PROGRAMOVÁNÍ NA PŘELOMU TISÍCILETÍ

Miroslav Virius

Fakulta jaderná a fyzikálně inženýrská ČVUT v Praze, Trojanova 13, Praha 2
virius@km1.fjfi.cvut.cz

Abstrakt
Příspěvek stručně shrnuje historii programování a tvorby aplikací a ukazuje trendy, které mohou určit její budoucnost.

1. První počítače

Úvahy o historii výpočetní techniky nemohou začít jinak než stroji nazvanými Difference Engine a Analytical Engine, které navrhl v první polovině 19. století Angličan Charles Babbage. Oba stroje měly být mechanické, založené na ozubených kolech a poháněné parou.

1.1 Mechanické počítače

Difference Engine, jehož zjednodušený funkční prototyp sestrojil C. Babbage r. 1822, měl sloužit k výpočtu tabulek polynomů a pomoci odstranit obrovské množství chyb v tehdejších ručně počítaných astronomických tabulkách užívaných pro námořní navigaci.

Při pokusu o plnou realizaci si ale uvědomil, že princip, z něhož vyšel, poskytuje širší možnosti, a r. 1837 navrhl novou verzi, označenou Analytical Engine, která již představovala univerzální mechanický číslicový počítač se vstupním a výstupním zařízením využívajícím děrné štítky, s pamětí a operační jednotkou řízenou programem zapsaným na děrných štítcích.

Poznamenejme, že C. Babbage ani jeden z těchto projektů nedokázal dokončit. To se podařilo až r. 1853 švédskému právníkovi Georgu Scheutzovi.

Instrukční sada tohoto předchůdce dnešních počítačů obsahovala i podmíněný skok, který umožňuje větvení algoritmu a vytváření cyklů. Babbageův stroj byl tedy v principu schopen provádět libovolně složité výpočty.

Za bezprostředního předchůdce dnešních počítačů bývají označována reléová zařízení Z_1, Z_2 (1939) a Z_3 (1941), která sestrojil v Německu Konrad Zuse. Neznal však Babbageovy práce a mezi instrukcemi jeho počítačů chyběl podmíněný skok. Nezávisle na něm sestrojil podobné zařízení r. 1939 v USA prof. John Atanasoff a použil je pro řešení systému diferenciálních rovnic, na něž narazil při řešení problémů z teoretické fyziky.

Velice důležitou roli ve vývoji počítačů a programování sehrála druhá světová válka. Ve Velké Británii byla pro účely prolomení kódu německého kódu Lorenz, užívaného nejvyšším velením, na základě teoretických prací A. Turinga sestrojen Colossus, patrně první skutečný počítač; jeho dokumentace byla však po válce z důvodu utajení zničena. Ve Spojených státech byly při výpočtech pro projekt Manhattan využívány děrnoštítkové počítací stroje IBM.

1.2 Počítače von Neumannova typu

Všechna tato zařízení se programovala propojováním funkčních bloků. V r. 1944 navrhl John von Neumann, aby instrukce, které řídí chod počítače, byly ukládány do operační paměti, neboť jde o druh dat. (V literatuře bývá také uváděno datum 1946.) Uskutečnění této myšlenky lze pokládat za počátek éry počítačů v dnešním smyslu – éry počítačů von Neumannova typu. Prvním počítačem von Neumannova typu byl EDSAC postavený v r. 1949 a jeden z prvních programů byl výpočet Ludolfova čísla π na cca 2000 míst.

I když na *principu* počítačů se od té doby nic nezměnilo, změnilo se nesmírně mnoho, pokud jde o programování. Zde se uplatnily především čtyři faktory, které na sebe navazují:

- Prudký pokles cen (o 6 řádů v průběhu 50 let),
- stejně prudký nárůst výkonnosti počítačů,
- jejich masové rozšíření a nasazení v nejrůznějších oblastech,
- propojení počítačů do celosvětové sítě.

Poslední dva faktory se sice uplatňují až v posledních několika letech, způsobily ale značné změny v chápání počítačů a samozřejmě i v programování.

