R A C I O N Á L N Í P R O J E K T O V Á N Í S L O Ž I T Ý C H S Y S T É M Ů

RNDr. Petr J i ř í č e k

Existují projekty mající proti jiným projektům větší neurčitost v předem prováděných odhadem řešitelského úsilí, doby řešení a jiných zdrojů. Dochází k tomu tehdy, jestliže se nový projekt liší od všeho, co jeho řešitelé vytvořili dříve. Takovým projektům také více hrozí neúspěch. Na něm se mohou podílet mimo jiné subjektivní příčiny - některé chybné názory a představy řešitelů. Článek ukazuje na pět dosti rozšířených takových mylných názorů.

Názory, se kterými budu polemizovat, nejsou zcestné za všech okolností. Jejich nebezpečí spočívá v tom, že nebyly dosud příliš vyvráceny, že se pokládají obecně za správné a že je metodikové někdy propagují. Pouze důkladná analýza některých vlastních neúspěchů mne oprávnila k tomu, abych přístupy z následujících pěti bodů prohlásil přinejmenším za diskutabilní.

1. Příliš dlouhý vývoj programového celku bez praktického používání

Jsou možné dvě strategie řešení projektu s velkou neurčitostí.

První spočívá v tom, že řešitel dostane dostatečně dlouhou dobu k řešení, po jejímž uplynutí se očekává hotový produkt již na dosti vysoké úrovni funkcí. Pokud se nepodaří takový produkt v této době vytvořit, prodlužuje se termín, zajišťují se další zdroje a podobně.

Druhá strategie spočívá v tom, že se řešitel snaží co nejdříve vytvořit něco, co bude hned sloužit uživateli, i za cenu extrémního zjednodušení řešení. Uživatel i všichni ostatní zúčastnění jsou si vědomi, že jde o první verzi a že se do budoucna počítá s dalším vývojem. Další rozvoj fungujícího systému probíhá ve stejném duchu - neustále se nové funkce zavádění do praxe, a to tak dlouho, pokud jsou k dispozici řešitelské kapacity a jiné zdroje.

První strategie bývá považována za velkorysejší, koncepčnější

a za přímější cestu k dlouhodobým cílům, kdežto na programové celky vzniklé druhým způsobem bývá pohlíženo jako na jakési slepence. Později ukážu, jak hluboce se v tomto hodnocení mýlíte.

2. Příliš detailní technické projekty

bývají požadovány metodikou, přestože v případě projektů s velkou neurčitostí jde o zřejmý rozpor s dialektikou poznání a praxe.

3. Přílišná snaha o perfektnost

doslova paralyzuje některé analytiky a programátory v jejich práci. V posledních deseti letech se s tímto jevem setkáváme i u starých praktiků, kteří poprvé mají pracovat s bankou dat. Vinu na tom asi mají jinak dobře míněné přednášky databankových odborníků. V nich se dovídáme, že databanky vyžadují úplně nový způsob myšlení, že databankové struktury je třeba projektovat velice opatrně, že se přitom dají udělat chyby, které už nikdy není možno napravit. Nač jsou potom jazyky pro popis dat, nač je nezávislost programů na datech, tak charakteristická pro databanky (jak se dovídáme v téže přednášce)?

4. Všechny funkce softwarového produktu je třeba navrhovat tak, jak to vyhovuje uživateli, ne tak, jak se lehce realizují na počítači

Tento názor implicitně avšak mylně předpokládá rozpor mezi oběma hledisky.

5. Snaha o obecnost

deformuje pouze určité typy programátorů, kteří ztratí hodně energie tím, že předpokládají další použití svých programů, modulů, procedur, navíc proti tomu, než žádá bezprostřední aplikace, na které zrovna pracují.

Dříve než přikročíme k polemice s těmito pěti názory, stojí za to, všimnout si jednoho jejich společného rysu: všechny jsou dobrými zásadami z jiných oborů obecné technické praxe. Odsud snad pramení přijímání těchto zásad i v tvorbě programů, aniž bychom si uvědomili specifika našeho oboru. Na technika je však kladen jiný požadavek, pro nás mnohem důležitější: požadavek nejzazší jednoduchosti řešení.

