Informační systémy

Specifikace, realizace, provoz

Jaroslav Král

Tato kniha vznikla s částečnou podporou grantové agentury ČR, grant 201/98/0532.

UNIX je technologická ochranná známka X/Open Company, Ltd.;

© Jaroslav Král, 1998
Obálka © Jan Hanuš, 1998

ISBN 80–86083–00–4

Obsah

Obsah

Obsah

Seznam obrázků

Seznam obrázků

Seznam tabulek

Úvod

Počítače stále více ovlivňují lidskou společnost. Používání počítačů je stále komplexnější a klade stále větší důraz na ukládání, vyhledávání a prezentaci informací. Základním prvkem využívání počítačů, nebo obecněji informačních technologií (IT), jsou softwarové produkty pokrývající tyto činnosti – informační systémy. Informační systémy (IS) jsou základním nástrojem změn organizace práce a zlepšení kvalitativních charakteristik podniků a organizací.

Kvalita informačních systémů a rozsah a kvalita jejich využívání do značné míry rozhoduje o úspěchu podniků i národních ekonomik. Moderní prostředky komunikace (např. Internet) značně rozšiřují možnosti a přínosy informačních technologií a zvláště informačních systémů. Informační systémy jsou technologickým základem revolučních společenských změn vedoucích ke společnosti nového typu – k informatické společnosti.

Informační systém je systém sběru, uchovávání, analýzy a prezentace dat určený pro poskytování informací mnoha uživatelům různých profesí. IS může a nemusí být podporován počítačem. IS často využívá automatizované (počítačem podporované) i neautomatizované („ruční“) způsoby práce. Volba optimální kombinace automatizovaných i neautomatizovaných činností je základním problémem návrhu IS.

IS musí disponovat prostředky sběru, kontroly a uchovávání dat. Data (v počítači posloupnosti nul a jedniček) musí být zobrazitelná ve formě srozumitelné uživatelům a použitelné pro činnosti uživatelů, musí poskytovat informaci. Informace závisí na znalostech a ne vždy zcela známých potřebách konkrétních uživatelů. Pro neškoleného pracovníka je dílenský výkres nesrozumitelná čmáranice. Jiné informace potřebuje skladník, jiné ředitel. Skladníka zajímá skladový list, ředitele trendy rozsahu zásob ve skladech. Různorodost, nejasnost a proměnlivost potřeb je dalším obtížným problémem budování IS. Zavedení IS může znamenat změnu náplně nebo dokonce zánik pracovních míst, změnu v hierarchii moci v podniku a v budoucnu pravděpodobně i v celé společnosti. S tím jsou spojeny problémy, na jejichž řešení nestačí pouze znalosti informačních technologií.

IS obvykle slouží jisté komunitě lidí –koncovým uživatelům, kteří s IS přímo pracují. IS je navržen jako nástroj podporující jisté činnosti. Funkce IS musí být podporovány vhodnými technickými prostředky, u automatizovaných funkcí hardwarovým vybavením a softwarem. IS nemusí nutně využívat automatizované činnosti. V tomto textu budeme předpokládat, že všechny diskutované IS obsahují automatizované činnosti, tj. alespoň zčásti pracují na počítačích.

Informační systémy pracují obvykle s velkými objemy dat, která jsou přístupná mnoha koncovým uživatelům současně. IS podstatně ovlivňují pracovní procesy i organizační strukturu podniků. Zkvalitňují operativu

(účetnictví, vedení skladové evidence, finanční operace, řízení výrobních procesů atd.), ať se jedná o banku, úřad či výrobní podnik. Data získaná na operativní úrovni spolu s informacemi dosažitelnými na veřejných sítích umožňují podstatně zkvalitnit práci managementu, zlepšit marketing a zkvalitnit činnost podniku a tím dosáhnout strategické výhody před konkurencí.

Organizaci používající IS budeme nazývat uživatelem nebo zákazníkem. Koncoví uživatelé jsou konkrétní osoby, které se systémem pracují nebo budou pracovat.

Vývoj a nasazení IS má řadu specifických rysů. IS nelze obvykle koupit a bez podstatnějších úprav použít tak, jak to známe např. u operačního systému WINDOWS 95. IS je často nutné vyvíjet od počátku. I u kupovaných informačních systémů je nezbytné informační systém v procesu zvaném customizace přizpůsobovat potřebám a požadavkům uživatele. V obou případech je třeba provádět analýzu potřeb a formulovat požadavky zákazníka. Formulaci požadavků na vlastnosti IS není obvykle schopen v dostatečné kvalitě provést budoucí uživatel. Je tedy nutné, aby přesnou specifikaci požadavků formuloval dodavatel systému. To není možné bez úzké spolupráce se zákazníkem.

Pro úspěch informačního systému je třeba řešit řadu problémů typických pouze pro IS, jako je potřeba mocenských změn v organizační hierarchii zákazníka po zavedení IS, zjišťování jeho skutečných potřeb, kontakty s těmi, co budou IS skutečně používat, zájem a podpora managementu zákazníka atd. Řešení těchto problémů spolu s případným rozhodnutím vyvíjet systém od začátku na míru nebo koupit customizovaný IS je velmi komplikovaná záležitost, jejíž nedostatečné řešení je jednou z hlavních příčin neúspěchů IS. Při řešení těchto problémů jsou potřeba specifické znalosti z oblasti týmové spolupráce, teorie organizace atd.

Problémy a techniky vývoje, zavádění i provozu IS musí být řešeny ve spolupráci dodavatele se zákazníkem. Je tedy nutné, aby základní znalosti a techniky používané při vývoji, customizaci a provozu IS byly srozumitelné i pro zákazníka. Zvláště je to patrné při specifikaci požadavků na funkce IS. IS je tedy do jisté míry vždy společným dílem dodavatele i zákazníka. To je rys, který není přítomen u jiných typů softwaru.

Základním problémem IS je formulace odpovědi na to, *proč* (s jakým cílem) je IS zaváděn. Při formulování cílů a důvodů zavádění IS a při podrobné formulaci požadavků (odpověď na otázku *co*) je nutné aplikovat řadu technik, které zahrnují kromě technologií používaných při vývoji všech softwarových systémů i zcela specifické obraty a metody. Pracnost počátečních etap vývoje IS (stanovení cílů, formulace požadavků) je pro IS daleko vyšší než u jiných typů softwaru. Tato skutečnost si vynucuje použití řady specifických technik vývoje resp. customizace IS.

Vývoj (výroba) softwaru včetně IS je technicko-organizační problém, pro jehož řešení byla vyvinuta řada technologií a know-how, které jsou systemizovány a rozvíjeny ve vědecko-technickém oboru softwarové inženýrství. Níže se budeme zabývat problémy vývoje, instalace a provozu IS z hlediska tohoto oboru. Důraz bude kladen na metody specifické pro IS. Vzhledem k významu počátečních etap vývoje IS a problémům s jeho instalací a provozem je důraz kladen na problémy stanovování cílů, specifikace požadavků softwarových systémů a také na problémy marketingu, spolupráce se zákazníkem (volba zákazníka, analýza rizik, organizace společných týmů) a některé sociálně společenské problémy, jako jsou počítačové nemoci z povolání. Jsou diskutovány i problémy volby hromadně dodávaných IS a jejich customizace.

Počátečními etapami vývoje a nasazení IS se zabývají kapitoly 1 až 12. Poněvadž se počátečních etap musí účastnit i pracovníci zákazníka, jsou informace zde obsažené důležité pro managementy softwarových firem i zákazníků, pro analytiky a informatiky obou stran a do značné míry i pro koncové uživatele.

Kapitoly 13 až 21 (kódování, softwarové metriky, softwarové procesy, softwarové normy, CASE systémy) jsou důležité pro informatiky, základní poznatky (využívání metrik a norem, požadavky na dokumentaci a procesní po-

hled na tvorbu softwaru, využívání CASE, problém intenzity zátěže při zavádění IS aj.) jsou nutné pro management a pracovníky obou stran. Kapitola 21 obsahuje studii jedné úspěšné realizace IS ve strojírenské výrobě.

U čtenářů předpokládáme úroveň počítačových znalostí obvyklou u studentů vyšších ročníků přírodovědných, ekonomických a technických směrů vysokých škol. Postačují i praktické zkušenosti s používáním informačních technologií.

Tento text se zabývá průmyslovou výrobou softwaru. Týká se tedy déle existujících softwarových firem, které podstata věci nutí dodržovat termíny, realizovat software většími týmy, provádět marketing a přísně kontrolovat práci všech svých zaměstnanců a neváhat použít případné postihy. Mnoho níže uvedených poznatků je důležitých i pro samostatně pracující programátory (např. při jednání se zákazníky, využívání standardů, používání metrik při kontrole kvality softwaru, počítačové nemoci z povolání aj.)

O autorovi

Profesor Jaroslav Král, DrSc., přednáší na Fakultě informatiky MU Brno a na Matematicko-fyzikální fakultě UK Praha problematiku vývoje informačních systémů. Prof. Král vystudoval Matematicko-fyzikální fakultu UK Praha v r. 1959. Od té doby pracuje jako informatik. Vedle teoretických prací, pojednávajících o formálních jazycích, stavbě kompilátorů a simulacích, se jako analytik a vedoucí projektu účastnil řady projektů zaměřených na kompilátory, makroprocesory, systémy přímého řízení procesu a výroby a vytvořil několik podnikových informačních systémů.

1

Software – technicky náročný (hi-tech) výrobek

Vývoj softwaru jako základní podmínky používání informačních technologií je komplikovaný úkol, pro nějž se stále ještě vytvářejí základní techniky a metody. Problémy jsou již v etapách vývoje softwaru, které bychom v klasické terminologii mohli nazvat předprojektová příprava (formulace odpovědí na otázky *proč* a *k čemu*). Situaci komplikuje i šalebné přesvědčení, že počítače pomohou jaksi samy od sebe, bez pečlivých analýz a úsilí obvyklých při instalaci klasických technologií, kde většinu otázek řeší technici a obsluha je věcí dělníků a technologů. IS však používá a tedy v jistém smyslu „obsluhuje“ i management. Management tedy musí plnit úkoly, na které není u klasických technologií zvyklý. Chce-li IS používat, musí se ho naučit používat. Je tedy v jistém smyslu ve stejné situaci jako poslední skladník. Musí si udělat čas na školení a lecos neobvyklého se naučit. Není také snadné přijmout zásadu, že dobré fungování a přínosy IS jsou výsledkem společné kvalitní práce všech uživatelů IS, managementu i řadových úředníků, kteří se tak stávají do jisté míry partnery managementu. Poněkud příznivější je situace v pozdějších etapách vývoje softwaru (řešení otázek *co* a především *jak*). Souhrn metod a zásad vývoje softwaru, jeho instalace a udržování v provozu je obsahem vědecko-technického oboru softwarové inženýrství (SI). Základní filozofií SI je pohled na tvorbu softwaru a jeho používání jako na výrobně-technický problém průmyslového charakteru.

Níže probereme problematiku informačních systémů z pohledu softwarového inženýrství. Budeme při tom vycházet z praktických zkušeností autora z vývoje řady informačních systémů a jiných softwarových produktů, které se svými kolegy nejen vyvíjel, ale také obvykle dokončil.

Vývoj softwaru je velmi nákladná činnost. Kopie softwaru se vytváří téměř zdarma a ani instalace softwaru nebývá obvykle příliš nákladná. Přesto jsou ceny softwaru vysoké a náklady na nákup softwaru převyšují náklady na hardware. Vývoj softwaru je do značné míry koncetrován do několika málo mamutích firem. Vysoké náklady na vývoj softwaru jsou důsledkem řady faktorů. Efekty informačních technologií jsou velmi často nepřímé.[1] Efektivní využívání IT vyžaduje vysokou kvalifikaci na straně dodavatelů softwaru a v případě IS i jeho uživatelů. Software je abstraktní objekt, který je jen obtížné zobrazit vcelku. Projekt letadla i jeho částí se může opírat o relativně snadné představy celku. Prostředky umožňující vidět softwarové systémy jako celek se teprve vytvářejí (viz kap. 11 a 12)

1. Příklad: Při prezentaci zkušeností se zavádění systému R/3 na konferenci System Integration ’97 bylo za hlavní přínos označeno zlepšení kvality dat, nikoliv tedy zlepšení ekonomických parametrů.

1 Software – hi-tech výrobek

Rychlé změny vlastností hardwaru si vynucují změny v technologiích využívání informatiky a změny metod vývoje softwaru. Úspěch informačních systémů závisí i na tom, do jaké míry dodavatelé informačních technologií dokáží zvládnout problematiku oboru, pro který je IS vyvíjen.

1.1 Problémy vývoje informačních technologií. Princip analogie

Vědomí, že realizace, instalace a používání informačních technologií je technický problém, usnadňuje samo o sobě řešení řady problémů. Mnohdy totiž stačí uplatnit při řešení nějakého problému (marketing, jednání se zákazníkem, možné sociální a jiné důsledky nasazení IS, vyhodnocování rizik, předprojektová příprava) obdobné obraty a techniky jako v jiných technických oborech. Je např. známo, že každá intenzivní práce (a práce s počítači je náročná) vede obvykle k zdravotním problémům. Lze tedy očekávat, že i informační technologie a IS zvláště přinášejí riziko vážných zdravotních problémů (kap. 4). Mnohdy tedy stačí uplatnit známé postupy – v případě pochybností, zda se postupuje správně, stačí často odpovědět na otázku, jak ten který problém řeší inženýři klasických technických oborů. Tuto zásadu budeme nazývat *princip analogie*.

Nedodržování známých postupů přípravy a řízení softwarových projektů, nevyužívání analogie (např. správné formulace a prověřování správnosti odpovědi na otázku *proč*) vede k rozsáhlým ztrátám. Podle výzkumu Standish Group nebylo v posledních letech v USA dokončeno 31 % softwarových projektů. Odhaduje se, že v r. 1995 utratila americká vláda 81 miliard dolarů na nedokončené projekty. Více než 52 % úspěšně dokončených projektů IS překročilo v USA náklady téměř trojnásobně. Pouze 20 % projektů skončilo v termínu a nepřekročilo plánované náklady (viz Šilha, 1995). Investice do informačních technologií překročily v USA v r. 1995 250 miliard dolarů. Podle uvedené studie byly příčiny zastavení projektů následující:

- nekompletnost nebo nejasnost požadavků na systém (22 %),
- nedostatek zájmu a podpory ze strany uživatelů (12 %),
- nedostatek zdrojů, tj. podhodnocený rozpočet a krátké termíny (11 %),
- nerealistická očekávání (10 %),
- nedostatečná podpora ze strany managementu dodavatele nebo odběratele (9 %).

Jako důležité faktory úspěchu byly uváděny

- zainteresovanost uživatelů (18 %),
- podpora managementu uživatele (16 %),
- jasně definované požadavky (15 %),
- dobré plánování (11 %),
- realistická očekávání (9 %),
- správná dekompozice úkolu (9 %),
- kompetentnost zúčastněných (8 %).

Analýza byla provedena na základě odpovědí 365 firem a pro více než 8000 aplikací. Je to tedy studie dosti reprezentativní. I u nás je dosti běžné, že se IS podaří zavést až po několika neúspěšných pokusech.

Většina faktorů úspěchu i neúspěchu informačních technologií (IT) se dá charakterizovat jako důsledek dodržování či nedodržování zásady analogie. Typické problémy jsou způsobovány nezainteresovaností uživatelů, nezájmem managementu, nerealistickým očekáváním, chybným rozpočtem atd. To vše lze charakterizovat jako nedodržení známých zásad přípravy a řízení technických projektů neboli neuplatnění principu analogie. Novost problematiky není omluvou – to by mělo spíše zvyšovat zájem a pozornost managementu a uživatelů.

Přes rozvoj nových metod a prostředků podpory vývoje softwaru (CASE systémy, vývojová prostředí, objektová orientace atd.) nedochází k podstatnému snížení podílu neúspěšných projektů. Mnozí dokonce tvrdí, že se situace spíše zhoršuje. Porušují-li se elementární zásady vedení projektu, je používání moderních technik stejné jako výměna žárovky, když nejde elektřina. Výše uvedená data o neúspěších vzbuzují podezření, že zákazník nevědomky jedná v souladu s následujícími „zásadami“: „Zaveďte u nás informační systém, je to moderní. Jsme vám za to ochotni dobře zaplatit, třeba i desítky milionů jsme ochotni pustit. Nechtějte ale proboha od nás, abychom si udělali jasno, k čemu to bude dobré, a abychom s vámi spolupracovali. Máme své práce nad hlavu“. Dochází tedy k nedodržování samozřejmých zásad. Se samozřejmou zásadou souhlasíme a řídíme se jí, pokud ji zformulujeme. Software je složitý i proto, že je při něm nutné dodržovat velmi mnoho jednoduchých zásad a snadno se něco opomene. Opomenutí samozřejmých zásad mívá velmi závažné důsledky.

U klasických technologií nikoho nepřekvapí, že je nutný rozbor přínosů, pečlivá příprava instalace a příprava obsluhy. U IS musí být míra spolupráce a nasazení managementu vyšší. IS ovlivňují celý chod podniku a mezi uživatele by měl patřit i management. Cena IS bývá značná. Spoluúčast při vývoji a přípravě provozu by proto měla být ze strany managementu uživatele i koncových uživatelů větší než u klasických technologií. Výše uvedená fakta však svědčí o opaku.

Je známo a shoduje se to i s výsledky výše uvedené studie, že případů, kdy projekt selže kvůli nezvládnutí technických problémů nebo technické nekompetentnosti řešitelů, je velmi málo. Proto budeme věnovat velkou pozornost především technikám a podmínkám pro úspěšné řešení otázky *proč* (cíle) a specifikaci požadavků (*co*) a tedy i „samozřejmostem“.

Problém je však hlubší. V rozsáhlé knize prof. Landauera *The Trouble with Computers*, 1995, jsou shrnuty výsledky statistických studií o efektivnosti investic do informačních technologií. Ukazuje se, že meziroční přírůstky produktivity práce v rozvinutých zemích po r. 1973 poklesly. Od r. 1973 se masově zavádějí IT. Analýza dat ukazuje, že pokles pravděpodobně není způsoben ropnou krizí.[2]

Jediná dvě odvětví, kde produktivita práce vzrůstala po roce 1973 rychleji než dříve, jsou materiální výroba a telekomunikace. Nejsou to však například banky. Překvapivý je pokles produktivity v ediční činnosti. To je zřejmě způsobeno tím, že se vydává více titulů v menších nákladech. U bank není zjištěn žádný pozitivní efekt velikosti podílu IT na investicích na hospodářské výsledky; pokud mají nějaký vliv, je spíše záporný. Podobná situace je i v jiných odvětvích. Interpretace tohoto faktu je obtížná a vyžaduje hlubší analýzu. Je např. možné, že se k IT uchylují firmy, které mají nějaké potíže, jako k pověstnému stéblu tonoucího. IT jim může opravdu pomoci, doklady pro to však zatím nejsou k dispozici.

Mohou být i jiné příčiny. Z historie je známo, že zvládání nových technologií trvá desetiletí. Třífázový motor byl vynalezen kolem roku 1880, přínosy jeho využívání se však projevily až po roce 1920. Poučný je případ bankomatů. Důvodem zavádění platebních karet je přání zákazníků. Pro banky jde o operaci, jejíž návratnost je sporná. Není dokonce vždy jisté, že karty šetří čas zákazníků. To se samozřejmě může změnit.

Použití Internetu je dalším příkladem rozporných přínosů. Velké množství informací je příjemné, ale ani při použití moderních prostředků není snadné najít právě to, co potřebuji. Mnozí z nás již pocítili tyranii elektronické pošty, která nám každý den ukradne hezkou řádku minut. Přínosy IS nejsou tedy jednoznačné a jsou často jinde, než se očekává.

2. Dá se ovšem namítnout, že se do statistik výkonů nezahrnují některé služby a výrobky, které dříve vůbec neexistovaly. Prosperita USA v posledních letech je pravděpodobně způsobena výnosy z informatických technologií, přílivem infodolarů.

Často diskutované téma je *informatická společnost* jako společnost založená na využívání moderních informačních technologií. Budoucnost se často líčí v růžových barvách. Informatická společnost skutečně znamená novou kvalitu. Informatická revoluce je v knize manželů Tofflerových (Toffler et al., 1994), přirovnávána k revoluci agrární a revoluci industriální. Jako taková mění strukturu společnosti a ohrožuje existující mocenské struktury. Nová situace vyžaduje nové myšlení. To vše vede ke krizím a k pokusům neměnit zvyklosti. I to je jedna z příčin problémů s informačními systémy. Ve stejné knize se na příkladu války v Perském zálivu ukazuje, že společnost využívající informační technologie má civilizační převahu nad industriální společností. Nositelem změn jsou především IS.

1.2 Inženýrský přístup k vývoji softwaru

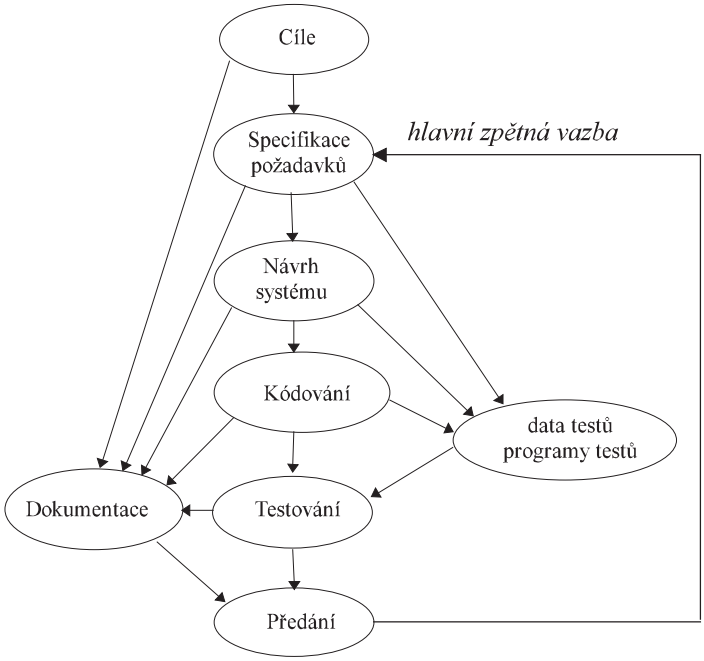
Inženýrský přístup k vývoji technických děl obsahuje následující prvky:

1. Předprojektové činnosti (marketing, vyhodnocování technického vývoje, atd.).
2. Pravidla formulování projekčních záměrů (cílů projektu).
3. Pravidla a metody vývoje výrobků (včetně dokumentace a zásad dekompozice).
4. Znalosti technologie a možností technických prostředků.
5. Pravidla organizace práce, analýzy rizik a oceňování prací.
6. Metody a organizace výroby.
7. Zásady předávání, oživování, provozu a údržby výrobku nebo technologie.

Náplň inženýrské činnosti se sice liší pro jednotlivé technické obory, výše uvedené aspekty v nich však existují vždy a mají dosti společného (princip analogie). Později vzniklé obory investují větší podíl zdrojů do vývoje (srv. stavebnictví a elektroniku) než do samotné výroby (vytváření kopií). Vyvrcholením tohoto vývoje je software, kde dochází do značné míry ke splývání výroby a vývoje. Výroba kopií je prakticky zadarmo. Vývoj softwaru probíhá v následujících etapách.

1. *Marketing a specifikace cílů (formulace problému)*, hledání odpovědí na otázky *proč* a rámcově *co*.
2. *Specifikace požadavků*, formulace přesné odpovědi na otázku *co*, zčásti na otázku *jak*. V této etapě se obvykle stanovují termíny řešení a zpřesňují ceny a stanovují i zdroje (kdo bude řešit a na jakém vybavení). Tato etapa bývá ukončena oponenturou testující, zda požadavky odpovídají cílům projektu, zda jsou konzistentní a splnitelné v daných termínech a s předpokládanými zdroji (studie proveditelnosti – feasibility study).
3. *Návrh systému.* Řešení technických otázek dekompozice, návrhu rozhraní, struktury dat, algoritmů a volba softwarových vývojových prostředků a systémového softwaru.
4. *Kódování (programování) částí*.
5. *Testování*: částí – unit tests, integrační, funkcí, předávací.
6. *Oživení a předání*: instalace hardwaru a základního softwaru, instalace systému, předávací testy, zkušební provoz.
7. *Údržba*: odstraňování chyb zjištěných za provozu, přizpůsobování novému hardwaru (HW) a změnám v použitém základním (systémovém) softwaru (ZSW), jako jsou databázové a operační systémy, a konečně vylepšování funkcí[3].
8. *Stažení z provozu.*

3. V anglické terminologii corrective maintenance, adaptive maintenance, perfective maintenance.



Obr. 1.1: Činnosti při metodě vodopádu.

Body 1. až 7. tvoří *životní cyklus softwaru*. Body 1., 2. a větší část 3. tvoří *analýzu systému*. Pracnost údržby podstatně převyšuje pracnost vývoje (viz odstavec 1.3.).

Pokud se vývoj softwaru uskutečňuje tak, že činnosti v bodech 1. až 5. jsou přísně odděleny a provádějí se striktně ve výše uvedeném pořadí, hovoříme o *metodě vodopádu*: stále padáme a nemůžeme se vracet; výsledek zjistíme, až dopadneme – až po předání. Metoda vodopádu má mnoho nedostatků. To vedlo k řadě variant vývoje softwaru (kap. 7, kap. 18).

Dokumentace, data testů a testovací software jsou nutné během vývoje softwaru a také pro zajištění efektivní údržby, především pro prověrku toho, zda se během údržby nenarušily funkce a nezhoršila spolehlivost systému (regresní testování – regression testing). Prostředky pro testování a testová data jsou vytvářeny na základě specifikací požadavků, návrhu a kódování částí. Dokumentace je výstupem všech ostatních činností. Dokumentace, výkonný software, testovací nástroje a data testů musí být k dispozici při předávání.

Návaznost činností při vývoji softwaru metodou vodopádu je zobrazena na obr. 1.1. Hlavní nevýhodou metody vodopádu je to, že se nedostatky a opomenutí specifikace požadavků projeví až při předávání systému a mnohdy až při provozu, a to je příliš pozdě, poněvadž opravy hotového produktu jsou velmi drahé. Hlavní zpětná vazba, reakce na nedostatky, tedy u metody vodopádu vede od etapy předávání (zjištění chyb) k etapě specifikace požadavků (opravy chyb).

Metoda vodopádu vlastně předpokládá, že jsme schopni neudělat v žádné z etap závažnou chybu. Tento předpoklad pro software obecně a pro IS zvláště neplatí.

Pro vývoj softwaru byla vyvinuta řada technik a metod:

a) Metody zkvalitňující sběr požadavků (interview, dotazníky, pozorování při práci, společný vývoj aplikace, sledování ucelených činností – tzv. procesní pohled) a jejich kvalitu (vnitřní a vnější oponentury), realizace funkcí mimo jakoukoliv pochybnost užitečných.
b) Varianty vývoje (inkrementální, iterativní a spirálový vývoj, využívání softwarových prototypů, viz kap. 7).

1.3 Životní cyklus customizovaných informačních systémů

Vývoj informačního systému je nákladná a dlouhodobá záležitost s dosti vysokým rizikem neúspěchu. Z těchto důvodů mnoho zákazníků volí nákup osvědčených systémů určených pro vícenásobné použití. Vzhledem k různorodosti požadavků zákazníků musí být IS přizpůsobovány požadavkům zákazníka – *customizovány*[4] – tak, aby vyhovovaly konkrétní situaci. Při customizaci je proto nutné stanovit cíle a specifikovat požadavky obdobně jako při vývoji. Vlastní přizpůsobení požadavkům zákazníka se provádí parametrizací – nastavením systémových parametrů jako jsou parametry udávající formáty dat (počet cifer před a za desetinou čárkou) nebo parametry stanovující strukturu obrazovek. Parametry také zařazují, blokují nebo modifikují jednotlivé funkce systému, např. sazbu DPH. Někdy je třeba jednotlivé menší části systému doprogramovat. Modernější systémy používají místo složitého nastavování parametrů specializovaný systém CASE (kap. 19) nízké úrovně (lower CASE), který generuje celý systém vždy znovu. Systémových parametrů bývá velmi mnoho, až tisíce, a jejich volba je velmi obtížná. Použití CASE silně snižuje pracnost customizace a umožňuje i vyšší obecnost. Parametrizace se u moderních IS stále více podobá kompletní generaci nového systému. Proto budeme tuto fázi customizace nazývat generací systému. Po generaci se systém otestuje a instaluje.

Customizace IS probíhá v následujících etapách:

1. Stanovení cílů.
2. Specifikace požadavků.
3. Generace: nastavení parametrů nebo generace systému, např. pomocí dedikovaného CASE systému; případně návrh, kódování a testování úprav a doplňků[5].
4. Instalace a zkušební provoz.
5. Podpora za provozu – údržba.

Dodavatel IS obvykle poskytuje podrobnou metodologii customizace. Nastavování parametrů však přesto bývá neobyčejně komplikované. Systémy, které spíše než nastavování parametrů systém generují, se customizují snáze.

Označme pracnost (spotřebu práce) vývoje IS číslem 100. Pak bude pracnost jednotlivých etap customizace obdobného IS odpovídat zhruba údajům v tabulce 1.1 (viz též kap. 15). V tabulce 1.1 je vyjádřen fakt, že dodavatel IS dává k dispozici metodologii specifikace požadavků a že za této situace je specifikace požadavků poněkud méně pracná. K největšímu snížení nákladů při customizaci dochází u údržby. Ta se totiž provádí pro všechny uživatele společně a na náklady údržby přispívají všichni uživatelé. Proto jsou náklady pro jednotlivého uživatele nízké. Náklady na údržbu u výrobce jsou ovšem velmi vysoké.

Poněvadž jsou etapy stanovování cílů a specifikace požadavků časově náročné, je doba customizace obvykle o polovinu, často však jen o třetinu, kratší než doba vývoje od počátku. Customizace systému R/3 firmy SAP trvá

4. Čteme kastomizovány.
5. Tato etapa se též poněkud zúženě nazývá parametrizací. Pokud se neprogramují nové části a neprovádí úpravy, lze hovořit o konfigurování systému.

	Vývoj	Customizace
1. Stanovení cílů	5	5
2. Specifikace požadavků	15–25	10–15
3. Návrh/generace	15–20	15–25
4. Kódování	15–20	cca 5
5. Testování/předvádění	35–40	cca 10
Celkem	100	35–65
6. Údržba	200	cca 30
Celkem včetně údržby	300	65–110

Tab. 1.1: Porovnání pracností jednotlivých etap při vývoji a při customizaci obdobného systému. Pracnost vývoje IS je normována hodnotou 100.

obvykle více než rok, někdy i více roků, své v tom hraje i schopnost zákazníka systém zvládnout a také zaplatit. Cena customizovatelných IS bývá vysoká. Hlavní výhodou je vyšší záruka, že systém bude pracovat správně. Hlavní nevýhodou customizovaných IS pro zákazníka je to, že IS nemusí přesně odpovídat jeho potřebám (je to přešívaná konfekce).

Použití generovatelných IS má závažné důsledky pro profesní skladbu firem provádějících customizaci. Drasticky roste potřeba obchodníků a analytiků a relativně klesá potřeba klasických informatických profesí, především programátorů. U zákazníka nemá použití customizace významnější vliv na pracnost. Jinými slovy rozsah prací na straně zákazníka při customizaci IS bývá obdobný (a nezřídka i vyšší) jako v případě, kdy dodavatel IS provádí vývoj od začátku. Specifikace požadavků se provádí v obou případech obdobnými technikami a se stejnými úkoly. Při customizaci ovšem bývají k dispozici propracované techniky specifikace požadavků. Customizace nesnižuje náročnost konverze dat a doplňování dat.

Customizace může být pro zákazníka dokonce pracnější, neboť často vyžaduje větší organizační změny, než by bylo nutné při vývoji IS od počátku. Generovaný IS může mít rovněž vyšší požadavky na obsah a formát dat. Generovaný systém obvykle nabízí velké množství funkcí. Zákazník pak mívá sklon používat i funkce, které nutně nepotřebuje, které se však musí naučit používat. To vše může vyvolat další náklady.

Pro dealera i distributora je důležité, jak velkou samostatnost má při customizaci. Některé firmy (SAP) nedovolují prakticky žádné úpravy systému pracovníky dealera či distributora. Pokud jsou změny nutné, provede je firma SAP. Není to ale pro zákazníka levná záležitost. Jiné firmy jsou v tomto směru podstatně liberálnější.

2

Formulace cílů projektu a základních požadavků

Nesprávná formulace cílů je jednou z hlavních příčin neúspěchu software. Správná specifikace cílů je tedy úkol velmi obtížný. Vzhledem k tomu, že specifikace cílů obvykle podstatným způsobem ovlivňuje věcný obsah hospodářské smlouvy mezi dodavatelem a odběratelem IS, musí být při stanovování cílů (otázka *proč*) rámcově známa odpověď na otázky *jak*, *do kdy*, *za kolik*. Bylo by sice vhodnější odpovědi na tyto otázky postupně zpřesňovat ve smlouvách uzavíraných na základě rámcové smlouvy, zákazníci však na takový způsob úpravy smluvních vztahů neradi přistupují. Nelze ovšem říci, že dobře činí. Problém formulace cílů je všeobecně podceňován, mj. proto, že prakticky vždy musí mít formu dokumentu v přirozené řeči a musí být srozumitelný. Řada požadavků proto musí mít spíše intuitivní formu. Volba cílů bývá spojena se specifikací základních (kritických) požadavků, bez jejichž splnění není systém považován za vyhovující.

2.1 Volba cílů softwarových systémů

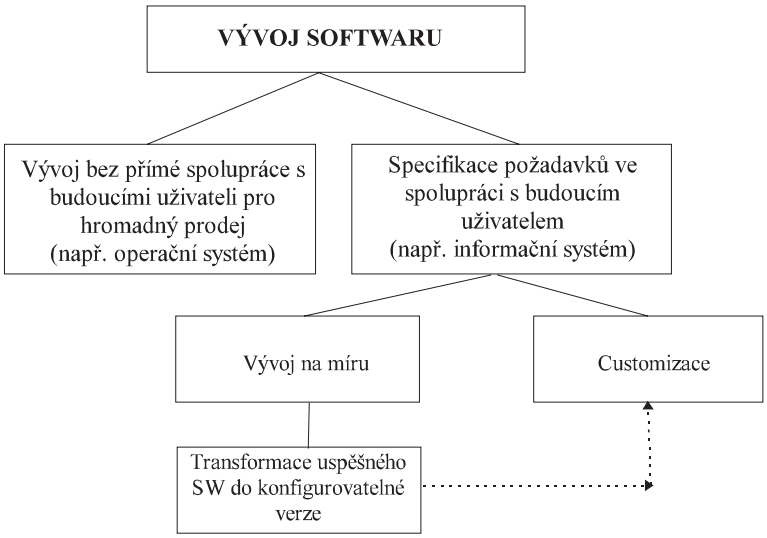
Míra spolupráce s uživatelem potřebná pro stanovení cílů a specifikaci požadavků se pro různé typy software liší. Nastávají následující typové situace (obr. 2.1).

a) Software vyvíjený pro hromadný prodej. Příkladem jsou operační systémy nebo nižší vrstvy komunikačních systémů nebo systémy CAD. Při vývoji software tohoto typu odhaduje výrobce potřebu produktu a v podstatě nezávisle na konkrétních uživatelích stanoví cíle produktu a specifikuje funkce. Software pak prodává hromadně bez nebo s malým rozsahem přizpůsobování požadavkům zákazníka (customizace). Vývoj bývá věcí rozsáhlých týmů a těžiště problémů bývá v tom, *jak* systém realizovat. Návrh systému kódování a testování u výrobce (alfa testování) se provádí bez interakce se zákazníky. Vybraní zákazníci pak testují hotový produkt (beta testování).

b) Software vyvíjený pro několik málo aplikací podle situace a požadavků konkrétních uživatelů. Typickým představitelem toho typu software jsou informační systémy vyvíjené na zakázku. V tomto případě je nutné ve spolupráci se zákazníkem stanovit cíle produktu, přínos (*proč*). Spolupráce se zákazníkem bývá nutná i při specifikaci požadavků (*co* se má realizovat). Při spolupráci se zákazníkem se používá řada specifických technik. Specifikace požadavků se v případě IS musí účastnit pracovníci zákazníka z různých úrovní řídící hierarchie včetně managementu. S řešením technických problémů nebývají větší obtíže. Hlavní rizika jsou

spojena s nedostatečnou úrovní spolupráce a s nedodržením pravidel obvyklých v klasických inženýrských oborech (opomenutí samozřejmých zásad).

Customizovatelné systémy (především customizovatelné IS). U systémů, u kterých se provádí customizace, je nutná spolupráce se zákazníkem obdobně jako v případě b).



Obr. 2.1: Varianty volby cílů softwarového produktu.

Při volbě cílů IS je vhodné dodržovat následující zásady (srv. Tapscott, 1995, str. 27–31):

1. Nezavádět IS jako prostředek okamžitého zlepšení nefunkční organizace podniku.

IS by neměl být nasazován do prostředí, které není organizačně a podnikatelsky v pořádku. Informatický folklór tuto zásadu charakterizuje sloganem: „Počítač je zesilovač – zesiluje pořádek i nepořádek“. Při nepořádcích je velmi málo pravděpodobné, že budou správně specifikovány cíle a požadavky. Současná změna organizačních pravidel a zavádění IS neúměrně zatěžuje pracovníky.

Řešení: Nejprve podnik uvést do rozumného stavu (např. pomocí konzultační firmy) a pak zavést IS. Při změnách organizace zohlednit vlastnosti IS, s jehož zavedením se počítá.

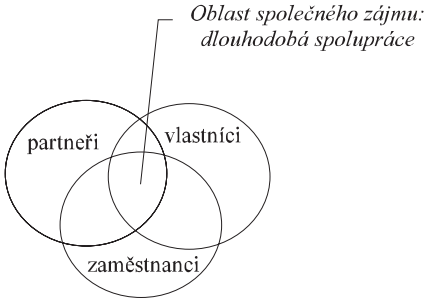
IS samozřejmě vytváří podmínky pro to, aby se podnik po jisté době provozu IS reorganizoval na základě zkušeností s provozem IS a také s využitím dat a služeb, které IS poskytuje. Usnadnění postupné restrukturalizace činností a organizace bývá významným přínosem IS. Pokud podnik vcelku dobře funguje, je možná opatrná restrukturalizace činností a organizace již při zavádění IS. To je součástí metodik některých dodavatelů IS. Restrukturalizace činností podniku/organizace je obsahem business process reingeneerig (BPR, viz níže). BPR je vhodné podporovat prostředky operativního řízení prací (workflow systems). Poněvadž se situace na trhu stále mění a poněvadž je žádoucí provádět BPR i během života systému, je třeba, aby byl IS snadno modifikovatelný.

2. Orientovat se především na strategické cíle.

Dobře fungující IS umožňuje podstatně zlepšit chování podniku na trhu tím, že mj. zrychlí reakce na požadavky odběratelů, změny na trhu a zkvalitní práci managementu, který má prostřednictvím IS k dispozici aktuální data důležitá pro rozhodování. Ze systémového hlediska umožňuje IS zlepšit zpětné vazby. Cíle tohoto druhu nazveme strategické. Taktické cíle se týkají operativy. Typické cíle tohoto druhu jsou snížení zásob, zkrácení výrobních časů, úspora pracovníků atd. Většina úspor na operativní úrovni (s výjimkou úspory pracovníků) přináší jen krátkodobé, často jen jednorázové úspory a neovlivňuje podstatně chování firmy na trhu a její dlouhodobé plány.

Přínosy strategické povahy podstatně převyšují přínosy taktické, poněvadž znamenají změnu kvality. Strategické cíle nezahrnují jen podporu činnosti managementu. Strategie (dlouhodobé plánování a koncipování koncepcí) musí vycházet z pochopení důvodů a podmínek existence podniku. Úkolem strategie je zajistit dlouhodobou prosperitu. To znamená neustále zkvalitňovat parametry podniku, především však zlepšovat chování podniku a kvalitu jeho výrobků a služeb na trhu. Při tom je třeba uvážit následující skutečnosti:

Podnik a obecně každá lidská organizace je průsečíkem zájmů více skupin, jejichž cíle nejsou identické, které však musí spolupracovat. Proto se chovají podobně jako koalice v politice. U podniků jsou to kromě státních a obecních orgánů následující skupiny (obr. 2.2) tvořící koalici v podniku:



Obr. 2.2: Koalice skupin v podniku.

Vlastníci. Primárním cílem vlastníků je dlouhodobě dosahovat co nejvyššího zisku. Současně mají zájem na tom, aby podnik existoval a přinášel zisk.

Zaměstnanci. Prvotním zájmem zaměstnanců je co největší stabilní příjem při minimální pracovní zátěži. Současně ale mají zájem na udržení pracovního místa a tedy nepřímo i na dlouhodobé prosperitě podniku.

Obchodní partneři (dodavatelé, odběratelé). Prvotním zájmem dodavatelů jsou co nejvyšší ceny dodávaného zboží či služeb při co nejměkčích termínech. Naopak odběratelé mají zájem o zboží v co nejlepší kvalitě a nejkratších termínech za co nejnižší ceny. Partneři mají obvykle zájem na dlouhodobé existenci podniku.

Místní mocenské orgány mají zájem na prosperitě a na tom, aby podnik zaměstnával co nejvíce lidí a pokud možno jim dobře platil.

Aby podnik dobře fungoval, je nutné, aby každá skupina omezila své bezprostřední zájmy tak, aby i ostatní skupiny měly z činnosti podniku rozumný prospěch. Jinak není možné zajistit dlouhodobou spolupráci. Z toho důvodu musí skupiny redukovat své požadavky, aby i ostatní členové koalice měli zájem v koalici zůstat. To je typická vlastnost koalicí. Koalice může existovat i na základě mlčky přijatých dohod nebo prostě na základě neformálně existujícího konsenzu.

Majitelé nemohou příliš odčerpávat zdroje podniku (podnik by neměl na investice a časem by nestačil konkurenci) ani příliš zvyšovat ceny (nikdo by drahé výrobky nenakupoval) ani příliš snižovat mzdy (odešli by zaměstnanci). Zaměstnanci musí své požadavky mírnit (mohli by být propuštěni, podnik by mohl zkrachovat). Partneři mají zájem o dlouhodobější spolupráci a proto musí omezovat své požadavky. Je tudíž důležité, aby budovaný IS rozšiřoval oblast společného zájmu zúčastněných a vytvářel podmínky dlouhodobé prosperity. IS by tedy měl kromě zlepšování služeb a kvality výrobků a zvyšování výnosů ve většině případů též zohledňovat legitimní zájmy zaměstnanců, jako je zlepšování kvality pracovního procesu, zvyšování kvalifikace, menší ohrožení zdraví. IS by pokud možno neměl zaměstnance ohrožovat existenčně. IS by měl zlepšováním chodu podniku vytvářet podmínky pro růst příjmů majitelů i zaměstnanců. IS by měl vycházet vstříc i zájmům obchodních partnerů zvyšováním kvality výrobků, zlepšováním platební disciplíny, lepším dodržováním termínů a rychlejší reakcí na požadavky. Zlepšování služeb obchodním partnerům (tj. zkvalitňování služeb podniku či organizace) je jedním z rozhodujících přínosů IS a mělo by být jedním z hlavních cílů IS.

Poněvadž zájmem všech zúčastněných je dlouhodobá úspěšná činnost, je třeba, aby IS zlepšoval dlouhodobou politiku a plánování.

IS by měl zlepšovat parametry podniku jako celku. Při tom je výhodné cíle specifikovat z pohledu ucelených činností – procesů (např. životní cyklus zakázky od obchodního jednání a objednávky až k odeslání zboží a uskutečnění plateb). Cíle by měly být formulovány převážně ve vztahu k externím procesům, tj. procesům, jimiž se podnik projevuje navenek. Procesy v podniku se většinou týkají podniku jako celku (srv. celý proces vyřizování zakázky) a jsou, co se týče obsahu, do značné míry nezávislé na konkrétní organizační struktuře podniku či úřadu. Cíle by měly být formulovány kvantitativně – vzhledem k číselným atributům procesů; např. zkrácení průměrné doby vyřizování objednávky o 20 %. Pokud je nutné stanovovat cíle kvalitativně, např. včasná informovanost managementu o problémech s objednávkou, je třeba navrhnout postup, jak se bude splnění těchto cílů kontrolovat.

Častou chybou je omezení cílů budovaného IS na úspory pracovníků, což je spíše taktický cíl, který by měl být důsledkem celkového zlepšení chodu podniku. Pokud bude IS úspěšný a zlepší hospodaření podniku, je pravděpodobné, že se pro pracovníky uplatnění najde, nebo že sami odejdou v rámci přirozené migrace. Existenčně ohrožení pracovníci mohou najít cesty, jak zkomplikovat instalaci a provoz IS. Strategické přínosy je možné hodnotit podle faktorů dobrého managementu zvaných 7S (viz Koonz, Weinrich, *Management*, Victoria publ., Praha, 1993). Jsou to Struktura podniku a úkoly, Styl managementu a podniková kultura, Systémy a organizace, Strategie, Společné cíle, Spolupracovníci, Schopnosti a know-how. Dobrý IS pozitivně ovlivňuje všechny tyto faktory.

3. Vylučovat nepodstatné nebo zbytečné požadavky.

Při stanovování cílů a nepominutelných (kritických) základních požadavků bývá snaha, aby IS pokrýval co nejvíce funkcí a činností (tuto snahu můžeme charakterizovat jako syndrom dortu pejska a kočičky podle známé pohádky Josefa Čapka O pejskovi a kočičce) bez řádného a nelítostného prověřování, zda je ta která funkce potřeba a zda se její automatizace skutečně vyplatí. Nepotřebné funkce nemohou být z principu správně navrženy (nebyly by nepotřebné) a tedy ani správně realizovány. Nepotřebné funkce ve fungujícím systému vždy nějak překáží (data, nabídky, velikost programů atd.), takže nakonec nepříznivě ovlivňují chod systému. Není tedy vhodné přistoupit na jejich implementaci, i když je zákazník za to ochoten zaplatit. Boj proti nadbytečným požadavkům a cílům je kupodivu značně obtížný. Osvědčují se oponentury, především je však nutná důvěra mezi zákazníkem a dodavatelem IS a řešení problémů ve vzájemné diskuzi. Účinné je použití softwarových prototypů, postupné

budování systému a techniky rychlého vývoje aplikací (rapid application development). Pokud se systém buduje od nejpotřebnějších funkcí, klesá u zákazníka rychle nadšení pro zavádění dalších funkcí s nejasnými přínosy.

Příkladem porušení tohoto principu podle všeho bohužel stále dosud je budování registru obyvatel ČR. Základní funkce (kdo je kdo, rodná čísla, bydliště a případně pojištění) bylo možno realizovat velmi rychle. Ve snaze implementovat všechny představitelné funkce nebylo dlouho k dispozici nic.[1] Snaha o bezbřehost funkcí má u nás dlouholetou tradici (od dob ASŘ) a mnohdy skrývá odpor k efektivnímu řešení. Snaha neoslabit své pozice vyvolává u úřadů ostrý odpor proti propojování datových bází, protože hrozí, že nebudou moci nakupovat IT ve vlastní režii.

4. Brát v úvahu kvalitu dat a složitost úkolu.

IS může poskytovat jen takové služby, pro které jsou k dispozici dostatečně přesná a aktuální data, jejichž pořizování není nadměrně pracné. Nelze dosáhnout vyšší kvality řešení, než umožňují data. Nelze například požadovat algoritmus optimálně rozvrhující práce pracovištím, nejsou-li data norem spolehlivá (kap. 21).

Kvantitativní požadavky při stanovování cílů je třeba formulovat uvážlivě, poněvadž zvláště u úloh kombinatorické povahy může splnění požadavků znamenat realizaci algoritmů neúměrně náročných na výpočtové kapacity. To se může snadno stát u úloh obsahujících rozvrhování prací. Podstatu problému si pro zjednodušení výkladu osvětlíme na úloze obchodního cestujícího. Úloha zní následovně: Je zadána skupina měst a dopravní spoje mezi nimi. Úkolem je projet všechna města za nejkratší čas. Nejlepší známý algoritmus řeší tuto úlohu v čase $k \times c^n$, kde $c > 2$ a n je počet měst. Zvětšením počtu měst o 1 se více než zdvojnásobí doba řešení. Pro velké počty měst jde tedy o prakticky neřešitelnou úlohu. Poznamenejme, že je mnoho důvodů pro hypotézu, že se lepší algoritmus nepodaří asi nikdy najít.

V daném případě však existuje efektivní algoritmus umožňující najít přibližné řešení, které se od optimálního řešení liší jen velmi málo. Poněvadž data o jízdních dobách nejsou přesná, a navíc mohou být ovlivněna dopravními zácpami, nemá smysl používat složitý algoritmus určující přesné řešení. Z nepřesných čísel nikdo nic rozumného nevypočte.

Tyto poznámky nemají význam pouze pro teorii. Řada projektů řízení výrob ztroskotala na přeceňování možností algoritmů rozvrhování a nedoceňování nepřesnosti dat a také na nedocenění toho, že mistr mnohdy rozvrhuje práci lépe než zdánlivě dokonalý algoritmus, který často neuvažuje všechny souvislosti. V praxi je také možné využít toho, že je úloha z nějakých důvodů jednodušší.

Sběr dat by neměl být pracný a hlavně by neměl narušovat pracovní rytmus. Pro úspěch projektů pružného řízení výroby bylo důležité, že veškerá komunikace dělníků se systémem se odvozovala od pohybu palet (kromě chybových situací) a nikoliv od tlačítek, které se mohou stisknout v nevhodný okamžik nebo se na ně může zapomenout.

5. Minimalizovat okamžité organizační změny.

Výše jsme několikrát uvedli, že je riskantní souběžně provádět organizační změny a oživovat informační systém. Základním předpokladem úspěchu je zákazník, který není organizačně nebo podnikatelsky v nepořádku. Takový zákazník bude asi i solventní a je větší pravděpodobnost, že bude mít rozumné požadavky a bude s ním rozumná

1. Na koncepci registru se pracovalo již v osmdesátých létech. Po roce 1990 práce zintenzivněly a prováděly se dokonce po více kolejích. Počátkem roku 1998 nebylo pro občany k dispozici prakticky nic. Přijatá koncepce IS státní správy byla natolik nákladná, že se její realizace nadlouho odložila.

spolupráce. Kvalita zákazníka se dá poměrně dobře odhadnout nejen z ekonomických dat. Dobrými indikátory jsou zájem a vstřícnost managementu, pořádek na pracovišti, pravidelný chod práce, kvalita sociálních zařízení a to, jak se dodržují dohody, jak se dodržuje čas a doba trvání schůzek a zda není obtížné se setkat se všemi potřebnými pracovníky.

Pokud podnik uspokojivě funguje, není vhodné překotně měnit jeho organizaci. Je to riskantní a pro provoz systému nevýhodné. Správně fungující podniky pracují tak, že každý ví, co má dělat za vzniku jisté situace. Nikdo nemusí mít přehled o všech detailech fungování podniku. Na to, jak řešit konkrétní situaci, si pracovníci obvykle vzpomenou, až když taková situace nastane. To je velký problém při specifikaci požadavků. Proto je důležité sledovat ucelené činnosti – procesy. Pracovníci budou schopni dobře pracovat jen s takovým IS, u kterého mají pocit, že rozumí jeho funkci. To lze splnit tehdy, když práce s IS při vzniku určité situace připomíná činnosti, které by se za stejné situace prováděly bez IS.

Pokud je nutné činnosti v podniku reorganizovat a nelze čekat na konec reorganizace, je možné zvolit následující kompromis. Nejprve se realizují ty části IS, které jsou důležité pro ekonomiku. S těmi pracuje menší počet pracovníků, kteří však mají vyšší kvalifikaci. Tyto části bývají navíc méně závislé na konkrétních podmínkách podniku (finance, sklady, účetní subsystém, zpracování objednávek). S využitím dat a služeb těchto subsystémů se provede postupná reorganizace dalších činností a doplní se další subsystémy. Zachovávání struktury činností nemusí nutně znamenat, že se nemění struktura řízení („pavouk“). Ucelené činnosti jsou totiž jen do jisté míry nezávislé na tom, která oddělení jsou odpovědná za určité kroky. Tuto nezávislost IS často ještě zvyšuje.

Pokud je v podniku již úspěšně používán nějaký IS, je možné postupně upravovat organizaci práce a měnit organizační schéma. Tento proces se nazývá restrukturalizace činností (business process reingeneering, BPR). BPR je možné provádět i při záměně starého IS novým. BPR je tedy většinou výhodné provádět postupně. Postupné provádění BPR je usnadněno takovou architekturou IS, která umožňuje snadné záměny částí a je otevřená pro integraci produktů třetích stran.

6. Spolupráce s koncovými uživateli.

Při volbě cílů a specifikaci základních požadavků je nutné kontaktovat všechny skupiny pracovníků, kteří budou systém používat, a také ty, u nichž lze předpokládat, že mají přehled o problémech podniku. Management uživatele proto musí vytvořit podmínky, aby analytici mohli všechny tyto pracovníky kontaktovat a využít jejich znalostí. Problém je v tom, že je nutné kontaktovat pracovníky prakticky ze všech úrovní hierarchie řízení, včetně řadových pracovníků. To nemusí všichni akceptovat – pán má spolupracovat s kmánem.

7. Modifikovatelnost a otevřenost IS.

Instalace IS je nákladná a dlouhodobá investice. Vývoj i customizace IS trvá měsíce i roky. Doba života IS musí být proto poměrně dlouhá, 10–15 let. Za tuto dobu dojde k několika změnám v hardwaru a základním softwaru (síťový software, operační systémy, databázové systémy). Během této doby se objeví nové informační technologie (za posledních deset let to byla počítačová grafika a grafická uživatelská rozhraní, podstatná modernizace databázových technologií, distribuované systémy, multimedia, technologie spojené s Internetem, využívání prostředků virtuální reality atd.) Během doby provozu systému dojde pravděpodobně i k podstatným změnám schopnosti uživatelů pracovat s IS.

Při instalaci IS je často nutné zajistit provoz existujících aplikací (legacy systems) – integrovat je do IS. Je žádoucí umožnit spolupráci, nebo lépe integrovat do IS produkty třetích stran. To je např. případ softwaru

řízení technologií (software ovládající technologii dodává výrobce technologie) či prostředků analýzy dat (datové sklady). Je třeba řešit i problém postupných inovací IS a jeho budoucí vyřazení z provozu. Tyto problémy by měly být zmíněny při specifikaci cílů a podrobněji formulovány při specifikaci základních (kritických) požadavků. Je dále žádoucí vymezit požadavky na následující cílové vlastnosti:

a) *Přenositelnost.* Na jakých hardwarových platformách a s jakými typy základního softwaru bude systém pracovat.
b) Jaké *databázové systémy* lze použít.
c) Rozsah *integrace produktů třetích stran* a existujících aplikací.
d) *Velikost systému.* Je třeba stanovit horní mez množství dat a počtu koncových pracovišť. Tento údaj silně ovlivňuje techniku řešení a volbu základního softwaru, především databáze. Při odhadu velikosti systému je někdy třeba vzít v úvahu, že přístup k IS je symbolem postavení v podniku. Na to musíme pamatovat při plánování umístění koncových pracovišť.
e) *Technické zabezpečení.* Technické otázky (*jak* a *na čem*) je třeba řešit co nejpozději. Předčasné řešení technických otázek odvádí pozornost od řešení problémů, které jindy než v časných fázích vývoje nelze řešit. Otázky, které musí být rozhodnuty poměrně časně, jsou vedle velikosti systému následující:

- využití stávajícího hardwaru a softwaru. To uživatel požaduje často. Za obvyklých okolností je využitelnost malá a úspory nevelké;
- přibližný odhad nákladů na nový hardware a modernizaci základního (podpůrného) softwaru (operační systém, sítě, databázové systémy).

Není to sice správná praxe, ale někdy je nutno se smířit s tím, že si zákazník vybere, od koho se hardware a software nakoupí. Pak je třeba být opatrný ohledně záruk za subdodávky.

2.2 Úskalí při volbě cílů

Použití IS vždy znamená změnu (snažíme se, aby zpočátku byla co nejmenší) vztahů v organizaci. Zde může dojít k opomenutí důležitých souvislostí.

Pro ilustraci uvedeme dnes již klasický příklad jedné z prvních aplikací lineárního programování před desetiletími. Byl řešen problém rozvozu mouky pekárnám. Výpočet ukázal, že je možná úspora 20 % nákladů. Neušetřilo se nic, dokonce došlo k nárůstu nákladů, protože vznikly potíže v plynulosti dodávek a některé pekárny nebyly ochotny měnit dodavatele.

U customizovaných IS mohou být potíže s dostupností nebo formátem dat a se změnami pravidel práce. Někdy není spokojenost prostě proto, že IS ohrožuje zájmy vlivných pracovníků, na příklad tím, že zmenšuje jejich oddělení. Jindy může být zdroj potíží v tom, že by někteří pracovníci raději viděli IS od konkurence. Příčinou obtíží může být i pocit vlivných pracovníků, že se dostávají na vedlejší kolej.

Pro úspěch řady systémů v průmyslu bylo důležité, že se automatizovaný systém choval k okolí obdobně jako systém neautomatizovaný, měl podobné rozhraní (viz kap. 21). U řídicích informačních systémů může být důležité, že se např. IS pro řízení dílny chová ve shodě s pravidly přijatými pro řízení dílen v celém podniku. Tuto otázku budeme diskutovat podrobněji v kapitole 21.

Cíle projektu jsou jedním z důležitých podkladů pro uzavření hospodářské smlouvy a měly by být její součástí. Pracnost a termíny řešení při tom silně závisí na typu úlohy. Nejdůležitější faktory ovlivňující pracnost a termíny jsou:

a) Míra interaktivnosti:
   - zpracování v dávce (nejjednodušší, nejméně pracné),
   - dotazovací systémy (datově orientované aplikace, kde frekvence dotazů vysoce převyšuje frekvenci změn, např. IS na Internetu),
   - systémy reálného času (RT systémy – real-time systems) s měkkou dobou odezvy (měkké – soft – RT systémy, příkladem jsou terminálové systémy; doba odezvy je krátká, není však podstatná chyba, musíme-li občas trochu počkat).
   - systémy reálného času s tvrdou dobou odezvy (tvrdé – hard – RT systémy; odpověď na podnět musí přijít vždy v určeném čase). Příkladem je jednotka intenzivní péče, avionika letadla či zbraňové systémy. Opoždění odpovědi může znamenat selhání (pacient zemře, letadlo nepřistane). Tyto systémy jsou nejpracnější.

   Poměr pracností jedné řádky programu pro dávkové zpracování a jedné řádky programů hard RT systémů je často více než 1:10. Důvodem vysoké pracnosti RT systémů je to, že při výpadku nelze vše pustit od počátku. Pracnost tvrdých RT systémů roste se zkracující se dobou odezvy. Tvrdé RT systémy vyžadují specifické (pracné) metody testování.
b) Počet uživatelů (pro jiné než dávkové systémy):
   - pro jednoho uživatele (nejjednodušší),
   - pro několik až desítky uživatelů,
   - pro stovky až tisíce uživatelů.

   Poměr pracností stejně rozsáhlých systémů 1:4, 1:10.
c) Rozsah dat (texty a číselná data):
   - IS s megabyty dat,
   - IS s gigabyty dat,
   - IS s terabyty dat.

   Terabytové databáze jsou na hranici možností současných informačních technologií.
d) Závažnost důsledků selhání IS.
   - prosté IS. Selhání neznamená přímé ekonomické ani jiné škody. Příkladem je IS o parametrech výroby pro management. Pracnost takových systémů je relativně nízká.
   - ekonomické IS. Selhání znamená přímou ekonomickou škodu. Příkladem jsou finanční a účetní systémy.
   - život ohrožující systémy (mission critical systems). Selhání systému přímo ohrožuje životy. Příklady: Jednotky intenzivní péče, řídící systémy atomových elektráren, řízení letadel, zbraňové systémy.

   Poměr pracností řádků stejně rozsáhlých programů činí i více než 1:20.
e) Rozsah ochrany systému.
   - nízká úroveň. Příklad: jednouživatelský systém a počítači nepřipojeném na síť.
   - běžná ochrana na lokální síti.
   - vysoká ochrana. Systém je dostupný z veřejné sítě, obsahuje vysoce utajované skutečnosti, lze provádět finanční operace.

Při specifikaci cílů a kritických požadavků je třeba s využitím hodnocení výše uvedených faktorů sledovat, zda systém nebude podstatně, tj. alespoň pětkrát pracnější než dosud vyvíjené nebo customizované systémy. Jako příklad uveďme zkušenost týmu, který softwarově zajišťoval let Američanů na Měsíc. Tým selhal při realizaci projektu SKYLAB, který byl datově podstatně náročnější.

K podstatnému nárůstu pracnosti dochází např. tehdy, je-li standardní IS obohacen o prvky přímého řízení procesů – o prvky řízení v reálném čase. K podstatnému nárůstu pracnosti dojde prakticky vždy, posune-li se

hodnocení libovolného z výše uvedených faktorů a jeden bod (řádku). Nebezpečí tohoto jevu je v tom, že se změna kvality faktoru jeví jako zdánlivě nevýznamná (např. počty pracovních míst, zabezpečení dat při výpadcích atd.). Vyřešení ochrany dat ve veřejných sítích (např. využití kryptografie) je nutnou podmínkou využití potenciálu IT k modernizaci pracovních a společenských procesů, jako je obchodování přes Internet a práce doma atd.[2]

Lze také postupovat tak, že některé části IS realizujeme jako aplikace nižší třídy náročnosti. Na jisté univerzitě bylo třeba vybudovat IS. Hlavním problémem byla správa prostředků na vědecké granty, kde byla nutná úzká spolupráce mezi vědeckými pracovníky a účetními. Bylo rozhodnuto, že plně interaktivní musí být pouze účetní subsystém (US) a ten že je vhodné koupit. Data z US byla exportována na WWW server WWWS, který pracovníci používali jako databázi informací o stavu prostředků grantů. WWWS byl dále použit jako prostředek sběru dat o nákupech z prostředků grantů. Sebraná data se importovala do US pod kontrolou účetního. Kontrola byla nutná z titulu hmotné odpovědnosti. WWWS byl systém bez přímých ekonomických důsledků a vzhledem k typu použití bez nutnosti řešit problémy spojené se souběžnou prací uživatelů a s restartem. To výrazně snížilo pracnost realizace.

2.3 Dvojí tvář informačních systémů

Jak jsme diskutovali výše, jsou IS používány dvojím způsobem:

1. Pro řízení každodenních operací (pro operativu – operativní IS – OIS).
2. Pro podporu rozhodování managementu (analýza dat, manažerské IS – MIS).

Pro OIS jsou typické následující vlastnosti:

a) Data, se kterými se v určitém okamžiku pracuje, nemají příliš velký rozsah a tvoří určitý uzavřený celek (např. skladový list a přímo související informace o zboží na listě, jako je zásoba na skladě). Určitý typ dat se vyskytuje v mnoha exemplářích – instancích: je mnoho skladových listů pro různé výrobky. Práce IS začíná vyhledáním relevantních dat.

b) Výběr a význam operací je možné definovat jednou provždy (viz operace při práci skladu) již během etapy specifikace požadavků. Pro OIS je přirozený objektově orientovaný přístup, při kterém se např. data skladového listu a operace nad nimi chápou jako celek.

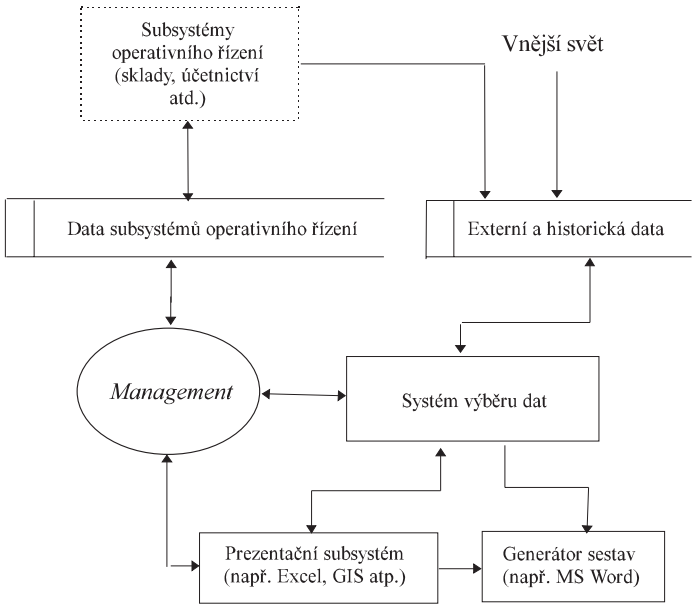
OIS přinášejí výhody spíše taktického rázu[3]. Strategické výhody poskytují ty funkce IS, které zvyšují kvalitu a rychlost rozhodování vyšších úrovní řízení podniku, především vrcholového managementu a také podpora činnosti prodejců a nákupčích. K tomu jsou nutná opatření na úrovni infrastruktury (dostupnost dat), především však výpočet takových údajů, jako je stav zásob, objem a trendy prodeje, průměrná výrobní doba a jiné ukazatele. Je třeba sledovat trendy a dynamiku změn v čase. Pro rozhodování managementu jsou výhodné dotazy ad hoc, jako je objem prodeje v různých regionech případně určitých prodejců atd. Pro takové IS (manažerské IS, MIS) jsou typické následující vlastnosti:

a) Zdrojem dat jsou operativní IS, a případně IS mimo podnik (Internet).

b) V IS se používají informace globálního charakteru vypočtené z velkých objemů dat (stav skladu).

2. Firma Lewis dosáhla pomocí Internetu toho, že je schopna dodat kalhoty na míru do tří dnů. Podobné techniky urychlují oběh zboží a zvyšují možnosti individualizace objednávek aut atd., (srv. Vořišek, 1997).

3. Taktické výhody jsou spíše takové, které se obvykle týkají vnitřního chodu podniku, mají spíše krátkodobé efekty a nemají významný vliv na chování podniku na trhu. Příkladem taktického přínosu je snížení stavu zásob a do značné míry i redukce počtu pracovníků. Strategické výhody jsou spíše dlouhodobé a mají významný vliv na chování podniku na trhu. Příkladem je inovace výrobků nebo zlepšení komunikace s obchodními partnery.



Obr. 2.3: Možná struktura manažerského informačního systému.

Operace jsou často založeny na ad hoc dotazech (manažer nemůže předem přesně stanovit, na co se bude nakonec ptát). IS tohoto typu nazveme manažerské (MIS). Oba druhy IS se poněkud liší v pohledu na data. U OIS se potřeba dat odvíjí od konkrétních přesně definovatelných operací. U MIS (a jejich zdokonalené variantě zvané exekutivní IS) nelze potřebná data předem přesně vymezit. Proto se pořizují všechna data, u kterých lze předpokládat smysluplné využití, samozřejmě za podmínky, že data jsou dostupná a jejich pořizování není příliš pracné. Často ovšem postačují data z OIS.

Požadavkům MIS docela dobře vyhovují databázové systémy umožňující SQL dotazy s exportem výsledků do nějakého systému pro prezentaci dat (v nejjednodušším případě tabulkového kalkulátoru, obr. 2.3). Výsledky dotazů lze exportovat do nějakého dokumentografického systému. V nejjednodušším případě může dobře posloužit MS Word.

V obecnějším případě mohou být využívány i specializované systémy. Osvědčuje se zobrazování marketingových dat (např. rozsah prodeje podle regionů do map prostřednictvím geografických IS nebo využívání specializovaných systémů analýzy časových řad pro analýzu trendů prodeje). Moderní IS umožňují integraci se systémy správy prací (workflow systems, viz např. IS firmy Lawson) a dalšími samostatnými aplikacemi (obr. 2.3).

OIS i MIS lze používat jako samostatné systémy. IS plnící současně funkce MIS i OIS umožňují podstatné zvýšení kvality řízení podniků a organizací. Někdy se označují jako exekutivní IS – EIS.

Knihovní (dokumentografické) informační systémy (elektronické knihovny) jsou navrhovány jako nadstavby knihovnických služeb. Využívají řadu specifických technik (vyhledávání v úplných textech).

2.4 Architektury informačních systémů

Informační systémy jsou nejvýznamnější oblastí aplikací informačních technologií. Informační systémy mohou mít různou architekturu, jsou aplikovány v nejrůznějších oblastech a mohou využívat nejrůznější techniky. Rychlý vývoj a velká konkurence přináší stále nové produkty a nová jména.

Z hlediska architektury se často uvádějí:

a) *Monolitní informační systémy*, které jsou koncipovány jako jeden celek.
b) *Federativní informační systémy.* Tyto IS jsou budovány jako soubor relativně samostatných systémů úzce spolupracujících prostřednictvím nadřazeného společného aparátu.
c) *Kooperující systémy* jsou volnější verzí federalizovaných IS. Kooperující IS jsou obvykle technicky realizovány jako soubor spolupracujících aplikací bez výrazného společného aparátu. Prvky této architektury jsou zmíněny v předchozím paragrafu (viz obr. 2.3).
d) *Distribuované IS* jsou takové IS, které existují na síti a jejich data i procesy jsou rozprostřeny po síti (nemají tedy jediný server). Variantou distribuovaných IS jsou globální IS rozprostřené na celosvětových sítích (Internetu). Na lokálních sítích lze i distribuované IS navrhovat jako logický monolit, tedy podobně jako IS nedistribuované. Distribuovanost je vlastnost do jisté míry nezávislá na tom, zda je IS monolitní, federativní či kooperující. Velké IS jsou často distribuované a kooperující.

2.5 Dokument „Stanovení cílů projektu“ (SCP, „Projektový záměr“).

Cíle projektu by měly být stanoveny ve formě písemného dokumentu. Účelem dokumentu je rámcově stanovit funkce a další vlastnosti projektu. Dokument budou posuzovat pracovníci různých profesí, a proto má být formulován srozumitelně, bez zbytečných podrobností, avšak dostatečně přesně. Odtud vyplývá, že dokument stanovení cílů (SCP) musí být ve většině případů spíš intuitivní než formálně přesný. Dokument je rozpracováván (a zpřesňován) v etapě specifikace požadavků. Nemá přitom docházet ke změně podstatných prvků cílů. SCP má obvykle následující strukturu.

1. Název projektu (případně identifikační kód projektu).
2. Shrnutí cílů (formulace problému, problem statement): Formulace celkového úkolu systému formou srozumitelnou i nečlenům týmu. Tato část má být stručná a vystihnout podstatu.
3. Vymezení uživatelů (kdo, kdy a za jakých okolností bude systém využívat, případně pro koho systém není určen).
4. Seznam nejdůležitějších funkcí spolu se stručnými popisy funkcí. Popis je formulován z hlediska uživatele.
5. Zásady pro dokumentování, použité normy.
6. Vazby na jiné projekty a systémy.
7. Rámcové požadavky na hardware (konfigurace, spolehlivost, ...) a požadavky na efektivnost zpracování.
8. Metody ochrany dat, žádoucí způsoby využívání dodávaného softwaru.
9. Požadavky na spolehlivost systému jako celku (doba mezi chybami, vzpamatování po chybě, ochrana dat, funkce nutné pro detekci chyb).
10. Předpokládané termíny realizace a náklady na realizaci.
11. Způsob předání.
12. Perspektivy realizovaného systému, jeho další rozvoj a zajištění údržby, pravidla šíření.

2 Formulace cílů

Dokument „Stanovení cílů projektu“ je základem specifikace požadavků. Specifikace požadavků má podobné členění jako dokument „Stanovení cílů projektu“, je však podstatně podrobnější a formálnější. Při podrobném rozpisu požadavků a jejich formalizaci se můžeme pod tlakem množství práce nechtěně odklonit od původního intuitivně motivovaného záměru. Je proto důležité, aby byl tento intuitivní záměr zachycen v dokumentu „Stanovení cílů projektu“[4]: Intuitivní popis cílů a funkcí značně usnadňuje údržbu programů po jejich předání do užívání (Guinares, 1985). Shrnutí cílů by nemělo být delší než několik málo stránek. SCP bude obvykle číst management a už z tohoto důvodu by to neměl být dlouhý dokument. Stručnost a výstižnost je však důležitá i pro všechny ostatní řešitele projektu, kteří jsou obvykle různých profesí. Cíle by měly být formulovány ve formě měřitelných nebo alespoň dostatečně konkrétních přínosů IS.

Dokument „Stanovení cílů projektu“ bývá výsledkem dlouhé intenzivní vysoce kvalifikované práce. Kvalita tohoto dokumentu je jedním z rozhodujících faktorů úspěchu. Je žádoucí, aby SCP bylo chápáno jako součást smlouvy. SCP je v různé formě součástí dokumentů vyžadovaných softwarovými firmami (viz např. kap. 20, kde se obdoba SCP nazývá „Marketing study“).

4. Klíčová část SCP je „Shrnutí cílů projektu“. Tato pokud možno stručná část vymezuje hlavní cíle projektu. V anglické literatuře se obvykle nazývá „Problem statement“.

3

Krátce z historie

Současný stav informačních technologií a IS má kořeny v minulosti. Pro odhad budoucího vývoje a pro porozumění současné situaci může být ohlédnutí zpět velmi užitečné. Principy počítače v současném smyslu byly zformulovány během druhé světové války a krátce po ní. Vznik počítače je spojen s vojenskými úkoly (dešifrování kódovaných depeší, výpočty atomové pumy). U vzniku počítačů stáli takoví velikáni matematiky, jako byli John von Neumann a Alan Turing a také firma IBM v čele s geniálním manažerem Thomasem Watsonem.

Počátkem šedesátých let umožnilo zvýšení spolehlivosti hardwaru (přechod k polovodičové technologii) a vynález vyhovujících periferií (tiskárny, magnetické pásky) rozsáhlé využití počítačů pro zpracování dat ekonomického charakteru dávkovým způsobem. V té době v podstatě existovaly pouze dávkové informační systémy (tj. pracující v dávce). Zlevnění hardwaru a vyřešení problému interaktivního přístupu k počítači umožnilo rozsáhlé nové aplikace, především v oblasti přímého řízení technologií a výrob a v informačních systémech, a přechod od dávkového k interaktivnímu způsobu práce s počítači.

Interaktivní práce s počítači usnadnila psaní programů, neboť práce s terminálem bývá efektivnější. Interaktivní úlohy jsou realizovatelné obtížněji než úlohy dávkové. Tento trend byl dále zesílen po objevu osobních počítačů s možností grafického rozhraní a jejich využitím jako inteligentních terminálů – klientů – v architektuře klient-server.

3.1 Vývoj programovacích jazyků

Za prvních deset let existence, kdy byly počítače používány téměř výhradně k vědeckotechnickým výpočtům, se přešlo od binárního absolutního programování k prvým jazykům se symbolickými adresami (assemblerům) a k jednoduchým jazykům vyšší úrovně. To umožnilo snížit mnohonásobně pracnost psaní programů. Do té doby bylo hlavním problémem psaní programů (kódování), zatímco problém formulace požadavků a dalších etap realizace nebyl tak tíživý (řešily se většinou z dnešního hlediska relativně jednoduché a obvykle dávno zformulované problémy).

Objevení programovacích jazyků vyšší úrovně znovu radikálně snížilo pracnost psaní programů. V průběhu asi šesti let (1955–1961), kdy byly definovány jazyky FORTRAN, COBOL, Algol, LISP aj., došlo ke snížení pracnosti kódování (psaní programů) v poměru $1 : 5$ až $1 : 20$. Od r. 1961 je vývoj v kódování a v celé oblasti realizace softwaru pozvolný. Tvrdí se, že v r. 1984 se s využitím všech moderních metod a vyšších nároků na parametry

počítačů programovalo jen asi dvakrát až třikrát „rychleji“ než v r. 1960. Změnu může přinést pokrok v metodách skládání softwarových komponent (spolupráce aplikací, kap. 11, kap. 7) a znovupoužívání objektů, které slibují být podstatnou revolucí v IT.

Po roce 1961 tedy přestalo být psaní programů hlavním problémem. Projevem tohoto faktu byl relativně malý úspěch nových programovacích jazyků. V osmdesátých létech bylo 85 % programů psáno v jazycích FORTRAN a COBOL. Fakta o realizaci řady projektů v šedesátých letech (např. v Bell Telephone Laboratories) ukazují, že podíl psaní programů kódování na celkové pracnosti realizace projektu se pohyboval okolo 15–20 %, což odpovídá situaci v 80-tých letech (Beck, Perkins, 1983). I to svědčí o tom, že pokrok v programování není překotný.

Zhruba od roku 1988 došlo k podstatné změně v tom smyslu, že místo jazyka COBOL jsou stále více používány databázové systémy a 4GL jazyky a vývojová prostředí s nimi spojená. Zvýšení produktivity (a spolehlivosti) však nebylo nijak dramatické. Úspěch operačního systému UNIX přinesl i úspěch jazyka C. Jistou popularitu a význam získávají jazyky založené na nových technologiích: jazyk PROLOG (deklarativní resp. logické programování), především Smalltalk a C++ a nejnověji Java (objektově orientované technologie).

Na prahu další revoluce stojíme právě teď. Využití sítí ve stylu sítě Internet, jazyk Java aj. znamená zcela základní změnu ve strategii využití informačních technologií, kdy budou IS montovány z jednotlivých komponent, které budou k dispozici na síti. Rozsah změny zatím jen tušíme (srv. kap. 13).

3.2 Jak se informační technologie používají

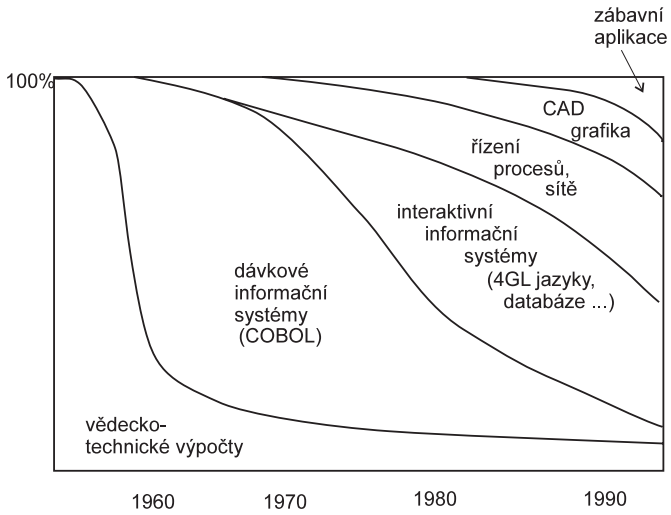
Oblast použití počítačů (obecněji informačních technologií) se v průběhu doby mění či spíše rozšiřuje. Výpočetní kapacita počítače určité ceny se zdesateronásobuje v průběhu několika málo let (odhad je 2 až 4 roky). To umožňuje využití nových informačních technologií vyžadujících velký výkon hardwaru, např. vizuální metody programování. Vývoj zpravidla probíhá tak, že nové oblasti aplikací pohlcují podstatně více výpočetní kapacity než dosavadní aplikace, které jsou samy o sobě stále dokonalejší a stále náročnější na výpočetní kapacitu. Z tohoto hlediska můžeme jednotlivé etapy vývoje charakterizovat podle rozhodujícího druhu aplikací následovně (obr. 3.1):

a) *Etapa vědeckých a technických výpočtů* (asi do r. 1960). Programovalo se v assembleru, později v jazyku FORTRAN (dnes FORTRAN77 a FORTRAN95). Hardware: elektronky, později tranzistory.
b) *Etapa dávkových ekonomických výpočtů.* (1960 – asi 1970–75) Ekonomické výpočty se díky své masovosti staly hlavním typem použití počítačů. Umožnila to vyšší spolehlivosti počítačů. Hardware: tranzistory, integrované obvody; převažující typ počítače: střediskový počítač. Dávkové zpracování. Hlavní programovací jazyky COBOL a FORTRAN.
c) *Etapa ekonomických výpočtů s možností interakce, terminálové systémy* (1970–1980). Přibývá systémů řízení technologií na bázi minipočítačů. Hardware: Vysoká integrace. Vedoucí typ počítače: Střediskový počítač s terminály, minipočítače.
d) *Etapa interaktivních informačních systémů nad lokálními sítěmi*. Aplikace pro osobní počítače (1980–1990). Hardware: velmi vysoká integrace, úspěchy v telekomunikacích. Vedoucí typ počítače: zprvu střediskové, postupně osobní počítače a propojování počítačů do lokálních sítí. 4GL jazyky.
e) *Etapa rozlehlých informačních systémů* (1990–). Postupně se prosazují aplikace pro zábavu, pro domácí počítače a zapojení značné části obyvatel na výkonné datové sítě. V podnicích a organizacích se prosazují informační systémy ve stále větší míře využívající technologii klient – server a postupně i složitější metody distribuované práce. Vše dohromady tvoří základní symptomy vzniku informatické společnosti. Hardware:

velmi vysoká integrace, vysoce výkonná komunikační média (optické kabely), úspěch technologie RISC. Vedoucí typ počítače: Vysoce výkonné osobní počítače a jejich výkonnější varianty – servery. Komunikační zařízení a multimediální prostředí. Stále zůstávají v použití sálové počítače (mainframe) jako centra sítí a silné využití mají i jednočipové počítače (např. v televizních přijímačích a při řízení technologií). Software: distribuované systémy, multimedia, rozlehlé IS, Internet.

Shrneme-li předchozí velmi stručný přehled, dojdeme k závěru, že asi jednou za desetiletí dochází k podstatným změnám v použití počítačů. K revolučním zvratům došlo ale v podstatě dvakrát. Kolem roku 1960 (zavedení programovacích jazyků a masové použití počítačů v ekonomických aplikacích) a v posledních letech (vytvoření datových sítí umožňující distribuovanou práci skupin a připojení značné části obyvatelstva na světové informační zdroje). Revoluce dneška vede ke vzniku nového fenoménu – k informatické společnosti. Význam a důsledky této revoluce je těžko předvídat. Procházka růžovým sadem to ale asi nebude. Již dnes lze ale konstatovat, že informatizace společnosti představuje novou výzvu a nové příležitosti a také nová rizika. Země, které nevsadí na nové technologie, vzdělání a znalosti, se propadnou do bídy a zmaru.

Při vývoji softwaru postupně vzrůstá podíl počátečních etap (tj. identifikace cílů, specifikace požadavků, předběžný návrh, podrobný návrh). Podíl kódování i u nově vyvíjených IS dnes činí významně méně než jednu čtvrtinu nákladů na vývoj softwaru. Tato situace musela vést a také vedla k pokusům o změnu v následujících směrech.



Obr. 3.1: Změny podílů jednotlivých druhů činností na celkové výpočetní kapacitě během historie používání počítačů.

1. Vytváření různých technik projekce a řízení projektu až k vytváření ucelených, počítačem podporovaných systémů zahrnujících kromě kódování i etapy projekce. Typickým projevem tohoto trendu jsou CASE systémy (kapitola 19).
2. Pokusy o „zlepšení“ programovacích jazyků tak, aby se usnadnil přechod od projektu k psaní programů. Prostředky programování se přibližují prostředkům návrhu. Nové vývojové prostředky zlepšují čitelnost

programů, usnadňují týmovou realizaci, zlepšují dokumentaci systému a usnadňují údržbu. Tvorbu softwaru usnadňují prostředky pro ladění a různé analýzy programů včetně prostředků automatické generace kódu uvnitř CASE a generace programů grafickými prostředky (vizuální programování jako Visual Basic, Visual C++, Visual Age, Power Builder atd.).

3. Změny metod používání počítačů (nové oblasti aplikace, interaktivní systémy umožňují snadnou práci s počítači i neprogramátorům atd.) a vytváření programových nástrojů pro podporu všech etap životního cyklu softwaru.

Pro moderní prostředky vývoje programů je typické, že je programovací jazyk a jeho kompilátor integrován do širšího kontextu softwarových nástrojů usnadňujících psaní programů, jejich testování a nejnověji i návrh systému. Během celé historie informačních technologií se vývojové práce postupně stále více a více přenášely na specializované firmy. Ještě v šedesátých letech bylo mnoho různých operačních systémů a dodavatelů kompilátorů. Informační systém nebo jeho zárodečná forma pracující v dávce byl většinou dílem uživatelů. Dnes existuje jen několik málo dodavatelů kompilátorů, počet úspěšných operačních systémů (OS) nepřevyšuje číslo deset. Prakticky všechny úspěšné OS, kompilátory a databázové systémy jsou dílem několika málo softwarových gigantů.

Postupně se měnila náplň IS. Vývoj IS lze rozdělit do tří etap:

1. Zpracování dat (DP – Data Processing) Automatizace informačních procesů dávkovým způsobem.
2. Interaktivní systémy řízení operací (operativy) – operativní IS – OIS.
3. Manažerské IS (MIS). IS zvyšující účinnost práce managementu zlepšením informační podpory a analýzy dat. Klíčovým cílem byla podpora činnosti managementu.
4. Integrované IS podniků s rozsáhlou podporou práce všech stupňů řízení (exekutivní IS – EIS). IS se stává prostředkem, který ovlivňuje aspekty práce všech řídících úrovní a podstatnou část činnosti ostatních pracovníků.

V současné době jsou i IS stále více dodávány specializovanými firmami. Jen výjimečně si IS vyvíjí uživatel pro svou vlastní potřebu. Zmenšuje se stále počet IS, které jsou vyvíjeny speciálně pro danou aplikaci, vyvíjeny na míru. Přibývá těch IS, které jsou vyvíjeny pro širší použití a přizpůsobovány požadavkům zákazníka (customizovány). Prosazuje se realizace IS prostřednictvím jediného dodavatele, který vše „dá dohromady“, provede systémovou integraci a systém oživí. Počet úspěšných výrobců IS se stále zmenšuje a obrat těch, kteří zůstávají, roste.

Tento trend může být zvrácen důsledky nových komunikačních technologií a rozvojem prostředků spolupráce aplikací. Není vyloučeno, že si uživatel svůj IS sám sestaví z komponent dostupných na Internetu.

3.3 Zdokonalování hardwaru a vlastnosti softwaru

Výkonnost hardwaru dovoluje používat nové techniky informačních technologií a vývoje softwaru. Grafické uživatelské rozhraní, GUI – graphical user interface (typickými představiteli jsou MS Windows a X-Window), bylo umožněno výkonnými procesory a grafickými kartami. Použití relačních databázových systémů – základu moderních informačních systémů bylo umožněno nejen teoretickými výsledky, např. metodami indexace, ale též zvětšením kapacity disků a zkrácením doby nastavení hlav disku. Distribuovanost zpracování je založena na teoretických výsledcích fyziky i informatiky, především na vyřešení problémů rychlého přenosu dat po síti.

Historie informatiky ukazuje, že se vývoj hardwaru (HW) i softwaru (SW) nedařilo dobře předpovídat na dobu delší než několik let. Současný stav informatiky naznačuje, že vše směřuje ke stále rozsáhlejším vzájemně

spolupracujícím IS, umožňujícím nové způsoby práce, jako je práce doma. To podstatně ovlivní metody vývoje softwaru a také vlastnosti základního softwaru (ZSW), tj. operačních systémů databází, vývojových nástrojů atd.

Životnost IS je 10–15 let a je tedy podstatně delší než doba, během níž morálně zastará hardware (3–5 let) a dokonce i základní software (5–8 let). Je tedy nejvýše žádoucí, aby IS nebyl během svého života podstatně ovlivňován změnami informačních technologií (HW a ZSW). IS by tedy měl být do značné míry nezávislý na HW a ZSW a jejich změnách. Tomu lze vyhovět např. tím, že IS bude v době uvedení do provozu navržen tak, aby byl schopný práce nad různými ZSW.

IS musí být modifikován během svého života a musí umožňovat integraci s novými aplikacemi. Musí být „otevřený". IS bude nutné po určité době podstatně modernizovat. Nový HW a SW komunikace a stav informačních technologií vedou k tomu, že nově budovaný IS je odlišný od starého. I poměrně drahý IS zaváděný mnohdy v průběhu řady let bude za poměrně krátkou dobu (deset let) vyměněn. Deset let je jen trojnásobek až pětinásobek obvyklé doby customizace.

3.4 Podíl ceny softwaru na ceně IS

Na základě faktů uvedených výše můžeme učinit tyto závěry o pracnosti softwaru:

1. V oblasti vývoje softwaru se ještě neustálily pevné pracovní postupy. U nerutinních (nových) úloh mají klíčový význam schopnosti vedoucího projektu.
2. Výkonnost hardwaru se od r. 1960 zvýšila mnohotisíckrát. I s použitím moderních interaktivních metod vytváření a ladění softwaru za situace, že nemusíme šetřit pamětí a časem, neprogramujeme ani zdaleka desetkrát „rychleji" než v roce 1962. Dnešní IS jsou velmi komplikované a jejich vývoj je proto neobyčejně nákladný.
3. Vlivem prudkého růstu složitosti softwarových (informačních) systémů a poměrně pomalého zefektivnění vývoje SW neustále narůstá cena softwaru. Cena softwaru moderních IS tvoří více než 70 % ceny většiny počítačů a podíl softwaru na ceně počítačů dále vzrůstá. Obrat obchodu se softwarem vzrůstá podstatně rychleji než ekonomika a dokonce i než obrat z hardwaru počítačů. Obrat obchodu se softwarem byl v roce 1995 stovky miliard dolarů, takže se od roku 1980 více než zdesateronásobil.
4. Přes pokroky v nástrojích vývoje softwaru (objektově orientované metody, CASE systémy, vizuální metody programování) se zatím nezdá, že by se v krátké době produktivita práce při vývoji softwaru zdesateronásobila. Zvrat je možné očekávat spíše v tom, že se zvětší šance určitý produkt použít vícekrát.
5. Stále více prací je věnováno údržbě softwaru. Trend růstu podílu údržby softwaru je v současné době oslaben úspěchem IS vyrobených pro mnohonásobné použití.
6. SW je vlastně, až na systémy dodávané v milionech kopií, stále dražší. Kvalita softwaru, především spolehlivost se však s vyšší cenou podstatně nezlepšuje.

4

Problémy a důsledky využívání informačních technologií. Počítačová ergonomie

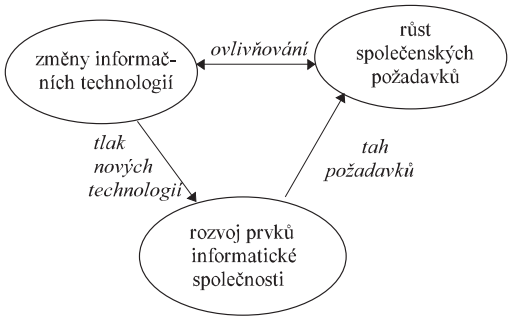
Stále širší využívání informačních technologií má velmi rozsáhlé důsledky pro celou společnost. Informační technologie (IT), především informační systémy, ovlivňují organizaci podniků a kanceláří, ovlivňují obsah a produktivitu práce v řadě činností (konstrukce, marketing, kancelářské činnosti, management, atd.) a globalizují světový trh. Všechny tyto změny zvyšují rychlost změn a zvyšují požadavky na kvalitu rozhodování. Stále více platí, že dostatečnou kvalitu řídicích a vývojových činností nelze zajistit bez uplatnění moderních informačních technologií. Vlivy informačních technologií nejsou vždy jen pozitivní. IT ohrožují pracovní místa, zrychlují mocenské a ekonomické změny, vyžadují vyšší přizpůsobivost a mohou poškodit zdraví těch, kteří je používají.

4.1 Informatická společnost

Rozvoj informačních technologií a zvyšující se požadavky společnosti vytvářejí dvojí tlak na rozvoj a modernizaci informačních systémů (obr. 4.1). Výsledkem je stále větší důraz na modernizaci a rozšiřování funkčnosti IS. Informační systémy používá stále větší procento populace. Tento proces zatím nejeví tendenci ke zpomalování, spíše naopak, jak se můžeme přesvědčit např. na revolučním vlivu celosvětových sítí, především Internetu. Počítače pronikají do nejrůznějších oblastí, mění pracovní náplň jedněch, likvidují pracovní příležitosti druhých a vytváří nové profese.

IT zvyšují potřebu vyšší kvalifikace a také potřebu rekvalifikace, neboť znalosti a dovednosti stále rychleji zastarávají. Ubývá rutinních činností. Roste význam nových informací a znalostí. Společnosti i národy, které to nepochopí a nebudou podle toho jednat, jsou odsouzeny do role outsiderů. Pro kvalifikované užívání IT je třeba, aby nebyly na školách zanedbávány přírodovědné předměty, především matematika a do značné míry i fyzika. Uživatelé IS musí stále investovat do školení personálu a zvyšování jeho kvalifikace. Tím spíše to platí pro dodavatele IS.

Přínosy IT jsou nevídané. Není neobvyklé, že moderní aplikace počítačů přináší zvýšení produktivity o 1 000 i 10 000 %. Počítače umožňují nové výrobní technologie a jako řídící centra robotů a moderních strojů podstatně mění charakter a podstatně zvyšují dynamiku výroby, např. v mnoha případech umožňují zkrátit inovační cyklus na jeden až dva roky. Tak významný jev s tak mnoha důsledky může vyvolat a také vyvolává řadu problémů. Musí být naší snahou negativní vlivy IT minimalizovat a nemůžeme se vymlouvat, že se zdají nevýznamné. Kdysi jsme



Obr. 4.1: Informatická společnost.

se totéž domnívali o automobilech, tepelných elektrárnách a pesticidech. Problémů s nasazením IT by si měli být vědomi především realizátoři softwaru.

4.2 Počítačové nemoci z povolání

Moderní způsob práce s počítačem předpokládá spolupráci s počítačem pomocí obrazovky a klávesnice. Není výjimkou, že pracovníci stráví před obrazovkou podstatnou část pracovní doby. Obrazovka je při tom vzdálena několik desítek centimetrů od očí. Dosud přetrvává představa, že práce u počítačů je spíše rekreace a zábava než tvrdá práce. Je to přesně naopak. Práce s počítači je mentálně namáhavá a může ohrozit zdraví psychické i fyzické.

Barevná obrazovka využívá při zobrazování různých zrakových iluzí. U nekvalitních obrazovek jsou problémy s ostrostí a kmitáním obrazu. Práce u obrazovky vždy namáhá zrak. Nekvalitní obrazovka může zrak i poškozovat. Kvalita zobrazení použité obrazovky a nepřítomnost zbytkového záření by měla být jedním z faktorů při rozhodování o koupi počítače. Zatím není dostatečně ověřen vliv dlouhodobé práce s obrazovkou na zrak zvláště u dětí. I zde je lépe omezit práci s obrazovkou u dětí na hodinu, dvě denně, raději méně, abychom se nedočkali nemilých překvapení. I u dospělých pracovníků je na místě opatrnost, zvláště proto, že již existují špatné zkušenosti s dlouhodobou prací u terminálů. Nadměrné namáhání zraku má i přímé ekonomické důsledky – zvyšuje únavu a snižuje produktivitu práce.

Namáhání zraku a tedy i stres při práci s obrazovkou lze snížit vhodným návrhem komunikace mezi člověkem a počítačem. Různé metody spolupráce mají různé požadavky na dobu, po kterou musí pracovník pozorovat obrazovku. Ani volba barev není bez významu. Je vhodné volit harmonické kombinace barev. Při vstupu větších objemů dat je výhodné vycvičit pracovníky, aby při práci nepozorovali obrazovku – tj. psali podobně jako kvalifikované písařky na psacím stroji. S takovým způsobem práce však musí počítat i návrh softwaru.

Uživatel by měl občas přenést pohled z obrazovky někam jinam, třeba se občas podívat oknem do přírody. Čistě grafické uživatelské rozhraní (GUI) je ergonomicky náročné a může mít i další nevýhody (kap. 14).

Dosti rozšířeným syndromem je ztráta ostrosti vidění, pálení očí večer a celkový pocit psychické nepohody. Vyskytují se i případy dočasné ztráty schopnosti vidění. To vše svědčí o nadměrné a tedy nutně škodlivé zátěži zraku. Dlouhodobé vlivy práce u počítačů na zrak se nepodařilo dokázat. Prokazatelné vlivy na zrak však indikují

silné přetěžování zraku a to nemůže být bez negativních následků. V medicíně se vliv nějakého faktoru prokazuje statistickými metodami. Pokud se vliv nepodaří prokázat, neznamená to, že neexistuje.

Práce s počítačem je mentálně namáhavá, avšak často fascinující. Nejsou výjimkou pracovníci, kteří prosedí u počítače nepřetržitě den i více. Nakonec se vypotácí na čerstvý vzduch se zarudlýma očima a prázdnou hlavou. Často dokáží za den či dva intenzivní práce udělat více než jiní za měsíc. U velkých projektů však bývá taková práce obtížně použitelná (problémy s dokumentací) a je to určitě práce na úkor zdraví. Schopnost některých pracovníků strávit u počítače mnoho hodin někdy připomíná opojení drogou. Podobné případy lze také pozorovat u dětí při hrách s počítači. Je to burcující zjištění. Snižuje totiž sociabilitu, především schopnost komunikovat s lidmi. To je s rostoucím podílem analytických prací, které vyžadují rozsáhlou spolupráci se zákazníky, nevýhodné nejen sociálně, ale i profesně. U dětí, které si zvyknou na počítač, který se dá vždy vypnout a který nabízí brutální hry, může dojít až k antisociálním postojům.

Potíže se zrakem mají převážně charakter diagnosticky neprokazatelných (subjektivních) potíží. Existuje řada nemocí s dosti dlouhou expoziční dobou, tj. dobou působení škodlivého faktoru před vznikem pracovního poškození, které se dají objektivně měřit přístroji.

4.2.1 Objektivně zjistitelné nemoci z práce s počítači

Mezi objektivně zjistitelné počítačové nemoci patří různé otoky a záněty. Zasažena bývá krční a bederní páteř a paže. Postižení paže mají delší expoziční dobu a bývají závažnější.

a) *Poškození rukou a paží z monotónní práce* (opakované zátěže, repetitive strain injuries, RSI). Při práci s nevhodně umístěnou myší a klávesnicí, např. na desce obyčejného kancelářského stolu, především při příliš dlouhodobé nepřerušované práci s počítačem, dochází v důsledku neustálého opakovaného, byť malého napětí svalů k různým patologickým změnám. Nejčastější formy RSI:[1]
   - Změny na loketním kloubu se zánětlivými procesy. Projevuje se úpornými bolestmi v lokti, zvláště při některých pozicích. Poškození dosti připomíná tzv. tenisový loket – nemoc tenistů.
   - Otoky nervových pouzder v ruce a v předloktí. Projevuje se brněním a bolestmi v prstech (palec až prostředník). Může dojít až k úplnému ochrnutí ruky případně i celé paže. Poškození je ve výjimečných případech trvalé.
   - Otoky a záněty pouzder šlach a záněty šlachových úponů. Projevují se bolestmi v paži vedoucími až k znehybnění.
   - Poškození kloubů a šlach v zápěstí.

   Obranou proti RSI je vhodná ergonomie pracoviště (viz níže) a pracovní režim: přestávky po hodině, uvolňovací cviky a velmi krátká přerušení práce po pěti až deseti minutách, max. 6 hodin práce, změny pracovního režimu – nepoužívat pouze myš a klávesnici, pracovat i s papírovými doklady, případně provádět i jiné činosti.

b) *Poškození krční páteře*. Při nevhodné poloze předloh a nevhodné poloze obrazovky dochází k přetěžování krční páteře a šíje. To vyvolává potíže podobné potížím pokladních samoobsluh. Proto se také toto poškození nazývá nemoc pokladních. Kromě otoků a bolestí v oblasti krční páteře dochází k tzv. nespecifickým projevům vyvolaných tím, že otoky v oblasti krční páteře nepříznivě ovlivňují vegetativní nervy, které vedou blízko páteře. To může vést k nepříznivému ovlivňování vnitřních orgánů, nejčastěji zažívacího traktu. Poškození

1. Viz též WWW stránku `http://www.eecs.hardward.edu/rsi/`. Podrobnosti o RSI lze nalézt v knihách (Pascarelli, 1993) a v (Quilter, 1997).

se projevuje bolestmi v zátylku, které mohou případně vystřelovat do paží a ramen, a poruchami činnosti vnitřních orgánů. Otoky a strnutí šíjových svalů vyvolané jednostrannou zátěží stlačují šíjové tepny zásobující mozek. To u citlivějších jedinců vyvolává prudké záchvaty silné nevolnosti až k bezvědomí. Záchvaty přicházejí někdy zcela nečekaně. Obranou je vhodná poloha předloh a obrazovky, kvalitní sedačka a přestávky v práci. Jako prevence záchvatů se někdy osvědčuje masáž šíjových svalů.

c) *Poškození bederní oblasti*. Toto poškození se projevuje bolestmi v kříži a je nejčastěji způsobeno nevhodným způsobem sezení a nekvalitní židlí a dlouhotrvající prací s počítačem. Může způsobit i záněty sedacích nervů (ischias).

4.2.2 Subjektivní potíže

Do této skupiny patří potíže, které lze jen obtížně změřit přístroji. Sem patří i výše uvedené potíže s ostrostí vidění večer a jiné zrakové potíže. Mnoho potíží je důsledkem stresu při práci s počítači. Z toho důvodu není vhodné používat různé „ sledovače výkonu“ a je žádoucí i jinak stres omezovat např. vhodnou organizací pracovišť, vhodným (přátelským) softwarem a také správnou organizací práce.

Výskyt subjektivních potíží, které jsou většinou psychosomatického typu, zahrnuje

- potíže zažívací (sklon k žaludečním neurózám, nadýmání aj.),
- nespavost a noční úzkosti,
- večerní ztráta ostrosti vidění, výjimečně i dočasná ztráta schopnosti vidět,[2]
- poruchy soustředění,
- celková psychická labilita s případným zhoršováním neuróz,
- pocit psychické nepohody a nespokojenosti často ve spojení se snížením výkonnosti.

Výše uvedené potíže mohou být vyvolány nevhodným uspořádáním pracovišť (neodstíněné vysokofrekvenční pole, vyzařování obrazovky, nevhodná místnost, těmto problémům je věnován paragraf 4.3).

Řada výše uvedených subjektivních potíží je typická pro těhotenství. Práce s počítači proto zvyšuje těhotenské obtíže. To může vést až k ohrožení průběhu těhotenství.

4.2.3 Jiné negativní vlivy informačních technologií.

Práce s počítači podstatným způsobem ovlivňuje psychiku. Může dojít až k patologickým jevům. Nejmarkantnějším příkladem jsou tvůrci počítačových virů a chorobná vášeň pro počítačové hry. U informačních systémů se mohou nepříznivé vlivy projevit ve sklonu „svádět vše na počítač“, s čímž souvisí i neodůvodněná víra v samospasitelnost informačních technologií. Používání osobních počítačů může dát příležitost hrát hry v pracovní době nebo užívat další počítačovou drogu – toulání po Internetu.

Informační systémy by měly být ze strany dodavatele i ze strany zákazníka chápány jako prostředek zvyšování kvality lidského mozku a prostředek podpory lidské intuice. Jinými slovy je třeba, aby IS vždy počítaly s lidským faktorem jako svojí integrální součástí a aby byla koncepce volena tak, že IS není pán, ale sluha. Zní to jednoduše, ale obtížně se to realizuje. Při realizaci IS je uplatnění tohoto principu vyjádřeno snahou několikrát se přesvědčit, zda některé činnosti nesvede lépe člověk ručně a snahou nekoncipovat IS jako nástroj, který sám „ze své vůle“ řídí lidi. Je třeba se držet zásady, že pokud není zcela zřejmé, že přenos nějaké činnosti na počítač přinese jednoznačný efekt, danou činnost na počítač nepřenášíme. IS má podporovat činnost lidí, nikoliv naopak.

2. Autor zná mezi svými kolegy dva takové případy. Ztráta vidění byla zřejmě vyvolána i přetěžováním mozkových zrakových center.

Pro úplnost se zmíníme o některých dalších aspektech používání informačních technologií. Počítač by neměl dětem a dospívajícím nahrazovat kamarády. Počítač drží řadu trumfů: nedělá podrazy, je vždy k dispozici, dá se navíc kdykoliv vypnout a nabízí nepřeberné množství her. Výsledkem může být a také bývá, že se dospívající jen obtížně zapojuje do lidské společnosti, neboť je orientován více na počítače než na lidské bytosti. Existují náznaky, že počítače pěstují „černobílé vidění světa“ a tím podporují tendence k extrémním politickým postojům.

Hrozba počítačových nemocí a psychická zátěž při práci u počítačů je značná a může při neúměrném zatížení vést k fyzickému či mentálnímu poškození dětí. To může nastat při výuce pomocí počítačů nebo při nekonečných hrách na tatínkově počítači.

4.3 Zásady hygieny práce u počítačů

Při práci s počítači je třeba se chránit před negativními vlivy. Ochranná opatření lze provádět v následujících oblastech

- pravidla pracovního režimu, organizace práce,
- vybavení pracovního místa (ergonomie pracovního místa),
- vybavení pracovišť,
- spolupráce IS s obsluhou (operátory).

Hlavní zásadou by však mělo být sledování vlastního zdravotního stavu. Je důležité si uvědomit, že i malé bolesti (paže, páteř, oči) mohou vést k závažným poškozením, a včas zjednat nápravu. Zdravotní potíže a stavy, jež jim předcházejí, výrazně snižují výkon pracovníků.

4.3.1 Ergonomie práce s počítači

Podle zkušeností se doporučují následující zásady práce s počítačem (předpokládá se, že pracovník bude pracovat s počítači měsíce až roky):

a) Přerušovat práci s počítačem po každé hodině asi na deset minut. Tomu lze vyhovět, střídáme-li práci s počítačem s jinými činnostmi (např. píšeme na papír).
b) Pracovat u počítače maximálně šest hodin denně čistého času.
c) Je výhodné střídat druhy zátěže (myš, klávesnice, psaní na papír, čtení sestav).

Během přestávek se doporučuje jednoduché cvičení a i během práce je vhodné cvičení jako „narovnat si záda“ a protáhnout se s případným opřením o opěradlo počítačové židle. Únavu očí snižuje relativně časté přenesení zorného pole z obrazovky jinam, stačí se podívat z okna.

Existují pokusy sledovat činnost pracovníka pomocí softwaru a upozornit ho na to, že by měl udělat přestávku. Pracovníci to však považují za spíše nežádoucí dozor a proto nejsou výsledky nejlepší.

Hlavní problém je v tom, že při tlaku termínů se často hodí všechna pravidla za hlavu. Lze očekávat, že záhy budou poškození z práce u počítače hodnoceny jako důsledek nedostatečné ochrany pracovníků. To povede k postihům vedoucích pracovníků a pravděpodobně i k postihům dodavatelů nesprávně koncipovaného IS a ke snížení renomé dodavatele IS a snížení pracovního výkonu a spokojenosti pracovníků.

4.3.2 Ergonomie pracovního místa

Správné vybavení pracovního místa podle ergonomických zásad podstatným způsobem omezuje vznik únavy a nepříjemných psychických stavů a nakonec i vznik počítačových nemocí z povolání. Zásady ergonomie se týkají

počítače, nábytku a celkové koncepce pracoviště. Zanedbávání ergonomických zásad vede nejen k nemocem, ale dávno před tím i k poklesu výkonu.

a) Ergonomie pracovního místa.

U počítače mají zásadní ergonomický vliv monitor a klávesnice. U klávesnice bývá na závadu nepříznivý chod kláves (lehký chod a tvrdý doraz) a rozložení kláves. Vliv a účinnost tzv. ergonomických klávesnic není dosud prověřen. U monitorů jsou důležité grafické vlastnosti, jako je rozlišovací schopnost (počet pixelů na řádku alespoň 1024, počet řádků alespoň 750) a obrazová frekvence (počet obnovení obrazů). Obrazová frekvence by měla být neprokládaně alespoň 70 Hz. Tato čísla by měla být vyšší u obrazovek větších rozměrů. Citlivost lidí k parametrům obrazovky je silně individuální. Vliv zbytkového záření a vysokého statického napětí je u moderních (LR) obrazovek omezen, stejně jako vliv vysokofrekvenčních polí. Pozor však na vyzařování zadní strany monitoru spolupracovníka, pokud není vzdálen alespoň 2 m. Kvalita obrazu závisí na řadě dalších parametrů. Proto se nevyplácí šetřit na monitoru (kvalita, velikost), sedí-li pracovník u něj mnoho hodin denně.

b) Ergonomie nábytku.

Nábytek pracovního místa by měl zajišťovat následující podmínky (viz obr. 4.2):

1. Monitor ve vzdálenosti od očí 50–60 cm, střed obrazovky 15 stupňů pod úrovní očí.
2. Klávesnice a myš v takové výši, aby loket mohl být u těla (nemusel se vyklánět od těla), předloktí mírně zvednuté od vodorovné polohy a ramenní část paže svisle dolů. Úhel mezi ramenní částí a předloktím je zhruba 90 stupňů. Požadavky 1 a 2 vedou k požadavku, aby klávesnice a myš byly na nižší desce, než na které stojí monitor. Tomu vyhovují speciální stolky pod počítače.
3. Stehenní část nohou by měla být vodorovná, lýtková část nohou kolmo. Vzhledem k různé výšce pracovníků to vyžaduje sedačku se stavitelnou výškou sedačky.
4. Sedačka by kromě stavitelné výšky měla mít opěrky rukou a vedle stavitelného opěráku i zařízení umožňující i pružný náklon sedací plochy. Pro nepříliš vysokou pracovní zátěž stačí běžné „počítačové“ sedačky, pro vyšší zátěž je nutno investovat do drahých ergonomicky řešených křesel.
5. Při práci se osvědčují opěrky pod předloktí upínané obvykle na pracovní desku. Opěrky silně snižují zátěž některých svalů a vliv opakované zátěže. Vyvolávají ale někdy pocit, že je pracovník ke stolu připoután.

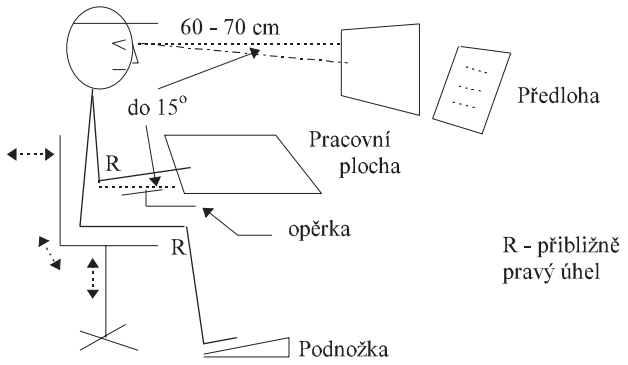
c) Jiné požadavky na pracovní místo.

Podklady pro vstup dat (papíry) by měly být na stojánku vedle monitoru (prevence nemoci pokladních). Pracovní místo by mělo být vybaveno individuálním osvětlením, někdy se požaduje dokonce stavitelný jas lokálního osvětlení, osvětlení místnosti by mělo být nepřímé. Monitor by neměl být umístěn proti oknu (oslňování) ani od okna (odlesky na obrazovce). Na obrazovce by neměly být vidět odlesky okna ani světel či jiných předmětů. Vhodné je okno vlevo, vyhovuje i okno vpravo.

Pracoviště by mělo poskytovat jistou úroveň soukromí a mělo by umožňovat výhled na něco příjemného, např. do přírody. Pracovník by měl být v dostatečné vzdálenosti od monitorů a počítačů kolegů, aby nebyl ovlivňován jejich hlukem a vysokofrekvenčním polem. Osvědčuje se i podložka pod nohy (viz obr. 4.2). Ergonomie pracovního místa může nejen omezit vznik nemocí z povolání, ale má také podstatný vliv na výkonnost a na kvalitu práce. To je méně nápadné a často se to podceňuje, poněvadž nedostatky pracoviště se projevují nepřímo a často se zpožděním. Návrh infrastruktury IS by tedy měl brát v úvahu i ergonomické vlastnosti pracovišť.

Ergonomické požadavky na pracovní místo lze shrnout do následujících bodů:

- kvalitní monitor,
- počítačový stůl (dvě desky, pro monitor a pro klávesnici a myš),



Obr. 4.2: Organizace pracovního místa.

- kvalitní židle,
- podnožka,
- kloubové opěrky pro předloktí,
- okna z boku,
- částečné soukromí,
- individuální osvětlení,
- výhled do přírody.

4.3.3 Ergonomie pracovního prostředí

Tento paragraf se týká pracovního prostředí skupin pracovníků zabývajících se většinu pracovní doby buď vývojem (dodavatel IS) nebo užíváním IS. Pracovní prostředí silně ovlivňuje výkonnost a kvalitu práce (počet chyb) a také spokojenost jednotlivce. Ještě podstatněji však pracovní prostředí ovlivňuje kvalitu a produktivitu týmové práce a osobní vztahy uvnitř týmů neboli vnitrotýmové klima. To má zásadní důležitost pro řadu technik, především pro vnitřní oponentury (kapitola 8).

Pro úspěch IS na straně zákazníka i pro efektivnost vývoje na straně dodavatele IS je vhodné pracovní prostředí významným faktorem a vyplatí se do prostředí investovat. Nejdůležitější požadavky na pracovní prostředí:

1. Jednotlivé místnosti pro nejvýše čtyři, raději však ne více než dva pracovníky s dostatečným soukromím.
2. Výhled do přírody. Okna po straně nejraději vlevo od pracovních míst.
3. Nevhodná jsou okna do jižních směrů (jih, jihovýchod, jihozápad). Důvodem jsou prudké změny světelných a tepelných podmínek.
4. Nepřímé umělé osvětlení celého pracoviště.
5. Místnost pro odpočinek a osvěžení a také pro skupinové aktivity (oponentury, neformální hodnocení práce) a jednání se zákazníkem.
6. Estetické prvky (květiny, obrazy) a hezký nábytek.
7. Každý pracovník by měl mít natolik snadný přístup k společným pracovním prostředkům, jako jsou např. tiskárny, aby pokud možno nerušil své spolupracovníky.

Úplné dodržení výše uvedených zásad může být finančně dosti náročné. Při troše snahy lze kvalitu pracovního prostředí podstatně zlepšit i při částečném splnění výše uvedených zásad.

4.3.4 Ergonomie softwaru

Pracovní zátěž uživatele lze podstatně snížit použitím kvalitního informačního systému. Informační systém by měl být uživatelsky příjemný. Dialog uživatele se systémem by měl být navržen podle pravidel uvedených v kapitolách 11 a 14.

Z hlediska ergonomie jsou významné následující zásady:

a) IS by neměl člověka nutit sledovat pozorně obrazovku po dlouhé hodiny. Je žádoucí, aby software nejen umožňoval, ale přímo vedl k střídání činností (ovládání pomocí myši, pomocí klávesnice, práce s tištěnými podklady), případně vybízel k přestávkám. Někdy to ale není snadné, např. pro CAD.
b) Návrh grafického rozhraní nemá vyžadovat příliš podrobné sledování obrazovky a měl by omezovat počet významem rozdílných prvků na obrazovce na minimum (na obrazovce nemá být více než 8 logicky, tj. typem významu rozdílných prvků).
c) Grafické rozhraní by mělo mít i znakovou variantu (to bohužel není vždy možné). Znakové rozhraní omezuje potřebu sledovat obrazovku a pracovat s myší a je to pro zkušeného uživatele mnohdy výhodnější než práce s myší. Zároveň to podporuje zásadu a).
d) Software má být nezáludný a pro uživatele „srozumitelný“, tj. uživatel má mít intuitivní představu, co se v IS děje. To snižuje počet chyb a stres.
e) Pro zrak a psychiku obsluhy nejsou vhodné prudké změny obrazu na obrazovce. Zrak zatěžují i některé kombinace barev. Tyto problémy je vhodné ověřit na prototypech (modelech obrazovek). Zátěž obsluhy zvyšuje i nejednotnost návrhu obrazovek pro podobné činnosti.
f) Je vhodné, je-li to možné, aby součástí pracovní náplně pracovníka byly i činnosti nevyžadující přímou a stálou interakci s IS. Tomu lze vyhovět např. tak, že IS některé činnosti nepokrývá. To může mít i jiný přínos, protože některé činnosti bývá efektivnější provádět „ručně“.

4.4 Důležitost dodržování ergonomických hledisek

Ergonomické zásady návrhu softwaru a pracovišť jsou často přehlíženy. Důsledky nedodržení ergonomických zásad se projevují nepřímo ve zhoršení výkonnosti, zhoršení vztahů mezi pracovníky, zvýšení počtu chyb atd. Expoziční doba počítačových nemocí je několik let. V případě prokazatelných poškození zdraví z práce u počítače lze v budoucnosti očekávat postih a značné náklady na náhrady za újmu na zdraví.

Vliv ergonomie nemůže být zanedbatelný, vstávají-li i mladší pracovníci od počítače s prázdnou hlavou, zhoršeným viděním a bolavým tělem. Jak jsme viděli, jsou známy i případy dočasné ztráty zraku. Zanedbání ergonomie může být z mnoha hledisek drahou záležitostí. Přetěžování a stres se projeví nejen sníženou výkonností a horší kvalitou práce, ale i vznikem zástupných problémů, větším sklonem ke sporům. To může vést ke ztrátám výkonu dávno před tím, než se nemoc projeví, a také k odchodu draze vyškolených pracovníků.

Ergonomii práce lze často podstatně zlepšit i poměrně malými investicemi do nábytku, monitorů a pracovního prostředí nejvíce zatížených pracovníků. Nevyplatí se šetřit tisíce a ohrožovat tím projekty v ceně mnoha milionů.

Na základě výše uvedených skutečností můžeme učinit závěr, že ergonomická hlediska by měla být zvažována při specifikaci požadavků a při návrhu systému. Zvažovány by měly být především následující aspekty:

1. Rozsah automatizovaných a neautomatizovaných činností.
2. Doba, po kterou budou pracovníci pracovat s počítačem.
3. Kombinování různých činností (myš, klávesnice, doklady na papíru, střídání práce s počítačem s jinými činnostmi).
4. Návrh rozhraní člověk – počítač (grafické/znakové/jiné).
5. Vybavení pracoviště (kvalita monitoru, kvalita vybavení) a pracovního prostředí. To může podstatně ovlivnit výši investic a rozhodování, které činnosti automatizovat.
6. Školení uživatelů o problémech hygieny práce.
7. Organizační opatření při provozu systému (např. pracovní náplň profesí).
   Další informace lze nalézt na `http://www.users.interport.net/~webdeb/`.

5

Marketing a zahájení analýzy projektu informačního systému

V této kapitole budeme blíže studovat problémy obchodní politiky a marketingu dodavatelů IS i jejich zákazníků. Řada zásad je stejná jako v jiných oblastech techniky, plyne tedy z principu analogie.

5.1 Důvody pro zavádění IS

Přínos IS by měl být především v oblasti strategických výhod (viz kap. 2). Využití IS je cesta, jak se vyrovnat s rostoucí složitostí rozhodovacích procesů spojených

- s rostoucím počtem skutečností, které je při rozhodování nutné brát v úvahu;
- se zkracováním doby na rozhodnutí;
- s růstem rizik z opožděného či chybného rozhodnutí;
- s důsledky rostoucí migrace pracovníků. To mj. vyžaduje, aby podnik nebyl závislý na žádném pracovníku a na informacích, které jsou známy jen jemu;
- se zrychlením inovací.
  Z marketingového hlediska jsou důležité následující potenciální možnosti IS.
- Lepší chápání vývoje na trhu a potřeb zákazníků.
- Zrychlení inovací výrobků a služeb. Tím docílit konkurenční výhody na trhu.

Pro zrychlení inovací je možné zkrátit dobu potřebnou k tomu, aby se správně rozhodlo, zda je inovace nutná a jaká by měla být, a zkrátit dobu vývoje a náběhu výroby.

- Zlepšení spolupráce se zákazníky (rychlost vyřizování objednávek, reklamace).
- Lepší informace o chodu podniku (informace o zásobách, lepší využívání kapacit, výrobní časy, prostoje, trendy prodeje atd.).
- Sledování obchodních charakteristik (trendy, rozložení poptávky podle kategorie a druhů zboží atd.).
- Využívání získaných informací pro optimalizaci, např. optimalizace výrobních dávek, optimální řízení zásob atd.
- Menší závislost na jednotlivých pracovnících.

Při klasickém neautomatizovaném způsobu práce existují činnosti vyžadující vysoký organizační talent a zkušenosti. Tyto činnosti nelze často provádět ve skupině. Tím se vytvoří nezdravá závislost na jednom pracovníkovi.

Stalo se, že plánovač výroby musel nastoupit do práce v pracovní neschopnosti, poněvadž výroba se bez něho prostě neobešla. Podobný efekt má každá situace, kdy jsou některé informace známy jen určitému pracovníkovi.

IS umožňuje zachytit i méně zjevné či téměř skryté skutečnosti, např. fakt, že existuje rezerva v prodeji u prodejce s vysokým obratem, poněvadž zásobuje více velkých zákazníků než ostatní. IS je drahé zboží, které poměrně rychle zastarává. Je proto vhodné uvést IS do provozu včas i za cenu, že budou zprvu zprovozněny jen hlavní funkce. Cenu totiž IS nepřímo zvyšují ztráty vzniklé tím, že IS během uvádění do provozu nepracuje a tedy nic nepřináší. Jsou známy případy, kdy odkládání uvedení IS do provozu pro nepodstatné maličkosti způsobilo ztráty z přínosů několikanásobně převyšující cenu IS. Stalo se dokonce, že kvůli odkladům nebyl vcelku vyhovující IS vůbec uveden do provozu. Přineslo to obrovské ztráty všem zúčastněným.

Klíčovou podmínkou dosažení přínosů IS je včasnost a dostupnost informací pro všechny, kteří ho potřebují ke své práci. Ani to nebývá snadné – pro mnohé bývá výhradní vlastnictví určitých informací zárukou mocenské pozice v podniku. Jiným skrytým zdrojem růstu nákladů na IS bývá nutnost příliš velkých organizačních změn, což prodlužuje dobu zavádění IS, snižuje výkon a zvyšuje rizika.

IS ve státní správě by měly sloužit především jako rychlý zdroj informací důležitých pro chod občanské společnosti. Pro občany by měly zajišťovat rychlý přístup k legislativním informacím a informacím důležitým pro hospodářskou činnost, jako je ověřování existence firem a jejich kvality, ověřování občanských průkazů při hospodářské činnosti, celní informace atd. IS mohou umožnit, aby úřady nevyžadovaly vždy znovu stejná data.

Pro vládu představuje IS efektivní nástroj kontroly chodu státního ústrojí. Dobrý celostátní IS může být velmi efektivním nástrojem boje proti kriminalitě, např. v boji proti odcizování aut, a dodržování pravidel občanské společnosti. IS ve státní správě umožňují redukci počtu úředníků a zefektivnění státní správy. Tato možnost však bývá zřídka využívána. Je totiž proti zájmům řady vlivných zájmových skupin.

5.2 Problém volby partnera

Software se stále více vyrábí a dodává pro mnohonásobné použití. Před dvaceti lety si prakticky každý podnik vyvíjel IS vlastními prostředky. Dnes je obvyklé, že IS dodává specializovaná softwarová firma a ta často instaluje koupený IS. Zahájení projektu musí začít navázáním kontaktů mezi partnery. Volba vhodného partnera je prvním předpokladem úspěchu.

Dobrá spolupráce musí být založena na oboustranném prospěchu. Pro dodavatele ani pro odběratele není nakonec výhodné, aby byl realizován nevyhovující systém za neadekvátní cenu. Špatně fungující systém poškozuje zákazníka, neboť mu nepřináší očekávané efekty, i dodavatele, jemuž kazí jméno a jemuž přináší přímé ekonomické ztráty při snaze nevyhovující IS oživit a při soudních sporech. I u IS platí, že není na místě příliš šetřit, např. použitím kritéria „beru nejlevnější nabídku“.

Dodavatel musí mít dostatek prostředků na vývoj nebo customizaci, oživení a zkušební provoz IS. Pokud se IS neoživí nebo oživený IS nepracuje dobře nebo není dostatečně podporován za provozu, je to obvykle škoda obou partnerů, ať je finanční vyrovnání jakékoliv. Při volbě zákazníka i dodavatele je základním kritériem jeho ekonomické zdraví.

U dodavatele je ekonomická úspěšnost nejspolehlivějším, i když ne naprosto spolehlivým indikátorem, že nedodá nekvalitní zboží a že neopustí trh a že tedy bude schopen systém udržovat. U úspěšného zákazníka je větší naděje, že bude schopen zaplatit. Stejně významné je, že u takového zákazníka je větší naděje, že nebude

od IS požadovat nerozumné funkce, jejichž realizace nemůže být v principu úspěšná. Chová-li se někdo racionálně v jedné oblasti, bude se asi chovat racionálně i v oblasti druhé.

Při volbě dodavatele jsou dobrým kritériem reference a především počet úspěšných projektů. Je vhodné se přesvědčit na místě, jaká je funkčnost a spokojenost s IS. Příliš velké množství referencí, stejně jako příliš velký meziroční nárůst základních ekonomických ukazatelů však může znamenat i menší podporu při customizaci a zavádění IS. Stejná úvaha platí pro vztah mezi dealerem IS a výrobcem IS.

Při hodnocení kvality partnera jsou důležité následující skutečnosti:

- Dodržování termínů schůzek, účast pozvaných, dochvilnost, účast na schůzce během celé doby jednání.
- Vstřícnost při jednání – snaha dospět k rozumnému řešení.
- Zajištění účasti všech, kteří jsou potřeba; platí i pro ředitele i pro posledního skladníka.
- Pracovní kultura; nikdo se neloudá, všichni mají co dělat, není nervozita.
- Pracovní prostředí; čistota a vybavenost kanceláří i dílen, kvalita a čistota sociálního zařízení.

Pro dodavatele IS je důležité, zda je zákazník schopen nalézt vhodného pracovníka, styčného důstojníka, odpovědného za spolupráci s dodavatelem IS a zajistit podle potřeby i spolupráci dalších pracovníků. Styčný důstojník by měl být vybaven dostatečnou pravomocí („téměř náměstek ředitele“) a být schopný spolupracovat a účastnit se vedení projektu. Zcela základní podmínkou je i přiměřený zájem managementů obou stran. Zájem managementu je přiměřený, jestliže

- je zajištěna spolupráce všech pracovníků uživatele podle potřeby nezávisle na postavení pracovníka v hierarchii řízení; management se sám aktivně účastní analýzy všech těch částí IS, které mají zlepšit jeho práci;
- management pružně vytváří prostor pro účast všech potřebných pracovníků, včetně sebe sama, na specifikaci požadavků na IS. Jinými slovy nedochází k tomu, že někdo, např. pan ředitel, nemá zrovna čas na předem domluvenou schůzku;
- management zajišťuje dostatek zdrojů potřebných pro řešení, včetně toho, že umožňují podřízeným účastnit se řešení;
- všichni se starají právě o ty záležitosti, které jim přísluší. Náměstek ředitele se tedy nestará o to, jak přesně bude vypadat dialog skladníka se systémem, požaduje pouze zajištění dat pro své úkoly.

Nepřiměřený zájem ze strany managementu může být velmi vážným rizikem. Stalo se, že zástupci managementu rozhodovali i o detailech tvaru obrazovek koncových uživatelů. Projekt se neustále upravoval, až dodavatel IS finančně vykrvácel a musel odstoupit od smlouvy. Ztráty na obou stranách převýšily několikanásobně cenu celého IS. Jen modul řízení skladů mohl přinést při obchodním obratu více než 100 mil. Kč úsporu ve výši více než 10 milionů Kč. A to byl pouze jeden taktický přínos.

Je zřejmé, že skladníkovi na tvaru obrazovky moc nezáleží – při každodenní praxi bude postupovat automaticky a bude mu záležet pouze na tom, aby mačkal co nejméně kláves. Proto pro něho nebude asi vhodné grafické rozhraní. Všem zúčastněným by mělo být jasné, že neúspěch bude nutně neúspěchem společným.

Obě strany uznávají kompetentnost partnera a mají k němu důvěru. Na straně dodavatele může z tohoto hlediska vadit nedostatečná znalost mikroekonomických kategorií a pojmů, především v porozumění principů organizace řízení podniků a řízení výroby (srv. Hospodářské noviny, 8. 1. 1996). Často je problém pouze v tom, že dodavatel IS nedovede propojit odborné ekonomické termíny s jemu známými vlastnostmi dodávaného software.

Oboustraně výhodný je tedy vztah partnerství (srv. sborník konference System Integration, 1997), ve kterém partneři pracují na společném díle, dbají o vzájemnou výhodnost, vycházejí si vstříc a snaží se maximalizovat nejen svůj vlastní prospěch. Vývoj a používání IS je dlouhodobá záležitost a dlouhodobá úspěšná spolupráce není bez dobrých partnerských vztahů možná.

