Kvalitní kód

- Výběr programovacího jazyka

Tabulka ukazuje poměr k jazyku C – kolik příkazů (řádků) kódu napsaného v jazyce C nahradí jeden příkaz (řádek) v daném jazyce.

C	1
C++	2.5
Java	2.5
C#	2.5
Perl	6
Python	6
Visual Basic	4.5

Vlastnosti programovacích jazyků

Assembler

Je jazyk nízké úrovně. Každý jeho příkaz odpovídá jedné instrukci procesoru. Assembler je tímto vždy specifický pro procesor, který je v použitém výpočetním systému. Užívá se výlučně v případech, kdy jsou takové požadavky na rychlost kódu nebo jeho velikost, že je nelze jiným jazykem splnit.

C jazyk

Je jazyk pro obecné použití vyvinutý na začátku 70. let. Má běžné konstrukce jazyků vyšší úrovně a bohatý soubor operátorů. Zároveň umožňuje široké využití ukazatelů (adres), což umožňuje efektivní programování blízké assembleru. Má poměrně slabý systém datových typů.

C++ jazyk

Objektově orientovaný jazyk založený na syntaxi jazyka C. Byl vyvinutý v 80. letech. Poskytuje deklarace tříd, polymorfismus, používání výjimek, šablony tříd. Má robustnější kontrolu datových typů, než má jazyk C. Obsahuje rozsáhlou a efektivní knihovnu šablon (STL).

Java

Objektově orientovaný jazyk, jehož syntaxe je podobná jazykům C a C++. Je určen pro používání na libovolné platformě. Zdrojový kód je konvertován do tzv. byte kódu (*byte code*), který může být použit v různých platformách v prostředí označovaném jako virtuální stroj (*virtual machine*). Java se v široké míře používá pro webové aplikace.

Javascript

Je interpretovaný jazyk pro skripty a je značně podobný jazyku Java. Slouží zejména k sestavování jednoduchých funkcí pro webové stránky.

C#

Objektově orientovaný jazyk pro obecné použití určený pro vývoj aplikací pro platformu Microsoft. Jeho syntaxe vychází z jazyků C++ a Java. Obsahuje řadu nástrojů usnadňující vývoj programů v platformě Microsoft.

Perl

Je jazyk pro práci s řetězci. Syntaxe je založena na jazyce C. (název: Practical Extraction Report Language). Je používán administrátory systémů pro skripty nebo generování a zpracování reportů. Rovněž je používán pro vytváření webových aplikací.

PHP

Skriptovací jazyk se syntaxí podobnou jazykům Perl, Javascript a C. (název: původně Personal Home Page, nyní Hypertext procesor). Užíván ve webových stránkách pro přístup a zobrazování databázových údajů.

Python

Je interpretovaný, interaktivní objektově orientovaný jazyk užívaný ve více prostředích. Nejčastěji je používaný pro psaní skriptů a malých webových aplikací. Obsahuje ale i prostředky pro vytváření rozsáhlejších programů.

Visual Basic

Objektově orientovaný jazyk založený na syntaxi jazyka Basic. Určený pro vytváření programů a webových aplikací v prostředí Windows.

Složitost systémů

Program	*Řádků kódu*
Linux	1.5 miliónů
Windows XP	40 miliónů
Vesmírná stanice	40 miliónů
Raketoplán	10 miliónů
Boeing 777	7 miliónů
Netscape	17 miliónů

➢ **Co je důležité při psaní programů**

Je zapotřebí způsob programování zvolit s ohledem na použitý programovací jazyk. Neměli bychom v jazyce programovat, měli bychom program přímo navrhnout pro daný použitý jazyk.

➢ **Kvalitní třídy**

- Soudržnost - třída se zabývá jen jedním tématem.
- Rozhraní třídy (veřejné operace) by mělo reprezentovat abstrakci na stejné, konzistentní úrovni.

```
class SeznamZamestnancu {
  void pridatZamestnance(Zamestnanec &zamestnanec);      // 1.
  void odebratZamestnance(Zamestnanec &zamestnanec);     // 1.
  Zamestnanec &prvniPrvekVSeznamu();     // 2.
  Zamestnanec &dalsiPrvek();             // 2.
```

```
  bool jePosledni();                    // 2.
}
```

Dvě úrovně abstrakce: 1. úroveň je „Zaměstnanec“, 2. úroveň je „Seznam“

```
class SeznamZamestnancu {
  void pridat(Zamestnanec &zamestnanec);
  void odebrat(Zamestnanec &zamestnanec);
  Zamestnanec &prvni();
  Zamestnanec &dalsi();
  bool jePosledni();
}
```

- Zapouzdření – atributy nejsou veřejné.
- Třída by neměla mít více než 7 atributů. Je statisticky ověřeno, že až 7 atributů si člověk dokáže zapamatovat, když dělá další činnosti, třeba sestavuje operace.
- V dědičnosti dodržet pravidlo „je“ (“is a”).
- Všechny operace definované v základní třídě by měly mít v děděných třídách stejný sémantický význam.

```
class Ucet {
  virtual void vypocitatUrok();
  // částka, kterou banka platí klientovi
}

class DebetniUcet: public Ucet {
  void vypocitatUrok();
  // zde má odlišný význam, je-li stav záporný, platí klient bance
}
```

