KIV/ZSWI 2003/2004
Přednáška 10

Návrh uživatelského rozhraní

* většinu aplikací můžeme rozdělit do dvou částí - aplikační logika a uživatelské rozhraní (user interface, dále jen UI)
* starší aplikace textové rozhraní, v současných systémech grafické uživatelské rozhraní (GUI) protože dovoluje podstatně více možností

Doporučení pro návrh UI

Mandel (1997) popisuje ve sv knize o návrhu uživatelských rozhraní tři často citovaná ”zlatá pravidla”:

* systém by měl reagovat na potřeby uživatele, tj. uživatel by ho měl řídit (nikoli naopak)
* rozhraní by mělo být konzistentní = všechny jeho části by se měly chovat podobně
* dobře navržené UI by nemělo mít požadavky na paměť uživatele

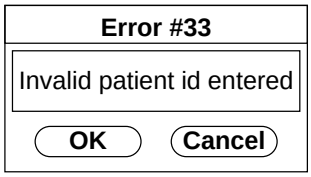
Tato pravidla můžeme trochu rozepsat:

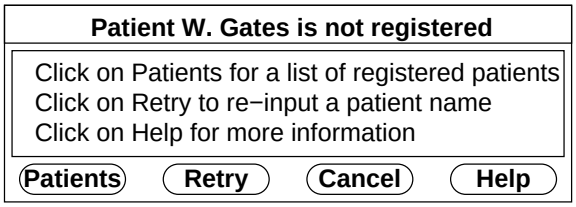
* systém by měl reagovat na potřeby uživatele, tj. uživatel by ho měl řídit
  - uživatel by neměl být nucen používat nějaké rozhraní jen proto, že se snadno implementuje; uživatel by neměl být nucen vykonávat nechtěné akce
    . např. pokud spustím fci spell check textového procesoru a uvidím místo, které bych chtěl opravit, aplikace by mě neměla uživatele nutit zůstat v režimu spell check
  - základní terminologie rozhraní by měla vycházet z aplikační domény (uživatel se pohybuje ve virtuálním světě aplikace, např. interaguje s objekty které se objevují na obrazovce)
    . např. v systému pro řízení letového provozu bude operátora informovat o letadlech, letových koridorech, radiomajících apod.; implementace (přístup k databázi, komunikace atd.) by měla být před uživatelem skryta
  - pokud uživatel udělá chybu, má mít možnost operaci předčasně ukončit (cancel) nebo se vrátit do stavu před vykonáním chybné akce (undo)
    . mechanismus ”cancel” - uživatelé budou nevyhnutelně při používání systému chybovat, UI má obsahovat možnost zrušení části nebo celé transakce (např. zadávání pacienta)
  - UI by mělo poskytnout přiměřené možnosti pro různé typy uživatelů; zkušení uživatelé budou chtít např. klávesové zkratky pro zrychlení přístupu k možnostem systému, na druhou stranu začátečníci potřebují vedení a nápovědu ve stylu kuchařky
    . např. některé volně šířené programy dovolují nakonfigurovat, zda je uživatel začátečník, středně pokročilý či pokročilý a podle toho uživatele v různé míře ”obtěžují” nápovědou

* UI by mělo být konzistentní, tj. všechny příkazy a menu by měly používat stejný formát
  - konzistence redukuje čas učení, protože znalost získanou v jedné části systému lze použít i v jiné části systému
  - UI by mělo být založeno na srozumitelné metafoře (analogii) z reálného světa
* s konzistencí souvisí princip nejmenšího překvapení - podobné akce vykonaná různých kontextech by měly mít podobné důsledky; pokud se systém zachová neočekávaným způsobem, budou uživatelé nemile překvapeni
  - např. pokud při kreslení jednoho objektu znamená stisk pravého tlačítka myši zrušení objektu a při kreslení jiného připojení další čáry, není UI aplikace konzistentní a může dojít k nemilému překvapení uživatele
  - UI by mělo být konzistentní i mezi podsystémy, např. klávesa Backspace by měla vždy dělat to samé (např. zrušit znak vlevo od kurzoru)

* dobře navržené UI by nemělo mít požadavky na paměť uživatele
  - UI by mělo omezovat potřebu uživatele pamatovat si předchozí akce a jejich výsledky (mělo by uživateli napomáhat zjistit či připomenout apod.)

