KIV/ZSWI 2003/2004
Přednáška 9

Moderní strukturovaná analýza

* Yourdon 1989
* místo čtyř modelů systému se zaměřuje přímo na nalezení esenciálního modelu systému, tj. logického modelu nového systému

* v moderní strukturované analýze se postupuje v těchto krocích:
 - vytvoříme model prostředí (definuje hranici mezi systémem a zbytkem světa)
  . nejprve definujeme účel systému (neměl by být delší než odstavec)
  . vytvoříme model kontextu systému
  . vytvoříme počáteční datový slovník definující data putující mezi systémem a terminátory
 - vytvoříme seznam událostí
 - na základě událostí vytvoříme předběžný model chování systému
 - model chování systému přestrukturujeme do konečného modelu chování (= esenciální model systému)
 - vytvoříme uživatelský implementační model (doplňuje esenciální model o informace nutné pro implementaci modelu)
 - konec analýzy, následuje návrh architektury, podrobný návrh a kódování

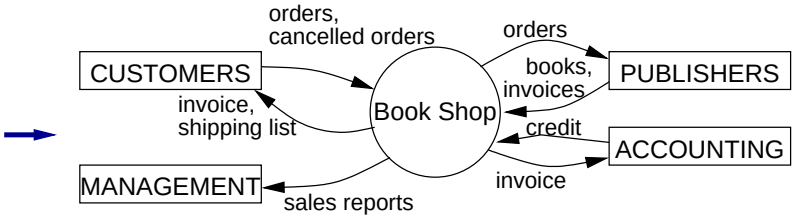
Vytvoření seznamu událostí

* události se klasifikují na:
 - událost datového toku (F) - událost která se projeví příchodem dat; např. událost ”přišla objednávka”
 - časová událost (T) - např. zpracování transakcí mezi bankovními účty nastane ve 3:00
 - řídící událost (C) - asynchronní událost např. v RT systémech

* identifikace událostí: procházíme každý terminátor, ptáme se jaké akce může provádět nad systémem (obvykle probíráme společně s uživateli, kteří hrají role terminátorů - analogicky jako při identifikaci případů použití v OO analýze)

EVENT LIST:

1. something happened (F)
2. ....



 - méně častá alternativa: vycházíme z ERA diagramu a hledáme události které způsobují vznik a zánik instancí entit a relací

* pro každého kandidáta na událost se ptáme, zda je skutečně událostí, tj. zda v jejím důsledku systém vyprodukuje výstup nebo změní svůj stav
* pro kandidáta na událost se ptáme, zda se všechny instance události týkají stejných dat (pokud ne, budou to nejspíš dvě různé události; cílem je rozlišit mezi různými událostmi, které se náhodou dějí společně nebo vypadají podobně)
* pak se pro každou událost ptáme ”musí systém nějak reagovat, pokud se událost neudá podle očekávání?” (tj. modelujeme odpovědi na chyby/poruchy terminátorů; např. zboží nepřijde v očekávané době, co musí systém udělat?)

Vytvoření předběžného modelu chování

* v klasické strukturované analýze se využívá postup shora dolů, pak ale často nastává tzv. problém ”paralýzy analýzy”: při definici velkého systému postrádáme vodítko jak vytvořit z kontextového diagramu DFD úrovně 1
* proto se v moderní strukturované analýze využívá identifikace odpovědí na události

