Ing. Karel Metzl
OKR Automatizace řízení,k.ú.o.,Ostrava

Ing. Josef Tvrdík
Ústav ekologie průmyslové krajiny ČSAV,Ostrava

ZDOKONALENÉ PROGRAMOVACÍ TECHNIKY (IPT)

1. Úvod.

Dříve, než jsme se odhodlali tento příspěvek napsat (nebo spíše dát dohromady), zvažovali jsme mnohokrát jeho užitečnost. O některých zdokonalených programovacích technikách (IPT - Improved Programming Technologies) se píše i v našich časopisech velice často. Mnohé z toho, čím se náš příspěvek zabývá, bylo probíráno i na předchozích havířovských seminářích. Zdá se nám však, že dosavadní pohled na IPT je příliš úzký a že jsou často pokládány pouze za součást strukturovaného programování. Ve svém příspěvku jsme se pokusili shrnout principy IPT a ukázat vztahy mezi jednotlivými technikami. Především bychom chtěli zdůraznit, že použití jednotlivých technik se navzájem nevylučuje, ale naopak podporuje a často je použití jedné techniky vázáno na použití jiné. Pokud je nám známo, v našem odborném tisku se tímto způsobem tématem IPT žádná práce dosud nezabývala a čerpali jsme proto z méně dostupné zahraniční firemní literatury. Příspěvek nevydáváme za své původní dílo. Je to kompilát z literatury na některých místech doplněný našimi názory a zkušenostmi z aplikace.

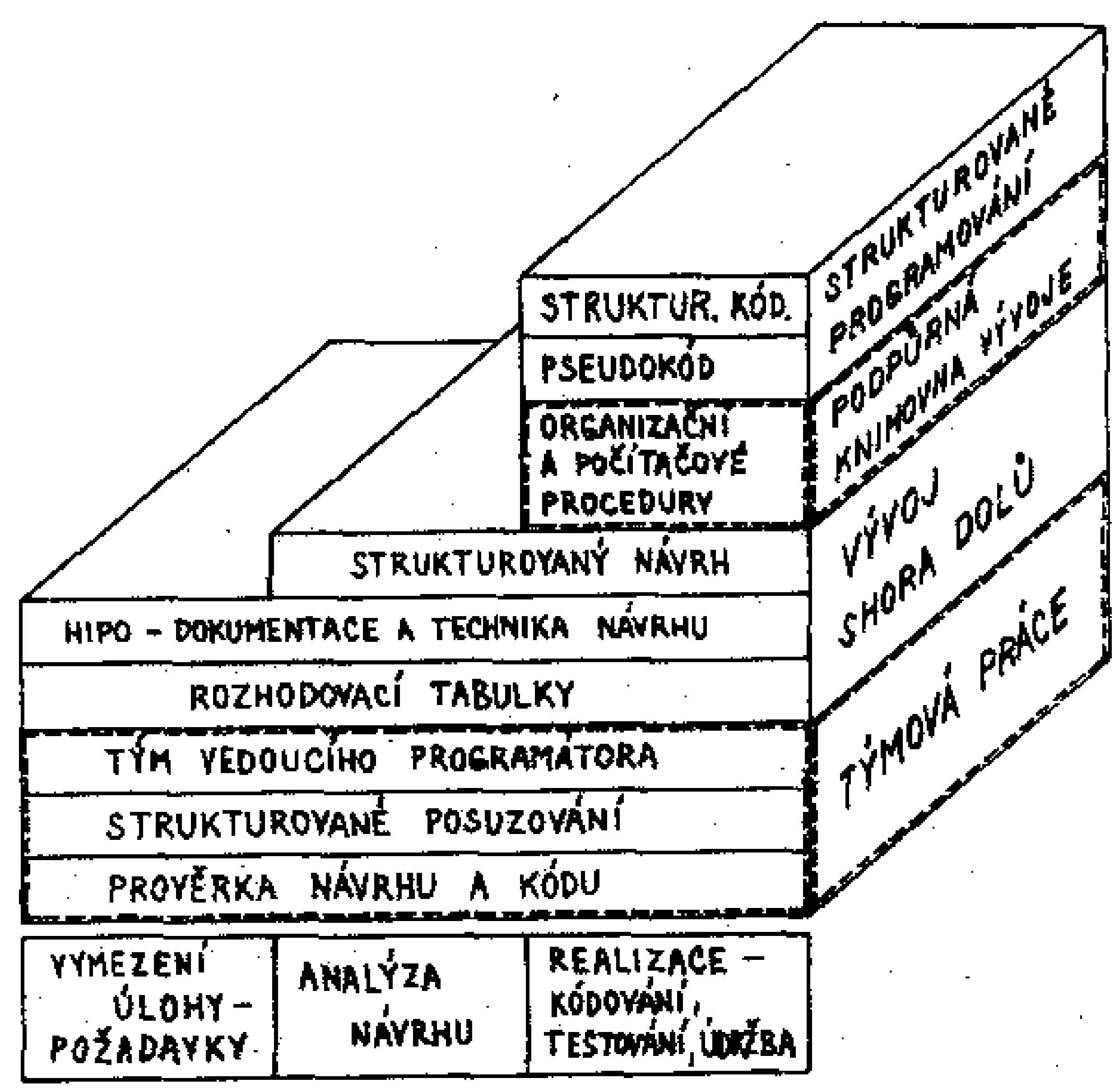
O příčinách vzniku zdokonalených programovacích technik se není potřeba příliš šířit. Vývoj hardware umožňuje stále složitější aplikace, uživatelé tyto složitější aplikace vyžadují. Rostou nároky na spolehlivost programového vybavení. Krátká životnost počítačů a nároky uživatelů přinášejí nutnost rychlého vývoje i velmi složitých programových systémů. Tato

úloha je zvládnutelná pouze organizovanou kolektivní prací. Značnou část pracovní kapacity programátorů (uvádí se často 70 %) pohlcuje údržba existujícího programového vybavení. Tím je vyvolána snaha o co nejvyšší efektivitu při vývoji nových a usnadnění údržby stávajících programů. Zdokonalené programovací techniky jsou racionalizačním prostředkem ve vývoji programového vybavení. Podotýkáme, že použití pojmu racionalizační prostředek je v málokteré jiné souvislosti tak výstižné jako v tomto případě.

Zdokonalené programovací techniky vznikly na základě analýzy procesu tvorby programového vybavení. Jsou motivovány snahou vytvořit pro jednotlivé části tohoto procesu, alespoň do jisté míry, formalizované postupy. Požadavek na formalizaci není přísný, spíše je zdravá snaha o co největší přístupnost a srozumitelnost pro účastníky procesu tvorby software (t. j. programátory, analytiky a uživatele). Důraz je kladen na rychlou a jednoduchou možnost aplikace.

Přehled prostředků IPT je uveden na obr. 1 [1]. Jak vidíme, lze prostředky rozdělit na <u>technické</u> (t.j. ty, které může využívat jedinec aniž by bylo nutné zavádět nějaká opatření v organizaci procesu vývoje software) a na prostředky <u>technicko-organizační</u> (v obr. 1 označeny čárkovaným rámečkem). Využití technicko-organizačních prostředků je podmíněno odpovídající organizací procesu vývoje programového vybavení. Individuelní využití některých částí je sice možné, ovšem za cenu nižší efektivnosti oproti využití hromadnému, podmíněnému organizací vývoje software.

Z obr. 1 je patrné, že prostředky IPT pokrývají nejen vlastní realizaci programů (t.j. kódování, testování a údržbu), ale i obě předcházející fáze tvorby programového vybavení, t.j. vymezení požadavků a fázi návrhu. Každá z těchto částí má svoji problematiku. S ohledem na zaměření zdejšího semináře určeného hlavně pro praktické aplikační programátory, budeme se ve svém příspěvku zabývat hlavně fází realizační a ostatních dvou fází se pouze okrajově dotkneme.



Obr. 1. Prostředky IPT

2. Týmová práce.

Vývoj aplikací pro jakýkoliv výkonější počítač je dnes nemyslitelný bez kolektivního zapojení mnoha pracovníků ve všech fázích vývoje programového vybavení. Jde nyní o to organizovat kolektivy účelně tak, aby pracovaly s vysokou efektivitou a vypracovaný kód splňoval všechny požadavky kladené na moderní software t.j. byl strukturovaný směrem shora dolů a vysoce přehledný.

