TECHNOLOGIE STRUKTUROVANÉHO PROGRAMOVÁNÍ
PROGRAMÁTOR - PROFESIONÁL ?

Dr. Jiří Suchomel
Orgaprojekt Praha

1. Úvod

V příspěvku se autor pokusil zasadit dříve na těchto místech přednášenou "Technologii strukturovaného programování - M.A.Jackson" do reálného prostředí návrhu a realizace programových systémů. Otázky spojené s touto problematikou nebylo možné objasnit při výkladu metody samotné a právě důsledné dodržování výchozích zásad zaručuje plnou úspěšnost metody v praxi.

Na realizaci programového díla řešícího danou úlohu v počítačovém prostředí se v podstatě podílejí tři profesní oblasti. U ž i v a t e l , specifikující úlohu, a n a l y t i k , akceptující specifikace úlohy a vytvářející tzv. systémový návrh, a konečně p r o g r a m á t o r , který realizuje vlastní programové řešení dané úlohy.

Uživatel je odborně fundovaný v oblasti svého problému, zná všechny své informační objekty, zná pracovní postupy nad svou informační základnou a má dostatečně přesnou představu o rozsahu a náplni všech činností ve své agendě. Uživatel nepotřebuje znalosti z oboru věd o počítačích.

2. Systémový analytik - programátor

Programování se tradičně odděluje od návrhu systému. Návrhář systému - systémový analytik - rozhodne, jaké soubory a programy v systému potřebujeme, a specifikuje je programátorovi. Programátor pak programy sestavuje podle těchto specifikací. Ukažme, jak tato dělba práce je absurdní, jak negativně ovlivňuje chápání programových systémů a způsob jejich realizace.

Za prvé, pomohla zvěčnit názor odvozený z prvních dávkových systémů, že existuje pevná hranice mezi úkoly návrhu systému a návrhu programu. Jiných metod, kritérií a nástrojů používáme, jsou-li prvky našeho návrhu programy a soubory, a jiných zase, jsou-li jimi podprogramy, příkazy programovacího jazyka a hodnoty v operační paměti. Tento rozdíl se nám nyní začíná stírat: Uvědomujeme si, že "řídícím jazykem prací" je programovací jazyk a že slovo "program" v kontextu komunikačního systému ztrácí hodně na svém významu.

Za druhé, zastřela povahu práce systémového analytika. Od něj se očekává, že bude dělat dvě velice různorodé práce: Musí analyzovat aplikační požadavky, aby bylo možné určit, co má systém provádět, má-li svému uživateli dobře sloužit, a zároveň musí navrhovat vyšší úrovně tohoto systému a konfigurovat programy a soubory tak, aby práce na počítači byla prováděna efektivně. První práce vyžaduje znalost odpovídajících úseků řízení, druhá práce vyžaduje znalosti z oboru věd o počítačích. Jen neobyčejně univerzální člověk dokáže oba tyto úkoly zvládnout úspěšně. V praxi se u analytiků často setkáváme s tendencí soustředit se na tu práci, kterou mají raději. Potřeby uživatele jsou příliš často zvažovány nedostatečně. Únavná a jednotvárná práce pochopení aplikace a naplánování ergonomicky zdravého systému je uspěchána, aby se mohlo začít s oblíbeným vytvářením vývojových diagramů a uspořádáváním formátů souboru a záznamů. Výsledné specifikace systému příliš často sestávají z povrchního popisu toho, co bude systém řešit, a z podrobného a láskyplného výčtu, jak to bude řešit.

Třetí vliv, vliv na programátory, se nás týká nejadresněji. Na mnoha instalacích nemá pracovník při aplikačním programování možnost postupu. Horní hranicí práce je návrhářská činnost - kterou se již zabýval analytik - a její dolní hranice je průběžně posunována nahoru zaváděním strojově nezávislých jazyků, generátorů sestav a programového vybavení počítače orientovaného na základnu dat. Technicky ctižádostiví

programátoři unikají do programování systémů, v němž se stanou znalci spletitého výrobcova technického vybavení počítače. Programátoři nároční finančně nebo postupově se stanou systémovými analytiky. A ti, kteří zůstanou u aplikačního programování, často propadají pochopitelnému, ale katastrofálnímu sklonu ke složitosti a důmyslnosti své práce. Se zákazem navrhovat cokoliv většího než program se vyrovnávají tím, že vytvářejí program dostatečně komplikovaný na to, aby mohl soupeřit s jejich odbornou kvalifikací.