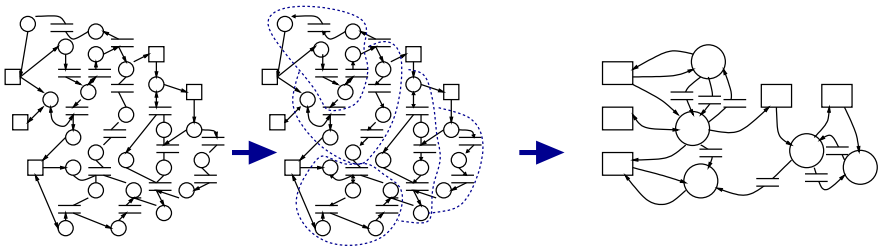
* postup pro vytváření předběžného modelu chování:
  - pro každou událost ze seznamu nakreslíme bublinu, očíslujeme jí podle čísla události
  - bublinu pojmenujeme podle odpovědi která by měla být sdružena s událostí (např. odpověď na ”zákazník zaplatil fakturu” je ”vlož platbu mezi příjmy”)
  - pokud je na událost více odpovědí, přidáme pro další nezávislou odpověď další bublinu (nezávislé odpovědi = mezi bublinami není komunikace)
  - k bublině nakreslíme vstupy, výstupy a potřebné paměti
  - identické bubliny sdružíme do jedné (identické bubliny mají stejný vstup, výstup a proces; např. různé typy objednávek mohou mít stejnou odpověď)
  - výsledný DFD zkontrolujeme oproti kontextovému diagramu a seznamu událostí

* výsledný model by měl
  - popisovat pouze logické procesy a podstatu transformace dat
  - zcela vynechat implementaci (nerozlišuje ani zda fci provádí člověk nebo existující počítačový systém) - proto vynecháme např.:
    . procesy jejichž účelem je přenos dat z jedné části systému do jiné
    . procesy pro verifikaci dat vzniklých uvnitř systému
    . procesy pro fyzický vstup a fyzický výstup (”vytiskni fakturu”)
  - je třeba rozlišovat mezi zdrojem informace (”obchodní zástupce”) a mechanismem pro vstup/výstup (”systém pro zadávání objednávek”); mechanismy vynecháme

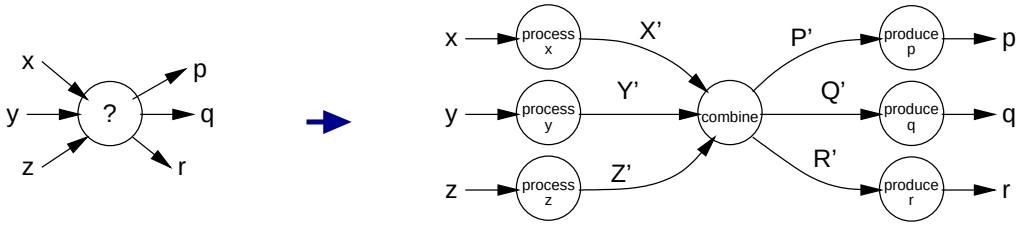
Dokončení modelu chování

* předběžný DFD chování systému bude zbytečně komplikovaný, zjednodušíme ho
* dokončíme specifikaci procesů
* dokončíme datový slovník

* zjednodušení DFD chování systému strukturováním
  - agregace = spřízněné procesy sdružíme do agregátů (= seskupení), které budou tvořit bubliny v DFD vyšší úrovně
  - každý agregát by se měl týkat úzce příbuzných odpovědí na události, tj. většinou procesů zpracovávajících příbuzná data
  - sdružovat bychom měli skupiny po cca 7+-2 procesech+pamětech
  - při tom máme příležitost ”schovat” paměti, které jiné procesy nepotřebují



* někdy procesy naopak nejsou primitivní a vyžadují další rozdělení
  - např. v následujícím příkladu: bublinu jsme nedokázali dobře pojmenovat, proto jí rozdělíme na primitivní procesy



* po dokončení modelu chování systému vytvoříme specifikaci procesů
* dokončíme datový slovník, ERA model

* výše uvedené informace tvoří tzv. esenciální model systému

Vytvoření uživatelského implementačního modelu
.....