Jednou z možných organizací, která byla prakticky vyzkoušena a popsána v literatuře, je zavedení hierarchicky uspořádaných týmů vedoucího programátora. Spolu s dalšími

metodami týmové práce jako jsou strukturované posuzování a prověrky návrhu a kódu, představuje změnu v přístupu od volně organizované skupiny pracovníků k vysoce strukturovaným týmům disciplinovaně pracujících specialistů.

2.1. Tým vedoucího programátora.

Tým vedoucího programátora je skupina pracovníků pracujících pod vedením zkušeného programátora-analytika nazývaného vedoucí programátor. Pevný kádr týmu je tvořen dalšími dvěmi členy: zástupcem vedoucího programátora a knihovníkem. Počet řadových členů je již různý a i v čase proměnný. Úzká spolupráce a tvůrčí prostředí týmu působí kladně na všechny členy, podporuje jejich odborný růst a pomáhá z nich vychovávat vedoucí či zástupce budoucích týmů.

2.1.1. Vedoucí programátor.

Vedoucí programátor musí být zkušený programátor-analytik schopný samostatné a iniciativní práce. Musí být důkladně seznámen s programovaným systémem a aktivně umět používat všech možností daných programovacími jazyky, systémy atd. Kromě vlastního administrativního i odborného vedení týmu sám detailně programuje kritické části a pro ostatní požadované části vypracovává specifikace. Nese plnou technickou odpovědnost za práci týmu ve všech směrech.

Již na první pohled je zřejmé, že povinnosti vedoucího programátora jsou značně rozsáhlé a různorodé. Vyskytly se proto názory, zda by ho nebylo možno některých zbavit. Je na příklad nutné, aby vedoucí programátor sám detailně kódoval? Při dnešní složitosti operačních systémů, přístupových metod, utilit, zdrojových jazyků atd. by vedoucí, věnující se pouze vedení týmu, ztratil kontakt s počítačem a schopnost vybrat ze široké škály nabízených možností tu nejvhodnější. Ztratil by tak postupně i autoritu u svých podřízených a schopnost technického vedení týmu. Nenahraditelná je však i jeho funkce vedoucího t.j. osoby odpovědné za

organizační disciplínu a dodržení rozpočtu, za výkonnost i kvalitu práce celého týmu.

2.1.2. Zástupce vedoucího programátora.

Pro znalosti a kvality zástupce platí stejná kriteria jako pro vedoucího programátora. Úzce s ním spolupracuje a musí být vždy připraven převzít jeho odpovědnost za vedení daného projektu, a to přechodně nebo i trvale. Stejně jako vedoucí programátor i on aktivně programuje. Často bývá pověřován různými specielními úkoly jako vyzkoušení alternativního řešení nebo vypracováním ladících dat atd.

2.1.3. Knihovník.

Knihovník je odpovědný za udržování nejzávažnější části celého projektu t.j. podpůrné knihovny vývoje. Z tohoto hlediska na něj nelze v žádném případě pohlížet jako na pomocnou kancelářskou sílu, ale je nutno ho považovat za plnohodnotného člena týmu - programátora. Kromě své základní funkce t.j. oddělení členů týmu od vlastního styku s počítačem a zbavení jich tak různých pomocných prací, bývá knihovník pověřován dalšími dodatečnými úkoly. Připravuje podklady pro strukturované posuzování, prověrky, sbírá různé statistické údaje pro vedení atd.

2.1.4. Členové týmu.

Každý tým je pružná jednotka, která může být doplňována dalšími programátory, analytiky či jinými specialisty v závislosti na rozsahu práce. Zjistí-li vedoucí programátor při rozboru shora dolů, že vznikajícími nároky by se tým neúnosně zvětšil, může přesahující část vyčlenit a přesunout do hierarchicky nižšího týmu vedoucího programátora k dalšímu detailnímu rozboru a kódování.

2.2. Strukturované posuzování.

Strukturované posuzování je vnitřní recenze vyvíjeného systému. Hlavním cílem je včasná detekce a odstranění chyb, rozšíření znalostí spolupracovníků o vyvíjené práci, možnost naučit je novým přístupům a technikám. Může být použito v různých fázích vývojového cyklu od celkového plánu projektu až po konečný produkt. Posuzuje se na příklad specifikace projektu, funkční návrh programu, detailní návrh programu včetně rozvrhu dat a spojení mezi moduly, výsledné kódy modulů, manuály pro uživatele a údržbu atd.

Základní charakteristiky strukturovaného posuzování jsou:

- Každá i dílčí práce všech členů týmu musí projít strukturovaným posuzováním.
- Vzhledem k tomu, že jde o vnitřní recenzi, organizuje posuzování sám autor. T.j. sestaví seznam účastníků (4 - 6) a určí termín a místo konání.
- Strukturovaného posuzování se zásadně neúčastní zástupci vedení a výsledků nesmí být použito jako podklad pro hodnocení zaměstnanců.
- Pro strukturované posuzování je předem určen přesně vymezený časový úsek, obvykle 1 až 2 hodiny. Není-li cíle dosaženo v tomto čase, určí se termín nové schůzky.
- Posuzovatelé obdrží posuzovaný materiál nejméně 4 až 6 dní předem, aby si ho mohli prostudovat a na schůzku si připravit dotazy.
- Při posuzování se klade hlavní důraz na vyhledávání chyb a slabých míst spíše než na jejich řešení a vůbec už nesmí jít o stanovení viníka.
- Obvykle je určen předseda, aby řídil posuzování a zaznamenával objevené chyby a nesrovnalosti. Konečný zápis je cenným dokladem pro autora, který na jeho podkladě provádí nutné úpravy.

Na začátku schůzky vystupují jednotliví posuzovatelé a vyjadřují své předem připravené názory na úplnost, přesnost a obecné kvality posuzované práce. Potom provádí autor

posuzovatele krok po kroku posuzovanou prací a simuluje jednotlivé funkce. Při výkladu se zabývá podrobněji úseky proti kterým byly vzneseny námitky a snaží se námitky vyvrátit. Vydatnou pomoc mu poskytuje HIPO dokumentace a různé složky podpůrné knihovny vývoje. Během autorova výkladu nacházejí posuzovatelé nové námitky a náměty pro řešení.

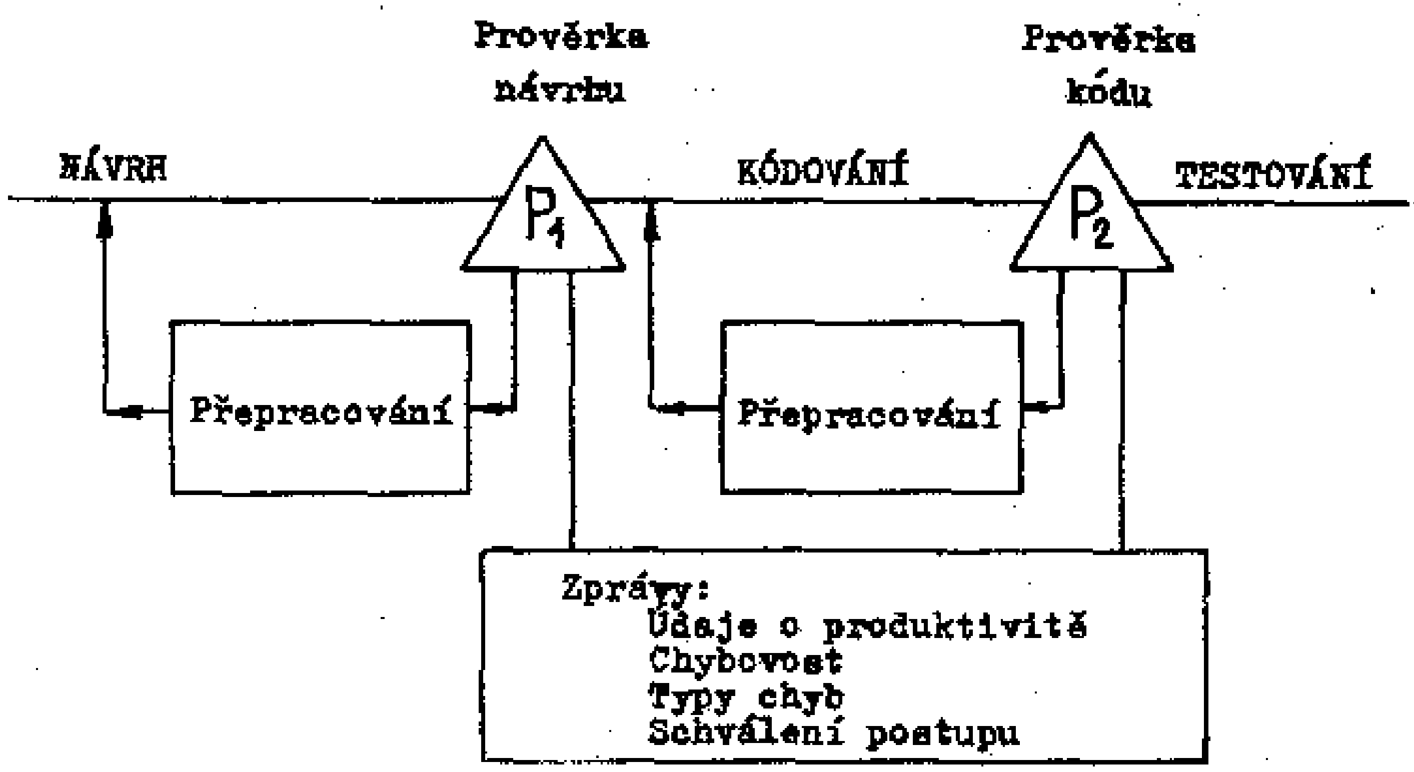
Základní podmínkou úspěchu strukturovaného posuzování je správný postoj všech zúčastněných. Od posuzovatelů se vyžaduje předběžná příprava, poctivé prostudování předložených materiálů a snaha vyhledat co nejvíce chyb a nesrovnalostí. Hlavní roli však musí sehrát autor. Musí vítat každou připomínku a povzbuzovat posuzovatele k námitkám. Takový postoj by se dal těžko očekávat od autora, který by věděl, že ho bude vedení hodnotit podle průběhu posuzování nebo podle rozsahu seznamu chyb a námitek.