Pokusme se v dalším ignorovat rozdílnosti mezi návrhem systému a návrhem programu a stanovit hlavní zásady obecného návrhu.

3. Programová složka

Pro danou úlohu budeme vždy navrhovat "programovou složku", přičemž nejsou důležité realizační prostředky a prostředí, důležité jsou pouze specifikace úlohy. Sem zahrnujeme datové objekty, nad kterými úloha pracuje, a algoritmy v úloze používané. Naší prvotní snahou bude vyřešit úlohu v obecné rovině, odděleně od specifičnosti dané konfigurace výpočetního systému.

Při návrhu programové složky řešící naši úlohu použijeme metodu návrhu shora dolů. Výsledný strukturální diagram je strom. Dekompozice úlohy je úspěšná pouze tehdy, daří-li se rozkládat specifikace úlohy tak, aby vazby mezi prvky struktury byly pouze svislé. Jinak řečeno: Rozložíme-li úlohu T na úlohy T1, T2, ..., pak kontext úlohy T1, T2, ... je celý určen jedině v T. Při dekompozici úlohy složené na úlohy jednodušší budeme uvažovat pouze tyto možnosti:

- Složená úloha obsahuje své jednodušší úlohy v jednoznačné sekvenci.
- Složená úloha obsahuje celý nezáporný počet jediné jednodušší úlohy.
- Složená úloha je jediná z celého kladného počtu úloh jednodušších.

- Úloha nemá části na dané rozlišovací úrovni.

Např. naše metoda nahrazuje rozklad úlohy rozkladem všech datových objektů figurujících v úloze. HIPO diagramy a příbuzné metody doporučují dekompozici funkcí v úloze.

Ale metoda návrhu shora dolů není jedinou metodou návrhu. Existuje také její opak, návrh zdola nahoru. U návrhu shora dolů vlastně říkáme: "Tato úloha je pro nás příliš složitá. Rozložíme si ji na řadu úloh menších a ty na ještě menší, dokud nedostaneme soubor úloh, které jsou dostatečně jednoduché, abychom je mohli vyřešit." U návrhu zdola nahoru místo toho říkáme: "Tento počítač mé úloze nevyhovuje, protože základní operace jsou příliš základní. Použijeme proto základních operací a vytvoříme výkonnější počítač a operace tohoto počítače použijeme k vytvoření ještě výkonnějšího počítače, který moji úlohu snadno vyřeší." U návrhu zdola nahoru se naší úloze věnujeme minimálně. Hlavním podnětem k práci je vytvoření nového počítače, který bude vhodnější než starý, a to i pro řešení naší úlohy.

Moderní metody návrhu programových složek jsou silným nástrojem návrhu a dávají velmi dobré výsledky. Přesto ne vždy je programátorská obec spokojena, žádá "lepší" metody pro řešení svých úloh nebo používá "osvědčené umělecké" či "pokusné" metody, které praxí stále upřesňuje. Proč?

4. Společné programové složky

Velkým nebezpečím v etapě návrhu programu jsou návrhy "společných programových složek" pro shodné dílčí úlohy. Pozdější realizace takovýchto programů zavléká do programového díla velké těžkosti při údržbě, tj. při činnostech reagujících na změny a doplňky ve specifikacích původní úlohy. Podobnými těžkostmi, ale v daleko větší míře, nám "prospívá" optimalizace v etapě návrhu. Jedinou možností, jak se vyhnout těmto nebezpečím, je dodržovat v etapě návrhu programové složky důsledně m e t o d u n á v r h u s h o r a d o l ů a nebát se velkého množství triviál-

ních úloh. Přínosem bude převedení složité úlohy na úlohu, jejíž řešení je pouze pracné.

Jestliže lze nepoužít v etapě návrhu "společné programové složky" ani metodu optimalizace, není to vždy možné v etapě realizace programového díla. O optimalizaci jen málo: Existují úlohy, kde optimalizace programového díla je pouze škodlivá. Přínos optimalizace je minimální a je znehodnocen nesrozumitelností a nesnadnou údržbou programového díla. Existují úlohy, kde optimalizace je nutná, ale i zde musíme mít nejdříve neoptimalizovaný návrh, a pak: Není možné bez nebezpečí zavlečení logických chyb optimalizovat optimalizované programové dílo.