- UI by mělo obsahovat nápovědu, při chybě by měla být poskytnuta smysluplná zpětná vazba v terminologii uživatele
- chybové zprávy by měly sdělit typ chyby a kde se chyba udála (oznámení "CHYBNÝ VSTUPNÍ ÚDAJ" nám nic neřekne
- chybové zprávy by měly být konstruktivní, a kdekoli je to možné, tam by měly napovědět, jak může být chyba opravena (levá zpráva je negativní, obviňuje uživatele z toho že udělal chybu, nepoužívá jazyk uživatele (patient id); pravá je pozitivní, říká že problém je v systému a poskytuje návod jak chybu napravit)





Webové aplikace by navíc měly uživateli poskytovat odpovědi na tři hlavní otázky (Dix 1999):

* kde jsem? - tj. jakou aplikaci právě používám, na jakém místě jsem v hierarchii stránek
* co mohu dělat? - tj. jaké funkce jsou nyní dostupné
* kde jsem byl a kam jdu? - srozumitelná navigace

Další poznámky:

* barva by neměla být primárním nositelem významu, protože cca 10% populace je v různé míře barvoslepých
* zvukovým efektům se pokud možno vyhýbáme
* důležité je uvažovat požadavky na čas odpovědi (délku i variabilitu); pokud je čas delší než cca 2s, je důležité, aby uživatel viděl, že se něco děje (např. zobrazením "teploměru")
* je třeba počítat s tím, že uživatelé prakticky nikdy nečtou manuál; ovládání musí být intuitivní, nápověda snadno dostupná

Poznámka pro zajímavost (doporučení pro konkrétní GUI)

Pro konkrétní GUI obvykle existují doporučení výrobce (style guide) pro návrh aplikace s využitím poskytovaných prvků GUI. Doporučení bude obsahovat například pravidlo, že defaultní hodnota ve formuláři má být nejpravděpodobnější volba, případně nejméně nebezpečná volba; pokud je defaultní hodnota odvozena z jiných hodnot zadaných uživatelem, pak ty hodnoty, ze kterých odvozujeme další, mají být blízko začátku formuláře atd.

Jako příklad uvedu "Java Look and Feel Design Guideline", kterou najdete na adrese http://java.sun.com/products/jlf/ed2/book/index.html .

[]

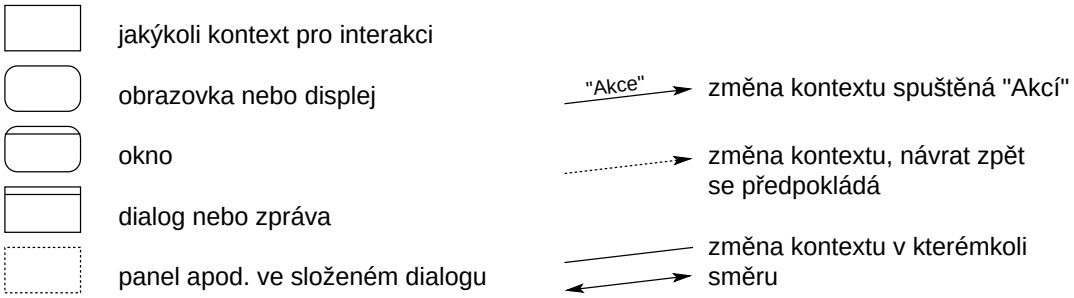
Návrh UI
.....

* můžeme vycházet ze scénáře, který se váže k případu použití
* procházíme scénář, identifikujeme hraniční objekty (podstatná jména) a akce (slovesa), týkající se komunikace uživatele se systémem
  - ptáme se: jaké informace bude uživatel potřebovat zadávat, měnit, prohlížet?
  - např. uživatel zadá {jméno} a {heslo}

