Ing. Josef BRZICKÝ
Ústav výpočetní techniky SPK, Praha

STRUKTUROVÉ PROGRAMOVÁNÍ A PRVÉ ZKUŠENOSTI S JEHO POUŽITÍM V JAZYCE PL/I

Pro současnou dobu je při posuzování nákladů na počítačové zpracování charakteristická tendence k růstu podílu softwarových nákladů na celkových nákladech. Na tomto růstu se na jedné straně podílejí zvyšující se dodávky programů nebo programových soustav od výrobních či specializovaných softwarových firem. Na druhé straně se téměř v každém výpočetním středisku stále výrazněji projevuje potřeba vypracovat programové prostředky vlastními silami pro krytí vlastních speciálních požadavků. Navíc se prohlubuje nedostatek programátorské kapacity. Proto se zákonitě věnuje stále více pozornosti otázkám racionalizace tvorby programů, zejména otázkám racionalizace techniky programování.

Objevila se nová hesla dne: strukturované programování /structured programming/ a metoda týmu vedoucího programátora /chief programmer team/. V některých zahraničních časopisech jsou tato hesla skloňována ve všech pádech. Jde v této souvislosti o náhlé převratné objevy, které rázem změní techniku programování? Domnívám se, že jde o logický vývoj vycházející z dosavadní programovací praxe.

Jaký je dnešní stav v oblasti techniky programování? Každý jednotlivý programátor povětšině programuje "jak mu jazyk narost", jeho programy mají své osobité rysy. Jestliže

říkáme, že každý programátor si vytváří svůj osobitý samorostlý styl, neznamená to, že tento styl je vždy dobrý. Spíše naopak. A není také divu, vždyť ve většině případů dostane začínající programátor kurs určitého jazyka, kde se dozví formální pravidla pro jeho užívání, ale nedostane návod, jak má efektivně klást jednotlivé příkazy jazyka za sebe, aby dosáhl cíle propočtu. A tak kromě kursu jsou mu vodítkem snad jen programy kolegů, které dostane jako vzor, a vlastní fantazie. Takto vytvořený individuální styl

- není efektivní při sestavení programu, sestavení programu trvá velmi dlouho,
- není vždy efektivní při provádění, zejména výpočet podle programu trvá příliš dlouho v důsledku přehmatů způsobených nepřehledností programového textu,
- ztěžuje zásahy do programu, a to nejen ze strany cizích programátorů, ale i vlastní zásahy autora. Ty jsou ve většině aplikací nutné, protože je třeba funkce programů rozšířit nebo změnit,
- svou nesystematičností je zdrojem chyb v programu, prodlužuje se doba ladění a vznikají nepříjemnosti s dodatečným zjišťováním chyb v době, kdy je program už v rutinním provozu.

Zejména poslední dva body mají v praxi značný negativní dopad. Zkušenosti ukazují, že mnohdy je nutno nesystematicky, živelně vybudované programové systémy v důsledku i navenek poměrně drobných úprav zadání podstatně přepracovávat či prakticky znova napsat. Tím produktivita podle odhadů klesá na polovinu své reálné hodnoty /při dodržení racionálního postupu/ i více. Odhaduje se, že v důsledku nejrůznějších časových ztrát je běžný programátor schopen vyprodukovat v dlouhodobém průměru pouze několik instrukcí ve vyšším programovacím jazyku denně. Ojedinělé pokusy různých organizací, zavést pro programování některá formální pravidla pro stavbu programu, vystřídalo široké hnutí

po racionalizaci programátorského řemesla. Podíváme se jaké požadavky přinášejí nově proklamované metody stavby programu, nejčastěji shrnované pod pojmem strukturované programování.

Jsou to trochu nesourodé požadavky od vyloženě primitivních /i když účinných/ až po požadavky zásadnějšího charakteru. Žádný z těchto požadavků není absolutní novinkou, vyplynul z praxe mnoha programátorů. Některé z nich se pokusím charakterizovat prostřednictvím příkladů ve vyšším programovacím jazyce PL/I /DOS/.

Z primitivních požadavků lze uvést požadavek na přehledné odstavcování programu. Tento požadavek lze v jazyce PL/I, který umožňuje velmi flexibilní zápis programu, lehce splnit. Znamená to různými úrovněmi začátků řádky programu /která se zpravidla rovná děrnému štítku/ výrazně odlišit logické části programu. Takovými logickými částmi jsou v PL/I-programu např. procedury, BEGIN-bloky, DO-skupiny, DO-smyčky, větve v příkaze IF, části popisů dat /zejména ve strukturách/. Do této skupiny požadavků patří také vytknutí návěští. Různou úrovní začátků řádky se zvýrazní počátek a konec jednotlivých logických částí, případně jejich hierarchická struktura.

