

## Osnova předmětu IB053

### Část A: Efektivita práce při tvorbě programu

1. Snížení chybovosti při tvorbě programu
2. Snížení doby potřebné k odstraňování chyb
3. Využití dříve napsaných částí programu
4. Nezávislost programu na pozdějších úpravách
5. Přenositelnost do jiných prostředí

### Část B: Efektivita programu

1. Mechanismus přístupu k datům
2. Implementace programových struktur
3. Rozdíl v interpretovaných a překládaných jazycích

## Historie předmětu IB053

Autor předmětu se od roku 1990 komerčně zabývá tvorbou software a z této praxe čerpá většinu zkušeností použitých v předmětu Metody efektivního programování. Protože původně na fakultě informatiky vyučoval programování v jazycích C/C++, využívá též zkušeností získaných na cvičeních k tomuto předmětu, protože zápočtové programy některých studentů jsou bohatou studnicí příkladů, jak by se nemělo programovat. Naštěstí těchto studentů postupně velmi ubývá. Výuka C/C++ tedy byla postupně obohacována o doporučení či metody jak (ne)programovat. Po několika letech autor přestal zcela vyučovat programování konkrétního jazyka a začal učit předmět Metody efektivního programování. Obsah tohoto předmětu se také postupně vyvíjí s novými zkušenostmi z praxe, ale též od studentů, kteří se na přednáškách zapojují do diskusí a vnášejí tak na problematiku nový pohled.

## Cvičení

- Srovnání efektivit C++ Win/Linux, Java, C#, PHP na shodném HW (dvě úlohy: výpočet nad daty v paměti, zpracování textového souboru)
- Vytvoření týmového díla (C++ Win/Linux, Java) – každý tým bude mít jednoho koordinátora (rozděluje práci, organizuje týmové porady, analýzu), účast v týmu = podmínka zápočtu

## Organizační pokyny

- rezervovány jsou 3 hodiny namísto dvou, ale počet hodin za semestr je vždy počet týdnů semestru krát dvě a tedy výuka v některých týdnech se neuskuteční
- rozdělení přednášek a cvičení určuje vyučující průběžně

## Část A: Efektivita práce při tvorbě programu

K čemu efektivita: Čas = peníze

### 1. Snížení chybovosti při tvorbě programu

- Důkladná analýza (*shora dolů*)
  - funkční (*rozdělení programu na moduly, moduly na třídy + interface, třídy na metody*)
  - datová (*objekty, DB-tabulky*)
- Dostatek času (*podvědomí pracuje za nás*)
- Práce v týmu (*Brainstorming*)
- Programování (*zdola nahoru nebo spíše odprostřed dolů a pak odprostřed nahoru → aby se dalo co nejdříve ladit*)
- Dobrá znalost programovacího jazyka (*chyby z neznalosti jazyka se špatně hledají*)
- Dobrá znalost vývojového prostředí a využívání jeho možností
  - refaktoring – přejmenování, přesunutí
  - Prostředky pro sestavování aplikace (MAKE, ANT, MAVEN)
- Soustředěnost při práci (*kolegové v kanceláři, hudba, TV, maily, prohlížení webu*)

## 2. Snížení doby potřebné k odstraňování chyb (ladění) (Analýza cizího programu, analýza vlastního programu s časovým odstupem)

(Člověk by neměl psát programy tak, aby se stal obětí své vlastní lenosti!)

- Čitelný zápis programu bez „hutných“ pasáží s mnoha vedlejšími efekty  

```
for (int i = 0; suma(pocet = index++) < max; new_line(i++) )  
{...}  
v jazyce C: int i; ... if (!i) { ... }
```
- složité logické podmínky rozdělit na jednodušší
- zamezit negaci negovaného
- boolean metody vracející pozitivní stav (hasRestriction x hasNoRestriction)
- nesnažit se za každou cenu zjednodušit logický výraz de Morganovými pravidly - zápis pak neodpovídá logice věci

např.  $!(a \ \&\& \ !b) \rightarrow !a \ || \ b$   
v zadání je: nesmí platit, že (**a** a současně neplatí **b**)  
špatně by se pak kontrolovalo, že  $!a \ || \ b$  odpovídá zadání

- Výpočty prováděné překladačem  

```
2 * 3.14  
6.28  
#define          PO CET_ZNAKU_ADRESY (40 + 30 + 5)  
#define          PO CET_ZNAKU_ADRESY 75
```
- Symbolické konstanty (*správná interpretace, hromadná změna; nevyužívat k jiným účelům*)  

```
jmeno[PO CET_ZNAKU_JMENA + 1];  
prijmeni[PO CET_ZNAKU_JMENA + 1]; - použití k nesprávnému účelu  
jmeno[PO CET_ZNAKU + 1]; - nejednoznačná konstanta
```
- Parametry procedur, proměnné (*výstižné, stručné, jednoznačné, pravdivé názvy, používat pouze k jednomu účelu*)  

```
promennaObsahujiciPocetPrvku  
pocetPrvku  
pocet  
poc  
p
```

p q r x y i j k – za určitých okolností lze (kdy?)