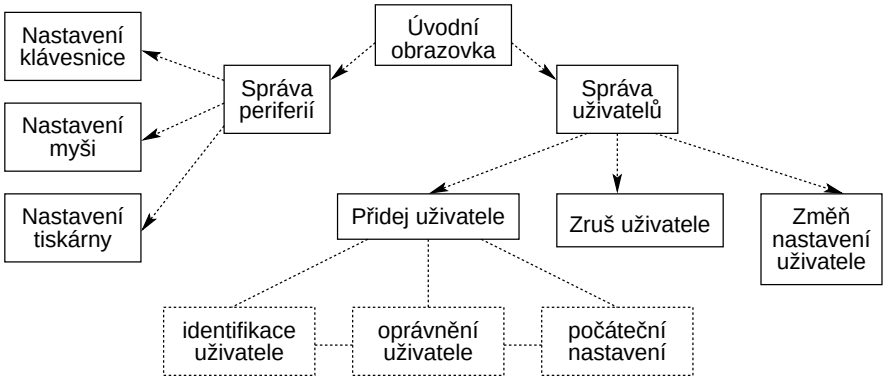
* výhodné je začít na papíře - čistý papír je pracovní plocha (časem se může stát oknem aplikace, dialogovým oknem apod.)
  - pro každou pracovní plochu si poznamenáme "materiál" (data) a "nástroje" (aktivní prvky pracující s materiálem)

- pracovní plochu pojmenujeme

* vytvoříme diagram procházení pracovními plochami (navigation map) -
  popisuje jak budou uživatelé moci přejít z jedné pracovní plochy do jiné
 - pro znázornění můžeme používat následující notaci (Constantine 1998):



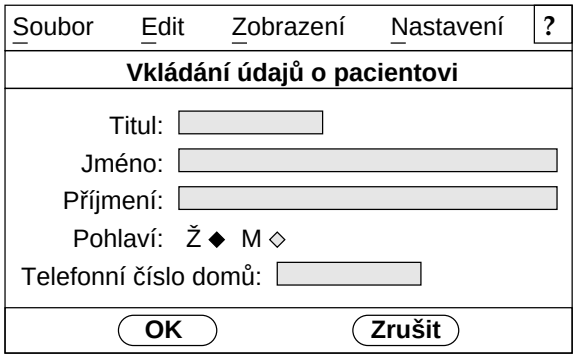
- například částečný diagram programu pro správu systému by mohl vypadat
  následovně:



Poznámka: pokud byste chtěli ke stejnému účelu použít UML, můžete použít
např. stavový diagram - události popíšeme klávesou, příkazem z menu atd.

* abstraktní prototyp převedeme na vzhled obrazovek
 - typicky se začíná tím, že každou abstraktní komponentu převedeme
   implementujeme jako jeden standardní prvek GUI (widget)
 - pak se snažíme zjednodušit způsob práce nebo ušetřit místo na obrazovce

* pro návrh UI běžných systémů můžeme použít diagramy rozvržení oken
  (window layout diagram, viz také [Page-Jones: Základy OO návrhu v UML.
   Grada 2001], str. 168)
 - zobrazuje přibližně návrh okna, neobsahuje všechny kosmetické doplňky
 - můžeme začít hrubým návrhem prototypu pomocí papírového modelu UI,
   nakreslením obrázku na počítači, případně v nástroji pro vytváření oken
 - informace ve formulářích by měla být požadována v logické posloupnosti
   (např. oslovení, jméno, příjmení; nikoli jméno, oslovení, příjmení)



- při návrhu vzhledu je nejvýhodnější evoluční vývoj (exploratory

development), kde uživatel návrh vyhodnocuje nebo spoluvytváří

Vyhodnocení UI

* UI je obtížné vyhodnotit bez otestování na skutečných uživatelích
  - dotazníky - co si uživatelé o rozhraní myslí
  - pozorování uživatelů při práci, při pokusu vyřešit úlohu ”myslí nahlas”
  - snímání typického využití systému videokamerou
  - vložení kódu který sbírá informace o nejpoužívanějších vlastnostech a nejčastějších chybách
* žádný ze způsobů nedetekuje všechny problémy UI

* dotazníky, otázky mohou být (1) jednoduché odpovědi typu ano/ne, (2) číselné odpovědi, (3) subjektivní oznámkování, (4) subjektivní procentuální odpověď; např.
  1. Jsou ikony srozumitelné? Pokud ne, které ikony jsou nejasné?
  2. Jak obtížné bylo naučit se základní práci se systémem? Ohodnoťte obtížnost na stupnici 1 až 5 (kde 5 je nejvyšší obtížnost).
  3. Kolik % fcí systému jste použila? (Všechny = 100%, žádné = 0%)
  4. Jak srozumitelné jsou chybové zprávy? ...

