KIV/ZSWI 2003/2004
Přednáška 11

Prototypování
=============

* na první přednášce jsem zmiňoval dva druhy prototypů:
 - prototypy ze kterých vyvineme konečný systém
  . vyvineme relativně jednoduchý systém, implementující nejdůležitější
    požadavky zákazníka
  . podle dalších požadavků přizpůsobujeme
  . měly by být vyvíjeny se stejnou kvalitou jako ostatní SW
 - throw-away prototypy - účelem je získání nebo ověření požadavků apod.
  . mají krátkou dobu života, je třeba je rychle vytvořit, snadno změnit

V dalším textu se budu zabývat pouze throw-away prototypy, tj. verzemi SW systému, které mají sloužit pro zjištění dalších informací o systému - nejčastěji v souvislosti se sběrem požadavků nebo v souvislosti s hledáním odpovědí na technické otázky (výkonnost apod.).

Tvorba prototypů by tedy v rámci přednášek logicky patřila ke sběru požadavků, ale z praktických důvodů ji uvádím zde.

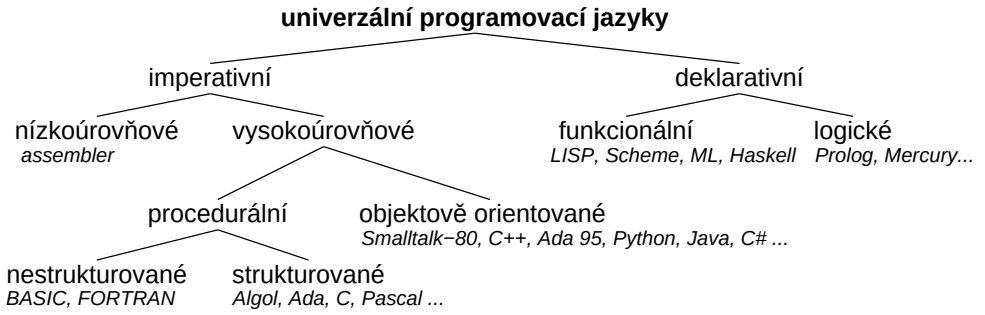
* experimenty ověřily intuitivně zřejmý předpoklad, že prototypy snižují
  množství problémů se specifikací požadavků (Boehm at al. 1984)

* prototypy se vytvářejí v následujících krocích:
 - definice účelu prototypu - například prototyp uživatelského rozhraní,
   prototyp demonstrující užitečnost systému zákazníkovi apod.
 - určení funkčnosti prototypu - co bude a co nebude prototyp obsahovat
  . při tvorbě throw-away prototypu obvykle rezignujeme na mimofunkční
    požadavky, jako je čas odpovědi, paměťová náročnost, spolehlivost
    (omezená kontrola chyb)
 - vytvoření prototypu
  . throw-away prototypy nemusejí být nutně spustitelné; užitečné (a levné)
    jsou i papírové modely uživatelského rozhraní apod.
 - vyhodnocení prototypu - nejdůležitější fáze, zde získáváme díky prototypu
   potřebné informace
  . pro otestování UI je třeba zvolit typického uživatele systému

Rychlé prototypování
----

* pro rychlé prototypování (rapid prototyping) se používají zejména:
 - dynamické vysokoúrovňové programovací jazyky
 - databázové jazyky
 - komponentově orientované programování

Vysokoúrovňové programovací jazyky
....



Programovací jazyky můžeme v zásadě rozdělit do následujících kategorií:

* impreravitní - posloupnost příkazů mění stav programu, jsou odvozeny od
  von Neumannova modelu počítače

* funkcionální - výpočet je zapsán pomocí fcí, které vracejí hodnotu
* logické - programy jsou vyjádřeny pomocí fakt a jejich vztahů

* klasické procedurální jazyky jako Pascal, C/C++, Ada, Java atd.
 - vytváření spolehlivých a rychlých programů
 - deklarace datových typů => specializace fcí, neumožňuje ”náhodnou spolupráci”, nutí programátora provádět explicitní volby brzy ve vývojovém procesu

