Ing. Vlastimil Cevela

Výpočetní středisko INGSTAV Brno

Systémový přístup a logická stavba programu

1. Úvod

Předkládaný referát se zabývá problematikou metodiky programování s využitím aplikace systémového přístupu pro tvorbu architektury programu. Po vymezení základních pojmů je uveden příklad řešení algoritmu slučování. Dále jsou popsány zásady uspořádané logiky a organizace programování používané na počítači TESLA 200 ve výpočetním středisku INGSTAV BRNO. Závěrem je uvedeno několik názorů na filozofii programátorského řemesla.

2. Systémový přístup

Přesto, že se jedná o věci obecně známé, nebude na závadu, když si úvodem krátce připomeneme některé pojmy.

Systém budeme definovat jako soubor prvků a jejich jistých vazeb mezi sebou i vzhledem k podstatnému okolí, který tvoří jednotně uspořádaný celek, sestavený za určitým cílem.

Prvek považujeme již za nedělitelný celek (černou skříňku), tj. zajímají nás vstupní parametry a jim odpovídající výstupy. Nezabýváme se vnitřním uspořádáním.

Rozlišovací úroveň potom určuje, co považujeme na urči-

tém stupni zkoumání za podstatné prvky a s jakými jejich vazbami musíme uvažovat k dosažení žádoucího cíle.

Za systémový považujeme tedy takový přístup k řešení problému, kdy se snažíme v každé fázi najít vše důležité, co ovlivňuje výsledek, aniž bychom se zabývali zbytečnými podrobnostmi. Při zpracování detailů na další, podrobnější rozlišovací úrovni postupujeme pak se stejnou filozofií.

Jinak, lapidárně řečeno - musíme vždy počítat s tím, že vše souvisí se vším, ale je nutno správně vystihnout, co je pro logiku věci na daném stupni přiblížení podstatné.

To, co zde bylo uvedeno se na první pohled zdá zcela jasné a samozřejmé, vždyť to nakonec odpovídá staré moudrosti, že je třeba "vyhmátnout jádro problému". Jakmile se však pokusíme o konkretní aplikaci při řešení určitého případu zjistíme, že věc nebývá ani zdaleka tak jednoduchá, jak na první pohled vypadá. Potíže vzniknou především v tom, že je třeba v souvislosti s různými rozlišovacími úrovněmi používat též různé stupně abstrakce.

Pro naše účely abstrahovat znamená vytvářet pomocí analýzy určitých jevů či konkretních pojmů nové pojmy obecné, tj. nepracujeme pak přímo s jednotlivými detaily s bohatou složitostí, ale s jistými jejich representanty, vykazujícími pouze ty vlastnosti a vztahy, které na dané rozlišovací úrovni potřebujeme jako podstatné. Použití různých stupňů abstrakce pak vyžaduje řešit úkol, co kdy považujeme za prvek, a které z vazeb je nutno na příslušné úrovni studovat.

3. Programování

Především je nutné zde připomenout, že ve všech dalších úvahách se omezujeme převážně na problematiku zpracování hromadných dat, i když některé závěry mohou platit též obecněji.

Pokusme se nyní formulovat, co v sobě zahrnuje činnost, běžně nazývaná programování. Dojdeme k následujícím třem základním skupinám prací, které mají jednak svoji specifickou

metodiku a jsou většinou rozlišeny i časovým odstupem:

Programování v užším slova smyslu řeší převedení myšlenkového algoritmu zpracování daného problému do programovacího jazyka příslušného počítače. Je to tedy provedení potřebné programové analýzy a vlastní zápis programu v kódu zdrojového jazyka.

Ladění si vymezíme jako provedení kompilace ze zdrojového jazyka do kódu stroje, ověření správné funkce navrženého programu a odstranění veškerých zjištěných formálních i logických chyb.

Údržba pak představuje dodatečné přizpůsobování novým požadavkům, tj. promítání potřebných úprav a změn do odladěného programu.

Vzhledem k tomu, že se stále zvětšuje rozsah programového vybavení a běžný život přináší stále nové požadavky a změny, jsou to právě nároky na údržbu, které kriticky ovlivňují produktivitu práce programátorů. Jedním z hlavních kriterií pro hodnocení programů se tak stává jejich portabilita a udržovatelnost. Podrobnější rozbor těchto souvislostí je možno nalézt v literatuře např. /10/, zde zůstaneme u konstatování, že se usilovně hledají cesty a metody pro zvýšení efektivnosti při navrhování programů.

Je možno vysledovat dvě tendence zaměřené k tomuto cíli. První z nich, která je staršího data se vyznačuje snahou formalizovat postupy zpracování určitých tříd úloh a dosáhnout někdy i značně širokou variabilitu možností za současného snížení pracnosti při tvorbě zdrojového programu. a zvýšení jeho přehlednosti a verifikovatelnosti.

Do této skupiny lze zařadit různé parametricky ovládané programové generátory a speciální jazyky jako je např. [illegible], RPG, a dále celý systém normovaného programování vyvinutý firmou UNIVAC. Tento systém zobecňuje zásady řešení rozsáhlé podtřídy úloh, avšak současně určuje jisté striktní normy, pokud jde o řízení programu a členění jeho funkcí do přede-

psaných bloků.

Druhý směr, nabývající na široké popularitě v posledních letech můžeme charakterizovat jako pokusy o aplikaci systémového přístupu na tvorbu programu. Když budeme podrobněji studovat praktiky modulárního programování, strukturovaného programování nebo budeme-li číst o postupném zjemňování, programování shora dolů, či o stratifikaci /17/,/7/ zjistíme, že zde existuje jeden společný základní záměr: definovat na určité rozlišovací úrovni všechny podstatné prvky a jejich vzájemné vazby tak, aby bylo dosaženo žádaného cíle a současně získána možnost pro dělbu práce a nezávislé řešení jednotlivých detailů.

Používá se modularizace, tj. vlastně rozkladu na jisté podprogramy nebo elementární procesy, určují se pravidla pro strukturu, která představuje soustavu vnitřních vztahů pro účelné uspořádání jednotlivých prvků, problém se postupně zjemňuje po jednotlivých rozlišovacích úrovních atd.

Využití metod na bázi systémového přístupu má některé velice zajímavé přednosti. Především jsou univerzální, tj. nejsou omezeny na žádnou předem specifikovanou oblast úloh. Další výhodou je, že k jejich využívání není nutné znát nová množství informací a pravidel o zápisech parametrů, nové instrukce a programovací jazyky, ale stačí pochopení logiky vztahu systému a prvku na jisté rozlišovací úrovni, a zácvik v používání příslušných programátorských praktik.