5. Základní programové složky

"Společné programové složky" v etapě realizace používají všichni programátoři, bez jakýchkoli rozpaků a s úspěchem. Ti, kteří programují ve vyšších programovacích jazycích, používají příkazy svého programovacího jazyka a neuvažují o tom, jak kompilátor realizuje stejné příkazy v různých místech textu. Někdy realizuje každý příkaz jednotlivě /MACRO-ASSEMBLER/, někdy společně /FORTRAN/. Příkazy nazýváme "základní programové složky" a není pro nás důležité, jak jsou realizovány a jak byly navrženy, důležité jsou specifikace základních programových složek. Dobrá znalost těchto specifikací dovoluje programátorovi používat základní programové složky v textu programu.

Pro to vše je jedině možné používat společné programové složky pouze jako základní programové složky. Tyto obecně použitelné programové složky nemohou vzniknout na základě návrhu shora dolů, návrh shora dolů může dát pouze impuls k realizaci takové programové složky; tyto programové složky mohou a měly by být vytvářeny postupem návrhu zdola nahoru. Tak uživatel návrhu shora dolů musí takové složky považovat za základní programové složky, se kterými pracuje a které tudíž nemají být vnitřně zkoumány.

Stará dobrá zavedená praxe: Snahou a povinností každého programátora je navrhovat tak, aby velká část jeho zápisu mohla být použita pro více než jeden účel, čímž se optimálně využije drahocenná paměť. Tato námitka obsahuje odpověď: Společné programové složky, jiné než základní programové složky, jsou prostředkem optimalizace.

6. Etapa návrhu a realizace

Řešitel úlohy - návrhář programových složek - bude tedy úspěšný, rozdělí-li důsledně etapu návrhu od etapy realizace a v každé etapě bude řešit pouze úlohy této etapy. V etapě n á v r h u : Zjišťuje úplnost a bezrozpornost specifikací úlohy, vytváří datové struktury úlohy, vybírá obecné typy základních programových složek pro splnění úlohy, vytváří strukturu programu spolu s podmínkami iterací a selekcí. Výsledkem jeho práce v etapě návrhu je "logické schéma" programu, realizované velmi obecným pseudokódem. Opačně, neumí-li návrhář zapsat svůj návrh řešení pomocí logického schématu, není jeho návrh hotový a pokoušet se o etapu realizace je bezpředmětné. V etapě r e a l i z a c e : Programátor popíše explicitně všechny datové položky, zapíše další formální náležitosti programu, vybere vhodné formáty pro zápisy podmínek iterací a selekcí, zvolí formáty základních operací a přepíše logické schéma do programovacího jazyka. Začlení program pod daný operační systém a program odladí.

7. K ladění programu

Naším prvořadým úkolem bylo vytvořit programy, které jsou samozřejmě správné. Existuje ale dostatečně velký prostor pro činění chyb: Při psaní příkazu se můžeme zmýlit v přesnosti jeho umístění, příkaz může být chybně zapsán atd. Jsme proto povinni program testovat, zda neobsahuje formální chyby. Nemůžeme však podstoupit tak velké riziko, že by náš program obsahoval logickou chybu. Všechny logické chyby měly být eliminovány v etapě návrhu.

Nemůžeme doufat, že logické chyby zjistíme pomocí testování. Logické chyby představují kombinace náhod a těch by k testování bylo příliš mnoho. Místo testování musíme přemýšlet a navrhovat tak, aby se testování logických chyb stalo zbytečným.

Je samozřejmě možné navrhnout pro odhalení logických chyb kvalifikovaný testovací soubor. Ale tento soubor nebude kvalifikovaný, nebudeme-li znát podrobně konstrukci laděného programu. A známe-li podrobně konstrukci programu, museli jsme projít celou cestu jeho návrhu, rozpoznat kvalitu tohoto návrhu a tedy i kvalitu programu. Pak víme, že program je správný, poněvadž jsme jej správně navrhli.

