spôsobených kolíziami modelov. Prvá sa zameriava na odhalenie kolízií pri projektoch, ktoré sa už začali. Druhá technika, keď sa používa na riadenie vývoja projektu, by mala umožniť celkom sa vyhnúť kolíziám medzi modelmi.

### Diagnóza

Na diagnózu kolízií v projekte, ktorý je už rozbehnutý sa používa technika revízie projektu pomocou pavučiny modelových kolízií. Táto technika si vyžaduje scenár, pri ktorom sa analytik dostáva pred úplne neznámy rozsiahly projekt, a žiada sa od neho aby urobil revíziu projektu a odporučil postup vývoja. V tomto scenári, analytik je konfrontovaný s množstvom informácií prezentujúcich projekt z rôznych aspektov, z ktorých sa on snaží vytvoriť si čo celkovejší obraz o problémoch vyskytujúcich sa v projekte.

Použitím pavučiny modelových kolízií sa táto reakčná situácia pre analytika môže obrátiť na proaktívnu. Namiesto pripravených prezentácií o projekte, manažment sa požiada o sumarizáciu hlavných držiteľov podielu v projekte a ich modeloch úspechu. Pokým sa toto pripravuje, analytik využije čas na identifikovanie základných kolízií medzi modelmi pomocou dostupných materiálov o projekte, a zakreslí ich do pavučiny. Po obdržaní správy od manažmentu sa tieto modely doplnia do pavučiny

a revízia pokračuje v diskusii o najzávažnejších identifikovaných kolíziách v modeloch a spôsoboch, ako sa s nimi vyrovnáť.

### Prevenencia

Na vyhýbanie sa kolíziám modelov je navrhnutý prístup k vývoju softvérových systémov pod menom MBASE (Model-Based System Architecting and Software Engineering). Použitím tohto prístupu možno identifikovať a vyhnúť sa kolíziám modelov priebežne počas vývoja softvérového projektu, skôr ako sa ich dôsledky negatívne prejavia na projekte.

MBASE používa modifikáciu špirálového modelu vývoju známu ako špirála typu výhra-výhra (Win-Win Spiral), ktorá rozširuje pôvodný špirálový model dvoma významnými spôsobmi:

- Používa model výhra-výhra na určenie cieľov, obmedzení a alternatív v každom cykle špirály.
- Používa sa skupina bodov zakotvenia (Anchor Points) v životnom cykle ako kritické body pre pokračovanie v manažmente projektu.

Špirála typu výhra-výhra zdôrazňuje vyjednávanie s držiteľmi podielov v projekte pomocou modelu úspechu výhra-výhra, ktorým sa určujú ciele, obmedzenia a alternatívy systému. Dôraz je i na manažmente rizík, ktoré sa včas identifikujú a riešia pomocou metód ako je napríklad prototypovanie.

MBASE zavádza tri body zakotvenia projektu, ktoré slúžia ako míľniky, v ktorých hlavní držitelia podielu v projekte prehodnocujú kľúčové body projektu: opis spôsobu prevádzky, výsledky prototypovania, opis požiadaviek, opis architektúry, plán životného cyklu projektu a odôvodnenie uskutočniteľnosti projektu. Na posledný bod je zvlášť kladený dôraz, lebo sa v nej odpovedá na kľúčovú otázku o pokračovaní projektu, ktorá v sebe zahŕňa výsledky všetkých ostatných bodov: "Ak budeme stavať projekt používajúc zvolené procesy a architektúru, bude on podporovať zvolený spôsob prevádzky, uskutoční výsledky prototypovania, uspokojí kladené požiadavky a skončí v rámci daného rozpočtu a časového plánu?". V prípade, že sa na túto otázku odpovie kladne, pokračuje sa vo vývoji po nasledovný míľnik. Viac informácií MBASE sa nachádza v literatúre [Boehm99a], [Boehm] a [Boehm99b].

### Z vlastnej skúsenosti

Opísané postupy a techniky sú založené na mnohých prípadových štúdiách veľkých projektov, ktoré sa skončili neúspechom. V každom z týchto projektov je možné identifikovať niekoľko závažných kolízií medzi modelmi, ktoré zapríčinili ich fiasko.

Keďže osobne som zatiaľ nezapríčinil neúspech žiadneho multimilionového projektu, môžem iba uviesť skúsenosti z projektov na ktorých som účinkoval alebo práve účinkujem. Aj keď sú to všetko

menšie projekty v rámci predmetov vyučovaných na fakulte, čo znamená že prebiehajú v dosť neštandardných podmienkach, je zaujímavé aplikovať opísané postupy aj na takéto projekty. Keďže niektoré z nich práve bežia, je možné výsledky takej analýzy použiť na vyjasnenie zdrojov problémov a ich zmiernenie.

Prvý z projektov, ktorý chcem spomenúť je záverečný projekt, ktorý som riešil v poslednom roku bakalárskeho štúdia na Fakulte. Zainteresované strany v projekte boli používateľ a vývojár, čím situácia projektu bola podstatne zjednodušená. Výsledok projektu mal byť interaktívna aplikácia so silnou orientáciou na užívateľské rozhranie. Už pri samom začiatku projektu bolo jasné, že ide o vysoko rizikový projekt, kvôli nedostatočnému poznaniu problémovej oblasti, slabej skúsenosti s vývojovými prostriedkami a jedným z použitých programovacích jazykov, a slabej zdokumentovanosti a rozširovateľnosti produktu, na ktorý sa mal projekt nadväzovať. Ako model procesu bol zvolený vodopádový model. Viedlo to ku kolízii modelov typu proces-úspech, medzi vodopádovým modelom a modelom úspechu IKIWISI. Model IKIWISI bol zvolený, aj keď v tom čase nevedome, z potreby vytvoriť používateľské rozhranie, ktoré by zodpovedalo požiadavkám používateľa. Táto kolízia sa riešila osvojením si neformálneho modelu procesu, ktorý sa podobal špirálovému modelu. Druhý typ kolízie, ktorý vznikol bol produkt-produkt, ktorý vznikol medzi modelom úspechu Premennivé požiadavky zo strany používateľa, a modelom úspechu Stabilné požiadavky zo strany vývojára. Ďalší typ kolízie viditeľný v pavučine modelových kolízií bol typu produkt-vlastnosť, kde prišlo do kolízie modelov Množstvo funkcií na strane užívateľa a Uľahčenie dodržania rozpočtu a rozvrhu na strane vývojára. Tieto kolízie neboli vyriešené a spôsobili mnoho ťažkostí počas celej doby trvania projektu. Záverečný projekt skončil pomerne úspešne, aj pri vysokých rizikách a problémoch, ktoré sa vyskytli.

Druhý z projektov je Tímový projekt, ktorý sa v čase písania tejto eseje blíži k finálnej fáze. Cieľom projektu je vytvoriť systém pre počítačovú podporu hodnotenia programov pre predmet Programovanie jazyku C a rieši ho tím zložený z piatich osôb počas dvoch semestrov. V jeho prípade by som chcel zdôrazniť iba jednu kolíziu modelov, a spôsob ako sa s ňou vyrovnalo. Táto kolízia modelov sa zjavila medzi modelmi typu úspech a produkt. Model úspechu bol zadaný ako podmienka výhry, znejúca že systém musí byť vyskúšaný v prevádzke do konca druhého semestra. Model produktu znel, že systém je určený pre podporu hodnotenia programov pre predmet Programovanie v jazyku C. Pre druhý model bol nevedome vytvorený nesprávny predpoklad, že je systém určený pre hodnotenie programov napísaných v jazyku C. Ako sa neskôr zistilo, pri písaní programov pre predmet Programovanie v C je povolené používať aj črtý jazyka C++. Možné riešenia boli zakázať používanie črt jazyka C++ v programoch pre tento predmet, alebo prerobiť systém tak, aby podporoval programy vytvorené aj v jazyku C++. Obe tieto riešenia boli nevyhovujúce, lebo ich nebolo možné uskutočniť termíne stanovenom v podmienkach výhry. K riešeniu sa prišlo

technikou vyjednávania o podmienkach výhry medzi držiteľmi podielu v projekte. Ako riešenie sa prijalo, že sa funkčnosť systému otestuje iba na podmnožine programov, ktoré budú napísané v jazyku C.

Ako z uvedených príkladov môžeme vidieť, kolízie modelov sú zdroje problémov ako u veľkých, tak aj u malých projektov. Neuvedomovaním si príčin týchto problémov, sa projekt veľmi rýchlo dostane do ťažkej situácie, často bez východiska. Početné veľké projekty z minulosti, ktoré sa udusili v svojich problémoch, stoja ako upomienka a výstraha novým generáciám manažérov.

Uplatnením postupov, ako sú technika revízie projektu pomocou pavučine modelových kolízií alebo systém MBASE je možné zdroje problémov odhaliť a eliminovať alebo im sa im včas vyhnúť a zabrániť tým, aby sa práve riadený projekt pridal k dlhému zoznamu tých neúspešných.

## Literatúra

- [Boehm] BOEHM, BARRY – PORT DAN: *Escaping the Software Tar Pit: Model Clashes and How to Avoid Them*.  
<http://sunset.usc.edu/TechRpts/Papers/usccse98-517/usccse98-517.pdf>.
- [Boehm99a] BOEHM, BARRY – PORT, DAN: *When Models Collide Lessons from Software Systems Analysis*. IT Pro, Január/Február 1999, s. 49 - 56.
- [Boehm99b] BOEHM, BARRY ET AL: *MBASE Life Cycle Architecture Milestone Package*. Proc. of the 1 st Working International Conference on Software Architecture, 1999.
- [Boehm00] BOEHM, BARRY – PORT, DAN – AL-SAID, MOHAMMED: *Avoiding the Software Model-Clash Spiderweb*. IEEE Computer, November 2000. s. 120-122.

## Tajomstvo úspešného softvérového projektu

Marián Šimo

**Abstrakt.** Esej opisuje vývojový systém pre tvorbu softvérového produktu. Snaží sa opísať niektoré problémy týkajúce sa tohoto systému a načrtnúť možné cesty ich riešenia. Ako podklad tejto eseje je článok Jamesa Bullocka v časopise *Computer* [Bullock99], ktorý sa zaoberá zlepšením procesu vývoja systému.

Vývoj softvérového produktu je pre softvérového inžiniera hlavnou pracovnou náplňou. Cesta od myšlenky vytvoriť systém po konečnú podobu vo forme programu, ktorý plní svoje poslanie je nesmierne dlhá a zložitá. V rámci jednotlivých etáp pri vývoji produktu číha na tvorcu softvérového systému mnoho potenciálnych problémov. Tieto problémy vyplývajú z podstaty tvoreného systému ako aj z použitých techník pri riešení.

### Vývojový systém

Vývojový systém predstavuje ľudí pracujúcich na projekte, činnosti a nástroje, ktoré realizujú proces vytvárania softvérového produktu. Ak si zadefinujeme, že vývoj softvérového produktu je transformácia požiadaviek na cieľový kód programu, potom vývojový systém je nástroj, pomocou ktorého sa táto premena vykoná. Vývojový systém môžeme teda chápať aj ako informačný systém, ktorý spracúva rôzne pohľady a opisy vytváraného softvérového produktu. Tento systém zahŕňa v sebe priamo procesy, ktoré implementujú postupnosti požiadaviek na kód. Okrem týchto procesov systém obsahuje aj podporné procesy ako napríklad zmena manažmentu, rozlíšenie rizík a samozrejme časový plán projektu.

Takisto ako iné systémy aj vývojový systém je popísaný vlastnými funkciami, ale obsahuje aj niekoľko atribútov, ktoré hovoria o tom, ako presne dodržiava vývojový systém túto funkcionálnosť. Niekoľko dôležitých atribútov vývojového systému:

- Objem procesu

- Kapacita procesu
- Súbežnosť prác
- Časový cyklus
- Operačné atribúty
- Rozšírenosť
- Meranie procesu

### **Objem procesu**

Je ťažšie spracovať 10000 vecí ako 100. Bude niečo, čo funguje pre 100 požiadaviek, fungovať aj pre 10000 požiadaviek?

Túto istú otázku si môžeme položiť aj pre ďalšie časti vývojového systému (výskyt chýb, kompilácia, zostavenie aplikácie a testovanie produktu).

Zoberme si príklad. Máme systém zložený z veľa malých komponentov. Zmenou nejakého rozhrania v systéme môže prísť k veľmi veľa zmenám v ďalších komponentoch, ktoré používajú toto rozhranie. Pre malý systém, kde nie je veľa komponentov, ktoré sú závislé na danom rozhraní, sa táto úprava nemusí dramaticky odzkradliť. Vo veľkom systéme, kde je veľmi veľa komponentov a taktiež ľudí, ktorí na tomto projekte pracujú vzniká problém. Zmenili sme rozhranie, zmenili sme ho všetkým komponentom? Dozvedeli sa o tomto aj ostatní členovia tímu, ktorí používajú toto rozhranie? Na ulahčenie riešenia tohoto problému môžeme použiť nástroj. V praxi sa používa metóda vybudovania vedomostnej základne. Spravidla sa vytvorí databáza, v ktorej sú uchované všetky dokumenty týkajúce sa projektu. Systém spravujúci takúto databázu musí byť schopný poskytovať na požiadanie informácie o zmenách, ktoré nastali, kto tieto zmeny vykonal a čo zmena zasiahla. Malo by platiť pravidlo, že do systému sa údaje dajú len pridávať. V žiadnom prípade by systém nemal umožniť mazanie dokumentov, pretože by mohla vzniknúť nekonzistencia materiálov o projekte a odzkradliť sa to môže v chybe návrhu systému. Naopak systém musí byť schopný podporovať v plnej miere správu verzií jednotlivých dokumentov, uchovávať všetky verzie a poskytnúť informáciu o rozdieloch medzi jednotlivými verziami dokumentu.

Pán Bullock vo svojom príspevku opisuje spôsob uloženia vťahov a závislostí jednotlivých elementov v projekte BSY-2 (bojový systém pre ponorku SeaWolf). Pracoval v skupine, ktorá mala na starosti udržiavanie niekoľkých návrhových špecifikácií v relačnej databáze. Špecifikácia sa vyznačovala veľkým počtom závislostí typu mnoho – mnoho. Tento typ závislosti nie je ideálny pretože spôsobuje nasledujúci problém. Jedna zmena v elemente môže zasiahnuť veľký počet iných elementov a príslušných dokumentov. Projekt dosiahol také rozmery, že manuálne prehľadávanie týchto dokumentov a sledovanie zmien, ktoré nastali bolo ťažkopádne a náchylné na pomýlenie sa. Tým, že špecifikácie boli uložené

v databáze, docielila sa automatizácia vyhľadávania zmien. Na vysledovanie konkrétnej zmeny v špecifikácii postačoval iba relatívne jednoduchý dopyt do databázy. Táto organizácia bola úspešná a pomohla v realizácii projektu. Problém, ktorý však nastal bola zmena štýlu práce. Zavedením siete pracovných staníc sa zmenil aj štýl práce na projekte. Celé publikovanie sa realizovalo na pracovných stanicach (decentralizovane). Stačilo mať v systéme niekoľko špecifikácií a celý systém sa stal neprehľadným, pretože existovalo aj niekoľko rôznych verzií toho istého dokumentu, ale každá na inom mieste. Zlyhala organizácia správy verzií. Zmenilo sa prostredie (hardvér), v ktorom bol vývoj systému realizovaný, nebol však tejto zmene prispôbený model vývojového systému.

### Kapacita procesu

Je reálne, aby bol spracovaný celý požadovaný objem v dostupnom čase?

Odpoveď na túto otázku nie je jednoduchá. Neexistuje univerzálne pravidlo, ktoré by objem vyhodnotilo a povedalo jednoznačnú a presnú odpoveď. Na pomoc v riešení takejto otázky nám prichádza niekoľko techník odhadov. Existujú dve skupiny odhadov:

- zhora nadol: Dekomponujeme systém na menšie podsystemy, ktoré potom odhadujeme. Je potrebné byť opatrný, pretože ľahko môžeme podhodnotiť drobné problémy tzv. implementačné detaily.
- zdola nahor: Systém vidíme ako súbor podsystemov. Z týchto podsystemov vytvárame systém. Je potrebné mať aspoň nejaký návrh systému. Pri tomto type odhadu je zvýšené riziko prehliadnutia niektorých činností v rámci celého systému, čo vedie k nepresnosti odhadu.

Najjednoduchším odhadom (zvyčajne aj najmenej presným) môže byť expertný odhad. Použijú sa tu údaje z predchádzajúcich projektov, ktoré sa už skončili. Na základe skúsenosti z týchto projektov sa stanoví čas a náklady na vyriešenie súčasného projektu. Tento odhad je nebezpečný, pretože nemusí s dostatočnou presnosťou odhadnúť výsledok. Vylepšiť takýto odhad je možné použitím údajov z podobného projektu. Pre určenie času potrebného na vyriešenie nejakého problému je vhodné poznať aký rozsah má daný problém. Pre vývoj softvéru existuje niekoľko metód ako odhadnúť rozsah softvérového projektu.

Najprimitívnejšia metóda odhadu je odhadnutie dĺžky textu programu. Dĺžka sa vyjadruje v rôznych parametroch (napríklad počet oddelovačov, počet dodaných príkazov, počet vytvorených riadkov, počet znakov v texte programu, veľkosť programových súborov, počet strán výpisu programu a pod.). Táto metóda odhadu závisí od použitého programovacieho jazyka, preto bola vyvinutá univerzálnejšia metóda funkčných bodov.

Základom metódy funkčných bodov je špecifikácia požiadaviek [Bieliková00]. Metódou sa vyhodnocuje počet služieb systému, ktoré sú

dostupné používateľovi (aplikácie komunikujúce s používateľom). Výborné odhady zložitosti softvérového systému sa dosahujú u databázových systémoch. Nakoľko metóda odhadu vychádza zo špecifikácie požiadaviek na systém a nie zo štruktúry výsledného programu, je nezávislá od použitého programovacieho jazyka. Pretože pre odhad používa kritérium počtu služieb komunikujúcich (dostupných) používateľovi, nehodí sa pre aplikácie zamerané na zložité vnútorné spracovávanie údajov. Metóda funkčných bodov bola v roku 1986 dopracovaná o ďalšiu charakteristiku – algoritmy so štandardnou váhou, aby sa odstránil spomenutý nedostatok.

Na čas potrebný na spracovanie požadovaného objemu výrazne vplýva aj produktivita spracovávateľa. Produktivita je definovaná ako množstvo výstupu za jednotku času. Rozdiely v produktivite softvérových inžinierov môžu byť niekoľko násobné. Menej produktívni softvéroví inžinieri môžu však vytvárať spoľahlivejšie softvérové produkty, ktoré sa ľahko udržiavajú a nie je potrebné vykonávať rozsiahle testovanie (ak to nie je požadované zákazníkom), čo v konečnom dôsledku môže znížiť náklady na projekt.

### **Súbežnosť prác**

Koľko ľudí alebo udalostí je zapojených v projekte naraz?

Možnosť súčasnej práce viacerých ľudí na projekte do značnej miery závisí od viacerých faktorov. V prvom rade musí byť urobený vhodný návrh systému, z ktorého vidieť jednotlivé elementy, ktoré vytvárajú systém. Je vhodné vytvoriť si časový plán a naplánovať spracovanie jednotlivých komponentov systému. Pri vytváraní časového plánu je nutné identifikovať všetky činnosti a závislosti medzi týmito činnosťami. Závislosti sú obmedzujúcim faktorom pri stanovovaní súbežnosti činností (napríklad činnosť nemôže byť vykonaná skôr ako bude dokončená iná činnosť).

Počet ľudí pracujúcich na danom probléme je vhodné obmedziť. Bohužiaľ neplatí pravidlo, keď urobenie niečoho trvá jednému človekovi hodinu, bude desiatim ľuďom trvať šesť minút. Nesmieme zabudnúť, že nie sme všetci rovnakí. Pri práci ľudí v tíme si musia ľudia vymieňať informácie medzi sebou. Zvyšovaním počtu ľudí v tíme narastá aj množstvo komunikácie. Toto množstvo rastie nelineárne (platí, že pri  $n$  ľuďoch je možných  $n \times (n-1) / 2$  komunikujúcich dvojíc). Táto komunikácia je „zdržujúci“ faktor, pretože kým človek komunikuje, neprodukuje systém. Od určitého počtu ľudí v tíme dochádza k väčšiemu zvýšeniu komunikácie ako je samotný prínos ďalšieho človeka v tíme. V konečnom dôsledku je tím menej výkonný ako s menším počtom ľudí. Táto hranica je individuálna pre každý projekt a neexistuje spoľahlivý vzorec na určenie jej hranice.

## Časový cyklus

Sú jednotlivé procesy vykonávané dostatočne rýchlo, aby projekt spĺňal časový harmonogram?

Aby sme zistili odpoveď na túto otázku, treba mať vo vývojom systéme vytvorený podrobný časový plán jednotlivých procesov. Určenie časových kvánt pre jednotlivé procesy je činnosť náročná, vyžadujúca skúsenosti ľudí, ktorí tento plán vytvárajú. Jednotlivé body plánu musia byť presne a jednoznačne definované pre ľahkú kontrolu stavu procesov.

Existujú metódy, pomocou ktorých dokážeme vyjadriť dĺžku trvania jednotlivých procesov. Najznámejšie sú PERT diagram a metóda kritickej cesty. Umožňujú zobrazenie jednotlivých činností v čase. Môžeme ľahko určiť časové oneskorenie, ktoré môže viesť ku kritickému oneskoreniu projektu voči časovému harmonogramu. Je vhodné zachytiť počiatočné fázy oneskorovania, pretože je v tejto dobe jednoduchšie a menej nákladné urobiť opravné opatrenia. Keď projekt dosiahne oneskorenie, je už veľmi ťažké dohnať plán. Najčastejšie riešenie oneskoreného projektu je vynechanie fázy testovania. Produkt potom naráža na problémy spoľahlivosti, pretože sa podcenila fáza testovania a neboli nájdené a odstránené aj závažné chyby.

## Operačné atribúty

Sú to atribúty požiadaviek na systém ako dostupnosť a spoľahlivosť. Sú kľúčové pre určenie kvality vývojového systému. Systém musí byť dostupný kedykoľvek, ktorémukoľvek vývojovému pracovníkovi tímu. Musí byť schopný doručiť informácie uložené v systéme na požiadanie. Zlyhávanie týchto základných funkcií vývojového systému môže viesť k neúspechu projektu.

## Dostupnosť

Ako ďaleko je systém dostupný. Dá sa poskytnúť každému, ktorý ho potrebuje použiť?

Táto otázka je v dnešnej dobe vo svojej podstate vyriešená. Rozmachom globálnych informačných sietí je publikovanie projektu nenáročné a prostredníctvom týchto sietí aj dostupné na celom svete. Geografická vzdialenosť ľudí tímu nie je už problémom, komunikačné média Internetu a telekomunikácii zakrývajú vzdialenosť.

## Meranie procesu

Ľubovoľný proces vývojového systému má nejaký spôsob svojho výkonnostného merania (napríklad presnosť, opakovateľnosť), ale predimenzovanie procesu na výkonnosť na úkor časového cyklu je rovnako zlé ako poddimenzovanie. Je dôležité merať charakteristiky všetkých procesov vo vývojom systéme. Namerané charakteristiky vypovedajú ako o kvalite procesu, tak aj o vývojovom systéme. Odchylky týchto charakteristík od štandardných hodnôt skúsenému manažerovi vývojového systému signalizujú, že v procese nie je všetko v poriadku.

Manažér môže podniknúť nápravné opatrenia, aby zamedzil vzniku väčších chýb v systéme.

Často týmto limitujúcim procesom je proces riadenia projektu. Taktiež najdôležitejší atribút daného procesu býva ten, na ktorý sa ani nepomyslelo. Väčšinou to býva práve časový cyklus alebo súbežnosť.

Nepochopením vývojového systému ako systému, môže v projekt zle zosúladiť rýchlosť vývoja systému s daným časom plánom alebo sa nemusí podariť správne určiť optimálnu rýchlosť vývoja systému. Pri veľa projektoch sa robia obe chyby naraz, čo vedie k nestíhaniu časového plánu, poprípade zlyhaniu projektu.

### Literatúra

- [Bieliková00] BIELIKOVÁ, M.: Softvérové inžinierstvo – Princípy a manažment. Slovenská Technická Univerzita v Bratislave, 2000.
- [Brooks95] BROOKS, FREDERICK P.: *The Mythical Man-Month: Essays on Software Engineering*. 2. vyd. Addison-Wesley, 1995. 336 s. ISBN 020135959.
- [Bullock99] BULLOCK, J.: Improving the Development System Model. *Computer*, Október 1999.

## Znovupoužitie – áno či nie? Ak áno, tak ako?

Zdroj: Barry Boehm. *Managing Software Productivity and Reuse*. IEEE Software, Sept. 1999.

Dušan Šucha

**Abstrakt.** *Softvérových manažérov často trápí myšlienka ako znížiť náklady na softvérový projekt bez toho, aby bola znížená kvalita výsledného produktu. Variant, keď sa náklady na vývoj softvéru znížia a jeho kvalita pritom vzrastie, možno považovať za hotový zázrak. Nejedná sa však o žiaden zázrak, ide o znovupoužitie. O tom, čo to znovupoužitie je, ako ho zaviesť do praxe, aké má výhody a nevýhody, pojednáva táto esej.*

Znovupoužitie je založené na veľmi jednoduchej, dobre známej myšlienke. Ak máme k dispozícii už hotovú, spoľahlivú vec, môžeme ju použiť a nemusíme pritom rozmýšľať nad tým, kto, kedy a ako ju vyrobil. Predídeme tak sklamaniu, ktoré zažil čert v rozprávke Čert a Káča, keď vynášiel už dávno objavené koleso a v neposlednom rade si tým ušetríme množstvo námahy, ktorú môžeme sústrediť na výrobu inej *znovupoužiteľnej súčiastky*.

Pri vývoji novej softvérovej aplikácie je tiež možné využiť už hotové, v minulosti vytvorené softvérové súčiastky a z nich „jednoducho“ poskladať nový produkt. Znížia sa tým náklady na vývoj, testovanie, dokumentovanie a údržbu týchto častí softvéru. Podľa Boehma [Boehm99] znovupoužitím je možné ušetriť až 47% nákladov na vývoj nového softvéru. Podľa Grissa [Griss/a] sa pri aplikácii znovupoužitia skrátí čas potrebný na vývoj softvéru na štvrtinu pôvodného času, kvalita takto vyvíjaného softvéru vzrastie päť až šesť krát, pritom sa však náklady na jeho vývoj a údržbu podstatne znížia. Softvérový produkt vytvorený znovupoužitím je schopný oveľa lepšie spolupracovať s okolitými systémami a je preň charakteristická i lepšia súdržnosť.

Efektívne využitie znovupoužitia ako nástroja znižovania nákladov a zvyšovania kvality výsledného produktu si v praxi vyžaduje komplexný

prístup. Nestačí zhromažďovať kúsky kódu do nejakej knižnice a ponúkať ich programátorom na využitie. Každá súčiastka musí byť dôkladne navrhnutá, naprogramovaná a zdokumentovaná. Iba takto kvalitne vyhotovená súčiastka je schopná bezproblémovo fungovať a spolupracovať s ostatnými časťami softvéru.

### **Organizačné zmeny v dôsledku zavedenia znovupoužitia**

Ak sa firma rozhodne zaviesť do svojho softvérového procesu znovupoužitie, musí sa pripraviť na rozsiahle organizačné zmeny. Klasický vývoj softvéru sa musí zmeniť na podporu vývoja a znovupoužitia súčiastok. Ľudia musia byť zaučení na využívanie existujúcich súčiastok a novej technológie. Vyžaduje si to v podstate prestavbu celej firmy, náročnú na čas i financie. Napriek tomu sa znovupoužitie považuje za najlepšiu cestu efektívneho vylepšovania softvérového procesu a znižovania nákladov.

Pri zavádzaní znovupoužitia sa musí prihliadať na základný procesný model znovupoužitia (obr. 1), ktorý pozostáva z viacerých častí:

### **Vytváranie súčiastok**

Ide o zriadenie množiny súčiastok pre znovupoužitie. Táto množina môže obsahovať zdrojové kódy, používateľské rozhrania, architektúry, testy a rôzne prostriedky na vývoj softvéru, ktoré boli vyvinuté, upravené či zakúpené v rámci organizácie.

### **Používanie súčiastok**

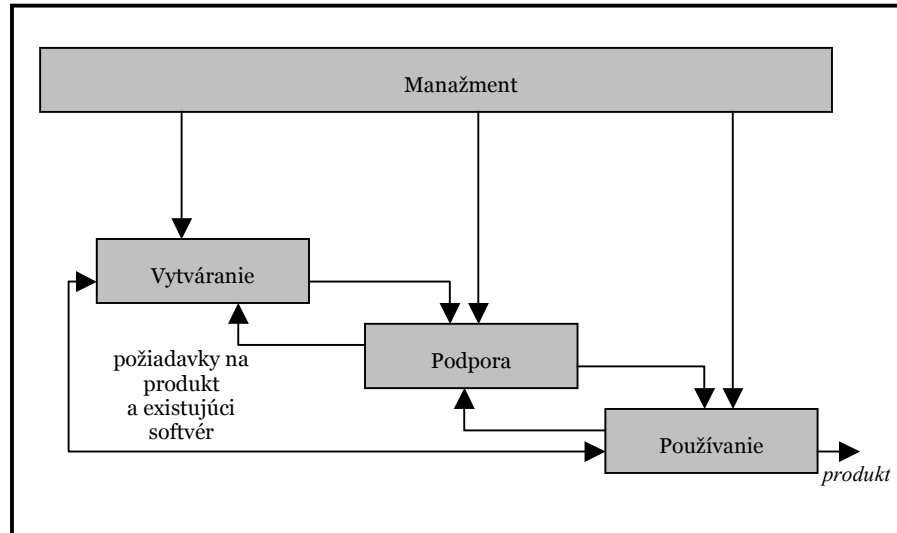
V tomto procese sa využívajú existujúce súčiastky na vývoj nových produktov (aplikácií alebo systémov). Tento proces zahŕňa analýzu požiadaviek na produkt, prehľadávanie množiny znovupoužiteľných súčiastok, ich adaptáciu na potreby produktu a samotný vývoj produktu.

### **Podpora**

Zabezpečuje podporu celého procesu znovupoužitia, manažment a udržiavanie množiny znovupoužiteľných súčiastok. Ide najmä o certifikáciu tejto množiny súčiastok, ich rozdeľovanie do knižníc, spätnú väzbu medzi ostatnými procesmi.

## Manažment

Tento proces zahŕňa najmä plánovanie, inicializovanie, zásobovanie, sledovanie, koordinovanie a zlepšovanie celého procesu znovupoužitia. Medzi ďalšie aktivity patrí plánovanie a vývoj novej množiny súčiastok, riešenie konfliktov v prípade, že potrebná súčiastka nie je k dispozícii, vydávanie pokynov pre prácu so súčiastkami.



**Obr. 1:** Základný model znovupoužitia (Zdroj: [Griss/a]).

V praxi Griss odporúča organizačnú štruktúru zodpovedajúcu uvedenému základnému modelu zloženú zo štyroch druhov tímov:

- Jeden alebo viac tímov, ktoré vytvárajú súčiastky na znovupoužitie.
- Viacero tímov, ktoré využívajú vytvorené softvérové súčiastky na vyhotovenie produktov.
- Jeden riadiaci tím, ktorý zvyčajne riadi samostatný manažér. Ten spolu so svojimi asistentmi dohliada na bezproblémový chod pracovného procesu, zabezpečuje potrebné financie a rieši prípadné konflikty medzi jednotlivými tímami.
- Jeden podporný tím, ktorý má za úlohu údržbu množiny znovupoužiteľných súčiastok, podporu tímov zameraných na znovupoužitie a zabezpečenie kvalitnej spätnej väzby v prípade nejakých konfliktov.

### Aplikácia znovupoužitia v praxi

Z praxe vyplynulo, že najvýhodnejšie je rozdeliť prechod na znovupoužitie do viacerých čiastkových krokov. V prvom kroku je potrebné vyvinúť znovupoužiteľné súčiastky, potom je potrebné vybudovať a nasadiť efektívny manažment, ktorý bude mať na starosti riadenie celého procesu znovupoužitia a nakoniec prispôbiť existujúci softvérový proces novým podmienkam.

Napriek všetkým snahám sa pri aplikovaní tejto techniky môžeme stretnúť s viacerými problémami. Uvediem tie najzávažnejšie:

### **Nerealizovateľné predstavy**

Problém spočíva v mylnej predstave, že stačí vybudovať nejaký sklad súčiastok a programátori začnú automaticky z tohoto skladu čerpať, čiže využívať pri svojej práci vopred vyhotovené súčiastky. Existuje konkrétny príklad, keď firma bezhlavo zhromažďovala veľké množstvo súčiastok, aby ich mohla potom využiť a nakoniec z nich nemala žiaden ošoh. Do hry vstúpilo niekoľko ďalších faktorov (neočakávané riziká, neplánované reakcie na určité stavy) a celá databáza asi 2000 súčiastok bola nanič.

### **Súčiastky versus používateľské rozhranie**

Naše úsilie dosiahnuť efektívne znovupoužitie zlyhá i v prípade, že vytváraný program nemá presne stanovenú základnú architektúru a dobre navrhnuté používateľské rozhranie. Ak bude program postavený z vysokokvalitných modulov, ktoré však budú chaoticky poprepájané a pridáme si k tomu predstavu nevyhovujúceho používateľského rozhrania, nie je ťažké odhadnúť, aký výsledný produkt dostaneme. Zoberme si príklad obyčajného bicykla. Neodvezieme sa dovedy, kým jednotlivé súčiastky starostlivo nespojáme presne podľa návodu.

### **Prílišné zovšeobecňovanie**

Knižničný informačný systém MEDLARS II bol postavený na vysokej úrovni abstrakcie tak, aby podporoval čo najviac knižničných informačných systémov v krajine. Jeho vyradeniu z prevádzky predchádzali dve finančne náročné zlepšenia hardvéru, ale ani po nich systém nebol schopný pracovať podľa predstáv používateľov.

### **Škálovateľnosť**

Kód napísaný v jazykoch štvrtej generácie sa veľmi ťažko rozkladá na jednotlivé samostatné komponenty. Program sám o sebe vystupuje ako samostatný komponent. Aj keď je možné ho ďalej deliť (veď všetko je deliteľné), treba k tomuto deleniu pristupovať naozaj veľmi citlivo. Ináč sa môže stať, že vytvorená aplikácia bude pracovať tak zle ako istý informačný systém dopravného inšpektorátu v New Jersey. Tento systém fungoval tak nespoľahlivo, že viac ako milión áut sa potulovalo v New Jersey bez obnovenej licencie.

### **Technologická zastaranosť**

V 70.-tych a na začiatku 80.-tych rokov získala firma TRW mnoho zákaziek v oblasti digitálneho spracovania vďaka veľkolepej architektúre svojho systému založenej na rozsiahlej množine znovupoužiteľných súčiastok. Napriek tomu bola v polovici 80.-tych rokov táto technológia nahradená vysoko výkonným distribuovaným spracovaním. Z uvedeného vyplýva, že aj kvalitný softvér vytvorený pomocou znovupoužitia je neustále vystavený silnému tlaku, ktorý so sebou prináša rýchly technologický vývoj.

### **Mutácia znovupoužitia**

Jedným často využívaným odvodeným spôsobom znovupoužitia je kopírovanie využiteľných častí zdrojového kódu na viaceré miesta v programe. Ušetrí sa tým čas na vývoj a testovanie, keďže miernymi úpravami vieme prispôbiť tento kód novým požiadavkám. Často sa však nevyhneme obrovským problémom s údržbou takto vyvinutého produktu. Chyby z jednej kópie sa rozšíria na viaceré miesta a ich odstraňovanie je potom veľmi namáhavé.

### **Sociálne a etické problémy**

Pri aplikácii znovupoužitia veľmi často dochádza k problémom vyplývajúcich z nedostatočnej dôvery používateľov softvérových súčiastok k práci vývojárov týchto súčiastok, pretože dostávajú do rúk čierne skrinky, na ktoré sa musia spoľahnúť. Programátori sa sťažujú i na stratu nezávislosti a možnosti vlastnej realizácie, pretože do ich predstáv nemusia vždy zapadnúť ohraničenia, ktoré stanovujú už hotové súčiastky.

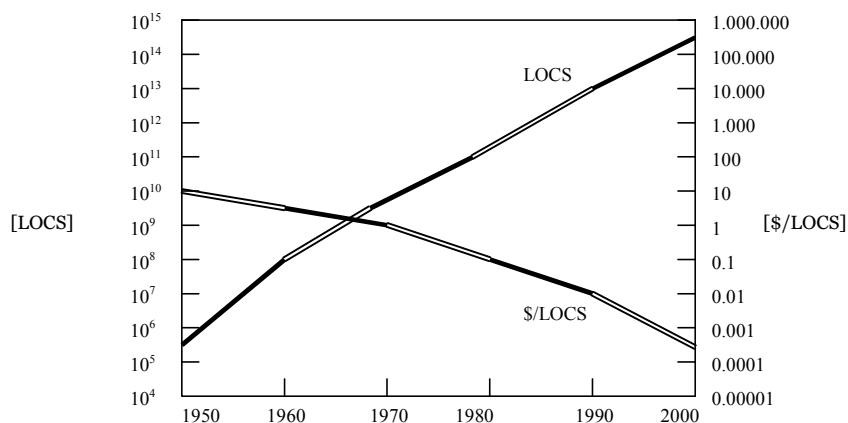
### **Znovupoužitie bez vedomia autora súčiastky**

V tejto časti mi nedá, aby som nespomenul veľmi zaujímavý spôsob znovupoužitia aplikovaný na našej škole, i keď je málo pravdepodobné, že by niekoho trápili sociálne a etické problémy s ním spojené. Ako sa už stalo dobrým zvykom na Katedre informatiky a výpočtovej techniky treba na získanie zápočtu odovzdať jeden či viac programov. Zopár pedagógov však nepovažuje za potrebné meniť zadanie týchto programov niekoľko rokov. A tak šikovnejší študenti (ktorí si to vedia dobre zariadiť) len zoženú hotový program (prehľadajú databázu znovupoužiteľných súčiastok), ktorý presne vyhovuje danej špecifikácii. Ušetria si tým náklady a čas na vývoj a testovanie programu, ale prídu o skúsenosti potrebné na jeho vyhotovenie...

### **Prínosy znovupoužitia**

Okrem výhod uvedených v úvode tejto eseje – 47%-né zníženie nákladovosti na vývoj softvéru, skrátenie času potrebného na vývoj softvéru na štvrtinu, zvýšenie kvality výsledného produktu atď., by som rád uviedol konkrétny príklad z projektov ministerstva obrany USA [Boehm99], ktorý výhody aplikácie znovupoužitia znázorňuje na obr. 2.

Vo svojej analýze sa neopieral o počet riadkov naprogramovaných v určitom programovacom jazyku, pretože určitý počet riadkov naprogramovaných v jazyku štvrtej generácie prináša so sebou oveľa väčšiu produktivitu ako ten istý počet riadkov v asembleri. Kvôli nezávislosti od programovacieho jazyka použil vo svojich štatistikách počet strojových inštrukcií pripadajúcich na daný kód (angl. LOCS). Sledoval počet strojových inštrukcií a náklady na jednu strojovú inštrukciu v rozmedzí rokov 1950 – 2000. Aj keď počet strojových inštrukcií vygenerovaných počas tohto obdobia rástol, náklady na jednu inštrukciu sa znižovali práve vďaka znovupoužitiu.



**Obr. 2:** Nárast počtu strojových inštrukcií a pokles nákladov na jednu strojovú inštrukciu pri zavedení znovupoužitia (Zdroj: [Boehm99]).