1.3 Československo: počítače a politika

U nás – a v celém východním bloku – se od počátku padesátých let uplatňovala Stalinova doktrína, podle níž byla kybernetika, jak se tehdy věda o řízení a počítačích nazývala, „buržoazní pavěda“. Tento postoj oficiálních míst přetrvával i po Stalinově smrti zhruba do poloviny 60. let.

Přesto byl r. 1957 v Československu postaven prof. Antonínem Svobodou a jeho spolupracovníky první československý počítač, a to ve Výzkumném ústavu matematických strojů, který v té době patřil Československé akademii věd. Nesl název SAPO (SAmočinný POčítač) a zahraniční literatuře je uváděn jako první počítač, který měl vestavěné mechanizmy pro ošetřování chyb, neboť používal samoopravné kódy. Počítač SAPO byl postaven z elektromechanických relé.

Samoopravné kódy byly použity i ve druhém československém počítači, označeném EPOS (Elektronkový POčítací Stroj, 1963). I tento počítač stavěl prof. Svoboda, před jeho dokončením však byl neustálými překážkami z oficiálních míst donucen emigrovat. Tím vývoj vlastních počítačů v Československu bohužel skončil; Československo tak ztratilo možnost stát se počítačovou velmocí, možnost, k níž za prof. Svobody směřovalo [9].

2. Programovací jazyky

Na vývoj programování se můžeme dívat jako na *vkládání vrstev abstrakce mezi stroj a programátora*. První vrstvou, kterou člověk mezi sebe a počítač vložil, představují programovací jazyky. Tyto jazyky se vyvíjejí podobně jako jazyky lidské – překotně, podle okamžité potřeby a často naprosto nelogicky. Přesto může pohled na jejich historii napovědět leccos o současných a budoucích trendech.

2.1 Strojový kód

První počítače se programovaly ve strojovém kódu. Jeden z největších problémů přitom představovaly absolutní adresy v programu: Přidání nebo odstranění jediné instrukce

znamenalo změnu všech adres za ní, a tedy i změnu odkazů na tyto adresy v programu. Je jasné, že opravy a údržba takového programu byly velice náročné. Kromě toho musel programátor podrobně znát i architekturu počítače, pro který svůj program psal.

2.2 Asembler

Náročnost programování ve strojovém kódu způsobila, že se hned od počátku začaly používat jazyky symbolických adres; dnes pro ně používáme souhrnné označení asemblery.

V jazyce symbolických adres jsou již číselné kódy instrukcí nahrazeny mnemotechnickými zkratkami. Jejich nejdůležitějším přínosem je ale zavedení symbolických adres: Tyto jazyky umožňují programátorům označit různá místa v programu návěštím a na toto návěští se v instrukcích odvolávat. Z programu tím zmizely absolutní adresy.

Cenou za toto zjednodušení ovšem je, že program napsaný v asembleru je nutno nejprve přeložit do strojového kódu. Musí tedy být k disposici překladač, pomocný program, který se o to postará. I když programátor, který používá jazyk symbolických adres, musí stále velmi podrobně znát architekturu počítače, je programování v tomto jazyce nesrovnatelně jednodušší než programování ve strojovém kódu.

2.3 První vyšší programovací jazyky

Jazyky symbolických adres pracují s ekvivalenty elementárních instrukcí počítače. Neumožňují zapsat např. výpočty v podobě, na jakou jsme zvyklí z matematiky. To umožňují až tzv. vyšší programovací jazyky. Jejich idea je přibližně stejně stará jako idea počítače von Neumannova typu. Vznik vyššího programovacího jazyka předpokládal K. Zuse již v r. 1945 a nazval ho *Plankalkül*.

Za první použitelný vyšší programovací jazyk se zpravidla pokládá Fortran, vyvinutý v letech 1954 až 1957 u firmy IBM týmem vedeným Johnem Backusem pro počítač IBM 704. Jejich cílem bylo vyvinout programovací jazyk, který bude snadno zvládnutelný a přitom povede k efektivnímu výslednému programu. Od verze II poskytoval Fortran možnost rozdělit program na několik samostatně překládaných částí.

Největší síla Fortranu spočívala v možnosti zapisovat poměrně složité matematické vzorce způsobem blízkým matematické praxi. (Ostatně dobová terminologie nehovořila o zdrojovém textu ve Fortranu jako o programu, ale jako o matematické specifikaci problému – program vznikl překladem matematické specifikace do strojového kódu.) Program bylo možno rozdělit na samostatně překládané podprogramy.