Nespolehlivým zákazníkům je lépe se vyhnout. V životě to však nebývá vždy možné. Rizika můžeme zmenšit následujícími opatřeními:

a) Je vhodné udělat ve firmě trochu pořádek před nasazením IS. Je žádoucí na základě vlastní analýzy nebo analýzy provedené poradenskou firmou provést, je-li to nutné, základní změny v podnikové politice a organizaci ještě před zahájením prací na vývoji IS. Tato opatření by měla být součástí smlouvy o instalaci IS. Slabé místo tohoto doporučení je cena poradenské firmy a také to, že poradenská firma není vlastně přímo odpovědná za úspěch budoucího IS. Problémy firem bývají v podnikové strategii a v rovině vztahů zaměstnanci – majitelé – zákazníci. Chybná řešení na této úrovni může IS jen stěží změnit. Pokud však podnik celkem dobře funguje, není optimální jeho organizaci měnit.
b) Je třeba podrobně analyzovat procesy v podniku a hledat nedostatky, snažit se je odstranit ještě před zahájením prací na IS, případně, není-li jinak možné, během specifikace požadavků.
c) IS by měl podporovat změny v podniku zjištěné a provedené v souhlase s body a) a b).
d) Smlouva by měla být uzavírána jako rámcová, každá etapa prací by měla být kryta další smlouvou, oponována a proplacena. To je schůdné, jsou-li brzy k dispozici fungující subsystémy. Zásady tohoto postupu by měly být součástí hospodářské smlouvy.

Existence informatického oddělení zákazníka je pro dodavatele IS výhodou, avšak jen tehdy, nepovažuje-li informatické oddělení dodávku „cizího“ IS za ohrožení svého postavení nebo ohrožení vlastní prestiže. Výhodou je proto, že pracovníci informatických oddělení mají představu o možnostech IS a je tedy naděje, že budou schopni IS provozovat. Mohou pomoci i při instalaci HW a ZSW. Nebezpečí, že budou IS sabotovat, protože se mohou cítit ohroženi, však není zanedbatelné a je vhodné přijmout opatření, aby k tomu nedošlo.

Pracovník zákazníka odpovědný za spolupráci při realizaci IS (styčný důstojník) by neměl být informatik. Informatik totiž většinou dostatečně nezná různé podnikové problémy a leckdy ho neberou pracovníci zákazníka „za svého“, není např. textilák. Je však žádoucí, aby se informatici zákazníka účastnili prací na vývoji/customizaci od začátku a aby u nich nevznikl pocit ohrožení či odstavení na vedlejší kolej a aby považovali IS zčásti za své dílo. Pocit ohrožení nebo nebezpečí poklesu vlivu by neměl vznikat ani u vlivných pracovníků zákazníka. Pokud zákazník nemá svoje informatiky, je vhodné ho přesvědčit, aby nějaké pro budoucí provoz IS najal, a to již v době specifikace požadavků. Informatici uživatele by se měli účastnit všech etap vývoje/customizace. Někteří výrobci IS, např. Lawson Software a zčásti i Peoplesoft, proto dokonce doporučují, aby customizaci prováděl za pomoci dealera jako konzultanta sám zákazník. Má to kromě většího ztotožnění zákazníka s produktem výhodu usnadnění budoucích modifikací IS.

Existuje ještě jeden ožehavý problém, který se skrývá pod názvy provize, případně úplatek. Přesto, že se dá máloco dokázat, tento problém existuje a zdá se být větší u větších podniků a státní správy. Lidé tam jsou méně závislí na výsledcích práce jejich organizací a také se méně znají. Velikost zakázky dává větší prostor pro nekalé praktiky. Tvrdí se, že dokonce existuje tržní cena úplatku v procentech za přijetí nabídky. Zde je každá rada drahá. Jistou ochranou může být snaha o regulérnost soutěží. Částečnou pomocí může být stanovení cílových odměn pro pracovníky, kteří se na zavedení IS ze strany zákazníka podílejí. Pokud jsou cíle IS stanoveny tak, že je lze měřit přímo, např. zkrácení doby vyřízení objednávky, nebo ve vazbě na zlepšení celkových ekonomických výsledků podniku, lze na dosažení cílů vázat cílové prémie. Řadu efektů IS lze však, především u strategických přínosů, měřit jen obtížně.

Pro jednání managementů větších firem je typická tendence k tzv. řešení pomocí minimaxu. Firma se snaží minimalizovat maximální riziko, které by mohlo nastat. V případě volby partnera je maximální riziko to, že partner nebude schopen splnit podmínky smlouvy, poněvadž zkrachuje nebo změní předmět své činnosti. Toto riziko je

pravděpodobnější u menších dodavatelů s krátkou tradicí. Proto velké firmy volí spíše renomované firmy jako dodavatele IS.

5.3 Převzít, nebo vyvíjet?

Během historie IT se stále méně software vyvíjí a stále více nakupuje. Uživatelé dnes IS obvykle nevyvíjejí. Vývoj IS i customizace prováděné specializovanými firmami a nikoliv přímo uživatelem jsou dnes standardem. Výhody a nevýhody customizovaného IS jsou shrnuty v tabulce 5.1. Tabulka je důležitá pro marketing dodavatele IS.

+	menší nebezpečí, že dodavatel opustí trh, customizovaný IS bývá obvykle podporován větší firmou;
−	neodpovídá přesně potřebám. To obvykle znamená menší účinnost a také vyšší náklady na reorganizace, které by jinak nemusely být nutné;
−	IS má i konkurence, takže neposkytuje podstatnou výhodu před konkurencí;
∗	vyšší nabídka funkcí, které však nemusí být vždy potřebné a pak zbytečně zvyšují nároky na obsluhu systému a také na hardware;
+	obsahuje know-how mnoha instalací, dodavatel většinou poskytuje přesné postupy pro zjišťování požadavků, instalaci, školení koncových uživatelů a oživování systému na místě;
+	ověřeno na více instalacích (reference, lze převzít zkušenosti);
+	úspora nákladů na vývoj a především údržbu;
−	vyšší nebezpečí, že je IS založen na zastaralých technologiích;
−	u cizích systémů nedostatečná lokalizace[1](potíže s českou legislativou a abecedou);
−	obtíže s integrací produktů třetích stran a existujících aplikací (např. SW pro řízení technologií)

Tab. 5.1: Výhody (+) a nevýhody (−), případně výhody i nevýhody (∗) customizovaného IS.

Potíže s legislativou a lokalizací jsou menší u těch customizovaných IS, které se používají ve více zemích, pokud možno nejen např. pouze v německy nebo pouze anglicky mluvících zemích.

Převzetí customizovaného IS (CIS) je tedy pro uživatele, někdy však pouze zdánlivě, spojeno s menším rizikem než vývoj od počátku. Uživatelé IS mají pocit, že tato rizika jsou u „slavných“ zahraničních IS minimální. To je pravda jen zčásti. Hrozba, že CIS přesně neodpovídá potřebám, je velmi vážná. Zastaralost CIS (mnohé CIS jsou založeny na koncepcích a technologiích starých řadu let) může spolu s nevyhovující funkčností způsobit, že doba života IS bude krátká. To znamená další náklady a opakování relativně nebezpečné operace nasazení IS. Problém je tím ostřejší, čím více lidí s IS pracuje a čím nižší vzdělání mají.

Bohatost funkcí CIS svádí movitější zákazníky k tomu, aby nakoupili všechny funkce naráz. To je spojeno s rizikem, že budou jednotliví pracovníci přetěžováni při zvládání systému, zvyšuje to tlak na ne vždy optimální změny v organizaci práce a nakonec to vede i k tomu, že nepotřebné věci budou překážet. Je to něco podobného jako nákup ne zcela spolehlivých nástrojů naráz pro celý big band či symfonický orchestr za situace, kdy hudebníci nedovedou na nové nástroje hrát.

Pro softwarovou firmu je vývoj „na míru“ nevýhodný tím, že po dokončení projektu musí začít zase od začátku. To lze zčásti vyrovnat vyšší cenou dodávky, kterou je ale málokdo schopen či ochoten zaplatit, a vhodnou architekturou systému umožňující znovupoužití podstatné části SW ve více (i nepodobných) systémech. Často

1. Tj. přizpůsobení jazyku, legislativě a místním zvyklostem místa nasazení.

na základě zkušeností z více zakázek „na míru“ vytvoří customizovaný IS umožňující opakované zhodnocení práce. Tato cesta je usnadněna použitím objektově orientovaných technologií. IS vyvíjený „na míru“ může přinést podstatné výhody proti jiným řešením. Předpokládá to však přesnou specifikaci a střídmost v požadavcích. To nebývá snadné.

U IS vyvíjených „na míru“ bývají potíže s údržbou a je větší nebezpečí, že dodavatelská firma nebude poskytovat dostatečnou podporu při provozu a údržbě. Není výjimkou nedodržení termínů. IS „na míru“ však může být kupodivu levnější než slavné CIS. To přitom nehovoříme o výše zmíněných skrytých nákladech. Maximální možné riziko je však u IS vyvíjených na míru větší, proto existuje při uplatňování principu minimaxu silná tendence příklonu k CIS. Výborný kompromis je „šít na míru“ a případně customizovat jen menší část IS a zbytek realizovat tak,že se využijí existující aplikace (např. e-mail, textový editor, tabulový kalkulátor atd.), případně existující CIS. Technologie spolupracujících aplikací diskutovaná níže asi brzy umožní, aby se IS mohl skládat z jednotlivých softwarových komponent, které si stáhne z Internetu.

V některých případech je vlastní vývoj jedinou možnou nebo alespoň jedinou dostatečně příznivou alternativou. Takový příklad jsou IS pro naše zdravotnická zařízení. CIS má výhodu v tom, že ušetříme čas (budeme moci dříve prodávat) a náklady na vlastní vývoj. Při volbě renomovaného výrobce CIS můžeme využít renomé jeho produktu a jeho know-how. Můžeme tak ale ztratit schopnost vlastního vývoje a staneme se silně závislými na výrobci CIS. Jeho případné problémy se přenesou i na naši firmu. V té souvislosti je vhodné poznamenat, že není CIS jako CIS. U některých se předpokládá, že všechny úpravy CIS dělá výrobce (příklad SAP), jiní výrobci umožňují poměrně rozsáhlou spolupráci dealerů a dokonce uživatelů při úpravách funkcí IS. Moderní metody výstavby otevřených systémů, především techniky spolupráce aplikací, velmi usnadňují tuto variantu spolupráce s výrobcem IS integrací moderních počítačem řízených výrobních technologií do IS. Otevřenost IS je důležitá pro výměnu IS za nový systém na konci jeho životnosti. Usnadňuje totiž přechod na nový IS postupnou záměnou jednotlivých komponent.

Při výběru dodavatele CIS je vhodné vyhodnocovat jeho ekonomické zdraví.

Při rozhodování softwarové firmy, zda přebírat či vyvíjet softwarový systém, je nutno zvažovat řadu faktorů. Uveďme ty důležitější:

- Je k dispozici vlastní systém, který by bylo možné použít jako základ CIS?
- Lze za rozumnou cenu získat CIS s funkčností vyhovující potřebám potenciálních zákazníků?
- Jaké jsou vlastní možnosti (počet a profesní struktura pracovníků a jejich schopnosti, finanční rezervy na vývoj atd.)? Jinak se bude rozhodovat firma, která má dostatek kvalitních programátorů, jinak ta, která má mnoho schopných analytiků.
- Lze CIS lehce adaptovat a integrovat s jinými aplikacemi? Je např. závislý na jedné databázi? Jak snadno se převádí do nového jazykového, legislativního nebo kulturního prostředí (lokalizuje)?
- Jaké jsou vlastnosti výrobce, jeho ekonomický růst, v kolika zemích je činný, jak spolupracuje?
- Jaké jsou technologické vlastnosti produktu? Jsou používány moderní technologie, jako je dnes technologie klient-server? Jak je systém otevřený? Jsou použité technologie blízké technologiím dosud ve firmě používaným?
- Pro jakou třídu zákazníků je CIS určen (obor: výroba, státní správa atp.; velikost: malé/střední/velké organizace, např. praktický lékař, sanatorium, nemocnice)?
- Jaká je nezávislost produktu na hardwaru a základním softwaru včetně databází?
- Je použití CIS ve shodě s koncepcí vlastního rozvoje?
- Bude možné zachovat silné stránky firmy, jakou formu spolupráce výrobce předpokládá?

- Jak silný je výrobce a jaké služby poskytuje?
- Jak je propracovaná metodika customizace?
- Pro zákazníka – jaká je kvalita výrobce a dealera?
- Pro dealera – jaká je forma spolupráce s výrobcem, renomé výrobce?
- Jaká je cena a dodací podmínky?

Rozhodnutí přebírat či vyvíjet CIS patří mezi nejriskantnější kroky softwarové firmy. Vývoj bude asi směřovat k takovým CIS, které budou připouštět snadné úpravy pro koncového uživatele integrací jeho existujících vlastních aplikací i cizích systémů a snadné programování funkcí specifických pro uživatele. Výrobců CIS nebude mnoho.

Provoz IS vyžaduje podporu ze strany dodavatele. Na tuto skutečnost je třeba brát zřetel při uzavírání smluv, ale také při plánování rozvoje softwarové firmy. Je obvyklé, že na provoz IS jedné až dvou nemocnic musí být u dodavatele vyčleněn jeden pracovník na plný úvazek. Dostatek pracovníků musí na provoz IS vyčlenit i uživatel. Mezi těmito pracovníky by měli být i informatici uživatele. Pokud zákazník nemá vlastní informatiky, měl by si je pro tento účel najmout.

5.4 Systémová integrace

Při realizaci IS má budoucí uživatel IS následující možnosti:

1. Vlastní vývoj a zajištění subdodávek a integrace systému vlastními silami. Tato varianta se dnes používá zřídka.
2. Omezení vlastního vývoje, nákup IS a provádění systémové integrace vlastními silami.
3. Převzetí IS na klíč. Dodavatel pak ručí za subdodávky a celkovou funkčnost. Hlavním cílem je integrace částí (vlastní vývoj u dodavatele je možný), oživení a provoz systému. Takový dodavatel se nazývá systémový integrátor a jeho hlavní činnost je systémová integrace.

Podporu a pomoc při zavádění IS poskytují

- konzultační firmy (poradenská činnost),
- dodavatelé částí (např. kabelových rozvodů, operačních systémů),
- systémoví integrátoři (fungují jako generální dodavatelé na klíč) ručí za subdodávky, integraci a oživení systému.

U IS vývoj směřuje k využívání služeb specializovaných systémových integrátorů. Systémová integrace se považuje za klíčový problém budování IS větších organizací. Z tohoto důvodu byla založena Česká společnost pro systémovou integraci pořádající každoroční konference s názvem System Integration a vydávající časopis Systémová integrace. Všimněme si doporučení systémových integrátorů jako příkladu, jak se poznatky z předchozích kapitol uplatňují v praxi.

Realizace IS prostřednictvím systémového integrátora (SI) má pro zákazníka řadu výhod. Systémový integrátor:

- striktně ručí za vše (i subdodávky);
- ve spolupráci se zákazníkem specifikuje cíle: hlavní funkce, co se zahrne a co se nezahrne do funkcí IS, pravidla pro rozhraní na jiné systémy;
- dekomponuje IS do subsystémů a stanovuje rozhraní mezi subsystémy;
- spolu se zákazníkem stanovuje a kontroluje harmonogram realizace;
- provádí integrační a předávací testy;
- stanovuje pravidla údržby a formy záruk;

- zaškoluje pracovníky uživatele pro práci s IS a uvádí společně s pracovníky uživatele systém do provozu.

Systémový integrátor by neměla být malá firma, aby byl softwarový integrátor schopen zajistit následující cíle a úkoly:

- ručení za celek,
- optimální volba hardwaru a podpůrného softwaru,
- kvalitní analýza: vymezení požadavků včetně rozhraní na okolí a obsluhu, dekompozice,
- optimální uplatnění datových sítí v IS,
- know-how tvorby týmů a koordinace jejich práce,
- realistické termíny a jejich dodržování,
- přesvědčivé a kvalitní předávací testy,
- kvalitní školení obsluhy a podpora provozu,
- přenos know-how na uživatele: znalosti problematiky IS, zkušenosti s instalací více IS, metody specifikace požadavků.

Pro práci systémového integrátora (SI) platí všechna výše uvedená pravidla pro instalaci IS. Podmínky úspěchu jsou podle zjištění České společnosti pro systémovou integraci zejména (srv. kap. 1 a 20):

- angažovanost managementu zákazníka: zájem, hmotná morální a organizační podpora, (srv. kap. 2);
- angažovanost, zájem a spolupráce všech koncových uživatelů včetně těch na nižších úrovních hierarchie;
- rychle použitelné funkce – brzy vzniká pocit, že se realizuje rozumná věc;
- zajištění týmové práce.

Při specifikaci požadavků se řeší především následující úkoly:

- vymezení klíčových problémů a klíčových (neboli kritických) požadavků,
- analýza předností a nedostatků stávajícího stavu,
- stanovení požadavků a jejich odsouhlasení klíčovými uživateli,
- stanovení podmínek pro uvedení IS do provozu a pro provoz IS,
- přínosy (měřitelné), kritéria úspěchu,
- kvalita realizovaných funkcí (popis),
- zajištění dat: jaká, kolik, kde vznikají, jak se zpracovávají a používají,
- vazby na jiné aplikace,
- kritéria volby HW a podpůrného SW,
- podmínky pro dialog IS s obsluhou: doby odezvy, použití grafického uživatelského rozhraní, možnost znakového rozhraní.

Většina systémových integrátorů (SI) pracuje s importovanými systémy. Ne vždy se však daří importované systémy dobře lokalizovat, někdy jsou importovány zastaralé systémy.

5.4.1 Problémy systémové integrace

Výhody spolupráce se systémovým integrátorem jsou zřejmé, prosazují se však pomalu z následujících příčin:

a) Systémového integrátora se snaží dělat i firmy, které pro to nemají předpoklady. Mnohé z nich nesplňují ani podmínku, aby byly dostatečně velké (alespoň deset pracovníků).
b) Zákazníci neoplývají finančními prostředky a služby kvalitních systémových integrátorů nejsou levné.
c) Chybí know-how a kvalifikace u koncových uživatelů i u informatiků pracujících u zákazníka potřebné pro spolupráci se systémovými integrátory a všeobecně pro aplikaci moderních informačních technologií.

d) Investice do IS, které často rozhodují o budoucnosti podniku, se nesledují se stejnou důsledností jako jiné investice. Jedním z projevů tohoto stavu je všeobecné podceňování úvodních fází řešení (specifikace cílů, výběr partnera), malá pozornost se věnuje personálnímu zajištění přípravy IS na straně zákazníka.
e) Často se, zvláště při vyhlašování veřejných soutěží, nemístně šetří. Je nedostatek „zdravých instinktů“ u zákazníků: kupovat jen to, co potřebuji, být za to ale ochoten zaplatit, vědět, co požaduji.
g) Často chybí spolupráce managementu, koncových uživatelů a pracovníků informatických útvarů zákazníka.
h) Data se nepociťují jako základní nenahraditelné bohatství podniku.

Problémy existují i u systémových integrátorů, mnohdy jsou obdobné jako u zákazníků:

- Podceňují se úvodní fáze projektu (to je do jisté míry stimulováno absencí zdravých instinktů u zákazníků a dalšími výše zmíněnými problémy – viz body b), c), g) výše. Nedostatečně se specifikují požadavky.
- Nezvládání týmové práce, především v týmech, jejichž velikost se mění s časem (najímané týmy, srv. kap. 10).
- Nedostatečná dokumentace, mnohé je dohodnuto jen ústně.
- Nedodržování základních zásad vedení projektů (kontroly, náprava skluzů).
- Potíže s organizací spolupráce se zákazníkem.

Je důležité si uvědomit uvedená nebezpečí. Při systematické práci s důrazem na eliminaci těchto problémů lze podstatně zmenšit rizika při zavádění IS.

5.4.2 Vedení projektu při systémové integraci

Být dobrým vedoucím vyžaduje talent. Vedoucí musí mít i dosti zkušeností. Kvalita vedoucího projektu rozhoduje o úspěchu projektu (kap. 10). U CIS je závislost na vedoucím projektu poněkud menší.

Specifikaci požadavků je nutné nechat schválit budoucímu uživateli IS. Rozhodující slovo by měli mít ti, kteří budou systém přímo používat, koncoví uživatelé. Zhruba 80 % požadavků byl měl formulovat management a koncoví uživatelé, zbytek informatici uživatele. Management provádí všeobecný dohled. Pracovníci managementu samozřejmě mohou být též koncovými uživateli, např. subsystémů podpory rozhodování, pak se účastní specifikace požadavků na tyto subsystémy.

Rozhodujícími předpoklady úspěchu na straně zákazníka jsou podle zkušeností systémových integrátorů:

- podpora a zájem managementů obou stran,
- moderní metody budování IS,
- spolupráce informatiků zákazníka,
- včasné dosažení použitelných dílčích výsledků,
- přijetí přiměřených cílů,
- realistická očekávání přínosu IS.

Na straně dodavatele jsou klíčové podmínky úspěchu:

- zájem a podpora vedení firmy: dohled, pomoc při problémech,
- alokace zdrojů,
- vhodný software,
- prosazení přiměřených cílů,
- kvalitní řešitelé.

Práce při systémové integraci je týmová. Doporučuje se sestavit tým následujícího složení:

1. *Dohlížecí výbor projektu.*
   Členové: Vedoucí projektu, zástupce managementů zákazníka a dodavatele, vedoucí řešitelských týmů, pomocné síly.

Úkol: Celkové věcné (a ekonomické) sledování projektu, řešení problémů, při nichž je nutná účast managementu, a problémů koordinace. Vedoucí projektu bývá pracovník dodavatele IS a jeho zástupcem je stý čný důstojník. Vedoucím projektu může pro snadno customizovatelný IS být i stý čný důstojník.

2. *Řídicí výbor projektu.*
   Členové: Vedoucí projektu, zástupce zákazníka, zástupce vedoucího, administrativa, vedoucí týmů.
   Úkol: Každodenní řízení projektu, řešení věcných problémů, koordinace týmů, přijímání rozhodnutí o projektu jako celku. Dodavatelé otevřených IS (např. Lawson Software) vyžadují, aby byl řídicí tým projektu tvořen převážně pracovníky zákazníka, pracovníci dodavatele pak fungují jako specialisté na systém a konzultanti. Tento přístup je podmíněn moderní architekturou IS uvedené firmy. Rozsáhlá účast pracovníků zákazníka v řízení projektu sama do značné míry automaticky zajišťuje angažovanost zákazníka během instalace IS a bezproblémový provoz. Pracovníci zákazníka pak vystupují spíše jako konzultanti.
3. *Řešitelské týmy.*
   Členové: Vedoucí, zástupce uživatele – zástupce těch, kteří budou užívat daný subsystém, specialisté pro daný subsystém, konzultanti, někdy i programátoři.
   Úkol: Customizace/vývoj subsystému, zavádění subsystému, dokumentace, návrh rozhraní s uživateli subsystému. Zajišťování systémových a jiných služeb.
   Poznámka: Jednotlivé týmy nemusí existovat, případně nemusí mít stálou velikost, po celou dobu projektu. Týmy pro subsystémy mohou samostatně řešit koncepci a metody řešení, specifikovat požadavky, účastnit se školení uživatelů a zavedení IS.

5.5 Problémy výběrových řízení

Většina IS se dnes realizuje na základě výsledků výběrových řízení. Způsob provedení výběrových řízení může být různý. Obvyklé schéma je založeno na následujících kritériích: Cena, termíny, funkce. Funkce se ve skute čnosti nepovažují za nejdůležitější kritérium. Kvalitativní požadavky, např. požadavek otevřenosti, se zřídka stanovují předem. Uchazečům o realizaci pak nezbývá než se podbízet a slibovat více, než je zdrávo. To představuje značné riziko pro úspěch budoucího IS (srv. Hausmann, 1995).

Tento postup je výhodný pro budoucího uživatele IS jen zdánliv ě. Dodavatel totiž musí mít dostatek prostředků na realizaci IS. Jinak bude realizován nepříliš dobrý systém. Jednání s mnoha účastníky soutěže prakticky vylučuje možnost s každým dostatečně důkladně jednat. Proto je volba vítěze soutěže dosti náhodná a může být ovlivněna ne zcela čistými praktikami. Dodavatel za této situace musí blufovat a doufat, že se věci nějak vyřeší. Nemá totiž na vybranou.

Vyhlašovatel soutěže někdy používá zrádnou taktiku „navrhněte funkce sami a my si vybereme“. Scestnost takového přístupu je na základě skutečností uvedených v kap. 2 a v této kapitole zřejmá. Zadavatel soutěže se snaží výsledek soutěže pojistit tím, že prosazuje, aby smlouva byla co nejpodrobnější. Poněvadž nebyla provedena řádná analýza, obsahuje smlouva mnoho nedomyšlených požadavků. Přesvědčení, že smlouva již obsahuje vše, vede k chybnému závěru, že není potřeba, aby zákazník spolupracoval při vývoji IS. To dále zhoršuje vyhlídky na úspěch IS.

Při organizaci veřejných soutěží se využívají služby poradců. Ani to není bez problémů. Poradce není obvykle bezprostředně závislý na kvalitě budoucího informačního systému. Poradce nemusí být nestranný. Není rovněž dostatečná kontrola zkušeností a znalostí poradců. Částečnou pomocí je využívání poradců renomovaných

konzultačních firem, jejíž odměna závisí na skutečně dosažených výsledcích. Rady poradce je i pak třeba brát jen jako „poradní“ hlas. Výhodou poradců bývá znalost ekonomické terminologie a zkušenost z velkého počtu podniků. Nedostatečná znalost ekonomické terminologie bývá jednou z příčin problémů (viz J. Komárek, Počítačové systémy nevyřeší nikdy všechno, Hospodářské noviny, 8. 1. 1996).

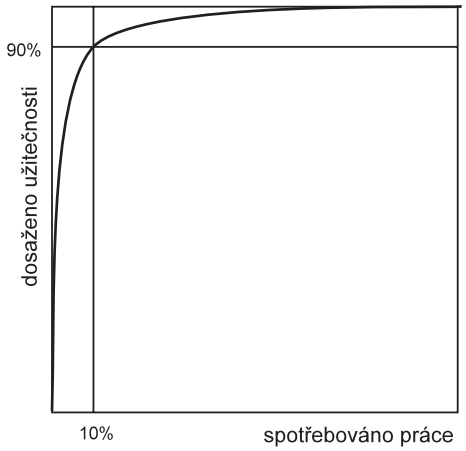
Osvědčuje se dvoustupňová procedura – nejprve se vyberou 2–3 účastníci soutěže do užšího výběru a s nimi se za úplatu provede specifikace cílů způsobem popsaným výše. Na základě výsledků se pak může provést kvalifikovaný výběr vítěze. Nevýhodou jsou vyšší náklady a větší pracnost pro zákazníka a delší termíny. Vyšší náklady se vrátí nejen ve vyšší spolehlivosti výběru. Dobré věci ze všech návrhů lze totiž požadovat od vítěze soutěže. Při výběru účastníků „do druhého kola“ je vhodné kromě kvality nabídky hodnotit následující skutečnosti:

- Měřitelný nebo alespoň kontrolovatelný přínos systému, vhodnost pro daný podnik.
- Reference.
- Realizovatelnost (architektura systémů, řešitelské kapacity dodavatele).
- Kvalita nástrojů, modernost architektury, možnost inkrementálního vývoje (kap. 7).
- Cena a termíny.

  Vítěz soutěže by měl získat ke spolupráci i ty pracovníky, kteří fandili konkurenci.

5.6 Existenční řešení

Jednou z důležitých příčin neúspěchu IS jsou nerealistická očekávání a vyžadování funkcí, jejichž přínos je sporný (srv. kap. 2). Nepotřebné funkce zvyšují cenu produktu a nemůže být s nimi spokojenost. Nevíme-li, co chceme, nemůžeme to dostat. Nepotřebné funkce bývají proto často právě ty, které dají nejvíce práce. Tento fakt se s jistou mírou nadsázky formuluje jako „zákon 90–10“. 90 % užitku přinášejí funkce, které byly realizovány 10 % pracnosti. Předpokládáme-li, že nejdříve se realizují nejužitečnější funkce můžeme „zákon 90–10“ zobrazit způsobem z obr. 5.1. Poznamenejme, že je někdy zmiňována mírnější verze známá jako „zákon 80–20“.



Obr. 5.1: Ilustrace zákona 90–10.

Poměrně účinným prostředkem, jak vyloučit nadbytečné funkce, je tzv. existenční realizace. Postupuje se tak, že se požadavky seřadí podle významu, přičemž se vybírají pouze ty, bez nichž není IS k ničemu dobrý (nepominutelné či kritické požadavky). Požadavky se sdruží do ucelených skupin a ty se pak, pokud možno postupně, realizují a uvádějí do provozu.

Problémy provozu přinesou realističtější pohled na problém. Strategie „všechno naráz“ ve spojení s přesvědčením „co nebude v požadavcích od počátku, to tam nikdy nebude“ značně přispívá k neúspěchu IS. Do značné míry vylučuje použití výhodných technik, jako je inkrementální vývoj (kap. 7), zvyšuje riziko, že použitelné části IS budou oživeny poměrně pozdě a že značné úsilí bude promrháno na nerealizovatelné zbytečnosti. Jednou z cest „existenčního řešení“ je technika rychlého vývoje aplikace (rapid application development – RAD).

5.7 Restrukturalizace činnosti podniku/úřadu (business process reingeneering)

Zavedení IS jen zřídka přináší úspory administrativních pracovníků. Informace se teď zpracovávají rychleji, ale je jich více. Příliš široká funkčnost systému (syndrom dortu pejska a kočičky) někdy dokonce vede k vyšší potřebě administrativních pracovníků a pracovníků informatických oddělení. Ti však zároveň o obsahu IS spolurozhodují. Proto mohou mít zájem na tom, aby byl IS co nejrozsáhlejší.

Z kap. 2 a z předchozích paragrafů víme, že je riskantní kombinovat instalaci IS se současně prováděnou reorganizací. Dlouhodobě je situace jiná. Snadná přístupnost a spolehlivost informací, možnosti různých analýz, orientace IS na ucelené procesy umožňuje, aby jeden vedoucí pracovník mohl řídit větší množství podřízených, kteří navíc mohu díky IS pracovat samostatněji. IS umožňuje, aby při řešení úkolů dynamicky a leckdy spontánně vznikaly neformální týmy napříč standardní organizační strukturou. Členové takového týmu tedy mohou mít různé nadřízené. IS také umožňuje, aby jeden pracovník prováděl činnosti, které dříve patřily do kompetence více oddělení, např. aby kompletně vyřizoval bezproblémové zakázky.

V delším výhledu může tato situace vést k tomu, že se zmenšuje výška organizační pyramidy, tj. počet organizačních úrovní. Musí samozřejmě existovat vůle skutečně střední úroveň řízení časem redukovat. To nebývá často pravda, zvláště v případě státních úřadů. Potřeba zachování práce středních úrovní může být někdy skrytým důvodem podivných požadavků. U dobře pracujících podniků je větší naděje, že se prosadí racionální řešení. Organizační pyramida se „zploští“. Snížení počtu řídících stupňů zvyšuje pružnost práce podniku[2]. Zároveň se snižuje význam formální hierarchie organizační struktury („organizačního pavouka“ – organogramu) a podporuje spolupráce a vznik neformálních týmů podle činností. Organizační struktura se může snáze přizpůsobovat požadavkům rozsáhlých nebo neobvyklých úkolů a snáze se formují řešitelské týmy. Trend redukce středních článků není pozorován u klasických úřadů, neboť jde proti zájmům úředníků. Není příliš silný ani u soukromých podniků, zvláště velkých (podobné důvody).

Pro řadu řídicích a organizačních prací nebývá už při používání IS nutné být přímo „v kanceláři“. Při práci na síti není důležité, kde se dané pracovní místo nalézá, může být i na jiném konci světa. To umožňuje pracovat zčásti nebo úplně doma (teleworking, homeworking). Může to být výhodné pro všechny, pro podnik i pro zaměstnance, především pro ty, kteří mají potíže s mobilitou, jako jsou invalidé. Domácí práce umožňuje snižovat dopravní zátěž měst.

Pro využití teleworkingu je třeba provést řadu opatření počínaje komunikační infrastrukturu a konče organizačními opatřeními u zaměstnavatele, včetně modifikací pracovní náplně. Mohou být i problémy se změnou pra-

2. To je samozřejmě proti zájmům pracovníků střední úrovně řízení a ti mohou zavedení IS sabotovat. S tímto rizikem je nutno počítat.

covních smluv a kontroly práce. Teleworking je vhodný především tam, kde je práci možno jednoduše „úkolovat“, tam, kde je pracovník sám zainteresován na výsledku (obchodníci, vývojoví pracovníci). Je samozřejmě nutné, aby práce nevyžadovala neustálý osobní styk. Za teleworking se považuje taková práce, která přesahuje 20 % pracovní kapacity, tj. alespoň jeden den v týdnu. Teleworking může být jedním z důležitých požadavků při specifikaci cílů. Tento požadavek bude s rozvojem širokopásmových multimediálních datových sítí stále častější.

Klasická organizace práce je typická tím, že na úkolu pracuje mnoho lidí, každý však jen po velmi krátkou dobu. S úkolem se něco děje jen po velmi malou část celkové doby vyřizování, jsou velké mrtvé časy. Většina mrtvých časů jde na vrub zdržením způsobeným přesuny informací mezi odděleními, k předávání informací dochází většinou jednou denně. Dostupnost informací spolu s možností, aby jeden pracovník prováděl ucelené činnosti, např. vyřizoval sám celou zakázku včetně objednávek výroby, umožňuje významné efekty.

Termín business process reingeneering (BPR) označuje postupy umožňující restrukturalizovat klasické způsoby spolupráce a zefektivňovat organizaci činností. Někteří dodavatelé CIS doporučují provádět BPR co nejrychleji. To je ovšem spojeno s řadou rizik. Při BPR je nutné uvážit, že pracovní činnosti jsou obvykle realizovány jako posloupnost kroků, z nichž některé souvisí se zvolenou organizací a nejsou z hlediska výsledku procesu rozhodující, např. přesun dokumentů z jednoho oddělení do jiného oddělení. Takové činnosti nazveme organizační. Jiné kroky jsou pro danou činnost podstatné a v podstatě nezávisí na organizačním členění. Takové činnosti nazveme výkonné nebo technologické. Příkladem technologické činnosti je příjem či výdej zboží ve skladu, technologická operace na obráběcím stroji či účetní operace. Při BPR je snazší měnit případně odstraňovat činnosti organizačního typu. Efekty mohou být velmi významné. Zde je však třeba postupovat opatrně a měnit jen to, co je opravdu nefunkční (Tapscott, 1996).[3]

Prochází-li zakázka deseti odděleními, je pravděpodobné, že bude obvykle vyřizována deset i více dní. Může-li pomocí IS vyřizování zakázky řídit jeden pracovník nebo ad hoc vzniklý tým pracující nad společnými daty, odpadnou zdržení vznikající předáváním zakázky mezi odděleními a doba vyřizování zakázky se zkrátí na několik málo dnů, někdy i hodin. Současně se zlepší kontrola průběhu zakázky a lze rychleji reagovat na různé problémy či dodatečná přání zákazníka. Osvědčuje se, když běžné bezproblémové případy řeší jeden pracovník. Složitější případy pak může řešit tým expertů, opět s podporou informačního systému.

Změny technologických operací jsou mnohem obtížnější a jsou spojeny s mnoha riziky. Obsah výkonné operace je často určen technologickými nebo legislativními požadavky a omezeními. Kvalita provádění výkonných operací často závisí na pracně získaných dovednostech vyžadujících dlouhé studium či školení a obvykle i delší praxi; to je případ účetního, skladníka či kvalifikovaného dělníka. Je nutno počítat i s nižší počítačovou gramotností příslušných pracovníků a s jejich menší schopností a často i menší možností měnit obsah své práce, který může být vymezen požadavky legislativy nebo technologie. Při automatizaci výkonných operací je proto žádoucí, aby se při zavádění systému co nejvíce podobaly své původní „ruční“ formě. Případné změny těchto operací je vhodné provádět postupně a potřebu takových změn pečlivě zvažovat. Výhodou je to, že změny výkonných operací jsou obvykle z hlediska IS lokální. Činnosti je často možné sdružovat.

Vzhledem k rychle se měnícím technologiím a globalizaci trhu roste význam permanentního vzdělávání a školení zaměstnanců. Instalace IS sama o sobě vyvolává potřebu zaškolování a rekvalifikaci. Organizace školení a zvyšování kvalifikace je důležitou a stále významnější složkou personalistiky a strategie rozvoje podniků. Bez školení a rekvalifikace lze jen obtížně využívat výhody, které přináší informační technologie (srv. kap. 13).

3. Enormní náklady na restrukturalizaci nových spolkových zemí v SRN byly pravděpodobně vyvolány i tím, že se úplně restrukturovaly dřívější socialistické podniky, tj. že se zcela zrušila jejich struktura a muselo se začít úplně od začátku. Je otázka, zda to byla ta nejsprávnější cesta.

5.8 Informatické pasti

Investice do IS jsou nemalé. Přínosy nebývají pokaždé přesvědčivé. Na druhé straně se mnohokrát prokázalo, že výpadek IS znamená ohrožení existence podniku. Příčiny nejasnosti přínosů jsme diskutovali výše (nevhodná volba funkcí a cílů, nejasně definované přínosy, neschopnost využívat nové technologie atd.). Důvody ohrožení podniku při dlouhodobém výpadku IS jsou v tom, že si na IS podnik zvykne a ztratí schopnost používat klasické metody řízení. Při výpadku IS pak nikdo neví co dělat. S jistou mírou nadsázky lze IS přirovnat k droze – její užívání nijak nezvýší výkon, nemá-li ji však narkoman k dispozici, má značné potíže a může i zahynout na detoxikační syndrom. Výsledky zavedení IS mohou a často i jsou zklamáním zvláště tehdy, spadneme-li do informatické pasti. Informatickou pastí rozumíme vznik takové nepříznivé situace při nasazování IS, které se lze poměrně snadno vyhnout, kterou však prakticky nelze změnit, pokud nastane. Nejčastěji uvázneme v následujících informatických pastích.

a) „Definitivní“ řešení, neboli všechno a hned.
   Snaha vše vyřešit jednou provždy je typickým příkladem informatické pasti. Pravděpodobnost, že do takové pasti spadneme, značně zvyšuje snaha o bezbřehou funkčnost IS (syndrom dortu pejska a kočičky). Syndrom dortu pejska a kočičky je jedním z projevů toho, že se vyžaduje systém, aniž je vyjasněný účel a přínosy takové investice. Jedním z kritérií pro nákup IS by tedy mělo být to, jak je IS snadno modifikovatelný a jak snadno by se prováděla záměna IS za jiný IS. IS by měl umožňovat postupné zavádění a snadnou modernizaci.
b) Nedomyšlená řešení.
   Jiný druh pasti hrozí při technice rychlého vývoje aplikace – realizují se částečná řešení bez toho, aniž je dostatečně jasné, jak bude fungovat celek. Pak bývá nutné začít zcela od počátku. Tomu se lze vyhnout relativně snadno – stačí realizovat jen to co je potřeba, tj. mít pro každý požadavek dostatečné důvody a sledovat od začátku cesty budoucí integrace, nebo začít od jádra, např. od systému účetnictví, které se postupně rozšiřuje.
c) Nevhodný IS.
   Tendence nakupovat customizovaný software může skrývat další past – koupí se systém, který zčásti nevyhovuje. Náprava je obtížná – pokud IS nevyhovuje, znamená to přechod na nový IS nebo drahé úpravy. Obojí je velmi nákladné.
d) Past údržby.
   Během života IS se mění hardware i podpůrný software. Je-li použitý SW k těmto změnám necitlivý, budou náklady na údržbu u uživatele i dodavatele velmi vysoké (nový hardware pak znamená vždy vícenáklady na přenos IS na novou platformu). Údržba znamená často jen krycí název pro další vývoj pod záminkou zlepšování funkcí. „Zlepšování“ ihned po instalaci bývá projevem nedostatečně kvalitního vývoje, resp. nesprávně nakoupeného softwaru. Vylepšování je velmi nákladné a je spojeno s rizikem, že se zhorší spolehlivost systému. Údržba IS vytváří nepříjemnou závislost zákazníka na dodavateli. Pracnost údržby a modernizace IS závisí na tom, jak kvalitně byl IS navržen a jaké technologie byly při jeho vývoji použity.

Informatické pasti jsou vážnou hrozbou. Ceny IS rostou závratnou rychlostí, kvalita a rozsah jejich funkcí neroste ani zdaleka tak rychle. Tak např. jedna transakce v bance stojí dnes stejně jako před dvaceti lety (Landauer, 1995). Někdy se bankovní operace provede rychleji a IS umožňuje používat bankomaty. To není mnoho, uvážíme-li rozsah investic do bankovního IS. Bankomaty se samozřejmě zavádějí jako důležitý produkt požadovaný zákazníky, ani těm všem nepřináší vždy podstatné výhody, banky však musí zvažovat všechny s tím spojené náklady a rizika.

5.9 Volba hardwaru a základního softwaru

Volba hardwaru a podpůrného softwaru by měla být provedena co nejpozději, nejlépe koncem etapy specifikace požadavků. Tuto zdravou zásadu není možné vždy dodržovat neboť:

a) Zákazník chce mít co nejdříve odhad celkových nákladů na IS a to bez představy, jaký hardware a software nakoupí, není možné.
b) Často je zájem využít dosavadní hardware. To není většinou výhodné a přináší to jen omezené úspory.

Jako mnohdy v životě je vhodné jít i zde cestou nepříliš škodlivého kompromisu. Hardwarové nároky softwaru rostou. Přibývají nové funkce, rozšiřuje se počet uživatelů. Operační a databázové systémy a aplikace jsou stále nenasytnější. Hardware by se měl proto navrhovat s jistou výkonnostní rezervou. Vzhledem k poklesu cen hardwaru o daném výkonu však není vhodné, aby byla rezerva příliš velká. To neplatí pro komunikační cesty, pokud je výměna obtížná z technických důvodů (obtížnost fyzické výměny kabelů) a z provozních důvodů (bývá nutné zajišťovat nepřetržitý provoz).

Nejlepší řešení je systém navrhnout tak, aby přirozený růst IS nebyl limitován vlastnostmi hardwaru, tj. aby IS byl nezávislý na hardwaru a základním softwaru. Jinými slovy je žádoucí, aby IS byl přenositelný (portabilní). Je výhodné používat portabilní operační systémy a s nimi spolupracující databázové systémy, ty jsou pak rovněž portabilní. Portabilita je nutná i proto, že se za dobu života IS HW a ZSW změní. Tento požadavek je splněn např. pro operační systémy UNIX a Windows NT a pro databázové systémy vyšší třídy, jako je ORACLE, SYBASE, Informix, Progress atd. Této podmínce zatím většinou nevyhovují levné databáze orientované na PC. Volba databáze je klíčovým problémem IS. Databáze by měla mít prostředky umožňující souběžnou práci více uživatelů a prostředky koordinace jejich práce při konfliktních požadavcích na změny v datech. To technicky znamená požadavek, aby databáze podporovaly transakce. Podpora standardu SQL je samozřejmostí.

Žádoucí je možnost podle potřeby balancovat zátěž klientů a serverů při aplikaci architektury klient – server. Databáze by měla být prověřena na aplikacích s rozsahem dat podstatně větším, než se zatím předpokládá v budovaném IS. S těmito podmínkami se zatím obtížně vyrovnávají databáze původně navržené pro malé aplikace pracující na PC. Týká se to především transakcí. Použití těchto databázových systémů na rozsáhlé aplikace (více než 100 pracovních míst) je zatím riskantní a znamená vysokou pracnost realizace. SW firmy s dobrými programátory jsou schopny tyto nevýhody eliminovat a těžit z nízké ceny databázových systémů pro PC.

Kvalita a možnosti databázových systémů se rychle zlepšují. To jednoznačně vede k tomu, že IS obvykle nepracují se systémem správy dat speciálně vyvinutým pro jeho potřeby. Lze také využít dlouhodobé stability předních počítačových firem (IBM, H-P, Digital atd.) a použít jejich software a hardware. Portabilita na vyšší modely vlastní výroby je u těchto firem zaručena. Nevýhodou může být menší volnost při volbě HW a databáze. Převažují výhody: dobrá úroveň systémové integrace, kvalita služeb, renomé dodavatelů u zákazníků atd. Velké firmy, jako je IBM, jsou zároveň výrobci hardwaru. Mají proto zájem vytvořit závislost na svých hardwarových výrobcích. Pro softwarové firmy může být spolupráce s velkými firmami ztížena podmínkami pro práci dealerů: poněkud menší samostatnost dealera, některé firmy síť dealerů nebudují, omezení při volbě databázového systému atd.

Při volbě HW a podpůrného softwaru a také při výběru CIS je třeba vycházet z toho, že v IS jsou nejcennější data. Hardware i software lze koupit či vyměnit. Jednou ztracená data nelze nahradit nikdy. Ochrana dat znamená vyšší nároky na hardware (zařízení na ukládání záložních kopií, záložní zdroje, využití metody dvojího zápisu, ochrana proti zneužití dat neoprávněnými subjekty atd.), ale také na software (větší spolehlivost, transakční zpracování dat atd.).

Volba HW, základního softwaru a IS rozhodujícím způsobem závisí na tom, jaké prostředky může zákazník do IS investovat. IS renomovaných firem jsou velmi drahé. Není výjimkou, že cena IS překročí hranici 100 milionů Kč.

5.10 Vyhodnocování rizik

Rizikem je míněn možný výskyt libovolného jevu, který může způsobit neúspěch nebo závažné potíže či ztráty. K odhadu rizik je nutné jednoznačně stanovit, co je vlastně úspěchem (srv. Grey, 1995). Je třeba vytipovat ty uživatele, jejichž stanovisko je rozhodující pro hodnocení výsledků. S tím souvisí potřeba stanovení kritérií či podmínek umožňujících rozhodnout, zda byly dosaženy plánované cíle či nikoliv. Absence takových měr a kritérií je sama o sobě rizikem.

Volba termínů, kdy se rizika vyhodnocují, se provádí ve vazbě na plán realizace projektu – co má být dodáno, funkčnost, potřebné zdroje a plán aktivit ve formě sítě používané při metodě kritické cesty (kap. 9). Ve vazbě na kritická místa plánu se identifikují jednotlivá rizika a jejich příčiny. Seznam rizik je zpřesňován během specifikace požadavků.

Po identifikaci rizik je třeba ocenit míru rizika vážící se na danou příčinu či kombinaci příčin. Odhad rizik může být od kvalitativního (vysoké, střední, nízké riziko) až k postupům založeným na kvantitativních modelech umožňujících uplatnění matematických případně simulačních metod.

Při odhadu rizik jsou nejčastěji používány následující techniky:

- stanovení seznamu případů (technické, obchodní, vnitřní, externí), které mohou mít nežádoucí následky,
- kvalitativní metody založené na skóre (velký vliv/zanedbatelný vliv, hodnocení vlivu nějakého faktoru stupnicí 1 až 10 atd.),
- kvantitativní techniky odhadu pravděpodobnosti a závažnosti vlivu příslušného rizikového faktoru na obvyklé plánovací charakteristiky jako čas a náklady.

Kvantitativní metody jsou většinou založeny na síťovém grafu projektu (kap. 9). Uzly grafu zobrazují činnosti. V uzlech jsou uváděny doby provedení příslušné činnosti. Při vyhodnocování rizik se pro každou činnost v síti odhadne rozložení pravděpodobnosti odchylek od doby řešení a odchylek od plánované ceny pro danou aktivitu. Rozložení pravděpodobnosti je odvozeno z trojice odhadů: minimálně možná hodnota, nejpravděpodobnější hodnota, maximální možná hodnota. Je možné zadávat i podrobnější specifikaci rozložení pravděpodobnosti, obvykle však bývá obtížné získat pro takovou specifikaci dostatek spolehlivých dat. Z pravděpodobností odchylek pro jednotlivé činnosti se zjišťují odchylky pro celý projekt užitím simulačních metod. Prostředky analýzy rizik jsou součástí některých CASE systémů a některých systémů řízení projektů. Odhad rizik je důležitou částí řízení větších softwarových projektů. Vědomí možných rizik může při adekvátní reakci samo o sobě podstatně snížit jejich pravděpodobnost. Důsledky rizikových faktorů lze snížit již tím, že je včas připravena reakce na výskyt rizikové události.

Pro ilustraci uveďme výčet nejběžnějších rizikových faktorů softwarových projektů:

Hardware: Opožděná instalace, nedostatečný výkon, nevhodné vlastnosti, nefunguje, jak bylo slíbeno – to se stává často při oživování počítačových sítí: chyby v kabeláži, nedodržení podmínek instalace, neúplná dodávka, poškození při dopravě, selhání dodavatele, nedodržení dohod, slabá podpora ze strany dodavatele, nedodržení záruk.

Podpůrný (základní) software: Opožděná instalace, nevhodný pro daný hardware, nesprávná či nevhodná funkčnost, nedostatečná dokumentace (zvláště záporných vlastností), nedostatečná podpora od dodavatele, chyby v konfiguraci.

Lidský faktor: Fluktuace, nemoci, nedostatečné schopnosti, nedostatečné nebo příliš pozdní školení, nedostatečná kvalifikace a zkušenosti, neschopnost týmové práce.

Management: Špatně stanovené termíny a cena, nedostatečné finanční krytí, změna manažera, organizační neschopnost vést projekt, špatně zvolený partner, chyby v hospodářské smlouvě, nevhodné stanovení cílů, nekvalitní plán realizace.

Uživatel: Slabá podpora spolupráce, nevstřícnost, není zajištěna spolupráce s koncovými uživateli, tj. s těmi, kteří budou IS skutečně používat, neúčast na společných pracích, neplatí, žádá stále změny, nezvládne systém, nelze získat potřebná data, změny podmínek u uživatele, např. změna majitele (zde je třeba se před následky bránit ve smlouvě), nebezpečí bankrotu, přechod na IS klade na uživatele příliš velké požadavky, nepřesnost či nespolehlivost dat.

Analýza rizik obvykle navazuje na analýzu kritických (nepominutelných) požadavků. Nesplnění kritického požadavku znamená výrazné riziko. Cílem analýzy kritických požadavků je nejen rozpoznat hlavní požadavky, ale také se souhlasem uživatele rozdělit požadavky na kritické, méně důležité a nepodstatné. Analýzu kritických požadavků lze rovněž použít k rozboru alternativ řešení – co realizovat a v jakém pořadí a v jakých kombinacích. U kritických požadavků se hodnotí rizika nevyhovění požadavku a také rizika a problémy spojené s implementací požadavku.

Pro každý kritický požadavek se hledá odpověď na následující otázky:

- má požadavek opravdu velký až kritický vliv na užitečnost IS?
- pokud ano, jaké konkrétní parametry činnosti uživatele ovlivňuje. Tyto parametry by měly být kvalifikovatelné.

Příklady: vyřizování zakázky se zkrátí z měsíce na čtrnáct dnů, snížení zásob o 10 %, platby se kontrolují týdně, atd.

Formálně se při analýze kritických požadavků postupuje následovně:

1. Stanovení podnikových cílů a kritických požadavků na IS. Vymezení priorit cílů. Pokud jsou cíle nezávislé nebo je nelze rozumně integrovat, je vhodné je řešit jako separátní projekty. S návrhem cílů musí souhlasit management.
2. Stanovení kritických oblastí výkonnosti: které důležité činnosti neexistují a které je třeba zlepšit kvantitativně i kvalitativně.
3. Pokrytí jednotlivých kritických oblastí funkcemi: která funkce jak rychle co řeší. Funkce je vhodné analyzovat na základě analýzy kritických oblastí výkonnosti (critical performance areas). Při tom se využívá techniky postupné dekompozice. Dekompozice je založena na rozkladu kritického požadavku na požadavky elementárnější. Tak např. požadavek rychlejšího vyřizování pohledávek se rozkládá na požadavek rychlejšího zaznamenávání požadavků do databáze, rychlejší analýzy existujících pohledávek a rychlejší generace urgencí. Rychlejší a úplnější analýza pohledávek může být rozložena do úkolu detekce ekonomicky zajímavých případů a do procesu rozhodování, jak s jednotlivými případy naložit.

5.11 Příklad analýzy kritických požadavků

Uveďme postup vyhodnocování rizik v metodologii SSADM na příkladu analýzy systému vyhodnocování pohledávek.

A) **Vyhodnocení kritických požadavků (KP)**.

a) *Stanovení kritických požadavků.*
Zkvalitnit práci při vyřizování pohledávek, např. rychlým zjišťováním neplatičů, a zmenšit provozní náklady na tuto činnost.
Stručně cíl: Hospodárně řešit pohledávky.

b) *Kritické oblasti výkonnosti.*
Hlavní požadavek: Hospodárně řešit pohledávky.
Kritické oblasti:

KP 1. Přesun pohledávky do oddělení fakturace nejpozději do vzniku práva účtovat. Při bližším pohledu se úkol přesunu pohledávky dělí na dva podúkoly:
KP 1.1. Ověření pohledávky: relevantnost, splnění formálních náležitostí.
KP 1.2. Registrace pohledávky.

KP 2. Sledování pohledávek ve stanovenou dobu po termínu splatnosti s cílem provedení nápravných akcí, jako jsou upomínky a žaloby. Úkol sledování pohledávek se člení na podúkoly:
KP 2.1. Vyhledání pohledávky.
KP 2.2. Vyhodnocení stavu pohledávky.

KP 3. Příprava akcí: Poloautomatická příprava podkladů pro urgence a soudní řízení. Příprava akcí se člení na:
KP 3.1. Generace dokladů.
KP 3.2. Odesílání dokladů.

KP 4. Aktualizace pohledávek: Reakce po pohybech na účtu, kam má přijít platba pohledávky, např. zastavení akcí u soudu po obdržení platby. Aktualizace pohledávek má podúkoly:
KP 4.1. Vyhledávání pohledávek.[4]
KP 4.2. Záznam změn.

KP 5. Efektivně provádět jednotlivé operace.

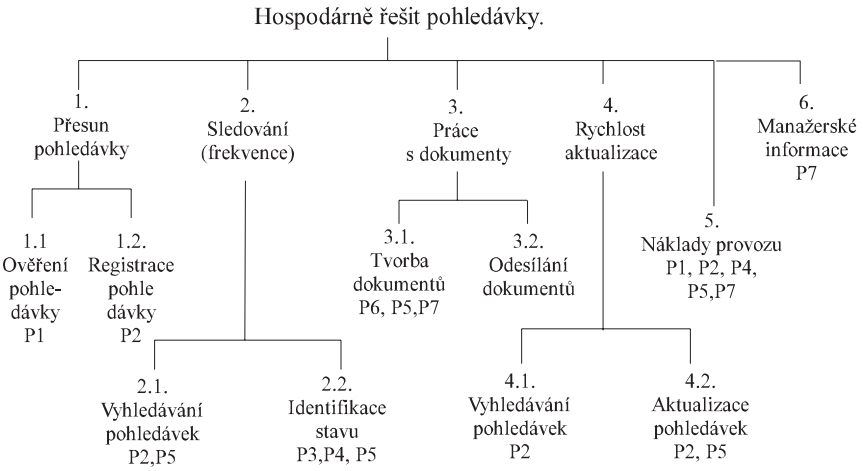
B) **Kvalitativní a kvantitativní požadavky zákazníka.**
Na základě analýzy kritických požadavků a analýzy situace v podniku byly zformulovány následující cíle:

Cíl 1. Počet pohledávek nesplacených více než 10 dnů po splatnosti k počtu splacených: Dnes 2:1, požadováno 10:9. Důvod cíle: Urgovat platby u většího procenta pohledávek.

Cíl 2. Náklady na urgenci jedné pohledávky snížit třikrát, tj. z 250 Kč na 70 Kč. Důvod cíle: Lze s pozitivním efektem vymáhat i malé pohledávky počínaje pohledávkami ve výši asi 120 Kč, lze ušetřit pracovníky v oddělení fakturace.

C) **Konkretizace požadavků do tvaru podcílů:**
C1 Snížit průměrnou dobu přesunu pohledávek ze 4 dnů na jeden den.
C2 Vyhodnocování a kontroly prováděné dosud jednou za měsíc provádět jednou týdně.
C3 Doba vyřizování korespondence: Den jako dosud, ale s menší pracností.

4. Nemusí být stejné jako KP 2.1.



Obr. 5.2: Přiřazení problémů a rizik kritickým požadavkům. Problém má být v grafu co nejníže, pokud se problém řeší ve více místech, přenáší se jeho řešení „společného předka“, tj. do nejníže položeného místa, kterým prochází cesty z kořene stromu do všech míst, kde se řeší daný problém.

C4 Aktualizace pohledávky: Provádět denně místo jednou za týden. Optimální by však bylo provádění ihned po tom, co je k dispozici potřebná informace.

R) **Povinné požadavky:**

R1 Pohledávky vymáhat podle zákona.

R2 Přístup k datům podle povinných norem.

D) **Vyhodnocení problémů**.

Stanoví se problémy. Pro každý problém se zjistí, ve kterých kritických požadavcích je nutno problém řešit.

P1 Chybně vedené pohledávky řešit v požadavcích:

KP 1.1 Ověření přesnosti pohledávek – párování s fakturami.

KP 5.1 Snížit provozní náklady na evidenci a sledování pohledávky.

P2 Neefektivní ruční práce. Vztah ke kritickým požadavkům:

KP 1.2 Registrace pohledávek.

KP 2.1 Vyhledávání pohledávek.

KP 4.1 Vyhledávání pohledávek.

KP 5.0 Pracnost sledovat při všech činnostech.

P3 Nedokonalá kontrola pohledávek. Řešit v kritickém požadavku:

KP 2.1 Sledování pohledávek. Dosud jednou za měsíc a tedy pomalu, často nepřesně.

P4 Adekvátnost a rychlost rozhodnutí, zda je nutná urgence. Souvisí s kritickými požadavky:

KP 2.2 Identifikace stavu účtu a potřebných opatření.

KP 5.0 Snížit náklady na provedení operací podle KP 2.2.

P5 Resty, jako je opožděná evidence plateb a změny adres partnerů:

KP 2.2 Identifikace stavu pohledávky.
KP 3.1 Tvorba dokladů (přesnost, úplnost dokumentů, včasnost).
KP 5.2 Vyhodnocení nákladů na provedení.

P6 Tvorba dokumentů (musí odpovídat právním požadavkům). Řešit v:
KP 3.1 Generace dokumentů.

P7 Tvorba měsíčních statistik – manažerské informace:
KP 5.3 Náklady na generaci dokumentů a statistik a používání interaktivních dotazů.

Osvědčuje se zobrazit požadavky a problémy graficky, jak je uvedeno na obr. 5.2.

E) **Návrh řešení:**
Problém P1 (viz. KP 1.1, KP 4.1) „Nesprávné pohledávky“: Vyžádání zásahu operátora, který bude mít právo přístupu k fakturám.
Problém P2 (KP 1.2, KP 2.1, KP 4.1, KP 5) Neefektivní ruční registrace. Řešit tím, že se pohledávky zpřístupní uložením do databáze, do které budou mít interaktivní přístup všichni oprávnění pracovníci.
Problém P3 (KP 2). Nedokonalá kontrola. Řešit automatickým vyhledáváním pohledávek podle předem známých i uživatelem zadávaných kritérií.
Problém P4. Stavy pohledávek (KP 2.2, KP 5). Výběr pohledávky se provádí na základě informací, že prošel termín určité činnosti.
Problém P5 Nedodělky („resty“, KP 2.2, KP 3.1, KP 3). Bude řešeno integrací dat (změna adresy se odvodí např. ze změny údajů na dodacím listě), vytvoří se aparát „párování plateb a faktur“ a prostředky evidence dat.

F) **Úspory:**
Na základě odhadů managementu stanoven cíl ušetřit 15 pracovníků. Při zahrnutí pojištění, daní a režie to znamená úsporu asi 3 mil. Kč za rok.
Úspory na prostředcích vázaných na faktury: Zkrácení průměrné doby proplacení o 14 dnů a výnosy z dříve neurgovaných pohledávek přinese okolo 5 mil. Kč za rok. Při několika desítkách pracovníků v oddělení fakturace musí být roční obrat firmy řádově stovky milionů Kč. Zlepšení platební kázně zákazníků přinese procenta obratu, tedy miliony Kč ročně.
Snížení skladových zásob přinese úsporu několik milionů Kč jednorázově.
Lepší informace pro management a lepší podpora obchodní činnosti, např. snížení počtu reklamací odběratelů, možnost častěji vyhovět přání zákazníků přinese pravděpodobně více než 10 mil. Kč za rok. Odhad je obtížný. Jen zvětšení obratu o deset procent přinese efekt v řádu milionů. Vyžaduje to však nástroje vizualizace dat, umožňující rychlou orientaci managementu. Je zřejmé, že v našem příkladě i podstatná úspora pracovníků nepředstavuje největší přínos IS. Ten je v celkovém zlepšení chodu firmy.

5.12 Shrnutí problémů počátečních etap vývoje a customizace IS

Základní problémy vývoje softwaru jsou v počátečních etapách vývoje. Pokusme se proto nyní o jejich shrnutí.

1. Podceňování počátečních etap.
   Hlavním problémem počátečních etap je podcenění jejich významu. Přežívá představa, že hlavní je napsat pěkný program či navrhnout pěknou obrazovku. Pokud však není jasné, co má systém přinést, nemůže být ani nejhezčí obrazovka správným řešením. Záludnost počátečních etap je v tom, že výsledkem práce jsou dokumenty, které skalní programátoři, a nejenom oni, odbývají slovem „řeči“. Dokumenty vytvářené během

počátečních etap vývoje IS mohou být zřídka kdy plně formalizovány. Nemohou tedy být ani prověřeny formalizovaným testem. U zákazníků jsou počáteční etapy podceňovány jako důsledek nedocenění komplikací a problémů při vývoji i customizaci IS.

Počáteční etapy jsou zdrojem největšího počtu chyb: specifikace asi 41 %, návrh 26 %, volba dat a volba rozhraní po 6 %. Odstraňování chyb vzniklých v počátečních etapách je zároveň nejnákladnější (kap. 15). Ztráty způsobené chybami jsou tedy alespoň z 80 % způsobeny chybami vzniklými v počátečních etapách vývoje. Chybám při specifikacích se nevyhneme ani u customizovaných systémů. Zde však trochu pomáhá know-how výrobce systému. Částečným řešením jsou metody inspekce a auditu (kap. 8).

2. Vedení projektu.
   Problémy počátečních etap jsou velmi často důsledkem problémů s řízením projektu. Minimum požadavků na řízení projektů vymezuje následující seznam:
   a) Musí existovat formálně jmenované vedení projektu. Vedení projektu tvoří vedoucí projektu, jeho zástupce a podpůrný aparát (srv. paragraf 4.4). Nutná je účast zástupce zákazníka (styčný důstojník). Ve vedení projektu by měli být řešitelé subsystému, pokud subsystémy řeší samostatné týmy. Je vhodné, aby se práce řešitelského týmu účastnili i další pracovníci uživatele.
   b) Existuje aparát stanovování úkolů a jejich kontroly, pravidla přejímání výsledků, kontrolní dny, vnitřní oponentury, audit.
   c) Jsou k dispozici metody vedení týmové práce a prostředky podpory práce v týmu.
   d) Zadání, úkoly a převzetí výsledků se zaznamenávají v písemné formě. Vedou se zápisy z jednání a schůzí. Všechna tato pravidla je nutné konkretizovat nejpozději koncem etapy „stanovení cílů“.
3. Organizační problémy.
   V podniku s neujasněnými cíli, s nejasnou dělbou práce a celkovým nepořádkem může nasazení IS přinést i zhoršení situace. IS nemůže odstranit problémy managementu podniku. Je nebezpečné provádět podstatné organizační změny současně se zaváděním IS. Současné změny organizace a zavádění IS jsou náročné pro pracovníky, především však ohrožují kvalitu specifikace požadavků. Pracovník zákazníka je obvykle schopen definovat požadavky pouze ve vztahu k existující situaci a z ní vyplývajícího vymezení pracovní náplně.
4. Syndrom dortu pejska a kočičky.
   Snaha o co nejširší až bezbřehou funkčnost ohrožuje především projekty pro státní správu. Nehrozí tolik u dobře fungujících soukromých firem, i když i tam se může projevit. U customizovaných IS jsou účinnou obranou cena licencí, úprav a obchodní podmínky (záruky).
5. Nadbytečná přesnost.
   Praktické dosažitelné cíle musí vycházet z kvality, tj. přesnosti a včasnosti dat a také z ceny jejich pořizování. Je proto nutné požadovat jen takovou přesnost řešení, kterou umožňují data a která je skutečně potřeba. Přílišná přesnost se často vyžaduje u algoritmů rozhodování (kap. 3, kap. 21).
6. Předčasné řešení technických problémů.
   Nebývá výhodné řešit technické problémy (*jak*, *na čem*) v době, kdy není dostatečně promyšlen cíl a požadavky na řešení. Rychlé řešení technických otázek vytváří zdání rychlého postupu prací a zakrývá podstatu problémů. Obranou jsou vnitřní oponentury prováděné např. formou inspekce požadavků a sledování ucelených aktivit (procesů) zákazníka. Předčasné řešení technických problémů může být skryto i v pokusech o příliš formalizovanou specifikaci požadavků.
7. Zamlčené předpoklady, opominuté souvislosti.

Řada požadavků zákazníka je založena na předpokladech, které zákazník považuje za tak samozřejmé, že je neuvádí. Pohled uživatelů je pohled optikou jejich každodenní činnosti. Tvoří tedy obvykle jediný kamínek mozaiky, kterou musí nakonec vytvořit dodavatel IS. Při popisu svých úkolů mají pracovníci tendenci opomíjet méně časté případy, především ty, při kterých spolupracují s jinými pracovníky na řešení nestandardních situací.

8. Nedostatečná analýza dat.
   Základní podmínkou fungování IS je dostupnost, přesnost a aktuálnost dat. Zdrojem dat je především operativa. U dat je třeba stanovit, kde se tvoří, kde se používají, frekvence vzniku, rozsah, doba uchovávání. Je žádoucí zmínit možné použití dat v budoucnosti a ověřit, zda jsou k dispozici spolehlivá data pro požadované funkce.[5] Takové činnosti se někdy provádějí nikoliv v rámci jednoho projektu, ale v rámci dlouhodobého plánování informací a informační politiky pro celou organizaci (strategic information planning)
9. Měřitelnost výsledků.
   Efekty nasazení IS by měly být stanoveny explicitně v počátečních etapách, především při formulaci cílů, a to pokud možno v měřitelné formě, tj. kvantitativně, nebo alespoň ve formě kontrolovatelných kvalitativních požadavků.
10. Volba termínů.
    Volba optimálních termínů je věcí zkušenosti, intuice a štěstí. Nejsou výhodné ani příliš ostré termíny, kdy hrozí nebezpečí, že bude realizován nekvalitní produkt a ten neuspěje. Příliš ostré termíny si vynucují šturmování, což zvyšuje pracnost. Nejsou však vhodné ani termíny příliš měkké. Ty vedou paradoxně také ke zvyšování pracnosti. Nepracuje-li se dostatečně soustředěně, rostou náklady u dodavatele systému, neboť se práce přerušují ve prospěch urgentnějších projektů, a pak se mnohdy musí začínat znovu (srv. kap. 15 a obr. 15.6).

5.13 Jazyk dokumentů počátečních etap budování IS

V úvodních etapách životního cyklu přecházíme od intuitivních a často i vágních představ a úmyslů k poměrně exaktní formulaci požadavků, tj. vymezení toho, co se bude nakonec realizovat. V této části životního cyklu je značné nebezpečí, že se odchýlíme od původního záměru. Specifikace požadavků musí zohledňovat vlastnosti počítače a dostupného softwaru a vyžaduje jistou praxi. Proto se neosvědčuje, když specifikaci požadavků dělá výhradně uživatel, který často pro stromy nevidí les.

Při specifikaci požadavků je ovšem potřeba ověřovat u budoucích uživatelů, zda to, co navrhujeme, pokrývá potřeby. Zde je problém společného jazyka partnerů. Specifikace požadavků musí být srozumitelná dodavatelům IS i budoucím uživatelům IS. To není snadný úkol, protože jazyk a pojmy různých profesí (např. informatik a ekonom) bývá velmi odlišný. Různé profese mívají různé požadavky na přesnost vyjadřování a různá kritéria pro to,

5. O tom, jak snadno může dojít k problémům, svědčí příklad českého zdravotnictví. Placení lékařů podle výkonů zaznamenávaných do IS bylo založeno na mlčky učiněném předpokladu, že je odhad pracností lékařských výkonů dostatečně přesný a že hlášené výkony lze pomocí počítačů kontrolovat. Obojí nebylo samozřejmě pravda, což se dalo odhadnout již na začátku. Oceňování výkonů bylo založeno na dosti nepřesných odhadech a množství dat nebylo možné ani s pomocí počítačů rozumně kontrolovat. Navíc byly i věcné problémy spojené s kontrolou a regulací. Fakticky byla nejspolehlivější data z doby státních plateb v dřívějších létech. To si ale nikdo nepřipustil, poněvadž by to znamenalo jinak koncipovat systém plateb, blíže k tomu systému, který je obvyklý v západní Evropě. To mnozí považovali za zpátečnické. Enormní investice do infrastruktury zdravotních pojišťoven byly do značné míry zbytečné. Vše významně přispělo ke krizi českého zdravotnictví.

co je pro řešení problému důležité a co nikoliv – mají různá základní paradigmata. Vzhledem k problému společného jazyka partnerů se při specifikaci požadavků na IS osvědčuje forma odborného článku, tj. forma používaná ve vědeckých a technických publikacích. Základem je přirozený jazyk. Takové řešení je prakticky nevyhnutelné u formulace cílů, kdy formulujeme intuitivní představy, a má mnohé výhody i při specifikaci požadavků. Poznamenejme, že popis funkcí v přirozeném jazyce má velký význam pro údržbu (srv. Guinares, 1985).

Použití jazyka odborných článků (včetně grafických metod, vzorců atd.) má při specifikaci požadavků na IS následující výhody:

1. Jazyk odborných článků umožňuje plynulý přechod od formulace cílů k formulaci požadavků. Specifikace požadavků se koncipuje jako zpřesnění cílů projektu; požadavky se formulují v jazyce blízkém jazyku dokumentu o cílech projektu. Při vnitřních oponenturách lze pak snáze ověřit, zda je realizován původní záměr.
2. Na používaném jazyce se mohou partneři poměrně snadno dohodnout.
3. Jazyk odborných článků umožňuje postupný přechod od vágních formulací k přesným definicím. Tato vlastnost je výhodná především v počáteční etapě prací, kdy nelze ještě vše definovat přesně a úplně – pak je výhodné, že můžeme být tak přesní, jak je to při dané úrovni znalostí možné.

Jazyk odborných článků je založen na částečné formalizaci přirozeného jazyka a má jako přirozený jazyk tyto nevýhody:

1. Dokumenty mohou i při konečné redakci obsahovat nejasné, nepřesné nebo víceznačné formulace.
2. Pro specifikaci v jazyce odborných článků se jen obtížně provádí matematický důkaz správnosti. Tento problém lze zčásti oslabit prováděním oponentur uvedených v kap. 8.
3. Jazyk je volen tak, že odpovídá okamžitým potřebám. Proto může být nedokonalý.
4. Jazyk odborných článků nelze mechanicky převést na program – prototyp. To ztěžuje tvorbu prototypů a oddaluje okamžik, kdy je možné ověřit správnost návrhu provedením funkcí systému; často lze testovat funkce až v okamžiku realizace cílových programů – a to může být příliš pozdě.

Specifikaci požadavků může být výhodné napsat ve specializovaném specifikačním jazyce. Předpokladem je, že buď už není nutná spolupráce se zákazníkem, nebo zákazník specifikačnímu jazyku rozumí. Specifikační jazyk mívá formu zvláštního „programovacího jazyka“, který je obvykle interaktivní a „neprocedurární“, tj. definuje spíše to, co je třeba, než přesný postup, jak toho dosáhnout. Někdy se používá programovací jazyk PROLOG. Existuje norma pro specifikace v jazyce Ada. Zápis specifikace požadavků lze za jistých okolností formálně převést do proveditelného programu – softwarového prototypu. Prototyp lze pak provést, a tím testovat správnost specifikací. Prototypy generované právě uvedeným způsobem ověřují pouze některé aspekty systému, jiné, jako je např. doba odezvy, nemohou ověřit. Existují úlohy, jako jsou numerické algoritmy a algoritmy kombinační, kde je tvorba prototypů značně ztížena; realizace prototypu je v těchto případech prakticky identická s realizací cílového programu. Specifikační jazyk musí být zpravidla srozumitelný oběma partnerům. To je velmi ostrý požadavek a daří se ho splnit jen při dlouhodobější spolupráci nebo tehdy, když programátoři vyvíjejí software obecného určení, a nemusí se tedy domlouvat s uživateli.

Mnohé specifikační jazyky jsou v mnoha směrech podobné jazykům programovacím. Požadavek používání specifikačního jazyka tedy může znamenat požadavek zvládnutí základů programování nebo přinejmenším požadavek přesného vyjadřování v oblasti, kde není uživatel odborník. Spolupráci s uživatelem může někdy usnadnit takový formální specifikační jazyk, který je „šit na míru“ řešené problematice. Specializované specifikační jazyky mohou přesně vyjádřit požadavky a tedy snížit nebezpečí nedorozumění. Tento efekt však přinesou jen tehdy, když jsou opravdu zvládnuty všemi zúčastněnými. V opačném případě mohou být výsledky horší než při

specifikacích formou odborného článku. V kap. 15 ukážeme, jak snadno může chybné použití formální metody matematické statistiky vést k chybným závěrům.

Formální metody specifikace mohou být používány jen velmi kvalifikovanými týmy a jsou vhodnější pro software, při jehož vývoji není nutná spolupráce s uživateli. Mají-li specifikační metody tvar matematických teorií, jak je tomu u algebraických specifikací, je v principu možné provést důkaz správnosti. Bohužel je tato metoda velmi pracná a u IS v podstatě nepoužitelná. Formalizované specifikační postupy mají následující výhody:

1. Umožňují přesné vyjadřování a snáze se při nich dodržuje disciplína.
2. Lze pro ně provést snáze důkaz správnosti.
3. Někdy mohou být automaticky transformovány na prototypové řešení a tím usnadňují prověření správnosti specifikací.
4. Lze snáze zkontrolovat, zda výsledné programy odpovídají specifikacím.

Formalizované specifikační jazyky mají následující nevýhody:

1. V případě IS je obvykle nutné, aby specifikačnímu jazyku rozuměli všichni zúčastnění. Pokud tomu tak není, nelze očekávat dobré výsledky. To je obvykle případ vysoce formalizovaných specifikací algebraického typu.
2. Zápis požadavků ve specifikačním jazyku je složitější a pracnější než u specifikací formou odborného článku.
3. U těch částí specifikace cílů, které nelze testovat na prototypech, je větší nebezpečí, že specifikujeme (přesně) něco jiného, než chceme (podobnost procesu specifikování s programováním znamená, že začínáme de facto programovat příliš brzy).

Bjørner, autor specifikačního jazyka VDM, používá následující obrat. Projekční skupina definuje rozhraní mezi částmi systému formálními postupy blízkými algebraickým specifikacím. Ty pak reformuluje v jazyce odborných článků pro jednání se zákazníkem.

U CIS má specifikace požadavků formu relativně formalizovaného dokumentu obsahujícího souhrn požadavků uživatele ve formě doporučované výrobcem systému. I při customizaci IS je vhodnější specifikace požadavků ve formě odborného článku než ve formě silně formální matematické teorie, která je vhodnější pro takové úkoly, jako je specifikace komunikačního protokolu, kdy je cíl znám a nezávisí na konkrétním zákazníkovi. Ve většině technických oborů je formalizace formou odborného článku s výkresy a vzorci standardní metodou, srv. např. projekt letadla.

U rozsáhlejších systémů je dokument Specifikace požadavků značně objemný. Pokud není dokument vhodně členěn, je nebezpečí, že nebude řešiteli navazujících etap zvládnut a často ani přečten se všemi z toho plynoucími důsledky. Je proto nutné, aby byly specifikace vhodně hierarchicky dekomponovány tak, aby každá úroveň hierarchické dekompozice byla snadno zvládnutelná jediným pracovníkem a byla pokud možno oponovatelná na jednu inspekci (kap. 8). Popis jedné úrovně hierarchie by měl proto být na několika málo stránkách. To je možné pouze při používání vhodných pravidel návrhu systému jako celku, jako je skrývání informace a metody dekompozice systému. Struktura specifikací tedy do jisté míry předznamenává metody spolupráce komponent systému a předurčuje i architekturu systému. Dekompozici by měl podporovat použitý specifikační jazyk.

Specifikace musí být rovněž snadno modifikovatelné, poněvadž:

- nebývají bez chyb,
- v době formulace specifikací nejsou známa všechna fakta a práce musí pokračovat – specifikace tedy musí být neúplné. Tento fakt má být ve specifikacích jednoznačně uveden ve formě umožňující snadno zjistit:
  (1) co chybí;
  (2) kde chybí;
  (3) jak bude doplněno;
- je nutné počítat s údržbou systému, a tedy se změnami za provozu systému.

5.14 Členění dokumentu „Specifikace požadavků“

Specifikace požadavků vychází z dokumentu „Stanovení cílů“. Vzhledem k výše zmíněné potřebě dekompozice specifikací požadavků je výhodné, aby dekompozice specifikací požadavků odpovídala dekompozici systému. Specifikace požadavků mají obvykle následující strukturu:

1. Název projektu a identifikátor projektu.
2. Úvod – shrnutí úkolů systému v obecně srozumitelné formě. Shrnutí má být krátké.
3. Vymezení uživatelů (kdo, kdy, jak bude systém používat) a způsobu využití produktu.
4. Perspektivy realizovaného systému (doba života, možnosti předání dalším uživatelům).
5. Způsoby vedení dokumentace.
6. Zajištění spolupráce mezi dodavatelem a uživatelem.
7. Dokumenty odkazované v textu.
8. Použité zkratky a anotovaný slovník všech termínů používaných v textu, u nichž je nebezpečí, že nebudou správně chápány.
9. Vazby na jiné projekty.
10. Požadavky na hardware, efektivnost a spolehlivost (doba mezi výpadky, ochrana dat, metody detekce chyb atd.).
11. Rozpis dat a funkcí.
12. Plán testů (specifikace testů, testových procedur a testových případů).
13. Vymezení obsahu dokumentace předávané uživateli.
14. Termíny realizace, plán realizace (je-li potřeba).
15. Ekonomické a organizační zajištění (odhad nákladů, kdo jsou řešitelé atd.).
16. Vymezení způsobu údržby a způsobu prodeje produktu dalším uživatelům.

Rozsah jednotlivých bodů závisí na konkrétní situaci. Jádro dokumentu je v kapitole Specifikace funkcí. Funkce systému mají být specifikovány z hlediska uživatele a nemá být při tom brán ohled na detaily realizace. Obvykle používané metody specifikace funkcí:

- popis algoritmů,
- zadání formou příkladů vstupů a výstupů,
- neprocedurální popis (popis účelu, resp. cíle funkce).

Pro větší projekty je vhodné body 11., 12., 13., 14. rozepsat pro jednotlivé subsystémy. Struktura dokumentu počínaje bodem 11 má pak následující tvar:

11.a. Dekompozice systému do subsystémů.
- 11.1. Popis dat a funkcí subsystému 1.
- 12.1. Plán testů subsystému 1.
- 13.1. Vymezení dokumentace části 1 předávané uživateli.
- 14.1. Termíny realizace subsystému 1.
- 11.2. Další subsystémy.

11.b. Metody integrace subsystémů.

5.15 Role zákazníka při specifikaci požadavků

Specifikace požadavků je základním dokumentem při vývoji i při customizaci IS. Zdálo by se, že vypracování dokumentu „Specifikace požadavků“ by mělo být dílem budoucího uživatele. Podle zkušeností však specifikace požadavků pouze zákazníkem značně zvyšuje pracnost realizovaného systému a zvětšuje pravděpodobnost neúspěchu. O důvodech, proč tomu tak je, jsme se zmiňovali na více místech. Shrňme je nyní:

a) Uživatel nebývá schopen omezit požadavky na rozumné minimum. Nevylučují se požadavky méně potřebné až neužitečné (syndrom dortu pejska a kočičky).
b) Uživatelé jsou si zřídka plně vědomi vlastností, možností a omezení moderních informačních technologií.
c) Uživatelé nemají dostatek zkušeností pro vypracování uceleného systému požadavků. Nezřídka jim chybí komplexní znalosti o fungování vlastního podniku na mikroúrovni. Nemají cit pro to, co lze použít, co lze snadno realizovat a kdy je realizace ohrožena.
d) Je větší nebezpečí, že se do požadavků prosadí zájmy těch „co jsou právě u toho“ a budou opomenuti ostatní uživatelé systému.

   Dokument „Specifikace požadavků“ by měl proto být vypracován ve spolupráci pracovníků dodavatele s pracovníky uživatele a oponován budoucími uživateli systému. Je tedy žádoucí, aby mezi členy týmu vyvíjejícího dokument „Specifikace požadavků“ byli pracovníci zákazníka. Je riskantní, specifikuje-li podrobné požadavky sám zákazník bez spolupráce s dodavatelem IS. Role zákazníka při specifikaci požadavků v důsledku shromažďování zkušeností s provozem informačních systémů a možnostem, které nabízejí nové informační technologie, postupně vzrůstá.

5.16 Zajištění vazeb mezi specifikacemi a ostatními etapami realizace softwaru

Vedoucí projektu by se měl účastnit všech etap prací na projektu. Míra účasti může být různá podle toho, které práce se svěřují „pomocným“ pracovníkům, zda jsou to spíše podpůrné práce, jako je testování, dokumentace a kontrola dodržování specifikací, nebo relativně samostatný vývoj nebo customizace dosti rozsáhlých celků s jasně definovanými vazbami na ostatní subsystémy, což je postup nutný u velkých systémů realizovaných desítkami lidí. Vedoucí projektu:

- řídí práce od formulace specifikací do předání hotového produktu,
- účastní se i návrhu a programování klíčových částí projektu (tvar dat, toky dat, přístup k datům, pravidla pro funkce rozhraní, schvalování změn).

Tato doporučení jsou v souladu se zkušenostmi předních týmů, nejsou však běžná v jiných inženýrských oborech, kde funkce projektanta prakticky končí předáním projektu. Realizace je pak dílem jiných pracovníků, i když zpětná vazba existuje – viz např. postup při výrobě prototypu. Způsob, kdy se projekt, návrh (analýza) a programování svěřují různým skupinám pracovníků, je do značné míry obvyklý v těch oblastech programování, kde převažují rutinní práce. Pro účely zbytku tohoto paragrafu budeme pracovníky odpovědné za specifikaci požadavků a návrh systému nazývat analytiky, pracovníky odpovědné za ostatní etapy vývoje softwaru programátory.

Jak organizovat účast analytika v pozdních fázích řešení softwaru a jak naopak účast programátora na analýze? Odpověď není jednoznačná, protože je nutné uvážit i takové faktory, jako je odpovědnost či snaha mít alibi, když věc nefunguje, personální a organizační možnosti atd. Účast programátorů na projekci a analytiků na realizaci je pro projekt výhodná. Nedodržení tohoto pravidla může přinést nárůst nákladů až o 200 % (kap. 15).

5.16 Zajištění vazeb mezi specifikacemi a ostatními etapami realizace softwaru

U customizovaných IS se pracnost etapy kódování a testování postupně redukuje a převažují analytické práce. Účast analytika při oživování systému je pak samozřejmostí. V každém softwarovém projektu není v okamžiku realizace záruka, že realizované funkce skutečně odpovídají záměrům zadavatele. Pro zmenšení pravděpodobnosti vzniku takové situace je třeba kromě jiného vytvořit podmínky, kdy jsou všichni zainteresováni na výsledku práce celého týmu a také jsou za svou práci plně odpovědni a nesou důsledky svých chyb. Jen tak se vytvoří situace, kdy se všichni nesnaží za každou cenu prosadit svoji vůli nebo prosadit svá řešení v sobecké snaze o ulehčení práce nebo z důvodů prestižních. Oba případy jsou časté a mohou ohrozit postup prací. Takový přístup k práci, při kterém jsou členové týmu schopni ustoupit, případně přijmout cizí řešení nebo provádět méně populární práce v zájmu celku, nazveme neegoistické jednání.

Vytvoření správných poměrů v týmu je velmi významné pro hladký přechod mezi jednotlivými etapami životního cyklu. Každá etapa bývá ukončena vnitřní oponenturou. Vnitřní oponentura se většinou týká dokumentů právě ukončené etapy.

Oponentury materiálů mají různé formy a názvy (inspection, walkthroughs, viz kap. 8) a bývají doplňovány kontrolními dny, jichž se účastní vedení firmy, a audity prováděnými nezávislými organizacemi. Vnitřní oponentury za podmínky, že účastníci postupují neegoisticky a že jim nevadí pročítání jejich programů, jsou velmi účinným nástrojem. Zvláště pečlivě je třeba provádět oponenturu specifikace požadavků. Obecně je při realizaci projektu třeba mít na paměti následující skutečnosti:

a) Realizace se uskutečňuje jako řada etap. Při zjištění problému v etapě n se musíme vracet k některé z předchozích etap.
b) Správnost etapy n se prověřuje vzhledem k etapě $n-1$ a $n+1$.
c) Každé etapě odpovídá vlastní dokumentace.
d) Někdy je nutné nechat některé relativně samostatné části projektu otevřené.
e) Při testování se musíme často řídit pouze intuicí, poněvadž relativně nejlépe je zpracován problém psaní programů, méně je známo, jak programy testovat, ještě méně je teoreticky zpracováno testování vnějších (uživatelských) funkcí a nejméně testování celku. Výsledky matematické teorie složitosti naznačují, že testování bude vždy do jisté míry i tvůrčí problém, který nebude nikdy možné plně automatizovat. Intuice bude tedy při koncipování testů asi vždy nezbytná.

Základním předpokladem úspěchu je jasná a konzistentní formulace požadavků. Poněvadž jsme schopni sledovat poměrně málo faktů současně, musí být požadavky předepisovány přesně, hutně a musí být vhodně strukturovány.

Mnohdy můžeme projektu porozumět jen tehdy, známe-li důvody přijetí rozhodnutí a požadavků a důvody odmítnutí jiných řešení a požadavků. Jsme tedy schopni vystopovat důvody požadavků. Proto hovoříme o vystopovatelnosti (traceability) požadavků.

Na vyšších úrovních návrhu je žádoucí, aby nebylo nutné se zabývat detaily organizace nižších úrovní návrhu. Tím dospíváme k dalším zásadě. Řešení technických podrobností lze na vyšších úrovních odložit a je možné je navíc řešit různými způsoby. Důsledkem je modularita, nezávislost a modifikovatelnost řešení. Změny mají být lokální a systém členěn na relativně malé subsystémy.

Specifikace požadavků musí vycházet z dobře formulovaných cílů a s možnou výjimkou inkrementálního vývoje, kap. 7, mají mít následující vlastnosti:

a) Mají být úplné ve všech důležitých aspektech, tj. v definici funkcí, požadavcích na efektivnost a ve stanovení vlastností rozhraní na uživatele a spolupracující systémy.

b) Měly by být testovatelné. Není vhodné formulovat požadavky, které nelze testovat. Příkladem je požadavek, aby program neobsahoval nedosažitelný kód, tj. část, která nemůže být nikdy provedena. Takový požadavek nelze v plné obecnosti ověřit. Jiným příkladem je požadavek, aby doba odpovědí byla obvykle nižší než 10 sekund. V tomto případě je nutné slovo „obvykle“ kvantifikovat, např. stanovením, že v 5 % případů může být doba odezvy mezi 10 až 20 sekundami.
c) Musí být bezrozporné, nesmí obsahovat požadavky, které jsou ve sporu. Příkladem je situace, kdy jeden požadavek stanoví, že události A a B se vylučují, a jiný stanovuje, že A a B probíhají souběžně.
d) Musí být modifikovatelné a srozumitelné. To vyžaduje, aby byl každý požadavek formulován právě jednou a vyskytoval se na jediném místě. Výhodné jsou různé rejstříky a tabulky vzájemných odkazů.
e) Musí být vystopovatelné, tj. u každého požadavku je možné zjistit důvody, proč byl formulován, a také jaké důsledky z daného požadavku vyplývají. U algoritmů realizujících některá zákonná opatření je důležité uvést přesný odkaz na paragraf, podle kterého je daný algoritmus realizován.
f) Specifikace požadavků by měla být použitelná i během provozu systému a při údržbě. Pro údržbu jsou důležité především všeobecné, nikoliv nutně formálně úplné popisy. Podrobné specifikace mohou být totiž z části nahrazeny tím, že se ověří, jak systém pracuje.

6

Techniky zjišťování požadavků

Specifikace požadavků v rozhodující míře definuje odpověď na otázku, *co* má být realizováno. Souběžně by měla být zpřesňována odpověď na otázku *proč*, konkrétněji proč má systém zajišťovat jednotlivé funkce. Zčásti se řeší i otázka realizovatelnosti, tj. otázka, *jak* systém realizovat. Technické otázky by se však měly řešit převážně v dalších etapách.

Etapy specifikace cílů a specifikace požadavků mají zásadní vliv pro úspěch budovaného systému. Chyby v těchto etapách mají vážné následky. Průzkumy ukazují, že vážné prohřešky v těchto etapách jsou spíše pravidlem než výjimkou. (srv. kap. 1.1 a kap. 15). Odstranění chyb ve specifikaci požadavků, na které se přijde pozdě, např. při testování nebo dokonce během údržby, je velmi drahé a může znamenat neúspěch projektu.

6.1 Techniky zjišťování požadavků na IS

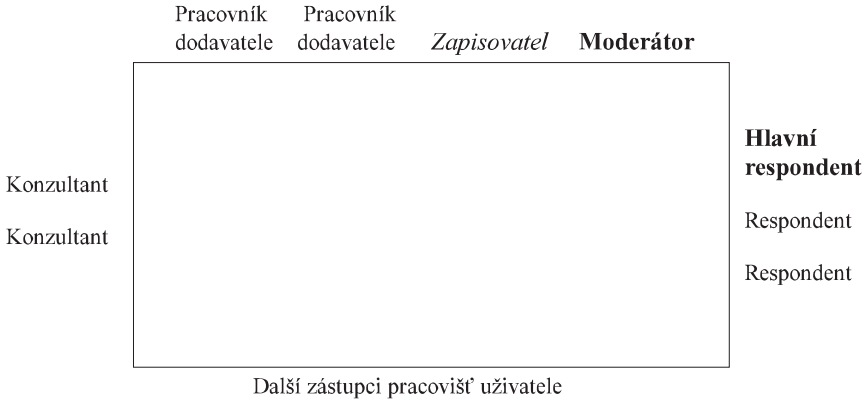
Existuje řada postupů zjišťování požadavků u pracovníků budoucího uživatele. Použitím moderních postupů zjišťování požadavků lze značně zmenšit riziko, ale nelze se úplně vyhnout tomu, že požadavky budou neúplné nebo nesprávné. Pro zpřesňování požadavků lze použít softwarové prototypy. Typickým příkladem softwarového prototypu je simulace dialogu uživatele s budoucím dosud nefungujícím systémem. Jinou cestou je postupné budování systému známé jako inkrementální nebo iterovaná realizace (kap. 7). Specifikace požadavků na IS vždy, i v případě použití CIS začíná zjišťováním požadavků u zákazníka. Používají se při tom následující metody:

a) Interview: Dobře připravený pohovor o tom, co pracovník uživatele (respondent) dělá, co potřebuje a co by mohl IS zlepšit či přinést.
b) Strukturované interview: Interview, při kterém se postupně odpovídá na otázky podle předem připraveného dotazníku.
c) Dotazníky: Požadavky se shromažďují pomocí dotazníků, které se rozesílají a které budoucí uživatelé vyplňují sami.
d) Studium dokumentů používaných zákazníkem.
e) Společný vývoj požadavků: Formulace požadavků skupinou pracovníků uživatele a konzultantů dodavatele IS.
f) Pozorování chodu prací u zákazníka.
g) Účast pracovníků dodavatele na pracích u zákazníka.
h) Analýza existujícího IS.

Obvykle se používá kombinace metod ad a), e) a někdy b) případně c). Ostatní metody se používají jako metody doplňkové.

6.2 Interview

Interview je nejčastěji používaná metoda zjišťování požadavků. Techniky interview při zjišťování požadavků mají velmi mnoho společného s interview v žurnalistice. Existuje rozsáhlá literatura týkající se pravidel vedení interview především v žurnalistice. Cenné jsou zvláště knihy (Davis, 1983) a (Steward, 1994).



Obr. 6.1: Zasedací pořádek při skupinovém interview.

Interview je předem připravený rozhovor vedený obvykle jedním dotazujícím obvykle s jediným respondentem. Dotazujícího budeme nazývat moderátorem. Interview může být vedeno ve skupině (viz obr. 6.1). Pak hovoříme o skupinovém interview. Interview při zjišťování požadavků má jisté zvláštnosti. Předpokládá dlouhodobější spolupráci a tím se liší od interview pro noviny. Jsou nutné přesné zápisy, někdy je třeba interview opakovat. Na tuto možnost je vhodné respondenta upozornit. Podstatným rysem interview při zjišťování požadavků je důraz zjišťování souvislostí. Interview nebývá krátkodobou záležitostí a může trvat až 4 hodiny. Pokud je interview delší než 90 minut, je vhodné dělat přestávky. Optimální délka interview je do hodiny a v žádném případě by interview nemělo být včetně přestávek delší než 4 hodiny. Pokud je záležitost komplikovaná, je výhodnější interview rozložit do více dnů. Opakování interview[1] bývá potřeba v těch případech, kdy se v průběhu vývoje projektu zjišťují skutečnosti, které je třeba dodatečně vyjasnit, a také tehdy, kdy se při následném vyhodnocování výsledků interview zjistí nejasnosti.

Interview při specifikaci požadavků má řadu rysů, které jsou typické pro výslech: následné interview, dlouhá doba trvání, hledání souvislostí a rozporů v tom, co se zjistilo, sledování zmínek o lidech, kteří se také účastní prací, potřeba výsledky interview zapisovat, pak z jednotlivých znalostí skládat mozaiku celku. Je však životně důležité, aby interview nikdy nebudilo dojem výslechu. Tento dojem může vzniknout velmi snadno např. tím, že ne dosti opatrně upozorníme na rozpory zjištěné při daném interview nebo na nesoulad s jinými interview nebo jistými

1. Takové interview se nazývá následné.

skutečnostmi. Pocit výslechu může vyvolat i zdánlivá maličkost. Ten, kdo realizuje interview, by neměl sedět tváří v tvář respondentovi, účastníci interview by tedy neměli sedět u protilehlých stran stolu. Pokud je u interview zapisovatel, je lépe, když respondent nesedí přímo proti dotazovanému ani proti zapisovateli. Těmto zásadám se dá obtížně vyhovět, pokud se interview účastní více osob. Při skupinovém interview je možné volit zasedací pořádek podle obr. 6.1. Účastníci interview musí být přesvědčeni o společném zájmu na výsledku projektu (srv. kap. 2.2). Nikdo z respondentů se nesmí cítit ohrožen nebo mít obavy, že nezvládne nové úkoly. Účinnost interview, které je i při dodržení všech zásad přípravy a vedení interview značně pracnou záležitostí, velmi závisí na psychologických faktorech a ty na schopnostech moderátora a na přípravě interview. To je podobné jako v žurnalistice. Nic nepůsobí odpudivěji, než když se někdo ptá zmateně, nepamatuje si či neví skutečnosti, které již byly řečeny nebo jsou zjevné. Pomáhají poznatky o funkcích a pracovní náplni respondenta.

Nepůsobí dobře, když moderátor není dochvilný. Pro respondenta znamená interview obvykle práci navíc. Je proto žádoucí, aby interview nebylo prováděno v době, kdy je respondent zavalen prací, jako např. účetní v době ročních uzávěrek. Zdvořilost, nikoliv však podlézavost, je samozřejmým požadavkem.

Jednat je třeba nezáludně a otevřeně. Vyplatí se zdůraznit, že se nepředpokládá propuštění respondenta a že IS nepřinese nadměrné pracovní zatížení. Nesmí se ale při tom klamat. Vznik nepříjemných situací lze často eliminovat výběrem respondentů. Dotazující by měl být kompetentní a také dojem kompetentnosti budit. Kompetentnost, přímost a znalosti jsou důležitými pomocníky při vedení interview. Dojmem kompetentnosti působí především skutečně kompetentní lidé, u kterých je patrné, že věci rozumí, že si vše pamatují a umějí dát věci do souvislostí. Při interview ve strojírenském podniku byly prolomeny ledy po tom, když dotazující došel k závěru, že při nebezpečí vzniku zmetku půjde kontrolor na pracoviště a že tedy obrobek nebude dopravován ke kontrole na centrální kontrolní pracoviště. Moderátor si totiž z předchozího interview pamatoval, že obrobek může vážit až 500 kg a že se složitě upíná na obráběcí stůl. Pracovník zákazníka odpovědný za projekt si tento fakt neuvědomil, respondent, jímž byl vedoucí provozu, samozřejmě ano. I nejposlednější dělník či úředník může poskytnou důležité informace, které budou ztraceny, pokud s ním nebudeme jednat jako rovný s rovným. Nelze ale takový postoj předstírat, protože to respondenti vycítí. Výrazem slušnosti je i to, že interview je vedeno v jazyce a termínech, kterým respondent rozumí. Je dobře se naučit jeho termíny, někdy ale dělá dobře, když se požádá během interview o vysvětlení. Nesmí to ale být často. Pokud se objeví nějaké nejasnosti a rozpory v tvrzeních, je vhodné navodit situaci tak, že si to i bez upozornění uvědomí respondent sám. To je Sokratova metoda dialogu. Vyžaduje značnou obratnost a také nadání. Přímé upozornění na rozpor bývá ke škodě věci. Někdy samozřejmě nelze jinak.

Vlastnosti a nadání moderátora jsou minimálně tak důležité jako všechna doporučení a zásady vedení interview. Volba moderátora je proto jedním z rozhodujících faktorů úspěchu. Mezi vlastnostmi moderátora musí být především schopnost budit sympatie a navázat kontakt. Je to schopnost, která se nedá úplně nacvičit, stejně jako schopnost jednat s lidmi, nebýt arogantní, pamatovat si řečené atd. Jsou to vlastnosti, které se dají naučit jen do jisté míry. Musí být spojeny s odbornou zdatností.

6.2.1 Průběh a zásady vedení interview

a) Interview je třeba dobře připravit. Je třeba předem shromáždit a vyhodnotit všechny informace, které by mohly mít vztah k respondentovi a jeho úkolům. Je třeba sestavit seznam problémů, o nichž je již před začátkem interview zřejmé, že by interview mohlo přispět k jejich řešení. Při přípravě interview se osvědčuje vycházet ze scénářů ucelených činností, jako je postup vyřizování zakázky, objednávání materiálů atd. Vychází se tedy

z činností a jejich návazností, z procesů[2]. Organizační zařazení pracovníků je z tohoto pohledu podružné. Tento přístup usnadňuje i budoucí organizační restrukturalizaci činností (business process restructuring, BPR). Při plánování interview se vychází z věcné pracovní náplně respondenta a pak se zjišťuje jeho organizační zařazení. Před zahájením interview je nutno mít souhlas pracovníků, jimž respondenti organizačně podléhají. To může být zajištěno obecnou dohodou na začátku prací, ale i tak je vhodné na konání interview vždy vedoucího upozornit.

b) Schůzku je třeba pečlivě naplánovat tak, aby nekolidovala s některými naléhavými povinnostmi respondenta, a zajistit vhodnou místnost.

c) K interview přijít včas. Představit se a uvést všechny informace o řešeném systému, které by mohly respondenta zajímat, např. proč se realizuje, přínosy, možnosti uplatnění respondenta po zavedení IS, je-li to zřejmé, jaký význam pro řešení mají informace od respondenta. Respondent musí být chápán jako partner.

d) Vlastní interview vést jako zdvořilý člověk, který nemaří čas a vede rozhovor tak, jak je obvyklé mezi znalými a slušnými lidmi. Na začátku interview se musí prolomit ledy, navodit přátelská atmosféra. Je proto vhodné nezačínat hned s vlastním interview, ale věnovat pár slov nutné společenské konverzaci. Po nenásilném přechodu na téma interview bývá vhodné požádat respondenta o sdělení, co dělá a co si o své práci myslí. Moderátor by měl zdůraznit potřebu spolupráce k oboustrannému prospěchu. Zde velmi pomáhá ukázat na to, že se s pracovníkem počítá i po zavedení IS. Musí to však odpovídat skutečnosti. Lhát se nevyplácí.
   Při interview je třeba dodržovat následující zásady:
   - pozorně poslouchat a neskákat zbytečně do řeči, ale také neposlouchat mlčky a projevovat zájem nepříliš častými drobnými zpřesňujícími poznámkami;
   - nechat respondentovi čas na rozmyšlenou při odpovědi na otázku, při váhání mu nenápadně pomoci jinou formulací otázky;
   - střídat otázky otevřené, vyžadující odpověď typu vysvětlení, s uzavřenými, na které je odpověď ano/ne. Uzavřené otázky se často kladou po tom, co moderátor shrne vlastními slovy to, co se v předchozích minutách dozvěděl a požádá o potvrzení, zda dobře porozuměl. Za uzavřené otázky se považují i takové otázky, na které lze odpovědět udáním nějakého faktu (příklad: kolikrát za měsíc řešíte reklamace?);
   - respondent může lehce odbočit od tématu, ale nelze připustit tlachání;
   - obvykle bývá vhodné se respondenta zeptat na to, co jej rozčiluje a co si myslí, že by se dalo zlepšit;
   - nepřipustit vznik pocitu, že interview je ztrátou času.

e) Pohled respondenta, zvláště v celkem dobře fungujících organizacích, je vždy neúplný. V dobře fungujících organizacích nezná nikdo detaily, jak přesně organizace jako celek funguje. Každý ví, za kým s čím jít a co udělat při vzniku určité situace, neví však, co dělají ti druzí. Je proto důležité „ty druhé“ znát. Při interview je nutné sledovat, kdo s respondentem spolupracuje. On sám si často nevzpomene ani na všechny takové situace, ani na to, s kým vším věci řeší. Je proto důležité zaznamenávat výroky, kde se vyskytují funkce a osoby spolupracující s respondentem, např. sledovat výroky typu „s tím jdu za skladníkem“.

f) Podstatná fakta ihned zaznamenávat. Je vhodné, aby zápis dělal pomocník moderátora. Lze použít diktafony do kapsy, na stole více ruší. S použitím diktafonu musí respondent souhlasit.

g) Interview by se mělo týkat následujících témat:
   - postavení respondenta a jeho pracovní zařazení;
   - ucelené scénáře jeho činností nebo kroky činností, za které je odpovědný;

2. Odtud procesní pohled a orientace na procesy

- vztahy: spolupráce s jinými pracovníky, souvislost s jinými činnostmi;
- co si myslí respondent o své práci a o tom, jak hodnotí jeho práci nadřízení a spolupracovníci;
- prověřovat existenci zvláštních situací (může existovat objednávka od zákazníka, který není na seznamu zákazníků?);
- opatrně zjišťovat názory respondenta na situaci v podniku. Do kritiky situace se ale zásadně nepouštět;
- při následném interview a někdy i při prvém interview lze s citem pro míru používat grafické prostředky, především diagramy toku dat (viz kap. 12) a někdy i E-R diagramy či organogramy („organizační pavouky"), případně jiná schémata.

h) Na konci interview je nutné výsledky interview shrnout a nechat předběžně schválit respondentem.

i) Konsolidace interview. Bezprostředně po interview je nutné zjištěné poznatky uspořádat a zařadit je do souvislostí. Jinými slovy zasadit další kamínek do mozaiky znalostí a požadavků. Při tom může vzniknout potřeba následného interview.

j) Osvědčuje se, aby byla v okamžiku, kdy je shromážděn ucelenější soubor požadavků, uspořádána schůzka řešitelů s klíčovými pracovníky uživatele (skupinové interview) s cílem provést kvalifikovanou analýzu a koordinaci souboru požadavků. Ve složitějších případech se osvědčuje vytvořit stálý tým pracovníků dodavatele i uživatele a IS vyvíjet ve spolupráci pracovníků obou stran (joint application development, JAD).

6.2.2 Situace ohrožující úspěch interview

Interview může být neúspěšné z řady příčin. Uveďme přehled nejčastějších případů:

a) Existenční ohrožení: pocit ohrožení postavení nebo dokonce obava ze ztráty zaměstnání u respondentů. Proti takovým pocitům je třeba bojovat, především podle zásad uvedených v kap. 2.2.

b) Mírnější variantou existenčního ohrožení může být pocit ztráty vlivu respondenta v podniku. Pocit může být racionální, dojde-li např. k redukci velikosti oddělení, jehož je respondent vedoucím, i zcela bezdůvodný vyvolaný iracionální obavou ze změn. Analýzou systému lze zjistit reálnost obav a případné negativní vlivy eliminovat volbou respondentů. Někdy je dokonce nutné upravit cíle projektu tak, aby nebylo ohroženo postavení pracovníků, bez jejichž podpory nelze IS uvést do provozu. I zde hraje velkou roli psychologie. Je-li odznakem moci přístup k IS, bude o něj bojovat každý dostatečně vlivný člen vedení. Pocit ohrožení může vzniknout i proto, že interview není provedeno se všemi pracovníky na jisté úrovni hierarchie. To může vyvolat u „outsiderů" negativní reakce. Vlivní členové vedení by nikdy neměli nabýt pocit, že jsou obcházeni, natož ohrožováni.

c) Respondent může mít tendenci odběhnout od tématu, např. trápí-li ho poměry v organizaci. Je pak nutno se vrátit k tématu starým známým obratem – proneseme formální frázi vyjadřující mírný zájem a vrátíme se k tématu interview.

d) Respondent a moderátor jsou odborníci různých profesí. Je proto velké nebezpečí nedorozumění. Odborné termíny je třeba definovat předem. Výskyt nových termínů je třeba zachytit a okamžitě vyjasnit jejich obsah.

e) Často se stává, že se respondent staví do pozice „co se ptáte, když jste expert". Takový přístup může být vyvolán existenčními obavami z budoucnosti, nevhodným chováním moderátora, ale také tím, že předtím už někdo s respondentem nevhodně jednal. V tomto směru se „vyznamenaly" některé poradenské firmy. Obranou je naprostá otevřenost, upřímnost a zdůraznění faktu, že úplné znalosti o tom, jak se provádí určitá činnost, má pouze respondent. Nebezpečným tématem jsou poměry v podniku. Moderátor se má vyhýbat vlastním hodnocením a o nedostatcích má nechat hovořit především respondenta. Důvodem nepochopitelných postojů mohou být osobní vztahy a celková nespokojenost s poměry v podniku.

Součástí interview je průběžné zaznamenávání klíčových skutečností, aby se dal průběh interview reprodukovat. U důležitých interview je vhodné mít zapisovatele, může to však rušit. U delších interview je dobré předem stanovit i dobu ukončení. Diktafon by neměl nahrazovat zápis, měl by se použít jen v případě pochybnosti o tom, co se řeklo při interview. Je vhodné si připravit pásky napřed a očíslovat je. Diktafon lze použít, souhlasí-li respondent s jeho používáním.

Moderátor interview nemá být oblečen tak, aby vyvolával averzi. Je-li respondent ředitel nebo úředník, je vhodnější oblečení společenské. Na nižších úrovních řídicí hierarchie je vhodné spíše kvalitní sportovní oblečení. Mladší moderátoři by neměli být oblečeni příliš vyzývavě, zvláště je-li respondent starší věkem a postavením.

Interview je při dodržení určitých předpokladů nejúčinnější metoda zjišťování požadavků pro IS vyvíjený od počátku. Osvědčuje se i pro systémy customizované, i když v tomto případě je častěji používáno strukturované interview.

Výsledky interview závisí na kvalitě moderátora. Vedení interview vyžaduje soubor vzácně se vyskytujících schopností. Dobrých moderátorů je proto nedostatek. Efektivnost interview závisí na vytvoření podmínek na straně odběratele: umožnění kontaktu s koncovými uživateli, podpora od managementu, vytvoření organizačních podmínek. Interview je poměrně pracné. Lze je pružně přizpůsobit okolnostem, může však zjišťovat nepodstatná fakta a opomenout podstatné skutečnosti. Kvalita interview se obtížně kontroluje. Interview je vhodné kombinovat s dalšími metodami uvedenými níže.

6.3 Strukturované interview

Strukturované interview je interview organizované jako vyplňování předem připraveného dotazníku, které se provádí ve spolupráci moderátora a respondenta. Tato technika je obvyklá u customizovaných IS. Vytvořit dobrý dotazník je obtížné. Struktura a obsah dotazníku je významnou částí know-how dodavatele IS. Výhodou dotazníků je standardizace dotazů a metod zaznamenávání odpovědí. Lze tedy dobře porovnávat výsledky interview různých respondentů. Strukturované interview je méně pracné než nestrukturované.

I výsledky strukturovaného interview silně závisí na osobě moderátora, avšak méně než při interview nestrukturovaném. Problémy volby respondentů jsou stejné jako při nestrukturovaném interview. Strukturované interview vyžaduje velmi dobrou přípravu. I pak je nebezpečí, že bude nutno použít i nestrukturované interview a další níže uvedené metody, poněvadž se mohou vyskytnout neočekávané okolnosti. Použití dotazníků zvyšuje nebezpečí, že se skutečnosti, s nimiž dotazník nepočítá, vůbec nezjistí.

Pro přípravu a vedení strukturovaného interview platí podobné zásady jako pro interview nestrukturované.

6.4 Rozesílání dotazníků

Pokud je k dispozici dotazník, může se použít také tak, že se pošle většímu počtu respondentů, kteří ho podle návodu samostatně vyplní. Tento postup má kromě zjevných výhod, jako je rychlý sběr požadavků, řadu nevýhod. Proto se tato metoda používá spíše jako metoda doplňková, vhodná spíše pro „vyhledávací průzkum“.

Poněvadž při vyplňování dotazníků není přítomen moderátor, hrozí nebezpečí, že respondenti nebudou při vyplňování dostatečně pozorní a nebudou věnovat vyplňování dostatek času. Na dotazník nemusí odpovědět všichni, kterým se poslal. Tyto problémy lze zmírnit vhodnými organizačními opatřeními, zcela eliminovat je

však nelze. Samo zpracování sebraných dotazníků může být dosti pracné a málo efektivní – mnoho informací se opakuje, někteří respondenti neodpoví správně.

Metoda může mít při vhodné přípravě i podstatné výhody. Poněvadž jsou odpovědi anonymní, mohou být upřímnější. Velký počet odpovědí někdy umožňuje zmapovat zvyky („kulturu“) různých částí organizace. Z tohoto důvodu se někdy dotazníky rozesílají v počátečních fázích specifikace požadavků. V rozesílaných dotaznících by měly převažovat uzavřené otázky s odpovědí typu ano/ne nebo zaškrtávání alternativ.

6.5 Studium dokumentů

Velmi často se před zahájením interview a jiných technik vyplatí provést podrobnou analýzu dokumentů obíhajících v podniku (co zajišťují, jaké obsahují údaje, rukama koho procházejí). Obvykle se provádí analýza dokumentů i během interview a při závěrečné redakci specifikace požadavků.

Výhodou studia dokumentů je poměrně přesné zjištění potřebných dat. Lze rovněž zjistit mnoho o podnikatelské kultuře zákazníka, jeho pracovním stylu a vyspělosti organizačních principů. Studium dokumentů silně zvyšuje porozumění pro to, co by mělo být cílem řešení, a může uspořit mnoho času. Hlavními nevýhodami jsou nejasné motivace a možná zastaralost dokumentů. Dokumenty se někdy nepoužívají k ničemu rozumnému, vytvářejí se jen ze setrvačnosti a jejich přínos pro fungování podniku je velmi malý. Toto nebezpečí lze snížit tím, že se ověří, kdo dokumenty a data v nich skutečně používá a proč.

Vytváření a oběh dokumentů, jejichž přínos je malý nebo žádný, není výjimečným případem. Na druhé straně dokumenty umožňují detekovat vazby a činnosti, které se jinak obtížně zjišťují. Studium dokumentů může nepříznivě ovlivnit vývoj IS tím, že „propaguje“ řešení, které už může být zastaralé. Studium dokumentů mnohdy umožňuje zjistit historii a odhalit důvody vedoucí k přijetí pozorované organizační struktury.

Výsledky studie každého dokumentu by měly být zaznamenány v následující formě:

- název dokumentu, stručný popis účelu,
- popis užití,
- cesta dokumentu v organizaci, kde vzniká, kde se modifikuje, kde končí,
- data,
- vazby na části realizovaného informačního systému.

Studium dokumentů je poměrně pracné. Všechny dokumenty také nemusí být k dispozici. Studium dokumentů je důležité pro návrh obrazovek, které by měly být podobné dokumentům vázaným na příslušnou činnost, např. výdejka ze skladu, recept v lékárně atd. Při studiu dokumentů je vhodné začínat od dokumentů „externích“, tj. těch, které zajišťují styk s okolím podniku a organizace. Příkladem jsou objednávky, faktury apod.

6.6 Pozorování na místě, účast na pracovním procesu

Pokud je obava z opomenutí důležitých aspektů řešení, lze jako doplňkovou metodu použít pozorování činností pracovníků zákazníka přímo na místě. Provádí se tak, že se díváme pracovníkům při práci „pod prsty“ a zaznamenáváme vše, co se děje: postup, doklady atd. Tato metoda je pracná. Je nutná psychologická příprava, aby se pozorovaní pracovníci chovali normálně a aby pozorování neodmítli. Pozorování obvykle zachytí jen některé činnosti. Činnosti prováděné zřídka, např. roční uzávěrky, nebo nepravidelně, např. reklamace odběratelů zboží, nemusí být zachyceny. Významnou výhodou je to, že lze zachytit scénáře činností a zjistit procedury

a kritéria rozhodování. Pozorování není ovlivněno názory respondentů ani nedostatky popisu „z druhé ruky“. Přináší přesnější pochopení cílů IS.

Ostřejší variantou pozorování je účast pracovníka dodavatele IS přímo v pracovním procesu. Jde o velmi pracnou metodu, která se používá v těch výjimečných případech, kdy se nedaří provést dostatečnou analýzu některých životně důležitých procesů budoucího uživatele IS.

6.7 Analýza stávajícího IS

Pokud má nový IS nahradit existující IS nebo nahradit existující aplikace, je to pro specifikaci požadavků příznivá okolnost, která je zaplacena vyšší náročností při konverzi existujícího SW a existujících dat na nový IS. Existující SW může být alespoň z části použit jako prototyp budoucího řešení. Poněvadž uživatel už má zkušenosti s provozem IS je větší naděje, že bude mít rozumné požadavky. Specifikace požadavků může být vztažena k funkcím provozovaného softwaru, samozřejmě za podmínky, že stávající SW není natolik nevyhovující, že jej nelze rozumně při specifikaci požadavků využít. Postupuje se následujícím způsobem:

1. Zformulují se:
   - problémy a požadavky, které není možné pokrýt stávajícím softwarem,
   - požadavky na modifikaci funkcí stávajícího systému,
   - požadavky a problémy s existujícími daty,
   - požadavky na zachování vyhovujících funkcí.
2. Na základě výsledků předchozího bodu se zformuluje přehled požadavků a problémů.
3. Provede se definitivní volba základních požadavků.
4. Vypracuje se specifikace požadavků.

Výhodou analýzy stávajícího SW je využití dříve provedených specifikací a zkušeností z provozu. Nevýhodou bývá nebezpečí převzetí zastaralých metod a problémy s konverzí dat.

6.8 Týmový vývoj specifikací požadavků

Týmový vývoj specifikací požadavků, kdy členy týmu jsou pracovníci softwarové firmy i pracovníci uživatele, se osvědčuje jako prostředek vývoje velmi kvalitních specifikací požadavků. Jde o velice pracnou metodu silně závislou na tom, zda bude zákazník schopen a ochoten uvolňovat své často nepostradatelné pracovníky na nezanedbatelnou dobu. Proto se týmový vývoj specifikací používá jako doplňková metoda v následujících situacích:

a) Zahajovací zasedání při zahájení prací na specifikacích. Zde se vzájemně seznamují ti, co budou na realizaci IS spolupracovat. Zároveň se specifikují důvody přechodu na nový systém. Dodavatel systému prezentuje své dosavadní zkušenosti v oboru.
b) Vyjasňování problémů a odstraňování rozporů v požadavcích na IS. Mezi problémy, které je nutné řešit, jsou vzájemně se vylučující požadavky různých pracovníků, nejasnosti v tom, jak spolu různé požadavky souvisejí atd. Schůzi je nutno dobře připravit, vypracovat seznam dosud zformulovaných požadavků. Pokud se schůzka koná po delší době, je vhodné, aby byla zahájena informací o postupu prací. Schůzi má řídit moderátor. Zápis ze schůze včetně přijatých rozhodnutí a zjištěných problémů je samozřejmostí. Zápis by měli podepsat zástupci obou stran.

c) Vnitřní oponentura: Formalizovaná varianta zasedání provedená na konci nějaké (delší) etapy. Oponuje se relativně uzavřená část projektu. Používají se techniky popsané v kap. 8.

Zobecněním modelu skupinového interview je společný vývoj aplikace (JAD – Joint Application Development), kdy se prací na všech etapách vývoje SW účastní i pracovníci uživatele. JAD se filozofií blíží výše uvedeným úkolům. Předpokládá však vyšší nasazení pracovníků uživatele. Osvědčuje se používání formálnějších metod jako jsou diagramy toků dat (srv. kap. 12), E-R diagramy (tamtéž), seznamy úkolů a jiné metody. JAD je požadováno při customizaci některých moderních IS, kdy dealer vystupuje spíše jako konzultant.

7

Varianty procesů vývoje softwaru

Ani aplikace všech technik specifikace požadavků uvedených v předchozí kapitole nezajišťují dostatečnou kvalitu specifikace požadavků. Klasický vývojový cyklus softwaru je znám jako metoda vodopádu (kap. 2.):

1. Úplně se specifikují požadavky na cílový produkt a provede se oponentura požadavků.
2. V etapách celkový a podrobný návrh, kódování, testování částí, testování integrační, testování funkcí, testování při předání a převzetí se systém oživí a uvede do provozu.

V praxi nebývá postup tak přímočarý. Zjistí-li se chyba v pozdějších etapách, je nutné se vracet k etapám předchozím. Hlavním nedostatkem metody vodopádu je fakt, že uživatel zjistí až v okamžiku předání, co se vlastně realizovalo, a může dojít k nemilým a hlavně drahým překvapením. Snahou je snížit pravděpodobnost takových případů. Za tímto účelem se používají následující obraty:

1. Každá etapa (často i některé její kroky) životního cyklu se podrobí různým formám kontroly (čtení kódu, inspekce, příp. jiné kontroly – viz kap. 8).
2. Specifikace se otestují pomocí prototypových řešení. Softwarový prototyp je částečně funkční model řešení realizující nebo simulující některé vlastnosti projektovaného systému. Tím se případné nedostatky odhalí již v etapě specifikace požadavků, tj. v době, kdy bylo zatím vynaloženo méně než 25 % nákladů.
3. Systém se vyvíjí postupně rozšiřováním jistého jádra o relativně samostatně vyvíjené přírůstky – inkrementy, nebo zpřesňováním a doplňováním funkcí. V prvém případě hovoříme o inkrementálním vývoji, v druhém o iterovaném vývoji.

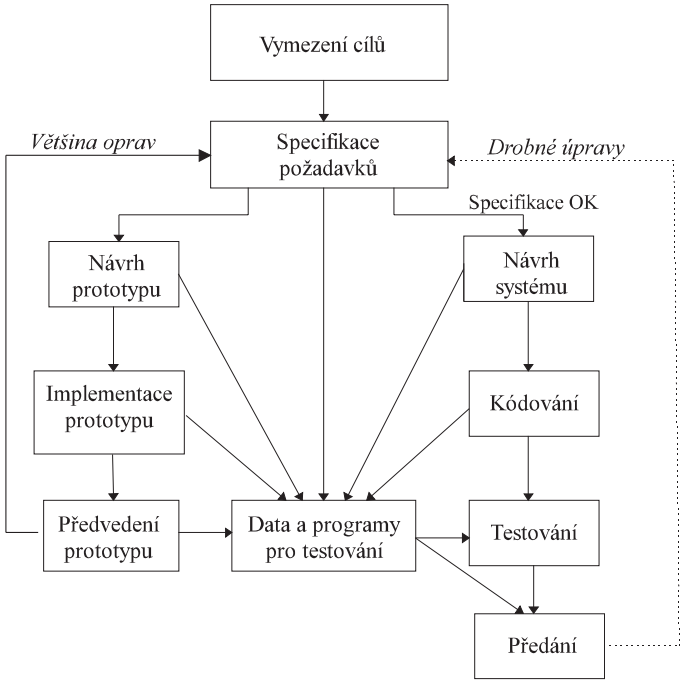
7.1 Softwarové prototypy a jejich použití

Softwarový prototyp se jako částečně funkční model cílového řešení vytváří z následujících důvodů:

- ověření správnosti a úplnosti specifikace požadavků,
- ověření úplnosti a správnosti funkcí a návrhu struktury systému,
- předběžný odhad nákladů a rizik realizace.

Existují následující základní typy softwarových prototypů:

a) *Potěmkin:* Model cílového systému, který simuluje obrazovky dialogů a tvar tiskových sestav. Vlastní výkonná část systému téměř nebo úplně chybí. Tento typ prototypu vlastně simuluje budoucí rozhraní systému. Návrh prototypů tohoto typu je součástí většiny CASE nástrojů (kap. 19).



Obr. 7.1: Používání softwarových prototypů. Větev nalevo může být provedena vícekrát.

b) *Neúplný:* Modeluje pouze některé funkce.
c) *Jiný kůň:* Systém je téměř úplný, ale funguje na jiném hardwaru nebo nad jiným základním softwarem. Tento případ je častý pro software vyvíjený pro jednočipové počítače.
d) *Hlemýžď:* Prototyp je realizován v jazyce, který neumožňuje dostatečnou cílovou efektivnost. Příkladem je prototyp v jazyce PROLOG.
e) *Nepříjemný:* Uživatelské rozhraní není příjemné, prototyp není dostatečně stabilní.
f) *Lajdák:* Nereaguje správně na chyby v datech a chyby obsluhy.

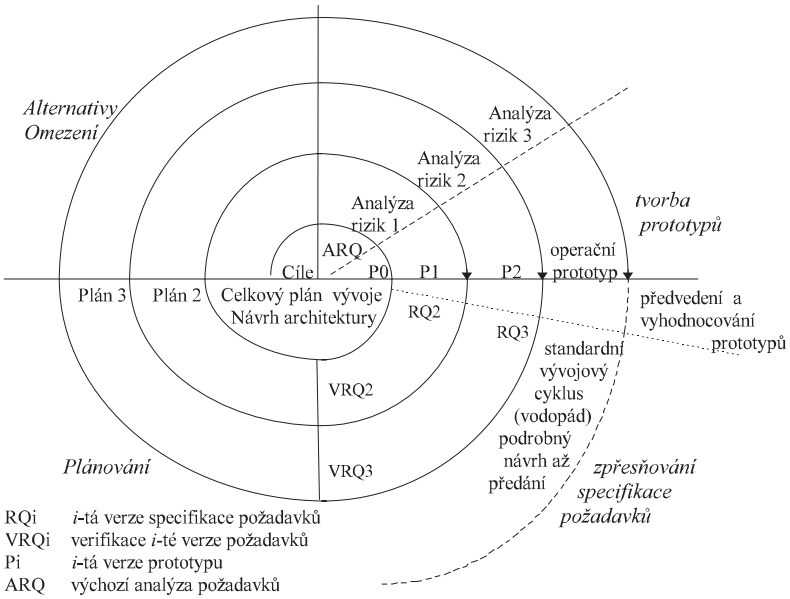
Prototyp je určen k ověření specifikace funkcí a není určen k cílovému řešení. Porušení této zásady nevede k dobrým koncům. Použití prototypu při návrhu softwaru je založeno na myšlence zpřesnit požadavky několikanásobným provedením etap návrh – kódování – předvedení pro prototyp a pak standardním způsobem realizovat cílový stav (obr. 7.1).

Prototypy se používají i při vývoji od začátku a vzácněji i při některých technikách customizace. Při customizaci se jako prototyp využívá částečně oživený systém. Hlavním nedostatkem použití prototypů v IS je fakt, že se jen zřídka podaří otestovat vlastnosti systému, které se projevují až při plném provozu, tj. s plným rozsahem reálných dat a při plné interakci s uživateli, např. při paralelním přístupu k datům. Data pořízená při prototypovém řešení lze často použít i při testování systému. V cílovém systému lze obvykle použít i tvary obrazovek z potěmkinovských prototypů.

7.2 Modely vývoje softwaru

7.2.1 Spirálový model

Pro velké projekty zobecnil Boehm schéma vývoje z předchozího paragrafu do tzv. spirálového schématu tak, aby do schématu byly zahrnuty prvky plánování, postupné zpřesňování požadavků, hodnocení rizik atd. Spirálový model je uveden na obr. 7.2. Návaznost činností se získá procházením spirály ze středu ve směru hodinových ručiček.



Obr. 7.2: Spirálový model vývoje softwaru.

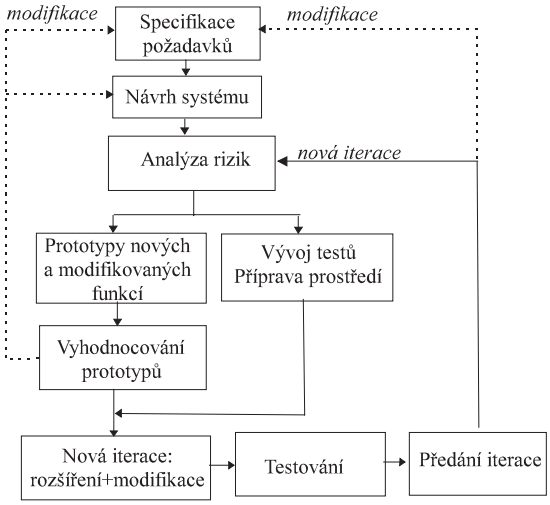
Spirálový model se od modelu z předchozího paragrafu liší v následujících aspektech:

a) Součástí každého cyklu je analýza rizik.
b) Ověřování požadavků se provádí v každé iteraci ve fázích: zpřesnění požadavků, verifikace požadavků, plán akcí, analýza rizik, návrh a realizace prototypu, předvedení a analýza funkcí prototypu.
c) Součástí metodologie je plánování prací a vyhodnocování alternativ řešení před vývojem prototypů.
d) Operační prototyp již zahrnuje vše potřebné pro návrh cílového řešení.

Spirálový model je vhodný pro takové IS (a SW systémy obecně), kde je značná míra nejistoty ve stanovení požadavků, a pro vývoj od počátku. Je vhodnější pro velké systémy, kde není na závadu jeho větší pracnost.

7.2.2 Iterační model

Iterační model se od spirálového modelu liší tím, že výsledkem každého cyklu je stále větší a lépe fungující část cílového systému. Prototypy se realizují pouze u těch požadavků, které jsou buď sporné nebo spojené s nějakými významnými riziky. Cílový systém je pak chápán jako monolitní celek. Schéma iteračního modelu je na obr. 7.3.



Obr. 7.3: Návaznost činností při iterativním vývoji softwaru.

