Vývoj programování a jeho další cesta

aneb

Od selského rozumu k objektům

Pavel Drbal

Programování prošlo od svého vzniku složitou, a zdá se, že i poučnou cestou. Trochu se po ní ohlédneme - pro zábavu i pro poučení - závěrem provedeme předběžné hodnocení významu objektového myšlení.

1. Dávnověk

Za vznik programování lze považovat geniální nápad von Neummanna, že instrukce je možné ukládat do paměti stejně jako data.

Poznámka. Všimněte si, že tentýž nápad se objevil v nedávné minulosti ještě jednou - jako programovatelné kalkulačky.

Hned ze začátku se ukázaly základní způsoby, jak ušetřit programátorskou práci:
- podprogramy (otevřené a uzavřené, tj. makra a procedury),
- překladače (interpretační i kompilační),

Poznámka. Počítače byly uzpůsobené pro počítání s čísly, proto některé programátorské obraty měly poněkud kuriózní charakter. Neexistovaly například instrukce pro práci s podprogramy, co však existovalo od počátku, byla instrukce skoku. Před odskokem do podprogramu se provedla instrukce skoku na následující instrukci, a ta se umístila na smluvené místo. Na závěr podprogramu se k této instrukci přičetla určitá konstanta (která zajistila návrat za odskok do podprogramu) a na takto vytvořenou instrukci skoku se skočilo.

Dnes se nám překladače jeví tak, že jejich základní myšlenkou je umožnit programátorovi formulovat problém v jemu blízkém jazyce a zápis v tomto jazyce automaticky převést do jazyku počítače. Tehdy však byla existence překladačů absolutní nutnost. Bez byť i primitivního překladače bylo totiž nutno programovat v absolutních adresách, tj. program musel být psán přesně tak, jak bude umístěn. Napsat takový program není nic těžkého, ale velká potíž je jej opravovat. Má-li se do takového programu vsunout byť jen jedna jediná instrukce, je nutno některou instrukci nahradit odskokem na volný kus paměti, tam uvést nahrazenou instrukci a novou instrukci a odskočit zpátky za nahrazenou instrukci. Dokážete si představit, jak takový program vypadal po několika opravách.

Poznámka. Tento "absolutní" způsob programování se používá dodnes. Aby se vyloučila nutnost složitých překladů velkých informačních systémů, existují prostředky jak nahradit instrukce v již přeložených programech. Často se tyto prostředky jmenují PATCH (česky ZÁPLATA). Programátoři mají za povinnost nechávat v každém programu několik set bytů volných pro případné záplaty. Počítače nebyly příliš rychlé, překlad rozumně použitelného programu trval i přes hodinu. Již dávno (v 50-tých letech) byla objevena obrana - oddělený překlad. Program se dělil na menší části, ty se překládaly samostatně, a speciální program je spojoval v žádoucí celek. Tento rys překladu se zachoval dodnes.

Za nejvýraznější objev té doby považuji zásobník (stack), který umožnil rozumný překlad aritmetických výrazů a nakonec se osvědčil tak, že se stal součástí současných počítačů.

2. Strukturovaný život

Počítače se staly výkonnými a vznikla potřeba vytvářet velké programové systémy. Tady se ukázal základní problém programování - programování je boj lidského rozumu se složitostí jevů. Člověk si se složitostí problému umí poradit vlastně jen jedním způsobem - rozdělit složitý problém na několik jednodušších, které již zvládne. Cest bylo několik.

Dvě úrovně programování

Programování se rozdělilo na "programování v malém", což je programování v běžných programovacích jazycích, a na "programování ve velkém", což je svazování programových modulů ve fungující celek. Toto modulární programování zůstalo jedním ze základních nástrojů programátorů dodnes.

Normované programování

Skupina pracovníků severoamerických vysokých škol analyzovala několik set osvědčených programů a vytvořila zobecněnou strukturu programu - základní cyklus. Vytvořila se spousta generátorů normovaných programů (každý slušný programátor měl svůj vlastní universální generátor) i specializovaný jazyk RPG. Základní vlastností normovaného programování bylo to, že činnost programu se řídila přečtenými daty. Jeho popularita klesla při přechodu od dávkového zpracování k interakčnímu, vznikla však obdoba - program řízený událostmi.