Uživatelský implementační model pokrývá:

* alokaci esenciálního modelu na lidi a stroje (které bubliny a paměti budou realizovány manuálně) - to musí rozhodnout uživatel
* rozhraní aplikace s uživatelem (volba vstupních a výstupních zařízení, formáty obrazovek, formáty výstupů)
* omezení, např. objemy dat, časy odpovědí na různé události atd., tj. mimofunkční požadavky na aplikaci

* vytvoření uživatelského implementačního modelu je oficiálně posledním krokem moderní strukturované analýzy požadavků.
 - následuje návrh architektury, podrobný návrh, implementace, testování

Poznámka:

Při analýze nemusejí být použity všechny nástroje (modely) - používají se vždy pouze ty nástroje, které mají v daném kontextu smysl. Například pokud jsou datové struktury jednoduché, nemusíme vytvářet ERA diagramy.

[]

Kromě klasické (DeMarcovy) metodologie a (Yourdonovy) moderní strukturované analýzy se v praxi používají další metodiky pro strukturovanou analýzu, např. SADT, SREM/RDD, SA/SD. V Česku je z nich asi nejpoužívanější SSADM (Structured Systems Analysis and Design Method); má v podstatě podobný záběr a nástroje jako DeMarcova a Yourdonova strukturovaná analýza (výstupy - DFD, model entit, ...), ale je to standard - velmi podrobně definované kroky včetně výstupů a předepsaných kontrol před přechodem k dalšímu kroku. Použití metodiky SSADM je v některých zemích podmínkou pro získání státních zakázek.

Návrh architektury
==========

* architektura softwaru je popis podsystémů (komponent) a vztahů mezi nimi
 - jednotlivé aspekty architektury se obvykle nazývají ”pohledy” (viewpoints)
 - např. fyzický pohled (umístění komponent), procesní pohled (týká se paralelismu), strukturální pohled (jak vývojáři rozdělí systém do implementačních částí) apod.
* při klasickém postupu vycházejícím ze strukturované analýzy nás bude zajímat zejména fyzický a strukturální pohled
 - nejprve rozhodneme, které části esenciálního modelu budou alokovány na které počítače (např. celý systém bude implementován na jednom počítači)
 - pak přiřadíme ”procesy” a ”paměti” alokované na stejný počítač jednotlivým procesům

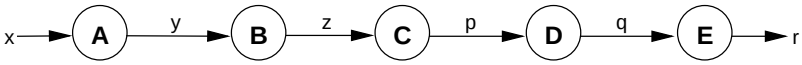
Podrobný postup návrhu architektury byl již popsán v 6. přednášce - pro strukturované metody vývoje je postup v zásadě stejný a proto ho zde nebudu opakovat. V mnoha případech je vhodné vycházet z tzv. architektonických stylů (makro-architektonických vzorů), které představují typické řešení architektury pro daný typ aplikace.

Příklady architektonických stylů uvedu později.

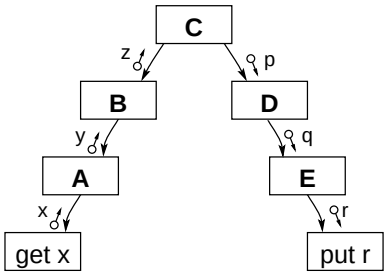
Strukturovaný návrh systému
==========

* někdy se také nazývá funkčně orientovaný návrh (function-oriented design)
* předcházela mu strukturovaná analýza a návrh architektury
 - produktem analýzy jsou:
  . model kontextu systému resp. DFD úrovně 0
  . množina diagramů datových toků (DFD)
  . specifikace činnosti elementárních procesů (pseudokódy,
    Nassi-Shneidermanovy diagramy, rozhodovací tabulky a stromy)
  . datový slovník
  . ERA diagramy
 - produktem návrhu architektury je rozhodnutí, které části esenciálního
   modelu budou přiřazeny jednotlivým podsystémům, případně na které počítače
   nebo procesory budou alokovány (např. celý systém bude implementován na
   jednom počítači)

* podrobný návrh aplikace probíhá v následujících krocích:
 - v rámci podsystémů transformace DFD na hierarchii podprogramů (každá
   bublina se stane podprogramem)
 - např. z DFD