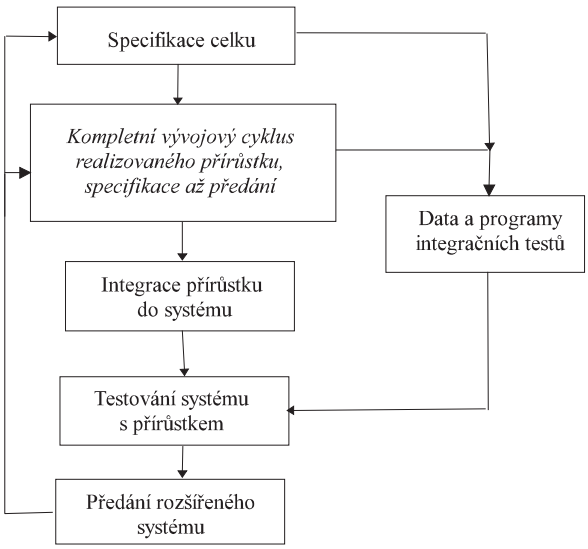
V každé iteraci se modifikuje i dosud vytvořená část systému, která tedy plní do jisté míry i funkci prototypu. Iterační vývoj lze přirovnat k neustále přestavovanému domu, ve kterém se přistavují nová patra, což si vynucuje úpravy již existujících částí domu. Iterativní a také inkrementální (viz následující paragraf) vývoj má následující výhody:

a) umožňuje dříve dospět k softwaru, který lze byť s menším výběrem funkcí použít v reálném provozu,
b) doplňované části jsou testovány v prostředí fungujících programů; to usnadňuje testování a snižuje potřebu prototypů a usnadňuje zavedení systému – nekoná se žádný velký třesk. I pro uživatele je výhodné, že může systém zavádět a tedy se ho učit používat postupně. Pro specifikaci požadavků je výhodné, že zákazník může využít své zkušenosti s fungujícím systémem. Jeho požadavky proto bývají v dalších cyklech rozumnější;
c) lze snadněji realizovat systém postupných plateb;
d) lze vyvíjet i v menším týmu;
e) vede k modifikovatelným a rozšiřitelným systémům;
f) lze snížit pracnost realizace (srv. kap. 15);
g) snadněji se integrují produkty třetích stran a existující aplikace.

Nevýhody:

a) celý systém bývá realizován za delší dobu,
b) u některých systémů je obtížné použít iterační či níže uvedený inkrementální model,
c) optimální velikost týmu bývá menší než u klasických postupů, takže lze jen obtížně zkracovat termíny realizace.

Používání prototypů má výhodu v tom, že se poměrně brzy testují funkce programů, nevýhodou je práce navíc při vývoji prototypů.



Obr. 7.4: Inkrementální vývoj. Obrázek nezachycuje činnosti spojené s vytvářením dokumentace.

7.2.3 Inkrementální vývoj

Inkrementální vývoj (IV) se podobá vývoji iteračnímu. IV je vhodný i pro vývoj velmi rozsáhlých systémů. Při inkrementálním vývoji se systém postupně buduje z jistého jádra, které se rozšiřuje o přírůstky.

Každý přírůstek – inkrement se realizuje kompletním vývojovým cyklem: specifikace požadavků, návrh, který je případně rozdělen do fáze návrhu celkového a podrobného, kódování + příprava testů, testování. Pak následuje integrace přírůstku a předání rozšířeného systému. Na přírůstek se tedy hledí jako na víceméně samostatný systém. Jednou vyvinutý přírůstek se zpravidla nemění. Existují techniky, jak přírůstek jako téměř samostatný a nezřídka i samostatně použitelný systém vyvíjet a pak integrovat (kap. 11). Inkrementální vývoj tedy připomíná výstavbu pavilonové školy. Pavilon (přírůstek) se postaví celkem nezávisle a propojí se s ostatními pavilony vhodnými koridory. Dříve postavené pavilony se téměř nemění.

Inkrementální vývoj není vždy možný, neboť je použitelný pouze za předpokladu, že funkce systému lze dekomponovat do relativně uzavřených celků. V IS však nebývá takový případ neobvyklý, viz např. subsystém účetní, subsystém řízení výroby atd. Při integraci přírůstků do systému je nutné obvykle provádět úpravy přírůstků, např. nahradit interakci s uživatelem výměnou dat mezi přírůstky. Moderní softwarové nástroje tento úkol velmi usnadňují. Moderní operační systémy (UNIX, Windows NT atd.) vytvářejí pro inkrementální model vývoje vhodné prostředí poskytující nástroje výměny dat a spolupráce aplikací. Výhodou je postupující standardizace rozhraní na databáze (jazyk SQL) a standardizační úsilí v oblasti API (Application Programming Interface – propojování aplikací). S těmito prostředky je možno vyvíjet přírůstky jako samostatné aplikace. IS je pak možné sestavovat jako stavebnici takových aplikací.

Inkrementální vývoj založený na spolupráci aplikací má řadu nesporných výhod, ke kterým se podrobněji vrátíme v dalších kapitolách. Inkrementální vývoj:

- je výhodný vývoj ve více týmech;
- usnadňuje integraci existujících aplikací a aplikací třetích stran;
- samotné přírůstky lze realizovat různými metodami a různými prostředky programování a vývoje;
- lze použít v případě řízení procesů (srv. Král, Demner, 1991 a kap. 11);
- IS lze snadno modifikovat, modernizovat a přizpůsobovat měnícím se potřebám zákazníka.

Hlavní problémem spolupráce aplikací je kromě vyšších, dnes však celkem snadno splnitelných nároků na SW prostředí a vyšší režie systému také změna způsobu myšlení vývojového týmu, které se podstatně liší od klasického myšlení orientovaného na realizaci jediného monolitního systému. To do značné míry platí i pro objektově orientovaný vývoj.

Poznamenejme, že jednotlivé přírůstky mohou pracovat na různých počítačích v síti, takže IS může pracovat distribuovaně. Architekturu klient-server lze tedy pokládat za variantu spolupráce aplikací podle výše zmíněné filozofie. Návrh systému klient-server však obvykle vychází z pohledu na systém jako jediný monolitní celek, který se dekomponuje až dodatečně. Takový systém se jen obtížně vyvíjí inkrementálně.

Propojování aplikací neznamená jen prosté spojování funkcí, ale přináší i novou kvalitu (srv. kap. 11 a 21). Spolupráce aplikací je technicky vcelku zvládnutelný problém, jak je vidět na takových produktech, jako jsou kancelářské systémy.

Pro širší uplatnění techniky inkrementálního vývoje je třeba kromě změny filozofie a metod dokončit vývoj standardů spolupráce aplikací (API) a najít spolehlivé nástroje integrace starších programů. Schopnost inkrementálního nasazení je důležitou předností některých customizovaných IS. Dává větší možnost při volbě metod spolupráce mezi výrobcem balíku a dealerem a mezi dealerem a uživatelem softwaru. Výhody inkrementálně customizovaného IS jsou na straně zákazníka obdobné jako u inkrementálně vyvíjeného softwaru. Dekompozice IS do samostatných částí je relativně schůdná, ale vyžaduje mnoho přemýšlení a pečlivou analýzu problému.

8

Vnitřní oponentury a dohled

Etapu specifikace požadavků je vhodné ukončit oponenturou nebo sérií oponentur zaměřenou na následující problémy:

a) Dodržení záměrů cílů projektu.
b) Splnitelnost požadavků.
c) Návrh a hodnocení plánu dalšího postupu.
d) „Správnost“ požadavků, tj. úplnost, jednoznačnost, bezrozpornost a otevřené problémy.

Výstupem je dokument „studie splnitelnosti“ (feasibility study, FS). Součástí FS mohou být výstupy částečných vnitřních oponentur specifikací požadavků. Oponentury FS by se měli účastnit zástupci vedení dodavatele i zákazníka, vedoucí týmů a klíčoví pracovníci obou stran.

Na FS bývají vázány platby podle hospodářských smluv. V případě použití prototypů je součástí FS i dokumentace návrhu prototypů a výsledků jejich testování. FS se vypracovává nejpozději před zahájením kódování, nejdříve však v okamžiku dokončení specifikací požadavků, u inkrementálního vývoje po specifikaci požadavků na přírůstek.

Promyšlené uplatnění akcí dohledu a auditu podstatně snižuje riziko neúspěchu a snižuje i pracnost realizace. Akce dohledu a oponentury jsou poměrně pracné a mezi informatiky dost nepopulární. Kvalitní provedení akcí dohledu a oponentur závisí na řadě okolností, často obtížně dosažitelných.

Dohled je kontrolní činnost prováděná vedením firmy. Má často formu kontrolních dnů. Obsahem dohledu je prověrka skutečností důležitých z manažerského hlediska, jako je dodržování termínů, organizační problémy atd. Audit je dohled prováděný nezávislou organizací. Audit prověřuje dodržování podmínek smlouvy, často včetně prověřování funkcí systému. Osvědčuje se, aby předmětem auditu nebyly pokud možno problémy technického rázu. Audit obvykle provádí „auditor“. Přehled metod auditu softwaru je např. v modulu HS4 systému uceleného informatického rekvalifikačního vzdělávání AMBI (kontakt Ústav informatiky a výpočetní techniky ČAV, Pod vodárenskou věží 6, Praha 9).

Některé formy auditu, např. audit účetních systémů nebo audit kvality podle normy ISO 9000, je oprávněna provádět pouze organizace, která je držitelem akreditace pro daný typ auditu. Pro řešení věcných problémů je velmi výhodné provádět průběžné oponentury, které mohou mít různé formy.

8 Oponentury

8.1 Pravidla provádění vnitřních oponentur

Vnitřní oponentury jsou kontrolní akce prováděné členy řešitelského týmu. Existuje řada variant oponentur lišících se úrovní formalizace a počtem účastníků. Všechny však mají následující společné rysy:

a) oponentur se účastní řešitelé, je možná účast spoluřešitelů ze strany budoucího uživatele;
b) cílem oponentur je detekce chyb, přehlédnutí chyby je selhání oponentury, které je nevýhodné pro všechny;
c) chyby se během oponentur neodstraňují, jen zaznamenávají;
d) detekce chyb nesmí být důvodem postihu jejich původců;
e) oponentura se týká ucelené části projektu a nemá trvat déle než jednu až dvě hodiny. Při delší době trvání klesá pozornost účastníků a účinnost oponentury.

Důvodem pravidla d) je zkušenost, že postihování původců chyb silně snižuje účinnost oponentur, účastníci se bojí „shodit“ kamarády. Porušení pravidla c) vede ke ztrátám času a snižování kvality oponentury.

Nejčastější formy vnitřních oponentur jsou:

Inspekce: Oponentury prováděné podle přísných pravidel v jedné nebo více fázích ve skupině.

Strukturované procházení, čtení kódu, revize: Strukturované procházení a revize (anglicky review) se provádějí při oponenturách větších celků, při přípravě inspekcí a analýze výsledků inspekcí. Pravidla provádění jsou méně striktní než u inspekcí.

Simulace: ruční nebo částečně automatizované procházení části programu se simulací výpočtu.

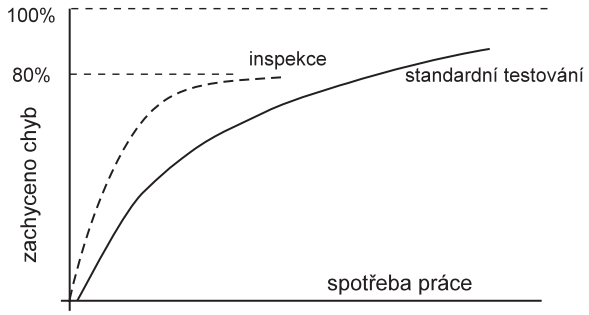
Oponentury jsou jedinou upotřebitelnou technikou odstraňování závad ve fázích stanovování cílů až k testování nebo předvedení prototypů. Jedním z předpokladů úspěšnosti vnitřních oponentur je kvalitní dekompozice specifikace požadavků. Důležitá je i psychologická příprava řešitelů, kteří se musí s technikami a cíli oponentur vnitřně ztotožnit.

8.2 Jednofázové inspekce (Fagan, 1979)

Jednofázové inspekce se provádějí podle následujících zásad:

1. Vytvoří se inspekční tým zpravidla o 3–6 členech. Členy skupiny jsou vedoucí – moderátor, 1–3 oponenti, předčitatel a zapisovatel. Předčitatel, jehož úkolem je prezentace materiálu při inspekci, může být autor příslušné části.
2. Jeden z členů inspekčního týmu, někdy i sám autor, pročítá specifikaci části nebo dokument nebo program a ostatní se snaží nalézt v prezentovaném materiálu chyby. Pokud je třeba souběžně kontrolovat více dokumentů, rozdělí se čtení mezi další členy týmu. Materiály mají dostat členové týmu několik dnů předem. O vzniklých problémech se dělá zápis.
3. Jedno zasedání nemá být delší než 1–2 hodiny, jinak se ztrácí pozornost zúčastněných. Zasedání organizuje a řídí moderátor.
4. Práce se nemá zúčastnit administrativní vedoucí. Jeho přítomnost může vyvolat obavy z postihů při odhalení chyb. Výsledky nesmí být podkladem pro hodnocení kvality pracovníků. Data o průběhu a výsledcích inspekce je dobré uložit do vhodné databáze pro další zpracování.
5. Práce by se měli účastnit ti, kteří budou profitovat z kvality inspekce, např. využijí její výsledky v dalších etapách vývoje systému, a ti, kteří vyvíjejí spolupracující subsystémy. Ti mají přímý zájem na kvalitě inspekce.
6. Účelem je problémy detekovat, nikoliv řešit. Výjimkou může být uživatelská dokumentace. Inspekce by měly být prováděny shora dolů (od celku k částem) po jednotlivých úrovních hierarchie dekompozice. Ze zkušeností

s podobnou technikou při ladění programů je známo, že lze takto s poměrně nízkou pracností odhalit až 80 % chyb. Snaha zvýšit tento podíl není zpravidla úspěšná. Je proto vhodné sledovat počet nalezených chyb za jednotku času a při poklesu tohoto ukazatele inspekci už dále neprovádět, viz. obr. 8.1[1]



Obr. 8.1: Rychlost odstraňování chyb při inspekcích a při testování.

Inspekce byly navrženy Faganem (Fagan, 1979) a s velkým úspěchem uplatněny u firmy IBM. Inspekce probíhá v těchto etapách:

1. Plánování, které provádí obvykle moderátor:
   - Připraví se materiály, které mají projít inspekcí. Materiály musí splňovat jistá kritéria, např. musí být uvolněny vedoucím vývojového týmu k inspekci.
   - Vyberou se vhodní členové inspekčního týmu.
   - Stanoví se termín inspekce.
2. Úvodní studium:
   - Účastníci týmu se školí o tom, co bude předmětem inspekce.
   - Jednotlivým účastníkům se stanoví role při inspekci.
3. Příprava:
   - S několikadenním předstihem se rozdají materiály.
   - Účastníci studují materiály, které jsou předmětem inspekce.
4. Vlastní inspekce pod vedením moderátora:
   - Zjišťují se chyby, neprovádí se žádné pokusy o nápravu. Chyby se zachycují v písemné formě obsahující: identifikátor záznamu, čas, identifikátor inspekce, paragraf, údaj o místě výskytu, např. stránka ZZZ, řádek YYY, popis problému.
   - Vypracuje se zápis a data z inspekce se uloží do databáze projektu. Tím končí vlastní oponentura.
5. Přepracování:
   - Autoři opraví materiály.
6. Kontrola:
   - Moderátor inspekčního týmu nebo celý inspekční tým ověří, zda chyby byly napraveny a zda opravy nevyvolaly další chyby. Kontrola je důležitá. Je totiž známo, že přibližně každá šestá oprava je chybná. Kontrola může mít opět povahu (následné) inspekce.

1. Na obrázku není zobrazeno statistické kolísání. Rozhodnutí, že počet detekovaných chyb již významně neroste, je vhodné založit na metodách matematické statistiky (srv. kap. 15).

- Je vhodné, aby následná inspekce byla provedena jiným nebo částečně obměněným inspekčním týmem.

Správně prováděná inspekce vyžaduje maximální úsilí a soustředěnou pozornost zúčastněných, a proto nemá být delší než 2 hodiny. Déle nelze udržet pozornost. Nedoporučuje se, aby se prováděla více než dvě sezení za den.

Role moderátora je zásadní. Moderátor má být speciálně školen a má mít k dané funkci předpoklady. V zájmu objektivity by neměl být členem řešitelského týmu, ale měl by mít zkušenosti s podobnými projekty. Moderátor musí přispět k vytvoření vhodného ovzduší v inspekčním týmu. Další členové týmu plní následující role:

- Autor textu/programu, může i chybět.
- Předčitatel: prezentuje dokumenty, jako kdyby je sám napsal.
- Zapisovatel.
- Oponenti: snaží se nalézt chyby.

Někdy, např. při posuzování struktury systému, může být tým větší. V takovém týmu mohou existovat i další role. Jednotliví oponenti mohou sledovat jen některé vlastnosti oponovaného, např. testovatelnost požadavků. Počet členů však nemá být větší než deset. Ze zkušenosti firmy IBM vyplývá, že ve fázi úvodního studia bývá produktivita inspekce cca 500 řádků za hodinu, při přípravě inspekce 120–150 řádků a při vlastní inspekci se projde kolem sta řádků za hodinu. Za jedno sezení lze tedy oponovat nejvýše 200 řádků. Studované materiály by měly být proto organizovány tak, aby uzavřené logické jednotky nebyly delší než 200 řádků. Jednotlivé inspekce se obvykle plánují podle zásad strukturovaného procházení, obvykle shora dolů (od celku k částem).

Strukturované procházení (viz. níže) a inspekce lze provádět pro specifikace požadavků, návrh systému, kódování a návrh testů. Kvalita inspekce zásadním způsobem závisí na kvalitě, psychologickém nasazení a na dobrých vztazích mezi účastníky inspekce, především však na kvalitě moderátora (viz Comm. of ACM, Vol 36, No. 1, Nov 1993, 51–62). Při nedodržení těchto zásad se inspekce často zvrhne ve formální záležitost právem považovanou za šikanu a ztrátu času.

Inspekce jsou kritizovány z následujících důvodů:

- Není dostatečně účinná kontrola kvality inspekce.
- Pravidla hodnocení jsou poloformální stejně jako pravidla vedení inspekce. To vytváří prostor pro neobjektivnost při hodnocení výsledků.
- „Slabší“ povahy se při inspekcích neprosadí jen proto, že někteří členové týmu jsou agresivnější.
- Není dostatečná podpora počítačem.
- Proces inspekce je zaměřen na hledání chyb, nikoliv na celkové zvýšení kvality ve smyslu ISO 9000 (srv. kap. 20).

Z těchto důvodů byla metoda inspekce dále rozvinuta, jak je uvedeno v následujících odstavcích.

8.3 Aktivní inspekce

Každá činnost, která není kontrolována, má tendenci „zplanět“, být neúčinnou. Kontrola inspekcí uvedených v předchozím paragrafu je možná až při vyhodnocování výsledků testů nebo dokonce až při předání. To je již často pozdě a vždy drahé. Proto byly navrženy techniky aktivní inspekce a metoda „zasetých chyb“ umožňující sledovat kvalitu inspekcí a tím zajišťovat vyšší aktivitu a pracovní nasazení inspektorů.

Při aktivní inspekci jsou spolu s „oponovaným materiálem“ zadávány „kontrolní“ otázky – jak co funguje a proč bylo zvoleno právě dané řešení atd. Tato metoda je zvláště vhodná pro kontrolu programů a návrh datových

struktur, kdy je možné požadovat rekonstrukci funkcí z kódu. Použití je možné i při inspekci specifikace požadavků, formulace otázek je však poměrně složitá.

Metoda zasetých chyb vypadá na první pohled poměrně bizarně. Do oponovaného materiálu se uměle zanesou („zasejí“) chyby. Po provedení inspekce se zjišťuje, kolik bylo nalezeno chyb zasetých a kolik chyb skutečných. Z těchto údajů lze odhadnout nejen účinnost práce týmu při inspekci, ale také množství skutečných, tj. „nezasetých“, chyb, které dosud nebyly nalezeny. Skutečně nechť z je počet nalezených zasetých chyb a Z celkový počet zasetých chyb. Nechť c je počet nalezených skutečných chyb a C jejich celkový dosud neznámý počet. Při dodržení pravidel náhodnosti zasetých chyb můžeme učinit předpoklad, že procento nalezených zasetých chyb je odhadem procenta existujících chyb, tj. $c/C \mathrel{\widehat{=}} z/Z$. Poněvadž známe hodnoty c, z a Z, můžeme provést odhad celkového počtu chyb $C \mathrel{\widehat{=}} c * Z/z$. $\mathrel{\widehat{=}}$ čteme pravá strana je odhadem levé strany. Slabé místo je v tom, že nelze zaručit, že Z/z je dobrým odhadem hodnoty C/c.

8.4 Vícefázové inspekce

Provedení inspekce je náročná činnost. Pro její zefektivnění je žádoucí, aby se inspekční tým mohl soustředit na základní problémy, a nebyl zbytečně rozptylován při práci takovými nedostatky, jako je nedodržování dohod volání podprogramů, mnemotechniky identifikátorů, pravidel typografického návrhu (pretty printing) a jiných standardů. Proto se některé činnosti při inspekcích provádějí separátně. Celá inspekce je pak vícefázový proces, kde pozdější fáze se mohou spolehnout na splnění podmínek kontrolovaných v předchozích etapách. Obvykle se v počátečních fázích inspekce prověřují víceméně formální záležitosti, jako je dodržování standardů. Nakonec se provádí jedna nebo více inspekcí způsobem uvedeným v 8.2 či 8.3. Formální záležitosti může zkontrolovat i jeden pracovník. Zkušenost ukazuje, že se podobně jako při programování při těchto kontrolách osvědčují spíše mladší pracovníci. Vícefázová inspekce může mít následující strukturu:

I. etapa: Inspekce prováděné jednotlivci.

Provádějí se kontroly formálních vlastností. Formální vlastnosti jsou takové, které by bylo možno v principu provést počítačem. Na odpověď, zda studovaný objekt má nebo nemá danou vlastnost, lze jednoznačně odpovědět ano/ne. Příklady témat inspekcí:

- celková struktura dokumentu,
- mnemotechnika zkratek, identifikátorů,
- přítomnost definic/úplnost deklarací,
- index a úplnost odkazů,
- programátorské standardy, jako je povinné deklarování a iniciace proměnných nebo komentování programu,
- typografické rozložení.

II. etapa: Skupinová inspekce.

Osvědčuje se následující varianta skupinové inspekce:

1. Oponenti dostanou předem materiály spolu s kontrolními otázkami jak co funguje, případně s požadavky, aby si připravili náměty ke zlepšení.
2. Oponenti individuálně pročítají dokument/program a řídí se seznamem otázek a pokyny, co mají sledovat. Znají jen oponovaný materiál spolu s nutnými vysvětleními vazeb na okolí v případech, že by jinak byl materiál nesrozumitelný.
3. Ve skupině se výsledky z bodu 2. jednotlivých respondentů porovnají, zjistí se nejasnosti.

4. Pak se provede oponentura podle zásad uvedených v 8.2.
5. Vše se zanese do zápisu.

8.5 Revize

Pod pojmem revize (anglicky review) se skrývá řada technik a obratů, jejichž společným rysem je, že jsou méně formalizovány než inspekce a mohou se použít na větší celky, např. na shrnutí výsledků několika inspekcí. Schéma revize je většinou následující:

a) Určí se moderátor. Ten si vybere oponenty.
b) Každý oponent dostane k analýze určitou část oponovaného materiálu s cílem nalézt problematická místa.
c) Provede se vlastní revize, na níž se
   – stručně specifikuje úkol revize,
   – uvedou a prodiskutují zjištěné problémy a nedostatky,
   – zhodnotí dodržování plánu práce, je-li to žádáno,
   – mohou vypracovat i doporučení organizačního charakteru,
   – zhodnotí kvalita materiálu, lze doporučit i přerušení prací.
d) Výstupem revize je souhrnné hodnocení s přílohami obsahujícími seznam problémů.

Revize může být uspořádána jako vícefázový proces, při kterém se postupně probírají jednotlivé části oponovaného materiálu nebo se shrnují výsledky jednotlivých inspekcí a jiných kontrolních akcí. Výhodou revize oproti inspekcím je větší flexibilita, možnost oponovat rozsáhlejší materiály a menší nároky na kvalitu členů oponentského týmu.

Nevýhodou je menší účinnost a menší možnosti měření kvality provedení. Pro rozsáhlejší materiály je to však jediný použitelný způsob kontroly. Revize je nejpoužívanější varianta oponentury.

8.6 Další techniky používané při oponenturách

V menších firmách se osvědčují takové formy oponentur, které mají spíše charakter přátelské výpomoci při hledání chyb. Oponentura se provádí ve velmi malém kolektivu o dvou až třech členech. Materiál prezentuje obvykle autor. Často se nedělá ani zápis, to ovšem nelze doporučovat. Hlavními technikami tohoto typu oponentur jsou:

a) *Procházení nebo strukturované procházení* (walkthrough). Podle jistých kritérií, např. podle stromu hierarchie dekompozice se ve dvou nebo tříčlenné skupině prochází text nebo program a analyzují se jeho funkce. Tato technika se osvědčuje jako příprava na formalizovanější metody oponování v budoucnu. Zvláště účinná je v případech, kdy autor textu neustále přehlíží chybu, kterou je schopen často sám rozpoznat, snaží-li se vysvětlit funkce daného místa pozornému posluchači. Pozorný a zkušený oponent dovede navíc vycítit problematická místa a tím zesílit zmíněný efekt. Podmínkou úspěchu procházení je dobrý vztah mezi členy „oponentského týmu“ a vysoké pracovní nasazení všech jeho členů. Výhodou je, že se jedná o techniku, kterou skoro každý někdy použil. Nevyvolává tedy pocit šikanování a byrokratického obtěžování jako inspekce.
b) *Simulace textů.* Řada technik charakteristických tím, že se při nich „ručně provádí“ činnosti/akce specifikované v materiálu. U programů se simuluje chování programu, dnes obvykle pomocí počítačů, a to často již před zahájením testů. Pro tento případ se používá též termín „čtení kódu“. Význam této techniky se zavedením interaktivního přístupu k počítačům, moderním prostředkům ladění programů a objektově orientovaných

technik poklesl, ale používá se stále. Simulaci scénářů uvedených ve specifikaci požadavků je i dnes často nutné kontrolovat „ručně“.

c) *Cleanroom* je vysoce formalizovaná metoda zahrnující i formální důkazy správnosti vyvinutá firmou IBM. Využití této metody při vývoji IS není příliš efektivní. Metoda je vhodnější pro systémy, jejichž cíle není třeba formulovat v úzké spolupráci se zadavateli, srv. kap. 1.

d) *Týdenní posezení u kávy.* U menších firem se velice osvědčují pravidelné schůzky, nejlépe při kávě na konci týdne, s neformální diskuzí na téma „jak jdou věci“. U důležitých zjištění se vyplatí pořídit dodatečně zápis. Existují případy, kdy se podobné sezení dělá každý den. To je případ některých vývojových center amerického ministerstva obrany. Pokud ve firmě panují dobré vztahy, mohou být takové neformální rozhovory účinnou metodou zjišťování vznikajících problémů. Navíc jsou tato sezení významná pro udržování dobrých vztahů mezi lidmi.

8.7 Oponentury zdrojových textů programů

Oponentura programů je založena na specifických technikách. Vstupem oponentury programů je specifikace požadavků, návrh systému a výpisy programů, které jsou již bez syntaktických chyb a obsahují křížové odkazy. Pokud jsou k dispozici vhodné softwarové nástroje, jako statické analyzátory programů, je výhodné předem využít jejich služeb a odstranit zjištěné nedostatky. Při oponenturách je nutné postupně projít celý program. Pořadí procházení může být různé. Nejčastěji se postupuje podle některé následující strategie:

1. *Čtení kódu, postup zdola.* Při tomto postupu se postupně zjišťují funkce různých částí programů počínaje podprogramy, ze kterých nejsou volány jiné podprogramy nebo – v případě rekurzivních programů – jsou rekurzivně volány pouze podprogramy dosud nejnižší nekontrolované úrovně. Z funkcí nižší úrovně se rekonstruují funkce vyšších úrovní, až se dospěje k funkcím celého systému. Funkce se tedy rekonstruují z programu. U objektově orientovaných programů se uvedeným způsobem kontrolují třídy a metody.
2. *Oponentura funkcí.* Při oponentuře se vychází z požadovaných funkcí systému a z nich se postupně odvozují funkční požadavky na nižší programové celky.
3. *Oponentura strukturovaným procházením shora.* Pro každou úroveň hierarchie dekompozice systému počínaje nejvyšší se ověřuje, zda daná úroveň programu realizuje ty funkce, které má realizovat za předpokladu, že nižší stupně hierarchie pracují správně.

Při všech třech postupech lze použít principy provádění inspekcí. Různé průzkumy ukazují, že pokud je vytvořeno vhodné mentální klima a správné návyky, je oponentura programů velmi účinným nástrojem, neboť

- odhalí až 80 % chyb,
- je to nejúčinnější metoda nacházení chyb v programech podle následujících kritérií:
- počet chyb nalezených za jednotku času (den),
- počet chyb na jednotku práce,
- počet chyb na jednotku nákladů.

Nejúčinnější a také nejefektivnější postup je ve všech uvedených kritériích čtení kódu realizované metodou inspekce. Pracnost inspekcí se snižuje používáním moderních prostředků vývoje softwaru.

Oponentury je třeba připravit, naplánovat, provést, vyhodnotit a pak sledovat postup odstraňování chyb. Kromě toho je žádoucí v průběhu oponentur informovat vedoucí projektu a sebraná data využít ke sledování trendů a pro zlepšování know-how softwarového projektu. V širším kontextu lze data inspekcí kromě zlepšení oponovaných

materiálů využít též ke zlepšování technik inspekce, řízení projektů a metod vývoje. Přitom lze využívat zpětné vazby z pozdějších fází řešení projektů, např. data o výsledcích testů a data o provozu systému. Je výhodné použít vhodný IS dat o projektu (srv. kap. 15).

8.8 Činnosti pro zajištění kvality

Z hlediska kroků potřebných pro dosažení požadované kvality budovaného IS se provádějí následující činnosti:

1. *Evaluace:* Tato činnost má za cíl celkové zhodnocení rozsáhlejších materiálů, vyhodnocování alternativ a také celkové vyhodnocování hotových produktů. Provádí se technikou revize nebo inspekce. Hlavní typ evaluace při vývoji a customizaci je oponentura požadavků (feasibility study). Hlavním cílem je prověření, zda jsou požadavky úplné, bezrozporné a ve shodě s cíli projektu. Sledují se možnosti nedorozumění a opomenutí důležitých předpokladů nutných pro funkci systémů.
2. *Verifikace:* Prověření zda
   - jsou specifikace v souladu s cíli projektu,
   - je návrh v souladu se specifikací požadavků,
   - je kód (programy) a doprovodné dokumenty v souladu s návrhem a specifikacemi požadavků.

   Obecně se verifikuje, zda výstupy etapy splňují požadavky vstupních dokumentů.
   Technika provedení: Inspekce / revize / procházení / čtení kódu.
   Kromě sledování toho, zda nedochází k nežádoucím odchylkám, se zde též sleduje, zda jsou termíny realizace reálné, zda nedochází ke změnám požadavků, zda jsou zdroje vyčleněné pro realizaci dostatečné a zda je k dispozici dostatečné know-how. Sleduje se rovněž dodržování dohodnutých norem a standardů.
3. *Validace:* Předvedení a praktické ověření správné činnosti.
   Technika provedení: testování.
4. *Audit:* Nezávislé prověření dodržování dohod a stavu plnění úkolů většinou pro potřeby managementu.

8.9 Vlastnosti členů oponentských týmů

V tomto paragrafu ve zkratce uvedeme vlastnosti, které jsou vítány pro jednotlivé role v oponentských týmech.

1. *Moderátor.* Neosvědčuje se autor materiálu – není dostatečně nestranný a nad věcí, může být ovlivněn vlastními omyly při práci. Moderátor musí být schopen navodit ducha spolupráce, motivovat členy k postoji: „Když nám něco unikne bude to škoda všech“. Musí přísně sledovat pravidla diskuze a vyžadovat je. Nesmí dopustit emocionální postoje zvláště typu: „Teď jsem ti ukázal, že jsi . . .“ nebo: „Máte radost z mých chyb“. Moderátora určuje vedoucí projektu obvykle z nějakého seznamu osvědčených moderátorů, někdy dokonce na doporučení autora oponovaného materiálu. Moderátor má být určen včas, není výjimkou, že je jmenován na začátku projektu. Je výhodné, aby moderátor měl neutrální vztah k autorům oponovaného materiálu, byl znalý problematiky a byl silně zainteresován na výsledku. Tyto požadavky jsou do jisté míry neslučitelné, a proto je nutno volit kompromis.
2. *Předčítající.* Může to být i autor, ale je lepší, pokud to není ani on ani moderátor. Snahou je co nejpřesvědčivěji prezentovat materiál. Postoj k materiálům by měl být neutrální. Předčítající je obvykle určován moderátorem.
3. *Zapisovatel.* Puntičkář, schopný vše důležité zachytit v dohodnuté formě. U zjištěných defektů přísně dbá na zachycení všech významných skutečností. Je určen moderátorem.

4. *Oponent.* Snaží se být neosobní, vyhýbat se výrokům jako „tvůj program“. Pečlivě projde materiál předem a snaží se o objektivnost. Propuštění chyby/defektu považuje za neúspěch. Je veden snahou přispět k řešení, nesmí ani náznakem hodnotit kvalitu autora oponovaného materiálu. Je výhodné, aby byl na kvalitě oponovaného materiálu nějak zainteresován, aby ho např. v budoucnu používal.

9

Řízení prací při vývoji softwaru

Řízení softwarových projektů má většinu rysů shodných s řízením projektů v jiných technických oborech. Je proto možné používat metody a nástroje určené pro řízení projektů, jako jsou např. produkty MS Project nebo organizační části Lotus Notes, podpora vedení projektu v informačním systému R/3 firmy SAP atd.

Se softwarem je ta potíž, že se vše rychle mění a nelze se příliš spoléhat na tradici a mnohdy ani na nedávné zkušenosti. Za této situace je nutné využívat intuici a počítat s nespolehlivostí dat, která pro vedení projektu potřebujeme. Řízení prací je i v softwarových projektech věcí zkušeností, rutiny a především specifického nadání. Nebývá dobré, když se administrativními otázkami řízení zabývají odborně nejzdatnější programátoři a analytici. Jednak to nemusí být schopní manažeři a navíc nejde sloužit dvěma pánům – chce-li někdo programovat i řídit, nebude většinou dělat ani jedno dobře. Manažer by ovšem měl být odborně na výši, aby pomáhal a nepřekážel požadováním zbytečných administrativních prací. Odborní vedoucí týmů musí být schopni spolupracovat s manažerem. Důležitá je volba optimální struktury týmů. Problému struktury týmu je věnována kapitola 10. Efektivnost prací zvyšují nejen prostředky řízení projektu, ale také prostředky podpory komunikace a spolupráce uvnitř týmu (groupware, síť, elektronická pošta atd) a prostředky elektronické podpory administrativy, jako je tvorba a správa dokumentů, sledování prací atd.

9.1 Databáze projektu. Infrastruktura projektu

Moderní informační technologie umožňují vytvoření databank projektu a nástrojů řízení projektů. Je vhodné využívat následující nástroje:

a) Databáze dokumentů včetně správy verzí a prostředky řízení a kontroly nápravy problémů.
b) Knihovny podprogramů a objektů včetně správy verzí[1].
c) Data o dohodách a termínech – lze používat vhodné nástroje vedení projektu.
d) Databáze hodnot metrik (kap. 15).
e) Elektronické formy spolupráce uvnitř týmů a mezi týmy (groupware).

Výše uvedené nástroje lze také využít k odhadu průběhu prací, např. pro zjišťování frekvence změn v jednotlivých částech projektu.

1. Správa verzí je součástí tzv. správy konfigurace (configuration management/configuration control).

Je důležité dodržovat pravidla přístupu a odpovědnosti: kdo je majitelem určitého dokumentu, z jakého důvodu, proč a kdo požadoval změny a kdo je schválil, časové razítko změny.

9.2 Plán zajištění kvality

Základním dokumentem sloužícím řízení prací při vývoji softwaru je dokument „Plán zajištění kvality“. Plán zajištění kvality obsahuje tyto hlavní položky:

1. Účel (jakých softwarových objektů se týká).
2. Seznam dokumentů, na něž se plán odvolává.
3. Popis organizace týmu a rozdělení odpovědnosti.
4. Seznam úkolů pro zajištění kvality ve vazbě na etapy životního cyklu, především pravidla provedení kontrol, oponentur a auditů.
5. Seznam dokumentů, které musí být vypracovány: specifikace požadavků, popis návrhu softwaru, plán verifikace a validace, zpráva o provedených testech, uživatelská dokumentace. Nepovinně: plán realizace softwaru, plán řízení konfigurace, manuál norem a procedur, případně další dokumenty.
6. Popis metod, praktik a konvencí, např. normy na kódování.
7. Prováděné inspekce, revize a audity. Sem patří např. inspekce všech etap životního cyklu, ověřování funkcí a různé manažerské přehledy.
8. Řízení konfigurace, tj. metody a prostředky kontroly toho, zda jsou spojovány správné moduly a jejich verze, řízení a kontrola změn.
9. Metody evidence a způsob řešení zjištěných problémů a závad.
10. Použité softwarové prostředky a použité metodologie.
11. Metody kontroly kódu; sem patří i požadavky na tvar knihoven a normy jejich použití.
12. Způsob ochrany médií a záznamy na nich: zálohování, ochrana před neautorizovanými zásahy, uchovávání verzí atd.
13. Pravidla kontroly subdodávek.
14. Pravidla údržby dokumentů nutných pro zajištění kvality.
15. Kontroly prováděné vedením nebo nezávislým kontrolním orgánem, audity.

V bodě 4 se obvykle požadují tyto akce:

1. Inspekce požadavků na software.
2. Inspekce předběžného návrhu, ověření technické proveditelnosti.
3. Inspekce návrhu, ověření, zda návrh odpovídá požadavkům.
4. Oponentura způsobu testování, jeho adekvátnosti a úplnosti metod.
5. Kontrola dodržení funkcí před předáním, ověření, zda funkce již realizovaného softwaru odpovídají specifikacím.
6. Fyzická kontrola úplnosti dodávky.
7. Průběžné kontroly. Obvykle se prověřují:
   - programy proti specifikacím,
   - správnost rozhraní,
   - implementační rozhodnutí – zda zajišťuje správnost funkcí,
   - testy – zda prověřují správnost všech funkcí.

Správa konfigurace / řízení konfigurace (configuration management, configuration control) je soubor opatření a nástrojů, které zajišťují, aby byly při kompletaci softwarového produktu použity správné verze jednotlivých součástí systému a aby byly včas dokončeny. Pro zajišťování těchto úkolů se někdy vypracovává plán řízení konfigurace. Plán řízení konfigurace má podobnou strukturu jako plán zajišťování kvality, s některými odchylkami, které souvisí s algoritmy zjišťování správnosti konfigurace a s pravidly pro provádění změn. Tato pravidla zahrnují konvence pro tvoření jmen a čísel verzí, pravidla práce s médii, zásady provádění změn, doporučení zásad práce a struktury dohlížecího výboru atd.

Plán řízení konfigurace obsahuje termíny realizace celku a jednotlivých etap. Stanovuje:

- odpovědnosti řešitelů a vedení projektu,
- vazby na ostatní dokumenty, především na plán zajištění kvality,
- termíny inspekcí a kontrol vázaných na vytváření konfigurace,
- způsob sledování změn rozhraní – specifikace rozhraní, postup přijetí změn, údržba dokumentů o rozhraní, ověření rozhraní „za běhu“ systému,
- použití organizačních postupů: zařazení realizovaného softwaru do vyššího celku, pravidla pro rozsah testů před zahrnutím části do celku atd.,
- metody správy konfigurace: stavba knihoven, práva přístupu, zásady ochrany, jištění, historie změn, vzpamatování po výpadku atd.,
- použití softwarových nástrojů a technik.

V některých operačních systémech jsou k dispozici prostředky usnadňující řízení konfigurace (viz SCCS v operačním systému UNIX). Mnohé moderní CASE systémy mají rozvinuté prostředky řízení konfigurace. Existují i samostatně nabízené systémy správy konfigurace. Prostředky řízení konfigurace jsou součástí některých CASE nástrojů a také některých vývojových prostředí. Řízení konfigurace odstraňuje jeden z vážných zdrojů problémů při vývoji softwaru.

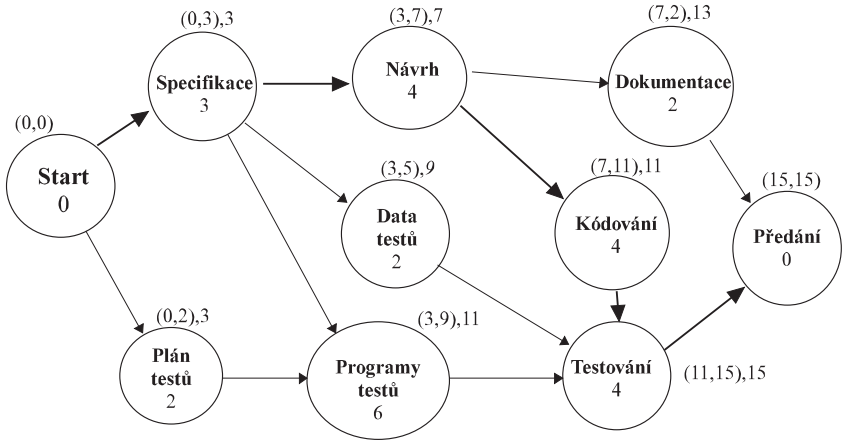
9.3 Síťové metody

Při řízení prací na větších projektech lze vyjádřit návaznost jednotlivých prací síťovým grafem (viz obr. 9.1). Uzly určují jednotlivé činnosti, čísla v uzlech udávají odhad doby řešení, hrany návaznost prací.

Z grafu na obr. 9.1 se určí kritická cesta, tj. posloupnost uzlů, jejichž doby řešení určují dobu řešení projektu. Každému uzlu na grafu se přiřadí dvojice čísel: nejdříve možná doba zahájení prací, nejdříve možná doba ukončení. První údaj se určí jako maximum dob ukončení předchůdců daného uzlu, druhý údaj je takto určená hodnota zvětšená o dobu provedení prací daného uzlu. Start má přiřazeny hodnoty $(0, 0)$, postupuje se od uzlu Start.

Jestliže je uzel P na kritické cestě ohodnocen dvojicí (a, b), je jeho předchůdce na kritické cestě ohodnocen dvojicí (c, a). Z této podmínky lze kritickou cestu zpětně určit, postupujeme-li od koncového k uzlu Start, a také určit nejkratší možnou dobu řešení. Zároveň je možné pro činnosti, které nejsou na kritické cestě, vypočítat nejdříve možné a nejpozději přípustné zahájení dané činnosti. Postupuje se od následníka k předchůdci a nejpozdější termín zahájení se odvodí z minima nejpozdějších termínů zahájení následníků zmenšený o dobu provádění. Popsaná metoda se nazývá metoda CPM, metoda kritické cesty (critical path method). Místo CPM lze použít metodu PERT založenou na úsečkových diagramech.

Programy na hledání kritické cesty jsou součástí většiny systémů podpory řízení projektů.



Obr. 9.1: Příklad použití metody kritické cesty.

Aplikace síťových metod v řízení softwarových prací trpí nepřesností v odhadech dob řešení. Rovněž návaznost prací může být jiná než se předpokládá. Může se například stát, že testovací programy musí vyvíjet i ti pracovníci, kteří vyvíjejí výkonné programy. Pak ovšem nemohou práce na testovacích programech probíhat současně s kódováním. Z tohoto důvodu se síťové metody v řízení softwarových prací používají spíše u velkých projektů a i tam se střídavým úspěchem. U složitějších systémů je vytvoření síťového grafu cenné třeba jen pro prosté stanovení návaznosti prací. Do síťového grafu můžeme zanést i výstupy jednotlivých etap, pokud to není zřejmé z názvů uzlů.

9.4 Deník projektu

Pro řízení prací je nutné rozumět projektu, tj. tomu, co je vlastně obsahem projektu, jakým způsobem je nebo byl systém realizován a znát důvody pro volbu přijatých řešení. Je dále vhodné zaznamenávat důvody volby cílů systému a použitých prostředků, problémy vzniklé při vývoji, každodenní úspěchy, ale také neúspěchy řešení, důvody úspěchu či neúspěchu projektu jako celku. Tyto informace jsou zapisovány do deníku projektu.

Deník projektu má být vytvářen členy řešitelského týmu a je používán během vývoje; je to jeden ze základních dokumentů usnadňujících budoucí údržbu systému. Deník projektu by neměl být rozsáhlý.

Deník projektu je určen k zachycení problémů, nápadů a důvodů realizace a k zachycení dynamiky realizace. Deník projektu může být nahrazen nebo doplněn „občasníky“ (newsletters), tj. nepravidelně vydávanými materiály, obsahujícími důležité aktuální informace a změny (viz Král, Demner, 1991). Deník by měl mít elektronickou formu. Zkušenosti ukazují, že je důležitým pomocníkem při údržbě.

9.5 Personální zajištění

Projekt musí být zajištěn odborně i administrativně. U velkých softwarových systémů je podobně jako u jiných technických děl vytvářena řídící skupina obdobné struktury, jako mají programátorské týmy (kap. 10). Jejím úkolem je především formulace plánů řešení a vytvoření rozumných způsobů jejich kontroly. Osvědčuje se vytvořit dostatečně hustou síť kontrolních bodů s termíny. Průběžnou kontrolu lze realizovat např. pomocí kontrolních dnů a různých forem oponentur. U velkých firem se někdy vytváří samostatná skupina dohledu, jejímž úkolem je studiem dokumentů a diskuzí s pracovníky ověřit, zda práce postupují podle plánu a pokrývají všechny požadavky. Sleduje se především vznik nových požadavků a požadavků na změny již existujících požadavků.

Při zajišťování úkolů je třeba stanovit, kolik lidí je třeba na práce nasadit a kdy mají zahájit práce. Při tom je třeba vycházet z toho, že řada prací má sekvenční charakter a nelze na ně obecně nasadit libovolný počet pracovníků. To se projevuje např. v nemožnosti zkrátit dobu řešení projektu pod jistou mez (kap. 15).

Při vývoji softwarového systému musíme navíc vzít v úvahu, že noví pracovníci musí obvykle zvládnout nové nástroje a metody, např. prostředky na vývoj daného systému, seznámit se s úkoly na projektu a naučit se spolupracovat se stávajícími členy týmu. Jedním z důsledků tohoto stavu je známý Brooksův zákon (Brooks, 1975):

„Zvětšení realizačního týmu v okamžiku, kdy se řešení opožďuje, způsobí další zvětšení skluzu (late team increase makes project late).“

Aby byly věci ještě složitější, odpověď na otázku „kolik lidí“ velice silně závisí na tom, jací pracovníci jsou k dispozici. Věc volby počtu pracovníků je tedy věcí umění řídících pracovníků a jejich zkušeností. Vyplatí se již na počátku najmout poněkud více pracovníků, než se zdá být bezprostředně potřeba. Především je třeba včas reagovat na vznikající potíže.

Při úvahách o počtu pracovníků můžeme využít modely průběhu velikosti týmu a odhady pracnosti, především funkční body. Odhady takto získané pak postupně zpřesňujeme (kap. 15, 16). Z organizačního hlediska lze postupovat dvěma způsoby:

a) Metoda stabilního týmu: Hledat pro existující tým práci.
b) Metoda najímaného týmu: Ke každému úkolu postupně najímat pracovníky podle okamžité potřeby. Kontinuitu prací zajišťuje jádro týmu, tj. vedoucí projektu a jeho nejbližší spolupracovníci, které je vytvořeno na začátku prací.

Metoda b) je možná jen u větších organizací, jinak nelze zajistit, aby byla pro všechny spolupracovníky stále práce, a je nutná při realizaci velkých systémů. Pro metodu b) jsou vhodné jen některé typy organizace týmů a je nutné počítat s menší produktivitou práce než u stabilních týmů (kap. 10).

9.6 Řízení prací a zbytečná administrativa

Výše uvedené metody jsou vhodné a použitelné v situaci, kdy je nějaký systém realizován velkým týmem. Pak je nutné, aby se prakticky všechna rozhodnutí a problémy zapisovaly. Na druhé straně malý systém může realizovat jeden člověk prakticky „na koleně“.

Při aplikaci metod řízení je vždy třeba usilovat o to, aby každá metoda byla používána jen potud, pokud její uplatnění přináší pozitivní efekt. U velkých projektů a velkých firem, jako je IBM, může být vhodné, aby dohled byl svěřen specializovanému týmu. Pro menší firmy a menší projekty je použití takových forem drahé a nemusí přinést očekávaný efekt. Jistou formu dohledu je však vhodné použít i zde. Čím větší úkol, tím větší pozornost musíme věnovat evidenci, plánování a kontrole průběhu prací, neboť u velkých projektů nemůže mít nikdo celý projekt

v hlavě. Rozdělením systému na části dosáhneme úspor především v těchto administrativních záležitostech. Další práce lze u menších realizací uspořit tím, že méně často dochází k nedorozuměním a nejasnostem koncepčního rázu a že práce v malých týmech bývá efektivnější. Plánování a sledování průběhu prací je součástí souboru činností známých jako process engineering (srv. kap. 18).

I u menších úkolů se vyplatí používat některé metody zdánlivě vhodné jen u velkých projektů. Jsou to především metody vedení dokumentace, normy psaní programů, evidence chyb a změn atd. Je s tím práce, ale pokud úkol trvá déle a pracuje na něm více pracovníků, vyplatí se. Někdy se vyplatí vytvořit malý tým špičkových pracovníků, který s prakticky neomezenou dobou řešení hledá koncepčně nové realizace. Takové realizace, jako je např. operační systém UNIX, mohou jen obtížně vzniknout v obrovském týmu s jeho administrativou ovlivňující i vlastní metody řešení. UNIX realizovali dva superprogramátoři před dvaceti lety a tento operační systém bude pravděpodobně používán i v budoucnosti.

Velké skupiny programátorů bývají pracovně přetížené. Není výjimkou, že pod tlakem úkolů nezbývá čas na školení programátorů a na vývoj softwarových nástrojů – musí se dělat jen „produktivní“ práce. To je krátkozraká politika. Význam softwarových nástrojů jsme již vícekrát zdůrazňovali. Snaha o to, aby programátoři dělali jen bezprostředně potřebné programy, se nutně musí negativně projevit na jejich profesionální úrovni a na kvalitě práce. Výsledkem je, že se problémy s nedodržováním termínů neustále zhoršují. Pod tlakem bezprostředních úkolů ztrácejí i programátoři zájem o další růst, změny metod práce a vývoj softwarových nástrojů. Je důležité, aby řízení projektu nedopustilo vznik takové situace. Podíl pracovních kapacit věnovaných zvyšování profesionálních znalostí a vývoj softwarových nástrojů by u každého pracovníka měl za delší časový úsek tvořit 25 % pracovní náplně. I při vývoji softwaru platí „spěchej pomalu“.

Tvůrčí pracovníci mají tendenci podceňovat dokumentaci a administrativní práce. Pokud se je nepodaří přesvědčit o nezbytnosti dokumentace a „úředničiny“ při řízení prací i při vývoji, bude každá administrativa zbytečná, neboť bude prováděna a zajišťována jen proto, že si to vedení přeje – a pak je to opravdu zbytečné.

Je tedy důležité, aby vedení firmy důsledným tlakem přesvědčovalo ve spolupráci s vedením projektu řešitele, nezřídka také pracovníky zákazníka, o nutnosti vedení dokumentace, a provádění kontrolních dnů a oponentur. Je třeba začínat od klíčových problémů, u kterých je vysoká hodnota poměru užitek / práce.

9.7 Vedení projektu a varianty životního cyklu softwaru

Základním problémem, který musí být řešen velmi brzy – bezprostředně po formulaci cílů projektu nebo nejpozději na počátku specifikace požadavků, je rozhodnutí, jaká varianta životního cyklu softwaru a jaká varianta řešení projektu bude zvolena. Při tom je nutné vzít do úvahy následující skutečnosti (kap. 1):

a) Druh softwaru podle míry rizik s jeho fungováním spojených:
   1. Prostý informační systém, ve kterém chyba nevede bezprostředně k ekonomickým ztrátám.
   2. Ekonomický informační systém. Chyby vedou bezprostředně k ekonomickým ztrátám.
   3. Software, na kterém závisejí životy. Příklad: zbraňové systémy, jednotky intenzivní péče, řízení technologií.
b) Velikost softwaru:
   1. Malý projekt (tisíce řádků programů).
   2. Střední projekt (desetitisíce či statisíce řádků programů).
   3. Velký projekt (statisíce až miliony řádků programů).

c) Kvalita řešitelů a zkušenosti s řešením podobných problémů:
   1. Kvalitní řešitelé: výkonní, mají zkušenosti s podobnými řešeními, schopní.
   2. Průměrní řešitelé.
d) Rozsah použití a kvalita použitých softwarových nástrojů:
   1. Kvalitní nástroje, využití moderních technologií
   2. „Klasické“ metody realizace IS: vychází se hlavně ze zkušenosti, žádné nebo téměř žádné vývojové nástroje a žádné prostředky tvorby dokumentace.

Volba varianty životního cyklu a rozsahu „administrativních“ opatření, např. plánování kroků a kontrolovatelných etap, závisí na kombinaci výše uvedených faktorů. Platí zásada, že vyšší číselné hodnocení znamená použití komplikovanějších metod vedení projektu. Metodika SSADM doporučuje čtyři varianty vedení projektu:

A) *Rapid delivery* obvykle při hodnocení a1–a2, b1, c1, d1). Doba řešení nepřekročí několik měsíců a řešení zajistí několik pracovníků (max. 5). Kontrolovatelné etapy: specifikace požadavků, testy, předání.
B) *Express*. Obdoba rapid delivery, varianta s průměrně kvalitním týmem; hodnocení faktorů a2, b1, c1, d1. Doba trvání asi rok. Počet řešitelů 5–10.
D) *Standard*. Vhodné pro méně schopné řešitele a větší projekty. Řešení do dvou let. Maximálně 15 řešitelů.
E) *Inkrementální*. Inkrementální realizace je optimální řešení při hodnocení a3, resp. b3. Pokud projekt s takovým hodnocením nelze realizovat inkrementálně je nutná maximální opatrnost.

Při zahájení projektu je nutno stanovit základní finanční (náklady) a časové parametry řešení. Zahájení projektu probíhá v následujících etapách:

1. Iniciace projektu: společná schůzka řešitelů, ustavení vedení projektu, stanovení termínů etap 2 až 4.
2. Rozbor věcných požadavků s uvážením skutečností uvedených v a) až d). Rámcové stanovení rozsahu prací a předběžná volba varianty vývoje. Zde je vhodné používat diagramy toků dat (kap. 12) a předběžné datové modely a vypracovat kostru řešení.
3. Stanovení plánu a rozpočtu a dále rozpis požadavků na zdroje – náklady na vývoj, vybavení, vývojové nástroje, náklady na provoz hotového systému, vyčíslení přínosů, ohodnocení rizik.
4. Zpřesnění organizačního zajištění a věcných podmínek řešení. Stanovení postupů při řešení problémů: změnové řízení, pravidla provádění činností, zajišťování kvality, správa verzí, pravidla styku, pravidla účasti zákazníka a jeho odpovědnosti, zajištění subdodávek, školení a pravidla informování členů týmu.

Počáteční etapy realizace se pro customizovaný IS příliš neliší od výše uvedeného schématu. Dodavatel customizovatelného IS většinou tvrdě vyžaduje dodržování svých metodik.

10

Práce v týmu

Větší projekty a větší úkoly při customizaci nemůže realizovat jednotlivec ani malá skupinka. Velké úkoly jsou řešitelné pouze ve velkých týmech. Úspěch projektu proto závisí na tom, zda se podaří vytvořit fungující tým. To je úkol vyžadující specifická opatření, znalosti a především schopnosti. Pro dobrou funkci týmu je třeba splnit řadu podmínek. Vlastnosti a schopnosti členů týmu se musí vhodně doplňovat. Mezi jeho členy nesmí být jedinci totálně neschopní týmové práce, např. „schopní všeho“. Práce v týmu musí být vhodně organizována. Význam organizace práce v týmu roste s jeho velikostí. I sama organizace týmu silně závisí na jeho velikosti. Ve větších týmech je nutné věnovat mnohem více kapacit administrativním záležitostem (plánování, kontrolní opatření, schůze, zápisy atd.).

Podíl produktivní práce klesá s rostoucí velikostí týmu. S velikostí týmu roste potřeba „byrokratických“ opatření, každý nemůže komunikovat s každým, vše se zapisuje. Práce ve velkém týmu je tedy méně zajímavá a proto se členové velkého týmů cítí obvykle méně spokojeni než členové menších týmů. Týmy by tedy měly být co nejmenší a administrativa co nejjednodušší. Nejlepší výsledky mají malé týmy do osmi členů. Moderní technologie tvorby software do jisté míry umožňuje realizovat i větší projekty jako soustavu menších projektů a tedy nevytvářet mamutí týmy.

V každém týmu je nutné vytvořit dobré vztahy mezi členy a dobrý postoj členů k týmu jako celku – dobré „klima“. Každý člen by se měl do značné míry ztotožňovat s týmem a jeho cíli. K tomu je nutné, aby v týmu nepřevládli sobci, lenoši a agresivní šplhouni a aby byl každý člen týmu jasně odpovědný za svou práci a byl za ni také správně hodnocen. Dodržování této zdánlivě jednoduché zásady patří k nejobtížnějším úkolům týmové práce (podrobnosti viz John Adair, 1994). Práce v týmu je dynamická záležitost. Tým může trvale dobře pracovat jen tehdy, je-li neustále pečováno o to, aby v něm nedošlo k nárůstu negativních jevů. Kvalita práce týmu silně závisí na lidské psychice. Pro softwarový tým platí zákonitosti platné pro týmy obecně. Začněme od obecných zákonitostí.

U větších skupin není možné, aby spolu po stránce pracovní komunikovali všichni členové. Ve velkých skupinách se nutně ztratí přehled o tom, jaké dohody mezi sebou učinili jednotliví členové týmu. Takových dohod může být až $n(n-1)/2$, kde n je počet členů týmu. Rozsah komunikace mezi členy týmu lze snížit tím, že zvolíme hierarchickou organizaci, ve které se všechny dohody o projektu uskutečňují přes vedení skupiny. Velké skupiny musí být proto přísně organizovány.

Z průzkumů (Leavitt, 1951, Shaw 1964, 1971) vyplývá, že je větší spokojenost s prací a vyšší produktivita u členů menších týmů s neformální organizací. Centralizace je výhodná tam, kde je nutné rychlé rozhodování

(např. v armádě) a u relativně jednoduchých a pracných činností, jako je shromažďování a předávání informací. V této souvislosti je vhodné připomenout výsledky analýzy v (Porter, Lawler, 1965). Tito autoři zjistili, že velikost skupiny je záporně korelována se spokojeností s prací a produktivitou. Ve větších týmech je větší absentérství a pracovníci se více snaží změnit místo. Průzkum se týkal větších organizací, platí však zřejmě i pro softwarové týmy. Ve skupinách, jejichž členy jsou muži i ženy, bývá méně problémů než ve skupinách čistě mužských nebo čistě ženských. V takových skupinách je méně pravděpodobný vznik ostrých osobních antipatií (ponorková nemoc), snáze se řeší problémy komunikace uvnitř skupiny. Většina pracovníků dává přednost práci ve smíšených skupinách.

Optimální je malá neformální skupina se členy obou pohlaví s neautokratickým vedoucím, který má přirozenou autoritu. Bohužel však existují úkoly, na které malý tým nestačí.

10.1 Psychologie týmové práce

Základní podmínkou úspěšné týmové práce je, aby členové týmu byli schopni úspěšně spolupracovat. To je možné jen tehdy, sejdou-li se v týmu vhodní lidé. Z hlediska motivací mohou být pracovníci klasifikováni do následujících skupin (Bass, Duntenman, 1963):

1. Pracovníci orientovaní na úkol, tj. pracovníci motivovaní především samotnou prací (workoholici).
2. Pracovníci orientovaní na spolupráci (kamarádi). Tito pracovníci jsou motivováni přítomností a prací spolupracovníků.
3. Pracovníci orientovaní především na sebe sama (sobci). Pro tyto pracovníky je hlavní motivací vlastní úspěch.

Tým může být úspěšný jen tehdy, je-li v něm dostatek talentovaných členů motivovaných spoluprací. Pracovníci orientovaní pouze na práci (workoholici) mohou být dobří vedoucí. Nesmí však v týmu převládat, neboť mají tendenci k organizační nekázni a neradi se podřizují týmové disciplíně. Pracovníci orientovaní na sebe bývají dobrými vedoucími, jsou-li zároveň motivováni prací. Mívají dostatek vůle ke zvládnutí většího kolektivu, chybí jim takt a těžko se smiřují s podřízeným postavením. Pokud je mezi členy týmu více sobců, lze očekávat problémy vyvolané bojem o moc.

Muži bývají častěji orientováni na práci samu, zatímco ženy jsou motivovány spíše spoluprací. I proto bývají úspěšnější skupiny, ve kterých jsou muži i ženy. V čistě mužských týmech bývá příliš mnoho členů aspirujících na vedoucí roli. Ještě méně se osvědčují čistě ženské týmy, ve kterých často bují intriky.

Studie softwarových týmů ve (Weinberg, 1971) naznačují, že se v týmu často vytváří vedoucí dvojice. Dvojice je tvořena specialistou na problém a specialistou, který fakticky organizuje práci členů týmu a řeší konflikty. Mezi informatiky je mnoho jedinců motivovaných prací. Proto bývá tak mnoho potíží při koordinaci jejich práce.

Ve skupinách tvořených individualisty bývají potíže s organizací spolupráce. Pak je nutné zavést tvrdé normy kontroly prací. Pokud se schopnosti členů týmu vhodně doplňují, nebývá tvrdý administrativní dohled nutný – stačí mírnější metody. Výsledkem bývá lepší spolupráce uvnitř týmu a hladší průběh prací. Spolupráce je možná jen při správném psychologického ovzduší v týmu. Vytvoření správného psychologického klimatu v týmu je jedním z hlavních úkolů vedení týmu.

Spolupráce se snáze organizuje, jestliže má většina členů týmu možnost účastnit se většiny etap životního cyklu tvorby software. Tím se dosáhne toho, aby každý věděl, jaká je funkce jím vyvíjené části v celém systému. Každý pracovník je pak také více zainteresován na úspěchu celku, lépe chápe úkol jím realizované části, nemá tendenci nesprávně „vylepšovat“ vlastnosti jím realizovaného software a někdy může přispět ke zlepšení specifikací

a návrhu jako celku. Rozsah a forma účasti mohou být různé. Tomuto doporučení není samozřejmě možné vyhovět v případě opravdu rozsáhlých monolitních projektů, kdy je nutná práce ve velkých týmech. To je další důvod, proč je realizace velkých systémů tak pracná a proč je výhodné členit projekt na malé části. Ve velkých týmech většinu práce spotřebujeme na kontrolu a administrativu.

Úspěch týmu silně závisí na vedoucím týmu. Vedoucím týmu rozumíme toho člena týmu, který je většinou členů uznáván a jehož rozhodnutí nebo doporučení jsou respektována – je vedoucím de facto. Je výhodné, když je takový vedoucí de facto vedoucím i de jure. V opačném případě mohou vzniknout potíže. Záleží však na konkrétní situaci. Nomenklaturní (administrativní) vedoucí týmu může např. zajišťovat administrativní záležitosti, zatímco faktický vedoucí týmu se věnuje své práci. Existují týmy, v nichž přináší taková dělba práce výborné výsledky. Administrativní vedoucí však musí souhlasit s tím, že v odborných záležitostech hraje podružnou roli. Vedoucí de facto musí naopak chápat potřebu administrativních prací a s administrativním vedoucím spolupracovat.

Vedoucí de facto bývá odborně nejzdatnější člen týmu. Nejsou řídké případy, že se takový vedoucí během různých fází realizace mění. Existence vedoucího de facto předpokládá dosti demokratický způsob vedení týmu. (Levin, 1939) ukázal, že v týmech s demokratickými vztahy mezi členy týmu bývá vyšší produktivita práce a členové týmu jsou s prací více spokojeni. V týmu převládají vztahy spolupráce nad vztahy konkurence. Demokratické vztahy v týmu jsou možné spíše u menších týmů – ještě jeden důvod k tomu, aby byl systém dekomponován a realizován několika menšími týmy.

Při vytváření týmu je výhodné, aby byl buď vedoucí de jure zároveň vedoucím de facto, nebo aby oba vedoucí dobře spolupracovali. Výhodná bývá taková organizace týmu, kdy schopný vedoucí dostane pracovníky, kteří mu umožní svými službami realizaci i poměrně rozsáhlých projektů. Bohužel i toto řešení má některé nedostatky (viz tým šéfprogramátora v 10.7.). V déle existujících týmech vznikají obvykle vztahy úzké spolupráce. Členové týmu považují cíle týmu za své vlastní, snaží se obhajovat tým jako celek a brání se zásahům do záležitosti týmu zvenčí. Tento postoj se označuje jako týmová loajalita. Týmová loajalita zvyšuje výkonnost týmu a zlepšuje vztahy v týmu. Mezi členy týmu však mohou po jisté době narůstat bez zjevného důvodu rozpory, což je jev známý jako ponorková nemoc.

U déle existujících týmů někdy přechází týmová loajalita do nekritické snahy, aby uvnitř týmu nedocházelo k žádným změnám a aby se používaly stále stejné postupy. Uvnitř týmu se neuvažují varianty řešení, neprovádí se důsledná kritika postupů řešení atd. Hlavní motivací týmu již nejsou cíle, které je třeba dosáhnout, ale především obrana zaběhnutých zvyklostí (Janis, 1972), nekritická obrana týmu a jeho rozhodnutí a někdy autokrativní nadvláda vedoucích členů týmů. Vzniká týmový šovinismus. Takovému zkostnatění týmu brání účast vnějších odborníků na řešení úkolů (např. při inspekcích). Oponování přijatých řešení uvnitř i vně týmu může být z tohoto hlediska cenné, pokud se nepociťuje jako šikanování. U projektů rutinnějšího charakteru lze použít metodu najímaného týmu vytvářeného vždy pro každý úkol znovu (viz níže). U zvláště rozsáhlých týmů může docházet k jevům připomínajícím politický boj ve společnosti.

Pro práci v týmu je důležitá komunikace mezi členy týmu – pracovníci se musí domluvit. Organizace spolupráce silně závisí na struktuře týmu, kvalitě zúčastněných a pracovních podmínkách. Poněvadž je počet komunikačních vazeb úměrný druhé mocnině velikosti týmu, jsou v podstatě dvě možnosti:

a) tým je malý a každý se domlouvá s každým přímo,
b) všechny dohody se uskutečňují prostřednictvím centrálního koordinátora.

V případě b) je třeba počítat s nárůstem administrativy a se zmenšením operativnosti při přijímání rozhodnutí. To je hlavní důvod zavádění nejrůznějších kontrolních a jiných opatření, jak jsme se s nimi seznámili v předchozích kapitolách.

Problémem mohou být „zakřiknutí“ členové týmu. Někteří členové týmu často komunikují, jiní se spíše stáhnou do ústraní. Je důležité, aby dominantní členové týmu dokázali podnítit ostatní členy k otevřenosti a aktivitě. Je vhodné zaznamenávat nebo si alespoň pamatovat, kdo kdy promluvil, zda jeho názor byl vzat v úvahu, a analyzovat, jak se členové týmu při diskuzi střídají a zda poslouchají i ti, co nediskutují. V diskuzi by měl vedoucí vystupovat jako moderátor diskuze všech ostatních a nikoliv jako dirigent. Diskutovat mají především členové týmu.

Ve skupinách vznikají různé osobní rozpory. Bývá to zvláště tehdy, je-li mezi členy týmu příliš mnoho pracovníků orientovaných na úkol nebo na svoji kariéru (příliš mnoho vůdců). V takových případech je vhodné tým reorganizovat, obvykle rozdělit.

Klíčovým problémem je vytvoření ovzduší, které podporuje neegoistické jednání. Neegoistické jednání je založeno na následujících zásadách:

a) chyby v programech a v dokumentech se považují za nutné zlo,
b) programy a dokumenty jsou považovány za společné dílo týmu, nikdo nepovažuje výsledky své práce pouze za svoje dítě, které je třeba vždy hájit,
c) při přijímání rozhodnutí je každý ochoten přijmout řešení optimální pro celý tým, i když to může znamenat dočasnou nevýhodu pro něho samého. Je samozřejmostí, že při dodržování této zásady nesmí být nikdo trvale znevýhodňován.

Neegoistické postoje jsou důležitou podmínkou úspěšnosti inspekcí – tedy podmínkou použití nejefektivnějších metod verifikace programů a kontroly kvality dokumentů. Vztahům v týmu prospívá i tzv. technika defenzivní práce, kdy jsou materiály vypracované kolegy před použitím oponovány a data od subsystémů vypracovaných kolegy jsou v programech kontrolována na relevanci. Tvorba softwaru je mentálně velmi namáhavá práce (viz McCue, 1978). Je proto důležité, aby informatici pracovali ve vhodných podmínkách:

1. Pracovníci by měli mít dostatek soukromí – možnost pracovat v klidu bez vyrušování.
2. Informatici by měli mít možnost pracovat při denním světle v prostředí s dobrou ergonomií.
3. Poněvadž je vývoj software dosti individuální práce a informatici bývají vyhraněné osobnosti, vyplatí se dát jim možnost upravit si pracoviště a pracovat tak, jak jim to při plnění zákonných podmínek vyhovuje (klouzavá pracovní doba, práce doma). Důležitý je snadný přístup k výpočetní technice.
4. Je důležité, aby se tým měl kde scházet, aby byla k dispozici zasedací a konzultační místnost.

Psychologie týmové práce je komplikovaný problém a my jsme se zmínili pouze o nedůležitějších faktech. Při práci s lidmi nelze postupovat šablonovitě. Podrobnosti lze nalézt ve (Tracz, 1979 a především v Adair, 1994).

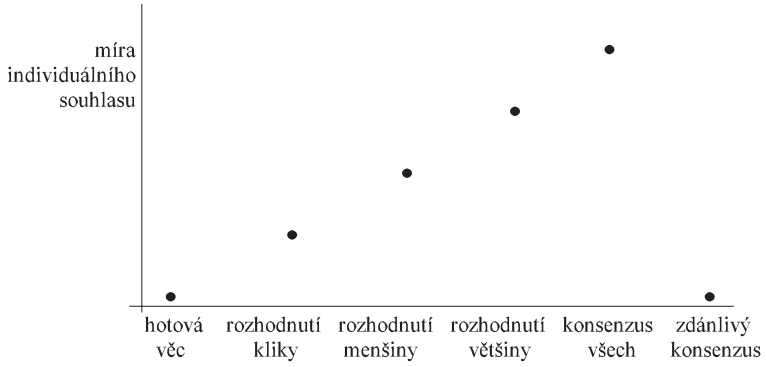
10.2 Pracovní procesy

Tým je skupina lidí spojených pro dosažení určitého cíle, u které je jasně explicitním rozhodnutím a především postojem členů týmu i okolí týmu vymezeno členství. Členové týmu se s týmem identifikují a přijímají společný cíl. Členové týmu se vzájemně potřebují, musí totiž pro dosažení cíle spolupracovat. Z tohoto důvodu přijímají členové týmu určité role; jsou do těchto rolí, nejlépe s vlastním souhlasem, určeni. Tým pracuje jako jednotný organizmus. Provádí koordinované akce k dosažení cíle. Životní cyklus týmu probíhá v následujících etapách:

1. Identifikace (výběr) úkolu.
2. Formování týmu:
   - určení vedoucího týmu, vytvoření jádra týmu,

- analýza hlavních rysů úkolů a cest jeho řešení, odhady pracnosti, rozdělení práce,
- získání dalších členů týmů, vytvoření podtýmů.

3. Krystalizace úkolů:
   - vymezování a zpřesňování jednotlivých dílčích úkolů a cílů,
   - diskuze a spory o způsobech řešení,
   - diskuze a spory o převzetí úkolů a rolí,
   - najímání dalších členů a formování podtýmů podle rozsahu prací.
4. Vyjasnění úkolů:
   - přijetí zásad řešení,
   - předběžný návrh struktury týmu,
   - přijetí cílů členy týmu, souhlas s cíli,
   - volba norem a pravidel práce.
5. Realizace:
   - definitivní stanovení struktury týmu a podtýmů,
   - definitivní přijetí rolí,
   - vlastní provedení úkolu.



Obr. 10.1: Míra skutečného souhlasu v závislosti na způsobu přijetí rozhodnutí.

U dlouhodobě existujících týmů není nutné provádět krok 2. Nevýhodou stálého týmu může být, že jeho struktura a zvyklosti nemusí být vhodné pro nový úkol. Rolí je míněno postavení jedince ve struktuře týmu zahrnující povinnosti, úkoly, pravomoci a odpovědnosti. Nevhodné stanovení rolí je vážnou chybou. Vede ke stresu a nízkému výkonu, projevům nervozity a nespokojenosti. Je důležité, aby pracovník roli přijal a necítil k ní odpor a samozřejmě na ni stačil. Nejčastější aspekty chybně stanovených rolí:

a) Konflikt rolí. Snaha vystupovat ve více rolích s konfliktními očekáváními, např. kamarád a vedoucí. Řešením je na některou roli rezignovat, to je většinou lepší, nebo role střídat, nikdy nehrát obě role současně.
b) Nekompatibilní představy o roli. Různí členové týmů mají o roli různé představy. Zpřesnění definice role je věcí vedoucího. Měla by se volit nejméně konfliktní interpretace z těch, které jsou kompatibilní s úkoly role. Role má vždy být dostatečně přesně vymezena, nesmí být víceznačná.
c) Příliš velký nebo nedostatečný rozsah úkolů pro roli. Řeší se úpravou náplně role.

Předpoklady pro plnění úkolů	Doplňují znalosti a dovednosti člena znalosti a dovednosti jiných členů týmu? Nedochází k duplicitě? Má člen vysokou úroveň profesionálních znalostí tam, kde se to požaduje? Je inteligentní? Je motivován k dosahování vynikajících výsledků a metod vzájemné spolupráce? Svědčí výsledky jeho dosavadní práce o správnosti odpovědí na předchozí otázky?
Předpoklady pro práci v týmu	Bude člen schopen úzce spolupracovat s ostatními při rozhodování a řešení problémů, aniž by docházelo k třenicím a neshodám? Naslouchá tomu, co druzí říkají? Je dostatečně pružný, aby převzal ve skupině různé role? Umí ovlivnit ostatní asertivním, neagresívním způsobem? Přispěje k morálce skupiny, nebo ji bude spíše narušovat?
Psychologické dispozice	Má potřebné odhodlání k dosažení cílů, chápe, že jich nemůže dosáhnout sám bez přispění druhých? Má smysl pro humor a žádoucí stupeň tolerance vůči ostatním? Vyvine se u něho pocit odpovědnosti za úspěch celého týmu, a ne pouze za úspěch jeho podílu na práci týmu? Je integrovanou osobností? Oceňuje realisticky své silné a slabé stránky?

Tab. 10.1: Vlastnosti člena týmu.

d) Role neodpovídá profesnímu zaměření, nebo není přijata členem týmu. V takovém případě je nutné buď pracovníka přesvědčit, aby roli přijal, nebo ji svěřil někomu jinému, nebo roli modifikovat.

Pro efektivnost práce týmu je důležitý způsob přijímání rozhodnutí a na ně se vážící stupeň individuálního souhlasu s rozhodnutím. Existují následující typové situace:

- postavení před hotovou věc: rozhodnutím vedoucího bez diskuze, nebo někdo udělal něco, co určuje způsob řešení;
- rozhodnutí v klice: obdoba předchozího případu, na řešení se však domluví malá skupina;
- dohoda menšiny: obdoba kliky, rozhodnutí však přijímá větší skupina;
- dohoda – konsenzus – většiny;
- všeobecný konsenzus;
- zdánlivý konsenzus.

Nejlepší zkušenosti jsou se všeobecným konsenzem (viz obr. 10.1). Nejhorší případ je zdánlivá dohoda. Velmi špatné zkušenosti jsou také se „stavěním před hotovou věc“, včetně diktátu kliky. Přijetí dohodou je optimální především v případě, kdy úkoly formulují profesně nejzdatnější členové týmu a přesvědčí o řešení ostatní.

Pokud jsou vytvářeny podtýmy, je nutné učinit opatření, aby se nedostaly do antagonistických vztahů. Je nutné přesně stanovit dělbu úkolů. Pomáhají vzájemná podpora, společná zasedání vedoucích atd.

10.3 Vedoucí týmu

Vedoucí týmu je role, na níž především závisí úspěch týmu. Platí to obecně, o to více pro softwarové týmy. Postavení vedoucího v týmu má být založeno na respektu k jeho odbornosti a výkonnosti a na vzájemné důvěře. Pro důvěru je důležité, aby byl psychologicky silná integrální osobnost a „dovedl podržet".

Vedoucím mohou být lidé různých typů. Schopnost vedení se dá do jisté míry při dostatečné úrovni nadání naučit. Vedoucí by měl budit pocit, že je platným členem týmu. Musí být ve vypjatých situacích schopen plně uplatnit i svoji pravomoc uvnitř týmu i navenek, nemělo by však k tomu docházet často. Dobrý vedoucí si k sobě vybírá kvalitní spolupracovníky, jen hlupák si vybírá ještě větší hlupáky, a to především takové, kteří kompenzují jeho slabé stránky. Má být schopen najít svého zástupce a spolupracovat s ním. Za hlavní psychologické rysy osobnosti úspěšného vedoucího se považují:

- Odborná a organizační kompetentnost.
- Vědomí vlastních předností a nedostatků.
- Jistá míra sebedůvěry až tvrdohlavosti, sebedůvěra, ne však arogance.
- Schopnost jasné formulace cílů a cest k jejich dosažení.
- Schopnost najít správné lidi a stanovit jim adekvátní úkoly a přesvědčit je, aby úkoly přijali za své.
- Schopnost přesně formulovat úkoly včetně realistických termínů. Úkoly by měly vyžadovat dostatečnou, nikoliv nadměrnou intenzitu práce, aby nedocházelo k lenošení a práci bylo možné stihnout.
- Dbát na profesní růst členů skupiny i svůj vlastní.
- Využívat nápady podřízených, nebýt přespříliš uzavřený a autoritativní a držet slovo.
- Schopnost přesvědčit a případně motivovat, budit důvěru a vytvářet správné vztahy uvnitř týmu.
- Být příkladem po stránce pracovní i charakterové, držet slovo, umět vynutit výkon a ocenit ho.
- Schopnost vychovávat svoje nástupce.
- Schopnost předvídání a včasného odhalování problémů a rizik.
- Loajalita k podniku a jeho cílům.
- Schopnost správně ohodnotit podněty zvenčí včetně změn v informačních technologiích.
- Být schopen rozumně hájit zájmy týmu a ovládat diplomatické jednání i s nečleny týmu, především s vedením a zákazníky.
- Být schopen tým podržet při neúspěchu. Nepanikařit.

Obr. 10.2: Skupiny úkolů a činností při týmové práci.

Efektivní vedení týmu vyžaduje řadu činností patřících do okruhů zobrazených na obr. 10.2. Operativní činnosti vedoucího týmu zahrnují:

a) Plánování. Za spoluúčasti členů týmu a na základě znalostí cílů a úkolů navrhovat, přidělovat a případně modifikovat úkoly a termíny jejich plnění.
b) Vysvětlování. Seznamování s cíli a hlavně s důvody, proč se to dělá tak a ne jinak. Rozbor úkolů a týmových standardů s uvážením připomínek členů týmu.
c) Kontrolní akce. Oponentury a akce dohledu s cílem kontroly úkolů a následných změn úkolů.
d) Podpora. Průběžné zjišťování vznikajících problémů. Řešení vznikajících sporů. Diskuze aspektů řešení. Kombinování stimulačních (odměny, pochvaly) a disciplinárních (výtky, pozdržení prémií) opatření.
e) Informování. Zajistit včasné informování o všem, co je pro členy týmu důležité nebo co by mohli za důležité považovat. Je nebezpečné, když se důležité informace šíří jako drby. Tento aspekt se často podceňuje.[1]
g) Regulace. Zajišťování a ovlivňování průběhu prací a termínů.
f) Hodnocení. Zajišťovat průběžné hodnocení, případně sebehodnocení kvality a postupu řešení členů týmu. K tomu lze využívat výsledky vnitřních oponentur (inspekce, review), kontroly i neformální schůzky. Výsledky hodnocení vhodným způsobem zveřejňovat a zaznamenávat.
h) Delegování pravomocí. Přenášení části pravomocí vedoucího na jeho zástupce a další členy týmu. Tento aspekt by měl zajistit chod týmu i při nepřítomnosti vedoucího a zároveň vytvořit prostor pro to, aby měl vedoucí také čas přemýšlet a měl „rezervy“ ve výkonu pro případ nenadálých událostí. Delegování pravomocí také zlepšuje klima týmu. Vedoucí má být schopen pozorně naslouchat, navodit partnerský vztah s členy týmu a zároveň budit respekt. Především však musí držet slovo a přijaté dohody. Pokud to v určitém případě není možné, je třeba vysvětlit, proč k tomu došlo.
ch) Udržování a rozvoj pozitivních postojů členů týmu:
   - Důvěra. Pocit, že se lze spolehnout na spolupracovníky.
   - Autonomie. Schopnosti člena jednat samostatně s možností volit pracovní režim a tempo bez zbytečné reglementace.
   - Iniciativa. Možnost uplatnění nápadů a metod členů.
   - Pracovitost.
   - Pocit bezpečí. Člen ví, že nebude neoprávněně postihován ani vystavován kritice a jiným tlakům.
   - Integrita. Člen nemění bezdůvodně své chování, nezklame.
i) Vedoucí musí tým podržet v krizových situacích, je psychicky odolný.

10.4 Neformální role v týmu

Člen týmu by měl splňovat řadu podmínek. Žádoucí vlastnosti jsou patrné z dotazníku v tab. 10.1. Při hodnocení členů týmu je vhodné hodnotit pracovní schopnosti a nadání, ale také zlozvyky.

V psychologii práce se rozeznávají následující pracovní role:

- *Využitelné role.*
  - Iniciátor: Iniciuje nové myšlenky, zásady řešení, změny organizace. Bývá slabší v dotahování věcí do konce.

1. V této souvislosti jsou zajímavé zkušenosti českého ředitele českého zastoupení firmy IBM. K svému překvapení brzy po svém nástupu do funkce zjistil, že komunikativnost směrem k podřízeným, kolegům i nadřízeným je v IBM považována za zcela zásadní předpoklad úspěšné práce.

  - Hledač informací, inovátor: Hledá stále nové informace, snaží se o přesnost, dovede najít rozpory v návrzích. Vhodný jako oponent. Bývá slabší při realizaci úkolů do konce.
  - Hodnotitel názorů (soudce): Snaží se vyjasnit „proč právě tak“, „proč to“.
  - Encyklopedista: Databáze zkušeností, hodnotí problém ve vztahu ke známému a podobnému: „když jsme dělali x, museli jsme . . . “.
  - Cizelér: Vyjasňuje dosud ne zcela přesné představy, „jde do detailů“, vyjasňuje důsledky.
  - Koordinátor: Dává věci do širších souvislostí, objasňuje vztahy, shrnuje znalosti, dovede koordinovat aktivity, které spolu souvisejí, a také dovede takové aktivity nalézt.
  - Navigátor: Schopný hodnotit, zda se řešení neodchyluje od cílů a jde správným směrem.
  - Šťoural: Dovede najít nedostatky, má cit pro rozpory a odchylky od dohodnutých standardů a zvyklostí.
  - Provozář: Zajišťuje provoz, je schopen organizovat a udržovat pořádek.
  - Hecíř: Chválí, oceňuje jiné, projevuje vřelost a solidaritu, „dovede vyhecovat“.
  - Harmonizátor: Dovede uklidňovat napětí, dovede podporovat rozvoj dobrých vztahů. Dovede uklidňovat spory a hledat vhodné kompromisy.
  - Realizátor: Miluje počítač a práci s ním. Rád programuje. Nerad píše dokumentaci a podceňuje počáteční etapy projektu.
  - Moderátor: Dovede zařídit, aby se všichni dostali ke slovu a žádný podnět nezapadl, povzbuzuje k vyjádření. Dovede shrnout výsledky diskuze.
  - Normovač: Prosazuje a podporuje vývoj standardů a získávání kvantitativních údajů o projektu (metrik).
  - Pozorovatel: Zaznamenává všechny aspekty a varianty řešení a používá sebrané informace pro pozdější hodnocení práce.
- *Nežádoucí role:*
  - Agresor: Závidí, destruktivně nesouhlasí, bezohledně útočí ve věcech pracovních i osobních.
  - Negativista: Cílem je zápor za každou cenu, zpochybňuje dohodnuté.
  - Exhibicioista: Předvádí se, chlubí se, prosazuje se.
  - Kecal: Tlachá, zdržuje.
  - Playboy: Svět je pro něho sexuální loviště, nic jiného ho nezajímá.
  - Vládce: Autoritativní, intrikuje, nedrží slovo.
  - Populista: Pasuje se na ochránce členů týmu.
  - Kanaďan: Miluje drsné vtipy.

Při řízení týmu je důležité detekovat potřebu rolí a zjistit, kteří členové týmu mohou hrát rozhodující role. Naopak je třeba eliminovat prostor pro rozvoj záporných rolí. Každá role vyžaduje jiný typ schopností. V týmu mohou někteří členové plnit více neformálních rolí. Vedle neformálních pracovních rolí jsou důležité neformální role přispívající ke stabilitě týmu.

10.5 Podvědomá schémata chování – psychohry

Psychohry v týmu jsou ustálená schémata podvědomého chování. Psychologie zná více než 30 takových schémat. Nebezpečí psychoher je v tom, že je lidé hrají, aniž si to uvědomují a aniž si jsou vědomi možných záporných důsledků.

Nejčastěji se vyskytuje hra „alkoholik“. Hra se podobá jednání alkoholika. V týmu roli alkoholika „hraje“ nevýkonný pracovník, kterého se snaží „napravit“ vedoucí. Ve špatných zvyklostem ho udržuje „utěšovatelka“: „Oni pro tebe nemají pochopení“, a kamarádi, kteří „alkoholika“ přesvědčují, že o nic nejde.

Hra „Ty máš taky máslo na hlavě, tak si nevyskakuj“. Využívají se slabosti partnerů i tehdy, mají-li oprávněné výhrady. Kritizovaný odvrací oprávněnou kritiku poukazováním na dřívější selhání těch, kteří kritizují.

Hra „To je z toho, co jste chtěli“. Odmítání kritiky s využitím toho, že se dotyčný musel podřídit nějakému rozhodnutí, např. nepoužívat příkaz skoku v programech.

Hra „Beru vše“. Člen týmu přijímá stále nové úkoly a pak se vymlouvá na přetížení.

Hra „Kanaďan“. Člen týmu ruší tím, že neomaleně provádí kanadské žertíky.

Hra „Ano, ale“. Odmítání disciplíny pod různými záminkami, např.: „Když nebudu používat příkaz skoku, bude to…“.

Vedoucí týmu musí sledovat, zda členové týmu nejednají podle některého z výše uvedených schémat jednání. Působení psychoher a intrik zvyšuje zatížení členů týmu a ztěžuje dodržování dohodnutých pravidel. Důležité je dosažení konsenzu při opatřeních zabraňujících „provozování“ psychohry.

10.6 Role zvláště schopných osobností v týmu

Je známo (kap. 15), že existují velké rozdíly ve výkonnosti programátorů a analytiků. Poměr 1:20 není výjimkou. Někteří jedinci jsou tedy schopni nahradit celý tým (Weinberg, 1971). Zdá se výhodné využívat zvláště talentované. To však bohužel není bez rizik. Výjimečné nadání se vyskytuje vzácně. Velmi nadané osobnosti bývají nekonformní při jednání s lidmi. Jsou často konfliktní, např. proto, že se domnívají, že ostatní by měli být stejně výkonní jako oni sami. Mají tendenci rychle uplatňovat nové nápady, aniž řádně dokončí své starší projekty. Zvláště nadaní se neradi smiřují s podřízenými rolemi v týmu. Jako vedoucí týmu se někdy osvědčují, ale většina výše zmíněných problémů zůstává.

Jednou z cest, jak využít schopnosti zvláště nadaných, je níže uvedený tým šéfprogramátora, ve kterém se vedoucí může věnovat pouze nejkvalifikovanějším pracem. Nejsnazší cestou využití špičkových pracovníků je nechat je pracovat samostatně a zajistit jim infrastrukturu služeb. Schopný jedinec pak může nahradit i několik desítek průměrných pracovníků. Takové řešení je však z manažerského hlediska dosti riskantní. Případný odchod špičkového pracovníka těsně před dokončením projektu téměř jistě znamená neúspěch projektu a obrovské ztráty ohrožující existenci firmy. Je totiž velmi pravděpodobné, že ne vše bude dostatečně dokumentováno. Hlavní zádrhel je ale v tom, že se pravděpodobně nepodaří rychle získat dostatečný počet pracovníků, kteří by mohli okamžitě pokračovat v práci na projektu. Odchod jednoho pracovníka z fungujícího týmu nemá při správné organizaci týmu tak fatální následky.

Právě provedená úvaha je příkladem manažerského přístupu známého jako princip minimaxu. Pro každé zvolené rozhodnutí (zde využití špičkového pracovníka) se uvažuje maximální možná ztráta (zde odchod pracovníka těsně před dokončením projektu). Volí se takové rozhodnutí, pro které je maximální možná ztráta minimální. Tuto podmínku splňuje v našem případě tým několika průměrných pracovníků místo jednoho vynikajícího. Z podobných důvodů převažuje tendence využívat customizované IS místo IS vyvíjených na míru podle potřeb zákazníka. V druhém případě je větší pravděpodobnost selhání a maximální možná ztráta větší. Je dosti pravděpodobné, že projekt selže a dojde i k ohrožení firmy.

I uplatnění metody minimaxu má svá rizika. Jako jinde v životě platí, že risk je zisk. Skutečně dobrá řešení lze jen zřídka dosáhnout bez podstoupení rizik. Vsadíme-li na průměrné pracovníky, dosáhneme pravděpodobně jen průměrných výsledků. Vsadíme-li na široce dodávaný IS, nezískáme podstatnou výhodu oproti konkurenci, která asi bude používat stejný nebo podobný IS. Z dlouhodobého hlediska není orientace na průměr perspektivní. Je věcí managementu najít vhodný způsob, jak využít špičkové pracovníky. Jít do rizik je však možné jen tehdy, jestliže případný neúspěch neohrozí firmu. Větší firmy mohou geniální pracovníky využívat ve výzkumných projektech. Příkladem výhodnosti takového postupu je operační systém UNIX.

10.7 Organizace softwarových týmů

Při budování týmu musí být zváženy problémy diskutované v předchozích paragrafech. Je výhodné, aby byla struktura a velikost týmu přizpůsobena struktuře realizovaného software a obtížnosti realizace jednotlivých částí systému. To často znamená, že se při rozdělování práce týmů poruší logická čistota návrhu systému. Je dobré, když se struktura systému navrhuje již s ohledem na možnosti jednotlivých týmů a je projednána s členy týmu nebo alespoň s vedoucími týmů.

Týmy v programování bývají poměrně malé (2 až 8 členů). Výhody malých týmů jsme zmínili výše. Uveďme ještě některé další (Sommerville, 1996).

1. V menším týmu je snažší dohoda norem kvality software: jak má být napsán, testován, dokumentován a předán.
2. V menším týmu se při společné práci mohou členové týmu učit jeden od druhého a tak si vzájemně vypomoci.
3. Snáze se používá neegoistické programování.
4. Členové týmu znají vzájemně svou práci, takže není takový problém, když někdo odejde. Mezi členy je větší důvěra.

Problémem mnoha organizací je nedostatek zkušených a schopných pracovníků. Odměnou schopným bývá služební postup na vedoucí místo, což problém zhoršuje. V lepším případě se z úspěšného analytika stane úspěšný obchodník, který se o realizaci systému nestará a starat nemá. V horším případě bude z úspěšného analytika neúspěšný obchodník. Všimněme si nyní blíže možných forem organizace softwarového týmu.

10.7.1 Horda

Tento typ organizace týmu byl používán zejména v pionýrských dobách programování. Práce se rovnoměrně rozdělí mezi několik nebo mnoho programátorů a každý z nich řeší svůj díl od počáteční analýzy, přes algoritmizaci, programování až po odladění. Rozdělení práce je tedy „lineární“, čistě podle objemu úkolu. Čím více lidí, tím více výsledků. Jedná se postup s mnoha úskalími. Horda se často kombinuje s metodou realizace zvanou velký třesk – všechno najednou. Tato metoda má též svá velká nebezpečí. Nicméně není nutné tento typ organizace zcela zavrhovat. Takovým způsobem byly již realizovány i skutečně velké projekty. Doporučit ji však můžeme pouze dobrým programátorům, kteří v omezeném počtu řeší nepříliš rozsáhlý a přiměřeně složitý problém, který lze navíc rozdělit tak, aby vazeb mezi jednotlivými částmi bylo co nejméně. Neformální organizace se mimo to může uplatnit ve skupině, která je součástí většího strukturovaného týmu a která řeší vymezenou dílčí úlohu.

U větších projektů přechází metoda hordy nepozorovaně do organizace, kdy se sám řešitelský tým neoficiálně zorganizuje kolem špičkových členů týmů. Pak se však nejedná o hordu, ale o jiný typ týmu – demokratickou skupinu. Často se stává, že se horda rozpadne na jednotlivce, kteří vše dělají na vlastní pěst. Takoví programátoři mají někdy přezdívku osamělí vlci.

Osobně známe člověka, který sám napsal kompilátor pro dosti úplnou podmnožinu jazyka ALGOL 60. Vždy se však jedná spíše o výjimku z pravidla. Mimoto osamělý vlk obvykle končívá svou práci v okamžiku, kdy ho to přestává bavit, tedy v době, kdy do úplného dokončení projektu zbývá ještě udělat mnoho a mnoho práce. Osamělý vlk v současné době bývá často programátor nabízející IS napsané ve Fox Pro nebo v podobném databázovém systému.

10.7.2 Demokratická skupina

Demokratická skupina je typ organizace podobný předchozímu, odstraňuje však jeho nejzávažnější nedostatek, totiž absenci vnitřní struktury a kontroly nad jednotností a správností realizace. V demokratické skupině existuje jednoduchá profesní dělba práce – jednotliví členové jsou vzájemně oponenty svých myšlenek a programů. Takový způsob práce pravděpodobně bezděky používá řada neformálních řešitelských skupin. U skutečně dobrých skupin se záhy vytvoří nevelké jádro profesně nejzdatnějších členů týmu, kteří svojí profesionální autoritou neformálně převezmou vedení skupiny. Pokud se sejdou skutečně dobří pracovníci, lze dělat zázraky. Neformálně vznikne strukturovaná skupina s organizací práce blízkou organizaci v týmu šéfprogramátora – viz obr. 10.3.

Má to ovšem svá rizika. Všichni musí být ochotni podřídit se disciplíně týmu, celý tým musí mít stále dost dostatečně náročné práce, jinak začnou intriky. Bývají i problémy generační, chybí starost o „mladou krev“. Mohou vzniknout potíže při řešení nepopulárních úkolů, ty však, jak zkušenost ukazuje, bývají často i nerozumné.

Demokratická skupina představuje velmi efektivní způsob organizace práce, je však méně vhodná pro řešení velkých ostře termínovaných úkolů. U větších úkolů je na závadu to, že demokratický tým může dobře fungovat, jen když není příliš velký, i když jsou i zde možné výjimky, a je stálý, tj. nemění podstatně svoji velikost a složení. V demokratickém týmu bývají špatně zajištěny některé méně atraktivní práce a služby. Pak si zajišťuje příslušné služby každý sám, příkladem je dokumentace, testování atd. S tímto problémem musí řízení projektu počítat.

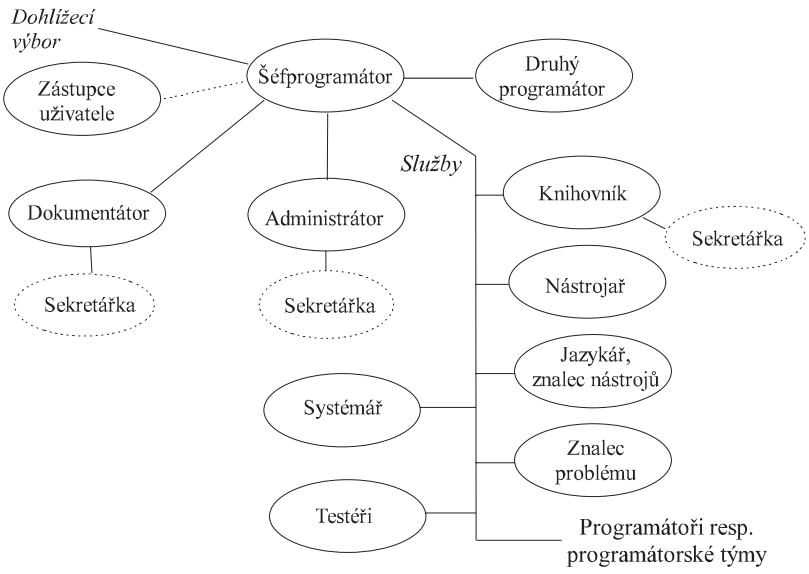
Demokratická skupina pracuje dobře, jen když je tvořena zkušenými a schopnými pracovníky. Složitější organizace týmu diskutované níže dokáže dosáhnout dobrých výsledků i v případě, kdy je většina členů méně zkušená a také méně schopná. Někdy jsou i v demokratickém týmu služby zajišťovány víceméně závaznými dohodami a svými vlastnostmi se blíží strukturovaným týmům uvedeným v následujícím paragrafu. Demokratickou organizaci týmu lze použít při práci v týmech pevné velikosti a přiřazování úkolů týmům a nikoliv při vytváření týmů k úkolům. Demokratická organizace není vhodná pro najímaný tým, protože neformální organizace a psychologické bariéry plynoucí z dlouhodobé existence týmu omezují rychlost integrace nových členů a uvolňování členů týmu při změnách rozsahu úkolů a intenzitě prací. U velkých firem je demokratický tým výjimkou.

U dlouho existujících demokratických týmů bývá tendence ke zkostnatělosti – tým se brání modernizaci práce, přijímání nových členů a vnějším vlivům a tendence k týmovému šovinismu, k nekritické obhajobě týmu, jeho práce a zvyklostí.

10.7.3 Tým šéfprogramátora

Tým šéfprogramátora je vhodný v situaci, kdy je k dispozici několik vynikajících odborníků a pak poměrně mnoho relativně nezkušených pracovníků.

Organizace týmu vychází z toho, že mnohé práce při realizaci software jsou práce relativně nenáročné, avšak zdlouhavé, řada prací je v podstatě „úřednická“ – udržování velkého množství informací, práce s dokumenty, psaní dokumentů atd. Tým šéfprogramátora je soustředěn kolem jádra týmu tvořeného těmito pracovníky (viz obr. 10.3):



Obr. 10.3: Struktura týmu šéfprogramátora.

Šéfprogramátor

je nejschopnějším členem týmu. Osobně provádí dekompozici problému, navrhuje architekturu a algoritmy řešení, osobně píše i programy, ladí je a píše dokumentaci. V zásadě je to člověk, který je schopen realizovat celý projekt sám, ovšem za předpokladu, že není rozptylován podružnými záležitostmi a nikdo mu v práci nepřekáží. Šéfprogramátor musí být velice talentovaný, mít dostatečné zkušenosti a rozvinuté kvality matematika a programátora.

Druhý programátor

je profesním dvojníkem vedoucího programátora. Může mít menší zkušenosti a v týmu zastává především funkci oponenta ideí vedoucího programátora. Často zastupuje tým vůči okolí. Je rovněž schopen dokončit celý projekt a zajišťuje tak vytvářené dílo před hrozbou nedokončení z důvodu odchodu vedoucího programátora. Vztah druhého programátora k vedoucímu není vztahem úplné podřízenosti, třebaže v originálním pojetí z (Baker, 1972) má šéfprogramátor i veškeré administrativní pravomoci, ale spíše vztahem odborného partnerství. Konečné slovo ve všech otázkách má ovšem šéfprogramátor týmu. Funkce druhého programátora umožňuje profesní růst schopných pracovníků. Ti ve spolupráci se zkušenými vedoucími programátory získávají cenné zkušenosti. To je další podstatná výhoda této formy organizace práce.

Někdy zajišťuje druhý programátor některé další funkce, např. koordinaci prací a komunikaci mezi členy týmu. Pokud je druhý programátor zkušený, mohou se v některých etapách práce role šéfprogramátora a druhého programátora vyměnit. Druhý programátor pak dočasně plní funkci šéfprogramátora. Jindy je výhodné svěřit druhému

programátorovi, zvláště pokud je zkušený, práce při návrhu některých částí projektu a odborné posuzování celého úkolu.

Dokumentátor

zajišťuje vytvoření úplné dokumentace projektu. Přestože vlastní dokumentaci v zásadě píše sám šéfprogramátor, výchozí „syrový“ text je potřeba podrobit redakci, ověřit konzistenci s dříve vytvořenými materiály, případně doplnit bibliografické odkazy apod. Dokumentátor rovněž zajišťuje redakci a rozmnožování textů a jejich distribuci mezi členy týmu i mimo něj (např. vytvořením databáze dokumentů).

Administrativní pracovník

se profesionálně zabývá administrativou týmu. I v administrativních otázkách má však konečné slovo vedoucí programátor.

Knihovník

odpovídá především za knihovny projektu a provádí většinu činností spojených s řízením konfigurace projektu.

V závislosti na typu a rozsahu projektu mohou být doplňovány další profese a služby. Jsou to především:

Specialista na jazyk

perfektně zná používané vývojové nástroje, především programovací jazyk. S každým novým programovacím jazykem nebo vývojovým nástrojem se zákonitě objevují i jedinci, kteří jsou nepřekonatelnými experty v jemnostech jeho používání a jejichž konzultace jsou široce využívány. Zatímco vedoucí programátor myslí spíše na úrovni celkové koncepce algoritmů a datových typů, specialista na jazyk mu poskytuje konzultace při interpretaci obtížných technických obratů. Není nezbytně nutné, aby byl přímo členem týmu, může své konzultace poskytovat i více týmům souběžně.

Systémový programátor

V týmu vedoucího programátora se stará především o vazby vytvořených programů na standardní software počítače a o plné využití jeho vlastností. Podle potřeby vytváří specializované softwarové nástroje pro vývoj projektu nebo navrhuje využití již existujících nástrojů.

Testér

se specializuje na otázky testování programů a algoritmů proti funkčním specifikacím, resp. navrženým algoritmům (validace). Od počátku spolupracuje s vedoucím programátorem, vyvíjí či přejímá vhodné testovací postupy, vytváří testové případy a procedury (kap. 13), provádí testy a ve spolupráci s prvním programátorem je vyhodnocuje.

Specialista přes problém

je pracovník, který důkladně zná řešenou problematiku. Někdy lze použít služby pracovníka zákazníka. To je však vhodné pouze tehdy, pracuje-li tento pracovník v týmu na plný úvazek.

Nástrojař

vytváří pomocné nástroje pro vývoj.

Pomocní programátoři

tvoří malé skupiny programátorů, kteří píší programy podle návrhů šéfprogramátora a druhého programátora v případě, že první programátor nemůže tuto práci stihnout sám. Potřeba pomocných prací se dosti často podceňuje a pak se jimi musí zabývat vysoce kvalifikovaní pracovníci.

Pomocné síly

U velkých týmů může vzniknout potřeba pomocných prací (sekretářské práce, příprava dat atd.). V tom případě je tým rozšířen i o tyto pomocné profese. Často to bývá sekretářka šéfprogramátora, případně dokumentátora.

U menších týmů může jeden člen týmu zajišťovat více služeb. U větších týmů mohou některé služby (např. testování) provádět vyčleněné skupiny. Skupiny pomocných programátorů mohou být strukturovány a rovněž tvořit týmy. Tím přecházíme k vícetýmové organizaci práce.

Při vyhodnocování efektů týmu šéfprogramátora byla zjištěna dvakrát až třikrát vyšší produktivita práce než např. ve skupině s demokratickou organizací. Není však jasné, kolik z toho bylo dosaženo díky kvalitě prvního programátora, tj. co lze skutečně připsat organizaci týmu.

Předností týmu šéfprogramátora je naprostá jednotnost návrhu a implementace systému, neboť je celý systém prakticky dílem jednoho člověka, a vysoká výkonnost týmu jako celku. Podřízení členové týmu ovšem musí mít specifické schopnosti, a přitom se mohou cítit odborně nevyužití (Shneiderman, 1980), což nepříznivě ovlivní jejich výkonnost. Může být i problém se získáváním členů týmu vhodných schopností. Hlavním nedostatkem tohoto typu organizace je ovšem naprostá závislost úspěchu či neúspěchu na kvalitách šéfprogramátora. Kolik tak dobrých superprogramátorů na světě existuje?

Tým šéfprogramátora je, striktně vzato, založen na snaze rozšířit rozsah projektů, které může koncepčně řešit jeden resp. dva lidé. Pokud nejsou příliš ostré termíny, může tak být několika pracovníky realizován systém, který by jinak musely dělat desítky až stovky lidí. Slavný operační systém UNIX v podstatě realizovali pouze dva lidé. V týmu šéfprogramátora mohou vzniknout potíže s kolísáním pracovní zátěže členů během řešení. Je zde uplatněna zdravá zásada maximálně využít schopnosti vedoucího programátora a zajistit i odborný růst nejschopnějších mladých pracovníků. Cenné je též vymezení obsahu služeb.

Vzhledem ke zmíněným problémům se tým šéfprogramátora osvědčuje dosti zřídka. Pokud se však sejdou příznivé okolnosti, jsou výsledky vynikající. Prvky týmu šéfprogramátora se vyplatí používat i v jiných formách týmové práce (např. systém služeb a jejich obsah). Tým šéfprogramátora může být v principu realizován neúplně vypouštěním pracovníků realizujících některé služby, které si pak musí zajišťovat ostatní členové týmu sami, až k samotné vedoucí dvojici.

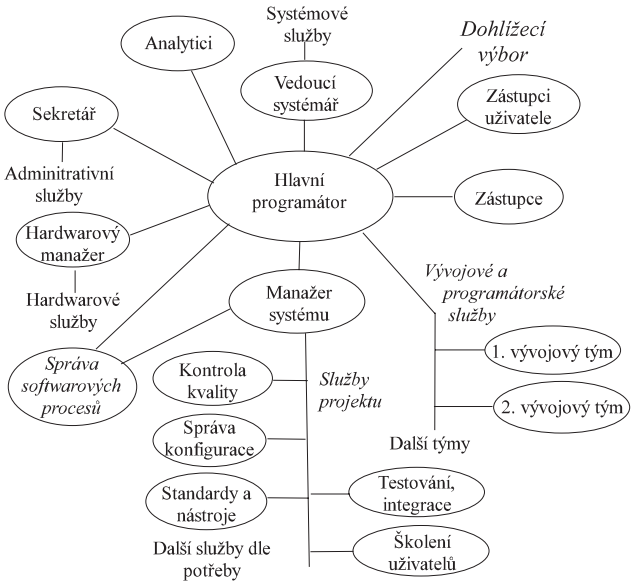
Tým šéfprogramátora je vhodný pro středně velké projekty (statisíce řádků). Dá se využít i při customizaci IS. V tom případě nejsou obvykle třeba služby nástrojaře a jazykáře a chybí většinou i programátoři. Zvláštním typem týmu šéfprogramátora je distribuovaný tým, koordinovaný vedoucím projektu, kde členové týmu rozptýlení po celém světě poskytují služby, hlavně programují. Principy týmu šéfprogramátora lze do jisté míry aplikovat i v demokratickém týmu na základě vnitřních dohod a znalosti potřeby funkcí. Především je žádoucí existence vedoucí dvojice šéfprogramátor – druhý programátor. V ČR je znám případ demokratického týmu, který byl

schopen takové dohody učinit a realizovat v malém kolektivu úspěšný operační systém DOS 4 pro střediskový počítač EC 1027.

Zásady spolupráce vedoucí dvojice by měly být uplatňovány ve všech týmech, včetně dvoučlenných. Využívání osamělých vlků je při vývoji software spojeno s vysokými riziky.

10.7.4 Supertým (tým hlavního programátora)

Opravdu velké úkoly nelze řešit ani v týmu šéfprogramátora. V tom případě je nutné podstatně zvětšit velikost týmu. To je však možné jen tehdy, jestliže je úkol řešen ve spolupráci více týmů. Jednotlivé týmy pak mohou zajišťovat (obr. 10.4):



Obr. 10.4: Struktura týmu hlavního programátora.

a) Jednotlivé služby (především testování, péče o systém a tvorbu nástrojů).
b) Samostatnou realizaci subsystémů (často jen programátorské dvojice).
c) Koordinaci prací. Pro koordinaci prací bývá vytvořen specializovaný tým (projektová skupina) následujícího složení:
   - hlavní programátor a jeho dvojník,
   - zástupci uživatele,
   - znalec problému,
   - skupina zajišťující celoprojektovou administrativu a dokumentaci (sekretariát),
   - u velkých projektů skupina zabývající se schématem vývoje produktu – softwarovými procesy (kap. 18).

Projektová skupina zpracovává vstupní dokumentaci, navrhuje celkovou architekturu a základní algoritmy řešení. Určuje filozofii společných dat, vypracovává metodická doporučení a závazné normy vedení dokumentace,

vybírá programovací jazyk. Projektová skupina realizuje pouze klíčové algoritmy, ostatní zadává k realizaci do systémové skupiny a do programátorských skupin. Zodpovídá za úspěšné dokončení projektu v požadovaném termínu.

Hlavní programátor je šéfprogramátorem ve smyslu výše zavedeného členění profesí. Vede tým, v kooperaci s druhým programátorem realizuje činnosti projektové skupiny a ve sporných otázkách přijímá konečná řešení. Čistě administrativní otázky nemusí být zcela beze zbytku v kompetenci hlavního programátora. I v tom případě je představitelem týmu zodpovědným vůči administrativnímu vedení. Je užitečné, aby hlavní programátor byl i vedoucím pracovníkem v hospodářském smyslu. Není ale nutné, aby se administrativou podrobně zabýval. Existuje totiž nebezpečí, že bude přetížen.

Druhý programátor projektové skupiny je ideovým partnerem hlavního programátora. Vazba mezi ním a hlavním programátorem je o něco volnější než v týmu šéfprogramátora, i když definitivní rozhodnutí přijímá i zde hlavní programátor. Při tom se uplatňují následující řešení: administrativně na stejné úrovni, jeden nadřízen administrativně druhému, přičemž ne nutně je první programátor administrativním nadřízeným. Organizace týmu by měla maximálně snižovat zátěž přetíženého hlavního programátora.

Projektovou skupinu doplňuje specialista na problém. Obvykle to bývá pracovník přímo z útvaru zadavatele projektu nebo zákazníka. V prvních fázích řešení především dohlíží na dodržování vstupních specifikací, v jeho průběhu se pak postupně připravuje na převzetí výsledného produktu do používání.

Programátorské skupiny představují nejnižší úroveň organizace. Skládají se obvykle ze dvou až tří členů a jsou organizovány jako demokratické skupiny. Vazba mezi členy je velmi volná, první programátor je určen víceméně jen z důvodu nutnosti jediného reprezentanta vůči projektové skupině – obsazení funkce prvního programátora se může po určitém čase i změnit. V programátorských skupinách se provádí podrobná specifikace rozhraní (interface) a detailní algoritmizace zadaných algoritmů a podle nich se implementují jednotlivé dílčí programy. Pokud jsou programátoři kvalitní, vyplatí se zapojit je i do prací na dekompozici a definici rozhraní a nechat jim pak relativní samostatnost při realizaci dohodnutých funkcí. To má značné výhody. Snižuje se zatížení vedoucí skupiny, práce běží klidněji. Tato varianta se blíží organizaci práce ve více týmech.

Realizace projektu v týmu hlavního programátora probíhá po úrovních. Na nejvyšší úrovni, v projektové skupině, se provádí základní dekompozice. Projektová skupina pak rozděluje úkoly, akceptuje konečná řešení a sama implementuje klíčové algoritmy systému. Odpovídá za definici rozhraní. Systémová skupina provádí dekompozici společných a speciálních algoritmů a po schválení projektovou skupinou je implementuje. Programátorské skupiny realizují na nejnižší úrovni dílčí algoritmy a programy. Sekretariát pak poskytuje vymezené služby všem členům týmu.

Tým hlavního programátora oslabuje největší nevýhody organizace šéfprogramátora – absolutní závislost výsledku na kvalitě vedoucího programátora týmu a možné nevyužití odbornosti ostatních členů týmu. I zde však úspěch či neúspěch projektu ve velké míře závisí na kvalitách a kondici hlavního programátora. To je ale obecný problém špičkových pracovníků v kterékoli týmové organizaci. Podobně jako v případě týmu šéfprogramátora lze i zde používat různé varianty organizace týmu hlavního programátora:

a) Úkoly pro pomocné programátory jsou voleny tak malé, že je lze zadat jednotlivcům.
b) Služby jsou omezeny např. díky metodám automatické generace dokumentace nebo využitím globálních služeb firmy.
c) Existují různé mezistupně směrem k organizacím vícetýmové práce (některé subsystémy mohou být odděleny a realizovány prakticky samostatně).

Tým hlavního programátora je vhodný i pro řešitelský tým s proměnlivou velikostí (najímané týmy). Je vhodný i pro velké projekty. Práce týmu hlavního programátora vyžaduje poměrně mnoho administrativy.

10.7.5 Multitýmová organizace (konfederace)

Mnoho projektů je nad síly jediného, byť i kvalitního týmu. Důvody mohou být v časové náročnosti nebo logické složitosti řešení, ale také v odborné specializaci již konstituovaných týmů. Týmy řešící např. problematiku přímého řízení technologií nebudou schopny dobře řešit i otázky plánování a podpory rozhodování managementu. Je užitečné, aby i vícetýmová organizace měla svou strukturu a řešení celku se neopíralo jen o neformální „sousedské“ dohody. Multitýmová organizace předpokládá existenci vedoucí skupiny, jejímiž členy jsou i všichni vedoucí jednotlivých řešitelských týmů. Týmy pak mají vnitřní, třeba i navzájem různou, organizační strukturu.

Jednotlivé funkce jsou analogií funkcí v týmech hlavního programátora. Rozhodujícím úkolem vedoucí skupiny projektu je dekompozice systému na jednotlivé úkoly tak, aby jednotlivé úkoly byly samostatně řešitelné jednotlivými týmy, které mohou samy postupovat obdobně. Dekompozice systému zahrnuje především tyto hlavní úkoly:

1. Definice částí, jejich funkcí a rozhraní mezi nimi.
2. Stanovení funkčních charakteristik a obsahu společné datové základny.
3. Stanovení norem vedení dokumentace a psaní programů.
4. Pravidla koordinace, kontrolní dny, pravidla dohadovacích řízení, inspekce.
5. Plánování koordinačních opatření.

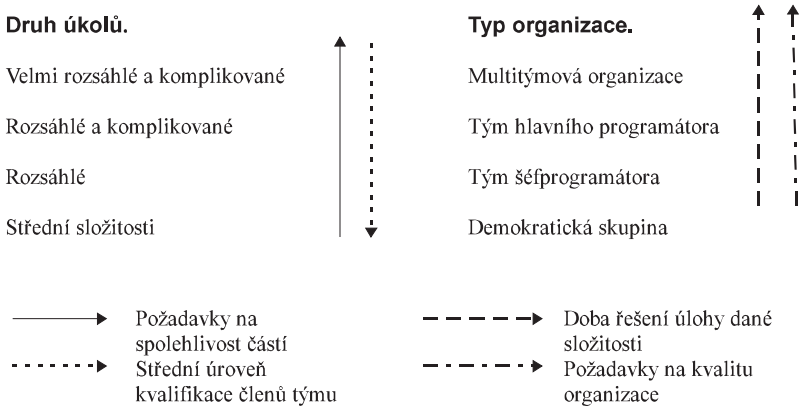
Multitýmová organizace se od organizace týmu hlavního programátora liší větší decentralizací prací a odpovědností. To je možné jen při použití vhodných technik a jen tehdy, když daný projekt takovou dekompozici na prakticky samostatné podúkoly připouští. Multitýmovou organizaci lze přirovnat ke konfederaci států, zatímco tým hlavního programátora lze přirovnat k poměrně centralizované federaci. Odtud plyne i hlavní problém konfederace – nebezpečí dezintegrace. Vztahy mezi týmy mají tendenci být vztahy konkurenčními a pokud se nepřijmou odpovídající opatření, zhoršují se, Směřují k podezíravosti, podceňování výsledků druhých týmů a přeceňování vlivu jejich neúspěchů atd., viz (Mankin a ost. 1994). Praxe ukazuje, že v případě, že se používá technika spolupracujících aplikací, lze konfederované multitýmy úspěšně využívat.

10.8 Kritéria volby týmové organizace

Efektivní týmová organizace musí být vhodná pro řešení daného problému. Je zbytečné zatěžovat nepříliš komplikovaný projekt rozsáhlou organizační strukturou a zanášet do dobře fungující organizace zbytečné otázky vzájemné komunikace jednotlivých programátorů. Naopak může být stejně škodlivé, jestliže je složitý projekt implementován bez řádné organizační struktury řešitelského týmu (obr. 10.5).

Kritéria pro výběr typu týmové organizace můžeme rozdělit na vnější, daná objektivními požadavky problému, a vnitřní, postihující naše subjektivní možnosti. Mezi vnější kritéria řadíme např. rozsah a logickou složitost problému, požadovanou dobu realizace nebo požadavky na stupeň spolehlivosti výsledného produktu. Vnitřními kritérii jsou pak kvalita a množství programátorů, které máme k dispozici, a jejich volná pracovní kapacita. Mohou totiž současně pracovat i na jiných projektech.

Na obr. 10.5 jsou doporučeny typy organizací pro vybrané problémy, klasifikované podle rozsahu, složitosti a požadované doby řešení.



Obr. 10.5: Volba typu organizace týmu.

Složitější metody organizace vyžadují značné množství administrativní práce všech zúčastněných. I celková pružnost řešení se zhoršuje (jednou učiněná rozhodnutí nelze snadno měnit).

Lze namítnout, že existují případy, kdy neformální organizace dává lepší výsledky než důkladně organizovaná práce v týmu. Trend směřující v podstatě od řemeslné neformální organizace práce k „tovární“ výrobě software v organizovaných týmech však bude zřejmě, především v „rutinních“ oblastech, nadále narůstat. To je další příklad zesilování inženýrských rysů v tvorbě software. Realizace velkých softwarových produktů ve velkých týmech s mnoha kontrolami, administrativou, metodami plánování a sledováním průběhu prací připomíná po stránce organizační v mnohém tvorbu velkých projektů v klasických oborech techniky.

Složitost organizace musí odpovídat složitosti a rozsahu úkolu. (Kiesler et al., 1994) poukazují na závislost efektivnosti týmu na nutném rozsahu komunikace mezi členy týmu. Je-li potřeba komunikace velmi nízká, není třeba sestavovat tým. Tým v tom případě pracuje neefektivně. Tým také pracuje neefektivně, není-li organizován. Pokud je potřeba komunikace mezi členy týmu velmi vysoká, pracuje tým s ostrými požadavky na formální procedury a s vysokou mírou strukturovanosti také neefektivně. Komunikaci v tomto případě zdržují procedury přijímání změn a rozhodnutí a procesy jejich kontroly. Nejefektivněji tedy pracují týmy vysoce organizované, avšak jen s mírnou potřebou komunikace mezi členy týmu a také týmy se střední až velkou potřebou komunikace mezi členy, ale jen se střední složitostí organizace. Tato situace je častější u menších týmů.

10.9 Infrastruktura podpory týmové práce

Moderní informační technologie nabízejí mnoho prostředků podpory týmové práce počínaje metodami spolupráce a softwarem podporujícím spolupráci a řízení projektu konče. V souladu s příslovím o kovářově kobyle nejsou tyto prostředky ke škodě věci při vývoji software doceňovány.

Výkon týmu pozitivně ovlivňuje dodržování zásad ergonomie a vytvoření správného psychologického klimatu (kap. 4). Je proto výhodné, aby měl každý soukromí, aby nebyl při práci zbytečně rušen (např. telefony, diskuze sousedů). Pro některé členy týmu i pro jejich výkon může být výhodné, když mohou pracovat doma. K tomu je však nutné vytvořit technické i organizační předpoklady.

10 Práce v týmu

Pro hladkou práci je důležitá společenská místnost, kde je možné jednat se zákazníky a provádět inspekce a jiné formy oponentur. Stejné použití má společenská místnost pro neformální setkání nad kávou a pro neformální diskuze o průběhu prací. Společná místnost má být vybavena promítacím zařízením, tabulemi a měla by působit příjemným dojmem, např. díky nábytku, květinám a výhledu. Je to důležitá vizitka firmy.

Práce týmu může být zefektivněna využitím prostředků podpory týmové práce. Pro menší týmy postačují standardní groupwarové prostředky jako Lotus Notes nebo jednodušší prostředky firmy Microsoft či jiná forma software na podpory práce v týmu (groupware). Pro větší týmy jsou žádoucí prostředky process engineering (kap. 18) a takové nástroje, jako je správa konfigurace.

Pro spolupráci v rámci týmu je důležitý přístup k síti, což umožní plné využití jak software podpory práce skupin tak moderní vývojové nástroje (CASE – kap. 19, grafické prostředky, informační systém projektu, moderní databáze atd.). Velké týmy by měly využívat i takové prostředky jako jsou systémy správy prací (workflow). Hlavní překážkou zavedení nástrojů podpory práce týmu nebývají náklady, ale ochota přijmout pravidla hry a disciplínu. Tyto problémy jsou podobné těm, se kterými se setkáváme u zákazníků při instalaci a oživování informačních systémů.

11

Softwarové architektury a návrh systému

Specifikace a návrh systému jsou silně ovlivňovány tím, jaká bude architektura systému. Systém může být koncipován jako monolit nebo jako více spolupracujících autonomních komponent. Návrh částí může být založen na klasickém strukturovaném přístupu nebo na objektově orientované technologii. V této kapitole se budeme především zabývat technikami vývoje systému metodou spolupráce aplikací. Zmíníme se rovněž o strukturovaných technikách a objektové orientaci při realizaci jednotlivých komponent.

11.1 Faktor času

Doba mezi podstatnými inovacemi hardwaru je 3 až 5 let. Doba mezi podstatnými inovacemi základního softwaru, jako jsou operační systémy a databázové systémy, je 5 až 7 let. Customizace IS trvá obvykle 1 až 3 roky. Vývoj od počátku bývá ještě zdlouhavější. Informační systém bývá používán i více než 10 let. Během doby života IS bude tedy nutné vyměnit hardware a často i operační systém. Vzniknou nové požadavky na propojení IS s nově vzniklými produkty. Dnes se IS propojují s prostředky řízení prací (workflow system) nebo s prostředky vyhledávání a vizualizace dat v rámci technik data mining v datových skladech.

Moderní informační technologie umožňuje tento problém řešit. Moderní řešení jsou založena především na následujících dvou vzájemně se doplňujících strategiích:

- dekompozice IS do souboru autonomních částí: aplikací, softwarových komponent, nebo softwarových agentů[1],
- objektově orientovaná specifikace, návrh a kódování jednotlivých částí s případným uplatněním prvků strukturované analýzy a návrhu.

Aplikace je v tomto textu chápána jako softwarová entita, která se chová jako samostatný program (proces) v multiprogramovém prostředí a která je po případných nepříliš náročných úpravách schopna samostatné existence: může být z hlediska uživatele použita jako samostatný plně funkční program – např. program hlavní knihy a účtů podvojného účetnictví. Pokud je vyřešen problém spolupráce aplikací, je do značné míry vyřešen i problém využití existujících aplikací (legacy systems). Může být ale obtížné vytvořit přístup – bránu – k datům existujících aplikací.

1. Softwarový agent je aplikace schopná migrovat po síti, vytvářet kopie – klony, spolupracovat s jinými agenty a někdy i zdokonalovat své funkce. Softwarová komponenta je obvykle menší část softwaru, obvykle jeden objekt nebo malá skupina objektů.

Informační systémy koncipované jako jednotně koncipovaná síť spolupracujících aplikací se společnými službami a centrální koordinací aktivit se nazývají *federalizované IS*. Pokud aplikace pracují bez centrální koordinace obdobně jako sítě peer to peer, hovoříme o *konfederovaných IS*.

Možnost realizace IS jako sítě spolupracujících aplikací závisí do jisté míry na požadovaných funkcích. Musí být však vzata v úvahu již při specifikaci požadavků, které by měly být dekomponovány tak, aby se jednotlivé skupiny požadavků daly realizovat jako samostatná část. Restrukturalizace požadavků je možná i později, je to však pracné a vnáší to do systému chyby. Dekompozice do samostatných aplikací umožňuje použití účinných metod ladění a sledování za provozu.

Spolupráce aplikací usnadňuje řešení řady softwarově inženýrských problémů, jako je postupná modernizace, používání produktů třetích stran, snižování pracnosti, lepší využívání hardwaru atd. Zároveň však umožňuje nové obraty. Níže uvádíme příklady použití obratů spojených s technikou spolupráce aplikací.

11.2 Dekompozice IS do sítě spolupracujících aplikací

Síť spolupracujících aplikací je tvořena aplikacemi, které spolu komunikují. Techniky komunikace aplikací využívají některou nebo více z následujících metod:

- výměna zpráv,
- přístup k datům spolupracujících aplikací,
- jiné metody shrnuté pod zkratkou API (application programming interface), kdy je spolupráce realizována přímo mezi výkonnými částmi (logikou) aplikací na základě vhodného protokolu.

Je zřejmé, že nejobecnějším prostředkem je neomezený přístup k datům aplikací třetích stran, neboť nepředpokládá, že cizí aplikace spolupráci zajišťuje. Nejsou tedy nutné žádné úpravy cizích aplikací. Styk s aplikacemi prostřednictvím přímého přístupu k „cizím datům“ je bezpečný, pokud jsou „cizí“ data přístupná jen pro čtení. To je případ většiny služeb Internetu. Nekontrolované změny dat jiných aplikací jsou velmi nebezpečné, neboť mohou vést k těžko odhalitelným chybám. Dobrý kompromis je použití technik aktivních databází.

Aplikace používající nějakou formu API musí cizím datům „rozumět“. Musí vědět, zda čtená posloupnost bitů nebo bytů je faktura nebo objednávka či něco jiného. Pokud se autoři komunikujících aplikací mezi sebou dohodnou, nevzniká žádný problém. To ale není vyhovující řešení, protože např. klienti přepravců nebo obchodních domů jsou z různých zemí, mají různé systémy. Klientela se také neustále mění. Proto je snaha definovat strukturu standardních obchodních a hospodářských zpráv. Toto úsilí je známo pod zkratkou EDI – electronic data interchange, elektronická výměna dat. Koordinace a standardizace EDI se ujaly OSN a ISO. Toto standardizační úsilí je známo pod zkratkou EDIFACT (Electronic Data Interchange for Administration and Transport, elektronická výměna dat pro administrativu a dopravu). Standardizace EDI je zatím neúplná. Schváleny jsou normy pro syntaxi (ISO 9735) a slovníky dat (ISO 7372). Velmi perspektivní řešení je jazyk XML dovolující definovat formát dat v rámci internetovského rozhraní, podrobnosti viz časopis BYTE z března 1998.

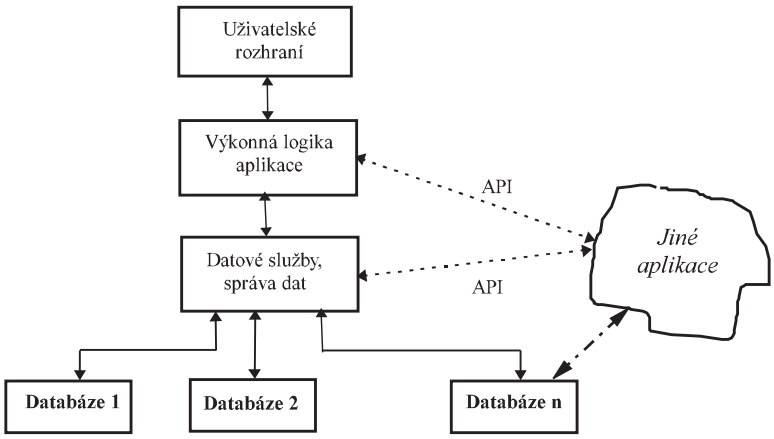
Aktivity EDI zahrnují i návody na návrh zpráv, implementaci a používání EDI a řadu dalších pomocných aktivit (viz `http://www.iata.org/edi/Abotedi.htm`|.[2] EDI se považuje za horké téma současnosti.

EDI je důležitou podmínkou elektronického obchodování na Internetu.

2. Jazyk KIF (formát výměny znalostí, Knowledge Information Format) definuje v principu obdobným způsobem strukturu zpráv mezi softwarovými agenty.

