Ing. Vilém Čimbura
Vědeckovýzkumný uhelný ústav, Ostrava

Ing. Josef Tvrdík, CSc
Ústav ekologie průmyslové krajiny ČSAV, Ostrava

PROBLÉMY NÁVRHU MODULÁRNÍCH PROGRAMŮ

1. ÚVAHY NA ÚVOD

Snad v každém odvětví lidské činnosti, které je natolik rozšířené a potřebné, že se stává profesí, se setkáváme s úsilím jednak o vytvoření množiny profesních dovedností a zejména standardních pracovních postupů, jednak o vymezení veličin charakterizujících a klasifikujících produkty této profese, případně jejich fáze. Veličiny i postupy se navzájem ovlivňují, vznikají nové na základě nových poznatků a stimulují tak dynamický vývoj profese k vyšší efektivnosti a racionalitě.

Při dnešních požadavcích na množství a kvalitu a narůstající složitost softwarových produktů nemůže snad už být o existenci profese programování /nebo pro terminologickou neujasněnost řekejme opatrněji analýzy a programování/ pochyb. Už ani bláhovci nepovažují programování za uměleckou činnost vyhraženou pouze vyjímečným jedincům, jejímž hlavním smyslem je neopakovatelný produkt s emocionální působností na autora a jeho nejbližší okolí. Praktické potřeby totiž dosti jasně a někdy velmi tvrdě vymezily nároky na programové vybavení a jednoznačně preferují racionální a solidní, tedy "řemeslné" přístupy. Mimochodem i pouhé nahlédnutí do světa umění nás přesvědčuje, že ani pro vznik uměleckého díla zpravidla nepostačuje okamžik geniálního tvůrčího nadšení a inspirace a že dokonce i umělecká

tvorba vyžaduje důkladnou přípravu a zvládnutí řemesla.

Přes značné pokroky, kterých bylo dosaženo v technologii programování v průběhu posledních dvou desetiletí, jsme však při vývoji a posuzování programového vybavení velmi často v situacích, na jejichž zvládnutí nemáme obecně platný nebo aspoň šířeji uznávaný postup. Diskuse často končí subjektivními tvrzeními typu "tento postup se mi líbí" nebo "návrh A je lepší než návrh B" a nemáme většinou prostředky, jak tato tvrzení objektivně doložit nebo vyvrátit. Jinak řečeno, chybí měřítka a veličiny charakterizující produkty a fáze tvorby programového vybavení. Vymezení těchto měřítek a veličin úzce souvisí s dosaženým stupněm formalizace procesu ve všech jeho stádiích. Tento stupeň je dán úrovní jednak možné a jednak využité abstrakce při hledání obecných postupů na základě jednotlivých řešení. Je zajímavé, jak oblast tvorby software, jejíž produkty jsou z určitých hledisek vysoce formalizované, sama formalizaci odolává.

Příčin je více. Především, variabilita a složitost problémů, pro jejichž řešení je software vyvíjen, je obrovská. Může jít prakticky o jakékoli problémy z reálného světa. Nepřekvapuje tedy, že hledání obecných prvků v jednotlivých řešeních je velmi obtížné, stejně jako hledání standardních postupů.

Dále vývoj software je proces, který je ovlivňován velkým počtem faktorů /požadavků, omezení, atd./. Analýze jejich vlivů a vzájemných interakcí není většinou věnována potřebná pozornost, takže nebylo dosaženo výsledků, které by faktory kvalitativně i kvantitativně hodnotily tak, aby je bylo možno účinně zohlednit při formalizaci procesu tvorby softwaru. Za těchto okolností je obtížné udělat pouhý výčet faktorů a jejich uspořádání podle důležitosti i pro specielní úlohu a je to proto z pohodlnosti nebo dokonce záměrně přehlíženo a vynechávéno. Z toho pak vyplývá minimum zkušeností s hodnocením skutečného vlivu faktorů a tedy

nedostatek podkladů pro abstrakce.

Další důležitou příčinou, proč proces tvorby software je málo formalizován je to, že lidská mysl dokáže obdivuhodně pracovat s vágními pojmy, takže je schopna za některých okolností dospět k hledanému řešení, aniž by užívala explicitně vyjádřených a formalizovaných postupů ve všech fázích procesu. Bez vágního myšlení a intuice se asi obejít nelze, zvláště při řešení dosud nezvládnutých problémů. Rozhodně si nemyslíme, že formalizace může odstranit všechny těžkosti při návrhu software a že by tedy měla být jediným cílem rozvoje metod. Nicméně bez pokusů o formalizaci a zobecňování je téměř nemožné získaná řešení a postupy sdělovat a uchovávat jinde než v hlavě autora.

Při formalizaci tvorby software bylo už dosaženo mnohých dílčích úspěchů. Většina autorů se v podstatných rysech shoduje na základním schématu procesu tvorby software, jeho členění do jednotlivých fází a jejich vzájemné souvislosti. Pro některé fáze existují i poměrně široce užívané formalizované metody, jako např. strukturované programování, normalizované programování, návrh programu na základě zpracovávaných datových struktur, atd., i když většina z nich je specializována jen na určité okruhy problémů. Průkazným výsledkem formalizace je rozšíření vyšších programovacích jazyků a takových prostředků jako jsou generátory a parametrické programy.

V souvislosti s tvorbou software se uvádějí i některé kvantitativní údaje. Nejčastěji se používá hodnocení pracnosti v "člověkodnech", "člověkoměsících" atd., případně velikosti produktu v počtu výroků zdrojového jazyka, které však nemají k vlastnímu procesu tvorby software, jeho fázím a hodnocení produktu vztah téměř žádný.

Existují i práce, které se pokoušejí o podrobnější a solidnější kvantitativní hodnocení některých fází např. [4], [5], ale tyto výsledky nejsou šířeji využívány, patrně i proto, že není dosud vyjasněn kontext využití.

Z charakteru procesu tvorby software a zejména variability a složitosti řešených problémů s nejvyšší pravděpodobností vyplývá, že univerzální prostředek, postup nebo metoda, zkrátka "kámen mudrců" asi neexistuje. Je tedy zřejmě namístě s krajní opatrností posuzovat propagaci metod a postupů, která používá charakteristik jako převratné, zcela obecné, univerzální a na cokoliv a vždy použitelné. Je třeba pečlivě vážit vlastnosti metody a používat ji v případech, jejichž charakteru odpovídá při vědomí cílů, kterých má být dosaženo. V opačném případě, pokud lze metodou vůbec dospět k řešení, může být její efekt problematický.

Za nejužitečnější v této situaci považujeme jednotící přístupy k celé tvorbě software, které vycházejí z jednotlivých fází procesu. Musí umožňovat výběr konkrétních alternativních postupů pro tyto fáze s ohledem na charakter řešeného problému, respektovat a hodnotit působení ovlivňujících faktorů /požadavků, omezení, atd./ pro různé varianty řešení.

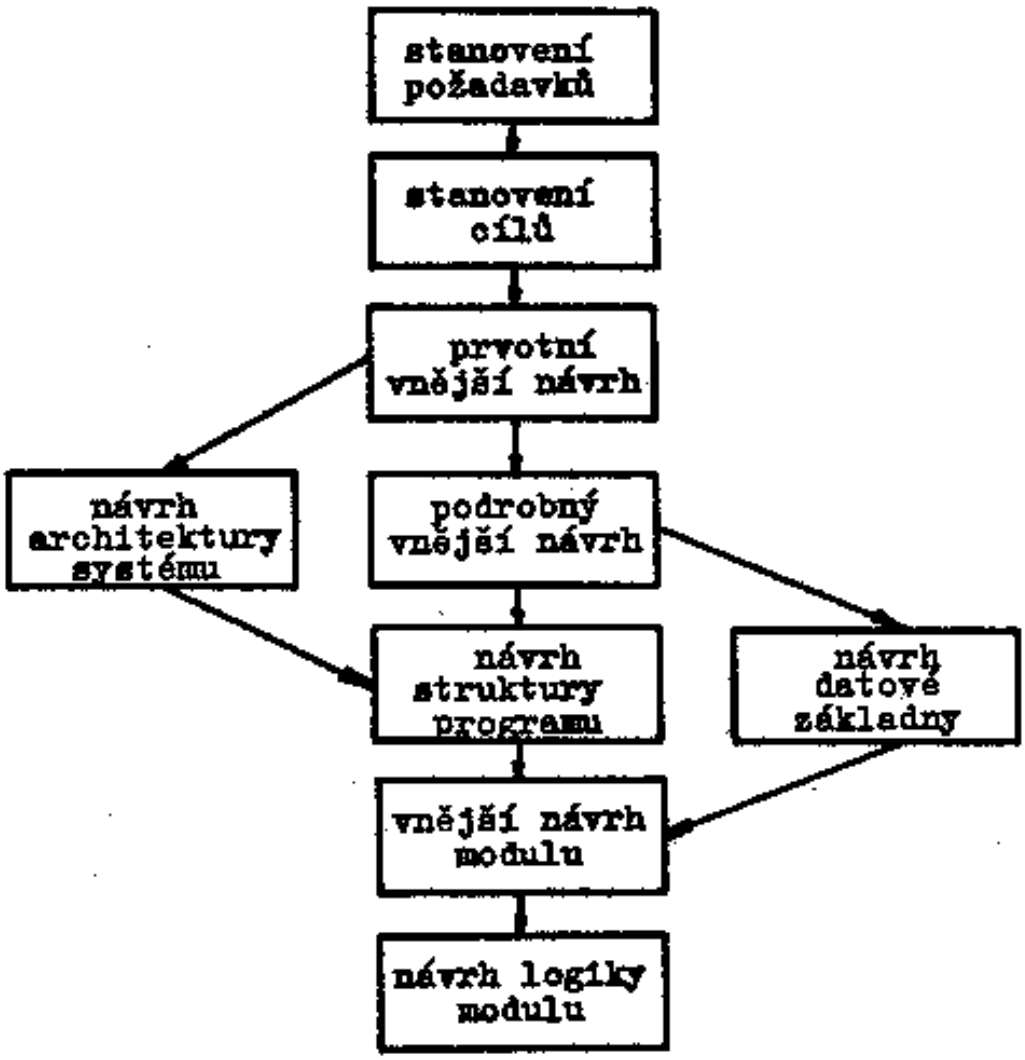
O takový přístup podle našeho názoru usiluje MYERS [1], [2], [3]. Některé myšlenky a postupy uvedené v těchto pracích považujeme za velmi přínosné a užitečné, a proto se jimi budeme zabývat v dalších odstavcích. Hlavním předmětem tohoto příspěvku je návrh modulárních programů. Klíčovým aspektem návrhu je dekompozice programů na jednotlivé moduly. Pro zdůraznění potřebných souvislostí uvedeme nejprve začlenění návrhu programů do celého procesu tvorby software a ideová východiska modularity.