Tento materiál si klade za cíl, nejprve na konkretním příkladě předvést ukázku, a potom s využitím druhé skupiny výše popsaných způsobů řešení formulovat hlavní technologická pravidla pro tvorbu účelně uspořádaných programů.

Problém racionalizace programovacího procesu je však třeba řešit i pomocí nových organizačních přístupů, proto bude i těmto otázkám věnována patřičná pozornost.

4. Příklad programového řešení problému SLUČOVÁNÍ

Algoritmy, které budou dále popisovány jsou psány v jazyku COBOL TESLA 200. Z důvodu zachování přehlednosti jsou v popisech uvedeny jen ty úseky programů, které jsou pro pochopení logiky jeho stavby na sledované realizovací úrovni nesbytně nutné. Přítomnost všech povinných oddílů a sekcí, vyplývajících z definice jazyka Cobol se implicitně předpokládá, i když jsou uvedeny jen některé. Rovněž nejsou uváděny úrovně a zobrazení (picture) pro jednotlivá pásma deklarací (popisovače, přepínače, FD popisy a věty vstupních souborů a pod.), poněvadž jsou z hlediska popisu logické konstrukce procedur nepodstatné.

Hloubka řešení algoritmů se omezuje na realizovací úroveň věty (recordu), jakožto logického prvku v systému procedury. Pro identifikaci jednotlivých vět je zaveden pojem POPISOVAČ (deskriptor, klíč), který reprezentuje větu při vyhodnocování logických vztahů, určujících způsob zpracování.

Problematika slučování patří mezi nejběžnější úlohy, programované v oblasti zpracování hromadných dat. Tento algoritmus je nutno řešit při veškerých změnových řízeních kartoték nebo matričních souborů, slučování údajů z více souborů, zatřiďování, připočítávání obratů k zůstatkům a pod. Výsledkem může být nějaký výpočet, tisk sestavy nebo výstup nového souboru atd.

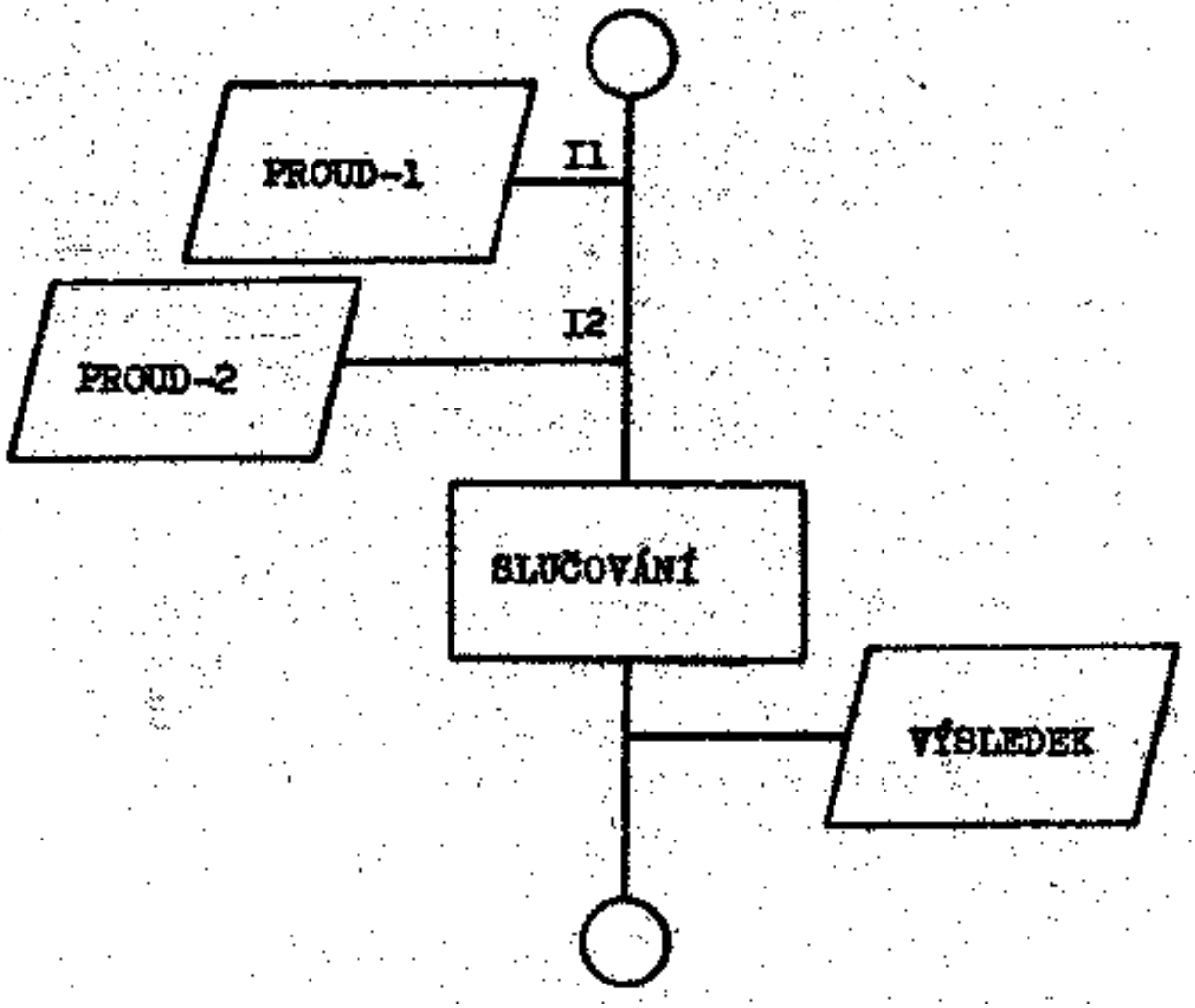
Uvažujeme nejprve případ sloučení 2 proudů dat a zpracování výsledků dle následujícího příkladu A:

Charakteristika A:

- vstupem jsou 2 shodně setříděné, sekvenčně uspořádané proudy dat I1 a I2 (soubory na magnetických páskách, na děrných štítcích a pod.)
- ve strukturách vět (záznamů) obou souborů je definován popisovač a na základě srovnání těchto popisovačů je provedeno zpracování

- výsledkem jsou 3 různé druhy zpracování s případným výstupem pro všechny možné případy: I1-NEMA-I2, I1-MA-I2, I2-NEMA-I1.
- předpokládejme, že logickým prvkem, který vstupuje do zpracování je každá věta ze vstupního souboru, tj. v žádném souboru není více než 1 věta se shodným popisovačem.