Moderní databázové systémy umožňují při vzniku jisté situace v datech, např. narušení integrity dat, vyvolat automaticky, nezávisle na aplikacích používajících danou databázi, proceduru – trigger – uloženou v databázovém stroji. Procedura na vznik takové situace reaguje, zablokuje např. nepřípustné změny. Trigger se tedy aktivuje automaticky při vzniku určité situace. Databáze s triggery tedy aktivně reaguje na změny – je *aktivní*.

Uživatelské rozhraní (UI) má řadu zvláštností. Je proto výhodné UI koncipovat jako relativně samostatný subsystém. Je rovněž výhodné jako logicky relativně samostatný subsystém realizovat i správu dat. Logická struktura aplikace je pak tvořena třemi vrstvami: uživatelským rozhraním, vlastní výkonnou logikou aplikace a subsystémem správy dat (obr. 11.1).



Obr. 11.1: Třívrstvá (three tier) struktura jednotlivé aplikace.

11.2.1 Datové sklady (data warehouse)

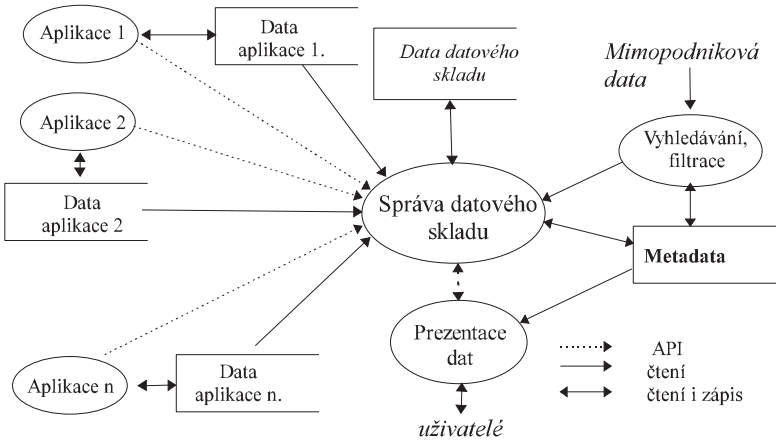
Může-li jedna aplikace prakticky bez kontrol měnit data jiné aplikace, existuje velké nebezpečí poškození dat a rozpadu funkcí celého systému. Takové chyby se velmi obtížně napravují. Problémy se zmenší, ne-li vyřeší, uskutečňuje-li se přístup k datům prostřednictvím mezilehlých aplikací (viz obr. 11.2), které:

- umožňují unifikaci spolupráce mezi jednotlivými výkonnými aplikacemi a bázemi dat;
- snižují počet cest informací a umožňují implementovat kontrolní operace, což značně usnadňuje ladění a lokalizaci chyb při výpadcích systému za provozu;
- specializované aplikace mohou realizovat řadu ochranných mechanizmů: přístupová práva, selektivní práva změn, testy smysluplnosti změn atd. Pomocná aplikace „skladník“ může poskytovat řadu služeb, jako jsou pokročilé prostředky analýzy dat;
- datakopectví (data mining), vizualizace a reorganizace dat. Touto cestou lze realizovat integrovaný systém správy dat – datový sklad (data warehouse).

Služby datového skladu mohou zahrnovat i vytváření kopií historických a externích dat a jejich prezentaci. Kopie některých dat jsou uloženy v datovém skladu. K tomu je nutné vhodná data vyhledávat, odfiltrovat nepoužitelná data a transformovat je do tvaru vhodného pro práci datového skladu. Bývá žádoucí, aby služby datového skladu

zahrnovaly i interaktivní přístup k aktuálním datům. I pro tato data jsou potřebné služby vyhledávání, filtrace a transformace pro prezentaci uvnitř datového skladu. Efektivní realizace funkcí datového skladu bývá založena na datech o datech (metadatech) definujících strukturu a organizaci dat a do značné míry i jejich prezentaci. Je to v jistém smyslu zobecnění definice struktury databáze v klasických databázích.

Datové sklady jsou dostupné jako produkty předních databázových firem. Správa skladu obvykle využívá data (metadata) definující transformace, filtraci a prezentaci dat.



Obr. 11.2: Možná architektura datového skladu.

Technika datového skladu umožňuje skrýt před aplikacemi implementační detaily (v jaké databázi jsou data uchována, kde jsou data na síti atd.). Podmínkou, někdy ne právě snadno splnitelnou, je možnost přístupu ke všem datům. Používání databází se standardem SQL přístup k datům značně zjednodušuje. Problém správné práce s daty samozřejmě zůstává. Datový sklad musí reagovat na měnící se požadavky. Nelze tedy podobně jako u IS předpokládat, že bude po vytvoření delší dobu využíván beze změn.

Aplikace používající služby datového skladu a pomocná aplikace „správa skladu“ mohou být provozovány na různých místech sítě. Dokonce i data každé aplikace mohou být uložena distribuovaně – na různých místech sítě. Rozhraní na externí data může umožňovat dočasnou spolupráci tržních subjektů tím, že se částečně a dočasně propojí IS spolupracujících podniků vytvářejících dočasnou tržní alianci (jako kdysi IBM a Microsoft). Data mining (DM, Fayyad et al., 1996) je soubor technik sloužících k rozpoznávání vlastností dat (patterns, obrazců) hodných pozornosti.[3]

Obrazce jsou základem poznatků. V našem případě je poznatkem pravděpodobná příčinná souvislost mezi kouřením a rakovinou plic. Existují pokusy automatizovat vyhledávání obrazců a dokonce i objevování poznatků (knowledge discovery, KDD). Základem těchto technik jsou metody matematické statistiky a zčásti i umělé inteligence. Techniky data mining a odhalování poznatků jsou předmětem intenzivního výzkumu.

Spolupráci aplikací můžeme v datovém skladu využít následujícím způsobem. Prezentační vrstva datového skladu může být realizována tak, že v ní lze jednoduchým způsobem využívat služeb systémů statistické analýzy

3. Příkladem může být zjištění, že z osob starších šedesáti let vedených v databázi, které vykouřily alespoň 300 000 cigaret, onemocnělo 95 % do pěti let rakovinou plic. Jiným příkladem je vizuálně ověřená charakteristika trendů prodeje (nárůst 5 % ročně).

dat, jako je GPSS, BMDP, Mathematica atd. Vyhledávání obrazců v datech vyžaduje za současného stavu řešení interakci s uživatelem systému. Tato spolupráce může využívat různé grafické systémy. Pro marketing jsou např. výhodné prostředky zobrazování výsledků prodeje do map prostřednictvím geografických informačních systémů. Odhalování poznatků, KDD, je pak netriviální proces identifikace validních, zajímavých, potenciálně použitelných a srozumitelných obrazců.

Pro KDD, ale nikoliv nutně pro DM, je nutné mít k dispozici jazyk L pro popis obrazců. Obrazec je pak výrazem v tomto jazyce. L nemusí být jazyk znakový, ten by se dal využít ve výše uvedeném příkladu, může to být i vhodný jazyk popisu grafických objektů. V tom případě však jazyk nevyžadující silnou interakci s uživatelem zatím chybí. Proces odhalování poznatků může umožňovat i postupné zpřesňování poznatků a měl by mít i jistou autonomii; jeho kroky by neměly být plně určovány rozhodnutími operátora. Existují pokusy, aby se při tom systém mohl i učit. Poznatky by měly být validní, tj. měly by platit nejen pro data, ve kterých byly nalezeny. Potenciální užitečnost je další důležitou podmínkou. Poznatky by měly „být k něčemu“. Existují systémy, které jsou schopny vygenerovat obrovské množství tvrzení platných s určitou pravděpodobností. Problém je, jak v této kupě sena najít jehlu tvrzení, která jsou pro uživatele zajímavá a něco srozumitelného mu říkají.

Proces odhalování/získávání poznatků, ať již automatizovaný či nikoliv, probíhá v následujících etapách:

1. Vymezení a porozumění oblasti aplikace (např. marketing), zjištění existujících poznatků, znalostí a cílů koncových uživatelů.
2. Vymezení/výběr dat relevantních pro daný cíl, vymezení charakteristik a vlastností, které je žádoucí analyzovat.
3. Předběžné zpracování dat: vyloučení šumů/chybných dat, řešení problému chybějících dat, uvážení trendů v čase.
4. Redukce dat: zmenšení počtu sledovaných atributů, hledání invariantů.
5. Výběr úkolů pro data mining (DM), např. hledání regresních vztahů.
6. Výběr modelu pro DM a jemu odpovídajících algoritmů. Tato část zahrnuje i případnou volbu příslušného, např. statistického balíku resp. aplikace.
7. Provedení DM. S DM interaktivně spolupracuje i koncový uživatel.
8. Vyhodnocování nalezených obrazců a jejich validace s případným opakováním výše uvedených kroků.
9. Zahrnutí získaných poznatků do systému dosavadních znalostí s případným řešením konfliktů.

DM využívá především následující techniky:

Klasifikace – volba funkce třídící objekty do určitých skupin. Příkladem může být třídění obrazů v grafické databázi, typy trendů na burze atd.

Regrese a odhady – nalezení funkce, která umožňuje odhad nějaké charakteristiky nebo jejího trendu, např. počet úmrtí na plicní rakovinu, z hodnoty jiných charakteristik, např. počtu vykouřených cigaret, pohlaví a věku.

Shluková analýza. Tato technika je založena na postupech zjišťování vzdálenosti objektů podle nějakého kritéria a jejich roztřiďování do shluků různých vlastností. Typickým případem je zjišťování zákazníků s podobným chováním.

Sumarizace. Při této technice se vizualizují a vyhodnocují vlastnosti dat na základě jejich některých globálních, většinou statistických, charakteristik.

Analýza závislostí. Při této technice se zjišťuje, zda jsou nějaké atributy statisticky závislé a případně jak silně jsou závislé. Relaci závislostí lze zobrazit grafem závislostí a ten dále analyzovat.

Z výše uvedeného vyplývá, že kvalitní prezentační vrstva datového skladu by měla využívat řadu aplikací poskytujících podporu analýze a sběru dat, především metod a nástrojů matematické statistiky, případně umělé

inteligence, a také prezentace dat (grafické systémy, generátory sestav) atd. S využíváním datových skladů je spojena řada problémů. Uveďme některé:

- Nutnost uchovávání obrovských souborů dat často distribuovaně.
- Zajímavá data je nutné vyhledávat v rozsáhlých celosvětových sítích. Při tom je třeba minimalizovat náklady na síťové služby a zkvalitňovat selekci dat. Tento problém se řeší uplatňováním techniky softwarových agentů. Softwarový agent (SWA) je aplikace schopná replikace a schopná migrovat po síti vyhledávající relevantní informace s co nejmenšími náklady. Žádoucí je, aby SWA byl schopen samostatně zdokonalovat svou činnost (učil se) a při tom spolupracoval s jinými agenty.

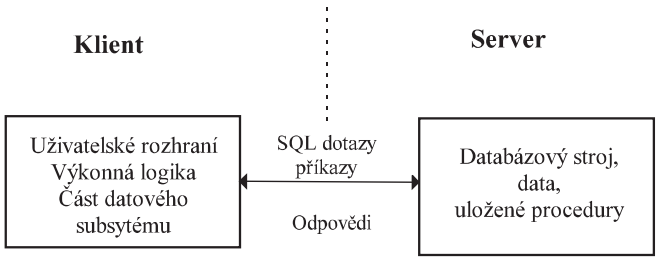
Řada technických problémů je spojena nejen s udržováním rozsáhlých dat (kde, rychlost, zálohování, atd.) ale také s problémy jejich efektivní analýzy. Problémem jsou multidimenzionální statistické úlohy, vylučování chybných dat (šumu) atd.

Kvalifikované využívání datového skladu je tedy možné jen se znalostmi oblasti aplikace (cíle, relevantnost a využitelnost poznatků), informatiky (databáze, sítě, grafické systémy, technologie realizace skládáním komponent případně spoluprací agentů), matematické statistiky (regrese, závislosti,...) a umělé inteligence. Požadavky na datové sklady se rychle vyvíjejí. Datové sklady jsou postupně integrovány do IS.

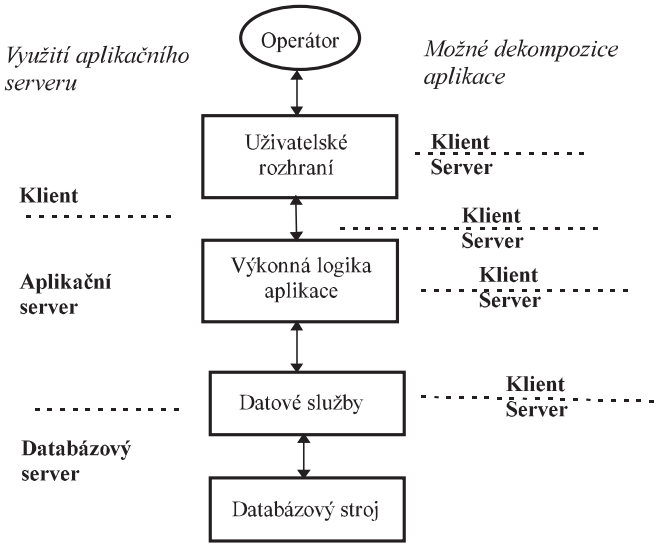
Náš výklad je v zájmu srozumitelnosti poněkud zjednodušen. V případě, že požadujeme, aby uživatel nebyl obtěžován tím, že je nutno při funkci systému přecházet mezi aplikacemi, je nutné aplikace obsahující vrstvu styku s obsluhou modifikovat tak, že komunikují s dalším procesem - správcem styku s obsluhou, který vytváří integrované rozhraní pro všechny aplikace naráz. Pro takto zvolenou architekturu je nutné, aby bylo možné rychle budovat a modifikovat rozhraní uživatele s aplikacemi tak, aby se rozhraní na více aplikací nelišilo od situace, kdy se pracuje s jedinou aplikací. Techniky, které to umožňují, se teprve vyvíjejí, jsou však ve velmi dobře použitelné formě již komerčně dostupné jako technika drill down nebo drill around firmy Lawson Software nebo Uniface od Oracle aj. Tyto nástroje umožňují, aby si rozhraní do značné míry definoval sám uživatel podobně, jako kdyby pracoval s databází. Podrobnosti implementace nejsou zveřejněny. Pravděpodobně je použita technika obdobná té, která se používá v databázových systémech pro prezentaci výstupů SQL dotazů. Za „řádek tabulky“ se u těchto metod považuje výsledek volání podprogramů jednotlivých aplikací. Podrobnosti jsou uvedeny níže.

11.2.2 Architektura klient – server

V tomto a v následujícím paragrafu uvedeme příklady ukazující, že dekompozice do tvaru jednotné budované sítě spolupracujících aplikací přináší nové kvality při vývoji IS.



Obr. 11.3: Klasické schéma spolupráce mezi klientem a serverem (tlustý klient).



Obr. 11.4: Možnosti dělení kódu aplikace.

První interaktivní IS byly realizovány na výkonném sálovém počítači se sítí jednoduchých („hloupých“) terminálů schopných realizovat pouze znakovou komunikaci. S příchodem osobních počítačů a jejich grafických prostředků se nabídla možnost využití grafického uživatelského rozhraní (GUI). Zároveň se objevila možnost provozovat na osobním počítači i část vrstvy logiky. Tato možnost byla využita následujícím způsobem:

Aplikace se rozdělí na dvě části – část klientovou a část běžící na výkonném centrálním počítači – serveru či na síti serverů. Sever může, ale nemusí být sálový počítač, v současné době bývá jako server častěji použit počítač, který spíše připomíná velmi výkonný osobní počítač než počítač sálový. Centry velmi výkonných IS bývají však i dnes sálové počítače. Na serverech se nejčastěji používá OS UNIX a firemní systémy jako AS/400 nebo MVS. Nejnověji proniká na servery i Windows NT.

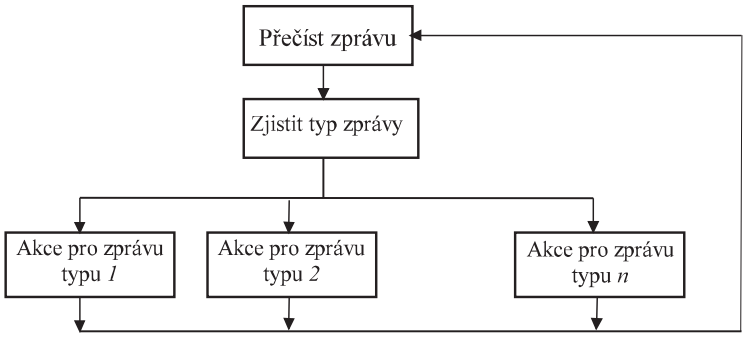
Klientová část na osobním počítači je obvykle provozována pod operačními systémy Windows, OS/2 nebo Macintosh.

Při dělbě aplikace mezi serverem a klienty jsou možné následující varianty:

A: a) Klient:
- vrstva styku s operátorem,
- vrstva výkonné logiky,
- část vrstvy správy dat (generace SQL příkazů / interpretace odpovědí).

b) Server:
- interpret SQL příkazů (databázový stroj).
- vlastní data.

Toto schéma (obr. 11.3) nevyžaduje žádné speciální programátorské obraty, ponĕvadž propojení mezi logikou na klientu a serveru lze uskutečnit pomocí nástrojů, které nabízí výrobce databáze. Je známo, že velikost softwaru neustále roste. Je-li většina logiky aplikace na klientu, znamená to, že je třeba neustále



Obr. 11.5: Vývojový diagram aplikace – příjemce zprávy.

zvyšovat kapacitu (výkon i rozsah paměti) klientů. Je-li klientů mnoho, znamená to neustálé nemalé investice. Tento problém je znám jako problém tlustého klientu (fat client). Tlustý klient není vhodný pro IS s mnoha pracovními místy.

B: Částečné řešení:
Přesun části výkonné logiky pomocí tzv. ukládaných procedur na server. Schéma z obr. 11.3 zůstává v platnosti s tím, že SQL příkazy jsou rozšířeny o správu triggerů. Tento obrat neřeší zcela problém tlustého klientu a není příliš vhodný ani pro použití datového skladu. U středně velkých systémů však představuje rozumný kompromis.

C: Optimální řešení:
S využitím technik objektově orientovaného návrhu a programování lze aplikaci rozdělit i na jiném místě, např. mezi vrstvou GUI a výkonnou logikou (obr. 11.4). Uplatněním této techniky lze optimalizovat zatížení klientu, serveru a sítě. Klient neobsahuje rozsáhlá data ani programy, je tenký (thin).

D: Třívrstvá architektura:
Pro opravdu velké systémy se aplikace rozdělí na více částí – klientovou, část logiky, která pracuje na dedikovaných počítačích – aplikačních serverech – a část, která bude na serveru nebo serverech zajišťujících správu dat.

Pokud je aplikace napsána objektově, lze poměrně snadno vytvořit klientový program umožňující spolupráci se všemi aplikacemi. V nejjednodušším případě může operátor přepínat mezi aplikacemi tak, že jednotlivé obrazovky představují vazbu právě na jednu aplikaci. Příkladem takového přístupu je přepínání aplikací v MS Windows. Dokonalejší systém umožňuje v rámci jedné obrazovky realizovat přístup k datům a ovládat více aplikací současně.

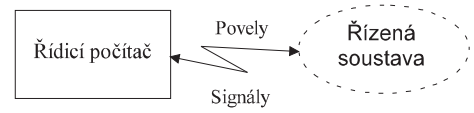
Technika realizace může být založena na následujícím obratu. Volání podprogramů či metod jednotlivých aplikací vracejících nějaké hodnoty lze chápat jako čtení řádků jisté virtuální tabulky. Kombinace volání lze chápat jako obdobu výsledku volání jistého dotazu do virtuální databáze. Výsledky tohoto dotazu lze tedy zobrazit unifikovaným způsobem do formuláře na obrazovce obdobně jako při zobrazování dotazů v jazyce SQL. Úkolem uživatele je pak formulovat vhodný pseudoSQL dotaz a jeho zobrazení na obrazovce. Zkušenosti s některými komerčními produkty naznačují, že je to docela schůdná cesta.

11.2.3 Příklad dekompozice s použitím výměny zpráv

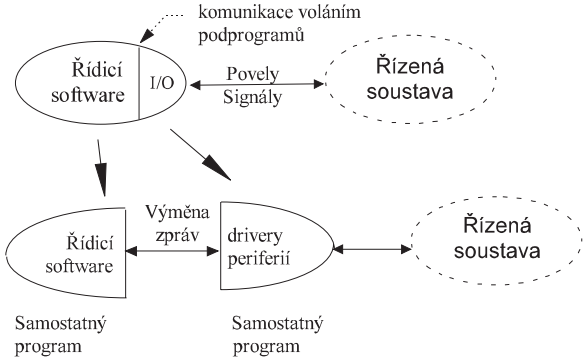
Až dosud jsme se zabývali technikami spolupráce aplikací, které byly založeny na komunikaci prostřednictvím dat uložených v databázích přístupných více aplikacím. Zde popíšeme techniku založenou na výměně zpráv. Tato technika je blízká, není však identická, objektově orientované metodologii ve smyslu (Booch, 1991). Její některé obraty přesahují rámec klasické objektové orientace.

V našem případě bude prostředkem komunikace mezi aplikacemi, které však mohou být realizovány i jako vlákna – threads – jedné aplikace. Výměnu zpráv je možné implementovat různými způsoby. Popíšeme způsob založený na koncepci poštovní schránky. Poštovní schránka S je zařízení, na které se může vázat fronta zpráv. Příkazem *SEND*(*S*, *Z*) se zpráva *Z* zařadí na konec fronty zpráv schránky *S*. Příkaz *WAIT(S, buffer)* pracuje následujícím způsobem:

a) Je-li fronta zpráv schránky *S* prázdná, čeká se do doby, než nějaký proces provede příkaz *SEND (S, zpráva)*.
b) Není-li fronta schránky *S* prázdná, vezme se první zpráva z fronty zpráv schránky *S* a uloží se do vyrovnávací paměti a vykonávání procesu pokračuje příkazem následujícím za příkazem *WAIT (S, buffer)*.



a) Soustava řízená počítačem



b) Oddělení ovladačů periferií realizujících styk s řízenou soustavou.

Obr. 11.6: Dekompozice procesu pracujícího v reálném čase.

Příkaz *SEND* tedy provádí odesílatel zprávy, příkaz *WAIT* příjemce zprávy. Pro jednoduchost nepopisujeme řešení situace, kdy se pokouší příkaz *WAIT* provádět více procesů současně. Tento problém se řeší podobně jako u softwarových semaforů. Aplikace – příjemce má strukturu popsanou zjednodušeně na obr. 11.5. Každá aplikace má tedy tvar nekonečné smyčky začínající čekáním na příchod zprávy. Pro vyjádření toho, že zprávy mohou být posílány mezi vlákny nebo nezávislými programy, budeme oba případy označovat slovem aktor. Operace *WAIT* je velmi snadno implementovatelná. V operačním systému UNIX jsou příkazy *WAIT, SEND* snadno realizovány

pomocí struktury *message_queue*. Pro realizaci programovacího systému, který by byl snadno modifikovatelný, je výhodné každému aktoru A přiřadit identifikační číslo (logické číslo) $ID(A) = j$. Každému aktoru A s $ID(A) = j$ je vzájemně jednoznačně přiřazena schránka S_j. Odeslání zprávy aktoru A je tedy ekvivalentní uložení zprávy do S_j. Výše uvedená realizace připouští, aby ze schránky mohlo vybírat zprávy více procesů. Máme-li k dispozici schránky, můžeme dále zdokonalit proces posílání zpráv. Předpokládáme, že každá zpráva má následující strukturu:

a) ID odesílatele,
b) ID adresáta,
c) pomocný údaj *VP*,
d) identifikátor typu zprávy *Typ* (kladné číslo),
e) vlastní obsah zprávy.

Strukturu zprávy napíšeme zkráceně (i, j, *VP*, *Typ*, *obsah*). Každý aktor obsahuje pole T s identifikačními klíči schránek ostatních aktorů. Je-li odesílána zpráva s logickým číslem adresáta k, provede se $SEND(T_k, z)$. Tento obrat je v principu velmi jednoduchý, je však neobyčejně silným nástrojem pro ladění systému. Předpokládejme, že v systému existuje zvláštní aktor „Pošta“ se schránkou S_p. Zprávy můžeme posílat přímo adresátu k (a pak T_k obsahuje identifikaci schránky S_k) nebo T_k obsahuje identifikaci schránky pošty S_p a pak pošta, má-li to předepsáno, může předávané zprávy archivovat nebo je poslat jinému adresátu, než bylo původně určeno, např. vypsat na terminálu. Zprávy může zároveň archivovat. Odpověď na zprávu (i, j, V, *Typ*, *obsah*) má tvar (j, i, *VP*, $-$*Typ*, *obsah odpovědi*). Odpověď se odesílá přes schránky obdobně jako zpráva. Příjemce může rozpoznat z organizačních údajů, že se jedná o odpověď, a z *VP* může identifikovat zprávu, na kterou odpovídá. Pozná to ze záporné hodnoty $-$*Typ*. Tím jsme implementovali operace potřebné pro přenos zpráv. Výhody navrženého řešení:

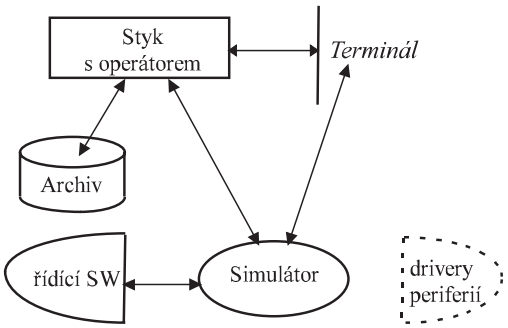
a) Lze snadno vytvořit takové prostředky, aby operátor u terminálu mohl simulovat dosud nenapsaný program. Stačí, aby zpráva byla vypisována na obrazovce místo zasílání původnímu adresátovi a aby na ni bylo možné z obrazovky odpovědět.
b) Aktory mohou být po dohodě o struktuře zpráv napsány různými týmy v různých programovacích jazycích.
c) S využitím archivace v Poště jsme celkem lehce získali výkonný prostředek pro ladění i kontrolu práce systému za provozu. Hledání chyb může začít analýzou archivu předávaných zpráv.

Zásadní nevýhodou výměny zpráv je to, že s výměnou zpráv musí „aplikace počítat“, musí být schopna výměny zpráv. To mnohdy neplatí, např. integrujeme-li do IS existující dříve napsanou aplikaci. Při realizaci softwarových agentů je možné odesílat nejen data, ale i programy a dokonce vlákna v rozpracovanén stavu. Tyto techniky jsou zatím ve stavu výzkumu.

11.2.4 Vývoj systému pracujícího v reálném čase.

Systém řízení v reálném čase je systém schopný odpovědět na podnět tj. uskutečnit výstupy do určitého času, zvaném doba odezvy, např. do 100 milisekund. To je obvyklá formulace jednoho ze základních požadavků na programy realizující přímé řízení procesů probíhajících v reálném světě. Systém je tvořen řízeným procesem a počítačem. Počítač řídí řízený objekt řídícími signály (povely) a je informován o stavu řízeného objektu stavovými signály (obr. 11.6). Tvůrci řídicího softwaru na počítači musí splnit následující požadavky:

1. Řídicí programy napsat a odladit před tím, než je realizován řízený proces, např. atomová elektrárna. Důvody:
   a) Po dokončení montáže řízeného objektu (technologie) nelze dlouho čekat na dokončení řídicích programů.
   b) Na již dokončeném systému nelze provádět rozsáhlé pokusy za účelem odladění řídicích programů – to je zdlouhavé, neekonomické a dokonce nebezpečné.



Obr. 11.7: Nahrazení řízeného systému simulačním programem.

2. Při poruchách za provozu je vhodné mít k dispozici prostředek, jak danou chybu simulovat na počítači, který není přímo napojen na řízený proces.

Z těchto požadavků plyne, že během vývoje je třeba řízený proces nahradit prototypem – simulátorem. Cílem je maximálně omezit změny v řídicích programech vyvolané nutností provádět simulaci. Ukažme, jak lze problémy realizace takových systémů řešit při použití techniky spolupracujících aplikací. Budeme postupovat takto: Části řídicích programů komunikujících s řízeným procesem soustředíme do specializovaných programů komunikujících s „výkonnými“ programy prostřednictvím zpráv. Postup rozdělení je na obr. 11.6. Operační systémy moderních počítačů mají prostředky styku s periferiemi podporující takové rozdělení. Pokud je systém dekomponován uvedeným způsobem, můžeme komunikaci se systémem nahradit komunikací s prototypem řízeného systému. Dostaneme situaci z obr. 11.7.

Jádro systému na obr. 11.7 je identické s jádrem systému z obr. 11.6. Při práci se simulátorem je jádro stejné jako pro práci s řízeným procesem. To lze využít i pro testování doby odezvy řídícího softwaru. Pro test doby odezvy musíme zřejmě:

a) určit dobu předání každé zprávy,
b) nějakým způsobem vyloučit dobu běhu simulátoru. V krajním případě může být simulátor nahrazen komunikací s obsluhou.

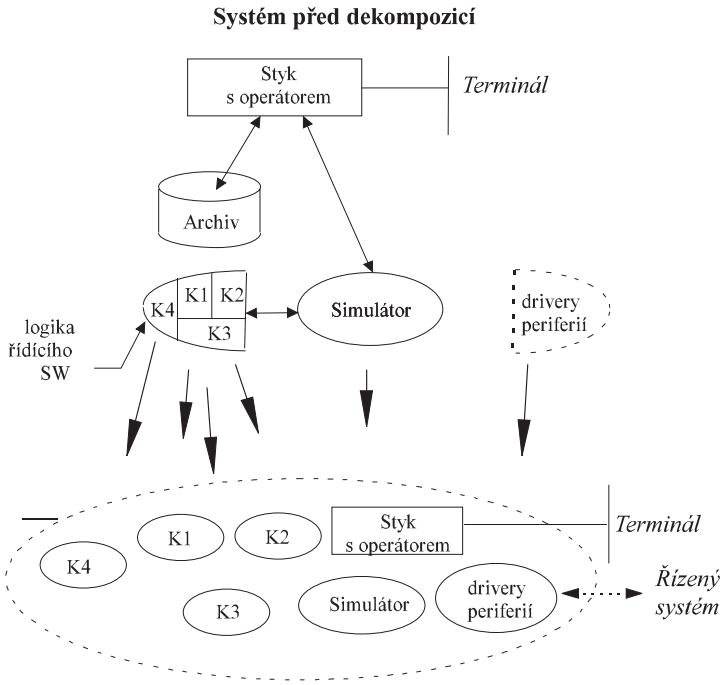
Řešení je následující: simulátor má z hlediska operačního systému nejvyšší prioritu: pokud může být spuštěn, je spuštěn, ostatní programy se pozastaví. Simulátor se spouští v těchto případech:

a) dostane-li zprávu,
b) systémový čas dosáhne hodnoty dané prvou událostí v kalendáři událostí (viz níže).

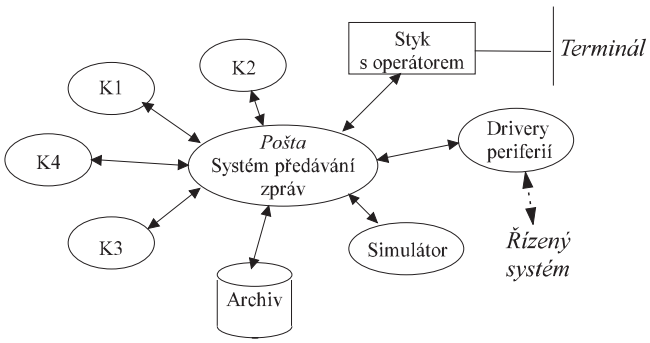
Při spuštění simulátoru se provedou následující akce:

a) zaznamená se čas systémových hodin do q;
b) převzaté i odesílané zprávy se archivují s časovým údajem $sc - cd$, kde sc je hodnota systémového času a cd celková doba běhu simulátoru;
c) pokud simulátor převzal a vyhodnotil všechny zprávy, sám se pozastaví. To se provede pomocí volání služeb operačního systému.

Z hodnoty systémového času uloženého v q a současné hodnoty systémového času se určí doba běhu simulátoru a o tuto hodnotu se zvětší celková doba cd běhu simulátoru. Po pozastavení simulátoru se spustí ostatní programy.



Dekomponovaný systém.
Komponenty spolupracují v principu každá s každou pomocí výměny zpráv nebo jinou formou API.



Realizace dekompozice s možností přesměrování zpráv.
Změnou adresáta lze provádět prototypování
Stejnou techniku lze využít při inkrementální realizaci.

Obr. 11.8: Postup dekompozice systému.

Vnitřní stavba simulátoru může být různá. Chování řízeného procesu můžeme definovat numerickým výpočtem, konverzací s obrazovkou apod. Simulátor je výhodné navrhnout s tzv. kalendářem, což je datová struktura K s položkami:

a) čas provedení,
b) identifikace akce,
c) parametry akce.

K je uspořádána podle času provedení. Simulátor po převzetí zprávy prochází kalendář a provede všechny akce s časem provedení ne větším, než je čas systémový, zmenšený o hodnotu cd. Akce mohou generovat zprávy i nové položky do kalendáře K. Tento obrat byl převzat ze simulačních jazyků. Pro účely oživování se tedy používají prostředky, které jsou nutné i pro cílový stav, což je jistě nejvýše žádoucí, neboť:

1. Archivace zpráv určených operátorovi systému je nutná i v cílovém stavu pro kontrolu práce operátora a vyhodnocování případných závad v práci systému.
2. Pokud vhodně navrhneme zobrazování zpráv na obrazovce, např. pomocí daty řízeného programu, můžeme pro řízení simulátoru použít aparát styku s operátorem.

Systém spolupráce mezi výkonnou logikou a simulátorem můžeme použít jako prostředek dekompozice výkonné logiky tak, jak je naznačeno na obr. 11.8. To umožňuje postupnou realizaci systému. Komunikace s dosud neimplementovanými komponentami může být totiž nahrazena např. komunikací s dispečerem. K podrobnému popisu diskutovaných metod realizace se vrátíme v kap. 21 v příkladu IS pružného výrobního systému.

11.2.5 Výhody a omezení dekompozice systému do sítě spolupracujících aplikací

Metoda spolupracujících aplikací zásadním způsobem závisí na prostředcích podpory interaktivní spolupráce více aplikací. Tento problém je snadno řešitelný pod operačními systémy podporujícími preemptivní multitasking a spolupráci se sítěmi. Takový operační systém podporuje „současný“ běh více aplikací/procesů a je zajištěno, že proces může být přerušen „zvnějšku“.

Tomuto požadavku vždy vyhovoval operační systém UNIX a některé firemní operační systémy, jako AS/400 nebo MVS. V novější době splňuje tato kritéria operační systém Windows NT. Na straně klientu lze využít i operační systémy, které tomuto kritériu nevyhovují, znamená to však jistá nepříliš bolestivá omezení. Pro extrémně krátké doby odezvy je někdy nutno použít specializované operační systémy.

Technologie spolupracujících aplikací tedy podporuje

- postupnou modernizaci IS záměnou některých aplikací jejich modernizovanými variantami.
- využití aplikací cizích výrobců,
- integraci existujících aplikací,
- postupný náběh systému (žádný velký třesk).

Spolupráce aplikací umožňuje řadu specifických implementačních obratů. Při vhodném využití (viz. předchozí paragraf) lze vytvořit systém nástrojů, které podstatně usnadňují detekci chyb při vývoji systému i při údržbě. Značné úspory při použití techniky spolupracujících aplikací plynou z té skutečnosti, že jednotlivé aplikace lze vyvíjet samostatně v menších týmech. V menších týmech je potřeba méně kontrol a dohledu. V kap. 15. je ukázáno, že rozdělení aplikace do sítě aplikací přináší podstatné úspory práce, i když jsou všechny komponenty psány znovu. Ještě větší úspory lze dosáhnout při údržbě a modernizaci a také tím, že lze využít existující komponenty. Rozdělení úkolu na více částí snižuje celkovou pracnost, neboť části jsou kratší a tudíž méně pracné (viz kap. 15.5.3). To však není jediná a ani nejvýznamnější úspora. Údržba softwaru je mnohem pracnější než vývoj. Dekomponovaný systém se snáze udržuje. Vlastnosti sítě spolupracujících aplikací podstatně usnadňují činnosti při údržbě a modernizaci

a přizpůsobování IS novým prostředím a novým požadavkům a samozřejmě celkovou modernizaci informačních technologií používaných zákazníkem (reingeneering).

Předchozí paragrafy ukazují, že metoda spolupracujících aplikací usnadňuje realizaci zcela nových technických obratů a pozitivně ovlivňuje funkčnost systému. Spolupracující aplikace mohou snadno využívat komunikaci v síti počítačů, včetně přístupu na rozlehlé sítě, jako je Internet. Je pak možné, aby např. prodejce komunikoval s podnikovým IS při uzavírání prodejní smlouvy přímo od zákazníka.

Realizace systému spolupracujících aplikací má i svá úskalí. Několikrát bylo zmíněno, že vyžaduje nový typ myšlení, který je odlišný od uvažování „v rámci jediného programu“. To se týká i objektově orientované metodologie. Výměna dat mezi aplikacemi, stejně jako podpora paralelní práce více aplikací, je podporována všemi moderními operačními systémy. Znamená ovšem značné zatížení výpočetních prostředků s možným zhoršením uživatelských vlastností. Tento problém s rostoucí výkonností hardwaru ztrácí postupně na významu.

Hlavní problém je samozřejmě věčný. Rozdělení do více aplikací je vhodné jen tehdy, existují-li relativně nezávislé skupiny funkcí. Tento předpoklad je u IS většinou splněn. Využití principů Internetu v podnikových sítích, známé jako Intranet, podstatně usnadňuje realizaci IS jako sítě spolupracujících aplikací. Doposud jsme mlčky předpokládali, že je systém dekomponován do komponent či aplikací staticky, tzn. je po svém dokončení tvořen právě těmi komponentami, které byly zvoleny či vyvinuty před instalací systému. Technologie softwarových agentů v principu umožňuje pracovat tak, že se jednotlivé agenty zapojují do spolupráce podle potřeby. Spolupráci aplikací lze přirovnat k metodě využívaní klasických knihoven podprogramů, využití agentů lze pak s jistou licencí přirovnat k práci s dynamicky připojovanými podprogramy (např. *.dll v systému Windows95).

11.3 Strukturované specifikace a návrh

Strukturovaná analýza a návrh je starší metodologií vývoje softwaru. V poslední době se stále více prosazují objektově orientovaná analýza a návrh popsané níže. O vztahu strukturovaných specifikací a objektově orientovaných technik se vedou diskuze. Dosti rozšířený názor tvrdí, že objektově orientované metody vytlačí metody strukturované. Tento názor není asi správný. Jsou oblasti, kde je strukturovaný pohled vhodnější. To asi platí pro některé části manažerských systémů a nejvyšší úrovně návrhu systému, pro návrh ve velkém, kde se používají strukturované techniky zobrazování sítě aplikací, modely dat atd. Je proto vhodné kombinovat oba přístupy podle potřeby.

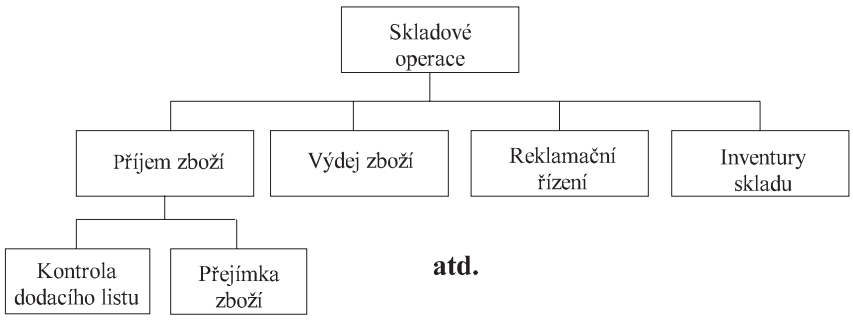
11.3.1 Strukturované specifikace a návrh jednotlivých aplikací

Podstatou strukturovaných specifikací a strukturovaného návrhu a psaní aplikací jsou tyto zásady:

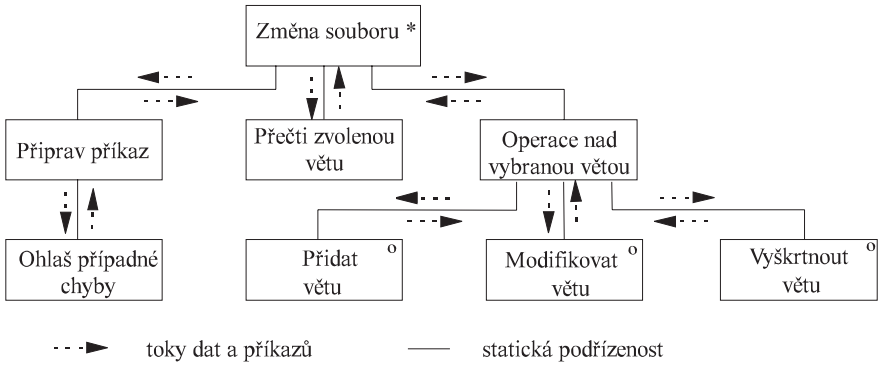
1. Aplikace je navržena hierarchicky jako soubor relativně nezávislých jednotek.
2. Celek se hierarchicky člení na úrovně. Každá úroveň obsahuje nevelký počet jednotek (modulů, programů), které jsou relativně nezávislé, mají úzké a srozumitelné rozhraní.
3. Pro pochopení funkce určité komponenty dané úrovně nejsou třeba znalosti o vnitřní struktuře ostatních komponent dané úrovně a komponent nižších úrovní.

   Dokumentace strukturovaných systémů obvykle obsahuje pro každou úroveň:

a) Popis úrovně jako celku.
b) Popisy funkcí jednotlivých prvků úrovně.
c) Popisy vazeb na ostatní úrovně (vstupy, výstupy, použití služeb jiných vrstev atd.).



Obr. 11.9: Příklad struktogramu hierarchické dekompozice aplikace.



Obr. 11.10: Vztah mezi hierarchickou dekompozicí a fungováním systému.

Hierarchická dekompozice se zobrazuje ve formě struktogramu (viz. obr. 11.9). Hierarchická dekompozice může být u jednodušších úloh využita i jako specifikace práce systému, jak je to vyjádřeno na obr. 11.10. Na obr. 11.10 jsou zobrazeny některé důležité charakteristiky toku řízení, např. fakt, že změna souboru se provádí opakovaně, že akce „přidat větu“, „modifikovat větu“ neprobíhají současně (tyto akce se pro danou větu vzájemně vylučují). Tyto skutečnosti jsou podle pravidel z (Jackson, 1983) zobrazovány následujícím způsobem:

1. Je-li na jedno provedení nadřízeného bloku daný blok proveden vícekrát, označíme to do jeho pravého horního rohu hvězdičkou.
2. Je-li pro daný blok proveden v každém okamžiku pouze jeden syn, poznačíme to kroužkem v pravém horním rohu všech synů.
3. Pokud nejsou podbloky označeny ani kroužkem ani hvězdičkou, provádí se v pořadí zleva napravo.

V souhlase s právě uvedenými pravidly se každá změna souboru z obr. 11.10 provede jako sekvence akcí: Připrav příkaz, Přečti zvolenou větu, Operace nad větou. Změna souboru může být provedena vícekrát. Akce Operace nad větou vyvolá v každém okamžiku právě jednu akci z výčtu: Přidat větu, Modifikovat větu, Vyškrtnout větu.

Reprezentace aplikace ve tvaru z obr. 11.10 je výhodná pro specifikaci jednoduchých dávkových programů a některých interaktivních systémů. Je méně vhodná pro návrh např. systémů řízení v reálném čase s komplikovanějšími vazbami mezi částmi systému – procesy. Reprezentace systémů tímto způsobem může být spojena s dalšími obtížemi, např. vyjádření existence obecně použitelných prostředků, složitější algoritmy. Přes tyto výhrady je použití reprezentace podle obr. 11.10 velmi výhodné u informačních systémů. Je však obvykle méně vhodné pro prvotní dekompozici velkého systému na jednotlivé části – samostatné aplikace, moduly, komponenty.

Použití struktogramů je výhodné při plánování a provádění vnitřních oponentur, a to i při používání objektově orientovaných technik popsaných níže. Strukturovaný návrh a specifikace vychází ze známého faktu, že člověk je schopen současně sledovat pouze nevelký počet objektů a jejich souvislostí. Uvádí se, že ne více než pět až sedm. Pochopení a kontrola strukturovaného návrhu jsou proto – při splnění právě uvedených podmínek – relativně snadné. Kontrola strukturované specifikace a návrhu se provádí tak, že se nejprve projde nejvyšší úroveň a pak postupně nižší úrovně. Při chybě se, počínaje od nejvyšší úrovně lokalizuje ten prvek systému, který může být zdrojem chyby, a v něm se postupuje obdobně. Tomuto způsobu kontroly se říká strukturované procházení.

11.3.2 Semistrukturovaný návrh

Strukturovaný návrh systému ve vrstvách používá jednotky uspořádané do jisté hierarchie. Takové uspořádání není vždy možné. (Parnas, Clemens, Weiss, 1985) navrhují dekompozici programů v poněkud jiné formě. Jednotkou dekompozice je u těchto autorů modul. Modul je prostředek strukturalizace celého systému. Modulární struktura systému definuje dekompozici systému do modulů a předpoklady, které mohou realizátoři modulů učinit o jiných modulech. Modul je obvykle tvořen z veřejně přístupných konstruktů. Pravidla použití modulů jsou založena na relaci „A vyžaduje přítomnost B“.

Moduly postihují statickou strukturu systému. Dynamické vlastnosti systému jsou zobrazeny procesy. Vazby mezi procesy nazveme procesní strukturou systému. Struktura procesů definuje způsob implementace jednotlivých aktivit z hlediska funkční dekompozice. Mezi procesy a moduly není jednoznačný vztah, jeden modul může realizovat více procesů, jeden proces může volat funkce více modulů. Části modulu mají k dispozici informace o datech všech programů v modulu. Data modulu jsou však přístupná jiným modulům pouze voláním vhodných podprogramů daného modulu. To umožňuje, aby byla realizace jiných modulů nezávislá na vnitřní struktuře daného modulu.

Dekompozice do modulů právě uvedeným způsobem není vždy snadná záležitost, poněvadž právě uvedený princip dekompozice mlčky předpokládá, že způsob spolupráce mezi moduly se nemění, že např. množina procedur, pomocí nichž nějaký modul komunikuje s jinými moduly, zůstává neměnná. To znamená, že navrhovatel systému dovede odhadnout pravděpodobnost změn. Takový odhad je založen na zkušenosti s podobnými systémy a na porozumění pro technologii hardwaru a softwaru.

Každý modul má být navržen dostatečně jednoduchý, aby byl snadno pochopitelný. Funkce modulu musí být zřejmá i bez znalosti vnitřní struktury modulu. Při studiu vnitřní struktury modulu má být snadné rozpoznat, které moduly jsou pro práci daného modulu významné. Při větších změnách a při počátečním návrhu musí být možné po stanovení pravidel pro rozhraní realizovat modul nezávisle na ostatních modulech.

V některých případech není možné omezit přímý přístup k informacím na programy v jediném modulu. Informace o hardwarových chybách musí být např. přístupné řadě modulů a tyto informace se mohou při změně hardwaru změnit natolik, že to ovlivní formu mezimodulární komunikace. Pak je třeba všechny tyto moduly změnit. Je však žádoucí, aby takových případů bylo co nejméně.

Moderní programovací jazyky usnadňují aplikaci výše uvedených principů. Platí to především pro jazyk Ada. Při objektově orientované technologii může být rozhraní modulu implementováno pomocí vhodných objektů. Modulární struktura v tomto pojetí je blízká architektuře spolupracujících aplikací.

11.3.3 Návrh systému ve vrstvách

Návrh systému ve vrstvách je vhodný především pro velké systémy, jako je operační systém. Jeho principy však lze uplatnit i u systémů vyvíjených jako více autonomních spolupracujících programů. Propagátorem tohoto postupu je Dijkstra (Dijkstra, 1976). Jedná se o techniku, která má úzký vztah k programování postupným zjemňováním. Principy metody jsou následující:

1. Celý systém se realizuje jako množina úrovní L_n až L_0, L_n je nejvyšší úroveň, tj. celý systém. L_n obsahuje operace systému použitelné externě.
2. Při programování úrovně L_i není známo nic o vlastnostech a dokonce ani o existenci úrovní L_{i+1} až L_n.
3. Na každé úrovni není nic známo o vnitřní struktuře ostatních úrovní, jsou známy „abstraktní operace“ – konstrukty nižších úrovní. Nějaká úroveň může obsahovat např. fyzické soubory, může je však skrývat za operace na „logické úrovni“. Úroveň L_{i+1} obvykle používá pouze konstrukty úrovně L_i.
4. Úrovně představují i jistý typ datových abstrakcí. Datovou abstrakcí míníme takový způsob práce s daty, který je definován pouze pomocí operací nad daty. Přístup k datům se tedy uskutečňuje pomocí volání podprogramů. Tento rys je typický i pro objektově orientované programy, kde místo podprogramů vystupují metody. Styk mezi úrovněmi je možný jen pomocí volání funkcí/metod. Globální data se používají zřídka.

Příklad: Úroveň L_0 obsahuje jádro operačního systému a základní operace synchronizace. Jen tato úroveň obsahuje informace o počtu procesorů. Úroveň L_1 provádí správu paměti (virtualizace). Úroveň L_2 řeší problémy vstupu a výstupu na fyzické úrovni atd. Poznamenejme, že moderní operační systémy obsahují úroveň L_{-1}, mikrojádro.

Metoda vrstev byla poprvé použita při realizaci operačních systémů. Je velmi vhodná pro objektově orientovaný návrh. Každá vrstva je pak tvořena skupinou tříd objektů. Členění systému na vrstvy je výhodné pro operační systémy, kde nejnověji tvoří nejnižší vrstvu, tzv. mikrojádro. Členění velkých systémů do vrstev v kombinaci s objektovou orientací podstatně přispělo ke zkvalitnění služeb moderních operačních systémů.

11.4 Objektově orientovaná analýza a návrh

Při specifikaci požadavků a při návrhu jednotlivých aplikací lze použít objektově orientovaný přístup. Tento přístup je mnohdy výhodné kombinovat se strukturovaným přístupem při návrhu architektury ve velkém – při dekompozici, volbě vrstev aj.. Objektově orientovaný přístup usnadňuje i dělení aplikací na menší jednotky, které se mohou chovat jako samostatné aplikace. Objektově orientovaný přístup ve smyslu (Rumbaugh, 1991), tak, jak jej známe z jazyků Smalltalk, C++ nebo Java, umožňuje propojení a bezpečnější využití výsledků specifikací požadavků při návrhu aplikace. Pokusme se nyní naznačit principy objektově orientovaného přístupu; podrobnější výklad lze nalézt v (Lorenz, 1995), (Rumbaugh, 1991), (Šešera, Mičovský, 1994), (Sochor, Richta, 1996).

11.4.1 Principy objektově orientovaného přístupu

V IS je mnoho činností podobných. Většina dokumentů při práci skladníka má záhlaví a výčet řádků. Jsou to dokumenty typu faktura, objednávka, dodací list, dobropis atd. Pro každý z dokumentů existuje řada podobných

nebo stejných operací, jako je založení dokumentu, vyplnění záhlaví, kde bude číslo zákazníka, datum vzniku, vazba na údaj zákazníkovi atd. Pro popis určitého typu dokumentu potřebujeme definovat data, která dokument obsahuje a operace, které se s daty provádějí.

Jednotlivé dokumenty nazveme objekty. Typ dokumentu, tj. jaké údaje obsahuje a jaké operace jsou pro něj definovány, je definován v tzv. třídě (class). Jednotlivé údaje se v objektově orientované terminologii nazývají atributy a operace se nazývají metody. Nové objekty, v našem případě jednotlivé obchodní dokumenty, se zavádějí instanciací tříd. Instanciace třídy může mít formu deklarace objektu.

Pro toho, kdo s objektem pracuje, je důležité vědět, jak se „volají“ metody objektu (volání metody se podobá volání podprogramu), tj. jaké je jméno metody a s jakými parametry se volá. Je důležité, že to, jak je metoda realizována, není „vidět“; dokonce je možné, že implementace metody může být do jisté míry oddělena od jejího rozhraní – od formy, jak je volána, viz kap. 11.2.

Intuitivním významem obdobné, ne však nutně stejné metody mohou mít pro různé třídy stejná jména, byť se změněným významem. Tato vlastnost se nazývá polymorfizmus.

Důležitou vlastností tříd je možnost definice nových tříd pomocí „dědění“. Pro výše uvedený příklad dokumentů můžeme nejprve vytvořit pomocnou třídu obsahující atributy a metody společné pro všechny dokumenty. Operací dědění se nejprve vytvoří nová třída – *potomek*, která má stejné atributy i stejné metody jako třída – *rodič*. Nově vzniklou třídu lze modifikovat tím, že definujeme nové nebo změníme zděděné metody nebo atributy. Takže např. z pomocné třídy „dokument se záhlavím“ vytvoříme třídu „faktura“, třídu „objednávka“ atd. Je možné i dědění od více rodičů – nová třída – *potomek* – zdědí atributy a metody více rodičů. Existuje i opačný postup zvaný abstrakce, při němž se k několika třídám vytváří rodič obsahující metody a atributy společné všem uvažovaným třídám. Metody instance třídy – objektu – mohou volat metody jiných objektů, které budou obvykle dostupné pomocí hodnot některých atributů daného objektu; tyto atributy mají obvykle hodnoty klíčů – identifikátorů objektů. Objekty proto mohou vytvářet komplikovanou síť připomínající síť vztahů mezi řádky tabulek v relačních databázích.

Objektově orientovaný přístup ulehčuje vytváření modifikací systému a opakované používání vytvořených tříd. Poněvadž má každý objekt vnitřní stav daný hodnotami atributů, je vhodné specifikovat, jak se objekt „vyvíjí“ v souvislosti se změnami jeho stavu. Je tedy žádoucí definovat scénář jeho „životního cyklu“.

11.4.2 Objektově orientované specifikace požadavků

Objektově orientovaný přístup lze velmi dobře použít už ve fázi specifikace požadavků, poněvadž na úrovni operativního řízení umožňuje dobře modelovat to, co se děje v reálném světě.[4]

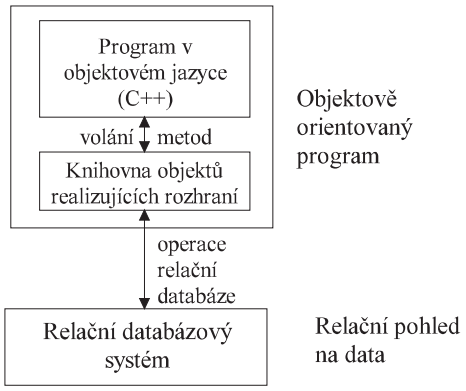
Doporučuje se však nepřecházet k objektově orientovanému popisu příliš brzy. Je vhodné vyjádřit požadavky v objektově orientované formě, která má tendenci být orientovaná spíše na detail, až v době, kdy je dostatečně jasná představa o souboru požadavků jako celku.

Formulace požadavků v objektově orientované formě se nazývá objektově orientovaná analýza (OOA). Výstupem OOA je objektově orientovaný popis systému, ve kterém se řeší co nejméně technických detailů, kde

4. Ve výše uvedeném příkladě je možné snadno stanovit:
   – jaké atributy má záhlaví dokumentu,
   – jaká data jsou v jednotlivých řádcích dokumentu,
   – jaké operace se s dokumentem provádí: vytvoření, oprava, potvrzení, zrušení,
   – jaký je životní cyklus dokumentu: jak vzniká, kde se mění, kdo změny provádí, jaké jsou vazby na jiné dokumenty či jiná data – např. na seznam zákazníků.

jsou však uvedeny všechny metody potřebné z hlediska uživatelů systému k provádění potřebných funkcí. I zde platí, že je vhodné co nejdéle zůstávat na úrovni modelování charakterizovatelné výrazem: „Popišme, jak by se to mělo dělat, zapomeňme při tom na chvíli, že něco z toho bude nakonec dělat počítač“.

Po formulaci požadavků v objektově orientované formě je třeba řešit technické problémy – navrhnout systém. I zde je možné a výhodné v případě, že vyvíjená aplikace má charakter operativního řízení, použít objektově orientovaný přístup – provést objektově orientovaný návrh (OOD – object oriented design). V OOD se doplňují detaily, jak systém implementovat. Používají se při tom operace dědění, modifikace metod a doplňování tříd. V našem příkladu dokumentů pravděpodobně bude zavedena třída reprezentující řádek dokumentu. Bude doplněna či zpřesněna implementace metod práce s jednotlivými řádky dokumentu. Zpřesní se rovněž definice grafického uživatelského rozhraní pro jednotlivé dokumenty.



Obr. 11.11: Objektově orientovaná nadstavba nad relační databází.

Stejný postup se uskuteční při přechodu od návrhu ke kódování v nějakém objektově orientovaném jazyce. Nejpřirozenějším způsobem realizace by bylo využití objektově orientované databáze. Objektově orientované databáze nejsou však zatím technicky zvládnuty. I v budoucnosti bude asi dlouho potřeba mít přístup k ohromným množstvím dat v klasických relačních databázích. Je tedy nutné vytvořit propojení mezi objektově orientovaným programem a daty v relační databázi. Možné řešení je uvedeno na obr. 11.11. V tomto případě lze využít knihovny tříd umožňujících přesunout data databáze do programu, např. C++, tak, aby práci s nimi bylo po jistou dobu možno chápat jako práci s příslušným objektem, který v jistém smyslu existuje i po skončení práce programu, je perzistentní.

11.4.3 Objektově orientovaný návrh a štěpení aplikací

Objektově orientovaný vývoj aplikace dává prostředky ke štěpení aplikace na části v architektuře klient – server (viz obr. 11.3). Jak bylo uvedeno výše, objekt je používán především prostřednictvím svých metod. To, jak jsou metody implementovány, je vázáno jedinou podmínkou – metoda pracuje tak, jak má. Je tedy možné, že při jedné implementaci pracuje metoda pouze s lokálními daty a při jiné je získává ze serveru.

Rozštěpení aplikace pak může být provedeno změnou implementace metod tak, aby nějaké objekty mohly být na klientu a jiné na serveru. Teoreticky vypadá takový postup jednoduše. V praxi je tato metoda úspěšná jen

tehdy, je-li, zhruba řečeno, rozštěpení provedeno na „úzkém místě“. To je tam, kde není nutné měnit implementaci velkého počtu objektů. U customizovaných systémů může být pro některé třídy k dispozici několik implementací. To, která implementace bude použita, závisí na tom, kde budou pracovat objekty dané třídy a objekty, jejichž metody daná třída používá.

Výhodou objektového přístupu je to, že změna implementace metod neovlivňuje podstatně implementaci tříd, které dané metody používají. To usnadňuje znovupoužitelnost tříd a usnadňuje rozdělení aplikace mezi klient a server. Implementace metod může být za jistých nepříliš obtížně splnitelných podmínek i v jazycích, které nejsou objektově orientované.

Každá aplikace může být členem sítě spolupracujících aplikací realizujících daný systém. Při vývoji systému se tedy postupuje v následujících krocích:

1. *Dekompozice ve velkém.* Systém se dekomponuje do aplikací – služeb, jejichž vnitřní struktura není známa, je známo pouze rozhraní aplikací, které se tedy chovají jako černé skříňky. Převažující typ spolupráce komponent je *asynchronní* – při vyžádání služby se nečeká na potvrzení, že se služba provedla. Základní implementační technika je výměna zpráv bez čekání na odpověď, případně spolupráce přes společnou databázi. Základní používaný grafický prostředek je diagram toků dat (kap. 12). Cílem dekompozice je zjednodušení vývoje, případně využití cizích nebo již existujících systémů. Používají se diagramy toků dat.
2. *Návrh a vývoj aplikací.* Každá aplikace se realizuje jako logicky jediný celek.
3. *Dekompozice v malém – štěpení komponent.* Jednotlivé aplikace se případně dekomponují na část, která bude pracovat na klientech, a na část, která bude pracovat na serveru. Aplikace se může v případě potřeby dekomponovat na více částí. Vnitřní struktura jednotlivých částí je známa (bílé skříňky) a aplikace se navrhuje jako celek. Převažující typ komunikace je *synchronní* – volající čeká na dokončení akce. Základní implementační technika je volání vzdálených procedur nebo metod. Používá se technika diagramů toků dat.

11.5 Případy použití (use cases)

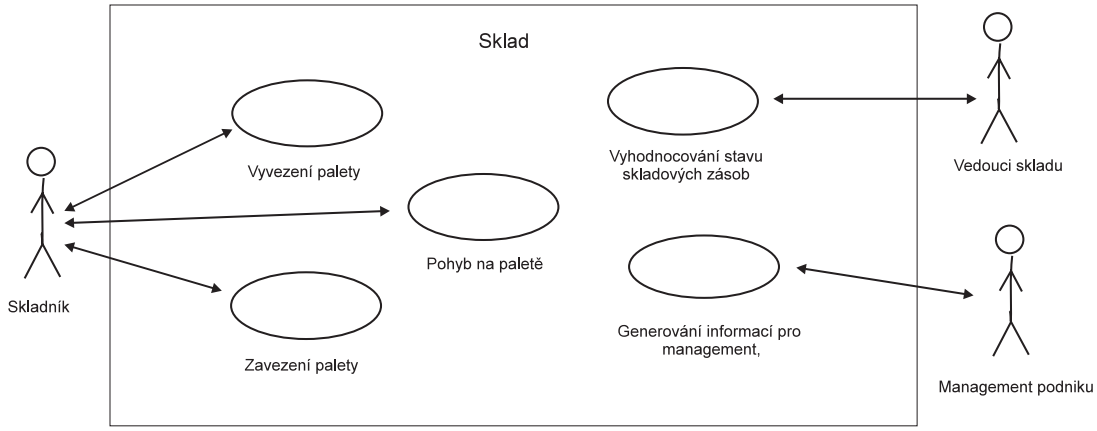
Jak bylo už několikrát uvedeno (srv. kap. 6), je nejvýše žádoucí vycházet při specifikaci požadavků z ucelených činností, jako je příjem zboží na sklad, provedení účetní operace na všech účtech, jichž se týká atp. Ucelené činnosti nazveme případy použití (use cases, UC).

V (Jacobson et al., 1995) je uvedena ucelená metodika jak postupně provést objektově orientovanou specifikaci a návrh systému tak, že se nejprve popíší jednotlivé případy použití a kdo tyto případy použití využívá. Situace se zobrazí ve formě z obr. 11.12. Účelem je zobrazit vazby mezi činnostmi navzájem a mezi obsluhou a činnostmi.

Při zpřesňování definice činností se jednotlivé činnosti definují jako soubor spolupracujících objektů. Objekty jsou trojí (kap. 12):

- *Objekty zajišťující rozhraní.* V našem příkladě objekt zajišťující grafické uživatelské rozhraní pro skladníka.
- *Objekty organizační.* V našem případě to může být objekt koordinující provádění jednotlivých kroků při přijímání zboží do skladu.
- *Objekty aplikační.* V našem případě např. objekty realizující jednotlivé kroky případu přijímání zboží. Krokem může být kontrola dodacího listu.

Po vymezení objektů se objekty rozdělí do skupin v závislosti na vzájemných vazbách, které mají být ve skupině široké, avšak mezi skupinami co nejužší, a na tom, kde je na síti nejvýhodnější ten který objekt umístit. Jednotlivé skupiny pak mohou tvořit samostatné moduly či programy. Spolupráce mezi skupinami objektů se zobrazuje



Obr. 11.12: Příklad grafických prostředků definice případů použití.

pomocí diagramů interakcí resp. diagramů výměny zpráv (podrobnosti viz kap. 12). V (Jakobson et al., 1995) je skupina objektů dále strukturována jako tzv. blok. V bloku jsou explicitně specifikovány objekty odpovědné za rozhraní mezi bloky a objekty výkonné. Vlastnosti objektů odpovědných za interakci se určují z diagramu interakcí.

Metodika UC je orientována na to, jak se bude činnost systému jevit obsluze. Prostředky postupné dekompozice systému a prostředky integrace produktů třetích stran nejsou dostatečně rozvinuty. UC jsou tedy vhodné spíše pro návrh monolitních aplikací, pro než není logická dekompozice nutná. To nevylučuje dekompozici monolitu na jednotlivá místa sítě, má-li pracovat distribuovaně. V případě potřeby je ovšem možné logickou dekompozici (např. v zájmy integrace existujících aplikací) provést, je k tomu ale nutné použít specifické nástroje mimo rámec metodiky UC.

Metodika případů použití je velmi úspěšná.[5] Je výsledkem více než dvacetileté zkušenosti autorů metodiky s vývojem systémů s prvky reálného času. Je proto silná především při specifikaci a návrhu operativních informačních systémů. Pro manažerskou část IS a obecně pro hromadné zpracování dat a jejich analýzu je dobře použitelná, avšak není již tak výhodná.

Při restrukturalizaci činností (BPR) se ještě před návrhem UC používají prostředky definující návaznost aktivit jednotlivých činností. Používají se při tom diagramy návazností aktivit (business thread process diagram, BTP). V BTP značí podle notace z CASE obdélník aktivitu, plná šipka povinnou návaznost, čárkovaná šipka nepovinnou návaznost aktivit. Neuzavřenou elipsou je možné označit čekání. Obdélníky nakreslené vedle sebe a uzavřené ve větším obdélníku se mohou provádět paralelně. Široké šipky kreslené obrysem značí podle kontextu buď událost (trigger) aktivující šipkou označenou aktivitu/proces nebo výsledek aktivity procesu. Je možné zadávat podmínky větvení a požadavky na stálé opakování aktivit. Ke každému prvku diagramu je možné připojit text definující význam prvku a jeho název a obsahující případně další informace. Pomocí BTP se nejprve zmapuje stávající stav. BTP se pak transformuje tak, aby definoval požadovanou strukturu činností po restrukturalizaci. Notace BTP

5. Výše zmíněná Jacobsonova kniha již vyšla v sedmi dotiscích s jednou revizí.

není ustálená. Zde uvedená notace pochází od autorů Hammera a Champyho z r. 1993. V návrhu systémů podle metodiky Jacobsona se používá diagram přechodů s prvky blízkými notaci BTP.

11.6 Softwarové vzory, sestavy a šablony

Pro určité typy úloh je výhodné používat osvědčené obraty a struktury – *vzory*.[6] Informační systémy se navrhují podle vzoru architektury klient – server nebo jako vícevrstvé struktury. Kompilátor je výhodné koncipovat podle vzoru lexikální analýza – syntaktická analýza – generátor kódu. Jádra moderních operačních systémů se koncipují podle vzoru jádro – mikrojádro. Uplatnění vzorů podstatně usnadňuje vývoj aplikací určitého typu. Softwarovými vzory se zabývá řada knih, např. (Buschmann et al., 1996). Podle této knihy je vzor soubor metod a pravidel jak definovat nějakou softwarovou entitu a soubor zásad, metod a postupů jak takovou entitu vytvořit.

Některé vzory definují architekturu a zásady dekompozice, jiné usnadňují postupné zpřesňování struktury a funkcí systému (návrh) a jiné se využívají při programování. Některé vzory jsou univerzální, např. techniky dekompozice do komponent a metody komunikace mezi komponentami, jiné jsou vhodné pouze pro určité aplikace (aplikační komponenty, viz výše uvedený příklad kompilátoru).

Vzory lze tedy rozdělit do tří kategorií

- Vzory architektury, např. spolupráce aplikací.
- Vzory návrhu, např. návrh spolupráce komponent pomocí roury v UNIXu.
- Idiomy neboli vzory programovacích obratů, obvykle v určitém programovacím jazyce.

Velmi schůdnou cestou použití vzorů jsou montážní sestavy (frameworks). Montážní sestava zajišťuje jistou skupinu funkcí. Při objektovém přístupu je sestava soubor tříd nebo objektů s rozhraním na metody objektů mimo sestavu odpovídajícím zásadám určitého vzoru. Integrace sestavy do systému se provede pouhým rozšířením systému o objekty skupiny.

Existují dva přístupy práce se sestavami. V prvém přístupu, který se zdá být výhodnější, se skupina chápe jako černá skříňka. Je možný i přístup, kdy je možné třídy skupiny dále modifikovat (bílá skříňka). To se ale vyplácí pouze tehdy, je-li nutno modifikovat existující skupinu pro mnohonásobné použití. Technologie Taligent rozeznává následující typy sestav:

- aplikační, sloužící jako služba více aplikacím, např. GUI,
- doménové, nabízející obecně použitelné funkce jisté oblasti, např. pohyby na účtech,
- podpůrné, které nabízejí rozhraní na základní software, jako jsou databáze a distribuované výpočty.

Šablona (template) je modifikovatelný idiom. Je to schéma výseku programu (můstek), jehož modifikací vznikají různé varianty určité funkce.

Různé typy montážních sestav (frameworks) jsou přehledně uvedeny v Communications of ACM z října 1997.[7]

6. Software patterns, frameworks, templates.
7. Někdy se framework chápe též ve smyslu stavebního celku, tj. jako rámecči schéma, do něhož lze vkládat sestavy a je při tom zaručen vznik funkčních aplikací.

12

Nástroje pro definici a vývoj softwaru

Moderní technologie vývoje a customizace IS se neobejde bez moderních nástrojů. Problém je v tom, že se tyto nástroje a metody velmi rychle vyvíjejí. Proto je třeba nástroje často modernizovat. Přes často proklamovanou univerzalitu nejsou jednotlivé nástroje použitelné v každé situaci. Proto v této kapitole uvádíme více variant vývojových nástrojů.

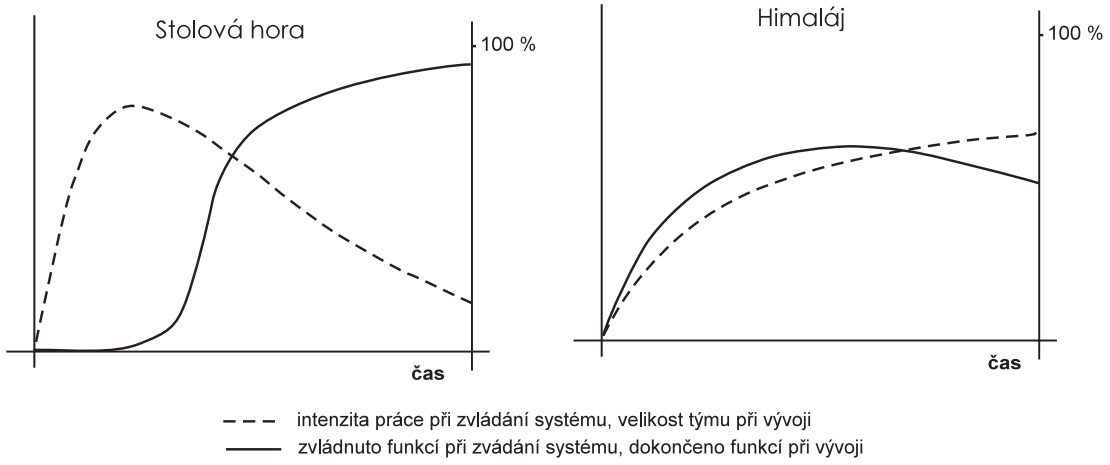
Koupě nástrojů a tím spíše jejich vývoj je nákladná záležitost. Další náklady jsou spojeny se zvládáním nově používaných prostředků. Obvykle uplyne dlouhá doba, než nástroje přinesou pozitivní efekty. To může znervózňovat a vést k nebezpečné tendenci „jít hned na věc“ jak ze strany managementu, to obzvláště, tak ze strany řešitelů.

12.1 Kolik investovat do nástrojů. Metody Himaláj a Stolová hora

Při návrhu architektury systému lze z hlediska harmonogramu prací postupovat v zásadě dvěma způsoby (obr. 12.1):

1. Poměrně dlouho vytváříme „pomocné prostředky“, jako jsou vývojové prostředky, prostředky pro ladění, různé dohlížecí programy, případně dedikované nebo nové programovací jazyky a makra. Do programování se neženeme bezhlavě, mnoho času spotřebujeme na přesné stanovení cílů, specifikace vstupů a výstupů, tvar dat, rozbor různých možností při volbě řešení a zkoumání logické konzistentnosti a úplnosti úlohy atd. Problém je v tom, že „dlouho není nic vidět“. Napřed se diskutuje a sepisuje, nejsou ale napsány žádné programy. Když už se programuje, tak to opět dlouho nejsou „užitečné“ programy, ale takové, které samy o sobě nejsou k ničemu. Když už se píší „užitečné“ programy, pak dlouho nepracují, protože jsou chyby v nich i v pomocných programech. Opravdu velké systémy ale v podstatě nelze jiným způsobem realizovat.
2. Můžeme však také postupovat metodou dávající rychle prvé výsledky, brzy „něco funguje“. Postupujeme tedy tak, že se okamžitě naprogramují bez pomocných prostředků jasné věci s tím, že se pak uvidí, jak „to dodělat“. Brzy máme výsledky a šéfové jsou spokojeni.

Později se ale často zjistí, že úplná realizace je velmi pracná a že zvolený postup je dosti drahý, poněvadž se musí leccos předělávat, počínaje specifikacemi požadavků přes návrh až k hotovým programům. To se samozřejmě může stát vždy, při právě uvedené metodě je to ale obvyklé. Místo dokončení za tři neděle („většina systému už chodí“) pak třeba nejsme hotovi ani za rok.



Obr. 12.1: Porovnání metody Himaláj a metody Stolová hora.

c	1/2-1/(2c)	max($Q(m)/n$)	zvýšení %
2	0.25	9/8	12.5
3	0.33	4/3	33.3
4	0.37	25/16	56.2
6	0.42	49/27	104.2
8	0.44	81/32	153.2

Tab. 12.1: Efekty zvýšení výkonu při použití nástrojů.

.

Pokud postupujeme správně, dochází při prvém způsobu práce v jistém okamžiku k rychlému náběhu celého systému a výsledný produkt mívá málo chyb. Při druhém způsobu nebýváme hotovi nikdy. Situace je znázorněna na obrázku obr. 12.1. V souladu s tvarem funkcí z obr. 12.1 nazýváme první metodu metodou Stolové hory – ke kopci po rovině, prudce do kopce a pak opět po rovině, druhou metodu pak metodou Himaláj – dost brzy se začne stoupat, ale vrchol je stále velmi daleko.

V praxi musíme někdy postupovat zlatou střední cestou, řekněme metodou Krkonoše, poněvadž se nám nemusí dařit navrhnout všechno hned na začátku správně, něco musíme realizovat až později, občas se mýlíme atd. Tvorba pomocných nástrojů přináší značnou úsporu prací, oddaluje však dobu, kdy budou k dispozici „užitečné“ výstupy. V řadě oblastí, jako jsou kompilátory, expertní systémy atd., je výhodné koncipovat programy jako tzv. daty řízené systémy, ve kterých je algoritmus do značné míry definován daty (tabulkami), nad kterými pracuje interpretační program. Tím lze dosáhnout značné obecnosti. I v tomto případě však jsou „užitečné“ výsledky k dispozici poměrně pozdě.

Všimněme si ve shodě s (Levy, 1987) podrobněji vlivu softwarových nástrojů. Předpokládejme, že máme k dispozici kapacitu n člověkoměsíců. Z n člověkoměsíců můžeme věnovat m člověkoměsíců na vývoj nebo

na zvládnutí kupovaných softwarových nástrojů. Použití těchto nástrojů zvýší ve zbytku času produktivitu práce $f(m)$-krát, takže objem „užitečných“ prací bude:

$$Q(m) = (n - m) \cdot f(m) \tag{12.1}$$

O $f(m)$ budeme předpokládat, že je to rostoucí funkce a že $f(0) = 1$, tj. žádné nástroje – žádná změna. $Q(m)$ nabývá maxima v bodě m, pro který je derivace $Q'(m) = 0$. Pro maximum funkce Q proto platí

$$0 = -f(m) + n \cdot f'(m) - m \cdot f'(m) \tag{12.2}$$

Uvažujeme případ $f(m) = 1 + c \cdot m/n$. Po dosazení do rovnice 12.1 dostaneme:

$$-1 - c \cdot m/n + (n - m) \cdot c/n = 0 \tag{12.3}$$

Hodnota v maximu má být větší než n, tzn. než je výkon bez použití nástrojů, takže c by mělo být větší než 1. Pak z rovnice 12.3 dostaneme:

$$m/n = 1/2 - 1/(2c) \tag{12.4}$$

V tabulce 12.1 jsou uvedeny hodnoty m/n, pro které pro dané c nabývá funkce $Q(m)$ maximální hodnoty. Snadno lze ověřit, že $f(m) > 1$ pro $m/n < 2 - 1/c$. Na základě hodnot uvedených v tab. 12.1 můžeme učinit následující závěry:

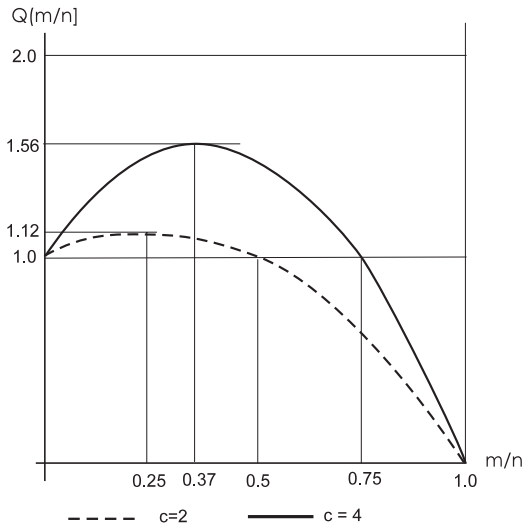
1. Na vývoj nových nebo na zvládnutí kupovaných nástrojů je vhodné věnovat téměř polovinu pracovní kapacity.
2. Znatelný přínos pro daný projekt přinese nový nástroj jen tehdy, zvýší-li produktivitu práce alespoň třikrát. Této podmínce vyhovuje např. specializovaný kompilátor.
3. Přínos nového nástroje se ovšem většinou neomezuje pouze na daný projekt. Přínosy v dalších projektech mohou být značné. Příkladem správnosti této úvahy je jazyk C.

Doba zvládnutí kupovaného nebo doba vývoje nového nástroje je dána vlastností nástroje samotného. To znamená, že m/n nebude přesně vyhovovat vztahu 12.4, takže skutečný efekt bude pak o něco menší, než je uvedeno v tabulce 12.1.

Funkci $f(m)$ jsme zvolili poněkud spekulativně. K obdobným výsledkům dospějeme i pro jiné volby tvaru funkce $f(m)$. Naše úvahy platí pro libovolně velká n. Při velkých n je však množství práce, které můžeme věnovat buď vývoji nebo zvládnutí nástrojů větší, lze tedy vyvinou dokonalejší nástroje. Hodnota c může proto být větší. Pro velká n je možné realizovat i nástroje vyžadující velké množství práce (např. kompilátor) a tedy nástroje účinnější. Naše výsledky však můžeme použít následujícím způsobem. Nechť nějaký nástroj vyžaduje pro využívání (vývoj/zvládnutí) m člověkoměsíců a přinese c-násobné zvýšení produktivity, $c > 2$. Pokud je $m/n < 1/2 - 1/(2c)$, je možné nástroj uplatnit. Je-li $m/n << 1/2 - 1/(2c)$, lze uvažovat o použití dalšího nástroje. Při použití dalšího nástroje se m zvětší na m_1 a c na c_1. Nový nástroj přinese zlepšení, je-li $m/n < 1/2 - 1/(2c_1)$.

Výsledek našich úvah je jednoduchý. Pro větší projekty je výhodné věnovat vývoji a zvládání nástrojů podstatnou část kapacit. Místo vývoje lze nástroje kupovat a kapacity věnovat tomu, abychom se je naučili používat (školení, zkoušení). U kupovaných nástrojů je nutno n snížit, poněvadž část prostředků musí být investována do nákupu nástrojů.

Zavádění systému lze rovněž realizovat metodou Himaláj nebo metodou Stolová hora (viz kap. 13). Systém je možné začít používat po krátkém zaškolení s využitím grafického rozhraní a nápovědy. Uživatelé se neseznamují se podstatnými skutečnostmi a celkovou strukturou systému. Výsledkem je nevyužívání všech možností systémů.



Obr. 12.2: Graf funkce $Q(m/n)/n$ pro různá c.

Důkladnější školení a vzdělávání přináší podstatné efekty. Školení a vzdělávání odpovídá zvládání nových metodik a nástrojů při vývoji systému, avšak zprvu nepřináší žádný efekt. Výše uvedený model naznačuje, že by se těmto „neproduktivním“ činnostem měla věnovat podstatná část kapacit uživatelů před zahájením provozu systému a během „záběhu“.

12.2 Návrh datových struktur, ER-diagramy

Volba datových struktur a metod přístupu k nim je ve většině aplikací zásadně důležitá. Důvodem je skutečnost, že volba dat a metody práce s daty jsou rozhodující pro snadnou modifikovatelnost a přenositelnost programů. I když problém implementace datového zabezpečení zařazujeme do etapy návrhu, hlavní funkce datového zabezpečení musí být řešeny v předchozích etapách. Některé aspekty volby struktury dat zasahují až do oblasti celkové dlouhodobé koncepce zpracování informací v dané organizaci (volba dlouhodobé strategie budování IS, Strategic Information Planning).

Data vytvářená často v průběhu mnoha let bývají používána mnoha programy a systémy. Často nelze předem rozhodnout, jaké operace s daty budou prováděny a k čemu všemu budou data používána. Struktury dat, se kterými program pracuje, mohou být – podobně jako samotné programy – během života IS modifikovány. Data jsou důležitou a stále významnější částí majetku podniku či organizace. Za této situace je nutné:

- navrhnout datové struktury tak, aby umožňovaly dlouhodobé cíle rozvoje podniku a podporu jeho strategických záměrů,
- navrhnout datové struktury a metody práce s nimi tak, aby při rozšíření dat o nové položky nebyla nutná častá a bolestná rekonstrukce datové základny,
- software koncipovat tak, aby změna datové základny nevyvolala změny programu,

- systém práce s daty navrhnout tak, aby mohly být snadno navrženy a realizovány další systémy.
- pořizovat i taková data, která nejsou bezprostředně použitelná, avšak jejichž sběr není drahý, a jsou vážné důvody předpokládat jejich smysluplné využití v budoucnu.

Data a metody práce s nimi mohou podstatně ovlivnit i vlastní formulaci požadavků a dekompozici systému. Optimální postup při navrhování systémů zpracování dat je tedy taková postupná (inkrementální) realizace, při které se data a systém práce s nimi navrhují tak, aby vytvářela datové prostředí, ve kterém bude realizováno mnoho IS.

Výhodné je užití obecného databázového systému. Pokud však nepostupujeme opatrně a používáme v programech databázové konstrukce přímo, může vzniknout situace, kdy budeme na daném databázovém systému tak závislí, že nebudeme moci přejít na modernější, ale nekompatibilní systém. Moderní databázové systémy splňují výše uvedené podmínky tím, že umožňují, aby mnohé práce s daty byly realizovány v jazyce SQL. Při návrhu datových struktur je potřeba především zkoumat ty vlastnosti dat, které plynou z podstaty řešeného systému, z jeho přirozených, „vrozených vlastností". Jsou to takové vztahy, jako pravidlo, že továrna má více oddělení, oddělení více zaměstnanců, zaměstnanec u dané organizace jeden plat (snad) a je členem jediného oddělení. Stroj může mít mnoho součástí a každá součást jediný popis, atd.

Užitečné jsou prostředky automatické generace grafického zobrazování vztahů mezi datovými objekty. Tyto grafické prostředky by měly vyjadřovat především vrozené vlastnosti dat nezávisle na způsobu, jak jsou data uložena v počítači, jak jsou používána a jaké jsou mechanizmy přístupu k datům. Necitlivost programů ke změnám dat lze dosáhnout následujícími obraty (Martin, McClure, 1983):

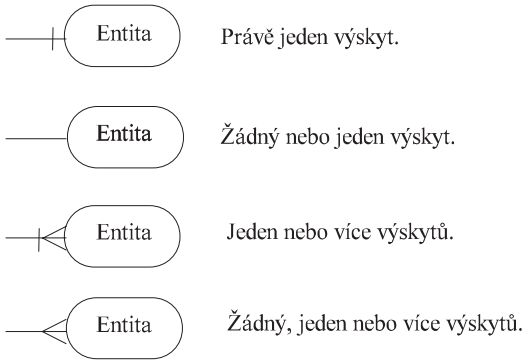
- Program P pracuje pouze s těmi atributy datových struktur, které potřebuje v tom smyslu, že přidání dalšího atributu *p* do datové struktury nevyžaduje změnu programu P, pokud program P s atributem *p* nepracuje.
- Pokud je nutné vytvořit nové atributy (např. sekundární klíče), neovlivní to programy, které je nepotřebují.
- Program je schopen zobrazit (zjistit) všechny vztahy mezi daty (hierarchii, vzájemné odkazy), se kterými pracuje.
- Deklarace datové základny jsou do programů generovány automaticky z knihoven. Totéž platí pro systémové konstanty. Tento požadavek splňují 4GL jazyky, které však zatím nejsou dostatečně standardizovány a jsou závislé na konkrétní databázi.
- Části programu závislé na struktuře databáze by měly být generovatelné na žádost přímo z definice dat.

Při návrhu obsahu a struktury dat je třeba předvídat, kdo všechno by mohl systém používat v budoucnu. Na základě zjištěných faktů je nutné vybrat systém prezentace dat. Je třeba postupovat uváženě – volba datového modelu může podstatným způsobem ovlivnit naše budoucí možnosti.

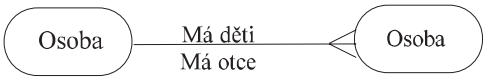
Vývojová prostředí moderních databázových systémů splňují většinu výše uvedených požadavků. Pro každou úroveň řízení se určí požadavky na informace a objekty, o nichž je potřeba uchovávat data. Vytvoří se seznam entit a vztahů mezi entitami, který bude základem definice databáze.

U uživatelů se ověří, zda lze pořizovat požadovaná data a zda data zabezpečují žádané funkce. Je třeba navrhnout, jak pracovat s existujícími daty. Budou souběžně používána, nebo se převedou do nového schématu?

Pro zobrazování vztahů mezi daty se používají diagramy entit a relací (ER-diagramy). Entita je v jednoduchém případě skupina (entice, vektor) hodnot – atributů. Množina takových entic se dnes nejčastěji implementuje jako tabulka a entita je „řádkem tabulky". Ve zbytku toho paragrafu budeme předpokládat, že entita odpovídá řádku tabulky a že pracujeme s databázovým systémem pracujícím s entitami uvedeného typu – s relační databází (podrobnosti viz (Pokorný, 1994, 1992)).



Obr. 12.3: Grafické prvky ER-diagramů. Při zachování podstaty se grafická forma ER-diagramů různých autorů liší, především při označování násobnosti – kardinality, s níž entita vstupuje do relace.



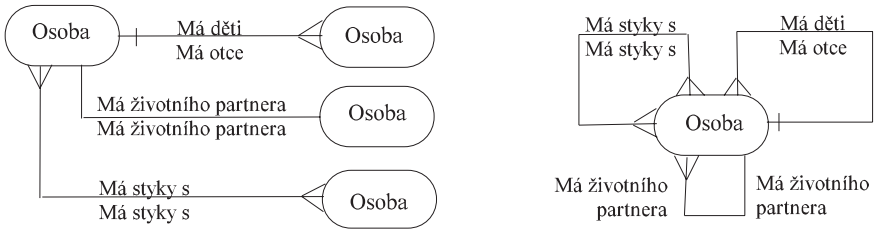
Obr. 12.4: Zobrazení vztahu mezi mužem a jeho dětmi. Muž nemusí mít žádné dítě, každé dítě má právě jednoho biologického otce.

Relace je vztah mezi entitami, např. mezi entitou Osoba ve významu Muž a entitou Osoba ve významu Dítě. Každá entita vstupuje do relace s jistou násobností. V našem příkladě má každé dítě právě jednoho biologického otce a muž může mít žádné, jedno nebo více dětí. Násobnost se vyjadřuje dohodnutými značkami. Tvar značek není dosud dostatečně standardizován. Příklad notace je uveden na obr. 12.3, vztah muže a jeho dětí je zobrazen na obr. 12.4.

Při definici dat je nutné analyzovat, zda údaje nejsou duplicitní, a duplicity až na dobře zdůvodněné výjimky odstranit. Výjimky mohou existovat při distribuovaném zpracování. Je dobré zavést vhodné standardy v pojmenovávání položek. U relačních databází je žádoucí převedení dat do normální formy (Pokorný, 1992).

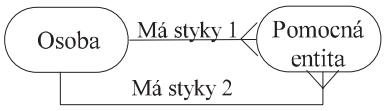
Definici dat by měli oponovat uživatelé. Uživatelé by měli sami provést inspekci návrhu. Zároveň se provede diskuze budoucího vývoje datového zabezpečení. Podle výsledků je nutné návrh upravit. Vývoj datové báze lze urychlit využitím moderních vývojových prostředků (CASE, vývojová prostředí). V průběhu návrhu je třeba uvážit možnosti hardwaru a různé normalizační operace definice dat a optimalizace. Na obr. 12.4 je definována relace „muž má děti“. Jeden muž může mít jedno i více dětí, také žádné dítě mít nemusí. V tom případě říkáme také, že jde o vztah typu 1:*n*. Všechny možnosti, které obecně přicházejí v úvahu na jedné straně relace, jsou: žádný nebo jeden výskyt, právě jeden výskyt, alespoň jeden výskyt, libovolný počet výskytů. Graficky jsou entity zobrazovány obdélníky se jménem typu entity, relace spojnicemi ukončenými značkami pro výše uvedené násobnosti. U spojnice se uvádí jméno relace.

Složitější příklad ER-diagramu na obr. 12.5. Varianta nalevo je používána v počáteční fázi datového modelování. Pro definici databázových tabulek je vhodné označení stejných entit ztotožnit, čímž dostaneme formu napravo.



Obr. 12.5: Příklad složitějšího ER-diagramu. Relace je typu *m*:*n*, mají-li obě strany relace vyšší násobnost než jedna (zde *má životního partnera*). Relace typu *m*:*n* není v relačních databázích přímo implementovatelná a pro databáze musí být transformována do tvaru z obr. 12.6.

.



Obr. 12.6: Odstranění relace typu *m*:*n*.

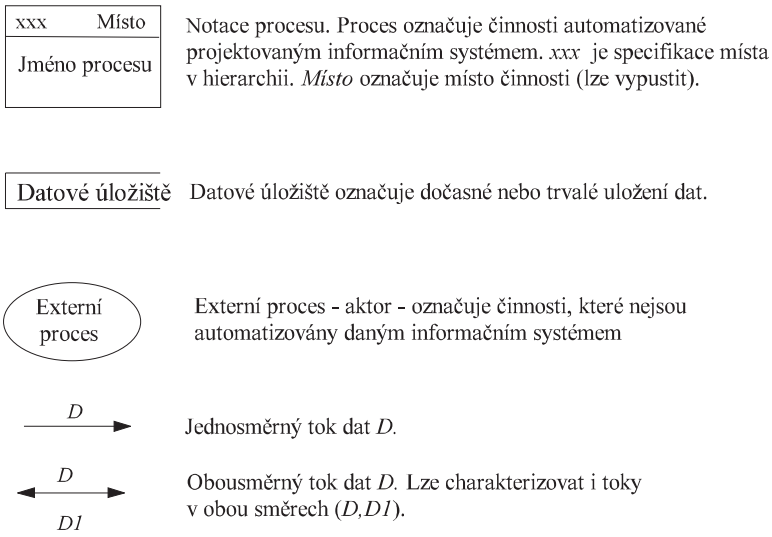
Po popisu dat na konceptuální úrovni je nutné data zobrazit těmi prostředky práce s daty (databázové systémy, systémy práce se soubory), které jsou na daném systému k dispozici. Je např. nutné doplnit atributy umožňující vyjádřit existenci relací pomocí klíčů identifikujících jednotlivé entity a nahradit relace typu *m*:*n* jinými relacemi.

Obr. 12.7: Chenova notace. Místo *n* může být konkrétní celé číslo, výčet nebo interval, např. 2:*n* značí minimálně dva výskyty, 3:4 tři nebo čtyři výskyty.

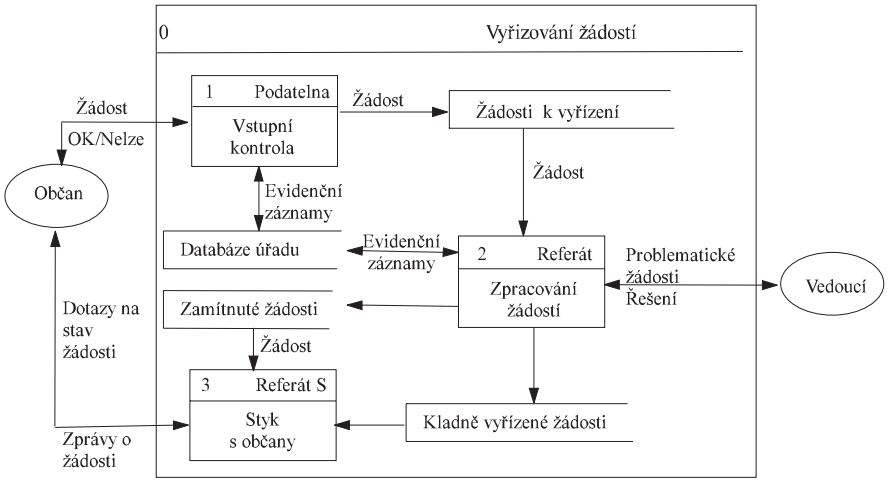
Při definici dat je často nutné provést úpravy z důvodů efektivnosti a menšího rizika ztráty, resp. porušení dat. Jsou to transformace odstranění *m*:*n* relací (obr. 12.6) a normalizace (Pokorný, 1991). Forma Chenova používaná např. v systému CASE – Cadre (obr. 12.7) má větší výrazové schopnosti při zobrazování složitějších vztahů mezi daty.

12.3 Diagramy toků dat

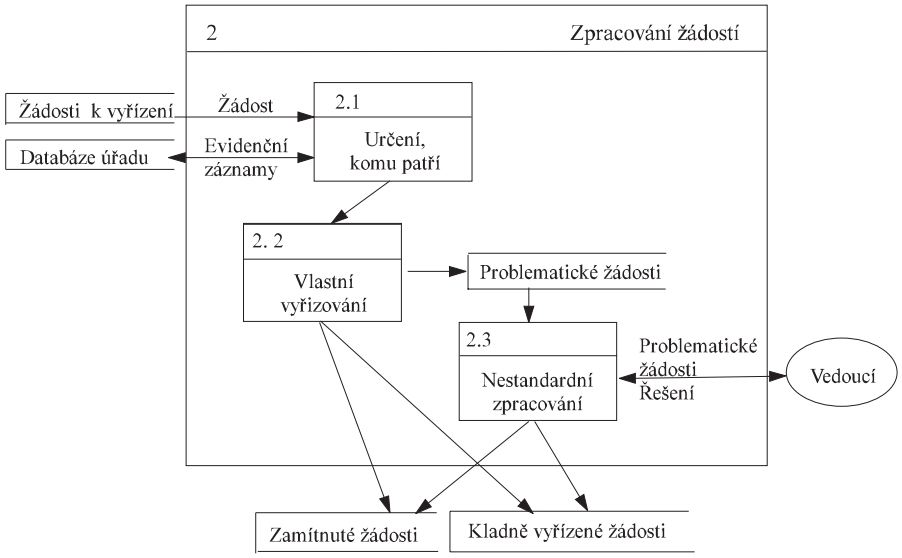
Základním grafickým prostředkem zobrazení struktury projektu jsou diagramy toků dat (data flow diagram DFD). Diagram toků dat je síť tvořená následujícími prvky:



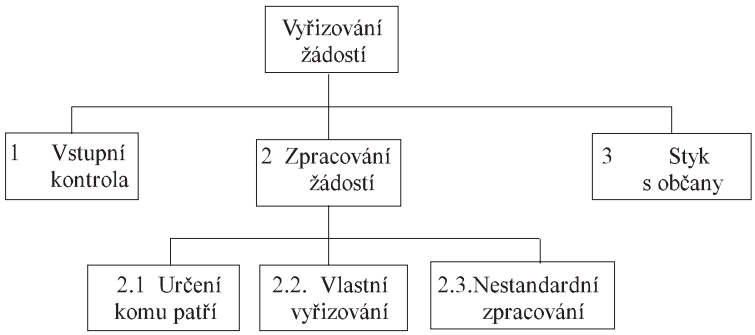
Obr. 12.8: Grafické prvky používané při definování diagramů toků dat (notace SSADM).



Obr. 12.9: Diagram toků dat systému Vyřizování žádostí.



Obr. 12.10: Dekompozice procesu Zpracování žádostí. Kontext diagramu musí odpovídat kontextu procesu Zpracování žádostí v diagramu Vyřizování žádostí.



Obr. 12.11: Hierarchie vytvořená postupnou dekompozicí systému Zpracování žádostí.

- Aktivity (procesy) – aktivity systému (značeny obdélníky).
- Datová úložiště (místa pro umístění dat, polootevřené obdélníky).
- Externí procesy – činnosti (aktivity mimo uvažovaný systém, elipsy).
- Datové toky mezi objekty předchozích typů. Příklad (velmi zjednodušeného) diagramu toků dat je na obr. 12.9.

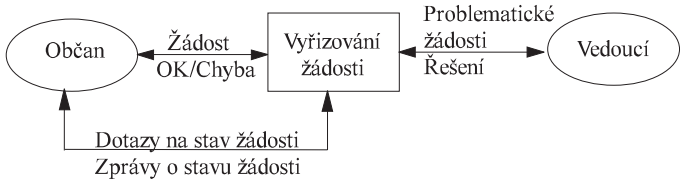
Jednotlivé aktivity mohou být dekomponovány do podřízené sítě dílčích aktivit. V našem příkladu lze dekomponovat proces Zpracování žádostí do diagramu na obr. 12.10. Strukturu dekompozice je možno zobrazit i ve formě struktogramu (obr. 12.11). Při návrhu diagramů toků dat se řídíme následujícími zásadami:

1. Spolupracují-li procesy interaktivně, je datový tok mezi nimi přímý, jinak se musí uskutečňovat přes datové úložiště.
2. Spojení s externími objekty může mít pouze proces, nikoliv datové úložiště.

Diagramy toků dat jsou velmi účinným nástrojem návrhu IS. Jsou intuitivně zřejmé a lze je použít při jednáních se zákazníky. Nevýhodou je nedostatečně formalizovaná forma a to, že v diagramech toků tak nelze jednoduše zobrazit některé požadavky souběhu (např. to, že data musí být k dispozici na více datových úložištích současně). Diagramy toků dat lze využít jako centrální část dokumentace, která pro každý prvek diagramu specifikuje důležité skutečnosti (např. u datových úložišť specifikuje buď přímo strukturu dat nebo odkaz na její definici). Procesy mohou být podle okolností samostatné aplikace nebo subprocesy jedné aplikace. Diagramy toků dat jsou součástí prakticky všech CASE systémů. CASE systémy jsou soubory nástrojů podporující vývoj softwaru, kap. 19) a používají se především jako prostředek celkového návrhu systému. Osvědčují se i jako prostředek komunikace se zákazníkem a to i v případě, že jsou vytvářeny bez CASE nástrojů. Notace jednotlivých CASE nástrojů se může lišit, principy však zůstávají stejné.

12.3.1 Postup vytváření diagramů toků dat

Vytváření diagramů toků dat (DFD) je intuitivně srozumitelný popis skutečnosti a může být proto založen převážně na intuitivním přístupu. To však vyžaduje poměrně vysokou úroveň schopností a velký rozsah zkušeností u řešitelů. Obtížnost a rizika úkolu vytvoření DFD lze zmenšit tím, že se celý proces vytváření DFD rozloží do následujících etap:



Obr. 12.12: Diagram kontextu systému Vyřizování žádostí.

1. Identifikace klíčových informačních toků a aktivit.
   Na základě rozboru činností, např. spolupráce se zákazníkem od vzniku objednávky až po její úplné vyřízení, se vystopují důležité informační toky mezi jednotlivými „aktory“, většinou organizačními jednotkami uživatele. Aktory vystupují jako zdroje nebo příjemci informací. Informace mívají tvar dokumentů, jako jsou objednávky, finanční operace, výrobní příkazy. Proto často návrh datových toků vychází z analýzy oběhu dokumentů. Vhodnou pomůckou jsou scénáře činnosti jednotlivých aktorů, které se používají při zpřesňování DFD.
2. Vytvoření prvotního DFD.
   Skutečnosti zjištěné v bodě 1. je vhodné zobrazit graficky. Aktory (obvykle jednotlivci nebo organizace mimo organizaci zákazníka i organizační jednotky zákazníka např. sklad, účtárna) jsou propojeny datovými toky v souhlase se skutečnostmi zjištěnými při analýze v bodě 1. Aktory mohou být zatím zobrazeny symboly vyhrazenými pro externí objekty.
3. Stanovení hranic realizovaného systému.
   V této etapě se rozhoduje, které činnosti kterých aktorů budou pokryty navrhovaným IS. Tyto činnosti se podrobně analyzují, uvažují se však jen ty, které mají vztah ke zpracování dat – jsou zdrojem dat nebo

jejich zpracovateli – a mohou být zefektivněny. Je vhodné uvažovat různé alternativy při stanovování hranic systému a rozhodování o tom, které informační toky je třeba v návrhu řešení uvažovat. Ty aktory, které jsou součástí systému, jsou v DFD nahrazeny procesy. Současně se vytvoří diagram kontextu, tj. diagram, ve kterém jsou uvedeny pouze externí objekty a to, co je plánováno automatizovat – všechny procesy jsou nahrazeny jediným procesem (obr. 12.12). Diagram kontextu by měl schválit zákazník. Je vhodné, aby posuzoval i ostatní diagramy toků dat.

4. Určení procesů a datových úložišť.
   a) Každý aktor uvnitř systému musí být nahrazen jedním nebo více procesy. Pokud je procesů mnoho (více než 10), je vhodné některé procesy sdružit do jednoho procesu a ten pak dekomponovat výše naznačeným způsobem.
   b) Každá informace vstupující do systému a vystupující ze systému musí být přijímána/odesílána nějakým procesem.
   c) Každému procesu se přiřadí jednoznačné identifikační číslo, „místo“[1] a jméno obsahující sloveso/slovesné podstatné jméno specifikující činnost procesu.
   d) Specifikace datových typů. Toky informací jsou nahrazeny označeními datových toků. Datový tok spojuje přímo procesy, je-li spolupráce procesu „on-line“, tj. není nutné data ukládat pro pozdější zpracování. V opačném případě, u běžných IS je to častá situace, musí být k dispozici datové úložiště (DÚ), do kterého se data ukládají pro pozdější využití. Datový tok je pak směřován do vhodného datového úložiště.

   Z datového úložiště jsou pak doplněny datové toky do procesů – příjemců. Datová úložiště se jednoznačně pojmenují, jméno by mělo charakterizovat obsah úložiště, očíslují a přiřadí se jim vhodný typ:

   Typ D – stálá data průběžně používaná pro každodenní provoz a údržbu systému (např. číselníky) a historická data pro potřeby analýzy.

   Typ T – dočasná (temporary) data používaná jako přechodná paměť pro pozdější zpracování, např. čekající zakázky, data pro tiskové programy.

   Typ M – manuální data.

   Znakem dobré volby procesů je úzké rozhraní, tj. malý počet datových toků, které vedou „do procesu“ (proces je konzumentem) a které vedou z procesu (proces je zdrojem dat).
5. Vyčištění DFD první úrovně.
   Zkontroluje se, zda vytvořený DFD obsahuje všechny datové toky nebo zda nejsou třeba další procesy provádějící transformace dat. Osvědčuje se postupně, počínaje procesy, které generují data pro externí objekty, hledat odpověď na otázky:
   - Má proces přístup ke všem datům, která potřebuje?
   - Lze použít data existujících datových úložišť, nebo je pro získání dat nutný další proces?

   Podle výsledků analýzy se doplní procesy a datové toky.
6. DFD nejvyšší úrovně.
   DFD nejvyšší úrovně je základním dokumentem návrhu systému. Má zobrazovat strukturu systému ve srozumitelné formě; prokázat, že analytik porozuměl systému; poskytnout základnu pro diskuze uvnitř řešitelského týmu a se zákazníkem; vymezit oblast, jíž se systém týká, a vytvořit základ pro dekompozici systému. Lze jej též využít pro prezentaci alternativ řešení.
7. Formální omezení na DFD:

1. Označení místa, kde se provádí příslušné činnosti.

a) Každý proces musí být navázán alespoň na jeden vstupní a alespoň na jeden výstupní datový tok. Totéž platí i pro datová úložiště. Výjimkou mohou být systémová data používaná pouze pro čtení.
b) Mělo by být možné vystopovat celý životní cyklus informace určitého druhu od vzniku přes kroky zpracování až ke zrušení.
c) U datových úložišť (DÚ) se odstraňují následující vlastnosti:
   – DÚ nefunguje jako zdroj dat pro žádný proces,
   – stejná data jsou uložena ve více úložištích[2],
   – úložiště obsahuje logicky nesouvisející data.
e) Vytvoří se datový model. Při tom je vhodné vyžadovat, aby každá entita byla pouze v jednom úložišti. Pod entitou rozumíme soubor atributů specifikující vlastnosti nějakého objektu, např. osoby v kartotéce personálního oddělení. Entity jednoho úložiště by spolu měly souviset. Pokud nesouvisejí, je vhodné úložiště rozdělit.
f) Fyzické informační toky se nahradí logickými tím, že se označí skutečně přenášenými daty a odstraní odkazy na média, používaná v reálu k přesunu dat. Toky, které jsou zbytečné, se odstraní. Jsou to takové toky, které přenášejí data obsažená v jiných vstupních datových tocích daného procesu.
g) Spojování procesů a odstraňování procesů, které nemění data. Spojeny by měly být procesy se stejnými vstupy. Spojují se procesy A a B splňující podmínku, že A má jediný výstup a ten je jediným vstupním tokem procesu B.

U systémů reálného času se používají DFD obohacené o další prvky, jako jsou řídící impulzy, stálé toky dat, např. od čidel, souběžnost prováděných akcí atd. Dobrý DFD splňuje následující další podmínky:

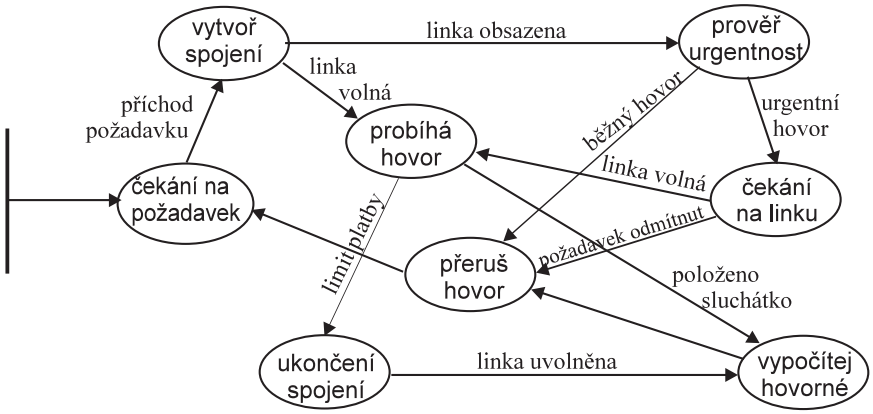
a) Procesy jsou charakterizovány slovesem nebo slovesným podstatným jménem, např. Zpracování objednávek. Pokud toho nelze dosáhnout, je to často proto, že proces nebyl vhodně navržen.
b) Výstupní datové toky by měly záviset na všech vstupních tocích. Nemělo by např. docházet k tomu, že jeden výstupní tok procesu závisí na dvou vstupních tocích a druhý výstupní tok téhož procesu na jiných dvou vstupních tocích. Takový proces je vhodné rozdělit.
c) Procesy by měly mít co nejméně vstupních a výstupních toků. Zmenšení počtu datových toků lze často docílit vhodným rozdělením funkcí systému mezi jednotlivé procesy.

8. Minimalizace počtu procesů.
   Hledají se procesy vhodné ke sloučení. Jsou to takové procesy, které mají stejné vstupy a výstupy, mají stejné „sousedy“, na které jsou vázány datovými toky externí objekty, úložiště a procesy, a které případně komunikují mezi sebou. Takové procesy se sloučí a následně se odstraní redundantní datové toky.
9. Dekompozice procesů, vytváření DFD nižších úrovní.
   Příklad návrhu DFD nižší úrovně je uveden výše (obr. 12.9, obr. 12.10). Vytváření DFD nižší úrovně si může vynutit změnu na DFD vyšší úrovně.
10. Vyčištění modelu.
    Na závěr je nutné odstranit redundance v datech, zpřesnit datový model (pomocí ER diagramů), odstranit redundantní datové toky, redundantní procesy a nelogické řazení procesů a provést úpravy podle výše uvedených požadavků na dobrý DFD.

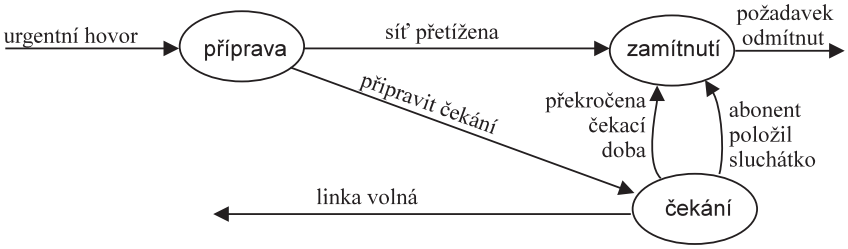
2. To u distribuovaných systémů nevylučuje replikaci dat. To je ale technické opaření, které nesouvisí s logickou strukturou systému.

12.4 Modelování dynamiky systému. Přechodové diagramy. Diagramy interakcí

V některých případech bývá výhodné popsat průběh aktivit, například postup zpracování nějakého dokumentu nebo průběh nějakých činností, ve formě přechodových diagramů. Přechodový diagram je zadán stavy, např. stav vyřízení žádosti, a přechody mezi stavy spolu s podmínkami, za kterých se má příslušný přechod uskutečnit. Příklad přechodového diagramu pro činnost telefonní ústředny je uveden na obr. 12.13; obr. 12.14 ilustruje, jak lze dekomponovat stav „Čekání na linku“.



Obr. 12.13: Schéma činnosti při propojování telefonního hovoru v telefonní ústředně.

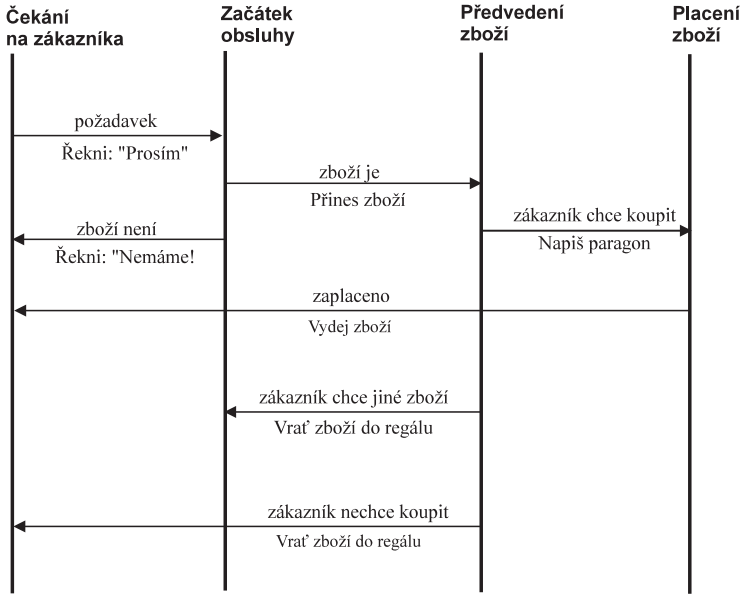


Obr. 12.14: Dekompozice stavu Čekání na linku.

Zobrazení aktivit ve formě z obr. 12.13. a obr. 12.14 připomíná diagramy toků dat. Šipka však neoznačuje předání dat, ale akci: většinou provedení nějaké části programu nebo příchod vnějšího podnětu. U jednotlivých přechodů může být udána buď akce, nebo podnět, nebo obojí. U akcí předpokládáme, že pracují nad nějakou společnou datovou bází. Provede se ta akce a ten přechod, který je možný buď při daném stavu databáze, nebo daném stavu vstupů, nebo obojím.

Přechod urgentní hovor se na obr. 12.13 provede, právě když je v datech údaj, že zpracovávaný hovor je urgentní, tj. volající chce čekat, až se uvolní linka, nemůže však přerušit probíhající hovor. Abstraktní stroje bývají vhodné pro návrh na poměrně nízké úrovni a pro systémy charakteru hromadné obsluhy.

Přechodové diagramy mohou mít různou grafickou formu. Forma, kterou zvolíme, závisí na vhodnosti pro daný účel.



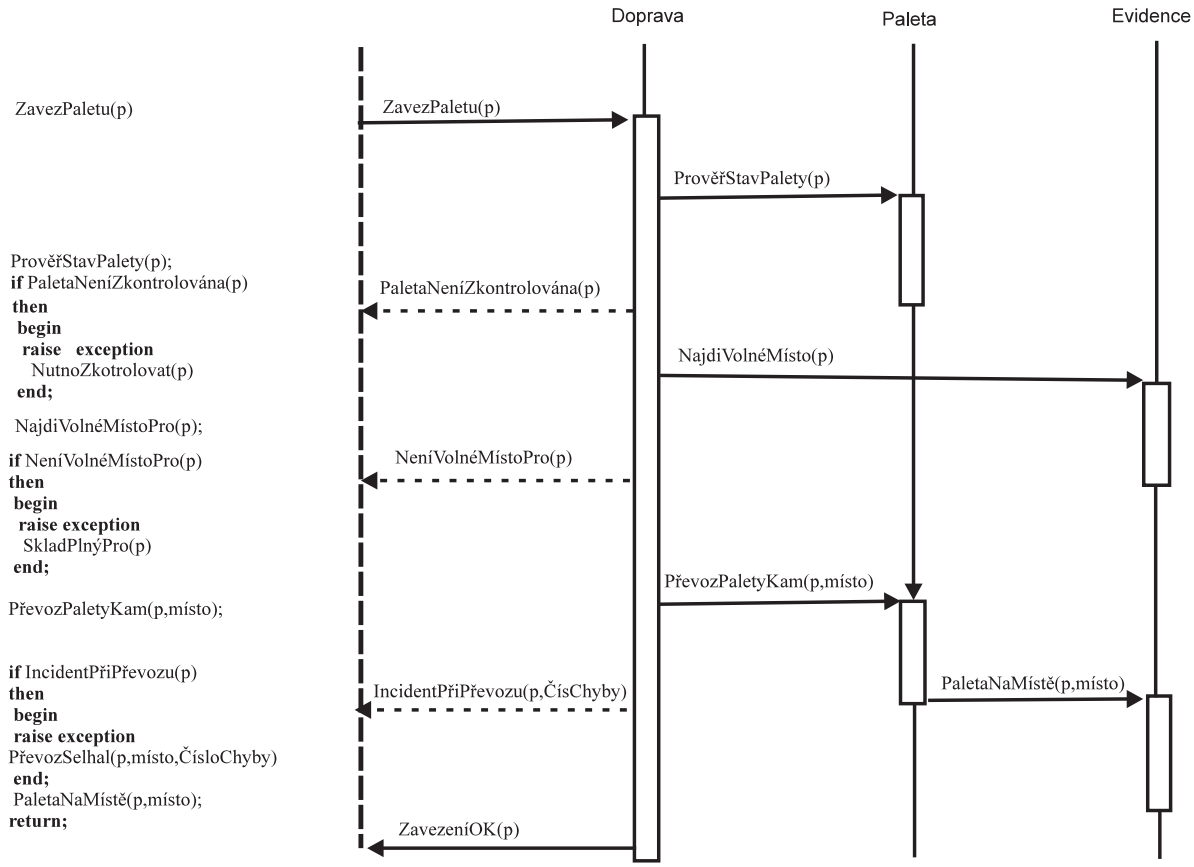
Obr. 12.15: Scénář (přechodový diagram) práce prodavače.

Při návrhu rozsáhlejších systému je výhodné popsat činnosti ve formě tzv. scénářů. Scénář je v podstatě abstraktní stroj, ve kterém jsou stavy zobrazeny jako svislé čáry, nad kterými jsou jména stavů (obr. 12.15). Stav je obvykle spojen s prováděním jisté činnosti a scénář definuje návaznost činností. Čas se vyjadřuje pohybem dolů po schématu.

Scénáře mohou být použity i pro popis komunikace mezi objekty nebo dokonce počítači na síti. V tom případě jsou stavy nahrazeny jmény subsystémů a šipky jsou ohodnoceny předávanými zprávami.

V (Jacobson et al., 1995) jsou scénáře použity jako prostředek objektově orientované analýzy a návrhu implementace případů použití (use case, UC). Případ použití (UC) je ucelená činnost (kap. 11). Jednotlivé UC mohou probíhat v principu souběžně. Každý stav, což může být i označení místa, kde se provádí určitá činnost, nebo skupina objektů objektově orientovaného programu, je chápán jako provádění určitého kroku UC. Kroky UC mohou také probíhat paralelně. Např. začneme-li mluvit na přednášce, můžeme souběžně kontrolovat hlasitost přednesu a reakci publika. Vyžádá-li si klient službu serveru, může po jistou dobu pracovat na svých dalších úkolech. Doba provádění činnosti nebo doba, kdy je daný stav aktivní, je vyznačena nahrazením části svislé čáry svislým úzkým obdélníkem v délce úměrné době činnosti. Obdélník začíná přechodem do daného stavu, např. vyvoláním metody, nebo příchodem zprávy. Přechody z daného stavu nebo odesílání zpráv je možné jen tehdy,

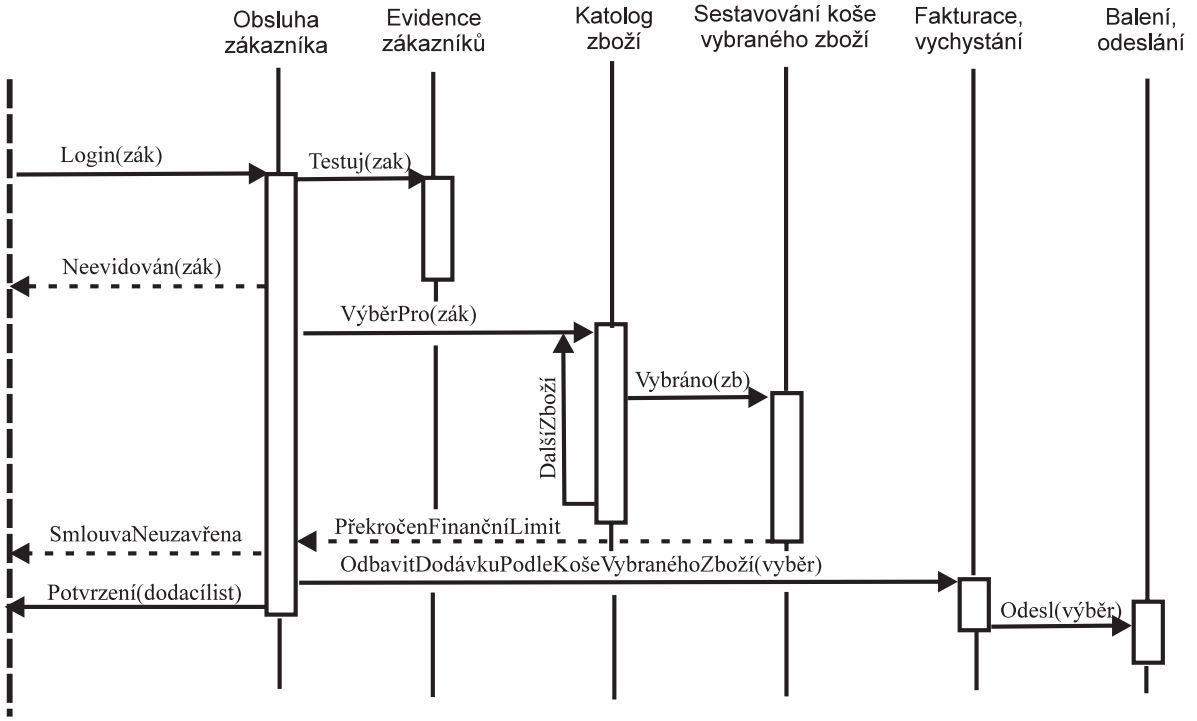
UC Zavezení palety.
Popis: Realizuje se na příkaz operátora z terminálu.
Podmínka provedení: Paleta *p* je na vstupním místě se správným obsahem. Obsah palety byl stanoven v případu použití Pohyb na paletě.
Postup: Nejprve se zkontroluje, zda nebyla paleta používána bez kontroly příliš dlouho; výjimka PaletaNeníZkotrolována. Pokud nebyla, vyhledá se automaticky volné místo pro paletu ve skladu (to závisí na typu palety). Není-li volné místo, vyvolá se výjimka SkladPlnýPro(p). Pak se uskuteční převoz, při kterém může dojít k incidentu ohlášeném výjimkou PřevozSelhal(p,místo,ČísloChyby).
Podmínky po úspěšném provedení: Paleta je fyzicky na správném místě a je na tomto místě vedena v evidenci ve skladu.
Reakce na výjimky: Ruční zásahy stanovené v provozním řádu.



Obr. 12.16: Diagram interakcí pro UC Zavezení palety do regálového skladu.

když je stav aktivní, což se graficky vyznačuje jako šipka „ze zdvojené čáry“. Pokud se čeká na provedení příkazu, nepřestane být stav aktivní v okamžiku odeslání zprávy nebo vyvolání metody cizího objektu.

V případě distribuovaných aplikací lze notaci INTD využít následujícím způsobem. Svislé čáry označují jednotlivá aktivní místa distribuované aplikace, např. jednotlivé servery, a také skupiny objektů v těchto místech.



Obr. 12.17: Diagram výměny zpráv systému elektronického prodeje. Svislé čáry označují moduly či skupiny objektů zajišťující danou činnost.

Pak má smysl, aby přechod nebo zpráva směřovaly i na stejný server, kde spustí paralelně pracující proces, jehož aktivita se vyznačí dalším obdélníkem posunutým mírně do strany (srv. Comm. of ACM, 10/97). K takto zobecněnému scénáři, pro který se používá často název diagram interakce (interaction diagram, INTD), se mohou připojit následující dokumenty:

- krátké shrnutí funkcí INTD;
- vstupní podmínky (preconditions) udávající podmínky, za kterých se jednotlivé UC definované v daném INTD uskutečňují;
- podrobný popis INTD v přirozené řeči. Důležitou součástí popisu je stanovení míst, kde dochází k nestandardním situacím, při nichž vznikají „výjimky“;
- popisy akcí při vzniku výjimek;
- výstupní podmínky (postconditions) udávající podmínky, které platí po provedení INTD;
- pseudokód definující práci scénáře, tj. text, který má strukturu programu s podmíněnými příkazy, cykly atd., jehož některé části jsou v přirozeném jazyce.

Starý zákazník	A	A	A	A	A	…	N	N
Běžný leasing	A	A	A	A	N		x	x
Rušení nájmu	A	A	N	N	A		x	x
Nový nájem	A	N	A	N	A		A	N
Zařadit zákazníka							x	x
Test platby	x	x	x	x	x			
Zrušení smlouvy	x	x						
Nová smlouva	x		x		x		x	
Faktura	x	x	x	x	x			
Úprava smlouvy	x	x	x	x				

Tab. 12.2: Rozhodovací tabulka popisující činnost leasingové firmy.

Příklad jednoduchého INTD s připojeným pseudokódem je uveden na obr. 12.16. Na obr. 12.17 je INTD definující činnost systému elektronického prodeje. INTD je odvozen od přechodového diagramu z obr. 12.15. Na obr. 12.17 jsou přechody ohodnoceny zprávami. Takový INTD se nazývá diagram výměny zpráv (message trace diagram).

INTD jsou důležitou technikou používanou při specifikaci požadavků, protože uceleně a velmi instruktivně zobrazují činnosti, např. průběh vytváření a zajišťování zakázky. Prostředky práce s UC a INTD jsou součástí moderních CASE systémů.

Notace přechodových diagramů byla rovněž zdokonalena v Rumbaughově verzi objektově orientovaného návrhu (Rumbaugh et al., 1991). Rumbaughova notace umožňuje poměrně přesně specifikovat způsob předávání informací a provedení asociovaných akcí. Je možné specifikovat i některé požadavky na synchronizaci akcí. Rumbaughova metodika se stala základem standardizačního úsilí OMG – Object Management Group. Výsledkem tohoto úsilí je např. norma CORBA definující spolupráci distribuovaných objektů.

12.5 Rozhodovací tabulky

Rozhodovací tabulky (viz např. Humby, 1976) představují způsob programování, ve kterém se maticovou (tabelární) formou zadávají podmínky a s nimi asociované akce. Obecný tvar rozhodujících tabulek je uveden v tab. 12.2. V horní části tabulky jsou po sloupcích zapsány všechny možné kombinace hodnot elementárních podmínek $P_1, \ldots, P_n$ (A a N zastupují ano a ne, lze uvést i jiná slova). Každý sloupec pak představuje konjunkci – současné splnění – elementárních podmínek, přičemž podmínka P_i vstupuje do konjunkce v záporu v případě, že v i-tém řádku je uveden znak N. A značí, že má daná elementární podmínka platit. Např. sloupec 2 v tab. 12.2 určuje, že platí složená podmínka Starý zákazník **a** Běžný leasing **a** Rušení nájmu **a ne** Nový nájem. x označuje, že na dané elementární podmínce v daném sloupci nezáleží. V dolní části tabulky jsou uvedeny přidružené akce $A_1, \ldots, A_k$. Každé akci je v tabulce vyhrazen jeden řádek. x v řádku akce a sloupci s stanovuje, že se daná akce uskuteční při současném výskytu podmínek udaných ve sloupci s. Význam tabulky je patrný z tab. 12.2.

Rozhodovací tabulky jsou obvyklým nástrojem používaným na podporu rozhodování. Jsou výhodné při rozhodováních, při kterých se vyskytuje více podmínek. Jako specializovaný nástroj mají i nevýhody:

- I po různých optimalizacích jsou značně velké.
- Nejsou hierarchické – nejsou k dispozici rozumné prostředky na popis akce zadané opět rozhodující tabulkou.
- Neobsahují explicitní prostředky synchronizace akcí.

- Některé podmínky se obtížně vyjadřují.

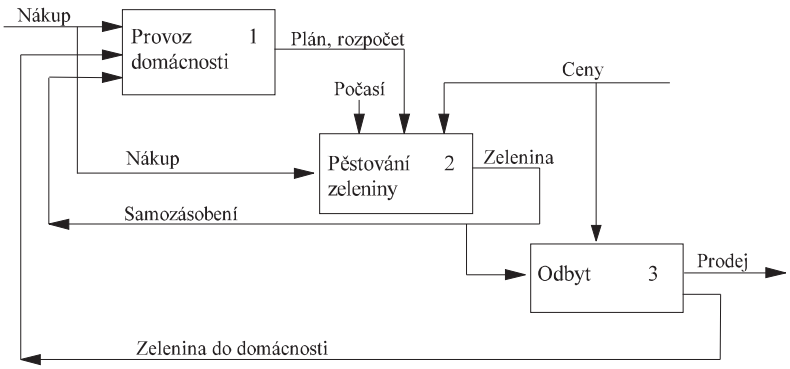
Rozhodovací tabulky mohou být výhodné jako forma výstupu IS – návod k rozhodování. Při návrhu IS lze někdy použít rozhodovací tabulky jako prostředek definice části systému řízené událostmi (nastane-li toto pak . . .).

Při specifikaci významu akcí je vhodné označit políčko akce písmenem a číslicí, jak je zvykem např. u tabulkových kalkulátorů, a pod tím označením akce dále specifikovat.

12.6 Metoda SADT

Metoda SADT (Structured Analysis and Design Technique) byla vyvinuta jako jeden z prvních prostředků, které do jednoduchého schématu zahrnuly jak procesy v diagramech akcí tak data v diagramech dat (Marco, McGovan, 1988). Diagramy akcí popisují ve formě blízké diagramům toků dat fyzické toky informací mezi procesy, diagramy dat popisují toky dat mezi datovými úložišti.