Griss po 15 ročných skúsenostiach so znovupoužitím vyjadril rozčarovanie nad tým, že príliš málo ľudí rozumie tomu, čo to znovupoužitie je a ešte menej ľudí pri svojej práci znovupoužitie systematicky využíva [Griss/c]. Existuje množstvo dobrých metód a pravidiel ako dosiahnuť efektívne znovupoužitie (OO technológie, softvérové vzory, komponenty, generátory, schémy rozdeľovania do knižníc) a budem oponovať pánu Grissovi, pretože sa u nás už celkom sľubne začínajú používať i v praxi. Nesmieme si však pri tom znovupoužitie zamieňať s trápny kopírovaním kódu a jeho prispôbovaním podľa potrieb alebo s obvyčajnými funkciami, pri ktorých sa tiež využíva raz napísaný kód na viacerých miestach. I keď opísané techniky tiež javia akési známky znovupoužitia, no ani tak ich zďaleka nemožno nazvať systematickým znovupoužitím. Pri skutočnej aplikácii znovupoužitia ide skôr o vytváranie samostatných objektov, komponentov a používanie generátorov na generovanie kostry kódu podľa nastavených parametrov.

Tých ostatných, teda tých, ktorí sa s myšlienkou znovupoužitia ešte nestotožnili treba presvedčiť, že znovupoužitie nie je len nafúknutá bublina, rozpitvaná teória o niečom, čo je už všetkým známe alebo množstvo práce s neistým efektom. Griss odporúča organizovať semináre, kurzy a prednášky na túto tému, aplikovať získané informácie v praxi a deliť sa o skúsenosti s inými, vydávať zrozumiteľné príručky a pravidlá, vyvinúť stratégiu zavádzania znovupoužitia do praxe krok po kroku, napísať knihu „Všetko čo chceme vedieť o znovupoužití“. 47%-né zníženie nákladov je síce sľubné, nikde som však nenašiel referencie na, s akými nákladmi musíme počítať pri zavádzaní znovupoužitia. I keď Boehm uvádza na konci svojho článku [Boehm99], že 47% je len začiatok, bez dôkladnej analýzy aplikácie znovupoužitia zo všetkých strán bude ťažké presvedčiť konzervatívnych programátorov a ich manažérov na zavedenie tejto techniky do ich praxe.

### **Literatúra**

- [Boehm99] BOEHM, BARRY: *Managing Software Productivity and Reuse*. IEEE Software, September 1999.
- [Griss/a] GRISS, MARTIN L.: *Domain Engineering In The Reuse-Driven Software Engineering*.  
<http://www.hpl.hp.com/reuse/papers/fusion2.htm>.
- [Griss/b] GRISS, MARTIN L.: *Reuse 2001-2004. What next, now that we have solved all reuse problems?*  
<http://www.hpl.hp.com/reuse/papers/fusion1.htm>.
- [Griss/c] GRISS, MARTIN L.: *Systematic software reuse: Architecture, Process and Organization are Crucial*.  
<http://www.hpl.hp.com/reuse/papers/wisr99-griss.htm>.



## Pripravte sa zlyhať

Michal Šrámka

**Abstrakt.** *S nekončiacim vývojom a prevratmi v oblasti technológie sa neustále mení aj trhové prostredie. Nové technológie spôsobujú úspešným a často aj veľkým firmám neprekonateľné prekážky. Existuje niekoľko liekov na manažovanie technologických zmien. Ale ktorý je ten pravý?*

V článku Manažovanie rýchlosťou svetla [Carrier99] autor uvádza, že dnes sa už nestačí zameriavať na potreby zákazníkov – tie prinášajú úspech teraz, ale zatiaľ objavenie inovácií, zdokonalení a zmien, na ktoré zákazníci čakajú. Táto skutočnosť dáva priestor novým firmám s ich produktmi, ktoré sa časom zlepšia a stanú sa reálnou konkurenčnou hrozbou na trhu.

### **Prelomové a zachovávajúce technológie**

Zoči voči neúprosnému tempu technologického pokroku firmy hľadajú návod, ako strategicky zavádzať a využívať nové technológie. Clayton Christensen, vo svojej knihe Dilema inovátora [Christensen99] uvádza návod, ako sa s takýmito problémami vysporiadať tým, že rozdeľuje technológie na prelomové (*disruptive*) a zachovávajúce (*sustaining*).

Zachovávajúce technológie sú také technológie, ktoré rozširujú existujúci produkt alebo službu a tým robia ich odbyt efektívnejším. Zachovávajúce inovácie spôsobujú, že produkt alebo služba je pre zákazníkov aj naďalej atraktívna, pretože produkty a služby sú sofistikovanejšie, vyrábajú a dodávajú sa rýchlejšie, sú lacnejšie a sú schopné poskytovať väčšie možnosti ako predchádzajúce verzie produktov – avšak len pre zákazníkov existujúceho trhu. Zachovávajúce technológie tento trh pomaly rozširujú, ale na nové trhy preniknú len zriedkavo.

Naproti tomu sú prelomové technológie také, že výrazne zmenia alebo úplne nahradia zaužívaný spôsob vykonávania vecí. Prelomové technológie – produkty alebo služby začínajú v malom, často pomaly a sú pre bežných používateľov príliš drahé. Príťažlivé sú len pre malú časť

používateľov a teda oslovujú len špeciálne trhy. Z marketingového pohľadu na existujúce trhy sú teda tieto technológie spočiatku bezvýznamné, ale zdokonaľujú sa rýchlejšie ako stúpajú požiadavky trhu a skôr alebo neskôr zaplavia tieto existujúce trhy a nahradia zachovávané technológie. Prelomové technológie sú často tie, ktoré spôsobujú zmeny aj v iných odvetviach – napr. Internet ako prelomová technológia podnietila vznik sieťových počítačov ako aj rôznych doplnkových zariadení (napr. *web-TV*).

Dobрым príkladom na priblíženie problematiky prelomových a zachovávajúcich technológií môžu byť sálové (*mainframe*) počítače. Sálové počítače dlhé roky prežívali zachovávanie technológií, stále vylepšujú pomer ceny a výkonu. Minipočítače boli naproti tomu prelomovou technológiou. Spočiatku nezvládali úlohy sálových počítačov – neboli teda pre ne konkurenciou a aj preto oslovili a presadili sa na úplne nových trhoch. Minipočítače nikdy neprekonali sálové počítače čo do kapacity, ale dokázali vykonať množstvo úloh, ktoré boli požadované od sálových počítačov a tým sa dokázali presadiť aj na týchto trhoch a dokázali ich z väčšej časti ovládnuť.

Spoločnosti často prehliadajú príležitosti ponúkané prelomovými technológiami, pretože príliš počúvajú svojich zákazníkov a vyrábajú a dodávajú to, čo zákazníci práve v tomto čase požadujú. Preto sa môže zdať, že počúvanie a uspokojovanie potrieb zákazníkov je nesprávny postup v riadení firiem.

Zákazníci pod "tlakom" starostlivosti o svoje potreby často o novinkách a ich využití ešte ani netušia a tak, keď sa nové technológie na trhu objavia, tak ich s počiatku odmietajú – vyčkávajú. Táto reakcia môže priviesť zákaznícky orientované firmy k strate ich hlavných výhod.

Clayton Christensen uvádza príklad, kde za celú dobu existencie trhu s nosičmi údajov (diskety, disky, pásky, mechaniky, ...) prišli vedúce firmy iba 2-krát na trh s produktmi ďalšej generácie. V ostatných šiestich prípadoch prišli s prelomovými technológiami iné spoločnosti a tým nahradili lídra na trhu.

### **Držať krok**

Vo svete biznisu sú známe prípady (napr. firma Intel s procesorom i8088), kedy pomohli prelomové technológie presadiť sa firmám na vysoko konkurenčných trhoch. Najprv sa produkty určené pre iné použitie presadili na malých trhoch, neskôr vytlačili z trhov špičkových konkurentov.

Zabehnuté firmy majú skôr sklon ignorovať prelomové technológie, pretože sú nové, neodskúšané a spočiatku ponúkajú nižšie zisky ako existujúce produkty. Preto dnešné vysoko konkurenčné prostredie vyžaduje iný manažérsky prístup k týmto neustálym zmenám. Viaceré kapacity v oblasti manažmentu odporúčajú nasledovné tri techniky ako úspešne konkurovať na rýchlo sa meniacich trhoch:

- rýchlo sa presúvať k novým produktom a novým trhom,
- byť flexibilným v stratégii a implementácii, a
- preskúmať všetky možnosti investovania.

Rýchle reakcie na nepredvídané príležitosti a konkurenčné hrozby vyžadujú organizačnú mobilitu. Napríklad firma Netscape, popri konkurenčnom boji na trhu prehliadačov WWW s firmou Microsoft, sa preorientovala na firmu, ktorá ponúkala riešenia pre intranet, neskôr pre extranet, zmenila sa na portál pre e-commerce a nakoniec bola rozdelená a odkúpená firmami Sun a AOL.

Iné kapacity v oblasti manažmentu, odchovanci Harvardskej univerzity, ponúkajú nasledovných desať stratégií ako sa stať vedúcim podnikom na trhu:

- prednosti sú dočasné,
- stratégia je rozmanitá, vyvíjajúca sa a komplikovaná,
- cieľom je znovuvynájdienie,
- žiť v prítomnosti,
- poučiť sa z minulosti,
- siahť do budúcnosti,
- zmeniť rytmus,
- zlepšovať stratégiu,
- rozšíriť stratégiu z podnikateľských (biznis) zámerov, a
- prepojiť zámery a trhy.

Sám Christensen vo svojej knihe navrhuje ako sa vysporiadať s prelomovými technológiami – ako a kedy vstúpiť na novovznikajúci trh. V podstate existujú dve protichodné možnosti ako na vstúpiť na trh: (1) skočiť na trh rýchlo alebo (2) počkať, kým priekopníci vyčistia cestu a vyriešia hlavné riziká.

Firmy, ktoré preferujú zachovávané technológie si môžu dovoliť počkať často aj niekoľko rokov, napriek tomu sa uchytia na trhu. Naproti tomu s prelomovými technológiami nemožno čakať – aj najmenšie oneskorenie môže viesť k veľkým nevýhodám. Napríklad firmy, ktoré vstúpili na trh tých nosičov údajov, ktoré boli vytvorené prelomovými technológiami počas prvých dvoch rokov mali šesť-krát väčšiu šancu uspieť ako firmy, ktoré na tento trh vstúpili neskôr.

Veľkosť firmy pritom nehrá veľkú rolu. Malé začínajúce firmy a veľké spoločnosti sa presadzujú približne rovnako. Je všeobecne známe, že skorý vstup na nový trh je výhodný.

Avšak menšie firmy sú zvyčajne kreatívnejšie, pružnejšie a zdá sa, že aj lepšie presadzujú nové prelomové technológie. Už spomínaná firma Netscape začínala s niekoľko desiatkami ľudí a produkovala výborný softvér, ktorý sa jej darilo na trhu aj presadzovať. Keď však začala byť príliš úspešná, tak podľa jedného z hlavných architektov prehliadača WWW Netscape Navigатора – Jamie Zawinskeho, stratila svoju flexibilitu. Vo väčších tímoch sa prevratné nápady strácali a potláčali, firma začala byť príliš orientovaná na zákazníkov – preferovala zachovávané technológie. Z týchto dôvodov firmu opustili aj viacerí zamestnanci, medzi inými aj Jamie Zawinski. Čiže je nutné nájsť kompromis medzi veľkosťou firmy (a tímov) ako aj medzi stratégiou vývoja nových prelomových technológií alebo preferovania zachovávaných technológií. Nesprávny kompromis má za následok možnú stratu kľúčových osobností a možnú stratu flexibility tak, ako sa to stalo práve v prípade firmy Netscape.

### **Neinvestovať do technológií kvôli technológii**

"Naše investície do technológií budú riadiť požiadavky podnikania, nebudeme investovať do technológií v záujme technológie." je jedna z viet označovaných v článku [Lewis98] ako veľmi originálna myšlienka.

A nie je to práve múdre vyhlásenie. Nie pre to, že by to vždy nebola pravda. Často neefektívnosť alebo zmeny v stratégii vedú biznis k zisteniu, že je potrebné zaobstarať novú technológiu. Pri veľmi málo udalostiach táto potreba ovplyvní aj rapidný vývoj u dodávateľov, skôr sa jednoducho nakúpia existujúce technológie. Ale výnimka potvrdzuje pravidlo a môže sa to stať.

Čo tento pohľad na vec ignoruje je však to, že nové technológie vytvárajú príležitosti pre zefektívnenie procesov, pre zlepšenie vzťahov so zákazníkmi alebo pre definovanie úplne nových trhov, v ktorých môže táto spoločnosť podnikáť. Toto sú príležitosti, ktoré riadiaci pracovníci nie vždy predvídajú. Napríklad, keby tomu tak bolo a spoločnosti by boli začiatkom sedemdesiatych rokov počúvali hobby-skupiny, ktoré hovorili "Viete, keby niekto z vás vytvoril elektronický tabuľkový procesor, my by sme si kúpili milióny osobných počítačov len preto, aby sme ho na nich spúšťali.", tak osobné počítače by sa rozšírili oveľa skôr.

Iným príkladom môže byť WWW – Tim Berners-Lee prispôbil SGML (*Standard Generalized Markup Language*) pre potreby medzinárodnej komunity fyzikov. Tým vynašiel WWW (*World Wide Web*) a aj nový multimiliónový, dnes už multimiliardový, biznis. Výkonní manažéri nepodnietili a neriadili vznik WWW. Ešte aj dnes niekoľko z nich považuje WWW za ešte úplne nevyčerpanú studňu možností.

Investovanie do technológií, ktoré podnietili výkonní manažéri alebo vedenie firiem všeobecne, môže pri správnych smeroch investícií viesť

k pokroku. Je to však iba časť celého problému investovania do technológií – tieto investície sú iba napodobovaním vedúcich firiem na trhu. Inými slovami, výkonný manažment firmy vníma tieto investície len ako odpoveď na otázky prežitia a udržania si pozície na trhu, nie ako investície na strategický rozvoj technológií.

Biznis nepoháňa technické inovácie, iba za ne platí. Manažéri si obyčajne nevedia predstaviť hodnotu niečoho, čo tu ešte nikdy nebolo a s čím sa ešte nikdy nestretli. Aj preto sú riadiaci pracovníci technologických oddelení vo firmách považovaní za neflexibilných – čakajú na zvyšok spoločnosti, kým nepožaduje novú technológiu.

Áno – technické inovácie musia vytvárať podnikateľské hodnoty. A nie, nie všetky technické inovácie budú vytvárať hodnoty pre podnikanie.

**M**anažéri firiem by sa mali sústrediť na zlepšenie strategického plánovania a na udržanie kreatívneho procesu myslenia.

Odborníci opäť odporúčajú: podnikateľská vízia – rozmanité nenáročné sondovanie v nových trhoch a neustále pozorovanie zmien súčasných a možných budúcich trhov. Takéto pozorovanie predpokladá časté pohľady do budúcnosti – čo môže budúcnosť priniesť, radšej ako neustále pracovať na prieskumoch rýchlo sa meniacich trhov.

Ak sa jedná o vedúcu firmu na trhu, je nutné mať na pamäti viac ako len aktuálne potreby zákazníkov. Udržovaním vývoja na zákazníkmi vyjadrených potrebách sa dá dosiahnuť najväčší profit, avšak inovácie a vývoj nových technológií bude potláčaný. Pretože nikto nevie, ako nové technológie použijú, firmy musia pozorovať zákazníkov.

Klasické analytické procesy a procesy rozhodovania vyžadujú určité kvantá informácií, ale pre nové a prelomové technológie takéto informácie ešte neexistujú. Firmy by sa okrem plánovania výsledkov mali plánovať aj zlyhanie. Nie každá firma musí zlyhať – ale ak zlyhá, vedomosti, ktoré sa získajú týmto zlyhaním určujú výhody do budúcnosti (napr. pre získavanie nových trhov). Takéto skúsenosti tiež môžu napomôcť rozptýliť obavy pri investovaní do nových oblastí, technológií alebo produktov.

Riadenie zmien a stratégie pri prelomových technológiách vyžaduje aktívnu účasť vývojárov, investorov a manažmentu. Vývojári vidia príležitosti a problémy pri hľadaní nových trhov pre nové technológie, identifikujú a určujú schopnosť firmy realizovať sa na týchto trhoch. Investori rizikového kapitálu, ktorí sponzorujú nové technológie majú úžitok z pochopenia ako treba prelomové technológie manažovať a presadzovať na trhoch. Nakoniec, implementátor prelomových technológií musí ustavične sledovať výhody prelomovej technológie, aby nedošlo k nahradeniu novej technológie starou a aby nová technológia dosiahla svoj úplný potenciál.

Vrcholový manažment by si mal uvedomiť, že prelomové technológie môžu firme priniesť úspech, ale aj neúspech a tým teda aj stratu trhu a dobrého mena firmy.

### Literatúra

- [Carrier99] CARRIER, L.: *Managing at Light Speed*. IEEE Computer, Júl 1999, s. 107-109.
- [Christensen97] CHRISTENSEN, C. M.: *The Innovator's Dilemma*. Harvard Business School Press, 1997.  
<http://www.disruptivetechnologies.com/> (part 1).
- [Lewis98] LEWIS, R.: *If you wait for business needs to drive technology buys, you will fall behind*. IS Survival Guide, 12. január 1998.  
[http://www.techinformer.com/crd\\_technologies\\_7031.html](http://www.techinformer.com/crd_technologies_7031.html).

## Testovanie požiadaviek a človek v úlohe testera.

Zdroj: J.Bach. Risk and Requirements – based testing [Bach99].

Marián Teplický

**Abstrakt.** Táto esej sa zaoberá dvami podstatnými problémami pri testovaní. Výberom vhodného testera, jeho vlastností a daností a dĺžkou dobou testovania a vplyvom jej skracovania na kvalitu produktu. Ďalej sa eseji zamiešľa nad štyrmi základnými zásadami testovania produktu v závislosti od požiadaviek a pokúša sa ich modifikovať. Esej sa snaží vyvrátiť opraviť zažitú predstavu, že testovanie je len nejaký druh automatizovaného procesu.

Testovanie, ako jedna s dôležitých činností pri vývoji softvéru, má svoje špecifiká a riziká. Keďže je to činnosť, ktorou do značnej miery môžeme ovplyvniť výslednú kvalitu produktu, nemali by sme ju v žiadnom prípade podceňovať. Pri zamyslení sa nad touto činnosťou, môžeme identifikovať aj nasledovné otázky. Kto má testovanie uskutočniť? Programátor, alebo zvlášť tým poverený človek? Aké má mať znalosti a zručnosti? Ako dlho má trvať testovanie? Ako testovať v závislosti od požiadaviek a rizík?

### Kto má testovanie uskutočňovať?

Samozrejme, že základné testovanie jednotlivých modulov a funkčnosti kódu vykonáva priamo programátor. Popri tom ako implementuje jednotlivé časti, testuje ich funkčnosť. Otázkou je, kto by mal uskutočňovať záverečné testovanie. Zrejme nie je vhodné, aby to bol programátor, ktorý daný produkt implementoval a to z nasledujúcich príčin. Programátor vie ako daný program presne funguje, je to preň vždy biela skrinka a tým je do istej miery určený jeho postup pri testovaní. Na druhej strane tester, ktorý nevie ako program presne funguje, nie je zaťažovaný týmito vedomosťami a preto má iný prístup k testovaniu činnosti programu. Ďalšou odlišnosťou môže byť, že programátor pochopil istým spôsobom špecifikáciu, ktorú implementoval. Musel

naštudovať problém do istej hĺbky, aby bol schopný realizovať špecifikáciu. Otázkou je, do akej hĺbky sa dostal a či mu neunikli nejaké skryté súvislosti, ktoré nemuseli byť viditeľné na prvý pohľad. Na druhej strane, ak je testerom človek, ktorý má s danou problematikou už isté skúsenosti, je schopný takéto záludnosti včas odhaliť.

Pekným príkladom je nasledujúca situácia. Pred časom som pracoval v istej firme, ktorá vyrába ovládacie zariadenia na tlačiarenské stroje. Ako pracovník vývoja som sa oboznámil s činnosťou tlačiarenského stroja a začali sme vyvíjať nový produkt. Program, ktorý točil motorčekmi ovládajúcimi prítlačné nože v požadovanom intervale. Spravil som program, ktorý bol funkčný a odovzdal som ho. Potom za mnou prišiel tester, že tento program by zničil celý stroj, lebo dva nože vedľa seba nemôžu mať posun väčší ako nejaké  $x$ . Nebolo to v špecifikácii pretože to bolo považované za samozrejmosť. Na jednej strane je to chyba špecifikácie, ale tá nie je skoro nikdy úplná ani jednoznačná, ale hlbšie znalosti testera o danej problematike zachránili stroj od prípadného zničenia. Takže, kto by mal testovať produkt? Mal by to byť tester nezávislý od programátora a ak je to možné, mal by mať prehľad o oblasti činnosti, pre ktorú sa daný produkt vyvíja. Tester by mal byť schopný dostatočne pochopiť oblasť, v ktorej bude daný produkt aplikovať.

Ako už s predchádzajúceho príkladu vyplýva, je rovnako dôležitá spolupráca medzi testerom a programátorom. Tester do istej miery môže ovplyvniť celkový vzhľad a činnosť produktu, hlavne ak má informácie čo sa v danej oblasti požaduje a čo zákazník očakáva. Najmarkantnejšia je táto schopnosť ovplyvniť produkt u produktov, kde je veľká konkurencia a kde o kúpe alebo nekúpe rozhoduje individuálny vkus používateľa. Typickým takýmto produktom sú počítačové hry. Pri ich testovaní často dochádza k prehodnoteniu jednotlivých postupov riešenia za účelom zvýšenia hrateľnosti.

Tieto a iné vlastnosti požadované od testera jasne ukazujú, že testovanie nie je len rutinná činnosť, ale požaduje sa pri ňom aj inteligencia, iniciatíva je to skrátka tvorivá činnosť. Ako poznamenal James Bach „Je potrebné opraviť zažitú predstavu, že testovanie je len nejaký druh automatizovaného procesu [Bach99].“

### **Ako dlho má trvať testovanie?**

V praxi sa stretávame s problémom, ktorý má podstatu v tom, že ako to správne vistihol pán F.P. Brooks: „Všetci programátori sú optimisti“ [Brooks95]. Na základe prílišného optimizmu a taktiež nepostačujúcich, alebo lepšie povedané nedostatočne presných nástrojov na odhad času potrebného na vývoj, sa často vypracuje plán, ktorý sa nedá splniť. Aj keď existujú isté postupy riešiace túto situáciu (uvedené napríklad v [Brooks95]), predsa len sa to väčšinou končí tak, že keď nastane čas odovzdania zákazníkovi jednoducho sa zoberie to, čo je hotové a odovzdá sa produkt dostatočne neotestovaný. Potom sme svedkami vydávania opravných doplnkov tesne po uvedení do predaja v prípade generických

produktov alebo oprave väčšieho počtu chýb počas používania programu u zákazníka. V oboch prípadoch prichádza k strate alebo aspoň narušeniu dobrého mena firmy.

Opäť jeden príklad s praxe. Firma, ktorá vytvorila a predáva informačný systém pre mestá bola požiadaná, aby doplnila systém o komunikáciu a výmenu dát s iným systémom. Keď prišiel čas odovzdať produkt, nebolo dokončené testovanie aj keď program bol čiastočne otestovaný a fungoval dobre. Po dodaní zákazníkovi však nastala istá vopred neotestovaná situácia a program nielenže spadol, ale navyše spravil chybu, ktorá znehodnotila databázu. Na základe zlých skúseností výrobcu toho druhého programu neodporučil svojim zákazníkom kupovať program tejto firmy a keďže mal dominantné postavenie na trhu, vznikli pre túto firmu oveľa väčšie straty ako v prípade oneskoreného dodania produktu a prípadnej penalizácie.

Aj s daného príkladu vyplýva, že testovaniu treba venovať dostatočný čas. Ten je samozrejme pre rôzne typy aplikácií v závislosti od miery škody spôsobenej pri krachu aplikácie rôzny. Iný bude pri programe pre atómovú elektrárňu a iný pre program pre tlačiarňu, ale všeobecne by sa nemal skracovať s dôvodu časového sklzu v projekte.

### **Ako testovať v závislosti od požiadaviek a rizík?**

Definujeme si najprv čo sú a ako vznikajú požiadavky. Požiadavky v tomto ponímaní nie sú špecifikácie od zákazníka, ale je to skupina myšlienok a kritérií, ktoré definujú kvalitu produktu. Táto skupina vzniká zlúčením požiadaviek zákazníka, sú to odpovede na otázku „Čo chceme vytvoriť?“ a reálnymi možnosťami našej firmy, čo sú zase odpovede na otázku „Čo môžeme vytvoriť“. Samozrejme už tu môžeme identifikovať prvé náznaky problémov, pretože odpovede na tieto otázky si môžu protirečiť. Vcelku by sme sa mali snažiť, aby obmedzenia firmy zásadne nedefinovali požiadavky zákazníka.

Testovanie môžeme definovať ako proces vývoja odhadu kvality produktu [Bach99]. Ináč povedané, je to nejaký proces, pomocou ktorého postupne zisťujeme kvalitu produktu v závislosti od objavených chýb. Možno by bolo vhodné doplniť túto definíciu o skutočnosť, že pri odhadovaní kvality nastáva aj oprava chýb, čím sa zase mení kvalita. Treba poznamenať, že podľa Dijkstra: „Testovanie nemôže preukázať, že v programe nie sú chyby. Môže iba ukázať, že tam sú.“ [Bieliková00] Z toho by vyplývalo, že podľa predošlej definície nemôžeme dospieť k poznaniu, že produkt je 100% bez chýb, ale len že je bez chýb len do tej miery, ako sme ho otestovali.

Štandardne sú definované štyri zásady testovania produktu v závislosti od požiadaviek(1).

- Bez stanovených požiadaviek nie je možné žiadne testovanie.
- Programový produkt musí uspokojiť naš stanovené požiadavky.

- Všetky testy by mali byť odvodené od jednej alebo viacerých stanovených požiadaviek, a naopak.
- Stanovené požiadavky musia zodpovedať testovaným podmienkam.

Podme sa teraz venovať bližšie jednotlivým zásadám a skúsme sa zamyslieť nad tým či skutočne vždy platia.

### **Bez stanovených požiadaviek nie je možné žiadne testovanie**

Samozrejme, ak je táto požiadavka nejaká základná a podstatná, napríklad že program môžeme ovládať myšou, je úlohou testera overiť, či program spĺňa danú požiadavku.

Vtedy je toto tvrdenie pravdivé. Ale treba si uvedomiť, že množina požiadaviek tak či tak nie je nikdy úplná ani jednoznačná. Tu záleží od schopností testera vystihnúť zdroje možných konfliktov. Treba skúmať význam a dôsledok požiadaviek a v spolupráci s programátorom či analytikom prispieť k zlepšeniu kvality produktu. Dobrý tester by mal nájsť aj rozdiely v konštatovaných požiadavkách a skutočnom programe, inými slovami identifikovať chyby či nepochopenie požiadavky. Tu sa opäť ukazuje že testovanie nie je len automatizovaná práca, ale tvorivý proces.

### **Programový produkt musí uspokojiť naň stanovené požiadavky.**

Toto tvrdenie je pravdivé, ak máme veľmi presnú a jednoznačnú skupinu požiadaviek. Ako som spomenul, už pri vytváraní skupiny požiadaviek, sa môže stať, že požiadavky si budú protirečiť. Vtedy je úlohou testera identifikovať, ktorá požiadavka je pre daný produkt prioritná. Nemožno zobrať skupinu požiadaviek a pridelovať im hodnoty či boli splnené alebo nie a spraviť s toho nejakú sumu. To by nám o kvalite veľa nepovedalo. Ak sú protirečivé požiadavky treba identifikovať priority a splniť hlavne tie s maximálnou prioritou. Je však pravdou, že ak takáto situácia nastane, znižuje to celkovú kvalitu produktu, pretože sú tu požiadavky, ktoré neboli splnené.

### **Všetky testy by mali byť odvodené od jednej alebo viacerých stanovených požiadaviek, a naopak.**

Ak uvažujeme o testovaní ako o overovaní požiadaviek, tak je táto požiadavka opodstatnená. Dôležité tu je, aby každá požiadavka mala pridružený aspoň 1 test. Pod pojmom 1 test môžeme rozumieť aj skupinu testov, ktoré postupne overia danú požiadavku. Duplicita v testoch nám nielen nevaďí, ale je vítaná. Bolo by dobre pokúsiť sa navrhnúť test tak aby overoval aj viac ako 1 požiadavku a to z toho dôvodu že, systém pri svojej práci bude spravidla uskutočňovať naraz viac ako jednu požiadavku a mala by sa otestovať aj vzájomná súbežnosť vykonávania požiadaviek.

### Stanovené požiadavky musia zodpovedať testovaným podmienkam

Je dôležité, aby sa požiadavka dala nejako kvantifikovať. Aby sa dala nejako odmerať, má nejakú hodnotu. Zväčša sú požiadavky definované ako niečo nápomocné, čo má viesť k dokonalej spoľahlivosti. Nie vždy sa musia brať doslovne ako nám ukáže nasledujúci príklad [Bach99]. Firma dostala za úlohu vyvinúť aplikáciu s dobrou odozvou na používateľský vstup 300 milisekúnd. Zrazu bolo treba kúpiť drahý citlivý prístroj na meranie času v milisekundách. Tak si zistili, že človek je schopný merať čas s odchýlkou 50 milisekúnd. Po preverení u zadávateľa, či je takáto presnosť dostačujúca zistili, že oni v skutočnosti nechceli presne 300 milisekundovú odozvu, ale len aby ich nový systém nebol tak pomalý ako ten starý.

Ak si zapracujeme tieto jednotlivé úvahy do prvotných štyroch zásad mohli by modifikované zásady vyzeráť asi takto:

- Zdrojom pre požiadavky je dokument o požiadavkách, ale tiež sa musíme snažiť porozumieť čoho sa problémy týkajú a tak definovať ďalšie požiadavky.
- V prípade výskytu konfliktných požiadaviek, treba určiť prioritu a výsledný produkt môže obsahovať nesplnené požiadavky, ale v minimálnej forme. Testovaním u zákazníka môžeme zistiť dôležité informácie o konfliktných požiadavkách.
- Pri rizikových situáciách sa snažíme aby aspoň jeden test bol pridružený ku každej požiadavke. Ak sa dá, tak sa jedným testom pokúsiť splniť viac požiadaviek.
- Konkrétne požiadavky definovať v takých podmienkach, ktoré zodpovedajú podstate toho, čo chceme testovaním dosiahnuť. Definovať ich s ohľadom na úžitok, riziko a vyznačiť ich mieru dôležitosti. Pri nejasnosti kontaktovať zákazníka.

Je jasné že tieto pravidlá sú vlastne len odporúčaním. Najpodstatnejší element pri celom testovaní je tester. Od jeho skúseností a schopností závisí úspešnosť jednak testovania, miera komunikácie s programátorom a používateľom a ostatné faktory ovplyvňujúce testovanie. Ak riziko alebo komplexnosť projektu presahujú schopnosti testera, mal by to ohlásiť a radšej nechať dokončenie na niekoho iného. Softvérové firmy by si mali plne uvedomiť potrebu kvalitného testovania a jeho vplyvu na kvalitu výrobku a tým aj povest' firmy a preto by vo vlastnom záujme mali do testovania investovať peniaze a hlavne čas.

### Literatúra

- [Bach99] BACH, J.: *Risk and Requirements – based testing*. Computer, Software Realities, Jún 1999, s. 113-116.

- [Bieliková00] BIELIKOVÁ, M.: *Softvérové inžinierstvo – Princípy a menežment*. Bratislava: STU, 2000. ISBN 80-227-1322-8.
- [Brooks95] BROOKS, FREDERICK P.: *The Mythical Man-Month: Essays on Software Engineering*. 2. vyd. Addison-Wesley, 1995. 336 s. ISBN 020135959.

## Čo robiť, keď zákazník nevie presne, čo chce?

Vladimír Trgo

**Abstrakt.** Esej sa zaoberá problémami, ktoré vznikajú pri špecifikácii požiadaviek na softvérový systém. Poukazuje na náročnosť úlohy analytika a nutnosť komunikácie so zákazníkom. Zdôrazňuje význam vytvárania prototypov – najmä pri zákazníkoch, ktorí nevedia, čo chcú. Zaoberá sa aj príčinami vzniku zmien v požiadavkách na softvérový systém. Základný zdroj, z ktorého som vychádzal pri písaní eseje, je článok jedného z klasikov softvérového inžinierstva pána Boehma [Boehmoo].

Softvéroví inžinieri sa snažia zlepšiť proces špecifikácie požiadaviek. Uvedomujú si, že aj nepatrná zmena požiadaviek na systém, ktorý vyvíjajú, môže vážne ovplyvniť náklady a časový plán projektu. Vývoj softvérového systému by sa nemal podobať na Brownov náhodný pohyb, ktorý závisí od momentálnej nálady zákazníka a softvérového inžiniera. Mal by sa riadiť istými princípmi.

Boehm v úvode [Boehmoo] opisuje staré dobré časy, kedy sa požiadavky definovali na začiatku projektu a počas jeho riešenia sa už nemenili. Softvérové tímy strávili nespočítateľné množstvo hodín overovaním špecifikácie. Snažili sa vytvoriť *perfektnú a detailnú špecifikáciu*, v ktorej presne uvádzali, čo zákazník požaduje. Až po úplnej špecifikácii požiadaviek nasledovali fázy návrhu a implementácie. Takýto spôsob vývoja softvéru označujeme ako vodopádový model. Vodopádový model však môžeme použiť len pri tvorbe jednoduchších systémov s dobre definovanými požiadavkami.

Špirálový model, ktorý navrhol Boehm, sa ľahšie vyrovnáva so zmenou používateľských požiadaviek. V každom cykle objavujú tvorcovia systému viacej a viacej detailov o funkciách systému a jeho architektúre. Vytváranie prototypov slúži na preskúmanie alternatív riešenia.

### Náročná úloha analytika

Konečne máme odvahu pripustiť, že na začiatku projektu často nevieme úplne presne, čo by mal systém robiť. Bolo by naivné si myslieť, že

zákazník od začiatku detailne vie, čo vlastne chce. Preto je intenzívny styk tvorcov systému so zákazníkom veľmi dôležitý [Reifer00].

Analytik musí preniknúť do problémovej oblasti, ktorej sa vyvíjajú softvérový systém týka. Jeho hlavnou úlohou je získať požiadavky od zákazníka. Analytik by mal získať jasnú predstavu definície požiadavky. Zapísať si nielen ako znie požiadavka, ale aj kto ju vyslovil a čo ho k tomu viedlo. Zákazník by mal vysvetliť, prečo je to tak.

Analytik by mal identifikovať, čo vlastne zákazník chce, aké sú jeho očakávania týkajúce sa výkonnosti systému a aké sú rozhrania s inými systémami. Získané požiadavky musí analytik zapísať a overiť. Mal by špecifikovať funkcie, ktoré má systém poskytovať a nie ako navrhnuť daný systém. Musí však vedieť poradiť, čo je a čo nie je technicky možné zrealizovať (najmä v súvislosti s ohraničeniami konkrétneho projektu).

Niekedy musí zjednotiť rôzne pohľady na výsledný systém. Zákazník a používateľ často nie sú totožní. Zákazník sa snaží o maximálnu efektivitu vynaložených prostriedkov, zatiaľ čo používateľ vyžaduje maximálne pohodlie počas práce so systémom. Rozličné skupiny používateľov môžu mať rôzne predstavy o systéme.

### **Podceňovať špecifikáciu požiadaviek sa nevypláca**

Niektoré vývojárske tímy podceňujú špecifikáciu požiadaviek [Berry]. Charakteristický výrok, ktorý počuť z úst analytika, je takýto: „*Ludia, začnite implementovať, ja idem vypátrať, čo zákazník chce!*“ Manažéri niektorých tímov síce uznávajú, že špecifikácia požiadaviek je veľmi dôležitá, no napriek tomu tvrdia: „*Nemáme čas na dôkladnú špecifikáciu, musíme implementovať, aby sme stihli termín odovzdania.*“

Nuž a napokon extrémny názor manažérov tímov, ktoré sa naplno vrhnú do vývoja systému bez stanovenia špecifikácie požiadaviek: „*Nikdy sme nenapísali dokument špecifikácie a napriek tomu sme stále úspešní.*“ Boehm prirovnáva tieto tímy k cestovateľom, ktorí sa vybrali na dlhú cestu bez mapy.

### **Vytvárať prototyp sa určite oplatí**

Pri prvom kontakte potenciálneho používateľa s prototypom systému dochádza často k dramatickej zmene požiadaviek, najmä tých, ktoré sa týkajú používateľského rozhrania systému. Dochádza k tzv. *IKIWISI efektu* (angl. *I'll Know It When I See It*).

Používatelia pred dodaním prototypu často nevedia povedať, čo vlastne chcú, no teraz to pomenujú vcelku presne. Takisto dokážu určiť, čo vlastne ani nechcú a nepotrebujú. Na základe týchto reakcií a pripomienok sa upresňuje špecifikácia požiadaviek na systém. Pomocou prototypovania si obe zúčastnené strany – analytik a zákazník vyjasňujú nielen požiadavky, ale aj ich interpretáciu. Pri prototypovaní platí

pravidlo: „*Pokiaľ obrázok má hodnotu tisíc slov, tak prototyp môže mať takú hodnotu ako tisíc obrázkov.*“

### **Použiť dodané výrobky?**

Dodané výrobky (*COTS, angl. commercial off-the-shelf*) môžu výrazne ušetriť čas vývoja a peniaze. Softvéroví inžinieri však musia nájsť ten správny výrobok, ktorý vyhovuje požiadavkám zákazníka.

Aj keď je obstarávanie dodaných výrobkov v súčasnosti stále viac populárne, úloha nájsť správny výrobok zostáva naďalej problematická. Dôkazom toho je prípad, o ktorom som sa dočítal v článku [Maiden98]. Systém, ktorý používali zamestnanci The London Ambulance Service, zlyhal kvôli zlému výberu dodaného výrobku. Po zavedení systému vznikol veľký zmätok, preto ho stiahli z prevádzky a vrátili sa k pôvodnému manuálnemu dispečingovému systému.

Softvéroví inžinieri môžu počas získavania požiadaviek robiť prieskum trhu kvôli identifikácii vhodných kandidátov na dodané výrobky. Preto je vhodné detailne získať tie požiadavky, ktoré dovoľia efektívnu selekciu týchto kandidátov. Kvôli tejto selekcii je takisto vhodné, ak sa požiadavky dajú merať. Toto je však ľahšie vysloviť ako v praxi uskutočniť.

Akonáhle sa používanie dodaných výrobkov stane ešte rozšírenejšie, požiadavky sa budú čoraz častejšie vyjadrovať v tvare: „*Chceme niečo podobné ako tamten produkt.*“ [Maiden98]

Musíme si však uvedomiť, že vlastnosti najlepšieho dostupného dodaného výrobku determinujú požiadavky na systém. Predstavme si, že zákazník požaduje jednosekundový čas odozvy systému na spracovanie transakcií. Najlepšie databázové systémy poskytujú dvojsekundový čas odozvy. Boehm kladie rečnícku otázku: „*Chystáte sa vytvoriť svoju vlastnú verziu Oracle alebo Sybase s nádejou, že ju spravíte dvakrát rýchlejšiu?*“ V tejto situácii musíme rozpoznať, že splniť takýto čas odozvy nie je adekvátna požiadavka.

### **Život je zmena**

Počas vývoja softvérového systému dochádza často k zmene požiadaviek. Dokonca Hall uvádza, že jediný zaručene platný fakt týkajúci sa požiadaviek na systém je to, že sa časom určite menia. Preto je zbytočné uvažovať o zmene požiadaviek ako o probléme [Hall97].