8. Platná a neplatná data

V souvislosti se specifikacemi úlohy a návrhem programových složek stojí otázka platných a neplatných dat. Každá programová složka provádí operaci nad datovými objekty. Budeme říkat, že data jsou pro programovou složku platná, je-li operace složky pro tato data specifikována. Naopak - data jsou neplatná tehdy, není-li operace složky specifikována. Je jasné, že data správná i chybná musí být daty platnými.

Např. mějme programovou složku "dělení (a,b,c,P)" s operací a=b/c . Požadujeme, aby a,b,c byly numerické položky a P jméno programové složky ve formě podprogramu. Bude-li c=0 , bude proveden podprogram P. Nyní všechny numerické hodnoty b,c, včetně c=0, jsou platnými daty programové složky "dělení". Pokusíme-li se však dělit položkou nečíselnou, tj. položkou neplatnou, operace není specifikována.

Specifikaci operace programové složky nelze zaměnit s předpověditelností operace programové složky. Máme-li k dispozici text programové složky a známe-li hodnoty datových položek, dokážeme vždy předpovědět operaci programové složky, i když se někdy ukáže, že tato operace závisí na náhodných činitelích, jako je např. uložení cílového programu ve feritové paměti. Nejen to, předpovězená operace je

silně závislá na použitých formátech příkazů jazyka. Při takové závislosti na podrobnostech zápisu je tato předpověď bezcenná. Je zásadně chybné zkoumat operaci programové složky pro neplatná data. Musí nám stačit konstatování, že operace není specifikována.

Neměli bychom se snažit o to, aby operace byla vždy specifikována? Není snad zásadou správného návrhu, aby každá programová složka ověřovala platnost svého vstupu? Neměl by programátor aktualizovat specifikaci programové složky tak, aby odrážela zápis, který on pořídil?

Ne, ne a ne.

Jistěže by každá programová složka, které vstupní data poskytuje přímo lidský činitel nebo nespolehlivý strojový zdroj, měla být navržena tak, aby všechna vstupní data byla platná a operace takto byla vždy specifikována. Ale složky, které získávají svá vstupní data z jiných složek téhož systému, musí být schopné spoléhat na správnou funkci těchto druhých složek.

Máme-li navrhnout programovou složku P pro danou operaci, např. výpočet mzdy za přesčasovou práci, která musí být kratší než 12 hod, a nemůžeme-li se spolehnout na platnost této relace, bude rozumné zapsat:

```
P   sel hod > 11 nebo min > 59
        do chybová rutina PX ;
P   or
        do P1 ;
P   end
```

Nyní musíme navrhnout programovou složku P1 a budeme-li trvat na tom, že operace P1 musí být specifikovány bez ohledu na hodnoty hod a min , musíme psát:

```
P1  sel hod, min chybné
        do chybová rutina P1X ;
P1  or
        do P2 ;
P1  end
```

Nyní ovšem P2 spoléhá na správnou funkci P1 stejně, jako P1 dříve záviselo na správné funkci P. Abychom se takové závislosti vyhnuli, musíme P2 psát jako ... atd. v nekonečném opakování zbytečného zápisu. Máme-li někdy dopsat programovou složku P, musíme se smířit s myšlenkou, že bude obsahovat složky, jejichž operace pro některá vstupní data není specifikována. Některé systémy zpracování dat byly sestaveny na uznávaném, i když protichůdném principu, že každá složka by měla plně ověřovat svá vstupní data. Tato zásada se sama popírá a nemůže být důsledně dodržována. Pak nevyhnutelně jsou některé kontroly prováděny dvakrát i častěji, a některé vůbec ne.

Rozdělme odpovědnost za platná data v systému takto:

- každá specifikace složky musí přesně definovat svá platná data,
- každá složka musí být navržena a zapsána za předpokladu, že bude zpracovávat platná data,
- je-li složka P1 částí složky P, pak P musí zajistit platnost dat pro P1.

Nakonec zjistíme, že je hrubou chybou programátora, když aktualizuje specifikace složky tak, aby odrážela jeho zápis. Není-li původní specifikace přesná, měl by ji objasnit dříve, než začne s navrhováním a kódováním. Jsou-li specifikace přesné, bude muset přesně odpovídat i realizovaná programová složka.

Literatura:

M.A.Jackson - Principles of Program Design
A.P. London, 1975

Sborníky - Programování '79 - '82, Technologie programování
Dům techniky ČSVTS Ostrava