DIAGRAM AKCÍ A0: *Zahradnictví.*

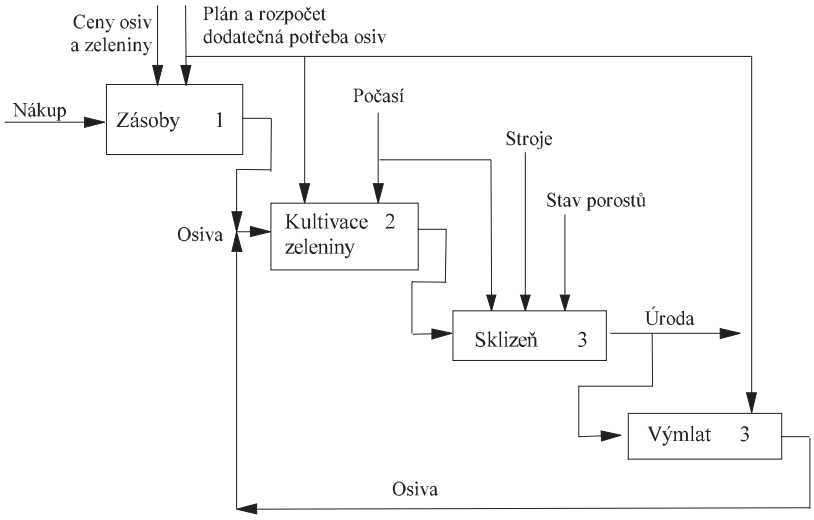


Obr. 12.18: Diagram akcí.

V diagramech akcí je možné pro každý proces/akci specifikovat vstupy (zleva), výstupy (napravo), podněty, doplňující informace a podmínky (shora) a zdroje a jiná fakta důležitá pro řízení (management) procesu (zdola). Diagramy dat popisují datové toky mezi strukturami dat, které jsou obdobou datových úložišť. Metoda se nestala základem žádného úspěšného CASE nástroje, byla však úspěšná při neautomatizovaných specifikacích a návrhu. Dnes se používá hlavně při specifikacích softwarových procesů – modelů postupu prací (viz kap. 18 a knihu Fairclough, 1996). Výhodou SADT v této oblasti je to, že má rozvinuté prostředky zobrazování skutečností významných pro řízení softwarových projektů a paradoxně i to, že při současném stavu znalostí není plná automatizace řízení softwarových procesů vždy výhodná. Podstata metody je patrná z obr. 12.18 až obr. 12.21. Další příklad použití lze nalézt v kap. 18. Obdélníky jsou umístěny zásadně na diagonále, tj. na spojnici levého horního a pravého dolního rohu. Součástí dokumentace metody SADT je přehled diagramů v následující formě (srv. obr. 12.18).

A0 Provoz zahradnictví

DIAGRAM AKCÍ A0.1 *Pěstování zeleniny*.



Obr. 12.19: Podschéma diagramu akcí A2.

A01 Provoz domácnosti
(Případná podschémata v A01)
A02 Pěstování zeleniny
A021 Dodávky
A022 Kultivace
A023 Sklizeň
A024 Výmlat
A03 Odbyt
atd.

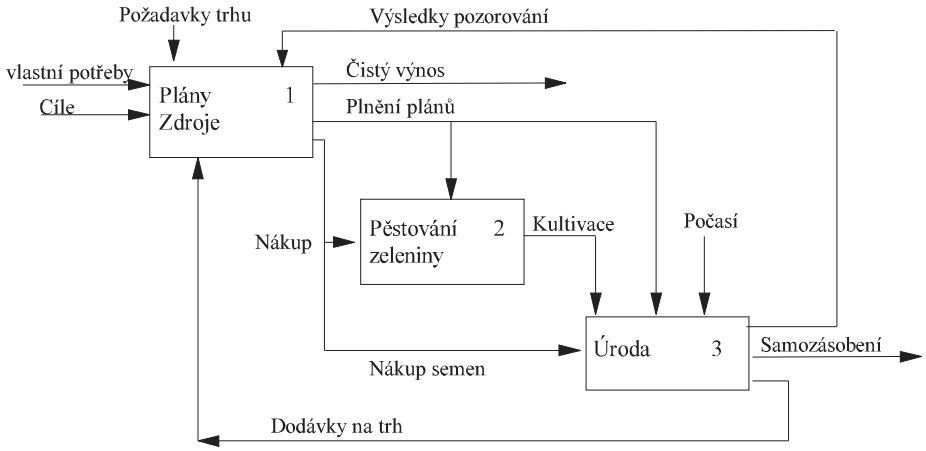
12.7 Objektově orientované metody

12.7.1 Životní cyklus softwaru budovaného objektově orientovanými metodami

Při objektově orientované specifikaci a návrhu je typické, že se podstatná část systému vytváří „skládáním“ objektů uložených v knihovně. Podle (Branson et al., 1992) se osvědčuje vývoj objektově orientovaného softwaru rozčlenit do následujících etap a kroků:

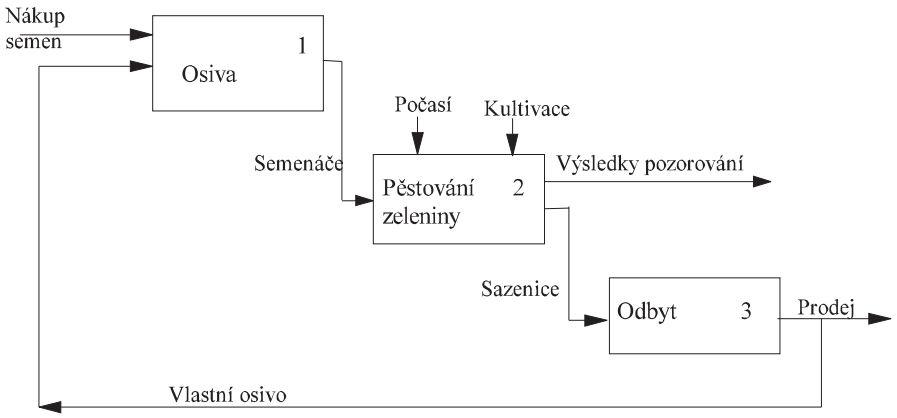
A) *Analýza*.
Úkolem je specifikovat požadavky hutně a úplně bez zavádění omezení plynoucích z použití určitého hardwaru a základního softwaru. Analýza se skládá z následujících kroků:

DIAGRAM DAT D0: *Zahradnictví.*



Obr. 12.20: Vrcholový datový diagram.

DIAGRAM DAT D 0.2: *Úroda.*



Obr. 12.21: Dekompozice datové struktury Úroda.

A1 Vytvoření modelu systému z pohledu zákazníka nezávislého na cílovém hardwaru a softwaru a detailech implementace. Obsahuje modelování podstatných (esenciálních) aktivit a esenciálních dat. Provádí se v následujících krocích:

a) Vytvoření uživatelského pohledu na systém; jak se uživateli budou jevit funkce systému a jaké informace k tomu potřebuje.
b) Modelování esenciálních aktivit.
c) Revize požadavků – oponentura výsledků kroků a) a b).
d) Definice dat: bližší rozbor typů dat a jejich vztahů, stanovení, které operace se váží na které údaje. Zde se může provést prvý návrh atributů tříd.
e) Zpřesnění esenciálního modelu. Na základě znalostí o datech se zpřesní a doplní specifikace esenciálních operací.
f) Revize externí struktury, jak se jeví uživateli z pohledu jeho činnosti, a revize specifikací.
g) Podrobná analýza požadavků. Závěrečná revize požadavků. Využijí se výsledky předchozí oponentury a provede se rozbor úplnosti a konzistentnosti požadavků, doplní se chybějící operace.

Výstupem je analyzovaný soubor požadavků. Objektová orientace nebývá v této etapě nutná, bývá však výhodná. Je výhodné celý proces organizovat jako postupné zjišťování skupin operací a činností, které by mohly být snadno implementovány objektově orientovanými metodami.

A2 Návrh kandidátů na esenciální třídy. Esenciální třída je třída definující základní funkce systému. Soubor esenciálních tříd tvoří esenciální model systému. Esenciální data a esenciální metody jsou data a metody esenciálních tříd. Na základě specifikace požadavků nebo již během specifikace se vytvoří diagramy toků dat, definice dat a ER-diagramy. Tyto dokumenty jsou základem výběru esenciálních tříd a jejich metod. Vychází se z externích entit, datových úložišť, vstupních a výstupních datových toků a specifikací funkcí procesů. Výstupem této etapy je prvý částečně formalizovaný návrh tříd, který ještě nebere ohled na implementační omezení.

B) *Návrh*.

Modifikace tříd esenciálního modelu tak, aby byly realizovatelné na daném hardwaru a softwaru. To vyžaduje definici pomocných tříd a postupný vývoj hierarchie tříd. Realizuje se v následujících krocích:

B1 Stanoví se omezení pro esenciální model tak, aby se vyhovělo omezením plynoucím z vlastností použitého hardwaru a softwaru. Esenciální aktivity a esenciální data jsou přiřazena konkrétním procesům a datovým strukturám. Esenciální aktivity jsou doplněny o aktivity a data, jejichž zavedení je nutné z důvodů omezení vyplývajících z vlastností použitého hardwaru a softwaru. Doplněné aktivity jsou přiřazeny procesům a data datovým systémům. Tím vznikne „inkarnační model“. V případě IS je obvykle nutné navrhnout propojení s relačními databázemi, protože objektové databáze nejsou zatím plně technicky zvládnuty a protože jsou existující data obvykle uložena v relačních databázích.

B2 Provede se syntéza tříd. Návrhy tříd z předchozích kroků jsou zpřesněny a z tříd se vytvoří hierarchie tím, že jsou společné atributy a metody odděleny a soustředěny do nadtříd a podtříd. Výsledné třídy jsou modifikovány tak, aby se daly použít vícekrát.

B3 Revize/inspekce s cílem analýzy a verifikace získaných tříd.

B4 Definice rozhraní je prováděna prostřednictvím metod objektů deklarovaných jako instance pro podporu operací rozhraní.

B5 Revize rozhraní definovaného v B4.

C) *Implementace*.

Metody tříd jsou naprogramovány ve vhodném jazyce a odladěny.

C1 Podrobný návrh. Specifikují se metody, které by měly implemetovat vždy jednu uzavřenou funkci systému. Specifikuje se logika, interakce a volání metod jiných tříd. Integritní omezení stanovené v A1 musí být při tom dodržena.

C2 Implementace – kódování.

C3 Inspekce kódu.

C4 Testování tříd. Jsou prověřovány metody skupin tříd vytvořením vhodných instancí tříd (objektů). Tato etapa se někdy zahrnuje do etapy integračního testování.

D) *Integrační testování.*

Z tříd se vytvoří instance tříd – objekty – a ty se integrují do fungujícího systému, který se testuje jako celek. OO metodika je vhodná pro inkrementální způsob testování. Testování začíná u malého jádra, které se postupně rozšiřuje.

E) *Předání.*

Při stanovování pravidel práce s třídami a akcí obsluhy je v řadě případů výhodné využívat přechodové diagramy (srv.. 12.4). OO metody mají rozvinutý systém notace přechodových diagramů.

Metodika z (Jacobson et al., 1995) vycházející z případů použití zahrnuje všechny etapy z doporučení IBM. Esenciální třídy jsou v tomto případě ty, které definují funkce a kroky případů použití.

Objektově orientovaná metodologie je tedy založena na třech pojmových okruzích a skupinách pohledů:

a) *Funkční modely.* Jsou ideově identické s diagramy toků dat v poněkud modifikované notaci. Funkční model je zobecněn o možnost zobrazení toků příkazů, nejenom dat, a o možnost tvorby a rušení procesů. Datové toky se mohou větvit a spojovat. Lze použít i notaci use case.

b) *Objektové modely.* Tyto modely zobecňují ER-diagramy a zobrazují vztahy mezi třídami (dědění, abstrakce, relace) i mezi objekty.

c) *Dynamický model.* Jde o různé varianty notace přechodových diagramů a diagramů interakcí, ve které jsou rozvinuty prostředky pro definici „vložených“ přechodových diagramů. Je možné stanovit akce prováděné při vstupu do stavu a při jeho opuštění stejně jako stanovit událost prováděnou při daném přechodu. Lze rovněž posílat zprávy mezi stavy a vázat uskutečnění přechodu mezi stavy na splnění určité podmínky.

Příklad grafických prostředků OO návrhu v Rumbaughově variantě je uveden v následujícím paragrafu. Objektově orientovaná analýza a návrh se jednoznačně osvědčují v operativních IS a dají se používat i v jiných případech. Metodika objektově orientované analýzy a návrhu se rychle rozvíjí. Existuje několik variant objektového přístupu.

Nejrozšířenější je přístup podle (Rumbaugh, 1991). Kromě toho se často používá i méně formalizovaná Boochova metodologie, která je bližší filozofii spolupracujících aplikací (Booch, 1991, 1995). S Boochovou metodologií lze výhodně kombinovat přístup Jacobsona, 1995, – use case – zmíněný výše. Existuje řada dalších variant objektového přístupu, jako je např. Fusion Method (Coleman, 1994) resp. metoda KISS (Kristen, 1994). Existence více variant objektových metodik svědčí o tom, že se objektové metody zatím plně nestabilizovaly.

Stručně řečeno, velké systémy je výhodné při používání objektové orientace vyvíjet v následujících krocích:

a) Dekompozice systému do aplikací nebo komponent bez specifikace vnitřní struktury aplikací – dekompozice ve formě černých skříněk. Aplikace pracují asynchronně, tj. obdobně jako služby, např. pošta, v lidské společnosti, při vyžádání služby nečekáme na poště na potvrzení, že dopis došel, v programu nečeká volající na výsledek.

b) Objektový návrh jednotlivých aplikací.

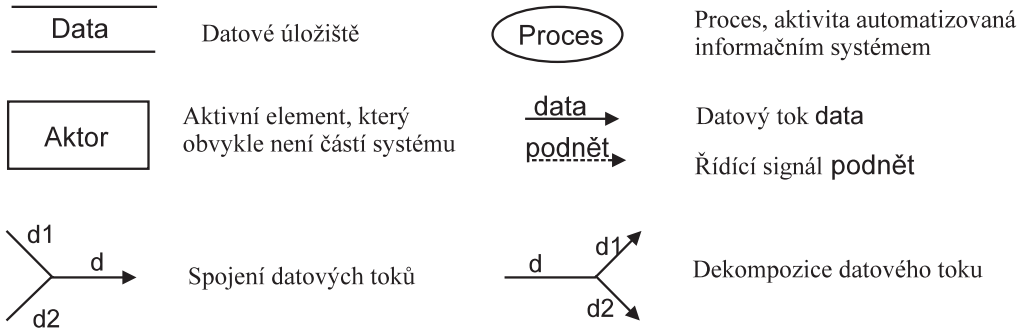
c) Rozštěpení konkrétní aplikace podle potřeb, např. na část pracující na klientu a část na serveru, určením, kde budou pracovat objekty určité třídy.
d) Odvození datových toků mezi částmi aplikace a generace diagramů toků dat mezi částmi aplikace, např. klienty a servery. Spolupráce na úrovni jedné aplikace je obvykle synchronní – při vyžádání služby se čeká na její provedení, technicky se provádí jako volání (vzdálených) procedur nebo metod.

12.7.2 Prostředky objektové analýzy a návrhu

Nejrozšířenější grafická notace a objektově orientovaná metodologie pochází od skupiny autorů (Rumbaugh et al., 1991). Zde shrneme základní principy jejich metodiky. Podrobnější popis metodiky lze nalézt i ve skriptech (Sochor, Richta, 1996).

Funkcionální model (diagramy toků dat)

Diagramy toků dat jsou prakticky identické s diagramy uvedenými v kap. 12.3. Odlišnosti jsou prakticky výhradně v notaci, jak je patrné z obr. 12.22.

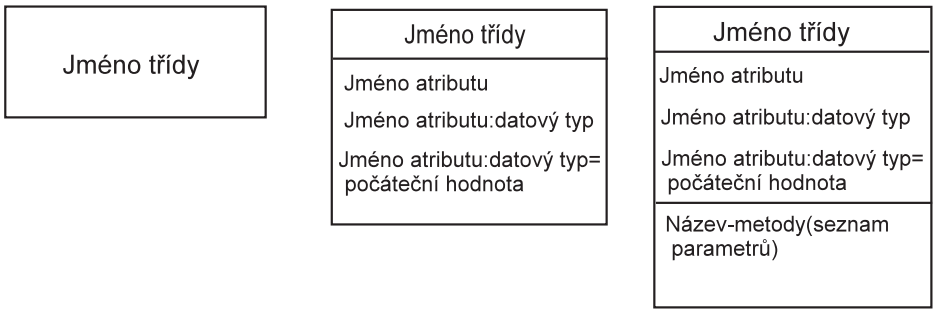


Obr. 12.22: Prvky Rumbaughovy notace diagramů toků dat. Formální požadavky: Šipky musí začínat či končit v úložištích, procesech či aktorech. Procesy a úložiště musí mít alespoň jeden vstupní a alespoň jeden výstupní tok.

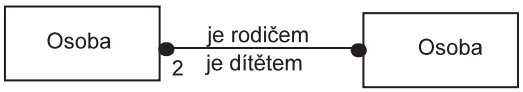
Diagramy tříd, modely tříd

Při grafickém zobrazování tříd se používá notace z obr. 12.23. Pro zobrazování vztahů mezi třídami a jejich vlastností je možné specifikovat následující skutečnosti:

A) *Asociace tříd.*
   a) Základní tvar asociace. Mezi třídami mohou být vztahy podobné jako relace mezi entitami v ER notaci. Pro tyto vztahy se používá označení asociace. Je však možno specifikovat vlastnosti asociací. Je možné explicitně stanovit počet výskytů a zároveň stanovit tzv. role, jak je patrno z obr. 12.24 (rodiče jsou právě dva).
   Při stanovování násobnosti lze zadávat výčet možností obdobně, jak je známo z řady aplikací pro osobní počítače.

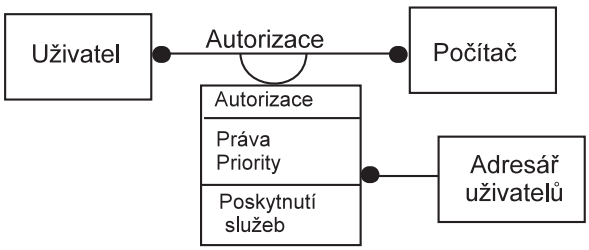


Obr. 12.23: Grafické zobrazování tříd.



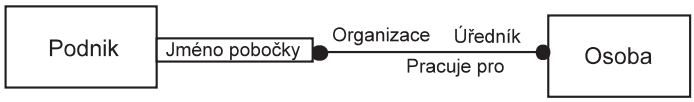
Obr. 12.24: Základní forma asociace tříd. Tečka označuje vyšší násobnost 0:*n*, kroužek násobnost 0:1, čára právě jeden výskyt

b) Asociace s vázanou třídou. Je možné definovat třídu vázanou na konkrétní asociaci. Příkladem může být specifikace přístupu v informačním systému. Třída vázaná na asociaci tříd pak definuje vlastnosti vztahu objektů v realitě, např. atributy přihlášení uživatele na počítači (obr. 12.25).



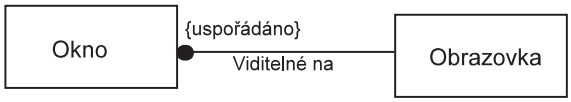
Obr. 12.25: Třída vázaná na asociaci jiných tříd.

c) Kvalifikovaná asociace. Je možno stanovit, který atribut – kvalifikátor – zajišťuje asociaci, která je pak kvalifikována, viz obr. 12.26. Na stejném obrázku je použita možnost stanovení *rolí* entit v asociaci (organizace, úředník).



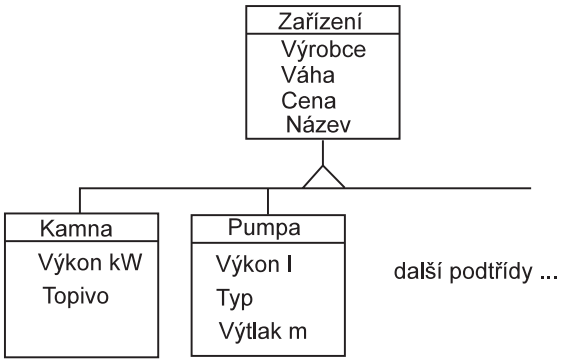
Obr. 12.26: Kvalifikace asociace.

d) Podmínky v relacích, omezení. Pro asociaci je možno stanovit některé další podmínky uzavřené do složených závorek { }. Je možno např. stanovit, že vždy existuje pořadí oken na obrazovce, tj. jak se okna překrývají. Princip je patrný z obr. 12.27.

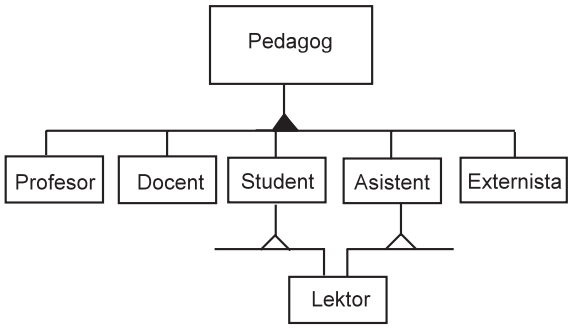


Obr. 12.27: Podmínky pro asociace.

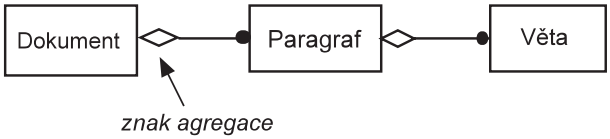
B) *Dědění.*
Vztah dědění mezi třídami se vyjadřuje způsobem patrným z obr. 12.28. Třídy Kamna i Pumpa mají některé atributy (Výrobce, Cena, Váha, Název) i metody (např. způsob objednávání) společné. Tyto společné atributy a metody jsou definovány jako atributy „rodičovské“ třídy (zde Zařízení) a „děděny“, tj mohou být používány u „potomků“. Potomek (třída Kamna) může v případě potřeby atributy i metody redefinovat. Jednou z objektových technik je abstrakce, při které se společné atributy a metody několika tříd přesunují do nově vytvořené rodičovské třídy.

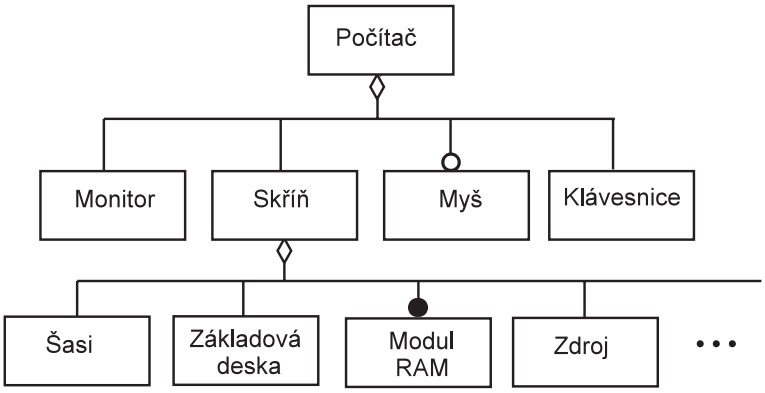


Obr. 12.28: Vztah dědění mezi třídami.



Obr. 12.29: Vícenásobné dědění.

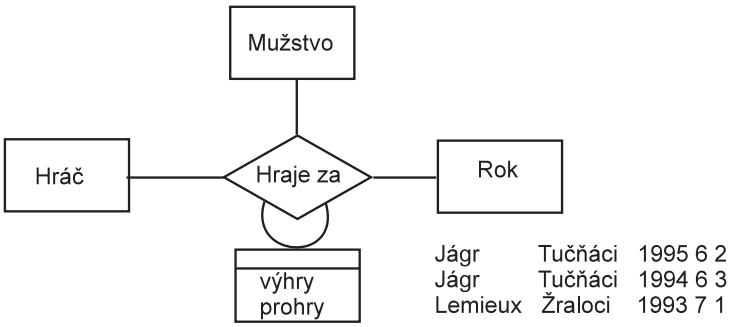




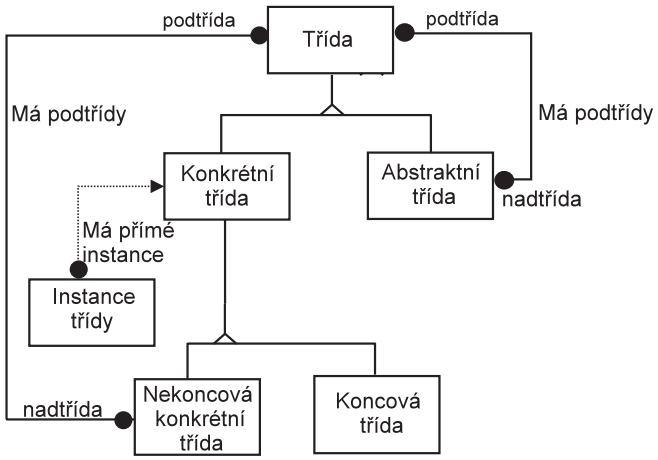
Obr. 12.30: Agregace.

Dědění může být od více rodičů, což se snadno zobrazí tím, že potomek má více rodičů (nadtříd). Může při tom dojít k nepříznivé situaci, kdy se dědí do různých rodičů dvě metody či dva atributy, které vznikly děděním od stejného zdroje a nejsou tedy odlišné (jak je patrné z obr. 12.29). Tomuto jevu se říká vícenásobné dědění. Vícenásobné dědění je nebezpečné, může vést k nepříjemným chybám a proto je explicitně vyjádřeno černým trojúhelníkem místo obvyklého znaku dědění v místě, které „vícenásobné dědění umožnilo“.

C) *Agregace.*
Agregace se používá pro označení vztahu „sestává se z“, je to vztah celku a částí. Podtřídy vytvořené agregací nemohou mít více než jednoho rodiče a jejich nově definované metody a atributy by měly být rozdílné od metod a atributů nadtříd, nedochází tedy k předefinování metod a atributů rodičů. Příklady jsou na obr. 12.30.



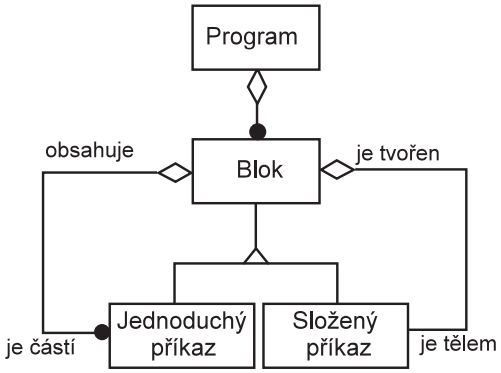
Obr. 12.31: Zobrazení ternární asociace.



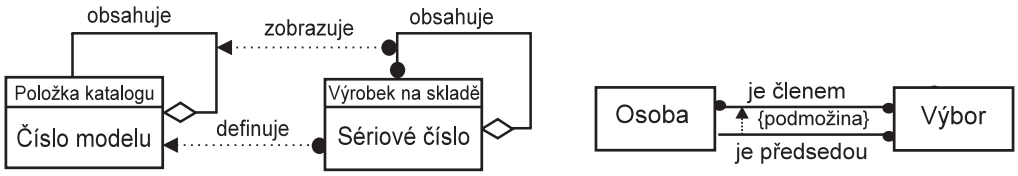
Obr. 12.32: Příklad rekurzivní struktury.

D) *Vícenásobné asociace.*
Příklad: Je třeba zaznamenat, za které mužstvo hrál v určitém roce určitý hráč, v kolika případech jeho mužstvo vyhrálo a v kolika prohrálo. Hráč hraje v určitém roce za určité mužstvo. Mužstvo má více hráčů. Řešení je na obr. 12.31.

E) *Rekurzivní struktury – hierarchie typů tříd, rekurzivní asociace.*



Obr. 12.33: Modelování struktury programu.

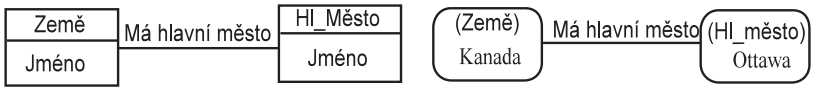


Obr. 12.34: Homomorfizmus mezi asociacemi a podmínky vztahů mezi třídami.

Vztahy mezi třídami můžeme modelovat schématem z obr. 12.32. Vztahy mezi třídami mohou být rekurzivní. Vztah je rekurzivní může-li být třída i nepřímo sama sobě nadtřídou či podtřídou. Rekurzivní mohou být i agregáty, jak je patrné z obr. 12.33

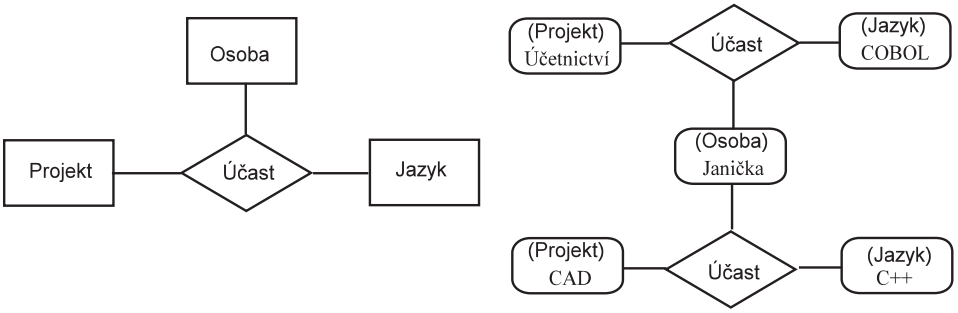
F) *Sémantické podmínky, homomorfizmy.*
Je možno stanovit logické vztahy mezi třídami a mezi asociacemi. Příklad podmínky pro třídy je na obr. 12.32, vztah „Má přímé instance“.
Příklady vztahů mezi asociacemi jsou na obr. 12.34. Vztah „zobrazuje“ mezi asociacemi je homomorfizmem ve smyslu matematickém – každé asociaci na jedné straně sémantického vztahu odpovídá (obvykle právě jedna) asociace na druhé straně vztahu. Vztah mezi asociacemi může mít též formu podmínky, ta je pak udávána ve složených závorkách. Příklad použití je na pravé straně obr. 12.34.

G) *Modelování objektů. Sítě objektů.*
Někdy je nutné modelovat skutečný stav dat – objektů. K tomu lze použít sítě objektů, tj. instancí tříd, popisující konkrétní vztahy mezi objekty. Vztahy mezi objekty se zobrazují podobně jako vztahy mezi třídami s tím, že se

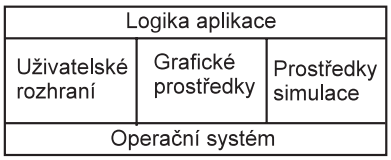


Obr. 12.35: Model tříd a odpovídajících objektů.

neuvádí některé informace, např. o násobnosti a metodách. Příklady modelování jsou na obr. 12.35 a obr. 12.36. S modelováním objektů je ta potíž, že se modely rychle stávají nepřehledné.



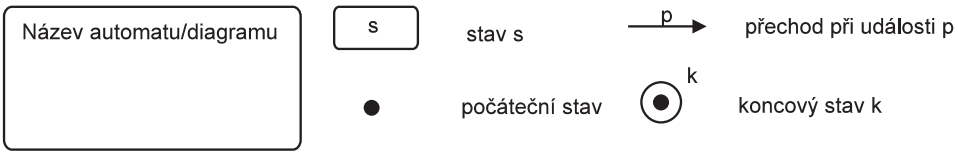
Obr. 12.36: Vztahy tříd a vztahy objektů.



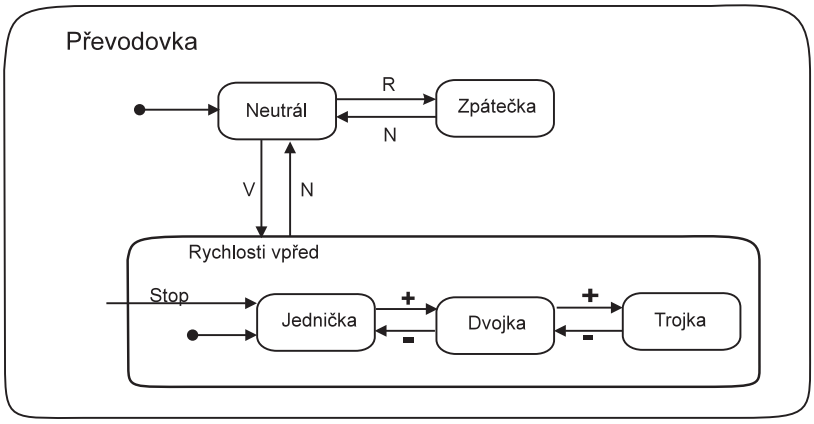
Obr. 12.37: Možná struktura modulů.

H) *Modul.*
Modul je logická konstrukce pro seskupování tříd, jejich asociací a zobecnění vytvářením nadtříd k existujícím třídám. Modul tvoří střední úroveň strukturalizace systému. Určitá třída může být částí více modulů. Systém se zobrazuje jako skupina modulů s vazbami mezi moduly odvozenými z vazeb mezi třídami obsaženými v modulech. V jednoduchých případech je možné zobrazit strukturu systému způsobem z obr. 12.37. Obrázek vyjadřuje fakt, že třídy modulů používají pouze metody sousedních tříd. V našem příkladě Logika aplikace nevolá přímo metody tříd definujících operační systém. Při volbě obsahu modulů se řídíme následujícími hledisky:

- Modul by měl být na jediném počítači – klientu nebo serveru.
- Vazby (asociace, volání metod, používání atributů) tříd jednoho modulu na třídy jiných modulů by měly být co nejslabší (úzké rozhraní).



Obr. 12.38: Grafické prostředky přechodových diagramů.



Obr. 12.39: Příklad přechodového diagramu.

Přechodové diagramy (dynamický model)

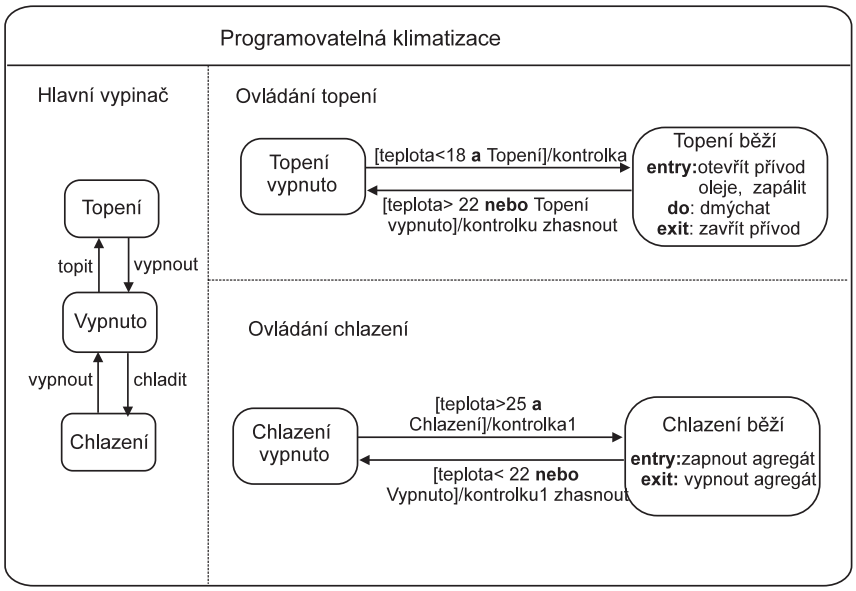
Základní prostředky zobrazení přechodových diagramů.

Přechodové diagramy využívají prvky z obr. 12.38. Příklady použití jsou na obr. 12.40 a obr. 12.41 (srv. obr. 12.13).

Diagram „Převodovka“ popisuje převodovku, která je ovládána tlačítky *STOP*, $+$, $-$, N, R, V. Při stisknutí V se ze stavu „Neutrál“ přejde do stavu „Vpřeď“, který je opět vnitřně strukturován. Přechodem do stavu „Rychlosti vpřeď“ přejde převodovka současně do podstavu „Jednička“. Do podstavu „Jednička“ přejdeme vždy při stisknutí tlačítka *STOP*. Do neutrálu lze přejít z každého rychlostního stupně. Syntaxe ohodnocení přechodu má tvar *událost[podmínka]/akce*. Ani podmínka ani akce nemusí být uvedeny, není-li to pro popis potřeba.

Událost může být spojena se seznamem hodnot atributů uzavřených v závorkách. Pokud je uvedena podmínka, nemusí být uvedeno jméno události. Použití uvedených prostředků je na obr. 12.40 zobrazující část řešení vytápění a chlazení v domku. Pro větší přehlednost jsou jednotlivé přechodové diagramy, z nichž každý vyjadřuje paralelně probíhající činnost, spojeny do společného diagramu. Zároveň se tím vyjadřuje skutečnost, že jednotlivé přechodové diagramy operují nad společnými daty a komunikují tak mezi sebou. Hranice jednotlivých přechodových diagramů jsou vyznačeny tečkovanými čarami. Na obr. 12.41 je ukázáno, jak je možné využít techniku vloženého přechodového diagramu „Čekání na linku“ s více koncovými stavy. V přechodových diagramech je tedy možné:

a) Stanovit logické a časové podmínky pro uskutečnění přechodu.
b) Stanovit akce prováděné
   - při vstupu do stavu *s* se značí **entry:** *událost*,
   - při přetrvávání ve stavu *s*, značí se **do:** *událost*,
   - při přechodu ze stavu *s*, značí se **exit:** *událost*,
   - při konkrétním přechodu, událost se zapisuje k přechodu.
c) Je možné s přechodem asociovat třídu. Při uskutečnění přechodu se pak vytváří instance objektu dané třídy. Je možné specifikovat i souběžnost (současné – paralelní provádění) a synchronizaci akcí. Tyto prostředky jsou však relativně nerozvinuty.



Obr. 12.40: Přechodový diagram, model práce klimatizace.

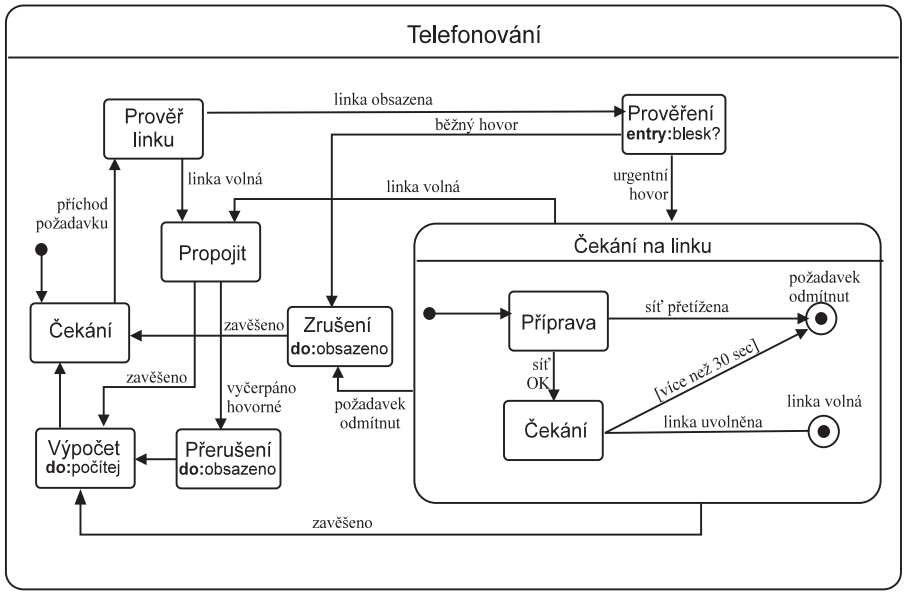
Celkově jsou OO metodologie vhodné spíše pro návrh jedné, byť případně distribuované, aplikace. Integrace existujících aplikací není dosud uspokojivě dořešena. Projevuje se to i v doporučení, ve kterém se DFD konstruuje z objektů jako poslední krok návrhu. Při současném stavu znalostí se zdá optimální následující postup:

- Návrh systému jako komplexu spolupracujících aplikací a popis diagramem toků dat.
- Jednotlivé nově vyvíjené aplikace realizovat OO technikou vytvořením objektového modelu, schématu modulů a případně DFD pro danou aplikaci. DFD je v tomto případě dekompozicí dané aplikace metodou bílých (průhledných) skříněk.

OO techniky se osvědčují, jedná se však o technologii, která přes nesporné úspěchy teprve dozrává. Problémy OO technik jsou široce rozebírány v diskuzi předních odborníků v článku „The Promise and the Cost of Object Technology.“ A Five Years Forecast, Comm. ACM No. 10, (1995), 33–55. Hlavní problémy OO technologie byly v diskuzi specifikovány následovně:

- Nedostatečná standardizace. Pod objekty se skrývají u různých autorů a různých výrobců konceptuálně různé entity.[3]
- Objektový přístup znamená změnu základních programátorských přístupů – paradigmat. To je značné obtížné. Nedostatek obecně přijatých norem způsobuje, že je dokonce nutné zvládat i více než jedno paradigma.

3. Objekt podle normy CORBA (CORBA, 1996, viz též přehled v Plášil, 1996) má málo společného s objekty v jazyce Smalltalk. Objekty ve smyslu Rumbaugh se liší od objektů ve smyslu Boochově. Softwarový agent není identický ani s objektem ve Smalltalku ani s objektem v normě CORBA. Objektově orientovaný software je pak obtížně přenositelný a hrozí, že se v důsledku rychlého vývoje metod stane záhy nepoužitelný.



Obr. 12.41: Model práce telefonní ústředny.

- OO technologie nepodporuje dostatečně hierarchickou dekompozici. V zásadě se pracuje na jedné úrovni hierarchie (pool of objects).
- OO technologie jsou vynikajícím prostředkem v těch případech, kdy se operace v softwaru „podobají“ akcím v realitě. Jsou méně vhodné pro práci s hromadnými daty a pro numerické aplikace. V IS jsou vhodné spíše pro popis operativních činností.
- U některých OO metod nejsou zanedbatelné i vysoké požadavky na zdroje[4].
- OO metody jsou relativně nový nástroj. Je proto nutné zmapovat jeho přednosti a meze. Nelze podlehnout iluzi, jak tomu bylo nejednou v minulosti, že se jedná o univerzální prostředek řešící všechny problémy. Je však mimo diskuzi, že je to prostředek velmi účinný. Zdá se však, že je poněkud přeceňován na úkor technologie spolupráce aplikací. OO technologie může být výhodně použita v kombinaci s technologií spolupracujících aplikací a metodami strukturovaných specifikací a návrhu.

Vývoj OO technologií směřuje k vyšší strukturovanosti s využíváním vzorů návrhu (patterns) a sestav (frameworks, viz závěr kap. 11). Metodologie založená na případech použití (use case, UC) vytváří jednotlivé UC jako soubor spolupracujících objektů, z nichž některé zajišťují rozhraní, jiné mají řídicí funkce a další definují funkce oblasti aplikace (Jacobson et al., 1995). Soubor objektů lze používat jako černou skříňku. V případě potřeby lze používat skupinu objektů jako bílé skříňky. Je to obecnější, avšak velmi pracné.

V poslední době sílí tendence sjednotit notaci v diagramech používaných při specifikaci a návrhu softwaru. Tyto snahy vedly k návrhu Unified Metod Language – UML, viz (Booch, Rumbaugh, 1995). Výsledkem je zatím

4. Např. to, že v metodě CORBA není udávána adresa, na které je přítomen objekt, jehož metodu voláme, znamená v některých případech silné zatížení sítě. V některých případech je totiž nutné rozesílat zprávy všem uzlům sítě.

spíše jen to, že ke starším notacím přibyla další. Sjednocení založené na hlubší unifikaci a integraci metodik není zatím v dohledu a možná není ani v plném rozsahu možné. UML je podporován většinou současných systémů CASE. Stále však platí, že vazby mezi diagramy různých typů nejsou dostatečně podporovány, např. vazby mezi diagramy přechodů a diagramy tříd.

12.7.3 Nezávislost implementací metod objektů

Implementace metod tříd závisí na tom, kde je objekt volající metodu, např. na klientu nebo na serveru, a kde je objekt vlastnící metodu. To ukazuje, že implementace metody by měla být závislá i na dalších parametrech, nejen na definici odpovídající třídy, která v podstatě stanovuje pouze rozhraní, např. způsob volání metod. Tuto úvahu můžeme dovést ještě dále. Je dokonce možné, aby implementace mohla být v různých programovacích jazycích, případně pro různý základní software. Programovací jazyky používané při objektově orientované analýze a návrhu nemusí být nutně objektově orientované. Objektovou orientovanost lze stimulovat i např. v jazyce COBOL. Je samozřejmě výhodné používat objektově orientovaný programovací jazyk, někdy to ale není možné, např. při modifikaci starších systémů.

Generace IS pak může probíhat následovně:

a) Určí se třídy tvořící daný IS.
b) Pro každou třídu se určí, zda bude na klientu či serveru; je přípustná obojí možnost.
c) Zvolí se jazyky implementace pro klienty a pro servery.
d) Na základě vazeb se vyberou z depozitáře správné implementace a vytvoří se zdrojový program.
e) Zdrojové programy se přeloží a sestaví s knihovnami doby běhu.

Tento přístup je možný, jen jsou-li k dispozici vhodné varianty implementace tříd. Jedna z možností je generace implementací pomocí vhodného nástroje podobného CASE nástrojům nízké úrovně (kap. 19). Příkladem komerční realizace tohoto přístupu je IS firmy Lawson Software a některé prostředky systému PowerBuilder.

13

Od kódování přes předání k provozu a údržbě informačního systému

13.1 Kódování

Etapa kódování patří k nejméně pracným a díky pokroku v programovacích nástrojích a jazycích také k nejméně problémovým etapám vývoje softwaru (kap. 15). Podíl kódování se v současné době dále snižuje díky následujícím faktorům:

a) Zavedení 4GL jazyků pro práci s databázemi.
b) Objektově orientované metody a objektově orientované programovací jazyky.
c) Metody vizuálního programování (Visual Basic, Visual C++).
d) Využívání vývojových prostředí integrujících využití všech výše uvedených faktorů, někdy i v kombinaci s CASE nástroji a vývojovými systémy (Delphi, Optima++, Developer 2000 atd.).
e) Stále širší využívání customizovaných IS.
f) Rozvíjející se možnosti integrace produktů třetích stran.
g) Rozvoj prostředků spolupráce aplikací.

Podíl kódování dnes nepřesahuje 25 % pracnosti vývoje či customizace. Přesto nelze problém kódování podceňovat. Většina softwarových firem totiž musí při vývoji i při customizaci vyvíjet software. Znalost programování je dobrým základem kvalitní specifikace požadavků. Kvalita programů silně ovlivňuje pracnost testování a náklady na údržbu.

Z hlediska technik programování jsou důležité objektová orientace a v nejnovější době vizualizace v kombinaci s objektovým přístupem. Vizuální programování umožňuje snadnou generaci programů uživatelského rozhraní. Obrazovka se graficky navrhne a pak se specifikují akce pro tlačítka, kontroly polí a nápovědi. Vizuální programování se osvědčuje při vývoji softwaru spolupracujícího s databázemi (intuitivní sestavování SQL dotazů s podporou grafického obrazu datového modelu a vyplňováním formulářů atd.). Vizuální programování lze používat při vývoji procesně nenáročných aplikací. Ve složitějších případech se používá v kombinaci s jinými postupy.

Vizuální programování se obvykle kombinuje s objektově orientovaným přístupem. Vizuálním způsobem jsou efektivně vytvářeny třídy pokrývající ty funkce systému, které jsou pracné, avšak které nejsou konceptuálně příliš složité. Typickým příkladem je graficky orientované uživatelské rozhraní – GUI. Jiné části systému bývá nutné vytvořit „klasickým“ způsobem. Při vizuálním programování se obvykle z grafické formy algoritmu generuje

meziprodukt ve formě „klasického“ programu, např. v C++. Vizuální programování velmi urychluje vývoj některých aplikací. To však neznamená, že není třeba dokumentace, jak se občas stává.

Ve zbytku této kapitoly se nebudeme zabývat vlastními technikami programování; jsou dnes v podstatě zvládnuty. Zaměříme se na problémy související s vedením projektu v etapě kódování.

13.1.1 Volba programovacího jazyka

Volba vyššího programovacího jazyka má na rychlost kódování a do značné míry i na pracnost testování poměrně malý vliv. Programování v assembleru je však nepoměrně pracnější. Produktivitu práce zvyšují moderní programovací prostředí a CASE systémy vytvářející ucelený systém integrující všechny etapy životního cyklu. To přináší podstatné výhody při údržbě a modifikacích. V současné době se šíří a používají následující jazyky[1].

FORTRAN 77, FORTRAN 90, FORTRAN 95.

Jazyky vhodné pro vědeckotechnické výpočty i na paralelních hardwarových architekturách. Pro IS mimo vědeckotechnickou oblast nejsou příliš vhodné s možnou výjimkou jazyka FORTRAN 95.

Jazyk C.

Byl vytvořen jako nástroj programování a přenosu operačního systému UNIX schopný nahradit v mnoha směrech i assembler. Je to v současné době velmi rozšířený jazyk používaný pro systémové programování, grafiku a některé části IS. Je poměrně záludný a je proto vhodný pouze pro profesně zdatné programátory.

Jazyk C++.

Objektově orientovaný jazyk původně koncipovaný jako nadstavba jazyka C nahrazující v řadě směrů i assembler. Je vhodný pro systémové programy. Vhodný pro profesně zdatné programátory. Existují vizuální varianty programování v C++ a integrovaná vývojová prostředí pro vývoj programů v C++. Programování v C i C++ je poměrně pracné a vyžaduje dosti vysokou zručnost a zkušenosti. C++ patří mezi jazyky jejichž možnosti rozvoje se dosud plně nevyčerpaly. Existují dobré standardy pro C++. Z hlediska metod programování má C++ řadu nectností.

Pascal.

Jazyk Pascal je jádrem úspěšného integrovaného vývojového prostředí Delphi firmy Borland. Delphi zahrnuje prostředky vizuálního programování, vazby na databázové systémy (SQL) a nástroje testování a dokumentace. Podpora jazyka Pascal od SW firem slábne.

Ada.

Jazyk Ada byl vyvinut v rámci projektu amerického ministerstva obrany jako prostředek programování systémů reálného času. V této oblasti je bez podstatné konkurence. Systém Ada zahrnuje rozsáhlý výběr podpůrných prostředků. Jazyk Ada je dobře standardizován a v nové verzi obsahuje objektově orientované rysy. Jazyk Ada se používá i jako specifikační jazyk, viz normu IEEE 990 – 1987 (1992).

4GL jazyky.

4GL jazyky jsou při vývoji IS široce používány a osvědčují se. Nevýhodou 4GL jazyků je to, že nebyly standardizovány a většinou jsou integrovány pouze s konkrétním databázovým systémem. Tuto nevýhodu zmenšuje to, že 4GL jsou procedurálními nadstavbami jazyka SQL, který je standardizován. Pro daný databázový systém jsou však optimalizovány a dovedou velmi dobře využívat jeho vlastnosti. 4GL jazyky jsou zahrnuty i do některých integrovaných vývojových systémů podporujících vizuální programování a intuitivní způsoby práce s databázemi.

1. Moderní programovací jazyk závisí do značné míry na vývojových prostředích, které jej podporují. Dobrým příkladem je Delphi pro jazyk Pascal

Přední databázové firmy investují do integrovaných vývojových prostředků pro 4GL jazyky značné prostředky. Nová norma SQL pokrývá mnoho operací, které bylo nutno dříve programovat ve 4GL jazycích.

Smalltalk.
Smalltalk je první obecně rozšířený důsledně objektově orientovaný jazyk. Prosazuje se i při vývoji IS. Vývoj programů v jazyce Smalltalk je částečně podporován vizuálními prostředky. Aplikace v jazyce Smalltalk bývají méně efektivní. Vazby na databáze jsou řešeny knihovnami tříd, úspěch aplikace proto závisí na kvalitě těchto knihoven.

Visual Basic.
Vizuální programování kupodivu nalilo novou krev do žil zdánlivě odepsanému jazyku Basic. Je vhodný spíše pro malé aplikace s grafikou.

COBOL.
Jazyk COBOL byl ze svého dříve monopolního postavení při programování IS vytlačen především 4GL jazyky. Doplatil trochu na svoji konzervativnost. Pro aplikace ve finanční sféře je COBOL stále ještě považován, asi právem, za nejspolehlivější nástroj. V jazyce COBOL bylo napsáno mnoho bankovních IS. Nejnovější norma jazyka umožňuje objektově orientované programování. Podpora nových verzí jazyka ze strany velkých výrobců je váhavá.

PROLOG, Lisp.
Tyto jazyky jsou vhodné pro specifické aplikace v oblasti umělé inteligence a pro některé techniky vytváření prototypů.

Eiffel
Kvalitní, avšak málo rozšířený objektově orientovaný jazyk.

Java
Je inspirován jazykem C++, neobsahuje však některé nebezpečné konstrukce a obsahuje některé výhodné konstrukce. Je to univerzální programovací jazyk používaný především pro práci s Internetem.

Jazyky FORTRAN, COBOL, Ada, Pascal, C se nazývají procedurální (3GL) jazyky, C++, Smalltalk a Java jsou objektově orientované jazyky umožňující objektově orientované programování. Objektově orientované rysy mají i nové verze jazyků Ada, COBOL a Pascal.

Řada nástrojů umožňuje vizuální programování pro více programovacích jazyků. Integrované prostředky, např. Delphi, podporují tvorbu aplikací s architekturou klient – server. Volba programovacího jazyka a programového prostředí závisí na řadě faktorů. Nejdůležitější je snadnost údržby a přenositelnost. Rozvoj rozlehlých počítačových sítí vytváří potřebu programovacích prostředků pro aplikace v síti. Nejznámějším pokusem v tomto směru je jazyk Java. Příklad jazyka Java ukazuje, že nový programovací jazyk má šanci na úspěch, pokrývá-li nějakou novou potřebu nebo podporuje nové metody. Podle analogie s biologií říkáme, že jazyk obsadil volnou niku. Úspěch nového jazyka v klasických oblastech obsazených zavedenými jazyky je z více důvodů nepravděpodobný. Nika je obsazena.

Jak je uvedeno výše, volba vyššího programovacího jazyka obvykle neovlivňuje zásadním způsobem produktivitu práce při vývoji softwaru. Volba vhodného programovacího jazyka však podstatným způsobem ovlivňuje způsob myšlení členů týmu a má značný vliv na rozsah prací při údržbě. Stále se používá programování v jazyce symbolických adres. Jazyk symbolických adres – assembler – je nutné použít v následujících situacích:

1. Ostrá omezení na dobu odezvy a využití paměti vylučující použití jazyka vyšší úrovně.
2. Při testech hardwaru bývá nutné zadávat libovolné – i nepřípustné – sekvence instrukcí.

3. Pro jednoúčelový hardware je vytvářen jednoúčelový program nepříliš velkého rozsahu. Příkladem jsou ovladače ve spotřební elektronice, kdy se nevyplatí vyvíjet překladač z vyššího programovacího jazyka a jsou ostré požadavky na rozsah programu a jeho efektivnost.
4. Je možné použít vyšší programovací jazyk, ale je nutné rozšířit jeho sémantiku, např. doplněním pojmu procesu do jazyka C++ nebo doplněním ovladačů nestandardních periferií. Jiným důvodem může být potřeba zajistit dostatečně rychlou odezvu naprogramováním kritických částí v assembleru.
5. Software byl navržen tak, aby mohl být snadno přenášen na nové počítače. Při přenosu je však často nutné naprogramovat v assembleru ty části, které zajišťují rozhraní na hardware nového počítače (drivery, tabulky generátorů kódu v překladačích atd.). Při tvorbě IS je použití assembleru pravděpodobné pouze v těch případech, kdy je vyvíjeno rozhraní na úrovni přímého ovládání technologických procesů. Programování v assembleru je podstatně pracnější než programování ve vyšších programovacích jazycích (kap. 15). Assembler je stále více vytlačován specializovanými programovacími jazyky vyšší úrovně, především jazykem C.

Při volbě vyššího programovacího jazyka musíme uvážit následující fakta:

1. *Požadavky zadavatele.* Zadavatel může požadovat určitý programovací jazyk nebo dává určitou omezenou možnost výběru. Důvodem takového požadavku může být snaha vyhovět předpisům. Rozumným důvodem požadavku na použití určitého programovacího jazyka může být potřeba, aby byl použitý programovací jazyk známý programátorům všech řešitelských týmů, případně programátorům zadavatele, kteří budou systém po určité době udržovat. Jiným důvodem může být potřeba zachovat jazyk aplikace při modifikacích.
2. *Vhodnost jazyka pro danou aplikaci.*
3. *Úroveň standardizace* (kvalita norem).
4. *Rozšířenost a podpora předních výrobců.*
5. *Kvalita vývojových prostředků* (vizuální prostředky, samodokumentující rysy, podpora ladění a testování, správa verzí, vývojová prostředí).
6. *Rozsah projektu.* Pro opravdu velké projekty může být výhodné vytvořit vlastní realizační jazyk. Příkladem je jazyk C a operační systém UNIX. Pro větší projekty musí být použitý kompilátor dostatečně rychlý, s dobrou diagnostikou chyb a dobrými prostředky pro výpočet křížových odkazů. Použijeme-li nějaký univerzální preprocesor nebo makroprocesor pro assembler, můžeme být časem velmi omezováni tím, že makroprocesory bývají pomalé a mají i některá další omezení, např. v syntaxi příkazů. Pak je vhodné navrhnout nějaký jednoúčelový programovací jazyk a napsat pro něj kompilátor.
7. *Znalosti realizátorů softwaru.* Je jistě výhodné nenutit programátory, aby se učili nový jazyk. Pro zkušeného programátora však není zvládnutí nového jazyku podstatný problém. Je rovněž žádoucí, aby byla většina projektů realizovaných softwarovou firmou napsána ve stejném jazyce a za použití stejných vývojových nástrojů. Zvyšuje to, zvláště při objektově orientovaném přístupu, možnosti opakovaného použití jednou vytvořených částí programů a snižuje to množství chyb programátorů, kteří si nemusí neustále uvědomovat rozdíly ve významu obdobných konstrukcí v různých jazycích, a ulehčuje to údržbu.
8. *Požadavky na přenositelnost.* Použitý prostředek by měl být nezávislý na hardwarové platformě a pokud možno použitelný se všemi hlavními operačními systémy.
9. *Použitelnost na zvoleném hardwaru.*
10. *Knihovny podprogramů* nebo *tříd.*

Riskantní je závislost na konkrétním dodavateli softwarových vývojových nástrojů. Volbou vývojového prostředí Delphi činíme předpoklad, že dodavatel, firma Borland, bude moci tento produkt delší dobu podporovat, že se tedy nedostane do potíží. Vizuální nástroje a integrovaná vývojová prostředí nejsou standardizovány. To

znamená vyšší závislost na dodavateli. Je důvodný předpoklad, že např. vizuální programování je výhodné jen pro jistý typ úloh, např. pro programování grafického uživatelského rozhraní.

13.1.2 Užitečné zásady psaní programů

Psaní programu navazuje na proces jeho návrhu. Tento proces můžeme chápat jako postupné zpřesňování popisu realizace tím, že vyšší funkce vyjadřujeme jako superpozici funkcí nižší úrovně.

Vývoj od nejvyšších úrovní můžeme použít i při psaní programů. Strukturovaný vývoj je obecnější metodický přístup, který se týká techniky vzniku systému ve formě hierarchického systému modulů nebo částí. Pokud při psaní modulů nepostupujeme odshora, musíme předpokládat, že návrh modulů nižších úrovní je tak kvalitní, že nebudeme muset daný modul v důsledku neočekávaných okolností zjištěných na vyšší úrovni měnit. Proto raději navrhneme modul, který realizuje spíše obecnější funkce. Pokud píšeme moduly počínaje nejvyšším, stává se méně často, že je nějaký modul třeba přepisovat. To je kromě jiného způsobeno tím, že vyšší modul používá obvykle několik modulů nižší úrovně a tvoří tedy programové prostředí pro více programových jednotek. Návrh a psaní programů shora dolů umožňuje zároveň „lepší kontakt se zadáním“ – snáze se dodržuje zásada, že se realizuje právě to, co je třeba. Pokud jsou obavy, že s některými moduly budou potíže, je možné případná rizika zmenšit tím, že se přednostně programuje kritický modul a moduly s ním spolupracující, případně moduly jemu nadřazené.

V technologii objektů výše uvedená zásada znamená realizaci systému počínaje třídami, jejichž metody zajišťují funkce nejvyšší úrovně, a seskupování těchto tříd do vyšších celků. Ve výjimečných případech je výhodné programování (návrh) provádět i od jiné než nejvyšší úrovně. To bývá v případech, kdy je výhodné, což bývá u velkých systémů, vyvíjet obecně použitelné programové prostředky: správu dat, prostředky testování, specializované archivační systémy atd. Budování takových systémů pak může být součástí celkové strategie rozvoje informačních systémů v podniku. Nejobtížněji odladitelné defekty ve větších programech vznikají tím, že okolí programu nedodrží přijatá, nezřídka však jasně neformulovaná pravidla, např. o vstupu dat, době provedení atd. Podobného typu jsou chyby ve vazbách mezi moduly uvnitř programu, jako je nedodržení tvaru parametrů při volání podprogramů atd. Osvědčuje se doplnit program o vypínatelné kontroly správnosti vstupů, výstupů, parametrů volání podprogramů atd. Tyto kontroly je možné vyřadit u systému, který již považujeme za odladěný. Žádný velký program však nemůžeme nikdy považovat za dokonale odladěný. Každý delší dobu používaný program se modifikuje a modifikace mohou do programu zanést chybu. Pokud k tomu nejsme nuceni hardwarovými omezeními, není příliš rozumné kontroly vyřazovat. Na úrovni dekompozice je vhodné provádět především kontroly rozhraní chránící programy před zavlečením chyb z okolí. Ochrana programů před zavlečením chyb z okolí se nazývá *defenzivní programování*.

Některá vývojová prostředí mají prostředky umožňující „podmíněný“ překlad části programů - některé vhodným způsobem označené části programu se mohou zadáním vhodné informace kompilátoru vynechávat nebo programovací jazyk obsahuje prostředky pro testování vstupů prostřednictvím tzv. výjimek. Např. je-li pro vstupní data splněna jistá podmínka, vyvolá se reakce systému, podobně jako např. při dělení nulou, viz jazyk Ada. Tento princip byl dále rozvinut v systémech řízených událostmi a v *aktivních databázích*, ve kterých vznik určité situace vždy vyvolá odpovídající akci.

Podmínky pro snadnou realizaci defenzivního programování je třeba vytvořit již v etapě návrhu systému a z části i v etapě specifikace požadavků, především při návrhu rozhraní mezi částmi systému. Pro defenzivní programování je výhodné vytvořit vhodné softwarové nástroje. Defenzivní programování je zvláště vhodné při objektově orientovaném přístupu. Při programování v týmu se často stává, že některá změna operace nebo funkce může být realizována podstatně snáze jinde, než se původně předpokládalo, nebo že zjištěnou závadu může snáze

napravit někdo jiný, než ten, kdo ji způsobil. V tom okamžiku je důležité, aby ten, jemuž tato změna či oprava přidělá práci, byl ochoten tuto práci přijmout, tj. aby stavěl zájem týmu a cílů nad své okamžité nepohodlí. Je věcí vedení projektu, aby nebyl nikdo z tohoto důvodu přetěžován. Je rovněž třeba, aby členové týmu neprosazovali na úkor celkového úspěchu svoje názory a řešení a programy psali tak, aby byly srozumitelné i ostatním, aby bylo v principu možné, aby programy dokončili i ostatní. Je tedy nutné postupovat neegoisticky – odtud i název *neegoistické programování* (srv. kap. 10).

Při psaní programů se vyplatí používat standardizovanou úpravu komentářů a přehledné rozvržení textu programů s případným využitím automatických prostředků. Každý program by měl být po odladění včetně dokumentace formálně převzat. Je nutné stanovit formální postup formulace požadavků na změny, jejich odsouhlasení a provedení. Jen tak lze u větších projektů udržet pořádek (kap. 20). Při programování a oponenturách modulů je vhodné začínat od těch modulů nebo tříd, které:

a) Mohou být využity při vývoji ostatních modulů nebo tříd. To bývají softwarové nástroje a základní služby realizovaného systému. Obvykle to znamená programovat shora dolů.
b) Obsahují pravděpodobně chybu. Z oponentur požadavků a návrhu systému je známo, že některé moduly nebo třídy jsou obtížně realizovatelné, neboť obsahují složité algoritmy či byly zdrojem obtíží při návrhu atd. Takové části je třeba programovat a testovat přednostně.

13.1.3 Řemeslo programátora a programovací jazyky

Psaní programů má mnoho rysů řemesla – nestačí vědět, jak se má programovat, ale musíme se převážně praktickou činností, programováním, naučit programovat. Něco jiného je vědět, a něco jiného je také prakticky umět. Hlavním nástrojem programátora je programovací jazyk a jeho vývojové prostředí. Zkušený a schopný programátor dokáže napsat dobrý program i v jazyce, který není pro daný účel příliš vhodný. Neschopný nebo nedostatečně připravený programátor dokáže napsat špatný program i v tom nejlepším jazyce. Pro zkušeného programátora tedy není to, v jakém jazyce píše programy, příliš důležité. Má-li však tu možnost, sáhne zkušený programátor po lepším nástroji. Jako v každém řemesle vyžaduje zvládnutí správných technik s kvalitními nástroji větší počáteční úsilí. Často se zdá, že by bylo výhodnější použít „méně čisté“ metody a neučit se ovládat komplikované nástroje – v jednoduchých případech se úkol vždy nějak zvládne a ve složitějších případech použijeme to, co je třeba. Je to však omyl. Staří mistři dobře věděli, jak je v každém řemesle důležité zvládnutí správných pracovních návyků a také správných způsobů práce s nástroji. Každý chápe, že k postavení kůlny snad stačí jedna tupá sekera, ke stavbě střechy větší stavby je však třeba tvrdá výuka, dobré nástroje, dobrý projekt a také fortel.

Ze zkušeností starých mistrů je také známo, že připustí-li se, aby učeň neprošel tvrdou školou výuky pracovních návyků a začal neuměle hned dělat něco „užitečného“ (stavět kůlny), je malá naděje, že z něho vyroste dobrý řemeslník. Bude z něho s velkou pravděpodobností pouze „fušer“. Abychom příměr dokončili, je samozřejmě zbytečné, aby se každý vyučil tesařem, většině bude stačit umět stlouci kůlnu. Avšak pro ty, u nichž se předpokládá, že budou profesionály, není příprava pouze na jednoduché práce vhodná. Profesionální nástroje dneška mají charakter integrovaných prostředí podporujících tvorbu dokumentace, specifikací a návrhu (CASE) a různé varianty práce s knihovnami. V případě IS převažuje orientace na kvalitní databázové systémy a integrovaná vývojová prostředí obsahují silné nástroje tvorby dokumentace, podpory testování, různé varianty metod programování (virtuální, znakové) a obvykle i podporu objektové orientace. Zvládnutí integrovaných nástrojů není snadné, vyžaduje mnoho práce a předpokládá poměrně vysokou kvalifikaci. Integrovaná prostředí nejsou standardizovaná a rychle se vyvíjejí. Doba mezi podstatnými inovacemi nepřesahuje několik málo let. Je tedy nutné zvládat stále nové nástroje.

Změna programovacího jazyka a vývojového prostředí znamená dosti velké riziko a zvyšuje pracnost vývoje. Přesto lze tvrdit, že změna vývojového prostředí nepředstavuje základní riziko. Role programovacích jazyků se asi dosti přeceňuje. Jako příklad uveďme zkušenost prof. Malíka z Matematicko-fyzikální fakulty KU Praha. Prof. Malík vyvinul v jazyce Simula 67, což je jazyk s objektovými rysy vhodný pro simulace, simulační model lidského srdce. Model byl tak dokonalý, že umožňoval nejen verifikovat diagnózy, např. místa poškození srdce při infarktu, ale také optimálně navrhovat funkce kardiostimulátorů.[2] Prof. Malík byl nucen přeprogramovat model do jazyka FORTRAN 77, který je pro takový účel dosti nevhodný. Přeprogramování systému o mnoha tisících řádků trvalo pouhé dva měsíce.

13.2 Testování

Nově napsané programy vždy obsahují chyby. Je proto nezbytné programy otestovat s cílem nalezení chyb nebo ověření, že systém funguje tak, jak má. Programy se testují tím, že se spouštějí pro testová data a ověřuje se, zda pro zadaná data pracují správně. Pro každý případ chybné práce (selhání, failure) je třeba nalézt příčinu. Testování programů je tedy proces, ve kterém se zjišťují selhání, příčiny selhání se lokalizují, čímž se zjistí místa v programech a zprostředkovaně v dokumentech (defekty), které je třeba změnit.

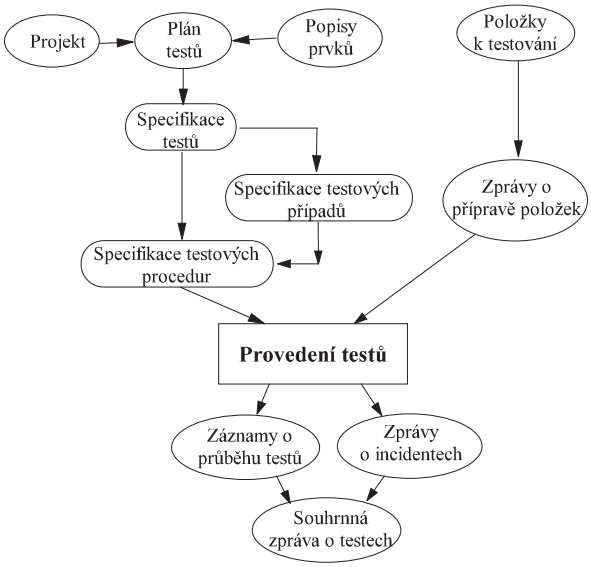
Opravený program se testuje a opravuje (ladí) dokud v něm testy odhalují selhání, přesněji dokud frekvence selhání systému neklesne pod určitou mez. Pro úspěšné ladění je třeba vytvořit podmínky již v předchozích etapách životního cyklu, především při návrhu systému a do značné míry i v etapě specifikace požadavků. Programy musí být navrženy tak, aby byly testovatelné. Výsledkem návrhu systému by měl být mimo jiné i plán testů umožňující snadný návrh a provedení testů (obr. 13.1).

Etapa ladění je velmi pracná, podstatně pracnější než psaní programů. Etapa ladění je všeobecně považována za nejobtížnější ze všech etap vývoje softwaru – je málo obecně použitelných metod testování a lokalizace chyb, a pokud se něco při testování jednoznačně osvědčuje, je to dokumentace testů: knihovny testů, plán testů, včasná příprava testů atd.

U velkých systémů jsou programy vyvinuté pro testování a lokalizaci defektů často rozsáhlejší než vlastní program. Problém testovatelnosti musí být řešen již při specifikacích požadavků. Testování probíhá v několika etapách – testování částí, testování při integraci, testování při předávání. Otestované části se po provedení testů částí postupně spojují do vyšších celků a celky se testují (integrační testování). U částečně realizovaného systému lze testovat správnost provádění zadaných funkcí. Pak se systém předvede uživateli a předá (testování při převzetí). Moderní vývojová prostředí nabízejí řadu nástrojů podpory testování a lokalizace. Výsledkem návrhu systému musí být podklady pro návrh plánu testů, vypracování testových případů (zadání příkladů) a testových procedur (posloupností testových případů). Testování využívá i výsledky oponentur všech etap vývoje softwaru, především oponentury programů, jako jsou inspekce a čtení kódu, a akce dohledu (kap. 8). U větších systémů se před prováděním testů provádějí minimálně tyto oponentury:

1. Oponentura požadavků na software.
2. Oponentura předběžného návrhu systému.
3. Oponentura návrhu softwaru (proveditelnost, pracnost, prověření, zda návrh odpovídá požadavkům).
4. Oponentura plánu testů (úplnost, proveditelnost, konzistentnost atd.).

2. To přineslo nemalou úlevu pacientům a bylo i komerčně úspěšné.



Obr. 13.1: Dokumenty potřebné k provedení testů a jejich návaznosti (vhodné pro reálné úkoly, podle ANSI 94).

13.2.1 Činnosti při testování

Na základě plánu testů se pro jednotlivé položky seznamu částí softwaru (modulů) specifikuje:

1. Jaké testové případy se uvažují. Testové případy mohou být součástí údajů o testu nebo mohou být vedeny ve specializované knihovně a pak se v popisu testu objeví pouze odkaz na testovaný případ. Testové případy jsou obvykle zadávány formou:
   - vstupy,
   - příkazy,
   - očekávané reakce systému,
   - výstupy.
2. Jaké testové procedury se předpokládají.
3. Popis vlastního testu. Popis obsahuje
   - cíle testu,
   - podmínky provedení,
   - způsob provedení,
   - obsah testu,
   - soubor testových procedur,
   - pravidla přijetí/zamítnutí testu,
   - pravidla pro přerušení testování,
   - rizika spojená s provedením testu.

Po provedení testových procedur se vypracují zprávy o zjištěných nedostatcích. Je vhodné průběh testů zaznamenávat do souboru test-log neboli žurnálu testů. Na závěr se vypracuje souhrnná zpráva o testech. Zpráva o zjištěném selhání/nedostatku má u testů tvar:

1. Identifikátor testu.
2. Zkrácený popis problému (abstrakt).
3. Podrobný popis problému
   - vstupy,
   - očekávané výstupy,
   - skutečné výsledky,
   - zjištěné anomálie,
   - doba kdy zjištěno,
   - v jakém prostředí testováno, např. použitý operační systém,
   - použité nástroje pro testování,
   - v které fázi testu zjištěno,
   - pokusy o opakování a jejich výsledky,
   - kdo testoval.
4. Důsledky selhání pro plán testů, specifikaci testů a testových případů; co doplnit nebo upravit.
5. Jak se dále pokračovalo: testování přerušeno, provedl se test *t* atd.

Souhrnná správa o testech obsahuje:

1. Zprávy o předání položek k testování.
2. Žurnál testů – záznamy o průběhu testů (test-log – v podstatě záznam kroků testů s relevantními informacemi o softwarovém prostředí, vstupech, výstupech atd.).
3. Zprávy o chybách nebo jejich shrnutí.
4. Souhrnná zpráva o provedení testů: co se vše testovalo, jaké jsou výsledky, zda produkt přijmout či nepřijmout.

Souhrnná zpráva o testech může být do značné míry vytvořena nástroji pro testování. Plán testů velkých softwarových produktů mívá následující strukturu (ANSI 94):

1. Identifikátor plánu testů.
2. Stručný popis cílů plánu: shrnutí úkolů, odkazy na relevantní dokumenty, přehled testovaných rysů.
3. Seznam částí softwaru, které se testují.
4. Testované funkce produktu.
5. Netestované rysy produktu, mohou-li být v této věci pochybnosti.
6. Kritéria přijetí /zamítnutí systému jako celku.
7. Stručná charakteristika jednotlivých testů.
8. Způsob realizace, pořadí testování a integrace částí, termíny.
9. Důvody přerušení testování a zajištění pokračování přerušených testů.
10. Seznam a termíny vyhotovení dokumentů k testům.
11. Požadavky na programové prostředí a prostředky testování.
12. Odpovědnosti a personální zajištění.
13. Termíny provedení testů a jejich návaznosti.
14. Rizika spojená s provedením testů.

Snadnost nebo obtížnost testování závisí do značné míry na architektuře systému, na použitých programovacích technikách a vývojových prostředích, kvalitě průvodní dokumentace a nástrojích testování. Systémy založené na nějaké metodě výměny zpráv nebo souborů a dekomponované do malých částí se testují snáze než systémy nedekomponované.

Návrh testů vždy vyžaduje značnou intuici. Důvodem je fakt, že řada úkolů, které si klademe při testování, patří mezi tzv. „algoritmicky nerozhodnutelné problémy“ (Davis, 1983). Příkladem nerozhodnutelného problému je např. požadavek, aby se při provedení testu prověřily (provedly) všechny části programu. Dá se ukázat, že neexistuje program, který by pro každý jiný program ověřil, zda neexistují instrukce, které nemohou být nikdy provedeny (mrtvý kód). Takový problém musí řešit člověk případ od případu znovu. Jde o v jistém smyslu tvůrčí problém, který nelze řešit mechanicky.

Algoritmicky nerozhodnutelných problémů existuje při testování mnoho. Při návrhu testů je proto méně možností postupovat podle nějaké univerzální, jednou provždy stanovené metodiky. To je důvod obtížnosti úkolu testování. Při návrhu testových případů je třeba, aby testové případy pokrývaly všechny rozdílně zpracovávané vstupy; pro každý vstup s logicky různým zpracováním by měl být navržen alespoň jeden testový případ.

Testové případy by měly zahrnovat zpracování „extrémních“ hodnot – např. soubor prázdný, s jednou větou, zpracování první, prostřední a poslední věty souboru, krajní hodnoty indexů, mezní hodnoty proměnných – a také způsob zpracování chybných vstupů – dělení nulou, hodnoty mimo povolený rozsah atd.

13.2.2 Organizační zabezpečení testů

Cílem testů je dosáhnout toho, aby software neobsahoval chyby. K dosažení tohoto cíle je samozřejmě třeba, aby byly testy koncipovány s cílem odhalit chybu. Jinými slovy navrhovatel testů musí testy koncipovat tak, jako by bylo jeho cílem dokázat, že program není správný. Pro realizátora softwaru bývá psychologicky obtížné převzít roli, ve které se má snažit dokázat, že jeho dílo není dokonalé. Dobří programátoři jsou často schopni roli nemilosrdného testéra převzít. Ve větších týmech (a u větších projektů) je obvyklé, že návrh testů, jejich příprava a provedení je svěřeno jiným pracovníkům než realizátorům softwaru. Vzniká tak služba testérů (kap. 10).

Testy a informace o jejich provedení jsou velmi cenným výsledkem vývoje softwaru. Vytvořená zásoba testů může sloužit pro kontrolu, zda při úpravách softwaru nedošlo ke zhoršení vlastností systému (regression testing), a jsou i cenným nástrojem při přenosu softwaru na jiný počítač. Podklady pro testy a programy testů musíme tedy chápat jako základní součást projektové dokumentace softwaru. Jsou předpokladem pro atest (certifikát) podle normy ISO 9000–3.

Praxe ukazuje, že při testování částí je někdy výhodná znalost struktury částí. V tom případě může být výhodné, aby některé testy částí navrhl a případně provedl programátor. U větších projektů je obvykle nutné, aby části testovali i nezávislí testéři bez znalosti vnitřní struktury částí (testování černých skříněk – black box testing). Testy integrační a funkční bývají u větších systémů navrhovány a realizovány výhradně nezávislými testovacími týmy. Testéři mohou pracovat zcela nezávisle. Je to sice účinné, ale velmi drahé. U IS se proto doporučuje, aby testéři úzce spolupracovali s ostatními členy týmu při přípravě, provádění a vyhodnocování testů.

Výsledkem práce při návrhu, realizaci a provedení testů by měly být softwarové nástroje umožňující snadné provedení testů a vyhodnocení výsledků testů i bez přítomnosti členů testovacího týmu.

V některých případech je úspěšné provedení testů podmínkou přiznání práva užívat název programovacího jazyka, který je ochrannou známkou. Tak např. název programovacího jazyka Ada lze používat, jen když je proveden soubor testů majitele ochranné známky Ada – ministerstva obrany Spojených států amerických. Jiné testové soubory – benchmark – byly vytvořeny nezávislými organizacemi pro nezávislé ověření správnosti

funkce a výkonnosti kompilátorů. Použití těchto testových souborů přinutilo řadu výrobců podstatně zkvalitnit kompilátory jazyka FORTRAN. Nevýhodou takových souborů testů je, že nejsou zaměřeny na lokalizaci místa chyby, podobají se spíše přejímacím testům.

13.2.3 Integrace

Po testech částí se části spojují a testuje se celek. Pokud se spojení provede naráz (metoda velkého skoku), pozdě se odhalí nedostatky v rozhraních, mnoho programů se napíše, aniž se vyzkouší, zkoušení modulů ve „skutečném prostředí“ se odkládá atd. Proto se používají metody „postupného nabalování“, kdy se části poměrně brzy spojují do vyšších funkčních celků. To umožňuje omezit rozsah pomocných simulačních programů, neboť nové moduly mohou být testovány pomocí dříve integrovaných částí. Metody spojování (integrace) jsou založeny na grafu vztahů mezi částmi daných relací „A potřebuje B“.

Integrace zdola začíná od modulů, které nepotřebují jiné moduly, a postupně se přechází na moduly, které potřebují pouze integrované moduly. Je to poměrně dobrá metoda, vyžaduje však mnoho pomocných programů, které generují data pro prověřené moduly. Poměrně pozdě se ověřují funkce systému, což je nevýhodné, neboť se opožďuje předvedení systému uživateli.

Integrace shora začíná od modulů, které nejsou potřeba v jiných modulech, a moduly, které jsou potřeba v daném modulu, se simulují. Podřízené moduly se naprogramují později a připojí do vznikajícího systému.

Výhody: Testují se cílové funkce, dá se dříve odhalit, zda lze úkol splnit či ne, testují se dříve rozhraní.

Nevýhody: Simulace nižších úrovní může být obtížná, je tendence programovat nejvyšší úroveň dost brzy, a pak se jen obtížně odhodláváme dělat opravy při výskytu problémů na nižších úrovních. Moduly jsou testovány jen ve svém okolí a nejsou tudíž univerzálně použitelné, což může být někdy i výhoda.

Modifikovaná metoda integrace shora. Při postupu shora se může stát, že modul, který je kritický[3], bude integrován a tedy zkoušen příliš pozdě. Pak lze postupovat shora, avšak tak, že v každé úrovni integrujeme pouze moduly nadřízené kritickému modulu. Někdy je nutné podobnou operaci provést i pro všechny moduly podřízené kritickému modulu.

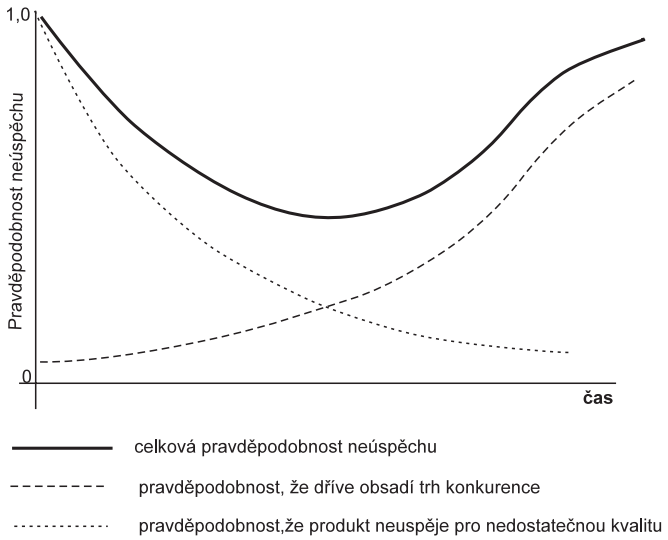
Metoda sendviče. Systém se rozdělí – vyšší část se integruje shora, nižší zdola. Tuto metodu je vhodné používat při programování operačních systémů nebo souborů programů. Uživatelské moduly se integrují zdola, společné služby shora. Moduly vyšší úrovně se mohou testovat nejprve každý zvlášť, avšak můžeme se dostat do situace podobné vzniku dvou tunelů při ražení z obou stran – uprostřed se jaksi nesejdeme.

Volba metody integrace závisí na realizovaném systému, počtu pracovníků, rozsahu pomocných prací, jistotě, že je projekt správný, možnosti likvidace pracovních špiček.

Integrační testy se provádějí obvykle za situace, kdy ještě není celý systém oživen. Integrační testy nejsou obecně identické s testy funkcí systému (function tests), při kterých se s využitím specifikace požadavků a systému ověřuje správnost realizace cílových funkcí systému. Testy funkcí se obvykle realizují na kompletním systému. Pokud je systém vhodně navržen, lze funkční testy provádět i na zčásti realizovaném systému. Je-li např. systém navržen ve vrstvách, lze funkce jednotlivých vrstev testovat do značné míry vzájemně nezávisle.

Je-li systém navržen jako soubor programů spolupracujících pomocí předávaných dat, je možné testovat jednotlivé programy zadáváním dat (zpráv, příkazů). Jednou z podstatných výhod objektově orientovaných programů je relativně snadná dekompozice systému a snadné doplňování monitorovacích metod do tříd.

3. Může v něm být mnoho chyb, není jasné, zda jsme schopni ho naprogramovat.



Obr. 13.2: Pravděpodobnost, že vyvíjený software neuspěje.

Testy funkcí a testy systému jsou základem předávacích testů, což je soubor testů systému, vypracovaných obvykle ve spolupráci s uživatelem, nutných pro převzetí systému uživatelem. Předávací testy nemusí být provedeny v místě instalace systému, někdy mohou být provedeny i na jiné konfiguraci (např. s jinou sestavou terminálových pracovišť), než bude systém instalován. V tom případě je ještě nutné provést testy instalační. Ty jsou obvyklé, instaluje-li se obecně použitelný systém, např. soubor programů.

Testy částí (unit tests), funkcí a testy integrační jsou prováděny realizátorem softwaru, testy předávací a testy při instalaci se provádějí za účasti uživatele. Často se až při nich zjistí, že bylo napsáno něco jiného, než to, co by bylo pro uživatele optimální. Nemůžeme říci „než to, co uživatel chtěl“, poněvadž není neobvyklé, že si uživatel ani řádně neuvědomuje, co by chtěl, pokud nevidí fungující systém. Toto nebezpečí lze zmenšit použitím prototypů a proveditelných specifikací (kap. 1, kap. 7).

Rozsah prací na testování softwaru tedy silně závisí na pořadí, v jakém jsou části zapojovány do systému. Na pořadí, v němž jsou části integrovány, silně závisí rozsah dat, která musíme vytvořit pro zajištění testů, a také rozsah prací na pomocných programech nutných pro provedení testů. U různých systémů bývá optimální po řadí testování částí a integrace různé. Velké systémy je nutno vždy integrovat postupně počínaje nástroji. Při velmi ostrých termínech realizace musíme často postupovat metodou blízkou metodě velkého skoku – proto bývá zkracování termínů realizace softwaru drahé (srv. kap. 15 a 16).

13.2.4 Testové metriky

Je výhodné vytvořit informační systém výsledků testů (viz kap. 15). Při vyhodnocování takto získaných dat je vhodné sledovat trendy v počtech zjištěných selhání systému a také v dobách potřebných pro odstranění příčin selhání (srv.. D. M. Marks, Testing Very Big Systems, McGraw Hill, New York, 1992). Software pro hromadné použití je testován u výrobce a rozesílán vybraným zákazníkům k tzv. beta testování. Testování u výrobce

se nazývá alfa testování. Výsledky beta testování je třeba vyhodnocovat statistickými metodami. O průběhu ladění se u velkých systémů doporučuje vést záznamy historie ladění, umožňující při vývoji i větších změnách vyhodnotit:

1. Počet modulů modifikovaných při vývoji/změně.
2. Počet chyb odstraněných v dané etapě.
3. Průměr chyb na modul.
4. Počet změněných příkazů.
5. Průměrnou dobu na lokalizaci a odstranění chyby.
6. Druhy a frekvence selhání systému způsobené příčinami podle následujícího členění:
   - chyba specifikací
   - chyba návrhu,
   - kódovací chyby,
   - selhání hardwaru,
   - chyba v reakci softwaru na selhání hardwaru.
7. Výčet modulů s největším (nejmenším) počtem defektů.
8. Výčet modulů, které jsou nejsložitější, tj. těch, pro něž nějaká metrika nabývá extrémních hodnot nebo překračuje nějakou hodnotu.

Informace o testech (nejdůležitější jsou data v bodech 1., 6., 7.) je vhodné vytvářet automaticky z hlášení o testování či výsledcích analýz. Body 1. a 7. je vhodné provádět i pro menší systémy. IS výsledků testů je vhodné integrovat s IS metrik. Údaje o testech lze při vhodné organizaci práce a vhodných nástrojích, jako jsou správa knihoven a IS o chybách, zjišťovat automaticky nebo s malou pracností. Pak nenarůstá nadměrně administrativa, kterou by členové týmu nelibě nesli a která by je odváděla od produktivní práce. Jen tak je možné zajistit dostatečnou spolehlivost údajů. Automatická generace metrik omezí vliv chybných nebo zastaralých údajů.

13.2.5 Kritéria ukončení testování

Je otázka, jak dlouho a jak důkladně je třeba testovat. Ani nejdůkladnější testování neodhalí u větších systémů všechny závady. Některé defekty vždy v produktu zůstanou a budou odstraňovány během údržby (srv. kap. 15.5). Při rozhodování, kdy je třeba systém předat, je nutné zvažovat dvě rizika:

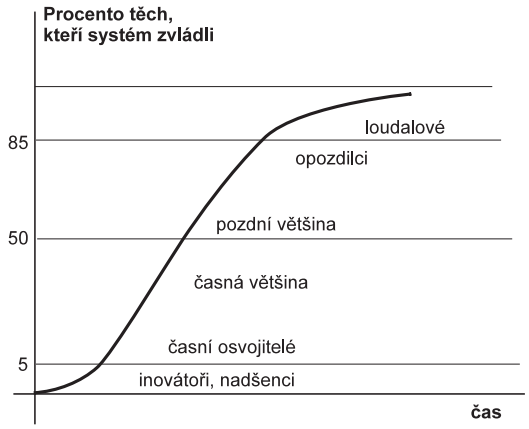
1. Příliš časné předání neodladěného systému zvyšuje pravděpodobnost, že systém neuspěje, tj. že při vývoji na zakázku zákazník buď systém nepřevezme, nebo nebude spokojen s jeho provozem, nebo že při vývoji pro hromadný prodej nebudou spokojeni zákazníci.
2. Prodlužování termínů zvyšuje nespokojenost zákazníka, neboť ztrácí přínosy systému do doby, než je systém použitelný, a vznikají náklady vyvolané potřebou, aby při dlouhém vývoji stále spolupracovali pracovníci uživatele. U produktů dodávaných opakovaně na trh roste nebezpečí, že dříve uspěje konkurence. Dlouho vyvíjený produkt přináší softwarové firmě náklady a nic neprodukuje.

Celková pravděpodobnost neúspěchu je výslednicí působení obou druhů rizik. Situaci ilustruje obr. 13.2. Všimněme si, že pravděpodobnosti obou rizik nejsou známy a že i při nejlepším odhadu zbývá jistá pravděpodobnost neúspěchu. Stanovení optimálního termínu je proto věcí zkušenosti a intuice. Kritérium ukončení testování je možné založit na sledování trendů metriky *PočetSelháníZaTýden* (srv. kap. 15.6). Je možné vyhodnocovat tuto metriku graficky a sledovat, kdy klesne pod zadanou mez (viz obr. 15.18). Žádoucí je při tom využívat metody matematické statistiky.

13.3 Předání do provozu

Po integraci, předávacích testech a případně testech instalačních přichází vyvrcholení práce – uvedení systému do provozu. To je poměrně kritická etapa; není-li totiž instalace dobře připravena, může snadno dojít k neúspěchu celého projektu. Je totiž třeba naučit pracovníky uživatele se systémem pracovat, což se neobejde bez změn pracovních návyků a někdy i bolestných změn mocenských vztahů. Nejednoduchá může být i údržba systému a jeho technické zabezpečení. To všechno jsou problémy známé i z jiných oblastí techniky. U softwarových produktů, které jsou nehmotné povahy, se často i běžné zásady technické praxe opomíjejí. Příprava uvedení do provozu zahrnuje obvykle tyto činnosti:

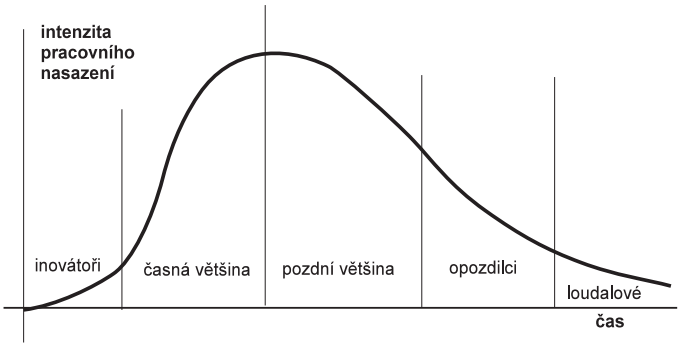
1. Školení pracovníků uživatele, tj. koncových uživatelů systému, a těch, kteří budou mít na starosti údržbu a provoz systému – „systémářů“. Je vhodné, aby rozhodující školení prováděli tvůrci systému, u šíře používaných systémů by to měli být dobře připravení lektoři.
2. Plánování přechodu na nový systém. Přechod na nový systém zahrnuje celou řadu etap, které je třeba pečlivě připravit. Některé aspekty přechodu musí být zajištěny softwarem a měly by být řešeny již v etapě specifikace požadavků. To se týká především konverze dat, vazeb na dosud provozované systémy plánů oživování a přechodu na nový systém. Bývá výhodné provozovat starý a nový systém souběžně, pak starý funguje jako prototyp a generátor dat pro oživení nového systému. Není-li možné provozovat starý systém souběžně s novým[4], je třeba uvážit, jak budou uživatelé zaškolováni – zda se použije prototyp či částečně oživený systém.
3. Navrhnout kritéria pro přijetí systému. Pro hladký průběh převzetí systému bývá vhodné stanovit kritéria převzetí, která by měla být v principu kontrolovatelná nezávislou skupinou pozorovatelů. Tato kritéria by měla být vypracována ve spolupráci řešitelů systému, jeho uživatelů, provozovatelů („systémových pracovníků“) a managementu.



Obr. 13.3: Průběh osvojování nového systému – křivka zvládání.

K převzetí velkého systému do provozu a údržby jsou obvykle požadovány tyto dokumenty:

4. Starý systém neexistuje, nebo je příliš odlišný od nového, nebo je souběžný provoz obou systémů příliš nákladný.



Obr. 13.4: Intenzita práce při učení.

1. cíle systému;
2. specifikace požadavků;
3. dokumenty týkající se návrhu;
4. zdrojové texty programů, předpokládá-li se údržba vlastními silami;
5. dokumenty o průběhu realizace a dokumenty o testech;
6. kopie deníku projektu – system journal (viz kap. 17 o softwarové dokumentaci);
7. záznamy testů a testové soubory, historie ladění;[5]
8. uživatelské a provozní manuály. Důležité jsou různé zaškolovací návody;
9. dohody o době zkušebního provozu a zárukách a způsobech odstraňování chyb;
10. dokumenty obsahující údaje o zárukách a záruční době. Pokud je předpokládáno, že údržbu bude provádět dodavatel softwaru, nemusí být některé dokumenty předávány, musí však být vypracovány pro potřeby údržby u dodavatele.

Pro testování možností údržby se někdy provádějí na hotovém systému:

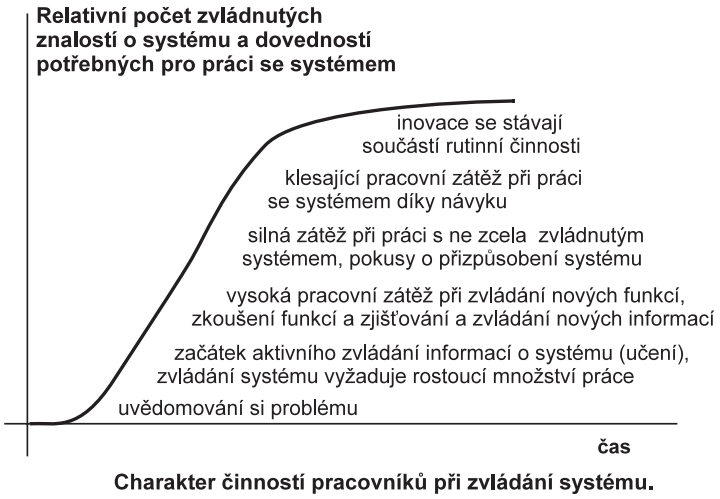
a) zkušební změny,
b) měření rychlosti nalezení záměrně zanesených chyb (seed errors).

Snadnost změn a rychlost nalezení chyb může být též ověřena během zkušebního provozu. Ve většině případů dává tento způsob lepší výsledky než zaseté chyby. Úspěch předání do provozu závisí na řadě faktorů souvisejících s poznatky psychologie a pedagogiky, které je proto třeba využívat v procesu zvládání nových znalostí a dovedností. Z obr. 13.3 vyplývá, že je výhodné se zaměřit na poměrně úzkou skupinu pracovníků, kteří systém zvládnou rychle a budou ho propagovat a také používat. Je vhodné se zaměřit na časné uživatele. U nadšenců hrozí nebezpečí, že se, pokud IS skutečně nutně nepotřebují pro svoji každodenní práci, nadchnou brzy pro něco jiného.

Křivka učení (obr. 13.5) nemá být příliš strmá a IS by měl poskytovat dobré služby i v situaci, že se většina funkcí zatím nepoužívá. Zvládnutí IS by nemělo nikdy znamenat nadměrnou intenzitu práce. Intenzitu učení můžeme vyjádřit ve tvaru z obr. 13.4. Intenzita učení při zvládání systému (tréninku) nesmí být nikdy příliš vysoká a samozřejmě doba učení nesmí být příliš dlouhá. Pro úspěch projektu je rovněž důležité, aby zpočátku bylo třeba poměrně malé úsilí k tomu, aby se daly používat základní funkce. To je často důležitější než celková pracnost

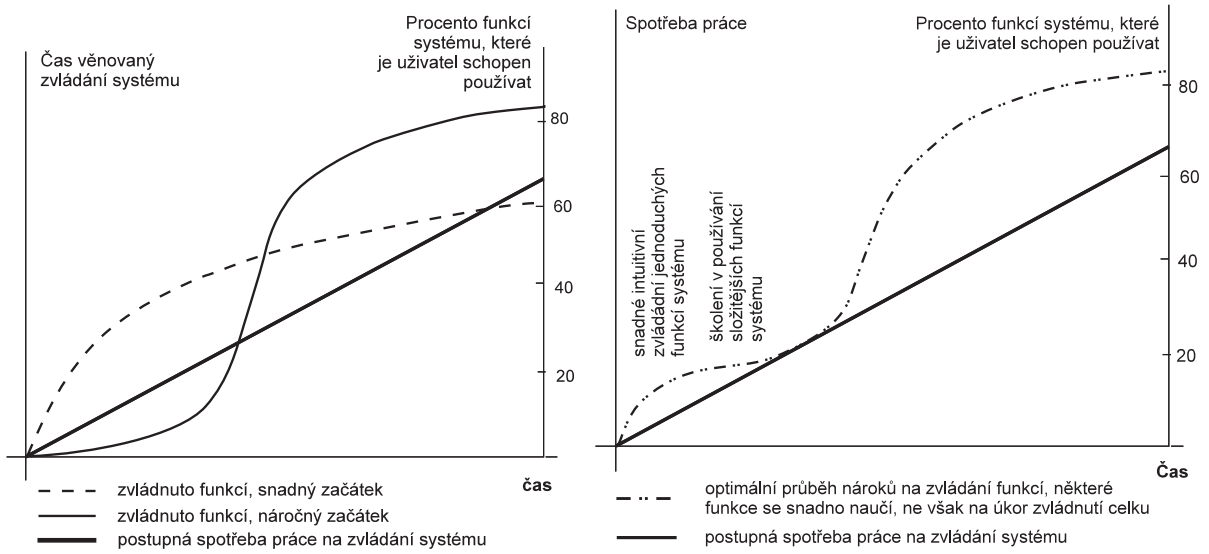
5. Výhodné je využívat IS projektu.

Obr. 13.5: Křivka učení.



Obr. 13.6: Činnosti při zaučování.

zvládnutí celku. O pravdivosti tohoto zjištění se můžeme přesvědčit z produktů Microsoftu. Možné případy jsou uvedeny na obr. 13.7, kde je pro jednoduchost intenzita práce, tj. množství práce za jednotku času, považována za konstantní. Zákazníci obvykle preferují rychlé výsledky. Po zvládnutí základních funkcí nemají už chuť se učit něco nového. Spadli do informatické pasti. Řešení bývá v kompromisu vyjádřeném na obr. 13.7 vpravo.



Obr. 13.7: Varianty zvládání systému (vlevo) a kompromis snižující náročnost učení bez podstatného snížení úrovně zvládnutí celého systému (vpravo).

Jednoduché funkce se naučí rychle. Potom, co již mají pracovníci přehled o základních funkcích, je vhodné provést školení o komplikovanějších funkcích systému.

13.4 Měnící se úkoly IS a jejich důsledky pro volbu technik vývoje

Zavedení informačních technologií silně zrychluje toky informací a zvětšuje rychlost změn. Vývoj IS v posledních deseti letech je charakterizován následujícími trendy:

a) Přechod od individuální práce k práci ve skupině a k práci s lokálními i rozlehlými sítěmi a k práci v rámci IS. Příkladem může být přechod od individuální práce sekretářky, která většinou používala samostatně pracující textový procesor, k práci ve skupině integrací subsystémů podpory práce sekretářky do celopodnikového IS s odpovídajícím rozšířením funkcí.
b) Stále rychlejší obměna používaného hardwaru: sálové počítače – lokální sítě s PC – globální sítě, podpora multimédií atd.
c) Rychlé změny softwarových nástrojů: široké uplatnění moderních operačních systémů, pokrok v databázových systémech, vývojová prostředí atd.
d) Posuny ve volbě cílů IS. Zvyšuje se důraz na informační pokrytí ucelených procesů, podporu managementu a na strategické aspekty fungování IS – na podporu kvalitativních změn fungování podniku či organizace.
e) Změny organizace práce a pracovní náplně. IS umožňuje měnit „pravidla hry v organizaci“.

Rychlost změn ve všech výše uvedených oblastech se zvyšuje. Po revoluci, jejímž nositelem byly osobní počítače a lokální sítě a jejímž nejmarkantnějším projevem bylo prosazení architektury klient-server, přichází další,

ještě podstatnější zvrat – globální sítě na Internetu. Dosah této revoluce lze jen odhadovat, rozsah změn lze však jen stěží podcenit.

Rostoucí výkonnost sítí a serverů umožňuje modernizaci metody spolupracujících aplikací a skládání komponent, (viz např. GaGöté, 1996). Použité metody a celková architektura IS tedy musí počítat s neustálou změnou požadavků, hardwarového a softwarového prostředí a se stále novými požadavky integrace aplikací a služeb celosvětových sítí. To je reálné pouze při uplatnění moderních softwarových architektur, pro které neznamená požadavek stálé změny a rozvoje žádnou větší komplikaci. Techniky, které to umožňují jsou objektová orientace, spolupráce aplikací, CASE systémy a využití nových technik práce s celosvětovými sítěmi, např. služeb WWW a obecně Internetu.

Otevřenost a komplexnost moderních IS si vynucují i změny v přípravě uživatelů. Konkrétní znalosti ovládání např. textových procesorů se stávají samozřejmostí. Stále důležitější je všeobecné povědomí o fungování sítí a možnostech IS v otevřeném prostředí. To platí především pro pracovníky managementu uživatelů IS. Jinými slovy je stále potřebnější všeobecná informatická vzdělanost. Z tohoto důvodu by měli pracovníci uživatelů IS podstoupit pravidelné rekvalifikační kurzy, které jsou zaměřeny na všeobecnou znalost informačních technologií.

13.5 Údržba

Údržba během života systému zahrnuje tyto hlavní činnosti (řazeno v pořadí provádění):

1. Převzetí.
2. Etapa investic:
   - Odstraňování chyb systému (corrective maintenance).
   - Zahrnutí změn hardwaru, standardů a operačního systému (adaptive maintenance).
   - Změny a vylepšení (perfective maintenance).
3. Etapa maximální užitečnosti:
   - Další vylepšení z podnětu uživatelů.
   - Kontrolní testování.
   - Vylepšování kódu a dokumentace.
4. Etapa zmenšování užitku:
   - Další vylepšování výkonnosti.
   - Vylepšení pro další uživatele.

Údržba systému je velmi pracná. U déle existujících středisek a softwarových domů podíl prací s údržbou postupně vzrůstá. S novějšími systémy bývá méně starostí. Jsou psány moderněji a obsahují méně změn – a nepořádků – vyvolaných dlouhodobou údržbou. Je více těch pracovníků, kteří si strukturu softwaru pamatují z dob, kdy software psali. U starších softwarových systémů je více důvodů k úpravám za účelem adaptace a zlepšení funkcí; tyto systémy tedy vyžadují více údržby. Požadavky na údržbu softwaru tedy po určité době vzrůstají. Starší výpočetní centra mají více starších programů, a proto i větší podíl prací na údržbě. Určité podniky nerady mění zavedené systémy z důvodů, které souvisí s problémy udržení systému v chodu a pracností a s riziky přechodu na nový systém. To je typické v bankovnictví. Tam je proto podíl prací na údržbě značný.

Ustavení specializovaných týmů na údržbu a oddělení těchto prací od vývoje a vývojářů přináší redukci nákladů na údržbu. Je při tom vhodné, aby se prací na údržbě zpočátku dočasně zúčastnili, je-li to ovšem možné, i autoři programů. Není ale vhodné, aby byla účast vývojářů trvalá. Údržba vyžaduje specifické znalosti a dovednosti.

Všeobecně platí, že faktory a postupy pozitivně ovlivňující vývoj softwaru ulehčují i údržbu. Jsou to především:

1. Moderní techniky návrhu a realizace.
2. Použití softwarových prostředků pro formátování textu programů a dokumentací.
3. Úplnost a kvalita dokumentace.
4. Elektronická podpora dokumentace.

Pro údržbu programů jsou důležité především správně komentované zdrojové programy s křížovými odkazy. Kromě toho musí být k dispozici vhodný popis architektury a filozofie řešení celého systému. Osvědčují se deníky projektu (kap. 17), protože jsou nejlépe schopny zachytit důvody některých rozhodnutí.

Poněvadž máme při údržbě k dispozici fungující systém, může být popis systému intuitivnější. Nemusí se snažit o podrobný a zcela přesný popis, neboť mnohé lze ověřit pokusem na fungujícím systému. Většinou stačí intuitivní popis systému ze specifikace požadavků. Cenné je zobrazení celkové struktury systému grafickými prostředky a popis rozhraní (interface), tj. metod a technik spolupráce subsystémů. Cenné jsou rovněž prostředky pro testování (knihovny testů a testových souborů) a také prostředky pro sledování historie ladění. Při úpravách programů za provozu je důležité vyhodnocování statistik o průběhu úprav zachycených v historii ladění nebo v IS projektu. To umožňuje včas rozpoznat postupnou ztrátu kvality systému nebo části, kterou je třeba znovu naprogramovat atd.

Problém údržby musí být vzat v úvahu nejen při vývoji softwaru, ale také při vlastní údržbě. Je nutné dbát na to, aby se úpravami nezhoršovala kvalita dokumentace. Při údržbě tedy musíme postupovat podobně jako při vývoji s tím, že v jednotlivých etapách (stanovení cílů, změn požadavků, návrh systému, kódování, testování, předání) upravujeme již existující dokumenty.

Údržbu usnadňují následující vlastnosti programů a dokumentů:

1. *Srozumitelnost.* Srozumitelnost zvyšují následující opatření:
   a) Každá část[6] by měla obsahovat komentář, obsahující
      – popis funkce; mělo by stačit několik vět,
      – popis proměnných/atributů nelokálních v dané části, které jsou v dané části používány nebo měněny,
      – seznam částí volajících procedury/metody dané části,
      – seznam částí, jejichž procedury/metody volají procedury/metody dané části,
   b) Důležité části by měly obsahovat komentáře s informacemi
      – o vstupech a výstupech,
      – o omezeních na provádění,
      – o předpokládaných datech a technickém vybavení,
      – o reakcích na chyby,
      – historie změn prostřednictvím odkazů na příslušné dokumenty,
      – datum vzniku a datum poslední opravy.
   c) Jsou dodrženy normy pro identifikátory, jako je jednoznačnost, mnemotechničnost, snadná přiřaditelnost k částem. Každá proměnná je používána pouze pro hodnoty jednoho logického typu.
   d) Obecná zásada (test 90 – 20). Programátor by měl být schopen po 20 minutách čtení části rekonstruovat 90 % části zpaměti (viz Shneiderman, 1980, str. 92–122), jinými slovy, 90 % modulu (třídy) zvládne za 20 minut.
2. *Modifikovatelnost.*

6. Modul, třída, metoda, tabulka v databázi atd.

a) Program musí být srozumitelný a psaný pokud možno ve vyšším programovacím jazyce.
b) Všechny systémové konstanty, jako je velikost tabulek, čísla kanálů atd., jsou zadány symbolicky a definovány na jednom místě modulu.
c) V paměti je místo pro rozšíření programu při úpravách.
d) Program neobsahuje žádné části kódu dvakrát. Takový kód se musí přesunout do společného podprogramu.
e) Algoritmy, které jsou v knihovnách, nejsou programovány znovu.
f) Software je obecný – lze ho provádět na různých konfiguracích hardwaru a základního softwaru a pro různé formáty vstupů a výstupů a dat.
g) programy jsou flexibilní, specializované funkce jsou koncentrovány do jednoho úseku programu, rozhraní je poměrně necitlivé ke změnám, rozhraní je programováno pomocí aparátu importovaných procedur/objektů.[7]

3. *Přenositelnost.*
   a) Program je psán pokud možno ve vyšším programovacím jazyce odpovídajícím normě, výhodné je použít objektově orientované techniky.
   b) Programy neobsahují vazby na konkrétní operační systém nebo jsou tyto vazby „skryty“ do malého počtu procedur, nebo tříd. Totéž platí o obratech, které závisí na hardwaru konkrétního počítače. Speciálně by neměly funkce programu záviset na velikosti slova počítače.
   c) Nestandardní programovací obraty, resp. vazby na hardware a operační systém počítače jsou zvláště pečlivě dokumentovány a vyznačeny v programech.
   d) Dokumentace i samotné programy usnadňují detekci míst, která je třeba změnit při přenosu. Dokumentace by měla usnadňovat případné úpravy.

Pracnost údržby závisí především na následujících skutečnostech:

1. Kvalita specifikace požadavků a stabilita požadavků. Změny požadavků jsou hlavní příčinou změn softwaru během údržby. Rozsah požadavků na změnu silně závisí na počtu uživatelů systému.
2. Doba, po kterou je systém provozován. Je-li program provozován dlouho, lze očekávat změny v důsledku změn požadavků, změn technického vybavení a operačních systémů. Některé informační systémy žijí dlouho, i více než 20 let.
3. Závislost na vnějších podmínkách. Algoritmus výpočtu daní závisí na zákonu – a ten může být změněn. Závislost na vnějších podmínkách je typická pro ekonomické aplikace.
4. Závislost na změnách hardwaru a základního softwaru. Rychlý vývoj počítačů a informačních technologií vyvolává potřebu častých změn softwaru na hardwaru a základním softwaru v různé míře závislých.
5. Charakteristiky použitých technik programování, jako jsou
   - modifikovatelnost (změny mají tendenci být snadné a lokální),
   - použitý programovací jazyk nebo vývojový systém,
   - styl jakým je systém navržen a napsán, a architektura systému,
   - kvalita testů inspekcí,
   - kvalita dokumentace,
   - použití moderních technik (CASE, objektová technologie, spolupráce aplikací, vývojová prostředí).
6. Kvalita provádění údržby. Je třeba, aby údržba nezhoršovala kvalitu programů a dokumentace.

7. Tato pravidla jsou známa již od šedesátých let. Přesto se neustále porušují. Nejmarkantnějším příkladem je problém roku 2000, který by při dodržování výše uvedených zásad nikdy nevznikl.

14

Vývoj uživatelského rozhraní

Návrh a vývoj uživatelského rozhraní je specifickou částí vývoje systému s vlastními problémy a metodami jejich řešení. Je proto vhodné koncipovat uživatelské rozhraní IS jako relativně samostatný subsystém. Metody návrhu vrstvy uživatelského rozhraní (UI – user interface) a především metody testování UI se liší od metod a postupu návrhu a realizace výkonné logiky a správy dat. Přehled metod návrhu UI je uveden v (Nielsen, 1993). UI do značné míry ovlivňuje spokojenost zákazníka se systémem. UI je důležité i z čistě obchodního hlediska. Je čímsi jako obalem softwaru – a kvalita obalu zvyšuje atraktivitu zboží. UI zároveň podstatně ovlivňuje složitost ovládání IS. Nekvalitní UI diskvalifikuje celý IS. UI dnes obvykle využívá grafiku a myš. Informační systémy dnes využívají grafické uživatelské rozhraní (graphical user interface, GUI). Kvalita UI silně ovlivňuje náklady na provoz IS. Ovlivňuje totiž počet chyb, jichž se dopouští uživatelé při práci se systémem. Doba zaškolování a také doba potřebná k provedení jednotlivých dialogů s IS také závisí na kvalitě UI. UI tedy může významně ovlivnit celkovou pracnost užívání IS. Nespokojenost uživatelů s IS bývá často způsobena nekvalitním UI.

Analýza 31 projektů v roce 1993 (Nielsen, 1993) ukázala, že vývoj UI spotřeboval od 4 % do 15 % celkových nákladů na projekt, s průměrem okolo 6 %. Tento podíl se považoval za příliš nízký, ideální by měl být okolo 10 %. UI významně ovlivňuje ergonomii práce s počítačem. Při hodnocení významu různých vlastností IS bývá na předním místě snadnost zaškolení, snadnost používání a kvalita dokumentace. Vzrůstá význam ergonomie. Podle průzkumů je váha požadavků na kvalitu UI mezi 20 % až 30 % váhy všech požadavků na IS. Význam UI vzrůstá, rozvíjí se prostředky návrhu GUI ve vizuálních prostředcích programování. Vzrůstá potřeba analýzy UI a testování UI.

Pravidla Evropské unie pro software explicitně stanovují, že software a tedy i UI

- musí být vhodný pro daný účel a plnit zvolené funkce,
- musí být snadno použitelný,
- musí aplikovat zásady softwarové ergonomie.

14.1 Hlavní zásady návrhu uživatelského rozhraní

Uživatelské rozhraní (UI) je výhodné vyvíjet jako relativně nezávislý subsystém (srv. obr. 14.1) v obdobném životním cyklu jako celý systém. Hlavní etapy vývoje UI jsou:

- Analýza požadavků, specifikace požadavků.

- Návrh UI.
- Realizace prototypů a jejich testování s uživateli.
- Integrace UI se zbytkem systému a testování UI společně s uživateli.
- Sledování vlastností UI za provozu a vylepšování vlastností UI.

Vývoj UI je silně ovlivňován následujícími skutečnostmi:

1. Při vývoji UI je ještě obtížnější než při návrhu funkcí odhadnout optimální vlastnosti UI, protože ty závisí na psychologii, dovednostech a znalostech budoucích uživatelů.
2. Testování UI je možné pouze tak, že systém testují uživatelé. Testování se provádí sledováním práce uživatelů.
3. Vlastnosti uživatelů se během používání systému vlivem zkušeností mění.
4. UI musí vždy počítat se začátečníky i s pokročilými.
5. Důležitou roli hrají problémy ergonomie (kapitola 4).

UI musí být navrženo v úzké spolupráci s uživateli. Návrhář UI většinou nedokáže dostatečně přesně odhadnout psychologii uživatelů, co jim bude vyhovovat a co nikoliv. Na druhé straně pro UI platí, že je nedokáže dobře navrhnout uživatel sám. Řešení tohoto dilematu je v prototypovém řešení UI vycházejícím z vlastností UI úspěšných systémů.

Prototypové řešení by mělo být ověřováno těmi, kdo je budou skutečně používat. Tak např. skladník ověřuje práci s UI potřebným pro provoz skladu a ředitel rozhraní manažerské části IS. Případy, kdy o UI rozhodují výhradně šéfové, např. náměstci, nebo pouze informatici odpovědní za IS, nekončívají dobře. Na testování prototypového řešení navazuje testování UI na postupně oživovaném systému.

Výsledky testů se používají jako podklad pro další zlepšování UI. Zlepšené verze je nutno opět testovat. Celý proces se několikrát opakuje (iterovaný vývoj, kap. 8), dokud nejsou vlastnosti UI vyhovující. Iterovaný vývoj je snáze realizovatelný, je-li UI realizován relativně samostatným modulem, který je koncepčně navrhován jako téměř samostatná aplikace (obr. 14.1).

Obr. 14.1: Architektura aplikace vhodná pro postupný vývoj uživatelského rozhraní.

Nápověda v UI má být chápána jako výpomoc spíše pro začátečníky a pro velmi zřídka se vyskytující komplikované situace. Dobré UI se na nápovědu nesmí příliš spoléhat. Časté používání nápovědy je příznakem nevhodného návrhu UI. Při analýze, návrhu a testování UI se využívají následující techniky (Nielsen, 1993):

a) *Pozorování uživatelů* a jimi prováděných úkolů a analýza dokumentů, které používají. Tato technika je součástí obecné procedury zjišťování požadavků (viz kap. 6).

b) *Scénáře.* Zaznamenávání ucelených postupů uživatele při práci.
c) *Myšlení nahlas.* Tato technika se používá jako prostředek zjišťování požadavků a především jako prostředek testování prototypového a pak i konečného UI. Při této technice uživatel za přítomnosti autora IS nebo testéra nahlas říká, co právě dělá a proč to dělá. Někdy se pořizuje zvukový záznam nebo video. To však může rušit a následná analýza záznamů je velmi pracná. Obvykle stačí, když si navrhovatel UI dělá poznámky.
d) *Heuristická evaluace.* Při této technice se hodnotí UI neformálně podle sady doporučení. Takových sad je celá řada (viz Nielsen, 1993) a některé obsahují až několik set doporučení. V (Nielsen, 1993) jsou uvedena následující hlavní kritéria používaná při heuristické evaluaci. Kritéria se do značné míry kryjí se zásadami správné specifikace požadavků na IS nebo jsou jejich jednoduchými modifikacemi:
   (a) Jednoduchý a přirozený dialog. Dialog má odpovídat intuitivnímu postupu a nemá obsahovat informace, které nejsou potřeba. Informace mají být zobrazovány v přirozeném a logickém uspořádání a způsobem, na který je uživatel zvyklý.
   (b) Jazyk dialogu je uživateli srozumitelný, užívá výhradně jemu známá slova a slovní spojení.
   (c) Dialog nezatěžuje zbytečně paměť. Je tedy navržen tak, aby si při jeho provádění musel uživatel pamatovat co nejméně. Příkladem porušení tohoto pravidla je editor vi v UNIXu. Pokud je nějaká informace potřeba, měla by být snadno dostupná.
   (d) Konzistentnost – obdobné situace vyžadují obdobnou reakci.
   (e) Zpětná vazba – uživatel by měl být vždy informován o tom, co systém dělá.
   (f) Jasně definované a zřejmé způsoby (exits), jak dialog ukončit nebo jak se vrátit v dialogu k předchozím krokům.
   (g) Horké klávesy: Dialog má umožňovat rychlé zahájení často používaných funkcí, např. stisknutím vhodné kombinace kláves.
   (h) Kvalitní hlášení chyb. Hlášení chyb má být v otevřeném jazyce, nemá obsahovat pouze kódy chyb, musí jasně definovat podstatu problému a případně obsahovat návod jak postupovat dále.
   (i) Prevence chyb. Často se vyskytující chyby uživatele se dají omezit zpřesněním dialogu.
   (j) Nápověda a dokumentace. Nápověda a dokumentace by měla být používána co nejméně. V případě potřeby by měla být dosažitelná snadno a standardním způsobem. Je vhodné mít k dispozici elektronickou formu dokumentace i papírové manuály.

   Je třeba, aby heuristické evaluace provádělo více hodnotitelů. Nielsenovy studie ukazují, že jediný vyhodnocovatel detekuje při evaluaci asi 1/3 problémů. 75 % problémů zachytí pět vyhodnocovatelů. Deset vyhodnocovatelů zachytí přibližně 85 % problémů. Optimální počet je pět vyhodnocovatelů.

14.2 Faktory akceptovatelnosti softwaru

Podle (Nielsen, 1993) lze faktory akceptovatelnosti softwaru shrnout do následujícího schématu:

1. Sociální a společenská akceptovatelnost.
2. Praktická akceptovatelnost.
   2.1 Užitečnost.
      2.1.1 Funkčnost.
      2.1.2 Použitelnost (Usability). Použitelnost silně závisí na následujících vlastnostech uživatelského rozhraní:

- snadnost naučení,
- efektivnost při používání,
- dobře se pamatuje, jak systém používat,
- málo chyb uživatele způsobených špatným ovládáním rozhraní,
- subjektivní příjemnost práce se systémem pro uživatele,
- dobré ergonomické vlastnosti.

2.3 Cena.
2.4 Kompatibilita, přenositelnost.
2.5 Modifikovatelnost.
2.6 Spolehlivost.
2.7 Dostupnost pro všechny uživatele.

Úkolem návrhu UI je zlepšit faktory uvedené v 2.2. Při hodnocení efektivnosti UI je třeba odlišovat chování nových uživatelů a uživatelů, kteří se systémem dlouhodobě pracují. Dlouhodobí uživatelé budou mít tendenci používat klávesové zkratky a různá urychlení, které většinou znamenají i větší nároky na paměť. Noví uživatelé dávají přednost takovým nástrojům, které nevyžadují rozsáhlé zaučování předem. Příkladem řešení, které vychází takovým uživatelům vstříc, jsou produkty Microsoftu. Tyto produkty jsou výhodné pro zvládání systému metodou pokusů a omylů. Výhody a nevýhody takového přístupu jsou uvedeny v kapitole 13.

Zkušení uživatelé často preferují znakové rozhraní nejen proto, že s ním lze pracovat rychleji, ale také proto, že nemusí tolik pozorovat obrazovku a nemusí tolik používat myš. Klávesové rozhraní je proto i ergonomicky výhodnější. V mnohých případech je však ovládání myší nenahraditelné. Je tedy žádoucí obě techniky (ovládání myší a ovládání klávesnicí) kombinovat pro dosažení optimálního výsledku.

Faktory použitelnosti lze poměrně dobře měřit. Lze implementovat automatický sběr metrik jako součást UI. Stačí zaznamenávat údaje o akcích jednotlivých uživatelů spolu s časovými údaji. Klasičtější postupy jsou uvedeny v (Nielsen, 1993). Subjektivní spokojenost lze zjišťovat pomocí dotazníků a analyzovat trendy v tomto hodnocení (srv. též kap. 15). Vyhodnocování prototypových řešení se může provádět i bez pomoci automatizace, stačí sledovat činnost uživatele a zaznamenávat vznikající problémy.

14.3 Životní cyklus uživatelského rozhraní

Náplň etap vývoje UI se poněkud liší od vývoje aplikace jako celku. Klade větší důraz na ověření kvalifikačních předpokladů a znalostí uživatele. Etapy vývoje UI podle (Nielsen, 1993) jsou následující:

ANALÝZA A SPECIFIKACE POŽADAVKŮ NA UI

A) *Poznání uživatele.*

Hlavním problémem návrhu UI je navázání kontaktu s těmi, kdo budou systém skutečně používat, s koncovými uživateli. Při vývoji UI je požadavek spolupráce s koncovými uživateli ještě ultimativnější než při specifikaci požadavků na funkce IS. Kontakt s uživateli musí být širší, především při testování UI. K tomu často management zákazníka nevytváří dostatečný prostor. Dodavatel IS se někdy obává, že při spolupráci vývojářů s uživateli vznikne situace, že koncoví uživatelé budou mít tendenci se obracet přímo na vysoce kvalifikované vývojáře a budou tak obcházet útvary odpovědné za údržbu. Analýza situace u uživatele se soustřeďuje do následujících oblastí:

(a) Kvalifikační předpoklady, znalosti a dovednosti koncových uživatelů: znalost práce s počítači resp. IS, vzdělání, věk, pracovní prostředí atd. Tyto skutečnosti jsou důležité pro odhad složitosti školení a potřebných vlastností rozhraní. Zkušenější obvykle nevyžadují tolik nápovědy v nabídkách a mohou snáze pracovat i se znakovým rozhraním. Pokud pracovník sdílí místnost s více pracovníky, je vhodné se vyhýbat zvukovým efektům při práci se systémem.

(b) Analýza úkolů. Návrh UI závisí na tom, co uživatel skutečně dělá – např. při konverzaci s klientem při zakládání účtu v bance. Z takto zjištěných skutečností se pak formulují cíle UI a slabá místa současné situace. Analýza úkolů je součástí specifikace požadavků. Je tedy žádoucí se při specifikaci požadavků zaměřit i na problémy interakce uživatele s IS.

(c) Analýza funkcí. Úkoly se skládají z jednotlivých uzavřených činností, jako je např. vyhledávání určité informace. Dialog se navrhuje tak, aby se nejčastější operace prováděly co nejefektivněji.

(d) Vývoj požadavků uživatele. Užívání IS mění vlastnosti uživatele – stává se zkušenějším, takže ho mohou zdržovat věci, které byly výhodné, když se zaučoval. Při používání IS se stává, že si uživatel uvědomí nebo najde možnosti, se kterými se nepočítalo. To je typický vývoj uživatele, který je třeba předvídat a připravit se na něj.

B) *Analýza UI podobných nebo konkurenčních projektů.*
Pokud existují podobné nebo konkurenční produkty, je výhodné prostudovat jejich UI, zjistit jejich přednosti a nedostatky. Tyto znalosti pak lze využít při návrhu uživatelského rozhraní.

C) *Stanovení kontrolovatelných cílů.*
Některé vlastnosti UI je možné předem kvantifikovat. Typickým příkladem je počet chyb uživatele za jednotku času. Jako cíl je možné stanovit nějakou hodnotu této metriky. Hůře se formulují požadavky na snadnost osvojení a rychlost ovládání.

D) *Stanovení finančního vlivu UI.*
Doba, po kterou musí pracovníci pracovat s IS, bývá značná. Uvažujeme systém se sto pracovními místy, u každého místa se pracuje po $1/3$ pracovní doby. Cena této pracovní doby je $1/3 \cdot$ *NákladynaHodinu* $\cdot$ *PočetHodinvMěsíci*. Při počtu hodin 200 do měsíce bude u terminálu spotřebováno $200 \cdot 100 \cdot 1/3 = 6666$ hodin. Při nákladech na hodinu pracovníka včetně režie 800 Kč UI přímo ovlivňuje pracovní kapacitu v ceně více než 5 milionů Kč měsíčně. Úspora 5 % času je ekvivalentní úspoře více než 250 000 Kč měsíčně.

NÁVRH UI

Při návrhu UI se používá řada technik, které jsou do značné míry nezávislé a mohou se vzájemné doplňovat.

A) *Paralelní návrh.*
Při návrhu UI se osvědčuje nezávisle navrhnout více variant a pak vytvořit syntézu použitím toho nejlepšího ze všech návrhů.

B) *Spoluúčast zákazníka.*
Spoluúčast je nutná při návrhu prvních verzí UI, nejlépe formou společného vývoje s tím, že první návrh formuluje vývojář.

C) *Koordinace návrhu.*
Cílem je sjednotit formu UI pro různé funkce systému.

D) *Heuristická evaluace (viz výše).*
Cílem je zjistit a napravit nedostatky uplatněním osvědčených zásad.

IMPLEMENTACE A TESTOVÁNÍ UI

A) *Realizace prototypů.*

B) *Předvedení prototypů UI.*
Uživatelé společně s vývojáři zkušebně používají prototypy dialogů typu Potěmkin.
C) *Postupné vylepšování návrhu a odstraňování opomenutí a nedostatků.*
UI je zkušebně používáno uživateli a postupně se vylepšuje.

TESTOVÁNÍ A PROVOZ UI

A) *Sledování metrik UI.*
Počty chyb a doba provádění se sledují za běhu systému. Je výhodné používat vhodné nástroje. Osvědčují se žurnály (log) dialogů a jejich analýza.
B) *Úpravy UI* podle zjištěných skutečností.

14.4 Zvláštnosti testování uživatelského rozhraní

Testování UI musí být prováděno ve spolupráci s budoucími uživateli. Test provádí člen skupiny uživatelů za přítomnosti vývojáře. Je třeba brát ohled na velké individuální rozdíly mezi jednotlivými uživateli a také mezi nezkušenými a zkušenými pracovníky. GUI bude např. daleko přístupnější těm pracovníkům, kteří mají nějakou zkušenost s nějakou formou Windows.

Ověřování UI je proto třeba provádět s větší skupinou pečlivě vybraných budoucích uživatelů. Bývá výhodné provést nejprve zaškolení v ovládání systémových prostředků. Optimální počet testujících je 3 až 6. Testéři pracují s tou částí UI, se kterou budou skutečně pracovat při provozu systému. Při organizaci testů je žádoucí navodit atmosféru spolupráce: „Pracujete na tom, aby vám to, co budete používat, sloužilo dobře“. Uživatelé nemají pracovat ve stresu. Výsledky testů by měly být anonymní.