Přestože hlavním cílem strukturovaného posuzování je objevit a "vychytat" co nejvíce chyb v úvodní fázi, kdy náklady na jejich opravy jsou nejnižší a jejich dopad je nejmenší, nelze podceňovat ani výchovnou a naučnou cenu posuzování. V týmech vedoucího programátora, kde se jako první posuzují moduly vypracované zkušeným vedoucím programátorem či jeho zástupcem, se strukturované posuzování stává školou nových přístupů i metod pro ostatní členy týmu.

2.3. Prověrka návrhu a kódu.

Prověrky, stejně jako strukturované posuzování, jsou určitým typem revize provedené práce. Zatím co posuzování je vnitřní recenze prováděná pouze členy týmu bez zásahu vedení, představují prověrky mnohem náročnější a disciplinovanější přístup k revidovanému systému. Prověrka je formálnější, má vyšší úroveň a poskytuje vedení různé kontrolní i souhrné zprávy, statistiky a indikuje kvalitu, kterou lze od prověřovaného systému v budoucnu očekávat. Umožňuje zasáhnout organizačně do vývojového procesu a stává se tak nástrojem pro řízení kvality a produktivity práce.

Rozeznáváme dva typy prověrek: prověrku návrhu a prověrku kódu. Jestliže práce kteroukoliv z prověrek neprojde, vrací se k opravám a modifikacím. V tomto případě je pak po přepracování nutná nová prověrka.



Obr. 2 Zařazení prověrek návrhu a kódu do vývojového procesu

Prověrka je řízena zpravidla čtyřčleným týmem inspektorů. Nejdůležitější osobou je předseda. Musí být zcela nezávislý na posuzované práci, jeho pohled na ni musí být objektivní, osvobozený od jakýchkoliv nátlaků. Nevyžaduje se, aby byl specialistou na posuzovaný systém, ale nezbytné jsou značné zkušenosti v oboru. Předseda koordinuje práci inspektorů, vede je a citlivě zajišťuje rovnováhu při provádění recenze. Na závěr prověrky shrnuje výsledky, připravuje zprávu pro vedení a s konečnou platností rozhoduje, zda práce splnila výstupní kriteria a může postoupit do další fáze, nebo zda se vrátí k přepracování. Kromě předsedy jsou v týmu inspektorů ještě autor prověřované práce, její realizátor, pracovník zajišťující ladění a eventuelně další nezbytní pracovníci logicky svázaných systémů.

3. Vývoj shora dolů.

Pojem "vývoj shora dolů" je užíván často ve dvou významech. Obecnější význam se týká postupu analýzy a funkční dekompozice úlohy ve fázi návrhu (obr. 1). Je možno jej označit jako uvažování shora dolů (top-down reasoning). Druhý význam pojmu vývoje shora dolů se týká postupu ve fázi realizační, t.j. postupu kódování, testování a integrace modulárního programového systému ve stadiu, kdy je už hotov návrh. Tímto druhým významem vývoje shora dolů resp. postupu vývoje programu shora dolů se zabýváme v odstavci 3.4.

3.1. Strukturovaný návrh.

Návrh programu je patrně nejtvořivější částí celého vývoje programového vybavení. Přestože existuje snaha stavit tuto činnost na pokud možno objektivní základ (jedna z možností je technika strukturovaného návrhu), zůstane zřejmě stále v navrhování programů jistý prvek umění, podobně jako existuje na př. v navrhování a konstrukci automobilů. Příčina je zřejmě v tom, že při návrhu programů je nutno pracovat s pojmy alespoň do jisté míry vágními na př. kvalita návrhu, jednoduchost návrhu, odpovídat struktuře problému atd. Publikované zjištění, že dva strukturované návrhy programů dvou různých autorů řešících nezávisle tentýž problém, se významně neliší ani strukturou ani počtem modulů, je sice dobrým vysvědčením o objektivnosti techniky strukturovaného návrhu, ale není to zjištění nijak překvapivé. Na automobilech dvou různých konstruktérů nás také nepřekvapuje, že obě auta mají kola, dveře, volant atd.

Strukturovaný návrh [1] je souhrn procedur a návodů, které slouží:

a. k demkompozici úlohy vymezené uživatelem v modulární návrh programu. Výsledkem tohoto postupu je modulární hierarchický diagram (někdy se nazývá strukturní schéma), který odpovídá obsahové tabulce HIPO sborníku

b. k identifikaci oblastí návrhu, ve kterých jiné uspořádání modulů nebo úprava struktury programu povede k "lepšímu návrhu".

Základním požadavkem strukturovaného návrhu je to, aby struktura programu odpovídala struktuře problému. Strukturovaný návrh je tedy výsledkem analýzy struktury problému, zvláště analýzy toku dat úlohou a transformací, ke kterým na těchto datech dochází. V tomto stadiu návrhář programu (programátor-analytik) by měl uvažovat o tom, <u>co</u> program musí dělat a nestarat se o to, <u>jak</u> nebo <u>kdy</u> má být v programu něco uděláno t.j. uvažovat funkčně, nikoliv procedurálně.

Připomeneme obsah několika pojmů důležitých pro techniku strukturovaného návrhu. <u>Modul</u> je úsek programu, který má externě známé jméno (na př. procedura v PL/I, programová jednotka ve Fortranu, CSECT v Assembleru). Modul má tři základní charakteristiky: logiku, interface a funkci. Pro strukturovaný návrh je důležitá pouze <u>funkce modulu</u>. Je to popis transformací, které nastanou, jestliže je modul volán. K tomuto popisu zpravidla vystačíme s větami obsahujícími sloveso a předmět. Na př. kontroluj vstupní parametry, tiskni výsledky, vypočti koeficienty korelace a p.

Prováděcí technikou strukturovaného návrhu je dekompozice založená na uvažování "shora dolů". Je to pětibodový postup obsahující analýzu struktury úlohy, analýzu toku dat úlohou a analýzu transformací vyskytujících se na datech:

1. Rozlož strukturu úlohy do několika (3 - 10) hlavních postupů.
2. V této struktuře úlohy identifikuj základní vstupní proud dat a základní výstupní proud dat.
3. Ve struktuře úlohy označ místo, ve kterém se vstupní proud dat ztrácí a místo, kde se výstupní proud dat poprvé objevuje.
4. Tyto dva body rozdělují strukturu úlohy do tří sekcí. Popiš tyto tři sekce jako tři funkce a definuj modul pro každou z nich. Tyto tři moduly se stanou podřízenými hlavnímu modulu programu. Nazývají se vstupní (source), transformační a výstupní (sink) modul.
5. Vezmi každý z těchto modulů, jeho funkci považuj za "podúlohu" a opakuj celý postup.

