SOFTWAROVÁ ARCHITEKTURA RESPEKTUJÍCÍ MENTÁLNÍ MODELY UŽIVATELE

Rudolf Pecinovský
Zbyněk Šlajchrt
ICZ a.s., Hvězdova 1689/2a, 140 00 Praha 4,
Katedra informačních technologií VŠE Praha
rudolf@pecinovsky.cz

ABSTRAKT:

O objektově orientovaném programování se předpokládalo, že v kódu sjednotí úhel pohledu programátora a koncového uživatele a poskytne tak oběma vyšší míru pochopení problémy a vyšší užitnou hodnotu. Objekty dokáží dobře, tj. tak, aby se na správnosti shodl programátor i uživatel, zachytit strukturu objektů. Při zachytávání jejich funkcionality však již často tak úspěšné nejsou. Jeden z přístupů, které se snaží zachytit uživatelovy kognitivní modely jednotlivých účastníků a interakcí mezi nimi, je architektura DCI, jejíž název vznikl jako zkratka z termínů *Data – Context – Interaction* a která je popsána např. v [3]. Příspěvek si klade za cíl seznámit s hlavními myšlenkami této architektury.

KLÍČOVÁ SLOVA:

OOP, objekty, role, interakce, metodika DCI.

1 ÚVOD

Alan Kay, který později vytvořil jazyk Smalltalk, napsal kdysi esej [2], která prezentovala jeho vizi osobního počítače, který je společníkem a svým způsobem i rozšířením svého uživatele. Tuto vizi se později snažil vtělit i do programovacího jazyka Smalltalk. Cílem Kaye i dalších pionýrů objektově orientovaného programování bylo zachytit v kódu mentální modely, které máme ve své mysli.

Když člověk pracuje s počítačem, tak současně přemýšlí a koná. Má-li být komunikace mezi člověkem a počítačem opravdu hladká, musí být model v programu počítače v souladu s modelem v mysli uživatele. Každá akce, kterou uživatel provede, totiž ve svém důsledku manipuluje s objekty v kódu. Je-li uživatelů mentální model v souladu s modelem, nad nímž pracuje program, odpadne při práci s programem většina chyb a překvapení.

2 VZOR MVC

Podívejme se na komunikaci uživatele s programem realizovaným implementací vzoru MVC (*Model – View – Controller*). Většina programátorů jej považuje za složeninu několika implementací návrhového vzoru *Pozorovatel*. Při hlubším pohledu si ale uvědomíme, že rámec existuje proto, aby oddělil reprezentaci informací od uživatelské interakce a budeme-li chtít postihnout všechny účastníky, měli bychom jej nazývat MVC-U (Model – View – Controller – User).

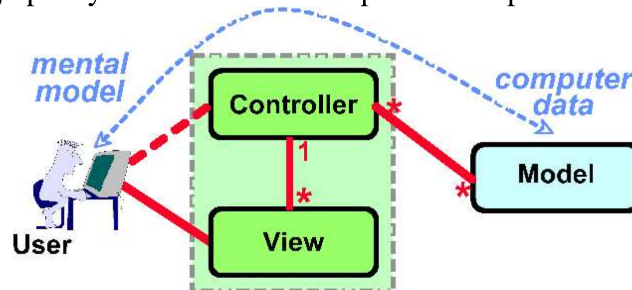
MVC-U je o vytváření spojení mezi *daty* v počítači a představami v hlavě uživatele. *Data* jsou reprezentací informací; v počítači je často reprezentujeme jako bity. Ale bity samy o sobě nic neznamenají. Začnou nabývat významu až v mysli uživatele, když s nimi pracuje. Mysl koncového uživatele dokáže tato data interpretovat – pak se stávají *informacemi*. *Informace* je termín, který budeme používat pro interpretovaná data. *Informace* je klíčovou součástí *mentálního modelu koncového uživatele*.

Dobře navržený program správně zachytí informační model uživatele v datovém modelu programu (nebo alespoň poskytne iluzi, že tomu tak je). Pokud to program dokáže, tak uživa-

tel vnímá paměť počítače jako rozšíření paměti vlastní. Pokud to program nedokáže, je třeba kompenzovat nastalý zmatek „překladem“.