Tato verze Fortranu však neobsahovala konstrukce jako složený příkaz nebo úplný příkaz IF, kromě polí nepodporovala uživatelem definované typy, neposkytovala nástroje pro práci s pamětí, nedovolovala rekurzi při volání podprogramů atd.

Jazyk Fortran byl určen především pro vědeckotechnické výpočty. Zpřístupnil programování, jež bylo do té doby záležitostí úzce vyhraněných specialistů, vědcům a inženýrům.

2.4 Problémově orientovaní programovací jazyky

Brzy po uvedení Fortranu se začaly objevovat další programovací jazyky – ostatně diverzita je základem evoluce všude, nejen v biologii. Přínos různých programovacích jazyků lze hodnotit z mnoha různých hledisek; my všimneme především následujících oblastí:

- Rozvoj základních vyjadřovacích schopností programovacích jazyků,
- vliv na styl programování.

Jazyk **Algol 60** uvedl do povědomí programátorské veřejnosti mimo jiné pojem bloku a blokové struktury programu. Poskytl také úplný příkaz `if` a zavedl rekurzivní podprogramy.

Cobol 60 vznikl jako jazyk pro zpracování hromadných dat (předchůdce dnešních databázových aplikací). Nabídl programátorům nehomogenní pole, konstrukci, která se obvykle označuje jako *struktura* nebo *záznam* a která představuje skupinu proměnných různých typů chápanou jako celek.

LISP byl publikován v r. 1960 skupinou Artificial Inteligence Group vedenou na M.I.T. prof. J. McCarthym jako jazyk pro symbolické manipulace, především pro zpracování seznamů. LISP je jazyk pro funkcionální programování – všechny řídicí struktury v něm vznikají skládáním funkcí. Tento jazyk ukázal, že imperativní programovací jazyky (jazyky, v nichž je program zapsán jako posloupnost příkazů) nepředstavují jedinou možnost. Podnítil také zájem o dynamické datové struktury.

Programovací jazyk **APL** publikoval na počátku 60. let K. E. Iverson. Za základní jednotku dat považoval pole a umožňoval s ním nejrůznější operace. Vynikal stručností zápisu, program v něm byl ovšem velice obtížně srozumitelný. Většího rozšíření dosáhl nakrátko až v 70. letech. Jeho implementace ukázala význam různých optimalizačních strategií – např. odloženého vyhodnocování – při práci s poli.

2.5 Univerzální programovací jazyky

Od počátku 60. let směřoval vývoj k univerzálním programovacím jazykům, tedy k jazykům, které budou poskytovat nástroje pro vědeckotechnické výpočty, zpracování hromadných dat, zpracování dynamických datových struktur a další – v té době třeba ještě neznámé – aplikace.

Prvním prakticky použitelným univerzálním programovacím jazykem byl **PL/I** (IBM, 1964). V principu šlo o spojení vlastností Fortranu, Algolu, Cobolu a Snobolu s některými novými rysy. Zavedl mj. mechanizmus zpracování výjimečných situací za běhu programu, možnost paralelního zpracování vstupních a výstupních operací a práci s ukazateli. Celkově však byl hodnocen spíše nepříznivě, neboť nabízel příliš málo reálné síly při obrovském rozsahu definice jazyka.

Jazyk **Simula 67** (O. J. Dahl, K. Nygaard) vznikl jako nadstavba Algolu 60; uvedl na scénu plnohodnotné objektově orientované programování – obsahuje třídy i dědičnost; polymorfizmus je řešen virtuálními funkcemi. Vedle toho seznámil širší programátorskou veřejnost s automatickou správou paměti (garbage collector) a nabídl prostředky pro práci se seznamy.

Dalším výrazným pokusem o univerzální jazyk byl **Algol 68**. Nabídl paralelní programování, základní prostředky pro synchronizaci procesů (semafor), práci s ukazateli, řízení alokace

paměti, možnost definovat nové operátory a měnit význam standardních operátorů, zacházet s funkcemi jako s datovými objekty atd. Přišel také s myšlenkou *ortogonality*: Programovací jazyk se má skládat z malého množství základních prostředků a pravidel pro jejich systematické kombinování a nesmí obsahovat náhodná omezení. Poznamenejme, že první rozumné implementace Algolu 68 se začaly objevovat až ve druhé polovině 70. let, doopravdy se tento jazyk ale nikdy nerozšířil.