Tento pětibodový proces probíhá rekursivně, dokud celá úloha není rozložena do modulů realizovatelných 40 až 60 výkonými příkazy kompilovatelného jazyka.

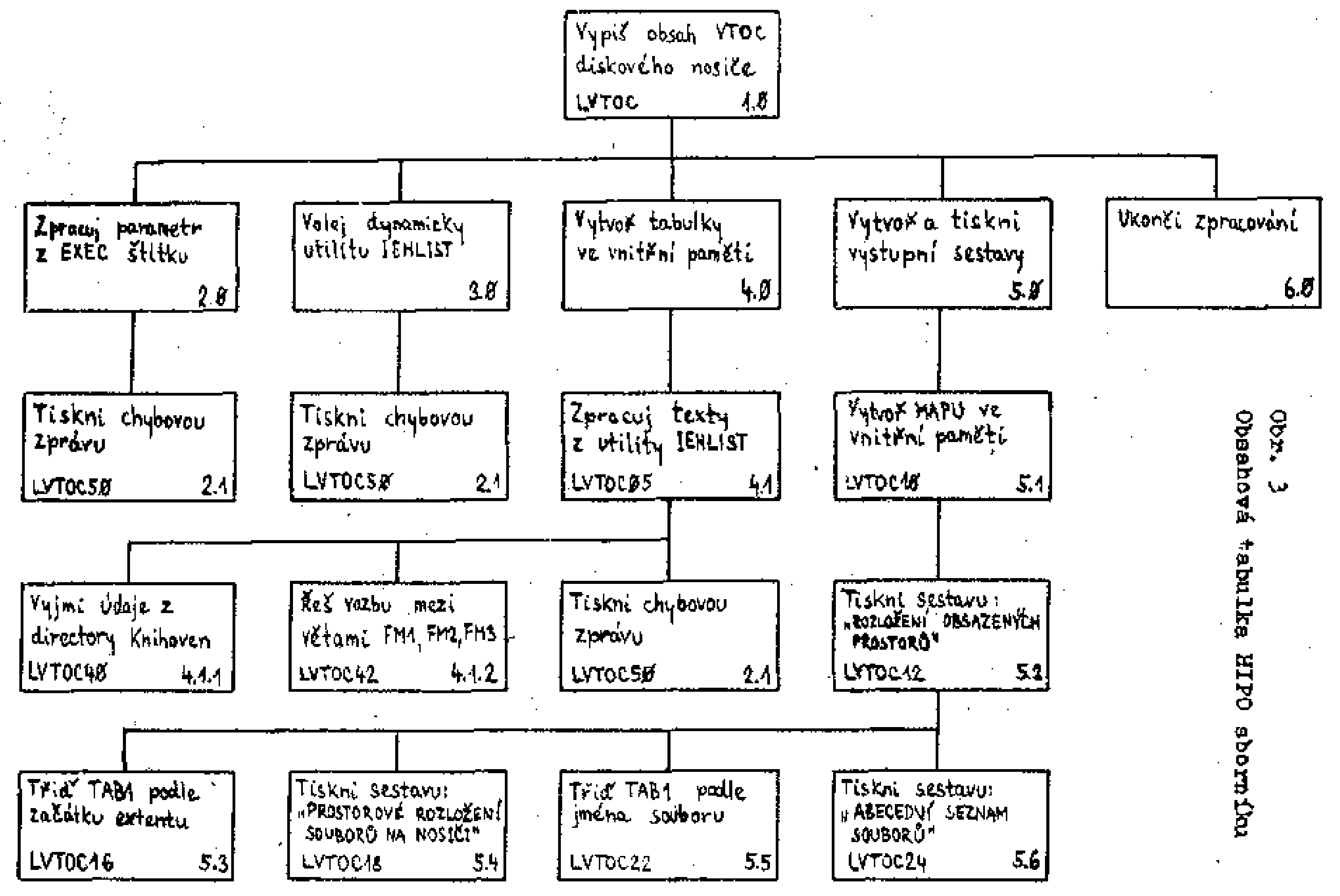
Kvalitu návrhu posuzujeme především z hlediska jeho jednoduchosti a z hlediska nezávislosti modulů. Obě kriteria jsou spolu svázána, neboť nezávislost modulů vede ke snížení složitosti návrhu. Výhoda takového jednoduchého návrhu s vysokou nezávislostí modulů je zřejmá: úloha je rozložena na nezávislé části. Tudíž programátoři mohou pracovat paralelně a tedy v souhrnu rychleji. Údržba a změny jsou snadnější, zpravidla se týkají jen jednoho nebo několika modulů. Změny v kterémkoliv modulu může provádět kterýkoliv programátor.

3.2. HIPO dokumentace a technika výstavby.

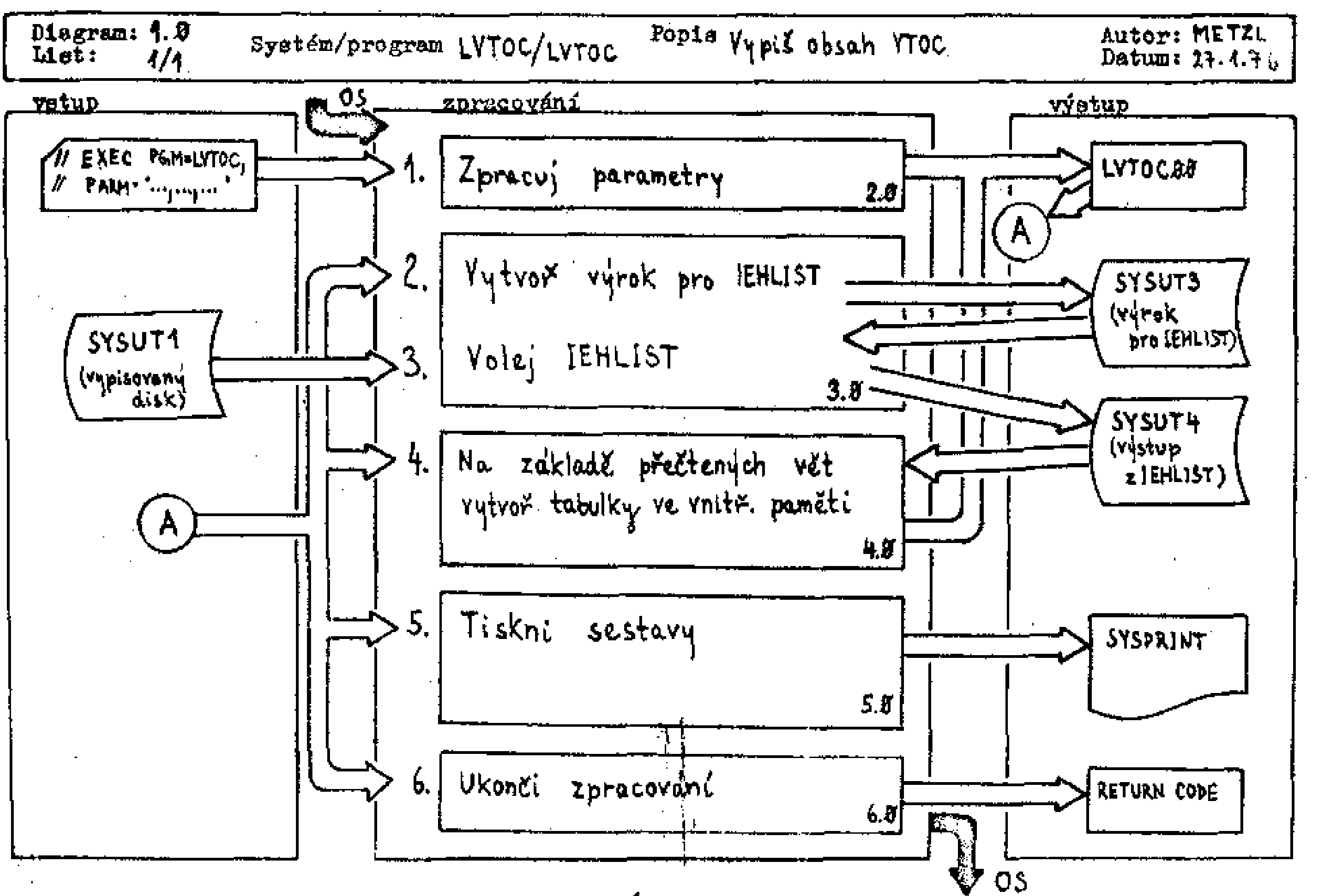
Dobrá dokumentace je důležitým předpokladem úspěchu každého projektu. Bohužel její vypracování se často odkládá až po skončení práce a pak se z důvodů časové tísně šidí. Při použití techniky HIPO vzniká dokumentace během celého vývojového cyklu, od stanovení požadavků až po konečný produkt jako vedlejší činnost procesu myšlení. Typický HIPO sborník se skládá z obsahové tabulky, přehledných i podrobných HIPO diagramů doplněných rozšířeným popisem event. jinými dokumentačními prostředky.