Strukturované programování

Typicky americkou akcí je snaha zvýšit produktivitu práce. Při zjišťování produktivity programátorů se přišlo na zajímavý jev - čím horší program, tím více příkazů GOTO obsahuje. Nejdříve vzniklo poněkud rozpačité hnutí GOTOless programming, Pak ale pan Dijkstra formuloval zásady dobře strukturovaných programů a pan Wirth vytvořil speciální jazyk pro výuku strukturovaného programování - Pascal.

Abstraktní data

Při řešení problémů modulárního programování vznikla myšlenka abstraktních dat (přesněji řečeno abstraktních datových typů). Podstatou je to, že data nejsou definována svou strukturou, ale že jsou definována svým použitím, že jsou definována operacemi, které vyžaduje uživatel. Takové představy dosti odporují zdravému selskému rozumu, což se také projevilo tím, že abstraktní data zůstala skoro výhradně na akademické půdě. Normované i strukturované programování proniklo do praxe, ale abstraktní data skoro ne.

3. Profesionalizace

Tvorba velkých programových systémů byla velmi obtížnou a nákladnou. Vylepšování stylu programování nepřinášelo výrazná zlepšení. Převládl názor, že problémy nejsou ve vlastním programování, ale v organizaci práce. Vznikla metodika pro tvorbu velkých programových systémů (nyní se označuje jako strukturální přístup nebo strukturálně orientovaný), která vycházela z rozboru životního cyklu programu (systému). Její podstatou je: nejdříve určit cíl, pak přemýšlet a modelovat a teprve pak programovat. Důsledkem je stratifikace programátorské činnosti a tedy rozrůznění programátorské profese na několik specializací. Přesně podle americké zásady - jestliže něco neumím, tak si na to vyškolím specialisty, ať se starají oni.

V té době již došlo k masovému rozšíření osobních počítačů a převaze interakčního zpracování nad dávkovým. V současné době programování zahrnuje několik desítek profesí.

V podstatě se jedná o to, že "dvojúrovňové" programování (ve velkém a malém, viz výše) se rozrostlo do většího počtu úrovní - základní jsou analýza, design (ve velkém) a implementace (v malém). Celkově má vývoj většího informačního systému několik desítek etap (ne všechny jsou programátorské) a je zapotřebí většího počtu specialistů (nebo lidí s několika specializacemi). Doby osamělých vlků, kteří psali geniální programy, ty už většinou minuly.

Poznámka. Navíc vidíme, že ve světě je několik inovačních center, ze kterých se šíří nové ideje, Evropa však mezi ně nepatří.

4. Objektově orientovaný život

Za nejdůležitější přínos objektů považuji skloubení různorodých programátorských idejí v jeden logický celek. Když se podíváme podrobněji na základní vlastnosti objektů, vidíme že vše již zde bylo.

- zapouzdření - základní myšlenka abstraktních datových typů a odkládání rozhodnutí (programming hiding),
- spojení dat a funkcí v jeden celek - myšlenka abstraktních datových typů,
- dědění - zveřejněno v Simule (Simula je jeden simulační programovací jazyk),
- virtuální funkce - myšlenka odkládání rozhodnutí (programming hiding),
- přátelské funkce a veřejně přístupná data - poznatek, že žádná pravidla nesmí být rigidní, že je nutno dopustit jejich porušování (například zpětné sledování v technologii strukturovaného programování),
- objektově orientované jazyky - jakákoliv idea má naději na úspěch, pouze je-li podporována programovým systémem nebo programovými jazyky.

Spojení všech těchto rysů v jeden celek je ovšem geniální počin. Není divu, že užití objektů doznalo takového rozšíření a popularity. Na druhé straně módnost objektů vede ke vzniku různých mýtů a obřadů. Je nutno si uvědomit, že objekty se používají za různými účely - a tedy také různým způsobem. Typickým zdrojem zmatků je to, že určitý způsob použití se užívá v oblasti, kam nepatří.

Nejdůležitější oblasti jsou tyto:

- programování
- tvorba nástrojů
- návrh systémů

4.1 Objektové programování

Objektově programovat znamená používat objekty jako stavební kameny, ze kterých se staví program. Nějaké dědění zde nemá žádný smysl. Daleko důležitější vztahy zde jsou:

- obsahuje,
- vlastní,
- používá,

a vztahy inverzní - je obsažen, je vlastněn, je používán.