Dalším primitivním, i když pro zvýšení srozumitelnosti programu velmi účinným prostředkem je použití samovysvětlujících identifikátorů. Tedy identifikátorů, z jejichž znění by byl na prvý pohled zřejmý jejich význam. Při dodržování této zásady je třeba nalézt vhodnou střední cestu tak, aby použité identifikátory neměly ani příliš mnoho znaků /PL/I umožňuje až 31 znaků/, ani příliš málo a tím byly nesrozumitelné. Zde dosti záleží na charakteru úlohy, nicméně i u vědeckotechnických úloh, kde je tradiční užívání velmi krátkých identifikátorů, by tato zásada měla být rozumně dodržována.

Většina vyšších programovacích jazyků dává programátorovi možnost zařazení poznámek do textu programu. PL/I to umožňuje v libovolném místě programu /všude tam, kde může být mezera/. Vysvětlující poznámky by ovšem neměly být formální, měly by

být zařazeny na místa při běžné četbě programu obtížněji pochopitelná. V praxi mohou být poznámky používány i k dokumentačním účelům. Často se doporučuje na počátku programu uvést v poznámkách základní charakteristiku programu včetně údajů o autorovi, datu napsání programu, čísla verse programu, nárocích na paměť, stupni utajení apod. Hlavní program pak může obsahovat na začátku čokonce jakýsi "obsah", kde v poznámkách jsou uvedeny názvy a charakteristiky podřízených částí.

Od těchto jasných a pravděpodobně obecně přijímaných požadavků se dostáváme k zásadnímu požadavku strukturovaného programování, kterým je hierarchický přístup k budování programu nebo systému programů s použitím modulární techniky. Hierarchický přístup znamená rozčlenění programu do několika stupňů ve formě stromu, kde na vrcholu je řídící modul, který vyvolává základní funkce zařazené jako podřízené funkční moduly ve druhém stupni hierarchického stromu, ty opět mohou dále volat podřízené moduly na třetím stupni atd. Hloubka hierarchie je dána složitostí a rozsahem řešeného problému.

Jako jednoduchý příklad strukturovaného hierarchického přístupu lze uvést z praxe Ústavu výpočetní techniky Státní plánovací komise vytvoření programového celku SVPI /Standardizovaný Vstup Plánové Informace/. Cílem SVPI je

a/ snímat plánové informace vyděrované jedním z několika standardizovaných způsobů do děrných štítků