Obsahová tabulka znázorňuje formou organizačního schématu strukturu sborníku a vztahy funkcí v hierarchii. Její pomocí si může uživatel zvolit příslušnou úroveň informací, případně najít určitý diagram bez listování celým svazkem. Na stránce obsahové tabulky se uvádí i legenda k symbolům užitým v diagramech sborníků a obsah jednotlivých diagramů. Příklad obsahové tabulky je uveden na obr. 3.

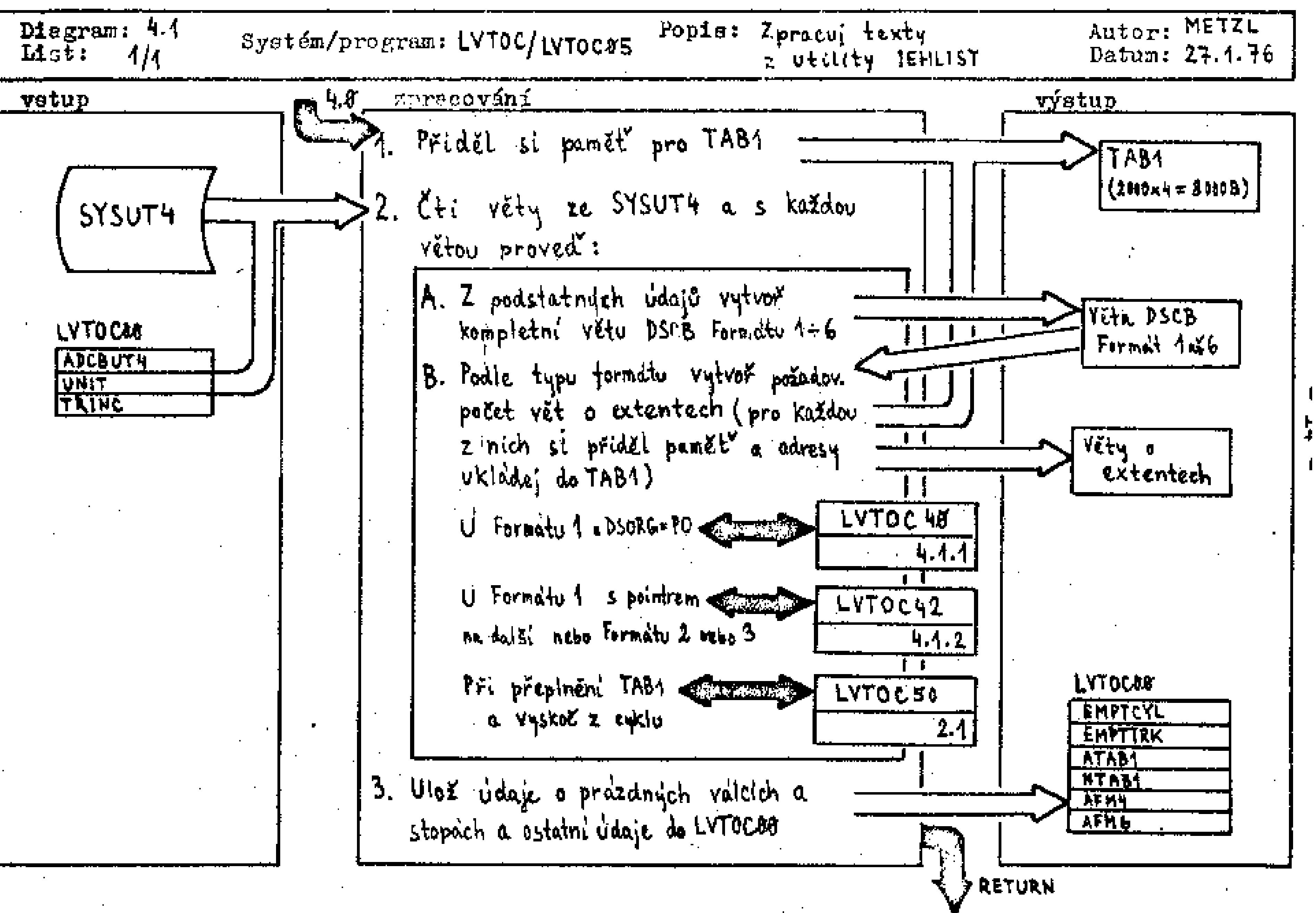
Přehledné i podrobné diagramy vyjadřují graficky požadované funkce spolu se vstupy a vznikajícími výstupy. Rozdíl mezi oběma typy diagramů je pouze v míře podrobnosti. Podrobné diagramy (zpravidla nižší hierarchické úrovně) bývají také



Obr. 3
Obsahová tabulka HIPO sborníku



Obr. 4 Ukázka přehledného HIPO diagramu



Obr. 4a Ukázka podrobného HIPO diagramu

častěji doplněny dalšími nezbytnými podrobnostmi v <u>rozšířeném popisu</u> a dokumentací jiného typu než HIPO na př. rozhodovacími tabulkami. Sekce zpracování v diagramu obsahuje serii očíslovaných pracovních kroků popisujících funkci, která se má provést. Každou funkci je vhodné specifikovat co nejmenším počtem slov, slovesa uvádět v rozkazovacím způsobu a soustředit se na funkci a ne na konkrétní provedení. Co dělat, ne jak dělat. Ukázky HIPO diagramů jsou na obr. 4.

Dokumentace HIPO se obvykle zpracovává postupně ve dvou až třech sbornících. <u>Sborník prvotního návrhu</u>, který připravuje navrhovatelská skupina podle požadavku zákazníků, popisuje pouze základní funkce a používá se jako pomůcka při návrhu. <u>Sborník podrobného návrhu</u> vychází ze sborníku prvotního návrhu, do kterého doplňují řešitelé další údaje a nižší úrovně HIPO diagramů spolu s jinou dokumentací. Během kódování se ještě doplňuje rozšířený popis o návěští použitá v programu a jiné informace týkající se implementace. Po zakódování je tak k dispozici úplná, přesná a snadno pochopitelná dokumentace. Někdy je výhodné vypracovat ještě třetí <u>sborník pro údržbu</u>. Je to v podstatě sborník podrobného návrhu, ze kterého se vyřadily některé diagramy nižší úrovně, původně určené pro realizaci.

HIPO dokumentace výborně splňuje požadavky kladené širokým okruhem uživatelů. Vedoucímu umožňuje získat celkový pohled na systém. Aplikační programátor podle ní kóduje své moduly a programátor - údržbář rychle určí funkce, ve kterých se má provést změna. Všechny tyto požadavky plní HIPO pomocí grafického vyjádření funkcí a jejich organizačního začlenění s postupně vzrůstající podrobností.

<u>3.3. Rozhodovací tabulky.</u>

Smyslem rozhodovacích tabulek je poskytnout jasnou a stručnou tabulární formu pro zachycení logiky rozhodování případně větvení modulu. Rozhodovací tabulka obsahuje soubor podmínek a soubor na nich závislých činností. Na první pohled

lze z ní odvodit, které činnosti se budou při splnění určité kombinace podmínek provádět. V jednom modulu může být i řada vzájemně (hierarchicky) propojených tabulek.

Deputát	1	2	3	4	ELSE
ŽADATEL = ?	PRAC	PRAC	DUCHOD	DUCHOD	
ŽENATÝ	Y	N	-	-	
ODPRACOVÁNO > ?	-	-	15	5	
PŘÍDĚL UHLÍ = ?	5,4	2,7	2,7	1,2	0
PŘÍDĚL DŘEVA = ?	2	1	0	0	0

Obr. 5 Ukázka rozhodovací tabulky

Na obr. 5 je uveden příklad zjednodušené rozhodovací tabulky pro stanovení přídělu deputátu pro pracující a důchodce v hornictví. Z předložené tabulky si čtenář na př. velmi snadno zjistí, že důchodce, který odpracoval v hornictví více než 15 let obdrží bez ohledu na rodinný stav 2,7 q uhlí.