* pozorování a nahrávání, které vlastnosti používají, jaké chyby dělají; analýza nahrané relace dovoluje pozorovat např. zda UI nevyžaduje příliš mnoho pohybů rukou (problém některých GUI je nutnost opakovaně přesouvat ruce mezi myší a klávesnicí - optimální je, pokud lze aplikaci ovládat pouze pomocí klávesnice)

* sběr statistik - zjištění nejčastějších operací => UI může být upraveno tak, aby tyto operace bylo možné zvolit nejrychleji

Architektonické styly

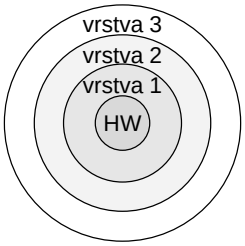
* architektonický styl (angl. architectural style, macro-architectural pattern) je znovupoužitelná abstrakce systému
  - v praxi se vyskytuje opakovaně v různých aplikacích
  - můžeme z něj vycházet při tvorbě vlastních aplikací
* vlastnosti jednotlivých stylů jsou známé z již existujících aplikací daného stylu (škálovatelnost, výkonnost, bezpečnost apod.)
  - styl slouží jako komunikační nástroj a základ pro managerské rozhodování (např. pro rozdělení do týmů, organizaci dokumentace projektu apod.)
  - styl je popsán:
    . množinou podsystémů a jejich typů (např. datové úložiště, proces, UI)
    . topologickým uspořádáním podsystémů
    . množinou sémantických omezení (např. datové úložiště nesmí měnit uložené hodnoty)
    . množinou interakčních mechanismů (volání podprogramu, událost, roura)

V dalším textu si ukážeme nejdůležitější architektonické styly.

Obecné struktury

Vrstvené systémy

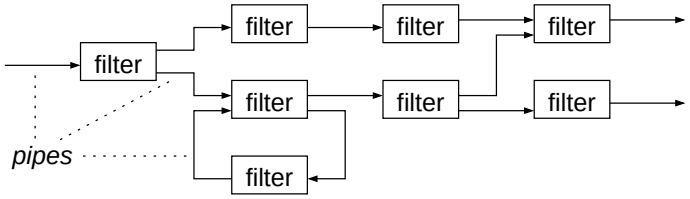
* systém strukturujeme do podsystémů které tvoří vrstvy, vrstva používá pouze služby nižších vrstev
  - používají se, pokud fčnost může být rozdělena na část specifickou pro aplikaci a generickou část použitelnou pro mnoho aplikací
  - další důvody:
    . pokud části systému mají být nahraditelné
    . pokud je důležitá přenositelnost do různých OS nebo HW
    . pokud můžete využít již existující vrstvu (OS, komunikaci po síti...)
  - používá se v mnoha SW produktech, například některé operační systémy



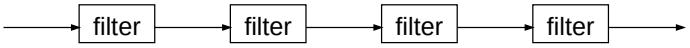
Styl dataflow
....

* používá se pokud se můžeme na systém dívat jako na posloupnost transformací
  zpracovávajících vstup a produkujících výstup
* výhodou integrovatelnost - relativně jednoduché rozhraní mezi komponentami
* dva hlavní podstyly - ”roury a filtry” a ”dávkově sekvenční” architektura

* roury a filtry (pipe-and-filter architecture)
 - podsystémy nazýváme filtry, jsou propojené rourami které přenášejí data
 - každý filtr pracuje nezávisle, pouze očekává data v určitém formátu a
   produkuje výstup v určeném formátu



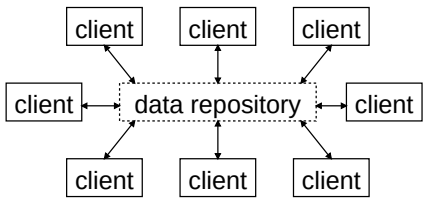
* dávkově sekvenční
 - pokud výše uvedená architektura degeneruje do jedné lineární posloupnosti
   transformací
 - vstupem dávka dat, aplikuje na ní posloupnost sekvenčních komponent (filtrů)



 - příkladem mohou být překladače programovacích jazyků, např. překladač
   jazyka C: zdrojový text nejprve zpracuje preprocesor, poté se provede
   lexikální a syntaktická analýza, optimalizace, nakonec generování kódu

Pasivní datové úložiště: styl repositář (repository)
....

