Objektové a strukturované programování

Pavel Drbal, Helena Jilková, VŠE Praha, KIT

Anotace

Tento článek ukazuje, že mezi objektovým a strukturovaným programováním je sice rozpor, který je zdánlivě nesmiřitelný, ale zároveň může být i velmi užitečný.

Úvod

Mezi objektovým a strukturovaným programování existuje rozpor, který je často pokládán za nesmiřitelný. Je to logické, paradigmata obojího se značně liší. Za zásadní odlišnost lze považovat četnost účelů. Strukturovaný program je zásadně jednoúčelový, vždy se spouští z jednoho bodu a vždy dělá totéž. Objekt je víceúčelový, nabízí vcelku neomezené množství vstupních bodů (metod), každý vstupní bod může sloužit jinému účelu. Lze to pokládat za pouhou technickou odlišnost (strukturovaný program lze pro jednotlivé účely parametrizovat, metody objektu v rámci vyššího celku slouží témuž účelu), velký rozdíl však je ve stylu myšlení. Navíc je objektový přístup módní a strukturované programování je zastaralé[1]. Je to podobné jako s auty, chůze je zastaralá, zrušme chodníky[2]. (Vidíme to i na naší podnikatelské kultuře. Podnikatel odjede autem do fitcentra, tam používá půl hodiny simulátor chůze a pak se odveze do podniku, místo aby do podniku došel)

Podívejme se na problém z jiného hlediska, z hlediska efektivity, v čem jsou zásadní rozdíly mezi oběma přístupy a jak se projevují. Pojem efektivity samotné je vágní v tom smyslu, že se vždy musí uvažovat hledisko, ze kterého efektivitu určujeme. Efektivita z hlediska údržby programu je něco úplně jiného, než efektivita z hlediska doby odezvy - a právě v těchto dvou hlediscích vidíme základní rozdíly mezi oběma přístupy.

Pružnost

Vzhledem k zapouzdření objektů má objektový přístup nedostižnou výhodu ve snadné upravitelnosti a opravitelnosti programů. Jednolitý (strukturovaný) program je na jakékoliv zásahy citlivý, protože změna na jednom místě se může projevit

---

[1] O tom svědčí frekvence přednášek na konferencích i obranný postoj zaujímaný v referátech

[2] To není žert, mnohá „moderní" města jsou pro chodce neprůchodná.

nečekanými důsledky i na vzdálených místech. Kdežto u objektů je vždy rozlišeno, jestli dělám změnu uvnitř objektu, nebo měním vazby mezi objekty. V obou případech změna nepronikne samovolně přes hranici objektu - na hranici objektu jsou jen ty změny, které explicitně zavedu.

Synergie

Obvykle se vykládá, že synergie objektu je větší než synergie skupiny procedur (odpovídajících metodám objektu). Je to samozřejmě pravda, metody objektu mají své vlastní (vzhledem k objektu globální) proměnné a vlastní podřízené procedury, a nikdo z vnějšku nemůže tyto vztahy a hodnoty narušit. Na druhé straně se nijak nezdůrazňuje, že síť objektů má menší synergii než jednolitý program s touž funkcí.

Synergie není příliš používaný pojem, možná by se dal nahradit pojmem provázanost - vždy to však znamená možnost ušetřit zdroje. Například synergický efekt školní jídelny je v tom, že není zapotřebí mít pro každého žáčka vlastní kamínka, vlastní hrnce a vlastní kuchařku - finanční přínos je zřejmý.

V čem je synergický efekt sítě objektů? Posloupnost příkazů „jednolitého" programu je v síti objektů rozbita do velkého množství malých procedur, které se navzájem vyvolávají.

Samozřejmě se i v „jednolitých" programech používají procedury, je však typické, že jsou „větší" a méně časté, že jsou náhražkou opakovaného psaní kódu. Kdežto v objektech má každý objekt svou dílčí odpovědnost, a k tomu, aby ji splnil, často potřebuje služby jiných objektů. Tyto řetězce poskytovaných služeb mohou být i dost dlouhé.

Zřetězení procedur zvyšuje nároky na paměť, protože jsou vyžadovány prostory pro lokální proměnné - ty jsou uvolňovány až po dokončení procedury, takže v případě řetězení procedur jsou lokální proměnné vyžadovány v podstatě současně. Daleko významnější jsou časové ztráty - volání procedury je jeden z časově nejdražších příkazů programu[3]

Šlo by pokračovat v tomto rozboru a v uvádění příkladů, stručně lze říci:

objekty poskytují pořádek v programu a platíme za to potřebou větší výkonnosti.

O koexistenci objektového a strukturovaného přístupu se obvykle mluví tak, že objekty jsou objekty a že když máme nějakou větší metodu, tak ji naprogramujeme strukturovaně. Podívejme se na věc jinak.

---

[3] Hodně to závisí na počítačovém prostředí i na jazyku, v assembleru to je lacinější, ale o všech vyšších programovacích jazycích na běžných počítačích to lze říci určitě

Převod objektů na strukturovaný program

V pozdních fázích designu optimalizujeme program. Jedna z optimalizaci je rozhodnutí o representaci objektů. Jinými slovy, máme rozhodnout, jestli nějaká věc není tak titěrná, že nestojí za to zavádět pro ni samostatné objekty. Typický učebnicový příklad:

> Objekt třídy „Linie" je kontejnerem dvou objektů třídy „Bod". V podstatě se jedná o úsečku, která je určena čtyřmi čísly - souřadnicemi. Asi bude rozumnější nepoužívat aparátu *kontejner* (kontejner „Linie" obsahuje dva objekty typu „Bod") a raději definovat objekty třídy „Linie" se čtyřmi atributy - souřadnicemi.

Naše úloha je poněkud obecnější - lze převést síť objektů na „jednolitý" program? Vypracujeme techniku, která to umožní.