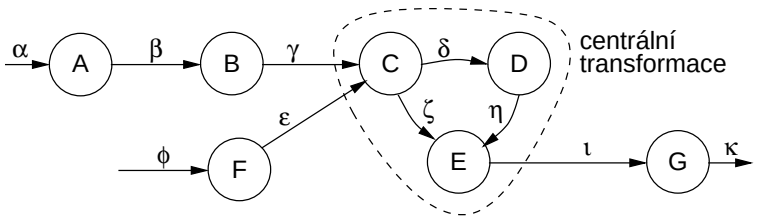
vzniknou podprogramy s následující hierarchií:



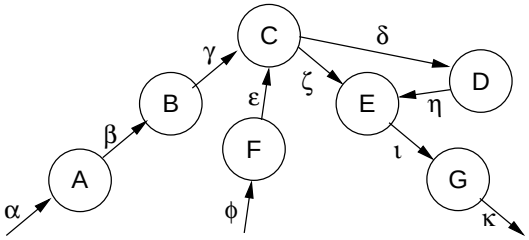
Transformace DFD na hierarchii podprogramů
.....

V obecném případě není převod DFD tak přímočarý jako v předchozím příkladu,
proto se používá následující postup:

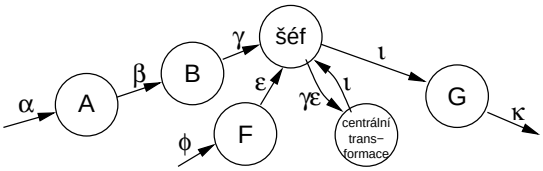
* vycházíme z DFD nejnižší úrovně (bubliny = elementární procesy); pro každý
  DFD provedeme:
 - identifikujeme centrální transformaci = část DFD popisující základní fce
   systému, nezávislá na implementaci vstupů a výstupů
 - dva způsoby hledání centrální transformace:
  . buď předpokládáme ”ideální svět”, kde vstupy nikdy neobsahují chyby,
    výstupy nemusí být formátovány apod.; odsekáváme všechny nepotřebné
    vstupní a výstupní proudy, co nám zůstane je centrální transformace
  . nebo najdeme ”střed” DFD odhadem (to je horší varianta, ale nic jiného
    nám nezbývá pokud nedokážeme centrální transformaci identifikovat jinak)
 - kolem centrální transformace nakreslíme obláček



* vytvoříme hierarchii procesů
 - hierarchie procesů bude vyjadřovat vztah nadřízenosti a podřízenosti
 - nejprve musíme najít kandidáta na šéfa
  . pokud je dobrý kandidát (v centrální transformaci), zvedneme ho a necháme ostatní bubliny viset dolů, viz obrázek (a)
  . pokud je potenciálních kandidátů více, vyzkoušíme který nám poskytne nejlepší návrh



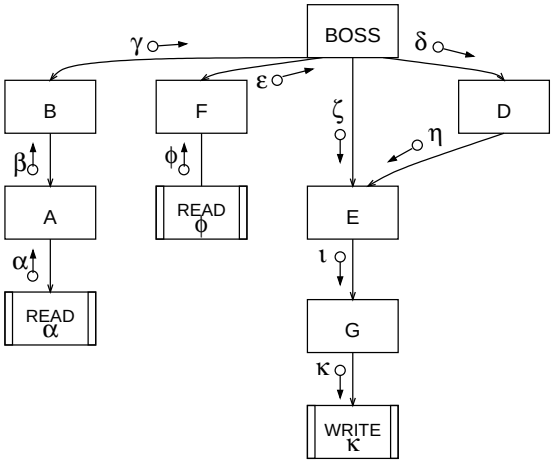
a) vyzvednutí šéfa



b) vytvoření nového šéfa

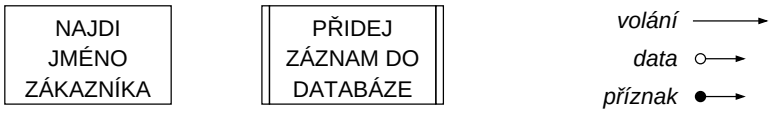
  . pokud není dobrý kandidát, vytvoříme nového šéfa, centrální transformaci nahradíme šéfem a původní transformaci pod šéfa zavěsíme (jako jednu bublinu, vstupy a výstupy budou proudit přes šéfa; viz obrázek (b))