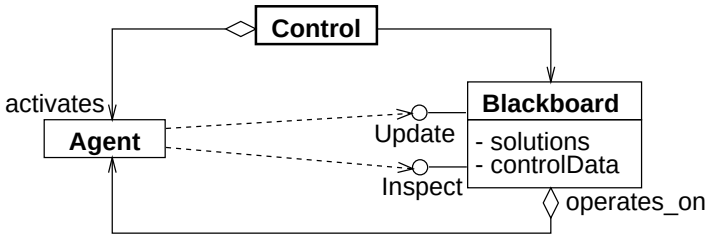
* centrem architektury je datové úložiště (databáze nebo soubor)
* ostatní podsystémy (klienti) k úložišti přistupují a čtou, přidávají, ruší
  nebo modifikují data



* tento styl se používá, pokud je požadováno uchování, výběr a správa velkého
  množství dat a pokud jsou data vhodně strukturovaná
* hlavní cíle:
 - integrovatelnost, klientské podsystémy pracují nezávisle
 - škálovatelnost (scalability) = možnost snadno přidávat nové klienty a data

Aktivní datové úložiště: styl tabule (blackboard)
.....

* množina agentů spolupracuje pomocí datového úložiště
 - úložiště je aktivní, posílá oznámení agentům o změně dat, která je zajímají
 - agent vyhodnotí obsah úložiště, případně vloží výsledek nebo částečné řešení
 - obrázek znázorňuje příklad uspořádání; agenti jsou aktivováni řídícím objektem/modulem, využívají rozhraní tabule pro přístup k datům



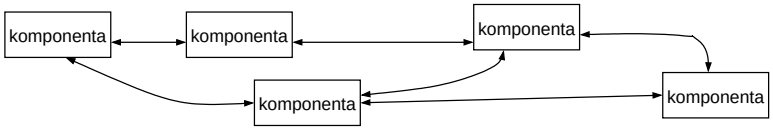
 - například systémy pro umělou inteligenci, rozpoznávání řeči apod.
 - tj. systémy kde neznáme vhodné uzavřené (algoritmické) řešení problému, ale umíme řešit pomocí agentů (např. znalostních agentů) kteří k řešení přispívají

Distribuované systémy

Pokud systém můžeme strukturovat jako množinu volně vázaných podsystémů, podsystémy mohou běžet na nezávislých strojích propojených sítí.

Peer-to-peer
.....

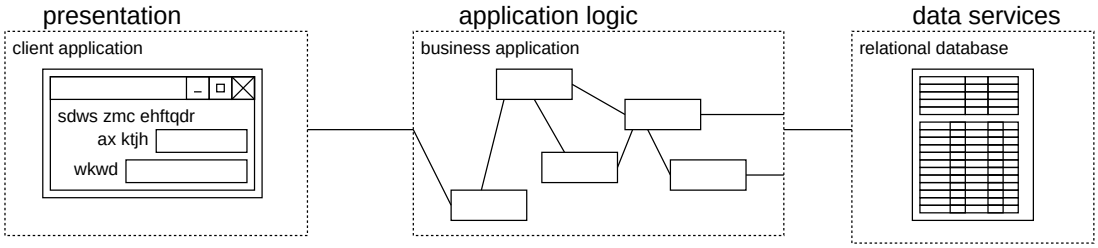
* komponenty spolu mohou navázat komunikaci - vyměňují si informace podle potřeby



Architektura klient/server
.....