Východiska technologie strukturovaného programování

Program děláme proto, aby ze vstupních dat vytvořil výstupní data. Postup technologie strukturovaného programování lze schematicky shrnout do těchto bodů:

1. Popíšeme výstupní data včetně jejich struktury.
2. Popíšeme vstupní data včetně jejich struktury.
3. Určujeme korespondenci mezi komponentami datových struktur, případně měníme datové struktury, aby došlo ke korespondenci. (Korespondence mezi komponentami datové struktury nastává, jestli si příslušná data odpovídají smyslem, počtem a pořadím.)
4. Jestliže vztahy mezi daty jsou plně určeny korespondencí, složíme datové struktury do jedné - programové struktury. (Skládáním datových struktur se rozumí ztotožnění těch uzlů struktury, mezi kterými je korespondence.)
5. Jestliže mezi datovými strukturami jsou rozpory, použijeme jednu ze tří technik pro řešení datových rozporů.
6. Do programové struktury doplníme operace - příkazy programovacího jazyka.

Ucelené programování

Cíl

Čím se technika strukturovaného programování liší od objektového přístupu? Vlastně jen tím, že objektově orientovaní lidé už nevidí žádná data, ale jen objekty! To je ale výhodou, jestliže si u dat představujeme i operace s nimi - je aspoň zřejmé, co do programu bude patřit.

Cílem tohoto textu je formulovat pravidla pro odvození uceleného programu (strukturovaného programu) ze sítě objektů, jinými slovy: najít pravidla pro tvorbu uceleného programu, která by byla analogická výše uvedeným pravidlům pro strukturované programování. Vzhledem k tomu, že se jedná o první pokus taková pravidla formulovat, necháme stranou datové rozpory.

Průvodní příklad

Při vysvětlování tvorby programu z objektů použijeme jako příklad primitivní hru, kde hrdina jménem Baltazar prochází bludištěm a cestou sbírá houby jako bonusy. Kroky Baltazara jsou řízeny kurzorovými klávesami. Cílem hry je projít bludiště v nejkratším čase. Zdi bludiště jsou neprůchodná políčka, bonusy jsou kladné nebo záporné (hříbky nebo muchomůrky). Počet kladných bonusů se odečítá od dosaženého času, počet záporných bonusů se přičítá.

Objektová struktura a životní cyklus objektů

Při objektovém přístupu je tvorba programu rozfázována do několika relativně nezávislých etap, z hlediska našeho bludiště to jsou:

⇒ objektový model, tj.. síť objektů, kde se určují
- ◊ vztahy mezi objekty (asociace),
- ◊ odpovědnosti, které objekt přebírá,
- ◊ služby, které objekt nabízí,

⇒ dynamické modely (pro jednotlivé objekty), kde se určuje chování objektu - možná že výstižněji lze říci, že se určuje algoritmus chování objektu.

Tyto dvě skupiny modelů můžeme vytvářet relativně nezávisle, navíc je chování objektu skryto uvnitř (zapouzdřeno). Objektový program funguje tak, že objekty navzájem po sobě vyžadují (a poskytují si) služby, které svými vnitřními algoritmy zpracovávají. Algoritmy jsou skryté a můžeme je měnit nezávisle na sobě, případně i dodatečně.

Vztahy mezi objekty (asociace) jsou několika typů, z našeho hlediska je zajímavé rozdělení na
- ◊ běžné asociace,
- ◊ agregace.

Běžné asociace jsou vztahy mezi objekty, které nějakým způsobem spolupracují, což je například slovně popsáno.

Agregace je typ asociace, kde je mezi objekty úzký vztah, který se dá vyjádřit jako „obsahuje" nebo (v opačném směru) jako „je částí". Objektu, který obsahuje druhé objekty, se říká „**kontejner**", obsažené objekty budeme označovat jako podřízené objekty.

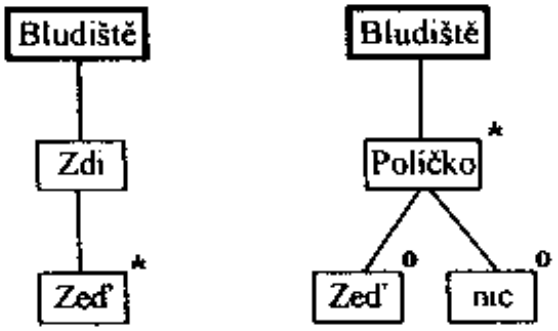
Každý objekt má svůj **životní cyklus**, který lze vyjádřit strukturogramem. Životní cyklus je souhrn všech akcí, které objekt může provádět v rámci zajišťování své odpovědnosti. Životní cyklus jsou vlastně všechny dovolené kombinace provádění metod objektu - lze to říci i opačně, metody jsou části životního cyklu objektu, které se provádějí na vnější impuls. Životní cyklus se obvykle určuje přechodovým diagramem.

> *V objektově orientovaném přístupu odpovědnost za dodržení životního cyklu má jednak objekt sám, který nesmí dopustit takovou kombinaci metod, která by narušila odpovědnost objektu, jednak jsou*

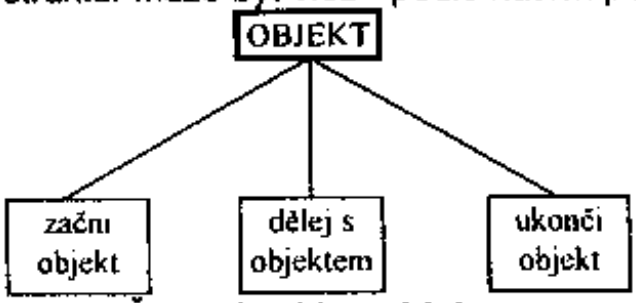
konkrétní etapy životního cyklu určovány impulsy (voláním metod) z okolí objektu.

Z formulace průvodní úlohy je zřejmé, že výchozími objekty jsou bludiště, prostředek pro měření času (stopky) a proud dat z klávesnice. Bludiště obsahuje políčka, na kterých mohou být bonusy (hřibky nebo muchomůrky) nebo může být políčko prázdné. Ostatně samotný hrdina je také objekt.