Svou protiargumentaci založím na třech principech: na zmíněném požadavku nejvyšší jednoduchosti řešení, na nezastupitelné roli uživatele při vývoji programových celků a na určitém názoru, jak mají vznikat obecné programy.

6. Polemika s body jedna až pět

Před rokem jsem na tomto semináři v diskusi citoval článek /1/ C.A.R.Hoara, ze kterého budu opět čerpat v následujících odstavcích. Článek se zabývá vnitřní rozporností známého pojmu "softwarové inženýrství". Protiklad mezi těmito dvěma slovy považuje za děsivý. Kombinace nového, ale již poskvrněného pojmu software se starou a váženou profesí inženýra ohrožuje svou rozporností: porovnejme ideály profesionálního inženýra s ideály, které přijali někteří programátoři.

Na příklad, "význačnou charakteristikou profesionála, ať už jde o lékaře, architekta nebo inženýra, je, že porozumí skutečným potřebám svého klienta nebo zaměstnavatele a to často mnohem lépe než ten klient sám. Má přitom tolik schopnosti a takové postavení, aby přesvědčil klienta, jaké jsou ve skutečnosti jeho zájmy a že je třeba opustit chiméry, které si klient ve svých představách vytvořil.

Potom profesionál na základě svých znalostí a zkušeností doporučí z řady známých a důvěryhodných technik ty metody a postupy, které v daných podmínkách dosáhnou žádaného účinku při nejmenších nárocích - při nejmenším obtěžování zákazníka, při nejmenších nákladech atd."

Když není rada přijata, "správný profesionál má dost profesionální integrity k tomu, aby na ten úkol nebo místo rezignoval".

Programátor často naopak "nerozpozná skutečné potřeby svého klienta, chce po něm především, aby si sám řekl, co chce. Ty nejkomplikovanější představy pak přivítá jako výzvu svému ostrovtipu. Mnozí programátoři ignorují známé postupy, které úspěšně použili jiní a raději dají průchod své vynalézavosti ..."

"Další charakteristikou profesionála, dobrého inženýra, je snaha snížit náklady a zvýšit spolehlivost svého výrobku". Snížení

nákladů a vzrůst spolehlivosti - to jsou zřejmě dva protichůdné požadavky. Inženýr "zpravidla shledává, že konflikt mezi oběma hledisky může být vyřešen pouze zachováním nejzazší jednoduchosti koncepce, specifikace, návrhu a implementace. ... Čím je projekt těžší, tím důsledněji je třeba trvat na nejzazší jednoduchosti řešení".

Jsou programátoři, kteří se "záměrně vyhýbají jakémukoliv zjednodušování. Příjemně je vzrušuje, když se angažují v projektech se složitostí mírně za hranicí jejich schopnosti porozumění a zvládnutí". Tento typ lidí může "jednou krásně uspět, ale při příští příležitosti třeba uvidí, že neexistuje způsob, jak rozlišit, co je mírně za, a co je totálně za touto hranicí".

Další věcí, na kterou musíme na začátku projektu s velkou neurčitostí myslet, je zapojení uživatele do řešení. Uživatel bývá dnes formálně jmenován do řešitelských týmů. To je jistě důležité, ale v souvislostech tohoto článku je především zajímavé, co bude uživatel v řešitelském týmu dělat. Jestliže s ním budeme neustále probírat, jak má výsledný produkt vypadat za tři roky, pak využijeme jen to, že z nás nejlépe ví, jak to dnes v jeho referátu chodí, nikoliv za tři roky. Pod zapojením uživatele proto rozumíme zapojení uživatele jako uživatele vznikajícího systému programů. Co nejrychleji mu dát aspoň něco, co mu bude sloužit.