2. PROCES TVORBY SOFTWARE

2.1 Členění procesu

Charakter činností, které se provádí v rámci procesu tvorby software, dělí celý proces do několika stádií - fází. Pro velké projekty uvádí [3] model procesu - viz obr. 1.

Je zřejmé, že model nezávisí na metodách použitých v jednotlivých fázích, ani se nemění s řešenými problémy a ani není podstatné, jaké je dělení mezi profesemi /analýza, programování/ v daném okamžiku. Vyjadřuje obecnou strukturu



Obr. 1. Model procesu tvorby software.

v procesu tvorby software a je základním krokem k formalizaci.

Ve fázi stanovení požadavků uživatel specifikuje, co očekává od výsledného produktu. Ve fázi stanovení cílů se určí přesně cíle a charakteristiky jednak produktu z hlediska uživatele a jednak vlastního projektu z hlediska jeho řízení a realizace. Fáze prvotní vnější návrh stanoví základní rysy interface uživatele k navrhovanému systému, aniž by se specifikovaly detaily. Dále se proces větví na paralelní činnosti, které probíhají současně a navzájem se ovlivňují. Interface uživatele k systému je propracován do co nejpřesnějších detailů /data, způsoby ovládání, atd./ ve fázi podrobný vnější návrh. Současně je navrhována architektura systému, tj. jeho členění na části - podsystémy, programy a definice interface

mezi těmito částmi. Po těchto fázích následuje návrh struktury programů systému s cílem definovat jednotlivé moduly, tj. jejich funkce, vzájemné vazby a obsahově interface. Pro každý modul jsou pak ve fázi vnější návrh modulů přesně definovány funkce, interface a další charakteristiky, jak se jeví z hlediska okolí modulu. Po tomto kroku jsou pak již ve fázi návrh logiky modulů navrhována řešení algoritmů a procedur jednotlivých modulů a posléze jejich kódování ve vybraném programovacím jazyku. Kromě toho paralelně s návrhem struktury programů probíhá návrh datové základny systému, kdy jsou specifikovány všechny soubory dat, báze dat atd. externí programům systému.

Pak následuje ladění, testování, uvedení do provozu, provoz a údržba, což se už přímo netýká navrhování software, a proto se jimi nebudeme zabývat.

Uvedený model je orientován na modulární programování, jak vyplývá z jeho posledních fází, což však není na újmu obecnosti. Jednotlivé fáze, jak jsou specifikovány, nemusí být v projektech menšího rozsahu explicitně přítomny a jsou sloučeny s jinými /např. prvotní a podrobný vnější návrh/, některé mohou chybět vůbec /architektura systému, jde-li o 1 program nebo návrh datové základny/. V zásadě je ovšem u každého projektu třeba definovat odpovídající model, jeho fáze a při vývoji je důsledně dodržovat i s příslušnými kontrolami. Sousední fáze se mohou sice částečně překrývat, i-tá fáze by však nikdy neměla být započata, dokud není dokončena fáze i-2. Lze si jen těžko představit, že by mělo smysl např. navrhovat strukturu programů a přitom nebyl znám interface uživatele k systému.

Při realizaci procesu může samozřejmě dojít k situacím, kdy se v některé fázi zjistí, že dosud rozvíjené řešení nesplňuje požadavky nebo nelze požadavky tímto způsobem buď efektivně nebo vůbec realizovat. Je pak nutno se vrátit k fázím předchozím a zpracovat novou variantu řešení. Dá se říci, že se to prakticky vždy vyplácí, zvláště očekáváme-li dlouhou dobu života systému nebo časté možnosti změn.

Z tohoto pohledu vyniká i důležitost fáze návrh struktury programu. Uvědomíme-li si totiž objemy lidské práce vložené do jednotlivých fází a zejména pak do ladění, testování, údržby atd., kde přibývají i náklady na strojový čas, je zjevné, že po tuto fázi /případně po vnější návrh modulů/ je objem prací ještě poměrně malý, a strojový čas dosud žádný. V této fázi je pak i poslední možnost provést strategické úvahy a ovlivnit pak vše, co po ní následuje. Chceme se v příspěvku proto zabývat právě návrhem struktury programů a poskytnout určitá vodítka a návody pro jeho realizaci.

2.2 Fáze stanovení cílů

Nemělo by však smysl zabývat se návrhem, kdybychom si nebyli vědomi požadavků na funkce a vlastnosti výsledného produktu, kdybychom nerespektovali ovlivňující faktory. Přesná specifikace těchto požadavků a omezení se provádí při stanovení cílů. Zásadní význam této fáze bývá často nedoceněn. Nejprve se zde určují cíle produktu z hlediska uživatele, z nich se pak odvodí cíle samotného projektu. Kromě ryze funkčních cílů se zde vymezují i nároky na další vlastnosti produktu a projektu. Uvedeme nejzávažnější.

Spolehlivostí se rozumí pravděpodobnost, že produkt bude splňovat žádané funkce v daném časovém údobí bez selhání. Za selhání je považován výskyt chyby, které způsobí, že produkt neprovádí to, co uživatel očekává. Při dnešní složitosti systémů, nárocích na jejich funkce a možné důsledky selhání je spolehlivost bezpochyby nejdůležitější vlastností produktu.

Adaptabilita charakterizuje snadnost, s jakou lze rozšiřovat a měnit funkce stávajícího systému.

Udržovatelnost charakterizuje náročnost a obtížnost detekce a odstranění chyb zjištěných při provozu. Dosud uvedené vlastnosti spolu úzce souvisejí a působí stejným směrem.

Obecnost řešení charakterizuje množství, mocnost, rozsah

a dosah uživatelem žádaných funkcí. Její zvyšování má zjevně za následek zvětšování složitosti systému, což zvětšuje mj. potencionální možnosti vzniku chyb. Působí proti již uvedeným vlastnostem.

Lidské faktory vystihují snadnost porozumění funkcím systému, snadnost jeho ovládání - tedy míru uživatelského komfortu a zabránění nesprávným manipulacím. I když zvyšování uživatelského komfortu opět implikuje zvýšení složitosti systému, další úroveň řízení interface uživatele k systému snižuje možnost nesprávných manipulací a zvyšuje na druhé straně spolehlivost.

Bezpečnost má obdobný charakter. Zamezení nepovolených přístupů k datům a ochrana před jejich zničením sice zvýší složitost systému, ale významně zvýší i jeho spolehlivost.

Úroveň dokumentace podmiňuje úroveň práce uživatele se systémem, možnosti jeho údržby a rozšiřování a nepochybně zvyšuje spolehlivost. Na druhé straně příliš složitá, obsáhlá a neúčelně redundantní dokumentace je sama o sobě složitým systémem, který je obtížné aktualizovat. Může tak být příčinou nesprávného ovládání, obtížného rozšiřování a prakticky nemožné údržby systému.

Náklady na projekt zahrnují náklady jak na vývoj produktu, tak na jeho údržbu. Vysoká spolehlivost redukuje náklady na údržbu. Je však podmíněna větším úsilím i náklady při vývoji.

Plán projekce musí být sestaven až s ohledem na specifikované vlastnosti a zohlednit důležitost, která je jim přikládána.

Výkonnost celého systému je ve velmi složitých vztazích k ostatním vlastnostem, např. zvýšení adaptability modulárním programováním má za následek dodatečné nároky na SČ dané dodržením hierarchické struktury, což ovšem zase zvyšuje spolehlivost. Triky sice obvykle zrychlují programy, ale znesnadňují údržbu, atd. Je třeba si také uvědomit, že výkonný není systém, který je sice rychlý, ale jako celek pracuje s chybami. Přitom rychlost odpovědi systému nebo jeho

průchodnost jsou důležitými požadavky uživatele.

Okolí systému a vztahy k němu charakterizují jak hardwarové možnosti pro realizaci systému, tak i jeho organizační a provozní začlenění /jak často bude využíván, s jakými objemy dat, kdo systém ovládá, vazby k jiným systémům, atd./. Tato omezení mohou značně ovlivnit ostatní vlastnosti.

Kompatibilita charakterizuje úroveň návaznosti systému na dosavadní verze, případně nutnost a formy vazby k jiným systémům.

Souhrn některých z uvedených vlastností se dá charakterizovat jako odolnost systému vůči vlivům okolí, a to jak provozního, tak uživatelského, analytického i programátorského. Odolnost systému např. z hlediska změn a údržby je chápána tak, že vůči těmto vlivům je odolná základní struktura systému a programů, takže tyto požadavky nezpůsobí jeho zhroucení. Odolnost má blízký vztah ke spolehlivosti.

Každá z těchto vlastností je charakterizována škálou úrovní obvykle kvalitativních nebo semikvantitativních. Jak z uvedeného stručného přehledu vyplývá, působí jednotlivé vlastnosti přímo nebo nepřímo proti sobě a tedy charakter výsledného produktu je nutně dán kompromisem mezi nimi. Aby tento charakter a tedy i kompromis, ke kterému se dojde při řešení, splňoval požadavky a nebyl libovolný, je třeba již na počátku celého procesu stanovit cíle produktu i projektu jako souhrn priorit a úrovní jednotlivých vlastností a vymezit tak rámec celému procesu. Je totiž nutno si uvědomit, že tento rámec - cíle se musí důsledně uplatňovat a musí být dodrženy při každém rozhodování o variantách i při řešeních detailních problémů návrhu. Tento rámec musí tedy být znám každému pracovníku, který se podílí na návrhu software, což nejlépe zajistí explicitní popis cílů /priorit a úrovní/ písemnou formou. Pro každou fázi se zdůrazní ty cíle, které se jí bezprostředně dotýkají. Rozhodování pracovníka pak není náhodné nebo závislé jen na jeho osobním názoru a zkušenostech, či dokonce líbůstkách, ale podložené znalostí celkového záměru, jednotlivých vlivů, vlastností a vztahů mezi

nimi tedy zdůvodnitelné a kvalifikované.

Dá se tedy bez nadsázky říci, že základní pracovní metodou při tvorbě software je "metoda kvalifikovaného kompromisu". Tento princip se totiž netýká jen diskutovaných cílů, ale i volby variant řešení s ohledem na jiné zde neuvedené vlastnosti. Uplatňuje se i při volbě metod pro realizaci jednotlivých fází. Pracovník by měl vždy uvažovat o žádoucích i nežádoucích vlivech každého rozhodnutí, každé volby varianty a pak kvalifikovaně a zodpovědně s ohledem na celkové i dílčí cíle rozhodnout. Musí si být vědom i důsledků takových rozhodnutí.

