

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
6. Moderní inženýrské metody
7. Další doporučení
8. Licence a formy šíření software

Část B: Efektivita programu

1. Optimalizace algoritmu
2. Mechanismus přístupu k datům
3. Implementace programových struktur
4. 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. Výuka C/C++ byla postupně obohacována o doporučení či metody jak (ne)programovat. Po několika letech autor přestal zcela vyučovat programovací jazyk 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.

Cíle IB053

Účelem IB053 je předat studentům znalosti a zkušenosti s praktickým každodenním vývojem software. Pojem efektivní programování je zde chápán ve dvou smyslech. Jednak z hlediska práce vynaložené na tvorbu programu, která má být samozřejmě co nejefektivnější a jednak z hlediska samotného programu, jehož implementace má být též co nejefektivnější, t.j. aby počítač při vykonávání programu prováděl minimum režijních (nevýkonných) činností.

Obsahem předmětu není samotný návrh algoritmu z hlediska jeho složitosti a ani výuka metod softwarového inženýrství (PB007, PA017, příp. PA102)

Řada přednášených metod a doporučení nebývá v praxi dodržována, mnohdy záměrně. Studenti by si měli umět udělat kritický pohled a být schopni posoudit, jestli je celkově dodržení výhodnější nebo naopak nevýhodnější.

Cvičení

- ~~• Napsání programu porušujícího co nejvíce zásad, účast = podmínka zápočtu~~
- Srovnání efektivity 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~~
- účast na cvičeních není evidována, ale bez prvotní domluvy na projektu a průběžného předvedení nelze na konci projekt předat

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~~
→ **letos neplatí**
- rozdělení přednášek a cvičení určuje vyučující průběžně, rozdělení celkového počtu hodin mezi přednášky a cvičení nemusí být přesně 1:1
- účast na přednáškách není vyžadována, ale je vřele doporučena, protože zkušenosti lze předat spíše ústním sdělením než písemně, ideálně samozřejmě dialogem

Látka, která z důvodu kolize s jinými předměty nebo z jiných důvodů již nebude přednášena a nebude ani u kolokvia

Látka, která z jiných důvodů nebude u kolokvia

Část A: Efektivita práce při tvorbě programu

K čemu efektivita: Čas = peníze

1. Snížení chybovosti při tvorbě programu

- Důkladná analýza (*shora dolů*)
 - funkční (*rozdělení celého systému na komponenty, jejich rozdělení na moduly/packages, moduly na třídy + interface, třídy na metody*)
 - datová (*objekty (třídy), DB-tabulky, UML <- víme všichni co to je?*)

S využitím programových prostředků pro zápis analýzy/modelování dat (Enterprise Architect, Papyrus, ArgoUML, ...), některé umí generovat zdrojový kód, některé ze zdrojového kódu rekonstruovat datový model.

Model na papíře je mnohem cennější než jen v počítači nebo jen v hlavě - je do něho vidět a dá se do něho čmárat.

Pokud není analýza (funkční i datová) dostatečně podrobná, bude nutné při vývoji vytvářet paralelně programátorskou dokumentaci (obyčejné komentáře, Javadoc, použití aplikací typu Confluence)

- Dostatek času (*podvědomí pracuje za nás*)
- Práce v týmu (*Brainstorming*)
- Programování (*zdola nahoru <- to se hezky řekne, ale špatně udělá nebo spíše odprostřed dolů a pak odprostřed nahoru → aby se dalo co nejdříve ladit*)

K ladění jakékoli funkce/metody potřebuji vstupní data, která nemám, pokud programuji odspodu -> nutnost psát kód, jehož jediným smyslem je testování a poté se zahodí.

S využitím programových prostředků JUnit, TestNG (Java), Arquillian (Java EE), NUnit (.NET) <- **vždy platí, že kvalita testu je přímo úměrná úplnosti sad testovacích dat**

Tyto prostředky testují funkčnost modulů na nižších úrovních, nikoli např. chování

aplikace při ovládání uživatelem. Úsporu času přinesou zejména při kontrole funkčnosti již odladěného kódu po dodatečných úpravách.

- Dobrá znalost programovacího jazyka (*chyby z neznalosti jazyka se špatně hledají*), knihoven a frameworků

```
char * strcpy ( char * destination, const char * source );  
char * retezec;  
strcpy(retezec, "AHOJ");
```

- Dobrá znalost vývojového prostředí a využívání jeho možností
 - refaktoring – přejmenování, přesunutí
 - automatizované generování kódu (setters+getters)
 - Prostředky pro sestavování aplikace (MAKE, ANT, MAVEN, Gradle)
 - view definition/declaration - přesunutí kurzoru na definici funkce/třídy
 - call hierarchy - vyhledání všech výskytů (volání funkce, užití proměnné)
- Soustředěnost při práci (*kolegové v kanceláři, hudba, TV, mail, 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) { ... }` správně je: `if (i != NULL)`
...