* transformace DFD na hierarchii bloků (každá bublina se stane blokem)
 - ”šéf” se stane řídícím blokem, ostatní budou jeho podřízení
 - vytvoříme počáteční strukturogram



 - šipky ukazují směr volání, přidali jsme bloky pro vstup a výstup
 - jména bloků by měla odpovídat jejich rolím v hierarchii, tj. nemusejí odpovídat názvům bublin - jméno by mělo sumarizovat i činnost podřízených bloků
  . příklad názvů bloků - postupně: ”získej položku” (blok pro čtení dat), ”získej transakci” (pomocí nižšího bloku vytvoří transakci), ”získej platnou transakci” (ověří platnost transakce)

 - strukturovaný design se znázorňuje pomocí tzv. strukturogramů (structure charts)
  . strukturogramy ukazují rozdělení systému do hierarchie bloků a jejich vzájemnou komunikaci
  . blok představuje proceduru nebo fci programovacího jazyka; blok se znázorňuje jako obdélník obsahující název bloku
  . knihovní procedury a fce (předdefinované bloky) se znázorňují pomocí zdvojení svislých čar obdélníka
  . volání podprogramu je znázorněno obyčejnou šipkou (od volajícího k volanému)
  . předávání zpracovávaných dat je znázorněno šipkou s prázdným kroužkem (šipka směřuje od odesilatele k příjemci dat)
  . předávání příznaků je znázorněno šipkou s plným kroužkem



blok *knihovní (předdefinovaný) blok*

Poznámka pro zajímavost (blok vs. modul ve strukturovaném designu)

V původní literatuře o strukturovaném návrhu se o blocích hovoří jako o ”modulech”. Termín ”modul” má ale dnes už jiný význam než v roce 1980, proto ho v souvislosti se strukturogramy raději nebudeme používat.

[]

* úprava strukturogramu
 - přidáme čtecí a zápisové bloky, bloky pro čtení z databáze apod.
 - centrální transformaci můžeme rozdělit podle DFD
 - přidáme bloky pro obsluhu chyb
 - pokud jsou potřebné, přidáme bloky pro inicializaci a ukončení programu
 - pokud je nesprávný výběr šéfa, změníme ho (nesprávný výběr šéfa se obvykle projeví tak, že pravý šéf signalizuje svému nadřízenému, co má dělat)

* ověření funkčnosti návrhu - implementuje strukturogram požadavky vyjádřené v DFD?
 - pokud si nějaký blok potřebuje vyžádat data, může to udělat?
 - pokud ne, můžeme změnit hierarchii volání

* pokud jsme výše uvedené kroky provedli pro jednotlivé DFD, máme k dispozici množinu nezávislých strukturogramů
 - vytvoříme nadšéfa, který bude volat jednotlivé šéfy
 - optimální je, pokud nadšéf obsahuje konstrukci case, která vyvolá některého podřízeného (např. výběrem položky v menu); v takovém případě bychom museli už DFD rozdělit podle typů transakcí

* tím končí fáze strukturovaného návrhu
 - hlavním účelem strukturovaného návrhu je rozdělení systému do bloků
 - výstupem je jeden nebo více strukturogramů, z bloků vzniknou podprogramy
 - strukturovaný návrh neřeší sdružení podprogramů do modulů (to probereme dále) nebo návrh vnitřku podprogramů (zde můžeme vycházet z pseudokódu, viz přednáška o kódování)

Charakteristiky kvalitních podprogramů

Dvě základní charakteristiky dobrého návrhu podprogramů je silná soudržnost a slabá provázanost.

Soudržnost

* soudržnost (cohesion) = jak blízký vztah k sobě mají operace uvnitř podprogramu
* existuje několik úrovní soudržnost: funkční, sekvenční, komunikační atd.

