Implementace a kódování

- vstupem implementace je návrh (design) SW systému
- návrh může být na různé úrovni podrobnosti
- kostra programu může být automaticky vygenerována CASE nástrojem

- aktivity a postupy ve fázi implementace:
  - návrh podprogramů, algoritmů, datových struktur (v případě, že návrh není příliš podrobný)
  - psaní a dokumentace kódu
  - a poté testování a ladění modulů, optimalizace

Implementace "zdola nahoru"

- návrh systému téměř vždy probíhá shora dolů, snažíme se velký problém (vytvářený SW systém) rozdělit malých částí, abychom je dokázali postupně vyřešit (tj. naprogramovat). Tento přístup se často nazývá "rozděl a panuj".

- implementaci je naproti tomu většinou probíhá 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: Výsledkem předcházejícího kroku je téměř funkční SW, který je většinou možné ještě podstatně vylepšit (pokud už nemáme předepsáno z návrhu):

- 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 hodné 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

- příklad metodiky pro kódování jednotlivých podprogramů – 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 získáme např. jako výstup strukturované analýzy:

```
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

- pseudokód transformujeme do kódu - postupně zjemňujeme až na úroveň, kdy je snadné doplnit skutečný kód
- pseudokód změníme v komentáře …

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

- nestrukturované jazyky typu FORTRAN a BASIC používaly řízení pomocí příkazu GOTO (příkaz GOTO spustí provádění instrukcí od udaného návěští)

- problém: lze vytvářet libovolně komplikované konstrukce => ze zápisu programu s GOTO není snadno viditelné dynamické chování programu (běh procesu)

- 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 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, ...).

Volba programátorských konvencí

- motivace - jedno ze základních pravidel zní: Zdrojový text programu musí být srozumitelný nejen pro počítač, ale především pro lidi (ostatní členy týmu) – a dohlédnout na to musí vedoucí projektu

- Fowler a knížka "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")

- možný přístup - při čtení vašeho kódu by se měl programátor cítit jako při čtení románu, ve kterém perfektně rozumí současnému ději a zároveň dokáže předvídat děj další. Funkce každého řádku by měla být zřejmá.
- např. pokud kód bude nějak zacházet s proměnnou, měl by si čtenář říci: "Mně bylo předem jasné, že uděláš přesně tohle!"

- a proč programátoři často nepíšou srozumitelný kód – protože je to otravná práce navíc, která se nikomu (téměř) nechce dělat

- a na co se vymlouvají
    - že jsou schopni napsat kód rychlejší – to už většinou neplatí (viz později)
    - nedostatek času – z velmi krátkého pohledu to tak je, z pohledu dlouhodobějšího je to přesně naopak
    - ochrana vlastních myšlenek – bez komentáře

- za nejlepší dokumentaci se považuje komentovaný a čitelný zdrojový kód
  - komentář – co program dělá (srovnejte s analýzou…)
  - samotný kód – jak to dělá (srovnejte s návrhem…)
  - čitelný zápis kódu šetří tvorbu dokumentace

- „Teď to napíšu nečitelně, protože nemám čas, a až ten čas mít budu, tak to přepíšu do čitelné podoby“ – bohužel nefunguje (proč?) -> kód je třeba psát čitelně hned (dokonce považováno za důležitější než abyste psali úplně bez chyb)

- 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 pokrývají i pojmenování souborů a jejich organizaci

Základní pravidla zápisu čitelného kódu

- kód musí být dobře a snadno čitelný i zapisovatelný
- vlastní pravidla jsou založena na konvencích jazyka nebo/a vývojového prostředí
- systém pravidel musí být dostatečně jednoduchý
- pravidla lze v odůvodněných případech porušit
- odsazení řádků odpovídá vnoření bloků kódu
- 1 řádek = 1 příkaz
- volné řádky oddělují logické celky

- existuje týmová shoda pro zápis standardních programových konstrukcí
- v jedné oblasti platnosti (třída, metoda) nepoužívejte identifikátory, které vypadají podobně (především začátky identifikátorů)
- používejte editory s barevným rozlišením syntaxe
- identifikátor co nejvýstižněji popisuje sémantiku objektu, ale také nesmí být příliš dlouhý
- používejte pouze jeden jazyk při pojmenování objektů (česky x anglicky)
- stanovte a dodržujte jasná pravidla pro velikost písmen u jazyků, které rozlišují velikost písmen
- nepoužívejte prefixy u názvu proměnných (případně velmi omezeně)

Odsazování bloků

- správné odsazení musí ukazovat logiku programu
- cíl: samodokumentující se 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: 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