Pohled zobrazuje *Model* na obrazovce. Poskytuje jednoduchý protokol jak předat informaci do a z *Modelu*. Srdce objektu *Pohled* předvádí data *Modelu* jedním speciálním způsobem zajímavým pro uživatele. Různé pohledy mohou pracovat se shodnými daty, tj. se shodnými modely, naprosto různými způsoby. Klasickým příkladem je možnost, aby jeden *Pohled* zobrazil táž data jako sloupcový graf, jiný jako koláčový graf a třetí jako tabulku čísel.

Ovladač iniciuje tvorbu *Pohledů* a koordinuje *Pohledy* a *Modely*. Obyčejně přijímá roli interpreta uživatelových pokynů, které obdrží jako stisky kláves či jiné události. Základem objektového pohledu je poskytnout uživateli iluzi přímé manipulace s daty.



Obrázek 1[1]: Rámec *Model-View-Controller-User*

3 PROBLÉMY S REPREZENTACÍ CHOVÁNÍ

Modely postavené na objektech dokáží dobře reprezentovat strukturu dat. Horší je to se zachycením chování. U jednoduchých objektů ještě nemáme problémy. Ty nastanou až ve chvíli, kdy chování objektů je složitější a účastní se jej více objektů.

Ukažme si to na příkladu. Mějme textový procesor, který se rozhodneme vybavit kontrolou překlepů. Který objekt by ji však měl zapouzdřit? Textový buffer? Slovník? Nějaký globální objekt, který bychom označili jako *KontrolorPřeklepů*? Každé z uvedených řešení má své nevýhody. Některá vedou ke špatné soudržnosti objektů, které mají na starosti danou kontrolu, jiná zase zvyšují provázanost mezi jednotlivými objekty.

4 V ČEM JE PROBLÉM

Doporučení pro začínající návrháře OO programů zní: Podstatná jména ve specifikaci řešené úlohy představují objekty, slovesa představují metody. Tato dichotomie přirozeně vypichuje dva koncepty, které mohou programovací jazyky vyjádřit. Objektově orientované jazyky (a především ty „čisté“) vyjadřují vše jako objekty či metody objektů. Řada jazyků k tomu sice používá třídy, avšak základem je, že nic neexistuje mimo objektový rámec.

Podívejme se nyní na objekt *SpořícíÚčet*. Skutečnosti, že mohu snížit jeho zůstatek a mohu z něj vybrat peníze jsou obě pojaty jako metody. Obě představují chování. Avšak tato chování se radikálně liší.

Snížení zůstatku je pouhou charakteristikou dat. Naproti tomu výběr odráží účel dat. Schopnost zpracovat výběr souvisí se sémantikou transakcí, interakcí s uživatelem, zpracováním chybových stavů a pravidly aplikační logiky. Tím daleko překračuje základní rámec datového modelu. Výběr je ve skutečnosti chováním systému a má za následek změnu jeho stavu, kdežto snížení zůstatku je to, co dělá účet účtem a dotýká se pouze stavu tohoto objektu. Tyto dvě operace jsou extrémně rozdílné z hlediska architektury systému, softwarového inže-

[1] Obrázky článku jsou převzaty z [3].

nýrství, očekávatelných požadavků na změny. Objektová orientace je však hází obě do jednoho pytle.

Klíčovou vlastností dobrých programů je, že v zájmu snadné správy oddělují věci, které se nemění, od těch, které se mění.

Doménový model reprezentuje (přesněji měl by reprezentovat) uživatelův mentální model věcí v jejich světě. Ten je v čase relativně stabilní. Objekty, které v něm vystupují, bývají navíc relativně hloupé. Základní doménové objekty reprezentují naše prapůvodní poznání o podstatě doménové entity daleko spíše než celý vesmír procesů a algoritmů, kterými klasický objektově orientovaný návrh zatěžuje prostřednictvím hromady případů užití.