2.6 Současné programovací jazyky

Pascal (N. Wirth, 1971) byl navržen jako jazyk pro výuku programování a znamenal revoluci ve výuce informatiky: Prakticky okamžitě nahradil dříve používané jazyky PL/I a Algol 68. Brzy však začal být používán i pro skutečné aplikace. Tento jazyk položil důraz na jednoduchost, přehlednost a dobré strukturování programu. Zavedl také výčtové typy.

Jazyk **C** (D. Ritchie, 1972) obsahoval podobné možnosti jako Pascal, vedle toho ale nabídl možnost odděleného překladu a prostředky pro nízkoúrovňové programování (adresovou aritmetiku), díky níž byl označován jako „přenositelný asembler“. Vzdor syntaxi, která je na první pohled poněkud kryptická, se rychle rozšířil a inspiroval vznik řady dalších jazyků (Objektive C, C++, Concurrent C atd.).

Jazyk **Smalltalk** (1973) byl prvním čistě objektovým programovacím jazykem. Vše – i třídy nebo části kódu – v něm představuje objekt. Setkáme se v něm s pojmy jako je metatřída. Tento jazyk předběhl svou dobu, neboť mohl efektivně běžet jen na dostatečně výkonných počítačích. Posloužil však jako nástroj pro rozvoj teorie objektového programování a objektových databází a ukázal výhodu integrovaného vývojového prostředí.

Jazyk **Ada**, vyvinutý z iniciativy Ministerstva obrany USA (1980), kladl maximální důraz na bezpečnost. Jeho novinkou bylo generické programování.

Jazyk **C++** (B. Stroustrup, 1985) byl nejpopulárnějším jazykem 90. let, a to i přesto, že se v té době prudce vyvíjel a prošel řadou změn. Do OOP uvedl vícenásobnou dědičnost.

Jazyk **Java** (Gosling, 1995) je sice syntakticky podobný C++, je ale určen pro tzv. *virtuální stroj*. To ale není v programování nic nového – s tím jsme se mohli setkat už v případě jazyka Basic (Kemeney, 1963). Je prvním široce používaným jazykem, který nabízí nástroje pro pokročilou práci s typy (reflexi). Java je čistě objektová; její nepřehlédnutelný úspěch je však především důsledkem její přenositelnosti a propracovaných knihoven pro nejrůznější podnikové aplikace – tedy technologického zázemí, které programátorům poskytuje.

V r. 2003 uvedla firma Microsoft platformu .NET, jejíž některé součásti byly později standardizovány mezinárodní organizací ISO. Programy pro tuto platformu se překládají do čistě objektového bajtového kódu označovaného **intermediate language** (**IL**).

Platforma .NET přinesla poměrně závažnou novinku, označovanou jako *atributy*. Jde o deklarativně zadávané dodatečné informace o datových typech, metodách, datových složkách, parametrech, modulech a dalších součástech programu. Tyto informace se ukládají do přeloženého programu jako tzv. *metadata* a lze je za běhu využít.

Vyšší programovací jazyky, které se překládají do IL, musely být upraveny, mj. proto, aby umožňovaly práci s atributy a využití automatické správy paměti.

3. Vývojová prostředí

Integrovaná vývojová prostředí, často označovaná zkratkou IDE, představují další vrstvu abstrakce, tentokrát mezi programátorem a překladačem. V této oblasti probíhá dnes největší část vývoje, neboť kvalitní IDE může podstatným způsobem zvýšit výkonnost programátora.

Historicky první IDE s sebou přinesl jazyk Basic. Tento jazyk byl od počátku interpretovaný, takže umožňoval spouštět úseky programu a hned vidět jejich výsledky. To umožnilo programátorům interaktivní práci při vývoji programů, která je podstatně efektivnější než v té době standardní dávkové zpracování (systém *closed shop*). Ovšem skutečný význam Basic získal, až když se terminály, umožňující interaktivní práci, staly standardní součástí výpočetních středisek.