```
xxxxxx
begin
  příkaz1
  příkaz2
end
```

1.

```
while (!done)
{
    příkaz1;
    příkaz2;
}
```

2.

```
while (!done)
 {
    příkaz1;
    příkaz2;
 }
```

3.

```
while not done
  begin
  příkaz1;
  příkaz1;
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 často doporučován styl 2, protože je konzistentní s odsazováním podle K&R, a pokud budete přidávat příkazy za if, nemůžete zapomenout přidat "{" a "}"

- potřeba použít "speciální" formátování; téměř vždy příznak špatně navržené metody/podprogramu nebo rozhraní

- 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, bohužel 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) ¨

- 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)

- několik různých tvrzení:

- podprogramy by neměly přesahovat cca jednu až dvě obrazovky (cca 50 řádků).
- do cca 200 řádek kódu samotná délka podprogramu neovlivňuje negativně chybovost ani srozumitelnost
- podprogram by 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

- ne: `x = a + b % c * d / e;`
  ale např.: `x = a + (((b % c) * d) / e);`

- ne: `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

- jak ne: x = x - fee(x1, x) + tt; // co to asi může znamenat?
- lépe: kredit = kredit - poplatek(zakaznik, kredit) + urok;

- názvy mají dodávat kódu význam, tedy proměnnou, metodu, třídu atd. označujeme srozumitelným názvem, který popisuje význam entity, kterou reprezentuje, např. pocetSedadel, jmenoTymu, pocet_mist_k_stani, apod.

- z čím větší části programu je název viditelný, tím pečlivěji ho musíme zvolit

- některá jména jsou ale příliš dlouhá na to, aby byla praktická (např. pocet_mist_k_stani) - 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)

- 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") = lokální proměnné, které se uvnitř jednoho podprogramu používají postupně pro několik různých účelů
  - jejich výskyt často signalizuje, že programátor problému ještě zcela nerozumí
  - měli bychom se jim spíše vyhýbat, pro každý účel vytvoříme 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.

- v jazyce C, 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.

Maďarská notace pro pojmenování identifikátorů

- maďarská notace - vznik ve firmě Microsoft (Simonyi asi 1984), dnešní rozšíření zejména díky rozhraní MS Windows
- název "maďarská notace" jednak protože identifikátor vypadá na první pohled nesrozumitelně a také protože Simonyi pochází z Maďarska

- k identifikátoru přidává prefix popisující funkční typ identifikátoru - 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), např. 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 (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:
  - 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 a dokumentace

- proč? – pokud je program kvalitní, bude se určitě dále rozvíjet, či přepracovávat – zapomeneme, proč jsme věci naprogramovali zrovna takhle, či mnohé úpravy bude dělat někdo jiný – důraz na čitelnost a dokumentaci programu

- čitelnost programu zahrnuje
  - výstižně pojmenované identifikátory
  - komentáře – doplňují, co v kódu není vidět (účel tříd a rozhraní, význam atributů, funkce metod, význam jednotlivých parametrů, bloků, řádek kódu, nestandardních úseků kódu, odkazy, zdroje informací)