* pro prototypování se používají především dynamické vysokoúrovňové jazyky
 - obsahují silné mechanismy pro manipulaci dat, správa paměti v režii jazyka (tj. programátor nemusí řešit problémy při alokaci a dealokaci paměti, na rozdíl např. od C kde programátor musí paměť alokovat/uvolňovat explicitně pomocí malloc() a free())
 - dynamické typování (typy argumentů nebo proměnných se nedeklarují)
 - předpoklad ”je lepší mít 100 fcí pracujících nad jednou datovou strukturou než mít 10 fcí pracujících nad 10 datovými strukturami”
 - příklady jazyků: awk, Javascript, Lisp/Scheme, Haslell, Perl, Prolog, Python, Ruby, Smalltalk, Tcl, Visual Basic...

* příklad č. 1: Python
 - objektově orientovaný interpretovaný jazyk, možnost procedurálního programování
 - elegantní a snadno naučitelná syntaxe, sdružování příkazů pomocí odsazování

```
def spocti(a, b):
    if a<b and a*b > b:
        return a
    else:
        return b
```

 - vestavěné vysokoúrovňové typy: seznam, slovník
 - knihovny s mnoha třídami usnadňujícími programování (např. regulární výrazy, komunikace po internetu, XML, GUI...)
 - Jython = v Javě implementovaný Python, dovoluje volání Pythonovského kódu z Javy a naopak (v Javě implementovaná část kódu může volat prototyp vytvořený v Pythonu)
 - nevýhody pro reálné aplikace: slabá správa paměti, pomalý (cca 3x pomalejší než Java)

* příklad č. 2: Haskell98
 - moderní funkcionální jazyk, tj. výpočet je prováděn vyhodnocováním funkcí
 - funkce jsou obvykle definovány množinou rovnic
 - levá strana výrazu obsahuje vzory, které se porovnávají se skutečnými argumenty
 - jako příklad - implementace algoritmu quicksort:

```
qsort []     = []
qsort (x:xs) = qsort less ++ [x] ++ qsort more
     where less = filter (<x)  xs
           more = filter (>=x) xs
```

* další jazyky pro některé typy prototypů:
 - Tcl - často prototypování grafických aplikací (Tk toolkit, který je dnes ale dostupný i z jazyků Python, GUILE atd.)
  . Tcl se většinou používá jako rozšiřovací jazyk pro aplikace v C nebo C++
  . nevýhoda: nemá dobře navržené datové struktury (do verze 8.0 pouze řetězce)
 - awk a Perl - orientovány především na zpracování textových souborů (vestavěné regulární výrazy apod.)
  . nevýhoda: Perl má problematickou syntaxi
 - Lisp (Common Lisp, Scheme) - funkcionální jazyk
  . hlavní datovou strukturou je seznam
  . minimální syntaxe
  . často se používá jako rozšiřovací jazyk aplikací (AutoLisp pro AutoCAD, GUILE pro volně šířené programy, elisp pro Emacs)
 - Prolog - logické programování, někdy simulace databází

* někdy se různé části prototypu vytvářejí v různých jazycích (pro danou část se volí nejvhodnější jazyk)

Databázové programování

* mezi aplikacemi zpracovávajícími data je velká podobnost
  - pro vstup a výstup obvykle množina formulářů nebo tabulek, zadaná data uložena do databáze
  - výběr dat z databáze, vytvoření výstupních sestav

* proto vznikly specializované jazyky pro manipulaci databáze, s nimi související nástroje pro definici UI
* pro nástroje + prostředí se používá pojem ”jazyky čtvrté generace” (fourth-generation languages, 4GLs)
* v 4GL prostředí typicky:
  - databázový dotazovací jazyk, dnes obvykle SQL
    . dotazy obvykle vygenerovány automaticky z formulářů vyplněných uživatelem
  - generátor UI
    . interaktivní definice formulářů pro vstup nebo zobrazování dat, jejich propojení, definice dovolených rozsahů vstupních hodnot
    . většina dnešních 4GL podporuje WWW formuláře
  - generátor výstupních sestav
    . pro definici a vytváření (tiskových) výstupů z informace obsažené v databázi

