Trocha historie – 3. část

V posledním dílu povídání o programování a o věcech, které s ním souvisí, se zaměříme na čím dál tím významnější úlohu vývojového prostředí.

Třetí vrstva abstrakce

V současné době se zdá, že samotné programovací jazyky nebo nové metodiky návrhu už nemohou dále zjednodušit a zrychlit vývoj aplikací. Tuto roli nyní přebírají podpůrné nástroje, které umožňují automatické generování kódu, a vývojová prostředí (IDE), v nichž jsou tyto nástroje často integrovány.

Basic

Programátoři se mohli s vývojovým prostředím poprvé setkat už před více než 30 lety, neboť první takové prostředí s sebou vlastně přinesl Basic. Tento jazyk byl totiž navržen jako interpreter, který umožňoval spouštět části programu a hned vidět jejich výsledky. Díky tomu mohli programátoři pracovat interaktivně, a tedy daleko efektivněji, než bylo obvyklé.

Připomeňme si, že v té době bylo pravidlem dávkové zpracování programů: programátor dodal operátorům zdrojový text programu na děrných štítcích (nebo třeba na magnetické pásce), vypsal průvodku a po nějakém čase si přišel pro vytištěné výsledky. Interaktivní Basic tedy představoval zásadní průlom, ovšem k tomu vyžadoval technické zázemí – terminál, který by interaktivní práci umožňoval. Výhody Basicu tedy začaly být zřejmé až ve chvíli, kdy se interaktivní terminály staly běžnou součástí výpočetních středisek, a později, kdy se implementace Basicu staly standardní součástí vybavení mikropočítačů. (Není bez zajímavosti, že na mnohých počítačích Basic dokonce nahrazoval operační systém.)

Je ovšem také pravda, že Basic byl jazyk, který se ke skutečným výpočtům příliš nehodil. (S dnešním Visual Basicem nemá kromě názvu a několika klíčových slov mnoho společného.)

Smalltalk

Vývojové prostředí bylo rovněž nedílnou součástí jazyka Smalltalk. Obsahovalo nejen nástroje pro vytvoření programu, ale i nástroje pro jeho ladění; vývojové prostředí bylo dokonale integrováno s jazykem, tj. skládalo se z objektů, které mohl programátor použít i v programu. Podobným způsobem byla vytvořena integrovaná prostředí i některých jiných čistě objektových jazyků.

Vývojová prostředí na PC

V našem povídání nelze opominout vývojová prostředí nejběžnějších překladačů na PC – už vzhledem k jejich masovému rozšíření. Mám tím na mysli např. prostředí Turbo Pascal, Microsoft C a další nástroje.

Ve srovnání s prostředím Smalltalku (nebo i s dnešními podobami špičkových vývojářských nástrojů) byla první integrovaná vývojová prostředí (IDE) značně jednoduchá a nevýkonná, ale i přesto nabízela mnohem více než tradiční příkazová řádka. Zpočátku tato IDE obsahovala vlastně jen textový editor, překladač, nástroje pro nastavování voleb a nástroje pro práci s projekty a se soubory. K nim později přibyly prostředky pro symbolické ladění, pro vytváření pomocných datových souborů (prostředky – resources – ve Windows) aj.

Šamani

Při programování aplikací pro Windows (ale i jiných) se ukázalo, že velká většina programů dodržuje jedno základní schéma. V “oknech” je to např. funkce WinMain, obsahující popis a registraci třídy okna, vytvoření, zobrazení a překreslení okna a cyklus výběru zpráv z fronty. Také “okenní” procedura je vlastně povinnou součástí programu a má své ustálené schéma – jeden příkaz, který podle druhu došlé zprávy určuje, co se má stát. Podobně existují ustálená schémata i při vytváření distribuovaných aplikací.

Velice brzy proto začala IDE nabízet služby různých nástrojů, které se podle okolností nazývají Wizard, Expert, SmartGuide atd. Dovolím si o nich nadále hovořit jako o “šamanech”. Po vyvolání šamana se většinou objeví dotazník v podobě několika dialogových oken, která zjistí, co si vlastně přejeme. Podle toho pak IDE vytvoří kostru aplikace, v níž komentářem označí místa, na která má uživatel doplnit svůj zdrojový kód.