Jak ukážeme dále Myers [1], [2] , [7] soustavně uplatňuje tento princip při návrhu modulárních programů.

3. IDEOVÁ VÝCHODISKA MODULARITY

Hlavní příčinou nespolehlivosti softwarových produktů je složitost jednak řešených problémů, jednak vlastního řešení, tj. produktu. Složitost objektu je chápána jako míra mentálního úsilí nutného k pochopení objektu 3 . Platí rovněž, že složitost objektu je funkcí vztahů mezi prvky systému.

Chápeme-li softwarový produkt jako systém, můžeme použít pro redukci složitosti při analýze i syntéze poznatků z obecné teorie systémů. Ta uvádí základní východisko pro redukci složitosti - chování systému je sledováno na základě chování jeho prvků a jejich vazeb, tzn. systém je rozčleněn na části a jsou předně definovány jejich vazby. Při členění se uplatňují tyto základní principy:

- Členění se provádí postupně v podrobnějších a podrobnějších rozlišovacích úrovních. V rámci úrovně je pak systém /podsystém, prvek/ a jeho chování mentálně snadněji zvládnutelné. Je to princip hierarchické struktury. Chování v rámci úrovně lze popsat chováním prvků nižší úrovně a jejich vazbami.

- Člení se na části /prvky/, které jsou co možná nejméně závislé na zbytku systému. Je to princip nezávislosti. Umožňuje pak zkoumat relativně nezávisle jednotlivé části,s tím, že jsou přesně a zjevně definovány vazby k okolí části /prvku/. Vazba musí být vždy přítomna, poněvadž jinak by prvek k systému nepatřil.

Nezávislost implikuje z druhé strany jednotící princip v každé části, členění tedy nemůže být náhodné a řídí se charakterem prvků v dané části. Znamená to pak, že i celá struktura systému je dána charakterem jednotlivých částí a není tedy náhodná. Má-li být tedy realizována syntéza systému, je znám popis jeho chování a vztahy k okolí, a řešení existuje, existuje duálně k popisu i taková struktura systému, která objektivně nejlépe splňuje požadavky a realizuje žádané chování za daných podmínek. Při řešení úlohy syntézy je proto klíčovým úkolem hledání, odkrytí této ideální struktury. Je proto nutné používat metod, které vycházejí z principu nezávislosti. Přístup může pak být různý, např. v informačních systémech se realizuje analýza i syntéza ze dvou základních hledisek: hledisko funkcí a hledisko informací. Jde o duální záměnu významu prvků a vazeb.

Vrátíme-li se tedy zpět k představě softwarového produktu jako systému a programu jako systému, můžeme tyto principy aplikovat i zde, pokud je to potřeba. Ta je zjevná, uvědomíme-li si,že monolitický program pro problémy jen trochu většího rozsahu může obsahovat desítky až stovky proměnných a až tisíce příkazů zdrojového jazyka. Algoritmus je nepřehledně a bez hierarchického uspořádání rozdroben do zápisu, takže sledovat potřebné souvislosti a vůbec mentálně zvládnout celý program je prakticky nemožné.

Je tedy nutno rozčlenit program na části a to ne náhodným způsobem, ale s ohledem na objektivní existenci nejvhodnější struktury řešení. Můžeme se jí přibližovat ze dvou uvedených aspektů - struktury funkcí realizovaných programem nebo struktury dat jím zpracovávaných. Při členění je třeba aplikovat jako klíčový princip nezávislosti a dále principy hierarchie a rozlišovacích úrovní. Přitom za nejvyšší se považuje úroveň

zadání, nejnižší úroveň je úroveň prostředku, který je již schopen přímo zajistit zpracování počítačem /nejčastěji programovací jazyk/. Mezi nimi jsou pak podle potřeby rozlišeny úrovně další. V každé z nich by se měly vyskytovat a řešit problémy stejného druhu z nějakého hlediska /způsob zpracování, celkového řešeného problému/. Řešení úrovně pak využívá určitých funkcí, které jsou buď k dispozici v již existujícím software /příp. hardware/ nebo jsou v procesu návrhu ad hoc vytvářeny. Souhrnu těchto funkcí se často říká virtuální počítač [9]. Používá se rovněž pojmu abstraktní počítač. Dá se říci, že se použitý programovací jazyk rozšiřuje o nově "vestavěné" funkce tak, že vzniká jazyk specializovaný pro řešení problémů úrovně. Pro rozlišovací úrovně se uvádí i pojem úrovně abstrakce. Členěním programu vznikají i rozhraní mezi částmi. Přes tato rozhraní realizuje část styk - vazbu s okolím a obráceně. Vzniká tak interface částí vůči zbytku programu nebo přesněji vůči jiným částem. Je proto vhodné věnovat značnou pozornost přesné specifikaci tohoto interface.

Tyto principy a koncept relativně nezávislých částí využívá jako základní východisko a jako metodu realizace modulární programování a koncept modulů. Přitom lze vidět i další sekundární, ale významné efekty jako např. možnost využití modulu i v jiných kontextech, programech, možnost nezávislého zpracování modulů, atd.

Uvedené principy neznamenají žádný nový přístup, jsou už desítky let součástí teorie systémů, byly používány ještě dříve v jiných vědách a jsou již dlouho známy pod méně přesným názvem "zdravý rozum". Mezi prvními je do programování důsledně uvedli Dijkstra, Wirth a další, odrazem je i top-down technika a zejména již zmíněné modulární programování.

4. ZÁKLADNÍ POJMY KONCEPCE SOUHRNÉHO NÁVRHU

Termín modul je užíván v různých souvislostech, a proto není jeho význam vždy jednoznačný. V této práci budeme vycházet z definice uvedené v [1], [2].

Modul je základní jednotka programové struktury. Tvoří jej skupina výkonných příkazů programu, která splňuje současně tato kritéria:

- je to uzavřená subrutina, tzn. výroky k sobě lexikálně patří, lze určit počátek a konec, může vlastnit lokální data, styk s globálními daty se děje pouze přes definovaný interface,
- může na ni být proveden odkaz z kteréhokoli jiného modulu programu,
- může být samostatně kompilován.

Modul je charakterizován funkcí, logikou, interface a kontextem.

Funkce modulu definuje to, co modul dělá. Slovní definice funkce modulu má mít tvar sloveso / předmět, tzn. co modul dělá a s čím. Např.: invertuj matici, setřiď seznam, získej další vstupní transakci atd. Je třeba se vyvarovat příliš obecných sloves jako zpracuj, udržuj, řiď, aj., které nevypovídají nic o charakteru funkce.

Logika modulu popisuje, jak modul realizuje svou funkci, tzn. popisuje interní algoritmus zpracování.

Interface modulu popisuje přesně vstupní i výstupní informace modulu co do významu i tvaru. Ve fázi návrhu je důležitá významová stránka interface.

Kontext modulu charakterizuje prostředí, ve kterém je modul v daném případě použit. Čím má modul méně vazeb ke kontextu použití, tím je nezávislejší a obecněji použitelný.

Předává-li modul A řízení modulu B /volá-li jej/, je modul A volající, modul B volaný. Předpokládá se, že volaný modul vrací řízení volajícímu, který pokračuje od příkazu

bezprostředně následujícího za příkazem volání. Modul C je podřízený modulu A, volá-li jej A nebo jiný podřízený modulu A. Podobně A je podřízený modulu C. Volá-li A modul B a B modul C, je A bezprostředně nadřízený B a C bezprostředně podřízený B.

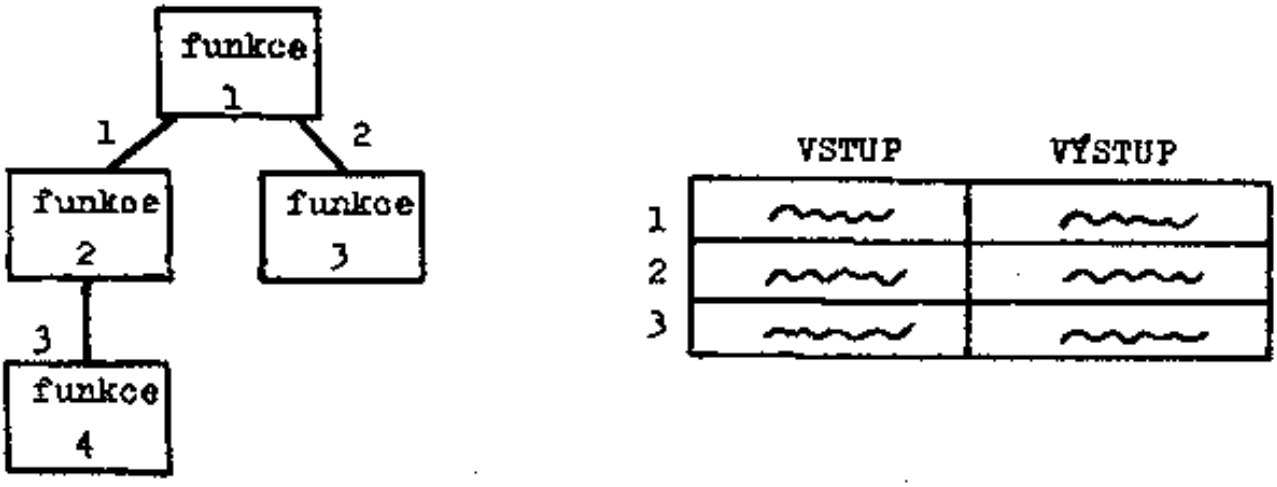
Program je tvořen hierarchickou strukturou modulů. Z principu nezávislosti uvedeného v kap. 3 pak vyplývá, že kvalita a odolnost programu je závislá na "kompaktnosti" jednotlivých modulů a na úrovni vazeb mezi moduly. K těmto charakteritikám je tedy nutné při návrhu programu přihlížet. Vychází z nich i Myersova koncepce modularity a souhrného návrhu struktury programu /composite structured designe/ [1],[2],[3],[7].

Jako klíčový nástroj pro posuzování struktury programu jsou v této koncepci navrhovány soudržnost modulu /module strength/ a spřaženost modulů /module coupling/.

Soudržnost modulu je vlastností jednoho modulu a je měřítkem toho, jak "silně" k sobě jednotlivé prvky modulu /t. j. příkazy nebo jejich sekvence/ patří. Spřaženost modulů se týká relace dvou modulů a lze ji chápat jako stupeň vzájemné vazby a ovlivňování modulů. Nejlepší modulární návrh programu je pak takový, ve kterém soudržnost jeho modulů je co nejvyšší a stupeň vzájemné vazby - spřaženost - co nejmenší.