Když se vás zeptáme, co může dělat spořící účet, jistě nám moudře odpovíte, že může zvýšit a snížit svůj zůstatek a podat o něm zprávu. Když ale řeknete, že může uloženou částku obhospodařovat, dostaneme se náhle do světa transakcí a interakcí s obrazovkou bankomatu nebo prověřovacím záznamem. Tím jsme ale vyskočili mimo objekt a hovoříme o propojení s hostitelem obchodní logiky, o které ale jednoduchý spořící účet nic neví.

Problém tohoto přístupu je následující: předpokládáme-li, že objekty zůstávají stabilní, a je-li veškerý kód v objektech, pak kde máme reprezentovat části, které se mění. Klíčovou vlastností dobrých programů je, že v zájmu snadné správy oddělují věci, které se nemění, od těch, které se mění. Odrážejí-li objekty stabilní část kódu, musí existovat ještě další mechanizmus, který by v kódu odrážel požadavky na změny a podporoval tak agilní vizi evoluce a spravovatelnosti. Jenomže objekty jsou stabilní a v objektově orientovaném programování není jiný mechanizmus.

Uvězněný uvnitř těchto umělých omezení přichází objektový svět s umělým řešením: k vyjádření „programování podle rozdílů“ (programming by difference) nebo „programování rozšiřováním“ (programming by extension) použijme dědičnost.

Protože dědičnost může vyjádřit různé variace základu, stává se rychle mechanizmem na zachycení rozdílů v chování oproti „stabilní“ základní třídě. Ve skutečnosti se tento přístup stává předzvěstí uznávané techniky označované jako princip otevřený-zavřený (*open-close principle*), který říká, že třída má být uzavřena vůči modifikacím (tj. má zůstat stabilní, aby odrážela stabilitu doménového modelu), avšak otevřena vůči rozšířením (přidávání nového, dříve neočekávaného kódu podporujícího nové chování). Toto použití dědičnosti pak vylezlo ze světa programování do nářečí návrhu.

V průběhu lety získaly staticky typované jazyky nadvládu podporovanou softwarovým inženýrstvím. Jedním důležitým aspektem systému statického typování je třída: konstrukce, která umožňuje překladači generovat efektivní kód pro vyhledávání metod a polymorfizmus. Dokonce i *Smalltalk*, jehož počáteční vize objektů a dynamického běhového prostředí byla skutečně vizionářská, přinesl oběť tomuto třídnímu kompromisu. Třída se stala implementačním nástrojem pro analytický koncept označovaný jako objekt. Toto přepnutí od dynamického ke statickému bylo začátkem konce zachycení dynamického chování.

5 NÁVRAT K ROLÍM

Zkusme se vrátit ke konceptu modelování postaveném na rolích a k metodě OORAM publikované v roce 1966 v [4]. Podle tohoto konceptu objekty zachycují jaké věci jsou, kdežto role zachycují kolekce chování, které popisují, co objekty dělají.

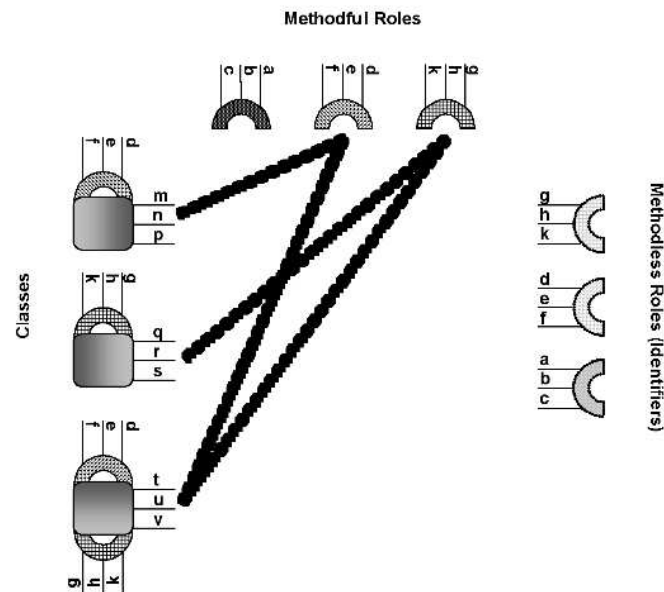
Hranice objektů a hranice rolí se liší. Objekty představuji místa zapouzdření doménových znalostí dat, kdežto role poskytují přirozené hranice sdružující kolekce operací, které jsou spolu logicky svázané a mohou se dotýkat celé řady objektů současně. Objektová orientace postavená na mechanizmu tříd nás nutí používat objekty pro oba druhy hranic.