Další významný krok ve vývoji IDE přinesl jazyk Smalltalk. Graficky orientované IDE je nedílnou součástí tohoto jazyka, neboť se skládá z objektů, které jsou součástí knihoven a lze je využívat i v programu. Toto prostředí také přineslo možnost interaktivního ladění.

Z hlediska rozšíření nelze pominout IDE programovacích jazyků C a Pascalu na PC od poloviny 80. let; masový prodej těchto nástrojů umožnil jejich rychlý vývoj. Jejich první verze v polovině 80. let neobsahovaly o mnoho více než editor, překladač a nápovědu; na počátku 90. let přibyl integrovaný ladicí program. Nástroje jako Visual Basic, Delphi apod. ukázaly v průběhu 90. let sílu vizuálního programování založeného na komponentách.

V dnešních IDE k základní čtveřici, tvořené překladačem, editorem, ladicím programem a nápovědou, přistupují další nástroje, které automatizují rutinní operace (průvodci atd.), generují opakující se části kódu, usnadňují orientaci ve zdrojovém kódu, profilování programu, vyhledají chyby ve zdrojovém kódu ještě před vlastním překladem, automaticky formátují zdrojový text atd. Stále častěji se také setkáme s integrovanými nástroji pro sledování historie zdrojového kódu, pro správu verzí, správu požadavků, podporu refaktoringu, automatické generování testovacích programů, tvorbu a nasazování podnikových aplikací atd. Nástroje, které se dříve označovaly jako CASE a byly dostupné jen nejbohatším firmám, se stávají integrální součástí IDE. Zatímco na konci 80. let se IDE zabývala *programy* a v první polovině 90. let *projekty*, dnes hovoří o *řešeních* nebo *skupinách projektů*.

4. Paradigmata a metodiky programování

Zpočátku se o programovacím stylu, o přístupu k programování, o metodice programování nebo o programovacím paradigmatu nehovořilo. Na konci 50.let vzrostla složitost programů a v důsledku toho i potřeba rozdělit program na více samostatných částí a tyto části pak opakovaně využít. Tyto požadavky podnítily vznik **modulárního programování**, které zavedlo modul jako základní jednotku programu. Modul je uzavřená, zpravidla samostatně překládaná součást, která prostřednictvím svého *rozhraní* poskytuje služby ostatním částem programu.

Přelom v přístupu k programování znamenal článek [2], který dokazoval, že kvalita programu je nepřímo úměrná počtu příkazů skoku, jež se v něm vyskytují. Experimenty, provedené nejen softwarovými firmami, toto tvrzení potvrdily.

Na konci 60. let se také začalo hovořit o *softwarové krizi*. Tento pojem vyjadřoval propastný rozdíl mezi prudce rostoucí úrovní hardwaru a úrovní tvorby softwaru. Výsledkem těchto diskuzí bylo **strukturované programování**, jehož základní ideje jsou:

- Program je složen ze tří základních programových struktur: posloupnosti (sekvence) příkazů, cyklu (iterace) a větvení (rozhodování).
- Každá programová struktura má právě jeden vstupní a právě jeden výstupní bod.

Zároveň se objevily i metodiky, které poskytovaly návod, jak ze zadání odvodit logicky správnou a přitom přehlednou strukturu programu. Některé s nich se opíraly o představu, že struktura programu je obrazem struktury dat – např. že opakující se data stejného typu na vstupu budeme zpracovávat cyklem apod. Asi nejznámější metodika strukturovaného programování pochází od M. Jacksona [3] a opírá se o tzv. Jacksonovy diagramy. Známé jsou také Yourdonovy diagramy toku dat [4]. Tyto metodiky naznačily, že programování v dohledné době přestane být uměním a stane se technologií s přesně danými postupy.

Objektově orientované programování (OOP), které se rozšířilo od 80. let, je logickým pokračováním předchozích dvou přístupů. Vychází z pojmu *třídy*, který je zpravidla abstrakcí třídy objektů z oblasti řešeného problému. *Objekty* jsou pak instance (proměnné apod.) těchto typů. Na třídu se lze dívat jako na modul, v němž jsou zapouzdřena data – jak data jednotlivých instancí, tak i data tříd jako celků. Operace, které lze provádět s daty tříd nebo objektů, se označují jako *metody*. Objektově orientovaný program se skládá z objektů, které si navzájem posílají zprávy (tj. volají své metody).