Šamani mohou podstatným způsobem zrychlit programování. Nejen proto, že za nás napíší třeba i stovky řádků zdrojového textu, ale hlavně proto, že je napíší syntakticky i sémanticky správně (samozřejmě pokud zadáme správné údaje). Na druhé straně ovšem mohou být poněkud svazující – např. uživatelé MS Visual C++ 5.0 si jistě vzpomenou, že aplikace vytvořená AppWizardem využívajícím knihovnu MFC musela vycházet z architektury dokument/pohled, a pokud jsme chtěli něco jiného, museli jsme “ručně” část vytvořeného kódu změnit.

Většina šamanů využívá speciální (nestandardní) knihovny dodávané s překladačem – IOC (IBM), MFC (Microsoft a dnes i jiné překladače), OWL (Borland), VCL (Borland) atd. To ale znamená, že uživatel musí vedle standardu použitého programovacího jazyka znát nejen danou knihovnu, ale i strukturu aplikace, kterou daný nástroj vytvoří.

Vizuální programování

V devadesátých letech se začal při programování uplatňovat nový postup – vizuální programování. Za první náznak bychom mohli považovat nástroje na vytváření prostředků (resources) určených pro programy pro Windows, jako byl např. Resource Workshop, které umožňovaly vytvořit některé součásti programu vizuálně pomocí myši na základě předdefinovaných součástí.

Poprvé jsme se s vizuálním programováním mohli ve významné míře setkat v prvních verzích Visual Basicu a později ve vylepšené podobě v nástrojích, jako je Delphi, C++ Builder, JBuilder, Power++, PowerJ, VisualAge apod. Princip je dnes už dobře známý: nejčastěji používané složky aplikací jsou

zapouzdřeny do komponent, obvykle implementovaných jako objektové typy. IDE je zpravidla nabízí na “paletách”, kde je reprezentují ikony.

Základní postup při vizuálním programování lze shrnout asi takto: Po vytvoření nového projektu (nebo na naši žádost) otevře IDE prázdné okno aplikace a vytvoří k němu odpovídající zdrojový kód. Do nabídnutého okna můžeme pomocí myši vkládat komponenty, které si vybereme z palet. Přitom prostředí generuje odpovídající zdrojový kód.

Většinu vlastností komponent můžeme nastavit ve speciálním okně již v době návrhu. IDE přitom obvykle ihned změní odpovídajícím způsobem zdrojový kód programu a vizuální návrh programu. Uživatel pak musí definovat odezvy programu na události, které mohou nastat – nejčastější událostí je “stisknutí tlačítka” v uživatelském rozhraní. I tady nám může IDE výrazně pomoci; může např. nabízet řadu předdefinovaných procedur typických pro určité komponenty nebo pro vztahy mezi nimi (uzavřít okno po stisknutí tlačítka apod.).

Také zde jsou výhody zřejmé – čím více kódu generuje IDE automaticky, tím rychleji programátor vytvoří aplikaci a tím méně má příležitostí k chybám. Přitom může ve skutečnosti umět méně, než kdyby programoval “klasicky”; např. v Delphi můžete napsat jednoduché databázové aplikace a znát přitom jen základní vlastnosti nějakých tří databázových komponent.

Zdá se tedy, že programování založené na komponentách nabízí řešení softwarové krize. Ovšem, jako obvykle, není to zadarmo. Programování založené na komponentách lze velice dobře přirovnat ke stavbě z panelů. Je rychlé a snadno zvládnutelné, ale drahé – vyžaduje speciální nástroje a výkonný počítač. Vytvořené programy jsou totiž zpravidla podstatně rozsáhlejší než podobné programy založené na “obyčejném” překladači a aplikačním rozhraní operačního systému. Dosažená řešení také nemusí být vždy tak kvalitní, jak bychom si přáli, neboť občas je třeba přizpůsobit se možnostem prefabrikovaných dílů (komponent).

CASE

Zejména při programování databázových aplikací založených na relačních databázích se setkáme s další variantou vývojových prostředí. Zde už je postup vývoje víceméně standardizován: Prvním krokem je vytvoření konceptuálního datového modelu, ve kterém se popíší jednotlivé entity, relace mezi nimi atd. Druhým krokem je vytvoření fyzického datového modelu, v němž již vezmeme v úvahu vlastnosti použitého databázového serveru. Zde na základě konceptuálního modelu popíšeme jednotlivé databázové tabulky, indexy, dotazy atd., které chceme ve své aplikaci použít. Ve třetím kroku se pak pomocí fyzického datového modelu vytvoří databázové skripty pro vytvoření databází, tabulek a indexů, dále dotazy v jazyce SQL atd.