Počas vývoja softvérového systému sa analytik prostredníctvom zjemňovania požiadaviek dozvedá od zákazníka, čo by mal systém robiť. Najlepší spôsob, ako to dosiahnuť, je neustále presnejšie vymedzovať požiadavky. Požiadavky musí analytik spravovať a zmeny v nich sledovať. Takto môže preukázať vývoj požiadaviek v čase. Kvôli tomu, že zmeny požiadaviek môžu nastať v najnevhodnejšom čase, ktorý dokonca ohrozuje termín odovzdania, je vhodné niektoré zmeny požiadaviek predvídať. To však vyžaduje od analytika značné skúsenosti s vývojom

podobných systémov. Našťastie počas dlhoročnej práce na riešení množstva projektov objavuje analytik medzi nimi určité súvislosti a v sebe schopnosť pozeráť sa na systémy z nadhľadu. Takisto by mal analytik porozmýšľať, u ktorých požiadaviek je pravdepodobnosť zmeny menšia.

Bolo by pekné, keby sme mohli navrhovať systémy, ktoré by boli natrvalo flexibilné, ale to nemôžeme. Je potrebné navrhnuť správnu mieru flexibility. Sústrediť svoje úsilie na veci, ktoré sa menia pomaly. Požiadavky vždy závisia od faktov z reálneho sveta a od prání používateľa. Fakty z reálneho sveta sa menia pomalšie ako prania používateľov. Dokonca aj keď dôjde k zmene detailov, základné fakty zostávajú rovnaké. Napr. lietadlo bude stále mať výšku letu a kurz, aj keby sa technológia ich merania zmenila.

Úlohou softvérového inžiniera je navrhnuť riešenia a implementovať požiadavky do systému. Nemal by argumentovať tým, že implementácia nie je možná alebo je príliš nákladná. Pri vývoji softvérového systému platí pravidlo: „*Zákazník má vždy pravdu.*“ S týmto konštatovaním úzko súvisí aj ďalšie pravidlo: „*Zákazník platí (alebo neplatí).*“

Softvéroví inžinieri sa často ocitnú zoči-voči veľkému množstvu požiadaviek na systém spolu s nedostatkom času a peňazí na ich implementáciu. Preto sa musia stanoviť priority jednotlivých požiadaviek. Rôzne priority určuje zákazník, iné koncový používateľ a iné softvéroví inžinieri. Preto musia nájsť „spoločnú reč“, ako tieto požiadavky usporiadať podľa priority. Musia vybrať základné požiadavky, ktorých implementácia najviac pridá systému na hodnote. Ak systém nebude spĺňať niektorú z týchto základných požiadaviek, nemožno ho odovzdať.

### **Moja osobná skúsenosť**

Keď som vyberal túto tému na esej, hneď som si spomenul na projekt Zber a spracovanie normatívnych dát pre elektromyografiu, ktorý riešime v rámci predmetu Tímový projekt. Na projekte spolupracujeme s lekárom z Fakultnej nemocnice. Tento lekár predstavuje typický príklad zákazníka, ktorý nevie, čo presne chce. Na túto skutočnosť nás už na začiatku projektu upozornili pedagógovia a študenti, ktorí na ňom pracovali pred nami.

Počas riešenia projektu sme pochopili náročnosť úlohy analytika – preniknúť do odbornej terminológie používanej v elektromyografii nebolo pre nás vôbec jednoduché.

Úvodnú špecifikáciu požiadaviek sme stanovili na základe analýzy systému, ktorý vytvorili študenti pred nami. Napríklad jedna z požiadaviek bola umožniť lekárovi rozhodnúť, ktoré z nameraných údajov zahrnie do databázy a ktoré nie. No na stretnutí so zákazníkom sme sa dozvedeli, že táto možnosť nie je vhodná kvôli novej subjektivite lekára pri takomto rozhodovaní.

Keďže momentálne projekt ešte prebieha, neviem, aký bude výsledok nášho snaženia. Jedno je už v tejto chvíli isté – na vlastnej koži sme pocítili úskalia vzťahu softvérového inžiniera a zákazníka s nejasnou predstavou o výslednom systéme.

**P**rečo písať špecifikáciu požiadaviek? Brooks to vyjadril vo svojej eseji No Silver Bullet [Brooks95] takto: „Najťažšia časť tvorby softvérového systému je presne rozhodnúť, čo máme vytvoriť. Žiadna iná časť tvorby systému nie je tak ťažká ako práve vytvorenie detailnej technickej špecifikácie vrátane prepojenia na ľudí, zariadenia a iné softvérové systémy. Žiadna iná časť nepoškodí výsledný systém tak ako nekvalitná špecifikácia.“ Preto by sme ju nemali podceňovať.

Kvalitná komunikácia so zákazníkom predstavuje základ úspechu projektu. Treba si uvedomiť, že zákazník na začiatku často nevie, čo vlastne chce. Softvéroví inžinieri majú nielen implementovať to, čo zákazník vie, ale mu majú pomôcť špecifikovať to, čo by bolo pre neho užitočné.

Nuž, údel softvérového inžiniera počas riešenia projektu je už raz taký – občas neobeduje, vypije zopár litrov kávy, niekedy sa primerane nevyspí, nadáva na zmeny požiadaviek, nič nestíha... a na konci projektu má dobrý pocit z vytvoreného systému, ktorý spĺňa požiadavky zákazníka (a ešte lepší pocit z narastajúceho konta v banke).

## Literatúra

- [Berry] BERRY, D. M.: *The Requirements Iceberg and Various Icepicks Chipping at it*. <http://www.student.math.uwaterloo.ca/~cs445/handouts/lectureSlides/W01/documents/iceberg.slides.pdf>. (15. 3. 2001).
- [Boehm00] BOEHM, B.: *Requirements that Handle IKIWISI, COTS and Rapid Change*. IEEE Computer, Júl 2000, vol. 33, no. 7., s. 99–102. <http://www.itpolicy.gsa.gov/mke/archplus/ieee-boehm.pdf>. (15. 3. 2001).
- [Brooks95] BROOKS, FREDERICK P.: *The Mythical Man-Month: Essays on Software Engineering*. 2. vyd. Addison-Wesley, 1995. 336 s. ISBN 020135959.
- [Hall97] HALL, A.: *Realistic requirement engineering: Dealing with change*. Interface – Newsletter of the Software Engineering Process Group, Máj 1997, vol. 6, no. 2, s. 1–2. <http://www.faa.gov/aio/common/nwsltrs/97v6no2.pdf>. (15. 3. 2001).

- [Maiden98] MAIDEN, N.A. – NCUBE, C.: *Acquiring COTS Software Selection Requirements*. IEEE Software, Marec/Apríl 1998, s. 46-56.
- [Reifero0] REIFER, D. J.: *Requirements Management: Search for Nirvana*. IEEE Software, Máj/Jún 2000, s. 45-47.

## Ako zlepšiť softvérové procesy

Podľa: Barry Boehm - Unifying Software Engineering and Systems Engineering.

Szabolcs Molnár

**Abstrakt.** Esej sa zaoberá unifikáciou softvérového a systémového inžinierstva. Približuje problémy, ktoré môžu vzniknúť v softvérových projektoch, keď nastane efekt tzv. separácie zainteresovania, ktorá môže viesť aj k zlyhaniu projektu. Zaoberá sa modelom vyspelosti procesu – CMM, ktorý slúži na klasifikáciu úrovné procesov. Uvediem niektoré konkrétne prípady, v ktorých výskumníci pokúsili zjednotiť softvérové a systémové inžinierstvo s CMM.

Podkladom tejto esej je publikácia Barry Boehma názvom Unifying Software Engineering and Systems Engineering, ktorá sa zaoberá myšlienkou, ako spojiť alebo zjednotiť procesy softvérového a systémového inžinierstva.

Niektorí ľudia, ktorí sa zaoberajú vývojom softvérových systémov si myslia, že softvérové procesy sú zbytočné, obmedzujúce a nepoužiteľné. Niektorí majú názor, že stačí iba zaplatiť najlepších ľudí a poskytnúť im všetky potrebné zdroje. Uznajú síce, že ich práca nebude najefektívnejšia a urobia mnoho zbytočných vecí, ktoré by mohli ušetriť, keby používali procesy, ale chyby neurobí iba ten človek, kto neurobí vôbec nič. Takýto prístup vývojárov avšak v žiadnom prípade nemôžeme považovať za vhodný. Veď ako by sme vyriešili väčšie projekty, keď nemáme k dispozícii žiadne podporné procesy? Používanie procesov však neprinesie automaticky so sebou úspešnosť projektu, musíme ich využívať efektívne. K tomu slúži aj zjednotenie systémových a softvérových procesov.

Prv než sa pohlúžime do problematiky, musíme si definovať, čo je vlastne softvérový proces. Učebnica Softvérové inžinierstvo. Princípy a manažment [Bieliková00], definuje tento pojem nasledovne: Je to „proces, zahŕňujúci technické aspekty a aspekty manažmentu vývoja softvéru; určuje abstraktnú množinu činností, ktoré sa majú vykonať pri vývoji softvérového výrobku z pôvodných požiadaviek používateľa.”

Základná klasifikácia softvérových procesov je nasledovná: procesy manažmentu procesu, procesy vývoja softvéru, procesy manažmentu softvéru a procesy manažmentu projektu.

Informačná technológia sa rozvíja neuveriteľnou rýchlosťou. Tento vývoj priniesol so sebou potrebu odstrániť predtým krvopotne vytvorené zmeny. V súčasnosti jednotlivé organizácie dokážu zmeniť pomalé, pasívne a oddelené softvérové a systémovo inžinierske procesy na normalizované procesy. Takýmto spôsobom vytvorené procesy lepšie vyhovujú rýchlemu vývoju dynamicky sa meniacich softvérových systémov, ktoré využívajú: agenty, celosvetová pavučina, multimediálne aplikácie alebo Internetová technológia. Ale nebolo to vždy také jednoduché.

### **Pokusy v minulosti**

V minulosti vykonanie kultúrnych zmien bolo ešte náročnejším procesom ako v súčasnosti. Na znázornenie Boehm uviedol nasledujúci príklad: V 70-tych rokoch spoločnosť názvom TRW vytvárala podnikovú kultúru softvérového inžinierstva okolo známeho sekvenčného vodopádového modelu. Usilovali sa vytvárať podnikové zásady a štandardy, tréningové kurzy riadené manažérom a prísne kontrolovali vytvorené zásady. V tom čase si ani neuvedomili, ako ťažko bude tieto zásady zmeniť. Problémy vznikli o niekoľko rokov neskôr, keď sa pokúsili niektoré predchádzajúce zmeny odstrániť.

V 80-tych rokoch sa mnohé veľké projekty realizovali vodopádovým modelom, ktorý má niekoľko vážnych nedostatkov. Tie sú napríklad: zákazník uvidí systém iba na konci projektu a preto neskoro odhalené nedostatky môžu vážne ohroziť úspešnosť projektu; malá prispôbivosť zmenám po rozbehnutí procesu; model zvyšuje riziko pri vývoji nových druhov aplikácií atď. Veľké projekty, ktoré boli realizované pomocou tohto sekvenčného prístupu, veľmi dobre znázorňujú nesúlad medzi sekvenčným, deduktívnym vodopádovým modelom a medzi potrebou umelých prístupov, ktoré podporujú nečakané zmeny pri tzv. user-intensive systémoch. Keď sa výskumníci v TRW pokúsili zaviesť prototypovanie do takých projektov, narážali na silný odpor zo strany softvérových inžinierov, ktorí tvrdili, že takéto prototypovanie nie je zrealizovateľné. Vyhlásili, že prototyp sa realizuje predtým, než by prešiel cez výskum kritického návrhu a toto absolútne znemožňuje efektívne využitie podnikových zásad.

V súčasnosti existuje viac noriem, ktoré slúžia na zlepšovanie softvérových procesov. Jedna z najznámejších je tzv. model vyspelosti procesu (Capability Maturity Model - CMM), ktorý bol vyvinutý na inštitúte softvérového inžinierstva názvom SEI-CMU [Paulko1]. Tento model klasifikuje vyspelosť a úroveň procesu. Základný model sa zaoberá tvorbou softvérových systémov. Model má 5 úrovní, ktoré sú: počiatočná, opakovateľná, definovaná, riadená, optimalizujúca. Tieto úrovne sú hierarchické a žiadnu z nich nemôžeme vynechať [Bieliková00]. Niektoré

ďalšie normy, ktoré súvisia so zlepšovaním procesu sú napríklad: ISO 9001 – norma pre manažment akosti a SPICE – norma pre ohodnotenie procesu. Ďalšia norma je napríklad IEEE-EIA 12207, ktorá pokrýva celý životný cyklus projektu. Elementy tejto normy môžeme namapovať na CMM elementy [Ferguson98].

V USA prvý príklad na integráciu bol integrovaný softvérový model vyspelosti procesu FFA, ktorý spojil softvérové inžinierstvo, systémové inžinierstvo a CMM. Ďalší podobný príklad na integráciu je CMMI, ktorý zjednotil softvérové inžinierstvo, systémové inžinierstvo a CMM model, a poskytoval aj možnosť integráciu ďalších CMM modelov [Boehm00].

### **Dedičstvo vodopádového modelu**

V CMM modeli verzie 1.1 sa ukazuje obrovské zlepšenie s porovnaním predošlými nedisciplinovanými konaniami, ale tento model ešte stále podporuje sekvenčný vodopádový prístup návrhu systému. Toto dedičstvo brzdí efektívny vývoj dynamicky sa meniacich systémov, teda nie je možné používať skoré prototypovanie, tzv. concurrent engineering, manažment rizík, a ďalšie užitočné kroky, ktorými by sa dalo zjednotiť a zjednodušiť softvérové a systémové inžinierstvo. Autori CMM modelu tvrdia napríklad: „Analýza a rozdelenie systémových požiadaviek nie je povinnosťou softvérových inžinierov, no je predpokladom ich práce“ [Paulko1]. Podľa Boehma sa toto vyhlásenie uplatní iba v niektorých prípadoch, ba dokonca takýto postoj softvérových inžinierov môže byť veľmi nebezpečný. Také nebezpečenstvo môže byť napríklad tzv. oddelenie zainteresovania (separation of concern), čo znamená, že softvéroví inžinieri sa nezaoberajú určovaním systémových architektúr a systémových požiadaviek, túto prácu ponechajú na iných. Keby sme súhlasili takýmto postojom, vývojári softvérových systémov by nepotrebovali žiadne činnosti, ktoré poskytuje softvérové inžinierstvo.

Boehm skúmal sociálne dôsledky takého postoja pri niektorých príležitostiach stretnutí výskumných pracovníkov, kde navrhovali letecké systémové architektúry. Pokiaľ hardvéroví a systémoví inžinieri diskutovali o tom, aké boli ich predchádzajúce systémové architektúry, softvéroví inžinieri zostali v pozadí. Čakali na niekoho, kto by im dal presnú špecifikáciu, čo by oni premenili na kód.

Žiaľ tú separáciu v značnej miere podporujú aj súčasné softvérovo inžinierske modely, ktoré sa sústreďia na abstraktné logické úlohy. V praxi však rozhodujúcim faktorom je výkonnosť a náklady softvérových systémov. Boehm hľadal v 16 nových učebniciach o objektovo-orientovanom návrhu už spomenuté výrazy, ale iba v 6 sa vyskytovalo v registri slovo „výkonnosť“ a iba v 2 slovo „náklady“. Ak rozdelíme úlohy na abstraktné, logické, oddelené podúlohy, môže byť ohrozená úspešnosť celého projektu.

### **Projekt, ktorý takmer zlyhal**

Takéto projekty môžu byť katastrofické alebo v „lepšom“ prípade blízko-katastrofické, ako uvidíme aj z nasledovného príkladu:

Na začiatku 80-tych rokov jedna veľká vládna organizácia uzatvorila zmluvu so spoločnosťou TRW na vývoj a analýzu informačného systému. S navrhnutým systémom by malo pracovať viac ako 1000 používateľov, ktorí nachádzali geograficky vzdialených miestach. Ďalšou požiadavkou systému bolo, aby systém používal dynamickú databázu. Spoločnosť TRW spolu so zákazníkmi si stanovili, že projekt bude vyriešený pomocou klasického vodopádového modelu a doba odozvy systému bude menšia ako 1 sekunda.

Požiadavky boli definované na viac ako 2000 stranách. Aby systém vyhovoval požiadavkám, museli by navrhnuť špeciálny podsystém na urýchlenie systémovej odozvy tak, aby používateľ dostal požadované údaje v najkratšom čase.

Navrhnutá hardvérová aplikácia mala viac ako 25 superpočítačov, ktoré sa starali o cache-ovanie údajov podľa daného algoritmu. Rozpočet systému odhadli okolo 100 miliónov USD. Vyplývalo to z komplexnosti hardvérovej a softvérovej architektúry.

Toto riešenie z hľadiska rozpočtu bolo veľmi neatraktívne, preto zákazníci a vývojári sa rozhodli, že navrhnu prototyp používateľského rozhrania systému, ktorých reprezentatívne vlastnosti sa dajú otestovať. Z testov sa dozvedeli, že aj 4 sekundová odozva systému by bola postačujúca pre 90 % používateľov. 4 sekundová odozva systému zredukovala cenu celého rozpočtu až na 30 milión USD.

V takýchto prípadoch môžeme zbrať nebezpečenstvá, že ak softvéroví inžinieri nezapoja do procesu špecifikácie systémových požiadaviek, môžu nastať vážne problémy, čo môže viesť k neúspešnosti celého projektu.

### **Podporné metódy**

Aktuálny náčrt modelu CMMI-SE/SW a jeho doplnenie (Integrated Process and Product Development – IPPD) sa nachádza aj na celosvetovej pavučine. Celý balík predstavuje novú paradigmu pre súbežné systémové a softvérové inžinierstvo. Obsahuje 3 nové oblasti procesov, ktoré sú: integrovaný tím (integrated team), spoločné zobrazenie (shared vision) a spolupracujúce vedenie (collaborative leadership). Tzv. concurrent engineering pozostáva z kombinovania softvérového inžinierstva, systémového inžinierstva a oblasti IPPD procesov, ktorá zahŕňa v sebe také procesy, ako sú napríklad: zákaznícke a produktové procesy, technické riešenie, plánovanie projektu a manažment rizík. Tento prístup pomôže premeniť zákazníckové požiadavky na funkčnú špecifikáciu, pomôže vytvoriť scenáre činnosti programu, používateľské rozhranie a pomocou tohto prístupu môžeme

vykonať aj testovanie. Takýmto spôsobom môžeme minimalizovať riziko i náklady a môžeme analyzovať primeranosť softvérového systému.

Metodológia CMMI tak poskytne softvérovým inžinierom hlavné miesto medzi vývojármi. CMMI okrem toho identifikuje aj dodatočné činnosti, ktoré sú nutné pre systémy a pre softvéry, aby bolo súbežné inžinierstvo úspešné. Samozrejme ani táto metóda nevyrieši všetko. Nedefinuje napríklad pracovné toky a metódy, ktoré sa používajú pri vykonávaní jednotlivých činností [Boehm00].

Ďalší prostriedok na podporu je DMR-BRA. Toto okrem toho, že podporuje začiatočnú obchodnú prípadovú analýzu, ktorá iba spája softvérové a systémové inžinierstvo s obchodným manažmentom, poskytuje aj uzavretú slučku so spätnou väzbou, ktorá zabezpečuje monitorovanie procesu [Boehm00].

Normu CMMI a spomenuté projektovacie metódy som opísal na demonštráciu možnosti pre zlepšenie procesov. Organizácie môžu tie zlepšenia používať na unifikáciu oddelených systémových a softvérových procesov, ktorými môžu ušetriť aj dosť vysoké náklady. Ako som už spomenul na začiatku, kultúrne zmeny ešte nikdy neboli také jednoduché ako v súčasnosti, ale treba si uvedomiť, že výber správnej metódy zo všetkých, nie je najľahší a nie je ani najpríjemnejší.

### Použitá literatúra

- [Bieliková00] BIELIKOVÁ, M.: *Softvérové inžinierstvo. Princípy a manažment*. Bratislava: Slovenská technická univerzita, 2000. ISBN 80-227-1322-8.
- [Boehm00] BOEHM, B.: *Unifying Software Engineering and System Engineering*. IEEE Computer, Marec 2000, s. 114-116.
- [Ferguson98] FERGUSON, J – SHEARD, S.: *Leaving Your CMM for IEEE/EIA 12207*. IEEE Software, September/Október 1998, s. 23-28.
- [Paulk01] PAULK, M. ET AL: *The Capability Maturity Model for Software*.  
<http://www.sei.cmu.edu/pub/cmm/Misc/cmm.pdf>.  
(Marec 2001).



## Ako urýchliť vývoj aplikácií?

*Esej je inšpirovaná pôvodnou esejou  
B. Boehm: Making RAD Work for Your Project [Boehm99].*

Radoslav Kováč

**Abstrakt.** Článok sa zaoberá metódami tvorby softvéru označovanými ako "Rapid Application Development" (RAD). Tieto metódy predstavujú v podmienkach dynamicky sa rozvíjajúceho trhu úspešnú stratégiu vývoja softvéru. Ich hlavným cieľom je skrátenie časového plánu dodania produktu zákazníkom.

**L**idská spoločnosť sa nachádza vo veku informácií. Digitálne informácie nám otvorili nové možnosti a informačné technológie zmenili množstvo našich každodenných aktivít. Informačné produkty prenikajú do všetkých odvetví ľudskej činnosti a nové technológie neustále vytvárajú priestor pre ďalšie príležitosti. V tomto dynamickom prostredí sa spontánne rozvíja trh so softvérom a ďalšími informačnými produktmi. Softvérové spoločnosti si hľadajú svojich zákazníkov a snažia sa riešiť problémy vznikajúce pri vývoji stále zložitejších systémov. To všetko sa deje pod neustálym časovým tlakom.

Čas je určite najkritickejším faktorom väčšiny softvérových projektov. Termíny a čas uvedenia na trh najvýznamnejšie vplývajú na úspech produktov, ktoré sú vytvárané pre trh s veľkým počtom zákazníkov a malým počtom potenciálnych dodávateľov [Card95] (typickým príkladom je zavádzanie nových služieb prevádzkovateľov mobilných telefónnych sietí). V takomto prípade zabezpečí skoré uvedenie produktu na trh získanie viacerých zákazníkov a získanie väčšieho podielu na trhu; taktiež je príslubom rýchlejšej návratnosti investícií, väčšieho rozšírenia, rozsiahlejšej podpory a vyšších ziskov z predaja nasledujúcich verzií a produktov. Existencia konkurenčných produktov alebo novších technológií môže spôsobiť zastaranie produktu, a preto sa často hovorí o konkrétnom časovom okne pre uvedenie produktu na trh a vyžadujú sa pevné termíny [Olsen95].

V súčasnosti sa pri vytváraní obchodnej stratégie softvérových spoločností diskutuje najmä o spôsoboch pre zlepšenie kvality procesu vývoja a tým aj výsledných softvérových produktov. Veľké spoločnosti sa

snažia splniť normy (napr. ISO-9000) kladené na výrobný proces a získať príslušný certifikát kvality. Kvalita sa však môže stať aj akýmsi neflexibilným ideálom, vďaka ktorému síce veľké firmy v silnom konkurenčnom prostredí získavajú najväčšie zákazky, no na trhu existuje (hlavne vďaka neustálemu prenikaniu nových technológií) stále dostatok príležitostí pre dynamickejšie spoločnosti s prispôsobivejšími stratégiami. Tieto dynamické spoločnosti dokážu pružnejšie reagovať na potreby trhu a dosiahnuť lepšiu kombináciu nákladov, dodacích termínov, dosiahnutých vlastností a kvality produktov. Minulosť ukázala, že aj produkty s nie práve najlepšimi vlastnosťami sa dokázali presadiť a spoločnosti, ktoré ich vyvinuli, si získali pozíciu, z ktorej ťažia dodnes. Preto tu existuje evidentný záujem o preskúvanie nových prístupov k vytváraniu produktov, ktoré by lepšie uspokojili požiadavky dnešného trhu.

Súborom takýchto metód, ktoré sa však v žiadnom prípade nesnažia poskytnúť jediné možné riešenie, sú techniky označované anglickým termínom "Rapid Application Development" (RAD). RAD sľubujú použitie metód a nástrojov, ktoré v konkrétnych situáciách môžu viesť k zníženiu nákladov na vývoj, ale najmä k skráteniu času vývoja i dodania produktu. Nemožno ich však použiť bez hlbšieho porozumenia jednotlivým cyklom softvérového procesu a odhalenia základných súvislostí, ktoré spôsobujú oneskorenie samotného dodania. Barry Boehm v súvislosti s použitím RAD techník [Boehm99] varuje pred ich aplikáciou formou tzv. "**Dumb RAD**" (DRAD). DRAD často vzniká, ak manažéri stanovia nesplniteľný termín dodania softvéru a projekt je tak od začiatku odsúdený na neúspech. Takéto rozhodnutia sa často prijímajú pod tlakom vonkajších okolností (vládne projekty, politické rozhodnutia) a súvisia s problémami pri obhajovaní nereálnosti naplánovaných termínov. V takomto prípade projekt určite nezachráni žiadne "zázračné" RAD techniky.

RAD techniky sa zameriavajú na skrátenie času trvania všetkých etáp softvérového procesu. Urýchlenie procesu možno dosiahnuť pomocou 4 fundamentálnych aspektov: akceleračných nástrojov, manažmentu, metodológie a ľudí [Agarwal00].

### Akceleračné nástroje

V softvérovom inžinierstve sa pre počítačovú podporu vývoja a manažmentu používajú tzv. CASE (Computer Aided Software Engineering) nástroje. Tieto nástroje sa často integrujú do komplexných prostredí, ktoré sa zameriavajú najmä na návrhové etapy softvérového procesu a vo veľkých tímoch dokážu zlepšiť kvalitu dokumentácie i celého procesu vývoja. Pri RAD projektoch sa však často vychádza z nedostatočnej špecifikácie problému (a preto sa prechádza viacerými etapami prototypovania) a pracuje sa najčastejšie v menších tímoch, kde by niektoré činnosti štandardných procesov vývoja zbytočne zdržovali. Preto sa za RAD nástroje považujú najmä jednoduchšie nástroje určené priamo pre generovanie aplikácií alebo pre vytváranie prototypov. Určite

však existuje množstvo ďalších nástrojov, ktoré dokážu ušetriť čas pri vytváraní dokumentácie, testovaní a pri práci v tíme.

Výhody použitia generátorov a vyšších programovacích i špecifikačných jazykov vychádza zo základného vnútorného problému softvéru – z jeho zložitosti. Pozrime sa preto na tento problém bližšie. Vytvorením počítačového programu sa snažíme donútiť počítač (ktorý máme k dispozícii), aby riešil naše problémy (ktoré vychádzajú z našich, čisto ľudských potrieb). V roku 1936 navrhol A. M. Turing matematický model automatu, pomocou ktorého vieme zapísať ľubovoľný dnes známy algoritmus. Takýto zápis (pomocou prechodovej funkcie) by však bol veľmi nepraktický, a preto aj von Neumannova koncepcia univerzálneho počítača (1946) bola oveľa prijateľnejšia pre zápis algoritmov riešiacich klasické výpočtové problémy. Odvtedy sa pre univerzálny počítač našlo uplatnenie vo všetkých oblastiach celej ľudskej spoločnosti a s užitočnosťou počítačov neustále narastajú požiadavky ich používateľov.

Človek teda musel nachádzať stále ďalšie spôsoby ako efektívne vytvárať programy pre počítač. Používanie vyšších štruktúrovaných programovacích jazykov prinieslo nevyhnutné zvýšenie čitateľnosti a samodokumentovateľnosti zložitých programov, čo umožnilo vytváranie väčších softvérových systémov. Tento zásadný skok (možno povedať, že o celý rád) v produktivite programátorov nebol odvtedy dosiahnutý žiadnymi ďalšími úpravami programovacích jazykov (pri zachovaní rovnakej vyjadrovacej sily). Ďalšie zvýšenie produktivity sa neskôr dosiahlo kontrolou kvality softvérového procesu [Cross]. Základným problémom však zostáva vysoká zložitosť samotného softvéru a neschopnosť vývojárov pracovať na návrhu, implementácii a testovaní takéhoto systému (preto je úlohou dobrého návrhu architektúry skryť pred programátormi zložitosť čo najväčšej časti systému).

Jedným z riešení zvýšenia produktivity pri vytváraní zložitých systémov je špecifikovanie ich správania čo najjednoduchším spôsobom, ktorý by nepreniesol zložitosť systému na jeho tvorcov. To však nie je možné dosiahnuť, ak sa pokúšame vyriešiť veľkú triedu rozličných problémov, lebo v takom prípade musíme použiť dostatočne univerzálny špecifikačný (programovací) jazyk a potom bude vyjadrenie riešenia nášho konkrétneho problému vždy zložitá (lebo podobne vieme vyriešiť aj ľubovoľný iný problém). Preto musíme použiť špecializovanejší špecifikačný prostriedok, v ktorom bude jednoduchšie zapísať požadované správanie. Snahou je teda vyjadriť program pomocou čo najvýstižnejších prostriedkov pre opis problémov konkrétnej aplikačnej oblasti a pritom sa nezaťažovať zložitou reprezentáciou pre technické prostriedky, ktoré budú daný program vykonávať. Iný prístup vychádza z princípu skrývania informácií, kedy sú pred každým tvorcom skryté tie časti systému, o ktorých zložitosti nemusí nutne vedieť. V takomto prípade môže byť vhodné použiť aj univerzálny programovací jazyk, avšak je potrebné dobre navrhnuť rozhrania medzi časťami systému a mať k dispozícii potrebné makrá, knižnice a súčasti. Pritom treba

myslieť aj na to, že integrácia súčiastok tiež nesmie viesť do nášho systému zbytočne veľa zložitosti.

RAD nástroje sa snažia eliminovať túto zložitosť a poskytnúť tvorcom systému len tie najnevyhnutnejšie prostriedky pre dosiahnutie požadovaného správania. Z toho však vyplýva, že ich možno použiť iba v určitých aplikačných doménach a nie sú univerzálne. Vytvorenie kompletného produktu použitím takýchto nástrojov sa označuje ako "**Generator RAD**" (GRAD) [Boehm99]. Zahŕňa použitie špecifikačných jazykov veľmi vysokej úrovne – napríklad jazykov 4 generácie (napr. Focus) alebo iných doménovo špecifických jazykov pre finančné aplikácie alebo pre riadenie priemyselných strojov. Medzi takéto nástroje možno zaradiť napríklad aj tabuľkové procesory. Iným príkladom môže byť použitie generátorov agentov pre multiagentové systémy, kedy sa zadefinuje správanie agenta na vyššej úrovni a použije sa generátor pre vytvorenie zdrojového kódu (napr. Zeus). Podobne aj zadefinovanie lexikálneho a syntaktického analyzátora špecifikovaným gramatiky jazyka (napr. programy lex a yacc). S obľubou sa tiež používajú skriptovacie jazyky, ktoré umožňujú pohodlnejšie zadefinovať správanie – napríklad pri inštalácii aplikácie (Installshield) alebo aj pre činnosť data warehouse servera.

GRAD sa najčastejšie aplikuje v ohraničených doménach, kde sú požiadavky zákazníkov dobre známe. V takom prípade umožňuje špecifikačný jazyk pohodlne zadefinovať požadovanú funkcionálnosť a generátor s využitím pred-vytvorených súčiastok dokáže vytvoriť samotný produkt splňujúci používateľské požiadavky. Používanie generátorov vyžaduje len minimálne programátorské skúsenosti a je možné pomerne rýchlo získať uspokojivé výsledky.

Základným problémom GRAD je slabá škálovateľnosť [Boehm99], pretože generátory umožňujú pohodlné vyjadrovanie často na úkor efektívnosti. V prípade väčších systémov potom môže rýchlo dôjsť k preťaženiu systému a jeho úplnému odstaveniu, čo môže mať katastrofálne následky pre celý projekt. Používanie vyšších špecifikačných jazykov a generátorov taktiež môže výrazne ohraničiť realizovateľné funkcie, a preto je niekedy potrebné myslieť aj na možnosti rozšírenia, ktoré špecifikačný jazyk priamo neposkytuje.

Ďalšou z foriem RAD je "**Composition RAD**" (CRAD) [Boehm99]. CRAD vychádza vo veľkej miere zo znovupoužitia väčších existujúcich súčiastok (tried knižníc) pre univerzálnejšie programovacie jazyky v kombinácii s nástrojmi urýchľujúcimi vývoj. V dnešnej dobe je nevyhnutnosťou využívať základné funkcie poskytované samotným operačným systémom. Využitie týchto funkcií spolu s ďalšími knižnicami pre podporu grafického výstupu, databázového spracovania a sieťových protokolov umožňujú veľmi rýchle vytvorenie veľkej skupiny stredne veľkých aplikácií. V procese implementácie sa osvedčilo použitie tzv. pomocníkov (angl. wizardov) pre generovanie šablón pre kód aplikácie a vizuálnych nástrojov pre vytváranie používateľského rozhrania.

Oblíbenosť týchto nástrojov (príkladom je prostredie Visual Basic, Delphi a množstvo databázových jazykov) súvisí s rozsiahlou hierarchiou tried a jednoduchosťou pridávania nových funkcií.

Pri vytváraní väčších systémov použitím CRAD prostredí treba opäť zvážiť škálovateľnosť. Najväčším rizikom však môže byť, ak sa neskúsení programátori nechajú viesť najjednoduchšími možnosťami, ktoré im prostredie priamo poskytuje, a potom vytvoria systém, ktorý je len veľmi ťažko modifikovateľný (to často súvisí s prepletením a vysokou zviazanosťou kódu používateľského rozhrania so samotnou funkcionalitou systému).

Vyvíjanie aplikácií pomocou GRAD a CRAD predpokladá vyššie náklady na vývojové prostredia, spájajú sa s nimi nové riziká, no sľubujú podstatné skrátenie času dodania produktu a zníženie ďalších nákladov. Väčšie používanie GRAD v budúcnosti predpokladá lepšie preskúmanie vhodných aplikačných domén a vytvorenie vhodných špecifikačných jazykov a generátorov pre čo najväčšiu skupinu aplikácií. CRAD vychádza zo znovupoužitia univerzálnych súčiastok a automatických nástrojov pre ich pohodlnú integráciu do vyvíjaných produktov. V tomto prípade bude dôležité vytvorenie ďalších knižníc súčiastok, ktoré by poskytli riešenia pre veľkú časť problémov konkrétnych domén.

### Manažment

Zatiaľ sme sa sústredili iba na možnosť skrátenia etapy implementácie. Pri pohľade na celý softvérový proces však môžeme nájsť viacero činností, ktoré by mohli výrazne urýchliť konečné dodanie produktu. Metódy RAD, ktoré sa o takýto komplexný pohľad pokúšajú, môžeme označiť termínom "**Full-scale RAD**" (FRAD) [Boehm99].

Pri odhadoch času trvania projektu sa často vychádza z grafu činností. Tento orientovaný graf zachytáva závislosti medzi identifikovanými činnosťami a podľa odhadov ich trvania je možné nájsť kritickú cestu, ktorá určuje trvanie projektu. Analýzou grafu činností môžeme určiť činnosti, ktorých prípadné oneskorenie by spôsobilo oneskorenie celého projektu. Zvážením týchto rizík môžeme dospieť k opatreniam voči takýmto neželaným situáciám (poistiť sa zabezpečením náhrady za kľúčový hardvér, prípadne zamestnancov).

Niektoré činnosti je niekedy možné dekomponovať a prípadne ich vykonať paralelne (ak nezávisia od konkrétnych prostriedkov) a tak ich odstrániť z kritickej cesty projektu. Príkladom môže byť nahradenie zdĺhavého procesu prehliadok návrhu systému samostatnými inšpekciami jednotlivých častí spolu s rýchlou spoločnou prehliadkou [Boehm99].

Skrátiť čas alebo úplne vypustiť niektoré činnosti je možné pomocou obstarania si ich u inej spoločnosti (angl. outsourcing). Predpokladá sa, že **obstarávanie produktov i služieb** mimo materskej spoločnosti bude v budúcnosti stále dôležitejšie najmä pri spoločnostiach podnikajúcich v oblasti elektronického obchodu [Kaghazian00].

Obstarávanie môže pomôcť znížiť náklady spoločnosti, zabezpečiť jej jednoduché získanie kvalitných zdrojov, ale pritom si ponechať vlastné zdroje na vlastné aktivity.

V prípade, že je čas kritickým faktorom projektu, je dobré vytvoriť plán tak, aby prioritne predpokladal zahrnutie všetkej základnej funkcionality do produktu a aby sa pri oneskorení projektu pozdržalo len dokončenie ostatných, menej významných funkcií. Pri vytvorení takéhoto plánu (čo sa musí odraziť aj v samotnom návrhu systému) je potom možné využiť aj prístup označovaný ako **nezávislé rozvrhovanie** (angl. *schedule-as-independent-variable*) [Boehm99]. V takomto prípade sa dokáže rozsah projektu dynamicky prispôbiť času, ktorý je k dispozícii, vynechaním funkcií, ktoré nespôsobia problémy v iných častiach systému. Pre každú požiadavku sa vlastne zdefinuje časový rámec, v rámci ktorého musí byť realizovaná, inak bude vypustená, aby nezdržovala dokončenie celého projektu. Plán potom obsahuje viacero takýchto ohraničení.

### Metodológia

Najväčšie zdržanie softvérových projektov spôsobuje opätovné prerábanie už vytvorených častí systému [Boehm99]. To vzniká neskorším došpecifikovaním niektorých požiadaviek alebo neskorým odhalením zásadných chýb návrhu a implementácie. Predísť vzniku mnohých chýb je možné zavedením viacerých preventívnych opatrení: vnútornými štandardami, disciplínou pracovníkov a manažmentom rizík (riziká v softvérových projektoch často súvisia so zavádzaním nových technológií). Skoré odhalenie chýb je možné zabezpečiť pomocou automatického testovania a prehliadok kódu.

Pri vytváraní špecifikácie systému sa RAD metódy často opierajú o prototypovanie a čo najväčšie zainteresovanie budúcich používateľov systému. Vytváranie prvotnej špecifikácie sa organizuje v rámci **spoločného stretnutia s používateľmi** (angl. *joint-application-development*). Pri ňom sa stretnú vývojári s používateľmi a metódou *brainstormingu* sa snažia zostaviť hrubý zoznam požiadaviek. V ďalších fázach sa iteračne vytvárajú prototypy, tie sa následne ohodnocujú používateľmi a podľa toho sa zjemňujú požiadavky. Takýto tesný kontakt s používateľmi umožňuje veľmi rýchlo identifikovať najdôležitejšie funkcie systému a sústrediť sa na ich implementáciu. Nevýhodou môže byť, že takýmto spôsobom sa do produktu pridávajú naviac funkcie, ktoré môže byť neskôr problém zrealizovať. S prototypmi súvisia aj ďalšie problémy: skoro viditeľné výsledky môžu spôsobiť unáhlenie manažérov pri odhadoch napredovania projektu [Gordon95].

Pri prototypovaní sa používajú prototypy na zahodenie, ktoré sa môžu vytvárať v generátoroch prototypov, ale aj evolučné prototypy pri menších projektoch. Pri evolučných prototypoch sa do produktu inkrementálne pridáva ďalšia funkcionality. Základnou nevýhodou takéhoto prístupu je, že nikdy neexistuje ucelený návrh kompletného

systému a tak sa ťažšie zabezpečuje kvalita výsledku. Z tohoto dôvodu je nevyhnutné vytvárať čo najflexibilnejší návrh, aby bolo možné pružne zapracovávať ďalšie požiadavky používateľov a odstraňovať nepotrebné moduly.

