JAZYK SIMULA A OBJEKTOVĚ ORIENTOVANÉ PROGRAMOVÁNÍ

PhDr. RNDr. Evžen Kindler, CSc.

SIMULA je programovací jazyk přes dvacet let starý, který obsahuje všechny myšlenky objektově orientovaného programování. V příspěvku bude uvedena jeho pozice ve vývoji (objektově orientovaného) programování a jeho základní myšlenky, zejména ty, jež reprezentují objektově orientované programování.

Klíčová slova : SIMULA, OBJEKTOVĚ ORIENTOVANÉ PROGRAMOVÁNÍ

1. Úvod - postavení jazyka SIMULA ve vývoji programování

Jazyk SIMULA je pevně normalizovaný standard a jako takový existuje od roku 1968. Jeho norma z r. 1968 je doplněním a slabší modifikací návrhu z konce r. 1966, prezentovaného na mezinárodní konferenci IFIP v Oslo v květnu 1967. Podle toho dostal jazyk přívlastek "67", který však v 80. létech vyšel z praxe. Jelikož tento návrh obsahuje vše, co se nyní, v 80. létech, nazývá objektově orientovaným programováním, lze už sám návrh (či oficielně jeho prezentaci v květnu 1967) považovat za vznik objektově orientovaného programování, byť sám termín pochází až z 80. let, kdy byl tento druh programování "znovuobjeven" jako velmi progresivní nejen pro programování a pro konstruování a udržování nejrůznějších softwarových výrobků, ale i pro znalostní systémy a umělou inteligenci.

Na jazyku SIMULA lze tedy demonstrovat všechny podstatné rysy objektově orientovaného programování (dále jen OOP); jelikož se však o OOP dozvíme z tohoto sborníku i prostřednictvím jiných sdělení, věnujeme několik slov i okolnostem, v nichž tyto myšlenky vznikly - jde o okolnosti, z nichž se některé právě v naší době vracejí.

Jméno SIMULA ukazuje vztah k simulaci systémů, tj. k experimentální technice, při níž se zkoumaný systém nahradí jeho počítačovým (simulačním) modelem, který slouží jako objekt experimentů místo skutečného, původního objektu. Už v 60. létech existovaly potíže s implementací "simulačních programů", které musely zobrazovat řadu složitých časoprostorových vztahů zkoumaného objektu, a tak během celé uvedené dekády probíhal intenzivní vývoj tzv. simulačních jazyků. Ty měly ulehčit stavbu simulačních pregramů tak, že uživatel v nich popsal ne algoritmus modelování, nýbrž modelovaný systém sám, a kompilátor provedl automaticky konverzi popisu na algoritmus. Už tehdy bylo objeveno cosi, co dnes existuje v prostředcích OOP (avšak ne samo) : předně to byl fakt, že složité dění zvládneme lépe, když děj rozdělíme na "objekty", které při něm interagují, vznikají a zanikají - zákonitosti "životů" těchto objektů lze precizovat snadněji, než zákonitost jejich velkého množství; za druhé to byl fakt, že objekty bývají podobné, takže je vhodné popsat "životní pravidla" celé skupiny (tzv. třídy), od které jednotlivé objekty reprezentují tzv. "instance"; konečně to byl fakt, že životní pravidla lze popsat v zásadě stejnými prostředky, jako algoritmy, podprogramy, výpočty atd., až na to, že v nich figuruje společný čas, v němž se životní pravidla různých objektů uplatňují, při čemž je možno současnou existenci objektů modelovat na monoprocesorové (Von Neumannově) architektuře vzájemným (automatickým) přerušováním "životů" objektů, tzv. kvaziparalelním systémem.

Tyto rysy lze najít např. u známých jazyků GPSS (z konce 50. let) a také u jazyka SIMULA I z roku 1964. Jeho autoři však byli tíženi - podobně jako ostatní - lavinou požadavků na simulaci systémů, na druhy popisů objektů i experimentování s nimi. Kvalitativními změnami jazyka SIMULA I došli k jazyku, který z reklamních důvodů nazvali rovněž SIMULA, avšak už na zmíněné konferenci r. 1967 sdělili, že jde o univerzální programovací prostředek. V něm totiž doplnili najednou tyto prostředky popisu systémů, které reprezentují OOP, a o nichž se blíže zmíníme dále : jelikož "životní pravidla" jsou tak jako tak složitá, je vhodné interakce v nich figurující

formulovat zvlášť a složit z nich životní pravidla - podle nich žijící objekt buď interakci sám provede nebo požádá o její provedení nějaký jiný objekt (ten mu vlastně provede "službu"); dále, jelikož v praxi jsou mnohé objekty v něčem podobné a v něčem zase ne, je vhodné zavést hierarchii jejich tříd podobně, jako třeba v přírodopise, čili zavést možnost definovat třídu jako obsahově obohací jiné třídy; konečně platí, že interakci, zavedenou v jedné třídě, lze negovat v třídě, zavedené na jejím základě, totiž zavést v ní tuto interakci s jiným obsahem.