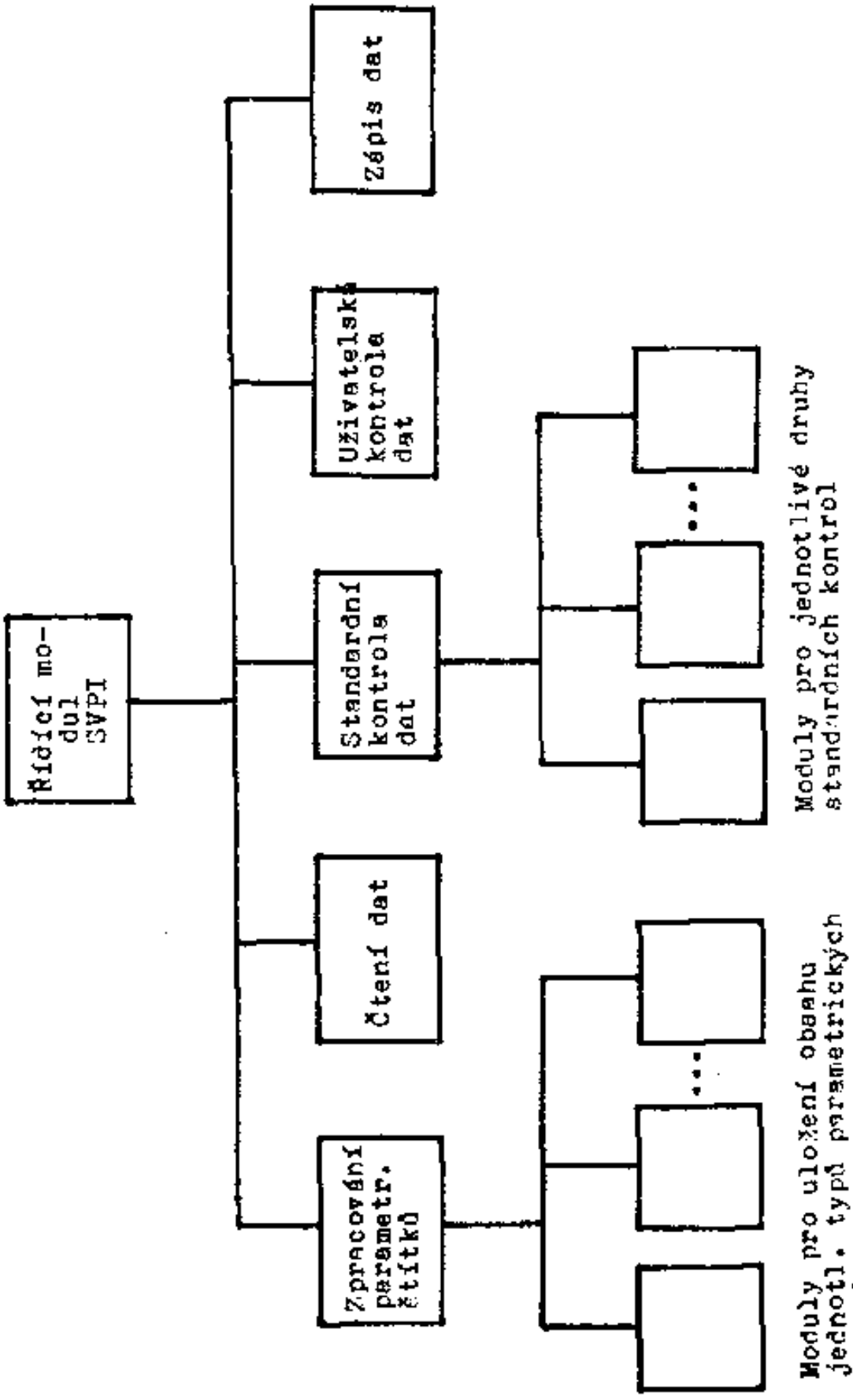
b/ provést kontroly specifikované v parametrických štítcích podle standardních schemat s výpisem chyb

c/ provést kontroly podle speciálních požadavků uživatele /které nejsou ve standardních možnostech/

d/ zapsat zkontrolované správné data standardizovaným způsobem na vnější paměti.

Schema na obr. 1 ukazuje rozdělení programového celku do modulů podle funkcí. Řídící modul vedle popisů a některých pomocných úvodních a závěrečných operací obsahuje volání pěti podřízených modulů ve druhém stupni, ty pak případně volají moduly

Obr. 1 Příklad hierarchické výstavby programového celku



na třetím stupni. Tento přístup má kromě jiného i tu výhodu, že dovoluje snadno použít overlay techniku šetřící vnitřní paměť.

Realizaci modulární výstavby v jazyce PL/I umožňuje existence příkazů PROCEDURE, END a CALL. Ty slouží definici procedur /= modulů/ a jejich vyvolání. Procedury mohou být interní a tedy nesamostatné, závislé na proceduře, ve které jsou obsaženy, a externí, které jsou předkládány samostatně. Podle toho co bylo řečeno, by se měl program vytvořený podle zásad strukturovaného programování skládat z několika modulů definovaných jako externí, případně interní procedury. V zájmu přehlednosti délka modulu by neměla přesáhnout jednu stranu tištěného výstupu při překladu. Každý z modulů je vytvářen sledem příkazů CALL a případně dalších příkazů jazyka. Pro strukturované programování je charakteristická tendence k odstranění, případně omezení příkazu GOTO z tohoto sledu.

Požadavek na likvidaci GOTO, který se objevoval od roku 1968 nejprve spíše v teoretické poloze, se ozývá přes bouři odporu u některých praktiků stále hlasitěji. Příkaz GOTO je označován za hlavní příčinu nepřehlednosti a nesrozumitelnosti programu. Je pravdou, že GOTO je zneužíváno. Jde však programovat absolutně bez použití GOTO? Teoretici dokazují, že ano. Ke tvorbě programu podle nich stačí tři prvky:

- prostý sekvenční sled příkazů
- rozhodování
- opakování

Rozhodování je v PL/I realizováno pomocí příkazu IF, opakování pomocí DO-smyček. Prostý sekvenční sled je tvořen všemi ostatními příkazy s výjimkou GOTO. Významné místo mezi těmito příkazy zaujímá už výše zmíněný příkaz CALL, který je spolu s příkazem DO hlavní náhražkou za GOTO. Tím příkaz CALL získává proti nejčastějšímu dosavadnímu použití, kterým bylo volání podprogramů, nový význam. V jednom případě však v subsetu

PL/I příkaz GOTO nahradit nemůžeme - v příkazu ON, kde vedle málo užívaného prázdného příkazu a klíčového slova SYSTEM, je GOTO jedinou praktickou možností.