Pri plánovaní sa taktiež zohľadňuje tento inkrementálny proces vývoja a definuje sa viacero míľnikov (spravidla 3 až 4), ktoré umožňujú lepšie sledovať postup prác na projekte.

### Projektový tím

Oproti klasickým softvérovým projektom závisí výsledok RAD projektov oveľa významnejšie od konkrétnych ľudí, ktorí na projekte pracujú. Na RAD projektoch pracuje zväčša menší počet vývojárov (3 až 6), ktorí často zastávajú viacero úloh. Veľmi dôležité sú ich skúsenosti s používaním RAD nástrojov, komunikáciou s používateľmi a schopnosť prijímať dôležité rozhodnutia. Aj tu platí, že do týchto ľudí sa oplatí investovať, pretože predstavujú najhodnotnejší zdroj organizácie.

Vývoj informačných produktov pre potreby dnešného dynamicky sa rozvíjajúceho trhu predstavuje výzvu pre veľké množstvo spoločností, ktoré vyvíjajú softvér v nestabilnom prostredí s meniacimi sa požiadavkami. V tomto prostredí sa skúmajú iné prístupy k vývoju softvéru a s existenciou nových podporných nástrojov sa stávajú RAD techniky veľmi úspešnou stratégiou. Tieto techniky predstavujú súbor odporúčaní pre odľahčenie klasických metodológií, ktoré sa sústredia na zabezpečenie kvality výsledkov softvérových procesov. Klasické metodológie často nie sú použiteľné v menších a stredne veľkých softvérových tímoch, ktoré navyše zápasia s nedostatkom času, a preto mnoho tímov hľadá nejakú reálnejšiu alternatívu pre proces vývoja softvéru. RAD techniky neposkytujú jednoznačný recept na úspech, ale sú skôr zmesou odporúčaní, ktoré môžu ukázať správny smer, ak sa vhodne prispôbia špecifikám konkrétnej aplikačnej domény.

### Použitá literatúra

- [Agarwal00] AGARWAL, R.: *Risks of Rapid Application Development*. Communications of the ACM, November 2000, s. 177-188.
- [Bieliková00] BIELIKOVÁ, M.: *Softvérové inžinierstvo - Princípy a manažment*. Bratislava: STU, 2000. ISBN 80-227-1322-8.
- [Boehm99] BOEHM, B.: *Making RAD Work for Your Project*. IEEE Computer, Marec 1999, s. 113- 117.
- [Card95] CARD, D.N.: *The RAD Fad: Is Timing Really Everything?* IEEE Software, September 1995, s. 19-22.

- [Cross] CROSS, S.E.: *Toward Disciplined Rapid Application Development*. Software Tech News, vol. 2, no. 1. <http://www.dacs.dtic.mil/awareness/newsletters/technews2-1/disciplined.html>. (27.03.2001).
- [Gordon95] GORDON, V.S., BIEMAN, J.M.: *Rapid Prototyping: Lessons Learned*. IEEE Software, Január 1995, s. 85-95.
- [Kaghazian00] KAGHAZIAN, K.: *Dynamic Process of Internet Companies: An Abstract Model*. 2000. <http://citeseer.nj.nec.com/334422.html>. (14.03.2001).
- [Olsen95] OLSEN, N.C.: *Survival of the Fastest: Improving Service Velocity*. IEEE Software, September 1995, s. 28-38.

## **DIEL II.**

### **Meranie v softvérovom inžinierstve**

ÚVOD.....	85
<b>MERANIE A ŽIVOTNÝ CYKLUS SOFTVÉRU</b>	
METRIKY V ŠTÁDIU ANALÝZY .....	91
MERANIE V ETAPE NÁVRHU.....	99
MERANIE V ETAPE IMPLEMENTÁCIE .....	111
MERANIE A PARADIGMY PROGRAMOVANIA.....	119
MERANIE PRI TESTOVANÍ, PREVÁDZKE A ÚDRŽBE SOFTVÉROVÝCH SYSTÉMOV .....	127
<b>MERANIE A MANAŽMENT, MERANIE PROCESU</b>	
MERANIE VYSPELOSTI SOFTVÉROVÉHO PROCESU .....	137
MERANIE PRODUKTIVITY .....	149
MERANIE SOFTVÉRU ZÁKAZNÍKOM (MERANIE PRODUKTU) .....	159



## Úvod

Priznám sa, že keď som prvýkrát počul o meraní v softvérovom inžinierstve, nevedel som, čo si pod týmto pojmom predstaviť. Meranie bolo pre mňa spojené len s „hmotnými“, „hmatateľnými“ objektami. Meranie sa však chápe ako proces určenia a vyjadrenia vlastností objektov (číslami, resp. symbolmi), pričom sa nehovorí, že sa musí jednať o „hmotné“ objekty. Aj program a ďalšie produkty životného cyklu softvéru, ba aj samotný proces vývoja softvéru majú atribúty, ktoré možno určovať, merať.

Meranie má nezastupiteľnú úlohu vo všetkých vedeckých a inžinierskych disciplínach. Meraním sa zistia potrebné vlastnosti, ktoré možno využiť v rôznych matematických modeloch.

V softvérovom inžinierstve sa bez merania takisto nezaobídeme.. Aký čas nám potrvá vývoj softvérového systému, ktorý zákazník chce? Koľko nás to bude stáť? Je návrh dobrý, nakoľko bude výsledný systém zložitý? Aký zložitý je daný program? Ktoré časti systému budú najkritickejšie a teda bude potrebné venovať im najväčšiu námahu pri testovaní? Aká námaha bude potrebná na testovanie? Nakoľko je systém udržiavateľný? Aká je produktivita jednotlivých pracovníkov? Nakoľko efektívny je náš softvérový proces? Odpovede na tieto otázky hľadá každá firma a nájdenie čo najpresnejších odpovedí je často kľúčové pre jej úspech. Možno pritom využiť práve merania, procesom merania odmerať vybrané vlastnosti a na základe určitých modelov z nich vyvodiť závery. Merať

Meranie softvéru nie je nová disciplína, ktorá „spadla z neba“. O meraní softvéru možno hovoriť už v súvislosti s určovaním časovej a priestorovej zložitosti algoritmov. V polovici päťdesiatych rokov sú prvé pokusy merať prácnosť jednotlivých štádií softvérového projektu. V r.1972 Halstead vydal prácu, ktorá položila základ tzv. softvérovej vedy. Tá skúma zápis programu, získava (nameriava) z neho niektoré údaje a snaží sa nájsť rôzne modely, za účelom určovania napr. optimálnej modularity, predpovedania počtu chýb pred testovaním, testovania plagiátorstva a ďalších. Ďalšie práce sa venovali využitiu merania na produktoch skorších fáz životného cyklu. Začiatkom 80. rokov už možno o meraní hovoriť ako o samostatnej disciplíne. V súčasnosti pokračuje vývoj v tejto oblasti a meraním sa zaoberá viacero inštitútov (napríklad, jedna z najuznávanejších organizácií Software Engineering Institute na univerzite Carnegie-Mellon má oblasť úloh meranie ako jeden z okruhov výskumných úloh).

možno nielen zápis programu, ale aj ďalšie produkty životného cyklu, ba dokonca aj samotný proces vývoja softvéru.

Z hľadiska **procesu vývoja softvéru**, meranie pomáha odhadnúť *časové termíny, náklady na projekt a kvalitu* výsledného produktu. Pretože tieto sú v svojej podstate protikladné (a zlepšovanie dvoch vedie k zhoršovaniu tretej), treba skúmať všetky tri. Odhad nákladov vedie k potrebe určiť celkovú prácnosť potrebnú na projekt. To si žiada vedieť určiť aj *produktivitu zdrojov* – pričom rozhodujúcu zložku nákladov softvérových produktov tvoria ľudia. Produktivita sa chápe ako podiel „množstva“ vyprodukovaného softvéru a spotrebovaných zdrojov. Známu mierou spotrebovaných zdrojov sú „človekomesiace“ (angl. person month), miery výstupu môžu byť napr. počet riadkov vyprodukovaného zdrojového kódu alebo napr. počet funkčných bodov (odhadneme počet a zložitosť funkcií, ktoré má výsledný softvér mať). Existuje niekoľko modelov, ktoré možno použiť na odhad nákladov.

Merat' možno aj *vyspelosť procesu* – napr. model vyspelosti SEI definuje 5 stupňov, na základe ktorých možno ohodnotiť vývojový proces danej organizácie. Na základe otázok, na ktoré odpovíme áno/nie (napr. „zbierajú sa štatistiky chýb v návrhu softvéru?) je vyspelosť softvérového procesu ohodnotená jedným z 5 stupňov. Na základe tejto miery potom získame predstavu, na akom stupni je náš proces momentálne a môže naplánovať, ako dosiahnuť ďalší stupeň. Treba zdôrazniť, že podľa štatistík je väčšina softvérových firiem len na prvých dvoch najnižších stupňoch a prechod na ďalší stupeň trvá spravidla dva roky.

Merat' tiež môžeme s motiváciou určiť prínosy rôznych techník, metód i nástrojov. Aké boli ich prínosy na výkonnosť tvorby softvéru či jeho údržbu? Tieto údaje sú dôležité pre rozhodovanie vedenia, či tieto prostriedky používať

**Životný cyklus softvéru** obsahuje nasledovné fázy: *analýzu a špecifikáciu požiadaviek, návrh, implementáciu, testovanie a údržbu*. V každej z nich možno využiť meranie jej produktov a ovplyvniť tak či už nasledovné fázy (rôzne odhady), alebo ako sme už spomenuli, vyhodnotiť prínosy určitých prostriedkov či použiť namerané údaje pri zhodnotení vývojového procesu.

Produktom etapy *analýzy a špecifikácie požiadaviek* je dokument obsahujúci špecifikáciu zákazníkových požiadaviek na systém. Z neho môžeme vyvodíť už spomínané funkčné body. Predstavujú predpovedný model, ktorý možno využiť najmä na odhad prácnosti. Model obsahuje niekoľko konštánt, ktoré autor (Albrecht) určil empiricky vo firme IBM. Pokiaľ na predpovedanie využijeme aj archív štatistík o starších projektoch firmy, môžeme dostať pomerne presné odhady (najmä ak firma už realizovala podobný projekt). Existujú aj ďalšie miery a predpovedné modely pre odhad prácnosti, z analytického modelu možno odvodiť (resp. odmerať) aj ďalšie atribúty.

Produktom etapy *návrhu* je popis architektúry systému, rozdelenie na jednotlivé podsystémy, dekompozícia na jednotlivé moduly, popis ich vzájomnej komunikácie a algoritmy pre jednotlivé moduly. Vlastnosťami návrhu je napr. súdržnosť (cohesion) a previazanosť (coupling). Najčastejšie uvádzaným pravidlom návrhu je *maximálna súdržnosť* a *minimálna previazanosť*. Ďalej, objektovo-orientovaný návrh má oproti štruktúrovanému niektoré špecifiká (odlišná paradigma), preto preň existujú osobitné miery. Možno tiež skúmať napr. návrh používateľského rozhrania (a odvodiť „zložitosť“ práce s ním pre užívateľa), pre navrhnuté algoritmy možno skúmať ich efektívnosť (časová a priestorová zložitosť). Okrem posúdenia kvality návrhu je možné identifikovať slabé miesta systému. Týmto potom môžeme prideliť väčšie množstvo prostriedkov pri testovaní/údržbe.

Produktom etapy *implementácie* je zdrojový kód. Existuje množstvo metrík, napr. počet riadkov, počet symbolov, cyklomatická zložitosť (počet ciest v programe, ktorými môže „prebiehať“ riadenie) a iné. Možno tiež napríklad odmerať súdržnosť, presnejšie ako v etape návrhu. Získané dáta môžu pomôcť pri testovaní a údržbe.

Meranie má význam aj pri *testovaní* a *údržbe*. Až v týchto etapách je možné otestovať spoľahlivosť. Pri testovaní sa hľadajú chyby, posudzuje sa kvalita výsledného produktu. Vážne chyby treba odstrániť, menej závažné sa často v softvéri ponechajú, napr. z dôvodu, aby sa nezhoršila vnútorná štruktúra či nezanesli nové chyby. Možno hovoriť o udržiavateľnosti modulu, na čo existuje viacero typov modelov ohodnocujúcich udržiavateľnosť. Jednou z nich je napríklad model HPMAS, vyvinutý v spoločnosti Hewlett-Packard, ktorý sleduje riadiacu štruktúru, informačnú štruktúru (voľba dátových štruktúr a techník tokov dát), v zápise programu ohodnocuje typografiu, pomenovanie a komentáre. Meranie má význam aj počas prevádzky, keď sa meria počet zlyhaní a porúch, čím možno určiť spoľahlivosť. Spoľahlivosť možno odhadnúť aj na základe niektorých modelov využívajúcich namerané výskyt chýb počas testovania. Zároveň je možné modelmi popísať zmenu spoľahlivosti v čase (obyčajne rast).

Uviedli sme príklady využitia meraní pri rôznych činnostiach softvérového inžinierstva. Zďaleka sa nejedná o úplný zoznam, našou snahou bolo len vymenovať niekoľko jednoduchých príkladov, na ktorých si čitateľ môže utvoriť základnú predstavu o meraní a jeho aplikáciách v softvéri inžinierstve. Podrobnejšie sa uvedenými témami budeme zaoberať v našej publikácii, kde mienime podať základný prehľad z o princípoch, používaných technikách a príkladoch využitia merania v praxi.

Meranie je pomerne mladou oblasťou, v ktorej neustále prebieha výskum. Treba povedať, že s meraním je spojených niekoľko ťažkostí, pre ktoré niektorí autori meranie kritizujú:

- meranie je v niektorých prípadoch subjektívne, záleží od „osobného úsudku“ merajúceho. Niektoré pojmy totiž ani nie sú exaktne

definované (napr. „zložitosť“, „udržiavateľnosť“) vzťahom nad objektívne merateľnými veličinami.

- meranie nedáva presné výsledky, tieto môžu byť až zavádzajúce. Model od Hewlett-Packard pre udržiavateľnosť bol kalibrovaný pomocou subjektívnych ohodnotení zamestnancov tejto spoločnosti a výsledný vzťah je:

$$\text{Udržiavateľnosť} = 171 - 5,2\ln(A) - 0,23(B) - 16,2\ln(C) + 50\sin(\sqrt{2,46(D)})$$
, kde

A,B,C,D sú miery, ktoré vieme namerať zo zápisu zdrojového kódu (napr. C udáva priemerný počet riadkov a D je percentuálny podiel komentárov v zdrojovom kóde.

Samozrejme, udržiavateľnosť ako taká je oveľa komplexnejší pojem a závisí od viacerých faktorov, než je v uvedenom vzťahu premenných. Taktiež, konštanty v tomto vzťahu boli určované subjektívne a pre pracovné prostredie firmy Hewlett-Packard. Preto pokiaľ by sme chceli aplikovať tento vzťah v našej firme priamočiaro, bezvýhradne mu dôverovať, mohli by sme zistiť, že jeho výsledky sú nepresné a nepoužiteľné.

Z uvedených dôvodov niektorí autori meranie „zatracujú“. S tým však nemožno súhlasiť. Od mier a modelov nemožno očakávať, že nám budú plniť úlohu „krištáľovej gule“, dávať presné a korektné odpovede (a od čoho možno?). Úlohou mier je len pomôcť nám pri rozhodovaní a v zmysle pomôcok sa aj používajú v mnohých firmách. Uvedený príklad pre určenie udržiavateľnosti sa používa v spoločnosti Hewlett-Packard. Ďalšie miery sa s úspechom použili a používajú v iných firmách, kde pomohli odhadovať náklady na projekt, ušetriť prostriedky či odpovedať na ďalšie otázky ktoré sme načrtli (na konci publikácie uvedieme príklady, aké skúsenosti mali s mierami konkrétne podniky). Čo sa týka subjektívnosti meraní, treba si uvedomiť problematiku a zložitosť exaktne definovať niektoré pojmy. Uvedené námietky teda nie sú „dôvodom“ pre zatratenie mier, ale skôr námetom, výzvou pre ďalší výskum v tejto oblasti.

V našej publikácii sa zameriavame najmä na meranie produktov jednotlivých etáp životného cyklu. Úvodné príspevky sú venované meraniu v etape analýzy (Marián Šimo), návrhu (Stanislav Hrk), implementácie (Vladimír Trgo) a meraním v etapách testovania, prevádzky a údržby (Szabolcs Molnár). Samostatný príspevok (Michal Šrámka) venujeme meraniu pri ďalších paradigmách tvorby softvéru, najmä objektovo-orientovanej paradigme. V publikácii sa dotýkame aj merania samotného procesu. Zaoberáme sa meraním vyspelosti procesu (Radoslav Kováč), samostatný príspevok venujeme meraniu významnej charakteristiky procesu tvorby softvéru – produktivite (Ján Pidych). V publikácii uvádzame príklady použitia metrík. Na záver sme začlenili príspevok tak trochu „z iného súdka“, v ktorom sa dotýkame tematiky

merania softvéru z pohľadu zákazníka (Peter Agh). Zaoberáme sa v ňom najmä meraním softvéru s cieľom určiť jeho prínosy pre zákazníka.

Z priestorových i časových dôvodov sme do publikácie nezaradili všetko čo s tematikou merania softvéru súvisí. Sústredili sme sa na podanie základného prehľadu tejto oblasti, pričom sme uviedli odkazy na zdroje s ďalšími informáciami. Dúfame, že naša publikácia poslúži ako úvodný materiál pre štúdium tejto problematiky, čím náš cieľ bude naplnený.

Peter Agh (editor)



## Metriky v štádiu analýzy

Marián Šimo

**Abstrakt.** Príspevok sa zaoberá dvoma metrikami, ktoré sú používané v štádiu analýzy. Podrobne vysvetľuje metriku funkčných bodov a metriku Feature Points, ktorá vychádza z metriky funkčných bodov (Function points). Pre názorné vysvetlenie metrik, príspevok obsahuje aj jednoduchý ukázkový príklad.

Analýza je počiatočná fáza projektu. Pomocou analýzy problému získavame celkový prehľad o riešenom probléme. Na základe analýzy v neskorších fázach projektu navrhujeme a realizujeme produkt.

Je dôležité v tejto fáze získať aj prehľad o množstve práce, ktorú je treba vynaložiť na úspešné dokončenie projektu. Takisto v tomto štádiu je vhodné približne odhadnúť aký veľký bude výsledný softvérový systém.

Metriky nám poskytujú návod, ktorým tieto odhady môžeme realizovať.

### Funkčné body

Metrika vznikla vo firme IBM. Autorom tejto metriky je A.J. Albrecht. V októbri 1979 bola táto technika prvýkrát predstavená na SHARE/GUIDE/IBM konferencii v Monterey, California. Dovtedajší spôsob merania softvéru neodrážal presne produktivitu, pretože vychádzal z počtu riadkov programu, ktorý závisí od použitého programovacieho jazyku.

Externé aspekty softvéru môžu byť ohodnotené na základe týchto piatich parametrov:

- počet logicky rôznych vstupov
- počet výstupov
- počet dopytov
- počet logických dátových súborov
- počet rozhraní

Metódou pokusu a omylu boli určené váhové faktory týchto piatich parametrov. Tieto váhy reprezentujú približnú náročnosť implementácie príslušných bodov.

**Tab. 1:** Váhové ohodnotenie parametrov softvéru.

Parameter	Nízka zložitosť	Stredná zložitosť	Vysoká zložitosť
Vstup	×3	×4	×6
Výstup	×4	×5	×7
Dopyt	×3	×4	×6
Dátový súbor	×7	×10	×15
Rozhranie	×5	×7	×10

Sčítaním jednotlivých váh parametrov získame celkový počet neupravených funkčných bodov. Pre realistickejšie ohodnotenie funkčných bodov konkrétneho projektu sa zaviedol mechanizmus úpravy funkčných bodov. Táto úprava spočíva v zohľadnení charakteristiky systému a výpočte koeficientu úpravy funkčných bodov (CAF).

Charakteristika systému sa určí ohodnotením 14 charakteristík systému bodmi v rozsahu 0 až 5 v závislosti od ich vplyvu na systém. Tieto charakteristiky sú nasledovné:

**Tab. 2:** Charakteristiky systému.

$v_i$	Charakteristika
1.	Dátová komunikácia
2.	Distribúované spracovanie dát
3.	Výkonnosť
4.	Konfigurovateľnosť
5.	Zložitosť transakcií
6.	On-line vstup údajov
7.	Efektívnosť
8.	On-line modifikácia údajov
9.	Zložitosť spracovania
10.	Znovupoužiteľnosť
11.	Jednoduchosť inštalácie
12.	Jednoduchosť obsluhy
13.	Viac inštalácií (viac organizácií)
14.	Budúce zmeny (rozšírenie)

Celková charakteristika systému sa určí súčtom bodov jednotlivých charakteristík (hodnota z rozsahu 0 až 70). Výsledný koeficient úpravy funkčných bodov sa vypočíta podľa vzorca:

$$CAF = 0,65 + \left( 0,01 \times \sum_{i=1}^{14} v_i \right)$$

Výsledný počet upravených funkčných bodov získame vynásobením počtu neupravených funkčných bodov vypočítaným koeficientom CAF.

Ukážme si túto techniku na konkrétnom príklade. V nasledujúcej tabuľke sú vyjadrené jednotlivé charakteristiky vývoja produktu v dvoch rozdielnych programovacích jazykoch. Použitá je metrika riadky programu.

**Tab. 3:** Paradox metriky riadkov programu.

Činnosť	Prípade A Assembler (10000 riadkov)	Prípade B Fortran (3000 riadkov)	Rozdiel
Požiadavky	2 mesiace	2 mesiace	0
Návrh	3 mesiace	3 mesiace	0
Kódovanie	10 mesiacov	3 mesiace	7
Integrácia/Testovanie	5 mesiacov	3 mesiace	2
Používateľská dokumentácia	2 mesiace	2 mesiace	0
Manažment/Podpora	3 mesiace	2 mesiace	1
Celkovo	25 mesiacov	15 mesiacov	10
Celkové náklady	125 000\$	75 000\$	50 000\$
Náklady na jeden riadok programu	12,50\$	25,00\$	-12,50\$
Riadky / 1 človekomesiac	400	200	200

Použitím tejto techniky dochádza k paradoxu, kde náklady na jeden riadok programu sú dvojnásobné u jazyka Fortran v porovnaní s jazykom Assembler, hoci celková cena je nižšia. Toto môže viesť k mylnému záveru, že použitie jazyka Fortran je menej efektívne ako použitie jazyka Assembler. Pri porovnaní oboch stĺpcov však zistíme, že v prípade A až 40 percent nákladov boli náklady na kódovanie programu, v prípade B to bolo len 20 percent. Ostatné náklady sú fixné.

Keď použijeme techniku funkčných bodov, musíme najskôr vyjadriť celkový počet funkčných bodov produktu. Majme systém, ktorý má strednú zložitosť vstupu, výstupu, dopytov, rozhrania a nízku zložitosť dátového súboru. Nech celková charakteristika systému je 30. Výpočet funkčných bodov bude nasledovný:

**Tab. 4:** Ukážková kalkulácia funkčných bodov.

Údaje	Váhy	Funkčné body
1 vstup	×4 =	4
1 výstup	×5 =	5
1 dopyt	×4 =	4
1 dátový súbor	×7 =	7
1 rozhranie	×7 =	7
Spolu neupravené		27
Úprava (CAF)		0,95
Upravené funkčné body		26

**Tab. 5:** Príklad metriky Funkčných bodov.

Činnosť	Prípad A Assembler (30 F.B.)	Prípad B Fortran (30 F.B.)	Rozdiel
Požiadavky	2 mesiace	2 mesiace	0
Návrh	3 mesiace	3 mesiace	0
Kódovanie	10 mesiacov	3 mesiace	7
Integrácia/Testovanie	5 mesiacov	3 mesiace	2
Používateľská dokumentácia	2 mesiace	2 mesiace	0
Manažment/Podpora	3 mesiace	2 mesiace	1
Celkovo	25 mesiacov	15 mesiacov	10
Celkové náklady	125 000\$	75 000\$	50 000\$
Náklady na jeden funkčný bod	4 166,67\$	2 500\$	1666,67\$
Funkčných bodov / 1 človekomesiac	1,2	2,0	-0,8

Pri vyjadrení predchádzajúceho príkladu pomocou metriky funkčných bodov náklady na jeden funkčný bod odzrkadľujú úsporu ak použijeme jazyk Fortran miesto Assembleru.

Metrika funkčných bodov poskytuje normalizované údaje o produktivite. Keď reálne náklady klesajú, taktisto aj náklady na jeden funkčný bod sa znižia. Ak reálne náklady stúpnu, stúpnu aj náklady na jeden funkčný bod.

Metrika funkčných bodov poskytuje softvérovým inžinierom nástroj na odhad veľkosti softvéru na základe analýzy implementovaných funkcií systému z pohľadu používateľa. Umožňuje predpovedať počet príkazov, ktoré musia byť napísané v programe alebo systéme.

Programovacie jazyky majú rôzne, ale za to charakteristické *úrovně*. Úroveň je priemerný počet príkazov potrebných na implementáciu jedného funkčného bodu.

Úroveň programovacieho jazyka poskytuje:

- možnosť predpovedať veľkosť softvérového projektu alebo počtu príkazov, ktoré budú potrebné, čo najskôr (už počas tvorby požiadaviek a návrhu systému)
- možnosť spätného určenia funkčných bodov už existujúceho softvéru bez namáhavého ručného určovania jednotlivých funkčných bodov
- možnosť jednoduchého prevodu veľkosti aplikácie napísanej v jednom programovacom jazyku (počet riadkov zdrojového kódu) na ekvivalentnú veľkosť, ak by bola aplikácia naprogramovaná v druhom programovacom jazyku
- možnosť odmerania produktivity projektov, ktoré sú programované vo viacerých programovacích jazykoch.

V niektorých programovacích jazykoch sú údaje zoskupené (napr. v objektoch, v štruktúrach a pod). Pre tieto jazyky odhadovanie veľkosti pomocou metriky Funkčných bodov poskytuje veľmi presné výsledky. Pre iné jazyky je metrika Funkčných bodov menej presná. Napríklad použitie metriky Funkčný bodov na systém programovaný jazykom COBOL neposkytuje presné výsledky, pretože rozptyl odhadu je až  $\pm 50\%$ .

### Feature Points

V roku 1986 firma Software Productivity Research, Inc. vyvinula experimentálnu metódu aplikovania metriky Funkčných bodov na systémový softvér ako napr. operačné systémy, systémy telefónnych ústrední, a pod. Pre vyhnutie sa právnych problémov s firmou IBM (jej metrika Funkčných bodov) bola nová metóda pomenovaná Feature Points.

Metrika Funkčných bodov bola pôvodne vyvinutá pre riešenie problémov merania klasických podnikových informačných systémov, preto nie je táto metrika optimálna pre použitie na meranie:

- systémov reálneho času
- systémového softvéru (operačné systémy)
- vnorené systémy
- komunikačný softvér
- riadiaci softvér technologických procesov
- ďalšie systémy (CAD, CIM, diskrétné simulácie, matematický softvér, a pod.)

Pri aplikovaní metriky Funkčných bodov na tieto systémy dostaneme skreslené výsledky, pretože tieto systémy majú veľmi veľkú vnútornú algoritmickú zložitosť. Preto meranie takýchto systémov vyžaduje

podobnú metódu ako je metrika Funkčných bodov, ale musí viac zohľadniť algoritmickejšiu zložitosť.

Metrika Feature points vychádza z metriky Funkčných bodov a definuje ďalší parameter *algoritmus* k existujúcim piatim parametrom z metriky Funkčných bodov. Metrika upravuje aj empirické rozloženie váh jednotlivých parametrov. Váha počtu logických dátových súborov je znížená z 10 na 7 voči metrike Funkčných bodov. Toto odráža menšiu významnosť dátových súborov v systémovej softvére ako v informačných systémoch. Váha nového parametra *algoritmus* bola stanovená na 3 pre systém priemernej zložitosti.

Samozrejme v systémoch kde je rovnaký pomer vnútorného spracovania<sup>1</sup> a logických dátových súborov obe metriky (Funkčné body a feature points) generujú rovnaké celkové počty bodov. Ak v systéme je viac vnútorného spracovania ako práce s dátovými súbormi (čo je vlastnosť systémovej softvére), metrika feature points generuje vyššie množstvo celkových bodov. Opačne, keď je pomer vnútorného spracovania voči dátovým súborom menší, metrika feature points bude generovať menší počet bodov ako metrika Funkčných bodov.

**Tab. 6:** Váhy parametrov metriky Feature points pre stredne zložitý systém.

Parameter	Váha
Algoritmus	×3
Vstup	×4
Výstup	×5
Dopyt	×4
Dátový súbor	×7
Rozhranie	×7

### Počítanie a určenie váhy algoritmu

Algoritmus definovaný štandardom softvérového inžinierstva znie:

„Algoritmus je usporiadaná množina pravidiel, ktoré musia byť popísané a implementované pre vyriešenie výpočtového problému“.

V ponímaní metriky Feature Points je algoritmus zadefinovaný nasledovne:

„Algoritmus je určitý výpočtový problém, ktorý je riešený príslušným počítačovým programom“

Algoritmy sa líšia medzi sebou zložitostou, preto zložitosť algoritmu je ohodnotená rozsahom hodnôt z intervalu 1 až 10. Zložitosť algoritmu obsahujúceho iba základné matematické operácie alebo len niekoľko jednoduchých pravidiel je ohodnotená minimálnou váhou 1. Algoritmus

---

<sup>1</sup> Pod pojmom vnútorné spracovanie sa myslí napríklad čakanie v slučke, počítanie a pod.

obsahujúci komplexné rovnice, maticové operácie a zložité matematické alebo logické operácie môže byť ohodnotený váhou 10. Váha pre bežné (stredne zložité) algoritmy používajúce bežnú matematiku môže byť empiricky určená na 3. Pretože vývoj tejto metriky nie je ešte ukončený, proces určovania váh je ešte stále vo vývoji.

### Čo si vybrať?

Pre aplikácie, v ktorých je algoritmus nie je dominantný faktor alebo algoritmus nie je zložitý, je vhodnejšie použiť metriku Funkčných bodov. Mnoho kancelárskych aplikácií patrí do tejto triedy aplikácií (napr. účtovnícky softvér, zákaznicke informačné systémy, podporné nástroje manažérov a pod.).

Pre aplikácie, ktorých zložitosť je známa alebo sa dá pomerne presne odhadnúť, u ktorých prevláda vnútorné spracovanie, je určená metrika Feature points. Do tejto triedy patrí veľa inžierskych, vedeckých a systémových aplikácií (systémy telefónnych ústrední, riadiace systémy procesov, vnorené systémy a pod.).

**Tab. 7:** Pomer bodov metrík.

Typ aplikácie	Funkčné body	Feature points
Dávková aplikácia IS	1	0,80
Interaktívna aplikácia IS	1	1,00
Databázova aplikácia	1	1,00
PBX prepájací systém	1	1,20
Riadenie procesu	1	1,28
Vnorená real time aplikácia	1	1,35
Automatizovaná výrobná linka	1	1,50

### Použitá literatúra

- [Capers96] CAPERS, J.: *Programming Languages Table*. Software Productivity Research, Inc., Marec 1996.  
<http://www.spr.com/library/olangtbl.htm>.
- [Capers97] CAPERS, J.: *What are Function Points*. Software Productivity Research, Inc., 1997.  
<http://www.spr.com/library/ofuncmet.htm>.
- [Garmus01] GARMUS, D. – HERRON, D.: *Function Points and Their Use in Information Technology*. Január 2001.  
<http://www.cobolreport.com/columnists/2davids/01152001.htm>.



## Meranie v etape návrhu

Stanislav Hrk

**Abstrakt.** Úvodné etapy vývoja softvéru, medzi ktoré patrí aj etapa návrhu, sú veľmi významné pre ďalší priebeh procesu vývoja. Predpokladom produkovania kvalitnejšieho softvéru je zlepšenie výstupov zo skorších etáp. Aplikovanie relevantných metrík na výstupy z týchto etáp umožňuje vylepšiť manažment v neskorších etapách, ako aj efektívnejšie a exaktnejšie odhadnúť kvalitu výsledného softvérového produktu v čase, keď je ešte pomerne ľahko modifikovateľný preventívnymi alebo nápravnými opatreniami. Tento príspevok sa zaoberá meraním v etape návrhu. Rozoberajú sa dôvody a ciele merania v tejto etape, atribúty návrhu, ktoré sa snažíme kvantitatívne vyjadriť, ako aj fázy návrhu, v ktorých sa meranie môže vykonávať.

Návrh systému je jednou z najcitlivejších etáp jeho vývoja. Rozhodnutia týkajúce sa architektonického návrhu, návrhu modulov, údajov a rozhrania systému do veľkej miery ovplyvňujú priebeh nasledujúcich etáp vývoja a kvality výsledného produktu. Chyby v návrhu odhalené v neskorších etapách predstavujú závažné problémy, ktorých odstránenie si vyžaduje rozsiahle prehodnocovanie a zmeny vo vytvorených častiach systému. Týmto sa stráca čas, zvyšujú sa náklady vývoja a znižuje sa celková kvalita výsledného produktu. Preto je pri rozhodovaní dôležité mať včas informácie o tom, aký dopad budú mať vykonané rozhodnutia na vyvíjaný systém.

### Ciele merania

Cieľom merania vo fáze návrhu je získať informácie, pomocou ktorých sa môžu odhadnúť charakteristiky budúceho systému. Vychádza sa pri tom z predpokladu, že štruktúra vytvoreného kódu zodpovedá štruktúre vytvorenej v etape návrhu. Táto vlastnosť umožňuje používať výsledky merania v etape návrhu na predpovedanie vlastností produktu v ďalších etapách vývoja. Problémom merania návrhu je, že veľká časť výstupov z tejto etapy nie je dostatočne formalizovaná, čím sa sťažuje objektívne

meranie a vyhodnocovanie výsledkov merania. Nárastom používania štandardizovaných metodík návrhu a CASE prostriedkov sa situácia na tomto poli postupne zlepšuje.

Včasnú uplatnenie merania je kľúčovým faktorom pre úspešný manažment vývoja softvéru. Informácie získané z výsledkov merania v etape návrhu sa používajú na:

- Včasnú identifikáciu chýb vo výstupoch z fáz špecifikácie a návrhu, čím sa znížia náklady na odstránenie týchto chýb zo systému.
- Pri výskyte alternatívnych riešení vo fáze návrhu výsledky merania pomáhajú zhodnotiť, ktoré riešenie je vhodnejšie.
- Identifikáciu zložitých modulov a modulov, ktoré sú náchylné k chybám. Modulom, ktoré sú identifikované ako problematické, sa pridelia dodatočné zdroje a skúsenejší programátori, s cieľom vyhnúť sa oneskoreniam a iným problémom v etapách implementácie, testovania a údržby daného modulu.
- Sledovanie kvality softvérového produktu a procesu vylepšené tým, že sa so sledovaním kvality začne v skorších etapách životného cyklu.
- Odhad miery možnosti znovupoužitia modulov v počiatočných etapách ich vývoja.

### Čo meriame

Pri meraní v rôznych etapách životného cyklu softvéru sa sledujú vybrané atribúty výstupu, ktorý daná etapa produkuje. Niektoré atribúty sú spoločné pre viacero etáp životného cyklu. Tak napríklad je možné uvažovať atribúty ako sú *zložitosť* alebo *veľkosť systému* pri špecifikácii, návrhu alebo v zdrojovom kóde, alebo *súdržnosť* a *zviazanosť* modulov pri návrhu a v zdrojovom kóde. Kvalita výstupu z danej etapy životného cyklu sa posudzuje podľa hodnôt získaných meraním vybraných atribútov. Napríklad všeobecne zaužívané pravidlo „dobrého návrhu“ je snaha o *minimálnu zviazanosť* (miera previazanosti elementov patriacich rozličným modulom) a *maximálnu súdržnosť* (miera previazanosti elementov v rámci jedného modulu). Je možné, že tieto dva koncepty sa medzi sebou budú vylučovať, a preto sa musí hľadať ich najlepší pomer z hľadiska kvality systému. Pritom sa nesmie zabúdať na iné kritériá kvality, akým je napríklad možnosť znovupoužitia modulov alebo mieru rozšíriteľnosti a modifikovateľnosti produktu.

Treba uviesť, že v literatúre existuje nejednoznačnosť v chápaní významu pojmov, ktorými sa označujú niektoré atribúty výstupov softvérového procesu významné pre definíciu softvérových metrik. V práci [Briand a kol., 1995] autori zaviedli dostatočne všeobecný model na popis vlastností charakterizujúcich jednotlivé atribúty (pomenované ako „koncepty“) softvérového systému. Model je založený na popise systému pomocou orientovaného grafu, v ktorom uzly predstavujú jednotlivé moduly systému na vybranej úrovni abstrakcie a hrany vyjadrujú vzťahy medzi nimi. Na základe tejto reprezentácie autori definovali axiómy pre

niektoré významné koncepty merania, ako sú *veľkosť*, *dĺžka*, *zložitosť*, *súdržnosť* a *zviazanosť*. Pomocou týchto axiém je možné overiť, či metrika spĺňa vlastnosti týkajúce sa konceptu, pre ktorý je navrhovaná.

Návrh systému sa vykonáva v troch na seba nadväzujúcich fázach [Salamon94]:

- *Predbežný návrh.* V tejto fáze sa príbuzné požiadavky zoskupujú do funkčných celkov a identifikujú sa závislosti medzi funkciami. Predbežný návrh môže byť reprezentovaný diagramami toku dát, štruktúrnymi diagramami na vysokej úrovni abstrakcie, alebo jednoduchým zoznamom požiadaviek kladených na podsystémy [Salamon94]. Výber metrik závisí od zvolenej reprezentácie systému. Napríklad pri reprezentácii pomocou diagramu toku údajov je možné merať zložitosť informačných tokov.
- *Podrobný návrh.* Pri podrobnom návrhu sa definuje celková architektúra softvérového systému. Funkcie a údaje sa pridelujú k jednotlivým častiam systému, definujú sa vnútorné rozhrania. Na reprezentáciu návrhu systému v tejto fáze sa najčastejšie používajú štruktúrne diagramy. Metriky, ktoré sa používajú v tejto fáze, sú napríklad *zložitosť informačných alebo dátových tokov*, *metrika externej zložitosti*, *zložitosť odvodená na základe reprezentácie pomocou grafov* a iné.
- *Návrh softvérových súčiastok.* V tejto fáze sa definujú algoritmy a údajové štruktúry, ako aj implementačné detaily. Návrh je reprezentovaný pseudokódom alebo popisom modulov v prirodzenom jazyku.

Okrem návrhu v týchto troch fázach je možné hovoriť aj o návrhu používateľského rozhrania ako oddelenej oblasti, ktorá ma veľký vplyv na konečné hodnotenie produktu používateľom. Pri návrhu používateľského rozhrania sa špecifikujú entity používateľského rozhrania (napríklad obrazovky) a ich vzájomné prepojenie. Reprezentácia návrhu používateľského rozhrania je viazaná na typ rozhrania, často sa však používajú diagramy používateľského rozhrania. Používanými metrikami sú napríklad *plytkosť rozhrania*, *kompaktnosť* a *navigovateľnosť smerom dole*, navrhnuté pre hypermediálne systémy. Tieto metriky zohľadňujú jednak kognitívnu záťaž na používateľa, jednak zložitosť štruktúry hypermediálneho obsahu.

## Niektoré metriky používané v etape návrhu

### Metriky pre zložitosť

Slovník IEEE [IEEGLLOSS] definuje zložitosť ako mieru toho, ako je návrh systému alebo komponentu ťažké porozumieť alebo preveriť.