*příklady použití velkých a malých písmen:*

```
proměnná;  
Metoda();  
PublicMetoda();  
privateMetoda();  
KONSTANTA
```

```
viceslovnáProměnná x viceslovná_proměnná  
iPocet, fVelikost, sJméno
```

- Globální proměnné (*nepřehlednost*)
- Komentáře  
(*stručné, jasné, pravdivé, neplýtvat, nekomentovat jasné věci (i++; apod.)*),  

```
a = findMax(array); // najdeme největší prvek  
a = findMax(array); // tady v tomto místě musíme najít  
                        // prvek, který je v tom poli největší  
a = findMax(array); // největší  
a = findMax(array); // vytiskneme největší prvek  
  
i++;           // i zvýšíme o 1  
i++;           // jak by zněl smysluplný komentář?
```

jak se dostane do programu nepravdivý komentář ?

co komentovat:

- obtížné pasáže
- jednotlivé bloky kódu provádějící ucelenou činnost
- interface (hlavičky metod, rozhraní tříd, modulu)
- Odstraňovat příčiny chyb, nikoli eliminovat jejich důsledky
- Použití Exceptions → zpřehlednění kódu  
(ale používat opravdu jen k řešení výjimečného stavu, nikoli jako řešení stavu, který je sice význačný, ale žádoucí, např. tabulka neobsahuje žádnou větu)
- Styl zápisu (*jednotný – graficky, odlišení proměnných, metod, konstant*)  
(často je řízen IDE - lze konfigurovat)  
(zejména při práci v týmu je potřeba jednotnost stylu zápisu)

grafický styl bloku (odsazování):

záhlaví {   }	záhlaví {   }	záhlaví {   }	záhlaví { }
------------------------	---------------------------	---------------------------	-------------

Mezery (každá neobvyklost musí mít své opodstatnění):

```
for (int i=0; i<5; i++) třída.metoda(x);
```

Přechody na nový řádek

příkaz 1;  příkaz 2;  příkaz 3;	příkaz 1; příkaz 2;  příkaz 3;
---	--------------------------------------

- Práce v týmu (je potřeba tým umět vést a je potřeba umět být členem týmu) - síla řetězu závisí na každém článku

v praxi: nepracuje-li jeden člen kvalitně, snižuje to kvalitu práce i dalších členů, např. když dám týmu neodladěné zdroj. texty funkce, kterou volá funkce jiného člena týmu, tak on pak nebude vědět, jestli je chyba v jeho kódu, zejména ve způsobu volání té funkce a nebo chyba až ve funkci - ke kvalifikovanému rozhodnutí je často potřeba jít až do zdroj. textu té funkce nebo v debuggeru funkci prokrokovat

### 3. Využití dříve napsaných částí programů

- vhodné rozčlenění programu (třídy/modules aplikačně nezávislé, částečně závislé, závislé)
- při psaní jakéhokoli programu je potřeba neustále myslet na to, jestli se právě psaný kód nebude ještě někdy hodit a tomu přizpůsobit způsob psaní
- obecný algoritmus vždy vyčlenit z méně obecného (aplikačně závislého kódu) – zejména u často používané funkčnosti lišící se jen např. zpracováváním daty (nejhorší je hotový zdrojový kód okopírovat a mírně upravit)
- neslučovat do třídy/module nesouvisející funkčnost
- OOP: používat abstraktní metody, interface
- tvořit knihovny (zdokumentované)
- využití částí programů napsaných cizími osobami (získané z internetu, např. [www.sourceforge.net](http://www.sourceforge.net), [apache.org](http://apache.org))
- využití cizích knihoven (volně šířených i placených)
  - např. pro konfigurační soubory (vyskytuje se prakticky v každém programu)
  - pro uložení dat (nejlépe SQLite, rovněž prakticky každý program potřebuje ukládat data)
  - XML (v Javě JAXB)
- zdrojové texty nekopírovat do jiných projektů, ale sdílet v rámci možností, které dává použitý jazyk a vývojové prostředí
- při úpravách kódu, který je sdílen mezi více projekty, dodržovat zpětnou kompatibilitu

## 4. Nezávislost programu na pozdějších úpravách

- symbolické konstanty (*zde za účelem parametrizace programu*)
- dobře rozdělený program do modulů a funkcí nebo tříd  
(*změna v jedné části programu se nesmí projevit jako vedlejší efekt v úplně jiné části programu*)
- prozíravost; odhad, co se reálně může v zadání později změnit  
(*nedůvěra je vysoce pozitivní vlastnost tvůrce software*)