Testování lze do jisté míry provádět na částečně funkčních prototypech. Vždy je však nutné nakonec provádět testy i na oživeném systému. Důvodem je potřeba testovat doby odezvy. Je obvyklé, že se na základě analýzy výsledků testů UI podstatně mění. Testovat je nutné i nové verze UI. To je další důvod, proč je třeba UI navrhovat jako relativně samostatný subsystém.

Testování UI probíhá v následujících etapách (Nielsen, 1993):

1. Příprava.
2. Zahájení. Při zahájení testů je vhodné zdůraznit následující skutečnosti:
   - účelem testů je testovat systém, nikoliv uživatele;
   - účelem testů je dosáhnout spokojenosti uživatelů;
   - je žádoucí, aby se všichni svobodně vyjadřovali;
   - poněvadž se jedná o testování, může výsledný systém fungovat poněkud jinak;
   - poprosit, aby o testu testující nehovořili, aby tím ostatní testéry neovlivňovali;
   - zdůraznit, že účast na testu je dobrovolná a lze jej kdykoliv ukončit a že výsledky testu jsou anonymní a důvěrné;
   - uvést, že jsou možné otázky, že je vítáno vyjádření pochybností, a že se má systém v budoucnu používat bez cizí pomoci;
   - požádat o „myšlení nahlas“, tj. poprosit, aby testující nahlas komentoval, co dělá a proč. Poprosit o pokud možno rychlou práci bez chyb.
3. Provedení testů. Testér má pracovat pokud možno samostatně.
   - Je důležité, aby testování nebylo přerušováno telefony, návštěvami atd.

- Je výhodné, když uživatel provede na počátku rychle a úspěšně nějakou část dialogu, je pak méně nervózní.
- Atmosféra při testování by měla být uvolněná.
- Nedávat najevo, že uživatel je pomalý nebo že dělá chyby.
- Vyloučit kibice, včetně šéfů testujícího.
- Zastavit testování, vznikne-li pocit, že testér toho má již dost.

4. Vyhodnocení testů.
   - Požádat uživatele o celkový názor, především o to, jak je spokojen.
   - Výsledky testů zaznamenat v dohodnuté formě.
   - Výsledky zavést do databáze projektu (pokud existuje).
   - Sestavit vlastní hodnocení.

Při vyhodnocování testů UI se používají následující metriky:

- Doba provedení určitého úkolu.
- Počet variant úkolů úspěšně provedených za určitou dobu.
- Poměr úspěšně provedených úkolů k počtu chyb.
- Doba potřebná pro nápravu chyby.
- Počet příkazů použitých během testů.
- Počet příkazů, které nebyly vůbec použity.
- Frekvence použití nápověd a manuálů.
- Počty kritických a pochvalných komentářů uživatele.
- Počet případů, kdy byl uživatel frustrován a kdy potěšen.
- Rozsah „mrtvých dob“, během nichž uživatel nekomunikuje.
- Počet případů, kdy uživatel nevěděl jak dál.

Pro hodnocení UI je třeba využívat data shromážděná při testování celého systému. Je vhodné pro tento účel používat IS metrik. Většinu výše uvedených metrik lze odvodit z databáze záznamů kroků dialogů (log)[1]. Záznamy mohou být tvořeny větami obsahujícími: id uživatele, typ záznamu (zahájení úkolu, konec úkolu, provedení příkazu), pomocné údaje a časové razítko. U hlášení chyb je třeba zaznamenávat i typ chyby.

14.5 Údržba uživatelského rozhraní

Během užívání IS se podstatně mění znalosti a dovednosti uživatelů. Ze začátečníků se stávají zkušení uživatelé. Postupný růst datové základny a často i počtu pracovních míst mění i chování systému. Jinými slovy podmínky, za nichž se UI testovalo, se během provozu podstatně mění. Je proto nutné práci s UI sledovat, např. využíváním dat generovaných automaticky, jak je uvedeno v předchozím paragrafu.

Je žádoucí zjišťovat stanoviska a názory uživatelů a vývoj jejich názorů a požadavků. K tomu je možno použít obdobné techniky jako při specifikaci požadavků na celý systém. Osvědčuje se interview (případně strukturované), dotazníky a společné hodnotící schůzky (někdy stačí i telekonference). Důležité jsou zprávy o problémech od zákazníka. Dotazníky by měly být nejvýše na dvě stránky (i s vysvětlivkami), nejraději na jednu. Jinak hrozí, že dotazníky skončí v koši.

Většina dotazů má být uzavřená (odpověď ano/ne) s možností komentářů, případně aby odpověď byla možná formou hodnotících známek (např. 1 až 10) či výběru z alternativ. Především je však vždy třeba zvážit, jak budeme

1. Záznam průběhu dialogu nazveme žurnálem.

Metoda	Etapa	Počet testérů	Hlavní výhody	Hlavní nevýhody
Heuristické evaluace	Návrh	0	Detekuje jednotlivé problémy použitelnosti, lze využít experty.	Není kontakt s uživateli.
Měření výkonů.	Testování UI, analýza podobných a konkurenčních produktů.	alespoň 10	Přesná data. Snadná kritéria porovnání.	Obvykle se nedaří porovnat všechny aspekty.
Myšlení nahlas.	Návrh, informativní evaluace.	3 až 5.	Levné. Zachycuje chyby v pochopení systému v poměrně reálné situaci.	Nepřirozené pro testujícího. Nevhodné pro zkušené. Ti rychleji jednají než mluví.
Pozorování.	Analýza a evaluace UI, tvorba scénářů.	alespoň 3 pro každý subsystém	Sleduje skutečné činnosti v reálu. Inspiruje návrh funkcí.	Subjektivní, obtížně kontrolovatelné, časově náročné, často se nenajde dost času na provedení.
Dotazníky	Analýza, sledování problémů za provozu.	Většina uživatelů	Anonymní. Lze opakovat, zachytí názory více uživatelů.	Nebezpečí, že důležití uživatelé neodpoví nebo dotazník přesně nepochopí.
Interview.	Analýza úkolů.	Několik	Flexibilní, lze jít do hloubky.	Časově náročné, náročné na kvality moderátora, subjektivní.
Žurnál (log)	Závěr testování, provoz.	Všichni uživatelé	Běží stále. Výhodné pro vyhodnocování efektivnosti a trendů.	Je nutno realizovat jednoduchý IS pro vyhodnocování.
Sledování názorů uživatelů.	Provoz.	Co nejvíce.	Lze sledovat trendy v názorech a potřebách uživatelů.	Obtížně se zajišťuje. „Proč se o to starat, když už to pracuje.“

Tab. 14.1: Přehled metod používaných při vývoji a modifikacích uživatelského rozhraní.

na zjištěná data reagovat, tj. jak ten který výsledek ovlivní modifikace UI. Schůzky hodnotící kvalitu UI jsou výhodné v tom, že se často zjistí neočekávané skutečnosti. Diskuze ve skupině, dotazníky a interview je vhodné připravit s využitím analýzy dat z žurnálu (log) dialogů uživatelů se subsystémem. Převážná většina problémů bývá způsobena několika málo příčinami.

14.6 Výhody a nevýhody grafického uživatelského rozhraní

Grafické uživatelské rozhraní (GUI) je víceméně standardem pro IS. Hlavní výhodou GUI je možnost intuitivní práce. GUI je výhodné pro učení metodou pokusů a omylů a má obecně malé nároky na paměť a většinou i dovednosti uživatele; uživatel si musí při používání GUI pamatovat méně než při znakovém rozhraní. Důležitý je i pocit neustále interakce s počítačem. Grafické rozhraní je nutností pro grafické aplikace. GUI je podmínkou pro vizuální metody programování. Nezanedbatelné je i to, že se GUI považuje za znak modernosti systému.

14.6 Výhody a nevýhody grafického uživatelského rozhraní

Použití GUI není bez problémů. Intuitivnost a menší nároky na zapamatování jsou výhodné hlavně pro začátečníky. Pro pokročilé uživatele může být práce s GUI pracnější než znakové rozhraní, může zdržovat. GUI je náročné na hardware a značně ztěžuje vytváření obdoby skriptů. GUI vyžaduje neustálou interakci s počítačem, neustálé sledování obrazovky a práci s myší. Jen zřídka lze zadávat příkazy „do zásoby“. Je proto ergonomicky značně náročné a často nadměrně pracné.

Při práci s GUI lze obtížněji sledovat historii práce. Bývá proto obtížnější analyzovat případné chyby a automatizovat definice obdobných činností, např. formou maker. Je proto žádoucí používat vedle GUI i znakové rozhraní včetně „horkých“ kláves (klávesových zkratek).

15

Měření softwaru, softwarové metriky

Při řízení prací při vývoji či customizaci a také při provozu IS je nutné sledovat kvantitativní (číselné) charakteristiky, jako je počet řádků programů, doba řešení, pracnost řešení atd., umožňující hodnotit průběh prací a odhadovat kvalitu softwaru a přijímat odpovídající opatření. Příkladem je sledování řešení těch částí, ve kterých je velmi mnoho závad. Pro kvantitativní charakteristiky softwaru budeme používat termín softwarové metriky. Sledování metrik, jako jsou trendy počtů selhání systému, nebo kvantifikované hodnocení systémů uživateli, je základem zajišťování kvality softwaru a bude brzy v souvislosti s používáním norem ISO 9000–3, ISO 9126 aj. nezbytností.

Při uzavírání smluv je nutné provádět odhady nákladů a doby řešení. Odhad je možné učinit pouze na základě zkušeností. Je však žádoucí znát, jaké byly náklady realizace a doba řešení obdobných projektů, čili je třeba mít k dispozici kvantitativní charakteristiky (náklady, doba řešení) softwaru dříve realizovaných projektů. Hodnoty metrik jsou tedy cosi jako paměť firmy.

Z dlouhodobého hlediska je možné využívat softwarové metriky k tomu, abychom mohli hodnotit přínosy různých metod vývoje softwaru a odvodit empirické zákonitosti, které mohou být použity

a) jako základ pro stanovení technicko-ekonomických podkladů pro řízení prací při tvorbě softwaru (normy pracnosti, odhady takových metrik, jako je pracnost či doba řešení) a uzavírání smluv (cena, termíny),
b) jako podklad pro hledání takových metod realizace softwarových produktů, které by přinesly podstatné snížení nákladů a doby vývoje a hlavně rozsahu prací při údržbě softwaru.
c) jako prostředek sledování spolehlivosti softwaru při provozu a podklad pro řídící zásahy během údržby,
d) jako prostředek sledování průběhu prací při vývoji (dodržování termínů, procento testovaných komponent, trendy počtů chyb, počty nově zanesených chyb, komponenty s největším počtem chyb, atd.).

Metriky mohou být z hlediska teorie měření různého typu (srv. Zuse, 1990, nebo Vaníček, 1995, nebo normu ISO 9126). Pro některé metriky, jako je délka programu, má smysl metriky sečítat. U jiných metrik, jako je např. míra spokojenosti s produktem, má smysl pouze uspořádání. U dalších metrik, jako je např. zařazení do určitých tříd, není definováno ani uspořádání.

15.1 Měření a řízení

Metriky jsou důležitou součástí podpory rozhodování moderního managementu. Různé aspekty a nebezpečí použití metrik při řízení jsou diskutovány ve sborníku (Harris, 1994). Zásady zde uvedené platí do značné míry pro každé

metriky a měly by být vzaty v úvahu při návrhu IS obecně i při vývoji softwaru. Metriky jsou použitelné jen tehdy, jsou-li k dispozici efektivní nástroje sběru dat a generace využitelných informací. Podle normy IEEE 1061–1992 (kap. 20) je nutné u softwarových metrik ověřovat relevanci, dostupnost, využitelnost a přínosy ve vztahu k nákladům. Podle (Sink, Smith, 1994) ve sborníku (Harris, 1994, str. 147–160), je nutno sběr a vyhodnocování metrik chápat jako základní součást strategických manažerských procesů softwarové firmy, jako základní nástroj jejího managementu.

Metriky a jejich vyhodnocování jsou součástí každého efektivního systému řízení a tedy i IS na jeho podporu. Systém měření je budován na základě požadavků uživatelů na informace. Na základě požadavků se identifikují data potřebná pro vyhodnocování požadovaných informací a pak se navrhne a realizuje systém sběru dat a vyhodnocování informací – informační systém. Při tom se berou v úvahu následující skutečnosti a zásady:

- Hlavním cílem je systém měření umožňující snadný přístup k metrikám a efektivní metody vyhledávání a vyhodnocování informací.
- Cílem měření není každodenní operativní dohled a kontrola. Systém orientovaný na kontrolu a dohled má tendenci ustrnout a nevyvíjet se. Navíc hrozí, že se nevyužijí možnosti, které systém nabízí pro vrcholové řízení.
- Systém měření nemá vyvolávat odpor. Odpor může být způsoben nevhodnými opatřeními managementu. Zviditelnění dobrých výsledků může vést management k rozhodnutí nadměrně redukovat zdroje pro daný úkol. Zviditelnění nevýkonnosti může vést k posílení zdrojů, později však i k postihům. Je důležité, aby většina cítila, že jsou metriky užitečné a poctivé zvýhodňují.
- Využívání informací může být ztíženo krátkozrakostí a profesionálními předsudky (nekritické přeceňování účetnictví a finanční operativy atd.).
- Informace a rozhodnutí mohou složitým způsobem záviset na několika metrikách. Snaha o zjednodušení může vést k nesprávným závěrům.
- Efektivnost využití systému sběru a vyhodnocování metrik závisí na tom, do jaké míry bude všemi akceptován. To závisí především na míře spoluúčasti uživatelů na koncipování a vývoji systému metrik. Spoluúčast zvyšuje i vyhlídky na další rozvoj a vylepšování systému.
- Systém by měl být koncipován jako otevřený, jak co se týče funkcí, tak z hlediska změn uživatelského rozhraní. Je třeba počítat se změnami v důsledku špatného odhadu potřeby funkcí i v důsledku změn potřeb za provozu.
- Systém měření by neměl obsahovat funkce a data bez jasného, ne nutně okamžitě uplatňovaného, účelu.
- Systém měření je integrální součástí systému řízení a je chápán jako prostředek podpory rozhodování a řešení problémů zvyšování výkonnosti.
- Měření – sběr metrik – je odděleno od vyhodnocování.
- Lidé by neměli mít pocit, že jsou systémem manipulováni, že jsou jen doplňkem systému.
- Cíle systému měření mají být formulovány jasně, na základě osvědčených technik, např. analýzou vstupů a výstupů.

Metriky jsou různého typu. Důležité atributy metrik:

- Frekvence zjišťování a aktuálnost. Pro operativní rozhodování jsou potřeba data v reálném čase nebo alespoň data aktuální. Pro management mohou být potřeba i data historická, např. pro vyhodnocování trendů.
- Potřebná přesnost. Jinou přesnost potřebuje operativa, např. účetnictví, jinou vrcholový management. Požadavky na přesnost podřídit tomu, jaká je dosažitelná přesnost dat.
- Snadnost vyhodnocování. Některé, především manažerské, informace je nutno zjišťovat velmi komplexními procedurami. Jiné jsou patrné přímo z dat bez vyhodnocovacích procedur.

15.2 Potíže s měřením softwaru

Metriky můžeme používat jako indikátory stavu projektu a kvality softwaru. Příkladem jsou počty zjištěných defektů, frekvence výpadků systému, procento otestovaných částí systému atd. Používání těchto metrik se v zásadě neliší od používání stejných nebo obdobných indikátorů v jiných oblastech techniky a není tedy spojeno s žádnými novými jinde neznámými problémy. Jiná je situace v případě, chceme-li používat softwarové metriky jako prostředek odhadu nebo, což je obdobný problém, pro odvození empirických závislostí, jako je závislost pracnosti na velikosti systému měřená ve vhodných jednotkách, např. řádcích programů. Hlavním problémem softwarových metrik je velký rozptyl pozorovaných hodnot. Důvody rozptylu dat jsou následující:

1. Produktivita práce programátorů (měřeno v jednotkách délky programu na jednotku času) silně závisí na typu realizovaného softwaru. Tak např. dávkové systémy a tvrdé systémy reálného času mají poměr produktivit měřených v počtech řádků za den 1:10 až 1:100.
2. Všechny kvantitativní charakteristiky programů silně závisí na kvalitě programátorů. Z experimentů (srv. Weinberg, 1971) i z praxe je známo, že:
   - mezi programátory jsou velké rozdíly v produktivitě práce, až 1:20, tj. poměr pracovního dne k pracovnímu měsíci;
   - výsledné programy mají poměr 1:10 v rychlosti i v délce programu; kratší programy píší obvykle lepší programátoři a tyto programy bývají i rychlejší;
   - programátoři jsou schopni vědomě ovlivnit různé charakteristiky programů, jako je délka programu, počet proměnných, rychlost práce programu atd., pokud mohou libovolně měnit ostatní vlastnosti programů. Jsou např. schopni napsat velmi rychlý program, pokud nemusí brát ohled na jeho délku;
   - prostředí, v němž se software realizuje se rychle mění. Mění se i paradigmata vývoje softwaru;
   - jednotlivé projekty se od sebe dosti liší, každý IS je z technického hlediska spíše originál než sériový výrobek. A pro každý originál jsou hodnoty metrik různé.
3. Hodnoty softwarových metrik jsou silně závislé na řadě dalších faktorů. Faktory ovlivňující pracnost jsou zejména
   - druh softwaru (viz kap. 1): míra interakce od dávkových až po tvrdé systémy reálného času, závažnost důsledků selhání, od nevýznamných škod, přes ekonomické ztráty až k ohrožení životů. V neposlední řadě je důležitá velikost systému;
   - ostré až obtížně splnitelné termíny realizace;
   - použití moderních projekčních technik a technik vývoje;
   - kvalita zúčastněných;
   - omezení hardwaru a softwaru. Pokud systém využívá zdroje jako je rychlost a paměť více než na dvě třetiny, vzrůstá značně pracnost řešení.

Zlé jazyky tvrdí, že za výše uvedených okolností jsou softwarové metriky sbírkou náhodných čísel. Při detailnějším pohledu není však situace tak pesimistická. Průměrné hodnoty různých softwarových metrik bývají relativně stálé. Je např. známo, že u programů střední složitosti bývá produktivita asi 3 000 řádků na programátora a rok. Větší produkty, např. kompilátory, bývají realizovány za 4–6 let a v polovině této doby „celkem fungují“. To indikuje existenci celkem stabilních empirických závislostí, které se dají použít pro odhady a také pro hodnocení účinnosti různých technik tvorby softwaru.

Mnohé metriky se využívají především v kontextu daného projektu. Jedná se především o metriky charakterizující kvalitu softwaru, jako je střední doba mezi poruchami. Jiným příkladem je sledování těch částí systému,

pro které nabývají metriky výjimečných hodnot. Pro mnohé metriky je pro daný projekt nutné sledovat trendy, např. trend počtu selhání systému, nebo procento otestovaných modulů.

Při opakovaných realizacích podobných projektů se hodnoty metrik obvykle příliš neliší. U principiálně nových řešení bývá odhad hodnot metrik nejistý, neboť tyto hodnoty značně kolísají. Rozptyl metrik se silně snižuje při dodržování standardů. Účinným prostředkem kontroly dodržování standardů jsou inspekce a správa konfigurace.

Kolísání hodnot metrik nebývá pro konkrétní řešitelský tým příliš výrazné – tým obvykle řeší pouze systémy určitého druhu s jistými poměrně úzce vymezenými vlastnostmi. Standardizace metod a postupů tvorby softwaru snižuje kolísání metrik. Většina softwarových firem v zájmu snížení rizik při řízení projektů nevítá příliš velké rozdíly v produktivitě. Inspekce umožňují lepší dodržování standardů a šíření výhodných řešení a omezování nevhodných technik. Pro metriky kvality, jako je počet selhání za jednotku času, doba mezi poruchami a doba nápravy chyby, námitka o rozptylu hodnot neplatí, neboť metriky kvality především vyjadřují vlastnosti daného produktu. Tyto metriky se vztahují k aktuálnímu stavu produktu, který buď je, nebo není dostatečně spolehlivý.

15.3 Druhy softwarových metrik

Softwarové metriky jsou dvojího druhu. První skupinu tvoří metriky, jako je délka programů, počet podprogramů/metod atd. Tyto metriky lze zjistit kdykoliv po skončení vývoje. V anglické literatuře se proto nazývají after process metrics. Jsou to metriky zjistitelné formální analýzou textů a jsou tedy zjevné – explicitní; budeme je proto nazývat *explicitní metriky*. Ostatní metriky, většinou jsou to metriky zjistitelné pouze během vývoje softwaru – in process metrics, nazveme *implicitní metriky*.

Nejdůležitější implicitní metriky jsou celková spotřeba prací – *Prac*, doba řešení – *Doba*, průběh velikosti týmu v čase *team* a produktivita *Prod* – počet jednotek délky (řádků) za jednotku času (měsíc) a *Fail* – počet selhání či výpadků za jednotku času. Metriky mohou být číselné hodnoty i posloupnosti hodnot, např. průběh velikosti týmu v čase. Většina metrik se týká programů. Řadu metrik (délka, různé strukturální údaje, např. počet funkcí a rozsah požadovaných dat, počet chyb zjištěných při inspekcích atd.) lze použít i pro dokumenty vznikající během počátečních etap vývoje softwaru.

Norma ISO 9126 (je již přijata i jako ČSN) orientovaná na kvantitativní charakteristiky kvality softwaru a řízení prací rozeznává metriky interní, tj. takové, které potřebuje řešitelský tým pro řízení a kontrolu prací, příkladem je aktuální procento otestovaných modulů systému, a externí, které charakterizují uživatelské vlastnosti produktu. Interní metriky mohou být explicitní, např. počet tříd v programech, i implicitní, např. podíl zkontrolovaných programů. V současné době je studováno několik set metrik. My se v dalším zaměříme pouze na ty metriky, které se běžně používají.

15.3.1 Explicitní metriky

Nejdůležitější explicitní metriky jsou:

Del – délka produktu v řádcích. U programů se nepočítají komentáře. *Del* programů se někdy udává v lexikálních atomech. Do metriky *Del* se někdy nezahrnují deklarace proměnných a záhlaví podprogramů, protože se ukazuje, že takto vyhodnocovaná metrika má příznivější vlastnosti. Přes intuitivní jednoduchost není *Del* právě jednoduché používat. Metrika *Del* je základem metodiky odhadu COCOMO (kap. 16).

	Del	*Noper*	*Nrnd*	*Soper*	*Srnd*
`begin`	1	1		1	
`var x,y:real;`	7	5	2	5	2
`x:= y*2.0+sin(x)+2.0`	12	7	5	5	1
`end.`	1	1		1	
Celkem	21	14	7	12	3

Tab. 15.1: Příklad výpočtu hodnot metrik jednoduchého programu v jazyce Pascal.

Srnd – rozsah slovníku operandů. Tato metrika se týká programů. Operand je buď konstanta (např. celé číslo 10, nebo řetězec znaků „xyz“, v terminologii programování literál), nebo proměnná, např. x. *Srnd* je pak počet logicky odlišných operandů vyskytujících se v programu (viz tab. 15.1).

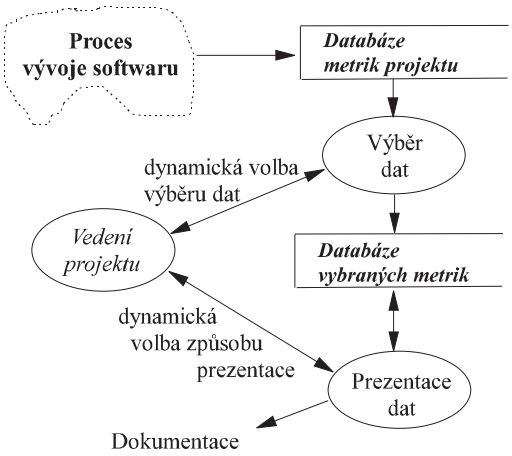
Nrnd – počet výskytů operandů v programech.

Soper – rozsah slovníku operací a delimiterů. Tato metrika udává, kolik program obsahuje významem různých znaků operací ($-$, $+$, $*$ atd.), jmen podprogramů (`sin`, `tan`, `put`), delimiterů (`: () begin if real` atd.).

`table failure`	záznam selhání / závad zjištěných při testech/oponenturách
`id`	identifikátor záznamu
`id-pers`	identifikátor osoby, která pořídila záznam
`cas`	časové razítko, kdy zaznamenáno
`cas1`	doba opravy
`chain`	vazba na defekty, které způsobily selhání
`kde`	údaj o místě, resp. o funkci, kde se projevilo
`kdo-opr`	kdo prověřil opravu
`popis`	text popisující projevy selhání
`opr`	explicitní potvrzení doby, kdy bylo skutečně opraveno
`table defect`	záznam o místě, které bylo třeba změnit jako příčinu selhání
`id`	identifikátor záznamu
`id-pers`	kdo zapsal
`kde`	identifikace místa výskytu
`vznik`	která etapa způsobila (resp. odkaz na příslušné dokumenty)
`cas-zj`	kdy zjištěno
`cas-opr`	kdy opraveno
`chain`	vazba na záznamy selhání, k nimž se vztahuje
`kdo-opr`	kdo provedl opravu
`popis`	text popisující defekt

Tab. 15.2: Data záznamu selhání a defektu. Mezi záznamy failure a defect je vztah m:n.

Noper – počet výskytů znaků operací, delimiterů a jmen podprogramů (metod) v programu Výpočet metrik jednoduchého programu v jazyce Pascal je v tab. 15.1. Délka je dána počtem lexikálních atomů. Tato metrika je součtem *Noper* a *Nrnd*.



Obr. 15.1: Informační systém metrik softwarového projektu.

Fun(f) – Tato metrika se vyhodnocuje pro každou funkci nebo metodu f. V programech se rovná po čtu parametrů funkce f. V návrhu specifikací udává pro určitou funkci počet různých vstupních a výstupních dat.

Fun(P) – počet podprogramů nebo metod, specifikovaných / navržených / naprogramovaných v dokumentu nebo programu P.

Tab(P) – počet databázových tabulek používaných v programu/modulu P.

Users – Maximální počet uživatelů, pro které je systém plánován.

McCabe – počet podmíněných příkazů (`if`), příkazů cyklu (`for`, `while`,...) a přepínačů (`case`) v programu. Metrika *McCabe* je dobrým indikátorem složitosti programů. Jejím nedostatkem je, že není citlivá na hloubku vloženosti podmíněných příkazů.

In, Out, Qer, File, Filee – složitost příkazů vstupu, výstupu, dotazů na terminál a operací se soubory interními a se soubory společnými s jinými aplikacemi. U databází složitost SQL dotazů. Tyto metriky se v následující kapitole používají v odhadech pracnosti a doby řešení.

FanIn, FanOut – míry indikující složitost rozhraní tříd, modulů a aplikací. *FanIn* udává počet logicky různých typů dat vstupujících do dané entity (např. modulu). *FanOut* je obdoba *FanIn* pro vystupující datové toky. Často se používá metrika $Fan1 = \sum_{modul\ i}(FanIn_i \cdot FanOut_i)^2$.

Následující metriky se používají pro objektově orientované specifikace, objektově orientované návrhy a programování.

Class(P) – počet tříd v návrhu či programu či specifikaci.

Attr(c) – počet atributů v třídě c.

Metod(c) – počet metod třídy c.

Par(c,m) – počet parametrů metody m patřící třídě c.

Asoc(c) – počet tříd, jejichž metody volá třída c.

Asoc(c,m) – seznam metod (včetně počtů jejich parametrů) cizích tříd, které volá metoda m třídy c.

Supercls(c) – počet nadtříd třídy c.

Subcls(c) – počet potomků třídy *c*.

Z metrik s „parametry“ se určují metriky globálně charakterizující ten který dokument či program, např. součet metrik *Supercls(c)* pro všechny třídy.

15.3.2 Implicitní metriky

Implicitní metriky jsou pro řízení prací na vývoji či customizaci softwaru nejdůležitější. Základním problémem řízení projektu je odhad implicitních metrik, jako je cena, pracnost a doba řešení. Nejdůležitější implicitní metriky jsou

Prac – pracnost (effort) realizace, spotřeba normojednotek práce na realizaci / customizaci softwaru nebo etapy realizace / customizace softwaru. Měří se obvykle v člověkoměsících.

Doba – doba v měsících potřebná k provedení softwarového díla případně doba potřebná k provedení jednotlivých etap či částí.

Prod – produktivita, počet jednotek délky vytvořených za člověkoměsíc.

team(t) – velikost týmu (počet osob) v čase t, měřeno od začátku prací. Tato metrika umožňuje postupné zpřesňování odhadů pracnosti a doby řešení během vývoje projektu.

Team – průměrná velikost týmu.

Fail(t,p) – počet selhání systému / části p detekovaných při testování či provozu v čase t. Při provozu hlásí selhání zákazníci. Obvykle se udává po dnech nebo týdnech. Tato metrika je důležitou mírou kvality. Při inspekcích je hodnota metriky *Fail* dána počtem chyb zjištěných při inspekci.

Defect(t,p) – počet míst v programech, která bylo nutno opravit pro odstranění selhání nebo pro nápravu selhání v čase t (den, týden) v části p, normalizovaný pro 1 000 řádků (1000 · počet defektů/délka).

Defect1(e1,e2,t,p) – počet defektů v části p vzniklých v etapě řešení $e1$ a zjištěných v etapě $e2$ v době t. Tato metrika a metriky z ní odvozené jsou účinnou mírou efektivnosti inspekcí a testů.

Prob(t) – počet problémů hlášených uživateli systému, tedy nikoliv jen počet selhání – např. problémy s instalací a ovládáním. Důležitá externí metrika pro vyhodnocování kvality produktu.

Satisf(t) – průměrná míra spokojenosti zákazníků v čase t. Spokojenost se udává ve stupnici 1 až 5 (nejlepší). Lze vyhodnocovat trendy. Existují metody odhadu vývoje úspěšnosti produktu na trhu z aktuálních hodnot metrik *Satisf* (Babich, 1992).

DobaOpr – průměrná doba opravy selhání.

Zmeny(f,t) – počet změněných míst souboru f v čase t (týdnu/dni). Tato metrika se snadno zjišťuje a její trendy mohou v průběhu prací poskytnout cenné informace.

MTBF(t) – (Mean Time Between Failures): střední doba mezi poruchami (v určitém období, např. týdnu, t).

PracInsp – pracnost inspekcí.

PracDefect(d) – pracnost odstranění defektu d.

15.4 Sběr a vyhodnocování metrik

Implicitní metriky jsou zjistitelné jen tehdy, jsme-li ochotni věnovat jejich zjišťování a vyhodnocování dostatečné úsilí. Sběr metrik se bohužel kromě takových dat, jako jsou hospodářské výsledky, velmi často považuje za šikanování a také jako nebezpečný prostředek postihu zúčastněných. Podobně jako při inspekcích je třeba se vyvarovat zneužití metrik pro postih pracovníků. Bez dobré vůle nelze při sběru metrik očekávat dobré výsledky.

Explicitní metriky je vhodné zjišťovat použitím nějakého formálního aparátu. Zvláště snadné je to u délky. Velmi jednoduché je zjišťování metriky *Změny*. Poměrně snadno lze zjistit hodnoty *Doba* a *Prac* za celý projekt. Problematičtější bývá velikost týmu. Nebývá výjimkou, že se někteří pracovníci účastní více projektů. Hodnoty metriky *team* je proto třeba upravovat podle toho, jakou část své pracovní kapacity projektu jednotliví pracovníci věnují, případně hodnotu metriky *team* měřit výkonem, tj. počtem hodin na člověka a den.

Zjišťování a vyhodnocování hodnot *Prac* a *Doba* pro části projektu, např. pro jednotlivé moduly či dokonce třídy, je nejlépe provádět pomocí IS. Záznam o selhání a opravě by měl minimálně obsahovat údaje z tab. 15.2. Tyto údaje by měly být ukládány a vyhodnocovány kombinací prostředků IS a vhodného nástroje pro prezentaci dat. Často postačuje i tabulkový kalkulátor, do něhož se data exportují.

Metriky *Fail* a *Defect* jsou hlavním prostředkem řízení kvality. Pro jejich zjišťování stačí při inspekcích, při testování a při úpravách dokumentů a textů zaznamenávat data o selháních a opravách. Tyto metriky musí být de facto vyhodnocovány při použití normy ISO 9000–3. Totéž platí pro metriku *MTBF*. Pokud data z tabulky 15.2 uložíme do IS, lze snadno vyhodnocovat informace o tom, jak účinně se opravují chyby, kolik chyb „prošlo“ inspekcemi a v kterých částech programů či dokumentů je nejvíce chyb. Pokud je u každého dokumentu či programu uveden autor, lze sledovat i výkonnost, především však spolehlivost pracovníků atd. Tyto údaje je možné kombinovat s hodnotami explicitních metrik pro vyhodnocování četnosti chyb na jednotku délky, zjišťování vazeb mezi strukturální složitostí (počet tříd, počet vazeb mezi třídami atd.) a implicitními metrikami (nejčastěji počet chyb). Možná struktura takového IS je na obr. 15.1.

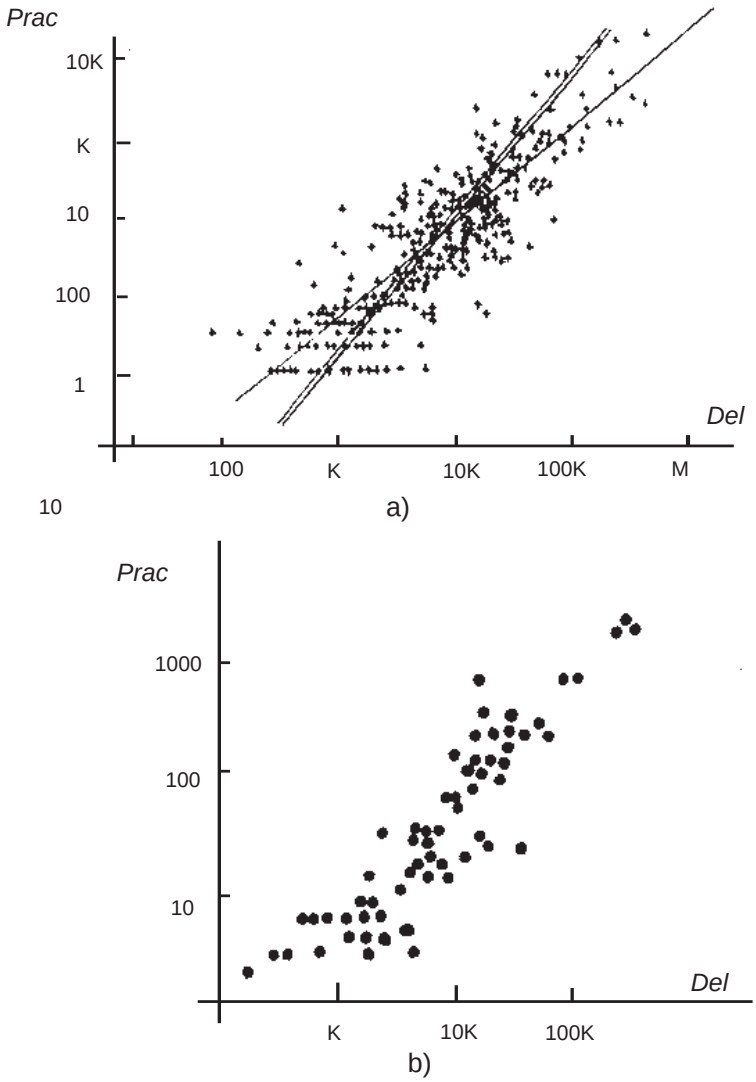
Analýza metrik může být velmi efektivní i při použití docela jednoduchých metod. Při provádění inspekcí lze sledovat počet zjištěných závad (metrika *Defect*). Programové moduly i dokumenty s vysokou hodnotou *Defect* budou pravděpodobně obsahovat mnoho závad i po opravách. Na takové moduly je vhodné zaměřit pozornost: provádět opakované oponentury, přepracovat dokument atd. Důvod tohoto postupu je následující. Je-li pravděpodobnost nalezení závady 75 % a bylo-li v nějakém modulu zjištěno patnáct závad, pak modul pravděpodobně obsahuje ještě přibližně pět závad, tj. 33 % nalezených chyb. V modulu, ve kterém byly nalezeny pouze dvě závady, nezbyla s velkou pravděpodobností závada žádná. Sledování modulů s malým počtem zjištěných závad je též důležité. Důvodem malého počtu zjištěných závad může být vyšší kvalita oponovaného materiálu (programu/dokumentu) nebo nižší účinnost oponentur nebo testů. Sledujeme tedy extrémní hodnoty metrik. Proto tento postup nazýváme *řízení na extrém*.

Řízení na extrém se používá i v případě explicitních metrik. Není žádoucí, aby byly velké rozdíly v explicitních metrikách, např. v délce, mezi moduly / částmi / dokumenty. Pokud k tomu došlo, je vhodné uvažovat o restrukturalizaci / přeprogramování. V případě objektově orientovaných technik je vhodné sledovat třídy s extrémními hodnotami počtu metod a počty tříd, jejichž metody dané třídy volají. Důležitou informaci obsahují i extrémní doby odstraňování defektů a doby reakcí na stížnosti na systém od zákazníků.

Z doby, kdy je tým největší a tedy i intenzita práce nejvyšší, lze poměrně přesně odhadnout dobu řešení *Doba* a spotřebu práce *Prac* (viz níže).

15.5 Empirické závislosti softwarových metrik

V této části uvedeme různé empiricky zjištěné vztahy mezi metrikami. Zjištěné vztahy jsou základem metod odhadu hodnot metrik důležitých pro uzavírání smluv (pracnost, termíny) a také pro hodnocení metodik vývoje softwaru. Ve zbytku této kapitoly budou c, C, c_1, $C_1, \ldots$ vhodné konstanty. Řada závislostí, které budeme



Obr. 15.2: a) Pracnost a délka programů, zbraňové systémy. b) Pracnost a délka programů, IBM.

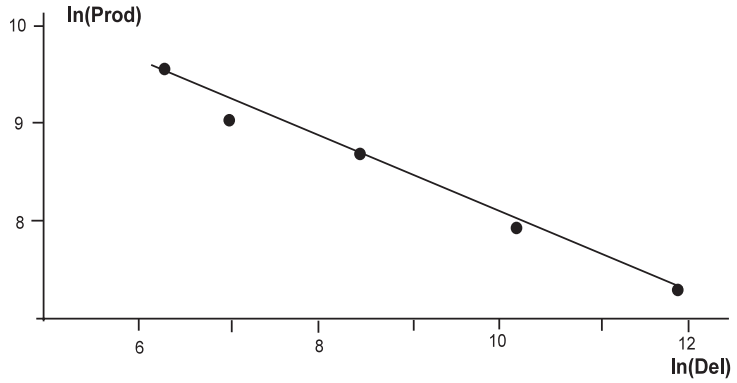
diskutovat, má charakter odhadu. Výrok „*B* je odhadem veličiny *A*“ zapisujeme.

$$A \mathrel{\widehat{=}} B$$

Uveďme příklad. Průměr vah nějaké skupiny obyvatel nějakého města je odhadem průměrné váhy všech obyvatel daného města. Skupinu obyvatel můžeme volit různě. Postupujeme-li tak, aby průměr odhadů B byl roven A, říkáme, že B je nestranným odhadem hodnoty A. Odhad je kvalitní, má-li malý rozptyl. Přesnou definici lze nalézt v libovolné učebnici matematické statistiky, např. v (Anděl, 1993). Pro programy platí, že

$$Del_{řádky} \mathrel{\widehat{=}} c \cdot Del_{atomy}$$

Tento odhad je nestranný a má při dodržování podnikových norem psaní programů malý rozptyl.



Obr. 15.3: Vztah mezi produktivitou a délkou programu. *Prod* je udávána v řádcích za rok, délka v řádcích. Za střední hodnoty délky programů je vzat střed intervalu logaritmů z tab. 1. Pro okrajové třídy jsou zvoleny hodnoty délky 1 000 000 a 300.

Z řady průzkumů je známo, že procento prací věnovaných různým etapám vývoje softwaru je celkem stálé. Pracnost kódování tvoří asi 20 % pracnosti vývoje softwaru a jen mírně klesá s rozsahem úkolu. Podíl součtu pracností návrhu a kódování, příp. kódování a testování je prakticky nezávislý na velikosti projektu (viz Beck, Perkins, 1983). V dalším budeme často studovat zákonitosti tvaru

$$A \mathrel{\widehat{=}} c \cdot B^t,$$

kde c je vhodná konstanta. Pro naše úvahy bude rozhodující hodnota exponentu t. Hodnota konstanty c bude přitom záviset na tom, zda bereme v úvahu celý životní cyklus softwaru nebo jeho část, hodnota exponentu však není vzhledem k výše uvedenému předpokladu ovlivněna tím, zda máme k dispozici data o celém cyklu vývoje softwaru nebo jen o některých etapách, např. o kódování a testování, jako je tomu v (Halstead, 1977).

Budeme vycházet z úvah a dat uvedených především v knihách Halsteada, 1977, a článku Walstona, Felixe, 1977. Poznatky a závěry těchto autorů se staly součástí řady softwarových norem (např. IEEE 1045). Data získaná různými autory jsou kompatibilní jen z části, neboť se např. týkají různých etap realizace softwaru a není vždy jasné, zda se týkají samostatných programů nebo programů, které jsou částí většího celku atd. Dají se však použít pro vysvětlení řady důležitých jevů. Poznatky výše uvedených autorů byly zahrnuty do softwarových norem, např. ISO 9126 a IEEE 1045. Data budeme analyzovat tak, jak je to obvyklé ve fyzice. Budeme tedy:

a) Odvozovat zákonitosti z empirických dat.

b) Zákonitosti budeme považovat za platné, pokud nejsou ve sporu s pozorovanými daty (a také ve sporu mezi sebou vzájemně) a pokud mají schopnost vysvětlovat a predikovat.

Bod b) budeme považovat za jistou formu experimentálního ověření zákonitostí. Při diskuzi o zákonitostech budeme používat metody nepřímých důkazů, jako jsou odkazy na trendy v metodologii programování, potvrzení existence nějakého jevu různými zákonitostmi, odvození nějakého zákona z jiných zákonů atd. Jedná se opět o přístup obvyklý ve fyzice, proto se studium empirických závislostí při tvorbě softwaru někdy nazývá softwarová fyzika. Musíme si však být stále vědomi toho, že zákony platí „v našem vesmíru“, tj. za „obvyklých“, ne však vždy úplně známých podmínek.

15.5.1 Pracnost a produktivita při programování

Řada souborů dat o realizaci softwaru dává možnost analyzovat faktory ovlivňující pracnost realizace softwaru.

Výsledky analýzy dat z obr. 15.2 standardními postupy metody analýzy regrese prováděné pro log(*Del*) jako nezávislou veličinu a log(*Prac*) jako závislou veličinu vedly shodně ke vztahu

$$\log(Prac) \mathrel{\widehat{=}} c_1 + t \cdot \log(Del), \tag{15.1}$$

čili po odlogaritmování

$$Prac \mathrel{\widehat{=}} c_2 \cdot (Del)^t. \tag{15.2}$$

Výsledky standardní regresní analýzy pro dva největší známé soubory dat obsahující údaje o délkách programů a jejich pracnosti ukázaly, že (obr. 15.2, viz Shooman, 1983)

$$0.94 < t < 0.97. \tag{15.3}$$

To je překvapující, poněvadž je všeobecně známo, že produktivita práce programátora (měřeno v jednotkách délky programů za jednotku času) je pro velké projekty menší. Toto zjištění potvrzují i fakta publikovaná v (Martin, 1985 a 1986); viz tabulka 15.3 a obr. 15.3. Z definice produktivity plyne

$$Del = Prac \cdot Prod. \tag{15.4}$$

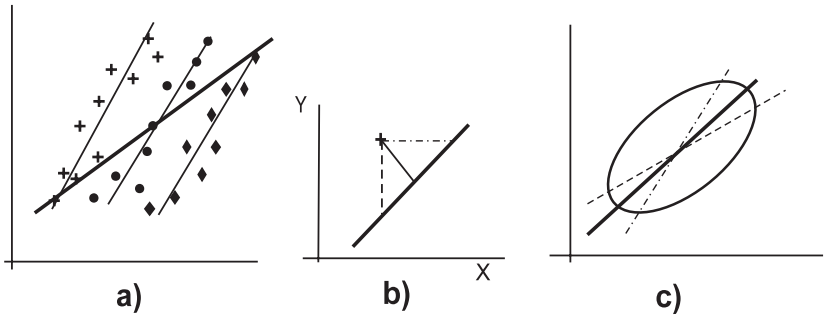
Platí-li však 15.2 a položíme-li $t = 1 + a$, dostaneme

$$\begin{aligned} Del &\mathrel{\widehat{=}} c_2 \cdot (Del)^{1+a} \cdot Prod, \\ Prod &\mathrel{\widehat{=}} 1/c_2 \cdot (Del)^{-a}. \end{aligned} \tag{15.5}$$

Podle tab. 15.2 (srv. obr. 15.3) by ale mělo být $a > 0$ $(1/10 < a < 1/4)$, avšak podle zveřejněných výsledků, viz 15.3, by mělo být $a = -0.05$. Tento zdánlivý rozpor je vyvolán chybným použitím analýzy regrese. Jinými slovy použití formálních matematických metod může vést k chybným závěrům v případě, že jim ne zcela přesně rozumíme. Tento fakt musíme mít na paměti, když zamýšlíme při vývoji IS používat formalizované metody, např. při specifikaci požadavků. Pokud nemáme používané metody plně zvládnuty, můžeme dosáhnout horších výsledků než při „obvyklém“ neformálním postupu. Formální metody mají řadu výhod a je vhodné je používat, vyžaduje to ale dostatečnou kvalifikaci. V každém případě by měly být doprovázeny neformálním intuitivním vysvětlením, tak jak je ostatně obvyklé i u dobrých matematických článků.

Typ aplikací	Průměrný počet řádků	Produktivita v řádcích za člověkorok
Extrémně rozsáhlé	$> 500\,000$	800
Velmi rozsáhlé	64 000 ... 500 000	1 300
Rozsáhlé	16 000 ... 64 000	2 000
Střední	2 000 ... 16 000	4 000
Malé	500 ... 2 000	8 000
Velmi malé	< 500	15 000

Tab. 15.3: Závislost produktivity práce programátora (měřeno v řádcích za člověkorok) na délce programu v řádcích. Podle (Martin, McClure, 1985a).



Vysvětlení rozporných výsledků analýzy regrese.
a) Regresní přímky pro jednotlivé týmy (tence) mají větší sklon než regresní přímka pro data všech týmů (silná čára).
b) ——— odchylka uvažovaná při ortogonální regresi,
------ odchylka uvažovaná při standardní regresi,
– · – · – odchylka uvažovaná, je-li Y nezávisle proměnná.
c) Regresní přímky pro data pokrývající elipsu. Čím je elipsa širší, tím je menší sklon regresní přímky pro standardní regresi a X jako nezávislou proměnnou.

Obr. 15.4: Vysvětlení zavádějících výsledků analýzy regrese.

Vraťme se nyní k problému růstu pracnosti s rozsahem programu. Chybný závěr, že $t < 1$, vychází z následující vlastnosti analýzy regrese. Předpokládejme, že máme k dispozici data od několika týmů. Pro tým T_i platí pro pracnost realizace $Prac_i$ vztah

$$\log(Prac_i) \mathrel{\widehat{=}} c_i + t_i \cdot \log(Del_i). \tag{15.6}$$

Data pro jednotlivé týmy se liší hodnotou c_i a rozdíly hodnot t_i jsou poměrně malé (obr. 15.4). Provedeme-li regresní analýzu pro všechna data, dostaneme

$$\log(Prac) \mathrel{\widehat{=}} c + t_0 \cdot \log(Del) \tag{15.7}$$

kde t_0 může být podstatně menší než všechna t_i, viz obr. 15.4.

Pro hodnocení účinku některých technik je třeba znát hodnotu koeficientu t pro jednotlivé týmy. K efektu vyjádřenému na obr. 15.4 dochází proto, že při vyhodnocování lineární regrese se v rovině se souřadnicemi hledá

	Zdroj	Nezávisle proměnná X	Závisle proměnná Y	koeficient regrese	koeficient korelace
1.	Norden,1978	$\log(Del)$	$\log(Prac)$	1.125	0.85
2.	Walston, Felix, 1977	$\log(Del)$	$\log(Prac)$	1.125	0.80
3.	Putnamn, 1978	$\log(P/D^2)$	$\log(Prod)$	-0.66	-0.98
4.	Walston, Felix, 1977	$\log(Del)$	$\log(Doba)$	0.44	0.62
5.	Walston, Felix, 1977	$\log(Prac)$	$\log(Doba)$	0.40	0.77
6.	Walston, Felix, 1977	$\log(Prac)$	$\log(Team)$	0.62	0.82
7.	Christensen, Fitsos, 1981	$\log(Srnd)$	$\log(Soper)$	0.33	0.78
8.	Fitsos, 1980	$\log(Srnd)$	$\log(Soper)$	0.48	0.69
9.	Wolberg, 1981	$\log(Srnd)$	$\log(Soper)$	0.45	0.88
10.	Wolberg, 1981	$\log(Srnd)$	$\log(Soper)$	0.29	0.86
11.	Halstead, 1977	Del	$Nrnd$	1.00	0.99
12.	Halstead, 1977	$\log(Del)$	$\log(Srnd)$	0.97	0.92
14.	Halstead, 1977	$\log(Del)$	$\log(Srnd)$	0.75	0.70
15.	Halstead, 1977	$\log(Del)$	$\log(Soper)$	0.48	0.50
16.	Halstead, 1977	$\log(Del)$	$\log(Soper)$	0.42	0.30

Tab. 15.4: Tabulka výsledků ortogonální regresní analýzy pro různé soubory dat. Data zřádků 9 až 16 se týkají malých podprogramů.

přímka $a + b \cdot X$ tak, aby součet čtverců odchylek zjištěných dvojic hodnot od hledané přímky byl minimální (obr. 15.4). Odchylky se měří ve směru osy Y. Pokud se považuje Y za nezávislou proměnnou, uvažují se odchylky ve směru osy X.

Existuje tzv. ortogonální regrese, při které se hledá minimum čtverců odchylek měřených kolmo ke hledané přímce (viz Cramér, 1946). Obecně máme pro nějakou množinu údajů tři regresní závislosti:

a) $\log(Prac) \widehat{=} c1 + f_2 \cdot \log(Del)$, odchylky jsou měřeny ve směru osy $Y = \log(Prac)$,
b) $\log(Del) = c_2 + f_2 \cdot \log(Prac)$, odchylky jsou měřeny ve směru osy $X = \log(Del)$,
c) $\log(Prac) = c_3 + f_4 \cdot \log(Del)$, odchylky jsou měřeny kolmo k regresní přímce.

Pro náš případ ($f_2', f_2, f_4 > 0$) platí vždy $1/f_2' > f_4 > f_2$. f_4 je tedy nestrannější odhad směrnice regresní přímky jednotlivých týmů než f_2. Jistý náhled na vlastnosti regresních přímek dává c) v obr. 15.4, vyjadřující výsledky analýzy regrese pro data pokrývající elipsu. Pro výše zmíněné soubory dat z obr. 15.2 dává ortogonální regrese hodnotu směrnice $t = 9/8$. Tento odhad je pravděpodobně snížen tím, že v datech se projevuje vliv zaokrouhlování hodnot pracnosti vzhůru na celý počet člověkoměsíců.

Můžeme tedy učinit závěr, že pro pracnost platí odhad

$$Prac \widehat{=} c \cdot (Del)^t, \quad t = 9/8. \tag{15.8}$$

Tento odhad již není v rozporu s průběhem produktivity zobrazeném na obr. 15.3.

15.5.2 Softwarové rovnice a jejich důsledky

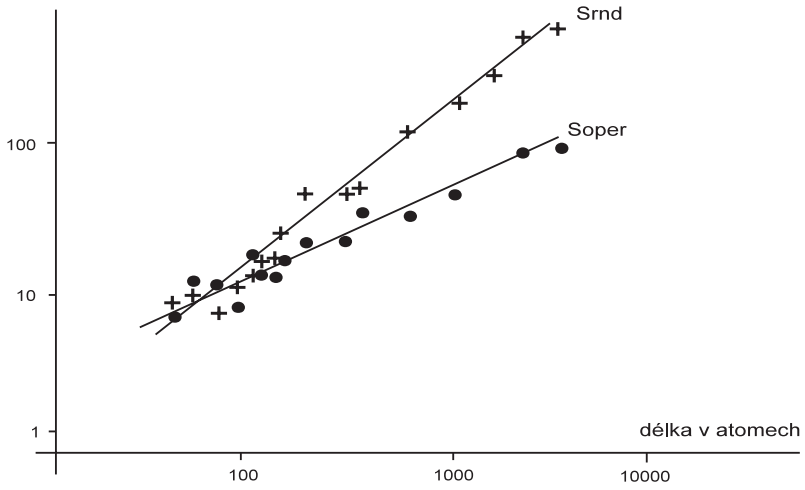
V knize (Halstead, 1977) je vyslovena hypotéza, že platí

$$Prac \widehat{=} c \cdot (Del) \cdot (Nrnd) \cdot Soper/Srnd \cdot \log(Srnd + Soper). \tag{15.9}$$

Z tab. 15.4 řádky 11 a 12 plyne, že pro řadu projektů

$$Nrnd \mathrel{\widehat{=}} c_6 \cdot Del, \tag{15.10}$$

kde $c_6 = 0.47$.



Obr. 15.5: Regrese pro *Srnd* (+) a *Soper*(•) pro krátké programy (Halstead, 1977).

Pro velké hodnoty délky *Del* se hodnota log(*Srnd* + *Soper*) mění málo, a proto lze hodnotu logaritmu aproximovat vhodnou konstantou. Vztah (15.9) pak dostane tvar

$$Prac \mathrel{\widehat{=}} c_7 \cdot Del^2 \cdot Soper/Srnd. \tag{15.11}$$

V klasických programovacích jazycích je u velkých programů míra růstu slovníku operací *Soper* určena růstem počtu procedur, u objektově orientovaných programů metod. Jestliže je program psán modulárně bez globálních proměnných s moduly/třídami omezené průměrné délky, lze očekávat, že

$$Srnd \mathrel{\widehat{=}} c_8 \cdot Del^b, \quad b > 1, \doteq 1. \tag{15.12}$$

Skutečně v programu psaném modulárně bude každý modul určité průměrné délky d obsahovat jistý průměrný počet q lokálních proměnných. Mezi lokální proměnné počítáme i formální parametry procedur a funkcí. Rozsah slovníku operací pak bude blízký hodnotě $(Del/d) \cdot q = c_8 \cdot Del$. To je skutečně pozorováno (viz tabulku 15.4 ř. 13, z části řádek 14, data v řádce 14 mají však značný rozptyl). Skutečný vztah mezi délkou a počtem operandů silně závisí na programovací metodice. Jestliže je totiž program psán pouze s globálními proměnnými, můžeme očekávat, že bude $Srnd \mathrel{\widehat{=}} c_9 \cdot Soper$, poněvadž pravidla vytváření nových proměnných jsou podobná pravidlům vytváření podprogramů. Pak ovšem dostaneme

$$Prac \mathrel{\widehat{=}} c_1 \cdot Del^2. \tag{15.13}$$

To bylo pozorováno pro pracnost prvých kompilátorů z jazyka FORTRAN, kdy nebyly lokální proměnné používány z důvodu snahy o úsporu místa v paměti. Z tabulky 15.4 můžeme odvodit (viz řádky 15, 16 viz též obr. 15.5)

$$Soper \mathrel{\widehat{=}} c_{12} \cdot Del^a, 0.25 < a < 0.4 \tag{15.14}$$

Výše jsme uvedli, že pro moderní programy platí $Srnd \mathrel{\widehat{=}} c \cdot Del$. Mělo by tedy platit

$$Soper \mathrel{\widehat{=}} c_{14} \cdot (Srnd)^a, \tag{15.15}$$

což je skutečně pozorováno (tabulka 15.4, řádky 7, 8, 9, 10). Po dosazení z (15.14 a 15.15 do rovnice 15.9 dostaneme

$$Prac \mathrel{\widehat{=}} c_{15} \cdot (Del)^{1+a},\ 0.25 < a < 0.4 \tag{15.16}$$

Dále platí

$$\begin{aligned} Srnd &\mathrel{\widehat{=}} c_{13} \cdot Del^b, \quad b \doteq 1, \\ Soper &\mathrel{\widehat{=}} c_{12} \cdot Del^a, \\ Prac &\mathrel{\widehat{=}} c_{15} \cdot Del^{2-b+a}. \end{aligned} \tag{15.17}$$

Vztah 15.14 je v kvalitativní shodě s 15.1. Vztahy 15.17 byly odvozeny pro malé programy. Pro velké programy je hodnota exponentu a v 15.17 příliš vysoká, poněvadž by podle rovnice 15.17 mělo platit přibližně

$$Prac \mathrel{\widehat{=}} c_{17} \cdot Del^{1+1/3}$$

a je pozorováno, že platí (15.4)

$$Prac \mathrel{\widehat{=}} c \cdot Del^{9/8} = c \cdot Del61 + 1/8$$

Tento rozpor vysvětlíme v následujícím paragrafu.

Vztahy 15.17 dokreslují vliv moderních programovacích metod, jako je nepoužívání globálních proměnných (pak je b velké) a používání pokud možno obecně použitelných podprogramů či metod nebo objektů (pak je a rovněž malé). Oba tyto postupy se používají a osvědčují se. Striktní zákaz používání globálních proměnných bývá často příliš omezující. Dobrý kompromis byl nalezen v objektově orientovaných technologiích, kde existují proměnné tří úrovní: lokální v metodě, atributy tříd a pro výjimečné situace globální proměnné.

Označme $S = Srnd + Soper$. Veličinu $V = S \cdot \log(S)$ nazveme objemem programu. Nejmenší možný objem programu je zápis ve formě procedury realizující požadovaný algoritmus. Takový zápis musí obsahovat:

1. Jméno procedury/funkce/metody P.
2. Parametry $p_1, \ldots, p_m$.
3. Znak konce seznamu parametrů.

Tedy například

$$P p_1 p_2 \ldots p_m)$$

Tento „program“ má dva operátory („P“ a „)“) a m operandů. Pro takový program je objem

$$V^* = (2 + m) \cdot \log(2 + m). \tag{15.18}$$

V^* je zřejmě v jistém smyslu dolní hranicí hodnoty objemů programů realizujících daný algoritmus. Při tom se předpokládá, že parametry jsou zvoleny tak, že každý parametr vyjadřuje údaj/údaje reprezentující jeden logicky samostatný vstup, že tedy parametry $p_1, p_2, \ldots, p_m$ nejsou uměle sdružovány do větších datových celků. Metrika V^* inspirovala velmi úpěšnou metodu odhadu pracnosti a doby řešení známou jako metoda funkčních bodů (function points, viz kap. 16). Úroveň L programu P definujeme jako poměr

$$L = V^*/V, \tag{15.19}$$

tj. program má tím nižší úroveň, čím je delší. Halstead vyslovil hypotézu, že platí

$$Prac \mathrel{\widehat{=}} c_{16} \cdot V/L,$$

a pro L navrhl použít odhad

$$L \mathrel{\widehat{=}} 2 \cdot Srnd/(Soper \cdot Nrnd).$$

Dosadíme-li tento poslední vztah do vztahu (15.19), dostaneme vztah (15.9). Halsteadovy metriky jsou součástí několika softwarových norem, např. normy IEEE 1061–1992.

15.5.3 Efekty dekompozice

Pracnost realizace programů roste rychleji než délka programů. Mějme nyní nějaký softwarový produkt délky *Del* s pracností $Prac_1$. Předpokládejme, že stejný projekt lze realizovat pomocí n programů, každý o délce Del/n. Označme pracnost takové realizace $Prac_n$. Předpokládejme, že na návrh a realizaci spolupráce těchto n programů nespotřebujeme žádnou práci. Spotřeba práce na každý program $c_{15} \cdot (Del/n)^{1+a}$. Pak ovšem

$$\begin{aligned} Prac_n &\mathrel{\widehat{=}} n \cdot c_{15} \cdot (Del/n)^{1+a} \\ &\mathrel{\widehat{=}} n^{-a} \cdot c_{15} \cdot Del^{1+a} \\ &\mathrel{\widehat{=}} n^{-a} \cdot Prac_1. \end{aligned} \tag{15.20}$$

Čili při výše uvedených předpokladech by klesla potřeba práce n^{-a}-krát. Pro $n = 32$ a $a = 0.2$ bude $n^{-a} = 1/2$, čili pracnost by klesla dvakrát. V praxi samozřejmě nebude úspora prací tak výrazná. Jednotlivé programy budou mít různou délku, celý produkt může mít o něco větší úhrnnou délku. Samotná dekompozice není lehký úkol, vyžaduje dosti přemýšlení a jistou práci musíme věnovat na vytvoření prostředků spolupráce jednotlivých programů.

Abychom zahrnuli do výpočtu pracnosti práce na realizaci rozhraní mezi programy, předpokládejme, že návrh dekompozice a vytvoření prostředků spolupráce programů zvětší pro n spolupracujících programů pracnost každého jednotlivého programu $c_2 \cdot n^q$-krát. Vztah 15.20 pak dostane tvar

$$\begin{aligned} Prac_n &\mathrel{\widehat{=}} c_2 \cdot n^q \cdot (Del/n)^{1+a} \cdot c_{15} \\ &\mathrel{\widehat{=}} n^{q-a} \cdot c_{21} \cdot Prac_1. \end{aligned} \tag{15.21}$$

Čili pracnost se pro $a > q$ zmenší až n^{a-q}-krát. Pokud je délka komponent přibližně konstatní s průměrnou délkou $K = Del/n$ (K zůstává pro zvětšující se délku konstatní), dostaneme $n = Del/K$ a rovnice 15.21 dostane tvar

$$\begin{aligned} Prac_n &= (Del/K)^{1+q} \cdot c_{20} \cdot K^{1+a} \\ &= Del^{1+q} \cdot c_{20} \cdot K^{a-q} \\ &= c_{21} \cdot Del^{1+q}, \quad c_{21} = c_{20} \cdot K^{a-q}. \end{aligned} \tag{15.22}$$

Jinými slovy míra růstu pracnosti je za těchto podmínek dána mírou růstu pracnosti vývoje rozhraní. Poněvadž se velké systémy vždy nějakým způsobem dekomponují, vysvětluje vztah 15.22, proč je exponent míry růstu pracnosti $1+a$ pro malé programy větší než pro velké programové komplexy. Při použití standardizovaných řešení je hodnota q ve vztahu 15.22 blízká nule. V dávkových metodách programování úloh zpracování dat se postupuje podle následujícího schématu:

a) Na základě návrhu požadovaných funkcí nebo dosavadní praxe se zvolí datové struktury a jejich organizace v souborech.
b) Pro jednotlivé funkce se navrhnou a realizují jednotlivé programy. Každý program implemetuje jednoduchý algoritmus, jehož vstupy i výstupy jsou soubory.
c) Programy se volí co nejjednodušší. Složitější funkce se člení na jednodušší a ty se pak realizují. To např. vede k tomu, že se návrh a realizace výstupu na tiskárnu odděluje od vlastní logiky programu.

Úspěch metod programování spolupracujících aplikací umožnil, aby se i interaktivní systémy koncipovaly jako sítě spolupracujících programů / aplikací, které mohou být téměř nezávisle realizovány, případně koupeny. Hlavním přínosem dekompozice tedy není pouze úspora práce podle vztahu (15.20). Pokud je systém dekomponován, je totiž podstatně snadněji udržovatelný. Defekty lze lokalizovat kontrolou komunikace mezi programy, změny jsou obvykle omezeny na jediný program. Systém je také modifikovatelný, modifikace lze provádět výměnou jednotlivých programů. Lze také snáze využívat produkty třetích stran a používat řadu specifických technik (kapitola 11). Možnosti technologie spolupráce aplikací nemohou být dosud plně využity, poněvadž zatím nejsou k dispozici obecně akceptované normy spolupráce aplikací.

15.5.4 Vliv napjatých termínů

Dobu řešení softwarového projektu nelze libovolně zkracovat. Termíny řešení projektu mohou být „mírné“ a jejich řešení pak nevyžaduje nadměrné úsilí a nebývá problém termíny zkrátit. Od jisté meze je zkracování termínů možné jen za podmínky prudkého nárůstu pracnosti, až nakonec nebude další zkracování termínů prakticky možné (obr. 15.5).

Pro určitý projekt existují tři typy termínů:

– Optimální – zkrácení termínů, pokud je dohodnuto včas, neznamená podstatný nárůst pracnosti.
– Napjaté – řešení v termínu je možné, vyžaduje však značné pracovní vypětí; zkrácení termínu znamená prudký nárůst práce.
– Nereálné – projekt nelze v daném termínu dokončit.
– Měkké – občas se vyskytne případ nepřiměřeně dlouhých termínů řešení. I v tomto případě může dojít k nárůstu pracnosti z toho důvodu, že není dostatečný tlak na systematickou a soustavnou práci (viz *b)* na obr. 15.5).

Z tabulky 15.4 řádek 5 a z analýzy dat z obr. 15.6, *a*), plyne, že

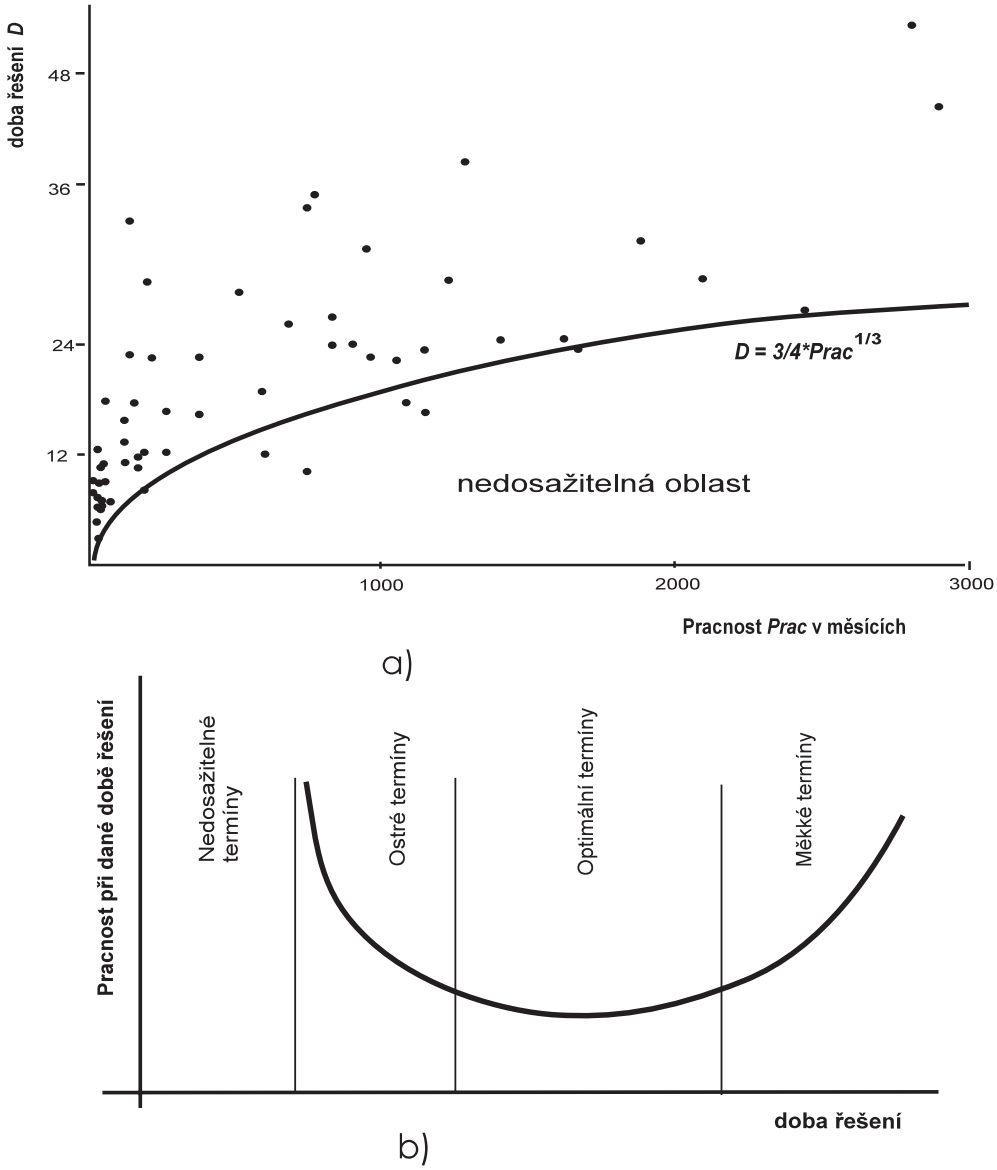
$$Doba \mathrel{\widehat{=}} c \cdot Prac^{d}$$

kde $0.3 < d < 0.5$. Pro téměř žádné projekty neplatí $Doba < 3/4 \cdot Prac^{1/3}$. Termíny větších projektů jsou obvykle napjaté. Pro takové projekty byla získaná regrese z řádku 3 tabulky 15.4. Podle tohoto řádku

$$\log(Prod) \mathrel{\widehat{=}} c_{25} - 0.67 \cdot \log(Prac/Doba^{2}) \tag{15.23}$$

čili po odlogaritmování

$$Prod \mathrel{\widehat{=}} c_{26} Prac^{-2/3} Doba^{4/3} \tag{15.24}$$



Obr. 15.6: a) Nedosažitelná oblast. Prakticky neexistují programy, pro které pro dobu řešení D platí $Doba < 3/4 \cdot Prac^{1/3}$. D – měsíce, $Prac$ – člověkoměsíce. b) Závislost pracnosti na době řešení. N – nedosažitelná oblast, A – oblast napjatých termínů, B – oblast stability, C – nepřiměřeně dlouhá doba řešení.

Poněvadž podle definice produktivity a pracnosti $Prac = Prod \cdot Del$, dostáváme z 15.24 vynásobením $Prac$ a úpravou „Putnamovu rovnici“ (Putnam, 1978)

$$Del \mathrel{\widehat{=}} c_{26} Prac^{1/3} Doba^{4/3} \tag{15.25}$$

Pro pracnost dostaneme z 15.25 a řádků 1 a 2 tabulky 15.4

$$(1/c) Prac^{8/9} \mathrel{\widehat{=}} c_{26} Prac^{1/3} Doba^{4/3}.$$

Takže

$$Doba \mathrel{\widehat{=}} c_{27} Prac^{5/9 \cdot 3/4} = c_{28} Prac^{0.41} \tag{15.26}$$

Což je v dobré shodě s řádkem 6 tabulky 15.4. Podle definice je doba řešení rovna pracnosti dělené průměrnou velikostí týmu, tj.

$$Doba = Prac/Team$$

Podle řádku 6 tabulky 15.4

$$Team \mathrel{\widehat{=}} c_{29} \cdot Prac^{0.62}$$

Po dosazení

$$\begin{aligned} Doba &\mathrel{\widehat{=}} Prac/(c_{29} \cdot Prac^{0.62}) \\ &= c_{30} \cdot Prac^{0.38} \end{aligned} \tag{15.27}$$

Vztah 15.25 můžeme použít o odhadu vlivu zkracování napjatých termínů. Proveďme následující myšlenkový pokus. Předpokládejme, že je nějaký projekt realizován nezávisle dvakrát s parametry:

$$Del_A, Doba_A, Prac_A$$

$$Del_B, Doba_B, Prac_B.$$

Nahraďme dále ve vztahu (15.25) znak $\mathrel{\widehat{=}}$ znakem rovnosti, tj. předpokládejme, že přibližně platí

$$Del = c \cdot Prac^{1/3} \cdot Doba^{4/3} \tag{15.28}$$

Vydělením „rovnic“ 15.28 pro obě realizace dostaneme, že přibližně platí

$$Del_A/Del_B = (Prac_A/Prac_B)^{1/3} \cdot (Doba_A/Doba_B)^{4/3}. \tag{15.29}$$

Poněvadž programy psané ve spěchu bývají delší, můžeme předpokládat $Del_A/Del_B \geq 1$. Ze vztahu 15.29 pak dostaneme

$$1 \leq (Prac_A/Prac_B)^{1/3} \cdot (Doba_A/Doba_B)^{4/3}.$$

Odtud plyne, že přibližně platí

$$Prac_A = Prac_B \cdot (Doba_B/Doba_A)^{4/3}. \tag{15.30}$$

Vezmeme-li projekt A jako etalon (tj. považujeme-li hodnoty $Doba_A$, $Prac_A$ za konstantní) dostaneme vztah

$$Prac_B \doteq c_{40} \cdot Doba_B^{-4}. \tag{15.31}$$

Za podmínek, ze kterých byla získána data v řádku 3 tabulky 15.4 (napjaté termíny realizace), dostaneme, že pro $Doba_B = 1.2 \cdot Doba_A$ spotřebuje projekt A dvaapůlkrát více práce než projekt B, neboť

$$Prac_A \doteq Prac_B \cdot (6/5)^4 \doteq 2 \cdot Prac_B. \tag{15.32}$$

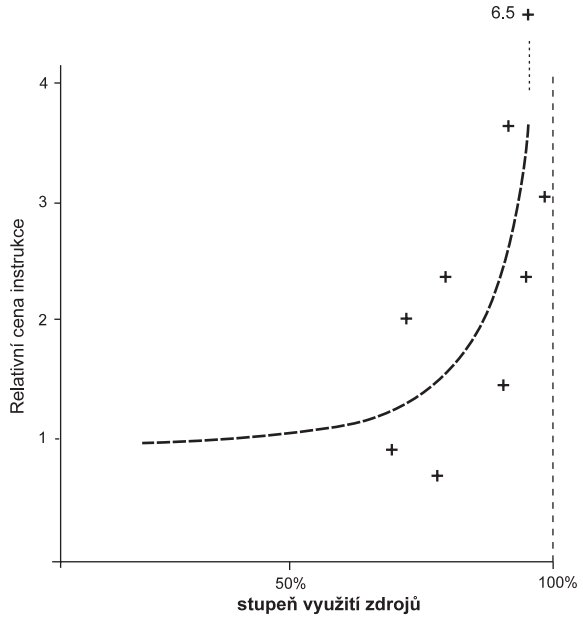
Zkrácení doby řešení na polovinu nebude tedy asi možné. Tento fakt lze ověřit na obr. 15.6, kde je jako nedosažitelná oblast vyznačena oblast roviny se souřadnicemi (*Doba*, *Prac*) vyhovujícími vztahu.

$$Doba_{měsíce} < 3/4 \cdot (Prac_{člověkoměsíce})^{1/3} \tag{15.33}$$

Pro data z obr. 15.6 platí

$$Doba_{měsíce} \mathrel{\widehat{=}} 2.5 \cdot (Prac_{člověkoměsíce})^{1/3}$$

Výše uvedené vztahy byly odvozeny na základě dat organizací pracujících metodou najímaného týmu, kdy se při zjištění potřeby tým okamžitě rozšíří, a napjatých termínů. Zkracování napjatých termínů zvyšuje neúměrně náklady a ohrožuje projekt, neboť nelze překročit hranice nedosažitelné oblasti. To bylo zahrnuto do metodiky odhadu pracnosti pomocí funkčních bodů (srv. kap. 16).



Obr. 15.7: Vliv hardwarových omezení na cenu instrukce reálných projektů.

15.5.5 Vliv hardwarových omezení

Zkracování napjatých termínů pod jistou mez zvyšuje pracnost a ohrožuje projekt jako celek. Podobné efekty lze pozorovat při snahách maximálně využít např. možností hardwaru. Vliv stupně využití rychlosti počítače nebo paměti ovlivňuje pracnost a dobu vývoje podle tabulky 15.5.

Využití hardwaru	Relativní cena instrukce programu	Doba řešení
50 %	1.00	1.00
60 %	1.08	1.00
70 %	1.21	1.00
80 %	1.47	1.05
90 %	2.50	1.18

Tab. 15.5: Vliv hardwarových omezení na cenu instrukce programů a dobuřešení. Cena instrukce při využití zdrojů na 50 % je normována na hodnotu 1. Podle (Boehm, 1981).

Cena při využití hardwaru na 50 % je v tabulce 15.5 normována hodnotou 1. Data z této tabulky jsou zobrazena v tab. obr. 15.7. Z uvedeného plyne, že se na hardwaru a základním softwaru příliš nevyplatí šetřit. Výjimkou jsou kromě takových exotických případů, jako je kosmonautika, a hromadně vyráběných produktů (televizory) i IS s velmi mnoha pracovními místy s omezenou funkcionalitou. Současný software vyžaduje stále výkonnější hardware. Pokud IS obsluhuje stovky pracovních míst, není rozdíl 30 000–40 000 Kč v ceně hardwaru jednoho pracovního místa zanedbatelný. Navíc hrozí, že modernizace základního softwaru si bude vyžadovat stále nové investice. Nárůst nákladů na hardware a základní software tvoří však vždy jen zlomek nákladů na celý IS. V architektuře klient-server lze mnoho ušetřit balancováním zátěže klientů a serveru. Šetřit na hardwaru se vyplatí u produktů, které budou vybaveny jednoúčelovým programovým vybavením a budou vyráběny v milionových sériích (např. programátor pračky) nebo tam, kde se musí šetřit na váze a příkonu (kosmický výzkum). Je to též účelné u sítí s mnoha pracovními místy. Tento poslední důvod byl inspirací k vytvoření koncepce síťového počítače. Všimněme si ale, že v případě síťového počítače (NC) nebývá hlavní přínos z úspor na ceně NC, ale z úspor spojených se zjednodušením správy systému, neboť se udržuje jen jedna verze softwaru, je menší nebezpečí, že do systému pronikne virus. Ve snaze „ušetřit“ se někdy nakupují takové konfigurace počítačů, které ve skutečnosti vylučují použití moderních metod vývoje softwaru, jako jsou objektová orientace, spolupráce aplikací nebo vývojové nástroje. Tento nešvar je rozšířen především při rozhodování o serverech. Výsledkem je enormní nárůst pracnosti a nedodržení termínů. Systém se pak prakticky nedá udržovat a modifikovat. Softwarové systémy mají obvykle po modernizaci podstatně větší nároky na hardware. Modernizace pak přijde hodně draho. K takovým rozhodnutím dochází i v případě investic, ve kterých tvoří náklady na počítače zlomek procenta celkových nákladů nebo kde se ovládají finanční toky v řádech miliard.

Při návrhu hardwaru je proto třeba uvážit i rychlé zastarávání hardwaru a rychle rostoucí požadavky základního softwaru na hardware. I z tohoto důvodu je žádoucí nakupovat hardware s rezervami nebo alespoň používat takové systémy, které lze v případě potřeby levně rozšířit. To neznamená plýtvání. Stačí jen, když se neuzavřou cesty dalšího růstu. Jako často v životě, nejlepší je zdravý kompromis.

Vybudování infrastruktury IS banky s několika sty pracovních míst je mnohomilionová investice. Pokušení ušetřit pár milionů, i když obrat banky je v miliardách, bývá velmi silné. Pak se šetří na serverech, sítích, komunikačním softwaru a často se dbá hlavně na barevnost obrazovek. Pokud nejsme dostatečně předvídaví, narazí růst IS brzy na meze. V praxi se např. ukazuje, že IS koncipovaný pro 50 pracovních míst nelze obvykle snadno rozšířit na 350 míst. Problémem nebývá pouze rychlost, ohrožena je spolehlivost.

15.6 Faktory ovlivňující produktivitu

V sedmdesátých letech byla provedena u firmy IBM analýza faktorů ovlivňujících produktivitu. Hlavní výsledky jsou následující (viz Brooks, 1995, Boehm, 1981, nebo přehled v Král, Demner, 1991)[1]:

1. Obtíže formuluje-li požadavky sám uživatel (3.5).
2. Zkušenost programátorů a jejich kvalifikace (3.1).
3. Podíl implementátorů, kteří se zúčastnili analýzy, $> 1/2$, $< 1/4$, (2.55).
4. Znalost vývojových nástrojů (3.15).
5. Zkušenost s projektem dané nebo větší složitosti (2.80).
6. Složitost uživatelského rozhraní (4.0)
7. Rozsah projektu (2.1).

Více než padesátiprocentní vliv na produktivitu mají následující faktory: Zkušenosti dodavatele v oblasti aplikace (1.5); změny za pochodu (1.50, to je velmi málo, zřejmě díky přísným metodikám firmy IBM, odkud data pochází); poměr průměrné velikosti týmu k době řešení projektu – lidé/měsíce, < 0.9, > 0.9, (1.76); podíl materiálů procházejících inspekcemi, $< 1/3$, $> 2/3$ (1.54). Vliv mají i takové faktory, jako je počet atributů v databázi nebo počet stránek dokumentace na 1 000 řádek programů a samozřejmě ostrost termínů a jiná omezení.

Uvedená studie se týká situace v sedmdesátých létech. Zkušenosti z posledních let naznačují, že výše uvedená zjištění zůstávají v platnosti i v dnešní době. Výjimkou je realizace uživatelského rozhraní, kde prototypování, vizuální a objektově orientované programování spolu s hlubším chápáním problému interakce člověk – počítač částečně snížilo vliv složitosti uživatelského rozhraní. Vliv uživatelského rozhraní však je stále velmi významný. Pozoruhodný je vliv účasti programátorů na analýze a vliv znalosti vývojového prostředí. Vliv zkušenosti a kvalifikace pracovníků jen potvrzuje, jak nebezpečné je zanedbávání péče o profesní růst.

15.6.1 Vliv programovacího jazyka

Spolehlivé empirické údaje o vlivu programovacího jazyka nejsou k dispozici. Ze zkušeností však lze učinit závěr, že není příliš významné, zda programujeme např. v COBOLu nebo Pascalu. Podstatný rozdíl je mezi assemblerem a procedurálními jazyky třetí generace a mezi programovacími jazyky procedurálními a těmi, které jsou objektově orientované a jsou podporovány efektivním vývojovým prostředím.

Zatím však nejsou k dispozici spolehlivá data, která by přínos objektové orientace spolehlivěji kvantifikovala.

Z hlediska pracnosti lze rozdělit programovací jazyky do následujících skupin:

1. Assemblery.
2. Procedurální jazyky třetí generace (C, Pascal, COBOL, do jisté míry i 4GL jazyky a Basic).
3. Objektově orientované jazyky (C++, Smalltalk, Eiffel, Java), modernizované verze jazyků COBOL, Ada a do jisté míry i 4GL jazyky podporované moderními vývojovými prostředími.

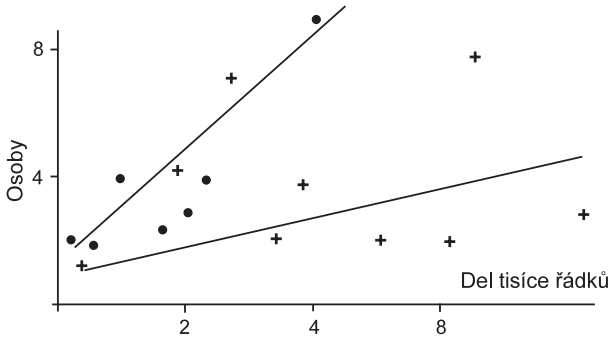
Pracnost lze podstatně snížit využitím vizuálních metod programování (Visual Basic, Power Builder, Visual C++ atd.) a vývojových prostředí. I zde nejsou zatím k dispozici spolehlivá data o přínosech a také o mezích použitelnosti metod vizuálního návrhu a programování.

V assembleru se programuje 5 až 10krát obtížněji než v procedurálních jazycích (obr. 15.9), při použití vizuálních metod programování je rozdíl ještě větší. To platí i pro údržbu. Rozdíly mezi procedurálními jazyky nejsou příliš výrazné. Objektově orientované jazyky mají výhodu ve snazším oživování systému, především však

1. Čísla v závorkách udávají poměr maximální a minimální produktivity pro různé hodnoty daného atributu.

snižují pracnost tím, že mnohé třídy lze použít s malými nebo žádnými úpravami ve více projektech. Objektově orientované systémy se snadněji udržují.

Hlavní rozdíly ve vlastnostech programovacího jazyka se projeví především při údržbě a při práci na více projektech. Programy v assembleru jsou prakticky nepřenosné, obtížně znovu použitelné a obtížně modifikovatelné. Assembler by měl tedy být používán výjimečně. Struktura programů v assembleru by měla být konceptuálně objektová – založená na třídách a objektech simulovaných vhodnými konstrukcemi v assembleru.



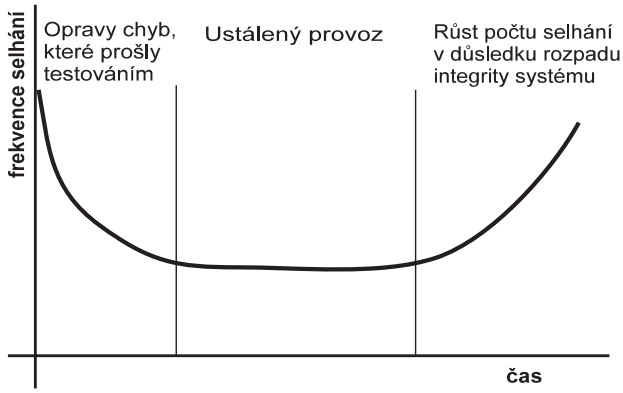
Obr. 15.8: Porovnání pracnosti údržby programů psaných v assembleru (+) a vyšším jazyce (•). *Del* v řádcích. Data ukazují, že $Osoby \mathrel{\widehat{=}} c \cdot Del$ a tedy $Prac_{údržba} \mathrel{\widehat{=}} c_1 \cdot Del$.

15.6.2 Výskyt defektů

Počet defektů v programové jednotce či dokumentu vzrůstá rychleji než délka. Velmi krátké programové jednotky a krátké dokumenty bývají většinou bez chyb. Delší programy či dokumenty jen obtížně přehlédneme, a proto od jisté délky počet chyb rychle narůstá, Tato hypotéza byla využita v (Halstead, 1977) k odhadu délky programu, od které je výskyt defektů podstatně pravděpodobnější. Nalezená mez je asi 300 lexikálních atomů. Odtud plyne, že by neměl mít modul psaný v assembleru více než 150 řádek. Tato zásada je u některých firem součástí norem psaní programů. Takové omezení na délku modulů je výhodné i pro provádění inspekcí. Ucelené části dokumentace by neměly mít větší rozsah než několik stránek. To je rozsah, který může být zvládnut při jedné inspekci. Slabé místo tohoto doporučení je v tom, že v některých, podle zkušeností nepříliš častých případech může být obtížné takové podmínce vyhovět. Sledování počtů defektů odhalených během testování umožňuje detekci slabých míst ve vyvíjeném softwaru. Taková místa se prozradí větším výskytem chyb.

Každý větší program či dokument obsahuje defekt. Vzniká otázka, zda se během údržby počet selhání systému zmenšuje. Ukazuje se, že počet selhání při údržbě nejprve klesá, později však opět narůstá. Vytváří tak typickou „vanovou“ křivku známou z teorie spolehlivosti. (obr. 15.10). Spolehlivost softwaru se tedy řídí stejnými zákonitostmi jako spolehlivost jiných složitých výrobků.

Při zjišťování zdrojů defektů při předávání bylo zjištěno, že asi 35 % chyb bylo způsobeno implementací, téměř dvě třetiny chyb tedy vznikly již ve stádiu formulace cílů, specifikace požadavků a návrhu. Podíl defektů v produktu předávaném do užívání je obdobný. Jiná situace je s náklady na odstranění chyb zjištěných během provozu. Zatímco náklady na odstranění chyb vzniklých chybným kódováním stojí při údržbě asi 1 % všech nákladů na odstranění defektů, chyby návrhu stojí 13 % a chyby v definici požadavků 82 % (4 % připadají na jiné



Obr. 15.9: Závislost počtu selhání při provozu softwaru na době provozu.

zdroje, viz Martin, McClure, 1984, srv. též kap. 1). Méně než polovina chyb se odhalí ve stádiu testování. Zbytek se odhalí při přejímacích testech a hlavně při provozu. Avšak náklady na odstranění defektů při provozu jsou třikrát větší než náklady na odstranění chyb ve všech předchozích etapách. Jsou-li náklady na nápravu chyb při specifikacích 1, jsou při testování 8–7, při provozu více než 50. Cena odstranění chyby je dána empirickou zákonitostí

$$Cena \mathrel{\widehat{=}} (Q)^i \cdot c_0 \tag{15.34}$$

c_0 je cena odstranění chyby v téže etapě, kdy vznikla. Q je počet etap vývoje softwaru (specifikace, návrh, kódování, testování, údržba), v nichž chyba existovala, avšak nebyla v nich odstraněna. Q bývá 3 až 5, tj. každá etapa cenu alespoň ztrojnásobí. Pro masivně prodávané produkty je cena chyby při údržbě ještě vyšší, než odpovídá vztahu 15.34. U mnohonásobně prodávaných produktů jsou ovšem náklady údržby na jednu instalaci ve srovnání s náklady spojenými s instalací u daného uživatele relativně malé, neboť se rozdělují na mnoho zákazníků.

15.6.3 Pracnost realizace jednotlivých etap životního cyklu. Problémy údržby

Analýza projektů řady organizací vedla k odhadům pracnosti jednotlivých etap vývoje nového nebo customizaci nakupovaného IS. Výsledky jsou uvedeny v tabulce 15.6, viz též obr. 15.9. Pracnost vývoje je ohodnocena číslem 100. Pracnost všech činností je vyjádřena v procentech pracnosti vývoje.

Rozsah prací při údržbě tvoří každoročně 10 až 25 % prací na vývoji softwarového díla. Odtud např. plyne, že vývoj trvající více než 7 až 10 let vlastně nikdy neskončí. Procentuální podíl jednotlivých etap realizace se u jednotlivých projektů značně liší. Definice požadavků je však u předních softwarových firem pracná záležitost. Poněvadž se etapy specifikace požadavků účastní obvykle poměrně malý tým a rozsah prací je značný, musí etapa definice požadavků zabírat značnou část doby řešení (20–50 %). Celková pracnost se při customizaci sníží, hlavně díky údržbě, alespoň třikrát při menším riziku neúspěchu. Doba řešení se však sníží pouze o 30–50 % díky tomu, že pracnost a časová náročnost specifikace požadavků zůstává vysoká. Podle zkušeností českých firem jsou náklady na customizované systémy z padesáti procent tvořeny náklady na poradenství, specifikaci požadavků a organizační změny. Náklady na HW a základní SW tvoří asi čtvrtinu nákladů. Nákup licencí customizace a oživení systému spotřebuje rovněž jen asi čtvrtinu nákladů. Výhoda customizovatelného IS je především v údržbě

		Vývoj	Customizace
1.	Specifikace cílů a potřeb	3–5	2–4
2.	Práce na definici požadavků	15–25	10–15
3.	Návrh	15–20	10–20
4.	Kódování	15–25	cca 5
4.1	Konfigurování		5–15
5.	Testování a předání	25–45	cca 10
	Vývoj/customizace celkem	100	30–50
6.	Údržba		cca 30
6.1	Opravy chyb	40	
6.2	Přizpůsobení změnám	70	
6.3	Vylepšení funkcí	90	
	Údržba celkem	cca 200	cca 30

Tab. 15.6: Podíly pracností jednotlivých činností vyjádřené v procentech vývoje systému dané kvality. Pracnost customizace se týká dealera.

a menším riziku neúspěchu projektu. Není však v podstatném zrychlení realizace. Náklady na údržbu CIS jsou pro jednu instalaci nižší proto, že se náklady na ni přenášejí na více zákazníků. Absolutně je u výrobce cena údržby customizovatelného softwaru vysoká a mnohonásobně převyšuje náklady na vývoj.

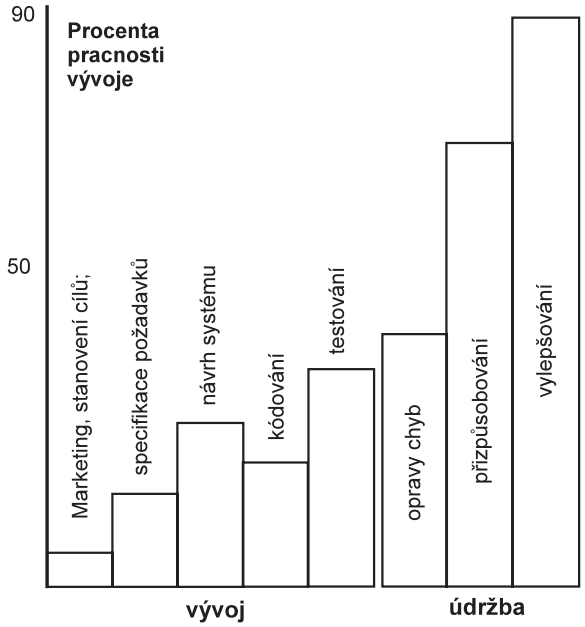
U déle žijících rozsáhlejších systémů tvoří odstraňování závad menší část prací na údržbě – méně než 20 %. Daleko podstatnější část prací představují práce související s přizpůsobeními softwaru vyvolané změnami hardwaru a základního softwaru. Mezi tyto práce patří přenos softwarového produktu na jiný počítač nebo úpravy programu s cílem využít nový typ periferie či novou operaci či proceduru operačního systému. Rozsah prací na údržbě silně závisí na typu softwaru, na době, po kterou je software provozován, a na kolika instalacích je daný softwarový produkt používán.

Během údržby se nejprve odstraňují chyby, které neodhalily testy, tím se snižuje frekvence selhání systému. Pak se upravuje. Úpravami se postupně narušuje logická konzistence systému, takže po jisté době počet chyb v programovém díle opět vzrůstá (srv. práce Lehmana a Beladyho, 1976).

Nutnost údržby rozhodujícím způsobem ovlivňuje požadavky na výstupy etapy vývoje softwaru (dokumentace, způsob psaní programů atd.), poněvadž kvalita výstupů vývoje softwaru silně ovlivňuje rozsah prací na údržbě a nakonec i to, zda bude software udržovatelný, a tedy i dlouhodobě provozovatelný. Tvrdí se, že jeden pracovník je schopen udržovat asi 15 000 výkonných řádků programů. Označme tuto charakteristiku ($K\check{R}/P$) a udávejme ji podobně jako délku *Del* v kilořádcích. Z definice platí pro potřebu $Prac_M$ na údržbu vztah

$$Prac_M = \frac{Del_{vyvinutého\ produktu}}{(K\check{R}/P)} \tag{15.35}$$

Hodnoty metriky $K\check{R}/P$ jsou známy pro zdrojové texty programů a pohybují se od 8 pro systémy reálného času až k 32 pro dávkové zpracování dat (Boehm, 1981). Pro údržbu systému v milionech řádků je nutno na údržbu vyčlenit desítky až stovky pracovníků. Produktivita při údržbě bývá v rozmezí 100 až 200 řádků nového kódu za člověkoměsíc. Zatím chybí spolehlivá data o rozsahu údržby systémů vyvinutých CASE systémy nebo



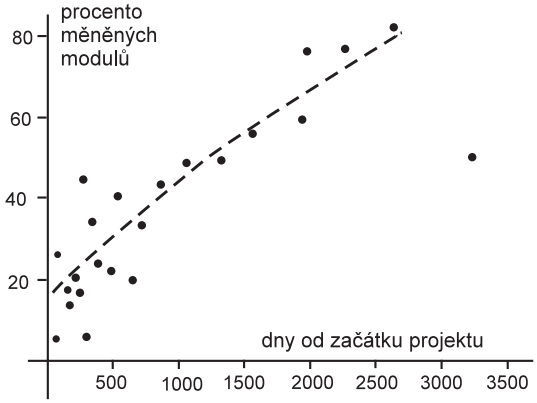
Obr. 15.10: Podíl prací jednotlivých etap životního cyklu.

integrovanými vývojovými prostředími a také pro objektově orientované systémy. Věří se, že takové systémy jsou méně náročné na údržbu.

Při studiu nákladů na údržbu systému je třeba zvážit následující fakta o údržbě velkých dlouho žijících softwarových projektů.

1. *Neustálá změna*. Velké systémy je nutné neustále upravovat, jinak se stávají postupně stále méně užitečné.
2. *Zvyšující se složitost*. Při neustálých změnách se zvyšuje složitost systémů, pokud se neprovádí údržba s cílem složitost zmenšit. Jinými slovy modifikace a doplňování funkcí musí být prováděno nikoliv „nalepováním“, ale i celkovou rekonstrukcí „hotových“ modulů. Údržba vede často k tomu, že se musí zvyšovat počet upravovaných modulů při každé větší úpravě. Příklad takového vývoje je OS pro počítače IBM 360 – viz obr. 15.11 Důsledkem je, že od jisté doby se začne zvyšovat počet selhání systému.
3. *Rozsah údržby* u velkých projektů je statisticky neměnný v čase; až na náhodné kolísání zůstává konstatní.
4. *Udržování znalostí*. Pro spolehlivý plánovitý vývoj musí být měněné programy uvolňovány do užívání tak, aby rozsah změn mezi vydáními nebyl příliš velký, aby byly změny zvládnutelné uživateli.
5. *Základní zákon* vývoje velkých programů. **Dynamika rozvoje velkých systémů je taková, že systém udržuje v podstatě svoje hlavní vlastnosti a kvalitativní charakteristiky stálé**, jsou zachovávány jejich vzájemné vztahy. Jedná se o jistou formu samoregulace systému. To znamená, že pokud má systém nevhodnou architekturu, nedojde během údržby k podstatnému zlepšení jeho vlastností. Proto přežil OS UNIX navržený počátkem sedmdesátých let své „současníky“ a dále se rozvíjí. Architektura UNIXu byla modernější než architektura tehdy vyvíjených komerčních produktů. Architektura UNIXu byla založena na velmi moderních

nástrojích, jako jsou paralelita práce a spolupráce procesů, jazyk C, utility jako SCCS atd., které umožnily přenositelnost a hladký rozvoj systému. Je tedy velmi důležité zvolit moderní, nikoliv pouze módní nástroje a technologie. U „správně navržených“ systémů může tedy být doba „ustáleného provozu“ velmi dlouhá i při poměrně rozsáhlé modernizaci.



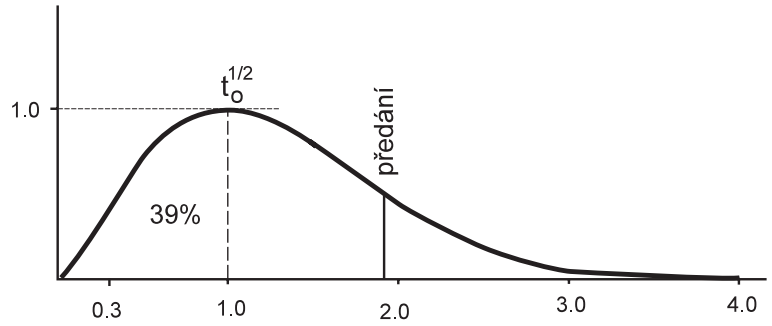
Obr. 15.11: Podíl počtu měněných modulů k počtu všech modulů pro jednotlivá vydání (•) OS IBM 360. Podle (Lehman, Belady, 1976). S časem podíl měněných modulů vzrůstá až na 80 %.

Fakt, že na každých asi 10–20 tisíc řádků softwarového produktu potřebujeme jednoho pracovníka na údržbu, musí být vzat v úvahu, chceme-li např. napodobit nějaký systém. Úpravy jsou nutné a systém se musí vyvíjet. Proto při desítkách programátorů nemůžeme převzít systém o milionech řádků. Nemohli bychom totiž systém vyvíjet. Proto je jediné řešení systém přeprogramovat, aby měl pouze statisíce řádků, byť s omezenou funkčností. Dobré řešení je použít moderní vývojový systém. Tím se rozsah udržovaných řádků de facto sníží. Touto cestou lze dosáhnout dobrých výsledků. Pokud postupujeme cestou pouhého napodobování, můžeme se do čkat nemilých překvapení.

Pro ty, kteří udržují software, je důležité rychle porozumět programům. Poněvadž mají k dispozici fungující systém, je důležité, aby byl k dispozici materiál dávající přehled. Proto je pracovníky údržby softwaru velice oceňován instruktivní popis systému v přirozeném jazyce (Guinares, 1985) a komentáře ve výpisech programů. Podrobný popis detailů implementace bývá potřeba méně. Pro údržbu bývá cenný i deník projektu (kap. 17). Zkušenosti s operačním systémem UNIX naznačují, že pro softwarové architektury založené na spolupráci relativně samostatných komponent je situace příznivější.

15.6.4 Průběh velikosti týmu

Při řešení nějakého úkolu není obvykle počet členů týmu stálý. To je zvláště patrné u organizací pracujících metodou hlavního programátora, u kterých se přidělují spíše programátoři k úkolům než úkoly k programátorům. V této situaci je řešitelský tým zprvu malý, dosahuje jistého maximálního počtu členů a pak se opět zmenšuje. Zmenšování počtu členů týmu je povlovnější než jeho růst. Velikost týmu lze tedy popsat doleva sešikmenou funkcí. Jako model velikosti týmu $team(t)$ v čase t je používána funkce, kterou zavedl Rayleight ve fyzice.



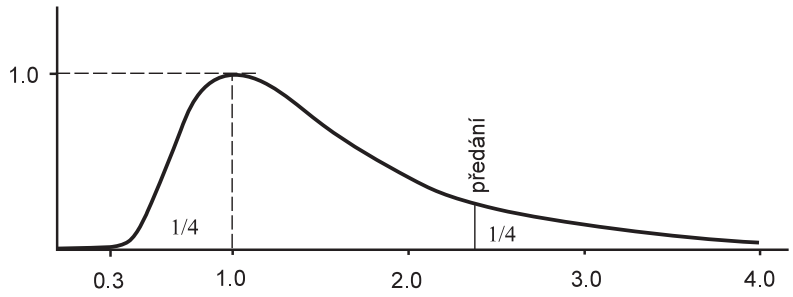
Obr. 15.12: Rayleightova křivka. Část plochy pod křivkou po předání jsou práce, které budou provedeny v rámci údržby (corrective maintenance). To, co se do okamžiku předání nestihne udělat, přechází do údržby, kde se projevuje jako neodstraněné chyby.

$$team(t) \mathrel{\widehat{=}} K \cdot t/t_0 \cdot \exp(-\frac{t^2}{2t_0}) \tag{15.36}$$

Zde jsou t_0 a K vhodné konstanty. $\sqrt{t_0}$ je hodnota času, kdy nabývá $team(t)$ (obr. 15.12) největší hodnoty. Zřejmě

$$\int_0^\infty team(t)\,\mathrm{dt} = K \tag{15.37}$$

tj. K udává celkovou spotřebu práce na projektu, není-li doba řešení omezena. Projekt musí však být předán v konečném čase P.



Obr. 15.13: Normalizovaný tvar Planckovy křivky $Planck(t) = 142.32 \cdot t^{-5}/(\exp(4.9651/t) - 1)$.

Tým má obvykle největší velikost v okamžiku testování částí (unit tests). Lze ověřit, že $\int_0^{\sqrt{t_0}} team(t)\,\mathrm{dt} = 0.39$, čili, že do okamžiku předávání části je podle Rayleightova modelu vykonáno 39 % celkové pracnosti, včetně té, která nakonec bude vykonána v rámci údržby.

Plocha pod křivkou od okamžiku předání představuje spotřebu práce k na nápravu chyb během údržby. Z výše uvedených dat o pracnosti údržby vyplývá, že by mělo být přibližně $k = 0.4 \cdot K$. Vzhledem k tomu, že je

odstraňování chyb prováděno pracovníky údržby a nikoliv vysoce kvalifikovanými vývojáři a tedy méně efektivně, můžeme předpokládat, že přibližně platí $k = 0.3 \cdot K$. Okamžik předání P by tedy měl být blízko $1.6 \cdot \sqrt{t_0}$ ($P < 1.7 \cdot \sqrt{t_0}$). Ve skutečnosti bývá $P > 2 \cdot \sqrt{t_0}$. Rayleightův model rovněž nevysvětluje prudký růst pracnosti při zkracování termínů realizace. Předpoklad pevného procenta spotřeby prací před dosažením maxima je nereálný.

Vztah 15.31 připomíná Wienův zákon pro zářivost absolutně černého tělesa. To je inspirací pro pokus modelovat průběh velikosti týmu modifikací Planckova zákona (Friš, Timorjeva, 1954). Nahraďme vlnovou délku v Planckově zákoně hodnotou $t = d^{-1}T + k$, kde T je čas od zahájení projektu a d a k jsou vhodné parametry. Považujme i ostatní konstanty v Planckově funkci za parametry. Tím dospějeme k modelu

$$teamp = \frac{c \cdot d^5 (T + k \cdot d)^{-5}}{\exp\left(\frac{D \cdot d}{T + k \cdot d}\right) - 1} \tag{15.38}$$

Tato křivka má 4 nezávislé parametry c, d, D, k. Křivka 15.38 má následující vlastnosti:

- Parametr D určuje polohu maxima a do značné míry i tvar křivky.
- Podíl práce do dosažení maxima (obr. 15.13) závisí na D a je blízký hodnotě 25 %. Tento podíl se zmenšuje, klesá-li hodnota D.
- Maximum funkce $teamp(T)$ je tím ostřejší, čím je D menší.
- V době maxima je ukončeno přibližně 25 % prací včetně nápravy chyb v době údržby (tj. 1/3 prací při vývoji). Je-li tedy přenecháno 25 % prací do údržby, bude systém předán asi za dvaapůl až trojnásobek doby dosažení maxima. Pokud je požadována záruka, že do údržby zbývá pouze 10 % prací, to je případ systémů, které mohou ohrozit životy, je předání možné očekávat po uplynutí 4.5 násobku doby, kdy tým měl maximální velikost (Král, 1993).

Prokládání křivky *teamp* pozorovanými daty dává velmi dobré výsledky. *teamp* nemá nepříznivé vlastnosti Rayleightovy křivky. (obr. 15.14, obr. 15.15).

Planckův model průběhu velikosti týmu můžeme dále zobecnit na model s pěti parametry

$$teamp1(T) = \frac{c \cdot (T + k \cdot d)^{-q-1} d^{q-1}}{\exp\left(D \cdot d/(T + k \cdot d)\right) - 1} \tag{15.39}$$

s parametry c, d, D, k, q. Má-li být celková pracnost konečná, musí být $q > 2$.

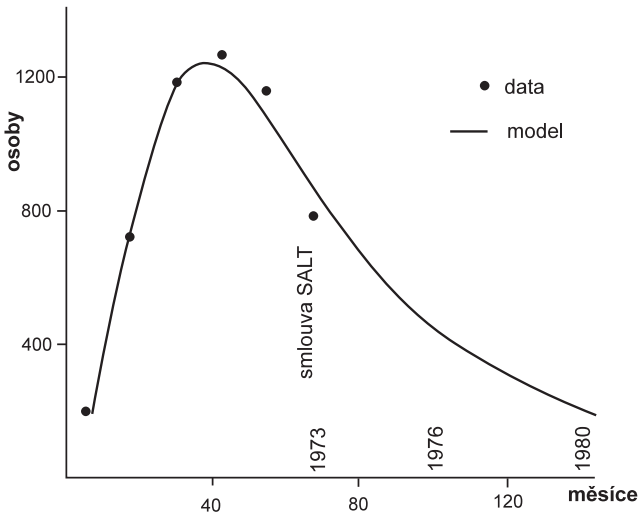
Hlavní závěr: *Špička výkonu týmů nastává mnohem dříve, než je vynaložena polovina práce na vývoj produktu.*

Tento fakt by měl být varováním před přehnaným optimismem. Často se pokles intenzity prací, tj. překročení vrcholu křivky, chybně spojuje s představou, že je „skoro hotovo“. Ve skutečnosti jsme málo za třetinou doby řešení a ve třetině spotřeby prací.

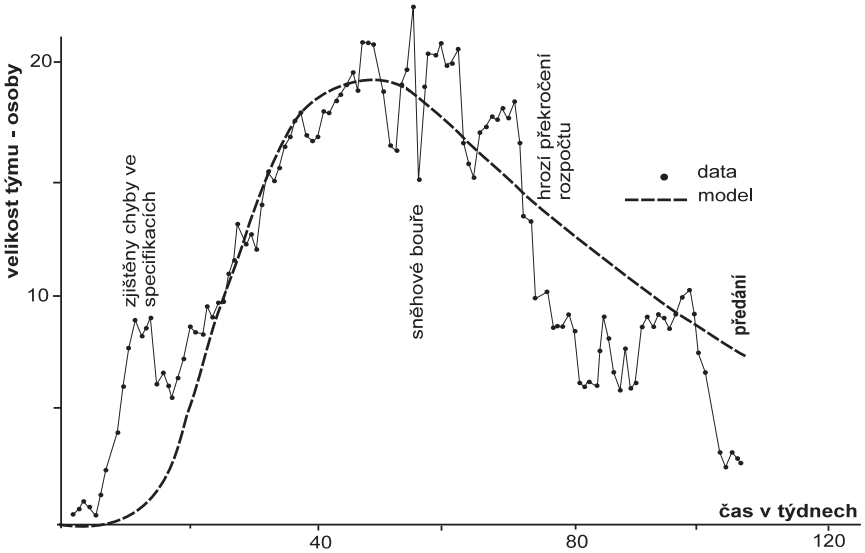
I v týmech pevné velikosti má intenzita práce na projektu podobný průběh jako velikost najímaného týmu. I tam pokles intenzity práce obvykle neznamená, že jsme blízko konce prací.

15.6.5 Využití času

Práce na kódování (psaní programů, nepočítáme-li testování) tvoří jen malou část nákladů na vývoj softwarových děl. Tento fakt je nepřímo potvrzován i studiemi struktury využití pracovního času programátorů. I zde se potvrzuje, že psaní programů je celkem „okrajová“ činnost. Kódování s testováním částí pokryje 1/5 až 1/3 pracovního času programátora – včetně psaní dokumentace. Struktura využití času programátorů – kódérů u velkých firem



Obr. 15.14: Planckův model velikosti týmů pro projekt Safeguard.



Obr. 15.15: Planckův model pro software spojení s ponorkami.

v USA je zachycena v tabulce 15.7. Údaje o podílu prací při vývoji softwaru jsou z Bell Telephone Laboratories z šedesátých let. Údaje o údržbě jsou ze sedmdesátých let. „Vlastní programování“ tedy tvoří menší část prací, dokonce i u specialistů programátorů – kódérů. To platí i dnes v důsledku používání moderních stále se měnících

Vývoj:	
Čtení programů a manuálů	16 %
Domluva o projektu uvnitř týmu	32 %
Psaní programů	13 %
Osobní záležitosti	13 %
Školení	6 %
Cestování	5 %
Testování	15 %
Údržba:	
Studium požadavků na úpravy	18 %
Studium dokumentace	6 %
Studium programů	23 %
Vylepšování dokumentace	6 %
Úpravy programů	19 %
Testování	28 %

Tab. 15.7: Co programátoři dělají.

technologií. Hlavní činností velkých týmů jsou domluva uvnitř týmu, čtení manuálů a dokumentů a teprve pak „vlastní práce na projektu“. Je proto přirozené, že se vytvářejí programové nástroje, umožňující zmenšit časové ztráty vyvolané např. potřebou neustálého čtení programů a dokumentace. To je důvodem úspěšnosti softwaru na podporu vývoje software – softwarových nástrojů - a na podporu práce ve skupině (groupware).

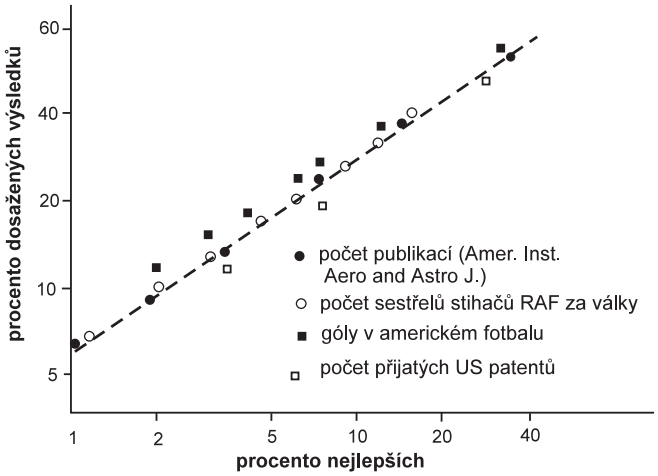
Z výše uvedeného plynou následující závěry:

a) Hlavní efekt pro týmovou práci má zrychlení komunikace a zmenšení potřeby komunikace uvnitř týmu. To je důležité jak pro programátory, tak pro analytiky.
b) Je důležité vytvořit infrastrukturu usnadňující práci v týmu. Snadno dostupná jsou následující opatření:
   - elektronické propojení členů týmů,
   - elektronická textová forma dokumentů, včetně hypertextu,
   - CASE systém, použití vývojových prostředí,
   - systém podpory správy dokumentů, včetně verzí,
   - systémy správy prací a řízení projektů (MS Project, workflow systems atd).

15.6.6 Role špičkových pracovníků

Vliv kvality zúčastněných pracovníků na kvalitu výsledného produktu je patrný ve všech oborech lidské činnosti, všeobecně však vzrůstá s podílem nerutinních (tvůrčích) činností na realizovaném díle. Při vývoji velkých softwarových děl je podíl tvůrčí práce poměrně značný. Zkušenost ukazuje, že volba schopného pracovníka do vedení týmu a vytvoření odpovídajících pracovních podmínek je zcela základní podmínkou. Velký projekt lze jen velmi obtížně realizovat bez kvalitního vedoucího projektu.

Rozdíly mezi programátory – profesionály jsou propastné. Není výjimkou poměr produktivit 1:20 (Weinberg, 1971). I z jiných oblastí lidské činnosti je známo, že podíl výsledků dosažených nejlepšími je značný. Nechť $a(t)$, $0 < t < 100$, značí procento výsledků dosažených t procenty nejlepších, tj. těch, co mají nejvíce výsledků.



Obr. 15.16: Vztah mezi procentem nejlepších a procentem jimi dosažených výsledků (podle Boehm, 1981).

Vztah $a(1) = 7$ značí, že 1 % nejlepších získalo 7 % výsledků. Je známo, především z oblasti vědeckotechnických informací, srv. obr. 15.16, že platí vztah ($1 < t < 20$)

$$a(t) = c \cdot t^{1/2}.$$

Při tom je $a(1) = 7$, $a(10) = 30$, $a(20) = 50$. Čili zhruba 10 % nejlepších udělá třetinu práce, 1 % nejlepších udělá více než 7 % výsledků, 40 % nejhorších neudělá skoro nic (viz obr. 15.16).

Lze se právem domnívat, že u opravdu kvalifikovaných prací špičkové obtížnosti je vliv talentu ještě vyšší a je také vyšší podíl těch, kteří na danou práci prostě nestačí, srv. špičkové vědecké obory. Vzhledem k tomu, že schopných vedoucích je málo, je přirozené, že se snažíme vytvořit podmínky, kdy jsou co nejméně zatěžováni „vedlejší“ prací. To je princip týmu šéfprogramátora (kap. 10).

Vzhledem k tomu, že vedoucí týmu má rozhodující roli v důležitých fázích formulace požadavků a návrhu systému a také při integraci, měl by se vedoucí týmu programátorů zúčastnit všech etap vývoje softwarového díla. Projekt by měl být od začátku do konce řešen pod vedením jediného vedoucího. Tuto zásadu je obtížné dodržovat, dobrých vedoucích je málo. Význam vedoucího není jen v tom, že navrhuje v jistém smyslu nejlepší řešení. Velmi důležitý je aspekt psychologický. Kvalitní odborník má obvykle přirozenou autoritu a je tedy v týmu respektován. Z toho důvodu není nutné a často ani vhodné, aby byl příliš zatěžován administrativními problémy a administrativním vedením.

Již jsme uvedli, jak velký je rozdíl ve výsledcích mezi programátory. Ještě významnější je vliv kvality vedoucích členů týmu na kvalitu specifikace požadavků. Tam je poměr kvality mezi nejlepšími a nejslabšími podstatně větší než 1 : 20. Využívání špičkových pracovníků při rutinních úlohách však není bez rizik, o kterých jsme se zmínili výše (kap. 10). Zopakujme, v čem je problém.

1. Pokud řešení závisí na pracovníkovi, který nahrazuje celý tým, je téměř jisté, že jeho odchod vážně ohrozí celý projekt a může ohrozit i firmu. Management proto často dává u rutinních úloh přednost technice „parního válce“ – raději využívá více průměrných pracovníků než jednoho špičkového.
2. Špičkoví pracovníci nejsou často dostatečně přizpůsobiví pro týmovou práci. Mají tendenci opomíjet dokumentaci a neradi dodržují podnikové normy. Výsledky jejich práce se proto obtížně udržují.
3. Špičkoví pracovníci mají častěji, než je managementu milé, tendenci přijímat nečekaná řešení a pouštět se na neprověřené cesty.

Využití špičkových pracovníků patří mezi nejobtížnější úkoly managementu. Bez nich však nemůže žádný podnik dlouhodobě prospívat. Nalezení kompromisu mezi strategií parního válce a využitím géniů je velmi obtížný manažerský problém. Některé firmy (Microsoft) používají techniku „být druhý, ale nemít přílišný odstup od prvého“. Sledují dobrá cizí řešení a ta pak rychle použijí. Ani to není bez rizik. Odstup může být přes všechnu snahu příliš velký. To byl do jisté míry případ vztahu Microsoftu a řešení, se kterým přišla firma Netscape. Faktory ovlivňující pracnost údržby jsou též studovány v kapitole 13.

15.7 Softwarové metriky a zajišťování kvality softwaru

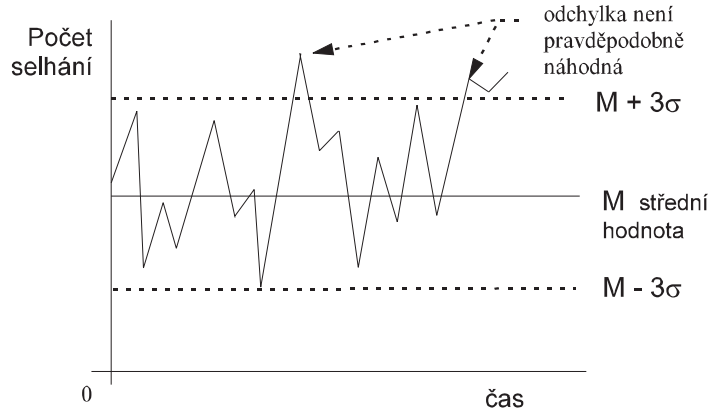
Kontrola kvality vyžaduje sledování dynamiky změn hodnot metrik mající vztah ke kvalitě. Jsou to odchylky od správné činnosti, střední doba mezi selháními, počet oprav v dokumentech a programech, průměrná doba do odstranění chyby (viz např. Kan, 1995). Základní metriky jsou uvedeny v paragrafech 15.2 a 15.3.

Pro kontrolu kvality je výhodné vytvořit informační systém, který by umožňoval evidenci a analýzu metrik. Informační systémy by měly umožňovat dotazy ad hoc a zobrazování trendů. Vhodným prostředkem zviditelňování metrik je např. tabulkový kalkulátor. Již s daty uvedenými v 15.3 lze vyhodnocovat trendy počtu selhání, doby do odstranění závad, zjišťování etap vzniku chyb atd. Struktura příslušného informačního systému pro analýzu metrik může mít strukturu z obr. 15.1.

Při zobrazování trendů se využívají grafické prostředky. Uveďme jednoduché případy použití:

a) Histogramy
   - počet a spokojenost zákazníků podle odpovědí na dotazníky,
   - počty chyb podle období,
   - procenta chyb podle závažnosti,
   - ... atd.
b) Čárové grafy se zobrazením cílového stavu. Čárové grafy pro déle existující systémy je výhodné zobrazovat ve formě obr. 15.17, kde se zobrazují
   - skutečné hodnoty,
   - průměr M,
   - hranice kontingenčního pásma, tj. $\pm 3 \cdot \sigma$, σ je směrodatná odchylka.

M a σ se zjistí z historických dat následujícím způsobem. Zvolí se dostatečně dlouhý interval pozorování s pozorovanými hodnotami $x_1, x_2, \ldots, x_n$. Pak je $M = 1/n \sum_{i=1}^{n} x_i$ a $\sigma = \sqrt{1/(n-1) \sum_{i=1}^{n} (x_i - M)^2}$. Překročení hranice kontingenčního pásma znamená podstatnou statistickou odchylku, která by měla být analyzována a její příčina odstraněna. Tato technika se dá použít pro detekci stavu, kdy je třeba systém přeprogramovat (obr. 15.9). Tento případ nastává tehdy, vybočují-li pozorované hodnoty z významně kontingenčního pásma. Přesné vyhodnocování je možné jen pomocí metod matematické statistiky.

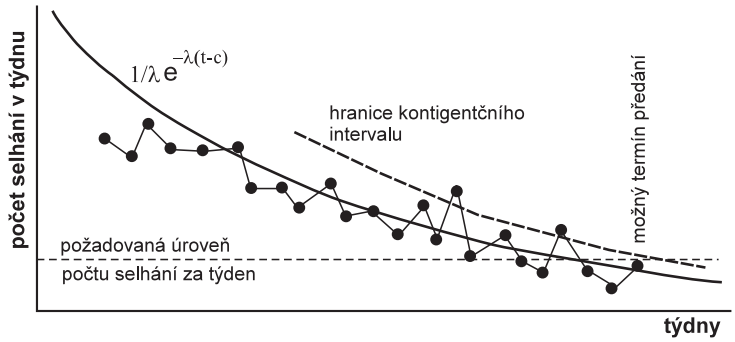


Obr. 15.17: Sledování stability pozorovaných hodnot.

Nejčastější metoda kontroly prací při testování je sledování počtu selhání za jednotku času. Při testování a odstraňování chyb klesá frekvence selhání zhruba podle zákona $1/\lambda e^{-\lambda(t-d)}$, kde λ a d jsou vhodné konstanty a t je čas. To lze použít následujícím způsobem:

a) Zjišťují se postupně dvojice hodnot *(čas, počet selhání)*, kde *čas* udává týden od začátku testování a *počet selhání* udává počet selhání v daném týdnu.
b) Zjištěnými body se metodou nejlepší shody, lépe však s uplatněním statistických metod, prokládá křivka pro neznámé parametry c, λ a d. Z průběhu funkce $c \cdot e^{-\lambda(t-d)}$ lze odhadnout, kdy systém dosáhne žádoucí kvality (obr. 15.17).

Graf z obr. 15.18. lze použít i pro včasnou detekci problémů při testování systému. Problémy se projeví podstatnými odchylkami od exponenciální křivky.



Obr. 15.18: Typický průběh frekvence selhání systému s proloženou exponenciální funkcí.

15.8 Role metrik v činnosti softwarových firem

Sběr a vyhodnocování metrik bývá zvláště u menších firem silně podceňován. Sběr metrik se často právem považuje za zbytečnou administrativní zátěž. Existuje obava ze zneužití a přijetí chybných závěrů. Tyto obavy jsou oprávněné při neopatrném vyhodnocování metrik, které nejsou založeny na dostatečně spolehlivých metodách zjišťování, sběru a hodnocení dat. Není výjimkou, že se metriky sbírají, ale neanalyzují se, nebo se při analýze nepoužívají moderní, např. statistické, nástroje. Z dlouhodobého hlediska se firmy nepracující s metrikami

Třída softwarového systému	Pracnost 1 řádku	Délka	Pracnost celku
Jednoduché dávkové úlohy	1–2	1–2	1–4
Interaktivní databázové systémy (běžné IS)	4–6	2–4	8–24
Databáze s prvky reálného času	5–8	2–8	10–64
Rozsáhlé projekty, selhání může ohrozit životy	> 10	> 10	> 100

Tab. 15.8: Obvyklé hodnoty metrik pracnosti instrukce, délka a celková pracnost typických řídsoftwarových projektů. Vše jako poměr k hodnotám v prvním řádku.

vystavují značnému riziku. Bez metrik se ztrácí podstatná část zkušeností z dokončených projektů. Bez metrik nelze včas detekovat vznik anomálních situací a z nich plynoucí problémy. Metriky umožňují sledovat důležité parametry při řešení projektu a detekovat příčiny problémů. V neposlední řadě umožňují sledovat vlivy použití moderních technik vedení projektu a použití vývojových prostředků. Sběr a vyhodnocování metrik vyžaduje norma kontroly kvality softwaru ISO 9000–3, ISO 9126 a řady připravovaných norem. Sběr metrik bude důležitým kritériem získání certifikátu kvality softwaru. Při vyhodnocování metrik bývá nutné používat statistické metody. To naráží na malou znalost matematické statistiky v softwarových firmách.

Pokud nejsou metriky používány, je výhodné postupně budovat jejich sběr a vyhodnocování, počínaje údaji, jejichž sběr představuje nejmenší zátěž. Mezi tyto údaje patří

- hlášení problémů od zákazníků,
- záznamy o provedení testů,
- údaje o počtech změn v souborech,
- údaje z kontrolních dnů a oponentur,
- ekonomické informace ze smluv.

Sběr metrik může mít sám o sobě pozitivní efekt, i když se metriky přímo nevyužívají pro řízení projektu. Je-li obecně známo, že vysoká hodnota metriky *McCabe* indikuje problémy, programátoři obvykle sami modifikují algoritmy tak, aby tato metrika neměla vysokou hodnotu. Výsledkem jsou kvalitnější programy.

15.9 Třídy softwarových systémů

Při plánování projektů lišících se od projektů dosud realizovaných je důležité sledovat parametry, které podstatně ovlivňují pracnost vývoje/customizace. Je nutné zvažovat, zda projekt nepřekračuje hranice třídy složitosti, do níž patřily dosud řešené projekty (tabulka 15.8). Důvodem maximální opatrnosti musí být podstatný vzrůst rozsahu projektu, počtu uživatelů a rozsahu dat. Za podstatný vzrůst se považuje pětinásobný růst těchto metrik. Takový nárůst, případně redukce jsou vyvolány přítomností či nepřítomností následujících faktorů:

– stonásobné zvýšení rozsahu dat,
– prvky tvrdého reálného času,
– chyba softwaru může ohrozit životy (mission critical systems),
– změna velikosti systému více než pětkrát.

Hrubý odhad změny pracnosti lze získat z tabulky 15.8 obsahující typické třídy softwarových systémů. Jednoduchými úlohami v dávce jsou míněny systémy pracující bez dialogu s uživateli. Tyto systémy je vždy možné koncipovat tak, že se výpočet dá vždy zopakovat. Interaktivní datově intenzivní systémy jsou systémy, které dnes pracují nad databázemi. Typickým příkladem jsou IS. Pro tyto systémy je nutné zajišťovat integritu transakcemi.

16

Metody odhadu pracnosti a doby řešení

Odhad parametrů na realizaci a termínů realizace softwarového produktu je obtížný z následujících důvodů:

1. Silná závislost všech parametrů výsledného produktu, jako jsou náklady, kvalita řešení atd., na kvalitě řešitelského týmu.
2. Rychle se měnící podmínky (vlastnosti hardwaru, měnící se způsoby používání počítačů, např. v poslední době přechod na interaktivní a distribuované způsoby práce atd.) silně snižují opakovatelnost nebo podobnost řešení. Jedná se tedy o stanovení pracnosti úkolu, který je do značné míry unikátní.
3. V programování se dosud neustálily pracovní postupy.
4. Vývoj je natolik rychlý, že ke změnám podmínek řešení (know-how, použitý hardware) může dojít i během řešení jediného úkolu.

Lze tedy očekávat, že každá metoda odhadu bude zatížena značnými chybami. Přesto je nějaký, byť hrubý odhad potřebný a nutný. Za této situace je nutné se do značné míry spoléhat na zkušenost a intuici odhadce. Musíme se však spolehnout pouze na ni? Odpověď je nikoliv. Je totiž známo, že čistě subjektivní odhady nákladů na realizaci softwarového díla bývají příliš optimistické. Lidová tvořivost tuto skutečnost zná jako tzv. Hofstadtlerův zákon, který s trochou nadsázky tvrdí: „V softwaru vše stojí a trvá déle, a to i tehdy, když provedeme korekci původního odhadu“[1]. Důvody pro tuto situaci jsou v lidské psychologii.