**Zložitosť informačných tokov (information flow)**

V [Henry81] je vytvorený model pre počítanie zložitosti modulu, ktorý berie do úvahy veľkosť modulu a jeho údajové prepojenie s ostatnými časťami systému. Zložitosť systému je definovaná nasledovným vzťahom:

$$\text{Zložitosť} = \text{dĺžka} * (\text{Fan-In} * \text{Fan-Out})^2$$

Chápanie veličiny dĺžka, vystupujúcej vo vzťahu pre zložitosť podľa [Henry81], zodpovedá definícii veľkosti podľa [Briand a kol., 1995]. Ako hodnota tejto veličiny sa môže použiť niektorá z metrík pre veľkosť. Niekedy sa ako dĺžka používa McCabeho cyklomatická zložitosť (pozri ďalej) [Karkas98].

Fan-In a Fan-Out predstavujú počet informačných tokov z iných modulov do daného, resp. z daného modulu do iných. Pod informačným tokom sa myslí vzťah medzi parametrami/premennými daného a ostatných modulov.

Táto metrika bola overená na meraní systému typu UNIX. Záverom bolo, že jej použitím sa môžu v systéme identifikovať moduly náchylné k chybám. Zvýšená hodnota zložitosti informačných tokov sa vyskytla u modulov, s ktorými skutočne boli problémy a vyžiadali si väčší počet zmien.

**Cyklomatická zložitosť (cyclomatic complexity) (McCabe)**

Metrika cyklomatickej zložitosti vychádza z reprezentácii štruktúry systému pomocou orientovaného grafu. Založená je na meraní počtu lineárne nezávislých ciest v grafe. Cyklomatická zložitosť systému reprezentovaného grafom  $G$  sa počíta pomocou jednoduchého vzťahu:

$$v(G) = \text{počet hrán} - \text{počet uzlov} + 2p,$$

kde  $p$  je počet prepojených uzlov v grafe  $G$ .

Takto definovaná zložitosť nespĺňa axiómy pre zložitosť uvedené v [Briand a kol., 1995], ale upravená hodnota  $v(G)-p$  tieto predpoklady spĺňa. Upravená cyklomatická zložitosť nedáva výrazne odlišné výsledky od pôvodnej, najmä pri rozsiahlejších systémoch [Briand a kol., 1995].

Cyklomatickú zložitosť je možné použiť na získanie relatívnej zložitosti rozličných návrhov, čo je použiteľné pri rozhodovaní medzi alternatívami návrhu. Aplikovaním metriky je tiež možné ohodnotiť oblasti systému podľa toho, koľko námahy a zdrojov im je potrebné venovať v ďalších fázach vývoja. Prednosťami tejto metriky sú jej jednoduchosť a možnosť aplikácie v skorých fázach vývoja systému. Nedostatkom je, že sa skúma iba zložitosť riadiacich tokov a zložitosť dátových tokov sa zanedbáva.

**Metrika pre zložitosť podľa [Card90]**

Táto metrika je navrhnutá pre počítanie zložitosti jednotlivých modulov, z čoho sa odvodí celková zložitosť systému. Autori rozlišujú tri typy zložitosti modulu:

- *Štruktúrna zložitosť*

$$S(i) = Fan - Out(i)^2,$$

kde  $Fan - Out(i)$  je počet výstupných vetiev z modulu  $i$ .

- *Dátová zložitosť*

$$D(i) = \frac{v(i)}{Fan - Out(i) - 1},$$

kde  $v(i)$  je súčet počtu premenných vstupujúcich a vystupujúcich z modulu  $i$ .

- *Systémová zložitosť  $C(i)$*  je daná súčtom štruktúrnej a dátovej zložitosti modulu  $i$ .

$$C(i) = S(i) + D(i)$$

- *Celková zložitosť modulárneho systému* (overall project complexity)  $P$  je daná súčtom systémových zložitostí všetkých modulov v systéme.

$$P = \sum_i C(i)$$

Metrika pre zložitosť podľa [Card90] nespĺňa axiómy pre zložitosť, lebo pri štruktúrálnej zložitosti je hodnota  $Fan - Out$  umocnená na druhú [Briand a kol., 1995].

**Metriky pre zložitosť založené na teórii grafov**

Požítie tejto triedy metrick si vyžaduje, aby návrh systému bol reprezentovaný pomocou silne spojitého grafu (graf, pri ktorom existuje cesta medzi ľubovoľnými dvoma uzlami). Uzly v grafe reprezentujú moduly systému a hrany reprezentujú spojenia medzi modulmi. Na základe tejto reprezentácie boli navrhnuté tri metriky pre zložitosť:

- *Statická zložitosť*, definovaná pomocou vzťahu

$$C = E - N + 1,$$

kde  $E$  je počet hrán a  $N$  je počet uzlov v grafe.

- *Zovšeobecnená statická zložitosť*, ktorá okrem spojení medzi modulmi berie do úvahy aj používanie zdrojov systému.

$$C = \sum_{i=1}^E \left( C_i + \sum_{k=1}^K (d_k \times r_{ki}) \right),$$

kde  $K$  je počet zdrojov v systéme,  $d_k$  je zložitosť alokácie  $k$ -tého zdroja (napríklad zložitosť procedúry, pomocou ktorej sa alokuje daný

zdroj), a  $r_{ki}$  je rovné 1, ak si  $i$ -tá hrana grafu vyžaduje alokáciu  $k$ -tého zdroja, v opačnom prípade je rovné 0.

- *Dynamická zložitosť*, vzťahujúca sa na zmeny v počte hrán grafu reprezentujúceho architektúru systému zapríčinené dynamickým vykonávaním programu (napríklad prerušením vo vykonávaní niektorého z modulov sa niektoré hrany stanú nedostupné, vymažú sa). Počíta sa zo vzťahu pre statickú zložitosť počas určitého časového obdobia.

Metriky pre zložitosť založené na teórii grafov sú orientované na meranie architektonického návrhu systému. Používajú sa v počiatočných fázach vývoja systému na určovanie kompromisov v návrhu systému a pri odhadovaní účinku zmien v návrhu systému na parametre kvality systému, ako sú napríklad zrozumiteľnosť alebo udržiavateľnosť.

### Metriky pre zviazanosť

Zviazanosť je podľa slovníka pojmov IEEE definovaná ako miera závislosti medzi softvérovými modelmi a spôsob, akým medzi sebou závisia. V literatúre môžeme nájsť aj definície zviazanosti ako napríklad: miera previazanosti medzi komponentmi, alebo miera, do akej moduly v softvérovom systéme závisia jedny na druhých [Salamon94].

### Zviazanosť podľa Fenton and Melton

Táto metrika uvažuje zviazanosť ako atribút každej dvojice modulov. Opiera sa o existenciu empirickej relácie usporiadania zviazanosti medzi modulmi od najpevnejšej po najvoľnejšiu [Varga99]:

5. *obsahová zviazanosť* (content coupling) znamená, že A má vetvenie na konkrétny vnútorný príkaz v B, mení lokálne dáta alebo príkazy vnútri B
4. *globálna zviazanosť* (common coupling) existuje, ak A aj B používajú spoločné (globálne) dáta
3. *riadiaca zviazanosť* (control coupling) znamená odovzdávanie parametru-príznamu modulom A, ktorý riadi logiku správania modulu B
2. *typová zviazanosť* (stamp coupling) sa vyskytuje, ak moduly A a B pri vonkajšej komunikácii používajú rovnaký dátový typ
1. *dátová zviazanosť* (data coupling) existuje pri komunikácii medzi modulmi pomocou parametrov, z ktorých je každý samostatný dátový prvok alebo homogénna dátová štruktúra neobsahujúca riadiace informácie
0. *žiadna zviazanosť* pri vzájomne nezávislých moduloch

*Miera zviazanosti*  $M$  medzi modulmi  $x$  a  $y$  sa definuje ako:

$$M(x, y) = i + \frac{n}{n+1},$$

kde  $i$  je maximum z čísel tried zviazanosti medzi modulmi  $x$ ,  $y$  a  $n$  je celkový počet prepojení medzi  $x$  a  $y$ .

Pomocou metriky zviazanosti medzi dvomi modulmi sa odvádza metrika zviazanosti celého systému, ktorá je definovaná ako medián zviazanosti všetkých modulov systému. Takto definovaná metrika nespĺňa axiómy pre zviazanosť podľa [Briand a kol., 1995].

### Zviazanosť na základe interakcií

V [Briand a kol., 1994] sa skúmajú závislosti medzi dátovými deklaráciami v rozličných moduloch alebo dátovou deklaráciou a podprogramom v inom module. Tieto závislosti sú nazvané interakciami a definované boli dva typy:

- Interakcia typu *Dátová deklarácia – Dátová deklarácia* (DD). Dátová deklarácia (DD) *A* interaguje s dátovou deklaráciou *B* práve vtedy, ak zmena deklarácie alebo použitia *A* zapríčiní potrebu zmeny deklarácie alebo použitia *B*.
- Interakcia typu *Dátová deklarácia - Podprogram* (DS). Dátová deklarácia interaguje s podprogramom, ak interaguje aspoň s jednou z jeho dátových deklarácií.

Na základe týchto dvoch typov interakcií sú definované metriky pre zviazanosť softvérovej súčiastky:

- *Vstupná zviazanosť* (import coupling) IC softvérovej súčiastky *sp* je počet interakcií typu DD medzi vonkajšími a vnútornými dátovými deklaráciami vzhľadom na *sp*.
- *Výstupná zviazanosť* (export coupling) EC softvérovej súčiastky *sp* je počet interakcií typu DD medzi vnútornými a vonkajšími dátovými deklaráciami vzhľadom na *sp*.

Uvedené metriky boli experimentálne použité na identifikovanie softvérových súčiastok náchylných k chybám. Overené boli na troch systémoch NASA na podporu satelitov implementovaných v jazyku ADA. Výsledky experimentu, ktorý zahŕňal aj iné metriky, poukazovali na to, že metriky založené na výstupoch z modulu (medzi nimi aj EC) nie sú významnými ukazovateľmi náchylnosti modulov k chybám. Metriky založené na vstupoch do modulu (kde patrí aj IC) sa osvedčili ako presné ukazovatele náchylnosti modulov k chybám.

### Metriky pre súdržnosť

Pojem súdržnosť sa v [IEEEGLOSS] definuje ako miera, v akej úlohy vykonávané jedným modulom medzi sebou závisia a spôsob, na ktorý závisia. V literatúre môžeme nájsť aj iné definície súdržnosti, ako sú napríklad: miera, v akej sú funkcie alebo elementy spracovania vo vnútri modulu medzi sebou spojené, alebo miera, v akej sú komponenty štruktúry modulu zjednotené v podpore vykonávania jeho funkcie.

### Súdržnosť na základe interakcií

Pri tejto metrike sa pozorujú interakcie typu DD a DS vo vnútri modulu. Definuje sa množina *súdržných interakcií CI* modulu *m* ako zjednotenie

množín interakcií typu DD a DS vo vnútri modulu s výnimkou interakcií typu DD medzi dátovou deklaráciou a formálnym parametrom podprogramu v module.

Na základe tejto definície bola definovaná metrika pomeru súdržných interakcií RCI softvérovej súčasti  $sp$  ako pomer počtu prvkov množiny súdržných interakcií  $C$  k počtu prvkov množiny všetkých interakcií  $M$  softvérovej súčasti  $sp$ .

$$RCI(sp) = \frac{|CI(sp)|}{|M(sp)|}$$

Táto metrika bola použitá v [Briand a kol., 1994] na odhad náchylnosti modulov k chybám v rámci rovnakého experimentu ako pri metrike zviazanosti na základe interakcií.

### Súdržnosť vo fáze návrhu podľa [Bieman98]

Pri tejto metrike sa vychádza zo závislosti medzi vstupmi a výstupmi modulu reprezentovanej grafom závislostí medzi vstupmi a výstupmi (input-output dependence graph, IODG). Každý vstup prispieva k jednému alebo viacerým výstupom z modulu. Vychádzajúc z reprezentácie pomocou IODG grafov sa v [Bieman98] definuje šesť relácií, pomocou ktorých je možné rozlíšiť jednotlivé úrovne súdržnosti (usporiadané od najslabšej k najsilenejšej):

1. *náhodná* - dva výstupy z modulu nezávisia medzi sebou ani od spoločného vstupu.
2. *podmienečná* - dva výstupy z modulu podmienečne závislé od spoločného vstupu (hodnota vstupu rozhoduje o vetvení programu).
3. *iteračná* - dva výstupy z modulu sú iteračne závislé od spoločného vstupu (hodnota vstupu rozhoduje o prerušení opakovania).
4. *komunikačná* - dva výstupy z modulu závisia od spoločného vstupu. Vstup sa používa na počítanie hodnôt oboch výstupov, ale sa podľa neho nerozhoduje pre niekty z výstupov ani o procese iterovania.
5. *postupná* – jeden výstup závisí od druhého.
6. *funkcionálna* – existuje iba jeden výstup z modulu.

Metrika súdržnosti vo fáze návrhu (design-level cohesion) DLC sa určuje na základe relácií medzi párami výstupov z modulu. Pre každý pár sa berie najsilnejšia úroveň súdržnosti medzi nimi. Výsledná súdržnosť modulu je najslabšia z úrovní súdržností párov výstupov z modulu.

Platnosť tejto metriky bola v [Bieman98] preskúmaná na systémových programoch operačného systému UNIX aj na programoch vytvorených študentmi. Výsledky analýzy ukázali, že výsledky merania súdržnosti vo fáze návrhu vo vysokej miere zodpovedajú výsledkom merania zdrojového kódu.

**Funkcionálna súdržnosť vo fáze návrhu podľa [Biemann98]**

Pri odvodení tejto triedy metrík sa vychádzalo z úvah použitých na vývoj analogickej metriky funkcionálnej súdržnosti určenej na meranie na základe dostupného zdrojového kódu [Biemann98]. Na meranie v etape návrhu bola pôvodná metrika funkcionálnej súdržnosti upravená tak, aby vychádzala z reprezentácie závislostí medzi vstupnými a výstupnými komponentmi modulu pomocou IODG grafu.

Metriky funkcionálnej súdržnosti vo fáze návrhu (DFC) sú vyjadrené na základe počtu tzv. *izolovaných* a *podstatných* komponentov a na základe *spojitosti* komponentov. Izolovaný komponent je taký, ktorý je vo vzťahu závislosti iba s jedným výstupom. Podstatný komponent je taký, ktorý je vo vzťahu závislosti so všetkými výstupmi. Spojitosť komponentu  $C$  je definovaná ako pomer medzi počtom výstupov, s ktorými je komponent viazaný reláciou závislosti, a celkovým počtom výstupov z modulu. V prípade, že modul má iba jeden výstup, spojitosť má hodnotu 1.

V [Biemann98] sú definované tri metriky pre súdržnosť:

- *Voľná súdržnosť*  $LC = \frac{D}{T}$
- *Pevná súdržnosť*  $TC = \frac{E}{T}$

(kde  $D$  je počet neizolovaných komponentov,  $E$  je počet podstatných komponentov a  $T$  je celkový počet komponentov v module).

- *Súdržnosť modulu*  $MC$  bola definovaná ako stredná hodnota spojivosti komponentov v module.

$$MC = \frac{\sum_i^T C_i}{T}$$

Tieto metriky boli použité pri rovnakom experimente ako pri metrike funkcionálnej súdržnosti vo fáze návrhu a dospelo sa k podobným výsledkom.

**Meranie používateľského rozhrania**

Kvalitu používateľského rozhrania môžeme hodnotiť z viacerých aspektov, pričom sa často opierame o subjektívne pocity a názory používateľov, ťažko použiteľné ako vstupné údaje pre meranie. Kvantitatívne hodnotenie používateľského rozhrania je možné získať viacerými spôsobmi. Je možné zakomponovať zber údajov počas fázy testovania do samého programu alebo založiť meranie na údajoch ako sú počet chýb priemerného používateľa pri práci s programom alebo cena prístupu k funkciám alebo údajom v systéme. Cena môže byť vyjadrená časom alebo počtom operácií, potrebných na prístup k danému prvku

systému. Napríklad pri web rozhraniach je to počet kliknutí myšou, potrebných na prístupenie hľadanej informácie, pri formulároch počet stlačení kláves potrebných na navigovanie v rozhraní.

Príkladom použitia metrik pre merania používateľského rozhrania sú metriky na meranie hypermedií (hypermediá sú kombináciou hypertextu a mutimedií). Tieto metriky môžu byť využité aj na meranie iných typov používateľských rozhraní.

### **Metriky na meranie hypermedií [Yamada a kol., 1995]**

V [Yamada a kol., 1995] sa skúmajú možnosti merania kvality návrhu hypermedií pre múzeum. Kvalita návrhu je ovplyvnená voľbou prvkov použitých na prezentáciu informácií (stránky, okná, panely, ..) a spôsobom ich prepojenia odkazmi. Štruktúra hypermedií je reprezentovaná grafom a metriky sa odvídzajú na základe matematickej teórie grafov.

Zavádza sa pojem *hĺbky* v hypermediálnom rozhraní, do ktorej sa používateľ môže dostať sledovaním odkazov. Hĺbka uzla v rozhraní sa definuje ako dĺžka najkratšej cesty, ktorou sa používateľ dostane z daného uzla do koreňového (uzla, ktorý má používateľ k dispozícii na začiatku práce). V [Yamada a kol., 1995] sa uzly triedia do jednotlivých „vrstiev“ v rámci rozhrania. Prechody medzi uzlami v rovnakej vrstve sú ohodnotené 0, hodnotia sa iba prechody medzi vrstvami. Hĺbka uzla v takto ohodnotenom grafe sa definuje ako vzdialenosť rozhrania (interface distance). Celkové ohodnotenie uzlov rozhrania sa nazýva *plytkosť rozhrania* (interface shallowness) ISh:

$$ISh = \frac{n(n+1)^2}{n(n+1) - \sum_i IDp_i},$$

kde n je počet uzlov (n>1), a Idpi je vzdialenosť rozhrania z koreňa do i-teho uzla.

Iná vlastnosť hypermedií je jednoduchosť prístupu k uzlom. Táto vlastnosť je vyjadrená ukazovateľom *kompaktnosť*, ktorý je definovaný vzťahom:

$$Cp = \frac{Max - \sum_i \sum_j C_{ij}}{Max - Min},$$

kde Cij je vzdialenosť z uzla i do uzla j, Min = n<sup>2</sup> - n, Max = (n<sup>2</sup> - n)C, C = max Cij (zjednodušené: C = n)

Cp nadobúda hodnoty z intervalu 0 až 1, pričom hodnoty blízke 0 poukazujú na nedostatočný počet prepojení medzi uzlami. V [Yamada a kol., 1995] sa tvrdí, že je pre používateľa dôležitá dostupnosť uzlov v smere od koreňu k hlbším vrstvám, pretože používateľ sa potrebuje rýchlo dostať zo začiatočného uzlu k uzlu ktorý ho zaujíma, pokiaľ potreba rýchleho prehľadávania všetkých iných uzlov v danej vrstve nie je až tak častá. Dostupnosť uzlov týmto spôsobom skúma metrika

*kompaktnosť smerom dolu* (downward compactness), vyjadrená vzťahom:

$$DCp = \frac{n(n-1)^3}{n(n-1) - \sum_i Dp_i},$$

kde  $Dp_i$  je hĺbka (nie vzdialenosť rozhrania) uzla  $i$

Tretia metrika, ktorá sa uvádza je metrika *možnosti navigovania v rozhraní* (interface navigability), ktorá zohľadňuje plytkosť rozhrania, ako aj kompaktnosť smerom dolu. Definovaná je ako lineárna kombinácia dvoch vyššie uvedených metrik.

Na overenie týchto metrik je dostupné iba empirické porovnanie, ktoré ukázalo, že poradie hodnôt ISh a  $Dp_i$  pre tri rôzne systémy je priamo úmerné preferenciám používateľa (na základe počtu prístupov podľa automatických záznamov).

V tomto príspevku sme opísali niektoré metriky týkajúce sa vlastností softvérových systémov a súčastok ako sú zložitosť, zviazanosť a súdržnosť. Spomenuté sú aj metriky pre meranie vlastností používateľského rozhrania. Uvedené metriky sa môžu využiť na predvídanie rôznych vlastností vyvíjaného systému, čo umožňuje včas odhaliť nepriaznivé smery vývoja a podniknúť zodpovedajúce korekčné opatrenia.

Kvôli ohraničenosti priestoru v publikácii, jednotlivé metriky boli spracované iba v krátkosti. Podrobnejšie informácie o jednotlivých metrikách, ako aj iné metriky ktoré neboli spomenuté v tomto príspevku možno nájsť v uvedenej literatúre.

## Literatúra

- [Bieliková00] BIELIKOVÁ, M.: *Softvérové inžinierstvo*. Bratislava: STU, 2000.
- [Bieman98] BIEMAN, J. – KANG, B.: *Measuring Design-Level Cohesion*.  
ftp://ftp.cs.colostate.edu/pub/bieman/tse98.pdf, 1998.  
(6. 4. 2001).
- [Briand94] BRIAND, L. – MORASCA, S. – BASILI, R.: *Defining and Validating High-Level Design Metrics*.  
ftp://ftp.cs.umd.edu/pub/papers/papers/ncstrl.umcp/CS-TR-3301/CS-TR-3301.ps.Z, 1994. (6.4.2001).
- [Briand95] BRIAND, L. – MORASCA, S. – BASILI, R.: *Property-based Software Engineering Measurement*.  
ftp://ftp.cs.umd.edu/pub/papers/papers/ncstrl.umcp/CS-TR-3368/CS-TR-3368.ps.Z, 1995. (6.4.2001).

- [Card90] CARD, D. – GLASS, R.: *Measuring Software Design Quality*. Prentice Hall, 1990.
- [Henry81] HENRY, S. – KAFURA, D.: *Software structure metrics based on information flow*. IEEE Transactions on Software Engineering, 1981, vol. 7, no. 5, s. 510-518.
- [Karkas,98] Karkas, U. – Sultanoglu, S.: *Complexity Metrics and Models*.  
<http://yunus.hun.edu.tr/~sencer/complexity.html>, 1998. (3.4.2001).
- [IEEE9821] *IEEE Std. 982.1-1988 – IEEE Standard Dictionary of Measures to Produce Reliable Software*. The Institute of Electrical and Electronics Engineers, 1988.
- [IEEEGLOSS] *ANSI/IEEE Std. 610.12 – IEEE Standard Glossary of Software Engineering Terminology*. The Institute of Electrical and Electronics Engineers, 1991.
- [Salamon94] SALAMON, W. – WALLACE, D.: *Quality Characteristics and Metrics for Reliable Software*. U.S. Department of Commerce / National Institute of Standards and Technology, September 1994.
- [Varga99] VARGA, M.: *Meranie softvéru*. Bratislava: Ekonomická Univerzita v Bratislave, Fakulta Hospodárskej Informatiky, 1999. Diplomová práca.
- [Yamada95] YAMADA, S. – HONG, J. – SUGITA, S.: *Development and evaluation of hypermedia for museum education: Validation of metrics*. ACM Transactions on Computer-Human Interaction, 1995, vol. 2, no. 4, s. 284-307.

## Meranie v etape implementácie

Vladimír Trgo

**Abstrakt.** Príspevok poskytuje úvod do najznámejších metrík, ktoré sa používajú v etape implementácie pri použití procedurálnej paradigmy programovania: počet riadkov zdrojového textu, Halsteadove metriky, cyklomatická zložitosť a metriky štýlu programovania. Stručný opis metrík dopĺňa názorný príklad ich výpočtu z uvedenej časti zdrojového textu programu.

Výsledkom etapy implementácie je zdrojový text programu. Vzhľadom na jeho charakter môžeme použiť metriky, ktoré sú presnejšie ako metriky v etapách analýzy a návrhu.

Meranie v etape implementácie patrí medzi najviac prepracované časti merania softvérového produktu. Prvé práce, ktoré sa týkali merania zdrojového textu, pochádzajú zo sedemdesiatych rokov 20. storočia. Napriek tomu (alebo práve preto) dochádza často k nejednotnosti pri chápaní určitých pojmov.

### Počet riadkov zdrojového textu

Počet riadkov zdrojového textu programu (angl. Lines of Code, LOC) je snáď najpoužívanejšia metrika veľkosti programu. Na prvý pohľad sa môže zdať, že každý programátor myslí pod týmto pojmom to isté, skutočnosť je však iná. Existuje množstvo rôznych odporúčaní pre počítanie riadkov zdrojového textu. Tieto odlišnosti vznikli kvôli počítaniu či nepočítaniu prázdnych riadkov, riadkov s komentármi, nevykonávaných príkazov (funkčné hlavičky, kľúčové slová begin a end v jazyku Pascal), viacerých príkazov v rámci jedného riadku a príkazov, ktoré sa nachádzajú na viacerých riadkoch. Takisto vznikla otázka, ako počítať znovupoužité riadky.

Často používaná definícia riadkov zdrojového textu hovorí, že pri počítaní sa neberú do úvahy prázdne riadky a riadky, ktoré obsahujú iba komentár. Všetky ostatné riadky započítavame. Tento spôsob je veľmi jednoduchý a môžeme ho ľahko zautomatizovať. Máme na výber niektorý

z existujúcich programov – „meračov“ zdrojového textu alebo si môžeme napísať vlastný.

Tento spôsob počítania riadkov má však jednu veľkú nevýhodu – citlivosť na formátovanie zdrojového textu. Programátori, ktorí píšu viacero rozptýlený kód, dosiahnu väčšiu hodnotu LOC ako programátori, ktorí píšu zhustene. Preto je vhodné zaviesť štandard počítania riadkov zdrojového textu, ktorý pokryje rôzne formátovanie. Umožní zhromaždiť údaje, ktoré potrebujeme na efektívne odhadovanie veľkosti. Programátori si budú môcť navzájom porovnávať svoje merania.

Park ustanovil smernice pre meranie veľkosti programu [Park92]. V zdrojovom texte procedurálneho programu môžeme identifikovať 5 typov príkazov: vykonateľné príkazy, deklarácie, direktívy prekladača, komentáre a prázdne riadky. Príkazy a riadky môžu vzniknúť programovaním (písaním), generovaním automatickým generátorom zdrojového textu alebo kopírovaním (znovupoužitím časti zdrojového textu bez zmeny).

Najjednoduchší spôsob určovania veľkosti je počítanie fyzických riadkov zdrojového textu. Jeden fyzický riadok sa rovná jednému príkazu. Oddeľovačom je špeciálny znak alebo dvojica znakov (newline alebo carriage return–line feed). Tento oddeľovač nezávisí od programovacieho jazyka, ale od použitého operačného systému.

Keď chceme počítať logické príkazy, musíme presne stanoviť, kde príkaz začína a kde končí. Na to slúžia oddeľovače príkazov (angl. statement delimiters). Počítanie logických príkazov (inštrukcií) nezávisí od fyzickej formy, v akej sa príkazy objavia. Tento spôsob počítania závisí menej od programátorského štýlu ako počítanie fyzických riadkov. Nezávisí od konvencií pomenovania – dlhé názvy nezvyšujú počet príkazov. Závisí však od použitého programovacieho jazyka.

Nevýhodou tohoto spôsobu počítania je ťažkosť definovania oddeľovačov. Park uvádza takýto príklad. Koľko logických príkazov je v nasledujúcom fragmente zdrojového textu?

```
if A then B else C endif;
```

Keď kládol túto otázku, zistil, že programátori vnímajú pojem logický príkaz rôzne [Park92]. Najviac z nich (53 %) videlo v uvedenom fragmente tri príkazy. Avšak dosť veľká časť opýtaných (21 %) ho považovala za jediný príkaz. V skutočnosti ide o dva a viac príkazov podľa toho, čo je C.

### Halsteadove metriky

Zdrojový text počítačového programu predstavuje vlastne sekvenciu symbolov, pričom každý symbol môžeme zaradiť medzi operátory alebo operandy [Richta98].

Medzi operátory patria kľúčové slová, aritmetické symboly, zátvorky, operátory porovnania a iné symboly (napr. čiarka a bodkočiarka). Medzi

operandy patria premenné a konštanty. V praxi je niekedy rozlišovanie medzi operátormi a operandami netriviálne. Názov funkcie sa počíta medzi operandy tam, kde sa funkcia definuje, ale medzi operátory tam, kde sa volá. Ďalšie charakteristiky (napr. vetvenie v programe, obtiažnosť riešeného problému) sa neberú do úvahy.

Počet rôznych operátorov označme  $n_1$  a počet rôznych operandov  $n_2$ . Potom veľkosť programového slovníka (angl. vocabulary)  $n$  definujeme ako súčet rôznych operátorov a operandov, teda  $n = n_1 + n_2$ .

Ďalší parameter, ktorý môžeme podľa Halsteada vypočítať, je dĺžka programu. Definujeme ju ako súčet počtu operátorov a operandov  $N = N_1 + N_2$ , kde  $N_1$  predstavuje celkový počet operátorov a  $N_2$  celkový počet operandov.

Halstead považuje programovanie za nedeterministický proces, počas ktorého vyberáme operátory a operandy z vopred daného zoznamu. Predpokladáme, že všetky operátory vyberáme s rovnakou pravdepodobnosťou. Ďalej predpokladáme, že rozhodovanie programátora opisuje binárny rozhodovací strom a operátor sa vyberá pomocou upresňujúcich rozhodnutí. Pre výber  $n_1$  operátorov potrebujeme  $\log_2 n_1$  takýchto rozhodnutí. Napr. výber jedného operátora zo štyroch realizujeme pomocou  $\log_2 4 = 2$  odpovedí typu áno/nie.

Ak sa v programe nachádza  $n_1$  operátorov, tak vykonáme celkovo  $n_1 \log_2 n_1$  rozhodnutí. Analogická situácia nastáva pri výbere operandov. Hodnotu  $N' = n_1 \log_2 n_1 + n_2 \log_2 n_2$  označujeme ako odhad dĺžky programu.

Zatiaľ čo dĺžka programu  $N$  je metrika, ktorú môžeme priamo zmerať z dokončeného programu, odhad dĺžky programu  $N'$  je metrika, ktorú môžeme vypočítať z aktuálnych alebo odhadovaných hodnôt  $n_1$  a  $n_2$ .

Ďalšia metrika, ktorú navrhol Halstead, je objem programu. Zodpovedá celkovému počtu rozhodnutí, ktoré potrebujeme vykonať pri voľbe každého z operátorov a operandov, ktoré sme použili v programe. Rozhodnutie medzi dvomi možnosťami môžeme zakódovať pomocou jedného bitu. Počet bitov potrebných na rozlíšenie medzi  $n$  operátormi a operandmi je  $\log_2 n$ . Objem programu  $V = N \log_2 n$  predstavuje minimálny počet bitov, pomocou ktorých môžeme zakódovať sekvenciu  $N$  operátorov a operandov.

#### Príklad [Churcher98]

```
procedure BubbleSort(var a : IntArray; N : Positive);
var j, t : integer;
begin
  repeat
    t := a[1];
    for j := 2 to N do
      if a[j - 1] > a[j] then
        begin
```

```

        t := a[j - 1];
        a[j - 1] := a[j];
        a[j] := t;
    end
until t = a[1]
end;

```

**Tab. 1:** Operátory a operandy v zdrojovom texte príkladu.

Operátor	Počet
;	4
[...] (index poľa)	8
:=	4
-	3
>	1
=	1
if ... then	1
repeat ... until	1
for...:=...to...do	1
begin ...end	1
procedure ...;	1
<b>n1 = 11</b>	<b>N1 = 26</b>

Operand	Počet
j	7
t	4
a	8
N	1
1	5
2	1
<b>n2 = 6</b>	<b>N2 = 26</b>

veľkosť programového slovníka  $n = n_1 + n_2 = 11 + 6 = 17$

dĺžka programu  $N = N_1 + N_2 = 26 + 26 = 52$

odhad dĺžky  $N' = n_1 \log_2 n_1 + n_2 \log_2 n_2 = 11 \log_2 11 + 6 \log_2 6 = 53,56$

objem programu  $V = N \log_2 n = 52 \log_2 17 = 212,55$

Halsteadova teória sa porušuje najmä „nečistotami“ programovania [Churcher98]. Medzi ne patria napr.:

- komplementárne operácie:  $i + 1 - j - 1 + j$
- viacznačné operandy: ten istý operand predstavuje dve alebo viac vecí v rôznych častiach programu
 
$$r = b * b - 4 * a * c;$$

...

$$r = (-b + \text{sqrt}(r)) / 2.0;$$
- nadbytočné operandy (synonymá): rôzne operandy predstavujú tú istú vec

- spoločné podvýrazy:

```
y = (i+j) * (i+j) * (i+j);
```

```
...
```

```
x = i + j; y = x * x * x;
```

### Cyklomatická zložitosť

McCabe zistil, že na posúdenie zložitosti programu nestačí primitívna metrika typu počet riadkov, ale je vhodnejšie vychádzať zo štruktúry programu [Mills88]. Štruktúru programu môžeme znázorniť pomocou grafu toku riadenia G, v ktorom každý uzol zodpovedá príslušnému bloku programu a každá hrana zodpovedá vetve programu. Graf má vstupný a výstupný uzol. Výstupný uzol sa musí dať dosiahnuť zo všetkých uzlov.

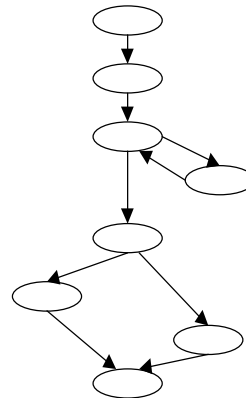
Cyklomatické číslo korešponduje s počtom lineárne nezávislých ciest v grafe. Môžeme ho vypočítať použitím vzorca z teórie grafov:

$v(G) = e - n + 2$ , kde  $e$  je počet hrán a  $n$  je počet uzlov.

#### Príklad

```
begin
  x:=y;
  for z:=1 to x do
    write ('');
  if a=x then
    write('a');
  else
    writeln;
end.
```

$v(G) = 9 - 8 + 2 = 3$



Cyklomatická zložitosť patrí medzi intervalové metriky. Napr. program s cyklomatickým číslom 6 je o 4 jednotky zložitejší ako program s cyklomatickým číslom 2.

### Metriky štýlu programovania

Berry–Meekingsove metriky slúžia na posúdenie štýlu programovania. Sledujú sa určité charakteristiky zdrojového textu programu. Berry a Meekings pridelili každej charakteristike váhu, ktorá indikuje jej príspevok k celkovému skóre. Najlepšie skóre má hodnotu 100. V zdrojovom texte programu napísaného v jazyku C sa sleduje týchto 11 charakteristík:

- dĺžka modulov (15 b.),
- dĺžka identifikátorov (14 b.),
- percento komentárových riadkov z celkového počtu riadkov (12 b.),

- percento odsadzujúcich medzier z celkového počtu znakov riadku (12 b.),
- percento prázdnych riadkov (11 b.),
- priemerný počet nemedzerových znakov v riadku (9 b.),
- priemerný počet medzier v riadku (8 b.),
- percento symbolicky zapísaných konštánt (8 b.),
- počet použitých rezervovaných slov (6 b.),
- počet súborov použitých formou „include“ (5 b.),
- počet príkazov „goto“ (–5 b. [Bieliková00] alebo –20 b. [Richta98]).

Berry a Meekings určili minimálne a maximálne hodnoty pre každú charakteristiku. Hodnoty mimo tohto rozsahu neprispievajú k celkovému skóre. Ďalej pre každú charakteristiku určili dve hranice medzi minimom a maximom, ktoré reprezentujú hranice ideálneho rozsahu. Všetky hodnoty vnútri tohto ideálneho rozsahu získajú maximálny počet bodov. Hodnoty, ktoré sú medzi minimom a maximom, ale mimo ideálneho rozsahu, dostanú body podľa ich vzdialenosti od ideálneho rozsahu.

#### Príklad

- charakteristika: dĺžka identifikátorov – priemerný počet znakov v názve identifikátorov, ktoré definoval programátor (14 b.)
- minimum: 4
- maximum: 14
- ideálny rozsah: 5–10

Berry–Meekingsove metriky môžu slúžiť na efektívne rozlíšenie programátorov s dobrým (skóre>60) a zlým (skóre<20) štýlom programovania [Bieliková00].

**I**nformácie, ktoré môžeme získať z metrík použitých v etape implementácie, majú menšiu hodnotu pri hodnotení procesu a výsledku než informácie z metrík, ktoré používame v skorších etapách životného cyklu softvéru (analýza, návrh). Môžeme ich však využiť, napr. pri testovaní, údržbe a v budúcich podobných projektoch.

## Literatúra

- [Bieliková00] BIELIKOVÁ, M.: *Softvérové inžinierstvo*. Kapitola 27. Meranie v softvérovom projekte. Bratislava: STU, 2000. s. 164-167. ISBN 80-227-1322-8.

- [Churcher98] CHURCHER, N.: *Metrics – prednášky z predmetu COSC314 Softvérové inžinierstvo týkajúce sa merania*.  
<http://www.cosc.canterbury.ac.nz/teaching/classes/cosc314/Docs/neville-ohps/metrics.pdf>, 1998.
- [Mills88] MILLS, E.: *Software Metrics. SEI Curriculum Module*. SEI-CM-12-1.1.  
<http://www.sei.cmu.edu/publications/documents/cms/cm.012.html>,  
[http://www.cvc.uab.es/shared/teach/a21291/apunts/SEI/Software metrics cm12.pdf](http://www.cvc.uab.es/shared/teach/a21291/apunts/SEI/Software%20metrics%20cm12.pdf), 1988.
- [Park92] PARK, R.E.: *Software Size Measurement: A Framework for Counting Source Statements*. Technical Report. CMU/SEI-92-TR-020,  
<http://www.sei.cmu.edu/publications/documents/92.reports/92.tr.020.html>, 1992.
- [Richta98] RICHTA, K. – Sochor, J.: *Softwarové inžinierství I*. Praha: ČVUT, Fakulta elektrotechnická, 1998. Časť 6. *Softwarová fyzika*. s. 185-204. ISBN 80-01-01428-2.



## Meranie a paradigmy programovania

Michal Šrámka

**Abstrakt.** Príspevok pojednáva o možných prístupoch merania softvéru pri použití alternatívnych paradigiem programovania k štruktúrovanej paradigme. Hlavná časť je venovaná objektovo-orientovanej paradigme programovania, menšie časti potom logickej a funkcionálnej paradigme programovania. Dôraz je okrem kategorizácií a definícií jednotlivých metrík kladený aj na ich interpretáciu.

V programovaní nehovoríme iba o metódach programovania, ale aj o spôsoboch programovania a o programovacích postupoch. Sú to špeciálne metódy, ktoré opisujú čiastkové riešenia – napr. v etape návrhu alebo implementácie.

Nasledujúce kapitoly pojednávajú o meraní softvéru pri rôznych prístupoch k ich tvorbe.

### Paradigmy programovania

Súhrn spôsobov formulácie problémov, metodologických prostriedkov ich riešenia, štandardných metodík rozpracovania sa označuje ako paradigma. Paradigmy sú teda názory, teórie, metódy, metodiky, praktiky a techniky, ktoré sa v danej oblasti uznávajú. V oblasti programovania sa podľa klasifikácie ACM dajú programovacie techniky (ekvivalent paradigiem) klasifikovať nasledovne:

- aplikatívne (funkcionálne) programovanie,
- automatické programovanie,
- súbežné programovanie, kam patrí
  - distribuované programovanie,
  - paralelné programovanie,
- sekvenčné programovanie,
- objektovo-orientované programovanie,

- logické programovanie,
- vizuálne programovanie.

Je zrejmé, že meranie softvéru musí byť rozlíšené podľa použitej metódy programovania.

V nasledujúcich častiach sú preto opísané špecifiká merania softvéru pri niektorých z horeuvedených paradigiem programovania. Najväčší priestor bude venovaný objektovo-orientovanej paradigme programovania a spomenuté budú aj paradigmy logického a funkcionálneho programovania a merania v týchto alternatívnych paradigmách.