Konceptuální datový model se obvykle popisuje graficky pomocí tzv. entitně-relačních diagramů; potřebné nástroje jsou dnes obvyklou součástí prostředí pro vývoj databázových aplikací. Na ně pak obvykle navazují nástroje pro převod konceptuálního datového modelu na datový model fyzický. Vytvořený fyzický datový model lze samozřejmě ještě upravit a na jeho základě pak automaticky generovat program. (Takový nástroj nabízí např. Oracle Developer 2000.) Podobným způsobem lze dnes navrhovat i vícevrstvé aplikace – známý je např. produkt Rational Rose.

Ještě v nedávné době byly nástroje tohoto druhu nesmírně drahé, takže si je mohly dovolit pouze velké firmy. Dnes se zvolna začínají stávat součástí běžných vývojových prostředí, i když zpravidla ve

značně omezené podobě. Např. součástí Visual C++ 6.0 Enterprise je Visual Modeller, který představuje omezenou verzi Rational Rose.

Ladění

Ruku v ruce s programováním jde vždy nezbytné ladění. Hledání logických chyb v programech bylo a je jednou z nejobtížnějších, nejprotivnějších a nejzdlouhavějších fází vývoje programu.

V prvních dobách byly základním nástrojem ladicí tisky a analýza zdrojového textu. Některé překladače k tomu nabízely možnost výpisu křížových referencí, tj. výpisu identifikátorů a míst jejich použití. Operační systémy také zpravidla poskytovaly v případě chyby výpis operační paměti v osmičkové nebo v šestnáctkové soustavě.

Možnosti, které z hlediska ladění představují vývojová prostředí, naznačil v polovině 60. let Basic. Vedle samostatného spouštění a testování jednotlivých částí programu nabízely některé implementace i možnost “animace”, tedy běhu programu s přestávkou po jednotlivých příkazech.

Skutečný rozvoj ladicích nástrojů přinesly ale až osobní počítače, neboť vzhledem k velikosti trhu mohly být podpůrné programy snadno dostupné. První ladicí programy byly samostatné, nebyly integrovány do vývojového prostředí a umožňovaly krokovat program (provádět odděleně jednotlivé příkazy), vkládat do něj zarážky a sledovat přitom hodnoty vybraných proměnných, případně výrazů. Princip ladicího programu je ve skutečnosti jednoduchý: Ladicí program nahradí instrukci laděného programu, před kterou se má program zastavit, instrukcí, která vyvolá přerušení, a původní instrukci si uschová. Jakmile laděný program na toto místo doběhne, dojde k přerušení a ladicí program převezme řízení. Nahradí instrukci přerušení původní instrukcí a čeká na pokyny uživatele.

S rostoucími nároky na programy rostly ovšem také nároky na ladicí nástroje. Ke standardnímu (dnes vlastně už asi minimálnímu) vybavení ladicích programů patří mj. následující možnosti:

• Krokování programů se vstupem do podprogramů nebo bez něj, a to na úrovni zdrojového textu nebo disasemblovaného přeloženého programu.

• Používání zarážek vázaných na místo, na počet průchodů, na splnění dané podmínky, na změnu hodnoty jisté proměnné atd.

• Průběžné sledování hodnot výrazů, lokálních proměnných, výsledků funkcí apod.

• Prohlídka operační paměti, stavu registrů, zásobníku, posloupnosti volání podprogramů apod. za běhu programu.

• Možnost měnit obsah proměnných za běhu programu.

• Ladění procesu běžícího na jiném počítači, schopnost ladicího programu připojit se k běžícímu procesu.

• Poskytování informací o jednotlivých vláknech (threadech) běžící aplikace.

Vedle toho se dnes zvolna stává samozřejmostí možnost editovat zdrojový text v průběhu ladění s tím, že se změny ihned promítnou do chování programu (alespoň v omezené míře). Ladění distribuovaných aplikací vyžaduje také možnost přecházet průběžně mezi jednotlivými běžícími programy (i na různých počítačích), popřípadě i mezi různými programovacími jazyky.