* nevýhoda: 4GL nejsou zatím nijak standardizované

Komponentově orientované prototypování

* komponentově orientované prototypování má obdobné výhody a nevýhody jako komponentově orientované programování:
  - nemusíme-li některé části prototypu navrhnout a implementovat, snížíme tím čas vývoje
  - na druhou stranu je často nutné přizpůsobit specifikaci tomu, jaké komponenty máme k dispozici

* extrémním případem je využití celých aplikací jako komponent
  - prototyp může být realizován např. jako objekt složeného dokumentu (text, část tabulky, zvukové soubory), které jsou udržovány různými aplikacemi (editor, spreadsheet, program pro přehrávání zvukových souborů)
  - nejpoužívanější mechanismus Microsoft OLE (Object Linking and Embedding)
  - výhoda: prototyp je vytvořen rychle
  - nevýhoda: pokud uživatelé nemají zkušenosti s použitými aplikacemi, může pro ně být matoucí funkčnost která pro prototyp není zapotřebí

* prostředí pro vytváření prototypů obecně obsahuje:
  - komponenty
  - rámec pro sestavování komponent - poskytuje mechanismus řízení + mechanismus pro komunikaci
    . jeden příklad je prostředí tzv. skriptovacích jazyků (scripting languages), mezi které patří jazyky příkazových interpretů, Python, Tcl apod.
    . dalším příkladem jsou obecné rámce pro integraci komponent (CORBA, DCOM, JavaBeans)

Volba programátorských konvencí

Implementaci by měly předcházet minimálně následující kroky:

1. pochopení řešeného problému
2. návrh architektury systému
3. výběr vhodného programovacího jazyka
4. výběr programátorského prostředí, které poskytuje vhodné nástroje
5. volba programátorských konvencí (pokud se nejedná o jednorázový program)

Body (1) a (2) už byly popsány v předchozích přednáškách. Výběr

programovacího jazyka a prostředí byl zmíněn v souvislosti s prototypováním.
Proto se budeme ještě zabývat bodem (5).

Motivace

Jedno ze základních pravidel zní: Zdrojový text programu musí být
srozumitelný pro lidi (ostatní členy týmu).

Jak poznamenává Fowler ve své knížce ”Refactoring”:

 Any fool can write code that a computer can understand. Good programmers
 write code that humans can understand.

* srozumitelnost důležitá zejména pro údržbu - co když je v programu nalezena
  chyba a původní programátor není k dispozici?
* pokud programátor nerozumí cizímu programu, může pro něj být jednodušší
  nesrozumitelný kód napsat znovu než ho převzít => snižuje produktivitu

Poznámka (kód pište pro ”průměrného programátora”)

 Při čtení vašeho kódu by se měl programátor cítit jako při čtení
 nejnudnějšího románu na světě. Funkce každého řádku by měla být zřejmá.
 Pokud kód bude nějak zacházet s proměnnou, měl by si čtenář říci: ”Mně bylo
 předem jasné, že uděláš přesně tohle!”

[]

* problém - pokud čtete kód vytvořený jinými programátory, je často obtížně
  srozumitelný kvůli jejich programátorskému stylu (formátování atd.)
 - programátorský styl (coding style) = soubor pravidel pro psaní zdrojových
   textů, týká se formátování, tvorby názvů, komentářů, předepsaného chování
   SW v určitých situacích (např. při chybě) atd.
 - čtení kódu vytvořeného ve stylu na který nejste zvyklí trvá vždy déle než
   pokud styl znáte
 - nesprávný či nekonzistentní styl => chybná interpretace
 - proto je styl kódu který budou číst další lidé podstatný
   => tým nebo týmy by se měly shodnout na souboru pravidel pro psaní
      zdrojových textů sdíleném celým projektem
  . nedokonalý systém je lepší než žádný
  . vzájemná srozumitelnost přednější než preference jednoho autora
  . na druhou stranu existují studie porovnávající čitelnost některých stylů
 - pozitivní důsledek jednotného stylu: kód bude pro všechny zúčastněné čitelnější