Meranie sekvenčných (štruktúrovaných) programov je opísané v iných častiach tejto knihy, preto nebude v tejto časti diskutované. Avšak mnohé princípy a aj konkrétne metriky a techniky merania v sekvenčnom programovaní sa dajú aplikovať aj na ostatné paradigmy programovania – napr. cyklomatická zložitosť štruktúrovaného programu alebo jeho konkrétnej funkcie sa dá použiť aj na meranie konkrétnej metódy objektu v objektovo-orientovanej paradigme programovania.

Jednotlivé časti začínajú stručným opisom tej-ktorej paradigmy programovania. Tieto opisy nie sú vyčerpávajúce a ich úlohou je len priblíženie konceptu a pojmov pre pochopenie jednotlivých metrik.

### **Objektovo-orientované programovanie**

V objektovo-orientovanom programovaní sa program chápe ako množina objektov. Objekt je štruktúra, ktorá zahŕňa údaje aj funkcie. Objekt má nasledovné charakteristiky:

- meno – jedinečný identifikátor,
- stav – reprezentovaný atribútmi
- správanie sa – množina povolených operácií (akcií)

Objekt si uchováva stav vo vlastných premenných a reaguje na podnety – akcie, ktoré vykonáva na základe definovaných metód.

Podobné objekty – čo do štruktúry a správania sa – vytvárajú triedy. Medzi triedami sa dajú definovať rôzne vzťahy. Jedným z definovaných vzťahov je napríklad delenie. Ďalšie vzťahy medzi triedami budú spomenuté pri jednotlivých metrikách.

V ďalšom texte sa pojem objektovo-orientované programovanie chápe ako objektovo-orientovaný vývoj (analýza, návrh, implementácia) systému.

### **Meranie v objektovo-orientovanom programovaní**

Metriky opísané v tejto časti majú dve úlohy – predpoveď chýb a predpoveď vynaloženého úsilia. Obe metriky vychádzajú z jednoduchého predpokladu, že čím je softvér komplikovanejší, tým viac bude obsahovať chýb a tým dlhšie bude trvať jeho vývoj. Metriky,

ich hodnoty a možnosti porovnania týchto hodnôt s predchádzajúcimi projektami by mali byť k dispozícii už v počiatočných fázach projektu, zatiaľčo metriky predpovedajúce chyby sa môžu (a uplatňujú sa) počas všetkých etáp vývoja softvéru.

Tradičné metriky nestrácajú zmysel pri meraní v objektovo-orientovanom programovaní. Niektoré metriky tu majú rovnaké uplatnenie, ako pri pôvodných určeníach (napr. LOC, funkčné body). Avšak väčšina týchto tradičných metrík berie ohľad na procedurálne a sekvenčné vlastnosti programov, a teda je pre meranie v objektovo-orientovanom programovaní menej vhodná.

Otázka teda znie, či existujú metriky, ktoré by dokázali viac charakterizovať špecifické vlastnosti a stavy v objektovo-orientovanom programovaní.

Takéto metriky existujú a dajú sa rozdeliť do dvoch skupín – metriky pre meranie tried (*class level metrics*) a metriky pre meranie interakcie tried z pohľadu celého systému (*system level metrics*) [Kolewe93]. Metriky pre meranie tried nám umožňujú merať zložitosť tried pri návrhu a implementácii. Naproti tomu metriky pre meranie interakcie objektov merajú závislosti medzi viacerými objektmi pri návrhu.

### **Metriky pre meranie tried**

Nasledovné metriky merajú zložitosti jednej konkrétnej triedy. Buď merajú zložitosť triedy vzhľadom na vnútornú stavbu alebo merajú vzťah skúmanej triedy k ostatným triedam. Metriky pre meranie tried nevnímajú viaceré triedy ako jeden systém.

#### **Váha metód v triede**

Metrika váha metód v triede (*weighted methods per class*) meria zložitosť správania sa triedy. Táto metrika sa dá definovať ako suma cyklotmatickej zložitosti pre každú metódu v triede.

Idea metriky je, že trieda s veľkým počtom jednoduchých metód je porovnateľne zložitá ako trieda s menším počtom komplikovanejších metód.

#### **Hĺbka dedenia**

Metrika hĺbka dedenia (*depth of inheritance tree*) sa dá definovať ako celkový počet predchodcov pre jednu konkrétnu triedu.

Táto metrika slúži na pochopenie zložitosti triedy. Čím je hĺbka dedenia väčšia tým je pochopenie triedy a jej metód náročnejšie.

#### **Spájanie tried**

Metrika spájanie tried (*class coupling*) sa dá formálne opísať ako počet všetkých vzťahov danej triedy s inými triedami.

Existuje spojenie medzi touto metrikou a konceptom abstrakcie. Abstraktné triedy zvyknú mať vyššiu hodnotu metriky spájania tried.

Vytvárajú sa väčšinou z dôvodu znovupoužitia. Výsledkom je, že neskoršia zmena jednej triedy sa stáva veľmi náročnou bez pochopenia väzieb medzi všetkými príslušnými triedami.

### Účinok triedy

Metrika účinok triedy (*response of a class*) je definovaná ako celkový počet metód v ostatných triedach, ktoré sú volané z metód danej skúmanej triedy. Formálne sa táto metrika počíta ako veľkosť množiny, ktorá pozostáva zo všetkých jej metód a zo všetkých metód iných tried volaných v metódach tejto triedy.

Táto metrika je iným príkladom spájania tried. Za účelom pochopenia danej triedy nestačí poznať o koľkých triedach musíme vedieť, ale musíme poznať aj určité detaily o každej z týchto tried. Teda čím väčšia hodnota metriky, tým komplikovanejšia je skúmaná trieda.

### Nesúdržnosť triedy

Metrika nesúdržnosť triedy (*lack of method cohesion*) meria súdržnosť danej triedy. Ak trieda má určitý počet metód, tak nesúdržnosť triedy je formálne definovaná ako počet navzájom rôznych množín určených ako prienik premenných použitých v metódach danej triedy.

Aj je hodnota tejto metriky veľká, tak existuje väčšie množstvo premenných v triede, ktoré sa nepoužívajú vo všetkých metódach tejto triedy. Táto metrika môže poukazovať na nutnosť rozdelenia triedy na menšie a viac súdržné triedy.

### Metriky pre meranie interakcie tried

Metriky v tejto skupine merajú viaceré (často všetky) triedy naraz za účelom zistenia zložitosti celého systému. Meria sa hlavne interakcia, komunikácia a zviazanosť medzi triedami.

### Počet potomkov

Metrika počet potomkov (*number of children*) meria rozsah vlastností triedy. Idea je, že všeobecne je lepšie mať v hierarchii tried väčšiu hĺbku ako šírku. Takisto platí, že čím viac potomkov, tým ťažšie bude ich pochopenie.

### Počet hierarchií tried

Metrika počet hierarchií tried (*number of class hierarchies*) je jednoducho definovaná ako meranie veľkosti systému. Táto metrika sa snaží spočítať počet podstatných modulov (zhlukov tried) z ktorých sa vlastne skladá celý systém.

Zhluk tried môžeme chápať ako ucelené moduly a teda táto metrika nám dáva triviálny pohľad na zložitosť systému.

### Počet zhlukov tried

Metrika počet zhlukov tried (*number of class clusters*) meria počet spojení medzi rôznymi triedami v systéme. Z toho vyplýva, že táto metrika podobne ako predchádzajúca meria aj počet zhlukov tried v systéme. Formálna definícia tejto metriky v systéme s daným počtom tried je počet rôznych (disjunktných) množín určených z množiny všetkých možných prienikov z množín tried, ktoré sú v ľubovolnej spojitosti s ľubovolnou inou triedou v systéme.

### Zložitosť asociácií

Metrika (*association complexity*) je meranie zložitosti štruktúry systému. Táto metrika je obdobou cyklomatickej (McCabe) zložitosti a je definovaná vzorcom:

$$AC = A - C + 2P$$

kde:

- A je počet ľubovolných asociácií (väzieb) v diagrame tried
- C je počet tried v diagrame tried
- P je počet neasociovaných (so žiadnou triedou nekomunikujúcich) tried

### Logické programovanie

Logické programovanie sa zakladá na postupoch, ktoré sa používajú pri dokazovaní teorém v logike prvého rádu. Hlavným zámerom použitia logického programovania je možnosť deklaratívneho opisu riešeného problému.

Napriek odlišnej syntaxe aj sémantiky pri programovaní a hlavne pri vykonávaní logického programu aj tu existujú postupy merania niektorých špecifických atribútov logických programov.

Prvá skupina mier je založená na počítaní základných syntaktických jednotiek v programe. Druhá skupina vychádza z vyjadrenia toku riadenia logických programov metódami teórie grafov.

### Syntaktické jednotky v programe

Logické programy majú hierarchickú syntax. Pre priblíženie, napríklad v Prologu sa program skladá z predikátov, predikáty z klauzúl a klauzuly z termov. Z toho vyplýva aj možnosť merania počtu jednotlivých štruktúr ako metrik rozsahu programu (obdoba metriky LOC alebo počtu funkcií v štruktúrovanom programovaní).

Možné je tiež merať riadiacu previazanosť predikátu s okolím – množstvo klauzúl, ktoré program obsahuje, množstvo iných predikátov od ktorých závisí daný predikát alebo množstvo predikátov, ktoré závisia od neho.

Ďalej je možné merať množstvo predikátov typu *rez*, *fail* alebo kombinácií *rez-fail*, typy rezov (zelený alebo červený rez) alebo množstvo extra-logických predikátov (predikátov s vedľajšími účinkami).

### **Teória grafov**

Pre vyjadrenie toku riadenia logických programov nie sú vhodné vývojové diagramy. Najlepšie toto riadenie vyjadrujú a-alebo grafy. Predikát je potom namapovaný na alebo-vrchol takéhoto grafu a každý a-vrchol zodpovedá jednej klauzule definovanej v predikáte. V prípade používania rezov alebo iných extra-logických predikátov je situácia komplikovanejšia.

V takýchto grafoch je potom možné (pomocou známych algoritmov teórie grafov) odhadnúť napríklad maximálny počet riešení predikátu, zistiť prítomnosť alebo neprítomnosť rekurzie, počty rekurzií, atď.

### **Funkcionálne programovanie**

Vo funkcionálnom programovaní je program zložený z funkcií, funkcionálov (funkcií operujúcich nad funkciami) a z nich zložených výrazov.

Moderné funkcionálne programy môžu obsahovať globálne premenné alebo v rámci funkcie môžu obsahovať aj sekvenčné časti (napr. v LISPe forma PROG)

Funkcionálne programy často využívajú rekurziu. Existuje niekoľko typov rekurzií, ktoré sa dajú v definícii funkcie jednoducho rozlíšiť:

- rekurzia s jednoduchým alebo s viacnásobným testom
- rekurzia na chvoste
- rekurzia s rozšírením
- rekurzia s podmienkovým rozšírením
- monotónna alebo nemonotónna rekurzia
- rekurzia na viacerých argumentoch
- viacnásobná rekurzia

### **Metriky vo funkcionálnom programovaní**

Okrem známych LOC a funkčných bodov môže mieru veľkosti programu udávať aj počet definovaných funkcií v programe.

Funkcie sa navzájom volajú – sú teda previazané. Previazanosť funkcií môžeme merať vektorom, kde prvky vektora pre danú funkciu sú počty volaní ostatných funkcií. Aritmetický priemer takéhoto vektora potom určuje dôležitosť konkrétnej funkcie.

Ďalšou metrikou je meranie počtu a typov rekurzií. Je zrejmé, že funkcia bude tým zložitejšia na pochopenie, čím zložitejšia bude rekurzia a čím viac typov rekurzie bude v danej funkcii použitých.

Cieľom príspevku nebolo navrhovanie a vytváranie nových špecifických metrik pre dané paradigmy programovania, ale skôr referovanie o stave, ktorý panuje v tejto oblasti merania softvéru.

Metriky pre objektovo-orientované programovanie existujú a poskytujú hodnotné informácie pre vývojárov a manažérov projektov. Jednak sú to „tradičné“ metriky, navrhnuté pre iné paradigmy programovania – hlavne pre štruktúrnú paradigmu ako napríklad McCabeova zložitosť alebo počet riadkov programu, a jednak sú to metriky špecificky navrhnuté pre objektovo-orientovaný vývoj softvéru. Tieto posledne spomenuté metriky prinášajú v meraní najväčšiu efektivitu.

Opísané metriky pre paradigmy logického a funkcionálneho programovania boli skôr empirického charakteru, jednoducho odvoditeľné od modelov a štruktúry daných programov.

Pre jednotlivé metriky existujú viaceré návody, ako ich interpretovať v kontexte vývoja softvéru. Nie je však ľahké stanoviť presné hranice hodnôt, napríklad, že hodnota 8 pre danú metriku je dva-krát zložitejšia alebo dva-krát horšia ako hodnota 4 pre tú istú metriku.

## Literatúra

- [Kolewe93] KOLEWE, R.: *Metrics in Object-Oriented Design and Programming*. Software Development Magazine, <http://www.eridani.com/metric.htm>, Október 1993.
- [Rosenberg98] ROSENBERG, L. H. – HYATT, L.: *Applying and Interpreting Object Oriented Metrics*. Software Technology Conference, Utah, USA, [http://satc.gsfc.nasa.gov/support/STC\\_APR98/apply\\_o/apply.pdf](http://satc.gsfc.nasa.gov/support/STC_APR98/apply_o/apply.pdf), Apríl 1998.
- [Varga99] VARGA, M.: *Meranie softvéru*. Bratislava: Ekonomická Univerzita v Bratislave, Fakulta Hospodárskej Informatiky, 1999. Diplomová práca.



## Meranie pri testovaní, prevádzke a údržbe softvérových systémov

Szabolcs Molnár

**Abstrakt.** Príspevok sa zaoberá základnými metrikami a modelmi, ktoré sa využívajú pri testovaní a pri prevádzke softvérových systémov. Počas testovania sa odhalia tie poruchy softvéru, ktoré sú zodpovedné za zlyhanie celého systému. V súčasnosti neexistuje metóda, ktorou by sme dokázali zistiť všetky chyby, ktoré obsahuje daný softvérový systém. Preto testovanie nemôže preukázať, že v programe nie sú chyby. Môže iba ukázať, že tam chyby sú. Počet všetkých chýb v softvéri sa zvyčajne odhaduje pomocou rôznych metód. Najvýznamnejšie vlastnosti softvéru, ktoré môžeme merať pri testovaní a prevádzke sú udržovateľnosť a spoľahlivosť. Na predpovedanie spoľahlivosti existuje niekoľko matematických modelov, ktoré sa používajú aj v súčasnosti. Na modelovanie udržovateľnosti predvedieme dve techniky, ktoré používa aj firma Hewlett-Packard pri testovaní svojich produktov.

**K**aždý softvérový systém by mal prejsť procesom testovania, počas ktorého sa pokúšame nájsť a odstrániť chyby, ktoré by spôsobili zlyhanie systému počas prevádzky. Testovanie systému však nezaručí, že nájdeme všetky chyby. Po zavedení systému do prevádzky, môžeme nájsť aj ďalšie nedostatky, ktoré znižujú kvalitu a použiteľnosť softvéru.

Každé zariadenie potrebuje, aby bolo pravidelne udržiavané. Nie je to inak ani pri softvérových systémoch. Práve preto je potrebné, aby aj tieto systémy boli ľahko udržovateľné.

Počas uvedených etáp dokážeme merať niektoré metriky softvéru, na základe ktorých dokážeme usúdiť spoľahlivosť, správnosť, kvalitu ale aj ďalšie vlastnosti softvérových systémov.

### Meranie pri testovaní

Proces testovania softvérových systémov je veľmi často poslednou obranou proti katastrofám, ktoré spôsobujú chyby a nedostatky pri vývoji

softvéru. V súčasnosti výpočtovú techniku využívame skoro všade. Pomocou počítača riadime elektrárne, výpočtovú techniku využívame pri bankových transakciách ale aj pri výučbe, atď. Keď preto program zlyhá, niekedy môžu nastať „aj dosť nepríjemné situácie“. Aby sme znížili pravdepodobnosť výskytu chýb, používame testovanie softvérových systémov [Voas95].

*Testovanie* je procesom, ktorý odhadne stupeň „prijateľnosti“, kde prijateľnosť je hodnotená podľa špecifikácie. Softvérovou verifikáciou môže byť: dynamické testovanie a statické testovanie softvéru [Bieliková00].

*Dynamické testovanie* vykonáme tak, že spustíme testovaný systém viackrát za sebou. Pri týchto testoch množina testovacích vstupov musí byť „prijateľná“. Testovacie vstupy sa vyberajú na základe testovacieho kritéria [Voas95].

*Statické testovanie* nevyžaduje vykonie programu. Pri týchto testoch sa sleduje zhoda so špecifikáciou a požiadavkami používateľa [Voas95].

Počas testovania môžeme sledovať niektoré dôležité vlastnosti softvéru.

- *Správnosť* znamená, že vytvorený softvérový systém spĺňa špecifikáciu, ktorá bola určená pre daný softvérový systém.
- *Použitelnosť* znamená, že vytvorený systém spĺňa používateľské potreby.
- *Výkonnosť* určuje, koľko a akých technických prostriedkov potrebuje náš softvérový systém počas prevádzky. Výkonnosť ovplyvní aj doba odozvy systému ale aj ďalšie ukazovatele.
- *Spôľahlivosť* sa definuje ako miera frekvencie a kritickosti zlyhania prevádzky výrobku [Bieliková00]. Jednoduchšie je to pravdepodobnosť, že systém počas určitého časového obdobia bude fungovať bez poruchy.

Na začiatku tohoto príspevku treba vyjasniť rozdiel medzi dvomi výrazmi, ktoré sú: *chyba* a *porucha*.

*Chyby* v softvéri vznikajú pri vývoji informačného systému a spôsobujú poruchy v softvérových systémoch. *Poruchy* sú zodpovedné za zlyhanie systému [Munson92]. Počas testovania môžeme nájsť poruchy systému, prípadne ich môžeme odstrániť. Veľmi často obsahujú programy aj menšie, drobné nedostatky, ktoré priamo nie sú zodpovedné za zlyhanie systému. Takéto „chyby“ v systéme často úmyselne nie sú odstránené, lebo pri zložitejších systémoch aj malé zmeny môžu spôsobiť ďalšie neočakávané a ešte vážnejšie chyby.

V praxi platí názor: „Neopravujte automaticky to, čo nespôsobuje haváriu“ [Bieliková00]. Na vysvetlenie tejto myšlienky uvedieme nasledujúci príklad podľa [Bieliková00]. Predstavme si, že počas testovania nájdeme niečo, „čo vyzerá“ ako chyba. Takéto chyby v žiadnom prípade neopravujeme sami, lebo je veľká pravdepodobnosť, že zavedieme

d'alšie chyby. Namiesto toho treba zadať žiadosť na zmenu a pri kontrole akosti potom zistia, či nájdenú chybu treba opraviť alebo nie.

Softvérové metriky zložitosti úzko súvisia s poruchami, ktoré sa vyskytujú v jednotlivých moduloch softvérového systému. Je priamy vzťah medzi niektorými metrikami zložitosti a medzi počtom atribútov pre poruchu, ktoré môžeme nájsť počas testovania systému. Iste totiž platí, že čím sú jednotlivé moduly zložitejšie, tým je pravdepodobnejšie, že nastane porucha.

Lipow vytvoril model, ktorý predpovedá počet porúch softvérového systému pre jednotlivé riadky. Tento model je založený na Halsteadovej softvérovej metrike. Crawford našiel viaceré metriky pre vyjadrenie zložitosti softvéru, ktoré podávajú ešte lepšie výsledky ako metriky založené na počte riadkov [Munson92].

Ako sme už povedali, v praxi niektoré chyby úmyselne nie sú odstránené. Z tohto zdôvodu má zmysel hovoriť o *počte známych neodstránených chýb*. Ak vážime počet chýb závažnosťou ich dôsledkov, môžeme dosiahnuť ešte lepšiu porovnateľnosť. Na ohodnotenie môžeme používať napr. trojstupňovú ordinálnu mierku (pozri [Varga99]).

Ďalšia metrika, ktorú môžeme merať počas testovania je *podiel bezchybných testovacích behov k celkovému počtu testovacích behov*, ktorá sa využíva na predpovedanie spoľahlivosti softvéru. Miera je normalizovaná do intervalu  $\langle 0, 1 \rangle$ . Problém je v tom, že hodnota tejto metriky závisí nielen od kvality softvéru, ale aj od toho, ako dobre bol otestovaný daný softvérový systém. Aby sme vedeli určiť mieru ako dobre bol daný softvérový systém otestovaný, potrebujeme poznať nielen počet nájdených chýb ale aj počet všetkých chýb v softvéri. Bohužiaľ, počet všetkých chýb sa dá iba odhadnúť, na čo sa používajú rôzne metódy.

Jedna z najznámejších je tzv. *úmyselné včlenenie známeho počtu chýb* do softvéru pred testovaním. Na to sa používa vzťah:

$$N = A.B / C$$

kde  $A$  je počet všetkých vložených chýb,  $B$  je počet všetkých chýb, ktoré sme našli počas testovania,  $C$  je počet nájdených vložených chýb a  $N$  je odhad celkového počtu chýb v softvéri.

Ďalšia metóda, ktorú môžeme používať na zistenie rozsahu otestovania je tzv. *analýza pokrytia*. Pri tejto metóde sa určí podiel testami pokrytých prvkov vnútornej štruktúry programu ako sú: príkazy, vetvenia, úseky medzi vetveniami, atď [Varga99].

### Meranie počas prevádzky

Prirodzenou mierou kvality softvéru v prevádzke je počet zlyhaní a porúch, ktoré nastanú počas prevádzky softvéru. Je veľmi dôležité, aby boli správne klasifikované tieto nedostatky. Kým poruchy ovplyvňujú použiteľnosť systému, zlyhania charakterizujú spoľahlivosť systému [Varga99]. Na modelovanie spoľahlivosti sa používajú matematické

modely. Matematické modelovanie spoľahlivosti má za cieľ sledovať zmeny spoľahlivosti produktu v čase. Pomocou dát o výskytoch zlyhaní počas testovania je možné všeobecné teoretické modely aplikovať na konkrétny produkt a odhadnúť neznáme parametre spoľahlivosti, ako to urobili v [Tian95].

Matematické modely sú navrhnuté na základe rozdielnych predpokladov. Uvediem niektoré matematické modely, ktoré sú publikované v [Tian95]:

- **Morandov geometrický model**

Pravdepodobnosť, že zlyhanie (čas okamihu zlyhania je  $T$ ) nastane v časovom intervale  $(t, t + \Delta t)$ , za predpokladu, že nenastalo pred  $t$ , je

$$z(t)\Delta t = P\{t < T < t + \Delta t \mid T > t\}$$

pričom pre  $i$ -te zlyhanie platí:

$$z_i(t) = D\phi^{i-1}$$

kde  $D$  a  $\phi$  sú konštanty.

- **Musaov základný model:**

Pre  $i$ -té zlyhanie platí:

$$z_i(t) = z_0(1 - (i - 1) / N)$$

kde  $z_0$  a  $N$  sú konštanty.

- **Littlewood - Verrallov Bayesian model**

Tento model naznačí čas zlyhania  $t_i$  pre  $i$ -tu periódu ako

$$f(t_i \mid \lambda_i) = \lambda_i e^{-\lambda_i t_i}$$

kde miera zlyhania  $\lambda_i$  je náhodná premenná s  $\Gamma$ -rozdelením s parametrami  $\alpha$ ,  $\psi(i)$ , tj.:

$$f(\lambda_i \mid \alpha, \psi(i)) = \frac{[\psi(i)]^\alpha \lambda_i^{\alpha-1} e^{-\psi(i)\lambda_i}}{\Gamma(\alpha)}$$

kde  $\psi(i)$  závisí od kvality programátorov a od obtiažnosti úloh. Toto sa najčastejšie opisuje lineárnou, kvadratickou alebo inou rastúcou funkciou, s parametrom  $i$ .

- **Goel - Okumatov NHPP (Nonhomogeneous Poisson failural arrival Process) model**

Predpokladom tohoto modelu je nehomogénne Poissonové rozdelenie zlyhaní, s počtom zlyhaní  $N(t)$ , ktoré je z intervalu  $(0, t)$  a je definovaná pomocou vzorca:

$$P[N(t) = n] = \frac{[m(t)]^n e^{-m(t)}}{n!}$$

kde funkcia  $m(t) = a(1 - e^{-bt})$ , kde  $a$  a  $b$  sú konštanty.

- **Musa - Okumatov logaritmický Poisson model:**

Je to ďalší NHPP model, kde predpokladom je taktiež nehomogénne Poissonové rozdelenie zlyhaní. Tento model má iba iný tvar funkcií  $m(t)$ :

$$m(t) = \frac{1}{\theta} \log(\lambda_0 \theta t + 1)$$

kde  $\lambda$  a  $\theta$  sú konštanty.

### Meranie počas údržby

Viacerí experti tvrdia, že udržovateľnosť systému je „najväčšia výzva našich čias.“ Udržovateľnosť je nesmierne dôležitou charakteristikou každého softvérového systému. Brooks vo svojej klasickej knihe *The Mythical Man-Month* vyslovil: „Celkové náklady udržovateľnosti predstavujú 40% alebo viac z celkového nákladu vývoja celého systému.“ Niektorí ďalší autori uvedú ešte väčšie odhady. Podľa Parikha táto hodnota sa pohybuje okolo 45 až 60% [Coleman94]. V [Bieliková00] táto hodnota je ešte vyššia, pohybuje sa okolo 70 až 80%.

Udržovateľnosť je charakteristika softvéru, ktorá ukazuje ako dobre sa dá udržiavať daný softvérový systém alebo jednotlivé moduly. Zahŕňa také vlastnosti softvéru ako: modifikovateľnosť alebo čitateľnosť. Po nameraní udržovateľnosti môžeme využívať získané poznatky pri rozhodovaní, veď ako sme už spomenuli, údržba je najnákladnejším procesom. Môžeme sa rozhodnúť tak, že využívame už existujúce komponenty alebo vybudujeme svoje, ktoré budú lepšie udržovateľné. Môžeme rozhodnúť aj tak, že časť systému, ktorá je ťažko udržovateľná navrhne znova.

V poslednej dobe na základe softvérových metrík boli vytvorené niektoré metódy pre kvantifikovanie udržovateľnosti softvéru:

### Hierarchický multidimenzionálny ohodnocovací model (HPMAS)

Tento model bol vyvinutý v Hawlett-Packard. Je založený na hierarchickej organizácii softvérových metrík. Metóda je podrobne opísaná v [Varga99] a [Coleman94]. Celý model sa rozdeľuje na 3 základné dimenzie:

- *Riadiaca štruktúra* zahŕňa charakteristiky, ako je systém alebo modul rozložený na algoritmy.
- *Informačná štruktúra* zahŕňa charakteristiky na výber a použitie údajových štruktúr a techniky tokov dát.
- *Typografia, pomenovanie a komentovanie* definuje charakteristiky ako: typografické úpravy, pomenovanie a komentáre v zdrojovom kóde.

Pre každú dimenziu z horeuvedených môžeme určiť samostatné metriky. Akonáhle sú definované všetky potrebné metriky pre danú dimenziu, môžeme si vytvoriť index udržovateľnosti pre danú dimenziu, ktorý je funkciou daných metrík. Nakoniec vypočítame celkovú hodnotu udržovateľnosti tak, že kombinujeme všetky 3 indexy.

Každá miera má svoj optimálny rozsah (trigger point range) zodpovedajúci najlepšej udržovateľnosti. Ak hodnota danej metriky je mimo tohto intervalu, udržovateľnosť softvéru je nízka. Napríklad, ak prípustná hodnota pre priemerný počet riadkov v jednotlivých moduloch je medzi hodnotami 5 a 75, hodnota pod 5 a nad 75 charakterizuje, že modul nie je dobre udržovateľný.

Celkovú udržovateľnosť pre jednu dimenziu môžeme vypočítať pomocou doplnkom do váženého priemeru percentuálnych odchýlok od týchto rozsahov [Coleman94]. Udržovateľnosť pre jednu dimenziu sa vypočíta podľa nasledujúceho vzorca:

$$DM_{\text{dimenzia}} = 1 - \frac{\sum w_i D_i}{\sum w_i}$$

kde  $DM_{\text{dimenzia}}$  je udržovateľnosť pre jednu dimenziu,  $w_i$  sú váhy a  $D_i$  sú pomerné odchýlky z intervalu  $\langle 0, 1 \rangle$ .

Celková hodnota udržovateľnosti sa rovná násobkom troch indexov v troch dimenziách, jeho hodnota sa pochybuje v intervale  $\langle 0, 100 \rangle$  a udáva sa v percentách. Najlepšie udržovateľný systém má index udržovateľnosti 100%.

Tento model bol kalibrovaný subjektívnym ohodnotením 16 softvérových systémov a testovali ho počas vývoja projektu AFOTEC (Air Force Operational Test and Evaluation Center).

HPMAS dokážeme úspešne používať pri analýze udržovateľnosti systému pred a po zmenami. Ako príklad uvedieme systém, ktorý vytvorili v Hawlett-Packard. Tento systém analyzovali pomocou HPMAS pred vykonanou zmenou aj po zmene. Výsledky sú zobrazené v tab. 1. Z tabuľky vidíme, že index udržovateľnosti, hoci pribudlo 150 riadkov, zostal v podstate nezmenený (0.4% je zanedbateľný rozdiel).

**Tab. 1:** Analýza udržovateľnosti systému pred zmenou a po zmene.

Metriky	Analýza pred zmenou	Analýza po zmene	Percentuálny rozdiel
<b>Celkový počet riadkov</b>	1086.00	1235.00	13.40 %
<b>Počet modulov</b>	13.00	15.00	15.40 %
<b>Priemer cyklotrickej zložitosti</b>	226.00	255.00	12.80 %

<b>Index udržovateľnosti pre HPMAS</b>	88.17	88.61	0.40 %
--	-------	-------	--------

**Tab. 2:** Analýza udržovateľnosti modulov pred zmenou a po zmene.

Trieda	Analýza pred zmenou		Analýza po zmene		Percentuál. Zmena
	Modul	Index udržovateľ.	Modul	Index udržovateľ.	
<b>1</b>	A	93.83	A	93.83	0.0
	B	93.82	B	93.82	0.0
	C	92.96	C	92.96	0.0
	D	84.41	D	84.41	0.0
<b>2</b>	E	86.24	E	89.00	3.2
	F	65.58	F	67.27	2.6
	G	88.06	G	85.83	-2.5
<b>3</b>	H	78.41	H'	83.05	5.9
	I	72.85	I'	63.15	-13.3
	J	67.75	J'	66.43	-1.9
	K	68.83	K'	66.67	-3.1
<b>4</b>	L	80.68			
	M	78.78			
			N	85.08	
			O	80.75	
			P	79.68	
			Q	69.68	

Túto metódu aplikovali aj na jednotlivé moduly systému. Úspešne ju používali na analýzu jednotlivých modulov pred zmenou a po zmene. V tab. 2 sú znázornené jednotlivé moduly, ktoré sú rozdelené do 4 tried. V prvej triede sa nachádzajú tie moduly, ktoré po vykonaní zmeny v systéme zostali nezmenené, takže ich percentuálna zmena je nulová. Tieto moduly sú veľmi dobre udržovateľné. V druhej triede sa nachádzajú tie moduly, ktoré boli zmenené pritom však meno modulu zostalo nezmenené. Pri týchto moduloch percentuálna zmena je menšia ako 5%. V tretej triede sa nachádzajú tie moduly, ktoré boli zmenené a dostali aj iný názov. V poslednej triede sú tie moduly, ktoré nevieme „namapovať“ na žiadny predchádzajúci modul pred vykonaním testu. Moduly v posledných dvoch triedach sú veľmi ťažko udržovateľné, lebo ich štruktúra sa zmenila a ešte nie sú dobre otestované. Údržbári, ktorí poznali predchádzajúce moduly, nebudú poznať tie, ktoré sme vytvorili neskoršie.

### Polynomiálna regresná analýza

Je to štatistická metóda, ktorá používa regresnú analýzu na preskúmanie vzťahov medzi udržovateľnosťou softvéru a medzi softvérovými metrikami. Táto metóda je podrobne popísaná v [Coleman94]. Na to, aby sme dokázali vyjadriť udržovateľnosť touto metódou, potrebujeme

definovať vzorec, ktorý je funkciou jednotlivých metrík. Tento model používala aj spoločnosť Hawlett-Packard, kde softvéroví inžinieri vytvorili model na základe intuitívneho stanovenia vzorca a pomocou kalibrácie subjektívnymi hodnotami. Základný regresný model mal štyri parametre. Tento vzorec vyzerá nasledovne:

$$\text{Udržovateľnosť} = 171 - 3.42 \times \ln(\text{aveE}) - 0.23 \times \text{aveV}(g') - 16.2 \times \ln(\text{aveLOC}) + \text{aveCM}$$

kde *aveE* je priemerná hodnota námahy (average effort), *aveV(g')* je priemer cyklomatickej zložitosti jednotlivých modulov, *aveLOC* je priemerný počet riadkov a *aveCM* je počet komentárov pre jednotlivé moduly v softvérovom systéme.

Predbežné výsledky ukazujú, že tento model je príliš citlivý na veľký počet komentárov, čo predovšetkým v malých moduloch nevhodne zvýši ukazovateľ udržovateľnosti. Na odstránenie tohto efektu nahradíme hodnotu počet komentárov (*aveCM*) s percentuálnym podielom komentárov v zdrojovom kóde (*perCM*), a celú funkciu obmedzíme koeficientom 50. Finálny model, ktorý obsahuje štyri metriky a používa sa aj v súčasnosti, má nasledujúci tvar:

$$\text{Udržovateľnosť} = 171 - 5.2 \times \ln(\text{aveVol}) - 0.23 \times \text{aveV}(g') - 16.2 \times \ln(\text{aveLOC}) + (50 \times \sin(\sqrt{2.46 \times \text{perCM}}))$$

Tento model bol porovnaný s predhádzajúcim a rozdiel medzi indexami udržovateľnosti bol menší ako 1.4.

Polynomiálnu regresnú analýzu taktiež môžeme používať na porovnanie udržovateľnosti celých systémov alebo na porovnanie jednotlivých modulov jedného systému.

Systém, na ktorom bol testovaný tento prístup, mal 236000 riadkov zdrojového kódu, bol napísaný v jazyku C pre operačný systém UNIX. Na základe indexu udržovateľnosti môžeme systém zaradiť do jedenej z troch kategórií. Ak index udržovateľnosti je menší ako 65 systém alebo testované moduly sú veľmi ťažko udržovateľné. Ak index udržovateľnosti je medzi hodnotami 65 a 85, systém alebo moduly sú mierne udržovateľné. Ak táto hodnota je väčšia ako 85, testovaný softvérový systém alebo moduly sú veľmi dobre udržovateľné. Musíme si však uvedomiť, že tieto hranice vytvárali softvéroví inžinieri v Hawlett-Packard. Tieto hodnoty boli prispôbené k ich technickému prostrediu a boli vytvorené veľmi intuitívne. V tab. 3 sú porovnané dva softvérové systémy. Systém-1 bol získaný z externého zdroja a počas testovania mu priradili index udržovateľnosti 89. Systém-2 je ukázkovým príkladom firmy Hewlett-Packard, ukazuje ako má vyzeráť dobre udržovateľný systém. Hoci Systém-1 má väčšiu hodnotu udržovateľnosti ako 85, s porovnaním tohoto indexu s indexom systému-2 vidíme značný rozdiel, takže na základe analýzy je systém-1 ťažko udržovateľný.

**Tab. 3:** Porovnanie dvoch softvérových systémov podľa regresnej analýzy.

Metriky	Systém 1	Systém 2
---------	----------	----------

<b>Ohodnotenie systému</b>	Nízke	Vysoké
<b>Prostredie</b>	UNIX	UNIX
<b>Programovací jazyk</b>	C	C
<b>Celkový počet riadkov</b>	236275	243273
<b>Počet modulov</b>	3176	3097
<b>Celkový ukazovateľ udržovateľnosti</b>	89	123

### Ďalšie techniky na kvantifikovanie udržovateľnosti

Existujú aj ďalšie metódy na kvantifikovanie udržovateľnosti softvéru, ktoré sú popísané v [Coleman94] a [Munson92]. My tu iba spomenieme niektoré najvýznamnejšie metódy:

- *Agregátna metrika zložitosti* charakterizuje udržovateľnosť systému ako funkcia entropie.
- *Analýza základných komponentov* je štatistická metóda na zníženie kolineárnosti medzi bežne používanými metrikami zložitosti. Takýmto spôsobom dokážeme identifikovať a znížiť počet komponentov, ktoré využívame pri zostrojení regresného modelu.
- *Faktorová analýza* je ďalšia štatistická metóda, kde metriky sú ortogonalizované do nepozorovateľných základných faktorov, ktoré sa potom využívajú pri modelovaní udržovateľnosti systému.

V tomto príspevku sme chceli čitateľa oboznámiť základnými metódami a technikami, ktoré používame pri testovaní, pri prevádzke a pri údržbe softvérového systému. Uviedli sme základné metriky, pomocou ktorých vieme zistiť vlastnosti softvéru práve v týchto etapách.

Testovanie je veľmi dôležitou etapou, počas ktorého sa snažíme nájsť všetky chyby softvérového systému. Samozrejme nikdy nemôžeme povedať, že systém už neobsahuje žiadne chyby. Testovaním môžeme potvrdiť iba to, že softvér chyby obsahuje.

V súčasnosti udržovateľnosť a spoľahlivosť sú veľmi dôležitými charakteristikami softvérových systémov. Pomocou niektorých metód dôkážeme tieto charakteristiky namerať a výsledky môžeme používať pri ďalších rozhodovaní.

Softvérové systémy sa budú stále meniť. Je iba na nás, či navrhujeme dobre udržovateľné programy, ktoré budeme ešte dlho používať a meniť, alebo ťažko udržovateľné, ktoré po prvej zmene budú jednoducho nepoužiteľné.

## Literatúra

- [Bieliková00] BIELIKOVÁ M.: *Softvérové inžinierstvo. Princípy a manažment*. Bratislava: Slovenská technická univerzita, 2000. ISBN 80-227-1322-8.
- [Coleman94] COLEMAN, D. M. ET AL: *Using Metrics to Evaluate Software System Maintainability*. IEEE Computer, 1994, vol. 27, no. 8, s. 44-49.
- [Munson92] MUNSON J. – KHOSHGOFTAAR T.: *The Detection of Fault-Prone Programs*. IEEE Transaction on Software Engineering, Máj 1992, vol. 18, no. 5, s. 423-433.
- [Tian95] TIAN, J. – LU, P. – PALMA, J.: *Test-Execution-Based Reliability Measurement and Modeling for Large Commercial Software*. IEEE Transaction on Software Engineering, 1995, vol.21, no. 5, s. 405-414.
- [Varga99] VARGA, M.: *Meranie softvéru*. Bratislava: Ekonomická Univerzita v Bratislave, Fakulta Hospodárskej Informatiky, 1999. Diplomová práca.
- [Voas95] VOAS, M. J. – MILLER, W. K.: *Software Testability: The New Verification*. <ftp://ftp.rstcorp.com/pub/papers/ieeesoftware95.pdf>/voas95software.pdf. (2001).

## Meranie vyspelosti softvérového procesu

Radoslav Kováč

**Abstrakt.** *Príspevok sa zaoberá meraním vyspelosti softvérových procesov. Podrobnejšie sú rozobrané výhody merania procesov a dve štandardné metódy ohodnocovania procesov: Model vyspelosti procesu (Capability Maturity Model) a norma ISO/IEC 15504. Stručne sú charakterizované aj niektoré ďalšie prístupy a čím sa navzájom odlišujú.*

Meranie je možné použiť vo všetkých etapách životného cyklu softvéru pri určovaní kľúčových atribútov výstupov jeho jednotlivých etáp. Tieto údaje sú podkladom pre riadenie softvérového procesu s cieľom dosiahnutia požadovaných vlastností jeho výsledku. Z pohľadu manažmentu softvérového procesu je meranie dôležitým nástrojom, ktorý umožňuje získavať dôležité informácie o stave projektu a zvyšuje tak viditeľnosť celého procesu. Získané údaje tvoria nevyhnutnú spätnú väzbu pre riadenie zmien a neustále zlepšovanie procesu a kvality jeho výsledkov. Aký je ale účel všetkých týchto činností?