Příklad A: (obr.1)

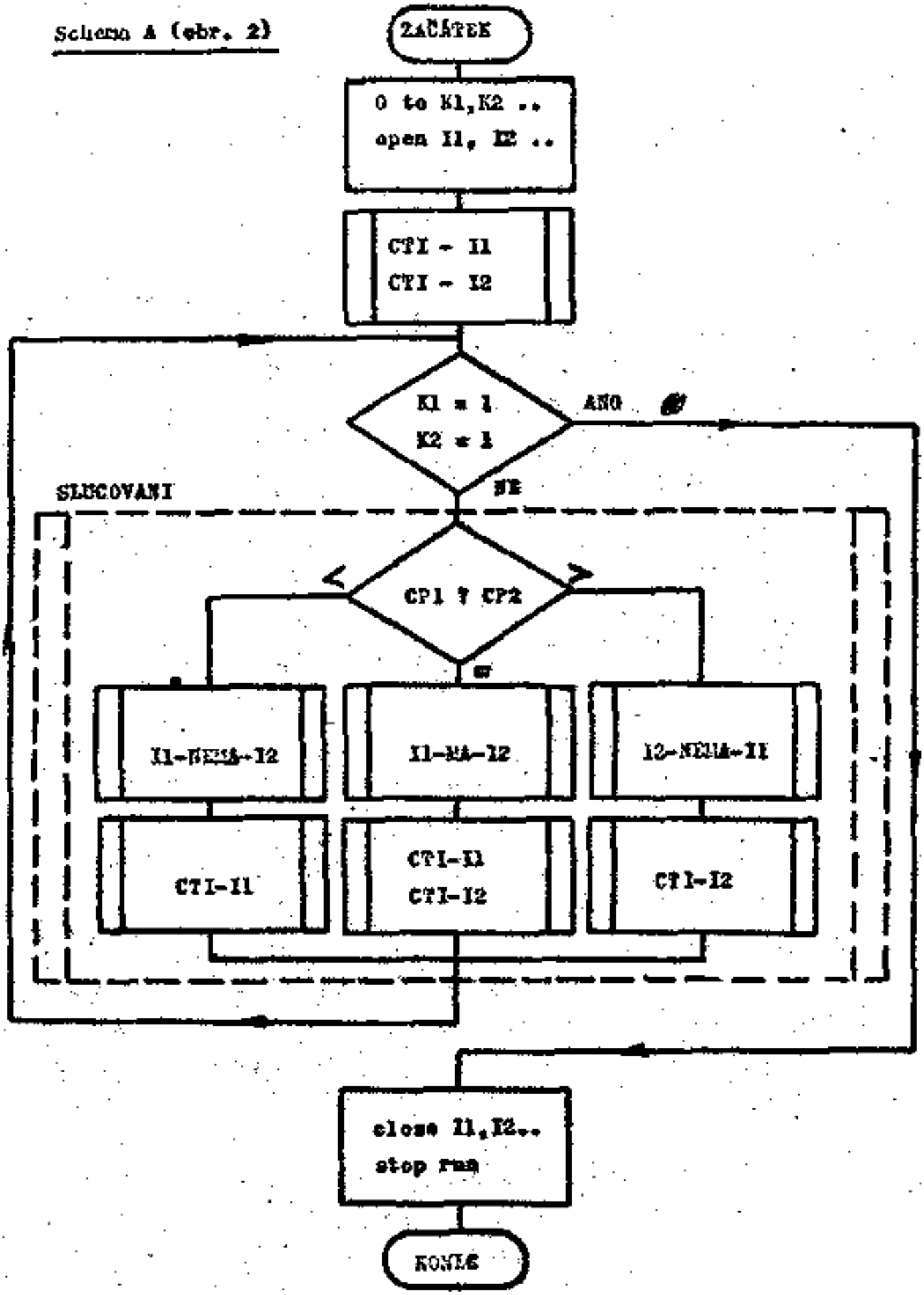


Program A

```
Data Division.
     I1, I2              (vstupní soubory)
     CP1, CP2            (čtené popisovače souboru I1 a I2)
     K1, K2              (přepínače: je skončeno zpracování
                          souboru I1, resp. I2 ? = 0 ne,
                          = 1 ano)

Procedure Division. (viz obr. 2)
     Move 0 to K1,K2 (+ příp. nulování počítadel).
     Open input I1,I2,....
     Perform CTI-I1, perform CTI-I2.
     Perform SLUCOVANI until K1 = 1 and K2 = 1.
     Close I1, I2, ...
     Stop run.
     -
SLUCOVANI.
     If CP1 < CP2 perform I1-NEMA-I2 perform CTI-I1
     else
     if CP1 = CP2 perform I1-MA-I2 perform CTI-I1
                  perform CTI-I2
     else
     if CP1 > CP2 perform I2-NEMA-I1 perform CTI-I2.
     —
CTI-I1, resp. CTI-I2.
     (čtení jedné věty příslušného souboru a naplnění
     odpovídajícího popisovače; v případě konce souboru
     se namísto čtení vloží 1 do K1 resp. K2 a max. hod-
     nota "FF" do čteného popisovače CP1 resp. CP2)
     —
I1-NEMA-I2, resp. I1-MA-I2, resp. I2-NEMA-I1.
     (jednotlivá zpracování dle potřebných algoritmů a
     případnými výstupy).
```

Schema A (obr. 2)



Vidíme, že řešení je skutečně velice přehledné, jsou patrny rozlišovací úrovně hlavní procedury s cyklem provádějícím vlastní slučování, logika rozdělení řízení programu mezi všechny 3 různé možnosti zpracování při nalezení nebo nenalezení vět se shodnými popisovači a jsou vymezeny podprogramy, které se pak zabývají detaily čtení, resp. zpracování.