Třetí vrstva

Dnes je již zřejmé, že integrovaná vývojová prostředí začínají hrát roli třetí vrstvy abstrakce mezi programátorem a hardwarem počítače. Jak víme, první vrstvou byl asembler, který zbavil programátora nutnosti starat se o konkrétní adresy v programu. Druhou vrstvu pak představují vyšší programovací jazyky, které nabídly vyjadřování v jazyce podstatně bližším člověku – nebo spíše řešenému problému – než stroji. Program zpravidla alespoň vzdáleně připomíná anglické věty, matematické zápisy apod.

Třetí vrstva pak zbavuje programátora závislosti na programovacím jazyku alespoň v prvních fázích vývoje programu. Umožňuje mu vyjadřovat své představy o funkci programu pomocí schémat a diagramů, sestavovat uživatelské rozhraní i některé funkční bloky aplikace z prefabrikovaných celků nebo specifikovat své představy o funkci budoucího programu vyplněním dotazníku apod.

V současné době však nemůžeme hovořit o skutečné nezávislosti programátora na programovacím jazyku. I když ve většině prostředí můžeme dnes nejjednodušší aplikace vytvořit, aniž bychom napsali jedinou řádku kódu, pro naprogramování čehokoli použitelného musí programátor umět jazyk, který stojí v pozadí, a v něm napsat těla řady procedur. IDE za něj ovšem napíše značnou část kódu a v mnoha případech ho zbaví potřeby detailně rozumět struktuře aplikace v daném prostředí (například v Delphi lze napsat plnohodnotnou aplikaci, aniž bychom něco věděli o fungování cyklu zpráv ve Windows, o způsobu překreslování oken, apod.). Navíc polotovary vytvořené v první fázi návrhu (uživatelské rozhraní programu, konceptuální datový model apod.) mohou být přenositelné mezi různými vývojovými nástroji, v současné době alespoň v rámci produktů jedné firmy.

Třetí vrstva tedy ještě není úplná, druhou vrstvu – programovací jazyk – je pod ní stále ještě velmi silně znát, ale přesto se další stupeň abstrakce začíná výrazně uplatňovat.

Na druhé straně ovšem vstupují do hry i další vrstvy, a to na úrovni procesoru. Z kdysi poměrně jednoduchého zařízení se stává nástroj, který se stará o ochranu paměti, o práci s virtuální pamětí, o výběr a přeuspořádávání instrukcí tak, aby se daly provádět paralelně, a přitom zůstal původní význam programu zachován. A objevují se už dokonce i úvahy o hardwarové implementaci garbage collectoru.

Co dodat

Naše třídílné povídání o programování se točilo kolem programovacích jazyků, stylů programování a vývojových prostředí. To samozřejmě nejsou všechny faktory, které způsob práce programátora ovlivňují. Navíc jsme se mnoha důležitých věcí jen dotkli, další podstatné jsme vynechali. Můj dojem je, že v současné době ustupuje význam programovacích jazyků do pozadí a vlády se ujímají nástroje pro vizuální programování.

Možná z toho bude mít řada lidí dojem, že programátoři – ta značně nepohodlná sorta lidí, kterým je třeba dobře platit, a kteří se přitom jen zřídka chovají tak, jak by si jejich šéfové přáli – už konečně zmizí v propadlišti dějin. Jenže programování není vlastně záležitost programovacího jazyka nebo ER diagramů; programování vyžaduje porozumění problému na jedné straně a možnostem počítačů na straně druhé, a přitom není příliš podstatné, ve kterém jazyce se vyjadřujeme; zda píšeme příkazy ve Fortranu, nebo zda kreslíme nějaké diagramy. Možná za několik let postačí, když si s počítačem prostě popovídáme – ale bude záležet na tom, co a jak mu to řekneme. Problém je v tom, že s rostoucím

výkonem a klesající cenou počítačů sice na jedné straně závratně rostou možnosti, které vývojové nástroje poskytují, ale na druhé straně ještě rychleji rostou požadavky na software.

Je těžké být prorokem

Počítače a vše, co s nimi souvisí, prošly vývojem, pro jehož rychlost snad neexistuje v dějinách techniky analogie. Proto je těžké cokoli předvídat. Velice pěkně to ukazuje výrok, který otiskl časopis Popular Mechanics v r. 1949, tedy v době, kterou mnozí ještě pamatují: “V budoucnosti možná nebudou počítače těžší než půldruhé tuny...”

Miroslav Virius