První, co musíme udělat, je nahradit datovou strukturu analogickým pojmem odvozeným z objektů. Zkusme chápat kontejner jako iteraci obsažených objektů. Vidíme to na obr. 1, kde je znázorněno, že „Bludiště" obsahuje zdi. (Zeď je políčko pro hrdinu neprostupné.) Vhodných struktur může být více - podle našich potřeb.



obr. 1 Objektová struktura

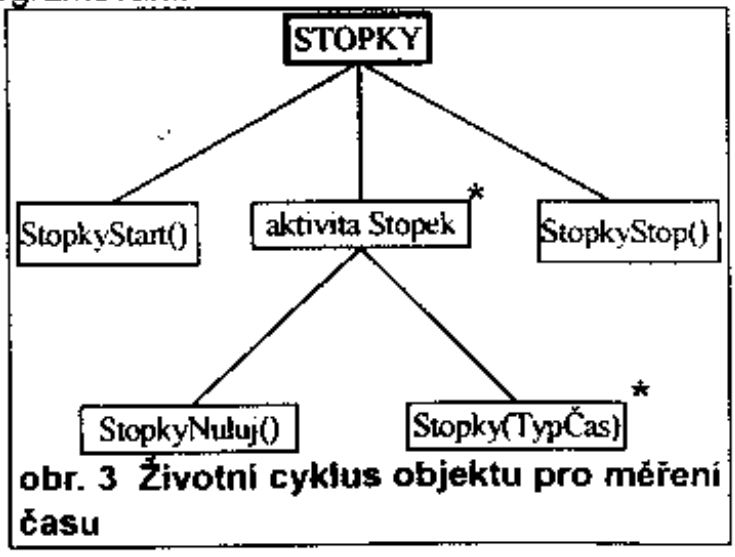


obr. 2 Životní cyklus objektu

Další možností je zachytit životní cyklus objektu, viz schéma na obr. 2.

Použitelné životní cykly ovšem musí být konkrétnější, viz obr. 3. Zase je nutno říci, že u jednoho objektu můžeme popsat různé struktury, respektive, že strukturu životního cyklu můžeme upravovat podle potřeb při hledání korespondence. Právě volnost v úpravách struktur je tvůrčí činnost v programování.

Ještě jednu věc je zapotřebí u objektů zdůraznit. Kontejner má svůj životní cyklus a obsažené objekty také mají svůj život. Rozhodně obecně nelze redukovat životní cyklus kontejneru na souhrn životních cyklů obsažených objektů.



obr. 3 Životní cyklus objektu pro měření času

Je tu poněkud zvláštní fenomén, že z jednoho objektu získáme více různých struktur, které navíc ještě spojujeme dohromady. Je nutno si uvědomit, že struktura objektu je účelovým pohledem na objekt.

Podívejme se na obr. 3 - zcela určitě si můžete představit i jiné struktury. Mohou to být i špatné struktury (StopkyStop je před StopkyStart) nebo pouze neužitečné (vícekrát za sebou Stopky Nuluj). Struktura na obrázku předjímá strukturu vytvářené hry (podrobněji níže v pravidle o úpravě struktur).

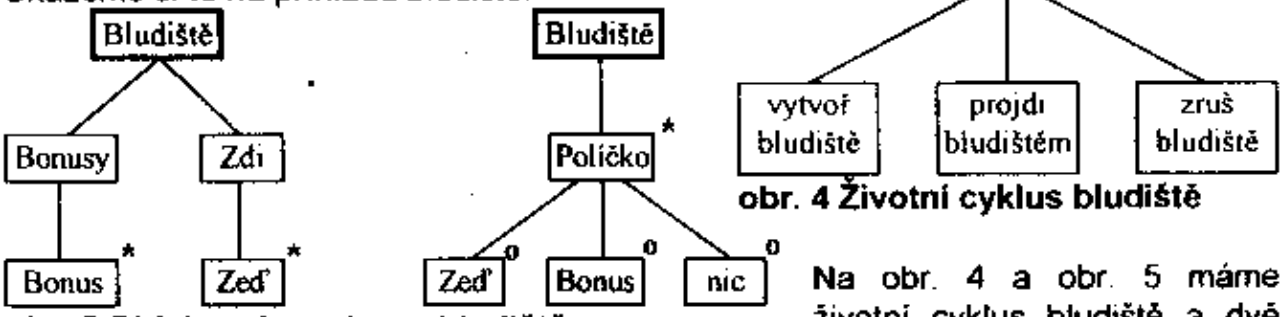
Pravidla pro tvorbu programů

Když se zamyslíme nad naším bludištěm, zjistíme, že jsou zde pravidla pro převod životního cyklu kontejneru do uceleného programu - a to pravidlo vhodného rozkladu a pravidlo propagace. (Modifikací těchto pravidel získáme pravidla další.)

Pravidlo propagace

Jestliže kontejner má vykonat nějakou akci, tj má realisovat etapy svého životního cyklu, má k tomu tři možnosti:
1. realisace na svých datech,
2. realisace na datech podřízených objektů,
3. kombinace obou způsobů.

Ukážeme si to na příkladu bludiště.



obr. 4 Životní cyklus bludiště

obr. 5 Objektová struktura bludiště

Na obr. 4 a obr. 5 máme životní cyklus bludiště a dvě možné verse jeho (kontejnerové) objektové struktury. Víme, že vytváření bludiště se týká vytváření jednotlivých zdí a bonusů, avšak procházení bludištěm se také týká všech políček, i neobsazených zdmi nebo bonusy. V terminologii objektů se tomu říká „propagace", tj. akce kontejneru se realizuje provedením akcí obsažených objektů.

Jinými slovy:

Jestliže chápeme kontejner jako iteraci podřízených objektů, tak akce kontejneru se realizuje pomocí iterace akcí podřízených objektů

Výše uvedený slogan je pouze slogan,obecněji to lze říci takto.