Koncepce souhrného návrhu dále obsahuje techniky a prostředky pro takový návrh struktury programů, který přihlíží k žádoucím i nežádoucím důsledkům soudržnosti a spřaženosti modulů. Obě části návrhu jsou podrobněji probrány v následujících dvou kapitolách.

Důležitou součástí koncepce jsou strukturní diagramy, které zobrazují hierarchickou strukturu programu a uvádějí i popis jednotlivých interface. Používá se grafických prvků [2] - obr. 2.



Obr. 2. Základní grafické prvky

V levé části obrázku je uveden vlastní diagram struktury. Obdélníky označují jednotlivé moduly, uvnitř je uvedena jejich funkce. Spojení mezi moduly jsou jednotlivé interface, které jsou číslovány pro účely popisu interface /tabulka v pravé části/. Pro každý interface je uveden zvlášť výčet vstupních informací a zvlášť výčet výstupních. Pokud je některá vstupní i výstupní, je uvedena v obou částech tabulky. Při kreslení diagramu je vhodné kreslit podřízený modul pod úrovní nadřízeného, bezprostředně podřízené moduly kreslit zleva doprava podle pořadí provádění. Na závadu nejsou i vazby mezi moduly nesousedících úrovní.

V [2] se používají kromě těchto základních i další grafické i další symboly, které však pro účely výkladu nejsou podstatné, případně může každý používat vlastních.

5. MĚŘÍTKA MODULARITY

5. 1. Soudržnost modulu

Soudržnost modulu navrhuje Myers [1], [2], [3] klasifikovat především podle těchto hledisek:

- zda modul provádí jednu nebo více funkcí,
- zda je při jednom volání modulu provedena jedna nebo více funkcí,
- zda má každá funkce modulu vlastní ENTRY,
- zda je mezi funkcemi modulu vztah a jaký.

Tato hlediska vymezují jednotlivé typy soudržnosti, které lze podle důsledků na nezávislost modulů, snadnost jejich údržby, adaptabilitu a možnost opakovaného použití seřadit do následující vzestupné škály soudržnosti:

1 - nahodilá soudržnost /coincidental strength/
2 - logická soudržnost
3 - klasická soudržnost
4 - procedurální soudržnost
5 - komunikační soudržnost
6 - funkční nebo informační soudržnost

Tato škála není lineární, stupeň 6 je značně silnější než kterýkoliv předchozí, stupeň 1 - 3 značně slabší než následující.

Skutečné moduly nemusí být charakterizovány jedním typem soudržnosti, rozhodující pro klasifikaci jsou podíly zastoupení a úrovně přítomných typů.

5. 1. 1 Nahodilá soudržnost

Modul má nahodilou soudržnost, jestliže provádí více funkcí, na jedno volání je provede všechny a není mezi nimi žádný zjevný vztah. "Funkci" takového modulu je prakticky nemožné slovně vyjádřit.

Nahodilá soudržnost vzniká v případech, kdy byl modul vytvořen dodatečnou "modularizací" existujícího programu /např. při nutnosti segmentace programu/, vylučováním duplicitního kódu z jiných modulů, na základě nevhodného standardu velikosti modulu /striktně zadán max. dovolený počet příkazů/. Ve všech případech jde o hrubé přehlížení charakteru funkcí a problému.

Funkce mají vztahy spíše ke zbytku programu než mezi sebou, soudržnost je malá, změny v modulu jsou velmi obtížné a mají důsledky v celém programu.

Použití modulu s nahodilou soudržností v jiném programu není možné.

5. 1. 2 Logická soudržnost

Modul má logickou soudržnost, jestliže provádí více funkcí, na jedno volání provede podle předaného kódu jen jednu z nich, vstupní bod modulu je jen jeden /funkce tedy nemají vlastní ENTRY/, mezi funkcemi je logický vztah, takže při hrubějším rozlišení je lze označit jako funkci jednu /s předpokladem, že pro různá volání jsou její činnosti rozdílné/.

Tuto situaci je možno ukázat na příkledu modulu s funkcí "aktualizuj kmenový soubor". Takový modul má však zabezpečovat funkce čtyři: kopíruj větu kmenového souboru, aktualizuj větu kmenového souboru, vynech větu kmenového souboru, vlož novou větu. Takový modul může obsahovat sekvence příkazů, které budou shodné pro více funkcí. Je velmi pravděpodobné, že programátor se dá zlákat k vícenásobnému využití těchto sekvencí, ovšem za cenu vzájemného svázání funkcí, případně k využití nějakého triku. Pak však změna v jedné funkci bude způsobovat obtíže, neboť ovlivní i ostatní funkce modulu.

Další podstatnou nevýhodou je, že interface modulu je společný pro všechny funkce, tzn. že stejných parametrů je používáno v různých kontextech a významech jak vně, tak uvnitř modulu. Poněvadž každá funkce nemusí vyžadovat stejný počet parametrů, vyskytují se v interface zbytečné parametry. Při přidání nové funkce s jiným větším počtem parametrů je pak třeba provést změnu interface, a to i ve všech modulech, které tento modul využívají.

Soudržnost modulu je malá vzhledem k značné samostatnosti jeho jednotlivých funkcí.

5. 1. 3 Klasická soudržnost

Modul má klasickou soudržnost, jestliže provádí více funkcí, na jedno volání je provede všechny a funkce jsou vázány procedurou zpracování /je nutno je provést v jednom časovém okamžiku práce programu/.

Příkladem této situace jsou běžné moduly pro inicializaci, ukončení, nulování ap. Moduly klasické soudržnosti vyvolávají zvýšení stupně vazby s ostatními moduly programu a jsou většinou nepoužitelné mimo kontex programu.

Někdy se modulům klasické soudržnosti nelze vyhnout, na př. modul s funkcemi "po výjimce diagnózuj chybu; jestliže je to možné, oprav chybu; pokračuj ve výpočtu" má klasickou soudržnost, kterou lze jen velmi těžko obejít.

5. 1. 4 Procedurální soudržnost

Modul má procedurální soudržnost, jestliže provádí více funkcí, na jedno volání je provede všechny a je mezi nimi vztah /pořadí/ daný procedurou řešení problému.

Soudržnost je vyšší než klasická, modul se totiž orientuje na problém. Moduly této soudržnosti často vznikají po použití vývojových diagramů sloučením více operačních bloků do jednoho modulu.

Příkladem procedurální soudržnosti je modul s funkcemi: testuj hodnotu, uprav tabulku v paměti a tiskni zprávu.

Modul má naději na použití i v jiných programech.

5. 1. 5 Komunikační soudržnost

Definice komunikační soudržnosti je shodná s definicí procedurální soudržnosti s tím, že jednotlivé funkce navíc komunikují daty. Znamená to, že se např. odvolávají na tatáž data /místa paměti/, nebo si postupně data předávají.

Příkladem je funkce modulu: vytiskni a vyděruj výstupní soubor; vypočti objem tělesa a tiskni jej.

Soudržnost modulu je vyšší, poněvadž přistupuje navíc vazba daty. Moduly této soudržnosti mohou být s velkou pravděpodobností používány i v jiných programech.

5. 1. 6 Funkční soudržnost

Modul má funkční soudržnost, provádí-li pouze jedinou funkci. Tato funkce může být primitivní, na př. "vypočítej

druhou odmocninu" nebo složité, vyžadující provedení skupiny dílčích funkcí /volání podřízených modulů/, např. "kompiluj zdrojový program v COBOLU". Rozhodující pak je, zda lze funkci skupiny popsat jako jedinou specifickou funkci. Pokud je to možné, modul, který ji realizuje má funkční soudržnost.

Funkcí se běžně rozumí jednoznačná transformace vstupní informace na výstupní, připouští se však i funkce bez vstupní nebo výstupní informace, např. zapiš větu na výstupní soubor, generuj náhodné číslo.

Soudržnost modulů je nejvyšší, nezávislost velmi vysoká, programy s nimi lze snadno udržovat, rozšiřovat a modulů lze využít i v jiných programech.

5. 1. 7 Informační soudržnost

Modul má informační soudržnost, provádí-li více specifických funkcí nad nějakými daty, strukturou dat, souborem dat nebo jiným zdrojem, který je uchováván tímto modulem. Každá funkce má svůj ENTRY /vstupní bod/, takže je vždy volána jen jedna. Jednotlivé funkce si nepředávají řízení.

Modul vzniká sloučením jednotlivých modulů s funkční soudržností, operujících nad týmž zdrojem /který jim byl předáván přes interface/, do jednoho modulu. Každému původnímu modulu odpovídá jeden vstupní bod nového modulu. Zdroj je pak ostatnímu programu neznám a je jím využíván na logické úrovni prostřednictvím neredundantních interface jednotlivých funkcí /tzv. "skrývání informace"/. Na tento druh soudržnosti lze převést i logickou soudržnost zavedením vstupních bodů a zrušením kódu funkce.

Přidání dalších funkcí nebo změny nečiní obtíže, změny v charakteristice zdroje /formát dat, atd./ se projeví jen uvnitř tohoto modulu. Nezávislost modulu je vysoká, lze jej používat i jinde.

5. 1. 7 Kategorizace modulů

Typ soudržnosti lze s určitou pravděpodobností stanovit už ze slovního vyjádření funkce modulu. Pokud věta, která funkci popisuje:

- je složená, obsahuje čárky, více sloves, provádí obvykle modul více funkcí; obsahuje-li dále spojku a, lze očekávat komunikační, procedurální nebo klasickou soudržnost; obsahuje-li nebo lze očekávat logickou soudržnost;
- obsahuje slova napřed, potom, další, když atd., bude mít modul procedurální soudržnost;
- obsahuje inicializuj, nuluj apod., bude mít modul klasickou soudržnost;
- obsahuje přívlastek předmětu, který neurčuje jeden objekt /např. zapiš všechna data/, lze očekávat logickou soudržnost;
- atd.

5. 2. Spřaženost modulů

Spřažeností modulů se rozumí jejich vzájemná vazba a ovlivňování. Myers [1, 2, 7] klasifikuje spřaženost dvou modulů především podle těchto hledisek:

- zda sdílí společné objekty /data, instrukce/, které jsou uvnitř těchto modulů;
- zda sdílí společné objekty ve svém okolí /data, datové struktury/;
- zda tato data mají globální či omezenou platnost;
- zda si předávají informace řídícího charakteru v jednom či obou směrech.