- složité logické podmínky rozdělit na jednodušší
- zamezit negaci negovaného
- boolean metody vracející pozitivní stav (`hasRestriction` x `hasNoRestriction`)
- ctít logiku programované věci

např. místo

```
if (name.indexOf("$") > -1) raději  
if (name.indexOf("$") >= 0) (protože nás zajímá, jestli je znak $ na  
některé pozici řetězce, přičemž pozice se počítají od 0) a nebo případně  
if (name.indexOf("$") != -1) (protože v dokumentaci je, že funkce vrací  
-1, pokud se podřetězec nenajde a nás zajímá situace, kdy se podřetězec najde)
```

- 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 POCET_ZNAKU_ADRESY (40 + 30 + 5)
```

```
#define POCET_ZNAKU_ADRESY 75
```

- Symbolické konstanty (*správná interpretace, hromadná změna; nevyužívat k jiným účelům*)

```
jmeno[POCET_ZNAKU_JMENA + 1];
prijmeni[POCET_ZNAKU_JMENA + 1]; - použití k nesprávnému účelu
jmeno[POCET_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) -> pokud je to možné, používat javadoc nebo obdobné systémy, ale pozor, jen popis parametrů metody opravdu nestačí, je potřeba k metodám a třídám uvést opravdu všechno, co musí programátor vědět, aby mohl třídy a jejich metody spolehlivě použít

Stojí za to zvážit jazyk komentářů s ohledem na osoby, které je mohou potenciálně někdy v budoucnu číst (i když aktuálně jsou to pouze místní programátoři, nemusí tomu tak být i v budoucnu; u nadnárodních společností vždy angličtina).

- Dokumentace dodatečných změn (vyplývajících z odstraňování chyb nebo ze změn zadání)
 - změny psát do komentářů
 - při použití systému pro shromažďování/hlášení/třídění chyb a požadavků (Bugzilla, Jira apod.) psát do komentářů reference na záznamy, kde si lze dodatečně připomenout důvod změny a posoudit tak později, jestli byla provedena dobře
- Evidence chyb zjištěných přímo při vývoji (pokud si vývojář sám sobě objeví chybu a z důvodu koncentrace na jiný problém ji nemůže/nechce odstranit ihned a nebo pokud se jedná o chybu jiného vývojáře podílejícího se rovněž na projektu)
 - v aplikaci (Jira, Bugzilla)
 - soubor v editoru (pohotovější řešení než aplikace, každou chybu na samostatný řádek, lze snadno měnit priority přesouváním řádků, mazat nebo škrtnat hotové)
 - systém papírků (zastaralý, ale stále funkční :-)
- 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)
Není až tak podstatné, jaký styl zápisu zvolíme, podstatné je ho pak dlouhodobě dodržovat.

Příklady pro inspiraci (nikoli doporučení pro styl zápisu):

grafický styl bloku (odsazování):

<pre>záhlaví { } </pre>	<pre>záhlaví { } </pre>	<pre>záhlaví { } </pre>	<pre>záhlaví { } </pre>
--	--	--	-------------------------------

Mezery (každá neobvyklost musí mít své opodstatnění):

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

Přechody na nový řádek

<pre>příkaz 1; příkaz 2; příkaz 3;</pre>	<pre>příkaz 1; příkaz 2; příkaz 3;</pre>
--	---

- 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 mu to nebude fungovat, ale 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

Motivace:

použití vlastního (resp. týmového) zdrojového kódu vícekrát

použití cizího zdrojového kódu (frameworky, utils)

napsání zdrojového kódu, který může využít cizí osoba

- 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/knihovny nesouvisející funkčnost
- OOP: používat abstraktní metody, interface
 - nepřehánět -> abstraktní část kódu je méně přehledná, většinou její vývoj je časově náročnější - vždy zvážit, kdy se vyplatí a kdy ne (zpravidla pokud bude použita 1-2x, tak se tento postup nevyplatí)
- tvořit knihovny (zdokumentované: nejen statický popis tříd a metod, ale i nezávislý popis realizace nejčastějších situací)
- využití částí programů napsaných cizími osobami (získané z internetu, např. www.sourceforge.net, apache.org, github, maven-repository)
- využití cizích knihoven a frameworků (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; lze použít i rozsáhlé frameworky typu hibernate)
 - XML (např. v Javě JAXB)
 - frameworky pro programování web. aplikací (AngularJS, jQuery+jQueryUI, Bootstrap aj.)