Je třeba připomenout, že uvedený algoritmus ošetřuje zpracování rovnocenných souborů; pokud např. soubor I1 představuje kartotéku a I2 změny, kterých může být pro jeden popisovač i více nebude při CP1 = CP2 prováděno CTI-I1. Tyto další aplikace však již ponecháme na svědavém čtenáři. Zaměřme nyní pozornost na další obecnější příklad B, kdy v obou vstupních souborech bude více vět se shodným popisovačem:

Příklad B - podprogramy CTI-I1, resp. CTI-I2,

- schema dle obr. 1 a charakteristika kromě posledního bodu zůstává stejná jako v příkladě A.
- logickým prvkem, který vstupuje do vlastního zpracování však není každá věta vstupního souboru, ale skupina vět se shodným popisovačem.
- čtení jednotlivých vět v rámci skupiny probíhá v cyklu tak dlouho dokud nedojde ke změně čteného popisovače; v průběhu tohoto čtení skupiny se v paměti postupně vytváří pracovní výsledek (např. načítáváním určitých hodnot).
- logika zpracování po skupinách pak zásadně vychází ze dvou fází: zahájení a zakončení.

  zahájení čtení skupiny představuje třeba nulování počitadel pro načítání, naplnění pracovního popisovače skupiny v paměti a pod.
  zakončení čtení skupiny potom obsahuje předání uloženého pracovního popisovače skupiny a pracovního výsledku k dispozici programu SLUČOVÁNÍ na vyšší rozlišovací úroveň, tj. do čteného popisovače souboru (CP1, resp. CP2) a do konečného výsledku; dále

zde musí být provedena signalizace, že skupina je přečtena.

Jak je z výše uvedeného zřejmé, prakticky celé schema programu, tak jak je uvedeno v příkladě A a na obr. 2 zůstává beze zbytku v platnosti. Pouze na nižší rozlišovací úrovni, tj. v podprogramech CTI-I1, resp. CTI-I2 není načtena pouze 1 věta, ale celá skupina - vlastně je vytvořen jistý pracovní výsledek, který se však navenek z pohledu vyšší rozlišovací úrovně chová shodně.

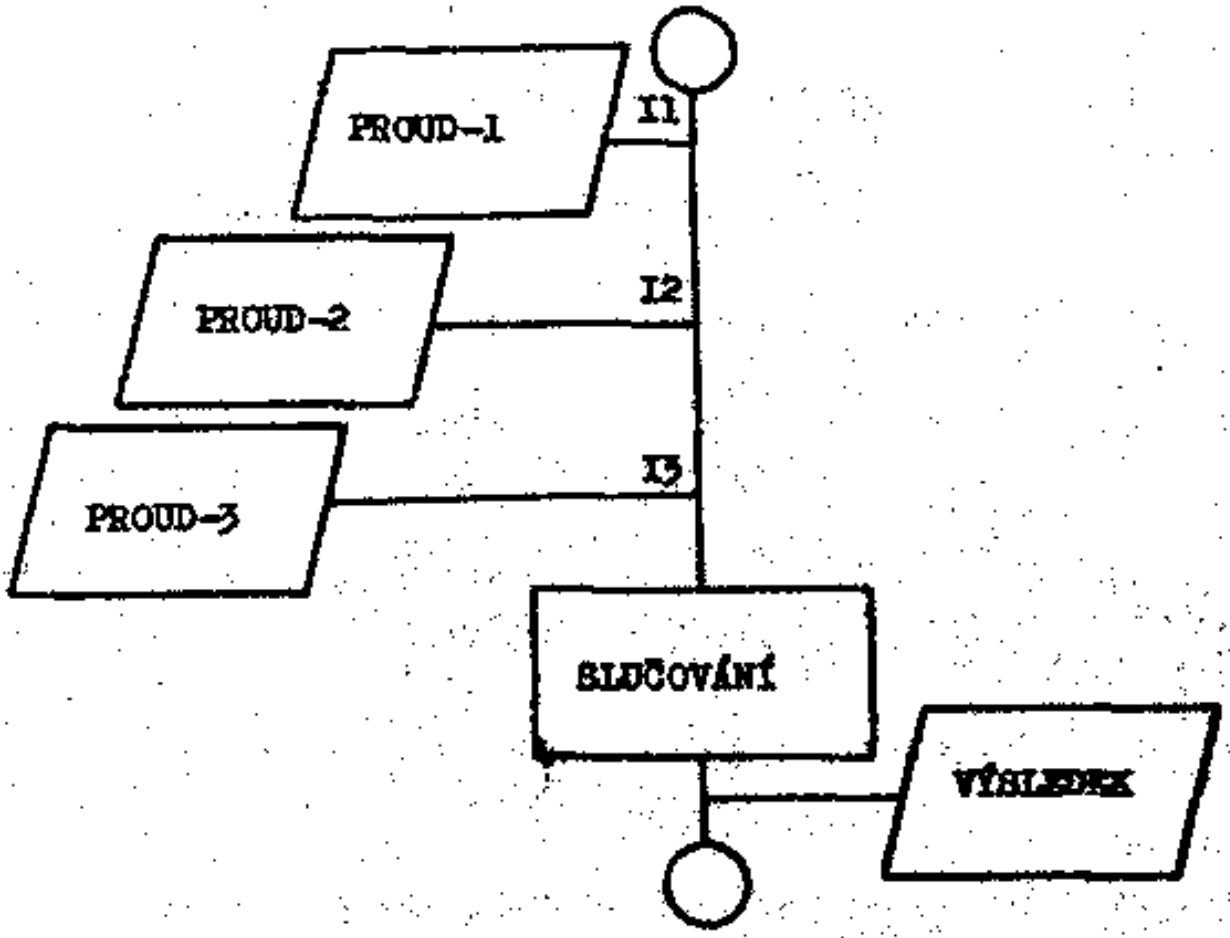
Vzhledem k omezenému rozsahu zde nelze rozepisovat celý algoritmus tak, jak je uveden v literatuře /11/ a /18/, ale snad je i z tohoto stručného popisu zřejmé, o co jde. Příklad B by měl ve spojitosti s příkladem A ukázat jednu z možností, jak lze vytáhnout logiku problému na vyšší rozlišovací úroveň nad detailní řešení vlastního zpracování.

Vraťme se nyní ještě jednou k programu A a k prostředkům, kterými je realizována logika procedury SLUČOVÁNÍ. Rozdělení řízení programu na 3 možné větve je ovládáno vztahem mezi popisovači CP1 a CP2. Je zřejmé že ten z popisovačů, který představuje minimum určuje, že se zpracovává věta (příp. skupina vět) pouze z jemu odpovídajícího souboru. Při CP1 < CP2 se tedy provede POUZE-I1 a CTI-I1, při CP2 < CP1 potom POUZE-I2 a CTI-I2, a při "minimu" CP1 = CP2 se zpracuje vlastně "POUZE"-I1-I2 a následným CTI-I1 a CTI-I2.