* programátorský styl se týká především
 - odsazování bloků
 - zalamování řádků, mezer a závorek
 - jmenných konvencí
 - komentářů

Poznámka (standardní konvence pro jazyk Java)

 Pokud je to možné, měl by být jednotný styl týmu založený na standardních
 konvencích daného programovacího jazyka. Např. pro jazyk Java jsou
 standardní konvence autorů jazyka zveřejněné na

  http://java.sun.com/docs/codeconv/

 Konvence pro jazyk Java oproti námi uváděným oblastem navíc pokrývají
 pojmenování souborů a jejich organizaci.

[]

Odsazování bloků
....

* správné odsazení musí ukazovat logiku programu
 - cíl: samodokumentující kód

- důsledky nesprávného či nekonzistentního odsazení: chybná interpretace, obtížně udržovatelný kód; například následující kód bude jinak interpretovat člověk a jinak počítač:

```
for (int i = 1; i <= 10; i++)
   leftboot = left[i];
   left[i]  = right[i];
   right[i] = leftboot;
```

* doporučuje se psát pouze jeden příkaz na řádku
* odsazování bloků tak, aby bylo vidět, které příkazy jsou v bloku
- čisté odsazování: lze v Adě, protože každá řídící struktura má svůj ukončovač:

```
začátek_bloku         while Color = Red loop
   příkaz1               příkaz1;
   příkaz2               příkaz2;
konec_bloku           end loop;
```

- simulované čisté odsazování: jako kdyby ”begin” a ”end” byly součástí řídící struktury
. styl ”Kernighan & Ritchie” v C
. de facto standard v C, C++ a Javě, v Pascalu se příliš nepoužívá

```
xxxxxx begin           while (c) {
   příkaz1                příkaz1;
   příkaz2                příkaz2;
end                    }
```

- begin-end hranice: za hranici bloku považujeme ”begin” a ”end”
. podle toho zda ”begin” a ”end” považujeme za součást bloku 3 varianty
. varianta 1 se používá v C i Pascalu, varianta 2 v C (styl GNU), varianta 3 v Pascalu

```
             1.              2.               3.
xxxxxx       while (!done)   while (!done)    while not done
begin        {                 {                begin
   příkaz1      příkaz1;          příkaz1;        příkaz1;
   příkaz2      příkaz2;          příkaz2;        příkaz1;
end          }                 }                end
```

* formátování jednopříkazových bloků
- mělo by být konzistentní s formátováním delších bloků
- v zásadě následující možnosti, každá má své výhody i nevýhody:

```
1.             2.               3.             4.
if (expr)      if (expr) {      if (expr)      if (expr) cmd;
   cmd;           cmd;         {
               }                   cmd;
                                }
```

- ve skupinových projektech se doporučuje styl 2, protože konzistentní s odsazováním podle K&R a pokud budete přidávat příkazy za if, nemůžete zapomenout přidat ”{” a ”}”

* někdy máte potřebu použít ”speciální” formátování; téměř vždy je to příznak špatně navržené metody/podprogramu nebo rozhraní
* například rozhraní pro jazyk C++ pro práci s Okny:

```
HRESULT hr = NějakéJménoFunkce(
          0,                 // Rezervováno, parametr musí být 0
          0,                 // Nedělej něco divného
          THIS|THAT|ANOTHER, // Nastav nějaké speciální příznaky
          BSTR(0),           // Rezervováno, musí vypadat takhle
          &something);       // Objekt který má být vyplněn hodnotou
```

- problémem výše uvedeného rozhraní je příliš mnoho parametrů

Poznámka pro zajímavost (automatické formátování)