1. Lidé obecně mají tendenci být příliš optimističtí a očekávají, že se záležitosti budou vyvíjet spíše příznivě. Mezi odhadem doby realizace t a skutečnou dobou realizace $a(t)$ platí vztah (Boehm, 1981) $a(t) \mathrel{\widehat{=}} 4/3 \cdot t$.
2. Dosavadní zkušenosti se berou v úvahu jen z části. Jestliže se projektuje nějaký systém, odhaduje se jeho rozsah analogií s podobným systémem. Přitom se mlčky soustřeďujeme na rozsah programů realizujících vlastní, „hlavní“, úkoly systému. Tyto části často vyžadují pouze menší část prací na systému. Většinu prací pohltí takové úkoly jako reakce na chybu, přesuny dat, konverze dat, generace návodů (HELP), kontrola vstupních a výstupních dat atd. Vlastní výpočty, „užitečné funkce“, zajišťuje někdy jen několik procent programů (Boehm, 1981, Král, Demner, 1991). Například v protiraketovém systému SAFEGUARD, který se nedokončil po uzavření smlouvy SALT, měly programy řízení protibalistických střel 789 tisíc řádků, programy pro diagnostiku a údržbu více než 100 tisíc řádků, různé pomocné programy pro simulaci a výcvik 840 tisíc řádků a různé vývojové nástroje 532 tisíc řádků.

1. Hofstadtlerův zákon je projevem tzv. softwarového folklóru.

3. Lidé nejsou obvykle plně seznámeni s celým rozsahem úkolu. Proto se často zapomíná na podpůrný software, provozní problémy, dokumentaci a školení personálu.
4. Nedostatečně se počítá se vznikem neočekávaných potíží. Doufá se, že se staré chyby nebudou opakovat, zapomíná se ale, že se mohou objevit chyby nové. Je tedy nejvýše žádoucí odhad nákladů a termínů založit na nějakých empirických zákonitostech, např. těch, které jsme studovali v kapitole 15. Pak bude odhad vycházet alespoň zčásti z objektivních údajů, o něž se bude moci opřít intuice kvalifikovaného odhadce.

V případech, kdy organizace získala dostatek zkušeností s realizací příbuzných systémů, může být odhad nejrůznějších metrik softwaru, jako je pracnost, rozsah dokumentace atd., dostatečně přesný. V každém případě musí být odhad svěřen kvalifikovanému odhadci. V dalším se omezíme na tři nejrozšířenější metody odhadu.

16.1 Odhad COCOMO

Odhady COCOMO vycházejí z odhadu délky programu v tisících nově napsaných řádků. Metodu COCOMO uvádíme jako příklad definice kalibračních konstant v odhadech. Navíc lze z postupu odhadu vysledovat míru vlivu některých faktorů, které lze ovlivnit organizací práce, a tím nalézt cesty ke snížení pracnosti vývoje. Prvý krok odhadu vychází z typu softwarového díla. Uvažují se tři typy projektů.

1. *Organický typ.*
   Tento typ zahrnuje spíše malé problémy, realizované malými týmy. Pro tento typ projektů jsou typické mírné normy a malá omezení na specifikaci rozhraní a možnost ovlivnit požadavky. Řešitelský tým si může např. vyžádat změnu zadání při obtížné realizaci původního zadání. Algoritmy a postupy řešení jsou dobře známy. Podmínky realizace jsou relativně stabilní. Nejsou ostré podmínky na termíny. Projekt není příliš velký, pokud nepřesahuje rozsah několika málo set tisíc řádků zdrojových programů. Požadavky na budoucí inovace a přenositelnost nejsou podstatnou podmínkou.
   Příklady: zpracování dat v dávkovém provozu, úpravy známého operačního systému nebo kompilátoru, běžný vědecko-technický software.
2. *Přechodný typ.*
   Typ mezi organickým a vázaným typem. Velikost projektu do 400 tisíc řádků pro případ, že kód není generován automaticky. Typické charakteristiky týmu a jeho úkolů:
   - v týmu jsou zkušení i nezkušení pracovníci,
   - tým má nepříliš velké zkušenosti z obdobných realizací,
   - řešená úloha je poměrně složitá.

   Přechodným typem softwaru bývají menší dedikované operační systémy, běžné transakční systémy, systémy řízení výrob, jednodušší systémy řízení vojsk a zbraní.
3. *Vázaný (embeded) typ.*
   Software musí pracovat za velmi ostrých omezení na výkonnost a dobu odezvy a je velmi rozsáhlý, až miliony řádků programů. Vývoj vyžaduje práci s komplikovanými softwarovými a hardwarovými systémy za ostrých předpokladů na předepsané funkce, spolehlivost, termíny, přenositelnost a modifikovatelnost. Požadavky lze jen obtížně měnit, stejně jako vazby na jiné systémy. Je obvyklé, že se řeší zcela nové problémy, se kterými se pracovníci dosud nesetkali. Obvyklý postup vývoje projektů vázaného typu: Specifikace požadavků a návrh systému je prováděn relativně malou skupinou, vývoj částí je prováděn velkým týmem, který programuje a testuje části souběžně. Jiný postup realizace zvyšuje dobu řešení. Pak bývá nutné provést více změn během

řešení, protože systém zastaral, spolupracující systémy byly realizovány dříve, a proto bývá nutné akceptovat to, jak pracují. Typické produkty tohoto typu jsou: nové rozsáhlé operační systémy, řízení kosmických lodí a letadel, složité zbraňové systémy, řízení atomových elektráren. Podprojekty velkého projektu mohou být různého typu. Odhady pro podprojekty se provádějí separátně.

Pro jednotlivé typy softwaru dostaneme následující odhady spotřeby práce *Prac* v člověkoměsících a doby vývoje *D* v měsících z délky *Del* programů v tisících řádků.

Organický typ

$$Prac \mathrel{\widehat{=}} 2.4 \cdot K \cdot Del^{1.05},\ D \mathrel{\widehat{=}} 2.5 \cdot Prac^{0.38} \tag{16.1}$$

Přechodný typ

$$Prac \mathrel{\widehat{=}} 3.0 \cdot K \cdot Del^{1.12},\ D \mathrel{\widehat{=}} 2.5 \cdot Prac^{0.35} \tag{16.2}$$

Vázaný typ

$$Prac \mathrel{\widehat{=}} 3.6 \cdot K \cdot Del^{1.20},\ D \mathrel{\widehat{=}} 2.5 \cdot Prac^{0.32} \tag{16.3}$$

K je pro jednodušší variantu metody COCOMO součin parametrů získaných z číselného hodnocení následujících atributů produktu:

Atributy produktu:

RELY – míra požadavků na spolehlivost.
DATA – míra rozsahu datové základny.
CPLX – složitost produktu.

Atributy počítače:

TIME – míra požadavků na dobu odezvy.
STOR – míra využitelnosti paměti.
VIRT – míra proměnlivosti OS počítače.
TURN – míra rychlosti oběhu úlohy počítačem.

Atributy týmu:

ACAP – míra schopnosti analytiků.
AEXP – míra zkušeností programátorů s podobnými aplikacemi.
PCAP – míra kvality programátorů.
VEXP – míra zkušenosti s počítačem.
LEXP – míra zkušenosti s programovacím jazykem.

Atributy metod vedení projektu:

MODP – míra použití moderních metod vývoje softwaru.
TOOL – míra použití moderních prostředků vývoje softwaru.
SCED – „ostrost“ požadavků na dobu realizace.

Postup stanovení odhadu:

1. Určí se typ softwaru.
2. Určí se hodnocení vlivu faktorů reprezentovaných jednotlivými atributy. Metoda hodnotí vliv jednotlivých faktorů stupnicí: velmi málo, málo, standard, mnoho, velmi mnoho, extrémně mnoho. Metoda obsahuje poměrně přesná pravidla, jak tato hodnocení provádět.
3. Určí se číselné hodnocení atributů z tabulek. Každý atribut má vlastní tabulku.
4. Určí se hodnota K jako součin takto získaných hodnot

5. V závislosti na typu softwaru se vypočtou hodnoty odhadu podle 16.1, nebo 16.2, nebo 16.3.

Hodnota K kolísá v poměru 1:50. Tak podle (Boehm, 1981) hodnocení atributu RELY je prováděno podle následujících kritérií: nepohodlí, malé ztráty, nahraditelné ztráty, velké ztráty, ohrožení lidí. Hodnocení extrémně mnoho se pro RELY neuvažuje. Pro tato hodnocení nabývá RELY po řadě hodnot 0.75, 0.88, 1.0, 1.15, 1.40. Z toho, co jsme uvedli výše, plyne, že je asi vliv ohrožení životů podceněn. Proto se COCOMO metoda stále rozvíjí a používá stále složitější metody volby kalibračních konstant. To je předmětem výzkumu pracovních skupin pro rozvoj metody COCOMO. Nevýhodou metody je vazba na metriku *Del*, která je známa poměrně pozdě a proto se musí odhadovat. Podrobnosti viz (Boehm, 1981) nebo (Král, Demner, 1991).

Odhad COCOMO se celkem úspěšně používá, vyžaduje však kvalifikovaného odhadce. Volba hodnot atributů je do značné míry subjektivní. Samotný výčet atributů je pro vedení projektu zajímavý. Ukazuje, jaké faktory mohou podstatně ovlivnit náklady a termíny řešení. Hlavní nevýhodou odhadu COCOMO je závislost na metrice *Del*. *Del* lze sice někdy poměrně přesně odhadnout, jsou však případy, jako je tomu u nových projektů, kdy je odhad obtížný. Metoda podceňuje vliv některých okolností, např. požadavek reálného času.

16.2 Odhad pomocí funkčních bodů

Důležitým krokem k objektivizaci odhadu termínu a pracnosti představují tzv. funkční body (Albrecht, Gaffney, 1983). Funkční body tvoří základ odhadu náročnosti realizace založené na Halsteadově hypotéze, že náročnost realizace je úměrná meznímu objemu V^* programu, tj. funkcí počtu m^* vstupních parametrů programů. Pro větší počty parametrů je

$$V^* = (2 + m^*)\log(2 + m^*) \doteq m^* \log(m^*). \tag{16.4}$$

Místo m^* je možné uvažovat jako míru počtu parametrů výraz

$$c \cdot F \tag{16.5}$$

kde c je vhodná konstanta a F je „informační objem“ definovaný vztahem

$$F = w_1 \cdot (IN) + w_2 \cdot (OUT) + w_3 \cdot (ENQ) + w_4 \cdot (FILE) + w_5 \cdot (FILEE) \tag{16.6}$$

Pro nejjednodušší variantu odhadu je $w_1 = 4$, $w_2 = 5$, $w_3 = 4$, $w_4 = 10$, $w_5 = 7$. Pro obecnější typy odhadů je možné váhy w_i odhadnout z charakteristik úlohy. Postup odhadu charakteristik *IN*, *OUT*, *ENQ*, *FILE*, *FILEE* je založen na následující proceduře.

V prvním přiblížení je *IN* počet logicky různých vstupů. Každý příkaz vstupu se počítá tolikrát, kolik obsahuje různých typů dat/proměnných; pokud se daný údaj vyskytuje se stejným formátem a se stejným zpracováním ve více příkazech vstupu, počítá se pouze jednou. Stejné údaje s různými formáty nebo různými algoritmy zpracování se počítají pro každý formát / zpracování znovu.

OUT se počítá pro výstupy obdobně jako *IN* pro vstupy.

FILE je počet interních logických souborů. Logický soubor je každá potenciálně neomezená posloupnost záznamů, která může být generována a udržována. Obvykle to jsou pracovní soubory. Počítají se logické soubory, nikoliv fyzické. Případ, kdy je délka souboru příliš velká, a posloupnost záznamů musí tedy být ve více fyzických souborech, považujeme za jediný logický soubor.

	Odhad pro	Průměrná relativní chyba	Směrodatná odchylka	Korelace délka/odhad
(a)	COBOL	0.223	0.736	0.854
(b)	PL/1	0.003	0.003	0.997
(c)	PL/1	0.007	0.057	0.997
(d)	PL/1	−0.002	0.057	0.997
(e)	PL/1	−0.002	0.057	0.997

Tab. 16.1: Kvalita odhadů délky pomocí funkčních bodů.

FILEE je obdobou *FILE* s tím, že daný logický soubor je sdílen více programy. Může se tedy jednat o společný soubor nebo o posloupnost zpráv předávaných z jednoho programu druhému programu. *ENQ* je obdoba *IN* a *OUT* s tím, že se jedná o příkazy typu dotazu, jako je např. „výstup s čekáním na vstup“. Příkladem je vyhledání údaje s daným klíčem nebo dotaz na terminál. Do *IN* a *OUT* se nesmí počítat případy zahrnuté do *ENQ* a *FILE*. Do žádných charakteristik nelze započítávat pomocné typy dat zavedené jen z důvodů technických omezení.

Funkční body F byly použity jako parametr následujících odhadů délky[2]:

$$\begin{aligned}
&(a) && Del = 118.7 \cdot (F) - 6.4, && \text{COBOL}\\
&(b) && Del = 73,1 \cdot (F) - 4.6, && \text{PL/1}\\
&(c) && Del = 13.9 \cdot (F/2) \cdot \log_2(F/2) + 5.3, && \text{PL/1}\\
&(d) && Del = 6.3 \cdot (F+2)\log_2(F+2) + 4.37, && \text{PL/1}\\
&(e) && Del = 6.3 \cdot (F \cdot \log_2(F)) + 4.5, && \text{PL/1}
\end{aligned} \tag{16.7}$$

Pro 18 programů v jazyce COBOL a 4 programy v PL/1 byly získány výsledky podle tabulky 16.1. Pro odhad spotřeby práce byly zkoumány následující vztahy[3]:

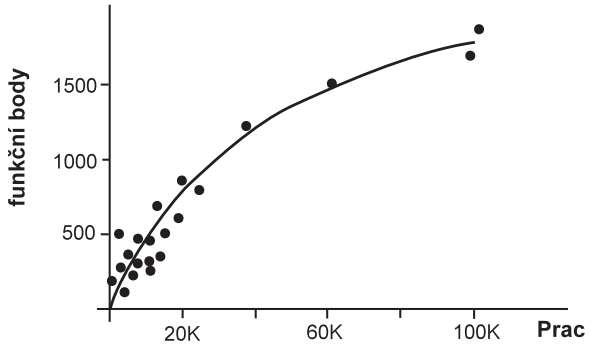
$$\begin{aligned}
&(a1)\ Prac = 54 \cdot (F) - 13.39,\\
&(b1)\ Prac = 10.75 \cdot (F/2)\log_2(F/2) - 8.3,\\
&(c1)\ Prac = 4.89 \cdot (F \log_2(F)) - 8.762.
\end{aligned} \tag{16.8}$$

Z tabulky 16.1 lze učinit závěr, že bylo dosaženo dobré kvality odhadu. Novější systémy odhadu využívají vztah $Prac \mathrel{\widehat{=}} c_1 \cdot (Del)^a$, $a > 1$ (srv. obr. 16.1). Tento vztah je používán v nových verzích odhadu pomocí funkčních bodů.

Odhad (16.8) má průměrnou chybu kolem 20 %. Chybu je možno zmenšit zavedením míry složitosti vstupů, výstupů atd. a opravou vah w_i podle výsledku a opravou hodnoty F podle dalších parametrů. Konstanty potřebné k odhadu jsou uvedeny v tabulce 16.2. Odhady složitosti se provádí pro každý „soubor“ zvlášť. Pravidla pro stanovení třídy složitosti jsou následující:

IN 1. Jednoduché vstupy: Vstupní věta obsahuje málo položek. Vstup se provádí do malého počtu vnitřních logických souborů.

2. Napravo od odhadu je uveden programovací jazyk, kterého se odhad týká.
3. *Prac* je v hodinách.



Obr. 16.1: Vztah zjištěných hodnot pracnosti *Prac* (v hodinách) pomocí funkčních bodů F. *Prac* roste rychleji než odhad, tj. pravděpodobná závislost $Prac \mathrel{\widehat{=}} c \cdot F^a, \quad 1 < b < 4/3$.

Charakteristika	Váha w_i při charakteristice		
	w_j-jednoduché	w_p-průměrné	w_s-složité
IN	3	4	6
OUT	4	5	7
FILE (vnitřní)	7	10	15
FILEE (externí)	5	7	10
ENQ	3	4	6

Tab. 16.2: Hodnoty konstant pro výpočet funkčních bodů.

2. Průměrně složité – nejsou ani jednoduché ani složité.
3. Složité: Složité věty, mnoho interních souborů je cílem vstupů. Návrh je silně ovlivněn požadavky pracovníků uživatele (uživatel si „vymýšlí“).

OUT 1. Jednoduché výstupy: Jeden až dva sloupce, jednoduché informace.
2. Průměrné: Více sloupců, částečné součty, více druhů transformací dat.
3. Složité: „Neprůhledné“ transformace dat, vzájemně závislé a složité odkazy na soubory, požadavky na rychlost provedení.

FILE, resp. *FILEE*
1. Jednoduché – málo typů vět, ve větách málo položek. Nejsou problémy s efektivností a vzpamatováním po chybě.
2. Průměrné – ani jednoduché, ani složité.
3. Složité – mnoho typů vět, mnoho položek ve větách, významné požadavky na efektivitu a restart.

ENQ – podobně jako *IN*, *OUT*.

Položme

$$IV = P(IN) + P(OUT) + P(FILE) + P(FILEE) + P(ENQ),$$

kde $P(.)$ se počítá podle tab. 16.3, ve které w_j, w_p, w_s jsou váhy uvedené v tab. 16.2 v řádku příslušném danému typu souboru. Pro takto zjištěnou hodnotu *IV* můžeme s využitím matematické statistiky odvodit pro svá konkrétní

$P(.)$	=	(počet souborů(.) jednoduché složitosti)$\cdot w_j$
	+	(počet souborů(.) průměrné složitosti)$\cdot w_p$
	+	(počet souborů(.) velké složitosti)$\cdot w_s$.

Tab. 16.3: Výpočet přínosů jednotlivých typů souborů.

data obdobu odhadů (16.7) a (16.8), kde klademe $IV = F$. To je pracné a ne vždy je k dispozici dostatek použitelných dat nutných k získání relativně spolehlivých odhadů. Proto byla metoda odhadu pomocí funkčních bodů zdokonalena následujícím způsobem. Odhad funkčních bodů *FP* získáme z *IV* vztahem

$$FP = IV \cdot TCA$$

kde $TCA = 0.65 + 0.001 \cdot DI$. *DI* je kalibrační parametr pracnosti vyjadřující vliv čtrnácti faktorů, z nichž každý je ohodnocen následující šestibodovou stupnicí:

0 – daný faktor neexistuje nebo nemá vliv,
1 – nevýznamný vliv,
2 – mírný vliv,
3 – průměrný vliv,
4 – významný vliv,
5 – velmi silný vliv na celou architekturu a programování.

Uvažované faktory jsou následující:

1. Ovládání a vstup dat přes síť (remote job entry, remote data entry).
2. Distribuované zpracování.
3. Ostré požadavky na výkonnost ovlivňují návrh, realizaci a instalaci.
4. Plné využití dané konfigurace.
5. Množství transakcí, které se provádějí, silně ovlivnilo návrh.
6. On-line vstup dat, např. vstup dat z terminálu.
7. On-line funkce, např. ovládání funkcí z terminálu.
8. Interaktivní přímé změny souborů.
9. Složitost zpracování:
   - mnoho bodů rozhodování a řídicích interakcí,
   - algoritmicky složité úlohy,
   - mnoho zpracování výjimek vedoucích k opakovanému provádění.
10. Obecná použitelnost výsledného produktu.
11. Snadná instalace a přenositelnost.
12. Snadnost práce se systémem za provozu.
13. Použití systému více organizacemi s různým způsobem využití a jinou podnikovou kulturou.
14. Snadnost změn.

DI je dáno součtem hodnocení všech faktorů. Neuvažují se faktory, jako je kvalita řešitelů či použití moderních programovacích technik. U velkých firem totiž mají tyto faktory stejné hodnocení pro všechny projekty a proto není nutné je brát v úvahu, poněvadž se eliminují kalibrací. Budou proto zahrnuty do níže uvedené konstanty *c*.

FP je kalibrováno pro konkrétní podmínky regresní analýzou pro *Prac* jako závislou proměnnou a pro *FP* jako nezávislou proměnnou. Tím získáme konečný odhad

$$Prac \mathrel{\widehat{=}} c \cdot FP + d \tag{16.9}$$

Lze také použít vztah

$$Prac = c \cdot FP^b \tag{16.10}$$

kde c a b lze odhadnout regresní analýzou pro logaritmy metrik *Prac* a *FP*. Volí se ten vztah, s nímž jsou u dané firmy lepší zkušenosti.

Pokud je málo dat z existujících aplikací, je možné použít odhad využívající průměrné množství funkčních bodů FP/M připadajících na jeden člověkoměsíc. Odhad pracnosti v člověkoměsících je pak dán vztahem $Prac \mathrel{\widehat{=}} FP/(FP(M))$. Čili pracnost nového projektu odhadneme jako podíl počtu bodů pro nový projekt děleného počtem bodů za měsíc. FP/M se zjišťuje z dokončených projektů.

Hlavní výhodou metody funkčních bodů je to, že ji lze použít již v počátečních etapách specifikace požadavků. Nevýhodou je, že neuvažuje podrobněji složitost algoritmů. Metoda je použitelná prakticky výhradně na odhad pracnosti datově náročných IS. Metodicky asi není správné, že metoda přiřazuje větší pracnost dekomponovaným systémům oproti systémům psaným jako jeden celek.

Metoda neužívá informace získané analýzou objektově orientovaných specifikací. Tyto nedostatky do značné míry odstraňuje modifikovaná metoda funkčních bodů (Function Points Mk II) popsaná v následujícím paragrafu.

16.3 Modifikovaný odhad funkčních bodů (*FP2*)

Metoda funkčních bodů má tu výhodu, že ji lze provést poměrně záhy po specifikaci požadavků. Nezohledňuje však složitost vnitřní struktury systému. Tento nedostatek do značné míry odstraňuje zdokonalená verze metody funkčních bodů Function Points Mark II. Metoda je podrobně popsána v knize (Symons, 1991). Pozdější zdokonalení této metody jsou podrobně popsána v knize (Treble, Douglas, 1996). Stručný úvod s příklady je uveden v (Shepperd, 1995).

Zdokonalená metoda funkčních bodů (*FP2*) je založena na pozorování, že každý IS je tvořen souborem logicky uzavřených akcí – transakcí, jako je vystavení objednávky, doplnění seznamu zákazníků, výdej zboží atd. Tyto transakce budeme, pokud je bude třeba odlišit od databázových transakcí, nazývat logickými transakcemi. Ve zbytku tohoto paragrafu míníme pod transakcí vždy transakci logickou. Podstatný přínos *FP2* je v tom, že odhady funkčních bodů celku se získají jako součet funkčních bodů elementárních akcí – transakcí. Postupuje se při tom v podstatě shodně jako u metody uvedené v předchozí kapitole s drobnými úpravami při výpočtu vah a hodnocení faktorů při výpočtu *TCA*. *FP2* poskytuje široké možnosti kalibrace konstant a parametrů v odhadech. Klíčovým problémem metody je identifikace transakcí. Identifikace vyžaduje jistou zkušenost. Výpočet hodnoty bodů je založen na vztahu

$$FP2 = VI \cdot Kalibrace \tag{16.11}$$

kde

$$VI = \sum_t VI_t, \tag{16.12}$$

Transakce	Vstupy	Vnitřní proměnné	Výstupy
Nový zákazník	53	1	3
Dotaz „je na skladě“	2	3	8
Záhlaví obj. – obj. přijata	20	2	1
Záhlaví obj. – obj. zamítnuta	8	2	10
Přidání položky	6	5	1
Odmítnutí položky	6	5	5
Objednávka – souhrn	2	4	4
Celkem	97	22	32

Tab. 16.4: Příklad výpočtu pro transakce prováděné při zpracování objednávek.

Součet je přes všechny transakce t. VI a VI_t je informační objem transakce t.

$$VI_t = WI \cdot p_{in} + WE \cdot p_{var} + WO \cdot p_{out} \tag{16.13}$$

Zde p_{in}, p_{var}, p_{out} jsou počty vstupů, proměnných, používaných v algoritmu transakce a výstupů. Počty vstupů a výstupů transakce t se určují podobně jako u funkčních bodů v předchozím paragrafu. *WI, WE, WO* jsou váhy. Metoda umožňuje kalibraci vah. Položme dále

$$TCA = 0.65 + c \cdot DI, \tag{16.14}$$

kde c je třeba kalibrovat. *DI* kvantifikuje vliv faktorů. *DI* je definováno vztahem

$$D = \sum_f DI_f, \quad f = 1, 2, \ldots, 19 \tag{16.15}$$

DI_f je vyjádření vlivu faktoru f podle následující škály:
0 – žádný vliv,
1 – průměrný vliv,
3 – kritický vliv.

FP2 uvažuje celkem 19 faktorů. 14 faktorů je převzato z *FP* (viz předchozí paragraf), doplněno je následujících pět faktorů:
15. Budování a udržování rozhraní na jiné aplikace.
16. Audit a ochrana dat a systému.
17. Umožnění přístupu k datům pro cizí systémy.
18. Vývoj školicích prostředků uživatele.
19. Zvýšené nároky na dokumentaci.

Z výčtu faktorů je patrné, že odhad není vhodný pro „tvrdé“ (hard) systémy reálného času a systémy, které mohou způsobit újmu na životech či zdraví. Vliv těchto faktorů není v procentech pracnosti, jako u výše uvedených 19 faktorů, ale v násobcích. *FP2* je v podstatě použitelný pouze pro typické IS, pro které je charakteristické množství dat a relativně málo algoritmicky jednoduchých operací nad nimi. Neosvědčuje se v případech operačních systémů. U systémů reálného času nejsou vhodné pro úroveň blízkou hardwaru, jako jsou drivery, a pro časově

Rezervace:	Pokoj	Cena	Rezervace	Zákazník	Smlouva	Operátor
Provedení	C	C	V	V	V	C
Dotaz	C	C	C			
Změna	C	C	M	C	M	C
Zrušení	C		Z	C	Z	C

Tab. 16.5: Matice událost/datová entita pro rezervaci hotelového pokoje. C – pouze čtení, V – vytvoření, M – modifikace, Z – zrušení.

kritické části. V matematických systémech nejsou vhodné pro odhad pracnosti vlastních matematických algoritmů. Tam je závislost mezi rozsahem dat a pracností algoritmu velmi slabá. Mohou se ale použít pro odhad pracnosti organizačních částí a UI.

Symons ve své knize provedl kalibraci vah pro řadu projektů a získal následující odhady

$$\begin{aligned} WI &= 0.58 \\ WE &= 1.66 \\ WO &= 0.26 \\ c &= 0.005 \end{aligned} \tag{16.16}$$

Navíc bylo zjištěno, že hodnota *TCA* ležela u 90 % zkoumaných projektů v rozmezí 0.75–0.95, takže lze hodnotu *TCA* s dostatečnou přesností nahradit hodnotou 0.85. Analýza *FP2* a *FP* pro existující projekty prokázala rychlejší růst *FP2* než růst *FP* v závislosti na velikosti projektu, což naznačuje lepší vlastnosti odhadu *FP2*. Pracnost jedné instrukce velkých systémů je větší než pracnost instrukce malých systémů.

FP2 je při jisté disciplíně práce vhodné pro vyhodnocování pracnosti objektově orientovaných systémů. Pro *FP2* je výhodné, aby se logické transakce kryly s metodami některých objektů. V tom případě je možné poloautomatické vyhodnocování metriky *FP2*. Klíčovým problémem je identifikace transakcí. Za transakci se považuje jednoznačná logicky uzavřená akce v realitě, což je součást specifikace. Je při tom třeba uvažovat všechny smysluplné varianty provedení transakce. Typické transakce jsou:

- přidat obchodního partnera do katalogu,
- zpracování záhlaví objednávky,
- zpracování řádků objednávky,
- výpočet mzdy.

Transakce není ve *FP2* přesně definována, takže různí hodnotitelé se nemusí při volbě transakcí shodnout. Někdo může považovat celé zpracování objednávky za jednu transakci, jiný – což se zdá přirozenější – oddělí zpracování záhlaví (jedna transakce) od zpracování řádku (druhá transakce). Praxe ukazuje, že za dosti snadno splnitelných podmínek nebývají rozdíly mezi experty velké. To platí zvláště tehdy, vymezují-li se transakce podle následujících zásad:

a) Logická transakce obvykle nezahrnuje více transakcí v databázovém smyslu, často se obě transakce kryjí.
b) Strukturu transakcí a jejich parametry lze poměrně přesně odvodit z datového modelu. Příkladem je odhad informačního objemu *VI* pro skupinu transakcí zpracování objednávek založený na datech z obr. 16.2. Možné hodnoty metrik vstupy/počet vnitřních proměnných/výstupy transakcí jsou pro náš příklad v tabulce 16.4.

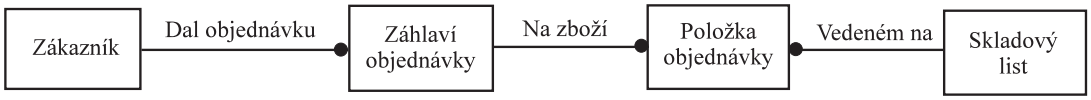
Etapa	Bez CASE		S CASE	
	Prac%	*Doba*%	*Prac*%	*Doba*%
Specifikace (analýza)	22	35	43	45
Návrh	15	15	38	35
Kódování, test. částí	46	25	2	2
Integrační testování	12	15	12	8
Test. funkcí, předání	5	10	5	10

Tab. 16.6: Procenta pracnosti a doby pro jednotlivé etapy životního cyklu podle (Symons, 1991). O CASE se předpokládá, že používá při generaci kódu. Předpoklad 2 % na kódování a testování částí při použití CASE je asi příliš optimistický.

Informační objem *IV* je dán vztahem $IV = 97 \cdot WI + 22 \cdot WE + 32 \cdot WO$, kde lze za *WI*, *WE* a *WO* dosadit z rovnic 16.16.

c) Při definování transakcí je výhodné využívat informace obsažené v diagramech toků dat. Analyzují se algoritmy prováděné v těch procesech, které nejsou dále dekomponovány do podřízených diagramů toku dat. Využívají se datové toky do/z procesu.

d) Mnohé CASE systémy vytvářejí V/U matice s řádky odpovídajícími událostem reálného světa a se sloupci odpovídajícími datovým entitám. Elementy matice jsou buď V – vytvoř a značí, že příslušná akce vytvoří či modifikuje daný údaj, nebo U – daná akce příslušný údaj pouze používá. Někdy se v matici udávají všechny datové akce: V – vytvoř, C – čti, M – modifikuj, Z – zruš[4]. Řádky matice velmi často odpovídají logickým transakcím (viz tab. 16.5).



Obr. 16.2: E-R diagram dat pro zpracování objednávek.

Důležité je odlišit transakce se stejnými vstupy a výstupy a rozdílnou funkčností. Např. změna rodinného stavu představuje celkem pět různých smysluplných přechodů mezi stavy svobodný – ženatý – rozvedený – ovdovělý.

V (Symons, 1991) je uveden postup kalibrace vah *WI*, *WE*, *WO* a hodnoty *c*. V konkrétním podniku je vhodné odvodit vztah

$$Prac \mathrel{\widehat{=}} f(FP2), \tag{16.17}$$

kde f značí hledanou závislost, podobně jako v případě funkčních bodů počítaných pro vstupy a rozhraní systému jako celku. Tak je možné s využitím analýzy regrese pro dříve realizované projekty odvodit odhad

$$Prac \mathrel{\widehat{=}} a \cdot (FP2) + d \tag{16.18}$$

nebo

$$Prac \mathrel{\widehat{=}} a \cdot (FP2)^b \tag{16.19}$$

4. V metodikách pocházejících z anglicky mluvících zemí se používají zkratky C – create, R – read, U – update, D – delete.

Hlavní parametr odhadu	Délka	*FP*	*FP2*
Standardizováno	ne	ano, zlepšuje se	ano
Přesnost definice	potenciálně možná[5]	část. subjektivní	téměř ano
Strukturovanost	ne	ne	ano
Snadnost pochopení	ano	jen zdánlivě	ano
Automatizovatelné	ano	obtížně	ano
Veřejné systémy	obojí	veřejné	veřejné
Rozsah použití	velký	velký	přiměřený
Organizace uživatelů	několik	IFPUG[6]	IFPUG
Školení	různé úrovně	ano	ano

Tab. 16.7: Aktivity na vývoji metod odhadu založených na metrikách *Del*, *FP*, *FP2*. Podle (Symons, 1991). *FP2* je automatizováno v některých CASE systémech.

Volí se ten model, který má lepší výsledky. Pro dobu řešení zvolíme odhad

$$Doba \mathrel{\widehat{=}} e \cdot (FP2)^b, \quad e = 1.5,\ b = 0.35, \tag{16.20}$$

Pro ostré termíny lze též využít vztah

$$Doba \mathrel{\widehat{=}} c_{13} \cdot Prac^{1/3}. \tag{16.21}$$

Pokud je k dispozici dostatečně velký soubor dat, je možné faktor e a exponent b v 16.10 zpřesnit statistickými metodami analýzy regrese. Pokud není k dispozici dostatečný soubor dat, lze použít následující vztah pro produktivitu odvozený v (Symons, 1991). Produktivita v bodech *FP2* za hodinu je pro $FP2 < 1000$ dána vztahem

$$Prod_H(FP2) = A \cdot (0.11 \exp(-((FP2 - 250)/575)^2) + 0.01 \cdot FP2^{1.1}/522) \tag{16.22}$$

kde $A = 1.6$ pro 4GL jazyky a $A = 1.0$ pro 3GL jazyky, např. pro COBOL. Hodnoty pro OO jazyky se zatím předpokládají stejné jako pro 4GL jazyky. Pro $FP2 > 1000$ klademe $Prod_H = 0.06$ pro 4GL jazyky. $Prod_H = 0.04$ pro 3GL jazyky. Pro odhad pracnosti $Prac(m)$ pro hodnotu m metriky *FP2* pak platí

$$Prac(m) \mathrel{\widehat{=}} m/Prod(m) \tag{16.23}$$

Takto získaný odhad se upravuje podle následujících pravidel:

a) Oprava na velikost. Zvětšit odhad pracnosti o 25 % a doby řešení o 20 %, jedná-li se o projekt více než třikrát větší, než byly dosud řešené projekty.
b) Oprava na známé. Snížit odhad pracnosti o 1/3 a dobu o 1/5, jde-li o reimplementaci dobře známého systému nebo řeší-li se podobný problém a jsou dobré vztahy se zákazníkem.
c) Oprava na problémy se specifikacemi. Zvýšit odhad pracnosti a doby až o 100 % při existenci následujících skutečností:
   – nepřesná definice funkcí,

5. Není jasné, zda do délky zahrnovat např. deklarace.
6. Information points user group, organizace uživatelů.

- uživatel se neúčastní prací na specifikacích ani jako oponent,
- uživatel musí provádět organizační změny,
- více uživatelů s různými požadavky, které jsou zčásti ve sporu,
- obtíže při specifikaci rozhraní na jiné systémy,
- změny požadavků za pochodu,
- změny prostředí za pochodu.

d) Nové dosud neznámé nebo nepoužívané metody a nástroje.
- použitá nová metoda, nevyžaduje nový nástroj – zvýšit pracnost o 1/3 a dobu o 1/5,
- použit nový nástroj, nemění se podstatně metoda – zvýšit pracnost o 10 %, dobu o 5 %,
- použita nová metoda podporovaná novým nástrojem – zvýšit pracnost o 50 % a dobu 30 %.

Tento odhad se provádí pro každou etapu zvlášť s použitím tab. 16.6.

e) Ostré termíny. Položme

$$SCF = \frac{\text{Požadovaná doba}}{\text{Odhad doby (po opravách výše)}}.$$

Nedopustit, aby $SCF < 0.5$. Pro $0.5 < SCF < 0.75$ zvýšit odhad pracnosti o 50 % až 100 %. Lepší cesta je snížit pracnost uplatněním následujících opatření:
- rozdělit projekt na části,
- vyloučit některé funkce,
- využít hotové části; to je možné zvláště pro objektově orientované systémy a při spolupráci aplikací,
- vyloučit rizikové faktory uvedené výše.

16.4 Zlepšování kvality odhadů v softwarových projektech

Problém odhadu parametrů softwarových projektů má zásadní důležitost pro podnikání v oblasti softwaru. Z toho důvodu byla vytvořena řada pracovních skupin, zabývajících se problémy odhadu. Tyto aktivity jsou zaměřeny především na tři výše uvedené metody odhadu. Zahrnují jak skupiny pracující na čistě komerčním základě, tak skupiny, výsledky jejichž práce jsou veřejné (public domain).

Vzhledem k prudkým změnám technologie, silné závislosti na obtížně ovlivnitelných faktorech, posunech v předmětu činnosti softwarových firem atd. nejsou výsledky odhadu nijak oslňující, jsou však prakticky použitelné. Současnou situaci popisuje tabulka 16.7. Nástroje odhadů nabízejí některé CASE systémy se střídavým, spíše menším úspěchem.

Z výše uvedeného plyne, že odhady lze kalibrovat, jen jsou-li dostupná data z mnoha projektů. To je možné pouze u větších softwarových firem. Odhady metrik pro customizaci nejsou zatím k dispozici.

17

Dokumentace

Dokumentace je důležitým nástrojem při projekci, realizaci i provozu a údržbě softwarového díla. Platí to jak o technické, tak o administrativní a ekonomické dokumentaci. Řádně vedená dokumentace často vyžaduje více práce než kódování (kap. 15). Práce na dokumentaci se ale při vývoji a hlavně údržbě bohatě vyplatí. Bez řádně vedené dokumentace nelze realizovat velké projekty. Dokumentace představuje do značné míry paměť firmy a to nejen po stránce věcné, ale i administrativní. Je důležité znát nejen technické aspekty realizací, ale i jejich ekonomické parametry, jako náklady, skluzy, spolehlivost partnerů atd.

Kvalita uživatelské dokumentace do značné míry rozhoduje o úspěchu či neúspěchu softwaru v praxi, zvláště v případě softwaru provozovaného na mnoha instalacích. Význam má jen taková dokumentace, která je

- aktuální,
- přehledná a srozumitelná,
- umožňuje opravovat či modifikovat programy bez rizika zavlékání dalších chyb,
- umožňuje snadno ověřovat správnost návrhu implementace
- umožňuje sledovat průběh prací.

Za současného stavu výpočetní techniky vedení dokumentace neznamená nutně jen „papírování“. Je možné využívat služeb počítače i pro redakci dokumentačních textů, udržování kartoték, testovacích souborů, vyhledávání křížových odkazů apod. Softwarové dokumenty mají obsahovat informace o tom,

- jak systém používat,
- jak systém instalovat a obsluhovat,
- jak systém udržovat,
- popis realizace,
- testové procedury, testovací data, hodnocení testů,
- ostatní dokumenty.

Při předání do užívání mají být k dispozici dokumenty (v pořadí důležitosti):

a) Manuály a uživatelská dokumentace.
b) Dokumenty pro údržbu:
   – zdrojové programy s komentáři a křížovými odkazy,
   – systémová dokumentace, základní dokumenty o vlastnostech softwaru. V této dokumentaci je třeba zachytit specifikace požadavků, cíle systému, metody dekompozice do programů a programů do částí, vazby mezi

komponentami a funkcemi systému, vlastnosti uživatelského rozhraní a rozhraní mezi komponentami a především plán testů a ostatní testovou dokumentaci.

Cenné bývají informace obsažené v deníku projektu. U velkých úkolů může být výhodné provádět kontrolu konfigurace i pro systémové dokumenty s využitím vhodných nástrojů. Pokud IS udržuje dodavatel, zůstávají dokumenty pro údržbu u něho.

17.1 Uživatelská dokumentace

Pomocí uživatelské dokumentace se uživatel obvykle poprvé seznamuje se systémem. Uživatelská dokumentace by měla poskytnout přesnou představu o práci systému. U IS je důležité, aby dokumentace nebyla psána jako reklama, důležitá je věcnost a „solidnost“ výkladu. Uživatel by neměl být nucen studovat celou dokumentaci, chce-li používat jednoduché funkce. Dokumentace by měla být strukturována tak, aby mohl uživatel studovat vždy právě jen to, co potřebuje ke své práci. Uživatelská dokumentace obvykle obsahuje následující dokumenty:

1. *Návod k instalaci* – vysvětlení, jak přesně uvést systém pro dané technické vybavení do provozu. Dokument se nevyžaduje, provádí-li instalaci dodavatel.
2. *Úvod do systému* vysvětlující co nejjednodušším způsobem, nejlépe na nějakých jednoduchých příkladech, jak začít se systémem pracovat. Je výhodné, usnadňuje-li toto počáteční seznámení nějaký program „učitel“ (demo, tutor) umožňující postupně zvládat i složitější činnosti.
3. *Popis funkcí* – seznam činností systému.
4. *Referenční manuál* – podrobný slovník funkcí systému, které může uživatel používat.
5. *Návod k použití* – učebnice používání systému.
6. *Příručka operátora*, je-li operátor potřeba. Dokumentace potřebná pro činnosti, které má operátor provádět při chodu systému.

Popis funkcí systému je obvykle formulován jako přehled požadavků na systém a na cíle implementace. Z této části by mělo být zřejmé, co systém může a co nemůže dělat. Cenné jsou malé příklady práce vystihující instruktivním způsobem funkce systému, např. operace přidat, zrušit nebo změnit záznam v databázi. Popis funkcí má především zlepšit intuitivní chápání systému. Popis funkcí tvoří doplněk úvodu do systému.

Úvod do systému je velmi důležitou, často však opomíjenou částí uživatelské dokumentace, především tehdy, kdy není prováděno školení uživatele. Stává se, že i dobrý software zapadne právě pro opomenutí této části uživatelské dokumentace. Návod pro začátečníky by měl na příkladech „běžného použití“ popsat všechny akce uživatele, včetně připojení do elektrické sítě, zapnutí počítače, zda např. svítí signální dioda, nebo co může být důvodem chyby atd. Úvod do systému by měl být napsán tak, aby bylo možné jednoduchý příklad provést podle návodu krok po kroku, podobně jako tomu bývá u nově zakoupeného videorekordéru.

Úvod do systému a popis funkcí jsou důležité i proto, že je pro zvládnutí práce se systémem nutné překonat počáteční bariéru, kdy uživatel ještě není schopen se systémem komunikovat a nedokáže systém přimět k rozumné reakci. Po překonání této počáteční bariéry v situaci, kdy jsou uživatelé již schopni provádět jednoduché práce, se mohou učit tím, že se systémem pracují. Takové učení už obvykle vyžaduje použití referenčního manuálu. Nejdůležitější vlastností úvodu do systému a popisu funkcí je srozumitelnost.

Referenční manuál je normou užívání systému. Nejdůležitější vlastností referenčního manuálu jsou úplnost a přesnost. Pokud je to možné, je výhodné použít pro přesnou definici formální prostředky, nesmí to však vést k obtížím při porozumění. Taková chyba se stala při definici některých programovacích jazyků, např. Algolu 68.

Referenční manuál může předpokládat znalost popisu funkcí a znalost úvodu do systému, čili znalost terminologie a základních pojmů. Je možné předpokládat, že čtenář spolupracuje při čtení manuálů se systémem, zkouší funkce. Opomíjenou částí jsou chybové stavy: důvod vzniku, místo vzniku, jak reagovat.

Návod k instalaci by měl vždy obsahovat údaje o paměťových médiích, na kterých je systém dodáván: médium, formát dat, způsob zápisu informace atd. Dále je třeba popsat minimální technické vybavení nutné pro práci systému, uvést, jak je třeba vytvořit pomocné soubory na disku, jak se systém startuje a jak se přizpůsobuje vlastnostem hardwaru (druhy tiskáren, grafických zařízení atd.). Pokud je nutné generování systému, musí být k dispozici příslušný program a návod k práci s ním.

Při tvorbě manuálů mohou pomoci specialisté na psaní technických textů. Míra účasti vývojářů musí být ale vždy vysoká. Existují firmy specializované na psaní manuálů. Taková firma použije podklady tvůrců softwaru a na základě vlastních zkušeností se softwarem napíše manuál. Tento postup se u menších firem velmi osvědčuje. Manuál napsaný tímto způsobem přispěl podle sdělení P. Vody k počátečnímu úspěchu jazyka Trilogy (P. Voda je autor jazyka). U velkých firem píší manuály specializované týmy spolupracující s autory systému, protože psaní manuálů vyžaduje jiné schopnosti, než vývoj softwaru.

17.2 Dokumentace pro údržbu

Pro údržbu bývá výhodné postihnout vazby uvnitř systému na úrovni atomických jednotek systému – datových struktur, proměnných a podprogramů. V tom nám mohou pomoci softwarové nástroje používané při vývoji programů. Při použití databázového systému může dobře komentované schéma databáze, které beztak musíme vytvořit, posloužit jako slovník datových typů atd.

Slovník dat je cennou pomůckou pro údržbu. Položka slovníku obvykle obsahuje tyto údaje:

- identifikátor objektu,
- význam objektu (identifikátor typu, proměnná, konstanta),
- typ hodnot (popis může využívat konstrukce použitého programovacího jazyka),
- oblast platnosti (lokální v…, exportovaný z…, používaný v…, na haldě),
- charakterizace povolených hodnot a význam důležitých konstant,
- křížové odkazy.

Pro databáze obvykle postačuje komentovaná definice tabulek v syntaxi SQL a E-R diagramy.

Slovník (pod)programů nebo tříd a jejich metod:

- identifikátor,
- typ použití (systém, databáze,…),
- stručný popis funkce nebo metod,
- popis rozhraní: tvar volání, kdo a jak používá a jaké předává parametry, jaká jsou používaná společná data,
- zobrazení vztahu „x voláno ze z“,
- u tříd popis atributů.

17.3 Umění psát dobrou dokumentaci

Kvalita dokumentace je stejně důležitá jako kvalita programu. Přes nejrůznější doporučení a normy bývá kvalita dokumentace softwaru často na dosti nízké úrovni. Dokumentace bývá neúplná, zastaralá a především nepřehledná.

Dokumenty bývají psány toporně a neinstruktivně. Dlouhé a šroubované formulace zatemňují význam a zhoršují čitelnost.

Programátoři jsou často přesvědčeni, že dokumentace je něco, co lze zvládnout levou rukou. Psaní dokumentace bývá pro programátora nezajímavá činnost, na kterou nerad myslí při mnohem zajímavějším vývoji programů. Tento postoj je mnohdy podporován sklony vedení projektu vyžadovat dokumentaci ne proto, že je potřeba, ale proto, že to je předepsáno. To se může negativně projevovat jak v požadavcích na obsah dokumentů, tak v postoji k vytvořeným dokumentům. Dokument se pouze zaeviduje a pak se uloží ad acta – dá se do almary čili almarizuje.

Tvorba dokumentů není ani snadná, ani levná. Je proto důležité, aby při tvorbě dokumentace byly uplatňovány procedury kontroly kvality podobně jako při tvorbě programů pomocí oponentur, inspekcí a procedur schvalování. Je to o to důležitější, že dokumentace nemůže být jiným způsobem otestována. Je-li to jenom trochu možné, vyplatí se dokumentaci „otestovat“ tím, že ji někdo před předáním skutečně použije.

Tvar dokumentů by měl být dohodnut předem a měl by být pokud možno jednotný, měl by být standardizován. Tak se snáze zajistí, že se na nic nezapomene. Jednotné členění dokumentů usnadňuje orientaci. Součástí dohod o tvaru dokumentů mohou být konvence pro číslování stránek, forma odkazů na stránky a na jiné dokumenty, metody číslování kapitol a podkapitol. Tyto zdánlivé maličkosti mohou značně zkomplikovat život, nejsou-li správně vyřešeny, zvláště u rozsáhlých dokumentů. Je důležité, aby dohodnuté normy nebyly příliš úzkoprsé, nestandardizovaly to, co se standardizovat nemusí. To je obecný problém všech norem. Pro strukturu mnoha dokumentů existují normy IEEE nebo ISO (kap. 20)

Pečlivě a včasně vypracovaná dokumentace je dobrým testem specifikací požadavků. Obtíže při psaní dokumentace jsou dobrou indikací špatného návrhu funkcí. Jinými slovy: jestliže se logika dat nebo jejich zpracování obtížně popisuje, znamená to pravděpodobně, že je špatně navržena.

Abychom se při psaní dokumentace vyhnuli šroubovaným obratům, musíme často zavést terminologii ad hoc. Neoddělitelnou součástí umění psát dokumentace je proto schopnost navrhovat a používat přiměřené názvosloví.

Dobře bývají dokumentovány ty produkty, u nichž uživatelská dokumentace nevznikla dodatečně, až po oživení systému. Dokumentace vytvářená předem nese riziko, že ne všechny rysy, které autoři slibují, budou implementované. Ukazuje se ale, že:

a) schopný tým dokáže dodržet minimálně 90 % svých příslibů;
b) při konečné revizi dokumentace do finální formy pak zbývá dost sil na pedagogickou stránku. Většina nepopulární práce je totiž již dávno hotova;
c) předběžná uživatelská dokumentace pokryje velkou část informací, která musí být beztak obsažena v kvalitní specifikaci požadavků.

17.4 Jazyková kultura v dokumentech

Kvalita dokumentace je silně ovlivňována literárním stylem, jímž je napsána. Přesný a instruktivní popis vyžaduje i v technické oblasti lehké pero a dobré vyjadřovací schopnosti. Dobré dokumenty jsou psány dobrou češtinou (nebo angličtinou). Informatici však často vládnou lépe programovacím než mateřským jazykem. Výuka češtiny na školách není dostatečně zaměřena na jazyk jako komunikační prostředek, ale spíše na biflování faktů, jako kdy se který spisovatel narodil atd. K tomu přistupuje to, že humanitní nadání nebývá u informatiků příliš rozvinuto. Komplikovanost problémů ztěžuje tvorbu kvalitní dokumentace.

17.4 Jazyková kultura v dokumentech

Jazyk odborných publikací se dosti liší od jazyka beletrie, není však pravda, že u odborného textu je otázka slohu vedlejší. Pro čtenáře není právě příjemné proplétat se při zvládání nových poznatků džunglí komplikovaných vět a slangovými termíny.

Psaní odborných textů vyžaduje průpravu a praxi. I pak nelze očekávat, že napíšeme dokumentaci snadno, rychle a najednou. Napsaný text by měl být pečlivě čten autorem i jeho spolupracovníky, oponován a upravován, dokud nejsme s výsledkem spokojeni. Neexistuje jednoduchý návod, jak dobře psát technické texty. Některé zásady je však dobré dodržovat. Mnohé z těchto zásad známe ze školních lavic (srv. Sommerville, 1996):

1. Je výhodné psát kratší věty. Každá věta má tvořit jednu informační jednotku. Pro čtenáře je nepříjemné si pamatovat více faktů současně jen proto, že jsme uvnitř jedné dlouhé věty[1]. V případě, že potřebujeme pracovat s více fakty, je výhodné použít výčet (seznam) případně zarámovaný větnou konstrukcí, např. Metody práce jsou následující: a) ...
2. Používáme-li často odkazy, je vhodné pro zvýšení čitelnosti nepoužívat pouze čísla kapitol, resp. paragrafů, ale občas se vyplatí uvést název paragrafu, např. při prvním výskytu odkazu v daném kontextu. Tak např. místo „Jak víme z kap. 15 ...“ volíme při prvém výskytu odkazu na kap. 15 formulaci „Jak víme z kap. 15, kde jsou uvedeny základní softwarové metriky,...“.
3. Snažíme se o maximální stručnost.
4. Nešetříme slovy tam, kde jsou třeba. To se týká především obtížných míst, kdy se někdy vyplatí tutéž věc vysvětlit dvakrát různými slovy nebo uvést instruktivní příklad. Je ale nutné, aby bylo zřejmé, že se věc vysvětluje ještě jednou.
5. Je třeba zajistit jednoznačnost používaných termínů. Tak např. pojem proces může mít v různém kontextu různý význam. Pro ty uživatele, u kterých je nebezpečí, že řadu termínů neznají, je vhodné doplnit anotovaný slovník používaných pojmů. Volba termínů závisí na tom, komu je dokument určen, zda odborníkům nebo širší veřejnosti.
6. Dokument by měl být dekomponován do paragrafů maximálně několik stránek dlouhých. Odstavce by obvykle neměly být delší než deset vět a měly by obsahovat relativně samostatný úsek informace.
7. Hrubé přestupky proti jazykové kultuře rozptylují čtenáře a zmenšují jeho důvěru k systému jako celku. Na druhé straně není správný jazykový purismus a zbytečně suchý a nezáživný výklad.
8. Dobrá grafická úprava silně zvyšuje čitelnost. Při psaní dokumentů je výhodné využívat editační systémy s různými typy písma.
9. Dokumenty mají být, podobně jako software, hierarchicky rozčleněny na nepříliš dlouhé paragrafy.

Nejúčinnější metodou zlepšování kvality dokumentace je inspekce prováděná formou popsanou v kap. 8. Při inspekci uživatelských dokumentů je přípustné nejen najít chybná nebo nevhodně napsaná místa, ale i v jednoduchých případech navrhovat úpravy.

Ke zlepšení kvality dokumentů lze použít různé formátovací programy a programy, které jsou schopny nalézt gramatické chyby, častá použití stejných slov apod. Dobrou přípravou je výcvik ve stylistice a v mluveném projevu.

1. Bohužel je někdy podstata věci, že musíme mít na paměti mnoho faktů současně.

17.5 Nástroje tvorby dokumentace

Pro tvorbu a údržbu dokumentů je výhodné využít softwarové nástroje a udržovat dokumentaci na počítači. Moderní způsoby práce s grafickou informací umožňují, aby dokumentace na počítači nebyla v žádném směru horší než dokumentace vytvořená klasickými metodami. Udržování dokumentace na počítači má mnoho výhod:

- na počítači mají všichni vždy k dispozici poslední verzi dokumentace,
- dokumenty se snadno používají a udržují; lze použít editory, případně dokumentografické a jiné systémy,
- dokumenty mohou být formátovány a připraveny k tisku přímo na počítači,
- lze provádět automatizovanou analýzu textů: generování odkazů, hledání překlepů, indexaci atd.,
- jeden dokument může být snáze vytvářen souběžně více pracovníky,
- usnadní se řízení prací při přípravě a údržbě dokumentů; lze provádět akce řízení konfigurace a kontroly změn obdobně jako u programů,
- dokumenty jsou snadno dostupné,
- lze použít výkonné nástroje vyhledávání.

Práce s dokumenty na počítači má i svá úskalí. Práce u obrazovky je ergonomicky namáhavá. Pohled na dokument prostřednictvím obrazovky někdy připomíná pohled na svět klíčovou dírkou, obrazovka zobrazí jen malou část dokumentu a na obrazovce nemohu listovat nebo čmárat tak snadno, jako v papírech na stole. Výhodná je hypertextová forma dokumentů.

Vyhledávací systém na počítači musí mít k dispozici indexy, podle kterých se vyhledává, a to může znamenat práci navíc. Obrazovka není vždy dostatečně velká k zobrazení celého listu textu a editor pro přípravu dokumentů nemusí být identický s editorem pro přípravu programů. Tyto zdánlivé maličkosti mohou značně zkomplikovat práci členům týmu. Pomineme-li problém rozměru obrazovky, musíme vyřešit problém volby textového editoru. Pro menší rozsah dokumentů vyhovuje libovolný textový editor s možností práce s grafickou informací. Pro větší systémy je žádoucí složitější prostředek umožňující číslování řádek, paragrafů, formátování, označování klíčových slov atd.

Nástroje pro vývoj, údržbu a práci s dokumenty jsou užitečné nejen při vývoji softwaru. Z výše uvedených příkladů však plyne, že pro softwarovou dokumentaci jsou potřeba editační systémy poměrně vysoké třídy. Pro správu dokumentů jsou vhodné plnotextové (full text) databáze. Tyto databáze pracují s úplnými texty dokumentů a mají silné nástroje vyhledávání a indexace.

17.6 Údržba dokumentace

Pokud je IS měněn, což je obvyklé, musí být měněna i dokumentace. Každá změna musí být promítnuta do všech dokumentů, jichž se týká. Oprava kódovací chyby může být promítnuta pouze do programu. Může však dojít i k úpravám, které musí být promítnuty do návrhu systému, dokumentů o testech, do uživatelské dokumentace a nezřídka i do specifikací požadavků. Je přirozenou snahou rychle opravit defekty v programech a úpravy v dokumentaci nechat na později. Z „později“ bývá často „nikdy“.

Programy pro práci s dokumenty by měly upozorňovat členy týmu na případy, kdy je pravděpodobně nutné upravit dokumentaci. Lze využít vztahu mezi částmi programů a částmi dokumentace a záznamy o úpravách. Uživatel by měl být informován o změnách v používaném softwaru při startu své úlohy. Podrobnosti je vhodné též popsat v „občasnících“ (newsletters) a v deníku projektu. Změny by měly být vyznačeny i v manuálech. U každého dokumentu musí být vyznačena verze, jíž se týká, a datum, kdy byl převzat do užívání, nejlépe na titulní straně.

Při přenosu softwaru z jednoho počítače na jiný počítač bývá nutné upravovat dokumentaci. Rozsah prací, které jsou k tomu nutné, závisí – podobně jako u programů – na tom, byla-li potřeba přenosu vzata v úvahu již při návrhu systému a při psaní dokumentace. Pro přenositelnost dokumentace je důležité, aby dokumentace byla kompletní, tj. obsahovala všechny důležité informace. Může se např. stát, že program pracuje s knihovnou matematických funkcí, která není identická s knihovnou v cílovém počítači. V tom případě není dokumentace úplná, odvolává-li se na knihovnu a nepopisuje vlastnosti používaných funkcí.

Při přenosu musíme dobře dokumentovat především ty části, které jsou závislé na typu počítače a podpůrném softwaru. Bývá to organizace souborů, pravidla tvoření jmen souborů, jazyk správy úloh a ovládání vstupů a výstupů. Pro usnadnění přenosu dokumentů můžeme použít stejný obrat jako u programů. Části, které je třeba při přenosu upravit, soustředíme do zvláštních sekcí. Tyto části při přenosu přepíšeme, zbytek může být přenesen beze změny. Moderní informační technologie problém přenosu velmi usnadňují.

Postupně se vyvíjejí prostředky rekonstrukce dokumentace z fungujícího systému. Tento proces je znám jako *reverse engineering*. Reverse engineering je velmi komplikovaný proces, který z principu nemůže být zcela dokonalý. Některá vývojová prostředí problém částečně obcházejí tím, že systém je definován pouze prostředky vysoké úrovně vývojového systému a bez vytvoření klíčových dokumentů ho nelze oživit a beze změny dokumentů ani změnit.

17.7 Administrativní a ekonomická dokumentace

Až dosud jsme rozebírali pouze dokumentaci, která je potřebná k technické realizaci softwaru a jeho používání. U organizací, které pracují jako dodavatelé softwarových prací, je nutné vypracovat řadu dokumentů a dokladů nutných k uzavření smlouvy, k fakturování prací atd. Rozsah této dokumentace je nemalý a v řadě podrobností se u různých organizací liší. Navíc se časem mění různé ekonomické předpisy. Omezíme se proto na základní fakta. Ekonomické a administrativní dokumenty tvoří několik skupin (srv. Baar, 1987).

1. Podklady pro uzavření hospodářské smlouvy:
   - odhady nákladů,
   - vlastní hospodářská smlouva,
   - protokol o předání/převzetí projektu
   - zápis o ukončení zkušebního provozu, atd.
2. podklady pro fakturaci a sledování nákladů:
   - pracovní výkazy pracovníků,
   - výkazy po úkolech (sumáře),
   - ostatní náklady: investice, provozní náklady aj.

Pro zkvalitnění odhadů nákladů a prevenci chyb u budoucích projektů je rozumné shrnout zkušenosti s řešením projektu do dokumentu „Vyhodnocení projektu“. V tomto dokumentu by měl být porovnán plán se skutečností a měla by být provedena analýza příčin odchylek a skutečností hodných pozornosti. Studium materiálů dokončených projektů může poskytnout mnoho cenných zkušeností začínajícím vedoucím týmů.

Administrativní dokumentace je pracná. Je důležité, aby se vedoucí týmu nevěnoval jen jí. Je ovšem nutné, aby vedoucí za kvalitu ekonomické informace odpovídal. Optimální tedy je, když práce spojené s vedením administrativní dokumentace rozdělí mezi členy týmu (viz též popis služeb v týmu v kap. 10) s tím že definitivní tvar dokumentů schvaluje vedoucí projektu. Vedoucí musí být odborně na výši, a proto se nesmí utopit v papírování.

Ekonomická a administrativní data lze využít pro evidenci a analýzu kvantitativních charakteristik vytvářeného softwaru (viz kap. 15). Výsledky analýzy mohou být využity pro zkvalitnění ekonomických rozhodnutí, při uzavírání smluv a při volbě strategie softwarové firmy.

17.8 Normy pro vedení dokumentace

Struktura a kvalita dokumentace je jedním z hlavních témat softwarových norem. Obecné zásady pro dokumentaci obsahuje norma pro zajišťování kvality softwaru ISO 9000–3.

17.8.1 Požadavky na kvalitu dokumentace podle ISO 9000

Podle normy ISO 9000–3 má dokumentace splňovat následující požadavky:

a) Dokumentace musí být vhodná pro účel, pro který je určena. Dokument má umožnit vhodně školenému pracovníkovi splňujícímu odpovídající kvalifikační předpoklady správně provádět to, k čemu je dokument určen nebo to, co předepisuje.
b) Každý dokument musí mít vlastníka, osobu či oddělení, který za dokument ručí. Vlastníkem nemusí být autor dokumentu.
c) Dokument má být oponován před tím, než je uvolněn pro používání. Průběh a závěry oponentury musí být dostupné všem oprávněným osobám. Zápis a závěry oponentury obsahují jméno schvalující osoby a jméno organizace.
d) Distribuování dokumentu musí být řízeno a zajišťováno následujícími opatřeními:
   – Musí být určena odpovědnost za originál (master copy) dokumentu.
   – Jsou vedeny záznamy o oběhu dokumentů.
   – Pokud je dokument v elektronické formě a je interaktivně přístupný, mělo by všem uživatelům být dáno na vědomí, že originál je v elektronické interaktivní formě.
   – Dokument by měl mít jasně vyznačenou verzi.
   – Číslování stránek má formu číslo stránky / celkový počet stránek.
   – Dokumenty jsou zničeny nebo označeny za neplatné v okamžiku, kdy jsou nahrazeny novou verzí.

17.8.2 Faktory kvality dokumentace

Při tvorbě dokumentace je třeba sledovat následující vlastnosti dokumentů (srv. normu IEEE 1298–1992):

a) *Srozumitelnost*. Dokument musí být srozumitelný pro ty, o nichž se předpokládá, že budou dokument po případném zaškolení používat. Struktura dokumentu a použité techniky jeho prezentace tedy závisí na tom, kdo bude dokument používat. I přesný dokument je nepoužitelný, nejsou-li mu uživatelé schopni rozumět.
b) *Úplnost*. Dokument by měl obsahovat všechny potřebné informace; u softwarové dokumentace to jsou především pravidla ovládání, seznam funkcí, podmínky či omezení pro používání a vlastnosti rozhraní. Součástí dokumentů mají být i reakce na chybná data či na chyby obsluhy a popis vazeb na standardy. U chybějících údajů by mělo být uvedeno, že chybí a kdy budou doplněny. Dokument by měl obsahovat index a odkazy na související dokumenty.
c) *Testovatelnost*. Požadavky a cíle mají být formulovány tak, aby se daly dobře ověřit. Výhodné jsou kvantitativní údaje. Je tedy žádoucí např. místo požadavku „odpověď přijde většinou do 5 sec“ stanovit např. „pouze 5 % odpovědí má dobu odezvy větší než 5 sec“. Pak je ale vhodné stanovit další omezení na maximální dobu

odpovědi. Doporučuje se uvést i metody ověřování splnění určitých požadavků v těch případech, kdy mohou být pochybnosti. Příklady netestovatelných požadavků: „Systém nespadne do věčného cyklu“ nebo „systém má uživatelsky příjemné rozhraní“.

d) *Modifikovatelnost*. Forma dokumentu umožňuje snadné a kontrolovatelné změny spolu se záznamem historie změn.
e) *Vystopovatelnost* (traceability). U specifikací požadavků a obecně při všech důležitých rozhodnutích má být vždy dostatek informací umožňujících určit důvody učiněných rozhodnutí, i odmítnutých řešení a voleb. Tyto důvody mohou být obsaženy v jiných dokumentech, na které však musí být v daném dokumentu odkaz.
f) *Jednotná struktura* (konzistentnost). Dokument by měl udržovat jednotnost termínů, obsahovat rejstřík a všechny důležité odkazy, dodržovat dohodnutou strukturu. Neměl by být v rozporu s jinými dokumenty.
g) *Jednoznačnost*. Dokument má být formulován tak, aby nepřipouštěl nejasnosti a dvojí výklad. Tento požadavek je poněkud v rozporu s požadavkem srozumitelnosti. Někdy je nutné volit kompromis nebo k danému tématu udržovat verze dvě: intuitivně srozumitelnou a přesnou. Obě řešení mají svá úskalí. Požadavek jednoznačnosti je závažnější pro konečné verze dokumentů a pro dokumenty, které jsou určeny pro vývojáře.
h) *Použitelnost* po celou dobu, kdy je potřeba. Dokument musí sloužit při vývoji i při údržbě. Během údržby ovšem lze např. funkce systému ověřit na fungujícím systému a nemusí být nutné je přesně popisovat. Jako jazyk dokumentů se u specifikace požadavků osvědčuje spíše jazyk odborných článků. Silně formalizovaný popis typu algebraických specifikací lze využívat u základního softwaru, jako jsou komunikační protokoly, operační systémy atd.
i) *Dostupnost*. Dokumentace má být lehce dostupná všem, kteří ji potřebují.
j) *Aktuálnost*. Dokumenty odpovídají aktuálnímu stavu projektu.

Větší projekt vyžaduje podrobnější dokumentaci. Pro stavbu kůlny potřebujeme také jednodušší dokumentaci než při projektování mrakodrapu. V softwaru se tento fakt nebere často v úvahu. Není výjimkou, že firma řešící dosud jednoduché projekty převezme úkol mnohonásobně větší, než bylo dosud obvyklé, a řeší ho zavedenými metodami. To je velmi riskantní. Požadavky na dokumentaci nepřímo vymezují i metody realizace projektu. U malých firem, které bývají založeny na individuálních kvalitách pracovníků, nebývá požadavek dokumentování populární a může vést až k odchodu pracovníků. Optimální je proto postupné zavádění norem dokumentace a vývoje. To je možné jen tehdy, roste-li velikost zakázek postupně, bez přílišných skoků.

18

Procesní pohled na tvorbu softwaru

Vývoj softwaru je proces, pro jehož modelování je vhodné používat specifické nástroje. Proces tvorby softwaru zahrnuje (viz sborník Garg, Jazayeri, 1995) aspekty plánování prací a integrace a koordinace různých aktivit.

Síť činností, jejich vazeb a podmínek jejich provedení nazveme procesem vývoje softwaru. Softwarové procesy vycházejí z postupů tvorby softwaru popsaných v kap. 7. Softwarový proces je moderní pojem zobecňující klasický pojem životního cyklu softwaru. Model tvorby softwaru, procesní model, zahrnuje prostředky řízení projektů a jejich integrace s vysloveně softwarovými aktivitami, jako je správa konfigurace, sběr a vyhodnocování softwarových metrik atd. Tvorba procesních modelů je specifická činnost, pro kterou se používá název „process engineering“. Tento termín se překládá jako procesní inženýrství, zůstaneme však raději u anglického termínu.

18.1 Softwarové procesy. Základní pojmy

Model procesu (obr. 18.1)je chápán jako generické schéma vývoje celé třídy softwarových produktů, z něhož se zadáním parametrů a případnou editací (instanciace) vytváří procesní model vývoje konkrétního produktu.

Rostoucí složitost vývoje moderního softwaru si vynutila chápání tvorby procesních modelů jako činnosti, jejíž principy a obsah jen do jisté míry závisí na vlastnostech konkrétního produktu. Jinými slovy: architektura a paradigmata procesu jsou do jisté míry nezávislé na architektuře a jiných vlastnostech produktu. Daný produkt lze realizovat podle různých procesních modelů. V dalším budeme používat terminologii z přehledového článku P. H. Feier, W. S. Humprey, Software Process Development and Enhancement, Concepts and Definitions, ve výše uvedeném sborníku (Garg, Jazayeri, 1996).

Softwarový proces je síť kroků nutných k dosažení stanoveného cíle – vytvoření nebo zdokonalení softwaru či customizace softwaru.

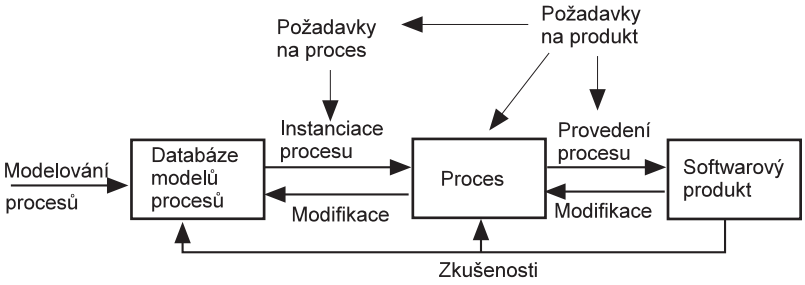
Krok procesu je z hlediska procesního nedělitelná akce procesu, která není dále strukturována. Příklad: Kompilace programové jednotky.

Prvek procesu je jednotka „zapouzdření“ (encapsulation) několika kroků procesu nebo prvků procesu. Příklad: Návrh tvořený kroky předběžný návrh a podrobný návrh.

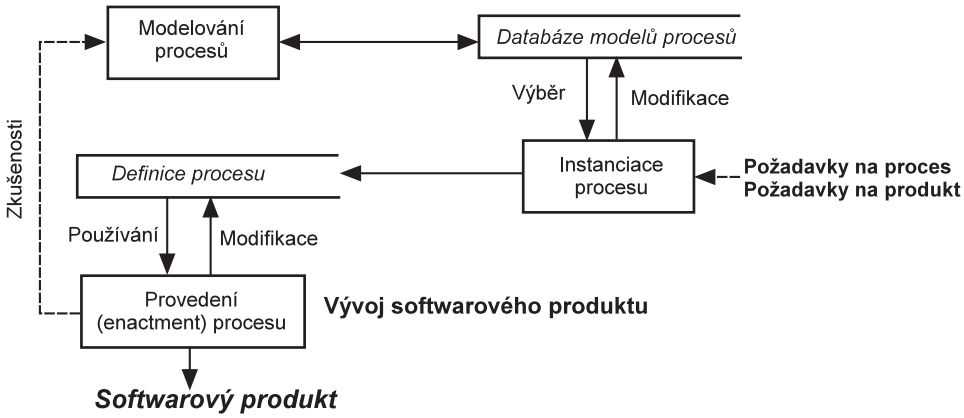
Agent – aktivní entita, obvykle člověk nebo počítač nebo softwarový systém, provádějící daný krok procesu.

Zdroje – prostředky nutné pro provedení určitého prvku procesu.

Process Engineering zahrnuje následující činnosti:



a) konceptuální schéma



b) schéma ve tvaru diagramu toků dat

Obr. 18.1: Operačně orientovaný pohled na model softwarového procesu a jeho použití.

Reprezentace procesního modelu. Použití částečně formalizovaných metod popisu a definice procesu, např. diagramů.

Analýza procesu. Po vytvoření modelu se provádí analýza, zda model vyhovuje určitým kritériím (nadbytečnost kroků, úplnost, chybné řazení kroků atd.).

Instanciace procesu. Abstraktní model je přizpůsoben pro vývoj konkrétního produktu specifikací výstupů kroků, zdrojů a agentů. Při tom se berou do úvahy technická organizační omezení a požadavky na vlastnosti produktu.

Provedení procesu (enactment). Provedení procesu může být úkolem lidí či softwarových nástrojů, nebo obou. Výsledkem provedení procesu je softwarový produkt.

Omezení provedení procesu. Proces a jeho kroky musí obvykle splňovat řadu podmínek. Příkladem jsou podmínky a povinnosti autorizace a dodržování standardů.

18.2 Životní cyklus softwarového procesu

Životní cyklus softwarového procesu je tvořen následujícími etapami.

1. *Analýza požadavků na softwarový proces.* Při této činnosti se na základě vlastností cílového produktu a prostředí, ve kterém bude vyvíjen, vlastností týmu a prostředků, které jsou k dispozici, a podmínek smlouvy se zákazníkem zvolí základní vlastnosti modelu procesu, např. inkrementální model s určitým počtem kroků.
2. *Vývoj modelu.* Cílem vývoje modelu je vytvořit proveditelný softwarový proces. Vývoj modelu procesu obvykle zahrnuje plán tvorby modelu, volbu architektury procesu, návrh modelu, instanciaci a validaci modelu provedením inspekcí, případně ověřením na pilotním projektu. Při tom se využívají prvky dříve vytvořených modelů, často stačí instanciace existujícího modelu.
3. *Přizpůsobení (tayloring).* Adaptace modelu pro konkrétní projekt zpřesněním významu jednotlivých kroků a doplněním informací.
4. *Plánování.* Stanovení termínů provedení jednotlivých kroků procesu.
5. *Instanciace.* Doplnění údajů o agentech (kdo co kde provede) a zdrojích (co je k provedení třeba). U velkých projektů se připouští i instanciace počátečních kroků procesu v situaci, kdy definice celého procesu ještě není dokončena.
6. *Evoluce.* Softwarové procesy jsou navrhovány jako schéma či plán budoucích činností. V průběhu provádění procesu dochází obvykle ke změnám v důsledku nově vzniklých nebo nově zjištěných skutečností. To může ovlivnit definici softwarového procesu, případně i model procesu.
7. *Provedení (enactment).* Podle modelu, který nyní slouží jako plán činnosti, se uskuteční vývoj softwaru. Při tom se připouštějí úpravy modelu při výskytu neočekávaných skutečností.

18.3 Provádění softwarového procesu

Softwarový proces lze chápat jako síť spolupracujících a na sebe navazujících aktivit pracujících synchronně (aktivita čeká na dokončení jí vyvolané aktivity) nebo asynchronně (ostatní případy). Při provádění procesu lze využívat jiný softwarový proces jako proces kontrolní. Je výhodné provádění procesu monitorovat. To lze zajistit sběrem dat o aktivitách procesu a vytvořením informačního systému pro analýzu záznamů aktivit procesů. Záznamy aktivit procesu obvykle obsahují:

- časový údaj,
- identifikaci kroku procesu,
- indikaci začátek/ukončení,
- parametry předávané při zahájení práce kroku,
- doplňující informace.

Pro analýzu komunikace mezi kroky procesu je výhodné využívat následující data:

- časový údaj,
- identifikátor komunikační akce,
- odesílatel,

- adresát,
- údaje o obsahu.

Výsledky analýzy takto získaných dat se využívají pro zlepšení modelu procesu. Explicitní součástí kroku procesu mají být pravidla a akce autorizace a schvalování, včetně pravidel předávání pravomocí. Tyto akce je žádoucí chápat jako standardní krok procesu a zaznamenávat je. Záznamy o průběhu procesu se využívají nejen k analýze průběhu procesu, ale mají také podstatný význam pro kontrolu plnění smluvních podmínek.

Při provádění softwarového procesu může docházet k situacím, které se podobají situacím při provádění softwaru. Může tedy docházet k incidentům nebo selháním. Součástí procesu jsou pravidla, jak pokračovat při vzniku incidentu (recovery) a jak provést dočasné nebo trvalé změny v modelu procesu a jakým způsobem jsou schvalovány změny ve struktuře procesu a jeho modelu.

18.4 Vlastnosti modelů softwarových procesů

Při hodnocení softwarových procesů se sledují následující kvalitativní ukazatele:

1. *Věrnost (fidelity)*. Vlastnost vyjadřující, do jaké míry se provádí to, co je stanoveno v modelu procesu.
2. *Vhodnost (fitness)*. Vlastnost vyjadřující, do jaké míry lze při přesném provádění procesu s rozumným úsilím dosáhnout plánovaného cíle. Jinými slovy do jaké míry zaručuje provedení procesu dosažení cílů projektu. Důvody malé vhodnosti mohou být různé – např. nepraktická doporučení nebo nedostatečná kvalifikace agentů.
3. *Přesnost.* Vyjadřuje správnost, úplnost a jednoznačnost modelu.
4. *Redundance.* Vlastnost vyjadřující počet takových kroků, které lze vynechat nebo nahradit jinými kroky. Redundantní kroky se používají pro definování alternativních postupů.
5. *Škálovatelnost.* Vlastnost vyjadřující rozsah velikosti projektů, pro něž je daný model procesu použitelný.
6. *Udržovatelnost.* Vlastnost vyjadřující, jak snadno lze model a jeho instance modifikovat.

Poněvadž je model softwarového procesu modelem paralelně pracujících aktivit, musí mít vlastnosti známé z teorie operačních systémů. Jsou to zejména:

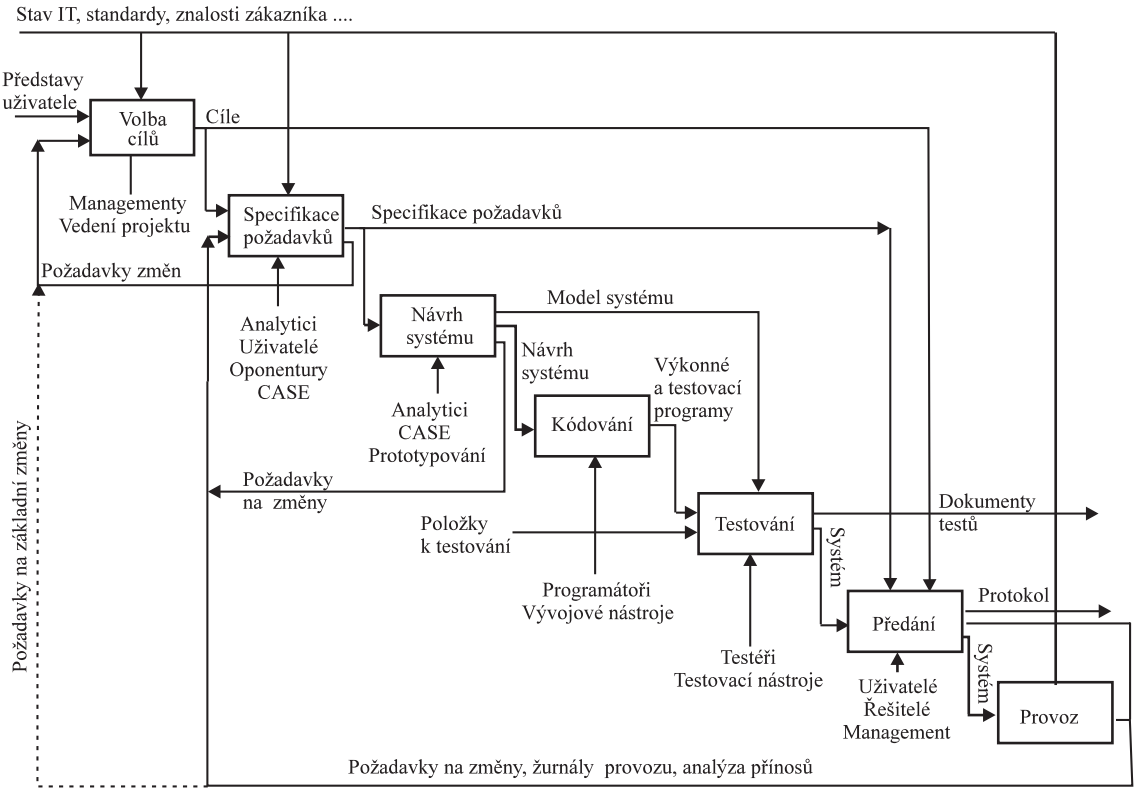
a) *Životnost.* Tato vlastnost vyjadřuje, že při provádění nedojde k uváznutí (deadlock), tj. že nedojde k situaci, kdy proces nemůže pokračovat a přitom není řádně ukončen.
b) *Robustnost.* Vlastnost vyjadřující stupeň ochrany před neautorizovanými změnami a stability při vzniku neočekávaných situací.
c) *Stabilita*. Stupeň ochrany vůči nesprávným změnám a celkové spolehlivosti. Typickým příkladem je zvyšování stability izolováním změn instance od změn modelu.
d) *Interakce s okolím.* Stupeň autonomie a rozsah interakce s jinými procesy.

Jednou z hlavních výhod modelování softwarových procesů je možnost formalizovaného zápisu průběhu řešení projektů. To umožňuje vytvoření databáze použitelné pro analýzu zkušeností. Databázi lze znovu využívat při vývoji nových procesních modelů a procesů.

18.5 Metody modelování procesů

Softwarové procesy jsou modelovány grafickými prostředky (diagramaticky) nebo jazykově. Diagramatické metody jsou inspirovány metodami a diagramy vyvinutými pro zobrazování softwarových produktů. Používají se mj. modifikace metody SADT (Marca, McGovan, 1988, Král, Demner, 1991), různé modifikace diagramů

DIAGRAM AKCÍ A0: *Vývoj metodou vodopádu*



Obr. 18.2: Metoda vodopádu v notaci SADT.

pro popis systémů reálného času a modifikace Petriho sítí, sítě s rozhodovacími uzly atd. Metoda SADT při specifikaci softwarových procesů je široce používána v knize (Fairclough, 1996).

Na obr. 18.2 je uvedena aplikace SADT pro zobrazení aktivit životního cyklu softwaru. Poněvadž je na tomto obrázku zobrazen poměrně vysoký počet aktivit, bylo by výhodné některé aktivity spojit, např. aktivity Návrh systému, Kódování a Testování, a pro tuto novou souhrnnou aktivitu vytvořit samostatný diagram.

Procesní modely vycházejí z různých koncepcí. Často se vedle diagramů SADT používají diagramy Petriho sítí. Pokrok technik procesního modelování je velmi rychlý.

Pro popis softwarových procesů se také používají jazyky pro popis sítí spolupracujících aktivit (např. Ada) nebo jazyky deklarativního typu. Blíže praktickým aplikacím jsou jazyky prvého typu. V praxi se používají především diagramy s doprovodnými texty. Vývoj kvalitních modelů softwarových procesů je pracný a obtížně se kontroluje. Existuje tendence k integraci grafických diagramatických prostředků a nástrojů práce s texty

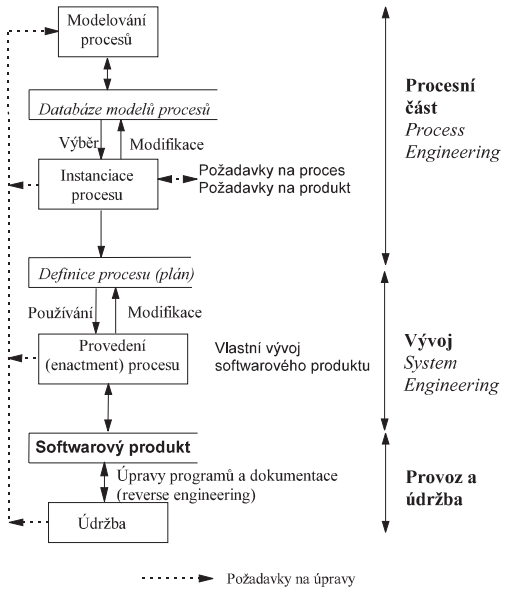
podobně jako je tomu u klasických CASE systémů. Podrobnosti lze najít ve sborníku (Garg, Jazayeri, 1995) a v metodě CMM (Carnegie Mellon U., 1995). V současné době je snaha o vytvoření integrovaného vývojového prostředí limitována tím, že jak po formální, tak intuitivní stránce není ještě dostatečně jasno, jaké konstrukce jsou pro modelování a práci se softwarovými procesy nejvýhodnější, např. není k dispozici vhodný jazyk pro definici Petriho sítí.

Většina doporučení týkajících se softwarových procesů je orientována na vývoj prostředků do značné míry inspirovaných zkušenostmi z informačních technologií. Vývoj softwaru je však technická činnost. Softwarové projekty jsou tedy projekty z oblasti techniky. Při modelování procesů lze tedy využívat prostředky používané při řízení projektů v jiných oblastech techniky, např.:

- systémy specifikace návaznosti prací, jejich rozvrhování a sledování (např. MS-Project nebo některé funkce Lotus Notes).
- systémy vhodné pro organizaci aktivit a kontroly pracovních toků (workflow systems).

Nadějná je tedy integrace nástrojů vhodných pro software, např. data o selhání a opravách, s obecnými nástroji řízení projektů a pracovních toků. Integrace však není právě jednoduchou záležitostí. Je zatím publikováno málo zkušeností s použitím prostředků řízení projektů obecného zaměření při řízení softwarových procesů.

Řízení softwarových procesů je integrální součástí metodiky SSADM4, metodiky CMM a je také součástí některých CASE systémů. Moderní CASE systémy směřují k tomu, aby se vývoj rozčlenil do etap navazujících činností a byl řízen z jednotného procesního pohledu. Pro takový přístup se používá termín procesně orientovaný (process centered). Procesně orientovaný má být tedy nejen samotný IS zaměřený na ucelené procesy zákazníka, ale i způsob, jak je prováděn jeho vývoj. Tvorba, modifikace, provádění a vyhodnocování softwarových procesů je náplní profese procesní inženýr. Význam této profese rychle roste.



Obr. 18.3: Návaznost činností při procesně orientovaném vývoji softwaru.

Zavádění procesního pohledu na vývoj softwaru je spojeno s potřebou měnit u softwarové firmy zavedená pravidla hry. Procesní orientace nutí ke spolupráci jednotlivá dříve spíše samostatná oddělení. To může být spolu s tím, že softwarový proces je prostředek řízení, pociťováno jako přílišné omezování pravomocí a snížení prestiže vedoucích oddělení či týmů a vyvolat nepříznivé reakce. Je to obdobná situace jako v případě BPR (business process restructuring) při vývoji IS. Procesní modely a jejich instanciace je proto třeba vytvářet ve spolupráci s vedoucími vývojových týmů a upravovat v průběhu řešení podle jejich připomínek. Modely procesů a jejich změny podléhají inspekcím a musí být schvalovány vývojáři, kteří jsou vlastně „uživateli“ modelů.

Procesní pohled na vývoj softwaru tedy chápe tvorbu softwaru jako postup s následujícími etapami:

a) Vývoj procesního modelu. Při tvorbě modelů lze využívat knihovny existujících modelů nebo jejich částí.
b) Instanciace procesního modelu. Výsledek je plán vývoje.
c) Vývoj softwarového produktu a jeho předání.
d) Údržba produktu.

Návaznost činností je zobrazena na obr. 18.3.

18.6 Stupně využívání softwarových procesů. Metodologie CMM

Využívání softwarových procesů může mít různou kvalitu, od řešení ad hoc k systematickému využívání a neustálé optimalizaci založené na statistické analýze dat. Kniha (Carnegie Mellon University, 1995) obsahuje podrobný návod, jak používat a zdokonalovat softwarové procesy. Zásady uvedené v knize jsou systematizovány do ucelené metodologie CMM (Capability Maturity Model). Cílem CMM je zvýšení spokojenosti uživatelů softwarových systémů, zlepšení kvality softwaru a omezení rizik spojených s vývojem softwaru zpřesněním odhadů při plánování a sledováním průběhu prací. CMM je i nástrojem zlepšování efektivnosti práce firmy a zmenšování rizik. CMM obsahuje postupy umožňující snižovat náklady, zkracovat termíny a eliminovat rizika spojená s migrací pracovníků.

CMM hodnotí vyspělost (maturity) organizací podle stupně a kvality využívání softwarových procesů (SWP). CMM definuje klíčové prvky efektivního využívání SWP. CMM je založeno na více než desetiletých zkušenostech a výzkumu SWP a představuje aplikaci principů komplexního řízení kvality (Total Quality Control) na vývoj softwaru. CMM definuje pět úrovní využívání (maturity level) SWP.

1. *Počáteční úroveň (initial level).* SWP existují většinou jen v neformální formě a definují se případ od případu od počátku. Nejsou zavedena pevná pravidla plánování a řízení projektů. Výsledky vývoje závisí spíše na kvalitě jednotlivců než na kvalitě organizace práce. Zkušenosti se na celopodnikové úrovni de facto nevyužívají, podnik jen v omezené míře zdokonaluje svou práci jako celek. Pokud nějaký pracovník z firmy odejde, jsou jeho zkušenosti pro firmu nenávratně ztraceny.
2. *Úroveň zajišťující opakovatelnost (repeated level).* Ve firmě jsou zavedena pravidla pro řízení projektů. Plánování a řízení projektů je založeno na zkušenostech s podobnými projekty, ale postupy se mohou případ od případu lišit. Řídí se však jednotnými zásadami platnými pro celou firmu. SWP nejsou standardizovány. Plány realizace jsou však realistické v důsledku využívaní předchozích zkušeností a je přísně sledováno, zda se dodržují. Při odchylkách od plánu jsou včas uskutečňována nápravná opatření.
3. *Úroveň definovaných procesů (defined level).* SWP jsou v rámci firmy standardizovány. Standardizace zahrnuje jak procesy softwarově inženýrské včetně specifikace a analýzy požadavků, tak manažerské. Součástí norem jsou nástroje kontroly a zvyšování efektivnosti práce. Standardy jsou založeny na zkušenostech a na osvědčených softwarově inženýrských metodách a postupech včetně organizace a řízení prací, infrastruktury

týmové spolupráce a školení pracovníků. Součástí standardů jsou i procedury přizpůsobování SWP potřebám a zvláštnostem konkrétních projektů. Je zajištěna kontrola dodržování požadavků na funkce systému, nákladů a termínů. Využívání SWP je založeno na odborném zázemí a na znalostech pracovníků firmy o aktivitách, rolích a odpovědnostech v SWP a podporováno skupinou vývoje a podpory SWP. Znalosti pracovníků jsou rozvíjeny pravidelnými školeními.

4. *Úroveň řízených procesů (controled level)*. Jsou definovány metriky kvality jak pro vyvíjený software, tak pro používané SWP. Je vyvinut systém sběru, sledování a vyhodnocování metrik jednotným způsobem v rámci celé organizace využívající celopodnikový IS metrik. Vyhodnocování dat je schopno odlišit náhodné fluktuace od statisticky významných změn. Firma je schopna vyhodnocovat trendy a odhadnout hodnoty důležitých metrik, jako jsou termíny a náklady. Je schopna odhadnout i přesnost odhadů stanovením kontingenčních intervalů, tj. mezí, do nichž s velkou pravděpodpobností padne odhadovaná hodnota.
5. *Úroveň optimalizovaných procesů (optimized level)*. Jsou zavedeny procedury neustálého vylepšování SWP. Je vytvořen tým hodnotící kvalitu procesů a navrhující jejich vylepšování včetně zavádění nejnovějších metod, postupů a nástrojů. Tým analyzuje příčiny úspěchů i neúspěchů a zobecňuje získané poznatky. Odlišuje při tom náhodné od zákonitého. Na základě analýz modifikuje používané SWP. Zlepšení se realizují formou dílčích změn SWP i jako zásadní inovace využívající nové metodologie a technologie.

CMM doporučuje, aby organizace procházela při využívání SWP postupně jednotlivými úrovněmi. Nedoporučuje se zavádět činnosti vyšších úrovní dříve, než jsou zavedeny a zvládnuty *všechny* činnosti a opatření nižších úrovní. Není např. vhodné vytvořit tým pro SWP (úroveň 3) na úrovni 2. Totéž platí pro standardizaci analýzy požadavků.

CMM doporučuje následující základní činnosti na jednotlivých úrovních:
Úroveň 2.

- Řízení a kontrola specifikace požadavků.
- Plánování na základě SWP.
- Dohled na SWP a jejich archivace.
- Řízení subkontraktů.
- Zajišťování kvality.
- Řízení konfigurace.

Úroveň 3.

- Koordinace a standardizace SWP v rámci organizace.
- Standardizace prací všech etap vývoje SW.
- Programy školení.
- Integrované řízení vývoje SW a SWP.
- Podpora týmové práce a spolupráce mezi týmy.
- Audity.
- Práce týmu podpory SWP.

Úroveň 4.

- Definice metrik SWP a systém jejich využívání.
- Kvantitativní metody řízení a plánování využívající metody statistické analýzy dat.
- Komplexní metody řízení kvality.

Úroveň 5.

- Prevence závad.

- Řízení postupných změn metod a praktik.
- Optimalizace softwarových procesů.
- Stálý tým analýzy softwarových procesů a jejich rozvoje.

Splnění požadavků úrovní 4 a 5 je v plné míře možné jen u velkých organizací, neboť jen ty mají k dispozici dostatek dat potřebných pro odlišení náhodných fluktuací od podstatných změn. Řada doporučení těchto úrovní je však do značné míry použitelná i u poměrně malých firem.

19

CASE

Vznik metodik a diagramatických značení při vývoji softwaru vedl k přirozenému požadavku automatizace vývoje softwaru s použitím počítačů. Odpověď na tento požadavek přišla ve formě nástrojů známých pod jménem CASE (Computer Aided Software Engineering). V současné době dodává CASE nástroje řada firem pro strukturované i pro objektově orientované metody vývoje. Trh s CASE systémy se rychle rozvíjí. Řada CASE nástrojů, zvláště nástrojů nižší úrovně, je integrována do moderních prostředí pro vývoj softwaru. Silné firmy integrují do svých vývojových prostředí nástroje dříve typické pro CASE systémy. Příkladem může být prostředek Oracle CASE.

CASE nástroje vyžadují přes svou zdánlivou jednoduchost a intuitivní zřejmost značně vysokou profesionální úroveň u těch, kteří je chtějí používat. Dalším předpokladem úspěchu použití CASE je vhodnost metod, na kterých je daný CASE systém založen, pro daný účel a také schopnost řešitelů přijmout tyto metody za své. Vývoj trhu s CASE nástroji – obrat řádu miliard USD v roce 1995 – a jejich cena a počet dodavatelů CASE systémů jasně ukazuje, že je o CASE nástroje zájem a že se osvědčují.

19.1 Druhy CASE systémů

CASE systémy se používají při specifikaci požadavků, návrhu, kódování a údržbě. Nástroje používané v těchto etapách se dosti liší a je obvyklé, že určité CASE nástroje pokrývají jen některé z těchto činností. Hranice mezi nástroji CASE a integrovanými vývojovými prostředími se postupně stírají. Z hlediska životního cyklu softwaru se nástroje CASE dělí do následujících skupin.

a) *Horní (upper) CASE* – CASE systémy používané při formulaci cílů projektu, analýze a specifikaci požadavků. Sem patří i nástroje pro plánování a vedení projektů, procesní inženýrství (kap. 18). Hlavním úkolem horního CASE je analýza organizace, v níž se má IS používat, zobrazení procesů v organizaci, definice klíčových informačních toků a prostředky dokumentování skutečností zjištěných při specifikaci požadavků.
   Použití: Specifikace cílů, počáteční fáze specifikace požadavků, řízení projektu.
   Hlavní cíl: Porozumění a specifikace systému jako celku.
   Hlavní nástroje:
   - diagramy toku dat a jejich varianty, např. SADT;
   - ER diagramy bez podrobné specifikace všech atributů;
   - prostředky pro řízení projektů (process engineering) a sledování ekonomických skutečností;

- dokumentografické systémy;
- popis základních vlastností systému prostředky objektově orientovaného modelování.

Metody horního CASE se částečně věcně i používanými prostředky překrývají s metodami a nástroji konzultačních firem. Metodika a prostředky CASE jsou však více zaměřeny na zpřesňování specifikace a celkového návrhu systému. Diagramy toků dat a ER diagramy jsou v horním CASE používány jako prostředek modelování situace u zákazníka a intuitivního popisu jeho požadavků.

b) *Střední (middle) CASE*. Nástroje střední úrovně zahrnují prostředky vhodné pro podrobnou specifikaci požadavků a návrh systému. Tato třída CASE je nejúspěšnější, neboť pokrývá činnosti, které nejsou předmětem činnosti konzultačních firem a firem zabývajících se tvorbou modelů podniků a řízením projektů a ani nejsou dosud ve větší míře integrovány do moderních prostředí pro vývoj softwaru.
Použití: Podrobné specifikace požadavků, návrh systému, dokumentace a vizualizace systému.
Podpora změn návrhu a lepší kontrola vazeb mezi návrhem systému a specifikacemi požadavků.
Obsahem středního CASE jsou metody a nástroje popisované v předchozích kapitolách, především v kap. 12.
Nástroje střední úrovně zahrnují především následující prostředky:
   (a) Diagramy toků dat včetně možnosti podrobnějšího popisu procesů, datových úložišť a datových toků.
   (b) ER diagramy s možností detailní specifikace atributů, matice V/C/U/Z (kap. 16), atd.
   (c) Pro objektovou metodologii diagramy OO technik – diagramy tříd a jejich vztahů, diagramy instancí, přechodové diagramy atd.
   (d) Systém správy dokumentů a správy konfigurace.
   (e) Systémy vyhodnocování metrik souvisejících s návrhem systému a specifikacemi požadavků.
   (f) Vývoj prototypů, většinou potěmkinovských, návrh rozhraní, generátory obrazovek a sestav.
   (g) Generátory definic dat. Někdy se generují pouze kostry definic, které se doplňují ručně.

   Hlavní cíle: Formalizace specifikace a návrhu s cílem usnadnění změn a snazší komunikace se zákazníky. Vytvoření modelů usnadňujících případně umožňujících generaci návrhu.
   Střední CASE jsou jádrem komerčně dodávaných CASE systémů. Úspěšně se používají především u větších firem.

c) *Dolní CASE* (Lower CASE). Dolní CASE obsahují nástroje na podporu kódování, testování a údržby a reverzního inženýrství. Nejcennější jsou následující nástroje.
   (a) Generátory kódu. Generátory kódu využívají výstupy CASE střední úrovně jako dat používaných při generaci kódu. Kód je generován s využitím ER diagramů, diagramů toků dat a vzorů typických programových obratů. Generátory kódu vytvoří obvykle polotovar, který pokrývá až 3/4 výsledného kódu a který je upravován programátorem do cílového stavu. Výhodou je, že zmíněný polotovar je většinou úplný z hlediska celkové logiky, doplňují se „detaily“. Generátory kódu se osvědčují především tehdy, kdy je možné převzít data pro generaci kódu mezi aplikacemi a kdy se opakovaně řeší podobné problémy. Moderní CASE systémy využívají vizuální metody generace kódu. To se obzvlášť osvědčuje při generaci SQL příkazů. Pokrok informačních technologií umožňuje postupné snižování rozsahu „ručních zásahů“ do generovaného kódu.
   (b) Prostředky reverse engineering. Skupina nástrojů umožňujících rekonstrukci dokumentace z existujícího softwaru nebo alespoň detekci míst, kde již existující dokumentace neodpovídá aktuálnímu stavu.
   (c) Prostředky sledování a vyhodnocování metrik kódu.
   (d) Prostředky a nástroje plánování a zajišťování kvality softwaru
      - sběr informací o průběhu testování, řízení testování,

- sběr a vyhodnocování dat, inspekcí a výsledků testů,
- pravidla přijímání prvků konfigurace,
- podpora plánování opatření na zajišťování kvality.

(e) Správa konfigurace (configuration management). Kontrola přebírání prvků konfigurace (relativně samostatných částí softwaru) a zajišťování toho, aby určitá aplikace/systém obsahovala správné prvky.

(f) Prostředky sledování a vyhodnocování práce systému.

Funkce, které nabízejí dolní CASE se do značné míry překrývají s funkcemi obecných vývojových prostředí. Standardní vývojová prostředí nepokrývají ty funkce generátorů kódu, které využívají výstupy středních CASE a některé aspekty zpětného inženýrství. Moderní CASE systémy se pokoušejí integrovat nástroje všech tří úrovní a obsahují i procesní část (obr. 18.1 a obr. 18.3). Činnosti při vlastním vývoji softwaru se někdy označují jako system engineering zatímco činnosti procesně orientované se označují jako process engineering.

19.2 Zkušenosti s CASE

CASE systémy přinášejí pozitivní efekty pouze tehdy, jsou-li metody, na kterých je příslušný CASE založen, blízké praxi týmů, které budou CASE používat. Je jen malá naděje, že CASE pomůže podstatně zlepšit kvalitu řízení softwarových projektů ve firmě, kde nebyla zavedena, byť nepsaná, rozumná pravidla vedení projektu. CASE podobně jako IS nemůže bez dalších opatření zavést pořádek. Je nebezpečné použít CASE založený na metodách, které nebyly alespoň částečně zvládnuty na jednoduchých příkladech.

Dalším úskalím může být podobně jako u IS apriorní odpor k novotám nebo přehnané naděje. CASE by neměl být poprvé použit v plné šíři v poměrně novém komplikovaném softwarovém projektu. Podle zkušeností autora se osvědčuje začít od grafických prostředků: ER diagramů a diagramů toku dat, u objektově orientovaných metod od diagramů tříd, instancí a přechodových diagramů, a postupně zvládat další prostředky ať už během daného projektu, nebo u dalších projektů.

Diagramy toků dat a ER diagramy jsou intuitivně srozumitelné i zákazníkům. To je podstatná, často i hlavní, výhoda CASE systémů. Není výjimkou, že je zákazník schopen po krátkém zaškolení najít nedostatky v diagramech bez potřeby hlubšího porozumění všech souvislostí. Velice se osvědčují návrhy obrazovek. Diagramy podstatně ulehčují i komunikaci v týmu a umožňují zpřesnit analýzu systému. To je pravděpodobně hlavní příčinou úspěchu středních CASE.

Dolní CASE nejsou již tak jednoznačně úspěšné. V oblasti generace kódu existuje řada konkurenčních nástrojů integrovaných do moderních vývojových prostředí. Výhodou generátorů kódu v CASE je to, že kód přesně odpovídá dokumentaci. Generátory kódu jsou většinou založeny na definici „polotovarů“ (můstků), které jsou pak využívány při generaci kódu. Tvorba můstků je poměrně pracná a vyplatí se pouze při mnohonásobné generaci podobných systémů. To je běžnější u větších softwarových firem. Při zavádění CASE je vhodné začít od diagramů a očekávat přínos hlavně při analýze a při specifikaci požadavků společně s uživatelem.

U větších firem zabývajících se vývojem softwaru se stává použití CASE v podstatě nutností. CASE nástroje mají zatím bohužel nedostatečné vazby na normu ISO 9000–3 a jiné normy a nepříliš rozvinutou podporu sběru a analýzy softwarových metrik. Moderní vývojové systémy, např. Borland C++ Design Tools, propojují prvky nástrojů střední úrovně přímo s psaním programů.

19.3 Volba CASE nástrojů

Při volbě CASE a moderních vývojových prostředí je žádoucí uplatnit následující kritéria.

a) Musí být vytvořeno vědomí potřeby formalizace metod vývoje a řízení prací.
b) Pro použití CASE je výhodné vytvořit předpoklady používáním metodologie vývoje blízké metodám, na nichž je založen CASE nebo vývojový systém.
c) CASE by měl obsahovat prvky procesního pohledu (process engineering) a měl by zahrnovat všechny nástroje střední úrovně a hlavní prvky horní úrovně.
d) Je výhodné, aby výstupy CASE umožňovaly spolupráci s vývojovými a grafickými prostředími, např. PowerBuilder, a různými databázovými systémy. Většina CASE je zaměřena na určitá vývojová prostředí, která bychom mohli nazvat „domovskými“. Pro tyto systémy bývá využití CASE nástroje snazší.
e) CASE systém by měl podporovat moderní architektury IS, především třívrstvou architekturu v kombinaci s možnostmi architektury klient – server, případně architektury klient - aplikační servery – datový server.
f) CASE systém by měl splňovat obecné podmínky pro moderní softwarové produkty: otevřenost, nezávislost na hardwaru a základním softwaru, kvalitní podpora ze strany dodavatele atd.

Volbu CASE nástrojů upravuje IEEE norma 1348–1995, IEEE Recommended Practice for the Adoption of Computer Aided Software Engineering (CASE) Tools. Tato norma uvádí několik desítek evaluačních kritérií pro CASE systémy. Kritéria zahrnují i podmínky platné pro každý softwarový systém, jako je snadnost zvládání, kvalita rozhraní atd. Zmiňme se stručně o některých dalších kritériích. Z hlediska celkové filozofie je důležitá oblast použitelnosti, podpora aktivit řízení projektu a také pro jak velké systémy je daný CASE nástroj určen.

U hardwaru a softwaru se vyhodnocuje, s čím může být CASE systém používán, jaký HW a SW a jaké metodologie, programovací jazyky a standardy podporuje. Důležitá je také kompatibilita s jinými nástroji. Těžiště CASE systémů je v modelování. Norma specifikuje požadavky na diagramy, prototypování, grafickou analýzu, vazby na formalizované specifikační jazyky, simulace a testování konzistence specifikací.

Pro implementaci se hodnotí možnosti syntaxí řízené editace, generace kódu ve více programovacích jazycích, analýza spolehlivosti, reverse engineering, restrukturalizace zdrojového kódu, analýza zdrojového kódu včetně výpočtu metrik a podpora ladění.

Pro testování se hodnotí prostředky definice testů, rozhraní na operátora testů, prostředky automatizace testů, regresní testování (opakování dříve provedených testů), analýza výsledků testů, analýzy výkonnosti, simulace prostředí a prostředky generace testových procedur.

U dokumentace se vyhodnocují prostředky editace textů, grafiky, editace podle formulářů, možnosti přípravy publikací (desk top publishing), podpora hypertextu, dodržování norem pro formu dokumentů a automatizace výběru dat do dokumentů a generace dokumentů.

CASE systém by měl poskytovat nástroje pro řízení konfigurace obsahující kontrolu přístupových práv a změn, prostředky uchovávání a analýzy posloupností změn, formování verzí, vyhodnocování stavu konfigurace a možnosti archivování. Podpora řízení projektu by měla obsahovat prostředky odhadu pracnosti a doby řešení, řízení aktivit a zdrojů, řízení testovacích procedur, řízení kvality a provádění změn. Žádoucí je propojení CASE s obecně použitelnými systémy řízení projektů a sledování prací (workflow).

CASE systémy sledují překotný vývoj metodik vývoje softwaru a proto rychle zastarávají. Nástroj starší než čtyři roky je obvykle morálně zastaralý. Investice do CASE nástrojů se proto vyplatí jen tehdy, jsou-li CASE nástroje systematicky a intenzivně využívány.

20

Softwarové normy

S rozvojem průmyslových rysů softwaru roste význam softwarových norem. Softwarové normy rychle zastarávají. Proto je u norem zaměřených na softwarové inženýrství stanovena zásada, že mají být modernizovány, tj. potvrzeny nebo přepracovány, každých pět let. Softwarové normy nejsou závazné ze zákona. Jejich dodržování je věcí dobrovolného rozhodnutí organizací, které software vytvářejí. Dodržování norem může být také smluvně vyžadováno odběratelem softwarových produktů. Autor normy neodpovídá za škody vzniklé v důsledku použití normy.

Právo deklarovat, že výrobek splňuje určitou normu, je vždy spojeno s možností právní odpovědnosti v případě, že se prokáže, že výrobek normě nevyhovuje. Právo označit, že výrobek vyhovuje normě, je u některých norem vázáno na schvalovací řízení (audit, atest). Podle normy ISO 9000–3 je právo označit software jako výrobek vyhovující normě vázáno na atestační řízení prováděné organizací vlastnící příslušnou akreditaci.

20.1 Tvorba norem

Softwarové normy standardizují různé aspekty softwaru. Klasickým příkladem jsou normy programovacích jazyků nebo normy síťových protokolů. V současné době probíhá intenzívní vývoj softwarově inženýrských norem. Normy mohou být různého typu a různé úrovně. Vždy však jsou výsledkem nejrůznějších kompromisů a do jisté míry fixují stav z doby svého vzniku.

a) *Firemní normy*. Normy stanovené výrobcem pro vlastní výrobky nebo interní pravidla pro vývoj a dokumentaci. Firmy mohou takové normy nabídnout k obecnému použití. Pokud se používání normy rozšíří, stane se normou de facto.
b) *Normy de facto* jsou technická řešení, která se obecně rozšířila a jsou přijata podstatnou částí firem a uživatelů pracujících v určitém oboru. Příkladem de facto norem jsou řešení přijatá některou silnou firmou, např. Microsoftem. De facto norma má všechny podstatné atributy normy, není však dosud schválena oficiální institucí odpovídající za vývoj a údržbu norem.
c) *Návrh normy, normy profesních organizací, oborové normy.* Při vzniku potřeby vytvořit normu může být vytvořením normy pověřena nějaká organizace. V softwaru vytváří normy především americká profesní organizace IEEE. Významným zdrojem oborových norem je americké ministerstvo obrany (DoD). Oborové normy jsou respektovány méně než normy IEEE, které mají vysokou prestiž a obvykle jsou schváleny i jako státní normy USA a často jsou i základem mezinárodních (ISO) norem. Proces vytváření normy

začíná ustavením pracovní skupiny (working group, WG) s úkolem vytvořit normu. Návrh normy je pak obvykle předložen odborné veřejnosti k diskuzi a pak schválen dosti složitou schvalovací procedurou. Návrh normy může vycházet z vhodné de facto normy. Ta může být přijata bez věcných změn. I pak je třeba vypracovat úplnou dokumentaci umožňující veřejné uplatnění normy. Oborová norma tedy může být přijata normalizačním úřadem určitého státu jako státní norma. V USA schvaluje normy normalizační úřad ANSI, u nás Státní úřad pro normalizaci a měření.

d) *Státní normy.* Státní (national) normy vznikají schválením nebo adaptací de facto norem nebo oborových norem profesních organizací jako norem státních nebo jsou vyvíjeny od počátku. Ve všech případech předchází proceduře schválení normy jednání v pracovních skupinách a veřejná diskuze normy. Většina softwarově inženýrských norem prochází etapami norma de facto – oborová norma vypracovaná v IEEE (IEEE norma), státní norma USA (ANSI norma). IEEE normy a ANSI normy jsou ve státech mimo USA nejprve používány jako normy de facto a dodatečně schváleny, nebo jsou zahrnuty do mezinárodních (ISO) norem a ty pak schváleny jako normy národní v jednotlivých státech. Počet celosvětově používaných softwarových norem vzniklých mimo USA je velmi nízký.

e) *Mezinárodní (ISO) normy.* Mezinárodní normy vznikají na základě aktivit International Standard Organization. Proces schvalování i zde začíná jednáním v pracovních skupinách a zahrnuje veřejnou diskuzi. Členské státy ISO schvalují ISO normu jako svoji státní normu. To je spojeno s oficiálním překladem textu normy do národního jazyka. Schválení ISO normy jako národního standardu není povinné.

V softwaru je významná především norma zajištění kvality ISO 9000–3. Další důležité mezinárodní softwarově inženýrské normy jsou:

ISO 9126: Software Quality Characteristics and Metrics. Tato norma je v současné době revidována a výčet doporučovaných metrik je podstatně rozšiřován.

ISO 12119: Information techniques – Software Packets – Quality and Testing Requirements. Tato norma obsahuje velmi důležitou část Product Description obsahující výčet skutečností důležitých pro rozhodnutí produkt zakoupit.

ISO 12207: Software Life Cycle Processes. Tato norma shrnuje pravidla návrhu softwarových procesů (kap. 18).

Všechny tyto normy jsou přijaty jako ČSN a jsou k dispozici v Úřadu pro normalizaci a měření.

Připravuje se zajímavá norma ISO 14597 Information Technology – Software Product Evaluation. Tato norma obsahuje doporučení pro hodnocení kvality softwaru dodavatelem i odběratelem.

20.2 Přehled softwarově inženýrských norem IEEE/ANSI

Americké softwarově inženýrské (profesní) normy IEEE poměrně rychle reagují na vývoj. Většina z nich byla schválena jako federální US normy (ANSI). S normami IEEE jsou dobré zkušenosti. Výhodou i nevýhodou těchto norem je, že obvykle standardizují poměrně úzký obor. K r. 1993 existovalo 22 IEEE norem pro oblast softwarového inženýrství. Originály norem lze získat v nakladatelství IEEE Computer Society Press, New York.

Důležité jsou terminologické normy IEEE. V době vytváření tohoto textu byly terminologické normy shrnuty do slovníku IEEE Standard Computer Dictionary, Compilation of IEEE Standard Computer Dictionaries, 1992 Edition, IEEE Press, New York, ISBN 1–55937–079–3.

20.2 Přehled softwarově inženýrských norem IEEE/ANSI

Kolekce softwarově inženýrských norem IEEE vydaných do začátku r. 1994 byla uveřejněna v knize IEEE Standards Collection, Software Engineering, 1994 Edition, IEEE New York, ISBN 1–55937–442-X, viz (ANSI94, 1994). V polovině roku 1996 byly k dispozici následující softwarové normy IEEE.[1]

730–1989, ANSI, IEEE Standard for Software Quality Assurance Plans.

828–1990, ANSI, IEEE Standard for Configuration Management Plans.

829–1983, (1991), ANSI, IEEE Standard for Software Test Documentation.

830–1993, ANSI, IEEE Recomended Practice for Software Requirements Specifications.

982.1–1988, ANSI, IEEE Standard Dictionary of Measures to Produce Reliable Software.

982.2–1988, ANSI, IEEE Guide for the Use of IEEE Standard Dictionary of Measures to Produce Reliable Software.

990–1987, (1992), ANSI, IEEE Recomended Practice for Ada as a Program Design Language.

1002–1987, (1992), ANSI, IEEE Standard Taxonomy for Software Engineering Standards.

1008–1987, (1993), ANSI IEEE Standard for Software Unit Testing.

1012–1986, (1992) ANSI, IEEE Standard for Software Testing.

1016–1987, (1993), ANSI, IEEE Guide to Software Design Descriptions.

1028–1988, (1993), IEEE Standard for Software Rewievs and Audits.

1042–1987, (1993), ANSI, IEEE Guide to Software Configuration Management.

1044–1993, ANSI, IEEE Standard Classification for Software Anomalies. Tato norma specifikuje pojmy jako selhání, defekt (místo, které je nutno opravit), error (lidské pochybení) a další pojmy.

1044.1–1995, IEEE Guide to Classification for Software Anomalies.

1045–1992, ANSI, IEEE Standard for Software Productivity Metrics.

1058.1–1987, (1993), IEEE Standard for Software Project Management Plans.

1059–1993, ANSI, IEEE Guide for Software Verification and Validation Plans.

1061–1992, IEEE Standard for Software Quality Metrics Methodology.

1062–1993, ANSI, IEEE Recommended Practice for Software Aquisition.

1063–1987, (1993), ANSI, IEEE Standard for Software User Documentation.

1074–1991, (1995), ANSI, IEEE Standard for Developing Software Life Cycle Processes.

1209–1992, ANSI, IEEE Recomended Practice for the Evaluation and Selection of CASE Tools.

1219–1992, ANSI, IEEE Standard for Software Maintenance.

1228–1994, ANSI, IEEE Standard for Software Safety Plans.

1233–1996, IEEE Guide for Developing System Requirements.

1298–1992, Software Quality Management System, Part 1: Requirements (Australian Standard AS 3363.1–1991).

730.1–1995, IEEE Standard for Software Quality Assurance Plans.

1175–1991, IEEE Standard Reference Model for Computer System Tool Interconnections.

1220–1994, IEEE Trial Use Standard for Application and Management of the Systems Engineering.

1348–1995, IEEE Recommended Practice for the Adoption of Computer-Aided Software Engineering (CASE) Tools.

---

1. Letopočet v závorkách je rok poslední revize, pokud byla revize uskutečněna. Poznámka ANSI značí, že je příslušná norma přijata jako federální norma USA.

1420.1–1995, IEEE Standard for Information – Software Reuse Data Model for Reuse Library Interoperability: Basic Interoperability Data Model (BIDM).

20.3 Hlavní zásady softwarové normy ISO 9000–3

ISO 9000–3 je adaptace normy ISO 9001 pro software. Tato norma získává popularitu a začíná se aplikovat i v USA. Norma ISO 9001 má oficiální název: *Systémy kvality: Model zajišťování kvality při projekci/vývoji, výrobě, instalaci a servisu – 1994.*

Oficiální název normy ISO 9000–3 je *Návod na aplikaci normy ISO 9001 při vývoji, dodávkách a údržbě softwaru – 1991*. ISO 9000–3 je tedy mezinárodní norma zaměřená především na kvalitu softwaru. Norma byla přijata jako česká (ČSN) norma a její český překlad je k dispozici na Českém úřadu pro normalizaci a měření. V souladu s obecnými zásadami norem ISO 9000 až ISO 9004 je možné softwarovému produktu přiznat certifikát, že vyhovuje normě ISO 9000–3. Certifikát se softwaru udílí, je-li ověřeno, že při jeho návrhu, vývoji, prodeji a údržbě jsou dodržovány zásady a prováděny činnosti v souladu s normou ISO 9000–3. Certifikát vystavuje autorizovaná firma, která musí projít složitým akreditačním řízením. Certifikát se vystavuje na základě dokumentů předepsaných normou a na základě inspekcí na místě.

Přiznání certifikace podle ISO 9000–3 je velmi pracná a časově náročná záležitost. Doba řízení k přiznání certifikace nebývá kratší než rok. Na první pokus získá certifikát jen asi 20 % žadatelů. Certifikace zaručuje pouze to, že byly při návrhu, vývoji, prodeji a údržbě prováděny činnosti v souladu s normou ISO 9000–3. Je ovšem známo ze zkušenosti, že takový produkt je obvykle kvalitní. Certifikát smí vystavovat pouze firma vlastnící příslušnou akreditaci.

20.4 Přehled normy ISO 9000–3

V této části uvedeme zkráceně hlavní doporučení normy ISO 9000–3. Pro hlubší porozumění lze použít např. knihu (Kehoe, Jarvis, 1996) ve které jsou doporučení normy ISO 9000–3 doplněna rozsáhlými komentáři.

ISO 9000–3 stanovuje pravidla umožňující aplikování normy řízení kvality ISO 9001 v organizacích vyvíjejících, dodávajících a udržujících software. Norma je koncipována jako návod pro případ, kdy obchodní smlouva mezi partnery vyžaduje prokázání schopnosti dodavatele vyvíjet, dodávat a udržovat softwarový produkt.

Softwarový produkt je podle normy úplný soubor programů, pravidel práce s nimi, dokumentace a dat nutných k oživení a provozu systému.

Verifikace softwaru je podle normy proces vyhodnocování výstupů určité fáze vývoje s cílem ověření správnosti a konzistence vzhledem k produktům a standardům, které jsou „vstupem“ této fáze. Hlavním nástrojem verifikace jsou oponentury.

Validace je procesem vyhodnocování softwaru s cílem zajištění toho, že software vyhovuje požadavkům. Základním nástrojem validace je testování.

Evaluace je vyhodnocovací procedura mající za cíl zhodnotit, do jaké míry splňuje nějaký produkt nebo organizace určitá kritéria, např. hodnocení, zda nějaký výrobek splňuje kritéria použitelnosti.

20.4.1 Zásady řízení kvality softwaru

1. *Povinnosti a odpovědnosti managementu.*
   Management projektu musí především zajistit, aby pracovníci dodavatele i zákazníka znali své úkoly a odpovědnosti. Povinností managementů obou stran je zajištění efektivní komunikace mezi dodavatelem i odběratelem. Management dodavatele proto musí
   - vytvořit podmínky a organizaci s jasně vymezenými rolemi a odpovědnostmi,
   - zajistit efektivní kontrolu plnění úkolů (obsah, přesnost, úplnost),
   - zajistit dodržování dohodnutých postupů,
   - účastnit se kontroly a revizí postupů a praktik s cílem zajistit jejich vhodnost a efektivnost.

   Management zákazníka
   - zajišťuje, aby dodavatel měl veškeré informace potřebné pro plnění smluvních závazků,
   - určuje zástupce zákazníka odpovědného za spolupráci na realizaci.

   Management dodavatele i odběratele musí zajistit, aby byly společné revize a přezkoumání (review) prováděny pravidelně a na základě dohodnutých pravidel.
2. *Systém zajišťování kvality.*
   Management musí nejprve formulovat cíle a zajistit podmínky pro to, aby plánované cíle byly dosaženy co nejefektivnějším způsobem. K tomu je nutno:
   - vymezit procesy a postupy,
   - na základě vymezených procesů a postupů vypracovat, kontrolovat a aktualizovat plány činností,
   - stanovit audity, vnitřní oponentury (review, inspekce) a testy pro zajištění kvality vytvářeného produktu a také pro řízení jeho vývoje,
   - stanovit, jak budou odstraňovány chyby detekované při revizích a testech.

   Norma stanovuje pravidla vnitřních auditů kvality a pravidla jejich dokumentování, včetně revizí a archivace dokumentů. Pro vnitřní audit jsou stanoveny podrobné postupy plánování auditů na základě důležitosti a stavu auditované činnosti. Musí být stanovena pravidla kontroly plnění závěrů auditu.

20.4.2 Systém řízení kvality v jednotlivých etapách životního cyklu

Norma nestanovuje, která varianta životního cyklu má být použita. Stanovuje však, které činnosti pro zajišťování kvality musí být provedeny. Životní cyklus standardizuje norma ISO 12207, která je přijata jako ČSN.

1. *Přezkoumání (revize, review) smlouvy.* Revize smlouvy má za cíl dosažení shody mezi dodavatelem a odběratelem o závazcích vyplývajících ze smlouvy a také shody v tom, jak budou řešeny případné budoucí problémy s plněním smlouvy. Cílem je, aby smluvní strany stejně chápaly
   - vymezení rozsahu smlouvy,
   - své organizační odpovědnosti,
   - rizika: termíny, rozpočet, omezení plynoucí ze závazků atd.,
   - vlastnická práva na produkt a vedlejší produkty.
2. *Specifikace požadavků na produkt ze strany zákazníka.*
   Podle normy obsahuje tento dokument požadavky na funkce a požadavky technické, např. požadavky na výkon, spolehlivost, zabezpečení atd. Dokument má také specifikovat vlastnosti rozhraní. Povinností obou stran je pracovat společně s cílem dosáhnout toho, aby specifikace požadavků byly úplné a kvantifikovatelné dříve, než začne vlastní vývoj produktu.

3. *Plán vývoje.*
   Plán vývoje stanovuje zdroje a termíny nutné pro zajištění dodávky produktu. Plán se sestavuje na základě požadavků zákazníka s ohledem na technické možnosti a postupy, které dodavatel použije při vývoji. Plán vývoje vymezuje
   - fáze vývoje,
   - vstupy a zdroje pro každou fázi,
   - pravidla sledování průběhu plnění,
   - metody a nástroje, které budou používány,
   - verifikační procedury (oponentury, audity, testování).

   Vstupy každé fáze musí být přesně specifikovány. Pro vstupy a výstupy by měly být k dispozici takové požadavky, které umožňují kvantifikovatelné testování použitelnosti, u výstupů se zaměřením na požadavky navazujících fází.
4. *Plánování jakosti.*
   Plánování jakosti má zajistit, aby byly prováděny aktivity verifikace a validace jakosti vyvíjeného produktu. Plány těchto aktivit mohou tvořit separátní dokument - plán zajištění jakosti – nebo mohou být zahrnuty do jiných plánů, jako je plán vývoje, plán testů a plán řízení konfigurace (configuration management plan). Tato část normy se do značné míry překrývá s jinými částmi normy. Z toho důvodu by měl být plán zajištění jakosti v podstatě chápán jako shrnutí pravidel a požadavků jiných aktivit, např. plánu testů.
5. *Návrh a implementace.*
   Návrh je základ implementace[2] softwarového produktu. Kvalita návrhu zásadním způsobem ovlivňuje kvalitu výsledného produktu. Návrh především ovlivňuje použitelnost (usability, kap. 12) produktu, náklady údržby a vylepšování produktu během života projektu. Norma zdůrazňuje význam následujících aktivit a nástrojů:
   - směrnice a pravidla návrhu,
   - metodologie návrhu,
   - metody používané při interním návrhu,
   - vazby a porovnání s návrhem podobných systémů u dodavatele.

   Norma stanovuje málo závazných pravidel pro fázi kódování. Doporučuje stanovení interních směrnic pro volby programovacích jazyků, jmen, tvaru programů a poznámek v programech. Všechny tyto směrnice by měly tvořit ucelenou metodologii vývoje zajišťující splnění požadavků odběratele.
   Výstupy všech etap vývoje mají být prověřovány prováděním revizí, s cílem prověřit a zajistit, aby konečný produkt vyhovoval požadavkům odběratele. Pod analýzou rozumí norma specifikaci požadavků a poslední redakci formulace cílů. Revize a inspekce mají rovněž prověřovat, zda jsou skutečně dodržována pravidla, směrnice a metody stanovené pro příslušnou etapu. Slouží zároveň pro ověření toho, zda byla provedena organizační opatření a použity postupy a metodologie zajišťování kvality.
6. *Testování a validace.*
   Průběh testování má být stanoven v plánu testů, který by měl zejména obsahovat:
   - testování funkcí a rozhraní, výkonu, uživatelského rozhraní; z organizačního hlediska testy částí, integrační testování, předávací testy, testy u uživatelů;

2. V české verzi normy se používá termín zavedení. Tento termín se nezdá být vhodný.

- seznam testových případů. U každého z testových případů se obvykle určuje číslo případu a jméno, jméno a číslo testované verze a komponenty, testovací procedury, očekávané výsledky, skutečné výsledky, hlášení o provedení;
- prostředí, v němž se testování provádí;
- zdroje nutné k přípravě testů a termíny včetně testových dat;
- zdroje a termíny potřebné k provedení testů;
- podmínky zahájení a ukončení testů;
- vnější podmínky a omezení;
- testovací nástroje;
- rizika (technická, rozpočtu a termínů).

Zdroje jsou personální (kdo bude testovat, kdo bude zajišťovat provoz), technické (hardware, software), organizační (vytvoření podmínek). Do zdrojů zahrnujeme i data. Výsledky testování musí být zaznamenávány a používány k následujícím účelům

- identifikace problémů zjištěných v testovaném produktu,
- identifikace oblastí, kde je třeba testy zopakovat,
- určení toho, zda proces testování vyhovuje.

Testování u uživatelů (field testing) musí být plánováno a provedeno ve spolupráci dodavatele a odběratele na základě přesně formulovaných podmínek formou blízkou dodatku ke smlouvě.

7. *Převzetí, předávací testy.*
   Převzetí je prováděno odběratelem. Cílem je prověření toho, zda produkt je akceptovatelný podle kritérií a postupů stanovených smlouvou. Předání má být provedeno na základě formalizovaného postupu písemně formulovaného s dostatečným předstihem před vlastním přebíráním systému. Plán převzetí systému by měl obsahovat termíny, zdroje, role a odpovědnosti pracovníků, pravidla pro přijetí a postupy řešení zjištěných problémů u dodavatele i odběratele. Obě strany mají při přebírání stejnou odpovědnost a musí úzce spolupracovat.
8. *Vytváření kopií, dodávka a instalace.*
   Vytváření kopií je povinností dodavatele. Instalace může vyžadovat spolupráci mezi dodavatelem a odběratelem. Rozsah spolupráce by měl být včas analyzován a dohodnut ve formě vhodného dokumentu. Plán instalace má kromě termínů obsahovat i personální zajištění, právo vstupu do míst potřebných pro instalaci, právo používat potřebná zařízení, systémy a samozřejmě právo testování zkušebního provozu. Instalace má být doložena psaným dokumentem schváleným oběma stranami.
9. *Údržba.*
   Údržba má z hlediska řízení kvality stejné vlastnosti jako vývoj. Analýza, návrh, implementace kódování a testování změn musí být plánováno a prováděno obdobně jako při vývoji. Dodavatel i odběratel se musí dohodnout na termínech a obsahu verze (release) produktu. Za údržbu se považují následující činnosti
   - odstraňování problémů,
   - změny rozhraní,
   - změny funkcí nebo zlepšování výkonu.

   Údržba by měla být podporována vhodnou organizační strukturou, která však musí být dostatečně pružná, poněvadž činnosti při údržbě nelze předem přesně plánovat. Tato organizace by měla zajišťovat pomoc při provádění změn. Při provádění změn musí stanovovat i priority změn a specifikovat výsledné efekty změn. Při údržbě musí sbírat data o selháních systému a provádění oprav. Tato data je nutné vyhodnocovat statistickými nástroji.

Při převzetí je nutné dohodnout pravidla pro uvolňování nových verzí produktu do provozu:
- jakým způsobem se provádějí úpravy/přechody mezi verzemi,
- popis změn funkcí mezi verzemi,
- postup, jak bude odběratel informován o současných a budoucích změnách,
- metody, jak se zajistí, aby změny nevyvolávaly další chyby,
- záznamy o tom, které změny byly provedeny, na jakých místech a u vícenásobných instalací na jakých lokalitách.

20.4.3 Aktivity při řízení konfigurace

Norma ISO 9000–3 vyžaduje pro zajišťování jakosti následující činnosti při řízení konfigurace:
- stanovení základních vlastností (baselines) produktu,
- správu verzí produktu,
- pravidla a odpovědnosti při procesu změn,
- procedury kontroly změn,
- sledování stavu procesů změn a variant vlastností produktu.