Základním problémem řešeným architekturou DCI je to, že lidé mají v hlavách dva různé modely jediné, unifikované věci označované jako bankovní systém.

- Jedním je datový model *co bankovní systém je*. Ten podporuje přemýšlení o bance s jejími účty.
- Druhým je algoritmický model interakcí *co bankovní systém dělá*. Ten má na starosti např. specifikaci akcí při převodu hotovosti mezi účty.

Uživatel rozpozná jednotlivé objekty a domény jejich existence. Objekt však musí implementovat také chování, které přichází z uživatelova modelu interakcí. Ty svazují objekt s jinými objekty prostřednictvím rolí, které jednotlivé objekty hrají v případech užití. Koncový uživatel má dobrou představu o tom, jak tyto dva pohledy „pasují“ dohromady. Uživatelé např. vědí, že jejich spořící účet má jistou zodpovědnost v roli *ZdrojovýÚčet* v případu užití *PřevodHotovosti*. Mapování mezi hlediskem rolí a hlediskem dat je také součástí uživatelova kognitivního modelu. Označujeme jej jako *Kontext* provádění scénáře případu užití.

Ukažme si vše na příkladě modelu na obr. 2.



Obrázek 2.
Kombinování struktur a algoritmů ve třídě

- Na pravé straně v něm jsou zachyceny abstrakce uživatelových rolí jako rozhraní, tj. jako schopnosti bez konkrétní implementace (methodless roles – role s deklarovanými, ale neimplementovanými metodami).
- Na vrchu najdeme role, které začínají jako klony abstrakcí vpravo, avšak jejich metody implementovány (methodful roles). Pro koncepty jako *ZdrojovýÚčet* v případu užití *PřevodHotovosti* můžeme definovat metody, které jsou nezávislé na konkrétním typu objektu, který bude hrát danou roli za běhu. Tyto role jsou *generické typy*.
- Na levé straně máme staré přátele: třídy. Obé, tj. role i třídy, žije v uživatelově hlavě. Tyto dvojice za běhu fúzují na jeden objekt.

Protože objekty pocházejí ve většině programovacích jazyků ze tříd, musíme zařídit, aby to vypadalo, jako že doménové třídy mohou podporovat aplikační funkce, které existují v oddělených zdrojích formalizmu rolí. Při překladu musí programátor čelit oběma uživatelovým modelům, tj. scénáře případu užití a entitám, nad nimiž pracuje.

Chceme programátorovi pomoci zachytit tyto modely odděleně ve dvou rozdílných programových konstrukcích které ctí dichotomii v uživatelově hlavě. Obyčejně považujeme třídy za přirozené místo, v němž sdružíme dohromady chování či algoritmy. Musíme ale podpořit také zdánlivý paradox, že každý z těchto konceptů fáze překladu koexistuje s druhým za běhu programu v jedné věci označované jako objekt.