* funkční soudržnost - podprogram vykonává právě jednu funkci
 - např. fce sin() bude mít funkční soudržnost, protože vykonává jedinou fci
 - fce sin_and_tan() by neměla funkční soudržnost, protože by prováděla dvě fce
 - podprogramy by měly být silně soudržné, tj. dělat pokud možno jen jednu věc
 - důsledkem silné soudržnosti je vyšší spolehlivost (to je prokázáno studiemi)

Z výše řečeného vyplývá, že funkce by neměly být víceúčelové (klasický špatný příklad: fce realloc(ptr, size) jazyka C zvětšuje alokovaný úsek paměti, zmenšuje ho, uvolňuje (pokud size=0), alokuje nový úsek paměti (ptr=NULL), atd.). Proto je to také nejhůře srozumitelná fce standardní knihovny jazyka C.

* sekvenční soudržnost - slabší typ soudržnosti než je funkční soudržnost
 - podprogram sestává z kroků, které se musejí provést v určitém pořadí
 - kroky postupně zpracovávají sdílená data, ale netvoří ucelenou fci

- například program bude provádět 10 kroků, pokud je prvních 5 v jedné fci a druhých 5 ve druhé, je to sekvenční soudržnost
- funkční soudržnost je lepší => fce sdružíme do jedné, operace bude ucelená

* komunikační soudržnost - ještě slabší typ
- operace provádějí zpracování stejných dat, ale jinak spolu nemají nic společného; např. změna dvou nesouvisejících prvků datové struktury
- z praktických důvodů ještě akceptovatelné

Poznámka (analogie soudržnost podprogramů pro třídy: soudržnost třídy)

Jedna z charakteristik dobře navržených tříd je soudržnost třídy; je to analogie soudržnosti podprogramů, avšak o jednu úroveň zapouzdření výše. Ukazuje vzájemný vztah sady atributů a operací tvořících rozhraní třídy. Podrobněji viz (Page-Jones 2001), kap. 9.3.

[]

Provázanost
....

* stupeň provázanosti (degree of coupling) = jak blízký vztah mají dva podprogramy
- např. fce sin(x) má slabou provázanost s ostatními podprogramy, protože jediné co potřebuje znát je jednoduchý vstupní parametr x
- poněkud silnější typ provázanosti nastává, pokud si podprogramy předávají datové struktury
- ještě silnější provázanost nastává, pokud procedury používají globální data atd.

V praxi obvykle není důležité určovat přesnou úroveň soudržnosti a provázanosti, ale je třeba se snažit o silnou soudržnost a slabou provázanost.

Vytváření modulů
----

* pro účely dalšího výkladu zadefinujeme následující termíny:
- podprogram = procedura nebo funkce
- modul = množina dat a podprogramů, například ”unit” v některých verzích Pascalu, zdrojový soubor v C apod.
  . bývá obvykle definován jako nejmenší jednotka zdrojového textu programu, kterou můžeme samostatně přeložit

Vylepšení modularity podsystému
....

Poté, co byla vytvořena struktura podsystému, můžeme jí dále vylepšit pomocí následujících pravidel:

* projdeme jednotlivé podprogramy, zmenšujeme provázanost a zvyšujeme soudržnost
- pokud je ve dvou podprogramech stejná činnost, může být vyjmuta do samostatného podprogramu
- pokud jsou dva podprogramy vysoce provázány, může být vhodné je spojit (zjednoduší se jejich rozhraní)
* omezení působnosti procedury: procedura by měla mít vliv pouze na podřízené procedury
- podřízených procedur nemá být velké množství (”low fan-out”)
* s rostoucí hloubkou strukturogramu se snažte o to, aby procedura byla volána vyšším počtem nadřízených procedur (”high fan-in”)
- strukturogram by měl mít tvar oválu, na spodní úrovni by měly být všeobecně užitečné procedury, které bude volat více nadřízených
* rozhraní každého podprogramu se snažíme co nejvíce zjednodušit