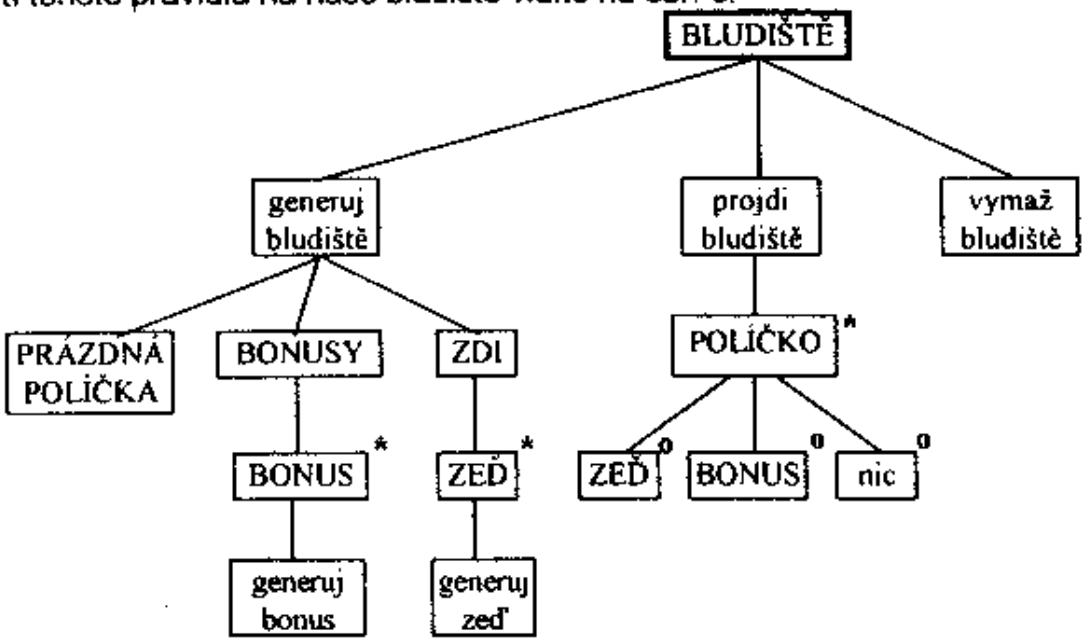
Objektová struktura kontejneru se kopíruje do struktury jeho akcí

Jinými slovy, akci celého kontejneru (akce je část životního cyklu) můžeme realizovat pomocí skupiny akcí podřízených objektů (částí jejich životního cyklu) Struktura této skupiny akcí podřízených objektů je odvozena z objektové struktury (tj. odpovídá struktuře podřízených objektů v kontejneru).

Pravidlo vhodného rozkladu

Strukturu kontejneru můžeme vyjádřit různými konstrukcemi. Zcela určitě můžeme volit mezi sekvencí a iterací - to tehdy, jestliže obsažené objekty jsou jednoho typu Jsou-li obsažené objekty různých typů, pak jsou možnosti bohatší - může to být

sekvence iterací objektů jednoho typu nebo iterace selekcí - dvě možnosti vidíme na obr 5. Jsou to bohaté možnosti, má to ale stinnou stránku, nevíme, kterou možnost zvolit.
Použití tohoto pravidla na naše bludiště vidíte na obr. 6.



obr. 6 Spojení životního cyklu a objektové struktury bludiště

Z výše uvedených úvah vyplynula prvá pravidla:

1	Popíšeme výstupní a vstupní data včetně jejich struktury.	Evidujeme objekty a určíme objektovou (kontejnerovou) strukturu.
1a		Určíme životní cykly objektů.
2	Určujeme korespondenci mezi komponentami datových struktur, případně měníme datové struktury, aby došlo ke korespondenci.	Provedeme propagaci podřízených objektů do životních cyklů kontejneru.

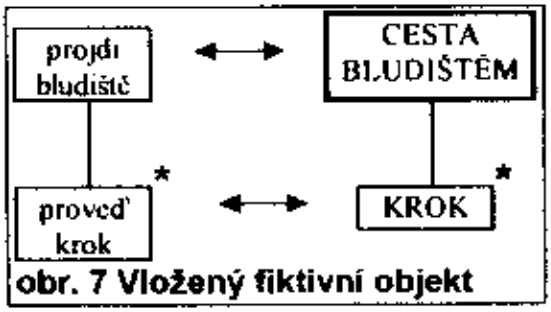
U obr. 6 vás určitě napadne, že volba struktur je intuitivní. Je to ještě horší, volba struktur je ryze účelová, byly voleny tak, aby se program dobře dělal. Podívejme se na to, proč je tomu tak.

U generace bludiště generujeme každé políčko právě jednou, tj. struktura příkazů odpovídá objektové struktuře kontejneru - jedná se tedy skutečně o prostou propagaci. Můžeme zvolit některou strukturu, pro náš případ můžeme vybírat ze struktur na obr. 5.

> *Volba struktury se později podstatně promítne do způsobu programování. Jestliže zvolíme sekvenci iterací (tak to je na obr. 6), pravděpodobně se to bude realizovat vygenerováním všech prázdných políček a pak se budou náhodně obsazovat zdmi a bonusy. Jestliže naopak zvolíme iteraci selekcí, pak při vytváření každého políčka se bude náhodně určovat, je-li to zeď nebo bonus, nebo jestli zůstane prázdné.*

Jiná situace je u etapy „projdi bludištěm". Rozhodně se hrdina nebude procházet po všech políčkách (například nemůže projít zdí), ani neplatí, že na každé políčka

šlápne jen jednou (může vycházet ze slepé uličky). Propagaci sice můžeme použít, nelze však použít objektovou strukturu.



obr. 7 Vložený fiktivní objekt

Na tento problém můžeme pohlížet dvěma způsoby:

◊ je to složitý problém, nezbývá než ho naprogramovat,

◊ zjednodušíme si problém tím, že hledáme nějaké datové prvky, na které by se příkazy vázaly.

Na obr. 7 vidíme, že si lze pomoci zavedením logického objektu „cesta bludištěm". Tento objekt vlastně zahrnuje všechny příkazy etapy „projdi bludištěm".
Návaznost příkazů na data je jasná, akci „projdi bludištěm" odpovídá posloupnost políček, kterými Baltazar prošel nebo projde. Ovšem podřízenými „objekty" nejsou políčka, ale kroky, tj. přechody z jednoho políčka na druhé, sousední.

Samozřejmě zde máme dvě možnosti, můžeme za podřízené objekty zvolit políčka i kroky. Protože se však jedná o ***řídicí*** *objekty, tj. objekty s výrazným dynamickým charakterem, volíme více dynamickou možnost. To se vyplatí v tom smyslu, že objekty korespondují s částmi algoritmu.*

Vypadá to, že jsme vyčerpali možnosti objektu bludiště, použijeme další objekty.
Na začátku práce jsme si vytvořili popisy objektů a jejich životních cyklů pomocí strukturogramů, o tom je předchozí text. Pro skládání objektů v jeden celek si vytvoříme další pravidla. (Objekt Stopky je na obr. 3, objekty Klávesy a Baltazar na obr. 8.)