Na příklad můžeme říct: "Na vaše oddělení přijde displej a my dva máme pro něj přupravit aplikaci. Rozdělme si to tak, že já ty programy napíši a ty je budeš používat. Cokoliv tě přitom napadne, to spolu probereme. A začneme tím, že ti během několika týdnů napíši textový editor, takže ten "Rozpis práce na příští týden", který každý čtvrtek rozesíláš kooperujícím oddělením, nebudeš muset pokaždé třikrát přepisovat kvůli dodatečným změnám jako dosud". To je na dlouhou dobu poslední náš návrh, co se bude dělat. Pak už se další náměty jen hrnou: Nešlo by, aby Rozpis práce navazoval na soubor zákazníků? (Ano, takový soubor je v agendě XYZ). A na soubor vozidel? (Tady je problém, dejme tomu nemusí být zrovna nasazen příslušný disk, vyřeší se to jinak, ovšem ne brzy). Atd. Nebo jsou-li našimi uživateli programátoři a analytici: Nemohl by překladač rozhodovacích tabulek připustit některé činnosti ještě před testováním zbývajících podmínek? Nemohl by překladač číselníkových modulů umožnit proměnnou délku odpovědi?

Zkušenost ukazuje, že tyto náměty jsou nenahraditelné. Řešitel sám nedokáže tak kvalitní dílčí cíle stanovit. Nevím proč, ale je to tak. Právě popsaným způsobem vznikají úspěšné produkty. Kromě toho při tomto způsobu komunikace uživateli daleko snadněji vysvětlíme nereálnost některých jeho představ. A má-li projekt nějaké větší konkrétně stanovené cíle, pak se touto cestou dají splnit. Nebo dokážeme argumentovat v případě, že se zjistí, že o ně nemá smysl usilovat. To tehdy, když byly formulovány s malou znalostí věci, což se zákonitě mohlo stát, jak učí teorie poznání.

Jestliže uživatel skutečně potřebuje funkci těžko na počítači realizovatelnou, je již obeznámený s dosavadním řešením natolik, aby společně s námi mohl uvažovat, jak svůj požadavek modifikovat. Pokud to jde, nalezneme spolu jiné, snazší a přitom také vyhovující řešení. Získáme nejen my, ale i uživatel sám, už tím, že jednodušší řešení je dříve hotovo. Dodržením zásady nejzazší jednoduchosti zachováváme dosavadní přehlednost programového textu a tedy také možnost snáze uspokojit nové požadavky uživatele v budoucnu.

Živý systém je dokonaleji vybaven pro svůj další rozvoj než krásně rozpracovaný projekt existující zatím jen v papírech.

Zásah do živého systému má svá úskalí, přesto se však tento způsob práce vyplatí. Je vždy třeba důkladně otestovat, zda jsem přidáním nové funkce neporušil funkce stávající, které by měly fungovat přesně stejně jako dříve, a to je pracné. Tím však také neustále prověřují již fungující celek. A jsem pro to dobře vybaven, protože systém není jen na papíře, ale funguje na počítači.

Druhým úskalím je skutečnost, že při větším počtu uživatelů nemohu zrušit atavizmy, přežívající funkce, které se dnes dají plně nahradit funkcemi novějšími, protože některý uživatel může starou funkci používat, tak jak se to kdysi naučil. Tento problém je skutečnou zátěží velkých firem známou jako břemeno kompatibility. Na úrovni našich produktů to ještě je snesitelné.

Každý zásah do živého systému musíme provádět promyšleně. Je třeba si přečíst celé pasáže programového textu, kterých se zásah týká. Jestliže odporuje duchu existujícího programu, pak příslušnou pasáž přeprogramujeme. Neustále, tak jako na začátku, usilujeme o nejzazší jednoduchost celkového řešení.

Mnohé z toho, co bylo dosud uvedeno, se dá promítnout do jiného hlediska, do problematiky speciálního a obecného v programování, viz /2/.

Vezměme si vyhraněnou situaci, kdy se můžeme rozhodnout vysloveně pro jednoúčelový speciální program anebo naopak pro program určený k obecnému použití.