Nyní můžeme přistoupit k nejobecnějšímu případu, kdy na vstupu pro slučování je obecně n proudů dat (souborů). Intuitivním ošetřováním všech možných případů bychom dostali pravděpodobně značně složitý program. Budeme-li však na tento problém aplikovat postup hledání minima popsaný v předchozím odstavci, redukuje se celá záležitost určení všech možností a větvení procedury na několik řádků velice přehledně uspořádaného textu.

Pro konkrétní ilustraci je dále uveden příklad C, který uvedeným způsobem řeší sloučení 3 proudů dat. Charakteristika přísluŠně odpovídá příkladu A a B, takže není znovu uváděna.

Příklad C (obr. 3)

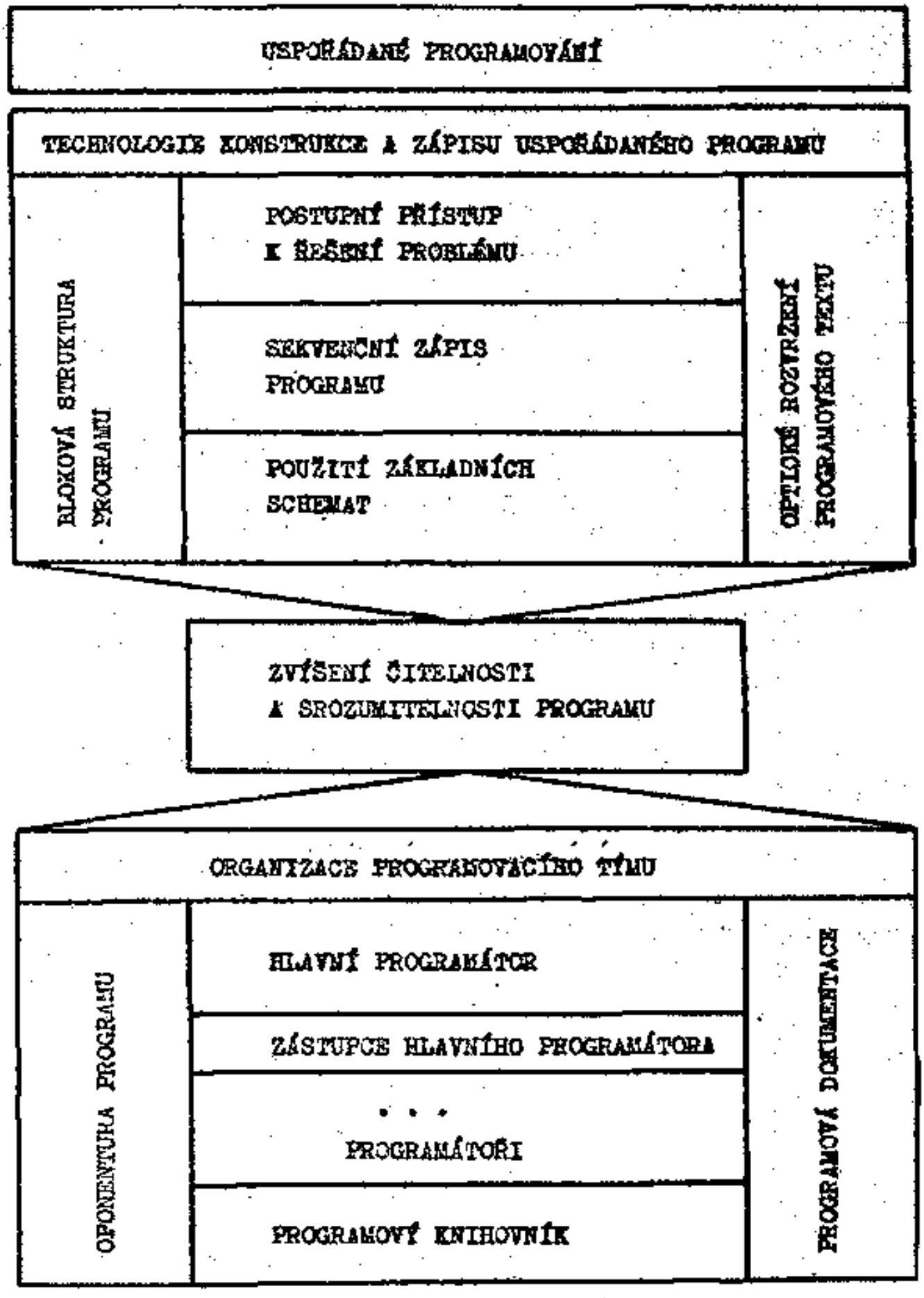


Program C

```
Data Division
     I1,I2,I3            (vstupní soubory)
     CP1,CP2,CP3         (čtené popisovače souborů I1,I2,I3)
     K1,K2,K3            (přepínače: je skončeno zpracování sou-
                          boru I1, resp. I2, resp. I3 ?
                          = 0 ne, = 1 ano)
Procedure Division.
     Move 0 to K1,K2,K3 (+ příp. nulování počítadel).
     Open input I1,I2,I3,...
     Perform CTI-I1, perform CTI-I2, perform CTI-I3.
     Perform SLUCOVANI until K1 = 1 and K2 = 1 and K3 = 1.
     Close I1,I2,I3,...
     Stop run.
```

```
SLUCOVANI.
     If CP1 < CP2 and < CP3 perform POUZE-I1 perform CTI-I1
     else
     if CP2 < CP1 and < CP3 perform POUZE-I2 perform CTI-I2
     else
     if CP3 < CP1 and < CP2 perform POUZE-I3 perform CTI-I3
     else
     if CP1 = CP2 and < CP3 perform POUZE-I1-I2 perform CTI-I1
                            perform CTI-I2
     else
     if CP1 = CP3 and < CP2 perform POUZE-I1-I3 perform CTI-I1
                            perform CTI-I3
     else
     if CP2 = CP3 and < CP1 perform POUZE-I2-I3 perform CTI-I2
                            perform CTI-I3
     else
     if CP1 = CP2 = CP3 perform SPOLU-I1-I2-I3 perform CTI-I1
                        perform CTI-I2 perform CTI-I3.
     —
CTI-I1, resp. CTI-I2, resp. CTI-I3.
     (čtení jedné věty nebo skupiny vět příslušného souboru
     analogicky s předchozími příklady A nebo B).
     —
POUZE-I1 až SPOLU-I1-I2-I3.
     (dílčí zpracování jednotlivých samostatných větví dle po-
     třebných algoritmů a případnými výstupy).
```

(obr. 4)