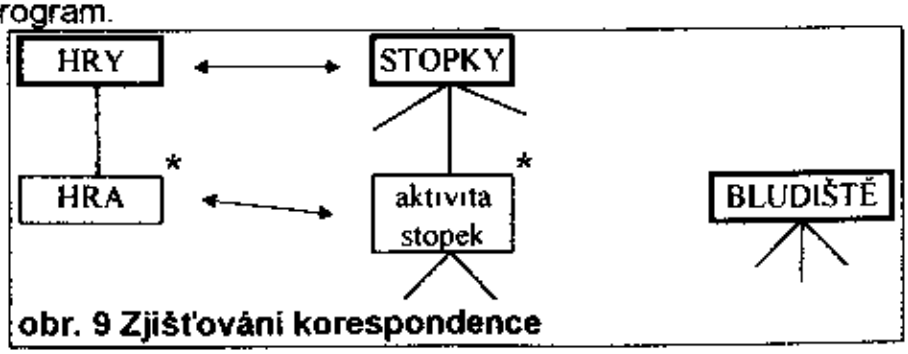
obr. 8 Další výchozí objekty hry

Pravidlo skládání objektů podle korespondence

Mezi dvojicemi objektů nebo mezi dvojicemi skládajícími se z objektu a z uzlu životního cyklu jiného objektu hledáme vztahy korespondence. Nalezené dvojice spojujeme v jeden strukturogram.

Korespondence je tehdy, jestliže mezi dvěma prvky je soulad ve

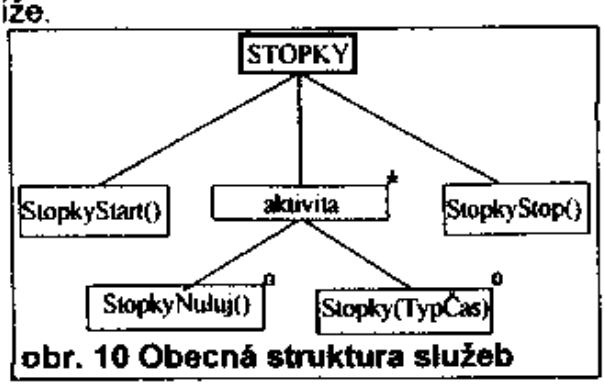
- smyslu,
- počtu,
- pořadí.



obr. 9 Zjišťování korespondence

Příklad vidíme na obr. 9. Vidíme horní úrovně životních cyklů tří objektů. (Je přidáno to, že jeden program je více her.) Vztahy korespondence mezi hrami a stopkami jsou vyznačeny šipkou. Je zřejmé, že na jednu sadu her potřebuji jedny stopky, na jednu hru jednu aktivitu stopek (od jejich nulování po zjištění času). Jednota smyslu je

zřejmá (hra potřebuje stopky), počet a pořadí také souhlasí, protože aktivita stopek je dána požadavky hry - o tom viz pravidlo níže.



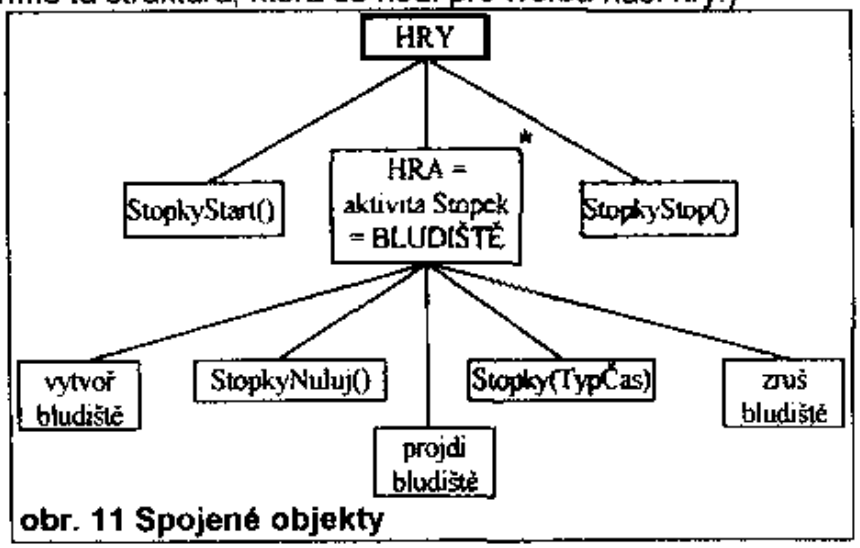
obr. 10 Obecná struktura služeb

Podobně to je u objektu Bludiště. Každá hra bude mít jiné bludiště, proto objekt Bludiště koresponduje objektu Hra (není vyznačeno šipkou).

Pravidlo úpravy struktur podle korespondence

U některých objektů je životní cyklus pružný, k danému objektu může být přiřazeno více myslitelných životních cyklů. Zvolíme takovou strukturu, která vyhovuje požadované korespondenci.

Obecnou strukturu zapisujeme podobně jako na obr. 10, kde stálé místo má jen start a stop stopek, kdežto ostatní dvě funkce se mohou používat v libovolném pořadí a libovolném množství (ale ne vše je smysluplné). Pro účely skládání objektů můžeme (a vlastně musíme) zvolit jednu konkrétní strukturu - samozřejmě ji zvolíme tak, abychom měli co nejlepší korespondenci. (Mnohdy to děláme podvědomě, nemusíme si to ani uvědomovat. To se stalo například v tomto článku, kdy jsme v obr. 3 zvolili jako první přímo tu strukturu, která se hodí pro tvorbu naší hry.)



obr. 11 Spojené objekty

Výsledek spojení objektů vidíme na obr. 11. Co si koresponduje, to se překrývá. Je zapotřebí zdůraznit jednu věc: při důsledné aplikaci těchto principů si uvědomíme, že korespondence a překrývání se týká zobecněných věcí - to jsou jednak samotné objekty[4], jednak uzly ve strukturách. Základních příkazů (funkcí) se překrývání nemůže týkat, o těch můžeme prohlásit, že se provádějí se stejnou frekvencí - po sobě.