* pokud úloha může být rozdělena na tvůrce požadavků (klient) a jejich vykonavatele (server)
* v systému je alespoň jeden klient a alespoň jeden server
* nejčastější podstyly stylu klient-server:
 - třívrstvá architektura (3-tier architecture)
 - tlustý klient (fat client)

* třívrstvá architektura (3-tier architecture)
 - funkčnost se rozděluje do 3 částí:
  . klienti - obsahují prezentační služby, obvykle jeden klient slouží jednomu uživateli
  . datové služby - obvykle jsou implementovány pomocí databázového serveru
  . aplikační logika - informace vytváří a modifikuje pomocí datových služeb, poskytuje informace klientům



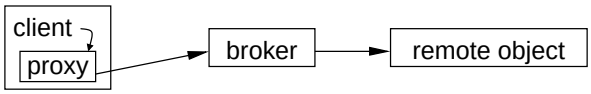
- v současné době nejoblíbenější, ”SW průmysl se neodvratně řítí tímto směrem” (citace z jedné přednášky)
- častá varianta: tenký klient - klientem je např. WWW prohlížeč

* tlustý klient (fat client)
- klient obsahuje prezentaci i aplikační logiku, využívá data z databáze

- oproti třívrstvé architektuře relativně snadný návrh, ale obtížná údržba

Broker

* cílem je, aby distribuovanost byla transparentní = objekt může volat metodu jiného objektu aniž by věděl, že objekt není lokální

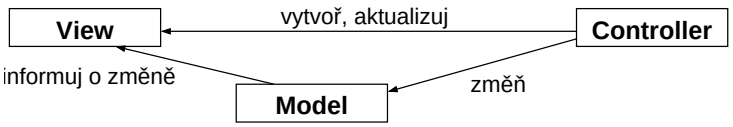


* proxy volá brokera, který zjistí, kde se nachází vzdálený objekt
* příklad: CORBA (Common Object Request Broker Architecture) - je to architektura+infrastruktura pro spolupráci aplikací prostřednictvím sítí

Interaktivní systémy

Architektura Model-View-Controller (MVC)

* architektura doporučovaná a široce používaná pro klasické i webové interaktivní aplikace



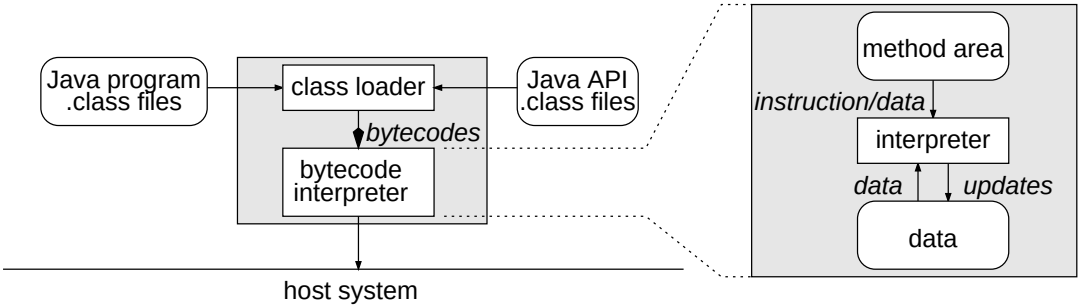
* odděluje funkční vrstvu aplikace (”Model”) od dvou aspektů uživatelského rozhraní (nazývaných ”View” a ”Controller”)
- Model = objekty, které budeme v aplikaci prohlížet a měnit (např. obchodní data)
- View (náhled) = zobrazení modelu; při změně obsahu modelu je vyvolána také změna zobrazení
- Controller = obsluhuje interakci uživatele s modelem
  . na základě změn modelu a vstupů uživatele (stisku kláves, výběru z menu)

```
    vybírá View který se má zobrazit
  . typicky jeden Controller pro množinu příbuzných fcí
* architektura MVC byla již popsána (viz str. 79 přednášek)


Ostatní styly
-------------

Virtuální stroj
...............

* pokud je výpočet buď velmi abstraktní nebo se předpokládá jeho běh na
  různých typech strojů
* příklady:
 - interprety, např. JVM
 - systémy založené na pravidlech, např. expertní systémy
 - procesory příkazových jazyků, např. /bin/sh
```




```
Co se výběry architektonického stylu aplikace týče, nejpříjemnější by bylo
mít katalog stylů (podobně jako existují katalogy pro návrhové vzory), kde by
bylo uvedeno ”pokud je problém podobný X, použijte styl Y”. Na rozdíl od
návrhových vzorů zatím vývoj tak daleko není...


Poznámka (návrh systému shora dolů)

 Alternativa metodičtějších přístupů (např. strukturovaných nebo objektově
 orientovaných metod) je neformální návrh systému shora dolů, kde vycházíme z
 architektonického návrhu:

 * architektonický návrh - identifikujeme a zdokumentujeme podsystémy a
   jejich vztahy
 * abstraktní specifikace podsystémů - pro každý podsystém vytvoříme
   abstraktní specifikaci podsystému a jeho omezení
 * návrh rozhraní podsystémů - pro každý podsystém navrhneme a
   dokumentujeme rozhraní (specifikace rozhraní musí být jednoznačná,
   aby bylo možné podsystém používat bez znalosti jeho vnitřního
   fungování)
 * návrh komponent - služby alokujeme komponentám, navrhneme rozhraní
   komponent
 * podrobně navrhneme a specifikujeme datové struktury
 * podrobně navrhneme a specifikujeme algoritmy
 * po návrhu následuje kódování

[]

Implementace a kódování
=======================

* vstupem implementace je návrh (design) SW systému
 - návrh může být na různé úrovni podrobnosti
 - obvykle popis modulů/tříd a jejich vztahů
 - v některých případech může být kostra programu automaticky vygenerována
   CASE nástrojem

* aktivity a postupy ve fázi implementace:
```