4.1.1 Jeden objekt obsahuje jiné objekty

Tak jako pole obsahuje prvky nebo záznam obsahuje údaje, tak může jeden objekt obsahovat jiné objekty. Kontejner říkáme tomu objektu, který obsahuje proměnné množství jiných objektů (množství proměnné v čase výpočtu).

Objekt může obsahovat jiné objekty zásadně dvěma způsoby:

1) Objekt je integrální částí jiného objektu (podobně jako data). Podřízený objekt vznikne při vzniku nadřízeného objektu a zanikne až se zánikem nadřízeného objektu.
Poznámka. Procedura nebo funkce objektu se označuje jako "metoda", pro data objektu nebyl zaveden všeobecně přijatý název. Někteří anglosasové označují data objektu jako "sloty". Při použití této terminologie můžeme říci, že slotem objektu může být číselný údaj, řetězec, záznam nebo jiný objekt.

2) Nadřízený objekt je kontejner, tj. obsahuje dynamický seznam jiných objektů, které mohou být během výpočtu přidávány nebo ubírány. Tento vztah může být rekurentní, podobně jako je rekurentní definice pole (tj. existují vícerozměrná pole).

Kontejner je něco více než pole objektů, kontejner má svou vlastní "individualitu", má své vlastní metody, které například zacházejí s podřízenými objekty.

4.1.2 Jeden objekt vlastní jiné objekty

Objekty typu ad 1) z předchozího odstavce jsou nejen obsaženy, ale i vlastněny. U kontejnerů to není samozřejmostí. Obsažený objekt nemusí být vlastněn. Jeden objekt může být umístěn ve více kontejnerech současně. Podobně třeba člověk může být členem rodiny, pracovního kolektivu i sportovního klubu. Pojem "vlastnění" si objasníme na příkladě: kontejner vlastní obsažené objekty, jestliže se zánikem kontejneru zaniknou i obsažené objekty. Nevlastnící kontejner může zaniknout a původně obsažené objekty existují dále.

Technická poznámka: kontejner většinou obsahuje ukazatele na objekty, ne objekty samotné. Proto může být existence kontejneru nezávislá na osudu obsažených objektů.

Někdy se pojem vlastnění ztotožňuje s porodem objektu, existuje myšlenkový proud, který tvrdí, že objekt může být vlastněn pouze tím kontejnerem, který dal objektu vzniknout. Tato představa mi připadá poněkud kuriózní.

Poznámka. Promítnu-li si tento názor na své nejbližší okolí, pak, ačkoliv jsem hrál určitou roli při vzniku své dcery, nemám pocit, že ji vlastním. Pokud ji vůbec někdo vlastní, tak to je nejspíš její muž.

Pojem vlastnění je trochu vágní, ve větších programech je ale zapotřebí udělat si mezi objekty pořádek, jinak nám v run-time budou bloudit objekty, které k ničemu nejsou.

4.1.3 Jeden objekt používá jiný objekt

To je základ programování. To, že jeden objekt použije jiný objekt znamená, že metoda jednoho objektu vyvolá metodu jiného objektu. Nic víc to není.

Poznámka. Z historických důvodů se tomuto volání metody říká "poslat zprávu" (send message). Důvod je prehistorický. V prvních programově orientovaných jazycích nebyly objekty integrální součástí programovacího jazyka, ale byly obhospodařovány samostatným objektovým systémem. Metoda objektu se volala tak, že se vyvolala funkce "message" tohoto systému. v této funkci bylo několik parametrů - jeden z těchto parametrů byla identifikace objektu (jeho jméno), druhým parametrem jméno metody, třetí a další parametry byly parametry určené pro metodu. Pak se poněkud nadneseně říkalo, že systém "sám" pozná, který objekt se má vybrat a která metoda se má provést. Samozřejmě se vybral objekt a metoda, jejichž jména byly uvedeny jako parametry funkce "message". Teď se s objekty zachází poněkud moderněji a více programátorsky.

Na vlastní technice programu se nic nemění. Například volání metody BFLM objektu A s parametry B a C se zapisuje v Pascalu i C++ takto:

A.BFLM(B,C)

ale uvnitř překladače se to chápe takto:

BFLM(B,C,A)

O trochu vzruchu se postarají nepřístupná data, chci-li pracovat se soukromými daty objektu, musím si udělat metodu, která mi je podá.