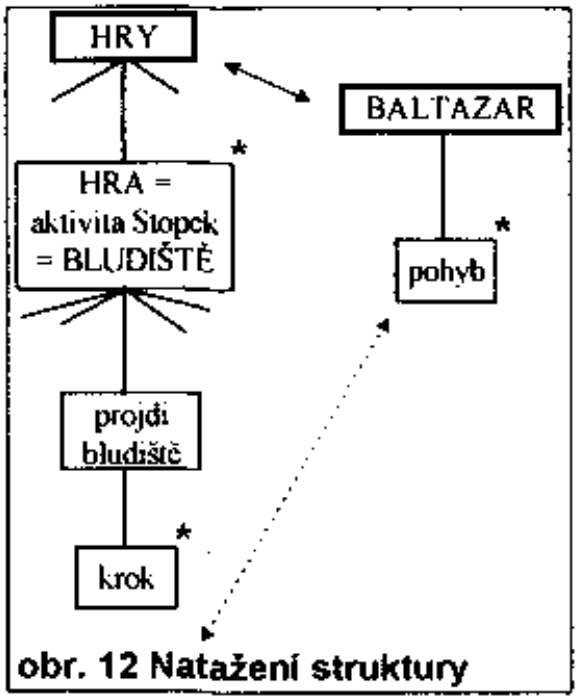
Pravidlo natažení struktur

Určitou částí pravidla úpravy struktur je natahování struktur - rozumíme tím toto: Zjišťujeme-li korespondenci dvou struktur a jsou splněny následující podmínky:

* na horních úrovních již je korespondence ustavena,
* mezi dalšími kandidáty je různý počet úrovní,

[4] Tím se liší naše koncepce, v objektovém programování je objekt určitá programová konstrukce, kdežto v tomto článku to je myšlenková konstrukce.

pak mohu vložit další úrovně tak, aby korespondence byla možná.



obr. 12 Natažení struktury

Příklad vidíme na obr. 12, kde je korespondence mezi hrami a Baltazarem (je jeden Baltazar pro všechny hry), kdežto prvek „pohyb" může korespondovat něčemu pod prvkem „krok". (Pohyb Baltazara je otočení vlevo, otočení vpravo a přechod na sousední políčko ve směru otočení - náš příklad je Baltazarovi poněkud poplatný.)

Strukturu životného cyklu Baltazara můžeme upravit tak, aby vyhovoval našim potřebám. To znamená vložit dvě iterace, jednu, která bude odpovídat opakovanému prvku „Bludiště", druhou, která bude odpovídat prvku „krok". Korespondence mezi nimi je zřejmá, takže po spojení bude jeden krok realizován iterací Baltazarových pohybů, což bude několik otočení a jeden přechod na sousední políčko.

Vlastnosti programů a jejich tvorby

Při tvorbě programů podle výše uvedených pravidel se projevuje několik výrazných vlastností, na které musíme brát zřetel.

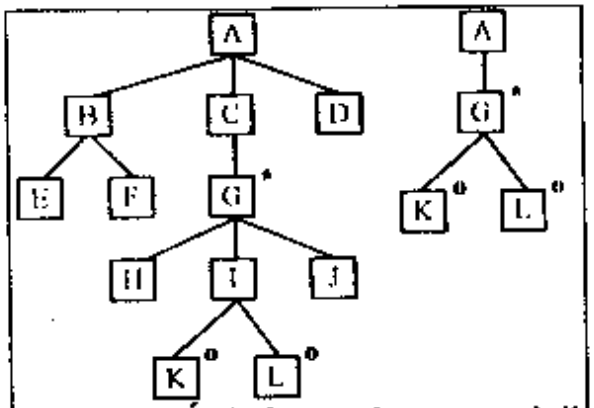
Vlastnost „rozpor struktur"

Ne vždy je možné spojit dvě struktury v jednu. Nejdříve si však určíme, co je na strukturách pro jejich spojování důležité a co může rozpor způsobit. Ve struktuře můžeme rozlišovat:

◊ přívěsky,
◊ jádro struktury (kostra),
◊ doplňky.

Všechny části dobré struktury mají svůj smysl a jsou potřebné, toto rozdělení je pouze z hlediska spojování struktur. Jako **přívěsky** označujeme větve a větvičky, které nemají partnera ve druhé struktuře. Do výsledného programu se samozřejmě promítnou, nijak však spojování neovlivňují. Například u stopek jsou takovými přívěsky „StopkyStart" a „StopkyNuluj".

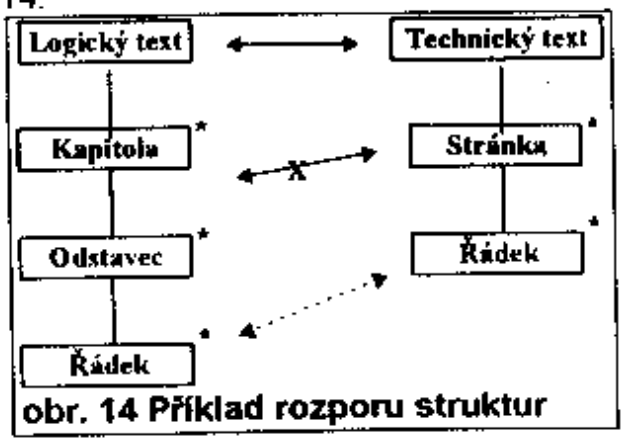
Jádro struktury - dalo by se říci kostru - tvoří iterace a selekce. Ty totiž určují počet dat nebo akcí komponentou representovaných - a počet je důležitou vlastností při určování korespondence. Oproti tomu sekvence neovlivňují korespondenci - i když obsahují potřebná upřesnění a zpodrobnění. Samozřejmě sekvence iteraci nebo selekcí do kostry struktury patří. **Doplňky** říkáme všem prvkům, které nepatří do jádra struktury.



obr. 13 Úplná struktura a její kostra

Příklad vidíte na obr. 13. Říkáme, že „G" má větší frekvenci než „A", což znamená, že počet dat nebo akcí representovaných komponentou „G" je větší než dat nebo akcí representovaných komponentou „A". Všimněte si, že „K" má menší frekvenci než „G", protože vše, co patří do „G", může patřit buď do „K" nebo „L".