V moderní terminologii OOP se nazývá první schopnost jako zasílání zprávy - objekt zašle (vyšle) zprávu jinému objektu, v níž požádá o službu - interakci (ta se nazývá selektorem); druhá schopnost se nazývá specializací - vyjdeme z jedné třídy, specializujeme ji, tj. změníme její obsah na jinou třídu; interakci, jejíž obsah lze definitoricky změnit, nazýváme virtuální. Virtualita hraje zvlášť významnou roli v situacích, kdy je nutno programový produkt změnit dle požadavků, které při jeho tvorbě byly neznámé.

Na závěr tohoto úvodu ještě informujme o následujících rysech : předně při simulaci, kdy se často provádějí rozsáhlé, opakované výpočty, je nutné, aby se bral v potaz výpočtový čas, čili aby vše, co lze, bylo provedeno při kompilaci a ne během výpočtu; to se týká zejména smysluplnosti - provedeli se nesmyslný příkaz ve větším programovém produktu, dostane se výpočet do tzv. divokého běhu a příčinu, tedy chybu, prakticky nelze objevit; to lze vhodně vyřídit "typovou disciplínou", která ovšem pro možnost specializace je spojena s problémy; autoři jazyka SIMULA se s nimi bezvadně vypořádali; mimo ně však vyšli z typové disciplíny jazyka ALGOL 60, z něhož převzali všechny (algoritmické) prostředky. To nám na jedné straně umožňuje zkrátit výklad o technických vlastnostech jazyka SIMULA (máme u nás totiž vynikající učebnice jazyka ALGOL 60), na druhé straně to implikuje další vlastnosti jazyka SIMULA, snad vývojově přesahující samo OOP. O nich a o podobných rysech, které plynou ze simulačních historických

kořenů, pojednáme v části třetí. Upozorněme též na to, že výtky, činěné jazyku ALGOL 60, se ve spojení se zásadami OOP jeví jako výhody.

2. Jazyk SIMULA - technické podrobnosti

Nechť P je podprogram v jazyku SIMULA s parametry x, y, z nichž první je aritmetického typu a druhý textového. Nechť P potřebuje jako svou lokální proměnnou booleovskou hodnotu, nazvanou s. Pak v jazyku ALGOL 60 (i v jazyku SIMULA) se podprogram P deklaruje takto :

```
procedure P (x, y); real x; text y;
begin Boolean s; ... příkazy ... end;
```

a vyvolá příkazem P (...,...), kde jsme vytečkovali skutečné parametry. V jazyku SIMULA lze zavést třídu objektů zcela podobně, jen místo slova procedure použijeme slova class a utvoření objektu, tedy instance třídy P se liší tím, že před vyvolání dáme slovo new, tedy např. new P(...,...). Takovýto objekt se chová podobně, jako vyvolaný podprogram, až na tyto odlišnosti : předně může dostat "jméno", a to příkazem R : -new P(...,...); pomocí tohoto jména je vyvolaný podprogram jakoby objektem, jehož nesené vlastnosti, zvané atributy, jsou jednat parametry (tedy x a y), jednak lokální proměnné (tedy s). Třída P se tedy chová jako definice obecné znalosti, nazvané P, a to znalosti objektů, které mají vlastnosti x, y a z a které "žijí" dle pravidel, daných příkazy, které jsme jen naznačili. Doporučujeme představit si jako příklad to, že P je obdélník, x, y a z jsou po řadě aritmetické atributy šířka, délka a uhlopříčka a příkazy jsou takové, že z délky a šířky vypočtou uhlopříčku. Význam příkazu R : -new obdélník (3,4) je zřejmý.

"Proměnná" R ovšem musí být někde zavedena jako "referenční", a to takto : ref(P)R. Při tom ref(P) a podobné konstrukce se mohou vyskytovat ve stejném kontextu jako např. real, Boolean či text, takže objekty mohou eventuelně mít za-

ko atributy také "reference" na jiné objekty. Pro znalce Pascalu je to jasné, jde o ukazatele. Ty se však nepíší s šipkami, takže např. R.x čteme jako "atribut x objektu, na který ukazuje R". Výraz v závorce, tedy u nás F, zajišťuje typovou disciplínu (viz opět analogii s Pascalem), takže smysluplnost výraz F.x může kontrolovat kompilátor.