To, že rozhodovací tabulka popisuje funkci a ne její konkrétní provedení v modulu, usnadňuje přímé začlenění do HIPO diagramů. Formalizovaný zápis nabízí navíc možnost vytvořit automatický překladač rozhodovací tabulky do zdrojového kódu. Změny a opravy se pak provádějí na úrovni zdrojové rozhodovací tabulky a o správnou aktualizaci kódu pro počítač se postará překladač. Překladač může mít vedle optimalizace kódu další přídavné funkce jako zjištění logické úplnosti a indikaci nadbytečných nebo sporných kombinací podmínek.

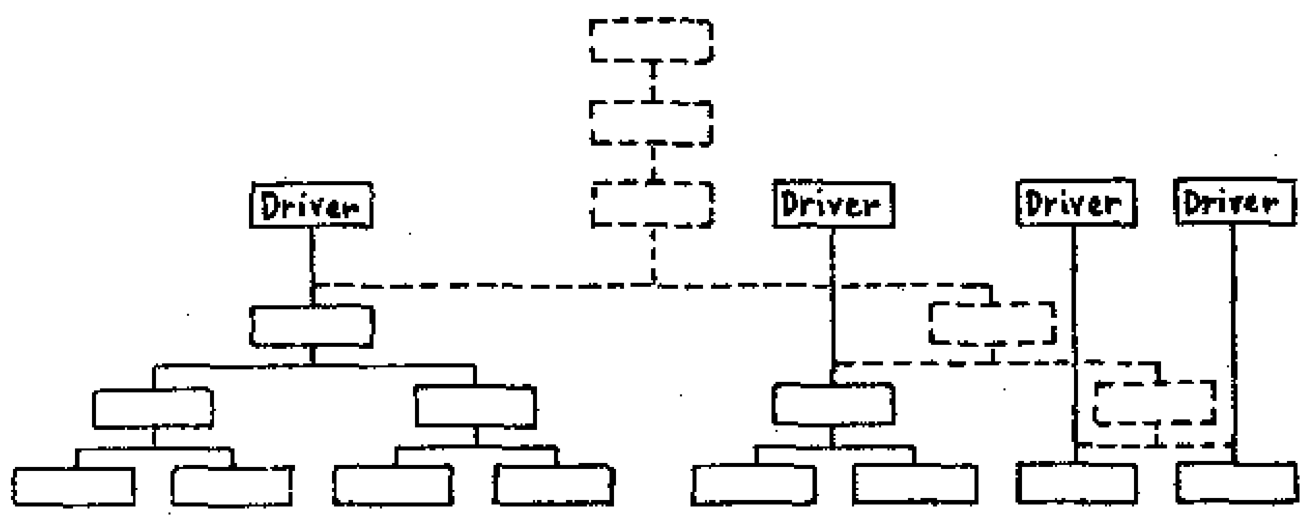
Problematika rozhodovacích tabulek je v naší literatuře dobře zpracována, jsou jí věnovány zvláštní semináře a značnou pozornost jí věnoval i havířovský seminář v loňském roce.

3.4. Postup vývoje shora dolů.

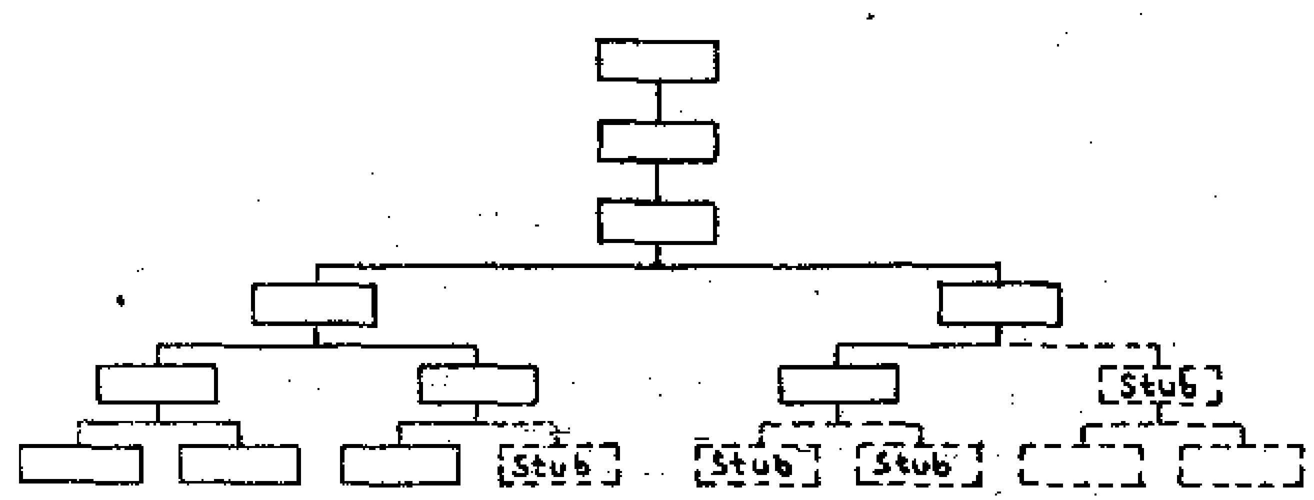
Při tvorbě a implementaci modulárního programového vybavení se donedávna téměř výlučně používal postup kódování a ladění zdola nahoru. Postup spočíval v tom, že program byl budován z postupně odladěných modulů nebo skupin modulů od nejnižší hierarchické úrovně (obr. 6). Tento postup vyžadoval vytvoření pomocných programů t.zv. driverů umožňujících odladění jednotlivých modulů nižší hierarchické úrovně a jejich částečnou integraci. Po odladění všech částí programu integrovat celý program, t.zn. spojit jednotlivé větve programu a odzkoušet jeho komplexní funkci. Bylo nutno programovat drivery, t.zn. produkovat programy, které pro vlastní funkci vyvíjeného software byly zbytečné a tudíž se po odladění příslušné části programu zahazovaly. Navíc integrace programu tímto způsobem byla často dosti komplikovaná, neboť změny v návrhu, ke kterým došlo během vývoje programu především v definici dat, vyvolávaly zásahy do již odladěných modulů nižší hierarchické úrovně případně driverů. Tyto zásahy zpravidla měly nepříznivý vliv na kvalitu programu, zvláště na jeho jednoduchost a průhlednost.

Postup vývoje shora dolů (obr. 7) naproti tomu začíná kódováním a testováním modulu nejvyšší hierarchické úrovně. T.zn., že místo odkud začít je zcela jednoznačně určeno - je pouze jediné, na rozdíl od metody zdola nahoru. Jediný výchozí bod neznamená, že implementace musí nutně probíhat po všech větvích paralelně. Na př. externí interface, t.j. ty větve programu, které řeší vstup nebo výstup dat, mohou být vyvinuty nejdříve, aby byla usnadněna práce na dalším vývoji programu.

Modul nejvyšší hierarchické úrovně pracuje s moduly nižší úrovně (zpravidla je volá), které však v tomto stadiu vývoje nejsou dosud realizovány. Je tedy nutno mít k dispozici v knihovně v tomto okamžiku moduly náhradní t.zv. program stubs. Náhradní modul (stub) neprovádí žádný "rozumný" výpočet, pouze existuje jako externě známé jméno.



Obr. 6 Tradiční postup vývoje zdola nahoru.



Obr. 7 Postup vývoje shora dolů.

Nejjednodušší verze náhradního modulu obsahuje pouze definiční příkaz, komentář s popisem funkce modulu a s poznámkou, že jde prozatím o stub a příkaz návratu. Je však možné další rozšíření. Náhradní modul může na př. produkovat ladící zprávu nebo simulovat paměťové či časové nároky budoucího skutečného modulu, pokud jsou tyto funkce pro vývoj daného systému významné. Náhradní moduly se předem kompilují a ukládají do příslušné knihovny. Při postupu vývoje shora dolů na nižší hierarchickou úroveň, t.zn. až na ně přijde řada, jsou pak tyto moduly doplněny příkazy realizujícími skutečnou funkci modulu; prozatímní již neplatné komentáře a případné příkazy pro ladě-

ní je nutno odstranit. Celý postup se opakuje tak dlouho, pokud nejsou všechny náhradní moduly substituovány moduly skutečnými, t.zn., že celý program je integrován a odzkoušen.