- 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

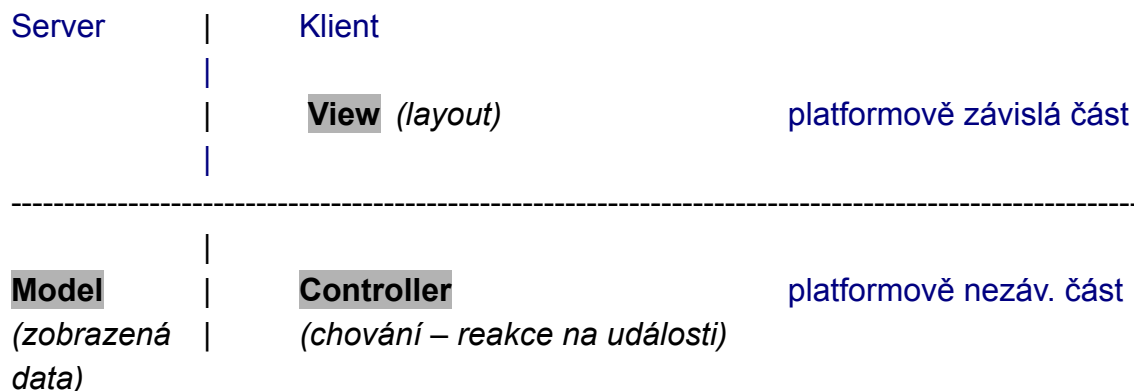
4. Nezávislost programu na pozdějších úpravách

- 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

Myšleno: použití jiného kompilátoru nebo dokonce překlad pod jiným operačním systémem. V případě webových aplikací možnost provozování pod různými prohlížeči

- orientace na standardní prvky jazyka, případně na dané verze/standards jazyka (C++ 1998, C++ 2003, C++11 apod., v Javě 1.5 - 1.9, 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 prakticky všechno standard, u C/C++ bývají rozdíly větší)
- užití knihoven jejichž jeden z účelů je zajištění kompatibility (na webu např. jQuery, desktopové aplikace např. GTK+ nebo OpenGL)
- 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 ve vlastnostech projektu – závisí na vývojovém prostředí, některá prostředí mohou v projektu mít víc konfigurací lišící např. právě předdefinovanými konstantami
 - moduly ve více verzích
 - rozdělené UI-dialogy: návrhový vzor Model-View-Controller



- znalost implementačních detailů (tj. to, co norma jazyka neřeší a nechává to na tvůrci kompilátoru) (např. *soubor s binárními daty*)
 - *implementace reálných čísel nemusí být vždy stejná*
 - *rozsahy datových typů (integer, long apod.)*
- spolupráce mezi částmi programů napsaných v různých prostředích/jazycích (kompilovaných) – další implementační detaily
Programm – DLL (pozdní vazba – řeší OS nebo VM)
Modul – Modul (volání – řeší kompilátor)
 - pořadí parametrů
 - implementace datových typů (reálná čísla, rozsah integeru)
 - 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. “Moderní” infromatické metody

(“moderní” = přišly na svět v relativně nedávné době) :-)

- Používat **návrhové vzory** (Singleton, Object-pool, Lazy-initialization, Observer, Model-View-Controller, DAO)
- Metody **agilního vývoje** (XP-extrémní programování a SCRUM; tento přístup je v praxi poměrně častý, patří spíše do softwarového inženýrství, zde zmiňujeme spíš z důvodu konfliktu s tímto předmětem -> před nasazením jakékoli metodiky je vhodné uvážit, jestli přinese víc než vezme)
 - postupné předávání software po menších dokončených celcích (v některých metodách se nazývají sprints) -> dřívější reakce na nepřesnosti v zadání
 - výhodné pro uživatele, který na začátku neví, co vlastně chce
 - absence plánování vývoje (protože vývoj je určován předchozími reakcemi)

uživatele) -> komplikovanější určení ceny software při předem dané funkčnosti; toto je zpravidla obráceně: zákazník stanoví výši investice, ale není jisté, co za to dostane

- preference funkčního softwaru před dodržením základních zásad, jako je dokumentace, komentáře, čitelný zdrojový text

spousta firem si na této metodě zakládá, ale je potřeba ji brát s rezervou: vysoké tempo se nedá vydržet stále a rovněž zanedbaná dokumentace se dřív nebo později negativně projeví, obzvlášť u rozsáhlých projektů

(podrobněji viz manifest agilního vývoje

<http://agilemanifesto.org/iso/cs/manifesto.html>)

- Použití netradičních databází (SQL x NoSQL, např. MongoDB)

- SQL: pevná struktura databáze je svazující, ale zajišťuje udržení pořádku

- SQL: léty vypracované mechanismy rozvržení struktury dat pro optimalizovaný přístup

- SQL existující frameworky a systémy pro usnadnění práce (hibernate, liquibase, ...)

- NoSQL: data uložená v různých fázích vývoje mohou mít různou strukturu, program na to musí umět reagovat

- NoSQL: nové možnosti optimalizace (související data, která jsou často potřeba současně, jsou uložena "na jednom místě" a tedy rychleji společně dostupná

- Mikroslužby (microservices), viz. např. <http://voho.eu/wiki/mikrosluzba/>

7. 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.**

- 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 smartphome: (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 aktuálnosti 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)