V prvních třech případech se vždy jedná o způsob sdílení určité oblasti paměti /vazba pozicemi vůči pevné adrese/, která je programu k dispozici, v posledním případě jde o porušení principu modulu jako "černé skříňky", tj. lze jej

používat na základě jeho vnější definice bez znalosti jeho vnitřní struktury. Podle úrovně vazby lze sestavit následující vzestupnou škálu soudržnosti [1],[2] :

1 - datová spřaženost /data coupling/
2 - spřaženost předávanou strukturou dat /stamp coupling/
3 - spřaženost řízením /control coupling/
4 - vnější spřaženost /external coupling/
5 - spřaženost sdílením vnější datové struktury /common coupling/
6 - obsahová spřaženost /content coupling/

Na rozdíl od soudržnosti, kdy cílem souhrného návrhu je dosáhnout co nejvyššího stupně soudržnosti jednotlivých modulů, je zde cílem dosáhnout co nejmenší možné vzájemné spřaženosti modulů. Neznamená to, že se vždy musí podařit dosáhnout minimální, tj. datové spřaženosti všech modulů. I při posuzování návrhu je třeba respektovat zdůvodněné kompromisy při návrhu programu.

Podotýkáme, že v programu mohou existovat moduly, které jsou na sobě zcela nezávislé. Takovým dvojicím odpovídá nultý stupeň na škále spřaženosti. Ostatní stupně budou probrány sestupně, neboť tento postup je srozumitelnější i přitažlivější /nejlepší nakonec/.

5. 2. 1 Obsahová spřaženost

Dva moduly jsou obsahově spřaženy, jestliže se jeden modul přímo odkazuje na obsah druhého. Jsou to zejména tyto případy:

- jeden modul modifikuje příkazy /instrukce/ v druhém;
- z jednoho modulu se provádí odskok do druhého modulu na místo, které není vstupním bodem;
- modul se odkazuje na data v druhém modulu, přičemž tato data nejsou deklarována jako externí;
- nemohou-li být moduly kompilovány samostatně /interní procedury, více CSECT v Assembleru s odkazy na společné symboly/.

Tato spřaženost je velmi nebezpečná pro spolehlivost programu. Naštěstí ve většině vyšších programovacích jazyků ji není možno vytvořit.

5. 2. 2 Spřaženost sdílením vnější datové struktury

Skupina modulů /každý modul s každým/ má tento stupeň spřaženosti, jestliže sdílejí společnou datovou strukturu globální platnosti, např. v nepojmenované COMMON oblasti u modulu v jazyku FORTRAN. Sdílení globálně platné struktury působí následující obtíže:

- Je snížena přehlednost programu, nejsou zjevné všechny odkazy na tuto strukturu. Sledování průběhu chodu programu je znesnadněno, což ztěžuje ladění, údržbu programu a identifikaci chyb.
- Zavádí závislosti mezi jinak nezávislé moduly. Změna v jednom modulu, která změní tuto datovou strukturu, vyžaduje změnu a rekompilaci ostatních spřažených modulů.
- Téměř vylučuje užití modulu v jiném kontextu i v případě, že obecností své funkce na toto použití aspiruje.
- Znemožňuje řízení přístupu k datům struktury v rámci programu, což je pro jeho spolehlivost životně důležité. Modul může zasahovat do dat, která mu nepatří, nebo je může nekontrolovaně používat. Pokud jsou proti tomu činěna opatření /dummy jména částí struktury/, je značné nebezpečí nepřesností při jejich realizaci.

Tyto obtíže je možno částečně odstranit převedením na nižší typy spřaženosti.

5. 2. 3 Vnější spřaženost

Dva moduly mají tento stupeň spřaženosti, jestliže se odkazují na tentýž globálně platný jednoduchý datový prvek /skalární proměnné, co do významu homogenní pole - vektor/. Vnější spřaženost je tedy velmi podobná spřaženosti sdílením globální datové struktury, odpadají však obtíže způsobené vazbou na pevné pozice v této struktuře. Vnější spřaženost

mají např. moduly, které spolu komunikují přes pojmenovaný COMMON obsahující jednoduchý datový prvek /ve FORTRANu/.

5. 2. 4 Spřaženost řízením

Spřaženost modulů tohoto stupně vzniká tehdy, jestliže jeden modul předává druhému jako parametry informace pro jeho řízení. Typickými prvky řízení jsou kódy požadovaných funkcí, indikátory a přepínače.

Technika předávání řídících parametrů je dosti rozšířená a nejsou podrobněji analyzovány její důsledky. Předávání řídících parametrů narušuje užitečnou zásadu realizace modulu jako "černé skříňky" charakterizované pouze její funkcí. Často vede k defenzivnímu kódování ve volaném modulu /kontrola obsahové správnosti předávaných řídících parametrů/. Předávání indikátorů /variantních hodnot návratového kódu/ volajícímu modulu je porušením hierarchie. Vede tak k řízení činnosti nadřízeného modulu. Volajícímu modulu by měla být vrácena jen informace, zda podřízený modul realizoval svou funkci či ne.

5. 2. 5 Spřaženost předávanou strukturou dat

Takto spřaženy jsou moduly, které si jako parametr předávají datovou strukturu, jejíž platnost není globální. Spřaženost tohoto typu je odkazy na pevné pozice vůči začátku struktury velmi podobná spřaženosti sdílením vnější datové struktury, ovšem bez nevýhod plynoucích z její globální platnosti. Tím je poněkud vyšší i možnost použití modulu mimo kontext programu. Někdy je nutnost vytvoření takto spřažených modulů vyvolána omezeními délky seznamu předávaných parametrů u některých kompilátorů.

5. 2. 6 Datová spřaženost

Pokud mezi dvěma moduly programu existuje nějaká vazba, která není žádného typu z předchozích odstavců, jsou tyto moduly datově spřaženy. Všechny vstupy a výstupy volaného modulu

jsou pak předávány jako parametry a tvoří je pouze jednoduché datové prvky, které nejsou předávány jako informace pro řízení. Vztah datové spřaženosti a spřaženosti předávanou datovou strukturou je analogický vztahu vnější spřaženosti a spřaženosti sdílením vnější datové struktury. Datové spřaženost je ve škále vzájemné vazby modulů nejnižší, je tedy žádoucí tohoto stupně při návrhu modulárních programů dosahovat. Pokud jsou moduly datově spřažené, je využití volaného modulu mimo kontext programu jednoduché a pravděpodobné.

5. 3. Další hlediska pro hodnocení modularity a návrhu

Kromě soudržnosti a spřaženosti modulů, dvou hlavních veličin charakterizujících modularitu a nezávislost modulů, existují i další faktory, z nichž se dá usuzovat na kvalitu programů. Mnohé z nich byly diskutovány i na předchozích seminářích [6].

Na tomto místě připomínáme požadavek jednoduchosti návrhu programu. Lze tvrdit, že ze dvou návrhů je při ostatních faktorech shodných lepší řešení jednodušší. Někdy je v rozporu s jednoduchostí požadavek na obecnost řešení. Tento požadavek je však často neodůvodněný, neboť v období návrhu "obecného" programu můžeme stěží odhadnout všechny budoucí požadavky a tento odhad se často podobá spíše proroctví [10]. Navíc je zpravidla jednodušší provést změny v dobře navrženém spolehlivém programu ze soudržných a nezávislých modulů než provádět třeba i menší úpravy v programu s aspirací obecnosti, avšak postrádajícím uvedené kvality.

Pro spolehlivost modulárních programů je důležité tzv. předvídatelné chování modulů. Takový modul se stejným vstupem pracuje vždy stejně, bez ohledu na to, kdy, odkud a po kolikáté je volán. Nemá vnitřní statickou paměť. Takový modul pak má i vlastnost "černé skříňky", pokud mu není předán prvek řízení.

Velikost modulů /počet příkazů ve zdrojovém jazyku/ je také užitečným vodítkem při návrhu programu. Příliš velké

moduly /asi 100 a více příkazů ve vyšším programovacím jazyku/ často indikují neúčelné spojení více funkcí v jednom modulu a nízkou soudržnost. Naopak příliš malý modul /méně než 10 příkazů/ nemusí provádět úplnou funkci. Tyto příkazy mohou být většinou začleněny do volajícího modulu bez újmy na jeho soudržnosti.

Důležitou vlastností dobrého návrhu je požadavek na to, aby jeho struktura odpovídala struktuře řešeného problému. Existují i další hlediska hodnocení návrhu programů, např. délky seznamů parametrů předávaných mezi moduly, mnemotechnická výstižnost jmen modulů, přiměřenost popisů funkcí modulů z hlediska jejich obecnosti či specielnosti atd., redundance interface, konzistence interface a funkce modulu, izolace systémově a instalačně závislých funkcí do zvláštních modulů, případně další hlediska. Obecně platná pravidla pro hodnocení podle nich nejsou, je třeba se řídit zkušenostmi a intuicí.