Za slovem begin však může být i podprogram, tedy cosi jako třeba procedure G(N..); ref(...)N..; begin...end; kde jsme jen vytečkovali parametry, jejich specifikace (zde konkrétnější jako referenční) a vlastní činnost podprogramu. SIMULA zpracovává i takové podprogramy, jako jakési atributy, takže R.G(T) je vlastně žádost objektu R, aby provedl podprogram G s parametrem T; ale ve skutečnosti je to metoda (G), realizující interakci mezi objekty R a T. Atributy (včetně metod) tvaru N.k, kde N je některý z formálních parametrů, se interpretují jako T.k, atributy, před nimiž není tečka, se interpretují jako atributy objektu R. SIMULA povoluje některé atributy "chránit", tj. blokovat proti čtení či změně přes uvedenou "tečkovou notaci"; lze je použít jen nepřímo pomocí metod, v nichž vystupují.

Nechť formulace třídy F, jak jsme ji uvedli výše, má před slovem class ještě "prefix" E. E musí být název nějaké jiné třídy. V tom případě mají objekty (instance) třídy F všechny atributy, zavedené pro třídu E (a ovšem i všechny explicitně zavedené pro třídu F) a jsou schopny provést kromě všech metod, zavedených explicitně pro F i všechny metody, zavedené pro E. Podobné pravidlo (zde blíže nespecifikované) platí i pro životní pravidla - v zásadě instance třídy F aplikuje jak pravidla, zavedené pro E, tak pravidla explicitně zavedená pro F. "Známost" F je tedy v tom případě specializací, obecnější známost E, říkáme, že F je podtřídou E. Když objekt třídy F dostane jméno E, určené dle typové disciplíny pro objekty třídy E - tedy zavedené výrokem ref(E)E - přijme kompilátor všechny konstrukce tečkové notace tvaru E.L, pokud L je atributem (i metodou) třídy E (pozor, ne třídy F).

Je-li ve třídě E zavedena třeba metoda M a ve třídě F jiná metoda s týmž jménem, vybere kompilátor jednu z nich podle právě uvedeného rozboru tečkové notace. Avšak např. ve třídě E lze zavést metodu jako virtuální : v tom případě hraje typová disciplína menší roli, důležitější je to, jak byl objekt, která ji má provést, generován, totiž, co bylo za příslušným new. Význam, náplň metody, se bere pomocí právě takového identifikátoru za new. Když tedy v nějakém programovém produktu použijeme příkazu tvaru P.G a G je virtuální, záleží na tom, který objekt byl před tím za P "dosazen". Můžeme např. doplnit programové vybavení o formulaci zcela nové třídy H se zcela novou náplní metody G, jednu instanci třídy H dosadit za P a někde uvnitř původního programového vybavení se příkazem P.G tato nová náplň metody G realizuje.

Jazyk SIMULA obsahuje v těchto prostředcích vše, co se dnes chápe jako podstatné pro OOP. Často používané entity, jako datové soubory, texty, seznamy, procesy při simulaci, prvky seznamů apod., jsou chápány jako instance "systémových" tříd, jakýchsi obdob standardních funkcí, známých z konvenčního programování.

3. Další - výhledové - možnosti jazyka SIMULA

SIMULA obsahuje řadu doplňků, které ulehčí programování, hledání chyb a obcházení typové disciplíny, avšak ty nejsou pro tento výklad podstatné. Podstatné - patrně překračující úroveň OOP - jsou tyto vlastnosti jazyka SIMULA :

a) Už simulační jazyky připouštěly spojit náplň třídy objektů s dějem, daným "životními pravidly"; SIMULA tuto možnost obsahuje také - když je generována instance třídy, začne podle těchto pravidel "žít", při čemž může vysílat zprávy jiným objektům a tak je během svého "života" žádat o provedení akcí; téměř všechny ostatní prostředky pro OOP toto nemají, takže uživatel nechá generovat instance tříd a pak musí alespoň jedné z nich poslat nějakou zprávu, čímž

se může vlastní výpočtový děj "rozjet".

b) Jak už jsme uvedli, životní pravidla simulačních jazyků podléhají kvaziparalelnímu provádění v rámci společného simulovaného času; SIMULA kvaziparalelismus zobecnila - přepínání může např. záviset na postavení v grafu nebo obecně na jakémkoliv stavu právě aktivního objektu.

c) Z jazyku ALGOL 60 převzala SIMULA blokovou strukturu a z ní mimo jiné plyne i to, že v náplni třídy (dejme tomu G) mohou být formulace jiných tříd (dejme tomu A, B, ...). Instance třídy G se pak chová jako "vidění světa" (exaktní teorie, jazyk, ..), v němž jsou smysluplně dány znalosti A, B, ... Několik takových instancí pak může najednou existovat jako různá pojetí vzájemně si konkurujících expertů v tomtéž oboru apod. S tím souvisejí problémy, neboť dosud nebyly exaktně studovány systémy, v nichž existuje v rámci jedné teorie několik individuí, které mají každý svou vlastní teorii, avšak zdá se, že SIMULA tyto problémy už úspěšně vyřešila (příkladem je zákaz přiřazení jména z jedné teorie prvku jiné teorii). To platí i pro případ, že samo G je formulováno uvnitř nějaké třídy.