Od dané třídy lze v OOP odvodit další třídy mechanizmem *dědičnosti*. Odvozené třídy zpravidla představují podtřídy (specializaci) bázové třídy, tj. třídy, od níž byly odvozeny. Pro instance odvozených tříd v OOP platí, že jsou zároveň instancemi bázové třídy. Dalším základním rysem OOP je *polymorfizmus* – možnost zacházet se všemi instancemi stejným způsobem (posílat jim stejné zprávy). V mnoha jazycích je polymorfizmus omezen na instance tříd odvozených od společné bázové třídy.

Také pro OOP se objevila řada metodik; v současné době získaly dominantní postavení metodiky založené na modelovacím jazyku UML [5]. Některé z nich vycházejí ze starších metodik pro návrh databází, např. entitně-relačních diagramů.

Logickým pokračováním OOP v distribuovaném prostředí současných počítačových sítí je **komponentové programování** založené na znovupoužitelných komponentách – samostatně vyvíjených a dodávaných třídách nebo skupinách tříd, které poskytují specifické služby. Typickým příkladem jsou komponenty tvořící grafické uživatelské rozhraní aplikací, poskytující připojení k databázím apod. Hojně se využívají v IDE různých vývojových nástrojů.

V nedávné době se objevilo **šablonové metaprogramování**. Jde o přístup založený na využití generických konstrukcí (šablon), který lze stručně charakterizovat jako snahu nechat místo vlastního programu počítat překladač. I když to vypadá na první pohled problematicky, lze ho využít při optimalizaci některých typů programů. Ukažme si velice jednoduchý, i když ne zcela typický příklad:

```
template<bool> class Assert;
template<> class Assert<true>{};
```

Tato deklarace primární šablony `Assert` s parametrem typu `bool` a její specializace pro hodnotu `true` představuje analogii makra `assert()`, ovšem vyhodnocovanou překladačem. (Primární šablona je deklarována, není však definována, nemá implementaci.)
Dnes se také hovoří se také o **aspektově orientovaném programování**, které řeší problém *aspektů* – vlastností společných jinak nesouvisejícím třídám. Mezi často skloňované pojmy patří i **agentově orientované programování**, založené na agentech, programových entitách schopných kontinuálně a autonomně plnit určité úkoly v distribuovaném prostředí.

5. Současný stav

V současné době jsou v centru pozornosti distribuované obchodní aplikace. To určuje hlavní směr vývoje nejen programovacích jazyků, jejich knihoven a podpůrných nástrojů, jako jsou propracovaná IDE, ale i metodik programování.

Pravděpodobně nejpopulárnějším programovacím jazykem je v současné době Java, která se zejména díky snadné přenositelnosti přeloženého programu od počátku prosazovala jako nástroj vhodný pro prostředí internetu. Narazíme na ni všude, kde není kritická rychlost odezvy. Její zásadní předností jsou bohaté a neustále se vyvíjející knihovny pro nejrůznější aplikace.

Vedle Javy patří k nejpoužívanějším jazyky pro skriptování v prostředí internetu na straně klienta i serveru (PHP, Perl, Python). Podobně jako Java poskytují propracované knihovny, i když zdaleka ne v takové šíři – alespoň prozatím. Pro výměnu dokumentů se používá značkovací jazyk XML.

Nejen preferovaným, ale v podstatě požadovaným přístupem je dnes objektově orientované programování. Při tvorbě aplikací se stále více uplatňují komponenty dodávané třetími firmami, často specializovanými právě na produkci softwarových komponent. Přes nesporný úspěch OOP je ale jasné, že objektové ani komponentové programování nepředstavují konečné slovo, neboť neposkytují nástroje pro programový popis příčin a následků, popis změn stavu systému apod.

Zcela dominantní postavení mají v současné době metodiky objektově orientovaného návrhu založené na modelovacím jazyce UML.

6. Co můžeme čekat?

Na závěr se zamyslíme nad tím, co můžeme ve vývoji softwaru očekávat v blízké i vzdálenější budoucnosti. Především je zřejmé, že v blízké budoucnosti definitivně přestane být programování chápáno jako umění a bude pokládáno za běžnou technologii.