Úkolem řízení konfigurace je
- jednoznačná identifikace každé položky konfigurace (configuration item),
- identifikace verzí softwarových položek, které společně tvoří určitou verzi daného produktu,
- stanovení stavu softwarového produktu (ve vývoji – k dodání – instalován),
- kontrola souběžných změn dané softwarové položky více osobami,
- koordinace změn v kopiích produktů provozovaných na více místech,
- identifikace a sledování všech akcí a změn vyvolaných požadavkem na změnu od inicializace až po provedení.

Základními prostředky koordinace změn jsou dokumenty Požadavek na změnu (Engineering Change Request, ECR), Návrh změny (Engineering Change Proposal, ECP) a Záznam o změně (Engineering Change Notification, ECN). ECR má obsahovat
- iniciátora změny,
- číslo (id) požadavku,
- krátký popis problému,
- důvod změny,
- softwarová položka (item) nebo dokument, jehož se týká,
- datum vzniku,
- priorita,
- analytik pověřený řešením,
- datum určení analytika,
- datum dokončení analýzy,
- číslo (id) odpovídajícího návrhu změny (Engineering Change Proposal, ECP).

Návrh změny (ECP) by měl obsahovat údaje:
- analytik, který ECP zformuloval,
- datum podání,
- vazby na ECR, které změnu vyžadují,
- krátký popis,

Závislosti požadavků			
Požadavek kontraktu	Specifikace požadavků	Položka návrhu	Softwarová položka
1.1.zzz	3.1xxx	Dmodul1	Smodul1
	3.4xxx	Dmodul2	Smodul2
	7.1xxx	Dmodul8	Smodul7
1.2.zzz	3.4xxx	Dmodul2	Smodul3
	6.3xxx	Dmodul6	Smodul8

Tab. 20.1: Matice vystopovatelnosti (traceability) požadavků.

Vazby testů			
Požadavek kontraktu	Specifikace požadavků	Testový případ	Softwarová položka
1.1.zzz	3.1.xxx	Tjméno1	Smodul1
	3.4.xxx	Tjméno1	Smodul2
	7.1.xxx	Tjméno7	Smodul7
1.2.zzz	3.4.xxx	Tjméno1	Smodul3
	6.3.xxx	Tjméno2	Smodul8

Tab. 20.2: Matice vysledovatelnosti testů.

- dokumenty, části programů a plány testů ovlivněné návrhem,
- odhad doby řešení a pracnosti.

Každý návrh změny má projít revizí a pak může být implementován. K tomu účelu se vytváří Záznam o změně (ECN), který by měl obsahovat následující informace

- osoba provádějící změnu
- datum, kdy byl úkol předán,
- datum ukončení úkolu a převzetí výsledků,
- vazba na ECP,
- krátký popis řešení,
- softwarová položka, jíž se týká,
- testér,
- datum ustanovení testéra,
- datum ukončení testů,
- vazba na výsledky testů.

Řízení konfigurace má být založeno na plánu řízení konfigurace vypracovaného dodavatelem. Plán řízení konfigurace má obsahovat

a) Seznam organizací zapojených do řízení konfigurace a jejich odpovědností.
b) Aktivity řízení konfigurace.
c) Používané nástroje, techniky a metodologie řízení konfigurace.

d) Jaký má být status položek v okamžiku, kdy vstupují do systému řízení konfigurace. Zde je nutný kompromis. Příliš časté zařazování položek do konfigurace a jejich časté změny jsou spojeny s nebezpečím velkých dodatečných změn spolupracujících položek. Příliš pozdní zařazení neumožňuje, aby spolupracující položky při svém vývoji mohly dostatečně využívat služeb zařazované položky. Doporučuje se zařazovat nejdříve ty prvky, které poskytují nejvíce „služeb“ ostatním prvkům.

e) Činnosti při řízení konfigurace:

1. *Činnosti zajišťující identifikaci konfigurace.*
   Tyto činnosti musí integrálně pokrývat všechny etapy vývoje a mohou být požadovány i při údržbě. Pro každou položku konfigurace musí být zajištěna
   - jednoznačná identifikace položky a verze,
   - funkční a technické specifikace,
   - vývojové nástroje, které mají vliv na funkční a technické specifikace,
   - rozhraní na jiné softwarové položky,
   - všechny dokumenty a soubory obsahující informace a data významná pro danou položku; identifikace položky má být navržena tak, aby umožňovala snadnou detekci vztahu mezi položkou a požadavky smlouvy se zákazníkem.
2. *Činnosti umožňující vystopovat změny.*
   Norma vyžaduje procedury umožňující vystopovatelnost produktu uvolněného k užívání. K tomu je výhodné, i když to norma výslovně nestanovuje, používat maticové schéma pro vystopovatelnost požadavků a vystopovatelnost testů. Matice může mít tvar z tabulek 20.1 a 20.2. V tabulkách označují Dmodul – jméno modulu návrhu, Smodul – jméno programového modulu a Tjméno – jméno testového případu. Údaje ve sloupci Specifikace požadavků znamenají specifikaci místa v dokumentu specifikace požadavků. Z uvedených příkladů jsou zřejmé vazby mezi testovými případy, programovými moduly, částmi návrhu a úseky specifikace požadavků. Tyto vztahy jsou typu m:n. Je výhodné pro tento účel vytvořit databázi umožňující generaci výše uvedených schémat. To lze usnadnit tím, že příslušné dokumenty a programy obsahují formalizované komentáře umožňující automatické generování dat do databáze.
3. *Kontrola změn.*
   Dodavatel navrhuje a provádí identifikaci dokumentace, revizi a autorizaci všech změn softwarových položek podléhajících systému řízení konfigurace. Před tím, než je změna přijata a provedena, musí být potvrzena její oprávněnost a musí být prověřen její vliv na ostatní položky. Při tom lze využívat data z ECP, ECR a ECN. Dodavatel má mít k dispozici metody zjišťování a oznamování změn všem, jichž se týkají, a prostředky umožňující vystopovat vazby mezi změnami požadavků a měněnými softwarovými prvky. Dodavatel by měl mít k dispozici prostředky pro záznam, správu a analýzu (tisk) statutu softwarových položek, požadavků na změny a implementace schválených změn. To prakticky znamená používání vhodného informačního systému, který by měl poskytovat minimálně následující informace
   - přehled modulů a testových případů, které byly verifikovány,
   - přehled otevřených požadavků na změny,
   - přehled otevřených návrhů změn,
   - počet otevřených požadavků na implementaci změn,
   - statistiky doby řešení změn.

20.4.4 Správa dokumentů

Správa dokumentů podléhá obecným zásadám normy ISO 9000. Pro dokumenty je třeba určit:

a) které dokumenty podléhají formálním pravidlům správy dokumentace,
b) postup schvalování a vydávání dokumentů,
c) pravidla provádění změn včetně rušení platností a vydávání nových verzí.

Dokumenty mohou být různých typů:

a) procedurální dokumenty stanovující strukturu systému kvality tak, jak má být použita během životního cyklu softwaru,
b) plánovací dokumenty zachycující plánování a postup všech aktivit dodavatele a jeho spolupráci s odběratelem,
c) dokumenty tvořící součást produktu, minimálně:
   - vstupy a výstupy jednotlivých etap,
   - plány a výsledky verifikací a validací,
   - uživatelská dokumentace,
   - dokumentace pro údržbu.

Všechny tyto dokumenty jsou oponovány autorizovanými pracovníky před tím, než jsou uvolněny k používání. Všechny dokumenty musí být k dispozici ve všech místech, kde jsou potřeba. Zastaralé dokumenty jsou neprodleně stahovány z oběhu. Při používání elektronické formy je nutné věnovat zvýšenou pozornost procedurám schvalování přístupových práv, distribuování a archivace. Změny dokumentů jsou oponovány a schvalovány těmi orgány, které oponovaly a schvalovaly originální verze dokumentů. Výjimky musí být schvalovány vedením projektu. Schvalující orgány musí mít přístup ke všem informacím potřebným pro oponenturu a přijetí dokumentu. Je-li to praktické, je žádoucí změny vyznačit buď v dokumentu, nebo v jeho přílohách.

Ke každému dokumentu existuje snadno dostupná informace, podle níž lze identifikovat aktuální verzi dokumentu. Cílem je vyloučit používání dokumentů, které už nejsou relevantní.

Požadavkům normy lze snáze vyhovět, vytvoříme-li databázi následujících údajů:

- jméno dokumentu (identifikátor),
- číslo verze dokumentu,
- datum vydání,
- místo,
- autor,
- vlastník,
- oponent,
- kdo schválil.

Dokumenty by po jistém počtu změn měly být znovu jako celek oponovány a znovu jako celek vydány.

Informace o průběhu a výsledcích akcí kontroly musí být k dispozici ke stálému použití. Z toho důvodu má dodavatel vyvinout prostředky pro identifikaci, indexaci, ukládání, údržbu a zpřístupnění záznamů o jakosti včetně záznamů o produktech subdodavatelů. Cílem je kdykoliv prokázat dosažení požadované kvality.

U záznamů vztahujících se ke kvalitě je nutné udávat dobu platnosti. Záznamy o kvalitě musí být dostupné všem oprávněným. U každého záznamu musí být zřejmé, které verze produktu (přesněji konfigurace) se týká. Záznamy o jakosti mohou být dostupné i odběrateli, je-li to stanoveno ve smlouvě. V tom případě je žádoucí stanovit dobu, po kterou to platí.

20.4.5 Měření, pravidla, nástroje

ISO 9000–3 se zabývá problémem metrik poměrně stručně, zřejmě proto, že se metrikami zabývá norma ISO 9126. Norma konstatuje, že neexistují univerzálně akceptované metriky kvality softwaru. Je však vymezena minimální množina metrik, které by měly být používány. Norma připouští a doporučuje používání metrik, o kterých se norma nezmiňuje, dohodne-li se na tom dodavatel s odběratelem. Doporučuje se sledovat:

- počty selhání (trendy, frekvence). Pro každé selhání se doporučuje vytvořit záznam o závažnosti selhání, době potřebné na odstranění defektů, počty dosud nevyřešených selhání atd. (srv. kap. 15).
- rozsah testového pokrytí (např. procento kódu prověřeného, procento testovaných funkcí),
- chybné opravy, tj. počet případů, kdy je opravu nutno opravovat,
- frekvence požadavků na opravy,
- spotřeba práce, doby řešení,
- metriky pro vyhodnocování funkčních bodů (kap. 16).

Metriky musí být sbírány a vyhodnocovány systematicky. Pro každou metriku je třeba mít nástroj pro vyhodnocování současné hodnoty a trendů. Pro některé metriky je vhodné sledovat, zda nepřekročují určité meze (řízení na extrém – viz kap. 15).

Dodavatel by měl sledovat ty metriky, které indikují kvalitu procesu vývoje a dodávky. Metriky tedy musí umožňovat sledování toho, jak kvalitně je realizován vývoj vzhledem k dohodnutým kritériím a zda bylo dosaženo požadované úrovně kvality ve stanovených termínech. Současně má být možné sledovat efektivnost vývoje především vzhledem k možnosti redukce pravděpodobnosti, že se do systému dostanou chyby nebo že chyby nebudou včas zjištěny.

Nástroje měření a zjištěná data by měla být užitečná pro vývojové pracovníky i pro management projektu.

20.4.6 Nákup a využití produktů třetích stran

V této části se stanovují pravidla pro použití produktů nebo služeb třetích stran a produktů odběratele, které mají být integrovány do výsledného systému. Je povinností dodavatele prověřit, že takové produkty vyhovují podmínkám plynoucím z požadavků na celý produkt.

Doklady o nákupu systémů třetích stran musí obsahovat údaje jasně popisující objednaný produkt či službu. Dodavatel systému provádí revizi nákupní smlouvy s cílem ověření splnění požadavků na funkce celku. To se provádí před uvolněním produktu k použití. Tento požadavek se týká softwaru i hardwaru.

Dodavatel musí vybírat subdodavatele na základě toho, zda jsou schopni splnit požadavky na subdodávku včetně požadavků na kvalitu. Dodavatel udržuje seznam akceptovatelných subdodavatelů.

Výběr subdodavatelů a rozsah kontroly subdodávek ze strany dodavatele celého systému závisí na typu produktu a, je-li to možné, na informacích o subdodavateli založených na dříve prokázaných schopnostech a výkonech. Dodavatel ručí za efektivnost kontroly kvality produktu jako celku.

Dodavatel je odpovědný za kvalitu prací subdodavatelů. To může znamenat, že dodavatel provede u subdodavatelů revize a jiné oponentury podle vlastních předpisů a pravidel. To musí být zahrnuto do smlouvy na subdodávku. Totéž platí pro přijímací testy subdodávek.

Dodavatel musí vytvořit prostředky pro validaci, ukládání, ochranu a údržbu produktů třetích stran a produktů odběratele, jsou-li integrovány do dodávky na smluvním základě. Produkty odběratele, které se ukáží být nevhodné pro použití v rámci produktu, musí být jako nevhodné oficiálně označeny a tento fakt musí být ohlášen odběrateli.

20.4.7 Zaškolování

Dodavatel stanovuje a udržuje postupy zjišťování potřeb zaškolování. Inženýři potřebují projít školením, aby mohli plnit svoje povinnosti při provozu systému. Je nutné zjistit jaké postupy, techniky, nástroje a metodologie musí pracovníci uživatele zvládnout, a zajistit odpovídající školení a zácvik. Školení je nutné plánovat a jeho průběh zaznamenávat včetně případných osvědčení o absolvování.

Při zjišťování potřeby školení lze využívat matici, ve které řádek odpovídá pracovníkovi, který je uveden na levé straně, a sloupec určité dovednosti. Prvek v i-tém řádku a j-tém sloupci udává, zda i-tý pracovník zvládl zde uvedenou dovednost nebo zda ji má zvládnout.

20.5 Vzor pro návrh softwarového procesu ISO 9000–3

Požadavkům normy ISO 9000–3 vyhovuje následující schéma vývoje, které může být upravováno podle konkrétní situace.

- *Fáze 1: Specifikace produktu a plánování produktu:*
  a) Dokument Marketingové požadavky: rozbor situace na trhu, koncepce produktu, základní cíle.
  b) Předběžný rozpočet a termíny.
  c) Podrobný rozpočet a termíny pro fázi 2.
- *Fáze 2: Technická specifikace a plánování:*
  a) Prototyp ověřující realizovatelnost (nepovinné).
  b) Specifikace požadavků.
  c) Plán vývoje softwaru.
- *Fáze 3: Návrh produktu:*
  a) Návrh softwaru.
  b) Specifikace testů systému.
- *Fáze 4: Implementace:*
  a) Kódování a testování částí.
  b) Integrační testování.
  c) Vývoj testových případů pro testy systému.
- *Fáze 5: Testování systému:*
  a) Provedení testů systému.
  b) Shrnutí výsledků testů (baseline system test results).
  c) Shrnutí funkčnosti systému.
  d) Aktualizovaná dokumentace produktu.
- *Fáze 6: Evaluace produktu,* testování a další činnosti prověřující, že produkt pracuje tak, jak bylo specifikováno:
  a) Alfa validace ve vývojovém týmu.
  b) Beta validace u (vybraných) uživatelů.
- *Fáze 7: Předání produktu (product release):*
  a) Uvolnění k výrobě, tj. ke kopírování a kompletaci.
  b) Předání zákazníkovi, přejímka.
- *Fáze 8: Údržba/vylepšování.*

Výstupem fáze 1 je vymezení potřeb zákazníka, stanovení základních vlastností produktu a předběžný odhad nákladů a termínů. Výstupem jsou tři dokumenty: Marketingové požadavky, Předběžný rozpočet a termíny a Rozpočet a termíny pro fázi 2. Podmínkou ukončení každé fáze je schválení dokumentů po revizi. Schválení dokumentu Marketingové požadavky podléhá schválení vedoucího projektu, technického vedoucího projektu, osoby odpovědné za provoz aplikací u zákazníka, zástupce marketingu a případně zákazníka.

Předběžný rozpočet schvaluje zástupce marketingu a vedení vývojového oddělení. Analýza marketingových požadavků může mít[3] následující průběh (**Marketingový plán** MP):

1. Stanovení cílů v následující struktuře: potenciální uživatelé, plánované funkce, technická a jiná omezení, hrozby a rizika.
2. Vstupy: Analýza trhu, přehledy situace, výsledky analýzy potřeb zákazníků atd.
3. Řešitelský tým.
   Základ: Marketingoví odborníci a pracovníci zákazníka.
   Širší tým: Vývojáři, obchodníci, odborníci přes danou oblast aplikací.
   Zástupci uživatele: Zástupci managementu, informatici, případně koncoví uživatelé nebo jejich přímí nadřízení.
4. Úkoly:
   a) Interview uživatele.
   b) Vytvoření předběžné verze dokumentu Marketingové požadavky (MR).
   c) Rozeslání pracovní verze MR zainteresovaným k posouzení.
   d) Společná revize (review) zástupců zákazníka a vývojářů, zjištění nedostatků MR. Návrh termínů a zdrojů.
   e) Návrh odstranění nedostatků.
   f) Úprava MR a MP.
   g) Opakovat c) až f) dokud není MP přijat
   h) Souhlas s MP stvrzen podpisy

Přílohy: Záznamy z jednání o zdokonalení MR.

Výstup: MP.

Podmínky ukončení: Oponentura a podepsání MP zástupci obou stran – vývojáři, marketingem, prodejci a managementem.

Struktura dokumentu **Marketingové požadavky** (marketing requirements, MR)

1. Přehled produktu:
   a) Popis.
   b) Strategické vlastnosti.
   c) Které funkce produkt zajišťuje a které nikoliv.
2. Předpoklady a rizika.
3. Předmět řešení:
   a) Diagram kontextu systému, jehož je produkt součástí.
   b) Rozhraní produktu:
      – vstupy,
      – výstupy,
      – dialogy, scénáře.

3. Tj. je v souladu s doporučeními normy.

4. Funkce:
   a) Vstupy, výstupy / krátký popis.
   b) Stavy u funkcí definovaných sítí přechodů:
      – definice stavů,
      – chování v každém stavu,
      – podmínky přechodu mezi stavy.
5. Popis funkcí z hlediska uživatele.
6. Použití:
   - četnost užití funkcí, podmínky provedení funkcí,
   - oblast vstupních hodnot, mezní hodnoty vstupních dat.
7. Konfigurace / kompatibilita.
   a) Hardware.
   b) Dřívější realizace téhož produktu.
   c) Současné produkty dodavatele.
   d) Produkty třetích stran.
8. Technická omezení:
   a) Časová.
   b) Paměťová.
   c) Parametrů sítě.
   c) Datová propustnost.
   d) Zabezpečení.
   e) Schopnost auditu.
   f) Standardy.
   g) Možnosti použití u jiných zákazníků a v cizině.
9. Analýza produktu:
   a) Oblast použitelnosti.
   b) Novost návrhu a implementace.
   c) Možnost změn požadavků.
   d) Jiné:
      – Hardware existující nebo nový.
      – Zpracování v reálném čase.
      – Existence rozhraní na produkty třetích stran.
      – Další aspekty.
10. Požadavky na spolehlivost a kvalitu.
11. Požadavky uživatele na dokumentaci.
12. Vytváření customizovaných kopií, instalace.
13. Plánované další funkce.
14. Odkazované dokumenty:
    a) Specifikace požadavků.
    b) Dokumenty třetích stran.
    c) Marketingový plán.
15. Předběžný rozpočet a termíny.

16. Přílohy.

Obdobnou strukturu jako analýza marketingových požadavků mají dokumenty Specifikace požadavků a Plán vývoje, kde se navíc stanovují termíny a vnitřní návaznosti.

Plán dokumentace:

Účel: Popsat technické dokumenty potřebné pro vyvíjený produkt.

Shrnutí obsahu dokumentů a určení zdrojového materiálu pro jednotlivé dokumenty.

Určení potřebných zdrojů, termínů a omezení.

Vstupy: Dokumenty Marketingové požadavky, Specifikace požadavků a Dokument definující vnější rozhraní systému.

Tým: Specialisté na technické publikace společně s vývojáři, marketingovými odborníky a testéry.

Úkoly:

1. Určení dokumentů, které bude třeba vytvořit či modifikovat.
2. U modifikovaných dokumentů určení částí, které je třeba modifikovat.
3. Určení podkladů pro nové dokumenty či modifikaci starých.
4. Nástin obsahu nových či modifikovaných dokumentů.
5. Identifikace zdrojů, rizik, termínů a omezení pro tvorbu dokumentů.
6. Vytvoření pracovní verze plánu dokumentování.
7. Revize, zjišťování problémů s plánem. Opravy.
8. Písemné schválení plánu.
9. Správa verzí dokumentů.

Podmínky ukončení: Oponentura a oficiální schválení.

Výstup: Plán dokumentace.

Plán návrhu systému.

Účel: Naplánovat práce při volbě architektury softwaru. Stanovení zásad implementace, definic reakcí na chyby, vlastností uživatelského rozhraní a struktury databáze.

Vstup: Specifikace požadavků, dokumentace produktů třetích stran a rozhraní.

Úkoly:

1. Dekompozice systému do komponent.
2. Definice vstupů a výstupů a algoritmů komponent.
3. Dekompozice programů komponent do modulů.
4. Návrh implementace modulů.
5. Popis reakcí na chyby.
6. Návrh databáze.
7. Určení postupu zpracování chyb v modulu.
8. Rozčlenění uživatelského rozhraní do oken a obrazovek.
9. Stanovení typu a umístění grafických objektů reprezentujících operativní informace.
10. Vytvoření prototypu obrazovek a oken.
11. Předvedení prototypu rozhraní vedení projektu, zákazníkovi a specialistům na zjišťování kvality.
12. Vytvoření dokumentu: Návrh softwaru.
13. Oponentura návrhu.
14. Náprava nedostatků.

15. Písemné schválení uživatelského rozhraní a reakcí na chyby vedením projektu, vedoucími marketingu a prodeje.

Dokument **Návrh softwaru** může mít následující strukturu:

1. Přehled produktu (převzato do značné míry ze specifikace požadavků):
   1.1. Popis.
   1.2. Strategický/hlavní účel.
   1.3. Problémy, které systém řeší.
2. Předpoklady a rizika.
3. Předmět řešení:
   3.1. Kontextový diagram systému.
   3.2. Schéma kontextu produktu.
      a) Vstupy.
      b) Výstupy.
4. Požadavky na implementaci. Seznam požadavků tvaru:
   4.x.1. Marketingový požadavek číslo x, $x = 1, 2, \ldots$
   4.x.1. Požadavek na implementaci x:
      a) činnost,
      b) podmínky startu,
      c) vstupy / výstupy,
      d) reakce na chyby,
      e) seznam problémů a jejich řešení.
5. Odvozené požadavky:
   5.1. Odvozený požadavek 1.
      a) činnost,
      b) podmínky startu,
      c) vstupy a výstupy,
      d) reakce na chybu.
   5.2. Odvozený požadavek 2.
      atd.
6. V případě, že lze systém charakterizovat přechodovým diagramem, uvést diagram přechodů a požadavky na stavy.
7. Technická omezení: Rozbor toho, jak technická omezení z dokumentu Marketingové požadavky ovlivní řešení.
8. Uživatelovy operace.
   8.1. Operace 1.
      a) Popis.
      b) Popis obsahu obrazovek, popis dialogů.
      c) Reakce na chyby.
   8.2. Operace 2.
      atd.
9. Specifikace hardwaru a základního softwaru.
10. Kompatibilita.
    10.1. S předchozím vydáním (release) téhož produktu.

10.2. S ostatními produkty dodavatele.
10.3. Se systémy třetích stran.
11. Definice dat.
11.1. Rozhraní na okolí systému.
11.2. Požadavky na spolupráci funkcí.
11.3. Data specifická pro daný hardware.
12. Testování.
12.1. Druhy testování.
12.2. Kvalifikační matice obsahující řádek pro každý testový případ a sloupec pro každou funkci. Do políček se zaznamenává, zda se daný testový případ týká uvedené činnosti.
13. Matice vysledovatelnosti požadavků (tab. 20.1).
14. Odkazované dokumenty.
14.1. Specifikace požadavků.
14.2. Dokumenty třetích stran.
14.3. Marketingový plán, marketingové požadavky.
15. Přílohy.

20.6 Poznámky k normě ISO 9000–3

Z výše uvedených faktů vyplývá značná pracnost spojená s uplatněním normy ISO 9000–3. Aplikace normy navíc vyžaduje, abychom výše zmíněné dokumenty přizpůsobili konkrétní situaci, celý postup si nechali předběžně schválit, ověřili si jej v praxi a pak požádali autorizované pracoviště o certifikace. Pravděpodobnost úspěchu při prvním pokusu je asi 20 %. Používat normu ISO 9000–3 se proto vyplatí pro větší systémy vyvíjené od začátku a spíše pro vícenásobné použití. Její použití je velmi pracné a pro menší projekty se tedy většinou nevyplatí. Norma neobsahuje doporučení pro customizaci.

Struktura normy se zdá být orientovaná spíše na starší softwarové technologie. Norma nevylučuje použití objektových technologií, nedoporučuje však žádné prostředky výhodné pro objektové technologie. Norma ISO 9000–3 je všeobecně více akceptována v Evropě a méně v USA. Norma ISO 9000–3 obsahuje mnoho správných zásad, např. požadavek spolupráce dodavatele s odběratelem, které je vhodné dodržovat, i když nemíníme realizovat všechna její doporučení.

21

Studie řídicího IS strojírenské prvovýroby

Přímé řízení výroby je oblast aplikací, které vyžadují velký rozsah customizace a které často musí být vyvíjeny od počátku. Důvodem je silná závislost funkcí na typu výroby a organizačních zvycích a také to, že IS pro řízení výroby používají pracovníci s nízkou počítačovou gramotností. Nižší kvalifikace pracovníků omezuje možnosti rychlých organizačních změn a změn pracovní náplně.

Řízení výroby je dobrým příkladem použití různých softwarově inženýrských technik, jako je dekompozice, dělba práce mezi IS a obsluhou, problém spolehlivosti dat atd. Podle (Landauer, 1995) jsou IS ve výrobě vedle komunikací v podstatě jediné IS, o jejichž pozitivním efektu na výrobu nelze pochybovat.

V této kapitole uvedeme příklad návrhu řídicího IS strojírenské dílny jako ilustraci výhod technologie spolupracujících aplikací. Příklad (dále zmiňovaný jako KS-PVS) vychází z úspěšného projektu řízení strojírenské dílny zajišťující prvovýrobu součástí obráběcích strojů.

21.1 Řízení průmyslové výroby

Proces nasazování a uplatňování výpočetní techniky v průmyslu je přímým odrazem vývoje její „užitné hodnoty“, tj. schopnosti efektivně řešit rozpor mezi neustále se zvyšujícími požadavky na rychlost a kvalitu rozhodování a vzrůstajícím objemem dat, jejichž zpracování je pro daný rozhodovací proces požadováno. Tato skutečnost se odráží i v rostoucím podílu aplikací pracujících v reálném čase, které dnes zasahují i do oblastí dříve převážně dávkového charakteru.

Tyto tendence jsou přímým důsledkem jak technologického pokroku a cenové dostupnosti informačních technologií, tak i v současné době probíhajících zásadních změn v organizaci, řízení i technologii průmyslové výroby, zdůrazňující požadavek „pružnosti“ (srv. Halevi, 1980).

Pojem pružně automatizované výroby vznikl jako reakce na stále se zostřující konkurenci, která dnes nutí výrobní podniky v řadě průmyslových odvětví přizpůsobovat výrobu stále rychleji požadavkům trhu a neustále zvyšovat její kvalitu a efektivnost při výrazném zkracování inovačních cyklů a také zkracování výrobních sérií. Organizace výroby založená na principech „tvrdé“ automatizace, kdy výrobní systém představuje posloupnost operací s pevně danou konfigurací technických prostředků, je efektivní v podmínkách hromadné výroby, pro malé a střední série je však poměrně nevhodná.

21 Příklad IS pro řízení výroby

Vznikem numericky řízených strojů (1952) a průmyslových robotů (1961) založených na spojení číslicového řízení a počítačové technologie vstupuje do výrobního procesu zcela nový prvek: typ výrobních operací je v principu měnitelný beze změny technologického zařízení pouze „změnou programu“. Jsou tak vytvořeny základní předpoklady pro vznik takových automatizovaných výrobních systémů, které jsou schopné vyrábět dostatečně široké spektrum výrobků s minimálními časovými a ekonomickými ztrátami nutnými pro přechod od jednoho typu výrobku k druhému. Abychom se dokázali orientovat i v poněkud širších souvislostech, zmíníme se i o problematice výroby integrované počítačem (CIM).

21.1.1 Výroba integrovaná počítačem (CIM)

Termín „výroba integrovaná počítačem“ (Computer Integrated Manufacturing, CIM) označuje vzájemné propojení všech prvků výrobního procesu do jednotného systému s cílem automatizovat kromě vlastní výroby i všechny, nebo alespoň rozhodující, informační a řídicí procesy, které jsou spjaty s výrobou a prodejem výrobků.

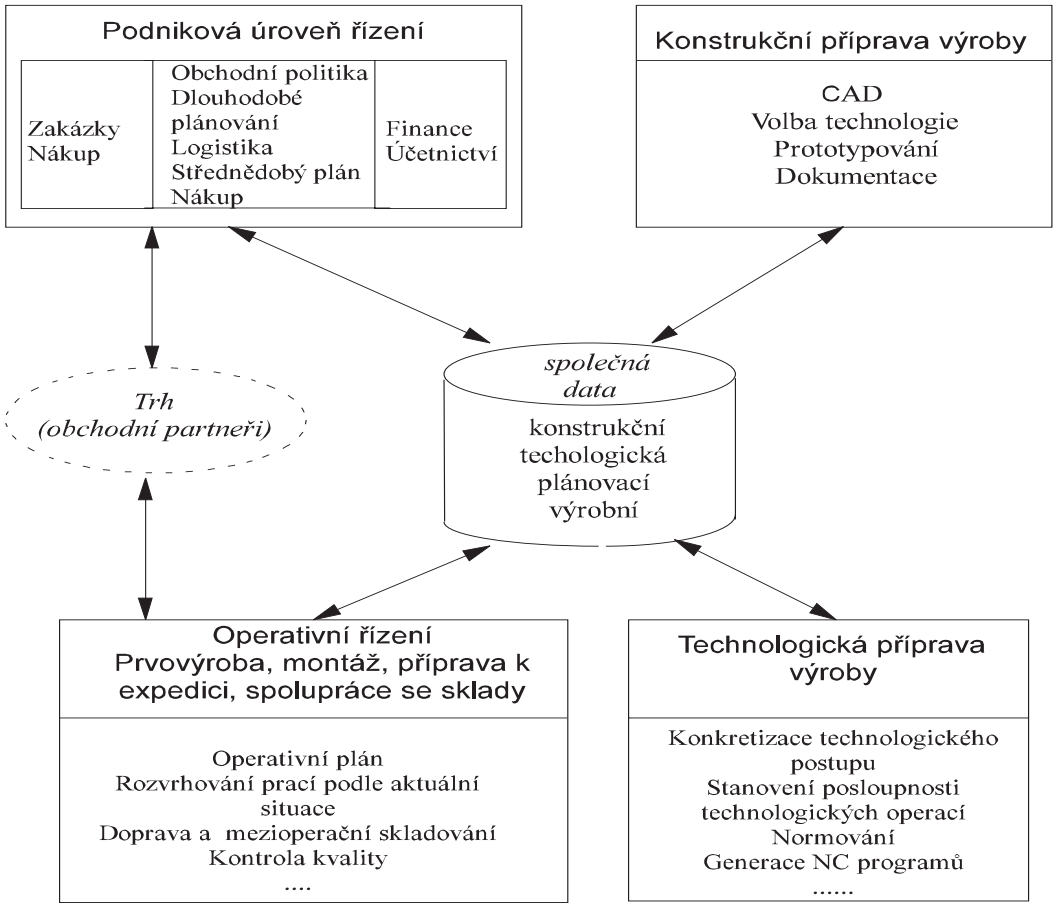
CIM zahrnuje tyto základní oblasti (obr. 21.1):

- sledování finančních toků, IS obchodní činnosti – objednávky, platby, bilancování výrobních kapacit, sledování zásob, podpor managementu,
- organizaci a řízení výroby na úrovni podniku,
- konstrukční přípravu výroby (Computer Aided Design, CAD),
- technologickou a technicko-organizační přípravu výroby (Computer Aided Process Planning, CAPP),
- organizaci a řízení výroby na úrovni provozu (Computer Aided Manufacturing, CAM).

CIM tedy integruje výrobu zboží s informačním systémem. Zavedení CIM umožňuje:

- zvýšit produktivitu zařízení a snížit jednotkové náklady,
- zvýšit pružnost výrobního systému díky podstatnému snížení nákladů na přestavbu výrobního zařízení při změně výroby,
- zlepšit kvalitu výroby a technickou úroveň výrobků,
- snížit rozpracovanost výroby a rozsah zásob,
- snížit potřebu pracovních sil při zlepšení pracovních podmínek díky snížení potřeby nebezpečné a namáhavé práce.

Při realizaci CIM se věřilo, že optimální postup vpřed je přes vytvoření „ostrovů automatizace“, jejichž propojováním vznikne ucelený systém. Tento předpoklad se nesplnil. Příčinou bylo zřejmě to, že „ostrovy“ zefektivnily jen jistou část podniku, neznamenaly však podstatný přínos pro činnost managementu a obchodní činnost. Integrace ostrovů je vlastně integrací zdola, která nemůže být příliš úspěšná, není-li zřejmé, jaké budou potřeby a omezení vyšších úrovní CIM. Ostrovy automatizace se nedařilo integrovat také proto, že nebyly k dispozici dostatečně výkonné softwarové prostředky podporující spolupráci aplikací. Podnikové IS se budují od IS podporujících některé aspekty operativy, především v oblasti finančních toků, obchodní činnosti, jako je správa objednávek a styk se zákazníky a správy zásob. Pro tyto činnosti byly díky jejich relativní příbuznosti u mnoha podniků vytvořeny generovatelné systémy, jako je R/3 firmy SAP či Oracle Financial či IS firmy Lawson Software. Tyto systémy se postupně rozšiřují tak, aby podporovaly činnost managementu a také aby integrovaly řízení výroby a tedy obsahovaly CIM jako svoji součást. Technologie spolupráce aplikací umožňuje integrovat části CIM do generovatelných systémů v rámci customizace.



Obr. 21.1: Struktura CIM.

21.1.2 Softwarové prostředky realizace CIM

Distribuovaný informační a řídicí systém výrobního podniku musí v podmínkách CIM umožnit zpracování jednotných dat ve fyzicky vzdálených uzlech vybavených dostatečnou informační kapacitou a „inteligencí“. Většina úloh musí být řešena v reálném čase s dobou odezvy pohybující se v rozmezí od několika milisekund do několika minut. To se týká i části chybových situací vyžadující odpojení chybného uzlu a jeho pozdějšího restartu. Tyto požadavky zvyšují náročnost realizace systému (kap. 1 a 15) a kladou vysoké nároky na kvalitu realizovaného systému.

Složitost některých rozhodovacích procesů vede k pokusům o uplatnění principů umělé inteligence (AI) v těch oblastech, kde je nutné a možné vytvořit prostředky podpory rozhodovacího procesu realizovaného ať už zcela automatizovaně nebo v přímé spolupráci s člověkem s využitím databáze znalostí (prognostika trhu,

plánování apod.). Plné využití principů AI je zatím brzděno značnými nároky programů používajících principy AI na paměť a čas počítačů, a také nedořešením řady softwarově inženýrských problémů (metody dekompozice, propojení algoritmů AI na širší programové okolí, jako jsou datové báze s programy psané v klasických programovacích jazycích atd.). Řešením může být koncepce softwarového agentu. SW agenty jsou používány především pro zjišťování a analýzu informací na síti včetně Internetu. Typickým příkladem mohou být agenty analyzující faktury, např. ty, které znějí na vysoké částky, a nabízející možná opatření. Podobně koncipované nástroje pomáhají řešit mnoho výrobních problémů.

Z obecnějšího hlediska umožňuje využití metod softwarového inženýrství při specifikaci cílů, návrhu a realizaci systému CIM oddělit vytváření jednotlivých funkčních subsystémů, nezávisle je ladit a postupně integrovat do větších celků. Tento postup je nezbytný pro dosažení takového stavu, ve kterém jsou jednotlivé systémy „montovány“ ze základních funkčních modulů, které jsou dodávány různými výrobci.

Důležitou vlastností IS ve výrobních zařízeních je právě to, zda jsou „otevřené“, tj. jak je snadné do nich integrovat další aplikace. Mezi aplikace, které je třeba integrovat, patřívají aplikace podporující práci vrcholového managementu a aplikace realizující řízení provozů a technologických procesů, které jsou z hlediska obchodního dnes považovány za nedílnou součást technologie. Jako samostatnou integrovatelnou aplikaci bývá výhodné realizovat některé části CIM – např. řízení dílen strojírenské prvovýroby. To byl předmět výše zmíněného úspěšného projektu.

Základní podmínkou realizace CIM je počítačová síť. Jejím úkolem je propojit nejrůznější typy výpočetních a řídicích systémů v prostředí, v němž není sice zátěž jednotlivých komunikačních tras enormní, avšak v souhrnu je objem komunikace v systému značný. Vytvořený komunikační systém musí být spolehlivý, ekonomický a použitelný pro nejširší okruh uživatelů v prostředí se silnými rušivými elektrickými poli.

Použití uzavřených lokálních datových sítí (LAN) je při řízení výroby spojeno s problémy s nekompatibilitou technických prostředků a systémů použitých v CIM. Problémy může vyvolat i to, že síť se postupně rozšiřuje.

Pokrok v síťových technologiích, např. širokopásmové sítě s vysokou propustností a principy, které se osvědčily v Internetu, umožňuje nová řešení. Slibné je využití principů Internetu na podnikové úrovni známé jako Intranet. Jeho použití na úrovni přímého řízení procesů není zatím dostatečně prověřeno.

21.1.3 Pružné výrobní systémy (PVS) ve strojírenství

Základním článkem organizace strojírenské výroby je závod. V něm se uzavírá celý cyklus od objednání výrobku, technického, hmotného i organizačního zabezpečení jeho výroby přes vlastní výrobu až po předání finálního výrobku odběrateli. Výrobní činnost závodu je rozdělena do řady provozů.

Strojírenský výrobní cyklus představuje řadu procesů, které lze rozdělit do tří základních skupin:

- přípravné procesy prováděné většinou na úrovni závodu;
- výrobní procesy probíhající na úrovni dílen a provozů;
- technologické procesy na pracovištích.

Těmto úrovním odpovídají v zásadě i tři základní úrovně řízení: závodní, provozní a technická. Přípravné procesy vytvářejí podmínky nutné pro zabezpečení výroby a zahrnují následující základní okruhy činnosti:

- obchodní zajištění výroby, její plánování a sledování na úrovni jednotlivých zakázek, hmotné a kapacitní zajištění a ekonomické vyhodnocení,
- konstrukční přípravu výroby,
- technologickou a technicko-organizační přípravu výroby,
- logistiku.

Informace o výrobku je v průběhu času postupně zpřesňována až na úroveň specifikace součástí, odpovídajících výrobních operací a jejich zabezpečení. Výrobní procesy probíhají na úrovni provozu a jsou chápány jako posloupnost dílčích technologických i netechnologických operací od výroby součástí až po konečnou montáž finálního výrobku. Technologické operace mění tvarové, např. obrábění, a fyzikálně chemické, např. tepelné zpracování, vlastnosti vyráběné součásti a provádějí její montáž do vyšších celků, netechnologické operace vytvářejí podmínky pro průběh technologických operací, jako je doprava, skladování a kontrolní operace. Charakteristickými a z hlediska podílu na celkové pracnosti rozhodujícími operacemi jsou ve strojírenské prvovýrobě operace obrábění (32 %) a montáž (34 %), z netechnologických jsou kapacitně i finančně nejnáročnější doprava a skladování. Pro klasickou organizaci práce je typické, že výrobní časy, tj. časový úsek od zahájení do dokončení výroby, jsou velmi dlouhé, avšak součet technologických časů, kdy se s výrobkem něco děje, je krátký. Většinu času se s výrobkem nic neděje. Uplatnění IT umožňuje výrazně zkrátit výrobní časy a tím zvýšit pružnost výroby. Jedná se tedy o podobné efekty jako u BPR.

Koncepce počítačem integrované výroby je založena na informatické infrastruktuře podporující vytváření koncepcí rozvoje závodu, zpracování dlouhodobého plánu, technické záměry a výrobní podklady až po přímé řízení výrobních operací. Funkce subsystémů CIM pokrývají především následující činnosti:

1. *Závod.* Na úrovni závodu se provádí zpracování veškerých agend důležitých pro chod podniku (marketing, zpracování zakázek, plánování, materiálně technické zásobování atd.). Závod musí zajistit kapacitní vytížení jednotlivých provozů. Plán práce provozů se odvozuje od střednědobého plánu výroby. Závod upravuje na základě informací o průběhu výrobního procesu své plány. Kvalita všech činností na úrovni závodu silně závisí na kvalitě informací o průběhu výroby. V této oblasti je jeden z hlavních přínosů CIM.
2. *Konstrukční příprava výroby (CAD).* CAD systémy vytvářejí geometrický model součásti, podporují konstrukční výpočty, generují podklady pro vlastní výrobu (výkresy, podklady pro technologickou přípravu výroby atd.). Výhodou CAD je značné zvýšení produktivity práce konstruktéra, zkvalitnění návrhu, zrychlení vývoje a ulehčení navazujících etap přípravy výroby, např. generace programů pro numericky řízené stroje. CAD není pouhým systémem na automatizované kreslení výkresů, je to skutečně systém pro navrhování, který má k dispozici velké soubory dat (grafických i negrafických) výpočtové prostředky a řadu speciálních periferií.
3. *Technologická a organizační příprava výroby* (Computer Aided Production Planning, CAPP). Tato část CIM pracuje s výstupy z CAD. Vytváří návrh technologického postupu výroby, programy pro řízení numericky řízených strojů a stanovuje technicko-organizační normy. Největší efekt mají aplikace počítačů při tvorbě programů pro numericky řízené stroje.
4. *Organizace a řízení výroby na úrovni dílny/provozu* (dílenské řízení výroby, CAM). Základem řízení dílny strojírenské výroby je soubor „operací“, tj. organizačně uzavřených pracovních akcí, např. obrábění jistého množství součástek na obráběcím centru nebo technická kontrola obrobků. Soubor operací je základem rozhraní řízení dílny na ostatní části CIM. Z operací zadaných střednědobým plánem, který může být vytvořen na podnikové úrovni a vychází z kapacitních rozvah, se vytváří operativní plán výroby – operace se přiřazují jednotlivým pracovištím. Pro chod výroby je nutné udržovat informace o průběhu výroby, tj. informace o tom, které operace se provedly pro danou výrobní zakázku Z, kde je materiál pro Z atd., a také informovat o stavu výroby nadřízené úrovně řízení.

Pružný výrobní systém představuje aplikaci principů pružné automatizace technologických a dopravně-manipulačních procesů a počítačového řízení na úrovni dílny/provozu. Pružný výrobní, resp. montážní systém (PVS) je nejčastěji skupina technologicky nebo předmětně uspořádaných pracovišť, obráběcích nebo montážních

center, navzájem propojených prostředky mezioperační dopravy a skladování. Celý technologický komplex je řízen sítí počítačů.

Při specifikaci požadavků na řídicí software pružného výrobního systému je výhodné PVS dekomponovat do následujících subsystémů.

a) Subsystém pracovišť. Pracovištěm je z hlediska řízení uzavřená jednotka definovaná rozhraním umožňujícím:
   1. Generaci požadavků na dopravu a skladování.
   2. Spolupráci se subsystémem řízení – oznámení o ukončení výrobní operace, požadavky na přípravu nástrojů atd.
   3. Generaci řídicích signálů, např. požadavek na zaslání programu pro numericky řízený stroj.

b) Subsystém dopravy a skladování. Úkoly subsystému jsou následující:
   1. Vedení evidence o mezioperačním skladu ve tvaru adresa – obsah.
   2. Odpovědi na dotazy tvaru „kde je paleta s obsahem O“, „kde je volné místo“ atd.
   3. Provádění příkazů tvaru „odvez z místa M paletu P“, resp. „přivez na místo M paletu P“. Přitom předpokládáme, že subsystém je schopen určit z informací o skladu údaje nutné k definici přepravní akce „převez odkud, kam, co“.

c) Subsystém organizace a řízení výroby. Tento subsystém udržuje operativní plán výroby a je schopen najít podle situace na pracovištích informace o další operaci pro dané pracoviště. Při nutných úpravách operativního plánu PVS spolupracuje subsystém s operátorem systému, s nadřízenými úrovněmi a s podřízenou technologickou úrovní PVS.

21.2 Příklad návrhu informačního systému pro pružný výrobní systém

Software pružného výrobního systému vycházel ve výše zmíněném konkrétním příkladě KS-PVS z následujících zásad.

21.2.1 Analýza základních požadavků a podmínek

Podstatou řízení prvovýroby ve strojírenské dílně je zajištění takového průběhu výrobního procesu, který zajistí splnění požadavků plánu a optimálně využívá kapacity výrobní soustavy. Plnění plánu je však neustále narušováno náhodnými a tudíž nepredikovatelnými poruchami výrobního procesu, takže v jistém okamžiku je třeba řešit vzniklé disproporce novým rozplánováním výrobních úkolů. Důvody odchylek jsou nap ř. onemocnění pracovníků, poruchy v zásobování, neplánované zásahy do výroby, jako je nutnost řešit havárie u zákazníků, především však nepřesnost dat technologických norem, na kterých je plánování založeno. Výrobní časy navíc závisí na řadě faktorů, jako je kvalita pracovníků, technický stav strojů a přesnost měření výrobních časů. Plánování a řízení výrobního procesu bylo v KS-PVS rozděleno do dvou základních úrovní – plánovací a řídicí.

Plánovací úroveň zahrnuje:

- vytvoření krátkodobého plánu výroby, který kromě požadavků střednědobého plánu bere v úvahu aktuální stav rozpracované výroby, optimální velikost výrobních dávek a „rozumné“ vytížení kapacit. Krátkodobý plán vytváří zásobu práce pro pracoviště víceméně bez ohledu na omezení plynoucí z návaznosti operací; krátkodobý plán zajišťuje v prvém přiblížení vytížení kapacit. Nebere ohled na většinu technologických omezení.

- dynamické rozvržení výrobních úkolů na jednotlivá pracoviště založené na požadavcích krátkodobého plánu, aktuálním stavu výrobního procesu a jeho předpokládaném průběhu. Dynamický rozvrh stanovuje pořadí, podle kterého jsou operace zadávány pracovištím nebo skupinám pracovišť. Tyto informace nazveme rozvrhem výroby. Rozvrh výroby je tvořen datovými strukturami obsahujícími informace o technologických operacích potřebných pro řídicí systém. Data operací definují mj. fronty prací na pracoviště a technologickou návaznost operací.

Řídicí úroveň: přiřazuje na základě rozvrhu výroby a hlášení pracovišť operace pracovištím a uskutečňuje i další činnosti operativního řízení výroby. Řídicí úroveň dále zajišťuje distribuci programů pro numericky řízené stroje. Rozhraní propojující plánování a řídicí úroveň umožňuje automatizovat v rámci řídicího systému PVS většinu procesů dílenského řízení výroby bez ohledu na to, zda se fronty prací vytváří ručně či automaticky, a nezávisle na tom, jakou strategii rozvrhování prací používají subsystémy plánování. Rozvrh prací může být řídicí úrovní předán ve formě front prací na pracoviště a plánovací úroveň může být dávkově informována o plnění plánu za plánovací období. Nakonec byla zvolena jiná varianta používající dialog obou úrovní. Pracoviště si v tomto případě vyžádá přidělení každé operace a hlásí provedení každé operace okamžitě po jejím ukončení.

Krátkodobé plánování je založeno na statickém modelu výrobní soustavy, vychází z použitelných kapacit, výrobních časů a požadovaných termínů odvádění výroby. Složitost úlohy a její časová náročnost jsou závislé na míře podrobnosti, s níž je operativní plán zpracováván. V rozhodující většině případů jsou však změny krátkodobého plánu v reálném čase nemožné. Doba práce rozvrhovacích algoritmů silně závisí na organizaci dat.

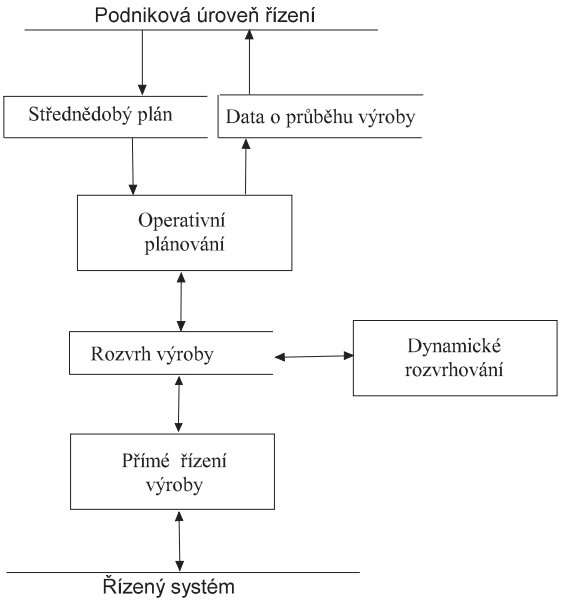
K vytvoření použitelného rozvrhu jsou zapotřebí správná data, která však často nejsou k dispozici. Časová náročnost generace rozvrhu může vést k situaci, kdy již v okamžiku dokončení nového rozvrhu skutečný stav výrobní soustavy neodpovídá stavu předpokládanému rozvrhem a pracně zhotovený podrobný rozvrh tedy není v podstatě k ničemu. Prakticky použitelný je tedy pouze takový rozvrhovací algoritmus, který buď netrvá příliš dlouho, nebo nemusí být spouštěn často.

Rozvrhovací algoritmus může pracovat rychle jen tehdy, je-li omezena velikost rozhodovacího prostoru, např. úplnou záměnností pracovišť, což je v běžném provozu nereálný předpoklad, nebo vytvořením variantních technologických postupů umožňujících nahradit výpadek jednoho typu pracoviště vyšším zatížením pracovišť ostatních. I při splnění těchto podmínek však bylo ve výše zmíněném systému zapotřebí „doladit“ vytvořený rozvrh ručně, po případě ponechat rozhodnutí o dalším výběru práce na pracovišti člověku, který za pomoci počítačem zpracovávaných a nabídnutých podkladů, front prací, aktualizoval rozvrh.

Proto byl zvolen následující postup: Fronty prací na pracovištích vytvoří již krátkodobé operativní plánování. Fronty prací definují rozvrh prací. Rozvrh je pak zpřesňován rozvrhovacími algoritmy. Cílem rozvrhovacích algoritmů je na základě odhadu budoucího průběhu výroby získaného simulací vytvořit dostatečnou zásobu práce pro všechna pracoviště na celé plánovací období. Rozvrh je konstruován tak, aby běžné odchylky skutečného průběhu výroby od průběhu plánovaného nevedly k nadměrným prostojům. Zkušenost ukázala, že takto formulovaný cíl byl reálný a že takto koncipovaný rozvrh definuje takový průběh výroby, který byl blízký optimálnímu.

V reálné situaci bývá často nutné do průběhu výroby zasahovat operativně. Důvodem je např. vznik urgentních neplánovaných požadavků a podstatnějších odchylek od plánovaného průběhu. V takovém případě bylo výhodnější nespouštět ihned algoritmy plánování, které mohou pracovat příliš dlouho a vyžadovat příliš mnoho dat, ale dát dispečerovi systému možnost nové operace zařadit příkazem z obrazovky přímo do front prací. Řešení tedy bylo v optimální kombinaci automatizovaných a ručních prostředků.

Aparát „ručních zásahů“ byl zároveň použit jako „prototyp“ plánovacích a rozvrhovacích algoritmů, což podstatně usnadnilo oživování systému. Je tedy nejvýše žádoucí, aby byl požadavek „ručních“ zásahů do



Obr. 21.2: Schéma vazeb mezi řízením dílny a podnikovým IS.

rozvrhu zahrnut do požadavků na systém[1]. Fronty práce vytvořené rozvrhem obsahují i takové práce, které jsou v okamžiku rozvrhování neproveditelné, neboť např. závisí na technologicky předcházející operaci, která ještě nebyla dokončena. Takové operace tvoří potenciální zásobu práce. Při výrobě se operace v závislosti na dokončení předchozích operací přesunují z potenciální zásoby práce do skutečné zásoby práce. Žádá-li pracoviště P práci, vybere se prvá proveditelná práce z fronty na P. Pořadí prací ve frontě může být změněno dispečerem systému. Dokončení operace na pracovišti je významným řídicím signálem, který je interpretován jako žádost o další práci. (viz obr. 21.2). Při výpadku pracoviště P jsou práce čekající na P přesunuty do fronty na jiné pracoviště $P1$, které mohlo práce převzít.

Hlavní výhoda navrženého postupu je v tom, že z hlediska dispečera a také okolí dílny se systém chová a je řízen obdobně jako v případě, kdy by nebyl používán IS. Úkolem dispečera je zajišťovat práci pro pracoviště. Pokud IS není používán, zjistí dispečer příliš pozdě, že pracoviště P nebude mít práci. Pak se obvykle nepodaří optimálně pracoviště vytížit a dochází k prostojům. S IS může dispečer reagovat s dostatečným předstihem. Metoda jeho práce se tedy nemění – je to v podstatě tzv. řízení „hašením průšvihů“. Tato zdánlivá maličkost je velmi významná. V KS PVS byl dispečer schopen spolupracovat s IS ihned, neboť se principy jeho práce nezměnily. Došlo k optimální kombinaci možností IS (evidence, zobrazování front) a dispečera (znalost řešení problémů, využití jeho znalostí a zkušeností).

Je samozřejmé, že s postupným zpřesňováním dat lze roli dispečera omezovat nebo, což je výhodnější, jeho zásahy více kombinovat s optimalizačními algoritmy. Nebývá to ale při malosériových výrobách, např. při

1. Další výhodou tohoto řešení bylo, že nevyžadoval restrukturalizaci činností, protože to nebylo nutné. Přijaté řešení však nevylučuje změny náplně činností, jsou-li potřeba.

výrobě obráběcích strojů výhodné. Teoreticky lze dospět zdokonalováním systému do situace, kde nebude dispečer potřeba. Dosáhnout tento cíl je velmi obtížné teoreticky a nereálné prakticky.

Data rozvrhu v KS PVS obsahují záznamy pro jednotlivé operace na pracovištích: na čem, kolik, odhad výrobního času, údaje umožňující zařazování operací do front a sledování návaznosti operací, požadované termíny provedení a stav – čeká na dokončení předchozích operací – čeká na nářadí – připravena – v práci – provedena.

Je překvapující, jak bylo někdy obtížné přesvědčit uživatele KS-PVS, že je nerozumné používat komplikované algoritmy pro nepřesná nebo neaktuální data z technologických norem a tyto algoritmy spouštět v reálném čase. Bylo také obtížné přesvědčit uživatele o výhodnosti ručních zásahů do rozvrhu jak z hlediska cílových funkcí systému, tak pro potřeby oživování. Pokud by byl uživatel svoje názory prosadil, bylo by došlo k podstatnému nárůstu pracnosti vývoje a zhoršení kvality systému. To možná vysvětluje překvapivý fakt, že je nevýhodné, aby specifikace formuloval uživatel sám bez spolupráce s informatiky a dodavatelem IS.

Při specifikaci požadavků je důležité uvážit psychologii obsluhy. Práce v systému by měla vyžadovat od obsluhy jen takové reakce, které jsou z jejího hlediska přirozené. Proto např. není v KS PVS signál o ukončení operace na pracovišti odvozován od stisknutí tlačítka, na které může pracovník snadno zapomenout nebo je stisknout v nevhodnou dobu, ale od pohybu palety. Podobně je požadavek na odvoz přepravní palety odvozován od pohybu palety, tj. od signálu o vložení palety do místa určeného pro odvoz.

Pracovníci PVS nesmějí mít dojem, že je jim systém nepřátelský, nebo že je pro ně práce v PVS nevýhodná. Špatný vztah k PVS může být vyvolán přílišnou intenzitou práce nebo snahou přesně a zbytečně kontrolovat každý pohyb pracovníků. Ti se v tom případě často snaží systém oklamat a často se jim to daří. Důsledky mohou být fatální. Tyto kontrolní funkce proto KS-PVS nezajišťuje.

Výše uvedené, zdánlivě málo ambiciózní požadavky vedly k realizaci systému, který se plně osvědčil[2], a kupodivu přinesly zvýšení výkonu dílny o 200 %. Poněvadž řízení dílny „na průšvih“ bylo konformní s řídicími zvyklostmi zbytku továrny, nebyla automatizovaná dílna pociťována jako cizí těleso a byla proto plně využívána[3]. Management továrny se záhy naučil využívat toho, že IS dílny obsahoval spolehlivá, aktuální a snadno využitelná data. Přesto, že se data týkají pouze prvovýroby, jsou využívána ke sledování stavu rozpracovanosti zakázek. Výše uvedené zásady kladou minimální nároky na podnikovou úroveň, která samozřejmě musí být schopna zajistit potřebná data. To ale není problém, poněvadž se jedná o data, která jsou nutná i pro konvenční způsob řízení. Pro další výklad je důležitý fakt, že součástí dílny je automatizovaný dopravní systém (regálové nakladače, indukční vozíky) a mezioperační sklad.

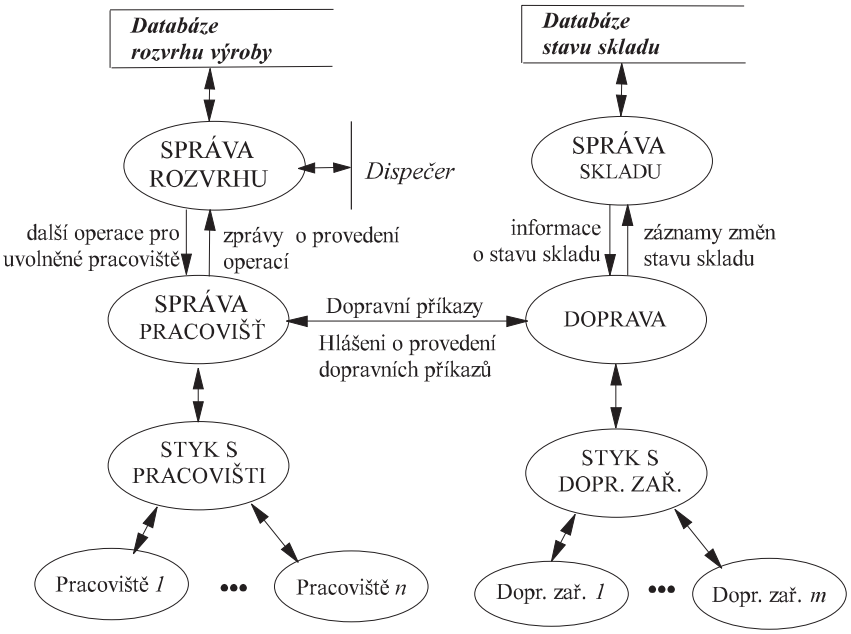
21.2.2 Architektura IS

Systém KS PVS zpracovává signály o událostech v řízené soustavě a s využitím rozvrhu a informací o situaci v dílně převádí signály na příkazy, které pak automaticky provádí.

V KS-PVS se osvědčila dekompozice do následujících komponent (obr. 21.3):

1. Styk s řízenou soustavou je koncentrován do následujících subsystémů. Část komunikující s dopravními zařízeními je oddělena od části komunikující s pracovišti (příslušné ovladače – drivery – jsou různé a bylo proto výhodné navrhnout separátní prototypy pro pracoviště a pro každé dopravní zařízení). Příslušné subsystémy jsou STYK S DOPR. ZAŘ a STYK S PRACOVIŠTI. Oba moduly společně budeme označovat STYK.

2. KS PVS pracuje již deset let k plné spokojenosti uživatele. Jediný problém představuje fyzické i morální opotřebení řídicího počítače.
3. Poznamenejme, že toto řešení nevylučuje použití libovolné plánovací strategie na vyšší úrovni řízení. Dobré plánování se, jak jsme uvedli výše, projeví tím, že bude málo případů, kdy musí zasahovat dispečer.



Obr. 21.3: Struktura KS-PVS.

2. Operace s frontami je soustředěna do subsystému SPRÁVA ROZVRHU. Jeho hlavním úkolem je zanést do rozvrhu změny vyvolané dokončením práce na pracovišti P a odeslat informace o další operaci pro pracoviště P.
3. Vyhodnocování situace na pracovištích provádí na základě signálů z pracovišť subsystém SPRÁVA PRACOVIŠŤ. Subsystém převádí signály z pracovišť na příkazy pro subsystém DOPRAVA tvaru „odvez z místa M paletu P“, „přivez na místo M paletu P“. Ze situace na pracovištích odvozuje systém zprávu „pracoviště P ukončilo práci na operaci“. Od subsystému SPRÁVA ROZVRHU přebírá zprávu „pracoviště P má provádět operaci O“.
   Subsystém DOPRAVA zpracovává příkazy „přivez/odvez na místo M paletu P“. Pro příkaz „odvez“ zjistí dotazem na subsystém SPRÁVA SKLADU tvaru „najdi místo“ adresu prázdného místa a odešle příkaz převozu „odkud, kam, co“. Po provedení akcí „nalož a vyložˮ se pomocí příkazů na místě M „naloženo/vyloženo“ aktualizuje v subsystému SPRÁVA SKLADU obraz stavu skladu.
4. Subsystém SPRÁVA SKLADU pracuje podle výše uvedených principů.
5. Subsystém SPRÁVA ROZVRHU musí spolupracovat, jak jsme viděli, s dispečerem. Bližší analýza potřeb ukáže, že spolupráce s dispečerem je potřeba i při řešení úloh subsystémů SPRÁVA SKLADU, DOPRAVA a SPRÁVA PRACOVIŠŤ. Proto je výhodné navrhnout subsystém styku s obrazovkou jako subsystém DISP, který může komunikovat se všemi ostatními subsystémy. DISP lze pak použít i jako prototyp (na obrázku není znázorněno).

Jednotlivé subsystémy byly koncipovány jako autonomně pracující aplikace – služby. Činnost každé aplikace je spuštěna příchodem takového požadavku, na nějž je aplikace schopna reagovat. Zpracování požadavku

spočívá v provedení operací nad datovou základnu a/nebo v odeslání dalších požadavků jiným aplikacím. Při realizaci systému mohou být aplikace SPRÁVA SKLADU, případně DOPRAVA koupeny jako samostatné celky a integrovány do systému. Obvykle je však nutné věnovat jistou práci realizaci rozhraní a customizaci. V našem případě musely být tyto subsystémy vyvinuty.

21.2.3 Datová základna

Zkušenosti z vývoje KS-PVS potvrdily důležitost správné a včasné volby obsahu dat, jejich logické struktury a metod práce s nimi. Systém spolupracujících aplikací umožňující poměrně snadnou realizaci datových abstrakcí řeší tento problém jen z části. I zde platí, že „kde nic není, ani aplikace nic nenajde“. Je proto důležité stanovit včas obsah dat, ne však nutně ihned jejich formu, a zvolit pokud možno jednotný datový model pro celý řídicí systém.

Pro správu dat je výhodné zvolit databázový systém s možností využití jazyka SQL a transakcí. Moderní databázové systémy umožňují:

- přístup k dané entitě prostřednictvím jednoho i více klíčů. Hodnota klíče může vyjadřovat nějakou podstatnou vlastnost prvku PVS reprezentovaného danou entitou, např. příslušnost frontě prací na pracoviště;
- existenci více entit se stejnou hodnotou klíče a možnost uspořádat je buď implicitně podle pořadí zařazování, nebo podle hodnoty některého dalšího ve větě obsaženého údaje. To umožňuje vytváření takových datových struktur, jako jsou fronty, posloupnosti technologických operací atd.

V KS-PVS je pro každou výrobní zakázku vložena do rozvrhu výrobní dávka (VD). VD je tvořena záhlavím, které mj. obsahuje počet plánovaných kusů a identifikátor VD, a posloupností výrobních operací (VO). VO mají atributy obsahující technologické informace o operaci. Ty jsou kopírovány z dat definujících technologický postup výroby příslušné součástky. VO dále obsahuje čtyři atributy: identifikaci pracoviště, na kterém se má operace provést, celé číslo určující pořadí ve frontě na dané pracoviště, identifikaci výrobní dávky a číslo určující pořadí operace v dávce. Pomocí těchto atributů lze měnit zařazení do front, přidávat či vylučovat technologické operace atd. Lze při tom s drobnými komplikacemi použít standardní relační databázi.

21.2.4 Výpadky

Výpadky systému jsou způsobovány výpadky nebo chybnou činností některého modulu PVS, případně chybou obsluhy. Způsob jejich řešení je jedním ze základních měřítek kvality řídicího systému a je rozhodující pro jeho provozuschopnost. Reakce na výpadek je netriviální záležitost, která je jednou z hlavních příčin složitosti realizace řídicích systémů a silně zvyšuje pracnost řešení. Proto bývá obtížné realizovat i zdánlivě jednoduché systémy. Základním předpokladem úspěchu je minimalizace důsledků výpadku určitého funkčního modulu na činnosti řídicího systému jako celku. Tato vlastnost je využitelná i při ladění bez přítomnosti řízené soustavy.

Architektura řídicího systému založená na principu komunikujících aplikací tuto podmínku splňuje. Činnost každé aplikace, přesněji její komunikaci s okolím, lze nahradit komunikací s obsluhou prostřednictvím terminálu. Například při výpadku automatického řízení zakladače ve skladu jsou příkazy pro zakladač zasílány obsluze, která zajistí jejich provedení „ručně“. Činnost ostatních modulů řídicího systému přitom není ovlivněna.

Další důležitou vlastností systému je schopnost automatického restartu po výpadku. Pro řešení tohoto problému bylo v KS-PVS důležité přijetí následujících dvou zásad:

- každá aplikace je odpovědná za provedení restartu v rámci své kompetence s tím, že v okamžiku restartu jsou vždy k dispozici aktuální údaje o fyzickém stavu řízené soustavy, jako jsou stavy aktivních manipulačních míst, zakladačů apod.,

- nesoulad mezi fyzickým stavem řízené soustavy a logickým stavem dat je řešen ve spolupráci s obsluhou.

Datová základna IS musí zajišťovat služby ochrany dat, zálohování, transakční zpracování a ochranu integrity dat při výpadcích. To v době realizace KS PVS silně znevýhodňovalo levné databázové systémy, které měly navíc tu nevýhodu, že nebyly vždy schopné se vyrovnat s rozsáhlými daty a požadavky na propustnost (rychlost vyřizování požadavků) a zajistit bezpečný souběžný přístup mnoha uživatelů (aplikací) k datové základně. Tyto problémy nebyly dodnes dostatečně dořešeny.

21.2.5 Uživatelské rozhraní

Jak je uvedeno v 21.2.2. subsystém DISP v KS PVS zajišťující styk s obsluhou musí komunikovat s většinou modulů v systému. DISP musí přijímat zprávy jednotlivých modulů dispečerovi systému a převádět je do formy vhodné pro zobrazení na obrazovce. Naopak zprávy od obsluhy musí převádět do formy vhodné pro adresáta a také z tvaru zprávy adresáta určit. Úkoly modulu DISP jsou tedy následující

1. Při vstupu od obsluhy je nutné z řetězce vstupních symbolů určit:
   - adresáta;
   - metodu dekódování;
   - řetězec dekódovat do binárního (vnitřního) tvaru.
2. Při výstupu zpráv je třeba:
   - zprávu převést do znakové formy;
   - odeslat zprávu na výstupní zařízení.
3. Poněvadž je nutné též archivovat zprávy pro případnou dodatečnou kontrolu činnosti obsluhy, je nutná archivace zpráv pro obsluhu příkazů obsluhy (kap. 11).
4. Zároveň je žádoucí u zpráv pro obsluhu žádajících odpověď kontrolovat, zda se odpovědělo a případně zopakovat zprávu, na kterou se zatím neodpovědělo.
5. Poněvadž subsystém styku s obsluhou je univerzálně použitelný a navíc se výběr zpráv dosti mění během vývoje systému, je vhodné požadovat snadnou modifikovatelnost zpráv a metod jejich kódování.

Řešení: Transformace zpráv je určena kódovacími tabulkami závisejícími na typu zprávy (kap. 11). Při výstupu má zpráva tvar:

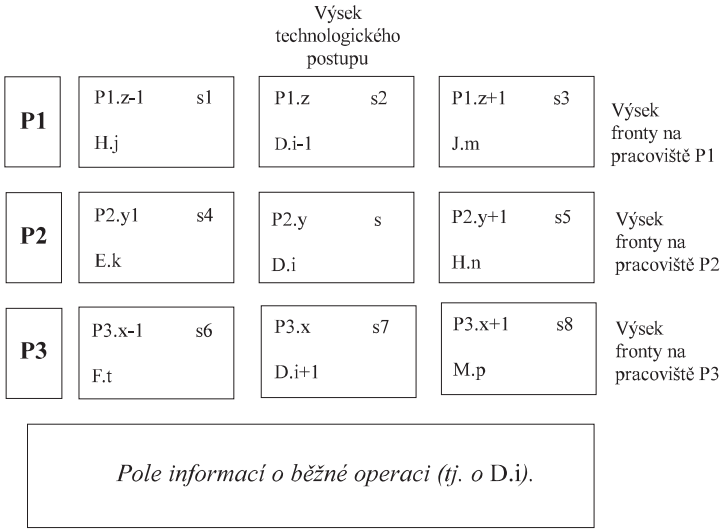
- typ zprávy,
- n dvojic: řetězec, podprogram kódování příslušného parametru.

Na obrazovce se objeví

1. Jméno zprávy
2. Pro každou dvojici
   - řetězec zadaný jako prvý člen dvojice,
   - výstup podprogramu specifikovaného druhým členem dvojice.

Při vstupu se provádí obrácená transformace.

Tabulku kódování je možné jednoduchým způsobem generovat. Poněvadž jsou podprogramy kódování / dekódování jednoduché, lze celý systém snadno přizpůsobit různým aplikacím. DISP realizuje kódování i dekódování zpráv. Dal by se tedy použít k testování jednotlivých aplikací, pokud bychom byli schopni zprávu místo adresátovi předat přes DISP obsluze a ta odeslala odpověď místo neexistujícího adresáta. DISP pak mohl v KS PVS fungovat jako prototyp dosud neexistujícího adresáta (srv. kap. 11). Zasílané zprávy lze archivovat v log. souboru spolu s časem odeslání zprávy. Tyto záznamy jsou účinným prostředkem ladění systému.



Obr. 21.4: Rozhraní usnadňující změny rozvrhu. Výhodné především při „řízení na průšvih“.

Velmi účinným grafickým prostředkem pro modifikaci rozvrhů KS PVS je grafické rozhraní z obr. 21.4. Na obr. 21.4 značí Di, DMo, DJm po řadě i-tou, j-tou, o-tou, m-tou operaci výrobní zakázky D (DM, DJ). Pu.v značí, že daná operace je ve frontě a pracoviště Pu na v-tém místě. Stav operace (dokončena, v práci, připravena, nepřipravena) je dán hodnotou atributu s. Hodnoty atributu s jsou na obrázku hodnoty s1, s2, atd. Operace uprostřed tabulky (operace (P2.y, s, Di)) je „běžná“. Prostřední sloupec na obrazovce tedy zobrazuje výřez z výrobního postupu obsahující aktuální operaci, řádky zobrazují úseky front prací na pracoviště uvedená nalevo.

Běžnou operaci lze změnit myší. Na větší obrazovce lze zobrazit více pracovišť a více míst ve frontách (např. tabulkou 5×5). V jednotlivých políčkách lze zobrazit i více dat (např. výrobní čas). Stav políčka lze vyznačit barvou. Pokud je to věcně možné, má dispečer právo měnit pořadí ve frontě a pracoviště příslušné operaci metodou „táhni a pusť“. Pro dispečera KS PVS bylo používání obrazovky přirozené, je intuitivně blízké tomu, nač byl zvyklý.

21.3 Zkušenosti z vývoje řídicích systémů

Poněvadž řídicí systémy zasahují do oblasti řízení, je žádoucí při návrhu systému zvážit, zda by se nevyplatilo předem realizovat některé menší změny v zaběhnutých zvyklostech. Často se tato možnost ani neuvažuje a hledají se důvody, obvykle falešné, proč to není možné. Je to přesně stejný přístup, jako bychom chtěli, aby numericky řízený soustruh byl obsluhován přesně stejně jako soustruh z minulého století. Stejně nebezpečné ovšem je měnit bez velice závažných důvodů zásadním způsobem organizaci práce a náplň činností. Toto nebezpečí je zvláště akutní u dovážených systémů. U KS-PVS se v tomto ohledu postupovalo racionálně.

Velice často vzniká dojem, že software je to, co zdržuje. To je zvláště typické u metody Stolové hory. Zkušenosti však učí, že příkladů, kdy věci selhaly z důvodu chyb v programování, je minimálně. O tomto faktu

jsme se již několikrát zmiňovali (kap. 1). To se potvrdilo i v KS PVS i u jiných autorem vyvíjených systémů. Zdrojem hlavních problémů byla nedořešená analýza systému jako celku, vazby na cizí subsystémy, nevyhovující hardware a nespolehlivá data a malý zájem a malá podpora ze strany managementu.

Nejčastější systémová závada je nedořešení vazeb mezi „ruční“ a „automatickou“ obsluhou. Nedořešení těchto vazeb vede k funkčním závadám, např. při dlouhodobějších výpadcích, nebo ke zbytečné práci, automatizuje-li se leccos, co lze dělat snáze ručně. Systém má být především spolehlivý a toho lze dosáhnout jen tehdy, neklademe-li si nerozumné cíle.

Že se nejedná pouze o teoretickou úvahu, o tom svědčí následující příklad. V projektu řídicího systému montážní linky tramvají se projektoval algoritmus automatického stavění zarážek, tj. zařízení, které zastaví posunovanou tramvaj na určeném místě montážního pásu. Montovaly se různé tramvaje, pro různé tramvaje se zarážka ustavovala na různých místech. Celý algoritmus nastavování zarážek byl zbytečný, poněvadž přesun tramvaje musí být z bezpečnostních důvodů vizuálně kontrolován na místě a obsluha může snadno nastavit narážku podle toho, co vidí.

Ještě bizarnější je případ relativně nespolehlivého regálového zakladače, který neustále hlásil neexistující chyby. Vývojáři systému se rozhodli některá chybová hlášení ignorovat na základě toho, že se z předchozích akcí zakladače dalo s velkou pravděpodobností, ne však s absolutní jistotou, usoudit, zda k chybě skutečně došlo či nikoliv. Algoritmus po roce provozování selhal a jen náhodou nebyl nikdo zraněn odlétajícími kusy železa z regálu, do kterého se zakladač (nosnost 1t) „opřel“ nakládacím zařízením. Vizuální kontrola byla přitom celkem snadná.

Při návrhu řídicího systému je třeba vycházet ze zásady, že se každá část systému může porouchat. Je proto třeba zajistit funkci systému i při výpadku některých jeho částí. Ty články systému, jejichž výpadek způsobí výpadek celku, nazveme kritické. V našem výše uvedeném příkladě řízení výroby je kritickou částí systém mezioperační dopravy, nejsme-li schopni zajistit, aby mohla být pracoviště zásobována náhradní dopravou (jeřáb, ještěrka). Řídicí systémy je nutné navrhovat tak, aby ty části, na kterých závisí chod celého systému, byly navrhovány s alespoň stoprocentní kapacitní rezervou. Zálohování počítače v KS-PVS bylo off-line, tj. zálohování bylo možné provést za 1–2 hodiny přenesením médií na záložní počítač, přepojením vstupů a výstupů a restartem.

Datová základna musí zajišťovat služby ochrany dat, zálohování, transakční zpracování a ochranu integrity dat při výpadcích. To silně znevýhodňuje levné databázové systémy, které mají navíc tu nevýhodu, že nejsou vždy schopné zajistit práci s rozsáhlými daty a požadavky na propustnost. Nebývají schopné zajistit bezpečný souběžný přístup mnoha uživatelů a aplikací k datové základně. Mnohdy je ale nutný kompromis. To byl i případ KS-PVS.

A

Profese informatik

Vývoj metod a způsobu využívání softwaru jednoznačně směřuje ke snižování podílu prací věnovaných etapám kódování a testování. Pracnost kódování a testování silně snižují nové prostředky vývoje softwaru, jako jsou vizuální vývojová prostředí, objektově orientované technologie, integrovaná vývojová prostředí, spolupráce aplikací, CASE atd. Kvalitní vývojové nástroje příznivě ovlivňují i rozsah údržby. U customizovaných systémů radikálně klesá rozsah údržby připadající na jednoho zákazníka. Potřeby kódování snižuje i možnost integrace aplikací třetích stran. Neklesají však podstatně nároky na účast pracovníků zákazníka i dodavatele při analýze, rozsah školení uživatelů, pracnost náběhu systému aj.

Výroba základního softwaru je silně koncentrována do několika mamutích firem. Důsledkem je, že potřeba pracovníků při vývoji základního softwaru výrazně klesla. Hlavní oblastí uplatnění informatiků je vývoj / customizace a provoz IS.

Při customizaci klesá výrazně pracnost všech etap životního cyklu s výjimkou etap specifikace cílů a specifikace požadavků a zčásti i etapy návrhu. To jsou však etapy, ve kterých se především rozhoduje *co* se má realizovat, a méně *jak* má být požadavkům vyhověno. Podstatně vzrůstá podíl prací, při kterých je nutná spolupráce se zákazníkem. Rychle rostoucí možnosti integrace aplikací třetích stran podporujících analýzu dat, integraci služeb globálních sítí a globálních informačních systémů lze využít pouze tehdy, účastní-li se vývoje IS i pracovníci se základními znalostmi manažerských věd, matematické statistiky a oboru aplikace, dnes nejčastěji ekonomie a finančnictví.

Při vývoji IS jsou tedy třeba odborníci se znalostmi informatiky a schopností zvládnout i základy znalostí jiných oborů.[1] Pro takové odborníky se někdy používá označení business information manager (BIM). Existují dvě varianty profesní orientace BIM

- pracovník s odbornou přípravou z oblasti aplikace s vyhovujícími znalostmi informatiky,
- informatik se základními znalostmi oboru aplikace nebo schopný tyto znalosti zvládnout.

Role BIM z výše uvedených důvodů stále roste. U oborů s delší tradicí využívání IS je tendence, aby roli BIM plnili spíše pracovníci profesně orientovaní na oblast aplikace. Příkladem takového vývoje je CAD.

1. Význam mezioborových znalostí roste i v jiných oborech, např. v genetice. Markantním příkladem je česká ekonomická reforma, ve které byla zřejmě podceněna role právního rámce fungování trhu. Pravděpodobně k tomu došlo i proto, že nebyli k dispozici ekonomové s právními znalostmi a právníci znalí ekonomických zákonů. Mnoho vládních ekonomů si asi význam právního rámce dodnes plně neuvědomilo. Podobně si význam mezioborových znalostí neuvědomují mnozí informatici, pro něž mnohdy svět mimo počítače jako by neexistoval.

V rozsáhlých IS podporujících práci managementu a také v nových oblastech aplikací informatiky se osvědčují spíše informatici. Příčinou není jen to, že se jedná o aplikace v oborech, kde je tradice používání počítačů poměrně krátká. U rozsáhlých systémů je důležitá kombinační schopnost, znalost možností informačních technologií a schopnost vytipovat a integrovat aplikace třetích stran, případně volit správně funkce při customizaci. Pro tuto činnost bývá vhodnější informatik. Musí však být schopen jednat s lidmi a zvládat myšlenkový svět oblasti aplikace. Důležitá je zkušenost z více realizací a samozřejmě i profesní příprava, ve které by neměly chybět přednášky umožňující pochopit i jiné myšlenkové světy, než je svět úzce chápané informatiky a počítačů. Mnohým absolventům informatických specializací vysokých škol se však často nechce za hranice říše počítačů do oblastí, kde musí chápat druhé a nejednou opatrně odvracet ne zcela domyšlené požadavky.

Počítačová kriminalita (Smejkal et al., 1995) je vhodným příkladem role BIM. Rychlý vývoj IT přináší rychlý vývoj počítačové kriminality jak po stránce skutkové podstaty, tak z hlediska technik provedení. Zločinci velmi často využívají nejnovější metody a postupy. Na to musí reagovat dostatečně rychle jak legistativa, tak soudy. Je nutné, aby soudy v procesech připouštěly nové typy důkazů a předmětů doličných, které často musí být v elektronické formě, a metody prokazování viny. Vazby počítačové kriminality na organizovaný zločin situaci dále komplikují. Boj s počítačovou kriminalitou nemůže být proto věcí jen právníka, který si podstatu některých kriminálních informatických technik dovede jen obtížně představit, ani informatika který naopak nemá dostatečné právnické znalosti. Úspěch je možno očekávat jen při spolupráci právníka se základními znalostmi IT a informatika se základními znalostmi práva.

Při aplikaci IS je nutné řešit i vysloveně systémové problémy, jako je např. návrh a realizace sítě, kde jsou znalosti informatiky důležité a stačí.

Shrneme-li, je nejvýše žádoucí, aby členy týmu vyvíjejícího nebo customizujícího IS byli pracovníci s následujícími předpoklady:

1. BIM s orientací na informatiku, profesně obvykle informatik. To bývají členové řešitelského týmu za dodavatele IS, případně informatici zaměstnaní u uživatele.
2. BIM s orientací na oblast aplikace – nejlépe zástupce zákazníka se vzděláním z oblasti aplikace a základními znalostmi informačních technologií.
3. Systémoví a provozní programátoři – „klasičtí“ informatici.
4. Vývojoví pracovníci účastnící se návrhu, kódování a testování – „klasičtí“ informatici.

Potřeba pracovníků s profilem 4 relativně klesá.

B

Revoluce v informačních technologiích

V současné době probíhá nová revoluce informačních technologií srovnatelná snad jen se zavedením polovodičů a programovacích jazyků třetí generace v začátku šedesátých let. Nositelem revoluce jsou moderní síťové technologie – širokopásmové sítě, Internet a rozvíjející se technologie, jako je WWW a jazyk Java. Vliv moderních síťových technologií je zásadní a bude dále vzrůstat. Moderní síťové technologie zásadním způsobem ovlivní nejen informační systémy, ale lidskou společnost jako celek. Bude možné, aby mnoho pracovníků mohlo pracovat doma. Sítě umožňují globalizaci informačních procesů na úrovni jednotlivých organizací, státní správy i celé lidské společnosti. Umožňují elektronický obchod podstatně zkracující řetězec prostředníků mezi výrobcem a koncovým spotřebitelem. Globalizují ekonomické procesy. Jsou nositelem revolučních změn v lidské společnosti (kap. 1).

Při vývoji softwaru umožní počítačové sítě plné využití všech výhod architektury spolupracujících aplikací a obohatí ji o další přednosti. Výkonná síť umožňuje ukládat na serverech nejen data, ale také aplikace. Pokud nejsou jednotlivé aplikace příliš rozsáhlé, je možné, aby je klient četl ze serveru. Je tedy žádoucí, aby byla úloha řešena souborem relativně malých spolupracujících aplikací. To umožní podstatně redukovat požadavky na vybavení klientů. Klient může na síti dobře fungovat bez pevných disků a rozsáhlých pamětí. Je proto možné navrhnout hardware klienta v konfiguraci, jejíž cena je zlomkem ceny standardního PC. Hlavním přínosem jsou úspory spojené se správou systému. Náklady na správu systému podstatně převyšují náklady na koupi klientských počítačů. Náklady na jedno pracovní místo se v USA odhadují na 8000 USD ročně. Koncepce síťového počítače tyto náklady snižuje tím, že se de facto udržuje pouze jedna verze softwaru na serveru a nikoliv mnoho verzí na všech klientech. Uživatelé však nelibě nesou omezení své autonomie a mohou tím způsobit, že se síťové počítače nakonec neprosadí.

O softwarovém agentu jsme se již zmínili na více místech. Softwarový agent je autonomně pracující aplikace schopná putovat po síti (mobile computing), dynamicky měnit své chování, vytvářet své kopie a spolupracovat s jinými agenty a aplikacemi. Podobně jako v případě objektů v objektové orientaci existuje řada variant agentů. V obecném případě putuje agent jako přerušený (pozastavený) proces, tj. včetně dat a stavu výpočtu, pracující obdobně jako procesy v preemptivních multiúlohových operačních systémech (UNIX, Windows NT atd.). Pro spolupráci agentů jsou vyvíjeny standardy pro koordinaci činnosti agentů a formáty dat pro reprezentaci znalostí, např. jazyky KQML a KIF. To vytváří nové impulzy pro vývoj technologií spolupráce aplikací např. tím, že

bude možné potřebnou aplikaci[1] snáze najít kdekoliv na Internetu a s použitím standardů rozhraní snadno využít v daném IS. Systémy tedy budou skutečně otevřené. V současné době se provádí v této oblasti intenzívní výzkum řešící řadu fascinujících problémů. Začíná se hovořit o agentově orientovaných technologiích.

Agentová orientace spolu s obecnější metodikou propojování aplikací dále sníží potřebu klasicky orientovaných informatiků a zdůrazní potřebu znalostí z oblasti aplikace. Prvé komerční produkty pracující podle této filozofie mají velmi příznivé uživatelské vlastnosti, a to jsme zatím v situaci „průzkumu možností“. Po standardizaci rozhraní mezi agenty nebo aplikacemi bude s použitím Internetu možné kombinovat nepřeberné množství komponent. Změní se zásadní paradigma návrhu a realizace softwaru, který v multimediální formě a formou virtuální reality ovlivní pracovní procesy i zábavní průmysl. Důsledky tohoto vývoje se naplno a s dramatickými důsledky projeví v příštím tisíciletí.

1. Aplikace jsou v tomto případě většinou dosti malé a proto se pro ně využívá i termín komponenta a místo spolupráce aplikací se používá termín skládání komponent. Terminologie není zatím stabilní. Někdy je komponentou třída ve smyslu objektově orientovaných jazyků. Třída je obvykle příliš malá jednotka. Use case a framework (kap. 12) seskupuje objekty do celků s rozsáhlejší funkcionalitou.

Literatura

[1] ACMS, (1981), ACM Comp. Surveys, Issue on the Psychology of Programming. Vol. 13, No. 1.
[2] Adair, J., (1994), Vytváření efektivních týmů, Management Press, Praha.
[3] Albrecht, A. J., Gaffney, J. E., Jr., (1983), Software Functions, Source Lines of Code, and Development Effort Predictions: A Software Science Validation, IEEE Transactions on Software Engineering, Vol. SE-9, No. 6, 639–645.
[4] Anděl, J., (1993), Statistické metody, Matfyzpress, Praha.
[5] Andriole, S. J., (1995), Managing Systems Requirements. Methods, Tools, and Cases, McGraw-Hill.
[6] ANSI 84, (1984), ANSI/IEEE Software Engineering Standards, John Wiley, New York.
[7] ANSI94, (1994), IEEE Standards Collection. Software Engineering, 1994 Edition. IEEE Comp. Soc. Press.
[8] Arnold, R. S., (ed), (1996), Software Reingeneering, IEEE Comp. Soc. Press, Washington.
[9] Ashworth, G., (1993), An Introduction to SSADM Version 4, McGraw-Hill.
[10] Assesment, (1993), An Assesment of Space Shuttle Flight Software Development Process, National Academy Press.
[11] Augustine, N.R., (1979), Augustine's Laws and Major Systems Development, Defense Systems Management Rewiev, 50–76.
[12] Baar, M., (1986), Hodnocení kvality programového vybavení, Sdružení uživatelů JSEP a SMEP, Sborník č. 30, KSNP, Praha.
[13] Babich, P., (1992), Customer Satisfaction: How Good is Good Enough. Quality Progress, Dec 1992, 65–67.
[14] Baker, F. T., (1972), Chief Programmer Team Management, IBM Systems Journal 11, No. 1, pp. 56–73.
[15] Baker, F. T., Mills, H. D., (1973), Chief Programmer Teams, Datamation 19, pp. 58–61.
[16] Bank, J., Kruger, M. J., (eds), (1993), Software Engineering: Methods and Techniques, John Wiley – Interscience, New York.
[17] Barker, R., Longman, C., (1992), CASE* Method. Function and Process Modelling, Addison-Wesley.
[18] Barnes, J. G. P., (1995), Programming in Ada, 4. ed., Addison-Wesley, London.
[19] Barr, A., Feigenbaum, E., (1981), The Handbook of Artificial Intelligence, William-Kaufman, Los Alamos.
[20] Basl, J., (1997), Analýza současné nabídky softwaru pro podnikové informační systémy, Computer World (CZ), 1–2/1997, 19–27.

[21] Bass, B. M., Duntenman, G., (1963), Behaviour in Groups as a Function of Self, Interaction and Task Orientation, J. Abnorm. Soc. Psychology, Vol. 66, 419–428.
[22] Beck, L. L., Perkins, T. E., (1983), A Survey of Software Engineering Practice: Tools, Methods, and Results, IEEE Transactions on Software Engineering, Vol. SE-9, No. 5, 541–561.
[23] Bechforooz, A., Hudson, F. J., (1990), Software Engineering Fundamentals, Oxford U. Press, UK.
[24] Beidler, J., (1996), Data Structures and Algorithms. An Object-Oriented Approach Using Ada, Springer V., New York.
[25] Bell, D., Morrey, I., Pugh, J., (1986), Software Engineering, a Programmer Approach, Prentice-Hall, Englewood Cliffs.
[26] Bennatan, J., (1992), Software Process Management: A Practical Application. McGraw-Hill.
[27] Bigus, J. P., (1996), Data Mining with Neural Networks. Solving Business Problems from Application Development to Decision Support, McGraw-Hill.
[28] Bjørner, D., Fisher, K., (1981), Špecifikácia Softwaru, Sborník SOFSEM ’81.
[29] Bjørner, D., Jones, C. B., (1982), Formal Specification and Software Development, Prentice-Hall, Englewood Cliffs.
[30] Boeing Co., (1979), Software Cost Measuring and Reporting, US Air Force—ASD Document D180-22813-1, Jan 1979.
[31] Böhm, B. W., (1981), Software Engineering Economics, Prentice-Hall, Englewood Cliffs.
[32] Böhm, B. W., Brown, J. R., Lipow, M., Macleod, G., Merrit, M., (1978), Characteristics of Software Quality, North Holland, Amsterdam.
[33] Booch, G., (1991), Object-Oriented Design with Applications. Benjamin/Cummings.
[34] Booch, G., Bryan, D., (1994), Software Engineering with Ada, Addison-Wesley, Reading.
[35] Booch, G., (1995), Object-Oriented Analysis and Design with Applications, 2. ed., Benjamin/Cummings.
[36] Booch, G., Rumbaugh, J., (1995), Unified Method for Object-Oriented Development. Version 8.0. Metamodel and Notation, Rational Software Co.
[37] Booch, G., (1997), The Best of Booch, Prentice-Hall.
[38] Branson, M. J., Herness, E. N., (1992), Process for Building Object Oriented Systems from Essential and Constrained System Model: Overview. In Proceedings of the 4th Worldwide MDQ Productivity and Process Tools Symposium. Vol. 1 of 2, No. 4, 1992, IBM Tornwood.
[39] Brooks, F. P., (1995), The Mythical Man Month, 2. ed., Addison-Wesley.
[40] Buckley, F. J., (1996), Implementing Configuration Management, Hardware, Software and Firmware, IEEE Comp. Soc. Press.
[41] Buschmann, F., Meunier, R., Rohnert, H., Sommerland, P., Stal, M., (1996), Pattern-Oriented Software Architecture. A System of Patterns, John Wiley.
[42] Buckley, F. J., Poston, R., (1984), Software Quality Assurance, IEEE Transactions on Software Engineering, Vol. SE-10, No. 1, 36–41.
[43] Carnegie Mellon University. Software Engineering Inst., (1995), The Capability Maturity Model: Guidelines for Improving the Software Process, Carnegie Mellon U. Press.
[44] Clegg, D., Baker, R., (1994), CASE* Method. Fast Track. Addison-Wesley.
[45] Cline, M. P., (1996), The Pros and Cons of Adopting and Applying Design Patterns in the Real World, Communications of the ACM, Vol. 36., No. 10, pp. 47–49.
[46] Clockin, W., Mellish, C., (1982), Programming in PROLOG, Springer V., Berlin.

[47] Coad, P., North, D., Mayfield, M., (1997), Object Models: Strategies, Patterns, and Applications, 2. ed., Prentice-Hall.
[48] Coad, P., Yourdon, E., (1991), Object-Oriented Analysis, Prentice-Hall / Yourdon Press.
[49] Coad, P., Yourdon, E., (1991a), Object-Oriented Design, Prentice-Hall / Yourdon Press.
[50] Cockburn, A., (1996), The Interaction of Social Issues and Software Architecture, Communications of the ACM, Vol. 36., No. 10, pp. 40–46.
[51] Coleman, D., et all, (1994), Object-Oriented Development. The Fusion Method. Prentice-Hall.
[52] Coleman, D., Khanna, R., (1995), Groupware: Technology and Applications, Prentice-Hall.
[53] Collins, D., (1995), Designing Object-Oriented User Interfaces, Benjamin/Cummings.
[54] Compton, S. B., Conner, G. R., (1994), Configuration Management for Software, Van Norstrand Reinhold.
[55] Conger, S., (1994), The New Software Engineering, Wadsworth Publ., Belmont, Ca.
[56] Cook, M., (1996), Building Enterprise Information Architecture, Prentice-Hall.
[57] Cotter, S., Potel, M., (1995), Inside Taligent Technology, Addison-Wesley.
[58] Cougar, J. D., Zawacki, R. Q., (1978), What Motivates DP Professional, Datamation, Vol. 24, No. 9.
[59] Crowler, R., (1985), The MAP Specification, Control Engineering, Oct. 1985, 75–104.
[60] Černý, J., Dvořák, P., (1985), Technologie vytváření software řídících systémů průmyslových výrob, Acta Polytechnica Vol. 17 (IV-2), 75–104.
[61] Davis, M., Weyuker, E., (1983), Computability, Complexity, and Languages, Fundamentals of Computer Science, Academic Press, New York.
[62] Davis, W. S., (1983), System Analysis and Design: A Structured Approach, Addison Wesley, Reading Mass.
[63] Demner, J., Král, J., (1978), Towards Reliable Real-Time Software, in Constructing Quality Software, Hibbard, Schumann (eds), North Holland, Amsterdam.
[64] Demner, J., Král, J., (1977), Týmová realizace softwarových systémů, Sborník SOFSEM '77, VVS Bratislava, Dúbravská cesta 3, pp. 309–320.
[65] Desfray, F., (1994), Object Engineering, The Fourth Dimension, Addison-Wesley.
[66] Deutsch, M. E., (1982), Software Verification and Validation, Prentice-Hall, Englewood Cliffs.
[67] Dijkstra, E. W., (1976), A Discipline of Programming, Prentice-Hall, Englewood Cliffs.
[68] DoD, (1980), Requirements for Ada Programming Support Environment Stoneman, US Department of Defence.
[69] DOGA, (1980), Dokumentační a generační prostředek pro podporu strukturovaného programování, popis použití, VÚMS Praha, dokumentace systému DOS 4.
[70] Donovan, J. J., (1994), Business Re-engineering with Information Technology, Prentice-Hall.
[71] Druckman, D., Singer, J. E., van Cott, H., (eds), (1996), Enhancing Organizational Performance. National Academy Press.
[72] Harris, D. H., (ed), (1994), Linkages. Understanding the Productivity Paradox. National Academy Press.
[73] Ebenau, R. G., Strauss, S. H., (1994), Software Inspection Process, McGraw-Hill.
[74] Edwards, K., (1995), Real-Time Structured Methods. Structural Design. John Wiley.
[75] Eliëns, (1995), Principles of Object-Oriented Software Development, Addison-Wesley.
[76] Fagan, M. E., (1979), Design and Code Inspection to Reduce Errors in Program Development, IBM Systems Journal, Vol. 15, No. 3, 182–211.
[77] Fayyad U., et al, (1996), Advances in Knowledge Discovery and Data Mining, MIT Press.
[78] Fairclough, J., (ed), (1996), Software Engineering Guides, Prentice-Hall.

[79] Fensel, D., (1995), The Knowledge Acquistion and Representation Language, Karl Kluwer Publ.
[80] Ferdinand, A. F., (1993), Systems, Software, and Quality Engineering. Applying Defect Behaviour Theory to Programming, Van Norstrand-Reinhold.
[81] Flecher, T., Hunt, J., (1993), Software Engineering and CASE, Addison-Wesley.
[82] Follprecht, J., (1986), Řízení strojírenských provozů, SNTL, Praha.
[83] Frances, N., Forman, I., (1995), Interacting Processes, Addison-Wesley.
[84] Friedman, L. S., (1984), Microeconomic Policy Analysis, McGraw-Hill.
[85] Friš, S. E., Timorjeva, A. W., (1954), Fyzika III, Nakl. ČSAV, Praha.
[86] Frolund, S., (1996), Coordinating Distributed Objects, An Actor Based Approach to Synchronization, MIT Press.
[87] Fučík, J., Král, J., (1986), The Hierarchy of Program Control Structures, The Computer J., Vol. 29, No. 1, 24–32.
[88] Fučík, J., Pavelka, J., (1981), Simulace a ladění v reálném čase, SOFSEM '81, VUSEIAR, Bratislava.
[89] Fuggetttayes, A., Wolf, A., (eds), (1995), Software Process, John Wiley.
[90] Gane, C., Sarson, T., (1979), Structured Systems Analysis. Prentice-Hall.
[91] GaGöté, R., (1996), OpenDoc, Small is Beautiful, Byte 2/96, p. 167.
[92] Garg, P. K., Jazayeri, M., (1995), Process-Centered Software Engineering Environments, IEEE Comp. Soc. Press.
[93] Gilb, T., (1977), Software Metrics, Wintrop Publ., Cambridge Mass.
[94] Glass, R. L., (1979), Software Reliabillity Guidebook, Prentice-Hall.
[95] Golberg, R., Lorn, H., (1982), The Economic of Information Processing, Vol. 1, 2, John Wiley, New York.
[96] Gomaa, H., (1983), The Impact of Rapid Prototyping on Specifying User Requirements, ACM Software Engineering Notes, Vol. 8, No. 2, pp. 17–28.
[97] Graham, I., (1995), Migrating to Object Technology, Addison-Wesley.
[98] Grey, S., (1993), Practical Risk Analysis and Management, John Wiley.
[99] Grey, S., (1995), Practical Risk Assesment for Project Management, John Wiley.
[100] Griffin, E. L., (1980), Real-Time Estimating, Datamation, Jun 1980, 188–198.
[101] Grochow, J. M., (1997), Information Overload! Creating Value with New Information System Technology, Prentice-Hall.
[102] Groover, M. P., (1987), Automation, Production Systems and Computer Integrated Manufacturing, Prentice-Hall.
[103] Guinares, T., (1985), A Study of Application Program Development Techniques, Communications of the ACM, Vol. 28, No. 5, 494–499.
[104] Halevi, G., (1980), The Role of Computers in Manufacturing Processes, John Wiley, New York.
[105] Halstead, H. M., (1977), Elements of Software Science, North Holland, New York.
[106] Hannaford, S., Poyssick, G., (1996), Workflow Reengineering, MacMillan / Hayden.
[107] Hansen, H. L. (1984), Software Validation, North Holland, Amsterdam.
[108] Hantler, A., King, S., (1976), An Introduction to Proving the Correctness of Programs, ACM Comp. Surveys, Sept. 1976.
[109] Harris, D. H., (ed), (1994), Organizational Linkages: Understanding the Productivity Paradox, National A. of Sciences, New York.

[110] Hausmann, O., (1995), Omezující vliv tržního prostředí na kvalitu informačních systémů. Sborník konference System Integration, KIT VŠE Praha.
[111] Henderson-Sellers, B., (1996), A Book of Object-Oriented Knowledge: An Introduction to Object-Oriented Software Engineering, 2. ed., Prentice-Hall.
[112] Henderson-Sellers, B., (1996), Object-Oriented Metrics, Measures of Complexity, Prentice-Hall.
[113] Hewitt, C., (1977), Viewing Control Structures as Patterns of Passing Messages, Artificial Intelligence, Vol. 8, No. 3, 323–364.
[114] Hickman, L., Longman, C., (1995), CASE METHOD Business Interviewing, Addison-Wesley.
[115] Hoffmann, B., Krieg-Brüknes, K., (1993), Program Development by Specification Transformation, Springer V.
[116] Honzík, J., (1986), Programovací techniky, JZD Agrokombinát Slušovice.
[117] Horowitz, T., (1975), Practical Strategies for Developing Large Projects, Addison-Wesley, Reading.
[118] Hsu, C., (1996), Enterprise Integration and Modelling. The Metadabase Approach, Kluwer Publ.
[119] Humby, E., (1976), Programy na základě rozhodovacích tabulek, Alfa, Bratislava.
[120] Humprey, W., (1995), A Discipline of Software Engineering, Addison-Wesley.
[121] Chapin, H., (1984), Software Maintenance with Fourth Generation Languages, ACM Software Engineering Notes, Vol. 9, No. 1, 41–42.
[122] Christensen, K., Fitsos, G. P., Smith, C. P., (1981), A Perspective of Software Science, IBM Systems Journal, Vol. 20, No. 4, 372–387.
[123] IBM, (1980), Software Development, IBM Systems Journal, Vol. 19, No. 4.
[124] IEEE1094, (1992), IEEE 1094–1992 Standard for Software Quality Metrics Methodology, viz IEEE94.
[125] Ince G., (1995), Software Quality Assurance. A Student Introduction, McGraw-Hill.
[126] Jackson, M.A., (1975), Principles of Program Design, Academic Press, New York.
[127] Jackson, M.A., (1983), System Development, Prentice-Hall, Englewood Cliffs.
[128] Jacobson, I., Christensen, M., Jonson, P., Övergaad, G., (1995), Object-Oriented Software Engineering. Use Case Driven Approach, Revised Printing, Addison-Wesley.
[129] Janis, J. L., (1972), Victims of Groupthink, A Psychological Study of Foreign Policy Decision, Houghton Miffrim, Boston.
[130] Jamsa, K., (1984), Object Oriented Design Versus Structured Design, ACM Software Engineering Notes, Vol. 9., No. 1, 43–48.
[131] Jemeljanov, S. V., (1987), Upravlenije gibkimi proizvodstvennymi sistemami, modeli i algoritmy, Mašinostrojenie, Moskva.
[132] Jones, T. C., (1979), The Limits to Programming Productivity, Guide and Share Application Development Symposium, Monterey, SHARE Inc, New York.
[133] Kalášek, P., Pavelka, J., Šebek, V., Štrausová, M., (1982), Tvorba řídicích systémů pro rozsáhlé technologické celky, Sborník SOFSEM ’82, VUSEIAR Bratislava.
[134] Kan, S. H., (1995), Metrics and Models in Software Quality Engineering, Addison-Wesley.
[135] Karolak, D. W., (1996), Software Engineering Risk Management, IEEE Comp. Soc. Press.
[136] Karson, E-A., (1995), Software Reuse, John Wiley.
[137] Keen, P. G. W., Gambino, T. J., (1983), Building a Decision Support Systems: The Mythical Man-Month Revisited, in Building Decision Support Systems, J. L. Bennett (ed.), Addison-Wesley, Reading, 132–172.

[138] Kehoe, R., Jarvis, A., (1996), ISO 9000–3. A Tool for Software Product and Process Improvement. Springer V., New York.
[139] Kernighan, B. W., Lesk, M. E., Ossanna, Jr. J. F., (1978), Document Preparation, Bell System Technical Journal, Vol. 57, No. 6, 2115–2135.
[140] Khosfahan, S., Bucklewitz, M., (1995), Introduction to Groupware, Workflow, and Workflow Computing, John Wiley.
[141] Kiesler, S., Wholey, D., Carley, K., (1994), Coordination as Linkages. The Case of Software Development Teams. In Harris, D. H., (ed.), Organizational Linkages: Understanding the Productivity Paradox. Nat. A. of Sci.
[142] King, S., (1976), Symbolic Execution and Program Testing, Communications of the ACM, July 1976.
[143] Kleindienst, J., Plášil, F., Tůma, P., (1996), CORBA and Object-Oriented Services, in SOFSEM '96, Theory and Practice of Informatics, LNCS 1175, 74–95.
[144] Knuth, D., (1968), The Art of Computer Programming, Vol. 1, Addison-Wesley, Reading.
[145] Koontz, H., Weihrich, H., (1993), Management, Victoria Publ. Praha/ McGraw-Hill.
[146] Kraft, S., (1977), Programmers and Managers, Springer V., Berlin.
[147] Král, J., (1980b), Parkinson Programming, SIGPLAN Notices, Feb. 1980, 41–48. Hlavní teze též v článku: Strukturované i nestrukturované programování v poněkud parkinsonovském vydání, Informačné systémy 1–1976.
[148] Král, J., (1986), Software Physics and Paradigms, Proceedings of Information Processing Congress, North Holland, Amsterdam, 129–134.
[149] Král, J., (1986a), Empirical Laws of Software Development and their Implications, Computer Physics Comm., Vol. 41, 385–391.
[150] Král J., (1993), Software Team Size Dynamics. Duration and Effort Effects. Scripta Facultatis Scientiarum Naturalium. Computer Science and Applied Mathematics, Vol. 23/1993, 36–45.
[151] Král, J., Břicháček, V., Fiala, J., (1983), Psychologické problémy softwarového inženýrství, Sborník SOFSEM '83, VVS Bratislava, 235–265.
[152] Král, J., Černý, J., Dvořák, P., (1987), Technology of the FMS Control Software Development, Proceedings of WIMA, Inst. of Cybernetics, Berlin.
[153] Král, J., Demner, J., (1991), Softwarové inženýrství, Academia, Praha.
[154] Král, J., Kostečka, V., Franek, J., Moudrý, J., (1979), Software pro přímé řízení a regulaci, Sborník SOFSEM '79, 167–199, VVS, Bratislava.
[155] Kristen, G., (1994), Object Orientation. The KISS Method, Addison-Wesley.
[156] Kroenke, D., Hatch, R., (1994), Management Information Systems, 3. ed., McGraw-Hill.
[157] Kubeck, L. C., (1995), Techniques for Business Process Redesign, John Wiley.
[158] Landauer, T. K., (1995), The Trouble with Computers, MIT Press.
[159] Lano, K., Haugton, H., (1994), Reverse Engineering and Software Maintenance, McGraw-Hill.
[160] Leavitt, H. J., (1951), Some Effects of Certain Communication Patterns on Group Performance, J. of Abnorm. Soc. Psychol. Vol. 8., No. 1, 38–50.
[161] Leebaert, D., (1995), The Future of Software, MIT Press.
[162] Lehman, M. M., Belady, L. A., (1976), A model of Large Program, Program Development, IBM Systems Journal, Vol. 15, No. 3, 225–252.

[163] Lehman, M. M., (1980), Programs, Life Cycles and the Laws of Software Evolution, Proc. IEEE, Vol. 68, No. 9, 1060–1076.
[164] Levin, K., Lippit, R., White, R.K., (1939), Patterns of Agressive Behaviour in Experimentally Created 'Social Climates', J. Social Psychology, Vol. 10, 271–299.
[165] Levy, L. S. C., (1987), Taming the Tiger, Software Engineering and Software Economics, Springer V., New York.
[166] Lewis, T. G., (1991), CASE: Computer Aided Software Engineering, Van Norstrand Reihold.
[167] Likeš, J., Machek, J., (1983), Matematická statistika, SNTL, Praha.
[168] Lientz, B. P., Swanson, E.B., (1980), Software Maintenance Management, A Study of the Maintenance of Computer Application Software in 487 Data Processing Organizations, Addison-Wesley, Reading.
[169] Lirov, Y., (1997), Mission Critical Systems Management, Prentice-Hall.
[170] Lorenz, M., (1995), Rapid Software Development with Smalltalk, SIG Books.
[171] Lorenz, M., Kidd, J., (1994), Object Oriented Software Metrics. A Practical Guide, Prentice-Hall.
[172] Lyu, R. M. (ed.), (1996), Handbook of Software Reliability Engineering, IEEE Computer Soc. Press.
[173] Macaulay, L. A., (1996), Requirements Engineering, Springer V.
[174] Mankin, D., Cohen, S. G., Bikson, T. K., (1996), Teams and Technology Fulfilling the Promise of New Organization. Harward Business School Press.
[175] Manna, Z., (1981), Matematická teorie programů, SNTL Praha.
[176] Marca, D., McGovan, C. (1988), SADT Structured Analysis and Design Techniques, McGraw-Hill, New York (monografie o metodě SADT).
[177] Martin, J., McClure, C., (1983), Software Maintenance, Prentice-Hall, Englewood Cliffs.
[178] Martin, J., McClure, C., (1985a), Diagramming Techniques for Analysis and Programmers, Prentice-Hall, Englewood Cliffs.
[179] Martin, J., (1985b), System Design from Provably Correct Constructs, Prentice- Hall, Englewood Cliffs.
[180] Martin, M. P., (1995), Analysis and Design of Information Systems, Prentice-Hall.
[181] Mattison, R., (1996), Datawarehousing. Strategies Technologies, and Techniques, McGraw-Hill.
[182] McCabe, T. J., (1976), A Complexity Measure, IEEE Transactions on Software Engineering, Dec 1976, p. 308.
[183] McCall, J. A., Herdon, M. A., Osborne, W. M., (1983), Software Maintenance Management, National Bureau of Standards, NBS Special Publication, Oct 1985.
[184] McCue, G. M. (1978), IBM Santa Theresa Laboratory – Architectural Design for Program Development, IBM Systems Journal, Vol. 17, No. 1, 4–25.
[185] McKeown, P. G., Leitch, R. A., (1993), Management Information Systems, Academic Press
[186] Metcalf, M., Reid, J., (1990), Fortran 90 Explained, Oxford U. Press.
[187] Miller, W., (1987), Software Tools Sampler, Prentice-Hall, Englewood Cliffs.
[188] Mills, H., (1976), Software Development, IEEE Transaction on Software Engineering, Vol. SE-2, Dec 1976, 265–273.
[189] Mohanty, S. N., (1981), Software Cost Estimation: Present and Future, Software—Practice & Experience, Vol. 11, No. 2, 103–121.
[190] Molnár, Z., (1992), Moderní metody řízení informačních systémů, Grada, Praha.
[191] Mowbray, T. J., Zahavi, R., (1995), The Essential CORBA: System Integration Using Distributed Objects, John Wiley.

[192] Mullet, K., Sano, D., (1995), Designing Visual Interfaces. Sun Microsystems, Mountain View.
[193] Murray, W., (1995), The Visual C++ Handbook, 3. ed., McGraw-Hill
[194] Myers, W., (1975), Reliable Software Through Composite Design, Petrocelli, Charter, New York.
[195] Myers, W., (1976), Software Reliability, Wiley, 1976 (ruský překlad Mir 1980).
[196] Myers, W., (1995), Professional Awareness in Software Engineering, McGraw-Hill.
[197] Neugebauer, T., (1997), Bezpečnost práce v administrativě, Economia Praha.
[198] Nettesheim, H., (1982), Programmentwurf und Programmdokumentation, VDI Verlag, Düsseldorf.
[199] Neumann, P., (1995), Computer Related Risks, Addison-Wesley.
[200] Nielsen, J., (1993), Usability Engineering, Academic Press.
[201] Nielsen, J., (1994), Multimedia and Hypertext. The Internet and Beyond, Academic Press.
[202] Nierstrasz, O., Tsirchritzis, D., (1995), Object Oriented Software Composition, Prentice-Hall.
[203] O'Connell, F., (1994), How to Run Succesful Process. The Silver Bullet. Prentice-Hall.
[204] OOPSLA, (1986), Proceedings OOPSLA '86 Conference, ACM Sigplan Notices, Vol. 21, No. 11.
[205] Orr, H., (1979), The Theory, Design and Implementation of Structured Data Base, SIGMOD International Conference on Management of Data, ACM.
[206] Orr, K. T., (1977), Structured System Development, Yourdon Press, New York.
[207] Parnas, D. L., (1972), On the Criteria to be Used in Decomposing Systems into Modules, Communications of the ACM, Dec 1972.
[208] Parnas, D. L., (1975), The Influence of Software Structure on Reliability, in IEEE International Conference on Reliable Software, IEEE Comp. Soc. Press, New York.
[209] Parnas, D. L., Clemens, P. C., Weiss, D. M., (1985), The Modular Structure of Complex Systems, IEEE Transactions on Software Engineering, Vol. SE-11, No. 3, 259–266.
[210] Parr, F. N., (1980), An Alternative to the Rayleight Curve Model for Software: Development Effort, IEEE Transactions on Software Engineering, Vol. SE-6, No. 3, 292–298.
[211] Pascarelli, M., Quilter, D., (1993), Repetitive Strain Injury: A Computer User Guide, John Wiley.
[212] Paton, N., Wiliams, H. M., Cooper, R., Trinder, P. H., (1994), Database Programming Languages, Prentice-Hall.
[213] Perry, W., (1995), Effective Methods of Software Testing, John Wiley.
[214] Pham Hoang, ed., (1995), Software Reliability and Testing, IEEE Comp. Soc. Press.
[215] Pokorný, J., (1992), Databázové systémy a jejich použití v informačních systémech, Academia, Praha.
[216] Pokorný, J., (1994), Dotazovací jazyky, Science, Veletiny.
[217] Polanský, D., (1997), Struktura a obsah dokumentace projektu informačního systému, Computer World CZ 5/97, 15–17, 6/97, 10–11.
[218] Pomberger, G., (1986), Software Engineering and Modula 2, Prentice-Hall, Englewood Cliffs.
[219] Porter, L. W., Lawler, E. E., (1965), Properties of Organization Structure in Relation to Job Attitudes and Job Behaviour, Psychol. Bulletin, Vol. 64, 23–51.
[220] Pour, J., Voříšek, J., (eds), (1995), System Integration '95, VŠE Praha.
[221] Pour, J., Voříšek, J., (eds), (1996), System Integration '96, VŠE Praha.
[222] Pour, J., Voříšek, J., (eds), (1997), System Integration '97, VŠE Praha.
[223] Poyssick, G., Hannaford, S., (1996), Workflow Reengineering, Adobe Press.
[224] Pressman, R. S., (1992), Software Engineering: A Practicioner's Approach, 3. ed., McGraw-Hill, New York.

[225] Prívara, I., (1988), Progress – systém podporující návrh a vývoj programov, Sborník SOFSEM '88, VUSEIAR, Bratislava.
[226] Putnam, L. H., (1978), A general Empirical Solution of the Macro Software Sizing and Estimation Problem, IEEE Transactions on Software Engineering, Vol. SE-4, 345–361.
[227] Quilter, D., (1997), The Repetitive Strain Injury Recovery Book, Walk Publ. Co.
[228] Quinn, J. B., Peters, T., (1992), Intelligent Enterprise. A Knowledge and Service Based Paradigm for Industry. Free Press, Simon and Schuster.
[229] Rae, W., (1995), Software Evaluation for Certification, McGraw-Hill.
[230] Ránky, P. Q., (1986), Computer Integrated Manufacturing, Prentice-Hall, London.
[231] Reifer, D. E., (1996), Software Management, 4. ed., IEEE Comp. Soc. Press.
[232] Rembold, U., (1986), Computer-Aided Design and Manufacturing, Springer V., Berlin.
[233] Rumbaugh, J., Blaha, M., Premerlani, W., Eddy, F., Lorensen, W., (1991), Object-Oriented Modelling and Design, Prentice-Hall.
[234] Ryan, T. W., (1997), Distributed Object Technology and Applications, Prentice-Hall.
[235] Sanders, J., Curran, E., (1994), Software Quality, Addison-Wesley.
[236] Shael, T., (1996), Workflow Management Systems for Process Organizations, Springer V.
[237] Shaw, M. E., (1964), Communication Networks. In: Advances in Experimental Social Psychology, Academic Press, New York.
[238] Shaw, M. E., (1971), Group Dynamics, The Psychology of Small Group Behaviour, McGraw-Hill, New York.
[239] Shaw, M., Garlan, D., (1996), Software Architecture. Perspectives of an Emerging Discipline, Prentice-Hall.
[240] Shepperd, M., (1995), Foundations of Software Measurement. Prentice-Hall.
[241] Shneidermann, B., (1980), Software Psychology: Human Factors in Computer and Information Systems, Winthrop Publishers, Cambridge.
[242] Shooman, M., (1983), Software Engineering. Design, Reliability, Management. McGraw-Hill.
[243] Schäfer, W., (ed.), (1995), Software Process Technology, LNCS 913, Springer V.
[244] Scheer, A. W., (1994), CIM, Computer Integrated Manufacturing, 3. ed., Springer V.
[245] Schenot, R., (1995), How to Sell Your Software, John Wiley.
[246] Schneider, V., (1978), Prediction of Software Effort and Project Duration – Four New Formulas, SIGPLAN Notices, Vol. 13, No. 6, 49–59.
[247] Schulmeyer, G. G., (1990), Zero Defect Software, McGraw-Hill.
[248] Sigfried, S., (1995), Understanding Object-Oriented Engineering, IEEE Comp. Soc. Press.
[249] Simon, A. S., (1995), Strategic Database Technology: Management for the Year 2000, Morgan Kaufmann Publ.
[250] Sink, D. S., Smith, G. l. Jr., (1994), The Influence of Organizational Linkages and Measurement Practices on Productivity and Management, in Harris, D. H., Organizational Linkages: Uderstanding the Productivity Paradox. Nat. A. of Sci., New York.
[251] Smejkal V., Sokol T., Vlček M., (1995), Počítačové právo. C. H. Beck/SEVT Praha.
[252] Sochor, J., Richta, K., (1996), Projektování softwarových systémů, vydavatelství ČVUT, Praha.
[253] Software Productivity Consorcium, (1995), The Software Measurement Guidebook, International Thompson Press.
[254] Sommerville, I., (1992), Software Engineering, 4. ed., Addison-Wesley.

[255] Sommerville, I., (1996), Software Engineering, 5. ed., Addison-Wesley.
[256] Sprague, R. H., Watson, H. J., (1996), Decision Support for Management, Prentice-Hall.
[257] Spurr, K., Layzell, P., (eds), (1990), CASE on Trial, John Wiley.
[258] Spurr, K., Layzell, P., Jenisson, L., Richards, N., (1994), Software Assistance for Bussiness Process Reengineering, John Wiley.
[259] Standardy státního informačního systému, (1996), I. a II., Úřad pro státní informační systém, Praha.
[260] Stay, T., (1976), HIPO and Integrated Program Design, IBM Systems Journal, Vol. 15, No. 2, 1976.
[261] Steward C. J., Steward C., (1994), Interviewing Principles and Practices, Longman C., CASE Method: Business Interviewing, Oracle Co. UK. Ltd, Berkshire, UK.
[262] Strauss, S. H., Ebenau, R.G., (eds), (1994), Software Inspection Process, McGraw-Hill.
[263] Swanson, E. B., (1988), Information Systems Implementation, Bringing the Gap between Design and Utilization, Irwin, Homewood, Il.
[264] Symons, C. R., (1991), Software Sizing and Estimating. MkII Function Point Analysis, John Wiley.
[265] Šešera, Ĺ., Mičovsky, A., (1994), Objektovo orientovaná tvorba systemov a jazyk C++, Perfekt, Bratislava.
[266] Šilha, L., Chaos, pouštění žilou pomocí projektů informačních technologií, Blue Pages, LBMS Česká republika, 4/95.
[267] Tapscott D., (1996), The Digital Economy. Promise and Peril in the Age of Networked Intelligence. McGraw-Hill.
[268] Van Tassel, D. (1978), Program Style, Deisgn, Efficiency, Debugging and Testing, Prentice-Hall, Englewood Cliffs.
[269] Teichroew, D., Hershey, E. A., (1977), PSL/PSA: A Computer Aided Technique for Structured Documentation and Analysis of Information Processing Systems, IEEE Transactions on Software Engineering, Vol. SE-3, No. 1, 41–48.
[270] Thayer, R. H., (1995), Software Engineering Process Management, IEEE Comp. Soc. Press.
[271] Toffler, A., Toffler, H., (1994), Creating New Civilization. The Politics of the Third Wave, Turner Publ., Atlanta, G.
[272] Töpfer, P., (1995), Algoritmy a programovací techniky, Prometheus, Praha.
[273] Tracz, W. J., Computer Programming and the Human Thought Process, Software—Practice & Experience, Vol. 9, No. 2, 127–138.
[274] Trebovanija i specifikacii v razrabotke programm, sbornik statej (1984), Mir, Moskva. V této práci je podrobný popis metody SADT.
[275] Treble, S., Douglas, N., (1996), Sizing and Estimating Software in Practice, Making Mk2 Function Points Work, McGraw-Hill.
[276] Turtle, Q. C., (1994), Implementing Concurrent Project Management, Prentice-Hall.
[277] UNO, (1987), Recent Trends in Flexible Manufacturing, United Nations.
[278] Van de Velde, W., Perram, J. W., (1996), Agents Breaking Away, Springer V.
[279] Van Vliet, H., (1993), Software Engineering: Principles and Methods, John Wiley and Sons.
[280] Vaníček, J., (1995), Lze normalizovat jakost software?, Magazín ČSNI 10, s. 205–210 a 11 s. 225–234.
[281] Vaníček, J. (1996) Měření a hodnocení kvality softwarových produktů, Habilitační práce, St. Fak. ČVUT, Praha.
[282] Višňák, K., (1996), Ad: Otazníky systémové integrace, Computer World CZ, 7–41, 17–18.

[283] Vodáček L., Rosický A., (1997), Informační management. Pojetí, poslání a aplikace. Management Press, Praha.
[284] Voříšek, J., (1997), Strategické řízení softwarového produktu a systémová integrace, Management Press, Praha.
[285] Walker, M. G., (1987), Managing Software Reliability, The Paradigmatic Approach, North Holland, Amsterdam.
[286] Walston, G. E., Felix, C. P., (1977), A Method of Programming Measurement and Estimation, IBM Systems Journal, No. 1, 54–73.
[287] Warburton, R. D. H., (1984), Managing and Predicting the Cost of Real-Time Software, IEEE Transactions on Software Engineering, Vol. SE-9, No. 5, 562–569.
[288] Ward, P. T., Mellor, S., (1985), Structured Development for Real-Time Systems, Vol. 1 – Vol. 3, Prentice-Hall, Englewood Cliffs.
[289] Warnier, J. D., (1977), Logical Construction of Programs, Van Norstrad, New York.
[290] Weinberg, G., (1971), The Psychology of Computer Programming, Van Norstrand Reinhold.
[291] Weiss, D. M., Basili, V. R., (1985), Evaluating Software Development by Analysis of Changes: Some Data from the Software Engineering Laboratory, IEEE Transactions on Software Engineering, Vol. SE-11, No. 2, 157–168.
[292] Wheeler, D. A., (1996), Wheeler, D. A., Brykczynski, B., (1996), Software Inspection, An Industry Best Practice, IEEE Comp. Soc. Press.
[293] Wiener, L. R., (1993), Digital Woes: Why We Should not Depend on Software, Addison-Wesley.
[294] Wills, L., Newcomb, P., (1996), Reverse Engineering, Kluwer Publ.
[295] WIMA, (1987), Proceedings of the V. Bilateral Workshop GDR – Italy with International Participation, Dresden. 1987, Central Institute of Cybernetics and Information Processes of the Academy of Sciences GDR, Berlin.
[296] Wirth, N., (1976), Algoritms = Data Structures + Programs, Prentice-Hall, Englewood Cliffs.
[297] Witt, B. I., Baker, F. T., Meritt, E. W., (1994), Software Architecture and Design. Principles, Models, and Methods, John Wiley.
[298] Wolberg, J. R., (1983), Conversion of Computer Software, Prentice-Hall, Englewood Cliffs.
[299] Wolberg, J. R., (1981), Comparing the Cost of Software Conversion to the Cost of Reprogramming, SIGPLAN Notices, Vol. 16, No. 5, 104–109.
[300] Wood, J., Silver, D., (1995), Joint Application Development, 2. ed., John Wiley.
[301] Wooldridge, M., Müller, J. P., Tambe, M., (1996), Intelligent Agents, Agent Theories, Architectures, and Languages, Springer V.
[302] Wysocki, R. K., Beck, R., Crane, D. B., (1995), Effective Project Management: How to Plan, Manage and Deliver Projects in Time and within Budget, John Wiley.
[303] Yourdon, E., (1989a), Structured Walkthrougs, 4. ed., Prentice-Hall.
[304] Yourdon, E., (1989b), Managing structured techniques, 4. ed., Prentice-Hall.
[305] Yourdon, E., (1993), Object-Oriented System Design. An Integrated Approach, Prentice-Hall.
[306] Yourdon, E., Argila, C. A., Rivera, P., (1997), Case Studies in Object-Oriented Analysis and Design, Prentice-Hall.
[307] Yourdon, E., Whitehead, K., Thomann, J., Appel, K., Nevermann, P., (1995), Mainstream Objects. An Analysis and Design Approach for Business, Prentice-Hall.

[308] Zelkowitz, M. V., (ed.), (1995), Advances in Computers, Vol. 41, Academic Press.
[309] Zuse, H., (1990), Software Complexity: Measurements and Methods, Walter de Gruyter, Berlin.
[310] Zuse, H., (1994), Foundations of the Validation of Object-Oriented Software Metrics, in Theorie und Praxis der Softwaremessung, R. Dumke, H. Zuse, (eds), Deutscher Universitätsverlag, Wiesbaden, 136–214.

Rejstřík

K

L

M

N

O

P

W

Z

SCIENCE, s.r.o.

687 33 Veletiny 212
tel / fax +420 - (0)633-671160
e-mail – science@traveller.cz
http://www.science.cz

Prodejna:

Šumavská 33
612 00 Brno
tel +420 - (0)5 - 41532261
– beletrie + počítačová literatura
pondělí–pátek 8^{00}–16^{00}

Doposud vyšlo:

Distribuované systémy, výpočty v sítích – Motyčková . . . . . 240,– Kč
Dotazovací jazyky – Pokorný . . . . . 250,– Kč
Informační systémy – Král . . . . . 460,– Kč
Jak publikovat na počítači – Kolektiv autorů . . . . . 225,– Kč
Principy a problémy operačního systému UNIX – Skočovský . . . . . 300,– Kč
Programování sítí operačního systému UNIX – Stevens . . . . . 775,– Kč
Programové prostředí operačního systému UNIX – Kernighan, Pike . . . . . 350,– Kč
Referenční příručka jazyka C – Harbison, Steele . . . . . 370,– Kč
Vše o Internetu, Průvodce uživatele & katalog zdrojů – Krol . . . . . 540,– Kč
Vyšší škola objektového návrhu v C++ – Horstmann (+disketa) . . . . . 530,– Kč
X Window, grafické rozhraní operačního systému UNIX – Macur . . . . . 230,– Kč

– Dále nabízíme knihy českých nakladatelů: **Computer Press**, **UNIS**, **Kopp**, **CCB** a dalších.
– Knihy si můžete objednat poštou na výše uvedené adrese, ale také telefonicky, faxem nebo přes Internet.
– U poštovních zásilek neúčtujeme poštovné ani balné, pokud obsahují alespoň 1 náš titul, popř. jejich cena přesahuje 500,– Kč. Do této částky účtujeme poměrnou část, pod 100,– Kč celé náklady.
– Knihkupci a distributoři mohou počítat s obvyklými rabaty.
– Do 4–6 týdnů vám také zajistíme jakoukoli knihu nakladatelství:

  O'Reilly / Prentice Hall / Addison Wesley / Wiley / Springer Verlag / McGraw Hill
  Kluwer / Osborne / IEEE / Macmillan / SAMS / Hayden Books / New Riders
  QUE / ZD Press / Waite Group Press / Borland Press / Lycos Press. . .

– Zahraniční tituly jsou dováženy na základě individuálních objednávek. Na skladě je jen menší množství.
– Platíte pouze (katalogovou cenu × devizový kurs) + 10 % (bankovní poplatky, režie) + 5 % DPH.

Jaroslav Král
Informační systémy
Specifikace, realizace, provoz

Vydalo SCIENCE, Veletiny jako svou 11. publikaci;
odpovědný redaktor Zdeněk Vincenc;
odborná korektura Luděk Skočovský, Jiří Sochor;
jazyková korektura Barbora Antonová;
obálka Jan Hanuš;
sazba písmem Times sázecím systémem LaTeX Jaroslav Král, Petr Sojka;
360 stran; 109 obrázků; 23 tabulek; první vydání;
doporučená cena 460,– Kč