- komentář – usnadňuje čtení kódu a nezahlcuje toho, kdo kód píše

Základní pravidla pro psaní komentářů

- neduplikují činnost kódu
- popisují účel kódu, to, nad čím by člověk musel přemýšlet
- je nezbytné komentovat
    - složitější datové typy (třídy, struktury, stromy, tabulky)
    - složitější algoritmy, obzvláště rekurzivní (např. sorty)
    - složitější cykly (abychom pochopili, jak přesně cyklus pracuje, musíme znát invariant cyklu)
    - rekurzi
    - implicitně uvažované předpoklady
    - hlavičky modulů (balíků, komponent)
        - autor, účel
    - hlavičky metod (funkcí, procedur)
        - sémantika, parametry, návratové hodnoty
    - změny v kódu – důvod, autor, datum změny

- v Javě (nejen) 3 druhy komentářů
  - řádkový (ŘK) – začíná dvojicí znaků //, končí koncem řádku – obvykle poznámka ke kódu v těle třídy nebo konstruktoru,
  - obecný (OK) – začíná dvojicí znaků /*, končí dvojicí znaků */
  - dokumentační (DK) – speciální typ obecného komentáře, začíná trojicí znaků /** - jsou zpracovávány javadocem

- pokračovací řádky obecných a dokumentačních komentářů je dobré, ale nepovinné začínat hvězdičkami – přehlednější kód

- komentáře ve zdrojovém kódu třídy
  - účel třídy – DK
  - význam public atributů – DK
  - význam private atributů – ŘK nebo DK
  - činnost metody, argumenty, význam hodnot, které mohou nabývat, účel argumentů, význam případné návratové hodnoty – DK
  - oddělení jednotlivých sekcí těla třídy – ŘK

- vkládání prázdných řádků pro přehlednost – nejčastěji mezi definice metod

Uspořádání prvků v těle třídy

- záleží i na osobních preferencích

- zajímavý a přehledný způsob – R. Pecinovský: Myslíme objektově v jazyku Java 5.0 – viz příklad

  - nejprve atributy, poté metody
  - nejprve statické členy, pak dynamické členy (platí pro atributy i metody)
  - konstanty před proměnnými
  - konstruktory mezi statickými a nestatickými metodami
  - ke konstruktorům i tovární metody
  - nejprve metody tzv. getry a setry, pak ostatní

Poznámka: Ani s komentáři se to nesmí přehnat.

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

- nástroj pro jazyk Java
- generuje HTML dokumentaci (pro jednotlivé třídy i celý projekt najednou) včetně křížových referencí na základě speciálně formátovaných komentářů ("doc comments") – uložena obvykle do *doc* podadresáře v adresáři aktuálního projektu

- obsahuje:
  - popis vlastností a použití dané třídy (z komentáře před hlavičkou třídy)
  - tabulky se základními informacemi o atributech, konstruktorech a metodách, abecední řazení
  - úplné komentáře k atributům, konstruktorům a metodám, abecední nebo programové řazení
  - pouze veřejně deklarované členy (dá se však vygenerovat i dokumentace k privátním členům)

- komentáře musí být umístěny před dokumentovanou třídou, metodou apod.
- v komentářích možno použít HTML (kromě H1-H3)
- dokumentaci můžeme vygenerovat, i když projekt (nebo jeho část) ještě není přeložitelná – může pak sloužit jako návod pro implementaci

- klíčová slova na úrovni 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"

- klíčová slova na úrovni atributů a metod

- @param název popis // popis parametru
- @return popis_hodnoty // popis návratové hodnoty
- @exception modul.JménoTřídy popis //totéž co @throws - generuje důvod, kvůli kterému metoda vyhazuje výjimku
- @deprecated náhradní_řešení // lze také na úrovni třídy nebo rozhraní