6.1 Hardware

I když se princip počítače nezmění (a to není jisté – na obzoru jsou kvantové počítače a úvahy o biologických a jiných principech), musíme počítat s důsledky propojení počítačů do celosvětové sítě – s tím, že je bude možné využít jako jeden superpočítač, který bude využívat periodických změn lokálního zatížení jednotlivých počítačů v síti v průběhu dne.

Poroste samozřejmě i výkon počítačů (rychlost, šířka slova …), a to vzdor tomu, že Moorův zákon brzy narazí na fyzikální meze možností. Důsledky nejspíš nelze odhadnout.

6.2 Začíná éra metaprogramování

Vývoj programovacích jazyků musí odrážet požadavky současného hardwaru a nejběžnějších druhů aplikací. Již několikrát se zdálo, že jejich vývoj je v podstatě ukončen, ale vzápětí se objevily poměrně zásadní novinky; jako příklad lze použít např. zavedení objektových typů v jazyce Simula. Lze tedy očekávat, že i v budoucnu se do syntaktické roviny budou přenášet konstrukce, které se dnes řeší např. pomocí knihoven.

Dnes jsme svědky nástupu jazyků kompilovaných pro virtuální stroj (jazyky překládané do bajtového kódu Javy nebo do IL), které přinášejí alespoň zdánlivou nezávislost na platformě. Překlad pro virtuální stroj přináší něco, co vypadá na první pohled samozřejmě, ale naprosto to samozřejmé není – binární kompatibilitu přeložených programů v tom smyslu, že např. od třídy definované v jednom jazyce lze odvodit potomka v jiném jazyce.

Největší vývoj lze očekávat v oblasti IDE. Mezi dnes nepříliš dobře zvládané problémy patří např. ladění vícevláknových aplikací. Běžnou záležitostí se zřejmě stane integrace nástrojů, dříve označovaných jako CASE, i do lacinějších verzí IDE, podpora vývoje všech druhů aplikací pro web atd. IDE přenesou tvorbu aplikací do roviny metaprogramování (generování programů), neboť postupně skryjí před programátorem používaný programovací jazyk.

6.3 Metodiky

Je těžké odhadovat budoucí vývoj metodik objektového návrhu. Lze ale očekávat, že standardem (a požadavkem, kladeným na řadového programátora) se stane znalost běžných návrhových vzorů a technik refaktoringu. Softwarové firmy začnou standardně využívat některé z technik agilního programování (extrémní programování aj.).
Z učebnic programování už doufejme zmizí kaskádní (vodopádový) model vývoje softwaru.

6.4 Evoluční programování

Skutečným problémem dnešního programování je zvládání vnitřní složitosti řešených úloh. Je pravděpodobné, že se programátoři inspirují způsobem, jakým podobné situace řeší příroda – pomocí mechanizmu evoluce. Lze si např. představit „populaci“ podobných programů, které řeší týž problém a přitom soupeří o možnost předat své „geny“ (nějaké vyjádření algoritmů, struktury, návaznosti stavů atd.) další generaci, přičemž kromě křížení se uplatňují i náhodné malé mutace. Po jistém počtu generací si z nich programátor vybere program, který nejvíce vyhovuje jeho požadavkům.

6.5 Nová a znovuobjevená paradigmata

Šablonové metaprogramování je zatím omezeno na C++; nicméně jakousi analogii šablon má obsahovat i JDK 1.5 a některá z pozdějších verzí IL. Nechme se překvapit, zda v těchto prostředích půjde jen o nástroj pro posílení typové kontroly nebo zda – podobně jako v C++ – otevře další možnosti. I když v současné době jde o akademickou záležitost, nelze vyloučit, že ve vzdálenější budoucnosti se stane obecně přijímaným paradigmatem. Ostatně OOP bylo čistě akademickou záležitostí více než 10 let.

Také aspektově orientované programování je zatím v počátcích – zabývají se jím především teoretici, s jeho implementací programovacích jazycích se setkáme jen výjimečně (AspectJ, AspectC++ [8]).
Agentově orientované programování ve skutečnosti není ničím novým, neboť s agenty jako s programovými entitami se setkáváme nejen v prostředí operačního systému UNIX již dlouhou dobu. To ale neznamená, že se tento přístup k tvorbě distribuovaných aplikací nemůže stát v budoucnu dominantním.