6. NÁVRH STRUKTURY PROGRAMU

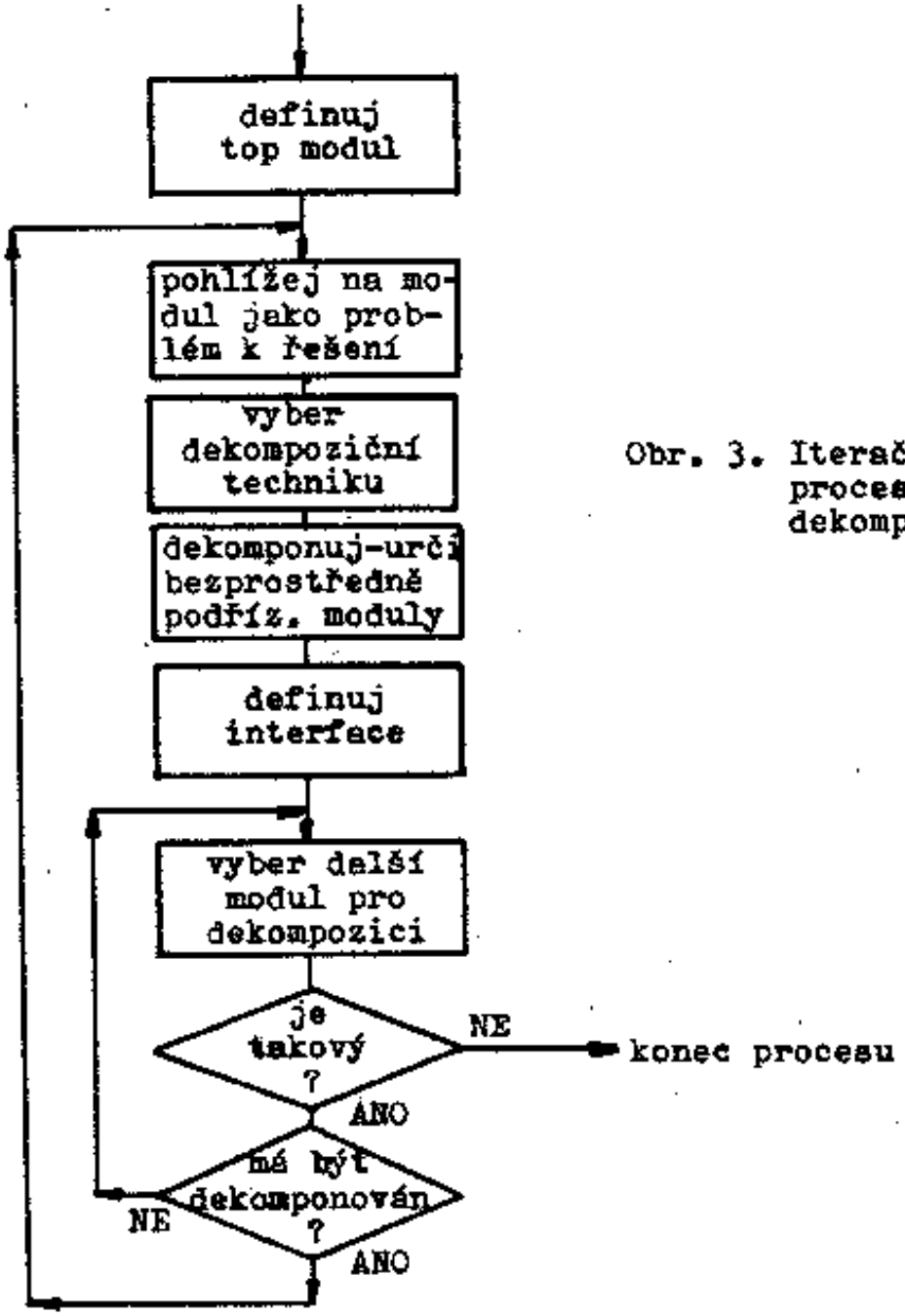
I v této fázi procesu tvorby software bude cílem hledání odůvodněného a kvalifikovaného kompromisu. Připomínáme proto opět kap. 2.2 - důležitost cílů projektu, kap. 3 a zejména kap. 5, která na základě hodnocení soudržnosti modulů , jejich spřaženosti a dalších vodítek pro návrh charakterizuje kvalitu navrženého programu a jeho částí.

6. 1. Proces návrhu struktury programu

Základní schéma tohoto procesu 2 /viz obr. 3/ je navrženo tak, že produkuje hierarchickou strukturu programů. Proces přitom postupuje shora dolů a je iterační.

Začíná definicí "top" /hlavního/ modulu programu, přesněji definicí jeho funkce. Pokud nelze definovat jedinou funkci, nebude mít top modul funkční soudržnost. Je pak vhodné udržet tento nižší typ soudržnosti v top modulu.

Další kroky se provedou nejprve pro top modul a pak



Obr. 3. Iterační proces dekompozice

postupně pro moduly vznikající dekompozicí. V prvním kroku se pohlíží na modul /jeho funkci/ jako na problém, který má být řešen. Podle charakteru problému se vybere vhodná dekompoziční technika /kap.6.2/ a k modulu se touto technikou navrhnou bezprostředně podřízené moduly. Definuje se interface těchto podřízených modulů co do druhu vstupních a výstupních informací bez přesné specifikace formátů apod. Tím je proces dekompozice modulu ukončen. Pokud je třeba dekomponovat další moduly vzniklé dekompozicí, postup se pro ně opakuje. Vznikají tak další moduly nižších úrovní. Při návrhu je třeba sledovat, zda se

nevytvářejí moduly s duplicitní nebo jen málo odlišnou funkcí, kterou by bylo možno upravit na stejnou. Takové duplicity je třeba odstranit. Může být také požadováno využití už existujících modulů z knihoven.

Postupně se tak vytváří modulární hierarchická struktura programu. Proces končí, není-li už další dekompozice modulů potřebná nebo vhodná. Pokud pro ni neexistují zjevné důvody, je toto posouzení obtížné. Nejsou k dispozici exaktní testy, užitečná však mohou být následující vodítka:

- Pokusit se před započetím dekompozice modulu představit si jeho logiku. Pokud je to snadné, bude modul přehledný a není třeba jej dále členit.
- Pokud nelze modul dekomponovat do bezprostředně podřízených modulů s funkční soudržností, není další dekompozice vhodná.
- Pokud nelze z jakýchkoli důvodů rozhodnout, je vhodnější v dekompozici dále pokračovat než ji ukončit příliš brzy. Při posuzování celého návrhu struktury nebo při návrhu s kódování logiky jednotlivých modulů je možno kdykoli se vrátit k vyšším úrovním, provést rekombinaci a zmenšit tak hloubku dekompozice. Snadnější je upravit program, který byl dekomponován příliš, než ten, který byl dekomponován málo.

6. 2. Techniky dekompozice

Techniky dekompozice vycházejí z toho, že funkce modulu je dána funkcemi všech bezprostředně podřízených modulů a jeho logikou. Dekompozice pak spočívá ve vydělení /odvození/ těchto dílčích funkcí /tj.modulů/ z funkce modulu dekomponovaného. Dílčí funkce mohou mít vztah k problému nebo potřebám způsobu zpracování. Podle charakteru, který převažuje, se liší i dekompoziční techniky. Technika funkční dekompozice /viz 6.2.1/ dělí modul do funkcí /modulů/ nižší úrovně zejména podle charakteru problému. Technika dekompozice STS /viz 6.2.2/ dělí modul obecně na tři moduly o specializovaných funkcích - první připravuje data pro ústřední transformaci, druhý ji realizuje, třetí rozptyluje její výstupní data v programu.

Technika transakční dekompozice /viz 6.2.3/ dělí modul do skupiny funkcí /modulů/, z nichž každá realizuje zpracování-transakci určitého typu.

Pro dekompozici kteréhokoli modulu je vybrána technika podle potřeby a charakteru řešení. Tedy použití některé techniky pro nějaký modul neznamená, že musí být používána v celém programu. V [2] se uvádí, že nejprve má být u každého modulu učiněn pokus aplikovat techniku STS, jejíž postup je nejvíce formalizován. Pokud není možno realizovat její první krok, je to známkou, že musí být použito jiné techniky.

Při dekompozici je vhodné uplatňovat i princip rozlišovacích úrovní v hierarchické struktuře - virtuální počítače. Je pravděpodobné, že čím více se při dekompozici celého programu respektuje charakter problému, tím víc se výsledná struktura blíží "objektivně nejlepší struktuře" řešení /viz kap.3/.

6. 2. 1 Technika funkční dekompozice

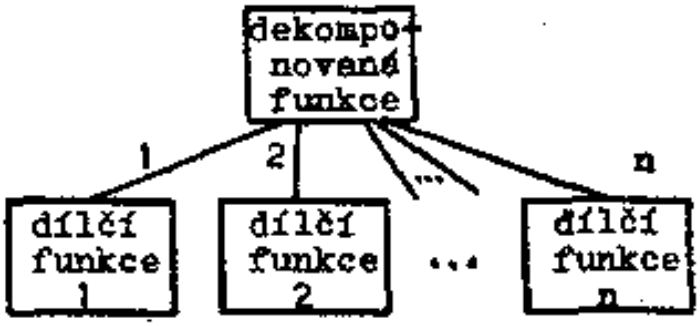
Technika vychází z funkční struktury řešení problému. Modul je dekomponován tak, že jeho funkce je zajišťována dílčími ad hoc vytvářenými funkcemi /tj. bezprostředně podřízenými moduly/, které jsou z ní vydělovány z různých důvodů.

Obvykle jde o provedení určité konkretní funkce /operace, zpracování, transformace/, jako např. invertuj matici, analyzuj řetěz znaků, kontroluj datovou položku atd. Důvodem pro vydělení je zejména struktura řešeného problému, případně i hlediska zpracování. Může jít také o záměr osamostatnit určitou funkci v případě, že:

- je poměrně obecná, počítá se s jejím použitím i jinde;
- se zvýší přehlednost struktury programu z hlediska algoritmu řešeného problému;
- počítá se s jejími různými alternativami, které budou buď zkoušeny nebo pak podle potřeby začleněny do struktury /např. různé metody řešení soustavy rovnic, instalačně závislé funkce OS, atd./;

- vyskytuje se jako společná dílčí funkce skupiny modulů;
- existuje ve struktuře modul s informační soudržností, který obhospodařuje nějaký zdroj, a je potřeba v jiné modulu realizovat dílčí funkci nad tímto zdrojem; tato funkce se pak přesune jako nový ENTRY do modulu obhospodařujícího zdroj;
- atd.

Při procesu návrhu je vhodné uplatňovat principy hierarchie tak, aby podřízené moduly, které vznikají dekompozicí, měly vzájemně si odpovídající úroveň. Proces dekompozice má výrazně tvůrčí a intuitivní charakter, není nijak formalizován a rovněž výsledná struktura nemá žádné charakteristické vlastnosti /obecný tvat je na obr. 4/. Proto nelze poskytnout zřejmě žádná další vodítka. Vzniklé podřízené moduly mají téměř výhradně funkční soudržnost a minimální vztahy k jiným modulům.