Návrh modulů

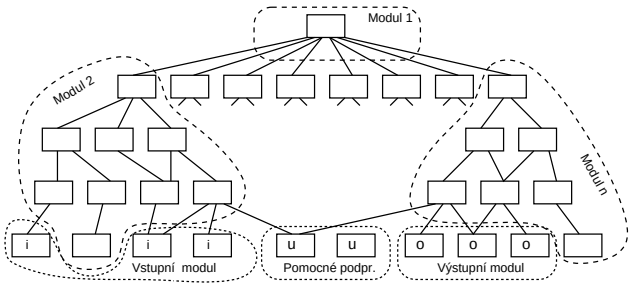
* perfektní modularita jednotlivých podprogramů = komunikace pouze pomocí jednoduchého rozhraní
  - skupina podprogramů může sdílet společná data, podprogramy pak nejsou perfektně modulární
  - pokud společná data skryjeme uvnitř modulu (budou přístupná pouze podprogramům uvnitř modulu) a zbytek programu může s modulem komunikovat pouze pomocí jeho oficiálního rozhraní, bude skupina podprogramů perfektně modulární, i když jednotlivé podprogramy perfektně modulární nejsou

* moduly procedurálních programovacích jazyků jsou vlastně ”poor folks’ objects”
  - sdružují data a operace nad daty, případně jinou navzájem související množinu služeb
  - moduly podporují OO koncepty abstrakce a zapouzdření, nepodporují dědičnost
  - dobrý modulární návrh podstatně usnadňuje údržbu (lokalizace změn)

* moduly je vhodné vytvořit pro všechny služby, jejichž implementace se může změnit nebo pro součásti, které lze využít v jiných projektech
  - uživatelské rozhraní: bude se pravděpodobně vyvíjet, zbytek systému by jím neměl být ovlivněn
  - HW nebo systémové závislosti: pro snadnější přenos systému do jiného prostředí (například operační systémy: moduly pro jednotlivé HW architektury, ovladače apod.; nebo Netscape - moduly pro práci se sítí ve Windows a v UNIXu)
  - vstupy/výstupy: snadnější změna formátu souborů, výstupních tiskových sestav apod.
  - správa dat - moduly pro každý typ dat; k datům přistupujeme pouze pomocí podprogramů, což usnadňuje změnu reprezentace apod. Např. modul implementující zásobník by měl procedury rozhraní init_stack(), push() a pop()
  - moduly pro navzájem příbuzné operace, např. manipulace s řetězci, statistické fce, grafické podprogramy apod. Např. modul obsahující trigonometrické fce sin(), cos(), tan() atd.

Vyváření modulů v návaznosti na strukturovaný design

* pokud navazujeme na strukturovaný design, máme k dispozici strukturogram
  - sdružíme související bloky nízké úrovně (bloky nízké úrovně budou nejčastěji vstupy, výstupy, přístup k databázi, pomocné podprogramy, uživatelské rozhraní)
  - zbytek strukturogramu rozdělíme na části, které jsou silně kohezivní a k ostatním částem se váží volně (snažíme se o minimální rozhraní mezi částmi)
  - bloky v každé části budou tvořit jeden nebo více modulů



Podle Page-Jonese (1980) se ukazuje jako nejlepší následující přiřazení modulů týmům programátorů:

* návrháři systému kódují moduly vyšší úrovně
* jednotlivé typy modulů nižší úrovně přiřadíme týmům, které s danou oblastí mají zkušenosti (databáze, uživatelské rozhraní apod.)
  - týmy mohou obsahovat i méně zkušené programátory, návrháři systému na ně dohlížejí

Poznámka (klasická chyba při návrhu modulů)

Mezi klasické chyby patří návrh modulů tak, aby odpovídal počtu dostupných programátorů. Správný návrh zapouzdřuje podprogramy do modulů podle výše uvedených kritérií (jako je zapouzdření příbuzných operací atd.) a na počtu programátorů by neměl přímo záviset.

[]

❇