Korespondenci určujeme mezi jádry struktur (i když přívěsky a doplňky ve strukturách zůstávají). Obvykle používáme zkratkovitého vyjádření „korespondence mezi strukturami", vždy se tím myslí „korespondence mezi jádry příslušných struktur". Někdy nelze mezi strukturami určit korespondenci - to bývá tehdy, jestliže není některá ze složek korespondence naplněna (smysl, počet, pořadí). Této situaci říkáme **rozpor mezi strukturami** a znamená to, že příslušné objekty nelze spojit v jeden ucelený program, tj. musíme to realizovat jako dva objekty (nebo program se speciálním podprogramem). Příklad je na obr. 14.



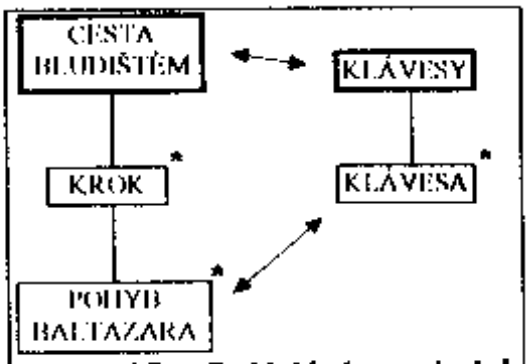
obr. 14 Příklad rozporu struktur

Vidíme zde dvě struktury, jedna popisuje autorsky chápaný text (logicky strukturovaný na kapitoly a odstavce), druhá je tiskařsky chápaný text, tj. iterace stránek. Kořeny obou struktur si korespondují (jedná se vlastně o tentýž text v různých stadiích rozpracovanosti). Korespondovaly by si i komponenty „Řádek", kdyby i nad nimi byla korespondence. Ale s komponentou „Stránka" nemůže korespondovat ani „Kapitola", ani „Odstavec", nelze je navzájem jednoznačně přiřadit - odstavec i stránka mohou přecházet se stránky na stránku, takže například nelze říci, že stránka je iterací odstavců nebo že odstavec je iterací stránek (obojí je možné). Nelze vytvořit jednolitý program, problém je možno řešit vytvořením dvou programů, které si předávají data prostřednictvím souboru, nebo jako dva objekty, jeden předává data druhému.

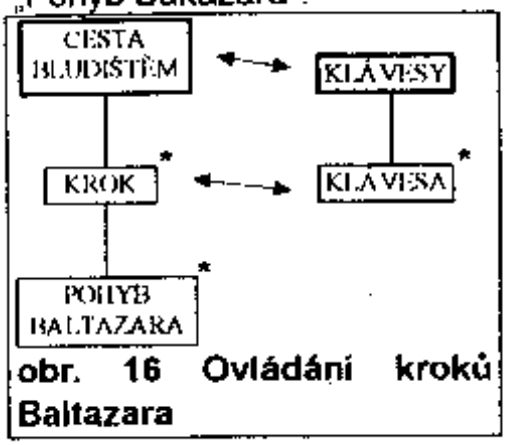
Vlastnost „konservace koncepce"

Určením korespondence volíme určitou koncepci programu, spojením struktur v jednu tuto koncepci zafixujeme. Ukážeme si to na obr. 15 a obr. 16.

Obrázky ukazují dvě možnosti, které máme při určování korespondence mezi tokem kláves a průchodem bludiště. Jedna z možností je přiřadit klávesu pohybu Baltazara, to znamená, že klávesami se řídí i otáčení, i přechod na sousední políčko. Úprava struktur sestává z toho, že komponentu „Krok" sdružíme s komponentou „Pohyb Baltazara".



obr. 15 Ovládání pohybů Baltazara



obr. 16 Ovládání kroků Baltazara

V druhém případě může být klávesa přiřazena kroku, takže na jednu klávesu provede Baltazar několik otočení a přechod na sousední políčko. Iterovaný prvek „Pohyb Baltazara" je pak přívěsek

Určování korespondence mezi strukturami je velmi pohodlný prostředek, který umožňuje stavět programy na míru problému, tj. dělat programy, které optimálně řeší daný problém.

Na druhé straně z této korespondence vyplyne určitá programová koncepce, která se v plné míře projeví až po dohotovení programu. Avšak měnit již vytvořený ucelený program je dost pracné - síť objektů je daleko pružnější a lze snáze měnit.

V naší hře na procházení bludiště má rozhodnutí o korespondenci klávesy závažné důsledky - rozhoduje se zde, jak se bude hra hrát. Jestliže se klávesa přiřadí pohybu Baltazara (to znamená, že na každé otočení je zvláštní klávesa, další je pro přechod na sousední pole), je hra pomalejší, hráči se častěji zmýlí a Baltazar naráží do zdi. Jestliže se klávesa přiřadí kroku Baltazara (to znamená, že jeden stisk klávesy program otočí Baltazara do příslušného směru a provede přechod), je hra rychlejší a s hladším průběhem. Tento výsledný styl hry je zakonservován v programu rozhodnutím o korespondenci.

Vlastnost „kompaktní data"

Objektové programování je ucelený styl programování, který je podporován v programovacích jazycích. Dá se říci, že objektové programování shrnuje zkušenosti, konvence a triky normálního rozumného programování a že je svým vlastním způsobem formalizuje. Jedna z těchto zkušeností je, že data jsou vedena v programu kompaktně, tj. že k sobě patřící data jsou držena u sebe a že jsou vybavena procedurami, které realizují základní operace s těmito daty. (Je to analogie atributů a metod, ale bez formalizace.)

Například je dost jedno, jestli jsou „Stopky" objektem a funkce „StopkyNuluj", „StopkyStart" a další jsou metody tohoto objektu, nebo jestli pro stopky existuje skupina proměnných a skupina funkcí, které s těmito proměnnými operují.

Je tedy rozumné, členit data do kompaktních skupin a často používané operace s těmito daty zapisovat jako funkce - souvislost mezi daty a funkcemi zachytit pomocí pojmenování funkcí - například tak jak to vidíte u stopek: jméno funkce je tvořeno jménem dat a slovem popisujícím činnost funkce.