4.1.4 Tvorba objektů

Program není nic jiného, než síť objektů. Chci-li objekty používat, musím je mít. Objekty mohu získat třemi způsoby:

- někdo už je připravil (tj. připravil objektové typy) a já je pouze použiji,
- objekty si vytvořím sám,
- vytvořím si nástroj pro tvorbu objektů pro můj program.

Výše uvedené dělení je velmi schematické, v praxi nejde jednotlivé způsoby od sebe úplně odlišit. Připravené nástroje jsou dosti obecné, pro konkrétní použití je většinou zapotřebí je upravit - dodělat. Vytvářím-li si objekty sám, snažím se ušetřit svou práci pomocí dědění - vytvářím nástroj. Přesto je zapotřebí si uvědomit, že práce musí jít v následujících krocích:

- V analýze zadání zjistit, jaké objekty budu potřebovat,
- vytvořit si tyto objekty,
- naprogramovat program.

Často se postupuje neuspořádaně, začne se programovat, pak se začne zdát, že je zapotřebí nějaký objekt, tak se začnou vytvářet objekty, po dalším programování se již vytvořené objekty mění - výsledek obvykle vypadá podle toho.

Jiní intenzívně přemýšlejí, jak vše realizovat objekty, i to, co je vysloveně neužitečné, takže program pak je vysloveně absolutně objektový, ale pořád šoupe s objekty místo aby něco dělal.

4.2 Tvorba objektových nástrojů

Objektovým nástrojům se většinou říká knihovna tříd, protože metoda jejich organizace je hierarchie tříd, kde třída dědí od svého předka vlastnosti (tj. metody a sloty), přidá k tomu něco vlastního a to vše poskytne svému potomku, aby to použil (po případné úpravě). Skutečné objekty se většinou odvozují z tříd na nejnižší úrovni hierarchie.

Třída - to je datový typ. Hierarchie tříd je způsob velmi efektivního využití programového kódu. Uvedeme si dva mechanismy.

Mohu například realizovat nějaký algoritmus jako soubor metod třídy (tj. metod budoucího objektu), z toho algoritmu však mohu vydělit proměnlivé části, které nenaprogramuji. Tyto části si doprogramuje uživatel třídy - programátor - podle svých potřeb.

Například mohu vytvořit třídicí algoritmus, nerealizuji však metodu, která porovnává dva prvky. Předpokládám však, že existuje metoda, která zjistí, který prvek ze dvou je menší. Tuto metodu doprogramuje programátor podle toho, jaké prvky třídí a jaké pořadí požaduje. S vynaložením malé námahy získá vyzkoušený algoritmus třídění speciálně šitý pro jeho potřeby.

Nebo třeba existuje třída, která má požadované vlastnosti (metody a sloty), pouze jednu metodu bych potřeboval jinou. Tak vytvořím svou třídu, ve které naprogramuji tu jednu žádoucí metodu, ostatní metody zdědí od již hotové třídy.

V manuálech knihoven tříd bývají u metod různé poznámky: "nikdy nepřepsat", "obvykle se přepisuje" apod.

Výhodou těchto knihoven je, že nám stačí útržkovité znalosti, abychom mohli objekty dodefinovat. Základní algoritmy neměníme, doprogramováváme jen to, s čím autor počítal, že se bude měnit a co popsal, jak to měnit.

Samozřejmě velmi záleží na kvalitě dokumentace.

4.3 Objektový návrh systémů

Životní cyklus informačních systémů (jejich vznik) se zhruba dělí na analýzu, design a implementaci. Přibližně můžeme říci, že

- úlohou analýzy je určit, jaké objekty použijeme ke tvorbě systému (objekty jakých vlastností) a jaké vztahy mezi nimi budou, programový systém chápeme jako síť objektů,
- úlohou designu je určit, jak budou objekty a jejich vazby konstruovány,
- úlohou implementace je jednak objekty konstruovat, jednak konstruovat program, tj. onu síť objektů.

O designu a implementaci jsme mluvili v předchozích kapitolách, nyní se věnujme analýze.

Vstupem do analýzy je účel systému - můžeme to chápat jako hrubě zadané funkce systému. Cílem analýzy je dostatečně podrobná struktura systému, která může sloužit jako východisko dalších etap návrhu. V tomto prostoru se lze pohybovat buď strukturálně nebo objektově. Oba přístupy poskytují určitá pravidla a nechávají prostor pro intuici.