d) Kombinováním b) s c) vznikají dosud zcela nedocenění možnosti popisů systémů, znalostí, teorií i programových prostředků, neboť např. jak G, tak A, B,... mohou mít kvaziparalelní systémy objektů, které v rámci těchto systémů jakoby "žily" paralelně. I když se úplné využití těchto možností jeví jako science fiction (systémová analýza 3. tisíciletí?), byly případné studie, využívající těchto možností, využity v praxi už dnes, např. při optimalizaci.

4. Závěrečné poznámky

Zásady, které byly uvedeny v 2. části, si může interpretovat každý, kdo zná nějaký konvenční programovací jazyk, podle tohoto jazyka a může jej takto alespoň na papíře rozšířit

na prostředek pro OOP. Ve skutečnosti bylo cosi velmi podobného už uděláno pro jazyk C (a tak vznikly jazyky C++ a OBJECTIVE C) a pro jazyk PASCAL (a tak vznikly jazyky OBJECT PASCAL a CWL), a to ovšem i včetně implementace (upozorňujeme na to, že v technických podrobnostech se tyto jazyky liší).

Vedle takového pojetí existují ovšem i pojetí "vstřícná", kdy entity konvenčních programovacích jazyků - např. čísla - se chápou jako objekty tříd a operace mezi nimi jako interakce. Sem patří jazyky se speciální syntaxí, která více než konvenční programovací jazyky připomíná spíše jazyky přirozené; příkladem je BETA či rodina jazyků SMALLTALK. Třetí, poněkud odlišný směr, reprezentující lispovské balíky, které svou syntaxí (zejména množstvím závorek) jazyk LISP vskutku připomínají.

SIMULA (a ostatní jazyky, vzniklé rozšířením konvenčních programovacích jazyků) umožňuje ovšem i konvenční programování. I v rámci OOP pak zachovává mnohé z výhod konvenčního programování, jako např. ekonomii použití paměti a času centrálního procesoru. Na rozdíl od mnohých novějších prostředků pro OOP nemá ve své normě prostředky pro práci v reálném čase a pro persistenci objektů, neboť dosud není ujasněna jejich optimální forma (obrovskou výhodou jazyka SIMULA je, že její norma nemusela být za 20 let měněna, takže programové produkty v ní jsou kompatibilní).

S návazností na konvenční programování je spjat i interfejs některých prostředků pro OOP na konvenční programovací jazyky. Tak SIMULA má nejen interfejs na ALGOL 60 (ten totiž existuje automaticky proto, že jde o rozšíření tohoto jazyka), ale i na FORTRAN (implementace pro střediskové počítače), na autokód a na jazyk C (implementace pro osobní počítače).

V naší zemi existují a jsou dostupné implementace obojího druhu. Před 3 léty byla zakoupena pasivní licence pro počítače JSEP a IBM 370 (a vyšší), takže u podniku DATASYSTÉM lze zaplatit v naší měně a získat instalaci (12. v pořadí a tedy

velmi kvalitní) pro počítače JSEP s operačním systémem OS či vyšším (existuje i možnost pro DOS - pomocí emulátorů). Do naší země dále přicházejí demonstrační verze jazyka SIMULA pro osobní počítače, které umožňují kompilovat moduly (obvykle třídy) do 300 řádek a z nich linkovat libovolné programové produkty - vše pod MS/DOS. O nákupu komerční verze pro osobní počítače se zatím jedná.

Literatura :

O.-J.Dahl, B.Myhrhaug, K.Nygaard : Common base language, Norsk Regnesentralen, Oslo, 1968 (1.vyd.), 1970 (2.vyd.), 1982 (3.vyd.), 1984 (4.vyd.). (Je to norma jazyka SIMULA)

SIMULA Standard. Simula a.s., Oslo, 1985. (je to rozšířený standard, implementovaný mj. pro osobní počítače)

E.Kindler, M.Brejcha : Programování a algoritmizace v jazyce SIMULA - kurs B. Dům techniky ČSVTS Plzeň, 1987 (základní kurz jazyka SIMULA, obsahující mj. v závěru i řadu učebnic a učebních textů tohoto jazyka československých i zahraničních).

PhDr. RNDr. Evžen Kindler, CSc.
Výpočetní centrum Karlovy University
Malostranské náměstí 25
118 00 Praha 1