Cieľom každého zákazníkneho softvérového projektu by malo byť vytvorenie kvalitného produktu a služieb a dosiahnutie spokojnosti používateľa. Pri rozsiahlejších projektoch však už k dosiahnutiu tohoto cieľa často nestačí iba dobrý tím a použitie vhodných nástrojov a technológií.

Kvalita výsledného softvéru významne závisí od procesu, ktorým sa tento produkt vyvíja. Táto závislosť je pri softvérových výrobkoch o to významnejšia, že výsledok je vo svojej podstate zložitý, často jedinečný a dosť závisí od úsilia i schopností všetkých jeho tvorcov. Z týchto dôvodov sa sústreďuje stále viac pozornosti na vlastnosti výrobných procesov a na možnosti ich zlepšovania. Preto vo väčších organizáciách narastá význam manažmentu softvérových projektov a presného zadefinovania procesu vývoja softvéru.

Ako však identifikovať, sledovať a riadiť kľúčové vlastnosti softvérového procesu, ktoré najväčšmi vplývajú na kvalitu výsledného softvéru? Základný problém je v tom, že procesy rôznych organizácií sa od seba dosť odlišujú, a preto je na úrovni procesu zložité určiť takéto charakteristiky, ktoré by boli navyiac jednoducho merateľné. Preto sa

prakticky ukázalo byť užitočné ohodnocovať procesy na vyššej úrovni. Pri takomto ohodnocovaní sa kladie dôraz na identifikovanie konkrétnych činností v rámci procesu a spôsobov akým sa vykonávajú.

Ohodnocovanie procesov sa opiera o určenie súhrnnej charakteristiky nazývanej **vypelost' procesu** (angl. process maturity). Vypelost' procesu vyjadruje potenciál schopností organizácie úspešne vyriešiť projekt, zahŕňa možnosti ďalšieho rastu týchto schopností a pripravenosť organizácie na zmeny v záujme zlepšovania procesu. Z tohoto pohľadu "nezrelé" organizácie vykonávajú a riadia proces často formou improvizácie. V takom prípade nie je možné presnejšie predpovedať výsledky procesu a prípadné problémy môžu byť katastrofou pre projekt. Oproti tomu organizácie s vyspelým procesom plánujú, riadia a vykonávajú činnosti v rámci projektov podľa overených a fungujúcich pravidiel, pričom sa pokúšajú neustále tento proces zlepšovať.

Ohodnocovanie vypelosti procesu predpokladá, že poznáme charakteristiky ideálneho procesu, ktoré vplyvajú na kvalitu jeho výsledkov a neustále zlepšovanie tejto kvality. Z tohoto dôvodu metódy ohodnocovania procesov vychádzajú z konkrétneho referenčného modelu procesov a určujú nakoľko sa ohodnocovaný proces približuje tomuto ideálu. Ideálny proces by mal byť explicitne definovaný, jednoducho kontrolovateľný, riaditeľný, modifikovateľný a inštitucionalizovaný v organizácii (definovaním štruktúry, štandardov a pravidiel). Vykonávanie takéhoto procesu by malo byť efektívne a malo by zaručovať kvalitu výsledkov. Neoddeliteľnou súčasťou vyspelého procesu je používanie merania pri získavaní údajov o vykonávaní procesu. Čím presnejšie je proces definovaný, tým viac činností je možné sledovať a na základe nameraných údajov lepšie riadiť celý proces.

Ohodnocovanie softvérových procesov sa v praxi vykonáva podľa konkrétnych štandardov, ktoré často definujú súbor otázok, na ktoré je potrebné odpovedať (buď iba áno/nie alebo podľa príslušnej škály). Otázky sa snažia identifikovať aké činnosti sa v procese vykonávajú a akým spôsobom. Podľa odpovedí sa nakoniec organizácii prizná dosiahnutá úroveň vypelosti predaním certifikátu. Jednotlivé komponenty ohodnocovania procesov sú:

- **referenčný model procesu** - je model vypelosti procesu, vzhľadom na ktorý sa vykonáva ohodnotenie. Medzi najznámejšie modely vypelosti patria: CMM (Capability Maturity Model), model projektu SPICE (Software Process Improvement and Capability dEtermination), Bootstrap alebo Trillium. Modely vypelosti majú rozličnú štruktúru, zväčša však definujú podobné úrovne vypelosti, požadované vlastnosti, ciele a okruhy činností.
- **metóda ohodnotenia** - definuje postup, ktorým sa ohodnotenie voči referenčnému modelu vykonáva, ako sa ohodnocujú vlastnosti realizovaných procesov a ako sa určí výsledná úroveň vypelosti organizácie. Metóda definuje rozsah ohodnocovania a všetky etapy

tohoto procesu: plánovanie, získavanie údajov, analýza údajov, vyhodnotenie a zlepšenie. Metódy sú viazané na konkrétne referenčné modely. Najznámejšie sú: SCE (Software Capability Evaluation) vzhľadom na CMM, ISO/IEC 15504 (ISO štandard pre ohodnotenie softvérového procesu) vzhľadom na SPICE, ISO 9001 (ISO štandard pre manažment kvality).

- **nástroje** - predstavujú nástroje, ktoré sa využívajú v celom procese ohodnocovania organizácie. Medzi najčastejšie používané nástroje vo fáze získavania údajov patria dotazníky, zoznamy kritérií, rozhovory (interview) a skupinové diskusie.

Modely vyspelosti procesov boli vytvorené tak, aby našli čo najširšie uplatnenie a je ich možné použiť vo viacerých kontextoch:

- **ohodnotenie vyspelosti procesu** (angl. process assessment) - za účelom identifikovania silných i slabých stránok organizácie a zistenia možných smerov zlepšovania sa. Robí sa najčastejšie samotnou organizáciou pre jej vlastné potreby. Môže sa však vykonávať aj za asistencie externých audítorov a viesť k prideleniu certifikátu s priznaním úrovne vyspelosti.
- **ohodnotenie schopností** (angl. capability evaluations) - pri výbere najvhodnejšieho dodávateľa softvéru nás často zaujíma ako spĺňa naše konkrétne požiadavky. V takomto prípade sa robí ohodnocovanie vzhľadom na požadované schopnosti, ktoré zodpovedajú našim požiadavkám.

### Model vyspelosti procesu

V roku 1984 bol v rámci kontraktu amerického Ministerstva obrany založený Inštitút softvérového inžinierstva (angl. Software Engineering Institute, SEI) na univerzite Carnegie-Mellon v Pittsburghu. Jeho úlohou bolo a je zvyšovať úroveň praktizovania softvérového inžinierstva a zlepšovať kvalitu reálnych softvérových systémov. O dva roky neskôr sa ich práce začali sústreďovať na softvérové procesy, čo neskôr vyvrcholilo definovaním Modelu vyspelosti procesu (angl. Capability Maturity Model, CMM). Tento model bol ďalej rozširovaný a zovšeobecňovaný a okrem klasického modelu pre softvér (označovaný ako SW-CMM) vznikli aj ďalšie: People CMM, Software Acquisition CMM, Systems Engineering CMM a Integrated Product Development CMM.

Model vyspelosti procesu definuje 5 základných úrovní, ktorými prechádzajú organizácie zlepšujúce svoj softvérový proces. Tieto úrovne zároveň umožňujú porovnávať softvérové procesy rozličných spoločností a pre konkrétnu organizáciu predstavujú návod pre neustále zlepšovanie jej procesov. Nasledujú stručné charakteristiky jednotlivých **úrovní vyspelosti** (angl. maturity levels):

1. **Východzia** (angl. Initial) - proces je charakterizovaný ad hoc postupmi, bez jednotných formalizovaných procedúr. Úspech

projektov silne závisí od schopností jednotlivcov a je ťažko predpovedateľný. Na tejto úrovni sa nachádza väčšina organizácií.

2. **Opakovateľná** (angl. Repeatable) - riadenie projektov vychádza zo základov riadenia kvality, vykonáva sa manažment konfigurácií a zmien, plánovanie a odhad nákladov.
3. **Definovaná** (angl. Defined) - inžinierske i manažérske procesy sú dobre definované a dokumentované, celý proces je dobre viditeľný a umožňuje efektívne využívanie získaných skúseností pri nových projektoch.
4. **Riadená** (angl. Managed) - procesy sú kvantitatívne merateľné, vytvára sa databáza informácií o projektoch, vďaka čomu je možné sledovať, analyzovať a plánovať pridelovanie prostriedkov a napredovanie projektov.
5. **Optimalizujúca** (angl. Optimizing) - organizácia sa sústreďuje na identifikáciu svojich slabých a silných stránok za účelom neustáleho zlepšovania všetkých procesov. Sú snahy automatizovať zber údajov a vytvárať analýzy užitočnosti zmien a zavádzania nových technológií.

Úroveň vyspelosti priamo vplýva na dodržiavanie termínov, nákladov a kvalitu výsledkov realizovaných projektov. Vyspelejšie organizácie dokážu zrealizovať projekt za kratší čas, pretože sa dokážu vyvarovať prípadných chýb; lepšie odhadnú termín dokončenia, projekt len zriedka prekročí rozpočet a výsledok častejšie dosiahne požadovanú kvalitu.

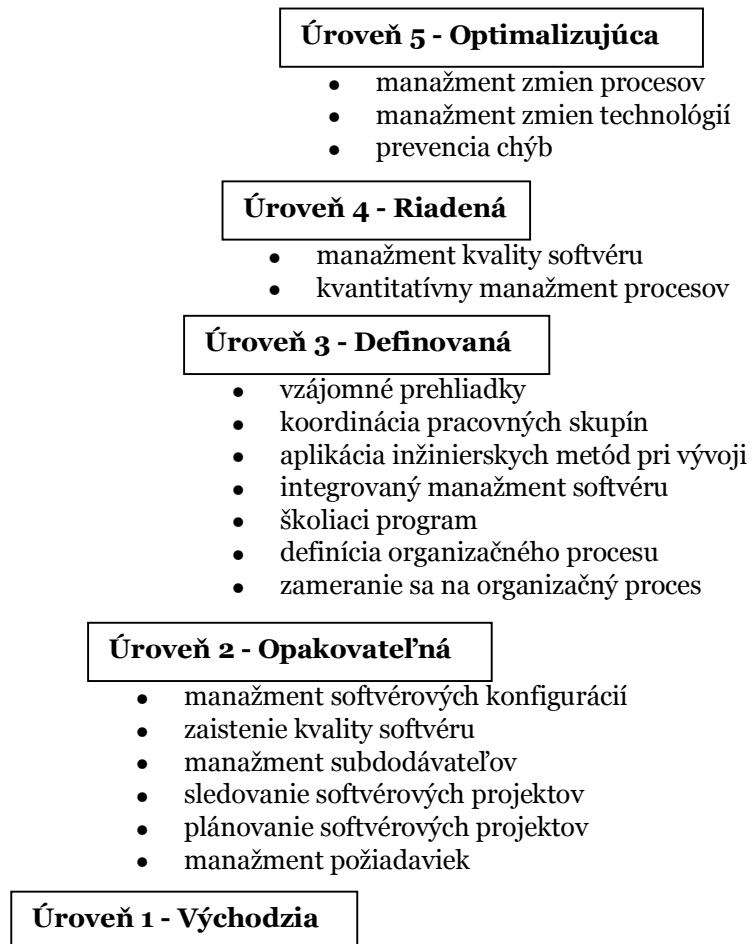
Už z charakteristík jednotlivých úrovní je zrejmé, že úloha merania narastá s vyspelosťou organizácie. Aké typy metrík sa používajú v jednotlivých úrovniach zachytáva tabuľka 1.

**Tab. 1:** Metriky prislúchajúce úrovniam vyspelosti (podľa [Shepperd95]).

Úroveň vyspelosti	Typ metrík	Príklady metrík
1. <b>Východzia</b>	základné	rozsah produktu, úsilie
2. <b>Opakovateľná</b>	projektové	funkčné body, plánované a aktuálne úsilie
3. <b>Definovaná</b>	produktové	počet modulov, cyklotmatická zložitosť, počet objektových rozhraní na otestovanie
4. <b>Riadená</b>	procesné + spätná väzba pre riadenie	úroveň znovupoužitia, miera dokončenia modulov v čase
5. <b>Optimalizujúca</b>	procesné + spätná väzba pre zlepšovanie procesu	(žiadne špeciálne odporúčania)

Spôsob, ktorým sú jednotlivé úrovne vyspelosti špecifikované, definuje základnú štruktúru Modelu vyspelosti procesu. Pre každú úroveň sú definované **klúčové procesné oblasti** (angl. Key Process

Areas) zahŕňajúce konkrétne skupiny činností. Vykonávaním týchto činností splní organizácia definované ciele modelu, vďaka čomu jej je možné priradiť schopnosti garantované konkrétnou úrovňou vyspelosti. Model vyspelosti procesu definuje spolu 18 kľúčových procesných oblastí prislúchajúcich jednotlivým úrovňam. Túto situáciu znázorňuje obrázok 1.



**Obr. 1:** Úrovne vyspelosti a ich kľúčové procesné oblasti.

Každá kľúčová procesná oblasť má definovanú skupinu **cieľov**, ktoré musí konkrétna organizácia splniť. Ciele vlastne jednoznačne definujú rozsah, ohraničenia a zámer danej procesnej oblasti. Spôsob, ktorým sa majú realizovať kľúčové procesné oblasti, je špecifikovaný skupinami **kľúčových postupov** (angl. Key practices) zoskupených do 5 sekcií podľa **spoločných črt** (angl. Common features): záväzok vykonávať, schopnosť vykonávať, vykonávané činnosti, meranie a analýza, kontrola vykonávania.

Jednou zo sekcií, do ktorých sú zoskupené kľúčové postupy pre jednotlivé kľúčové procesné oblasti, je teda aj meranie a analýza. Táto

sekcia zhrňa aké typy meraní sa vykonávajú v danej kľúčovej procesnej oblasti. Používanie merania je taktiež definované aj v niektorých činnostiach ostatných sekcií. Napríklad pre 3. úroveň vyspelosti je pri kľúčovej procesnej oblasti "aplikácia inžinierskych metód pri vývoji softvéru" definované meranie funkcionality (napr. vytváranie prehľadov alokovaných požiadaviek) a kvality (napr. počty, typy a závažnosti nájdených chýb) softvéru a s tým súvisiacich činností v rámci trvania projektu. Pre 4. úroveň vyspelosti je ako príklad pri oblasti "manažment kvality softvéru" uvedené meranie nákladov súvisiacich s nekvalitným softvérom a nákladov pre dosiahnutie vyššej kvality. Špecifikácia Modelu vyspelosti procesu [Paulk93] zväčša uvádza meranie súvisiace s naplánovanými činnosťami, so sledovaním ich vykonávania (najmä nákladov, úsilia a efektívnosti) a na vyšších úrovniach aj meranie súvisiace so sledovaním zlepšovania celého procesu.

Pri ohodnocovaní procesu konkrétnej organizácie podľa Modelu vyspelosti procesu je jedným z najzákladnejších nástrojov tzv. Dotazník vyspelosti (angl. Maturity Questionnaire). Dotazník sa podobne ako aj celý model vyspelosti zameriava na otázky súvisiace s činnosťami v rámci procesu. Je štruktúrovaný do 18 častí podľa 18 kľúčových procesných oblastí (asi 6-8 otázok na každú). Otázky sa zväčša zameriavajú na preskúmanie splnenia cieľov kľúčových procesných oblastí a netýkajú sa priamo jednotlivých kľúčových postupov. Odpoveď na každú otázku sa vyberá zo štyroch možností: *áno* (znamená, že postup je zavedený a konzistentne sa vykonáva), *nie*, *neviem* alebo *nie je aplikovateľné*.

Ohodnocovanie organizácie bolo najskôr vykonávané iba na základe analýzy získaných odpovedí na otázky v pôvodne vypracovanom dotazníku, ktorý je popísaný v [Humphrey87]. Pre priznanie príslušnej úrovne vyspelosti bolo potrebné kladne zodpovedať 90% tzv. kľúčových a 80% všetkých otázok pre danú úroveň. Prax však ukázala, že tieto otázky nepokrývali úplne všetky kľúčové procesné oblasti a odpovede nedokázali vždy odhaliť všetky dôležité charakteristiky procesu organizácie. Z tohoto dôvodu sa dnes používa nový Dotazník vyspelosti [Zubrow94], ktorý sa viac zameriava na kľúčové procesné oblasti Modelu vyspelosti procesu a používa sa len orientačne na získanie prvotnej predstavy o zavedených postupoch v činnostiach organizácie. Preto je možné pri každej odpovedi uviesť aj prípadný komentár pre audítora. Tabuľka 2 obsahuje príklady otázok Dotazníka vyspelosti.

**Tab. 2:** Príklady otázok Dotazníka vyspelosti (so zameraním na meranie).

<p><b>Úroveň 2</b> – Plánovanie softvérových projektov          Používa sa meranie pri určovaní stavu naplánovaných aktivít v rámci projektu (napr. dosahovanie míľnikov pre naplánované aktivity v porovnaní s plánom)?</p>
<p><b>Úroveň 3</b> - Aplikácia inžinierskych metód pri vývoji softvéru          Používa sa meranie pri určovaní funkcionality a kvality softvérových produktov (napr. počty, typy a závažnosť nájdených chýb) ?</p>
<p><b>Úroveň 4</b> - Manažment kvality softvéru          Porovnávajú sa výsledky merania kvality so stanovenými cieľmi pre kvalitu softvérových produktov za účelom zisťovania dosahovania cieľov kvality?</p>

## ISO/IEC 15504

Návrh normy ISO/IEC 15504 je výsledkom snahy o vytvorenie medzinárodného štandardu pre ohodnocovanie softvérového procesu. Táto norma vznikla v rámci projektu SPICE (Software Process Improvement and Capability dEtermination), ktorý bol zriadený organizáciami International Organization for Standardization a International Electrotechnical Commission (ISO/IEC) v roku 1993.

Tento štandard sa opiera o referenčný model softvérového procesu, ktorý je definovaný normou ISO/IEC 12207 (Software Life Cycle Process). Procesy sú rozdelené do 5 procesných kategórií (pričom pre každú kategóriu je definovaných 4 až 8 procesov):

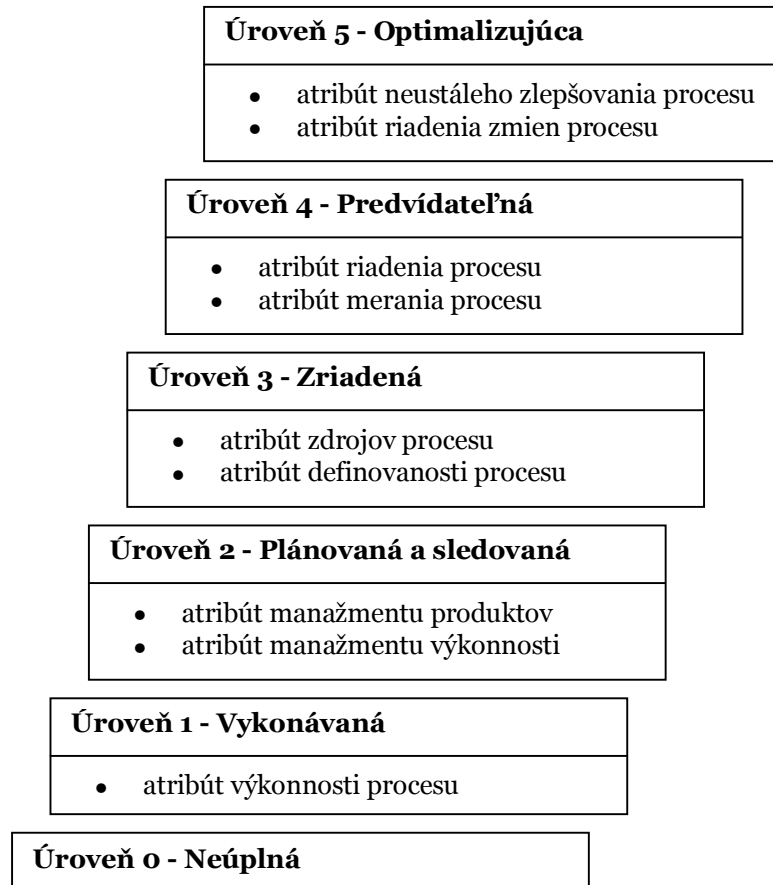
- **zákaznícko-dodávateľské** (angl. Customer-Supplier, CUS) - kategória zahŕňa procesy, ktoré priamo súvisia so zákazníkom: získanie softvéru, manažment požiadaviek, dodanie a prevádzka softvéru, zákaznícke služby.
- **inžinierske** (angl. Engineering, ENG) - zahŕňa špecifikáciu, návrh, implementáciu, testovanie a údržbu softvéru.
- **podporné** (angl. Support, SUP) - zahŕňa podporné procesy, ktoré sa vykonávajú v rámci iných procesov: vytváranie dokumentácie, manažment konfigurácií, zaistenie kvality, verifikácia a validácia produktu, vykonávanie prehliadok a auditov, analýza problémov.
- **manažérske** (angl. Management, MAN) - zahŕňa všeobecné postupy, ktoré sa používajú pri riadení projektov a procesov: manažment projektu, kvality, rizík a dodávateľov.
- **organizačné** (angl. Organization, ORG) - zahŕňa riadiace procesy, ktorými sa zabezpečuje dosahovanie obchodných cieľov organizácie: definovanie a zlepšovanie procesov, zabezpečovanie ľudských zdrojov a infraštruktúry.

Norma definuje 6 **schopnostných úrovní** (angl. capability levels), ktoré sú podobné úrovniam vyspelosti modelu CMM:

0. **Neúplná** (angl. Incomplete)
1. **Vykonávaná** (angl. Performed)
2. **Plánovaná a sledovaná** (angl. Performed)
3. **Zriadená** (angl. Established)
4. **Predvídateľná** (angl. Predictable)
5. **Optimalizujúca** (angl. Optimizing)

Pre každú z týchto úrovní (okrem 0.) sú určené **procesné atribúty**, ktoré musí ohodnocovaný proces splniť. Procesný atribút vlastne predstavuje merateľnú charakteristiku procesu, ktorá odzrkadľuje jeho vyspelosť. Každý atribút sa klasifikuje jedným zo 4 stupňov: *nedosiahnutý* (angl. Not achieved), *čiastočne dosiahnutý* (angl. Partially achieved), *zväčša dosiahnutý* (angl. Largely achieved), *úplne dosiahnutý* (angl. Fully achieved).

Pre každú schopnostnú úroveň je potrebné získať úplné dosiahnutie všetkých atribútov nižších úrovní a zároveň sa pridávajú jeden až dva nové atribúty, ktoré musia byť aspoň zväčša dosiahnuté. Spolu je definovaných 9 procesných atribútov. Napríklad 4. úroveň definuje procesný atribút merania, ktorý určuje do akej miery je vykonávanie procesu podporované používaním merania pri zabezpečovaní dosahovania cieľov procesu. Pre dosiahnutie 4. úrovne je potrebné získať stupeň tohoto atribútu aspoň "zväčša dosiahnutý". Obrázok 2 zachytáva nové atribúty, ktoré je potrebné získať pre priznanie konkrétnej schopnostnej úrovne.



**Obr. 2:** Požadované atribúty pre schopnostné úrovne.

### Porovnanie s ďalšími prístupmi

Okrem dvoch najznámejších metód na ohodnocovanie softvérových procesov, ktoré už boli prestavené v tomto príspevku, vzniklo ešte mnoho ďalších. Tieto metódy a štandardy často vychádzali z iniciatívy väčších softvérových spoločností, ktoré sa snažili prispôbiť existujúce metódy ohodnocovania procesov svojim potrebám a zohľadniť pri tom aj vlastné štandardy.

Rozličné metódy a štandardy na ohodnocovanie softvérových procesov sa odlišujú najmä:

- **štruktúrou referenčného modelu** - rôzne metódy definujú model ideálneho procesu rôzne. Procesy sú zoskupované do rozličných kategórií a sú im priradované rozličné vlastnosti, atribúty alebo ciele.
- **pokrytím procesných oblastí** - rôzne štandardy zahŕňajú procesy súvisiace s rozličnými etapami životného cyklu softvéru a taktiež

manažérske procesy. Požiadavky na jednotlivé procesy sú rozpracované často na rôznej úrovni podrobnosti.

- **metódou ohodnocovania** - ohodnocovanie sa robí rôznymi formálnymi i neformálnymi metódami. Najčastejšie vychádza zo spoločných diskusií a viac či menej sa pridržiava výsledkov získaných z odpovedí na otázky dotazníkov.
- **spôsobom priradenia výsledného ohodnotenia** - najčastejšie je výsledkom ohodnotenia priradenie konkrétnej úrovne vyspelosti procesu organizácie. Výsledkom však môže byť aj priznanie certifikátu, ktorý hovorí o splnení požadovaných kritérií. Výsledky ohodnotenia taktiež zahŕňajú odporúčania pre zlepšovanie procesu.

K ďalším známym metódam a štandardom na ohodnocovanie softvérových procesov patria:

- **sústava medzinárodných noriem ISO 9000** - definuje kritériá pre zabezpečovanie kvality výrobných procesov všeobecne. Konkrétne norma ISO 9001 pokrýva etapy návrhu, vývoja, produkcie, inštalácie a služieb súvisiacich s produktom. Smernice pre jej aplikáciu na softvérové procesy obsahuje norma ISO 9000-3. Norma ISO 9001 pozostáva z 20 klauzúl, ktoré definujú požiadavky na jednotlivé vykonávané procesy. Ohodnotenie organizácie vykonávajú špeciálne vyškolení audítori. Výsledkom ohodnotenia je pridelenie všeobecne akceptovaného certifikátu, ktorý zaručuje, že procesy vykonávané organizáciou spĺňajú definované požiadavky kvality.
- **Bootstrap** - táto metodológia ohodnocovania softvérových procesov vznikla v rámci projektu iniciovaného Komisiou Európskeho spoločenstva v roku 1989. Bootstrap priamo definuje svoj vzťah k normám ISO 9000. Procesy sú na najvyššej úrovni zoskupené do 3 kategórií: organizačné, metodologické a technologické. Pri ohodnocovaní sa využíva dotazník, pričom odpovede na otázky sú zo štvorstupňovej škály. Proces sa neohodnocuje ako celok, ale ohodnocujú sa zvlášť jeho jednotlivé atribúty. Pre ohodnocovanie atribútov sa používajú úrovne vyspelosti definované modelom CMM.
- **Trillium** - metóda bola vyvinutá konzorciom telekomunikačných spoločností na čele s Bell Canada. Je založená na modeli CMM. Výsledkom ohodnotenia je taktiež jedna z piatich úrovní vyspelosti, ktoré zodpovedajú úrovniam CMM. Trillium sa zameriava na odvetvie telekomunikácií a pokrýva širšiu škálu produktov (nielen softvér).

Jednotlivé metódy sa od seba vo viacerých črtách odlišujú a často je zložité určiť, ako by bolo možné vzťahnúť výsledné ohodnotenie jednej metódy na ohodnotenie inej. Napríklad požiadavky na certifikát ISO 9001 zahŕňajú niektoré oblasti, ktoré sa nenachádzajú v žiadnej z kľúčových procesných oblastí modelu CMM. K takýmto napríklad patrí poskytovanie služieb zákazníkovi alebo dodávka a inštalácia softvéru. Organizácie, ktoré získali certifikát ISO 9001 by nemali mať problém so

splnením väčšiny cieľov 2. a 3. vyspelostnej úrovne CMM. Avšak aj organizácia nachádzajúca sa na 1. vyspelostnej úrovni môže získať ISO 9001 certifikát.

**P**rocesná orientácia predstavuje veľmi významný prístup k zabezpečeniu kvality softvérových produktov. Vďaka mnohým skúsenostiam s úspešnými i neúspešnými softvérovými projektmi bolo možné vytvoriť referenčné modely procesov tvorby softvéru a zadefinovať metódy pre ohodnocovanie reálnych procesov voči týmto ideálnym referenčným modelom. Metódy ohodnocovania majú často rozličný charakter a používajú rôzne prístupy, pretože ohodnocované procesy sú veľmi rozdielne a tieto metódy sa snažia o vytvorenie celkového pohľadu na vlastnosti procesu. To je tiež dôvod, prečo je určovanie vyspelosti softvérového procesu také náročné a vytyka sa mu, že používané metódy sú subjektívne. Napriek tomu predstavuje ohodnocovanie vyspelosti procesu dôležitú techniku merania procesu, ktorá nám poskytuje cenné informácie pri vytváraní kvalitných softvérových produktov.

### Literatúra

- [Humphrey87] HUMPHREY, W. S. – SWEET, W. L.: *A Method for Assessing the Software Engineering Capability of Contractors (CMU/SEI-87-TR-23)*. Pittsburgh, Pa.: Software Engineering Institute, Carnegie Mellon University, 1987.
- [Niessink98] NIESSINK, F. – VLIET, H. VAN: *Towards Mature Measurement Programs*. In Proceedings of the Euromicro Working Conference on Software Maintenance and Reengineering, 1998. s. 82-88.
- [Paulk93] PAULK, M. et al.: *Key Practices of the Capability Maturity Model, Version 1.1 (CMU/SEI-93-TR-25)*. Pittsburgh, Pa.: Software Engineering Institute, Carnegie Mellon University, 1993.
- [Paulk95] PAULK, M.: *How ISO 9001 Compares with the CMM*. IEEE Software, Jan. 1995. s. 74-83.
- [Shepperd95] SHEPPERD, M.: *Foundations of Software Measurement*. Prentice Hall, 1995. 234 s. ISBN 0-13-336199-3.
- [SCE94] SOFTWARE CAPABILITY EVALUATION PROJECT: *Software Capability Evaluation Version 2.0, Method Description (CMU/SEI-94-TR-6)*. Pittsburgh, Pa.: Software Engineering Institute, Carnegie Mellon University, 1994. 210 s.

- [Werth94] WERTH, L. H.: *Lecture notes on Software Process Improvement. (CMU/SEI-93-EM-8)*. Pittsburgh, Pa.: Software Engineering Institute, Carnegie Mellon University, 1994. 31 s.
- [Zahran98] ZAHRAN, S.: *Software Process Improvement*. Addison-Wesley, 1998. 447 s. ISBN 0-201-17782-X.
- [Zubrow94] ZUBROW, D. ET AL.: *Maturity Questionnaire (CMU/SEI-94-SR-7)*. Pittsburgh, Pa.: Software Engineering Institute, Carnegie Mellon University, 1994. 57 s.

## Meranie produktivity

Ján Pidych

**Abstrakt.** Meranie produktivity je dôležitou súčasťou projektu. Tento príspevok opisuje hlavné dôvody, spôsoby a ciele merania produktivity. Venuje sa niektorým problémom, ktoré meranie produktivity sprevádzajú. Taktiež sú spomenuté niektoré modely odhadov nákladov na vývoj softvérových produktov. Pri opise procesu tvorby softvéru sú podrobnejšie opísané najčastejšie organizačné štruktúry.

Čo je produktivita asi všetci vieme, aj keď len intuitívne. Označujeme ňou množstvo výstupu za jednotku času, alebo nákladov. Ale, ako to je s produktivitou v softvérovom projekte? Čo ovplyvňuje produktivitu tvorby softvéru a ako ju môžeme zlepšiť?

Pre mnohých je meranie produktivity tvorby softvéru len ťažko predstaviteľný pojem a pozerajú naň s veľkou nedôverou. Napriek tomu je meranie produktivity veľmi dôležité. Pre lepšie pochopenie merania produktivity, by sme si mali položiť a zodpovedať nasledujúce otázky [Scacchi94]:

- Prečo merať produktivitu tvorby softvéru?
- Kto a ako má merať produktivitu tvorby softvéru?
- Čo sa má merať?

Vyjasnením týchto otázok, by meranie produktivity malo stratiť svoje „rúško tajomstva“ a mali by sme sa naň pozerieť ako na nedeliteľnú súčasť softvérového inžinierstva.

### Prečo merať produktivitu?

Dnes poznáme veľa dôvodov, prečo by sme mali merať produktivitu tvorby softvéru. Medzi najčastejšie spomínané dôvody patria : zistenie a porovnanie výkonnosti zamestnancov, zistenie možného urýchlenia projektu, či zníženie ceny vývoja. Lenže tieto údaje môžeme využiť aj ďalšími spôsobmi :

- lepšie odhadnúť množstvo práce, ktorú pracovníci úspešne zvládnu - určenie realistických plánov,
- vyhnutie sa najímaniu nepotrebných zamestnancov,
- skoršia identifikácia problémov počas vývoja,
- možnosť uplatnenia osobných odmiern, čo môže kladne motivovať zamestnancov,
- odhalenie medzier vo výkonnosti pracovníkov, ktorí môžu byť určení na školenia.

Ako vidíme dôvodov na meranie produktivity je viacero. V súčasnosti, rozhodujúcu zložku nákladov softvérových produktov, tvoria ľudské zdroje. Je preto nevyhnutné správne naplánovať ich využitie podľa očakávaného postupu riešenia projektu. Dôležitým faktorom je pritom práve produktivita.

### Kto a ako má merať produktivitu?

Voľba, kto má merať, závisí od dôvodov, respektíve dát, ktoré nás zaujímajú. Meranie môžu vykonávať sami členovia tímu, pomocou pravidelných správ. Takýto zber údajov je najjednoduchší a poskytuje informácie o osobnej výkonnosti pracovníkov. Pokiaľ sa na základe týchto správ rozhoduje o odmeňovaní, je potrebné vykonávať kontrolu správnosti údajov. Kontrola je dôležitá aj kvôli skutočnosti, že zamestnanci často podávajú správy, ktoré ich manažéri chcú počuť.

Inou úrovňou je meranie produktivity, ktorú vykonáva manažér tímu alebo projektu. Jeho hlásenia slúžia pre vyššie riadenie a zameriavajú sa na výkonnosť a postup práce z hľadiska celého projektu. Aj v tomto prípade, je nutná kontrola pravdivosti podávaných správ. Manažér sa takisto väčšinou snaží previesť projekt a postup prác v najlepšom svetle.

Druhou alternatívou k meraniu produktivity je využívanie externých pracovníkov, ktorí nie sú členmi tímov na vytváranie správ o výkonnosti tímu. Výhodou externých pracovníkov je objektívnejšie hodnotenie, nevýhodou vyššia cena – platíme ľudí navyše.

Na podporu merania je možné použiť podporné softvérové prostriedky. Sú najpresnejšie, najrýchlejšie a takisto najlacnejšie. Na ich nasadenie, musí vo firme existovať pevná štruktúra dokumentov a spôsobu ich archivácie. Všetky stretnutia, návrhy, výsledky a testy musia byť zdokumentované, aby softvérové prostriedky mali čo merať. Nedokážu však zachytiť „prácu pomimo“ – aktivitu na neformálnych stretnutiach, poskytovanie konzultácií a rád medzi zamestnancami.

Takisto je pomocou tohto prístupu problematické zachytiť „kvalitu výstupu“, často sa spracúvava iba objem práce. Na produktivitu taktiež vplýva okrem schopností pracovníkov množstvo ďalších faktorov – akosť procesu vývoja softvéru, spoľahlivosť a výkonnosť výpočtového prostredia, či náročnosť projektu [Bieliková00]. Mnohé z týchto skutočností nedokáže automatizované meranie zohľadniť a výsledkom sú opäť skreslené údaje. Preto je nevyhnutné, aby tieto generované správy dopĺňal človek.

Kvalita procesu merania produktivity súvisí s kvalitou samotného procesu tvorby softvéru. Proces merania produktivity je ovplyvnený používaním (respektíve nepoužívaním) CASE prostriedkov. Pre využívanie CASE prostriedkov pri vývoji je typické, že množstvo práce sa vykonáva pomocou výpočtovej techniky a navyše štandardným postupom. Toto umožňuje viesť prehľad o toku dokumentov a tým skvalitniť zber údajov. Na meranie produktivity je takto jednoduchšie nasadiť softvér, ktorý dané merania vykonáva automaticky. Formalizácia pracovných postupov ponúka použitie ďalších meraní, pretože máme štandardizované dokumenty z celého životného cyklu softvéru.

Pre úspešné aplikovanie merania produktivity musíme zabezpečiť také spôsoby merania, ktoré sú presné, opakovateľné a zachytávajú skutočný stav. Meranie produktivity je špecifické najmä preto, pretože ju nie je možné merať priamo. Produktivita je skôr manažérska, ako technická disciplína [Bieliková00]. Pri jej meraní treba používať aj vlastnú intuíciu a skúsenosti.

Preto sa odporúča otestovať navrhnutý systém merania v skúšobnej prevádzke. Takto môžeme otestovať, či proces merania neprodukuje skreslené údaje a či môžeme z nameraných údajov získať informácie ktoré požadujeme. Takisto je potrebné sa pripraviť na neúplne údaje. Nemôžeme predpokladať, že vždy získame všetky údaje. Ako si z navrhovaných opatrení môžeme všimnúť, proces merania neprebíha v ideálnom prostredí, ale dokýkajú sa ho rôzne problémy. Tak ako ostatných ľudských činnostiach. Preto je potrebné proces merania v čo najväčšej miere prispôbiť pomerom panujúcim v prostredí, v ktorom chceme meranie vykonávať.

### **Čo sa má merať??**

Keď sa už rozhodneme merať produktivitu, stojíme pred voľbou čo vlastne máme merať. Pred začiatkom vývoja softvéru je nutné navrhnuť metódy práce tak, aby meranie pokrylo čo najväčší objem prác projektu. Pokiaľ sa uspokojíme s meraním vyprodukovaných riadkov kódu alebo funkčných bodov, získame prehľad o produktivite a objeme prác programátorov, ale nebudeme vedieť nič o efektívnosti stretnutí tímu, o komunikácii v tíme, o testovaní, alebo o využívaných zdrojoch. Preto by sme do oblasti merania mali zahrnúť celý proces tvorby softvérového produktu, nielen samotný softvérový produkt.

## Softvérový produkt

Pri softvérovom produkte nás zaujíma najmä rozsah vyprodukovaných zdrojových súborov a množstvo chýb v nich. Na meranie rozsahu zdrojových súborov sa v súčasnosti používajú dva prístupy :

- meranie dĺžky textu programu
- meranie pomocou funkčných bodov

Tieto metriky tu nebudeme podrobne opisovať (pozri príspevok *Metriky v etape analýzy*), pripomenieme len ich základné charakteristiky.

Meranie dĺžky textu programu patrí k tradičným a najčastejšie používaným metrikám. Výhodou tohto prístupu je jednoduchosť, možnosť automatického sledovania a porovnania s historickými údajmi. Medzi hlavné nevýhody patrí závislosť od programovacieho jazyka, programovacieho štýlu a neposkytuje údaje o akosti programu.

Meranie s využitím funkčných bodov je technika nezávislá od programovacieho jazyka. Táto metrika sa používa aj pri odhade zložitosti softvéru. Najlepšie vystihuje databázovo-orientované systémy, nie je vhodná pre systémy so zložitým vnútorným spracovaním. Pre takého systémy sa používa technika feature points [Richta01], ktorá upravuje pôvodnú techniku funkčných bodov. Pri meraní produktivity sledujeme množstvo vytvorených funkčných bodov za jednotku času.

Z týchto metrick sa odvádzajú ďalšie sledované hodnoty. Medzi často používané patria: cena na riadok kódu (funkčný bod) a počet chýb na riadok kódu (funkčný bod) [Frakes96].