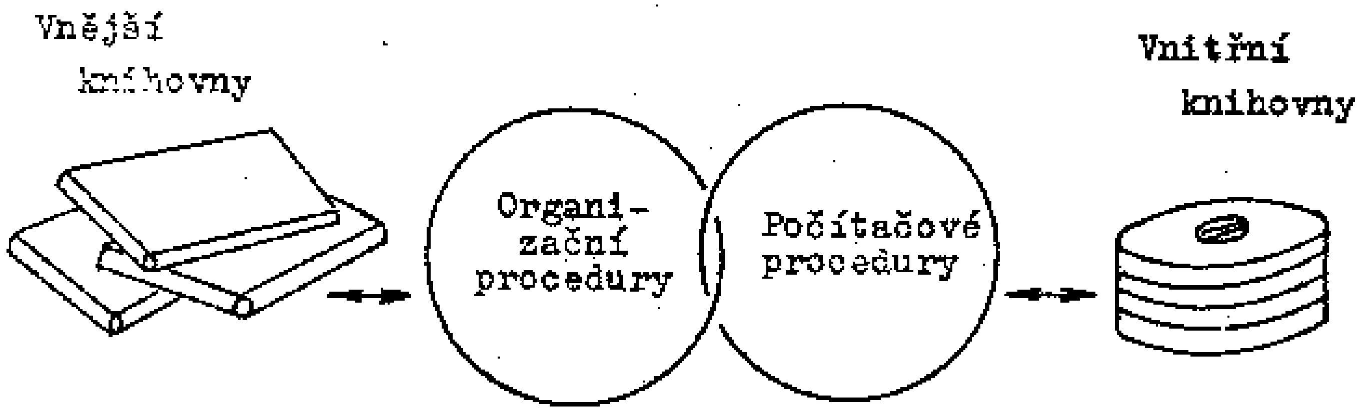
Výhody postupu shora dolů jsou zřejmé. Není nutno produkovat programy s dočasnou funkcí (drivery), které se po použití zahazují. Rozsah "zbytečného" kódování v náhradních modulech (stubs) je zanedbatelný vzhledem k rozsahu kódování driverů. Celý program nebo systém je průběžně integrován během vývoje jaksi mimochodem bez zvláštního úsilí. Funkce celého programu je od počátku průběžně ověřována. Program od prvního ladění může produkovat výstupy ve formě požadované zadáním. Tyto výstupy se během vývoje přibližují zadanému požadavku i co do obsahu. Ve srovnání s postupem zdola nahoru odpadá ve vývoji programu testování jednotlivých modulů, takže celý postup shora dolů je i časově kratší.

Uvedený postup a úvahy vycházely z literatury, především z [1]. Postup je zřejmě bez výhrad použitelný pro úlohy, které je možno označit za rutinní. Často však se nacházíme v situaci, kdy některou funkci programu nebo modulu máme v úmyslu realizovat postupem nebo programovací technikou, kterou nemáme dosud ověřenou (na př. využití některých dosud nepoužívaných možností vyššího programovacího jazyka, nově vytvořený algoritmus pro některou složitější funkci modulu a p.). V tomto případě je vhodné vytvořit driver a umožnit si tak důkladné odzkoušení funkce testovaného modulu, modelovat méně běžné stavy atd. Tento postup nelze považovat za nějaký prohřešek proti pravidlům vývoje shora dolů, neboť při vývoji shora dolů vždy funkce modulů na nižší hierarchické úrovni považujeme za realizovatelné. Pokud tedy máme pochybnosti o technické schůdnosti realizace, můžeme, ale dokonce ji musíme ověřit.

4. Podpůrná knihovna vývoje.

Základním principem moderních programovacích metod je skutečnost, že všechna data, kódy, dokumentace i počítačové výstupy (dobré i chybové) přestávají být soukromým vlastnic-

tvím jednotlivých programátorů a stávají se obecně přístupnými částmi podpůrné knihovny vývoje. Čtyři základní složky podpůrné knihovny vývoje jsou na obr. 8



Obr. 8. Základní složky podpůrné knihovny vývoje

Vnitřní knihovny obsahují na mediích počítače (nejčastěji discích) veškerá data potřebná pro projekt včetně zdrojového kódu, rozpisů dat, object a load modulů, výroků pro spojování a ladění. Stav vnitřních knihoven se odráží ve vnějších knihovnách obsahujících nejčerstvější výpisy všech částí vnitřní knihovny, ale i archivní svazky nahražených výpisů. Počítačové procedury se vztahují k vnitřním knihovnám a zajišťují na př. údržbu knihoven, kompilování zdrojových modulů, spojování a ladění, zabezpečení a obnovu knihoven, výpisy stavů atd. Organizační procedury jsou vlastně předpisy pro styk uživatelů s knihovnou. Formalizují zápisy požadavků na kompilace a ladění, předpisy pro zakládání opravených stránek do vnějších knihoven a předpisy pro archivaci.

Při použití týmu vedoucího programátora v klasickém pojetí pracují programátoři výlučně s vnějšími knihovnami a organizačními procedurami. Zásahy na vnitřních knihovnách provádí pomocí počítačových procedur pouze jediný vyčleněný pracovník - knihovník, který zároveň odpovídá za správný stav podpůrné knihovny vývoje jako celku. Používá-li se interaktivního způsobu ladění z terminálů, závisí práce s podpůrnou knihovnou vývoje na typu terminálů, jejich počtu a vlast-

nostech použitého software. U nejprostšího způsobu má k terminálu přístup pouze knihovník, který jeho prostřednictvím plní tytéž úkoly jaké plnil při dávkovém zpracování. Při vyspělejším postupu zapisuje knihovník veškerý nový kód z programovacích formulářů přes terminál a řadoví programátoři používají jiné terminály k zápisu změn a požadavků na chody. Nejvýhodnější způsob je dát každému programátórovi vlastní terminál obrazovkového typu, ze kterého může zapisovat veškerý nový kód, požadavky na změny, kompilace i ladění. Podpůrná knihovna vývoje používaná ve spojení s obrazovkovými displeji a vyspělým software dává všem programátorům i vedení okamžitý přístup do celé vnitřní knihovny. Bez ohledu na použitý přístup má funkce knihovníka, jako správce podpůrné knihovny vývoje, stále své opodstatnění. Jako minimum zabezpečuje vnitřní knihovny, sbírá počítačové výstupy, doplňuje a udržuje pořadače vnějších knihoven, zaznamenává a shromažďuje statistické údaje pro vedení. Knihovník bývá pověřován i generováním náhradních modulů (stubs).

Použití podpůrné knihovny významně zlepšuje kontrolu stavu práce na systému. Vzhledem k tomu, že vyvíjený systém se kompletuje spojitě, odráží obsah knihoven dosažený stupeň vývoje. Úplnost lze objektivně měřit tím, jaká část systému na knihovně je již v provozu.

5. Vlastní programování.

Při vývoji programového vybavení nejméně jednou nastane stadium, ve kterém je nutno od uvažování funkčního, používaného při navrhování struktury programu a při komunikaci se zadavatelem, přejít k uvažování procedurálnímu umožňujícímu realizovat funkci příslušnou posloupností příkazů.

Prozatím užívané zdrojové jazyky (i ty nejvyšší) většinou nejsou pro počáteční stadium vývoje programu vhodné, především pro svou přílišnou podrobnost a vysoké formální nároky. V této fázi vývoje je dosud rozšířeno užívání vývojových diagramů přes jejich zřejmé a na mnoha místech uváděné nedostatky.

5.1. Pseudokód.

Místo vývojových diagramů lze při navrhování a tvorbě programových částí užít t.zv. pseudokód. Je to jazyk umožňující jednoznačný zápis programu formou blízkou běžnému jazyku. Pseudokód nebyl navržen jako kompilovatelný jazyk, má velice málo formálních pravidel a poskytuje svému uživateli dostatečnou volnost vyjadřování myšlenek na vhodné úrovni podrobnosti.

Zavedení pseudokódu bylo inspirováno strukturovaným programováním, především snahou o použití vymezených jednoduchých řídících struktur při tvorbě programů. Jak bylo dokázáno [6], některá rozšíření a diskuse na př. v [7], každý správný program t.j. program s jedním vstupním a jedním výstupním bodem, neobsahující nekonečnou smyčku nebo nedosažitelný kód, lze zapsat s použitím tří základních řídících struktur, a to sekvence příkazů, výběru a iterace (opakování zakódovaného postupu dokud podmínka platí). Zápis těchto tří struktur je základem pseudokódu. Na programy složené z uvedených základních struktur je možné aplikovat formální metody důkazu korektnosti programu i když zřejmě ještě delší dobu nebudou běžně používány.