Pro literaturu o strukturovaném programování je typické malé množství praktických příkladů. Pokusíme se dokázat, že užití příkazu GOTO lze v programu omezit.

V prvém příkladě jde o provedení několika sekvenčních sledů příkazů, z nichž některé jsou v závislosti na dvou podmínkách přeskakovány. Nejprve tradiční způsob s použitím příkazů GOTO:

```
     IF SPOTREBA  > 1000 THEN GOTO POKR1;
     .
     .  příkazy 1
     .
     IF KOEF * 1.05 THEN GOTO POKR2;
     .
     . příkazy 2
     .
POKR2: .
       . příkazy 3
       .
       GOTO POKR3;
POKR1: .
       . příkazy 4
       .
POKR3:
```

V tomto jednoduchém případě vystačíme pro odstranění GOTO se zavedením ELSE větve a s DO-skupinami:

```
     IF SPOTREBA < = 1000 THEN DO;
                      .
                      . příkazy 1
                      .
                      IF KOEF¬= 1.05 THEN DO;
                                          .
                                          . příkazy 2
                                          .
                                          END;
                      .
                      . příkazy 3
                      .
                      END;
                 ELSE DO;
                      .
                      . příkazy 4
                      .
                      END;
```

Ve druhém příkladě se pokusíme odstranit příkaz GOTO z testování konce souboru a ze smyčky pro čtení souboru. Tradiční postup:

```
CTI: READ FILE (DS) INTO (STITEK);
     IF SUBSTR (STITEK,1,5) = '99999' THEN GOTO POKR;
     .
     . příkazy 1
     .
     GOTO CTI;
POKR:
```

Lze nahradit tímto postupem

```
DCL EOF BIT (1) INIT ('0'B);
DO WHILE ¬EOF;
          READ FILE (DS) INTO (STITEK)
          IF SUBSTR (STITEK,1,5) = '99999' THEN EOF = '1'B;
                                           ELSE DO;
                                                .
                                                . příkazy 1
                                                .
                                                END;
END;
```

Ve třetím příkladě je testování konce souboru provedeno obvyklejším způsobem pomocí příkazu ON. Zde bohužel v subsetu PL/I, který je v této přednášce pro příklady používán, příkaz GOTO odstranit nelze. Tradiční postup:

```
ON ENDFILE (DS) GOTO POKR;
CTI: READ FILE (DS) INTO (STITEK);
     .
     . příkazy 1
     .
     GOTO CTI;
POKR:
```

S omezením GOTO:

```
Nadřízený modul:    DO WHILE ¬EOF;
                       CALL CTI;
                       END;
```

```
Podřízený modul:   CTI: PROC;
                        ON ENDFILE (DS) GOTO KNC;
                        READ FILE (DS) INTO (STITEK);
                        .
                        . příkazy 1
                        .
                        RETURN;
                       KNC: EOF = '1'B;
                      END CTI;
```

V tomto příkladě je řešení bez GOTO na prvý pohled značně komplikovanější. Nicméně zjednodušení, větší přehlednost, se projeví v nadřízeném modulu zejména tehdy, je-li sled "příkazy 1" delší a složitý.

Ve čtvrtém příkladě je ukázka převedení postupu, který užívá GOTO pro opakování a přeskakování, na postup využívající DO-smyček:

```
SUMA: IF BETA = 1000 THEN GOTO POKR;
      .
      . příkazy 1
      .
      IF OMEGA = 3 THEN GOTO SUMA;
OPAK: IF ALFA <500 THEN GOTO MEZ;
      .
      . příkazy 2
      .
      . GOTO OPAK;
MEZ:  .
      . příkazy 3
      .
      GOTO SUMA;
POKR:
```