USPOŘÁDANÉ PROGRAMOVÁNÍ
TECHNOLOGIE KONSTRUKCE A ZÁPISU USPOŘÁDANÉHO PROGRAMU
BLOKOVÁ STRUKTURA PROGRAMU
POSTUPNÝ PŘÍSTUP K ŘEŠENÍ PROBLÉMU
SEKVENČNÍ ZÁPIS PROGRAMU
POUŽITÍ ZÁKLADNÍCH SCHEMAT
OPTICKÉ ROZTŘÍZENÍ PROGRAMOVÉHO TEXTU
ZVÝŠENÍ ČITELNOSTI A SROZUMITELNOSTI PROGRAMU
ORGANIZACE PROGRAMOVACÍHO TÝMU
OPONENTURA PROGRAMU
HLAVNÍ PROGRAMÁTOR
ZÁSTUPCE HLAVNÍHO PROGRAMÁTORA
. . .
PROGRAMÁTOŘI
PROGRAMOVÝ KNIHOVNÍK
PROGRAMOVÁ DOKUMENTACE

Předchozí příklady měly být pokusem o konkretní předvedení, jak lze přistupovat ke tvorbě architektury programu. Při jejich konstrukci bylo použito některých praktik z oblasti výše zmíněného "druhého směru", tj. strukturovaného programování, resp. přístupu shora-dolů. V dalším bude v hrubých rysech popsána metodika, která vznikla syntézou poznatků z literatury (především /1/, /2/, /3/, /4/) a programátorské praxe a je pod názvem USPOŘÁDANÉ PROGRAMOVÁNÍ a úspěchem používána ve VS INGSTAV Brno.

5. Uspořádaná logika a organizace programování

Pokus o formulaci podstaty věci vypadá asi následovně:

"P r o g r a m y j e t ř e b a k o n s t r u o v a t t a k , a b y b y l y č i t e l n é a s r o z u m i t e l n é . P o t o m j e m o ž n o p r o v á d ě t k o n t r o l u a o p o n e n t u r u j e j i c h l o g i k y a u p l a t n i t v e v ý v o j o v é m t ý m u ú č e l n o u o r g a n i z a c i a d ě l b u p r á c e . V ý s l e d k e m p a k j e z m e n š e n á c h y b o v o s t , r y c h l e j š í l a d ě n í a o v ě ř o v á n í n a p o č í t a č i a p o d s t a t n ě z j e d n o d u š e n á ú d r ž b a".

Praktické řešení tohoto problému, které by vedlo k dosažení uvedených cílů lze rozdělit do dvou sfér - jednak na technologii uspořádaného zápisu vlastního programu a pak na současně aplikovaný nový organizační přístup (obr.4).

5.1. Technologie konstrukce a zápisu uspořádaného programu

Logika uspořádaného programu vychází z následujících zásad:

- POSTUPNÝ PŘÍSTUP K ŘEŠENÍ PROBLÉMU

  K programové analýze a vlastní tvorbě programu je nutno přistupovat po jednotlivých realizovacích úrovních.

To znamená vždy nejprve definovat logické prvky odpovídající příslušné úrovni a jejich vzájemné vazby. Teprve potom je možno řešit detaily jednotlivých prvků, avšak i zde je nutno dále zachovávat výše uvedený postup. Takovým přístupem, pokud je ovšem rozvážně používán, lze značně zjednodušit sledování logiky v programovém textu. V podstatě vlastně nejde o nic jiného než o důslednou aplikaci systémového přístupu na logickou konstrukci programu.

- SEKVENČNÍ ZÁPIS PROGRAMU

Program je třeba psát tak, aby byla pokud možno přímá korespondence mezi statistickou formou programu, tj. polohovým uspořádáním v zápisu zdrojového programu a dynamickým průběhem při jeho provádění. Předpokládejme např. následující programový úsek v jazyce Cobol.

```
1. if A>30 go to 3.
   if A>20 go to 2.
   move 8 to B.
   go to 4.
2. move 7 to B.
   go to 4.
3. move 6 to B.
4. exit.
```

Pro oči, vytrénované v jazyce Cobol jistě není problémem brzy postihnout všechny souvislosti. Pokud však návěští 3 a 4 budou o několik stran dále, bude třeba značného listování k tomu, aby se zjistilo, co program zamýšlí. Oč snadněji se lze v tomto programu orientovat, je-li zapsán příslušně uspořádaným způsobem:

```
1. if A>30 move 6 to B
   else
   if A>20 move 7 to B.
   else    move 8 to B.
   exit.
```

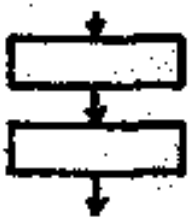
Dodržování sekvenčně uspořádané formy se striktně vyžaduje všude tam, kde je to možné.

V souladu s výše uvedeným je zde však nutné poznamenat, že požadavek sekvenčního přístupu vznášíme na instrukce jako prvky systému, kterým je PROGRAM. Pokud některý detail má své specielní řešení, prvkem pak není výkonná instrukce, ale volání příslušného uzavřeného podprogramu. Na každé další rozlišovací úrovni v systému PODPROGRAM potom platí shodné sekvenční zásady.

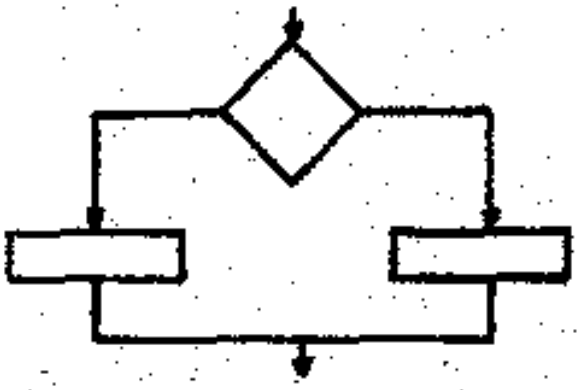
- POUŽITÍ ZÁKLADNÍCH SCHEMAT

Jednotlivé programové funkce, kterákoliv kombinace rozhodnutí a jakýkoliv druh logiky mohou být vždy vyjádřeny použitím jednoho ze tří základních schemat.
Jedná se o:

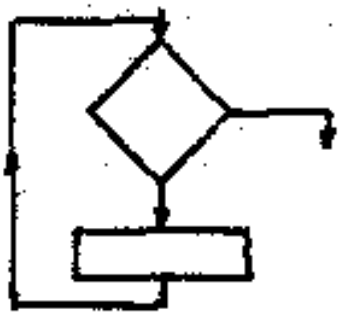
a) jednoduchou posloupnost



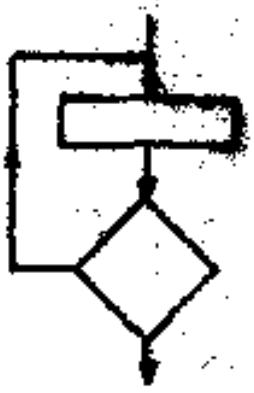
b) prosté větvení (výběr)



c) cyklus



nebo



Každé z nich je charakterizováno jedním jednoduchým vstupem a jedním výstupem.
Dále k těmto schematům přistupuje přepínač na více větví, resp. vypočtený skok, který je vlastně zevšeobecněním případu ad b) a případný abnormální výstup z cyklu navíc.
Celý program je potom vytvořen různými kombinacemi a slučováním uvedených základních schemat. Pro jednotlivé úseky však kromě uvedených výjimek musí být vždy dodrženy požadavky jednoho vstupu a jednoho výstupu. Někdy bývá jako podstata věci uváděno, že není používán příkaz GO TO. Ověřená skutečnost je však taková, že jsou-li dosud popsaná pravidla správně a důsledně aplikována, není mnoho příležitostí pro jeho užití.

- BLOKOVÁ STRUKTURA PROGRAMU

Dále je třeba zmenšit rozsah programů a složitější problémy řešit pomocí jednotlivých modulů. Délka procedury by měla být v mezích srozumitelných jednotek - takových, které lze najednou přehlédnout, tedy asi 50 řádků neboli 1 stránka výpisu z tiskárny. Protože má takový úsek jeden vstup a jeden výstup, odpadá úplně jakékoliv listování a je možno podstatně lépe koncentrovat pozornost na logiku věci.

- OPTICKÉ ROZVRŽENÍ PROGRAMOVÉHO TEXTU

K čitelnosti a srozumitelnosti programu je možno velice účinně přispět vhodnou grafickou úpravou výpisu. Jedná se o použití "mluvících" návěští a identifikátorů, vhodný vysvětlující komentář a v neposlední řadě příslušné odstavce a mezery, které názorně ukazují vymezení jednotlivých logických částí, respektive úrovní.
Dobrá přehlednost textu zdrojového programu je neocenitelná především při nutnosti zásahů po delším časovém odstupu.

5.2. Organizace programových týmů

Vytvořením obecnější srozumitelného zdrojového programu se nám otevírá nové pole působnosti při kontrole a řízení programátorské práce. Proces tvorby programů se mění se soukromého umění na veřejnou praxi a to umožňuje podstatné ovlivňování kvality a produktivity. Pro vývoj programového systému lze vytvořit pracovní tým, ve kterém panuje dokonalé rozdělení a dělba práce mezi jednotlivými specialisty. Tento způsob je podstatně výhodnější, než rozmnožování mezi všeobecně zaměřené pracovníky, kteří mají shodné mnohonásobné starosti se všemi problémy komunikace a integrace.

Jádro programového týmu sestává z hlavního programátora, jeho zástupce a programového knihovníka. Toto jádro je schopno plnit komplexní úkoly od programové analýzy přes kódování a ladění programů až po dokumentaci včetně zajišťování potřebné údržby. V závislosti na rozsahu projektu je tým dále doplněn 2 až 3 programátory, případně dalšími specialisty.

Technickým vedoucím týmu je hlavní programátor, který udržuje organizační disciplínu a nese zodpovědnost za projekt. Podstatnou částí náplně jeho práce však také je navrhovat a kódovat centrální kritické segmenty programového systému. Vymezuje rovněž programy nebo moduly pro své spolupracovníky a kontroluje jejich vývoj.

Zástupce hlavního programátora se účastní návrhu klíčových problémů a veškerých důležitých akcí do té míry, aby byl schopen kdykoliv převzít vedení projektu. Plní v případě potřeby též funkci výzkumného asistenta v programové strategii a taktice a umožňuje tak hlavnímu programátorovi plné soustředění na nejdůležitější problémy.

Programový knihovník je zodpovědný za údržbu knihoven a za přípravu, dokumentaci a provedení všech ladicích a ověřovacích prací. Není to však pouhý společný asistent

programátorů, ale rovnocený člen týmu. Zajišťuje, aby byly v platném stavu knihovny a výpisy zdrojových programů, připravuje použitelné programy v jazyku stroje a běžná ladicí data. Další členové týmu mohou pracovat efektivněji, s větším pořádkem a s méně zbytečnými omyly. Kromě toho je kdykoliv možné kontrolovat stav a průběh prací. Vedlejším výsledkem této centralizované činnosti je významné šetření papírové a administrativní práce programátorů.

5.3. Poznatky z ověřovací praxe

Výše popsané zásady uspořádaného programování byly ve VS Ingstav aplikovány při tvorbě několika desítek programů, převážně v jazyce Cobol Tesla 200. Je prokazatelnou skutečností, že při dodržování pravidel dle kapitoly 5.1. se výrazně snižují počty nutných kompilací a zvyšuje kvalita a logická správnost výsledných programů. Je pravdou, že je nutno věnovat větší péči vlastnímu programování, tj. tvorbě logiky zdrojového programu, ale docílený zisk při zjednodušení ladění a údržby tyto nároky mnohonásobně převýší.

Z pravidel organisace pracovních týmů, popsaných v kapitole 5.2. je především důsledně využívána osoba programového knihovníka k zajištění údržby zdrojové knihovny COBOL-LIBR i pro obhospodařování odladěných programů na uživatelské knihovně projektu (ULP). Zatím se nepodařilo docílit, aby se hlavní programátor podílel ve větší míře přímo na kódování nejdůležitějších míst a to především z časových důvodů. Ponětvadž je trvale zatížen návalem jiné, převážně analytické a organisační práce, čekání na jeho programátorské výsledky by mohlo často brzdit celý průběh vývoje programů.

Kontrolu logiky programů se však daří realisovat tím, že každý program má kromě řešitele určeného též oponenta, zpravidla některého ze zkušenějších pracovníků. Ten potom dokáže svým nezaujatým přístupem buď sám odhalit řadu chyb, nebo funguje jako "katalysátor" a svými věcnými dotazy

urychlí správnou reakci autora. Oponentura se provádí nejprve nad hrubým vývojovým diagramem a architekturou programu a potom nad prvním opisem štítků zdrojového textu. Teprve po opravení zjištěných chyb se zadává kompilace Cobol. Při oponentuře se klade důraz nejen na logickou a věcnou správnost, ale i na vzhledovou formální úpravu programů, tj. čitelnost a srozumitelnost.