- příklad:

```
/**
 * Toto je komentář pro třídu <code>Hello World</code>.
 *
 * @author  Adam Votruba
 * @version 1.1 (13.3.2006)
 * @see     java.lang.Object
 */
public class Hello {
    /** Popis atributu. */
    int x = 0;

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

- popis javadoc na http://java.sun.com/products/jdk/javadoc/
- podobné nástroje jsou dostupné i pro další jazyky nebo jsou jazykově nezávislé - přehled na http://www.codeassets.com/doc_tools.htm

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
- některá omezení dána už architekturou systému (výkonnost databázového serveru, propustnost komunikačních linek) – zrychlení kódu se navenek neprojeví
- optimalizace je drahá činnost, tj. je třeba důkladně zvážit, zda je nutná
- při optimalizaci je riziko zanesení chyb do funkčního kódu
- zvýšení výkonnosti aplikace - spíše změny v návrhu než v kódu

Paralelní zpracování instrukcí
- cache procesoru
- zpracování několika instrukcí najednou (instrukce musí být vzájemně nezávislé)

- rychlost zpracování závisí na sekvenci instrukcí, dostupnost dat v cache – těžko se odhaduje bez detailní znalosti

- 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
    - nutné optimalizovat pouze kritická místa programu
    - kritická místa dnes není možné určit bez měření, protože moderní překladače provádějí poměrně agresivní optimalizaci (dokáží přeskupit celou sekvenci kódu)

optimalizátor – „má raději“ více jednoduchých příkazů než-li příkazy složité – může pak vybírat z více variant (nejčitelnější kód je pak i nejrychlejší) -> nemá smysl přemýšlet nad zápisem kódu z hlediska rychlosti

- nástroje - profilery - zjištění času stráveného v jednotlivých částech kódu
- např. v gcc a gprog

```
$ 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
```

- kde má smysl optimalizovat:

  - práce s diskem -> využití cache
  - databázové operace -> indexy, minimalizace databázových dotazů, uložené procedury
  - volba algoritmů -> typicky algoritmy třídění pro malá a velká data

- optimalizace nejčastěji se opakujících bloků kódu (profiler) -> experimenty (jednotlivě a nutné okamžité testování za stejných podmínek), neobvyklé konstrukce a povolená porušení pravidel psaní kódu
- cykly – opakované nastavování hodnoty proměnné (změna se provádí jinde), ukládání dat na disk ad.

Další poznámky a postřehy

- vývoj programovacích jazyků
  - snaha umožnit programátorům, aby psali srozumitelné programy, základní techniky (přiřazovací příkaz, cykly apod.) se desítky let nemění
  - roste složitost aplikací – techniky pro správu kódu

- každá funkčnost musí být naprogramována pouze jednou

- v kódu se nesmí objevovat magická čísla
  - pozor na výčtový typ

- dobře napsaný podprogram
  - jedna funkčnost
  - málo parametrů
    - v případě potřeby složitější datový typ (vstupní i výstupní – Transport object) nebo přetížení (varianta s největším množstvím parametrů identifikuje hlavní funkčnost, ostatní volají hlavní variantu, dosazují předdefinované hodnoty na místa chybějících parametrů)

- podprogram komunikuje s okolím přes parametry

- ošetření chybových stavů
  - podprogram musí definovat, co se stane, pokud dojde k chybě (očekávané i neočekávané)
    - signalizace návratovou hodnotou
    - mechanismus výjimek

- datové struktury
  - Reference a ukazatele
  - Uvolňování paměti
  - Přetečení bufferů

- Objektový model, hierarchie tříd
- Moduly, třídy
- Globální proměnné
- Programátorské triky
- Externí data

Balíčky (packages) v Javě

- části programu (většího řádu než třídy) obsahující související třídy

- hierarchické uspořádání

- úzus – třídy (soubory) jednoho balíčku se ukládají do jednoho adresáře

- zařazení třídy do balíčku (nazev_balicku = adresář, ve kterém je soubor uložen)

package nazev_balicku

- konvence – název balíčku pouze malá písmena

- třída může být v jediném balíčku

- balíčky mohou obsahovat podbalíčky (stromová struktura) –

  syntaxe *rodičovskýbalíček.podbalíček*

- potom tzv. plné názvy tříd (název balíčku+třídy)

- příkaz *import* (na začátku souboru, před může být jen *package*)
  - třídy
  - všech tříd balíčku - *
  - podbalíček není součástí balíčku¨

- statický import - *import static* – import statického členu (členů) – použití ?

- balíček java.lang – importován implicitně
- přístupová práva v rámci balíčku