<u>Řešení:</u>

```
DO WHILE BETA ¬= 1000
   .
   . příkazy 1
   .
   IF OMEGA ¬= 3 THEN DO;
                         DO WHILE ALFA <500;
                            .
                            . příkazy 2
                            .
                          END;
```

```
                    . příkazy 3
                    .
                    END;

END,
nebo
Nadřízený modul: DO WHILE BETA ¬= 1000;
                    CALL VYP;
                 END;

Podřízený modul: VYP: PROC;
                 .
                 . příkazy 1
                 .
                 IF OMEGA = 3 THEN RETURN;
                 DO WHILE ALFA < 500;
                    .
                    . příkazy 2
                    .
                 END;
                 .
                 . příkazy 3
                 .
               END VYP;
```

V tomto příkladě, právě tak jako ve třetím bylo porušeno pravidlo, které prosazují někteří teoretici strukturovaného programování: <u>modul má mít pouze jeden vstupní a pouze jeden výstupní bod.</u> V praxi se však druhá část tohoto požadavku splňuje s obtížemi.

Bylo uvedeno několik spontánně vytvořených ukázek pro programování bez GOTO. Nová řešení si nekladou nároky na větší efektivnost při výpočtu. Hlavním cílem je dosáhnout přehledného sekvenčního sledu příkazů, snadno čitelného i pro neautora. Nicméně nedomnívám se, že je nutno brát likvidaci příkazu GOTO stoprocentně. Někde je to spojeno s příliš velkými komplikacemi. Ostatně v literatuře někteří autoři doporučují např. nelikvidovat vypočítávané GOTO, které je v jazyce PL/I realizováno formou GOTO návěští - proměnná /výraz/.

Nový hierarchický, vertikální způsob návrhu programového schématu /je někdy také označován jako přístup top - down/ si vyžaduje na rozdíl od tradičního horizontálního způsobu, vycházejícího z tradičního blokového schématu, často nový analytický přístup. Ne vždy lze snadno přeprogramovat starší "horizontální" programy bez nové programové analýzy.

Z hlediska volby programovacích jazyků lze pro strukturované programování dobře použít zejména PL/I a Algol. Fortran a Cobol jsou již méně vhodné, i když i u nich lze metody strukturovaného programování použít. Assembler je relativně nejméně vhodný.

V souvislosti se strukturovaným programováním se často hovoří o <u>konceptu pevně organizovaných programátorských týmů s jedním vedoucím</u> /chief programmer team/. Tento koncept byl vytvořen v USA firmou IBM. Podle autorů projektu přinesl velké zvýšení efektivity programování a minimum chyb ve vytvořených programech. Zde nejde o vlastní techniku programování, ale o racionalizaci organizace programování větších softwarových systémů. Navrhuje se vytvoření týmu vedeného velmi dobrým programátorem, kde by dále byl zajišťovací programátor, sekretář týmu a 3 - 5 programátorů. Navíc může mít tým ještě úředníka, který obstarává všechny administrativní práce.

Vedoucí programátor programuje hlavní moduly a řídí a kontroluje tvorbu všech ostatních modulů, které tvoří ostatní programátoři. Zajišťovací programátor je pobočníkem vedoucího, podrobně zná všechny moduly, je připraven kdykoliv zaskočit za vedoucího. Sekretář týmu zprostředkovává všechny ladící běhy /programátoři tedy sami práce nezadávají/, udržuje dokonalou dokumentaci o projektu včetně evidence všech úspěšných i neúspěšných běhů, jak v písemné formě, tak ve zvláštní knihovně DSL /development support library/.

K základním rysům konceptu týmu vedoucího programátora patří:

- specializace programátorů pro jeden typ programovací práce

- přesné vymezení vztahů mezi specialisty
- disciplína, teamwork
- užití strukturovaného programování
- užití DSL, což je knihovna uložená na vnějších pamětech počítače, kde jsou všechny vyvíjené programy udržovány ve standardizované formě pod správou sekretáře týmu.

S konceptem týmu vedoucího programátora je spojena řada standardů, které musí programátoři dodržovat. Ty zato umožňují, že částem vytvářeného projektu rozumí všichni členové týmu. Tím přestává být tvorba programů soukromou věcí jednotlivých programátorů, ale "veřejnou záležitostí".

Tolik stručná informace o některých progresivních směrech v technice a organizaci programování. I když většina uvedených postupů není v programovací praxi zbrusu nová, je poučný systematický přístup k racionalizaci programovací práce. Tato tendence je zřejmě obecná a je na nás, abychom včas tento nový proud zachytili.