Z praktických důvodů (jednoduchost nebo zkrácení a přehlednost zápisu) je vhodné užívat při návrhu programů i dalších řídících struktur respektujících principy strukturovaného programování. Jsou to příkazy iterace typu UNTIL (t.j. opakování probíhá pokud není splněna podmínka s testováním podmínky až na konci sekvence příkazů, případně příkaz opakování s podmínkou implicitně obsaženou v indexujícím parametru)a příkaz pro vícenásobné větvení - přepinače (CASE).

Další řídící struktura - příkaz skoku - je sice prohřeškem proti ortodoxnímu pojetí strukturovaného programování, avšak v některých zvláštních případech je využití příkazu skoku v pseudokódu účelné nejen z hlediska efektivnosti výsledného kódu, ale také z hlediska jednoduchosti a průhlednosti zápisu programu jak v pseudokódu, tak zejména pak po přepisu do zdrojového programovacího jazyku.

Název	Vývojový diagram	Pseudokód	Poznámka
Sekvence	a, b	a b	
Výběr	ano, a, p, ne, b	IF p THEN a ELSE b ENDIF	p je podmínka Větev ELSE může být prázdná, v tom případě ji lze vynechat
Iterace – WHILE	a, ano, p, ne	DOWHILE p a ENDDO	p je podmínka Testuje se předem a při splnění se provede činnost a
Iterace – UNTIL	ne, a, p, ano	DOUNTIL p a ENDDO	Podmínka se testuje po činnosti a při nesplnění se činnost opakuje
		DO i=c_1 TO c_2 BY c_3 a ENDDO	Podmínka impl. obsažena v ind. parametru c_1 počátek c_2 konec c_3 krok
Přepínač	f_1, f_2, f_3, f_{n+1}	CASE e_1 OF e_1: f_1 e_2: f_2 e_3,e_4: f_3 e_n: f_n ELSE: f_{n+1} ENDCASE	Lze užít i jiné srozumitelné formy zápisu

Obr. 9 Řídící struktury a jejich vyjádření v pseudokódu

Přehled řídících struktur, jejich vývojové diagramy a zápis v pseudokódu je uveden na obr. 9. Povšimněme si, že na každou z těchto řídících struktur lze pohlížet jako na realizaci funkce úseku programu s jedním vstupem a jedním výstupem, t.zn., že funkční bloky A, B mohou být na podrobnější úrovni realizovány pomocí uvedených řídících struktur.

Pseudokód má řadu užitečných vlastností. Umožňuje snadno čitelný zápis na libovolné úrovni podrobnosti. Zápis v pseudokódu lze snadno převést do kompilovatelného zdrojového jazyka a při tom udržet přehlednost programu. Pseudokód je účinný nástroj při vyhledávání logických celků v programu. Navíc pseudokód je i účinným prostředkem dokumentace programu, zvláště pokud je zápis v pseudokódu součástí komentářů zdrojových programových modulů. Tato forma dokumentace se na rozdíl od vývojových diagramů dá velice snadno udržovat a pořízení není časově náročné.

Způsob zápisu v pseudokódu uvedený na obr. 9 není samozřejmě jediný možný. Zvláště příkaz CASE nabízí několik dalších variant zápisu. Diskutabilní je rovněž použití anglických výrazů, mohly by být nahrazeny českými ekvivalenty. Domníváme se však, že způsob zápisu navržený na obr. 9 (velká písmena pro klíčová slova, odstavcování, ukončování struktur klauzulí END..) je přehledný a že zápis je srozumitelný bez předchozí znalosti formálních pravidel pseudokódu a tudíž lze tuto formu doporučit jako základ, který může být podle potřeby modifikován. Při modifikaci je však nutno mít na zřeteli, že pseudokód má poskytovat programátorovi co největší pohodlí zápisu postupu při zachování jednoznačnosti zápisu a ne jej svazovat nadbytečnými formálními pravidly.

5.2. Strukturované programování.

Definice dobře strukturovaného programu a diskuse efektů, které lze strukturováním programu dosáhnout je předmětem polemik mnoha autorit v oblasti programování. Nesporně však tyto polemiky v mnoha směrech příznivě ovlivnily rozvoj

metodiky vývoje programového vybavení. Mnoho sporů je vedeno o to, zda strukturované programování znamená úplné vyloučení příkazu skoku nebo zda je možno jisté omezené užití tohoto příkazu dovolit. Domníváme se, že tyto spory se netýkají nejnaléhavějších stránek vývoje programového vybavení a navíc jsou reálnému světu programátora dosti vzdálené.

Všechny z běžně užívaných programovacích jazyků obsahují příkaz skoku jako jednu z řídících struktur a kódování v těchto jazycích (máme na mysli především Fortran a Cobol) bez využití příkazu GOTO je téměř vyloučené. To znamená, že ve zdrojovém programu v současné době není možno příkazy skoku zcela vyloučit. Diskuse o vhodnosti GOTO se může omezit na užívání této řídící struktury na úrovni pseudokódu. Doporučujeme, aby příkaz skoku byl mezi dovolenými řídícími strukturami pseudokódu. Jeho využití může v některých případech čitelnost programu dokonce zlepšit. Existují totiž celé skupiny algoritmů, jejichž zápis bez příkazu skoku není jednoduchý. Pokud mají být bez příkazu skoku zapsány, je nutno buď některou část kódu zapsat dvakrát nebo použít t.zv. výhybek. Oba tyto způsoby mají svá úskalí. Při opakování kódu je nebezpečí opomenutí při opravě úseku programu vyskytujícího se dvakrát a programy využívající složitých výhybek jsou velice obtížně čitelné, často i pro samotného autora.

Za žádoucí vlastnosti programu jsou požadovány funkční správnost, možnost rychlého zakódování a odladění, srozumitelnost a přehlednost zápisu, efektivnost, snadná modifikovatelnost a přenosnost mezi programátory. O těchto vlastnostech programu se však rozhoduje především ve fázi návrhu a část je jich silně ovlivněna úrovní dokumentace. Vše to by měl mít programátor na zřeteli nejen v případech, kdy rozhoduje o využití příkazu skoku, ale v průběhu celého návrhu programu a vlastního programování.

6. Závěr.

Prostředky zdokonalených programovacích technik poskytují metodický návod pro celý proces vývoje programového

vybavení. Ve svém příspěvku jsme se úmyslně zaměřili na ty části, které mají blízký vztah k aplikačnímu programování a snažili jsme se zdůraznit, že jednotlivé složky se ve svých účincích navzájem podporují. Na druhé straně bychom však nechtěli vzbudit dojem, že o IPT platí heslo "všechno nebo nic". Jednotlivé prostředky se mohou stávat pracovním nástrojem postupně a to nejen v rámci výpočetního střediska nebo skupiny, ale i jednotlivce.

Literatura.

1. Učební texty IPT technik fy IBM
2. STEVENS W.P.,MYERS G.J.,CONSTANTIN L.L.: Structured design, IBM System Journal 13(2),115-139(1974)
3. WIRTH N.: Program development by stepwise refinement, Comm. ACM 14(4),221-227(1971)
4. WIRTH N.: On the composition of well-structured programs, ACM Computing Surveys 6(4),247-259(1974)
5. KNUTH D.E.: Structured programming with go to statements, ACM Computing Surveys 6(4),261-301(1974)
6. BÖHM C.,JACOPINI G.: Flow diagrams, Turing machines and languages with only two formation rules, Comm. ACM 9(5), 366-371(1966)
7. LEDGARD H.F.,MARCOTTY M.: A genealogy of control structures, Comm. ACM 18(11),629-639(1975)
8. An introduction to structured programming in PL/I, manuál IBM GC20-1777-1,2.vydání(1977)
9. STAY J.F.: HIPO and integrated program design, IBM System Journal 15(2),143-154(1976)
10. Van LEER P.: Top-down development using a program design language, IBM System Journal 15(2),155-170(1976)
11. BRZICKÝ J.: Realizace schemat strukturovaného programování v pseudokódu a jazyce PL/I, Sborník Metody programování počítačů III. generace Havířov 1977,str.44-56
12. POLACH O.: Dokumentační prostředek HIPO, Sborník Metody programování počítačů III. generace Havířov 1976, str.269-282.