- návrh podprogramů, algoritmů, datových struktur
- psaní a dokumentace kódu, strukturované programování
- testování a ladění modulů, optimalizace

Implementace ”zdola nahoru”

Návrh systému téměř vždy probíhá shora dolů, protože se snažíme velký problém (vytvářený SW systém) rozdělit do tak malých částí, abychom je dokázali postupně vyřešit (tj. nakonec naprogramovat). Tento přístup se často nazývá ”rozděl a panuj”.

Implementaci je naproti tomu většinou vhodné realizovat zdola nahoru:

* začínáme s podprogramy nejnižší úrovně, které ihned testujeme
  - pro testování podprogramu můžeme vytvořit jednoduchý hlavní program, který bude pouze volat testovaný podprogram s příslušnými parametry
* jakmile je podprogram funkční, je třeba ho zdokumentovat
  - v komentáři k podprogramu uvedeme co podprogram dělá, význam vstupních a výstupních argumentů a případné návratové hodnoty
* po vytvoření všech potřebných podprogramů nižší úrovně můžeme napsat podprogram vyšší úrovně a otestovat ho atd.
* nakonec napíšeme hlavní program

Vytvářením systému zdola nahoru postupně zvětšujeme vyjadřovací sílu jazyka, který pro psaní systému používáme. Ve chvíli, kdy nám začnou velké části systému pracovat, stane se to pro nás také motivací k dalšímu programátorskému úsilí.

Poznámka (další vylepšení)

Výsledkem předcházejícího kroku je téměř funkční SW, který je většinou možné ještě podstatně vylepšit:

* pro každý podprogram je vhodné se podívat, jakým způsobem je volán; často je výhodné podprogram rozšířit o činnost, která se provádí vždy před a po volání příslušného podprogramu
* volají-li se dva nebo více podprogramů vždy ve stejném pořadí, je vhodné tyto podprogramy sdružit vytvořením dalšího podprogramu
* vykonává-li podprogram několik typů činností, může být vhodné ho rozdělit na podprogramy pro jednotlivé činnosti.

[]

Vytváření podprogramů z pseudokódu
---

* vlastní kódování je vysoce individuální záležitost, obvykle nebývá podřízeno nějakému SW procesu

* jako příklad metodiky pro kódování jednotlivých podprogramů uvedu vytváření podprogramů z pseudokódu (PDL-to-code process, McConnell 1993)
  - vstupem je podrobný návrh v pseudokódu - ten převezmeme nebo vytvoříme
  - na základě pseudokódu vytváříme kód, z pseudokódu se stanou komentáře

Vytváření pseudokódu
.....