* některé týmy nechají programátory používat styl odsazování jaký kdo chce a

```
   použijí např. ”indent -kr -i8 -l80” před přidáním kódu do repositáře
   projektu
 * automatické formátování ale občas vede k méně přehlednému kódu, než je kód
   formátovaný ručně

 * program indent(1) v Linuxu
  - formátuje zdrojové texty jazyka C - mezery, odsazení, umístění
    programových závorek, komentáře
  - předdefinované styly:
   . Kernighan & Ritchie (-kr)
   . Berkeley (-orig)
   . GNU (-gnu)
  - nejdůležitější parametry:
    -iN         ... odsazení příkazů v bloku o N mezer
    -lN         ... délka řádku bude N znaků
    -br         ... složené závorky budou na stejné řádce jako ”if”, ”while” atd.
    -bli0 -bliN ... složené závorky na další řádce odsazené o N znaků
    -diN        ... odsazení jmen proměnných od typu v deklaracích na sloupec N

 * obdobné programy existují i pro jiné jazyky, např. astyle pro C, C++ a
   Javu, Jalopy pro jazyk Java

[]


Zalamování a vkládání prázdných řádků
.....................................

* řádek by neměl být delší než je obvyklá šířka obrazovky (např. 80 znaků pro
  znakový terminál)
* dlouhé řádky je nutné zalomit na logickém místě
 - související věci ponechat na stejném řádku
 - na pokračovacím řádku odsazení podle úrovně ”vnoření” zalamovaného místa

  fd = open(name, O_WRONLY|O_CREAT|O_TRUNC|O_APPEND,
            S_IRUSR|S_IWUSR|S_IRGRP|S_IROTH);

 - některé konvence doporučují zalamovat před operátorem (např. konvence pro
   jazyk Java, GNU konvence), jiné za ním (např. Delphi; je třeba se řídit
   vybranou konvencí)

  if (queue == NULL && foo_this_is_long && bar > win (x, y, z)
      && remaining_condition) ...

  if (queue == NULL && foo_this_is_long && bar > win (x, y, z) &&
      remaining_condition) ...

* prázdným řádkem je vhodné od sebe oddělit logické celky:
 - jednotlivé sekce v programu, podprogramu, třídě nebo metodě (např. lokální
   proměnné od prvního příkazu apod.)
 - skupinu souvisejících příkazů
 - jednotlivé podprogramy nebo metody
 - komentáře

Poznámka (k délce podprogramů)

 Již dlouhá léta se traduje, že podprogramy by neměly přesahovat cca jednu až
 dvě obrazovky (cca 50 řádků). Studie však prokázaly, že do cca 200 řádek
 kódu samotná délka podprogramu neovlivňuje negativně chybovost ani
 srozumitelnost. Podprogram by tedy měl být dlouhý přesně tak, jak je
 zapotřebí.

[]


Používání mezer a závorek
.........................

* K&R doporučují kolem operátorů obvykle zapsat mezeru, např.
```

```
  x = x * (y + 1);

* uvnitř výrazů se doporučuje vkládat závorky a mezery pro lepší
  srozumitelnost
 - tedy nikoli: x = a + b % c * d / e;
     ale např.: x = a + (((b % c) * d) / e);
        nikoli: z = x / 2 + 3 * y;
     ale např.: z = x/2 + 3*y;

* za čárkou a středníkem má následovat mezera, např.

  foobar (x, y, z);  // nikoli: foobar (x,y,z);


Jmenné konvence
...............

* dobré názvy jsou nejdůležitější složkou programátorského stylu
 - příklad chybného pojmenování: x = x - fee(x1, x) + tt;  // co to asi může znamenat?
 - příklad lepšího pojmenování: kredit = kredit - poplatek(zakaznik, kredit) + urok;

* názvy mají dodávat kódu význam
 - z čím větší části programu je název viditelný, tím pečlivěji ho musíme
   zvolit
 - proměnnou, metodu, třídu atd. bychom měli označit srozumitelným názvem,
   který popisuje význam entity kterou reprezentuje
 - například pocetSedadel, pocet_mist_k_stani, jmenoOlympijskehoTymu apod.
  . výše uvedené názvy jsou samy o sobě srozumitelné
  . některá jména jsou ale příliš dlouhá na to, aby byla praktická (z výše
    uvedených poslední dvě)
  . výzkum ukázal, že psychologické optimum je cca 8-20 znaků
 - názvy delší než 20 znaků je vhodné konzistentně zkrátit, např. použít