- Neměly by se dědit atributy nebo operace, které v dědící třídě nejsou zapotřebí.
- Neměli bychom překrýt (*override*) operace, které k tomu nejsou určeny (nemají specifikaci *virtual*), což třeba v jazycích C++ a Java je možné.
- Je podezřelá abstraktní základní třída, která má jen jednu dědící třídu. Může to nastat, když si připravujeme do budoucna možnost vytvořit další dědící třídu. Nicméně při jedné dědící třídě základní třída zbytečně komplikuje kód. Základní třídu bychom měli vytvářet, jen pokud je to nutné.
- Je podezřelá základní třída, jestliže v dědící třídě některé její operace implementujeme metodou, která nic nedělá.

  Například mějme třídu *Ucet* a v ní operaci *vypocitatDebetniUrok*. Děděním třídy *Ucet* vytvoříme třídu *SporiciUcet*. U spořicího účtu není přípustný debetní stav a operaci *vypocitatDebetniUrok* implementujeme metodou, ve které není žádný výpočet. Zde viditelně operace *vypocitatDebetniUrok* neměla být v základní třídě *Ucet* uvedena, ale měla z ní být odvozena nová třída *UcetSDebetnimStavem* pro dědění účty, které debetní stav umožňují, a operace *vypocitatDebetniUrok* by měla být až v ní.

- Hloubka dědění by neměla být příliš velká. Nad 3 úrovně začíná být dědění málo přehledné.
- Třídy by měly modelovat objekty reálného světa nebo abstrakci reálných objektů.

  Kruh, čtverec – reálné objekty.

  Tvar – abstrakce objektů se specifickým tvarem.
- Jména tříd by měla být podstatná jména nebo jmenné fráze. Vhodná jsou jména, která vyjadřují obsah a účel třídy, například *Zakaznik, Ucet*. Nevhodná jména jsou vágní jména jako *Procesor*, *Data*, *Informace*.

➢ **Kvalitní funkce**

- Soudržnost funkce.

  Soudržnost funkce označuje, jak silně jsou operace obsažené ve funkci vzájemně svázány. Tj. kolik činností a jaké funkce vykonává. Například funkce *zmenitVelikost* má vysokou soudržnost, protože dělá jen jednu činnost – změní velikost nějakého objektu. Naproti tomu funkce *otocitAPosunout* má nízkou soudržnost, protože dělá dvě různé činnosti – otočí objekt a posune ho. Zde by měly být dvě samostatné funkce *otocit* a *posunout*.

  Ideální jsou funkce s vysokou soudržností. Běžně se ale v programech vyskytují funkce, které mají nižší úroveň soudržnosti:

  Přijatelné soudržnosti:

  - Sekvenční soudržnost.

    Funkce obsahuje operace, které musí být dělány v určitém pořadí, neboť jedna navazuje na výsledek předchozí. Příklad může být funkce, která nejprve vypočítá diskriminant kvadratické rovnice a následně vypočítá kořeny rovnice.
  - Komunikační soudržnost.

    Funkce obsahuje operace, které používají stejná data, ale nejsou vzájemně nijak svázány. Příklad může být funkce, která vytiskne údaje a rovněž je uloží do souboru. Zde obě operace jsou na sobě nezávislé, jen používají stejná data.
  - Časová soudržnost.

    Ve funkci jsou operace, které probíhají ve stejnou dobu. Typickým příkladem může být funkce *inicializace*, která dělá různé operace související s inicializací okna. Ty spojuje jen skutečnost, že se všechny dělají ve stejný čas.

  Nežádoucí soudržnosti:

  - Procedurální soudržnost.

    Operace ve funkci spolu nesouvisí, jen probíhají ve stanoveném pořadí. Například ve funkci je zpracováno křestní jméno, následně příjmení a pak adresa, což je dáno jen pořadím, jak jsou tyto údaje čteny ze vstupu. Je lepší, aby pro vstup každého údaje byla samostatná funkce a ty byly volány ze samostatné funkce *cteniOsoby*.
  - Logická soudržnost.

    Více operací je sloučeno do jedné funkce a která z nich se provádí, je určeno vstupním parametrem funkce. Název logická soudržnost je odvozen ze skutečnosti, že operace je vybrána pomocí podmíněného příkazu nebo přepínače na základě

hodnoty parametru funkce. Je vhodnější místo jedné funkce, které se předává parametr, mít více funkcí, každá z nich realizující jednu z možností a místo předání parametru příslušnou z nich zavolat.

- Náhodná soudržnost.

  Mezi operacemi ve funkci není žádný viditelný vztah. Taková funkce by se dala označit i "bez soudržnosti". Je účelné zde přepracovat návrh funkce a její implementaci.

- Volba jména funkce.