6.6 Budeme programovat v XML?

Rozšiřitelný značkovací jazyk XML byl původně navržen jako jazyk pro publikování a výměnu dokumentů. Poměrně rychle se však rozšířil i do jiných oblastí, takže v současné době se používá k popisu komponent, k popisu nasazení distribuovaných aplikací atd. Mnohé z těchto oblastí lze již bez nadsázky označit za programování. Ve skutečnosti ale XML ve spojení s vhodným DTD umožňuje popsat téměř cokoli, takže si lze snadno představit, že se stane jakýmsi univerzálním programovacím jazykem:

```
<class name="Okno" base="Frame" visibility="public">
     <data name="i" type="int" value="12"/>
     <method... >
          <!-- .... -->
     </method>
</class>
```

I když to vypadá zbytečně složitě, lze XML využít jako nástroj pro vytvoření univerzální syntaxe společné pro různé programovací nástroje – a to může být za jistých okolností užitečné.

6.7 Budeme programovat?

Poslední otázka může vypadat absurdně, neboť s počítači se dnes setkáme prakticky všude – od zbraňových systémů přes osobní automobily a fotografické přístroje až po ledničky a pračky. Přesto odpověď nemusí být kladná.

Jednou z možností je, že se potvrdí předpověď statistického modelu růstu lidské populace, který prezentoval prof. S. Kapica v Praze v červenci 2003 [6]. Podle něj není růst počtu obyvatel exponenciální, ale hyperbolický. V důsledku toho dojde v průběhu nejbližších dvaceti let v lidské společnosti k analogii fázového přechodu, k něčemu, jako je tání ledu, tedy např. k rozpadu států a státní moci. (První náznaky se již objevují…) Jeho důsledky budou podle Kapicova modelu pro populaci v Evropě a v severní Americe velmi nepříznivé, původní obyvatelstvo těchto zemí v podstatě vymizí – a mimo tyto oblasti počítače příliš nepronikly.

Lze si ale představit i jiné důvody, proč by mohlo programování zaniknout nebo téměř zaniknout. Snahy o zavedení patentového práva do oblasti programování by mohly skončit absurdními situacemi, kdy bychom studentům při přednáškách říkali něco jako „cyklus je velice účinný nástroj, ale nesmíme ho použít, protože je chráněn tím a tím patentem… V těchto snahách jde nepochybně také o skrytý boj proti freewarovým programům.

Počítač také může začít hrát roli orwellovského „Velkého bratra“, který nás sleduje na každém kroku. Přitom nemusí jít o politiku, ale o zájmy firem. Už dnes se objevují náznaky –

např. jistý přehrávač videozáznamů posílá (prý statistické) informace o tom, co právě přehráváme. Umíte si představit programování v takovém prostředí?

To je však problém celkové krize etiky, netýká se jen programování.

Literatura:

1. Petzold, C.: Code – The Hidden Language of Computer Hardware and Software. Microsoft Press, Redmond 2000. ISBN 0-7356-1131-9
2. Dijkstra, E.: Goto Statement Considered Harmful. Communications of the ACM, vol. 11, No. 3 (March 1968), p.147
3. Jackson, M.A.: System Development. Prentice Hall 1983
4. Yourdoun, E., Constantine, L: Structured Design. Yourdon Press, 1978
5. Booch, G., Rumbaugh, J., Jacobson, I.: The Unified Modeling Language Reference Manual. Addison-Wesley 1999. ISBN 0-201-30998-X
6. Kapica, S.: The Statistical Theory of Global Population Growth. Přednáška na konferenci Advanced Study Institute – Spin and Symmetries, Praha 15. 7. 2003
7. Virius, M.: Programování a programová prostředí. In Jazyk Pascal po 25 letech, VUT Brno, 1998. ISBN 80-214-1234-8
8. http://aosd.net/technology
9. Vysoký, P.: Počítače z Loretánského náměstí. http://www.vumscomp.cz/Svoboda.html (Převzato z časopisu Vesmír 11/1999)