   srozumitelné prefixy/postfixy (jako jsou anglické Sum, Max, Min, Ptr)

* existují další konvence pro pojmenování řídících proměnných cyklů,
  logických proměnných, konstant, tříd a metod apod.
 - řídící proměnné cyklů - pokud jsou cykly krátké, používají se často
   jednoznakové názvy jako i, j, k; např. v jazyce C:

     for (i = 0; i < BUFFER_SIZE; i++) ...

 - pokud je smyčka delší než několik řádek, má i zde smysl použít popisné
   jméno, např. v Pascalu:

    for TeamIndex := 1 to TeamCount do begin
       for EventIndex := 1 to EventCount [ TeamIndex ] do ...

* logické proměnné - měly by mít pozitivní jméno podmínky
 - např. česky chyba, konec, nalezeno atd.
 - nebo anglicky done, error, found, success (případně isDone, isError,
   isFound, isSuccess)

* výčty a pojmenované konstanty - často velkými písmeny, např.
  VELIKOST_BUFFERU nebo BUFFER_SIZE (v C, Javě), případně
  Okraj.VYSTŘEDIT nebo BorderLayout.CENTER (v Javě)

* dočasné proměnné (temporary variables, často názvy ”tmp”, ”tem”)
 - dočasná proměnná = lokální proměnná, která se uvnitř jednoho podprogramu
   používá postupně pro několik různých účelů
 - jejich výskyt je často varující příznak toho, že programátor problému
   ještě zcela nerozumí
 - dočasným proměnným bychom se měli spíše vyhýbat, pro každý účel bychom
   měli vytvořit samostatnou lokální proměnnou se smysluplným názvem
 - kromě zvýšení čitelnosti to usnadní optimalizaci dobrým překladačům

* v objektově orientovaných jazycích které rozlišují malá a velká písmena se
  často používá konvence pocházející z jazyka Smalltalk:
 - jméno třídy a jméno konstruktoru začíná velkým písmenem, např. Point,
```

```
   Rectangle, Image, ImagePanel apod.
 - jméno metody, proměnné atd. malým písmenem, např. metody addMouseListener(),
   paintComponent(), proměnné point, rectWidth, imageHeight apod.

Poznámka (neformální jazykově závislé konvence - C)

 Pro konkrétní programovací jazyky vznikly další konvence. Pokud budete
 programovat v jazyce C, brzy zjistíte, že (až na výjimky) jsou názvy
 proměnných a funkcí tvořeny malými písmeny a podtržítkem (”pocet_sedadel”,
 ”pocet_mist_k_stani” apod.), pro lokální proměnné se používají krátké názvy
 (”c” a ”ch” pro znaky, ”p” pro ukazatel, ”s” pro řetězec) apod.

[]


Poznámka pro zajímavost (Maďarská notace pro pojmenování identifikátorů)

 * maďarská notace - vznik ve firmě Microsoft (Simonyi asi 1984), dnešní
   rozšíření zejména díky rozhraní MS Windows
 * viz závěr poznámky (mínusů je více než plusů, tj. maďarskou notaci
   NEDOPORUČUJI používat pokud nemusíte)

 * k identifikátoru přidává prefix popisující funkční typ identifikátoru
 * název ”maďarská notace” jednak protože identifikátor vypadá na první pohled
   nesrozumitelně a také protože Simonyi pochází z Maďarska

 * základní myšlenka pojmenovat hodnoty jejich funkčním typem, aby
   programátor nemusel název proměnné a fce dlouho vymýšlet
  - ”funkční typ” dvou proměnných je stejný, pokud je nad oběma možné provést
    stejné operace (tj. nebere se v úvahu pouze reprezentace, ale také význam)
  - například pokud je operace setPosition(x, y) v pořádku zatímco
    setPosition(y, x) je nesmysl, nemají ”celá čísla” x a y stejný funkční typ
  - funkční typy jsou pojmenovány krátkými indikátory, které si volí
    programátor; neměly by to být obecné názvy, protože s nimi jsou potíže
    (např. ”color” je obecný název, ale v aplikaci můžeme mít víc funkčních
    typů pro uchovávání barev; proto raději názvy jako ”co”, ”cl”, ”kl” apod.)
  - například funkční typy pro textový procesor by mohly být:
    ”wn” (okno), ”row” (řádek textu), ”fon” (font), ”f” (boolovská hodnota -
    flag), ”ch” (znak - character) apod.
 * ze základních funkčních typů můžeme konstruovat další typy pomocí prefixů
  - standardní prefixy:
```