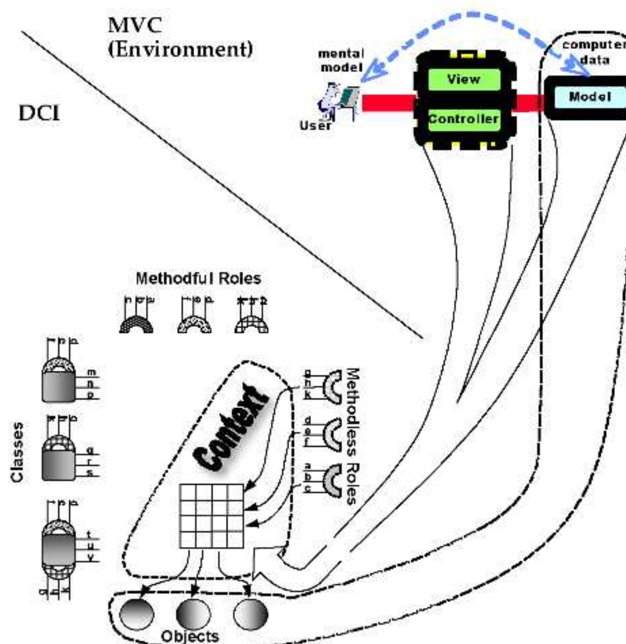
Uživatelé jsou schopni kombinovat části těchto pohledů ve svých hlavách. Proto vědí, že *SpořícíÚčet*, jehož hlavním účelem je udržovat informace o právě dostupných prostředcích, může být požádán, aby se stal zdrojovým účtem při převodu peněz. Takže bychom měli být schopni vzít operace ze scénáře převodu hotovosti a přidat je, byť trochu hloupě, k objektu *SpořícíÚčet*.

Obrázek 2 ukazuje takové slepování logiky rolí (oblouky) a logiky tříd (zaoblené obdélníky). *SpořícíÚčet* již má operace, které mu umožňují provádět skromný úkol informování o zůstatku a jeho snižování a zvyšování. Tyto operace definuje jeho doménová třída. Víc operací vztažených k scénáři případu užití však přichází od rolí, které daný objekt hraje.

Zatím je vše nastaveno tak, že každý objekt získá veškerou logiku v době překladu, aby mohl podporovat libovolnou roli, o jejíž hraní bude požádán. Když však budeme dostatečně chytří, můžeme to udělat také tak, že dostatečnou logiku injektujeme objektu až za běhu až ji bude potřebovat k podpoře aktuálně hrané role.

6 ARCHITEKTURA DCI

Vraťme se k našemu příkladu s převodem hotovosti ze spořícího na investiční účet. Budu-li chtít provést převod, potřebuji, aby můj *SpořícíÚčet* byl schopen hrát roli *ZdrojovýÚčet* a můj *InvestičníÚčet* roli *CílovýÚčet*. Představte si, že bychom mohli magicky slepit metody rolí s jejich objekty a pak spustit interakce. Každá metoda role by se spustila na objektu, s nímž by byla slepena, což je přesně to, jak to uživatel vnímá.



Obrázek 3: Mapování rolí na objekty

Situace je znázorněna na obr. 3. Šipka od řadiče a modelu ke kontextu označuje mapování jehož zdroji jsou objekty modelu.

Všechny objekty, které jsou potřebné k převodu, jsou prozatím v paměti, uživatel má proces převodu v mysli. Musíme vybrat kód, který realizuje daný algoritmus, a pak již stačí sdružit správné objekty se správnými rolemi a nechat kód běžet.

Jak je ukázáno na obr. 3, algoritmus a mapování rolí na objekty je vlastnictvím objektu *Context*, jenž ví jak najít či získat objekty, které se stanou skutečnými aktéry v tomto případu užití, a „obsadit" je do příslušných rolí v daném scénáři. V typické implementaci bude v každém případu užití objekt *Context*, který bude obsahovat identifikátor pro každou roli obsaženou v daném případu užití. Bude tak svazovat identifikátory rolí se správnými objekty.

Pak už jen odstartujeme spouštěcí metodu na „vstupu“ role pro daný *Context* a kód poběží. Může běžet nanosekundy a nebo roky, bude však odrážet uživatelův model výpočtu.

Nyní máme kompletní architekturu DCI:

- *Data*, která žijí v doménových objektech, které jsou zakořeněné v doménových třídách,
- *Context*, který na požádání uvádí živé objekty na jejich pozice ve scénáři,
- *Interakce*, které popisují uživatelovy algoritmy v termínech rolí, přičemž oboje lez najít v uživatelově hlavě.