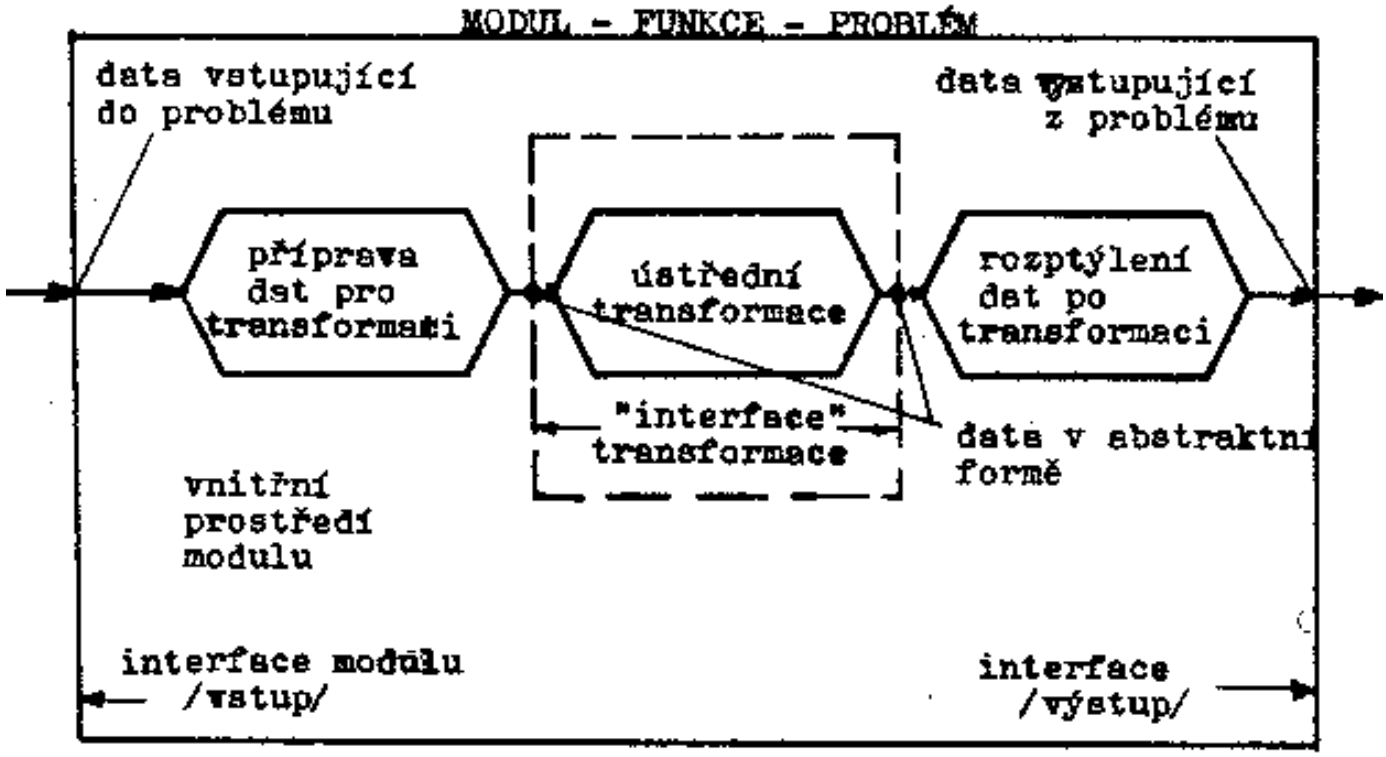


Obr. 4. Obecný tvar struktury po funkční dekompozici.

6. 2. 2 Technika dekompozice STS.

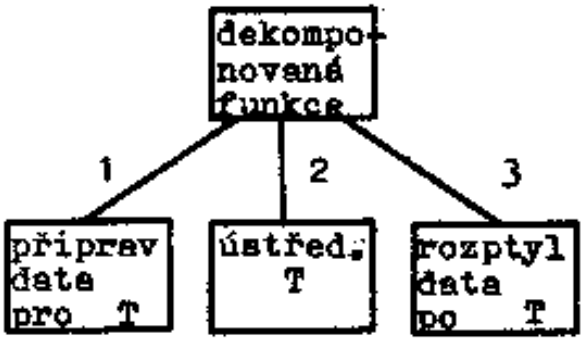
Technika vychází z představy funkce jako transformace dat vstupních na výstupní, formálně vyjádřeno VÝSTUP=T/VSTUP/. Kromě vlastního způsobu transformace /algoritmu/ k její definici patří také definice a přesné specifikace struktur dat vstupních i výstupních, jak je transformace vyžaduje či produkuje. Vstupní i výstupní data tak tvoří logické entity o vysoké úrovni abstrakce. Takto chápaná transformace je zcela nezávislá na kontextu použití a je vytvořena na obecné úrovni.

Má-li modul realizovat nějakou funkci, obsahuje transformaci, která je z hlediska této funkce hlavní - ústřední. Poněvadž se transformace odehrává na obecné úrovni bez ohledu na kontext, je ji třeba do prostředí modulu "zasadit", vytvořit jí podmínky jaké požaduje. Znamená to vybudovat takový "interface" vůči vnitřnímu prostředí modulu, který odpovídá její definici. Problém se pak rozpádá na tři vysoce nezávislé dílčí problémy - přípravu dat pro ústřední transformaci, vlastní transformaci a rozptýlení dat po ní do prostředí modulu /programu/. Odsud je i zřejmý význam toku dat problémem. Data jsou nejprve ve formě odpovídající styku problému s okolím /interface modulu/, tedy ve formě, která nejvíc odpovídá vnějšímu charakteru problému. Tato forma se mění v abstraktní nutnou pro transformaci; po transformaci pak v opačném smyslu opět do formy charakteristické pro problém. Tok dat je takto sledován z hlediska funkce modulu /obr.5/.



Obr.5. Tok dat problémem.

Funkce může být takto principielně dekomponována do tří specializovaných modulů, výsledná struktura je na obr. 6. Dekompozice je označována STS, což je zkratka celého názvu



Obr. 6. Struktura po dekompozici STS.

	VSTUP	VÝSTUP
1	...	data v abst. formě pro transformaci
2	vstupní data v abstr. formě	výstupní dt. v abstr. formě
3	výst. data v abstr. formě	...

... - dáno interface modulu a jeho prostředím

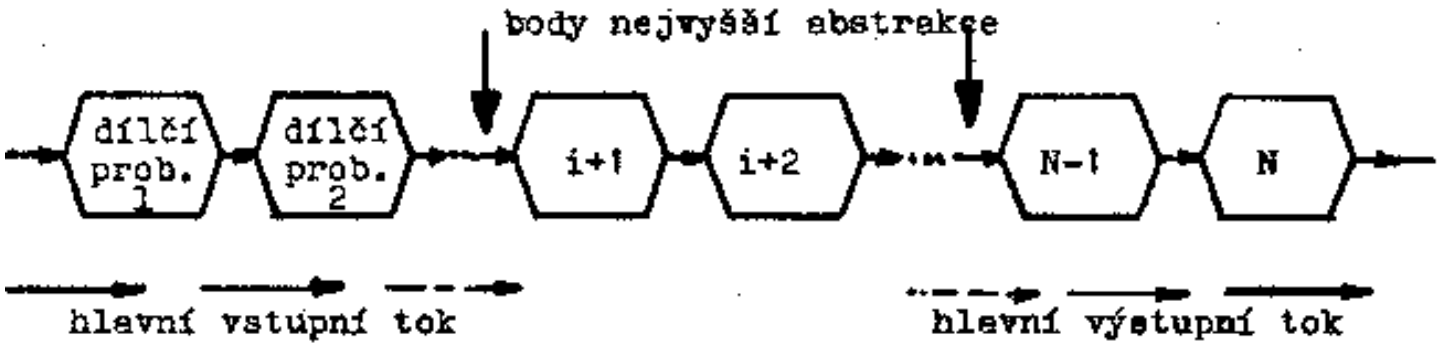
source/transform/sink, který vystihuje charakter metody. Uvedené úvahy implikují i postup při její eplikaci [1,2,3,7] :

1 - Identifikuj a načrtni strukturu problému z hlediska toku dat problémem. Znamená to zkoumat strukturu problému jako řadu dílčích problémů /cca 3-10/. Účelem není popsat proceduru zpracování, ale tok dat problémem, či závislost dílčích problémů na datech. Pokud dílčí problémy označíme 1,2,..,N, musí platit, že problém i je závislý na některých /nebo na všech/ výstupech problémů 1,2,.., i-1 a není závislý na výstupech problémů i+1, i+2,..., N. /obr.7/. Pokud nelze takto popsat strukturu problému - tok dat, je třeba použít jinou techniku dekompozice.

2 - Identifikuj hlavní vstupní a výstupní tok dat v této struktuře. Za tok dat se považuje logický souhrn informací; je nezávislý na fyzické reprezentaci nebo fyzickém V/V zařízení. Z téhož V/V zařízení může vstupovat do problému více toků dat. Nejprve je třeba určit všechny vstupní toky dat do problému. Pokud je jen jeden, je to hledaný hlavní tok. Je-li jich více, je hlavní ten, který je z hlediska funkce a zpracování rozhodující, případně řídící. Podobně se postupuje při hledání hlavního výstupního toku. Oba musí být určeny; označí se ve struktuře problému /obr.7/.

3 - Najdi ve struktuře problému body nejvyšší abstrakce. Jsou

to body, ve kterých hlavní vstupní tok naposledy a hlavní výstupní tok poprvé existují jako logické entity. Body nalezneme procházením struktury problému. Při sledování hlavního vstupního toku se tento stává abstraktnější a abstraktnější z hlediska původního charakteru, až je nalezen bod, kde se dá říci, že za ním už vstupní tok neexistuje. Podobně sledujeme hlavní výstupní tok v opačném směru, až určíme bod, před kterým už hlavní výstupní tok neexistuje. To jsou pak hledané body nejvyšší abstrakce /obr.7, schematicky/.



Obr. 7. Schematické znázornění struktury problému- toku dat, hlavní toky, body nejvyšší abstrakce.

4 - Využij těchto bodů k dekompozici problému a definuj bezprostředně podřízené moduly. Body dělí problém do tří nejméně závislých [2] dílčích problémů. Popíšeme je jako samostatné jediné funkce. Budou to funkce tří bezprostředně podřízených modulů /vstupního, transformačního, výstupního/. Pokud by nebylo možno popsat dílčí problémy jako jedinou funkci, byly zřejmě špatně určeny body abstrakce, nebo se technika nehodí pro daný problém, případně vyžaduje dílčí modifikace. Takto je dekompozice modulu ukončena.

Modifikace techniky

Tato technika není dogmatem a je možno ji podle potřeby přizpůsobovat při zachování hlavních principů. V některých případech není třeba možno určit z více vstupních toků dat hlavní, poněvadž třeba dva mají z hlediska problému stejný

význam. Pak se provede dekompozice z hlediska obou vstupních toků a vznikne tak více dílčích problémů a modulů /např. dva vstupní moduly, pro každý hlavní tok dat/.

Někdy také úplně chybí transformace, např. kopíruj soubor z A do B. Oba body abstrakce splynou v jeden a modul pro transformaci chybí.