prefix	*anglicky*	*význam*
a	array	pole
c	count	počet, např. počet znaků, záznamů apod.
d	difference	rozdíl mezi dvěma proměnnými stejného typu
e	element of an array	prvek pole
g	global variable	globální proměnná
h	handle	popisovač, např. popisovač souboru apod.
i	index	index do pole
m	module-level	proměnná modulu
p	pointer	ukazatel

```
   - např. ”arow” = pole prvků typu ”row”
   - datové struktury mají vlastní typy (název typu by neměl být odvozen z
     prvků datové struktury, protože reprezentace typu se může snadno změnit
     aniž by se změnil jeho význam)
   - datové typy jsou pojmenovány stejnými zkratkami jako funkční typy, tj.
     v programu bychom našli deklarace jako:

       WN wnMain;
       ROW rowFirst;

 * pravidla pro pojmenování hodnot (proměnných): funkční typ volitelně
   následovaný kvalifikátorem
  - např. v názvu ”rowFirst” je ”row” typ a ”First” je kvalifikátor; typ by
```

```
    měl být od kvalifikátoru vhodným způsobem oddělen, např. v C velké písmeno
  - příklady identifikátorů v maďarské notaci:

        ch          datová struktura pro znak
        cch         počet znaků
        ach         pole znaků
        achInsert   pole znaků pro vložení
        echInsert   prvek pole znaků pro vložení
        hwn         popisovač okna (typ "okno" jsme si pojmenovali "wn", viz výše)

 * výhody
  - standardní konvence (to je užitečné samo o sobě)
  - snadná tvorba názvů

 * nevýhody:
  - hlavní nevýhoda - vytvořená jména nejsou vždy informativní (např. "hwn"
    neříká o jaký typ okna se jedná - nevím zda je to hlavní okno, help, menu apod.)
  - spojuje význam dat s jejich reprezentací
  - mnoho uživatelů maďarské notace používá místo funkčních typů základní
    typy programovacích jazyků (int, long apod.) - což je zbytečné (překladač
    základní typ dat zná)

[]



Komentáře
.........

* doplňují, co v kódu není vidět (sémantika, odkazy, zdroje informací,
  záměr/účel, nestandardní operace)
 - komentování modulů, proměnných, podprogramů, bloků, řádek kódu

* komentování modulů
 - každý modul by měl začínat komentářem, který stručně popíše účel modulu
   (případně jiného zdrojového souboru aplikace)
 - další část komentáře obvykle popisuje copyright apod.
 - např.:

  /* farm.c -- obsahuje funkce pro zakládání a rušení farem.
   *
   * Copyright 2003  Lukáš Petrlík  <luki@kiv.zcu.cz>
   */

* komentování deklarací proměnných, zejména globálních; např.

   wrkstate *worker;         /* Seznam obsahující stav všech pracovníků. */
   farminfo *farm;           /* Seznam obsahující popis všech farem. */
   int nfarms;               /* Celkový počet farem spravovaných aplikací. */

* komentování podprogramů
 - činnost podprogramu
 - argumenty, význam hodnot které mohou nabývat, účel argumentů
 - význam případné návratové hodnoty

* komentování bloků kódu apod.
 - komentář by měl být odsazen stejně jako komentovaný kód
 - před nebo za komentářem je vhodné vynechat prázdný řádek, abychom komentář
   snadno našli

        foobar(buff);

        /*
         * POSIX.1 specifies that if the O_APPEND flag is set, no intervening
         * file modification operation is allowed (see POSIX.1:1996, section
         * 6.4.2.2, lines 226-228).
         */
        write(fd, buff, strlen(buff));
```

```
* komentáře nepřehánět, vysvětlit především co a proč program dělá

Poznámka (nástroje pro kontrolu programátorských konvencí)