Jak je ukázáno na obr. 3, o kontextu můžeme uvažovat jako o tabulce, která mapuje metody rolí na metody objektů. Kontext ví, který objekt bude v zadaném scénáři hrát kterou roli. Kód v *Řadiči* nyní pracuje s obchodní logikou převážně v rámci kontextu: každá drobná znalost objektu může být zapsána v termínech rolí, které jsou převedeny na objekty prostřednictvím kontextu.

7 MOŽNÉ IMPLEMENTACE – TVÁŘE (TRAITS)

Možné implementace jsou založeny na konceptu označovaném *trait,* který překládáme do češtiny termínem *tvář*. Je-li role analýzou konceptu v mysli uživatele, pak tvář je obecný návrh konceptu představujícího role. Jehož implementace se jazyk od jazyka liší.

V C++ můžeme tváře reprezentovat jako šablony, jejichž členské funkce jsou v době překladu posbírány z konkrétních tříd tak, aby výsledné objekty poskytly za běhu jak chování doménové třídy, tak šablony tváří.

```
. . . .

template <class ConcreteAccountType> class TransferMoneySourceAccount {
public:
   void transferTo(Currency amount) {
        beginTransaction();
        if (self()->availableBalance() < amount) {
        . . . .
}

. . . .

class SavingsAccount:
    public Account,
    public TransferMoneySourceAccount<SavingsAccount> {
public:
    void decreaseBalance(Currency amount) {
        . . . .
    }
}

. . . .
```

V jazyku Scala jsou tváře implementovány jako jazykové konstrukce nazvané *traits*, jejichž metody mohou být injektovány do objektů při jejich zřizování.

```
. . . .

trait TransferMoneySourceAccount extends SourceAccount {
  this: Account =>

  // This code is reviewable and testable!
  def transferTo(amount: Currency) {
    beginTransaction()
```

```
    if (availableBalance < amount) {
        . . . .
    }
}

. . . .
val source = new SavingsAccount with TransferMoneySourceAccount
val destination = new CheckingAccount with TransferMoneyDestinationAccount

. . . .
```

8 VLASTNOSTI DCI

- Pro zachycení hlavních konceptů, které se vyskytují v případu užití, používáme role.
- K zachycení hlubších doménových konceptů vyplývajících ze zkušeností a neformulovaných znalostí používáme objekty jako nepříliš chytrá data.
- DCI řeší integritu doménových tříd i rolí tak, že výsledné programy respektují princip otevřený-zavřený.
- DCI přirozeně vyhovuje agilnímu způsobu vývoje softwaru. Umožňuje programátorům, aby se přímo napojili na mentální model uživatele.

9 ZÁVĚR

Článek představil základní myšlenky architektury DCI. Ukázal na problémy, které se tato architektura snaží řešit, a současně naznačil způsoby, jak se její řešení promítne v různých programovacích jazycích.

LITERATURA

[1] *IFIP-ICC Vocabulary of Information Processing*; North-Holland, Amsterdam, Holland. 1966; p. A1-A6.

[2] KAY Alan: *A Personal Computer for Children of All Ages*, Xerox Palo Alto Research Center, 1972. http://www.mprove.de/diplom/gui/Kay72a.pdf

[3] REENSKAUG Trygve; COPLIEN James O.: *The DCI Architecture: A New Vision of Object-Oriented Programming*. http://www.artima.com/articles/dci_vision.html

[4] REENSKAUG Trygve; WOLD P.; LEHNE O. A.: *Working with objects – The OOram Software Engineering Method*. Manning Pubns Co © 1995, 366 stran, ISBN 1-884777-10-4, http://heim.ifi.uio.no/~trygver/1996/book/WorkingWithObjects

[5] SCHÄRLI Nathanael, NIERSTRASZ Oscar, DUCASSE Stéphane, WUYTS Roel, BLACK Andrew: *Traits: The Formal Model*. Technical Report, no. IAM-02-006, Institut für Informatik, November 2002, http://scg.unibe.ch/archive/papers/Scha02cTraitsModel.pdf