  Jméno by mělo vyjadřovat, co funkce dělá. Nic neříkající, vágní jména by se neměla používat:

```
udelatVypocet();
zpracovatVstup();
```

  Dobře sestavené jméno:

```
vypocitatUrok();
```

  Z něho je zřejmé, co funkce počítá.

- Jména by se neměla odlišovat jen číslicemi. Situace: Máme rozsáhlou funkci s mnoha řádky kódu a některé její části jen vyjmeme a vytvoříme z nich samostatné funkce. Například ve vyhledávacím stromu, ve kterém jsou uloženy záznamy účtů, přidáváme nový účet funkcí

```
pridatUcet(Ucet &ucet);
```

  To může vyžadovat vyvážení vyhledávacího stromu po přidání, které dáme do samostatné funkce

```
pridatUcet1();
```

  Jméno je zde použito nevhodně, nevyjadřuje, co funkce dělá. Lepší by bylo jméno

```
vyvazitStromUctu();
```

- Struktura jména.
  - Vrací-li funkce hodnotu, jméno funkce by mělo být popisem návratové hodnoty

```
double korenRovnice();
int pero.soucasnaBarva();
      |          |
    objekt   funkce vracející aktuální nastavení barvy
```

  - Nevrací-li funkce hodnotu, typická struktura jména je

    sloveso + jmenná fráze popisující, co je předmětem funkce

```
vytisknoutObjednavku();
```

    Je-li předmět funkce reprezentován jménem objektu, ve kterém je daná funkce členskou funkcí, stačí uvést jen sloveso

```
objednavka.vytisknout();
   |              |
objekt          funkce
```

- Vrací-li funkce hodnotu, jméno funkce by mělo voleno tak, aby odpovídalo datovému typu návratové hodnoty. Například funkce se jménem *mezeryNaKonci* může vracet počet mezer, které jsou na konci, nebo logickou hodnotu udávající, zda na konci jsou mezery. V prvním případě zvolíme raději jméno *maMezeryNaKonci*.
- Někdy lze ve jménu vynechat nějaké slovo, pokud význam zůstane zachován. Místo *zmenitNaMalaPismena* stačí napsat *naMalaPismena*.
- Vyhýbáme se slovům s nejednoznačným významem. Například v následujícím volání funkce pojmenované *filtrovat*

  ```
  filtrovat(rok<2010)
  ```

  není zřejmé, jaký je výsledek.
  - Jsou to roky před rokem 2010?
  - Nebo jsou to naopak roky, které nejsou před rokem 2010?

  Jiný příklad může být funkce *ohranicitText*. Význam této funkce může být, že například odstraní mezery, pak ji nazveme *odstranitMezery*. Pravděpodobnější je význam funkce, že omezí maximální délku textu. Pak bychom ji nazvali *zkratit*.

➢ Parametry funkcí

- Měly by být v pořadí
  - první: vstupní parametry
  - za nimi: vstupně/výstupní parametry
  - poslední: výstupní parametry

```
void editovatRetez(const char *vstupniRetez,
                   char *vystupniRetez);
```

- Jestliže více funkcí používá podobné parametry, měly by být ve stejném pořadí.
  Jazyk C – záporný příklad:

```
int fprintf(FILE *stream, const char *format, ...);
int fputs(const char *str, FILE *stream);
```

  V první funkci je odkaz na soubor zadán v prvním parametru, ve druhé je naopak soubor zadán v posledním parametru. Je obtížné si zapamatovat, na kterém místě parametr je v jednotlivých funkcích.
- Parametr, který slouží k uložení statusu nebo údajů o chybě by měl poslední. Je to navíc výstupní parametr, což i odpovídá zásadě, že takové parametry jsou na konci.
- Parametr by neměl být používán jako lokální proměnná – změnou své hodnoty ztrácí svůj původní význam, což může být matoucí.

```
void foo(int pocet)
{
  while (pocet--) {...}
}
```

- Funkce by neměly mít nadměrný počet parametrů. Psychologický výzkum ukazuje, že si lze zapamatovat parametry, pokud jich není více než 7.

- **Délka funkce**

  Statisticky je ověřeno, že v kratších funkcích bývá nižší procento výskytu chyb než v dlouhých funkcích. Vhodnější jsou proto kratší funkce. Jestliže funkce má více než 200 řádků, je zapotřebí zvážit, zda je to účelné.

- **Polymorfismus**

  V některých případech se využitím polymorfismu dá efektivně nahradit přepínač (příkaz *case*).

  Mějme třídu, která umožňuje kreslit různé tvary podle typu tvaru:

```
switch (objekt.typTvaru)
{
  case KRUZNICE: objekt.nakreslitKruznici();
                 break;
  case CTVEREC:  objekt.nakreslitCtverec();
                 break;
   ...
}
```

  Zde by bylo účelné sestavit základní třídu a jejím děděním vytvořit třídy pro jednotlivé tvary a v nich budeme mít virtuální funkci *nakreslit*, kterou můžeme volat příkazem *objekt_daného_tvaru.nakreslit()*.