Uspořádanou formou bylo možno též poměrně snadno srozumitelně popsat logickou konstrukci základních, nejčastěji používaných algoritmů pro konverzi, slučování, součtování a pod. Tyto materiály /18/ se staly vítanou pomůckou, především pro začínající programátory.

6. Závěr

Okolo nových metod, které jsou předmětem tohoto příspěvku se poměrně dost hovoří i píše, ale jejich praktické aplikace nejsou zatím zdaleka prováděny v takové míře, jak by si zasluhovaly. Je to veliká škoda už proto, že k použití filozofie systémového přístupu jako nástroje pro ovládnutí logiky programu není třeba čekat na žádné nové programové jazyky, speciální školení a pod., ale stačí se zamyslet nad problémem a najít v sobě odvahu překonat trochu konvence.

Často se v této souvislosti ohnivě diskutuje o používání či nepoužívání příkazu GO TO. To však vůbec není podstata věci! Je zcela jasné, že pokud v assembleru není instrukce PERFORM nebo DO, bude nejspíš potřeba ji nahradit skokem do podprogramu a návratem zpět. Na druhé straně je však nutno konstatovat, že nejlepší cesta, jak se např. v jazyku Cobol naučit konstruovat program uspořádaným způsobem je zřeknutí se příkazu GO TO úplně. Toto tvrzení je založeno na praktickém ověření, že takový postup skutečně dovede programátora k přehodnocení dosavadní volnosti myšlenkových postupů k poměrně přísně uspořádané formě. Musí se totiž daleko hlouběji zamyslet nad logickou konstrukcí procedury, nad vstupy na jednotlivých rozlišovacích úrovních a celkově se dostane

podstatně blíže k vystižení jádra problému.

Cílem našeho snažení by mělo být ovládnutí procesu výpočtu, tj. dokonalé zvládnutí logiky každého programu. Většinou však máme co činit s mnoha značně složitými vzájemnými vztahy a vazbami, pracujeme s instrukcemi, jejichž počty jdou do stovek a šíře a podrobnost lidského poznání je přece jen jistým způsobem limitována. Je třeba si tedy problém, v našem případě algoritmus programu, přizpůsobit tak, aby bylo možno řešení provést s podstatně menším počtem ovlivňujících faktorů. A to je právě cesta využití rozlišovacích úrovní a prvků při systémovém přístupu. Ovládat věci, to vlastně znamená stát v jistém smyslu nad nimi, tj. být na vyšší rozlišovací úrovni. A potom je možno také vidět další širší souvislosti, které by jinak unikly.

Dalo by se zde ještě hovořit o konkrétních výsledcích oponentur, o vztazích mezi programátorskou individualitou a kolektivní týmovou prací a o řadě dalších zkušeností. To by ovšem znamenalo daleko překročit plánovaný rozsah tohoto příspěvku, takže rozbor těchto otázek ponechme pro některou jinou příležitost.

7. Soupis podkladů a související literatury

/1/ D.D.Mc. Cracken, Revolution in Programming, Datamation, Vol. 19, No 12, Dec. 1973, 50-52

/2/ J.R. Donaldson, Structured Programming, Datamation, Vol. 19, No 12, Dec. 1973, 52 - 54

/3/ E.F. Miller, G.E. Lindamood, Structured Programming Top-down Approach, Datamation, Vol. 19, No 12, Dec. 1973, 55 - 57

/4/ F.T. Baker, H.D. Mills, Chief Programer Teams, Datamation Vol. 19, No 12, Dec. 1973, 58 - 61

/5/ Demner, Organizace řešitelského týmu v projektu Beta, Informační systémy 4/1974, 361 - 368

/6/ J. Hořejš, Strukturované programování, Sborník ze semináře VVS ČSM Bratislava - Sofsem 74, 11 - 58

/7/ J. Hořejš, Strukturované programování I. a II., Informační systémy 2 + 3/1975

/8/ J. Hořejš, Principy strukturovaného programování (strukturování dat a programů), Sborník ze 3. sympózia SVTS Bratislava - Algoritmy vo výpočtovej technike 1975, 338 - 349

/9/ V. Čevela, Využití prostředků jazyka Cobol Tesla 200, pro uspořádanou logiku a organizaci programování, Sborník přednášek "Používání jazyka Cobol", DT ČVTS Pardubice 1975, 55 - 65

/10/ V. Čevela, Uspořádaná logika a organizace programování, Mechanizace automatizace administrativy 2/1975, 62-66

/11/ V. Čevela, Algoritmy základních úloh hromadného zpracování dat metodou uspořádaného programování, Knižnica Algoritmov III. diel z 3. sympózia SVTS Bratislava - Algoritmy vo výpočtovej technike 1975, 41/400 - 41/412

/12/ V. Čevela, Uspořádaná logika a organizace programování, Sborník ze semináře "Racionalizace analyticko-programátorské práce a provozu VS", Pobočka ČVTS Ingstav Brno 1975

/13/ V. Čevela, Norma řízení VS Ingstav a zkušenosti s jednotnou metodikou budování ASŘP, Sborník ze semináře "Racionalizace analyticko-programátorské práce a provozu VS", Pobočka ČVTS Ingstav Brno, 1975

/14/ M. Tuček, Technologie konstrukce a zápisu uspořádaného programu, Sborník ze semináře "Racionalizace analyticko-programátorské práce a provozu VS", Pobočka ČVTS Ingstav Brno, 1975

/15/ B. Lacko, Standartizace programování v praxi výpočetního střediska, Informační systémy 4/1974, 351 - 359

/16/ J. Kroiský, Strukturované programování a prvé skúšenosti s jeho použitím v jazyce PL/I, Sborník ze semináře "Metody programování počítačů III. generace", DT ČVTS Ostrava, 1975

/17/ M. Šilar, Metody navrhování programového vybavení, Mechanizace automatizace administrativy 11/1975, 443 - 446

/18/ V. Čevela, Algoritmy základních úloh zpracování hromadných dat metodou uspořádaného programování, Interní materiál E00002 VS Ingstav Brno, květen 1975

/19/ ACM - Computing Surveys, Special Issue: Programming, Vol. 6, Num. 4, Dec. 1974

/20/ A.P.Jeršov, Estetika a lidské aspekty programování Metro (PCRS) 2/1973, 10 - 13

Brno, únor 1976