Shrnutí

Uvedený postup tvorby ucelených programů lze použít ve dvou situacích:
- při převodu části programového systému na strukturovaný program (důvodem může být efektivnost),
- při tvorbě nového programu.

Obě situace se liší tím, že v první již máme všechny objekty k dispozici a pouze z nich odvozujeme program, kdežto ve druhé situaci musíme objekty rozpoznat.

Východiskem rozpoznání objektů jsou vstupní a výstupní požadovaná data. Ty seskupíme do logicky provázaných celků, které považujeme za objekty. Každá taková skupina implikuje základní operace se svými daty. O každé datové skupině si vytvoříme představu o její roli při požadovaném zpracování - to je východisko pro její životní cyklus. Ze znalosti o předpokládaném algoritmu vytvářeného programu (a ze zkušeností s podobnými programy) lze také odvodit potřebné objekty.

Celý postup můžeme shrnout do těchto etap:
- Zavedení objektů a životních cyklů
- Korespondence mezi strukturami
- Vytvoření struktury programu
- Doplnění operací do struktury

Náplně jednotlivých etap jsou následující.

Zavedení objektů a životních cyklů

Hlavní náplní etapy je evidence objektů a tvorba jejich životních cyklů. Pro složitější objekty - kontejnery - se vytváří objektová struktura. Do životních cyklů kontejnerů se propagují prvky životních cyklů podřízených objektů - viz **Pravidlo propagace** a **Pravidlo vhodného rozkladu.**

U hodně složitých životních cyklů hledáme dílčí data, která umožní definovat dílčí objekty, nebo složitější podstrukturu bereme jako náplň řídicího objektu. Tímto způsobem složité ad hoc struktury rozbijeme do životních cyklů (a jejich struktur) jednotlivých dílčích objektů, které v další etapě skládáme v celek pomocí korespondence.
Výsledkem této etapy jsou strukturogramy reprezentující jednotlivé objekty.

Korespondence mezi strukturami

V této etapě slaďujeme jednotlivé struktury tak, aby z nich šlo vytvořit jednu strukturu. Používáme k tomu pojmu korespondence (souladu podle smyslu, počtu a pořadí datových údajů representovaných komponentami struktur). Nejdříve se určí

korespondence mezi kořeny struktur (buď mezi dvěma kořeny, nebo mezi kořenem jedné struktury a vnitřní komponentou jiné struktury). Pak se určuje korespondence v nižších úrovních. Nesmí se vynechat žádná úroveň kostry (jádra) struktury - viz **Pravidlo skládání objektů podle korespondence**, přičemž se struktury upravují pomocí dovolených úprav - viz **Pravidlo úpravy struktur podle korespondence** a **Pravidlo natažení struktur.**

Při určování korespondence se hlídá, jestli nedochází k rozporu, viz **Vlastnost „rozpor struktur"**. Určování korespondence je důležité také v tom smyslu, že se zde definitivně určují vlastnosti programu, viz **Vlastnost „konservace koncepce"**

Protože orientace na objekty vede k drobení struktur, tak se již zde spojují drobnější struktury ve větší celky - ovšem jen ty, kde je možné spojení zřejmé.

Vytvoření struktury programu

Zdánlivě mechanické spojování struktur do jedné celkové (programové) struktury. Podstatné na této etapě je zjišťování, jestli nedochází k rozporům struktur, a odkrytí rozhodnutí o koncepci. I v této etapě se občas rozpoznají nové objekty a provedou se pro ně počáteční etapy postupu.

V případě rozporu struktur se program rozdělí do více navazujících programů nebo se realizuje jako několik (větších) objektů.

Všechna rozhodnutí o koncepci se zaznamenají do dokumentace programu i s důvody, které k danému rozhodnutí vedly.

Doplnění operací do struktury

Na závěr se do programové struktury doplní operace, tj. listy životních cyklů objektů.

Revize programové struktury

Samozřejmě se provede revise programové struktury, nejlépe myšlenou simulací nebo strukturovanou oponenturou. Při tom se provádí fragmentace struktury, tj. vyjmutí podstruktur ze kterých se utvoří procedury (z několika podobných podstruktur se vytvoří obecná procedura) nebo se skupina dat a příslušných funkcí vyčlení jako samostatný objekt - viz **Vlastnost „kompaktní data"**

Závěr

Uvedený postup ukazuje, že z objektových paradigmat může vycházet jak objektové, tak i strukturované programování a že kriteriem pro volbu mezi těmito způsoby by měl být účel - snaha po dosažení větší pružnosti nebo větší rychlosti.

Uznání

S nápadem odvodit strukturovaný program z objektů přišel Bohumír Soukup z Uherského Hradiště.

Literatura

[1] Jackson,M.A.: System Development, Academic Press, London, 1978
[2] Jackson,M.A.: Principles of Program Design, Academic Press, London, 1975
[3] Drbal P., Vaníček J.: Technologie strukturovaného programování, Kancelářské stroje k.ú.o. Praha 1983
[4] Drbal P., Jilková H.: Metody a technologie progamování, VŠE, Praha 1986
[5] Booch G.: Object-Oriented Analysis and Design, The Benjamin/Cummings, Redwood City 1994
[6] Coad, P., Yourdon, E.: Object-Oriented Analysis, Prentice-Hall 1991
[7] Coad, P., Yourdon, E.: Object-Oriented Design, Prentice-Hall 1991
[8] Jacobson, I.: Object-Oriented Software Engineering, Addison - Wesley 1994
[9] Mellor, S.J., Shlaer, S.: Object-Oriented Systems Analysis, Prentice-Hall 1988
[10] Rumbaugh J.: Object-Oriented Modeling and Design, Prentice-Hall 1991
[11] Šešera, L., Mičovský, A.: Objektovo-orientovaná tvorba systémov a jazyk C++, PERFEKT Bratislava 1994
[12] Drbal P. a spolupracovníci: OOMT Objektově orientované metodiky a technologie, skripta VŠE, Praha, 1997, 300s. ISBN 80-7079-740-1
[13] Drbal P.: SGP Baltazar - programování pro děti a rodiče, učebnice pro základní školy, Praha, prosinec 1997, 261s. ISBN 80-7226-044-8