Strukturovaný přístup má jako výsledek síť funkcí a dat - a to dosti podrobnou. Tím se na jedné straně zjednodušuje implementace, na druhé straně se zvětšuje prostor pro intuici.

Objektově orientovaný přístup zmenšuje prostor pro intuici dvěma způsoby. Navržené objekty tvoří přirozenou hranici konce analýzy, a tato hranice je výše než u strukturovaného přístupu (je méně podrobná).

Na druhé straně lze počáteční hrubě zadané funkce systému rozdělit podle logických kritérií na domény. Ne že by tato možnost při strukturální analýze nebyla, strukturální analýza má však jedinou možnost: realizovat domény jako moduly, tj. dosti isolované podsystémy. Objektově orientovaný přístup má dvě možnosti, realizovat domény jako isolované moduly, navíc je však může "rozpustit" v jednotlivých objektech - toto rozhodnutí se však provede až v designu. Z hlediska analýzy jsou domény samostatné podsystémy.

To, že v objektu jsou pohromadě data i funkce, to je velká výhoda. Strukturální analýza poskytuje pravidla pouze pro normalizaci dat (tj. pro odstranění redundance). Praxe však vyžaduje i pravidla pro zavedení redundance. Při objektově orientovaném přístupu logika věci naznačuje nutnou redundanci. Navíc lze docela dobře i odhadnout cenu, kterou musíme zaplatit za redundanci nebo normalizaci.

Za největší přínos objektově orientovaného přístupu pro analýzu považuji to, že objekty (prostředek pro modelování skutečnosti) svým charakterem více odpovídají prvkům reality. Práce je snazší, jestliže modelující prvek má vlastnosti modelovaného výseku reality.

4.4 Vlastnosti objektového přístupu

Předchozí text uvádí spíše výhody objektového přístupu. Objektový přístup soustřeďuje nové ideje v jeden kompletní aparát - prostředek pro podporu programování. Má ovšem i svou druhou stránku - záporné vlastnosti.

Nejnápadnější zápornou vlastností je to, že objektově realizované programy jsou pomalejší. To zpomalení je velmi výrazné - i více než o řád. Velmi pěkně to je vidět na Windows. Porovnáme-li dva algoritmicky stejně složité programy realizované přímo v DOS a prostřednictvím Windows a požadujeme-li stejnou dobu odezvy, potřebujeme pro Windows mnohonásobně výkonnější počítač. Je to způsobeno tím, že hlavní přednost Windows - aparát oken - je realizován objektově. Objektová realizace však umožňuje snadné využití tohoto aparátu. Použití objektů umožňuje, že kterýkoliv program může plně využít aparát oken. Bez objektů se nepovedlo, aby nějaký podobný prostředek doznal tak širokého využití.

Mnoha lidem nevadí, že si musí pořídit podstatně výkonnější a dražší počítače, aby mohli používat programy s okny.

Za nebezpečný však považuji jiný rys objektů. Velmi se zjednodušuje programování. Technické problémy jsou skryty uvnitř objektů, programátor nemusí znát podrobnosti jejich řešení a často je ani nezná. Věnuje se návrhu sítě objektů, i složité problémy se stávají jednoduchými.

Objektově orientované programování vyžaduje jiný způsob myšlení, návyky ze strukturovaného přístupu jsou velmi málo použitelné. Pro návrh programů se používá zdravý selský rozum - jinými slovy, navrhuji se neprofesionálně. U nepříliš složitých sítí objektů to samozřejmě nevadí, nemá smysl používat složité aparáty na jednoduché problémy. Také není důvod, aby nějaké složité aparáty vznikaly.

Poznámka. Výše uvedená úvaha neplatí pro tvorbu objektových nástrojů. To je složitý problém a rychle se profesionalizuje. Do tvorby objektových nástrojů se totiž nyní přesunula složitost programování.

Vývoj půjde dále, bude zapotřebí zvládat stále složitější sítě objektů, které jsou nad síly intuitivního rozboru. Vyvstane nutnost zvládnout tvorbu složité sítě objektů. Historie se bude v principu opakovat. Dříve jsme se snažili systematizovat tvorbu sítě příkazů, nyní je zapotřebí systematizovat tvorbu sítě objektů.

Autor :

RNDr. Pavel Drbal, CSc.
VŠE, katedra IT, nám. W.Churchilla 4, 13000 Praha 3
tel: 24095437

soukromá adresa:
Domažlická 8
13000 Praha 3