Příkladem takové situace může být zpracování sociologického výzkumu. Jde o jednorázovou akci s nejasnými možnostmi podobných požadavků v budoucnu. Známe dotazník a požadované tabulky četností (kombinace otázek). Rozhodneme-li se pro speciální jednorázový program, je naše úloha poměrně lehká. Každou tabulku (z asi stovky nebo dvouset požadovaných) deklarujeme jejím jménem, každá otázka dotazníku má své jméno, čteme respondenta za respondentem, překontrolujeme přípustnost přečtených hodnot a jednoduše přičítáme podle přečtených indexů na správná místa jedničky. Nakonec všechny tabulky zpracujeme jednotnou subrutinou. Program je logicky velmi jednoduchý. Každá specialita se dá jednoduše naprogramovat. Např. jedna tabulka má být zpracována zcela jiným postupem než všechny ostatní - není nic lehčího.

U obecného programu musíme napřed přečíst parametry - popis dotazníků, zadání tabulek. Musíme navrhnout adekvátní způsob uložení těchto informací do paměti, což nemusí být jednoduché. Pak teprve čteme respondenta za respondentem a interpretací uložených parametrů provádíme vše co je třeba. Jedná se asi o desetinásobek práce ve srovnání s jednoúčelovým programem. A zbytečně:

Není totiž vůbec jisté, že se další podobné požadavky vyskytnou. Avšak i když se vyskytnou, dostaneme příště mírně jiný typ zadání, např. ještě nějaký nový typ otázek dotazníku, nějaký nový typ tabulky. Požadavky mohou být příště jiné také na základě výsledku řešení prvního problému daného typu. Vidíme, že obecný program nebyl přes naše úsilí koncipován dostatečně obecně.

Aplikujeme-li tedy na tuto situaci kritérium nejsazší jednoduchosti, musíme se rozhodnout pro jednorázový speciální program.

Přes jeho jednoúčelové použití se snažíme napsat takový program na dobré úrovni - přehledně, jasně, srozumitelně. Tím si vytváříme univerzálnější prostředek, než by byl obecný program. Každou další

a další podobnou úlohu sice programujeme znova - avšak většinou přebíráme části starších podobných programů, ve kterých se dobře vyznáme. Přitom pečlivě promýšlíme ty části, které jsou tentokrát nové.

Jestliže se původní předpoklad ukázal být pravdivým a skutečně nastala velká potřeba výpočtů daného typu, teprve pak se stává z neustálého programování téže úlohy skutečná zátěž - jednotvárná mechanická dřina. Teprve nyní je čas k napsání jednoho univerzálního programu. Ale teprve nyní jsme dospěli v poznání problematiky zpracování sociologických výzkumů tak daleko, že můžeme napsat kvalitní univerzální program. Jinak bychom to nemohli dokázat. Práce na prvním, druhém, na eventuálních dalších jednoúčelových programech nebyla zbytečná, byla to nutná příprava k vytvoření hodnotného softwaru. A byli jsme tímto postupem povinni vůči zadavateli této práce. Šlo o nejzazší jednoduchost řešení.

Nejsme vždy v této situaci, kdy máme volit mezi oběma krajnostmi. Ale i v malém, v našich běžných programech, neustále hrozí nebezpečí zbytečného zobecňování, zbytečného pamatování na budoucí odlišné použití. Nechráníme na různých místech slepé přípojky pro strýčka Příhodu - nadbytečné parametry podprogramů, rezervní výplňky v datových strukturách, nepoužívané boční externí vstupy do modulů apod. To vše je většinou zbytečný balast. Každá takové věc se dá přece do programu dodělat později, až to bude zapotřebí. Ovšem za předpokladu, že se v něm pak vyznáme. A na to bychom měli především dnes dbát. Nejzazší jednoduchost řešení a dobrá srozumitelnost programu jsou nejjistější zárukou a jedinou rozumnou možností pro eventuální pozdější zobecnění. Jsou to známky dobré profesionální úrovně.

Literatura

/1/ Hoare, C.A.R.: The engineering of software: a startling contradiction. Programing methodology, A Collections of Articles by Members of IFIP WG2.3, Springer-Verlag N.York 1978

/2/ Jiříček, P.: Generátory, přednosti a nevýhody této techniky, Metody programování počítačů třetí generace, Havířov 1975

/3/ Volák, J.: Úsporné programování, Metody programování počítačů třetí generace, Havířov 1976