Metoda se také poněkud liší podle toho, zda je dále dekomponován vstupní, transformační nebo výstupní modul. Pak platí:

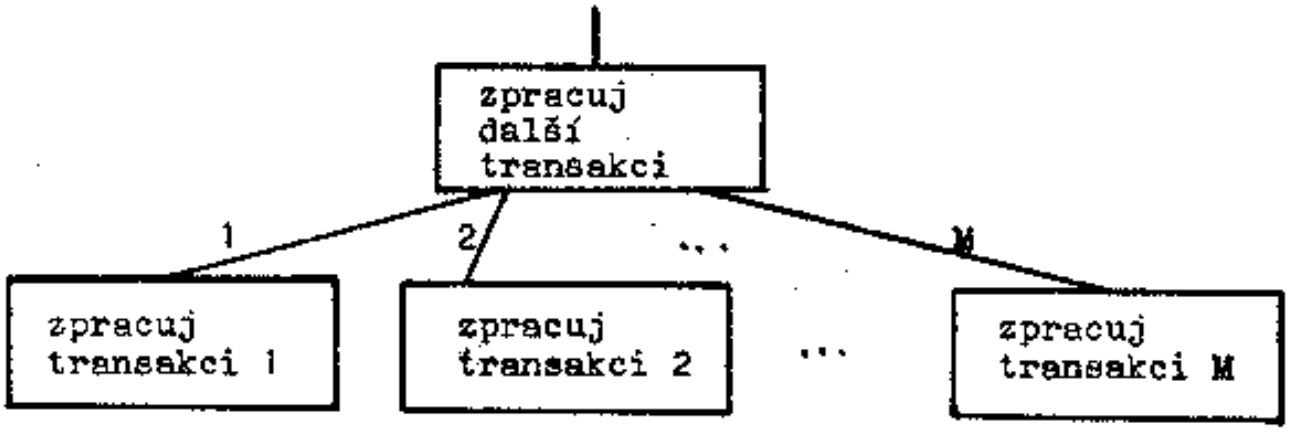
- Je-li dekomponován vstupní nebo transformační modul a vrací výstupní data nepřímo, bude mít podřízený výstupní modul. Pokud vrací výstupní data přímo /parametry/, podřízený výstupní modul mít nebude. Logika tohoto podřízeného výstupního modulu je totiž přímo součástí dekomponovaného modulu.
- Je-li dále dekomponován výstupní nebo transformační modul a dostává data nepřímo, bude mít podřízený vstupní modul. Pokud dostává data přímo /parametry/, podřízený vstupní modul mít nebude. Logika tohoto podřízeného vstupního modulu je totiž součástí dekomponovaného modulu.

6. 2. 3 Technika transakční dekompozice

Předpokládejme, že dekomponovaný modul má zajistit zpracování jedné z třídy alternativních odlišných funkcí/transakcí/ podle informace o typu funkce, který je mu předán. Přitom jednotlivé funkce mají stejnou úroveň a význam a realizují vždy odlišný soubor dílčích operací pro každou transakci-typ zpracování. Problém má selekční charakter, přičemž počet variant je znám. Modul je pak tzv. řízený daty, bude mít pravděpodobně logickou soudržnost. Vhodná technika pro dekompozici je transakční a vede na charakteristickou strukturu /obr. 8/.

Výhodou je, že moduly zpracovávající transakce jsou na sobě zcela nezávislé /i když třeba používají společné funkce v nižších úrovních/; modul realizující určitou transakci nemusí "znát" nic o jiných typech transakcí. Transakce jsou

tedy rovněž nezávislé, je možno snadno přidat další nebo některé vyloučit. Změna se odehraje jen v dekomponovaném modulu, který jediný má přehled o možných typech. Nemusí však znát nic o jednotlivých transakcích a jejich zadáních, poněvadž je předává všem stejným způsobem. Tato koncepce tak zvyšuje míru nezávislosti v programu. Dá se říci, že je specializovaným případem funkční dekompozice.



Obr.8. Struktura po transakční dekompozici modulu. Počet transakcí je znám, je roven v této verzi modulu M.

6. 3. Poznámky k aplikaci

I když je proces návrhu poměrně jednoduchý a návrhář vždy ví, co má v následujícím kroku dělat, nelze jej považovat za "kuchařku", neříká totiž vždy jednoznačně jak. Dekompoziční techniky i proces sám vyžadují orientaci v problému, speciální znalosti /kap.5/, kompromisy při rozhodování, takže zkušenosti a tvůrčí schopnosti jsou nezbytné. Celý přístup spíše poskytuje návrháři návod, jak použít některých prostředků a vodítek pro návrh, přičemž se nebrání adaptacím podle charakteru problému. Výsledkem jsou programy s vysoce soudržnými moduly s malou vzájemnou spřažeností. Typická je pro ně vysoká adaptabilita, snadná údržba a vysoká odolnost.

Velkým kladem tohoto přístupu je i to, že se v průběhu procesu návrhář podrobně seznámí s problémem a jeho strukturou. Může se pak stát /a dosti často se stává/, že po ukončení návrhu cítí, že by bylo možno realizovat návrh jinak, kvalitněji.

S ohledem na zvýšení odolnosti a délku života programu by se neměl rozpakovat provést příslušné změny, případně navrhnout celou strukturu znovu. Při návrhu velkých nebo komplikovaných programů je také možno nejprve navrhnout jen základní, jednodušší verzi oproštěnou od dílčích funkcí. Takto vzniklý návrh lze pak v dalších verzích jimi doplnit, případně po tomto ujasnění základních souvislostí realizovat celý znovu.

Po ukončení procesu návrhu se ještě struktura podle hledisek /kap.5/ optimalizuje /např. sdružení funkcí do modulu s informační soudržností/; po optimalizaci následuje kontrola ve dvou úrovních- statické a dynamické. Měli by se jí zúčastnit i návrháři předchozí a následující fáze procesu tvorby software. Při statické kontrole se ověří, zda návrh programu vyhovuje specifikaci a má požadované výstupy pro další fázi, tzn. přesné definice funkcí modulů a jejich interface /obsahově/. Přitom se prověřuje např.: zda mají všechny moduly informační nebo funkční soudržnost, pokud ne, tak proč; zda je mezi moduly jen vazba daty, když ne, tak proč; zda si lze snadno představit logiku všech modulů, zda je každá funkce definována jasně, atd. /viz [2], str.127/. Při dynamické kontrole se simuluje práce programu a prověřují jeho funkce.

Další fází návrhu v procesu tvorby software je vnější návrh modulů. Stručně uvedeme hlavní principy. Obsahem vnější specifikace modulu jsou všechny informace nutné pro jeho volání a nic více /nesmí tedy obsahovat popisy logiky/. Nemá také obsahovat informace o volaných modulech a kontextu použití. Obsahuje tyto hlavní informace:

- jméno modulu,
- funkci modulu,
- seznam parametrů,
- přesný popis všech vstupních parametrů /formáty a další atributy/,
- přesný popis všech výstupních parametrů, a popis které výstupy vznikají ze kterých a jakých vstupů,
- vnější účinky činnosti modulu /vnější programu nebo systému/.

7. ZÁVĚR

Koncepce navrhování modulárních programů a posuzování jejich kvality, tak jak ji předkládá MYERS v poměrně rozsáhlých pracích [1,2,3,7] neznamená nějakou převratnou změnu v technologii programování. Je přirozenou syntézou opírající se o dlouhodobé praktické zkušenosti i pokroky dosažené ve formalizaci procesu tvorby software.

Za důležitou vlastnost předkládané koncepce považujeme její jednoduchost, která implikuje snadnou použitelnost. Navrhovaná klíčová kritéria pro posuzování modularity programů /soudržnost a spřaženost/ mají platnost relativní a současně všechny"nectnosti"ordinálních veličin. Důležité je používat je vždy z hlediska cílů projektu. Přínos koncepce spatřujeme především ve dvou oblastech. Za prvé významně rozšiřuje nástroje v navrhování složitých programů, i když spíše vždy naznačuje směr než zcela přesný návod. Přitom je otázkou, zda právě to není další výhodou při aplikaci v nejrůznějších podmínkách. Druhý významný přínos pak spatřujeme v tom, že rozšiřuje možnosti jazyka, kterým se lze k navrhování programů, modularitě a kvalitě programů vyjadřovat.

Předložená koncepce je vhodná především pro navrhování středních až velkých programů se složitou strukturou, kterou je obtížné popsat na základě analýzy problému jinými prostředky. Podle našeho názoru je vhodná pro uživatelský software, vědeckotechnické výpočty, simulace, software operačních systémů, on-line aplikace, atd.

V tomto příspěvku už bylo věnováno dosti místa předmostem modularity. Považujeme za důležité upozornit i na některé obtíže, ke kterým modularita vede. Zvyšuje se počet členů v knihovnách /jak zdrojových, tak cílových/ a jsou tak kladeny vyšší nároky na označení a evidenci modulů. Požadavek na víceúčelové využití hotových modulů vyžaduje i věnovat větší pozornost dokumentaci a způsobu kódování. Všechny tyto záležitosti jde však zvládnout jednoduchými prostředky.

Nakonec jsme si ponechali vyjádření o vhodnosti zveřej-

nění tohoto příspěvku, který zcela nesplňuje v jiných souvislostech požadované vlastnosti "původní" práce. Je spíše pokusem o prezenteci, doplnění a zhodnocení Myersova přístupu. Domníváme se, že je užitečný především pro význam, který má a může mít aplikace této koncepce pro zkvalitnění procesu tvorby software a jeho produktů. Pokud se některými záměrnými či nechtěnými zjednodušeními a zkráceními vytratila srozumitelnost příspěvku, s omluvou odkazujeme čtenáře na citované prameny. Zároveň se omlouváme i G.J.Myersovi, pokud jsme vlastní interpretací a zohledněním vlastních zkušeností jeho koncepci významově posunuli.

Jsme vděčni RNDr. J.Zlámalovi, který nás před několika lety na práce G.J.Myerse upozornil. Zvláště pak děkujeme RNDr.J.Peškovi, Ing.K.Metzlovi, P.Jiříčkovi, p.m., RNDr.Č. Losertovi a P.Šimoníkovi za užitečné a inspirující diskuse o předkládaných tématech.

LITERATURA

1 MYERS G.J., Reliable Software through Composite Design, Van Nostrand Reinhold Comp., 1975

2 MYERS G.J., Composite/Structured Design, Van Nostrand Reinhold Comp., 1978

3 MYERS G.J., Software Reliability, Principles and Practices, J.Wiley, sons, 1976

4 COMER D., HALSTEAD M.H., A Simple Experiment in Top-Down Design, IEEE Transactions on Software Engineering, 5, 105-109 /1979/

5 HERKEL´I., Omožnostiach kvalitatívneho hodnotenia programov cez kvantitatívne veličíny, sborník seminář "Programování 79", DT ČSVTS Ostrava

6 METZL K., TVRDÍK J., Zdokonalené programovací techniky, tamtéž

7 STEVENS W.R., MYERS G.J., CONSTANTIN L.L., Structured Design, IBM System Journal, 13, 115-139/1974/

8 TRACZ W.J., Computer Programming and Human Thought Process, ACM Comp. Sci.Conference, Atlanta, /1977/

9 RAJLICH V., Úvod do teorie počítačů, SNTL, Praha, 1979

10 VOLÁK J., Úsporné programování, sborník semináře "Metody programování počítačů 3.generace, DT ČSVTS Ostrava, 1976