 Pro kontrolu programátorských konvencí existují nástroje. Například pro
 programy v Javě existuje nástroj Checkstyle, který umí kontrolovat všechny
 zde uvedené konvence, včetně jmenných konvencí.

[]

Nástroje pro dokumentaci: javadoc
---------------------------------

* javadoc je nástroj pro jazyk Java
 - generuje HTML dokumentaci včetně křížových referencí na základě speciálně
   formátovaných komentářů (”doc comments”)
 - komentář začíná /** a končí */ (pokud více řádků, mohou začínat hvězdičkou)
 - komentáře musí být umístěny před dokumentovanou třídou, metodou apod.
 - v komentářích možno použít HTML (kromě H1-H3)
 - klíčová slova pro odkazy apod. na úrovni modulu a třídy

* na úrovni modulu a třídy:

   @author jméno       // bere se v úvahu pouze pokud zadán parametr -author
   @version verze      // bere se v úvahu pouze při zadaném parametru -version
   @see JménoTřídy     // generuje ”See Also: JménoTřídy”

* na úrovni atributů a metod

   @param název popis                // popis parametru
   @return popis_hodnoty             // popis návratové hodnoty
   @exception modul.JménoTřídy popis  // totéž co @throws - generuje ”Throws”
   @deprecated náhradní_řešení       // lze také na úrovni třídy nebo rozhraní

* příklad:

  /**
   * Toto je komentář pro třídu <code>Hello</code>.
   *
   * @author  Lukáš Petrlík
   * @version 1.0 (25.4.2003)
   * @see     java.lang.Object
   */
  public class Hello {
      /** Příklad komentáře atributu. */
      int x = 0;

      /** Popis metody <code>main</code>. */
      public static void main(String[] args) {
          System.out.println(”Hello World!”);
      }
  }

* popis javadoc naleznete na http://java.sun.com/products/jdk/javadoc/
* podobné nástroje jsou dostupné i pro další jazyky nebo jsou jazykově
  nezávislé, např. nástroje Doc++, RoboDoc, atd.
 - přehled najdete na http://www.codeassets.com/doc_tools.htm

Poznámka (nástroj Checkstyle)

 Výše zmíněný nástroj Checkstyle umí kontrolovat i Javadoc komentáře.

[]

Optimalizace programu
---------------------

* program by měl být tak efektivní, jak je od něj požadováno, nikoli tak, jak
  je to technicky možné
```

```
 - přílišné zaměření na výkonnost zhoršuje čitelnost a udržovatelnost kódu
 - optimalizace je drahá činnost, tj. je třeba důkladně zvážit zda je nutná
 - při optimalizaci je riziko zanesení chyb do funkčního kódu

* na výkonnost se můžeme zaměřit na dvou úrovních: strategické a taktické
 - strategie:
  . nejde změnit/vyladit design?
  . můžeme použít jiný algoritmus, změnit datové struktury?
 - taktika - optimalizace kódu:
  . cca 20% programu konzumuje 80% času (Boehm 1987, podobně Bentley 1988)
  . nutné optimalizovat pouze kritická místa programu
  . kritická místa dnes není možné určit bez měření, protože moderní
    překladače provádějí poměrně agresivní optimalizaci

* nástroje - profilery - zjištění času stráveného v jednotlivých částech
 - např. volně šířené nástroje gcc a gprof:

   $ gcc -pg program.c      # přeloží program.c a vloží kód pro profilování
   $ ./a.out                # přeložený program spustíme, vytváří gmon.out
   $ gprof a.out gmon.out   # gprof vypíše statistiky (doby strávené ve fcích apod.)

* nejčastější zdroje neefektivity:
 - přístup k souborům
 - podprogramy pro formátovaný tisk
 - operace v pohyblivé řádové čárce
 - stránkování (bude popsáno v předmětu KIV/ZOS)
 - volání služeb operačního systému (taktéž viz KIV/ZOS).
```

❇