## 5. Přenositelnost programu do jiných prostředí

- orientace na standardní prvky jazyka  
(v Javě je to jedno – pozor pouze na verze Javy, v jiných jazycích existuje spousta kompilátorů, které více nebo méně dodržují standardy, v C++ např. pozor na Templates – nejsou všude)
- použití pouze standardních knihoven, tříd a funkcí  
(v Javě je to – až na verze Javy - jedno)
- vyčlenit platformově (co to je? → Windows/Linux, KDE/GNOME atd.) nebo jinak závislé části  
(podmíněný překlad (co to je?, v Javě není – řeší se pomocí interface a tříd implementující daný interface, tyto třídy se případně umístí do různých jar-souborů) – konstanty pro podmíněný překlad v jednom souboru nebo případně v projektu – závisí na vývojovém prostředí, moduly ve více verzích, rozdělené dialogy: návrhový vzor Data, View, Controller)

Server

Klient

**View** (layout)

platformově závislá část

**Data**

**Controller** (chování – reakce na události)

platformově nezáv. část

- znalost implementačních detailů  
(soubor s binárními daty – implementace reálných čísel nemusí být vždy stejná)
- spolupráce mezi částmi programů napsaných v různých prostředích/jazycích (kompilovaných) – další implementační detaily  
Programm – DLL (pozdní vazba – řeší OS)  
Modul – Modul (volání – řeší kompilátor)
  - pořadí parametrů
  - implementace datových typů (reálná čísla)
  - umístění prvků ve struktuře
- používat konstrukce nezávislé na implementačních detailech nebo případně závislé konstrukce předdefinovat na jednom místě (zač. modulu, hlavičkový soubor) (počet bitů v int, short, long apod.)

~7



## 6. Další doporučení

- Pořádek ve zdrojovém textu i po dokončení → využitím refaktoringu (pokud vývojové prostředí umožňuje) přejmenovat proměnné, metody i třídy, přesunout metody mezi třídami nebo dokonce moduly nebo projekty
- Jakmile něco nefunguje podle očekávání, hledat pomoc na internetu vhodně formulovaným dotazem na google (málokdy se stane, že jsme první, kdo danou situaci řeší). Vysoká pravděpodobnost nalezení řešení je na [stackoverflow.com](https://stackoverflow.com).
- Použití sofistikovaných prostředků k úpravě zdrojových textů (editor Sublime)

- Ve zdrojovém textu využívat prostředky k tomu, k čemu jsou určeny např. For-cyklus:

```
for ( inicializace proměnné cyklu ; podmínka pro další iteraci ;           přechod na  
další iteraci )
```

```
for (int i = 0; i < počet; i++)  
for (String str = inputString; !str.isEmpty(); str =  
      str.substring(pozice)) ← (toto je ještě na hranici únosnosti)  
for (File f = new File(name); osoba.getPlat() > 20000.0;  
     osoba.hledej(hledOsoba))
```

- Využití enumerace v cyklu (pokud jí jazyk disponuje – Java od 1.6, C# nebo např. PHP)

```
for (int i = 0; i < seznamOsob.size(); i++)  
{ Osoba osoba = seznamOsob.get(i); ... }  
for (Osoba osoba : seznamOsob) { ... }
```

- Každá větev ve switch (Java, C/C++, C#, PHP aj.) má mít svůj break pokud nemá, je potřeba okomentovat

```
switch (druh) {  
case POCITAC:  
...  
break;  
case NOTEBOOK: (zde nebude komentář, protože to není samostatná větev)  
case NETBOOK:  
...  
break;  
case TABLET:  
...  
// dále stejně jako u smartphone: (okomentováno nepoužití break)
```

```
case SMARTPHONE:
...
break;
case MOBIL:
...
} (poslední větev break nepotřebuje, ale měl by se uvést! Proč?)
```

- Parametrizace programu pomocí konstant v projektu – případně s využitím podmíněného překladu, pokud jsou požadovány i odchylky ve funkčnosti (ideálně pokud vývojové prostředí umožňuje definovat více konfigurací projektu) – použije se v případě, že je potřeba udržovat více verzí jednoho programu (projektu) např. pro více zákazníků, kteří mají specifické požadavky (**nejhorší alternativou je okopírovat všechny zdrojové texty do jiného adresáře a tam upravit**)
- Umět negovat podmínku ( $!a \&\& !b \rightarrow a \ || \ b$  a nikoli  $a \ \&\& \ b$ ) (*pokud už to z nějakého důvodu je potřeba*)
- Udržování aktuálností databáze, struktury konfiguračního souboru
  - databáze, konf. soubor apod. má v sobě jako jednu z položek aktuální verzi a v programu musí být kód, který porovná tuto verzi s verzí programu a v případě nesouladu ví, co všechno musí aplikovat, aby výsledná struktura byla aktuální
  - tools pro databáze: liquibase, flywaydb
- Goto? (existuje, ale používat v opodstatněných případech)  
(každý program s lib. množstvím goto lze přepsat, tak aby neobsahoval jedině)