* pseudokód můžeme získat např. jako výstup strukturované analýzy, například:

```
PROCES 3.2.1: ”Vydej stvrzenku”

vytiskni hlavičku stvrzenky
celková-hotovost = 0
DO WHILE v tabulce ”peníze” jsou nezpracované záznamy
    do ”platba” načti záznam z tabulky ”peníze”
    vytiskni obsah záznamu ”platba”
    k celkové hotovosti ”celková-hotovost” přičti částku z ”platba”
vytiskni ”celková-hotovost”
```

* pokud nemáme pseudokód, musíme ho navrhnout - postupujeme od obecného ke konkrétnímu
 - napíšeme komentář popisující účel podprogramu
  . ověříme zda je činnost podprogramu dobře definována, zda zapadá do architektury a zda alespoň nepřímo odpovídá požadavkům
  . definujeme účel podprogramu - musí být definován tak podrobně, aby podprogram bylo možné implementovat
  . popíšeme vstupy a výstupy (včetně globálních proměnných které vytvářený podprogram ovlivní), způsob obsluhy chyb
 - pokud je manipulace s daty podstatnou součástí činnosti, navrhneme hlavní datové struktury
 - podprogram pojmenujeme
 - vytvoříme vysokoúrovňový pseudokód podprogramu
  . pseudokód nebude používat prvky cílového jazyka, bude popisovat záměr
 - vytvořený pseudokód zkontrolujeme
  . kontrola a oprava pseudokódu je podstatně snazší než kontrola a oprava výsledného programu!

* je to opět iterativní proces, chyby je třeba opravovat ihned, jak je naleznete

Transformace do kódu
....

* pseudokód transformujeme do kódu takto:
 - pseudokód postupně zjemňujeme až na úroveň, kdy je snadné doplnit skutečný kód
 - pseudokód změníme v komentáře
 - napíšeme deklaraci podprogramu
 - pod každý komentář doplníme kód
 - kód zkontrolujeme, doplníme zbývající části (chybějící deklarace proměnných, ošetření chyb apod.)

* po ”ruční” kontrole kódu podprogram syntakticky zkontrolujeme překladem
 - při překladu povolíme všechna varování překladače, případná varování vyřešíme

Strukturované programování
----

* strukturované programování = používání řídících konstrukcí tak, aby každý blok kódu měl pouze jeden vstupní a jeden výstupní bod
 - jednoduché, hierarchické struktury pro řízení běhu
 - základní řídící konstrukce: sekvence, větvení, iterace

* základní myšlenka v Dijkstrově článku ”Go To Statement Considered Harmful” (1968, viz http://www.acm.org/classics/oct95/)
 - nestrukturované jazyky typu FORTRAN a BASIC používaly řízení pomocí příkazu GOTO (příkaz GOTO spustí provádění instrukcí od udaného návěští)
 - příklad v jazyce BASIC:

```
10 A=10
20 B=5
30 IF A<B THEN GOTO 60
40 PRINT ”A je větší než B”
50 STOP
60 PRINT ”A je menší než B”
70 END
```

 - problém: lze vytvářet libovolně komplikované konstrukce => ze zápisu programu s GOTO není snadno viditelné dynamické chování programu (běh procesu)
 - řešení: strukturované programování, příklad v jazyce Python:

```
a = 10
b = 5
```

```
        if a < b:
             print ”A je menší než B\n”
        else:
             print ”A je větší než B\n”
```

- strukturované programování zlepší produktivitu oproti nestrukturovanému až o 600%, čitelnost kódu zlepší o cca 30%
- strukturované programování podporují všechny současné procedurální programovací jazyky (obsahují strukturované řídící konstrukce typu ”if-then-else”, ”while”, ”repeat-until”, ”for”)

* řešení výjimečných situací pouze strukturovanými konstrukcemi může být zbytečně komplikované
- proto proto má většina jazyků více způsobů jak opustit řídící konstrukci (např. příkazy ”continue”, ”break”, ”return”, případně ”goto”)
- používat obezřetně pro řešení situací, jako je předčasné opuštění vnořených cyklů při chybě

Moderní jazyky vycházejí z myšlenek strukturovaného programování, ale posouvají je dále (objekty/třídy, výjimky, ...).

❇