## Odhad úsilia a nákladov

S produktivitou súvisí aj odhad úsilia a času potrebného na vývoj softvéru. Modely odhadu nákladov rozdeľujeme podľa spôsobu odhadu do troch kategórií :

- odhady na základe odporúčaní expertov,
- odhady na základe analógie s už dokončenými projektami,
- algoritmické odhady.

Algoritmické odhady sú výhodne najmä pre ich objektívny odhad, pretože nie sú ovplyvňované ľudským faktorom [Boras096]. Jeden z často uvádzaných algoritmických modelov je model COCOMO (Constructive Cost Model). COCOMO definuje potrebné úsilie a čas [Masse97]:

$$PM = a \cdot KDSI^e \quad TDEV = c \cdot PM^d$$

$PM$  určuje prácnosť projektu v človekomesiacoch,  $KDSI$  veľkosť produktu v tisícoch inštrukcií (variant zdrojových riadkov),  $a$  je koeficient produktivity (pozri tab. 1),  $e$  charakterizuje zmenu nákladov v závislosti od veľkosti produktu (nelinearitu prácnosti a veľkosti, tab. 1).

*TDEV* je čas potrebný na vývoj, *c* a *d* sú parametre, ktoré sa určujú podľa charakteru projektu. Hodnota *c* sa uvádza 2.5, parameter *d* (tab. 1) závisí od módu vývoja.

**Tab. 1:** Hodnoty parametrov pre model COCOMO.

Mód vývoja	a	b	d
Organický	3.2	1.05	0.38
Prechodný	3.0	1.12	0.35
Viazaný	2.8	1.20	0.32

Módy vývoja vyjadrujú stupeň zložitosti systému [Bieliková00]. Organický mód je charakterizovaný relatívne malým tímom, ktorý pracuje na aplikácii zo známej oblasti. Predpokadá sa stabilný hardvér, známe algoritmy a malé náklady na komunikáciu. Veľkosť do 50 KDSI. Prechodný mód predstavuje stredný stupeň zložitosti systému. Sú kladené vyššie požiadavky na komunikáciu, čas a veľkosť systému. Veľkosť do 300 KDSI. Viazaný mód reprezentuje najzložitejšie systémy. Predpokladajú sa rôzne ohraňovania, zmeny požiadaviek počas vývoja, atď. Do tohto módu patria riadiace, interaktívne a operačné systémy.

Ďalšie prispôbenie modelu COCOMO na konkrétny produkt prebieha pomocou faktorov produktivity, napr. požadovaná spoľahlivosť, zložitost produktu, skúsenosti vývojového tímu a ďalšie [Varga99]. Vo fáze odhadovania, pred začiatkom implementácie sa počet inštrukcií (riadkov kódu) nahrádza funkčnými bodmi. Nevýhodou tohto nahradenia je skreslenie odhadu.

Z modelu COCOMO vychádzajú aj ďalšie modely ako napríklad TUCOMO, ktorý sa odlišuje len v hodnotách parametrov. Zmenené hodnoty sú uvedené v tabuľke 2 [Boras096].

**Tab. 2:** Hodnoty parametrov pre model TUCOMO.

Mód vývoja	A	b
Organický	2.74	0.76
Viazaný	6.85	0.66

Ďalšie modely odhadu využívajú funkčné body. Medzi tieto modely patria FPPROD (Function points productivity) a FPREG (Function points linear regression). Model FPPROD bol vytvorený na základe meraní 25 austrálskych organizácií v rokoch 1983 až 1987 na 112 projektoch. Úsilie vyjadrené v človekomesiacoch definuje vzťah :

$$\text{Úsilie} = FP / p$$

Kde FP je množstvo funkčných bodov, *p* je index produktivity. Pri meraniach projektov, ktorých implementácia prebiehala v jazykoch COBOL a PL/1 bol index *p* stanovený na 17.48 [Boras096].

Model FPREGR je štatistický model. Sleduje závislosť úsilia od množstva funkčných bodov. Úsilie je definované :

$$\text{Úsilie} = a + b * FP$$

Autori tohto modelu (B. Kitchenham, K.Käbsälä) sa rozhodli úsilie opísať lineárnou závislosťou. Hodnoty parametrov ( $a = 3.96$ ,  $b = 0.06$ ) získali lineárnou regresiou z údajov 40 projektov vyvíjaných 9 fínskymi organizáciami.

### Znovupoužitie

Pri tvorbe softvérového produktu zohráva výraznú úlohu znovupoužitie. Opätovné využívanie existujúcich modulov ovplyvňuje množstvo potrebného úsilia a času. Preto sa pri odhadovaní nákladov a meraní produktivity táto skutočnosť prejavuje použitím modifikovaných modelov merania (napríklad modifikovaný model COCOMO). V literatúre [Varga99], [Frakes96] publikované modely porovnávajú predpokladaný nárast produktivity pri znovupoužití s nákladmi, ktoré si znovupoužitie vyžiada :

$$C = (b + (E/n) - 1) * R + 1,$$

kde  $C$  je pomer celkových nákladov na vývoj produktu so znovupoužitím k nákladom bez znovupoužitia,  $b$  je pomer nákladov na znovupoužitie modulu k nákladom na vývoj nového modulu,  $E$  je koeficient, o ktorý sa náklady na vývoj modulu predrazia, ak má byť znovupoužiteľný (t.j.  $E > 1$ ),  $n$  je počet znovupoužití toho istého produktu, na ktoré sa rozloží investícia do jeho znovupoužiteľnosti,  $R$  je podiel modulov produktu, ktoré sú znovupoužité. Najmenší počet znovupoužití, pri ktorom je zisk zo znovupoužitia pokryje náklady :

$$N_o = E / (1 - b)$$

Určiť koeficienty pre tento model nie je jednoduché. Je náročné odhadnúť hodnotu  $R$ , pretože nie je jasné, či merať riadky zdrojového kódu (preloženého kódu), alebo pomer modulov. Pri parametri  $b$ , nie sú zrejmé ani položky, ktoré majú náklady obsahovať (nájdenie vhodného komponentu, oboznámenie sa s ním, jeho integrácia do aplikácie). Model sa dá ďalej rozšíriť o úsporu nákladov údržby, pretože znovupoužitie zvyčajne spojené s vyššou spoľahlivosťou.

Počas vývoja softvérového produktu je zaujímavé sledovať množstvo úsilia vynaloženého na prepracovanie modulov. Za prepracovanie považujeme všetko úsilie, ktoré sa vynaloží na module, od okamihu jeho prvého označenia za dokončený. Prerábanie modulov patrí medzi hlavné príčiny oneskorenia vývoja softvérových produktov a negatívne ovplyvňuje celkovú produktivitu.

### Proces tvorby softvéru

Proces tvorby softvéru taktiež ponúka niekoľko zaujímavých možností sledovania produktivity. Patrí medzi ne sledovanie efektívnosti

komunikácie (tímovej, medzitímovej, so zákazníkom), organizácie, alebo rozdelenia tímov.

Dôležitým aspektom tímovej spolupráce je komunikácia. Práve problémy v komunikácii spôsobili neúspech mnohých projektov [Brooks95]. Pre meranie komunikácie je nevyhnutné ju v čo najväčšej miere archivovať – zápisy s tímových porád, zo stretnutí so zákazníkom, zo stretnutí medzi tímami, či príspevky zasielané pomocou elektronickej pošty na rôzne elektronické konferencie. Pri meraní komunikácie môžeme sledovať širokú škálu úrovní od tímovej až po osobnú. Takto môžeme ľahšie lokalizovať miesto vzniku problémov. Zaujímavá je takisto možnosť sledovania komunikácie v rôznych časových mierkach (presnosť napríklad na dni). Mieru a efektívnosť komunikácie silne ovplyvňuje rozdelenie tímov a ich organizácia.

Organizácia práce je dôležitým faktorom, ktorý ovplyvňuje výkonnosť tímu. Organizačná štruktúra a komunikácia spolu úzko súvisia. Kvalitu organizačnej štruktúry je možné opísať nasledujúcimi dimenziami [Bieliková00]:

- *stupeň formalizácie* – vyjadruje do akej miery sú špecifikované, písomne vypracované očakávané prostriedky a výsledky práce,
- *stupeň centralizácie* – týka sa spôsobu delegovania právomocí na rozhodovanie a na vykonanie prác v rámci organizácie,
- *stupeň zložitosti* – závisí od počtu špecifických prác a od počtu organizačných jednotiek, resp. Oddelení.

Pre topológiu je dôležité, aby kopírovala tok informácií v projekte. Vhodná topológia projektu minimalizuje tzv. neformálnu topológiu, t.j. skutočný tok informácií sa minimálne odlišuje od formálnych komunikačných kanálov. V prípade nevyhovujúcej štruktúry narastá miera neformálnej komunikácie. Preto sa musíme snažiť prispôbiť organizačnú štruktúru požiadavkám vývoja projektu. Poznáme niekoľko variantov topológie :

- funkcionálna topológia,
- projektová topológia,
- maticová topológia,
- sieťová topológia.

Pri *funkcionálnej topológii* sa organizačné jednotky vytvárajú podľa špecializácie. Koordinácia projektu sa robí medzi jednotlivými funkciami. Pri tomto prístupe je výhodou efektívne využívanie zdrojov – úzka špecializácia na jednotlivých postoch. Na druhej strane je ťažko merať efektívnosť projektu a mieru príspevku jednotlivých pracovísk.

*Projektovú topológiu* tvoria oddelenia vytvorené na základe projektov. Pri realizácii rozsiahlych projektoch sa táto topológia kombinuje s funkcionálnou. Vtedy sa jednotliví členovia tímu uvolnia zo svojich pôvodných pozícií a stávajú sa členmi projektového tímu. Medzi

výhody takejto organizácie patrí pružnosť, súdržnosť (z toho vyplývajúca jednoduchšia komunikácia a koordinácia), jasné vzťahy a jednoduché meranie výkonnosti. Nevýhodou tohto prístupu je nízka špecializácia – experti nie sú využití počas celej doby projektu a duplicita zdrojov.

*Maticová topológia* vychádza zo snahy o kombináciu funkcionálnej a projektovej topológie. Táto organizácia sa snaží prevziať výhody a minimalizovať nevýhody predchádzajúcich organizácií. Pracovník je zaradený do funkcionálneho oddelenia a zároveň pridelený ku konkrétnemu projektu. Takýto spôsob poskytuje pružnosť pri reagovaní na zmeny, ale kladie vysoké nároky na komunikáciu. Výhodou maticovej organizácie je efektívne a účelné využívanie zdrojov a dobrá prispôsobivosť zmenám. Nevýhod je takisto viacero – pracovníci majú dvoch priamych nadriadených, náročné vymedzovanie právomocí a problémy pri prideľovaní zdrojov na projekty.

*Sieťová topológia* vznikla na základe požiadaviek charakteristických pre softvérové produkty – neustále zmeny požiadaviek veľkého množstva vzájomne sa prekrývajúcich projektov. Počas projektu sa projektové organizácie vyvíjajú, po skončení projektu sa automaticky nerozpadávajú, ale stávajú sa z nich trvalé organizácie. Takto vznikajú nové úlohy pre manažment pri riadení siete paralelne sa realizujúcich projektoch.

Pri pohľade na organizačnú štruktúru, môžeme merať nielen množstvo komunikácie, ale aj konkrétny tok dokumentov, vyťaženie špecialistov, či využitie technických prostriedkov. Sledovanie vyťaženie pracovníkov prebieha pomocou analýzy ľudských zdrojov. Využíva sa celá skupina výsledkových a výkonnových metrík (výkonnostne limity, čerpanie rozpočtu, a ďalšie). Pri technických prostriedkoch nás zvyčajne zaujíma ich vyťaženie, náklady na ich prevádzku, administráciu, alebo ich poruchovosť [Učeno1].

**M**eranie produktivity patrí medzi dôležité procesy softvérového projektu. Ak poznáme produktivitu jednotlivých elementov procesu môžeme ho ľahšie riadiť a tým zvyšujeme šancu na jeho úspešnú realizáciu. Musíme si uvedomiť, že produktivitu ovplyvňuje množstvo faktorov. Je to disciplína skôr manažérska ako technická. Neexistuje dokonalá metóda merania produktivity [Davis95]. Pri meraní produktivity treba používať intuíciu, skúsenosti a treba zohľadniť špecifiká projektu. Meraním je potrebné pokryť celý proces tvorby softvéru. Od tvorby produktu, cez komunikáciu, organizačnú štruktúru až po využívanie technických prostriedkov. Treba nájsť vhodný kompromis medzi objemom a kvalitou výstupu. Pracuje tím efektívne, keď výstupom je množstvo neefektívneho kódu? Jednotlivci nie sú produktívni, keď dosahujú osobné maximum, ale keď maximum dosahuje tím ako celok. Manažéri si takisto musia uvedomiť, že žiadna technológia nemôže nahradiť najpodstatnejšiu časť projektu – ľudský vklad. Pri zvyšovaní produktivity by sme sa všetci mali orientovať práve na lepšie využitie (často podceňovaných) ľudských zdrojov.

**Litaratúra**

- [Bieliková00] BIELIKOVÁ, M.: *Softvérové inžinierstvo*. Vydavateľstvo STU, 2000.
- [Boras096] BORASO, M. – MONTANGERO, C. – SEDEHI, H.: *Software cost estimation, technical report*. Università di Pisa. <http://citeseer.nj.nec.com/cache/papers2/cs/15728/ftp:zSzzSzftp.di.unipi.itzSzpubzSztechreportszSzTR-96-22.pdf/boras096software.pdf>. (25.5.2001).
- [Brooks95] BROOKS, FREDERICK P.: *The Mythical Man-Month: Essays on Software Engineering*. 2. vyd. Addison-Wesley, 1995. 336 s. ISBN 020135959.
- [Davis95] DAVIS, A.: *201 princípov tvorby softvéru*. McGraw-Hill, 1995.
- [Frakes96] FRAKES, W. – TERRY, C.: *Software reuse: Metrics and models*. ACM Computing Surveys, 1996, vol. 28, no. 2, s. 415-435.
- [Masse97] MASSE R. E.: *Software Metrics: An Analysis of the Evolution of COCOMO and Function Points*. <http://www.python.org/~rmasse/papers/software-metrics/>.
- [Richta01] RICHTA, K.: *Softwarové metriky*. Prednášky k predmetu Sofvarové inžinierství 2, ČVUT. <http://cs.felk.cvut.cz/~richta/www-si2/WS3.HTML>.
- [Scacchi94] SCACCHI, W.: *Understanding Software Productivity*. Advances in Software Engineering and Knowledge Engineering. D. Hurley (ed.), 1995, vol 4, s. 37-70. [http://www.usc.edu/dept/ATRIUM/Papers/Software\\_Productivity.html](http://www.usc.edu/dept/ATRIUM/Papers/Software_Productivity.html).
- [Varga99] VARGA, M.: *Meranie softvéru*. Bratislava: Ekonomická Univerzita v Bratislave, Fakulta Hospodárskej Informatiky, 1999. Diplomová práca.
- [Učen01] UČEN: *Metriky v informatice*. Praha: Grada Publishing, 2001.



## Meranie softvéru zákazníkom (meranie produktu)

Peter Agh

**Abstrakt.** *Merat' softvér má zmysel aj z pohľadu zákazníka. Zákazník sa podieľa na riadení projektu, potrebuje preto určité informácie, z ktorých niektoré sa získavajú meraním. Zároveň, zákazník – zákaznícka firma musí vedieť identifikovať a kvantifikovať prínosy zo zavedenia, resp. inovácie informačného systému. V oboch prípadoch sa pritom takisto merajú produkty životného cyklu, resp. samotný proces, ale z „iného“ pohľadu. Osobitosťou je, že sa (najmä pri hodnotení prínosov) berú do úvahy aj napr. „ekonomické údaje“ o zákazníckej firme, pretože sú nevyhnutné pre vytvorenie komplexného obrazu o firme a jej činnosti a objektívne určenie prínosov. Tejto tematike sa stručne venuje nasledovný príspevok, ktorý nepochybne je „z iného súdka“, avšak pre zaujímavosť sme sa rozhodli ho do našej publikácie zaradiť. Zamieriava sa najmä na meranie produktu (vytvoreného informačného systému), s cieľom určenia jeho prínosov pre zákazníka.*

**P**redchádzajúce príspevky boli venované tematike merania softvéru (či už meraniu produktov jednotlivých etáp procesu vývoja, resp. samotného procesu vývoja ako takého). Uvažovali sme pritom, že meranie sa prevádza na strane vývojára a slúži pre jeho internú potrebu. Našu publikáciu sme sa rozhodli doplniť aj týmto príspevkom, ktorý je z „iného súdka“ – je venovaný meraniu softvéru zákazníkom.

Zákazník, rovnako ako vývojár informačného systému (uvažujeme túto aplikačnú oblasť), tiež potrebuje určité údaje, charakterizujúce určité atribúty výsledného produktu (informačného systému), procesu vývoja produktu a prípadne aj medziproduktov ostatných fáz životného cyklu. Tieto môže využiť dvojakým spôsobom:

- Na jednej strane, zákazník je *spoluriešiteľom* projektu. Podieľa sa na *riadení* projektu, „leží na ňom zodpovednosť“ za niektoré kľúčové rozhodnutia. Preto potrebuje aktuálne informácie o priebehu

projektu, výsledkoch jednotlivých fáz atď. V súvislosti s meraním, mnohé údaje získané pri meraní procesu vývoja alebo meraní výsledkov jednotlivých fáz životného cyklu sú relevantné nielen pre vývojára, ale aj pre zákazníka. Príkladom je určenie kvality procesu vývoja.

- Na druhej strane, zákazník potrebuje určiť, aké *prínosy* má pre neho výsledok projektu. Zákazník musí sledovať a určiť, nakoľko výsledok projektu *zefektívnil* proces, pre ktorého podporu bol určený. Tieto informácie sú relevantné pre manažment zákazníka, ako údaj o návratnosti vložených investícií. Avšak, pri skúmaní návratnosti investícií do IT treba uplatniť iné postupy ako v „klasických“ metódach určenia návratnosti investícií, pretože treba zohľadniť niektoré špecifiká IT.

Treba upozorniť, že cieľom (predmetom) projektu nemusí byť len vytvorenie a zavedenie informačného systému (ďalej len IS), ale ním môže byť aj inovácia, rozšírenie pôvodného, prípadne sa môže jednať o určité služby, a pod (v ďalšom texte kvôli stručnosti hovoríme len o prínosoch vytvoreného/inovovaného IS).

Takisto treba spomenúť aj to, že uvedené kategórie majú „širšie využitie“. Napríklad, nameranie prínosov výsledkov projektu môže slúžiť nielen ako údaj o návratnosti investícií, ale zároveň nepriamo poslúžiť ako určitá „spätná väzba“ pre manažment zákazníka, napr. pre prehodnotenie vhodnosti a úspešnosti komunikácie a spolupráce zákazníckej firmy a dodávateľa. Podrobný výpočet všetkých možností však presahuje rozsah a zameranie našej publikácie.

### **Kto má merať?**

Meranie údajov môže vykonávať:

- buď priamo zákazník sám
- alebo sa údaje merajú na strane vývojára a zákazník ich len zbiera a aplikuje.

Prvý prístup sa uplatňuje najmä pri metrikách využívaných na určenie prínosov IT, druhý najmä pri metrikách používaných za účelom získania údajov relevantných pre podporu riadenia (napr. vyspelosť procesu).

### **Podpora riadenia a meranie prínosov**

V prípade meraní za účelom podpory riadenia namerané údaje slúžia ako podklad na rozhodovanie vedenia projektu zloženého zo zástupcov zákazníka a vývojára, spolu s ďalšími údajmi (ako napr. dodržiavaníu časových termínov). Meraním za účelom podpory riadenia sa bližšie nebudeme zaoberať – meranie zväčša vykonáva priamo tím vývojára, najpoužívanejšie metriky sme už opísali v predchádzajúcich príspevkoch a vyrátavanie možných metrick spolu s ich interpretáciou v rámci

možných stratégií riadenia a rozhodovania je osobitná tematika, ktorej rozsah a zameranie presahuje rámec našej publikácie.

V ďalšom texte sa sústredíme na metriky na meranie produktu. Rozlišujú sa dve významné skupiny metrík:

- *metriky použitia a metriky prevádzky IS*: *metriky použitia* merajú charakteristiky IS a efektívnosť informačnej podpory z používateľského hľadiska, *metriky prevádzky* merajú efektívnosť a výkonnosť prevádzky IS a využitie zdrojov.
- *metriky efektov zo zavedenia/ inovácie IS*, ktoré slúžia na vyjadrenie prínosov zavedenia či inovácie IS.

### **Meranie použitia a prevádzky IS.**

Metriky, ktoré sú komplexne zamerané na opis a hodnotenie úrovne poskytovaných informatických služieb, resp. informačnej podpory, sú obvykle včlenené do tzv. zmluvy o úrovni poskytovaných služieb – *Service Level Agreement (SLA)*.

SLA zahrňuje skupinu metrík. SLA je pojem, ktorý primárne vznikol ako nástroj na meranie úrovne služieb, ktoré poskytuje vytvorený informačný systém, resp. meranie úrovne „dodanej“ služby. Namerané hodnoty SLA potom podmieňujú výšku platieb, resp. uplatnenie sankcií vo vzťahu k dodávateľom (vývojárom). (podrobne [Corbet97]).

Treba si uvedomiť, že už *snaha presnejšej formulácie požiadavkov* na informatickú podporu zvyšuje pravdepodobnosť úspešnosti projektu. Zároveň, vyjadrená SLA determinuje „obtiažnosť“ projektu – celkové náklady na projekt.

Významnou črtou SLA je, že kategorizuje používateľov do skupín. Jednotliví používatelia nemajú jednotné požiadavky a preto je vhodné rozdeliť ich do (vhodne zvolených) skupín, ku ktorým možno uplatniť diferencovaný prístup pri rôznych udalostiach a podnetov od používateľov, napríklad:

- hlásení porúch
- požiadaviek na zmeny súčasného IS
- nových požiadaviek, teda na rozlíšenie definovanej úrovne poskytovaných služieb

Pre jednotlivé kategórie používateľov možno stanoviť štandardy HW, štandardné používateľské rozhranie, tréningové programy atď. Môže sa stanoviť odlišná interpretácia meraných parametrov pre jednotlivé kategórie používateľov. Príslušnosť k skupinám zároveň môže určovať aj prioritu v prípade kapacitných obmedzení (priradovaní zdrojov).

Príslušnosť používateľa k danej skupine nie je daná postavením v hierarchii firmy, ale v tom, do akej miery daný používateľ potrebuje informačnú podporu podnikového IS pre realizáciu svojej pracovnej činnosti.

Príklad kategorizácie používateľov: (príklad podľa [Corbet97])

- *komerční používatelia*: koncoví používatelia systému, využívajú IS priebežne. Vysoká závislosť na IS, „znesú“ však kratšie výpadky systému a menšie potiaže s obsluhou.
- *komerční používatelia 24*: koncoví používatelia systému, vyžadujú celodennú podporu informačným systémom. Vysoká závislosť od IS, „znesú“ však kratšie výpadky systému a menšie potiaže s obsluhou.
- *bežní používatelia kancelárskych aplikácií*: používatelia systému s nepravidelným prístupom k systému. Znesú aj dlhšie výpadky.
- *pokročilí používatelia*: napr. pracovníci technickej podpory, vývojoví pracovníci. Vysoká závislosť na IT. Vysoká závislosť na IS, „znesú“ však kratšie výpadky systému a menšie potiaže s obsluhou.
- *VIP*: komerční používatelia, kriticky závislí od IT podpory. Výpadky sú nežiadúce, vyžaduje sa ľahká obsluha a krátky vybavovací čas

Je potrebné stanoviť kategórie používateľov, definovať SLA, stanoviť procedúru eskalácie problémov a určenie štandardov.

V nasledujúcom odstavci uvedieme výpočet základných údajov a metrik SLA. Predtým však ale treba ozrejmiť pojmy *tvrdých* a *mäkkých metrik*, ktoré sa v štandardných opisoch SLA používajú. Ako *tvrdé* sa označujú objektívne merateľné metriky, *mäkké* metriky sú subjektívne merané, merané auditným spôsobom. Príkladom mäkkej metriky je *kvalita používateľskej príručky*, príkladom tvrdej metriky je *doba odozvy systému*.

SLA obsahuje nasledovné údaje, združené do niekoľkých kategórií :

*Základné podmienky a pravidlá:*

- Kategórie používateľov, počet používateľov v jednotlivých kategóriách
- Objem poskytovaných služieb (pomer počtu dokumentov/dokladov spracovávaných IS k celkovému počtu spracovávaných dokumentov/dokladov)
- Meranie- postup, spôsob, periodicita, spôsob prezentácie výsledkov
- Overovanie- postup, spôsob, periodicita, spôsob prezentácie výsledkov overovania správnosti merania
- Spôsob realizácie podpory (napr. na mieste, vzdialene)
- Náväzné podporné služby (napr. školenie)
- Cena, platobné podmienky
- Pravidlá pre zmenu služby
- Práva a povinnosti oboch strán (podmienky. súčinnosti)

**Tvrde metriky:**

- Dostupnosť (v % vyjadrený pomer času disponibility aplikácie na danom zariadení k celkovej pracovnej dobe, za určité časové obdobie)
- Bežná a maximálna prípustná (kritická) doba odozvy na požiadavky (pričom sa rozlišujú rôzne typy požiadaviek)
- Bežná a maximálna prípustná (kritická) doba riešenia požiadaviek (pričom sa rozlišujú rôzne typy požiadaviek)

*Mäkké metriky:*

- ostatné metriky pre danú službu (kvalitatívne ukazovatele typu „akceptácia“, „potvrdenie realizovaného školenia“, „hodnotenie používateľa“, „hodnotenie lektora školenia“, „hodnotenie účastníka školenia“ a pod.)

V súvislosti s tvrdými metrikami, ktoré sú súčasťou SLA, sa v niektorých prípadoch zavádzajú až tri úrovne parametra:

- *štandardná*: požadovaná hodnota parametra za štandardných podmienok
- *minimálna*: hranica, pod ktorú parameter *nikdy* nesmie klesnúť
- *motivačná*: „ideálna“, nadštandardná úroveň

Použitie týchto úrovní sa často vzťahuje na platobné podmienky (zачytené v platobnej zmluve) – nedosiahnutie štandardnej úrovne spôsobí nižšie platby, nedosiahnutie minimálnej sa považuje za „nedodanie“ a je dôvodom na uplatnenie (v zmluve definovaných) sankcií a naopak, dosiahnutie motivačnej úrovne je dôvodom na finančný bonus. Podrobnosti sa nachádzajú v platobnej zmluve.

Metriky prevádzky IS sú zamerané na jednotlivé aspekty hodnotenia efektivity prevádzky IS a efektivity využívania zdrojov. Príkladom metrick sú: počet chýb a zlyhaní za určité časové obdobie, priemerná doba odozvy na požiadavku, počet požiadaviek na zmeny za určité časové obdobie (požiadavky možno kategorizovať podľa ich závažnosti), náklady na zmeny (riešenie požiadaviek na zmeny) za určité časové obdobie, priemerné, minimálne, maximálne využívanie zdroja (zdrojov), definované ako podiel času, keď sa zdroj využíva k celkovému času, kedy je zdroj k dispozícii (sledované za určité časové obdobie), atď... (podrobne v [Učeň01])

**Meranie efektov zavedenia/inovácie IS.**

Snaha o efektívne vyčíslenie prínosov projektov zavedenia (implementácie), resp. inovácie IT do podniku sa objavila už pri prvých takýchto projektoch.

Chápanie prínosov zo zavedenia IT sa vyvíjalo v čase. V minulosti sa ako prínos chápala samotná *implementácia* IS. V súčasnosti sa však ako

prínos chápe *dosiahnutie meraných efektov* garantované dodávateľom. Inými slovami, kým v minulosti sa ako cieľ chápalo samotné vytvorenie IS, v súčasnosti sa ako cieľ chápe práve dosiahnutie špecifikovaných cieľov, pre ktoré bol daný IS vytváraný. Dôležité je, aby ciele boli špecifikované a merateľné.

Základné kritériá pre podporu rozhodovania a výber projektov zavedenia či inovácie IS do podnikov možno rozdeliť do troch kategórií:

- manažérske kritériá
- technicko-organizačné kritériá
- finančné kritériá

Každá kategória uplatňuje iný uhol pohľadu na projekt a má svoje vlastné metódy a kritériá pre hodnotenie investície do IS. Používajú sa pritom ako tvrdé, tak aj mäkké metriky (ďalšie príklady tvrdých metrík uvedieme v nasledovnom odseku).

Manažérske kritériá sú napríklad, či projekt podporuje strategické zámery, či projekt je v súlade s informačnou stratégiou, či je významný pre zaistenie konkurenčnej výhody, či má vysokú pravdepodobnosť dosiahnutia prínosov a efektov z výsledného IS, či splňuje požiadavky noriem kvality a či splňuje legislatívne požiadavky. Pre každé kritérium existuje viacero nástrojov a metód na jeho sledovanie.

Technicko-organizačné kritériá sú napríklad: funkčnosť, bezpečnosť, kompatibilita s existujúcimi systémami a cena. Zaraďujú sa sem aj možné riziká vyplývajúce z nasadenia IS (ako napr. možnosť nízkej výkonnosti systému, neposkytnutie služby na dostatočnej úrovni, obtiažna údržba, neodpovedajúca bezpečnosť, nespoľahlivosť a pod).

Finančné kritériá možno rozdeliť na: metódy, u ktorých sa ako kritérium hodnotenia prioritne berú *úspory nákladov*, metódy, ktorých kritériom hodnotenia je *vykazovaný zisk* a metódy, ktorých kritériom je *peňažný tok z investície*.

Podrobnejší opis spomenutých kritérií možno nájsť v [Učeňo1].

Účinným nástrojom pre sledovanie efektov zo zavedenia/inovácie IS sú metriky. Pre vytvorenie objektívneho „obrazu“ je nutné vhodne skĺbiť tvrdé a mäkké metriky. Vo všeobecnosti však neexistuje postup a model vhodný pre každý projekt. Možno stanoviť základný postup a základný súbor metrík, avšak vždy sa treba prispôsobiť špecifikám konkrétneho projektu a zákazníka.

Platí, že sa treba sústrediť na tie aspekty softvéru a oblasti firmy, ktoré majú na jej podnikanie zásadný vplyv. Komplexné meranie by bolo síce ideálne, no je časovo a finančne neuskutočniteľné.

Výber metrík a spôsobu ich interpretácie je dôležitým krokom, obzvlášť keď je dodávateľ (vývojár) zmluvne zainteresovaný na na efektoch z vývoja/inovácie IS. Príklady metrík uvádzame v nasledujúcom odseku.

Ward uskutočnil prieskum v predných britských spoločnostiach [Ward94], ktorý odhalil tri základné príčiny problémov objektívneho zhodnotenia prínosov zo zavedenia/inovácie IS:

- mnoho podnikových manažérov dáva prednosť pasívnej roli pri projekte (nerozumejú sa do IT, sú zaneprázdnení, a pod.)
- podnik používa na hodnotenie investícií do IT rovnaký postup ako pre hodnotenie investícií do iných oblastí, pričom však nebere do úvahy špecifiká IT, strategické dopady investície a pod.
- minimum podnikov prevádza dôkladnú dokumentáciu identifikácie a vyčíslenia prínosov. Pritom sa ale často sústreďujú len na niektoré relevantné atribúty.

Preto, pokiaľ podnik chce účinne sledovať a merať náklady a prínosy investícií do IS, mal by zaujať aktívny prístup k príslušnému projektu zavedenia/inovácie IS, s jasnou predstavou jeho cieľa a obsahu, vybavený metodikou a súborom metrík pre meranie výkonnosti. Pasivita a nejasné predstavy riadiacich pracovníkov firmy zákazníka znemožňujú efektívne meranie prínosov.

### **Príklady tvrdých metrík**

V nasledujúcej tabuľke uvádzame príklady tvrdých metrík slúžiacich na vyhodnotenie efektov (prínosov) zo zavedenia či inovácie IS. Časť týchto metrík je „neinformatického“, skôr „ekonomického“ charakteru. Je to prirodzené, pretože snahou je merať prínosy IS z pohľadu „ekonomickej činnosti“ zákazníka. Zároveň treba tieto údaje doplniť aj čisto ekonomickými ukazovateľmi firmy, ktoré síce s meraním softvéru nemajú nič spoločné, avšak sú takisto nevyhnutné pre vytvorenie komplexného obrazu o firme a posúdenie prínosov v jednotlivých odvetviach podnikania zákazníka (napr. porovnanie údajov o celkovom zisku z podnikania pred a po zavedení či inovácie IS, pričom údaje treba upraviť vzhľadom na koeficient inflácie). Uvedené metriky sú prevzaté zo [Učeňo1] (skrátene). Uvádzame ich spolu s pôvodnou kategorizáciou podľa *konkurenčných faktorov*.

<b>Konkurenčný faktor</b>	<b>Príklad metriky</b>	<b>Komentár</b>	<b>Cieľ</b>
<i>Riadenie vzťahov so zákazníkmi, predaj</i>	Doba odozvy na zákazníkovu požiadavku/sťažnosť		Min
	Doba reakcie na dopyt	Doba od obdržania dopytu až po odoslanie ponuky	Min
	Doba reakcie na objednávku	Počet dní od obdržania objednávky až po expedíciu	Min
	Dodacia spoľahlivosť	Oneskorenie expedície v dňoch, rozdiel reality a plánovaného termínu	Min
	Dodacia presnosť	Pomer objednané/dodané v merných jednotkách	Max
	Dostupnosť infraštruktúry firmy pre zákazníkov	udané ako podiel dňa, počas ktorého sú informácie o firme a jej produktoch dostupné (napr. prostredníctvom www) jej zákazníkom	Max
	Podiel e-commerce na celkovom predaji firmy	podporná metrika	Max
	Objem predaja	v Sk za určité obdobie	Max
	Náklady na ľudské zdroje	Priemerné osobné náklady za obdobie, náklady na rast kvalifikácie	Min
<i>Cena (výrobné náklady)</i>	Vlastné výrobné náklady	Tento ukazovateľ je ovplyvnený externými faktormi. Aj tak sa však zväčša doporučuje sledovať ho pri hodnotení efektu z inovácie IS, za predpokladu, že portfólio vyrábaných výrobkov je stabilné	Min
	Náklady predaja	Má veľký význam sledovať ho u projektov zameriavajúcich sa na elektronické obchodovanie	Min
	Produktivita práce		Max
	Stav výrobných zariadení	Pomer zdržaní výroby/predaja z dôvodov zlého technického stavu k celkovému počtu odpracovaných hodín	Min
	Využitie prostriedkov	Doporučuje sa sledovať pre vybrané prostriedky. Aj v súvislosti s IS.	Max

<i>Kvalita procesov a produktov</i>	Kvalita procesov	Počet nezhôd za obdobie (v prípade IS, sledujeme nezhody s IS pre zamestnancov i zákazníkov firmy)	Min
	Náklady na nekonformitu	Náklady na „zmetky“ v pomere k objemu výroby v Sk (za obdobie)	Min
	Pomer reklamácií k tržbám	Percentuálne vyjadrenie pomeru reklamácie/tržby	
<i>Schopnosť inovácie</i>	Počet novouvedených produktov	Počet nových alebo inovovaných produktov uvedených na trh za obdobie	Max
	Priemerná doba zavedenia nového produktu		Min
	Ekonomická návratnosť vývojových investícií	vzhľadom na IS (zavedenie, inovácia,...) toto predstavuje pre zadávateľskú firmu kľúčový údaj	Max
	Stav riešenia vývojových úloh	Sklyzy za obdobie	Min
<i>Pozícia na trhu</i>	Celkový počet zákazníkov	Zákazníci sa kategorizujú do skupín podľa veľkosti celkového obratu obchodu s nimi	Max
	Podiel na celkovom objeme trhu	Podiel z celkového objemu tržného segmentu daného oboru	Max
<i>Finančné výsledky</i>	Tržná cena akcie	Sleduje sa na vybranom kapitálovom trhu	Max
	Ekonomická pridaná hodnota	Ekonomický zisk	Max
	Hodnota pridaná trhom	Hodnota podniku- celkový investovaný kapitál	Max
	Stav konta	Debet v Sk za obdobie Kredit v Sk za obdobie	Min Max
<i>Ostatné</i>	Náklady na inštaláciu	výrobných zariadení, ale aj IS	Min
	Náklady na údržbu		Min

V tomto príspevku sme sa zaoberali tematikou merania z pohľadu zákazníka, špeciálne meraniu výsledného produktu – vytvoreného/inovovaného IS. Na produkte bolo možné merať dve skupiny charakteristík. Prvá skupina charakterizuje „vonkajšie“ atribúty produktu (chápaného ako „čierna skrinka“), ktoré sledovali napr. „ľahkosť“ použitia (pohľad používateľa), ďalej atribúty ako „spoľahlivosť“ či „efektívnosť“ a pod. (metriky použitia a prevádzky IS). Druhá skupina slúži na vyjadrenie jeho „užitočnosti“, t.j. sledujeme, nakoľko projekt spĺňa účel, pre ktorý bol určený (metriky efektov a prínosov IS). Uviedli sme súbor základných metrík. V druhej skupine sa medzi „jej“ metrikami

nachádzali aj metriky merajúce „ekonomické“ ukazovatele. Ich zaradenie je kvôli úplnosti prirodzené, veď sú nevyhnutné pre vytvorenie kontextu a komplexného objektívneho obrazu o činnosti zákazníka a prínosoch IS pre tieto činnosti.

Zaujímavosťou je, že existuje snaha docieľiť používanie týchto metrík v obchodných zmluvách. Vývojári (dodávatelia) sa často bránia a argumentujú tým, že dosiahnutie želaných efektov je predsa dané nielen kvalitou výsledného produktu, ale aj množstvom ďalších externých faktorov (napr. vzťahom zamestnancov zákazníka k používaniu IS pri svojej práci, atď...). To je nepochybne pravda, otázkou je rozumný výber metrík a spôsobov ich použitia (napr. priradenie vhodných váh).

Prax ukazuje, že uplatnenie spomenutých metrík (najmä merania efektov zo zavedenia/inovácie) so sebou prináša dva významné psychologické aspekty. Na jednej strane, pracovníci dodávateľa (vývojára) sú už od počiatku projektu motivovaní k orientácii na efekty u odberateľa. Zároveň ale, aj účastníci za odberateľa sú motivovaní na dosiahnutie definovaných efektov. Tieto sú exaktne definované a ich naplnenie je cieľom projektu. Zákazník sa musí aktívne podieľať na tvorbe projektu. Musí vedieť exaktnejšie vyjadriť svoje potreby. Preto, pokiaľ je tento prístup zavedený v rozumnej miere, môže byť pre vývojárov – softvérových inžinierov prínosom.

## Literatúra

- [Corbet97] CORBET, M.F.: *Redefining the Corporation. Bringing Order to a New Industry*. Outsourcing Leadership Forum, Outsourcing Institute. [www.outsourcing.com](http://www.outsourcing.com), 1995/1996.
- [Učeň01] UČEŇ, P.: *Metriky v informatice*. Praha: Grada Publishing, 2001.
- [Ward94] Ward, J.: *Information Systems – Delivering Business Value*. Manchester: BIT'94 Proceedings, 1994.

Bc. Peter Agh  
Bc. Stanislav Hrk  
Bc. Radoslav Kováč  
Bc. Dušan Lacko  
Bc. Szabolcs Molnár  
Bc. Ján Pidych  
Bc. Marián Šimo  
Bc. Michal Šrámka  
Bc. Dušan Šucha  
Bc. Marián Teplický  
Bc. Vladimír Trgo  
Bc. Zoltán Varga

**Meranie v softvérovom inžinierstve  
a eseje o manažmente softvérových procesov**

1. vydanie  
Náklad 1 výtlačok  
168 strán  
Rok vydania 2001